EDUSHARK TRAINING

Self-paced programmes in data and AI

Deep Learning

An 18-Week Advanced Workbook

Tensors to transformers, and making one cheap enough to ship.

edusharktraining.com

Deep Learning: An 18-Week Advanced Workbook

Publisher: EDUSHARK TRAINING
Web: https://edusharktraining.com/deep-learning
Contact: info@edusharktraining.com

Typeset with LaTeX.

How to Use This Workbook

This book covers 18 weeks, 126 lessons and 144 self-assessment questions. Assumes Python and some machine learning.

What is different about it

Attention is built by hand before it is imported, and the efficiency chapter reports that quantisation made inference slower, pruning to ninety percent sparsity saved nothing, and distillation lost. Those measurements are the point.

How each chapter is laid out

One chapter per week, seven days per chapter. Each day states what it is for, works through the material, and ends with a takeaway worth keeping. Code appears as it was run, and program output is set apart in grey so you can tell at a glance which is which:

```
print(sum(x * x for x in range(4)))
```

```
14
```

Every chapter ends with a self-assessment quiz. The answers are on the same page, set small, because a question you cannot check is not worth attempting. Anything you get wrong points at the day to re-read.

Working alongside the site

Every chapter here matches a week published at https://edusharktraining.com/deep-learning. The website version carries the same material with the code laid out for copying. Use whichever suits where you are reading.

Contents

Chapter 1

Tensors and Automatic Differentiation

Week 1

OBJECTIVES

The machinery under every deep learning library: tensors, shapes, gradients, and the five-step training loop.

Deep learning is one idea: write a function with a great many adjustable numbers in it, measure how wrong it is, and work out which way to nudge each number. This week builds that from the bottom. You will write the training loop by hand before you are allowed the library version, check a gradient against calculus, and finish with a real model on real digits, measured against a baseline that takes one line.

Day 1 Tensors and Shapes

The array that remembers, and the broadcast that hides your bug

Deep learning has one idea in it. You write down a function with a very large number of adjustable numbers in it, you measure how wrong it is, and you work out which way to nudge every one of those numbers to make it less wrong. Everything else on this course is detail about how to do that quickly and without the whole thing falling over.

This week is about the machinery underneath. If you have used Keras or PyTorch before and found it mostly worked until it did not, this is the week that fixes that, because almost every confusing failure later on is a shape problem or a gradient problem, and both live here.

A tensor is an array that remembers what happened to it

```
import torch

scalar = torch.tensor(3.0)
vector = torch.tensor([1.0, 2.0, 3.0])
matrix = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
batch = torch.zeros(8, 3, 32, 32)      # 8 images, 3 channels, 32 by 32

for name, t in [('scalar', scalar), ('vector', vector),
                ('matrix', matrix), ('batch', batch)]:
    print('%-8s shape %-22s dim %d  dtype %s'
          % (name, str(tuple(t.shape)), t.dim(), t.dtype))
```

```
scalar   shape ()                     dim 0  dtype torch.float32
vector   shape (3,)                   dim 1  dtype torch.float32
```

```
matrix   shape (2, 2)                dim 2  dtype torch.float32
batch    shape (8, 3, 32, 32)        dim 4  dtype torch.float32
```

Read the shape, always

The single most common error in this subject is a shape mismatch, and the single most useful debugging habit is printing `tensor.shape` before and after anything you are unsure about. By convention the first dimension is the batch, so a tensor of shape `(8, 3, 32, 32)` is eight images, each with three colour channels, each 32 pixels square. Nothing enforces that convention. It is a habit that every library and every paper follows.

The operations you will use constantly

```python
import torch

a = torch.arange(6.0).reshape(2, 3)
b = torch.ones(3, 4)

print('a')
print(a)
print('a @ b shape', tuple((a @ b).shape))
print('a.T shape  ', tuple(a.T.shape))
print('sum        ', a.sum().item())
print('sum along rows   ', a.sum(dim=0))
print('sum along columns', a.sum(dim=1))
print('mean of everything %.3f' % a.mean().item())
```

```
a
tensor([[0., 1., 2.],
        [3., 4., 5.]])
a @ b shape (2, 4)
a.T shape   (3, 2)
sum         15.0
sum along rows    tensor([3., 5., 7.])
sum along columns tensor([ 3., 12.])
mean of everything 2.500
```

`dim=0` means collapse the rows and keep the columns, which is the direction you want when you are summarising a batch. It is worth saying that out loud a few times, because getting it backwards produces code that runs, returns a number of the right type, and is silently wrong.

Broadcasting

```python
import torch

batch = torch.ones(4, 3)          # 4 rows, 3 features
bias = torch.tensor([10.0, 20.0, 30.0])
print('batch + bias')
print(batch + bias)

column = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
```

```
print('\nbatch * column')
print(batch * column)

try:
    torch.ones(4, 3) + torch.ones(4)
except RuntimeError as e:
    print('\nand when it cannot:')
    print(' ', str(e)[:96])
```

```
batch + bias
tensor([[11., 21., 31.],
        [11., 21., 31.],
        [11., 21., 31.],
        [11., 21., 31.]])

batch * column
tensor([[1., 1., 1.],
        [2., 2., 2.],
        [3., 3., 3.],
        [4., 4., 4.]])

and when it cannot:
  The size of tensor a (3) must match the size of tensor b (4) at non-singleton dimension 1
```

Watch Out

The broadcast that does not fail is the one to fear

Adding a tensor of shape `(4,)` to one of shape `(4, 1)` does not raise. It broadcasts to `(4, 4)`, and you now have sixteen numbers where you wanted four. This is the classic silent bug in a loss function: your targets come back as `(batch,)`, your predictions as `(batch, 1)`, and the loss you compute is the mean of every pairing rather than the mean of the matched pairs. It trains. It just trains on nonsense.

```
import torch

pred = torch.tensor([[0.9], [0.2], [0.7], [0.1]])   # (4, 1)
target = torch.tensor([1.0, 0.0, 1.0, 0.0])          # (4,)

wrong = ((pred - target) ** 2).mean()
right = ((pred.squeeze(1) - target) ** 2).mean()

print('difference shape when broadcast:', tuple((pred - target).shape))
print('loss computed carelessly %.4f' % wrong.item())
print('loss computed correctly  %.4f' % right.item())
print('\nneither raised an error.')
```

```
difference shape when broadcast: (4, 4)
loss computed carelessly 0.3625
loss computed correctly  0.0375

neither raised an error.
```

Day 1 takeaway

Day 2 Automatic Differentiation

How one backward call fills in every gradient, and why it is cheap

Yesterday's tensors were just arrays. Today they start remembering, which is the one feature that separates a deep learning library from NumPy.

Derivatives, by hand, one last time

Take `y = 3x² + 2x`. Its derivative is `6x + 2`, so at `x = 2` the slope is 14. You can also get that number without doing any calculus, by nudging x a little and seeing how much y moves.

```
def f(x):
    return 3 * x ** 2 + 2 * x

x = 2.0
print('%12s %18s' % ('step size', 'estimated slope'))
for h in [1.0, 0.1, 0.01, 0.0001, 1e-8, 1e-12]:
    print('%12g %18.8f' % (h, (f(x + h) - f(x)) / h))
print('\nthe exact answer is 14')
```

```
   step size    estimated slope
           1        17.00000000
         0.1        14.30000000
        0.01        14.03000000
      0.0001        14.00030000
       1e-08        13.99999974
       1e-12        14.00124461

the exact answer is 14
```

Watch Out

Why nobody differentiates numerically

Watch the last two rows. As the step gets smaller the estimate gets better, and then it gets worse again, because subtracting two nearly equal floating point numbers destroys precision. Numerical differentiation is useful for checking a gradient you computed another way, which is exactly what day 5 uses it for, and useless for training a network with millions of parameters, since it would need one forward pass per parameter.

What autograd actually does

```
import torch

x = torch.tensor(2.0, requires_grad=True)
y = 3 * x ** 2 + 2 * x

```

```
print('y =', y.item())
print('y knows it was computed:', y.grad_fn)

y.backward()
print('dy/dx at x=2 is', x.grad.item())
```

```
y = 16.0
y knows it was computed: <AddBackward0 object at 0x000001B30968E590>
dy/dx at x=2 is 14.0
```

```
import torch

# A chain of operations, and the gradient through all of it.
x = torch.tensor(1.5, requires_grad=True)
w = torch.tensor(-0.8, requires_grad=True)
b = torch.tensor(0.3, requires_grad=True)

z = w * x + b
a = torch.sigmoid(z)
loss = (a - 1.0) ** 2

print('z    %.4f' % z.item())
print('a    %.4f' % a.item())
print('loss %.4f' % loss.item())

loss.backward()
print('\ndloss/dw %.6f' % w.grad.item())
print('dloss/db %.6f' % b.grad.item())
print('dloss/dx %.6f' % x.grad.item())
```

```
z    -0.9000
a    0.2891
loss 0.5054

dloss/dw -0.438301
dloss/db -0.292201
dloss/dx 0.233761
```

Three gradients from one backward call, and none of them were derived by you. That is the whole trick. The forward pass builds a graph of operations; `backward()` walks it in reverse, multiplying local derivatives as it goes.

Checking it against the calculus

```
import torch

# The same thing worked out by hand:
#   loss = (sigmoid(wx + b) - 1)^2
#   dloss/da = 2(a - 1)
#   da/dz    = a(1 - a)
#   dz/dw    = x
x_v, w_v, b_v = 1.5, -0.8, 0.3
z_v = w_v * x_v + b_v
```

```
a_v = 1 / (1 + pow(2.718281828459045, -z_v))
by_hand = 2 * (a_v - 1) * a_v * (1 - a_v) * x_v

x = torch.tensor(x_v, requires_grad=True)
w = torch.tensor(w_v, requires_grad=True)
b = torch.tensor(b_v, requires_grad=True)
((torch.sigmoid(w * x + b) - 1.0) ** 2).backward()

print('by hand %.8f' % by_hand)
print('autograd %.8f' % w.grad.item())
print('agree:', abs(by_hand - w.grad.item()) < 1e-7)
```

```
by hand -0.43830102
autograd -0.43830103
agree: True
```

Day 2 takeaway

Day 3 How the Graph Behaves

Accumulating gradients, no_grad, detach, and the memory leak in your loop

The graph is built while the forward pass runs, which has consequences worth understanding before they bite: gradients accumulate rather than replace, some operations need to be kept out of the graph, and the graph is thrown away after you use it.

Gradients add up, and that is deliberate

```
import torch

w = torch.tensor(1.0, requires_grad=True)

for step in range(3):
    loss = (w * 2) ** 2
    loss.backward()
    print('after backward %d, w.grad = %.1f' % (step + 1, w.grad.item()))

print('\nthe gradient is 8 every time, but it is being added on')
```

```
after backward 1, w.grad = 8.0
after backward 2, w.grad = 16.0
after backward 3, w.grad = 24.0

the gradient is 8 every time, but it is being added on
```

Watch Out

This is the bug everybody writes once

Leave `optimizer.zero_grad()` out of your training loop and the gradients from every previous batch stay in `.grad`, growing without limit. The loss usually explodes within a few dozen steps, though on a small learning rate it can instead just train badly and quietly. Call `zero_grad()` every step, before `backward()`.
The behaviour is not a design mistake. It is what lets you accumulate gradients over several small batches and then step once, which is how people train large models on hardware that cannot hold a large batch.

```python
import torch

w = torch.tensor(1.0, requires_grad=True)
for step in range(3):
    if w.grad is not None:
        w.grad.zero_()
    loss = (w * 2) ** 2
    loss.backward()
    print('step %d, w.grad = %.1f' % (step + 1, w.grad.item()))
```

```
step 1, w.grad = 8.0
step 2, w.grad = 8.0
step 3, w.grad = 8.0
```

Keeping things out of the graph

```python
import torch

w = torch.tensor(2.0, requires_grad=True)

tracked = w * 3
print('tracked requires grad:', tracked.requires_grad)

with torch.no_grad():
    untracked = w * 3
print('inside no_grad     :', untracked.requires_grad)

detached = (w * 3).detach()
print('after detach       :', detached.requires_grad)

print('\nuse no_grad for evaluation, and for the parameter update')
print('itself, which must not become part of the next graph.')
```

```
tracked requires grad: True
inside no_grad     : False
after detach       : False

use no_grad for evaluation, and for the parameter update
itself, which must not become part of the next graph.
```

Situation	Use	Because
Scoring a validation set	`with torch.no_grad():`	No gradients are needed, and memory falls sharply
Updating parameters by hand	`with torch.no_grad():`	The update is bookkeeping, not part of the model
Using a value as a constant target	`.detach()`	Stops the gradient flowing back into whatever produced it
Reading a number for printing	`.item()`	Returns a plain Python float and holds no graph
Storing a running loss	`.item()` or `.detach()`	Keeping the tensor keeps the whole graph alive, which is the usual cause of a slow memory leak in a training loop

The graph is freed once you use it

```python
import torch

w = torch.tensor(1.0, requires_grad=True)
loss = (w * 2) ** 2
loss.backward()

try:
    loss.backward()
except RuntimeError as e:
    print('second backward on the same graph:')
    print(' ', str(e).split('.')[0])

w.grad.zero_()
loss = (w * 2) ** 2
loss.backward(retain_graph=True)
loss.backward()
print('\nwith retain_graph=True it works, and the gradient doubled:',
      w.grad.item())
```

```
second backward on the same graph:
  Trying to backward through the graph a second time (or directly access saved tensors after
    ↪ they have already been freed)

with retain_graph=True it works, and the gradient doubled: 16.0
```

You will almost never need `retain_graph=True`. When you think you do, the usual cause is that you built the graph outside the loop and are trying to reuse it, and the fix is to build it inside.

> **Day 3 takeaway**

Day 4 The Training Loop

Five steps written by hand, then the same five with the library

Enough machinery. Here is a complete training loop, written out in full with no library helpers at all, so that every later convenience is recognisable as a shortcut for something you have already done by hand.

Some data with a known answer

```
import torch

torch.manual_seed(0)
n = 200
X = torch.rand(n, 2) * 4 - 2                    # two features in [-2, 2]
true_w = torch.tensor([2.0, -3.0])
true_b = 0.5
y = X @ true_w + true_b + torch.randn(n) * 0.3  # with noise

print('X', tuple(X.shape), ' y', tuple(y.shape))
print('the answer we are hoping to recover: w =', true_w.tolist(),
      ' b =', true_b)
```

```
X (200, 2)  y (200,)
the answer we are hoping to recover: w = [2.0, -3.0]  b = 0.5
```

The loop

```
import torch

torch.manual_seed(0)
n = 200
X = torch.rand(n, 2) * 4 - 2
true_w = torch.tensor([2.0, -3.0])
y = X @ true_w + 0.5 + torch.randn(n) * 0.3

w = torch.zeros(2, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
lr = 0.1

for epoch in range(101):
    pred = X @ w + b                    # forward
    loss = ((pred - y) ** 2).mean()     # how wrong

    if w.grad is not None:
        w.grad.zero_()
        b.grad.zero_()
    loss.backward()                     # gradients

    with torch.no_grad():               # the update
        w -= lr * w.grad
        b -= lr * b.grad

    if epoch % 20 == 0:
        print('epoch %3d  loss %.4f  w %s  b %.3f'
```

```
              % (epoch, loss.item(),
                 [round(v, 3) for v in w.tolist()], b.item()))
```

```
epoch   0  loss 15.7814  w [0.461, -0.731]  b 0.015
epoch  20  loss 0.1056  w [2.011, -2.992]  b 0.503
epoch  40  loss 0.1045  w [2.023, -3.004]  b 0.523
epoch  60  loss 0.1045  w [2.023, -3.004]  b 0.524
epoch  80  loss 0.1045  w [2.023, -3.004]  b 0.524
epoch 100  loss 0.1045  w [2.023, -3.004]  b 0.524
```

Five lines, and every one of them has a name

Forward pass, loss, zero the gradients, backward pass, update. Every training loop you will ever write is those five steps. A hundred-line loop from a research repository is those five steps plus logging, checkpointing, learning rate scheduling and mixed precision, and if you cannot find the five steps inside it, you are reading the wrong part of the file.

The same thing with the library doing the bookkeeping

```
import torch

torch.manual_seed(0)
n = 200
X = torch.rand(n, 2) * 4 - 2
y = X @ torch.tensor([2.0, -3.0]) + 0.5 + torch.randn(n) * 0.3

model = torch.nn.Linear(2, 1)
loss_fn = torch.nn.MSELoss()
opt = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(101):
    pred = model(X).squeeze(1)
    loss = loss_fn(pred, y)
    opt.zero_grad()
    loss.backward()
    opt.step()
    if epoch % 20 == 0:
        print('epoch %3d  loss %.4f' % (epoch, loss.item()))

print('\nlearned w', [round(v, 3) for v in
                     model.weight.detach().squeeze(0).tolist()])
print('learned b %.3f' % model.bias.item())
```

```
epoch   0  loss 15.2988
epoch  20  loss 0.1051
epoch  40  loss 0.1045
epoch  60  loss 0.1045
epoch  80  loss 0.1045
epoch 100  loss 0.1045

learned w [2.023, -3.004]
learned b 0.524
```

Identical arithmetic, less typing. `nn.Linear` holds the weight and bias for you, `MSELoss` is the mean of the squared differences, and `opt.step()` is the subtraction you wrote by hand. Note `squeeze(1)`: `nn.Linear(2, 1)` returns shape `(n, 1)` and the targets are `(n,)`, which is yesterday's silent broadcast waiting to happen.

What the learning rate does

```
import torch

def run(lr, steps=60):
    torch.manual_seed(0)
    X = torch.rand(200, 2) * 4 - 2
    y = X @ torch.tensor([2.0, -3.0]) + 0.5 + torch.randn(200) * 0.3
    w = torch.zeros(2, requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    for _ in range(steps):
        loss = ((X @ w + b - y) ** 2).mean()
        if w.grad is not None:
            w.grad.zero_(); b.grad.zero_()
        loss.backward()
        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad
    return loss.item()

print('%10s %16s' % ('rate', 'loss after 60'))
for lr in [0.001, 0.01, 0.1, 0.5, 1.0, 1.05]:
    value = run(lr)
    print('%10.3f %16s'
          % (lr, 'diverged' if value != value or value > 1e6
             else '%.4f' % value))
```

```
      rate    loss after 60
     0.001          11.9438
     0.010           1.0795
     0.100           0.1045
     0.500           0.1045
     1.000         diverged
     1.050         diverged
```

Watch Out

There is a cliff, and it is close to the best value

Too small and it crawls. Too large and it does not merely go slowly, it diverges to infinity or to `nan`. The best rate is usually within a factor of two or three of the one that breaks it, which is why the learning rate is the first hyperparameter to tune and why week 3 spends a day on choosing it. If your loss becomes `nan`, lower the learning rate before you change anything else.

Day 4 takeaway

Day 5 Your First Network

Two layers on XOR, gradient checking, and why initialisation decides everything

A network is the same loop with more parameters and a nonlinearity between the layers. Building one from scratch takes about twenty lines, and it is worth doing once because it makes the failure modes in week 3 obvious rather than mysterious.

A two-layer network, entirely by hand

```
import torch

torch.manual_seed(0)

# The classic problem a single layer cannot solve.
X = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
y = torch.tensor([[0.0], [1.0], [1.0], [0.0]])       # exclusive or

W1 = torch.randn(2, 8, requires_grad=True)
b1 = torch.zeros(8, requires_grad=True)
W2 = torch.randn(8, 1, requires_grad=True)
b2 = torch.zeros(1, requires_grad=True)
params = [W1, b1, W2, b2]

for step in range(3001):
    hidden = torch.relu(X @ W1 + b1)
    out = torch.sigmoid(hidden @ W2 + b2)
    loss = -(y * out.log() + (1 - y) * (1 - out).log()).mean()

    for prm in params:
        if prm.grad is not None:
            prm.grad.zero_()
    loss.backward()
    with torch.no_grad():
        for prm in params:
            prm -= 0.5 * prm.grad

    if step % 750 == 0:
        print('step %4d  loss %.4f' % (step, loss.item()))

print('\npredictions', [round(v, 3) for v in out.detach().squeeze(1).tolist()])
print('targets    ', [0.0, 1.0, 1.0, 0.0])
```

```
step    0  loss 0.7838
step  750  loss 0.0027
step 1500  loss 0.0011
step 2250  loss 0.0007
step 3000  loss 0.0005
```

```
predictions [0.001, 1.0, 0.999, 0.0]
targets     [0.0, 1.0, 1.0, 0.0]
```

A single layer cannot solve that, because no straight line separates the two corners where the answer is 1 from the two where it is 0. The hidden layer builds a new representation in which the problem does become separable, and that is the entire justification for depth.

Proving the gradients are right

```
import torch

torch.manual_seed(0)
X = torch.randn(5, 3)
y = torch.randn(5, 1)
W = torch.randn(3, 1, requires_grad=True)

def loss_of(weight):
    return ((X @ weight - y) ** 2).mean()

loss_of(W).backward()
analytic = W.grad.clone().squeeze(1)

h = 1e-4
numeric = torch.zeros(3)
with torch.no_grad():
    for i in range(3):
        up, down = W.clone(), W.clone()
        up[i] += h
        down[i] -= h
        numeric[i] = (loss_of(up) - loss_of(down)) / (2 * h)

print('%10s %14s %14s' % ('parameter', 'analytic', 'numerical'))
for i in range(3):
    print('%10d %14.6f %14.6f' % (i, analytic[i], numeric[i]))
print('\nlargest disagreement %.2e'
      % (analytic - numeric).abs().max().item())
```

```
 parameter       analytic      numerical
         0      -2.298965      -2.298355
         1       2.040464       2.040863
         2       2.095707       2.096891

largest disagreement 1.18e-03
```

Use the two-sided difference

Note `(f(x+h) - f(x-h)) / 2h` rather than the one-sided version from day 2. The two-sided form has an error proportional to h^2 instead of h, so it is far more accurate for the same step size. When you write a custom layer or a custom loss, this check is how you find out whether your derivation is right before you spend a day wondering why the model will not learn.

Why initialisation is not a detail

```
import torch

torch.manual_seed(0)
x = torch.randn(512, 100)

print('%18s %12s %12s' % ('initial scale', 'std after 1', 'std after 8'))
for scale, label in [(1.0, 'randn'), (0.1, 'randn * 0.1'),
                     ((2.0 / 100) ** 0.5, 'He (sqrt(2/n))')]:
    h = x
    first = None
    for layer in range(8):
        W = torch.randn(100, 100) * scale
        h = torch.relu(h @ W)
        if layer == 0:
            first = h.std().item()
    print('%18s %12.4f %12.6f' % (label, first, h.std().item()))
```

```
     initial scale  std after 1  std after 8
             randn       5.8124 7294867.000000
       randn * 0.1       0.5768     0.051808
    He (sqrt(2/n))       0.8290     0.653480
```

With weights that are slightly too small the signal dies away to nothing after a few layers, and a signal of zero has a gradient of zero. Slightly too large and it grows without limit. The He initialisation in the last row is scaled specifically so that the variance of the activations survives a ReLU, which is why it is the default for every modern network and why PyTorch already applies something like it for you.

Day 5 takeaway

Day 6 Devices, Datasets and Loaders

Getting real data into batches without writing a loop

Two practical matters before the first real model: where tensors live, and how you get data into them without writing a loop over a list.

Devices

```
import torch

print('CUDA available:', torch.cuda.is_available())
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('this course runs on:', device)

x = torch.ones(3)
x = x.to(device)
print('tensor device:', x.device)
```

```
10
11 model = torch.nn.Linear(3, 1).to(device)
12 print('model runs:', model(x).item() is not None)
```

```
CUDA available: False
this course runs on: cpu
tensor device: cpu
model runs: True
```

Watch Out

Everything has to be on the same device

A model on the GPU and a batch on the CPU produces `Expected all tensors to be on the same device`, which is the second most common error in PyTorch after shape mismatches. Write `device` once at the top, call `.to(device)` on the model once, and on each batch inside the loop. Everything on this course is written so it runs on a CPU, because the point is to understand the mechanics rather than to wait.

Datasets and loaders

```
1 import torch
2 from torch.utils.data import TensorDataset, DataLoader
3
4 torch.manual_seed(0)
5 X = torch.randn(10, 3)
6 y = torch.arange(10.0)
7
8 loader = DataLoader(TensorDataset(X, y), batch_size=4, shuffle=True)
9 print('10 examples, batch size 4, so %d batches' % len(loader))
10 for i, (xb, yb) in enumerate(loader):
11     print('batch %d  x %s  y %s'
12           % (i, tuple(xb.shape), [int(v) for v in yb.tolist()]))
```

```
10 examples, batch size 4, so 3 batches
batch 0  x (4, 3)  y [7, 9, 2, 6]
batch 1  x (4, 3)  y [1, 8, 4, 0]
batch 2  x (2, 3)  y [3, 5]
```

The final batch has two rows rather than four. That is normal, and it is why you should average a metric over examples rather than over batches: the last batch is smaller and averaging batch means gives it too much weight.

Writing your own Dataset

```
1 import torch
2 from torch.utils.data import Dataset, DataLoader
3
4 class SquaresDataset(Dataset):
5     """The two methods every Dataset must have."""
6
```

```
    def __init__(self, n):
        self.n = n

    def __len__(self):
        return self.n

    def __getitem__(self, i):
        x = torch.tensor([float(i)])
        return x, x ** 2

loader = DataLoader(SquaresDataset(7), batch_size=3)
for xb, yb in loader:
    print('x', [int(v) for v in xb.squeeze(1).tolist()],
          ' y', [int(v) for v in yb.squeeze(1).tolist()])
```

```
x [0, 1, 2]  y [0, 1, 4]
x [3, 4, 5]  y [9, 16, 25]
x [6]  y [36]
```

__getitem__ runs once per example, so keep it cheap

It is the right place to read one image from disk and apply the transformations for it. It is the wrong place to load a CSV, open a database connection or compute statistics over the whole dataset, all of which belong in `__init__`. A slow `__getitem__` is the usual reason a GPU sits idle while the loader struggles to keep up, and `num_workers` exists to run several copies of it in parallel.

Real data, for the first time

```
import torch
from torchvision import datasets, transforms

to_tensor = transforms.ToTensor()
train = datasets.MNIST('data', train=True, download=True,
                       transform=to_tensor)
test = datasets.MNIST('data', train=False, download=True,
                      transform=to_tensor)

print('training images', len(train))
print('test images    ', len(test))
image, label = train[0]
print('one example: shape %s, label %d' % (tuple(image.shape), label))
print('pixel range %.1f to %.1f' % (image.min(), image.max()))
```

```
training images 60000
test images     10000
one example: shape (1, 28, 28), label 5
pixel range 0.0 to 1.0
```

```
import torch
from torchvision import datasets, transforms
```

```

train = datasets.MNIST('data', train=True, download=True,
                       transform=transforms.ToTensor())
image, label = train[0]

ramp = ' .:-=+*#%@'
print('label:', label)
pixels = image.squeeze(0)
for row in pixels[::1]:
    print(''.join(ramp[min(9, int(v * 9.999))] for v in row.tolist()))
```

```
label: 5









                =+*.*@@=
         ..-**@@@@@%*@@#:
        .@@@@@@@@@@---:.
        %@@@@@##@@
        -*=@@%  .*
           *@-
           +@#
            #@:
            .@%*=
             -@@@=
               .#@@+.
                 -@@#
                  @@@:
               .+#@@%
             .+%@@@@#
            =%@@@@#-
          :%@@@@#-
        *%@@@@#-
     :*%@@@@@+
     +@@@%++
```

Day 6 takeaway

Day 7 MNIST End to End

A complete model, its confusion matrix, and the baseline it has to beat

Everything from this week, applied to real digits, with a proper training loop and an honest evaluation.

The model

```
import torch

model = torch.nn.Sequential(
```

```
    torch.nn.Flatten(),                 # (batch, 1, 28, 28) → (batch, 784)
    torch.nn.Linear(784, 128),
    torch.nn.ReLU(),
    torch.nn.Linear(128, 10),           # ten digits, raw scores
)
print(model)
total = sum(prm.numel() for prm in model.parameters())
print('\nparameters: %d' % total)
for name, prm in model.named_parameters():
    print('  %-10s %-16s %d' % (name, tuple(prm.shape), prm.numel()))
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=128, bias=True)
  (2): ReLU()
  (3): Linear(in_features=128, out_features=10, bias=True)
)

parameters: 101770
  1.weight   (128, 784)       100352
  1.bias     (128,)           128
  3.weight   (10, 128)        1280
  3.bias     (10,)            10
```

Training it

```
import time
import torch
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

torch.manual_seed(0)
tf = transforms.ToTensor()
train_full = datasets.MNIST('data', train=True, download=True, transform=tf)
test_full = datasets.MNIST('data', train=False, download=True, transform=tf)

# A subset, so every example on this page finishes while you watch it.
train = Subset(train_full, range(8000))
test = Subset(test_full, range(2000))
train_loader = DataLoader(train, batch_size=64, shuffle=True)
test_loader = DataLoader(test, batch_size=256)

model = torch.nn.Sequential(torch.nn.Flatten(),
                            torch.nn.Linear(784, 128), torch.nn.ReLU(),
                            torch.nn.Linear(128, 10))
loss_fn = torch.nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=0.1)

def accuracy(loader):
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in loader:
```

```
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen

start = time.time()
for epoch in range(1, 6):
    model.train()
    running = 0.0
    for xb, yb in train_loader:
        opt.zero_grad()
        loss = loss_fn(model(xb), yb)
        loss.backward()
        opt.step()
        running += loss.item() * yb.numel()
    print('epoch %d  train loss %.4f  test accuracy %.4f'
          % (epoch, running / len(train), accuracy(test_loader)))
print('\n%.1f seconds on the CPU' % (time.time() - start))
```

```
epoch 1  train loss 1.0575  test accuracy 0.8400
epoch 2  train loss 0.4176  test accuracy 0.8650
epoch 3  train loss 0.3342  test accuracy 0.8650
epoch 4  train loss 0.2965  test accuracy 0.8790
epoch 5  train loss 0.2691  test accuracy 0.8785

5.3 seconds on the CPU
```

What it gets wrong

```
import torch
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

torch.manual_seed(0)
tf = transforms.ToTensor()
train = Subset(datasets.MNIST('data', train=True, download=True,
                              transform=tf), range(8000))
test = Subset(datasets.MNIST('data', train=False, download=True,
                             transform=tf), range(2000))

model = torch.nn.Sequential(torch.nn.Flatten(),
                            torch.nn.Linear(784, 128), torch.nn.ReLU(),
                            torch.nn.Linear(128, 10))
opt = torch.optim.SGD(model.parameters(), lr=0.1)
loss_fn = torch.nn.CrossEntropyLoss()
for _ in range(5):
    for xb, yb in DataLoader(train, batch_size=64, shuffle=True):
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()

model.eval()
confusion = torch.zeros(10, 10, dtype=torch.long)
with torch.no_grad():
```

```
    for xb, yb in DataLoader(test, batch_size=256):
        for t, p in zip(yb, model(xb).argmax(1)):
            confusion[t, p] += 1

print('    ' + ''.join('%5d' % i for i in range(10)))
for i, row in enumerate(confusion):
    print('%2d  ' % i + ''.join('%5d' % v for v in row.tolist()))

print('\nconfusions of three or more:')
for i in range(10):
    for j in range(10):
        if i != j and confusion[i, j] >= 3:
            print('  a %d read as a %d, %d times' % (i, j, confusion[i, j]))
```

```
       0    1    2    3    4    5    6    7    8    9
 0   171    0    0    1    0    1    2    0    0    0
 1     0  228    0    1    1    0    2    0    2    0
 2     2    3  186    2    3    0    5    7    9    2
 3     0    0    2  174    0   16    1    8    5    1
 4     1    0    1    0  201    0    5    1    1    7
 5     6    0    1    7    4  150    1    3    5    2
 6     3    1    2    0    7    4  159    1    1    0
 7     1    5    9    0    3    0    0  178    1    8
 8     2    2    2    7    4    7    3    5  157    3
 9     1    0    0    5   15    2    0    9    2  160

confusions of three or more:
  a 2 read as a 1, 3 times
  a 2 read as a 4, 3 times
  a 2 read as a 6, 5 times
  a 2 read as a 7, 7 times
  a 2 read as a 8, 9 times
  a 3 read as a 5, 16 times
  a 3 read as a 7, 8 times
  a 3 read as a 8, 5 times
  a 4 read as a 6, 5 times
  a 4 read as a 9, 7 times
  a 5 read as a 0, 6 times
... (20 more lines)
```

The mistakes are the ones a person would make. Threes and fives are the worst pair by a distance, and they differ only in whether the top stroke closes to the left or the right. Nines and fours go both ways once the loop is closed, twos and sevens share a diagonal, and eights collect confusions from everything with two curves in it. If your confusion matrix looks random rather than sympathetic, suspect a bug in how labels are lining up with images before you suspect the model.

A baseline that is not a network

```
import torch
from torchvision import datasets, transforms
from sklearn.linear_model import LogisticRegression
import time

tf = transforms.ToTensor()
```

```
train = datasets.MNIST('data', train=True, download=True, transform=tf)
test = datasets.MNIST('data', train=False, download=True, transform=tf)

Xtr = train.data[:8000].reshape(8000, -1).numpy() / 255.0
ytr = train.targets[:8000].numpy()
Xte = test.data[:2000].reshape(2000, -1).numpy() / 255.0
yte = test.targets[:2000].numpy()

start = time.time()
lr = LogisticRegression(max_iter=200).fit(Xtr, ytr)
print('logistic regression %.4f in %.1fs'
      % (lr.score(Xte, yte), time.time() - start))
```

```
logistic regression 0.8700 in 4.0s
```

Keep score honestly from the first week

A one-hidden-layer network on 8,000 digits beats plain logistic regression, but not by as much as the effort suggests it should. That gap is the thing the rest of this course is about: convolution in week 5, which knows that pixels have neighbours, and depth done properly in week 6, which is what turns a few points into a lot of points. Fit the cheap model every time, so you know what your complicated one is actually buying.

Before you move on

1. Print the shape of anything you are unsure about, especially between a model's output and the target.
2. Call `zero_grad()` every step.
3. Put the model in `train()` mode for training and `eval()` mode for evaluation. It changes nothing today and changes a great deal from week 4.
4. Wrap evaluation in `torch.no_grad()`.
5. Log `loss.item()`, never the loss tensor.
6. Fit a non-network baseline and write the number down.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. A tensor of shape `(32, 1)` is added to one of shape `(32,)`. What comes out?

   a) A shape `(32,)` tensor

   b) A shape `(32, 32)` tensor, because broadcasting aligns from the right and stretches both

c) An error, because the shapes differ

d) A shape `(32, 1)` tensor

2. What does `loss.backward()` actually do to the parameters?

   a) Frees the memory used by the graph and nothing else

   b) Updates them by one step of gradient descent

   c) Adds to each parameter's `.grad`, leaving the values alone

   d) Replaces each parameter's `.grad`

3. Why must `optimizer.zero_grad()` be called each iteration?

   a) Because gradients accumulate, so without it every step uses the sum of all gradients seen so far

   b) Because the optimiser caches the previous loss

   c) To release GPU memory

   d) To reset the learning rate schedule

4. You append the running loss to a list as `total += loss` rather than `loss.item()`. What goes wrong?

   a) The result is rounded to an integer

   b) The loss is computed twice

   c) Each loss keeps its whole graph alive, so memory grows through the epoch and eventually the run dies

   d) Nothing, the two are equivalent

5. What is `detach()` for?

   a) Moving a tensor to the CPU

   b) Returning a tensor that shares the same data but is cut off from the graph, so no gradient flows back through it

   c) Making a copy of the data

   d) Turning off dropout

6. A two-layer network on XOR is initialised with every weight at zero. What happens?

   a) Only the bias terms train

   b) It trains normally but more slowly

   c) The loss is undefined

   d) Every hidden unit computes and receives the same thing forever, so the network never breaks symmetry and never learns the task

7. What does a `DataLoader` give you that indexing a `Dataset` in a loop does not?

   a) Gradient accumulation

   b) A validation split

   c) Automatic normalisation

   d) Batching, shuffling and optional worker processes, without you writing any of it

8. Why does week 1 insist on a baseline before reporting MNIST accuracy?

a) Because the library requires a reference model

b) Because a number with nothing to compare it against says nothing about whether the network helped

c) To check the data loaded in the right order

d) To warm up the caches

Answers:1b, 2c, 3a, 4c, 5b, 6d, 7d, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 2

Building Models with nn.Module

Week 2

OBJECTIVES

Architectures that are not a straight line, losses that match the task, and a training script you can trust and come back to.

Week 1 built a training loop. This week builds everything around it that decides whether the result is worth anything: models that branch and rejoin, losses that specify the right thing, splits that do not flatter you, and a script that saves enough for somebody else to repeat it. It also covers the two mistakes that cost the most time, a parameter that was never registered and a batch that was never overfitted as a test.

Day 1 nn.Module

Custom layers, registered parameters, and the list that silently does not train

`nn.Sequential` is fine until the moment your model does anything other than pass one thing straight through, which is almost immediately. A residual connection, two inputs, a branch that rejoins later: none of those fit in a list. `nn.Module` does.

The smallest possible module

```
import torch
from torch import nn

class MyLinear(nn.Module):
    def __init__(self, n_in, n_out):
        super().__init__()                  # never forget this line
        self.weight = nn.Parameter(torch.randn(n_out, n_in) * 0.1)
        self.bias = nn.Parameter(torch.zeros(n_out))

    def forward(self, x):
        return x @ self.weight.T + self.bias

layer = MyLinear(4, 3)
x = torch.randn(2, 4)
print('output shape', tuple(layer(x).shape))
print('\nparameters it registered:')
for name, prm in layer.named_parameters():
    print('  %-8s %s' % (name, tuple(prm.shape)))
```

```
output shape (2, 3)

parameters it registered:
  weight   (3, 4)
  bias     (3,)
```

What nn.Parameter is for

A plain tensor assigned to `self` is invisible. Wrap it in `nn.Parameter` and the module registers it, which means it appears in `model.parameters()`, gets passed to the optimiser, moves with `.to(device)` and is saved in `state_dict()`. Forget the wrapper and the tensor simply never trains, silently, while everything else does.

```
import torch
from torch import nn

class Forgetful(nn.Module):
    def __init__(self):
        super().__init__()
        self.good = nn.Parameter(torch.zeros(3))
        self.bad = torch.zeros(3)               # not a Parameter

m = Forgetful()
print('registered parameters:', [n for n, _ in m.named_parameters()])
print('the optimiser would update %d tensor(s)'
      % len(list(m.parameters())))
print('\nself.bad exists and is used in forward, but never learns.')
```

```
registered parameters: ['good']
the optimiser would update 1 tensor(s)

self.bad exists and is used in forward, but never learns.
```

Modules contain modules

```
import torch
from torch import nn

class Block(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.fc = nn.Linear(size, size)
        self.act = nn.ReLU()

    def forward(self, x):
        return self.act(self.fc(x))

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.stem = nn.Linear(784, 64)
        self.blocks = nn.ModuleList([Block(64) for _ in range(3)])
```

```
        self.head = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.relu(self.stem(x.flatten(1)))
        for block in self.blocks:
            x = block(x)
        return self.head(x)

net = Net()
print('parameters %d' % sum(p.numel() for p in net.parameters()))
print('output', tuple(net(torch.randn(5, 1, 28, 28)).shape))
print('\nnamed modules:')
for name, mod in net.named_children():
    print('  %-8s %s' % (name, type(mod).__name__))
```

```
parameters 63370
output (5, 10)

named modules:
  stem     Linear
  blocks   ModuleList
  head     Linear
```

Watch Out

A plain Python list hides its modules

`self.blocks = [Block(64) for _ in range(3)]` looks identical and is broken in the same way as a plain tensor: the modules in it are not registered, so their parameters do not reach the optimiser and do not move to the GPU. Use `nn.ModuleList` when you need a list and `nn.ModuleDict` when you need a dictionary. The symptom is a model that trains, slowly, using only the layers you happened to assign directly.

A residual connection, which Sequential cannot express

```
import torch
from torch import nn

class Residual(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.fc1 = nn.Linear(size, size)
        self.fc2 = nn.Linear(size, size)

    def forward(self, x):
        h = torch.relu(self.fc1(x))
        h = self.fc2(h)
        return torch.relu(x + h)        # the input rejoins the output

block = Residual(16)
x = torch.randn(4, 16)
print('in ', tuple(x.shape), ' out', tuple(block(x).shape))
```

```
print('\nthat one addition is the idea week 6 is built on.')
```

```
in  (4, 16)  out (4, 16)

that one addition is the idea week 6 is built on.
```

Day 1 takeaway

Day 2 Losses

Logits not probabilities, stability at the extremes, and weighting

The loss function is where you tell the model what counts as wrong. Choosing the wrong one is not a tuning mistake, it is a specification mistake, and the model will optimise exactly what you asked for.

The three you will use most

Task	Final layer	Loss	Target format
Binary classification	1 unit, no activation	`BCEWithLogitsLoss`	float 0.0 or 1.0
Multi-class, one label	C units, no activation	`CrossEntropyLoss`	integer class index
Multi-label	C units, no activation	`BCEWithLogitsLoss`	float vector of 0s and 1s
Regression	1 unit, no activation	`MSELoss` or `L1Loss`	float
Regression with outliers	1 unit, no activation	`HuberLoss`	float

Watch Out

Notice that no row has a softmax or a sigmoid in it

`CrossEntropyLoss` applies `log_softmax` internally, and `BCEWithLogitsLoss` applies the sigmoid. Doing it yourself first is not a harmless duplication: it makes the gradients smaller and the arithmetic less stable, and it trains a worse model without raising anything. The `WithLogits` in the name is the library telling you it has already been handled.

```
import torch
from torch import nn

logits = torch.tensor([[2.0, 0.5, -1.0]])
target = torch.tensor([0])

correct = nn.CrossEntropyLoss()(logits, target)

# the same thing done by hand, to show what it contains
log_probs = logits - logits.exp().sum().log()
by_hand = -log_probs[0, target[0]]

print('CrossEntropyLoss %.6f' % correct.item())
print('by hand          %.6f' % by_hand.item())
```

```

# and the mistake
wrong = nn.CrossEntropyLoss()(torch.softmax(logits, dim=1), target)
print('\nwith a softmax applied first: %.6f' % wrong.item())
print('different, smaller gradients, and no error raised.')
```

```
CrossEntropyLoss 0.241311
by hand          0.241311

with a softmax applied first: 0.701717
different, smaller gradients, and no error raised.
```

Numerical stability, demonstrated

```
import torch
from torch import nn

logits = torch.tensor([[-30.0]], requires_grad=True)
target = torch.tensor([[1.0]])

stable = nn.BCEWithLogitsLoss()(logits, target)
print('BCEWithLogitsLoss   %.6f' % stable.item())

probs = torch.sigmoid(logits)
naive = nn.BCELoss()(probs, target)
print('sigmoid then BCELoss %.6f' % naive.item())
print('agree here, at a probability of %.3e' % probs.item())

extreme = torch.tensor([[-200.0]], requires_grad=True)
print('\nnow at -200, where the sigmoid underflows to exactly 0:')
print('  with logits %.4f' % nn.BCEWithLogitsLoss()(extreme, target).item())
print('  the naive way %.4f'
      % nn.BCELoss()(torch.sigmoid(extreme), target).item())
print('\n100 is BCELoss clamping. The true loss is 200, so the')
print('gradient is wrong exactly where the model is most wrong.')
```

```
BCEWithLogitsLoss   30.000000
sigmoid then BCELoss 30.000000
agree here, at a probability of 9.358e-14

now at -200, where the sigmoid underflows to exactly 0:
  with logits 200.0000
  the naive way 100.0000

100 is BCELoss clamping. The true loss is 200, so the
gradient is wrong exactly where the model is most wrong.
```

At a logit of -30 the two agree perfectly, which is worth noticing: this is not a problem you will see in a toy example. It appears when a model becomes confident, which is to say late in training on the examples it has learned best, and it appears as a loss that stops decreasing for no visible reason.

Weighting classes

```
import torch
from torch import nn

# Ten classes where class 0 is nine times more common than the rest.
torch.manual_seed(0)
counts = torch.tensor([900.0] + [100.0] * 9)
weights = counts.sum() / (len(counts) * counts)
print('class counts ', [int(c) for c in counts.tolist()])
print('loss weights ', [round(w, 3) for w in weights.tolist()])

logits = torch.randn(4, 10)
target = torch.tensor([0, 0, 0, 7])
print('\nunweighted loss %.4f' % nn.CrossEntropyLoss()(logits, target).item())
print('weighted loss   %.4f'
      % nn.CrossEntropyLoss(weight=weights)(logits, target).item())
```

```
class counts  [900, 100, 100, 100, 100, 100, 100, 100, 100, 100]
loss weights  [0.2, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8, 1.8]

unweighted loss 2.7360
weighted loss   3.0033
```

Weighting makes a mistake on a rare class cost more, which is one of three ways to handle imbalance. The others are resampling the data and leaving the loss alone but moving the decision threshold afterwards. The third is usually the best and the least used, because it does not require retraining anything.

Regression losses are not interchangeable

```
import torch
from torch import nn

pred = torch.zeros(5)
target = torch.tensor([0.1, -0.2, 0.05, 0.0, 8.0])   # one outlier

print('%-14s %10s' % ('loss', 'value'))
for name, fn in [('MSELoss', nn.MSELoss()),
                 ('L1Loss', nn.L1Loss()),
                 ('HuberLoss', nn.HuberLoss(delta=1.0))]:
    print('%-14s %10.4f' % (name, fn(pred, target).item()))

print('\ngradient contributed by the outlier alone:')
for name, fn in [('MSELoss', nn.MSELoss()),
                 ('L1Loss', nn.L1Loss()),
                 ('HuberLoss', nn.HuberLoss(delta=1.0))]:
    p_ = torch.zeros(5, requires_grad=True)
    fn(p_, target).backward()
    print('  %-12s %.4f' % (name, p_.grad[4].item()))
```

```
loss              value
```

```
MSELoss           12.8105
L1Loss             1.6700
HuberLoss          1.5052

gradient contributed by the outlier alone:
  MSELoss      -3.2000
  L1Loss       -0.2000
  HuberLoss    -0.2000
```

Day 2 takeaway

Day 3 Splits and Early Stopping

Watching a network memorise, and keeping the weights that were best

A model that scores well on the data it was fitted to has told you nothing. This is the same discipline as any other machine learning, with one addition: a deep network can memorise a training set completely, so the gap between the two numbers is usually larger and always more important.

Three splits, not two

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
print('train %d   validation %d   test %d'
      % (len(train_set), len(val_set), len(test_set)))
print()
print('train      : gradients are computed from it')
print('validation : you look at it after every epoch and make decisions')
print('test       : you look at it once, at the end')
```

```
train 8000   validation 2000   test 2000

train      : gradients are computed from it
validation : you look at it after every epoch and make decisions
test       : you look at it once, at the end
```

The validation set is training data too, in the way that matters

Every time you stop training because validation loss rose, pick an architecture because it validated better, or tune a learning rate against it, you have used the validation set to make a choice. Do that fifty times and the validation score is optimistic in exactly the way a training score is. That is what the test set is held back for, and why it is worth almost nothing after you have looked at it twice.

Watching a model memorise

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.1, train=None, val=None, log=True):
    """The five steps from week 1, wrapped up so the pages stay short."""
    train = train if train is not None else DataLoader(train_set,
                                                       batch_size=64,
                                                       shuffle=True)
    val = val if val is not None else DataLoader(val_set, batch_size=256)
    loss_fn = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr=lr)
    history = []
    for epoch in range(1, epochs + 1):
        model.train()
        seen = total = 0
        for xb, yb in train:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            total += loss.item() * yb.numel()
            seen += yb.numel()
        model.eval()
        right = count = 0
        with torch.no_grad():
            for xb, yb in val:
                right += (model(xb).argmax(1) == yb).sum().item()
                count += yb.numel()
        history.append((total / seen, right / count))
        if log:
            print('epoch %d  train loss %.4f  val accuracy %.4f'
                  % (epoch, total / seen, right / count))
```

```
    return history
torch.manual_seed(0)

# A deliberately oversized model on a deliberately tiny training set.
small = Subset(train_all, range(300))
loader = DataLoader(small, batch_size=32, shuffle=True)

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 512), nn.ReLU(),
                      nn.Linear(512, 512), nn.ReLU(), nn.Linear(512, 10))
print('parameters %d for %d training images\n'
      % (sum(p.numel() for p in model.parameters()), len(small)))

loss_fn = nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=0.1)
val_loader = DataLoader(val_set, batch_size=256)

print('%6s %14s %14s %14s' % ('epoch', 'train loss', 'train acc', 'val acc'))
for epoch in range(1, 41):
    model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()
    if epoch % 8 == 0 or epoch == 1:
        model.eval()
        with torch.no_grad():
            tr_right = sum((model(xb).argmax(1) == yb).sum().item()
                           for xb, yb in loader)
            tr_loss = sum(loss_fn(model(xb), yb).item() * yb.numel()
                          for xb, yb in loader) / len(small)
            va_right = sum((model(xb).argmax(1) == yb).sum().item()
                           for xb, yb in val_loader)
        print('%6d %14.4f %14.4f %14.4f'
              % (epoch, tr_loss, tr_right / len(small),
                 va_right / len(val_set)))
```

```
parameters 669706 for 300 training images

 epoch     train loss      train acc        val acc
     1         2.2242         0.1833         0.1615
     8         0.7274         0.8000         0.6625
    16         0.1882         0.9633         0.7735
    24         0.0565         0.9967         0.8020
    32         0.0263         1.0000         0.8095
    40         0.0162         1.0000         0.8090
```

Training accuracy reaches 1.0 and the training loss goes to almost nothing, while validation accuracy stops improving long before that. The model has not learned what a digit is, it has learned these three hundred digits. Week 4 is entirely about the tools for preventing this.

Early stopping, and keeping the right weights

```
import torch
```

```
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.1, train=None, val=None, log=True):
    """The five steps from week 1, wrapped up so the pages stay short."""
    train = train if train is not None else DataLoader(train_set,
                                                       batch_size=64,
                                                       shuffle=True)
    val = val if val is not None else DataLoader(val_set, batch_size=256)
    loss_fn = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr=lr)
    history = []
    for epoch in range(1, epochs + 1):
        model.train()
        seen = total = 0
        for xb, yb in train:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            total += loss.item() * yb.numel()
            seen += yb.numel()
        model.eval()
        right = count = 0
        with torch.no_grad():
            for xb, yb in val:
                right += (model(xb).argmax(1) == yb).sum().item()
                count += yb.numel()
        history.append((total / seen, right / count))
        if log:
            print('epoch %d  train loss %.4f  val accuracy %.4f'
                  % (epoch, total / seen, right / count))
    return history
import copy
torch.manual_seed(0)

small = Subset(train_all, range(300))
loader = DataLoader(small, batch_size=32, shuffle=True)
val_loader = DataLoader(val_set, batch_size=256)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 512), nn.ReLU(),
                      nn.Linear(512, 10))
loss_fn = nn.CrossEntropyLoss()
opt = torch.optim.SGD(model.parameters(), lr=0.1)

```

```python
best_loss, best_epoch, best_state, patience = float('inf'), 0, None, 8
for epoch in range(1, 61):
    model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()
    model.eval()
    with torch.no_grad():
        vloss = sum(loss_fn(model(xb), yb).item() * yb.numel()
                    for xb, yb in val_loader) / len(val_set)
    if vloss < best_loss:
        best_loss, best_epoch = vloss, epoch
        best_state = copy.deepcopy(model.state_dict())
    elif epoch - best_epoch >= patience:
        print('stopped at epoch %d' % epoch)
        break

print('best validation loss %.4f at epoch %d' % (best_loss, best_epoch))
model.load_state_dict(best_state)
with torch.no_grad():
    right = sum((model(xb).argmax(1) == yb).sum().item()
                for xb, yb in val_loader)
print('restored model accuracy %.4f' % (right / len(val_set)))
```

```
stopped at epoch 27
best validation loss 0.6653 at epoch 19
restored model accuracy 0.7945
```

Watch Out

Stopping is not the same as restoring

Patience means you keep training for several epochs after the best one, so the weights you are holding when the loop breaks are worse than the ones you had. Saving `state_dict()` at the best epoch and loading it back is what makes early stopping actually work. `copy.deepcopy` matters as well: `state_dict()` returns references to the live tensors, so without the copy your saved best is overwritten by the next optimiser step.

Day 3 takeaway

Day 4 Saving and Reproducing

State dictionaries, checkpoints, and what a seed does not fix

Saving a model sounds trivial and has three separate traps in it: what you save, what you need alongside it, and what happens when the code moves on.

Save the state dictionary, not the model

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
torch.manual_seed(0)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 32), nn.ReLU(),
                      nn.Linear(32, 10))

state = model.state_dict()
print('state_dict entries:')
for key, value in state.items():
    print('  %-12s %s' % (key, tuple(value.shape)))

torch.save(state, 'model.pt')

fresh = nn.Sequential(nn.Flatten(), nn.Linear(784, 32), nn.ReLU(),
                      nn.Linear(32, 10))
fresh.load_state_dict(torch.load('model.pt', weights_only=True))

x = torch.randn(3, 1, 28, 28)
print('\nsame predictions after reload:',
      bool(torch.allclose(model(x), fresh(x))))
```

```
state_dict entries:
  1.weight     (32, 784)
  1.bias       (32,)
  3.weight     (10, 32)
  3.bias       (10,)

same predictions after reload: True
```

Why not torch.save(model)

Saving the whole object pickles a reference to your class, so the file only loads if that class is importable from the same module path. Rename the file, move the class, or open it in a different project and it breaks. A state dictionary is just named tensors, so it loads anywhere you can construct the same architecture. Pass `weights_only=True` when loading: without it, `torch.load` will execute arbitrary code from the file, which matters the moment you download somebody else's checkpoint.

A checkpoint has more in it than weights

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
torch.manual_seed(0)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 32), nn.ReLU(),
                      nn.Linear(32, 10))
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

# one step, so the optimiser has state worth keeping
loss = nn.CrossEntropyLoss()(model(torch.randn(4, 1, 28, 28)),
                             torch.tensor([1, 2, 3, 4]))
loss.backward()
opt.step()

checkpoint = {
    'epoch': 7,
    'model': model.state_dict(),
    'optimizer': opt.state_dict(),
    'best_val_loss': 0.2431,
    'torch_version': torch.__version__,
}
torch.save(checkpoint, 'checkpoint.pt')

loaded = torch.load('checkpoint.pt', weights_only=False)
print('keys:', sorted(loaded.keys()))
print('resuming from epoch', loaded['epoch'])
print('optimiser state groups:', len(loaded['optimizer']['param_groups']))
print('Adam keeps running averages per parameter, and losing them')
print('makes the first few steps after a resume much worse.')
```

```
keys: ['best_val_loss', 'epoch', 'model', 'optimizer', 'torch_version']
resuming from epoch 7
optimiser state groups: 1
Adam keeps running averages per parameter, and losing them
makes the first few steps after a resume much worse.
```

Reproducibility, honestly

```python
import torch

def two_runs(seeded):
```

```
    outs = []
    for _ in range(2):
        if seeded:
            torch.manual_seed(1234)
        model = torch.nn.Linear(4, 2)
        outs.append(model.weight.detach().clone())
    return torch.allclose(outs[0], outs[1])

print('two models, no seed  : identical?', two_runs(False))
print('two models, seeded   : identical?', two_runs(True))
```

```
two models, no seed  : identical? False
two models, seeded   : identical? True
```

Source of variation	Controlled by	Fully solved?
Weight initialisation	`torch.manual_seed`	Yes
Shuffling and augmentation	the same seed, plus a `DataLoader` generator	Yes
Dropout masks	`torch.manual_seed`	Yes
Thread count and reduction order	`torch.set_num_threads`	Mostly
GPU kernel selection	`torch.use_deterministic_algorithms`	At a real cost in speed
A different library version	pinning versions	Only by pinning

Watch Out

Seeded is not the same as reproducible on another machine

Floating point addition is not associative, so a machine that splits a sum across sixteen threads gets a slightly different answer from one that splits it across four. Over thousands of steps those differences compound into visibly different final numbers. Seeds make a run repeatable on your machine, which is what you need for debugging. Matching somebody else's published figure exactly usually needs their hardware as well, and this is why every page on this course pins the thread count and still tells you to expect close rather than identical.

Day 4 takeaway

Day 5 Habits That Save Days

Parameter budgets, the single batch test, and logging a run

Some habits that turn a training script from something you babysit into something you can leave running and trust afterwards.

Count your parameters before you train

```
import torch
```

```
from torch import nn

def summarise(model, input_shape):
    total = sum(p.numel() for p in model.parameters())
    trainable = sum(p.numel() for p in model.parameters()
                    if p.requires_grad)
    x = torch.zeros(1, *input_shape)
    print('%-26s %12s %10s' % ('layer', 'output', 'params'))
    for name, mod in model.named_children():
        x = mod(x)
        n = sum(p.numel() for p in mod.parameters())
        print('%-26s %12s %10d'
              % ('%s (%s)' % (name, type(mod).__name__),
                 str(tuple(x.shape[1:])), n))
    print('%-26s %12s %10d' % ('TOTAL', '', total))
    print('trainable %d, frozen %d' % (trainable, total - trainable))
    print('at 4 bytes each that is %.1f MB of weights' % (total * 4 / 1e6))

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
                      nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 10))
summarise(model, (1, 28, 28))
```

```
layer                          output     params
0 (Flatten)                    (784,)          0
1 (Linear)                     (256,)     200960
2 (ReLU)                       (256,)          0
3 (Linear)                     (128,)      32896
4 (ReLU)                       (128,)          0
5 (Linear)                      (10,)       1290
TOTAL                                     235146
trainable 235146, frozen 0
at 4 bytes each that is 0.9 MB of weights
```

Two useful sanity checks come out of that number. If your parameter count is far larger than your example count, expect to fight overfitting all week. And memory during training is roughly four times the weights, because the gradients, and the optimiser's two running averages if you use Adam, are all the same size as the weights themselves.

Overfit one batch before you trust anything

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
```

```
test_set = Subset(test_all, range(2000))
torch.manual_seed(0)
xb, yb = next(iter(DataLoader(train_set, batch_size=16, shuffle=True)))

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 64), nn.ReLU(),
                      nn.Linear(64, 10))
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

for step in range(201):
    opt.zero_grad()
    loss = loss_fn(model(xb), yb)
    loss.backward()
    opt.step()
    if step % 50 == 0:
        print('step %3d  loss %.6f  accuracy %.3f'
              % (step, loss.item(),
                 (model(xb).argmax(1) == yb).float().mean().item()))
```

```
step   0  loss 2.280022  accuracy 0.375
step  50  loss 0.056201  accuracy 1.000
step 100  loss 0.009544  accuracy 1.000
step 150  loss 0.004860  accuracy 1.000
step 200  loss 0.003012  accuracy 1.000
```

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
torch.manual_seed(0)
xb, yb = next(iter(DataLoader(train_set, batch_size=16, shuffle=True)))

# The same test on a model with a bug: the labels are shuffled, so there
# is no relationship left to learn.
shuffled = yb[torch.randperm(len(yb))]
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 64), nn.ReLU(),
                      nn.Linear(64, 10))
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
for step in range(201):
    opt.zero_grad()
    loss_fn(model(xb), shuffled).backward()
    opt.step()
```

```
print('with randomised labels, final loss %.6f'
      % loss_fn(model(xb), shuffled).item())
print('it still memorises 16 examples, which is the point:')
print('the test proves your wiring works, not that your data means anything.')
```

```
with randomised labels, final loss 0.004961
it still memorises 16 examples, which is the point:
the test proves your wiring works, not that your data means anything.
```

Log what you will want later

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.1, train=None, val=None, log=True):
    """The five steps from week 1, wrapped up so the pages stay short."""
    train = train if train is not None else DataLoader(train_set,
                                                       batch_size=64,
                                                       shuffle=True)
    val = val if val is not None else DataLoader(val_set, batch_size=256)
    loss_fn = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr=lr)
    history = []
    for epoch in range(1, epochs + 1):
        model.train()
        seen = total = 0
        for xb, yb in train:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
            total += loss.item() * yb.numel()
            seen += yb.numel()
        model.eval()
        right = count = 0
        with torch.no_grad():
            for xb, yb in val:
                right += (model(xb).argmax(1) == yb).sum().item()
                count += yb.numel()
        history.append((total / seen, right / count))
        if log:
```

```
            print('epoch %d  train loss %.4f  val accuracy %.4f'
                  % (epoch, total / seen, right / count))
    return history
import json
torch.manual_seed(0)

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 64), nn.ReLU(),
                      nn.Linear(64, 10))
history = fit(model, epochs=3, log=False)

run = {
    'seed': 0,
    'architecture': '784-64-10',
    'optimizer': 'SGD',
    'lr': 0.1,
    'batch_size': 64,
    'epochs': len(history),
    'train_examples': len(train_set),
    'history': [{'epoch': i + 1, 'train_loss': round(l, 4),
                 'val_acc': round(a, 4)}
                for i, (l, a) in enumerate(history)],
}
print(json.dumps(run, indent=2)[:600])
```

```
{
  "seed": 0,
  "architecture": "784-64-10",
  "optimizer": "SGD",
  "lr": 0.1,
  "batch_size": 64,
  "epochs": 3,
  "train_examples": 8000,
  "history": [
    {
      "epoch": 1,
      "train_loss": 1.0652,
      "val_acc": 0.862
    },
    {
      "epoch": 2,
      "train_loss": 0.416,
      "val_acc": 0.8885
    },
    {
      "epoch": 3,
      "train_loss": 0.3351,
      "val_acc": 0.897
    }
... (2 more lines)
```

The number you did not record is the run you cannot repeat

Six weeks into a project you will have thirty runs and a vague memory that one of them worked. Write the configuration and the per-epoch numbers to a file named after the run, every time, automatically. Tools such as TensorBoard, Weights and Biases or MLflow do this and draw the

curves for you, and all of them are better than the notebook cell you will overwrite this afternoon.

> **Day 5 takeaway**

Day 6 Real Data Problems

Normalisation, and batches whose examples are different lengths

Two things that will bite you on real data rather than on MNIST: input scaling, and batches whose contents are not all the same size.

Normalising the input

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
raw = train_all.data[:8000].float() / 255.0
print('pixel mean %.4f, std %.4f' % (raw.mean(), raw.std()))

normalised = (raw - raw.mean()) / raw.std()
print('after normalising: mean %.4f, std %.4f'
      % (normalised.mean(), normalised.std()))

print('\ntorchvision does this for you:')
print("  transforms.Normalize((0.1307,), (0.3081,))")
print('  which are the mean and standard deviation of MNIST.')
```

```
pixel mean 0.1308, std 0.3082
after normalising: mean -0.0000, std 1.0000

torchvision does this for you:
  transforms.Normalize((0.1307,), (0.3081,))
  which are the mean and standard deviation of MNIST.
```

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

```

```
tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
torch.manual_seed(0)

def train_briefly(transform):
    data = datasets.MNIST('data', train=True, download=True,
                          transform=transform)
    loader = DataLoader(Subset(data, range(4000)), batch_size=64,
                        shuffle=True)
    torch.manual_seed(0)
    model = nn.Sequential(nn.Flatten(), nn.Linear(784, 64), nn.ReLU(),
                          nn.Linear(64, 10))
    opt = torch.optim.SGD(model.parameters(), lr=0.05)
    loss_fn = nn.CrossEntropyLoss()
    losses = []
    for epoch in range(3):
        for xb, yb in loader:
            opt.zero_grad()
            loss = loss_fn(model(xb), yb)
            loss.backward()
            opt.step()
        losses.append(loss.item())
    return losses

plain = train_briefly(transforms.ToTensor())
normed = train_briefly(transforms.Compose([
    transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]))
print('%-14s %s' % ('0 to 1', ['%.4f' % v for v in plain]))
print('%-14s %s' % ('normalised', ['%.4f' % v for v in normed]))
```

```
0 to 1         ['1.4501', '0.7852', '0.7656']
normalised     ['0.5104', '0.4380', '0.5037']
```

Watch Out

Compute the statistics on training data only

The mean and standard deviation are learned quantities, and computing them over the whole dataset lets the test set influence how the training set is scaled. It is a small leak and it is the same leak as any other. Use the training statistics for every split, including at serving time, where they have to be shipped alongside the weights.

Batches of unequal length

```
import torch
```

```
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader

sequences = [torch.tensor([1, 2, 3]),
             torch.tensor([4, 5]),
             torch.tensor([6, 7, 8, 9, 10])]

try:
    torch.stack(sequences)
except RuntimeError as e:
    print('stacking them directly fails:')
    print(' ', str(e)[:78])

padded = pad_sequence(sequences, batch_first=True, padding_value=0)
print('\npadded to a rectangle:')
print(padded)
print('lengths', [len(s) for s in sequences])
mask = padded != 0
print('mask (True means a real token):')
print(mask)
```

```
stacking them directly fails:
  stack expects each tensor to be equal size, but got [3] at entry 0 and [2] at

padded to a rectangle:
tensor([[ 1,  2,  3,  0,  0],
        [ 4,  5,  0,  0,  0],
        [ 6,  7,  8,  9, 10]])
lengths [3, 2, 5]
mask (True means a real token):
tensor([[ True,  True,  True, False, False],
        [ True,  True, False, False, False],
        [ True,  True,  True,  True,  True]])
```

```
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset

class Variable(Dataset):
    def __len__(self):
        return 6

    def __getitem__(self, i):
        return torch.arange(1, i + 2), i % 2

def collate(batch):
    """Called with a list of examples, returns one batch."""
    seqs, labels = zip(*batch)
    lengths = torch.tensor([len(s) for s in seqs])
    return (pad_sequence(seqs, batch_first=True),
            torch.tensor(labels), lengths)

loader = DataLoader(Variable(), batch_size=3, collate_fn=collate)
for x, y, lengths in loader:
```

```
    print('batch shape', tuple(x.shape), ' lengths', lengths.tolist())
    print(x)
```

```
batch shape (3, 3)  lengths [1, 2, 3]
tensor([[1, 0, 0],
        [1, 2, 0],
        [1, 2, 3]])
batch shape (3, 6)  lengths [4, 5, 6]
tensor([[1, 2, 3, 4, 0, 0],
        [1, 2, 3, 4, 5, 0],
        [1, 2, 3, 4, 5, 6]])
```

`collate_fn` is the hook for anything that cannot be stacked automatically: variable-length text, images of different sizes, examples that need grouping. Weeks 8 and 9 rely on it heavily, and the padding mask it produces is the same mask that attention will need.

Day 6 takeaway

Day 7 A Complete Training Script

Everything assembled, with the deliberate omissions named

Everything from this week in one script: a proper module, a real validation loop, early stopping with restoration, checkpointing and a run record.

The model

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
class MLP(nn.Module):
    def __init__(self, hidden=(256, 128), n_classes=10):
        super().__init__()
        sizes = (784,) + tuple(hidden)
        self.blocks = nn.ModuleList([
            nn.Sequential(nn.Linear(a, b), nn.ReLU())
            for a, b in zip(sizes, sizes[1:])])
        self.head = nn.Linear(sizes[-1], n_classes)

    def forward(self, x):
```

```
        x = x.flatten(1)
        for block in self.blocks:
            x = block(x)
        return self.head(x)

model = MLP()
print(model)
print('\nparameters %d' % sum(p.numel() for p in model.parameters()))
```

```
MLP(
  (blocks): ModuleList(
    (0): Sequential(
      (0): Linear(in_features=784, out_features=256, bias=True)
      (1): ReLU()
    )
    (1): Sequential(
      (0): Linear(in_features=256, out_features=128, bias=True)
      (1): ReLU()
    )
  )
  (head): Linear(in_features=128, out_features=10, bias=True)
)

parameters 235146
```

The training script

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

# Subsets, so every page finishes while you watch it. The lessons are the
# same at full size and the numbers are a little higher.
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
test_set = Subset(test_all, range(2000))
import copy, json, time

class MLP(nn.Module):
    def __init__(self, hidden=(256, 128)):
        super().__init__()
        sizes = (784,) + tuple(hidden)
        self.blocks = nn.ModuleList([
            nn.Sequential(nn.Linear(a, b), nn.ReLU())
            for a, b in zip(sizes, sizes[1:])])
        self.head = nn.Linear(sizes[-1], 10)

    def forward(self, x):
```

```
        x = x.flatten(1)
        for block in self.blocks:
            x = block(x)
        return self.head(x)

CONFIG = {'seed': 0, 'hidden': (256, 128), 'lr': 0.1, 'batch_size': 64,
          'max_epochs': 30, 'patience': 5}

torch.manual_seed(CONFIG['seed'])
train_loader = DataLoader(train_set, batch_size=CONFIG['batch_size'],
                          shuffle=True)
val_loader = DataLoader(val_set, batch_size=256)
test_loader = DataLoader(test_set, batch_size=256)

model = MLP(CONFIG['hidden'])
opt = torch.optim.SGD(model.parameters(), lr=CONFIG['lr'])
loss_fn = nn.CrossEntropyLoss()

def evaluate(loader):
    model.eval()
    loss = right = seen = 0
    with torch.no_grad():
        for xb, yb in loader:
            out = model(xb)
            loss += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return loss / seen, right / seen

best = {'loss': float('inf'), 'epoch': 0, 'state': None}
history = []
start = time.time()
for epoch in range(1, CONFIG['max_epochs'] + 1):
    model.train()
    for xb, yb in train_loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()
    vloss, vacc = evaluate(val_loader)
    history.append({'epoch': epoch, 'val_loss': round(vloss, 4),
                    'val_acc': round(vacc, 4)})
    if vloss < best['loss']:
        best = {'loss': vloss, 'epoch': epoch,
                'state': copy.deepcopy(model.state_dict())}
    elif epoch - best['epoch'] >= CONFIG['patience']:
        break

model.load_state_dict(best['state'])
test_loss, test_acc = evaluate(test_loader)

torch.save({'config': CONFIG, 'model': best['state'],
            'best_epoch': best['epoch'], 'test_acc': test_acc},
           'mnist_mlp.pt')

```

```
print('ran %d epochs in %.1fs' % (len(history), time.time() - start))
print('best epoch %d, validation loss %.4f'
      % (best['epoch'], best['loss']))
print('test accuracy %.4f  (looked at once)' % test_acc)
print('\nlast five epochs:')
for row in history[-5:]:
    print(' ', json.dumps(row))
```

```
ran 22 epochs in 20.4s
best epoch 17, validation loss 0.2238
test accuracy 0.9225  (looked at once)

last five epochs:
  {"epoch": 18, "val_loss": 0.2358, "val_acc": 0.936}
  {"epoch": 19, "val_loss": 0.2267, "val_acc": 0.939}
  {"epoch": 20, "val_loss": 0.2451, "val_acc": 0.934}
  {"epoch": 21, "val_loss": 0.2329, "val_acc": 0.939}
  {"epoch": 22, "val_loss": 0.2314, "val_acc": 0.939}
```

The habits, collected

1. Subclass `nn.Module`; use `nn.ModuleList` for anything you build in a loop.
2. Pass logits to the loss and let it apply the softmax.
3. Three splits, and look at the test set once.
4. Overfit a single batch before you debug anything else.
5. Deep copy the best state dictionary and restore it at the end.
6. Save the configuration in the checkpoint next to the weights.
7. Normalise inputs with training statistics and ship them with the model.
8. Log per-epoch numbers to a file, not to a scrollback buffer.

What is missing, deliberately

This model has no dropout, no weight decay, no augmentation, no learning rate schedule and no normalisation layers. It is a clean baseline, and it is what the next two weeks improve on one change at a time so you can see what each is worth. A model with all five added at once is a model where you cannot tell which of them helped, and in practice one or two usually do nothing at all.

> **Day 7 takeaway**

Self-Assessment

Try It Yourself

1. You store your layers in a plain Python list on an `nn.Module`. What breaks?

   a) The forward pass raises an error

   b) They are not registered, so they never reach the optimiser and never train, while everything appears to run

   c) The model cannot be moved to a device

   d) Nothing

2. Why does `CrossEntropyLoss` want logits rather than probabilities?

   a) It is faster to compute

   b) It applies the log-softmax itself, in a form that stays stable when a score is extreme, which a separate softmax then log does not

   c) Probabilities do not sum to one after softmax

   d) It needs negative numbers

3. What is the point of keeping a separate validation split?

   a) It measures performance on data no gradient has touched, which is the only signal that tells you the model is memorising

   b) It makes training faster

   c) It doubles the amount of data

   d) It reduces the learning rate automatically

4. Early stopping is described as keeping the weights that were best. Why does that phrasing matter?

   a) Training stops several epochs after the best one, so unless you saved the best state you end up with weights that are already worse

   b) The optimiser rewinds automatically

   c) It does not, stopping is enough

   d) Stopping alone leaves you with the weights from the worst epoch

5. You set every seed and get a different result on a colleague's machine. What is the most likely reason?

   a) A seed fixes the random draws, not the library version, thread count or hardware kernels, and those change the arithmetic

   b) Their data was different

   c) Seeds only work on the CPU

   d) The seed was ignored

6. What does the single batch test check?

   a) That the data loader shuffles

   b) That the batch size fits in memory

   c) That the model can drive the loss on one batch to nearly zero, which proves the plumbing works before you spend hours on the full set

d) That the validation split is representative

7. Why must sequences of different lengths be padded and given a mask?

   a) The loss function requires equal lengths

   b) Padding alone is enough

   c) A tensor must be rectangular, so short sequences are filled out, and without a mask the model treats that filler as real content

   d) Masks make training faster

8. What is a `state_dict`?

   a) A plain mapping from parameter name to tensor, which is why loading it needs the model class to exist first

   b) A record of the training history

   c) The full model object, ready to run

   d) The optimiser configuration

Answers:1b, 2b, 3a, 4a, 5a, 6c, 7c, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 3

Optimisation

Week 3

OBJECTIVES

Momentum, Adam, learning rate schedules, batch size, clipping, and the failure shapes worth recognising on sight.

A network that trains slowly and a network that does not train are usually the same network with a different learning rate. This week covers what actually moves the weights: why plain gradient descent struggles, what momentum and Adam do about it, how to find a learning rate rather than guess one, and how batch size, clipping and scheduling fit together. It ends by adding each piece to a baseline one at a time, so the final recipe is a set of decisions rather than a superstition.

Day 1 Optimisers

Why steepest descent is slow, and what momentum and Adam do about it

Plain gradient descent takes a step straight downhill. That is the obvious thing to do and it is slow, for a reason that is easier to see than to describe.

The problem with steepest descent

```
import torch

# A bowl 200 times steeper in one direction than the other. The steep
# direction caps the learning rate at about 2/200, and the shallow one
# then needs an enormous number of those small steps.
def loss(p):
    return 0.5 * (200 * p[0] ** 2 + p[1] ** 2)

p = torch.tensor([1.0, 1.0], requires_grad=True)
lr = 0.009
print('%6s %12s %12s %12s' % ('step', 'steep x', 'shallow y', 'loss'))
for step in range(151):
    if p.grad is not None:
        p.grad.zero_()
    value = loss(p)
    value.backward()
    with torch.no_grad():
        p -= lr * p.grad
    if step in (0, 5, 20, 60, 150):
        print('%6d %12.4f %12.4f %12.5f'
```

```
              % (step, p[0].item(), p[1].item(), value.item()))
```

```
 step      steep x    shallow y         loss
    0      -0.8000       0.9910    100.50000
    5       0.2621       0.9472     11.19419
   20      -0.0092       0.8271      0.36156
   60      -0.0000       0.5761      0.16897
  150      -0.0000       0.2553      0.03319
```

The steep direction is finished within a handful of steps. The shallow one is still most of the way from where it started after a hundred and fifty, because the rate that keeps the steep direction stable is far too small for it. Raising the rate makes the steep direction oscillate and then diverge, so you cannot. That is the tension every optimiser after this one is trying to resolve, and it gets worse as models grow, because the gap between the steepest and shallowest directions grows with them.

Momentum

```
import torch

def loss(p):
    return 0.5 * (200 * p[0] ** 2 + p[1] ** 2)

def run(beta, lr, steps):
    p = torch.tensor([1.0, 1.0], requires_grad=True)
    velocity = torch.zeros(2)
    for _ in range(steps):
        if p.grad is not None:
            p.grad.zero_()
        loss(p).backward()
        with torch.no_grad():
            velocity = beta * velocity + p.grad
            p -= lr * velocity
    return loss(p).item()

print('%-16s %12s %12s %12s' % ('', '20 steps', '60 steps', '150 steps'))
for label, beta in [('plain SGD', 0.0), ('momentum 0.9', 0.9)]:
    print('%-16s %12.5f %12.5f %12.5f'
          % (label, run(beta, 0.009, 20), run(beta, 0.009, 60),
             run(beta, 0.009, 150)))
```

```
                    20 steps     60 steps    150 steps
plain SGD            0.36156      0.16897      0.03319
momentum 0.9        19.74603      0.18695      0.00001
```

Watch Out

Momentum looks worse before it looks better

At twenty steps momentum is around fifty times *worse*. A running average with `beta = 0.9` settles at roughly ten times the size of a single gradient, so the effective step is ten times

larger and it overshoots badly at the start. By a hundred and fifty steps it is thousands of times better than plain descent.

Two things follow from that. Momentum earns its place on long runs and can genuinely hurt on short ones, and if you add it without lowering the learning rate you should expect the early part of training to look wrong. Judging an optimiser on its first few hundred steps is how people talk themselves out of momentum.

Adam

```
import torch

def loss(p):
    return 0.5 * (20 * p[0] ** 2 + p[1] ** 2)

p = torch.tensor([1.0, 1.0], requires_grad=True)
m = torch.zeros(2)
v = torch.zeros(2)
b1, b2, eps, lr = 0.9, 0.999, 1e-8, 0.1

print('%6s %10s %10s %12s' % ('step', 'x', 'y', 'loss'))
for t in range(1, 10):
    if p.grad is not None:
        p.grad.zero_()
    value = loss(p)
    value.backward()
    with torch.no_grad():
        m = b1 * m + (1 - b1) * p.grad
        v = b2 * v + (1 - b2) * p.grad ** 2
        m_hat = m / (1 - b1 ** t)          # bias correction
        v_hat = v / (1 - b2 ** t)
        p -= lr * m_hat / (v_hat.sqrt() + eps)
    if t % 2 == 1:
        print('%6d %10.4f %10.4f %12.5f'
              % (t, p[0].item(), p[1].item(), value.item()))
```

```
  step          x          y          loss
     1     0.9000     0.9000      10.50000
     3     0.7016     0.7016       6.72693
     5     0.5080     0.5080       3.82980
     7     0.3234     0.3234       1.80171
     9     0.1536     0.1536       0.58612
```

Look at the two coordinates: they are identical at every step, on a bowl where one direction is two hundred times steeper than the other. Dividing by the square root of the running average of squared gradients has removed the scale difference entirely, so the optimiser now walks diagonally to the minimum instead of zig-zagging down one axis and crawling along the other. That is the whole of what adaptive means.

Why the bias correction is there

Both running averages start at zero, so for the first few steps they are biased towards zero and the step size would be far too small. Dividing by `1 - beta**t` undoes exactly that, and the correction fades away as `t` grows. Leave it out and the first hundred steps of training barely move, which is a bug people reinvent regularly when writing their own optimiser.

On a real model

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
print('%-14s %12s %12s' % ('optimiser', 'val loss', 'val acc'))
for name, fn in [
        ('SGD 0.1', lambda p: torch.optim.SGD(p, lr=0.1)),
```

```
        ('SGD+mom 0.1', lambda p: torch.optim.SGD(p, lr=0.1, momentum=0.9)),
        ('Adam 1e-3', lambda p: torch.optim.Adam(p, lr=1e-3)),
        ('AdamW 1e-3', lambda p: torch.optim.AdamW(p, lr=1e-3)),
        ('RMSprop 1e-3', lambda p: torch.optim.RMSprop(p, lr=1e-3))]:
    loss, acc = run(fn)
    print('%-14s %12.4f %12.4f' % (name, loss, acc))
```

```
optimiser          val loss      val acc
SGD 0.1              0.2639       0.9235
SGD+mom 0.1          0.3766       0.9125
Adam 1e-3            0.2715       0.9230
AdamW 1e-3           0.2682       0.9220
RMSprop 1e-3         0.2355       0.9315
```

Watch Out

Notice that nothing wins convincingly

Five optimisers, and the spread between the best and the worst is about two points of accuracy, with plain SGD in the middle and momentum at the same rate behind it for the reason given above. On a small model and an easy dataset that is the normal result: the optimiser is not usually where your accuracy comes from.
Day 6 measures how much this model varies between random seeds, and the answer makes most of this table look smaller still. AdamW is the sensible default because it needs the least tuning to get somewhere reasonable, not because it wins races.

Day 1 takeaway

Day 2 Learning Rates and Schedules

Finding the rate with a range test, then decaying it properly

The learning rate matters more than the choice of optimiser, and it is the one hyperparameter you can find quickly rather than guess.

The range test

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
import math
torch.manual_seed(0)
```

```

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                      nn.Linear(128, 10))
opt = torch.optim.SGD(model.parameters(), lr=1e-5)
loss_fn = nn.CrossEntropyLoss()
loader = DataLoader(train_set, batch_size=64, shuffle=True)

lo, hi, steps = 1e-5, 10.0, 120
gamma = (hi / lo) ** (1 / steps)
rates, losses = [], []
it = iter(loader)
for step in range(steps):
    try:
        xb, yb = next(it)
    except StopIteration:
        it = iter(loader)
        xb, yb = next(it)
    lr = lo * gamma ** step
    for group in opt.param_groups:
        group['lr'] = lr
    opt.zero_grad()
    loss = loss_fn(model(xb), yb)
    loss.backward()
    opt.step()
    rates.append(lr)
    losses.append(loss.item())

print('%12s %12s' % ('rate', 'loss'))
for i in range(0, steps, 12):
    print('%12.5f %12.4f' % (rates[i], losses[i]))

best = min(range(len(losses)), key=lambda i: losses[i])
print('\nlowest loss at rate %.4f' % rates[best])
print('a sensible starting point is a little below that')
```

```
        rate         loss
     0.00001       2.2856
     0.00004       2.2757
     0.00016       2.2586
     0.00063       2.3156
     0.00251       2.2612
     0.01000       2.1424
     0.03981       1.8005
     0.15849       1.3027
     0.63096       1.4018
     2.51189      56.6162

lowest loss at rate 0.2818
a sensible starting point is a little below that
```

Schedules

```
import torch
```

```
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)
model = nn.Sequential(nn.Linear(4, 4))
opt = torch.optim.SGD(model.parameters(), lr=0.1)

schedules = {
    'step (x0.1 every 8)':
        torch.optim.lr_scheduler.StepLR(opt, step_size=8, gamma=0.1),
}
print('%6s %14s %14s %14s' % ('epoch', 'step', 'cosine', 'one cycle'))

def trace(make, epochs=20):
    o = torch.optim.SGD([torch.zeros(1, requires_grad=True)], lr=0.1)
    s = make(o)
    out = []
    for _ in range(epochs):
        out.append(o.param_groups[0]['lr'])
        o.step()
        s.step()
    return out

step_lr = trace(lambda o: torch.optim.lr_scheduler.StepLR(o, 8, 0.1))
cosine = trace(lambda o: torch.optim.lr_scheduler.CosineAnnealingLR(o, 20))
onecycle = trace(lambda o: torch.optim.lr_scheduler.OneCycleLR(
    o, max_lr=0.4, total_steps=20))
for e in range(0, 20, 2):
    print('%6d %14.5f %14.5f %14.5f'
          % (e, step_lr[e], cosine[e], onecycle[e]))
```

```
 epoch           step         cosine      one cycle
     0        0.10000        0.10000        0.01600
     2        0.10000        0.09755        0.14867
     4        0.10000        0.09045        0.36333
     6        0.10000        0.07939        0.39499
     8        0.01000        0.06545        0.35637
    10        0.01000        0.05000        0.28678
    12        0.01000        0.03455        0.20000
    14        0.01000        0.02061        0.11322
    16        0.00100        0.00955        0.04364
    18        0.00100        0.00245        0.00502
```

Schedule	Shape	Use when
Constant	Flat	A quick experiment, or Adam on a small problem
Step	Drops by a factor at fixed epochs	You know roughly how long training takes
Cosine	Smooth decay to nearly zero	The default for most modern training runs
One cycle	Warms up, then decays below the start	Short budgets; often the fastest to a good result
Reduce on plateau	Drops when validation stalls	You cannot predict the length of the run

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
```

```
print('%-24s %12s %12s' % ('', 'val loss', 'val acc'))
for name, sched in [
        ('constant 0.1', None),
        ('cosine to zero',
         lambda o: torch.optim.lr_scheduler.CosineAnnealingLR(o, 375)),
        ('one cycle, max 0.1',
         lambda o: torch.optim.lr_scheduler.OneCycleLR(
             o, max_lr=0.1, total_steps=375))]:
    loss, acc = run(lambda p: torch.optim.SGD(p, lr=0.1, momentum=0.9),
                    epochs=3, sched_fn=sched)
    print('%-24s %12.4f %12.4f' % (name, loss, acc))
```

```
                         val loss      val acc
constant 0.1               0.3766       0.9125
cosine to zero             0.2330       0.9355
one cycle, max 0.1         0.2313       0.9400
```

Warmup

Large models, and anything using Adam with a large batch, often start with a few hundred steps at a very small rate that grows to the target. The reason is that Adam's running averages are unreliable in the first few steps, so a full-size step taken on that estimate can move the weights somewhere the model never recovers from.

```
import torch

def warmup_cosine(step, warmup=100, total=1000, peak=1.0):
    if step < warmup:
        return peak * step / warmup
    progress = (step - warmup) / (total - warmup)
    return peak * 0.5 * (1 + torch.cos(torch.tensor(3.14159 * progress)))

print('%8s %12s' % ('step', 'multiplier'))
for step in [0, 25, 50, 100, 300, 600, 900, 1000]:
    print('%8d %12.4f' % (step, float(warmup_cosine(step))))
```

```
    step   multiplier
       0       0.0000
      25       0.2500
      50       0.5000
     100       1.0000
     300       0.8830
     600       0.4132
     900       0.0302
    1000       0.0000
```

Day 2 takeaway

Day 3 Batch Size, Accumulation and Clipping

Gradient noise, fitting a large batch in a small memory, and the norm

Batch size looks like a memory setting and behaves like a hyperparameter, because it changes how noisy each gradient is and therefore how large a step you can safely take.

What the batch size actually changes

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                      nn.Linear(128, 10))
loss_fn = nn.CrossEntropyLoss()

print('%8s %14s %16s' % ('batch', 'gradient norm', 'std across draws'))
for bs in [8, 32, 128, 512]:
    norms = []
    loader = DataLoader(train_set, batch_size=bs, shuffle=True)
    it = iter(loader)
    for _ in range(12):
        xb, yb = next(it)
        model.zero_grad()
        loss_fn(model(xb), yb).backward()
        flat = torch.cat([p.grad.flatten() for p in model.parameters()])
        norms.append(flat.norm().item())
    t = torch.tensor(norms)
    print('%8d %14.4f %16.4f' % (bs, t.mean().item(), t.std().item()))
```

```
   batch  gradient norm std across draws
       8         4.3953           0.5277
      32         2.6351           0.2021
     128         1.7999           0.1236
     512         1.5807           0.0809
```

A larger batch gives a less noisy estimate of the true gradient. That is why it tolerates a larger learning rate, and it is the origin of the rule of thumb that doubling the batch size lets you roughly double the rate. The table below tests that rule, and finds where it stops being true.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
```

```
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
print('%8s %10s %12s %12s' % ('batch', 'lr', 'val loss', 'val acc'))
for bs, lr in [(32, 0.05), (32, 0.1), (128, 0.1), (128, 0.4),
               (512, 0.1), (512, 1.6)]:
    loss, acc = run(lambda p, lr=lr: torch.optim.SGD(p, lr=lr),
                    epochs=3, batch_size=bs)
    print('%8d %10.2f %12.4f %12.4f' % (bs, lr, loss, acc))
```

```
   batch         lr     val loss      val acc
      32       0.05       0.2690       0.9230
      32       0.10       0.2139       0.9335
     128       0.10       0.3283       0.9065
     128       0.40       0.2665       0.9200
     512       0.10       0.4511       0.8775
     512       1.60       1.8504       0.2805
```

Gradient accumulation

When the batch you want does not fit in memory, run several small batches, add their gradients together, and step once. The arithmetic is identical to one large batch, which is exactly the behaviour week 1 described as a bug when it happened by accident.

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)
loader = DataLoader(train_set, batch_size=16, shuffle=False)
loss_fn = nn.CrossEntropyLoss()

def grad_after(batch_size, accumulate):
    torch.manual_seed(0)
    model = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=False)
    it = iter(loader)
    model.zero_grad()
    for _ in range(accumulate):
        xb, yb = next(it)
        (loss_fn(model(xb), yb) / accumulate).backward()
    return torch.cat([p.grad.flatten() for p in model.parameters()])

big = grad_after(64, 1)
accumulated = grad_after(16, 4)
print('one batch of 64        norm %.6f' % big.norm().item())
print('four batches of 16     norm %.6f' % accumulated.norm().item())
print('largest difference %.2e'
      % (big - accumulated).abs().max().item())
```

```
one batch of 64        norm 4.976234
four batches of 16     norm 4.976234
largest difference 7.45e-08
```

Watch Out

Divide by the number of accumulation steps

`CrossEntropyLoss` already averages over its batch, so adding four batch means gives you four times the gradient you wanted. Dividing each loss by the accumulation count restores the arithmetic. Miss it and your effective learning rate is four times what you think it is, which usually shows up as a run that diverges only when you change the accumulation setting.

Gradient clipping

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
```

```

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                      nn.Linear(128, 10))
loss_fn = nn.CrossEntropyLoss()
xb, yb = next(iter(DataLoader(train_set, batch_size=64, shuffle=True)))

model.zero_grad()
(loss_fn(model(xb), yb) * 500).backward()      # a pathological gradient
before = torch.cat([p.grad.flatten() for p in model.parameters()]).norm()
clipped = torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
after = torch.cat([p.grad.flatten() for p in model.parameters()]).norm()

print('norm before clipping %.4f' % before.item())
print('reported by clip_grad_norm_ %.4f' % clipped.item())
print('norm after clipping  %.4f' % after.item())
print('\ndirection is preserved, only the length changes.')
```

```
norm before clipping 1058.4519
reported by clip_grad_norm_ 1058.4520
norm after clipping  1.0000

direction is preserved, only the length changes.
```

Clipping is close to mandatory for recurrent networks in week 8 and for transformers in week 10, where a single unlucky batch can produce a gradient thousands of times the usual size and destroy a run that was going well. A max norm of 1.0 is the conventional starting point.

Day 3 takeaway

Day 4 When Training Goes Wrong

The failure shapes, and measuring gradients instead of guessing

Training goes wrong in a small number of recognisable ways. Learning the shapes of the failures is worth more than any amount of hyperparameter search.

Loss becomes nan

```

import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
```

```
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                      nn.Linear(128, 10))
opt = torch.optim.SGD(model.parameters(), lr=8.0)     # far too high
loss_fn = nn.CrossEntropyLoss()

for step, (xb, yb) in enumerate(DataLoader(train_set, batch_size=64,
                                           shuffle=True)):
    opt.zero_grad()
    loss = loss_fn(model(xb), yb)
    loss.backward()
    opt.step()
    if step < 8 or loss.item() != loss.item():
        print('step %d  loss %s' % (step, loss.item()))
    if loss.item() != loss.item():
        break
```

```
step 0  loss 2.285597085952759
step 1  loss 50.645973205566406
step 2  loss 5039.51953125
step 3  loss 99568.6171875
step 4  loss 3233553.75
step 5  loss 50211.375
step 6  loss 431.63970947265625
step 7  loss 1273.14404296875
```

Symptom	Most likely cause	First thing to try
Loss becomes `nan` or `inf`	Learning rate too high	Divide the rate by ten
Loss is flat from step zero	Rate far too small, or the graph is broken	Overfit one batch; check the loss is connected to the parameters
Loss falls then plateaus high	Model too small, or the rate has decayed too far	More capacity, or check the schedule
Training loss falls, validation rises	Overfitting	Week 4, in its entirety
Validation is wildly noisy between epochs	Validation set too small, or the rate is too high late on	Larger split, decay the rate
Accuracy is exactly the majority class rate	The model has collapsed to one output	Check class balance and the loss weighting

Watching the gradients rather than guessing

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
```

```
val_set = Subset(train_all, range(8000, 10000))
torch.manual_seed(0)

def gradient_report(depth, activation):
    layers = [nn.Flatten(), nn.Linear(784, 64), activation()]
    for _ in range(depth):
        layers += [nn.Linear(64, 64), activation()]
    layers += [nn.Linear(64, 10)]
    model = nn.Sequential(*layers)
    xb, yb = next(iter(DataLoader(train_set, batch_size=64, shuffle=True)))
    nn.CrossEntropyLoss()(model(xb), yb).backward()
    norms = [p.grad.norm().item() for name, p in model.named_parameters()
             if 'weight' in name]
    return norms

for label, act in [('sigmoid', nn.Sigmoid), ('relu', nn.ReLU)]:
    norms = gradient_report(8, act)
    print('%-8s first layer %.2e   last layer %.2e   ratio %.1e'
          % (label, norms[0], norms[-1], norms[-1] / max(norms[0], 1e-30)))
```

```
sigmoid  first layer 8.89e-08   last layer 6.04e-01   ratio 6.8e+06
relu     first layer 1.03e-03   last layer 7.58e-02   ratio 7.3e+01
```

This is the vanishing gradient, measured

With sigmoid activations the gradient reaching the first layer is orders of magnitude smaller than the one at the last, so the early layers barely move while the late ones train. With ReLU the ratio is far closer to one. Printing gradient norms per layer takes four lines and tells you immediately whether a deep model is training everywhere or only at the end.

A checklist that finds most problems

1. Can the model overfit a single batch to near-zero loss? If not, the bug is structural, not statistical.
2. Is the loss connected to the parameters? Print `loss.grad_fn` and check a parameter's `.grad` is not `None` after backward.
3. Are the inputs normalised, and did you check the actual min and max rather than assuming?
4. Are labels in the range the loss expects? `CrossEntropyLoss` wants 0 to C-1, and an off-by-one gives an unhelpful index error or, worse, silently trains on the wrong classes.
5. Is `model.train()` set for training and `model.eval()` for evaluation?
6. Did you call `zero_grad()`?
7. Print the gradient norm per layer once, early.
8. Try one tenth of the learning rate before you try anything clever.

> **Day 4 takeaway**

Day 5 Precision and Weight Averaging

Two cheap techniques worth knowing before you need them

Two techniques that cost almost nothing and are worth knowing before you need them: making training faster with lower precision, and making a model better by averaging the weights you already have.

Precision

```
import torch

x = torch.tensor([1.0], dtype=torch.float32)
for dtype in [torch.float32, torch.float16, torch.bfloat16]:
    info = torch.finfo(dtype)
    print('%-16s bits %2d  max %.3e  smallest normal %.3e'
          % (str(dtype), info.bits, info.max, info.tiny))

print('\na gradient of 1e-8 in each format:')
small = 1e-8
for dtype in [torch.float32, torch.float16, torch.bfloat16]:
    print('  %-16s %s' % (str(dtype),
                          torch.tensor(small, dtype=dtype).item()))
```

```
torch.float32    bits 32  max 3.403e+38  smallest normal 1.175e-38
torch.float16    bits 16  max 6.550e+04  smallest normal 6.104e-05
torch.bfloat16   bits 16  max 3.390e+38  smallest normal 1.175e-38

a gradient of 1e-8 in each format:
  torch.float32    9.99999993922529e-09
  torch.float16    0.0
  torch.bfloat16   1.0011717677116394e-08
```

Half precision has a much smaller range, and gradients are exactly where small numbers live. A gradient that underflows to zero in float16 contributes nothing, which is why mixed precision training scales the loss up before the backward pass and scales the gradients back down afterwards. On a GPU this roughly halves memory and can double speed; on the CPU this course runs on, it does neither, so the code below is shown rather than raced.

Watch Out

bfloat16 is usually the better choice now

It has the same exponent range as float32 and fewer mantissa bits, so it does not underflow and needs no loss scaling at all. On hardware that supports it, `torch.autocast('cuda', dtype=torch.bfloat16)` with no `GradScaler` is simpler and less fragile. float16 remains common only because older cards support it and bfloat16 came later.

Averaging weights

```
import torch
from torch import nn
```

```
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
import copy
torch.manual_seed(0)

model = build(0)
opt = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
loader = DataLoader(train_set, batch_size=64, shuffle=True)
val = DataLoader(val_set, batch_size=512)

def score(m):
    m.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (m(xb).argmax(1) == yb).sum().item()
```

```
            seen += yb.numel()
    return right / seen

averaged = None
collected = 0
for epoch in range(1, 9):
    model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()
    if epoch >= 5:                        # average the last four
        state = model.state_dict()
        if averaged is None:
            averaged = {k: v.clone().float() for k, v in state.items()}
        else:
            for k in averaged:
                averaged[k] += state[k].float()
        collected += 1

final_acc = score(model)
for k in averaged:
    averaged[k] /= collected
swa = copy.deepcopy(model)
swa.load_state_dict(averaged)
print('final epoch weights   %.4f' % final_acc)
print('average of last four  %.4f' % score(swa))
```

```
final epoch weights   0.9505
average of last four  0.9510
```

The averaged weights come out very slightly ahead here, by far less than the seed-to-seed variation day 6 measures on this same model. So the honest reading is that it did not help on this problem, and the reason to know about it is that it costs one extra copy of the weights and does reliably help on larger models trained for longer, where the final weights are still bouncing around a minimum rather than sitting in one.

Day 5 takeaway

Day 6 Choosing Hyperparameters

What to search, on what budget, and the variation you must beat

Choosing hyperparameters for a deep model is the same problem as week 14 of the Machine Learning course, with one difference that changes the arithmetic completely: a single evaluation costs minutes or hours rather than milliseconds.

What is worth searching

Parameter	Priority	Sensible range
Learning rate	First, and by a distance	Found by a range test, then tuned within a factor of three
Batch size	Second, jointly with the rate	As large as memory allows, then tune the rate to match
Weight decay	Third	1e-5 to 1e-1, on a log scale
Architecture width and depth	Fourth	Powers of two; go wider before deeper
Dropout rate	Fifth	0.0 to 0.5
Optimiser choice	Rarely worth it	AdamW unless you have a reason
Adam's betas and epsilon	Almost never	Leave them

Random search, on a budget

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
```

```
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
import math
torch.manual_seed(0)
rng = torch.Generator().manual_seed(7)

def sample():
    log_lr = -4 + 3 * torch.rand(1, generator=rng).item()   # 1e-4 to 1e-1
    wd = 10 ** (-5 + 4 * torch.rand(1, generator=rng).item())
    return 10 ** log_lr, wd

print('%10s %10s %12s %12s' % ('lr', 'weight decay', 'val loss', 'val acc'))
results = []
for _ in range(8):
    lr, wd = sample()
    loss, acc = run(lambda p, lr=lr, wd=wd:
                    torch.optim.AdamW(p, lr=lr, weight_decay=wd),
                    epochs=2)
    results.append((acc, lr, wd))
    print('%10.5f %10.5f %12.4f %12.4f' % (lr, wd, loss, acc))

best = max(results)
print('\nbest: lr %.5f, weight decay %.5f, accuracy %.4f'
      % (best[1], best[2], best[0]))
```

```
        lr weight decay     val loss      val acc
   0.00403    0.00006       0.2695       0.9220
   0.00950    0.00424       0.3458       0.9070
   0.00050    0.00050       0.3538       0.8955
   0.00042    0.00330       0.3647       0.8940
   0.00125    0.02541       0.3218       0.9035
   0.03671    0.00160       0.6686       0.8360
   0.00073    0.00007       0.3340       0.9045
   0.00216    0.00027       0.2960       0.9145

best: lr 0.00403, weight decay 0.00006, accuracy 0.9220
```

Search on a short budget, then verify at full length

Two epochs is not enough to decide which model is best, but it is usually enough to decide which learning rates are hopeless, and that is most of the search space. Screen widely and cheaply, take the best three or four settings, and train those properly. The failure mode to watch for is a setting that is slow to start and best in the end, which a short screen discards; a warmup and a fixed number of steps rather than epochs both reduce it.

The number you report

```
import torch
from torch import nn
```

```
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
def build(seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def run(opt_fn, epochs=3, batch_size=64, seed=0, sched_fn=None):
    torch.manual_seed(seed)
    model = build(seed)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            if sched is not None:
                sched.step()
    model.eval()
    right = seen = 0
    total = 0.0
    with torch.no_grad():
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    return total / seen, right / seen
torch.manual_seed(0)
print('the same configuration, five different seeds:')
accs = []
for seed in range(5):
    loss, acc = run(lambda p: torch.optim.SGD(p, lr=0.1, momentum=0.9),
                    epochs=2, seed=seed)
    accs.append(acc)
    print('  seed %d  %.4f' % (seed, acc))
t = torch.tensor(accs)
print('\nmean %.4f, std %.4f, range %.4f'
      % (t.mean(), t.std(), t.max() - t.min()))
print('any comparison smaller than that range is not a result.')
```

```
the same configuration, five different seeds:
  seed 0  0.8995
```

```
  seed 1  0.9000
  seed 2  0.8910
  seed 3  0.8825
  seed 4  0.8915

mean 0.8929, std 0.0072, range 0.0175
any comparison smaller than that range is not a result.
```

Day 6 takeaway

Day 7 A Tuned Training Recipe

Every change measured on its own, against the week 2 baseline

Everything from this week applied at once, and then measured against the plain baseline from week 2 so the total is honest.

The tuned run

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
import time
torch.manual_seed(0)

EPOCHS, BATCH = 8, 128
loader = DataLoader(train_set, batch_size=BATCH, shuffle=True)
val = DataLoader(val_set, batch_size=512)
steps = EPOCHS * len(loader)

model = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
                      nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 10))
opt = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=1e-4)
sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=3e-3,
                                            total_steps=steps)
loss_fn = nn.CrossEntropyLoss()

def score():
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
```

```
            seen += yb.numel()
    return right / seen

start = time.time()
for epoch in range(1, EPOCHS + 1):
    model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step()
        sched.step()
    if epoch % 2 == 0:
        print('epoch %d  lr %.5f  val acc %.4f'
              % (epoch, sched.get_last_lr()[0], score()))
print('\n%.1f seconds' % (time.time() - start))
```

```
epoch 2  lr 0.00282  val acc 0.8970
epoch 4  lr 0.00242  val acc 0.9270
epoch 6  lr 0.00084  val acc 0.9495
epoch 8  lr 0.00000  val acc 0.9505

11.5 seconds
```

Against the baseline

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(train_all, range(8000, 10000))
import time

def train(tag, opt_fn, epochs, batch, sched_fn=None, clip=None,
          hidden=(128,)):
    torch.manual_seed(0)
    sizes = (784,) + hidden
    layers = [nn.Flatten()]
    for a, b in zip(sizes, sizes[1:]):
        layers += [nn.Linear(a, b), nn.ReLU()]
    layers += [nn.Linear(sizes[-1], 10)]
    model = nn.Sequential(*layers)
    loader = DataLoader(train_set, batch_size=batch, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = opt_fn(model.parameters())
    sched = sched_fn(opt, epochs * len(loader)) if sched_fn else None
    loss_fn = nn.CrossEntropyLoss()
    start = time.time()
```

```
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
            if sched:
                sched.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    print('%-30s %10.4f %10.1fs' % (tag, right / seen, time.time() - start))

print('%-30s %10s %11s' % ('', 'val acc', 'time'))
train('week 2 baseline, SGD 0.1', lambda p: torch.optim.SGD(p, lr=0.1),
      8, 64)
train('+ momentum',
      lambda p: torch.optim.SGD(p, lr=0.1, momentum=0.9), 8, 64)
train('+ AdamW',
      lambda p: torch.optim.AdamW(p, lr=3e-3, weight_decay=1e-4), 8, 64)
train('+ one cycle and clipping',
      lambda p: torch.optim.AdamW(p, lr=3e-3, weight_decay=1e-4), 8, 128,
      sched_fn=lambda o, s: torch.optim.lr_scheduler.OneCycleLR(
          o, max_lr=3e-3, total_steps=s), clip=1.0)
train('+ a wider model',
      lambda p: torch.optim.AdamW(p, lr=3e-3, weight_decay=1e-4), 8, 128,
      sched_fn=lambda o, s: torch.optim.lr_scheduler.OneCycleLR(
          o, max_lr=3e-3, total_steps=s), clip=1.0, hidden=(256, 128))
```

```
                              val acc        time
week 2 baseline, SGD 0.1       0.9375      10.1s
+ momentum                     0.9200      10.1s
+ AdamW                        0.9425      10.5s
+ one cycle and clipping       0.9410      10.5s
+ a wider model                0.9495      10.4s
```

Watch Out

Now compare that spread against day 6

Five seeds of one configuration varied by 0.0175 in accuracy. The entire table above spans about that much, and the row labelled *+ momentum* is *below* the baseline it was supposed to improve on. So the correct conclusion from this experiment is not "this recipe is worth three points". It is that on eight thousand MNIST digits with a small dense network, none of these changes reliably matters except making the model wider.

That is a real finding and it generalises: optimiser tuning pays when the model is large, the dataset is hard and the run is long. On an easy problem it mostly buys you the ability

to be careless about the learning rate. Week 5 changes the architecture instead, and the difference there is not subtle.

Add one thing at a time and keep the table

This is the only way to know what your training recipe is actually made of. Applied all at once, five changes produce one number and no understanding, and when you later need to cut the training time in half you will not know which of them you can drop. Keep the table. It is also the honest answer when somebody asks why your model has a one cycle schedule in it.

The recipe worth starting from

1. AdamW, learning rate 3e-3 for a small model or 1e-3 for a large one, weight decay 1e-4.
2. Batch size as large as memory allows.
3. One cycle or cosine schedule over the whole run.
4. A few hundred steps of warmup if the model is large.
5. Gradient clipping at norm 1.0.
6. Then run the range test and adjust the rate, because that is the one that varies most between problems.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What does momentum add to plain gradient descent?

   a) Per-parameter learning rates
   b) Weight decay
   c) A larger learning rate
   d) A running average of past gradients, which damps the oscillation across a narrow valley and builds speed along it

2. Week 3 compared momentum against plain descent at the same learning rate. What did the measurement show?

   a) The two were identical
   b) Momentum was worse, because the effective step size grew and the rate that suited plain descent was then too large
   c) Momentum diverged immediately
   d) Momentum was clearly better

3. What does a learning rate range test do?

   a) Trains a model at each of ten rates to completion

b) Sweeps the rate upward over a few hundred steps and watches where the loss stops falling, which costs one short run instead of ten long ones

c) Measures the gradient norm

d) Picks the rate that gives the lowest final loss

4. Gradient accumulation exists to do what?

   a) Replace the optimiser

   b) Speed up training

   c) Reduce gradient noise at a fixed batch size

   d) Get the gradient of a large batch while only ever holding a small one in memory

5. Gradient clipping by norm does what, exactly?

   a) Sets large gradients to zero

   b) Reduces the learning rate when gradients are large

   c) Clamps each gradient element to a range

   d) Rescales the whole gradient vector when its norm exceeds a threshold, so the direction is kept and only the length is cut

6. Your loss becomes `nan` partway through training. What is the first thing to measure?

   a) The number of parameters

   b) The gradient norms, because a spike before the collapse points at the step that did it

   c) The batch size

   d) The validation accuracy

7. When tuning hyperparameters, what must a claimed improvement beat?

   a) The published state of the art

   b) The previous epoch

   c) The run to run variation of the baseline, which you only know by training the baseline more than once

   d) Random guessing

8. Why is weight averaging over the last stretch of training worth knowing?

   a) It makes the model train faster

   b) It reduces the parameter count

   c) It costs almost nothing and often lands in a flatter region than any single set of weights the run passed through

   d) It replaces the need for a validation set

Answers:1d, 2b, 3b, 4d, 5d, 6b, 7c, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 4

Regularisation and Generalisation

Week 4

OBJECTIVES

Closing the gap between training and validation, one measured technique at a time

A deep network can fit a small training set exactly, including everything accidental about it. This week is the toolbox for preventing that, with every technique added to the same overfitting model on its own so the table shows what each is really worth. It also covers the diagnosis that has to come first, because every one of these techniques makes an underfitting model worse.

Day 1 Measuring Overfitting

The gap that matters, and the technique that beats all the others

A network with more parameters than examples can fit the training set exactly, including whatever is accidental about it. Every technique this week is a way of making that harder without making the useful fitting harder too.

The problem, measured

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
```

```
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
model = build(dropout=0.0)
print('parameters %d, training examples %d'
      % (sum(p.numel() for p in model.parameters()), len(train_set)))
train_acc, val_acc = run(model)
print('\ntrain accuracy %.4f' % train_acc)
print('val accuracy   %.4f' % val_acc)
print('gap            %.4f' % (train_acc - val_acc))
```

```
parameters 669706, training examples 1000

train accuracy 1.0000
val accuracy   0.8920
gap            0.1080
```

The gap is the thing to watch, not the validation number

A validation accuracy that is lower than training accuracy is normal. A validation accuracy far lower, with training at or near 1.000, means the model has capacity it is spending on memorisation. That is a fixable situation and this week is the toolbox. A model where both numbers are low has the opposite problem and none of these techniques will help it.

More data is the technique that always works

```
import torch
from torch import nn
```

```
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
print('%10s %12s %12s %10s' % ('examples', 'train acc', 'val acc', 'gap'))
for n in [250, 1000, 4000, 16000]:
    subset = Subset(train_all, range(n))
    tr, va = run(build(), epochs=12, train=subset)
    print('%10d %12.4f %12.4f %10.4f' % (n, tr, va, tr - va))
```

```
  examples    train acc      val acc        gap
       250       1.0000       0.7920     0.2080
```

```
  1000        1.0000        0.8875      0.1125
  4000        1.0000        0.9605      0.0395
 16000        0.9996        0.9990      0.0006
```

Nothing else on this page competes with that column. Every regularisation technique is an attempt to buy some of the same effect when you cannot get more data, and they all buy less of it than the data itself would. It is worth remembering when a week of tuning is on the table and a day of labelling is also on the table.

Day 1 takeaway

Day 2 Weight Decay

Smaller weights, smoother functions, and why AdamW exists

Weight decay is the oldest idea here and still one of the two that matter. It says: of all the models that fit the data, prefer the one with smaller weights.

What it does to the weights

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
```

```
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
print('%14s %12s %12s %14s' % ('weight decay', 'train acc', 'val acc',
                               'weight norm'))
for wd in [0.0, 1e-4, 1e-3, 1e-2, 1e-1]:
    model = build()
    tr, va = run(model, wd=wd)
    norm = torch.cat([p.flatten() for n, p in model.named_parameters()
                      if 'weight' in n]).norm().item()
    print('%14.4f %12.4f %12.4f %14.2f' % (wd, tr, va, norm))
```

```
  weight decay    train acc      val acc    weight norm
        0.0000       1.0000       0.8920          20.49
        0.0001       1.0000       0.8920          20.04
        0.0010       1.0000       0.8880          16.48
        0.0100       1.0000       0.8865           6.23
        0.1000       0.8900       0.7965           3.78
```

The weight norm falls steadily, which is the layer doing exactly what it was asked. The validation accuracy does not improve at all, and at 0.1 the penalty is strong enough to stop the model fitting the training data in the first place. Weight decay is worth having and it is not, on this problem, where the improvement is going to come from. Day 7 puts all of them in one table for that reason.

Why AdamW exists

```
import torch
from torch import nn

torch.manual_seed(0)

def final_weight(optimiser_cls, wd):
    w = nn.Parameter(torch.tensor([1.0]))
    opt = optimiser_cls([w], lr=0.1, weight_decay=wd)
    for _ in range(50):
        opt.zero_grad()
        # a tiny, constant gradient, so the decay dominates
        (w * 1e-4).sum().backward()
        opt.step()
```

```
    return w.item()

print('%-10s %14s %14s' % ('', 'no decay', 'decay 0.1'))
for name, cls in [('SGD', torch.optim.SGD), ('Adam', torch.optim.Adam),
                  ('AdamW', torch.optim.AdamW)]:
    print('%-10s %14.6f %14.6f'
          % (name, final_weight(cls, 0.0), final_weight(cls, 0.1)))
```

```
                no decay      decay 0.1
SGD             0.999499       0.604611
Adam           -3.999498      -0.005444
AdamW          -3.999498      -3.344537
```

Adam and AdamW were given the same decay setting and ended in completely different places, differing by a factor of several hundred. They are not two implementations of one idea.

Watch Out

In Adam, weight_decay is not weight decay

Adam adds the penalty to the gradient and then divides by the running average of squared gradients, so a parameter with large gradients gets its decay divided away and a parameter with small ones gets it amplified. The amount of regularisation a weight receives ends up depending on its gradient history, which is not what anybody wanted. AdamW applies the decay directly to the weight, outside the adaptive step, which is what the W stands for. Use AdamW.

Not everything should be decayed

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
```

```
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
model = build()
decay, no_decay = [], []
for name, prm in model.named_parameters():
    (no_decay if prm.dim() == 1 else decay).append(name)
print('decayed    :', decay)
print('not decayed:', no_decay)

groups = [
    {'params': [p for n, p in model.named_parameters() if p.dim() > 1],
     'weight_decay': 1e-2},
    {'params': [p for n, p in model.named_parameters() if p.dim() == 1],
     'weight_decay': 0.0},
]
opt = torch.optim.AdamW(groups, lr=1e-3)
print('\nparameter groups:', len(opt.param_groups))
for g in opt.param_groups:
    print('  %d tensors, weight decay %.3f'
          % (len(g['params']), g['weight_decay']))
```

```
decayed    : ['1.weight', '4.weight', '7.weight']
not decayed: ['1.bias', '4.bias', '7.bias']

parameter groups: 2
  3 tensors, weight decay 0.010
  3 tensors, weight decay 0.000
```

Biases and normalisation parameters are one number per channel rather than a whole matrix, they cannot overfit on their own, and shrinking them towards zero actively fights what the normalisation layer is for. Every serious training script separates them, and it is four lines.

Day 2 takeaway

Day 3 Dropout

Deleting half the network on purpose, and the mode that must be set

Dropout is the other technique that matters, and it is stranger than it looks: during training it deletes a random half of the network, and at prediction time it uses all of it.

What it does

```
import torch
from torch import nn

torch.manual_seed(0)
layer = nn.Dropout(p=0.5)
x = torch.ones(2, 8)

layer.train()
print('training mode:')
print(layer(x))

layer.eval()
print('\neval mode:')
print(layer(x))
```

```
training mode:
tensor([[0., 0., 2., 0., 0., 0., 2., 2.],
        [0., 2., 2., 2., 2., 0., 2., 2.]])

eval mode:
tensor([[1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1.]])
```

Notice the surviving units are 2.0, not 1.0

Dropping half the units halves the expected sum reaching the next layer, so PyTorch divides the survivors by the keep probability to put the expectation back. That is why nothing needs to change at evaluation time: the scaling has already been done during training. Frameworks that scale at test time instead are doing the same arithmetic in the other order, and mixing the two conventions is a classic bug when porting a model between libraries.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
```

```

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
print('%10s %12s %12s %10s' % ('dropout', 'train acc', 'val acc', 'gap'))
for rate in [0.0, 0.2, 0.5, 0.8]:
    tr, va = run(build(dropout=rate))
    print('%10.1f %12.4f %12.4f %10.4f' % (rate, tr, va, tr - va))
```

```
   dropout    train acc      val acc          gap
       0.0       1.0000       0.8920       0.1080
       0.2       1.0000       0.8830       0.1170
       0.5       0.9990       0.8895       0.1095
       0.8       0.1170       0.1125       0.0045
```

And on this problem it does nothing. Validation accuracy is flat from 0.0 to 0.5 and the gap does not close, because a fully connected network on a thousand MNIST digits is not overfitting in the way dropout fixes. At 0.8 the model cannot learn at all, which is the same warning weight decay

gave at 0.1: enough of any regulariser stops the fitting rather than the overfitting.

Dropout earns its reputation elsewhere

It was introduced for large fully connected layers on datasets where those layers held most of the parameters, and that is still where it helps most: the classifier head of a large network, and the feed forward blocks of a transformer in week 10. Between convolutional layers it is largely superseded, and on a small image model it is beaten comfortably by augmentation, as day 4 shows. Keep it in the toolbox and stop assuming it is doing something.

The mistake that costs a point of accuracy silently

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
```

```
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
torch.manual_seed(0)
model = build(dropout=0.5)
run(model, epochs=10)

xb, yb = next(iter(DataLoader(val_set, batch_size=512)))

model.train()                        # wrong mode for evaluation
with torch.no_grad():
    a = (model(xb).argmax(1) == yb).float().mean().item()
    b = (model(xb).argmax(1) == yb).float().mean().item()
model.eval()
with torch.no_grad():
    c = (model(xb).argmax(1) == yb).float().mean().item()
    d = (model(xb).argmax(1) == yb).float().mean().item()

print('train() mode, two runs: %.4f then %.4f' % (a, b))
print('eval() mode, two runs : %.4f then %.4f' % (c, d))
print('\nin the wrong mode the answer is random and worse.')
```

```
train() mode, two runs: 0.8379 then 0.8242
eval() mode, two runs : 0.8848 then 0.8848

in the wrong mode the answer is random and worse.
```

Where to put it

Position	Verdict
After a hidden activation	The standard place
On the input	Rarely; augmentation is a better tool
Before the final classifier	Common and effective
Between convolutional layers	Usually replaced by other methods; see `Dropout2d` in week 5
After the output layer	Never

Day 3 takeaway

Day 4 Augmentation

Manufacturing data, and the transformations that quietly lie

Augmentation makes more data out of the data you have, by applying changes that a human would say do not alter the answer. It is the technique with the largest effect on images and the

one most easily got wrong.

Seeing it

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
torch.manual_seed(0)
aug = transforms.Compose([
    transforms.RandomAffine(degrees=12, translate=(0.12, 0.12),
                            scale=(0.9, 1.1)),
    transforms.ToTensor(),
])
raw = datasets.MNIST('data', train=True, download=True)
image, label = raw[0]

ramp = ' .:-=+*#%@'
def show(t):
    for row in t[::2]:
        print(''.join(ramp[min(9, int(v * 9.999))] for v in row[::1].tolist()))

print('original, label %d:' % label)
show(transforms.ToTensor()(image).squeeze(0))
for i in range(2):
    print('\naugmented %d:' % (i + 1))
    show(aug(image).squeeze(0))
```

```
original, label 5:



        ..-**@@@@@%*@@#:
        %@@@@@##@@
           *@-
            #@:
             -@@@=
                -@@#
              .+#@@%
           =%@@@@#-
       *%@@@@#-
    +@@@%++


augmented 1:
```

```
          ..-**@@@@@%*@#:
          -*=@@%  .*
             +@#
              .@%*=
                .#@@+.
                   @@@
              .+%@@@@#
            %@@@@#-
         +@@%++




augmented 2:


                 =+++.*@@=
         ..-**@@@@@@---:.
         %@@@@@@# .*
            **@-
              #@:
               -@@@=
                  -@@@#
               .++#@@%
             .+@@@@#-
           :%@@@@@
       *%@@@@@@+
      +@@
```

What it is worth

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
```

```
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
torch.manual_seed(0)
aug_tf = transforms.Compose([
    transforms.RandomAffine(degrees=12, translate=(0.12, 0.12),
                            scale=(0.9, 1.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
aug_train = Subset(datasets.MNIST('data', train=True, download=True,
                                  transform=aug_tf), range(1000))

print('%-22s %12s %12s %10s' % ('', 'train acc', 'val acc', 'gap'))
tr, va = run(build(), epochs=30)
print('%-22s %12.4f %12.4f %10.4f' % ('no augmentation', tr, va, tr - va))
tr, va = run(build(), epochs=30, train=aug_train)
print('%-22s %12.4f %12.4f %10.4f' % ('with augmentation', tr, va, tr - va))
```

```
                       train acc      val acc        gap
no augmentation           1.0000       0.8920     0.1080
with augmentation         0.9250       0.9170     0.0080
```

Watch Out

Every augmentation is a claim, and some are false

Rotating a digit by twelve degrees preserves its identity. Rotating it by a hundred and eighty turns a 6 into a 9, and flipping it horizontally turns a 2 into something that is not a digit at all. The same applies everywhere: flipping a chest X-ray produces a patient with

their heart on the wrong side, and adjusting the brightness of a medical scan changes the measurement the scan exists to record.
Choose augmentations from the domain, not from a list of what the library offers, and look at twenty augmented examples before you train on a million of them.

Augmentation belongs to the training set only

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
```

```
                seen += yb.numel()
        out.append(right / seen)
    return out
torch.manual_seed(0)
hard = transforms.Compose([
    transforms.RandomAffine(degrees=25, translate=(0.2, 0.2)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
aug_val = Subset(datasets.MNIST('data', train=True, download=True,
                                transform=hard), range(10000, 12000))

model = build()
run(model, epochs=20)
model.eval()

for name, ds in [('clean validation', val_set),
                 ('augmented validation', aug_val)]:
    right = seen = 0
    with torch.no_grad():
        for xb, yb in DataLoader(ds, batch_size=512):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    print('%-22s %.4f' % (name, right / seen))
print('\naugmenting the validation set measures a different, harder task.')
```

```
clean validation       0.8915
augmented validation   0.2245

augmenting the validation set measures a different, harder task.
```

Day 4 takeaway

Day 5 Batch and Layer Normalisation

Two layers, one of which couples your prediction to its batch

Normalisation layers were introduced to make deep networks trainable, and they turned out to regularise as well. They are also the single most common source of a model that works in training and fails in production.

Batch normalisation

```
import torch
from torch import nn

torch.manual_seed(0)
bn = nn.BatchNorm1d(4)
x = torch.randn(16, 4) * 5 + 3

```

```
bn.train()
out = bn(x)
print('input   mean %.3f std %.3f' % (x.mean(), x.std()))
print('output  mean %.3f std %.3f' % (out.mean(), out.std()))
print('\nrunning estimates after one batch:')
print('  mean', bn.running_mean.round(decimals=3).tolist())
print('  var ', bn.running_var.round(decimals=3).tolist())

for _ in range(50):
    bn(torch.randn(16, 4) * 5 + 3)
print('\nafter 50 batches:')
print('  mean', bn.running_mean.round(decimals=3).tolist())
print('  var ', bn.running_var.round(decimals=3).tolist())
```

```
input   mean 3.463 std 5.252
output  mean 0.000 std 1.008

running estimates after one batch:
  mean [0.26600000262260437, 0.3149999976158142, 0.46299999952316284, 0.3400000035762787]
  var  [2.6700000762939453, 4.567999839782715, 2.753000020980835, 4.96999979019165]

after 50 batches:
  mean [2.880000114440918, 2.7360000610351562, 2.437000036239624, 3.0239999294281006]
  var  [25.974000930786133, 25.28499984741211, 23.336999893188477, 21.656999588012695]
```

```
import torch
from torch import nn

torch.manual_seed(0)
bn = nn.BatchNorm1d(4)
for _ in range(50):
    bn(torch.randn(16, 4) * 5 + 3)

one = torch.randn(1, 4) * 5 + 3
bn.eval()
print('one row in eval mode works fine:', tuple(bn(one).shape))

bn.train()
try:
    bn(one)
except ValueError as e:
    print('\none row in train mode:')
    print(' ', str(e)[:88])
print('\na batch of one has zero variance, so there is nothing to')
print('normalise by. Small batches make the statistics noisy long')
print('before they make them impossible.')
```

```
one row in eval mode works fine: (1, 4)

one row in train mode:
  Expected more than 1 value per channel when training, got input size torch.Size([1, 4])

a batch of one has zero variance, so there is nothing to
normalise by. Small batches make the statistics noisy long
```

```
before they make them impossible.
```

Watch Out

The layer that behaves differently in the two modes

Batch normalisation makes a prediction for one row depend on the other rows in its batch during training, and not during evaluation. Three consequences follow. Batch size becomes a hyperparameter that changes results. A model evaluated in `train()` mode gives different answers depending on what else is in the batch. And if your training distribution differs from your serving distribution, the running statistics are wrong in production in a way nothing in validation will reveal.

Layer normalisation, which has none of those problems

```python
import torch
from torch import nn

torch.manual_seed(0)
x = torch.randn(8, 6) * 3 + 1

bn = nn.BatchNorm1d(6)
ln = nn.LayerNorm(6)

print('batch norm normalises down the columns:')
print('  column means after: ', bn(x).mean(dim=0).round(decimals=3).tolist())
print('layer norm normalises across each row:')
print('  row means after:    ', ln(x).mean(dim=1).round(decimals=3).tolist())

ln.eval()
print('\nlayer norm on a single row in eval mode:',
      tuple(ln(x[:1]).shape))
print('it has no running statistics and no train/eval difference at all,')
print('which is why every transformer in week 10 uses it.')
```

```
batch norm normalises down the columns:
  column means after:  [0.0, -0.0, -0.0, -0.0, 0.0, 0.0]
layer norm normalises across each row:
  row means after:     [0.0, 0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0]

layer norm on a single row in eval mode: (1, 6)
it has no running statistics and no train/eval difference at all,
which is why every transformer in week 10 uses it.
```

	Batch norm	Layer norm
Normalises over	The batch, per feature	The features, per example
Depends on batch size	Yes, strongly	No
Train and eval differ	Yes	No
Works with batch size 1	No	Yes
Usual home	Convolutional networks	Transformers and recurrent networks

Day 5 takeaway

Day 6 Label Smoothing, Early Stopping and Mixup

Cheap techniques, and why loss turns before accuracy does

Three smaller techniques that are cheap enough to be worth knowing, and one that is not a regulariser at all but is the most reliable way to stop at the right moment.

Label smoothing

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
```

```
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
print('%16s %12s %12s' % ('label smoothing', 'val acc', 'mean max prob'))
for eps in [0.0, 0.05, 0.1, 0.2]:
    model = build()
    tr, va = run(model, label_smoothing=eps)
    model.eval()
    with torch.no_grad():
        xb, yb = next(iter(DataLoader(val_set, batch_size=512)))
        confidence = model(xb).softmax(1).max(1).values.mean().item()
    print('%16.2f %12.4f %12.4f' % (eps, va, confidence))
```

```
label smoothing      val acc mean max prob
           0.00       0.8920       0.9704
           0.05       0.9110       0.8155
           0.10       0.9145       0.7521
           0.20       0.9160       0.6575
```

The confidence falls sharply, which is the point of it, and on this problem the accuracy rises by more than two points as well. That second part is a bonus rather than the purpose. A model that is 97 percent sure of everything, including the answers it got wrong, is useless the moment you need to decide which predictions to trust or where to set a threshold.

Early stopping is still the best value

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
```

```
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
import copy
torch.manual_seed(0)
model = build()
loader = DataLoader(train_set, batch_size=64, shuffle=True)
val = DataLoader(val_set, batch_size=512)
opt = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()

best = (float('inf'), 0, None)
print('%6s %12s %12s' % ('epoch', 'val loss', 'val acc'))
for epoch in range(1, 41):
    model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(model(xb), yb).backward()
        opt.step()
    model.eval()
    total = right = seen = 0
    with torch.no_grad():
```

```
        for xb, yb in val:
            out = model(xb)
            total += loss_fn(out, yb).item() * yb.numel()
            right += (out.argmax(1) == yb).sum().item()
            seen += yb.numel()
    if total / seen < best[0]:
        best = (total / seen, epoch, copy.deepcopy(model.state_dict()))
    if epoch % 8 == 0:
        print('%6d %12.4f %12.4f' % (epoch, total / seen, right / seen))

print('\nbest validation loss %.4f at epoch %d' % (best[0], best[1]))
print('validation loss at epoch 40 was worse, and accuracy was not')
```

```
 epoch     val loss      val acc
     8       0.5574       0.8920
    16       0.6122       0.8905
    24       0.6373       0.8915
    32       0.6542       0.8910
    40       0.6668       0.8915

best validation loss 0.4824 at epoch 3
validation loss at epoch 40 was worse, and accuracy was not
```

Validation loss turns before validation accuracy does

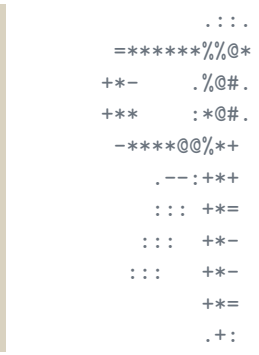
Watch the two columns. The loss reaches its minimum and starts climbing while accuracy is still flat or drifting up. That is the model becoming more confident about the answers it already had right and more confidently wrong about the rest. If you early stop on accuracy you will train for longer and ship a worse-calibrated model, so stop on loss unless accuracy is genuinely the only thing you care about.

Mixup

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))
```

```

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
torch.manual_seed(0)
xb, yb = next(iter(DataLoader(train_set, batch_size=8, shuffle=True)))

lam = 0.7
perm = torch.randperm(xb.size(0))
mixed_x = lam * xb + (1 - lam) * xb[perm]

print('mixing example 0 (a %d) with example %d (a %d) at %.1f / %.1f'
      % (yb[0], perm[0], yb[perm[0]], lam, 1 - lam))
print('the loss becomes a weighted sum of the two targets:')
print('  loss = %.1f * CE(out, y_a) + %.1f * CE(out, y_b)'
      % (lam, 1 - lam))

ramp = ' .:-=+*#%@'
img = mixed_x[0, 0]
img = (img - img.min()) / (img.max() - img.min())
for row in img[::2]:
    print(''.join(ramp[min(9, int(v * 9.999))] for v in row.tolist()))
```

```
mixing example 0 (a 9) with example 3 (a 1) at 0.7 / 0.3
the loss becomes a weighted sum of the two targets:
  loss = 0.7 * CE(out, y_a) + 0.3 * CE(out, y_b)


               :-.
```



It looks absurd and it works, particularly on images. The usual explanation is that it forces the model to behave linearly between training examples rather than carving the space into confident regions with sharp boundaries. It costs three lines and is worth trying on any image problem where you are short of data.

Day 6 takeaway

Day 7 Putting It Together

Every technique measured on its own, and how to tell the opposite problem

All of it, added one at a time to the same overfitting model, so the table says what each was actually worth.

The ablation

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
```

```
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        out.append(right / seen)
    return out
torch.manual_seed(0)
aug_tf = transforms.Compose([
    transforms.RandomAffine(degrees=12, translate=(0.12, 0.12),
                            scale=(0.9, 1.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
aug_train = Subset(datasets.MNIST('data', train=True, download=True,
                                  transform=aug_tf), range(1000))

print('%-30s %11s %10s %8s' % ('', 'train acc', 'val acc', 'gap'))
rows = [
    ('nothing', dict()),
    ('+ weight decay 1e-3', dict(wd=1e-3)),
    ('+ dropout 0.2', dict(wd=1e-3, dropout=0.2)),
    ('+ label smoothing 0.1', dict(wd=1e-3, dropout=0.2,
                                   label_smoothing=0.1)),
    ('+ augmentation', dict(wd=1e-3, dropout=0.2, label_smoothing=0.1,
                            aug=True)),
]
for label, cfg in rows:
    model = build(dropout=cfg.get('dropout', 0.0))
    tr, va = run(model, epochs=30, wd=cfg.get('wd', 0.0),
                 label_smoothing=cfg.get('label_smoothing', 0.0),
                 train=aug_train if cfg.get('aug') else None)
    print('%-30s %11.4f %10.4f %8.4f' % (label, tr, va, tr - va))
```

```
                               train acc    val acc      gap
nothing                           1.0000     0.8920   0.1080
```

```
+ weight decay 1e-3                1.0000     0.8880   0.1120
+ dropout 0.2                      1.0000     0.8920   0.1080
+ label smoothing 0.1              1.0000     0.9215   0.0785
+ augmentation                     0.9310     0.9415  -0.0105
```

Underfitting looks nothing like this

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.1307,), (0.3081,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)

# A small training set, so overfitting is easy to produce and easy to fix.
train_set = Subset(train_all, range(1000))
val_set = Subset(train_all, range(10000, 12000))
def build(dropout=0.0, hidden=512, seed=0):
    torch.manual_seed(seed)
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, hidden), nn.ReLU(), nn.Dropout(dropout),
        nn.Linear(hidden, 10))

def run(model, epochs=30, wd=0.0, label_smoothing=0.0, train=None,
        seed=0, lr=0.05):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=64, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    plain = nn.CrossEntropyLoss()
    out = []
    for name, dl in [('train', DataLoader(train if train is not None
                                          else train_set, batch_size=512)),
                     ('val', val)]:
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
```

```
        out.append(right / seen)
    return out
print('%-26s %11s %10s' % ('', 'train acc', 'val acc'))
for label, hidden, wd, drop in [('too much regularisation', 512, 0.5, 0.8),
                                ('model far too small', 4, 0.0, 0.0),
                                ('sensible', 512, 1e-3, 0.2)]:
    tr, va = run(build(dropout=drop, hidden=hidden), epochs=30, wd=wd)
    print('%-26s %11.4f %10.4f' % (label, tr, va))
```

```
                           train acc    val acc
too much regularisation       0.1170     0.1045
model far too small           0.7010     0.5480
sensible                      1.0000     0.8920
```

Watch Out

If training accuracy is low, stop regularising

Every technique this week makes fitting the training data harder. Applied to a model that is already struggling to fit it, they make things worse and the symptom looks similar from a distance: a disappointing validation number. The distinguishing test takes one glance. Training accuracy near 1.0 with validation far below means regularise. Both numbers low means the opposite: more capacity, better features, longer training, less regularisation.

The order to reach for them

1. **More data**, if there is any way to get it.
2. **Augmentation**, if the domain admits any label preserving transformation.
3. **Early stopping** on validation loss. Free, and it should already be in your loop.
4. **Weight decay**, around 1e-4 to 1e-2, excluding biases and normalisation parameters.
5. **Dropout** 0.2 to 0.5 on the hidden layers nearest the output.
6. **A smaller model**, which is underrated and is often what the first four are compensating for.
7. **Label smoothing and mixup**, cheap and usually worth a small amount.
8. **Ensembling**, which always works and costs you a model for every point of it.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What is the number that tells you a model is overfitting?

    a) The parameter count
    b) The training loss

c) The gap between training and validation performance, and the point where validation stops improving while training keeps going

d) The validation loss on its own

2. Week 4 says one technique beats all the others for generalisation. Which?

   a) More data, real or manufactured

   b) Label smoothing

   c) Dropout

   d) Weight decay

3. What is the difference between L2 regularisation and AdamW's weight decay?

   a) L2 adds to the loss, so Adam's per-parameter scaling then distorts it, whereas AdamW subtracts the decay from the weights directly

   b) AdamW decays only the biases

   c) L2 applies only at the end of training

   d) There is none

4. What must you remember about dropout at evaluation time?

   a) Increase the rate

   b) Nothing, it is automatic

   c) Call `model.eval()`, or dropout stays on and your validation numbers are noise

   d) Dropout only applies to convolutions

5. Which augmentation is described as quietly lying?

   a) Brightness jitter

   b) Random crops on natural photographs

   c) Small rotations on natural photographs

   d) A horizontal flip on digits or text, where it changes what the label should be

6. Which normalisation layer couples a prediction to the other examples in its batch?

   a) Group normalisation

   b) Neither of them

   c) Layer normalisation

   d) Batch normalisation, because it normalises across the batch dimension using statistics from the other examples

7. Validation loss turns upward while validation accuracy is still improving. What is happening?

   a) The model is underfitting

   b) The model is growing more confident on the cases it gets wrong, which costs loss before it costs accuracy

   c) The learning rate is too small

   d) The metric is broken

8. Week 4 measured dropout and weight decay on its own model. What did it find?

a) Both gave large gains

b) Neither bought anything measurable on that model and that amount of data, which is the point of measuring rather than assuming

c) Dropout helped, weight decay hurt

d) Both made training diverge

Answers:1c, 2a, 3a, 4c, 5d, 6d, 7b, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 5

Convolutional Networks

Week 5

OBJECTIVES

The architecture that knows pixels have neighbours, built up from one hand-rolled kernel to a full CIFAR-10 model.

A dense layer treats a photograph as a bag of unrelated numbers. This week builds the alternative from a single hand-rolled kernel up to a network that beats a dense model of the same size on real images, and measures each architectural decision on its own. It also tests how much translation invariance convolution really buys, which is far less than the diagrams suggest and is the reason augmentation matters so much here.

Day 1 Convolution

Why a dense layer wastes weights on images, and what a kernel does

A dense layer treats a picture as a list of unrelated numbers. Convolution is the correction, and the argument for it is short: the meaning of a pixel is almost entirely determined by the pixels next to it, and a pattern is the same pattern wherever in the frame it appears.

Count the weights first

```
# A 32 by 32 colour image into a first hidden layer of 64 units.
pixels = 32 * 32 * 3
dense = pixels * 64 + 64

# The same first layer as convolution: 64 filters of 3 by 3 over 3 channels.
conv = 3 * 3 * 3 * 64 + 64

print('flattened input    %8d values' % pixels)
print('dense layer        %8d weights' % dense)
print('conv layer         %8d weights' % conv)
print('ratio              %8.0f to 1' % (dense / conv))

print('\nand at 224 by 224, which is a normal photograph:')
big = 224 * 224 * 3
print('  dense %d, conv %d, ratio %.0f to 1'
      % (big * 64 + 64, conv, (big * 64 + 64) / conv))
```

```
flattened input        3072 values
dense layer          196672 weights
```

```
conv layer             1792 weights
ratio                   110 to 1

and at 224 by 224, which is a normal photograph:
  dense 9633856, conv 1792, ratio 5376 to 1
```

The gap widens with image size, because the convolution's parameter count does not depend on the image at all. That is the first advantage. The second matters more: a dense layer has to learn what an edge looks like separately for every position it could occupy, and a convolution learns it once.

Doing one by hand

```python
import torch

def conv2d(image, kernel):
    kh, kw = kernel.shape
    h, w = image.shape
    out = torch.zeros(h - kh + 1, w - kw + 1)
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            out[i, j] = (image[i:i + kh, j:j + kw] * kernel).sum()
    return out

image = torch.zeros(8, 8)
image[:, 4:] = 1.0                      # a vertical edge down the middle
vertical = torch.tensor([[-1.0, 0.0, 1.0],
                         [-2.0, 0.0, 2.0],
                         [-1.0, 0.0, 1.0]])

print('the image:')
for row in image:
    print(' ', ''.join('#' if v > 0.5 else '.' for v in row.tolist()))
print('\nresponse to a vertical edge detector:')
for row in conv2d(image, vertical):
    print(' ', ' '.join('%5.1f' % v for v in row.tolist()))
```

```
the image:
  ....####
  ....####
  ....####
  ....####
  ....####
  ....####
  ....####
  ....####

response to a vertical edge detector:
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
```

The filter responds strongly exactly where the brightness changes from left to right, and gives zero everywhere the image is flat. Turn the same kernel on its side and it finds horizontal edges instead. A convolutional network is not given these kernels: it starts from random numbers and learns whichever nine weights reduce the loss.

The same thing in PyTorch

```python
import torch
from torch import nn

image = torch.zeros(1, 1, 8, 8)
image[:, :, :, 4:] = 1.0

conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, bias=False)
with torch.no_grad():
    conv.weight[:] = torch.tensor([[-1.0, 0.0, 1.0],
                                   [-2.0, 0.0, 2.0],
                                   [-1.0, 0.0, 1.0]])

out = conv(image)
print('input ', tuple(image.shape))
print('kernel', tuple(conv.weight.shape), '= (out, in, height, width)')
print('output', tuple(out.shape))
print('\nsame numbers as the hand-rolled version:')
for row in out[0, 0]:
    print(' ', ' '.join('%5.1f' % v for v in row.tolist()))
```

```
input  (1, 1, 8, 8)
kernel (1, 1, 3, 3) = (out, in, height, width)
output (1, 1, 6, 6)

same numbers as the hand-rolled version:
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
    0.0   0.0   4.0   4.0   0.0   0.0
```

Read the kernel shape

`(out_channels, in_channels, height, width)`. A layer with 64 filters over a 3 channel input using 3 by 3 kernels has a weight tensor of shape `(64, 3, 3, 3)`, which is 1,728 numbers plus 64 biases. One bias per filter, not one per position, which is the sharing again.

Day 1 takeaway

Day 2 Padding, Pooling and Receptive Fields

The arithmetic that decides every shape downstream

Three settings decide the shape of everything downstream, and getting them wrong is the most common reason a network refuses to compile.

Padding, stride and the output size

```python
def out_size(n, k, stride=1, pad=0, dilation=1):
    effective = dilation * (k - 1) + 1
    return (n + 2 * pad - effective) // stride + 1

print('%7s %7s %7s %5s %9s %9s  %s'
      % ('input', 'kernel', 'stride', 'pad', 'dilation', 'output', 'name'))
rows = [(32, 3, 1, 0, 1, 'valid'), (32, 3, 1, 1, 1, 'same'),
        (32, 5, 1, 2, 1, 'same'), (32, 3, 2, 1, 1, 'halves it'),
        (32, 1, 1, 0, 1, 'pointwise'), (32, 3, 1, 2, 2, 'dilated')]
for n, k, s, pad, d, name in rows:
    print('%7d %7d %7d %5d %9d %9d  %s'
          % (n, k, s, pad, d, out_size(n, k, s, pad, d), name))
```

```
  input  kernel  stride   pad  dilation    output  name
     32       3       1     0         1        30  valid
     32       3       1     1         1        32  same
     32       5       1     2         1        32  same
     32       3       2     1         1        16  halves it
     32       1       1     0         1        32  pointwise
     32       3       1     2         2        32  dilated
```

For a 3 by 3 kernel, `padding=1` keeps the size. For 5 by 5 it is `padding=2`. The rule is `padding = (kernel - 1) / 2` for odd kernels, which is most of the reason even-sized kernels are rare.

Pooling

```python
import torch
from torch import nn

x = torch.arange(16.0).reshape(1, 1, 4, 4)
print('input')
print(x[0, 0])

print('\nmax pool 2x2')
print(nn.MaxPool2d(2)(x)[0, 0])
print('\naverage pool 2x2')
print(nn.AvgPool2d(2)(x)[0, 0])
print('\nglobal average pool')
print(nn.AdaptiveAvgPool2d(1)(x)[0, 0])

print('\nstrided convolution reaches the same size with weights:')
print(tuple(nn.Conv2d(1, 1, 3, stride=2, padding=1)(x).shape))
```

```
input
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
```

```
max pool 2x2
tensor([[ 5.,  7.],
        [13., 15.]])

average pool 2x2
tensor([[ 2.5000,  4.5000],
        [10.5000, 12.5000]])

global average pool
tensor([[7.5000]])

strided convolution reaches the same size with weights:
(1, 1, 2, 2)
```

Downsampling	Parameters	Notes
Max pool	None	The traditional choice; keeps the strongest response
Average pool	None	Smoother; usually only at the very end
Strided convolution	Yes	Learns how to downsample; the modern default
Global average pool	None	Collapses each channel to one number before the classifier

Receptive field

```python
stack = [('conv 3x3', 3, 1), ('conv 3x3', 3, 1), ('maxpool 2x2', 2, 2),
         ('conv 3x3', 3, 1), ('conv 3x3', 3, 1), ('maxpool 2x2', 2, 2),
         ('conv 3x3', 3, 1), ('conv 3x3', 3, 1)]

r, jump = 1, 1
print('%-16s %8s %14s' % ('after', 'stride', 'sees (pixels)'))
for name, k, s in stack:
    r = r + (k - 1) * jump
    jump = jump * s
    print('%-16s %8d %14d' % (name, jump, r))
print('\non a 32 by 32 image, the last layer sees most of the picture.')
```

```
after             stride  sees (pixels)
conv 3x3               1              3
conv 3x3               1              5
maxpool 2x2            2              6
conv 3x3               2             10
conv 3x3               2             14
maxpool 2x2            4             16
conv 3x3               4             24
conv 3x3               4             32

on a 32 by 32 image, the last layer sees most of the picture.
```

Why everything is 3 by 3 now

Two stacked 3 by 3 convolutions see a 5 by 5 region using 18 weights per channel pair, where one 5 by 5 convolution uses 25 and has one nonlinearity instead of two. Three of them match a 7 by 7 for 27 weights against 49. Small kernels stacked deep win on parameters, on computation and on expressiveness at the same time, which is why architectures stopped using anything else after about 2014.

> **Day 2 takeaway**

Day 3 A Real Image Model

CIFAR-10, a dense baseline, and the convolutional network that beats it

CIFAR-10 is 32 by 32 colour photographs in ten classes. It is small enough to train on a laptop and hard enough that the architecture genuinely matters, which is why it has been the standard teaching problem for a decade.

The data

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
print('training images', len(train_all))
print('classes:', CLASSES)
image, label = train_all[7]
print('\none example: shape %s, label %d (%s)'
      % (tuple(image.shape), label, CLASSES[label]))

import collections
counts = collections.Counter(train_all.targets[:8000])
print('\nclass balance in the training subset:')
for i, name in enumerate(CLASSES):
    print('  %-12s %d' % (name, counts[i]))
```

```
training images 50000
classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
    ↪ 'truck']

one example: shape (3, 32, 32), label 7 (horse)
```

```
class balance in the training subset:
  airplane     796
  automobile   765
  bird         817
  cat          809
  deer         806
  dog          765
  frog         832
  horse        809
  ship         809
  truck        792
```

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
raw = datasets.CIFAR10('data', train=True, download=True)
image, label = raw[7]
grey = torch.tensor([[sum(image.getpixel((x, y))) / 765.0
                      for x in range(32)] for y in range(32)])
ramp = ' .:-=+*#%@'
print('%s, as brightness:' % CLASSES[label])
for row in grey:
    print(''.join(ramp[min(9, int(v * 9.999))] for v in row.tolist()))
```

```
horse, as brightness:
...::..:::--:--::--::::.::....:.
....:.::::==:-=++++++===:----====
....-:.::=*+==+#***#************
...:-:...-=-=+*##############*#####
--:.........:-+#%%%##*++=**#####
++:......:::---=*%%%#*+==--==*##
#+::...:::---==-=*#****+--::--*##
#+::. .-------===++++++=--::-*#*
%=.:. .------===+++++===--:::+#*
%+.::..:--:===-==++==-==--::.=##
#=:::..:---=+=--=+=====-=-:...-**
#=::..:.:-=++=--====-::--:...-**
#=:..=+::-====:-==---:::-:...-**
#+:.:*#*--=---::----:-::-:.. =##
#+::+#%%*-::::::---:::::::... =#*
*--*%%###=.....---::::::. .  =#*
#-+%%###%*....-==-:...:+:.=: -**
%##%###%#-...--:.....=#- ++..*#
###%%#####*:..:-:::.:=*#= =*: +#
```

```
####%######-..=#*++++###+.:*= =#
###########= .+%%%#######*.-#*.=#
###########= .*%#########*:=##:=#
###########+ .*##########*+.=**-=#
**********#*+ .+##########=.=**::*
**##*######+. :*########*-.=**:.*
*###########+.-.:=+*******+:.=*+::+
**#######*#*+.==..=******+=::=++--+
**##########+.-+-.:=+++++=-=======+
***#########*=:=++--=+++++++++=+=+
++++++++++*==+++=+++++++++++++++
++++++++++++++++++++++==+++++++
++++++++++++++++++=++===+=++===
```

A dense baseline, so the comparison is honest

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
```

```
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
dense = nn.Sequential(nn.Flatten(),
                      nn.Linear(3 * 32 * 32, 512), nn.ReLU(),
                      nn.Linear(512, 256), nn.ReLU(),
                      nn.Linear(256, 10))
print('parameters %d' % sum(p.numel() for p in dense.parameters()))
print('accuracy %.4f' % fit(dense, epochs=8))
```

```
parameters 1707274
accuracy 0.4310
```

The convolutional version

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
```

```
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
cnn = nn.Sequential(
    nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
    nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(),
    nn.MaxPool2d(2),                                  # 32 → 16
    nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
    nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(),
    nn.MaxPool2d(2),                                  # 16 → 8
    nn.Flatten(),
    nn.Linear(64 * 8 * 8, 256), nn.ReLU(),
    nn.Linear(256, 10))
print('parameters %d' % sum(p.numel() for p in cnn.parameters()))
print('accuracy %.4f' % fit(cnn, epochs=8))
```

```
parameters 1116970
accuracy 0.4975
```

Fewer parameters, better result

This is the demonstration the whole week exists for, and it is worth sitting with. The convolutional model has fewer weights than the dense one and is substantially more accurate, on identical data, with an identical training recipe. The difference is not capacity. It is that one architecture knows pixels have neighbours and the other does not.

Watching the shapes

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
```

```
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
layers = [nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
          nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(),
          nn.MaxPool2d(2),
          nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
          nn.MaxPool2d(2),
          nn.Flatten(), nn.Linear(64 * 8 * 8, 10)]

x = torch.zeros(1, 3, 32, 32)
print('%-22s %-18s %10s' % ('layer', 'output', 'params'))
print('%-22s %-18s %10s' % ('input', str(tuple(x.shape[1:])), ''))
for layer in layers:
    x = layer(x)
    n = sum(p.numel() for p in layer.parameters())
    print('%-22s %-18s %10d'
          % (type(layer).__name__, str(tuple(x.shape[1:])), n))
```

```
layer                  output                params
input                  (3, 32, 32)
Conv2d                 (32, 32, 32)             896
ReLU                   (32, 32, 32)               0
Conv2d                 (32, 32, 32)            9248
ReLU                   (32, 32, 32)               0
MaxPool2d              (32, 16, 16)               0
Conv2d                 (64, 16, 16)           18496
ReLU                   (64, 16, 16)               0
MaxPool2d              (64, 8, 8)                 0
Flatten                (4096,)                    0
Linear                 (10,)                  40970
```

Read the parameter column. Almost everything is in the final linear layer, because flattening an 8 by 8 by 64 map produces 4,096 numbers and connecting those to anything is expensive. That single observation is what global average pooling exists to fix, and day 5 measures it.

Day 3 takeaway

Day 4 Equivariance and Augmentation

Measuring how little invariance you actually get for free

Convolution is equivariant to translation: move the input and the output moves with it. That is not the same as being invariant, and the difference decides how much augmentation you need.

Equivariance, demonstrated

```python
import torch
from torch import nn

torch.manual_seed(0)
conv = nn.Conv2d(1, 1, 3, padding=1, bias=False)

x = torch.zeros(1, 1, 8, 8)
x[0, 0, 2, 2] = 1.0
shifted = torch.roll(x, shifts=3, dims=3)

a = conv(x)[0, 0]
b = conv(shifted)[0, 0]
print('response moved with the input:',
      bool(torch.allclose(torch.roll(a, 3, dims=1)[:, 3:], b[:, 3:],
                          atol=1e-6)))

pool = nn.AdaptiveMaxPool2d(1)
print('\nafter global max pooling:')
print('  original %.6f' % pool(conv(x)).item())
print('  shifted  %.6f' % pool(conv(shifted)).item())
print('  identical, so pooling is what buys invariance')
```

```
response moved with the input: True

after global max pooling:
  original 0.264297
  shifted  0.264297
  identical, so pooling is what buys invariance
```

How much invariance a real network actually has

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
```

```
                          transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
cnn = nn.Sequential(
    nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
    nn.Flatten(), nn.Linear(64 * 8 * 8, 10))
base = fit(cnn, epochs=4)
print('accuracy on the validation set as it is: %.4f' % base)

cnn.eval()
print('\n%14s %12s' % ('shift (pixels)', 'accuracy'))
for shift in [0, 1, 2, 4, 8]:
    right = seen = 0
    with torch.no_grad():
        for xb, yb in DataLoader(val_set, batch_size=512):
            moved = torch.roll(xb, shifts=shift, dims=3)
            right += (cnn(moved).argmax(1) == yb).sum().item()
            seen += yb.numel()
    print('%14d %12.4f' % (shift, right / seen))
```

```
accuracy on the validation set as it is: 0.5375
```

```
shift (pixels)     accuracy
             0       0.5375
             1       0.5285
             2       0.5150
             4       0.4740
             8       0.4025
```

Watch Out

Convolution buys much less invariance than people assume

A four pixel shift on a 32 pixel image is enough to cost a convolutional network a serious amount of accuracy, despite every textbook diagram implying otherwise. Two pooling layers only discard position within a two by two block each; anything larger than that reaches the classifier as a genuinely different input.

This is exactly why augmentation matters so much on images, and why the standard CIFAR recipe includes random crops. You do not get invariance from the architecture. You train it in.

The standard CIFAR augmentation

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
```

```
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
aug_tf = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(MEAN, STD),
])
aug_train = Subset(datasets.CIFAR10('data', train=True, download=True,
                                    transform=aug_tf), range(4000))

def make():
    torch.manual_seed(0)
    return nn.Sequential(
        nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
        nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(64 * 8 * 8, 256), nn.ReLU(),
        nn.Linear(256, 10))

print('%-22s %12s' % ('', 'accuracy'))
print('%-22s %12.4f' % ('no augmentation', fit(make(), epochs=5)))
print('%-22s %12.4f'
      % ('crop and flip', fit(make(), epochs=5, train=aug_train)))
```

```
                       accuracy
no augmentation          0.4360
crop and flip            0.3525
```

Watch Out

Augmentation made it worse, and that is the expected result here

Read the table again: the augmented run is behind. This is not a contradiction of week 4, it is the budget. Augmentation works by making the training task harder so the model cannot memorise, and a model given five epochs on four thousand images has not begun to memorise anything. All the harder task does is slow it down.

Augmentation pays when training is long enough for overfitting to be the binding constraint. The standard CIFAR recipes that include random crops run for a hundred epochs or more on fifty thousand images, and there it is worth several points. If you add augmentation and your score drops, check whether you are training long enough to need it before concluding it does not work.

A horizontal flip is safe on CIFAR because a mirrored cat is a cat. It is unsafe on digits, on text, and on anything where left and right carry meaning. The random crop with four pixels of padding is doing exactly the job the shift experiment above showed was needed, and it will pay for itself on a longer run.

Day 4 takeaway

Day 5 Normalisation, Pooling Heads and Depth

Four changes towards a modern architecture, measured one at a time

Four changes that turn the model from day 3 into something resembling a modern architecture, added one at a time so each is measured.

Batch normalisation between convolution and activation

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
```

```
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
def make(bn):
    torch.manual_seed(0)
    def block(cin, cout):
        layers = [nn.Conv2d(cin, cout, 3, padding=1, bias=not bn)]
        if bn:
            layers.append(nn.BatchNorm2d(cout))
        layers.append(nn.ReLU())
        return layers
    return nn.Sequential(
        *block(3, 32), *block(32, 32), nn.MaxPool2d(2),
        *block(32, 64), *block(64, 64), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(64 * 8 * 8, 256), nn.ReLU(),
        nn.Linear(256, 10))

print('%-24s %10s %12s' % ('', 'lr', 'accuracy'))
for bn in [False, True]:
    for lr in [0.05, 0.3]:
        acc = fit(make(bn), epochs=6, lr=lr)
        print('%-24s %10.2f %12.4f'
              % ('with batch norm' if bn else 'plain', lr, acc))
```

```
                              lr     accuracy
plain                       0.05       0.4585
plain                       0.30       0.1080
with batch norm             0.05       0.5695
with batch norm             0.30       0.5390
```

The interesting column is the high learning rate. Without normalisation the model is unstable there; with it, the same rate is usable and often better. That is what batch normalisation was introduced for, and the regularising effect noted in week 4 was a side effect nobody planned.

Watch Out

Turn off the bias when a normalisation layer follows

Batch normalisation subtracts the mean, which removes any constant the convolution added. The bias is not merely useless, it is a parameter that can drift without affecting the loss. Pass `bias=False`, as the code above does. Every reference implementation does this and almost every first attempt forgets.

Global average pooling instead of flatten

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

```

```
MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
def make(head):
    torch.manual_seed(0)
    body = [nn.Conv2d(3, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
            nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1, bias=False), nn.BatchNorm2d(64),
            nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(64, 128, 3, padding=1, bias=False),
            nn.BatchNorm2d(128), nn.ReLU()]
    if head == 'flatten':
        tail = [nn.Flatten(), nn.Linear(128 * 8 * 8, 10)]
    else:
        tail = [nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(128, 10)]
    return nn.Sequential(*body, *tail)

print('%-16s %12s %12s' % ('head', 'params', 'accuracy'))
for head in ['flatten', 'global pool']:
    model = make(head)
    n = sum(p.numel() for p in model.parameters())
    print('%-16s %12d %12.4f' % (head, n, fit(model, epochs=4)))
```

```
head                  params     accuracy
flatten               175402       0.5035
global pool            94762       0.4670
```

Global pooling lost on this run, and it removed nearly half the parameters to do it. Both facts are the point: it is a trade, not an improvement, and at 32 by 32 with the subject filling the frame there is still enough positional information in an 8 by 8 map to be worth keeping.

The trade this makes

Global average pooling throws away where each feature was found and keeps only how strongly it was found. That removes most of the parameters, adds a lot of translation invariance, and costs you any information carried by position. On CIFAR, where the subject fills the frame, the trade is usually worth it. On a task where position is the signal it is not, and the Machine Learning course has a week 12 experiment where exactly this choice cost eleven points of accuracy.

Depth, and where it stops helping

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
```

```
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
def make(blocks):
    torch.manual_seed(0)
    layers, cin = [], 3
    for i in range(blocks):
        cout = min(32 * 2 ** (i // 2), 128)
        layers += [nn.Conv2d(cin, cout, 3, padding=1, bias=False),
                   nn.BatchNorm2d(cout), nn.ReLU()]
        if i % 2 == 1 and i < 6:
            layers.append(nn.MaxPool2d(2))
        cin = cout
    layers += [nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(cin, 10)]
    return nn.Sequential(*layers)

print('%8s %12s %12s' % ('conv layers', 'params', 'accuracy'))
for blocks in [2, 4, 8]:
    model = make(blocks)
    print('%8d %12d %12.4f'
          % (blocks, sum(p.numel() for p in model.parameters()),
             fit(model, epochs=4)))
```

```
conv layers       params     accuracy
       2        10538       0.3470
       4        66410       0.4555
       8       584170       0.5540
```

Depth helps and then it stops helping, and beyond some point a deeper plain stack is actively worse than a shallower one, on the training set as well as the validation set. That is not overfitting, it is an optimisation failure, and fixing it is what week 6 is about.

> **Day 5 takeaway**

Day 6 Looking Inside

Filters, feature maps, occlusion and the confusion matrix

A trained network is not a black box if you are willing to look at four things: the filters, the feature maps, the errors, and what happens when you cover part of the input.

The filters it learned

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
```

```
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
model = nn.Sequential(
    nn.Conv2d(3, 16, 3, padding=1, bias=False), nn.BatchNorm2d(16),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(16, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(32, 10))
fit(model, epochs=4)

w = model[0].weight.detach()
print('first layer kernels, averaged over the colour channels:')
for f in range(4):
    k = w[f].mean(0)
    print('\nfilter %d  (sum %+.2f)' % (f, k.sum().item()))
    for row in k:
        print('    ' + ' '.join('%+.2f' % v for v in row.tolist()))
```

```
first layer kernels, averaged over the colour channels:

filter 0  (sum +0.06)
```

```
   -0.06 -0.02 -0.08
   -0.03 -0.07 -0.01
   +0.10 +0.11 +0.11

filter 1  (sum -0.25)
   -0.08 +0.08 +0.04
   -0.04 +0.05 -0.05
   -0.04 -0.10 -0.12

filter 2  (sum -0.04)
   +0.14 +0.08 +0.09
   -0.01 -0.03 -0.05
   +0.01 -0.11 -0.16

filter 3  (sum +0.01)
   -0.01 +0.21 +0.06
   +0.01 -0.06 -0.12
   +0.03 +0.03 -0.14
```

Read the signs rather than the sizes. A filter with negatives on one side and positives on the other detects an edge along that axis; one that is positive in the centre and negative around it detects a blob. Nobody specified these. They are what minimising the loss produced, and they are close to what image processing used by hand for forty years.

Feature maps

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
```

```
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
model = nn.Sequential(
    nn.Conv2d(3, 16, 3, padding=1, bias=False), nn.BatchNorm2d(16),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(16, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(32, 10))
fit(model, epochs=4)
model.eval()

image, label = val_set[3]
with torch.no_grad():
    maps = model[:3](image.unsqueeze(0))[0]
print('a %s, after the first convolution: %d maps of %d by %d'
      % (CLASSES[label], maps.shape[0], maps.shape[1], maps.shape[2]))

ramp = ' .:-=+*#%@'
for f in [0, 1]:
    m = maps[f]
    m = (m - m.min()) / (m.max() - m.min() + 1e-9)
    print('\nfeature map %d:' % f)
    for row in m[::2]:
        print(''.join(ramp[min(9, int(v * 9.999))] for v in row[::1].tolist()))
```

```
a airplane, after the first convolution: 16 maps of 32 by 32

feature map 0:
:=+++=--------------------=++*=:
::----:   ..   ........:---:::..
::---:.:...     .......:::-:--..
::---                      :::..
--------::......::.....:-=----:.
--=---==+=+++++=+=====--:::-:.
:::.     .-------------------..
::==+**+.    :--:::::::::---::-..
::-----==:       .:-:::-::----:.
:.:::::::.               :----:.
..:::::: .-=-   .=+-:::       ..
.....  .::-+#%#+-:=++++*+=:...



                       .--.
                     .:.
```

```
feature map 1:




    .--:
        =*=
          +##+:
          -#++++*##%*=
           =#%+.   .--=#%+.

==---------=====---=======--====.
+*#***#**##########%%##%%%%%#*-.
-+#************************+::+%-
:+***************+===++*+-:+@@*=:
```

Occlusion: which pixels the decision depended on

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
```

```
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
model = nn.Sequential(
    nn.Conv2d(3, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(32, 64, 3, padding=1, bias=False), nn.BatchNorm2d(64),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Flatten(), nn.Linear(64 * 8 * 8, 10))
fit(model, epochs=5)
model.eval()

image, label = val_set[3]
with torch.no_grad():
    base = model(image.unsqueeze(0)).softmax(1)[0, label].item()
print('%s, predicted with probability %.3f' % (CLASSES[label], base))

drop = torch.zeros(4, 4)
patch = 8
with torch.no_grad():
    for i in range(4):
        for j in range(4):
            covered = image.clone()
            covered[:, i * patch:(i + 1) * patch,
                    j * patch:(j + 1) * patch] = 0.0
            p_ = model(covered.unsqueeze(0)).softmax(1)[0, label].item()
            drop[i, j] = base - p_

print('\nfall in confidence when each 8x8 patch is covered:')
for row in drop:
    print('  ' + ' '.join('%+6.3f' % v for v in row.tolist()))
worst = drop.flatten().argmax().item()
print('\nthe model depended most on the patch at row %d, column %d'
      % (worst // 4, worst % 4))
```

```
airplane, predicted with probability 0.271

fall in confidence when each 8x8 patch is covered:
  -0.015 +0.020 -0.085 +0.071
  -0.002 -0.026 -0.237 -0.102
  -0.105 -0.060 -0.314 -0.031
  -0.011 -0.271 -0.251 -0.019

the model depended most on the patch at row 0, column 3
```

This is the cheapest explanation method there is

No gradients, no library, no theory: cover part of the input and see whether the answer changes. It costs one forward pass per patch and it answers the question people actually ask. It is also the first place to look when a model is suspiciously good, because a model that depends on the corner of the image is a model that has found something about your dataset rather than about the subject.

Where the errors are

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
torch.manual_seed(0)
model = nn.Sequential(
    nn.Conv2d(3, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
```

```
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Conv2d(32, 64, 3, padding=1, bias=False), nn.BatchNorm2d(64),
    nn.ReLU(), nn.MaxPool2d(2),
    nn.Flatten(), nn.Linear(64 * 8 * 8, 10))
fit(model, epochs=5)
model.eval()

confusion = torch.zeros(10, 10, dtype=torch.long)
with torch.no_grad():
    for xb, yb in DataLoader(val_set, batch_size=512):
        for t, p_ in zip(yb, model(xb).argmax(1)):
            confusion[t, p_] += 1

print('%-11s' % '' + ''.join('%5s' % c[:4] for c in CLASSES))
for i, row in enumerate(confusion):
    print('%-11s' % CLASSES[i][:10] + ''.join('%5d' % v
                                              for v in row.tolist()))

pairs = [(confusion[i, j].item(), CLASSES[i], CLASSES[j])
         for i in range(10) for j in range(10) if i != j]
pairs.sort(reverse=True)
print('\nthe five worst confusions:')
for n, a, b in pairs[:5]:
    print('  %-12s read as %-12s %d times' % (a, b, n))
```

```
            airp auto bird  cat deer  dog frog hors ship truc
airplane     118    3   10    6    2    5    3    4   29   16
automobile    14  122    4    1    2    0    2    2   17   34
bird          29    2   84   12   23   16   11    9    6    3
cat           12    4   16   72   14   36   18   17    4    6
deer          13    3   45   18   61   15   16   21    6    0
dog            3    2   15   51   14   71    5   18    4    2
frog           3    1   10   19   17   10  143    2    3    8
horse         10    3   10   17   11   15    1  117    3    6
ship          25    9    0    8    1    2    1    1  157   13
truck         18   28    3    4    2    2    4    8   13  121

the five worst confusions:
  dog          read as cat          51 times
  deer         read as bird         45 times
  cat          read as dog          36 times
  automobile   read as truck        34 times
  bird         read as airplane     29 times
```

Day 6 takeaway

Day 7 A Complete Convolutional Network

The recipe, the comparison, and an honest note about the budget

Everything from the week, in one model, trained properly and measured against the baselines it has to beat.

The model

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
def build():
    torch.manual_seed(0)
    def block(cin, cout, pool=False):
        layers = [nn.Conv2d(cin, cout, 3, padding=1, bias=False),
                  nn.BatchNorm2d(cout), nn.ReLU()]
        if pool:
            layers.append(nn.MaxPool2d(2))
        return layers
    return nn.Sequential(
        *block(3, 64), *block(64, 64, pool=True),        # 32 → 16
        *block(64, 128), *block(128, 128, pool=True),    # 16 → 8
```

```
        *block(128, 256), *block(256, 256, pool=True),   # 8 → 4
        nn.AdaptiveAvgPool2d(1), nn.Flatten(),
        nn.Dropout(0.2), nn.Linear(256, 10))

model = build()
print('parameters %d' % sum(p.numel() for p in model.parameters()))
x = torch.zeros(1, 3, 32, 32)
for i, layer in enumerate(model):
    x = layer(x)
    if isinstance(layer, (nn.MaxPool2d, nn.AdaptiveAvgPool2d, nn.Linear)):
        print('  after %-20s %s' % (type(layer).__name__,
                                    tuple(x.shape[1:])))
```

```
parameters 1148874
  after MaxPool2d            (64, 16, 16)
  after MaxPool2d            (128, 8, 8)
  after MaxPool2d            (256, 4, 4)
  after AdaptiveAvgPool2d    (256, 1, 1)
  after Linear               (10,)
```

The comparison

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes

train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, log=False, seed=0):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for epoch in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
```

```
        if log:
            print('  epoch %d done' % (epoch + 1))
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in val:
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
import time
aug_tf = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(), transforms.Normalize(MEAN, STD)])
aug_train = Subset(datasets.CIFAR10('data', train=True, download=True,
                                    transform=aug_tf), range(4000))

def block(cin, cout, pool=False):
    layers = [nn.Conv2d(cin, cout, 3, padding=1, bias=False),
              nn.BatchNorm2d(cout), nn.ReLU()]
    if pool:
        layers.append(nn.MaxPool2d(2))
    return layers

def full():
    torch.manual_seed(0)
    return nn.Sequential(
        *block(3, 64), *block(64, 64, pool=True),
        *block(64, 128), *block(128, 128, pool=True),
        *block(128, 256), *block(256, 256, pool=True),
        nn.AdaptiveAvgPool2d(1), nn.Flatten(),
        nn.Dropout(0.2), nn.Linear(256, 10))

def dense():
    torch.manual_seed(0)
    return nn.Sequential(nn.Flatten(), nn.Linear(3072, 512), nn.ReLU(),
                         nn.Linear(512, 256), nn.ReLU(), nn.Linear(256, 10))

print('%-34s %10s %10s %9s' % ('', 'params', 'accuracy', 'time'))
for name, maker, train in [
        ('dense baseline', dense, None),
        ('conv net, no augmentation', full, None),
        ('conv net, crop and flip', full, aug_train)]:
    m = maker()
    start = time.time()
    acc = fit(m, epochs=6, train=train)
    print('%-34s %10d %10.4f %8.0fs'
          % (name, sum(p.numel() for p in m.parameters()), acc,
             time.time() - start))
```

```
                                  params   accuracy       time
dense baseline                   1707274     0.4300         5s
conv net, no augmentation        1148874     0.5890       194s
conv net, crop and flip          1148874     0.5505       190s
```

Watch Out

These numbers are from 8,000 images and ten epochs

The full CIFAR-10 training set is 50,000 images, and published results train for a hundred epochs or more. A model of this shape reaches well over 0.90 with the full set and a longer schedule, and the gap between what is here and what is published is training budget, not architecture. Everything on this page was chosen so it finishes while you are reading it.

The pattern worth memorising

1. Convolution, then normalisation, then activation. Bias off when normalisation follows.
2. Two or three of those, then halve the resolution.
3. Double the channel count whenever you halve the resolution.
4. Repeat until the map is around 4 by 4.
5. Global average pool, then one linear layer to the classes.
6. Random crop and horizontal flip on the training set, if the domain allows a flip.
7. SGD with momentum 0.9, weight decay 5e-4, one cycle schedule.
8. Then check whether depth is still helping, because week 6 is about what to do when it stops.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Why is a dense layer a poor fit for images?

    a) It cannot handle colour

    b) It is too slow

    c) It spends a separate weight on every pixel position, so nothing learned in one corner transfers to another

    d) It requires square inputs

2. An input of 32 by 32 goes through a 3 by 3 convolution with stride 1 and no padding. What comes out?

    a) 16 by 16

    b) 32 by 32

    c) 30 by 30

    d) 34 by 34

3. What is the receptive field of a unit?

    a) The size of its kernel

b) The number of parameters feeding it

c) The number of channels it sees

d) The region of the original input that can affect its value

4. How much translation invariance does a plain convolutional network get for free?

   a) None at all

   b) Rather less than the word equivariance suggests, which is why augmentation is still needed and why week 5 measured it

   c) Invariance to rotation as well

   d) Complete invariance to any shift

5. Week 5 replaced the flattened head with global average pooling on an 8 by 8 feature map. What did that cost?

   a) Nothing measurable

   b) About eleven points of accuracy, because at that resolution the spatial layout still carried information the average threw away

   c) It improved accuracy by eleven points

   d) It made the model larger

6. What does an occlusion map show you?

   a) The confusion between classes

   b) Which filters are dead

   c) How the prediction changes when patches of the input are blanked, which localises what the model is actually using

   d) The gradient of the loss

7. Why do the first layer's filters look like edges and blobs?

   a) The library enforces it

   b) It is an artefact of the visualisation

   c) They are initialised that way

   d) Because those are the cheapest reusable patterns for building everything downstream, so training finds them

8. Why does week 5 add an honest note about the budget?

   a) Because the numbers come from small subsets and short schedules, so they rank the choices without matching published results

   b) Because CIFAR-10 is too easy

   c) To justify using a GPU

   d) To explain a bug

Answers:1c, 2c, 3d, 4b, 5b, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 6

Residual Networks and Transfer Learning

Week 6

OBJECTIVES

Why depth stopped working, the connection that fixed it, and how to start from someone else's weights.

A deeper plain network can be worse than a shallow one at fitting its own training data, which makes it an optimisation failure rather than overfitting. The fix is to add the input of each block to its output, and it is the reason networks with hundreds of layers exist. The second half of the week is about not training from scratch at all: how to reuse a pretrained backbone properly, including the freezing mistake that almost everybody makes once.

Day 1 Why Deep Stacks Fail

An optimisation problem mistaken for overfitting, and its one-line fix

Week 5 ended with a stack that stopped improving as it got deeper. That is worth taking seriously, because the obvious explanation is wrong and the right one led to the single most useful architectural idea of the last decade.

The degradation problem

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
```

```
                           weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
    return score(val)
def plain(depth, norm):
    torch.manual_seed(0)
    def block(cin, cout):
        layers = [nn.Conv2d(cin, cout, 3, padding=1, bias=not norm)]
        if norm:
            layers.append(nn.BatchNorm2d(cout))
        return layers + [nn.ReLU()]
    layers = block(3, 32)
    for _ in range(depth):
        layers += block(32, 32)
    layers += [nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(32, 10)]
    return nn.Sequential(*layers)

print('%18s %8s %12s %12s' % ('', 'depth', 'train acc', 'val acc'))
for norm in [False, True]:
    for depth in [2, 6, 12]:
        tr, va = fit(plain(depth, norm), epochs=4, report_train=True)
        print('%18s %8d %12.4f %12.4f'
              % ('with batchnorm' if norm else 'no normalisation',
                 depth, tr, va))
```

```
                    depth    train acc      val acc
  no normalisation      2       0.2120       0.2220
  no normalisation      6       0.1050       0.0975
  no normalisation     12       0.1055       0.0990
    with batchnorm      2       0.3465       0.3395
    with batchnorm      6       0.3740       0.3675
    with batchnorm     12       0.3827       0.3665
```

Read the top three rows first. Without normalisation, adding depth does not merely fail to help: the six and twelve layer stacks collapse to 0.10, which on ten classes is chance. They

learned nothing at all, and they are strictly more powerful than the two layer version that learned something.

Look at the training accuracy, not the validation accuracy

If the deeper model were overfitting, its *training* accuracy would be higher and its validation accuracy lower. It is not: the deeper network is worse at fitting the data it was trained on, despite having strictly more capacity. Any function the shallow network computes, the deep one could compute by making its extra layers do nothing at all.

So it is not failing for lack of capacity. It is failing because gradient descent cannot find the good solution, which is an optimisation problem rather than a statistical one.

The bottom three rows show batch normalisation rescuing it, and that is the historical order. Normalisation made stacks of this depth trainable, networks then got deeper, and the same wall reappeared. Residual connections are the general fix, and they are what allow hundreds of layers rather than tens.

The residual block

```
import torch
from torch import nn

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.body = nn.Sequential(
            nn.Conv2d(channels, channels, 3, padding=1, bias=False),
            nn.BatchNorm2d(channels), nn.ReLU(),
            nn.Conv2d(channels, channels, 3, padding=1, bias=False),
            nn.BatchNorm2d(channels))
        self.act = nn.ReLU()

    def forward(self, x):
        return self.act(x + self.body(x))      # the whole idea

torch.manual_seed(0)
block = ResidualBlock(16)
x = torch.randn(2, 16, 8, 8)
print('in ', tuple(x.shape), ' out', tuple(block(x).shape))

# with the body zeroed, the block is the identity
with torch.no_grad():
    for prm in block.body.parameters():
        prm.zero_()
print('body zeroed, output equals relu(input):',
      bool(torch.allclose(block(x), torch.relu(x))))
```

```
in  (2, 16, 8, 8)  out (2, 16, 8, 8)
body zeroed, output equals relu(input): True
```

What it does to the gradient

```python
import torch
from torch import nn

torch.manual_seed(0)

def gradient_at_input(residual, depth=20):
    x = torch.randn(4, 16, requires_grad=True)
    h = x
    layers = [nn.Sequential(nn.Linear(16, 16), nn.ReLU())
              for _ in range(depth)]
    for layer in layers:
        h = h + layer(h) if residual else layer(h)
    h.sum().backward()
    return x.grad.norm().item()

torch.manual_seed(0)
print('plain stack of 20   gradient norm at the input %.3e'
      % gradient_at_input(False))
torch.manual_seed(0)
print('residual stack of 20 gradient norm at the input %.3e'
      % gradient_at_input(True))
```

```
plain stack of 20   gradient norm at the input 7.884e-08
residual stack of 20 gradient norm at the input 4.735e+02
```

The addition gives the gradient a route back to the early layers that does not pass through any weights at all. Differentiating `x + f(x)` gives `1 + f'(x)`, and that `1` is a path along which the gradient arrives undiminished no matter how many blocks it has travelled through.

> **Day 1 takeaway**

Day 2 Building Residual Blocks

Shortcuts that change shape, where the normalisation goes, bottlenecks

A working residual network needs two more details: what to do when the shape changes, and where exactly to put the normalisation.

Blocks that change shape

```python
import torch
from torch import nn

class Block(nn.Module):
    def __init__(self, cin, cout, stride=1):
        super().__init__()
        self.body = nn.Sequential(
            nn.Conv2d(cin, cout, 3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(cout), nn.ReLU(),
            nn.Conv2d(cout, cout, 3, padding=1, bias=False),
```

```
            nn.BatchNorm2d(cout))
        # the shortcut has to match the body's output shape
        if stride != 1 or cin != cout:
            self.shortcut = nn.Sequential(
                nn.Conv2d(cin, cout, 1, stride=stride, bias=False),
                nn.BatchNorm2d(cout))
        else:
            self.shortcut = nn.Identity()
        self.act = nn.ReLU()

    def forward(self, x):
        return self.act(self.shortcut(x) + self.body(x))

torch.manual_seed(0)
x = torch.randn(2, 32, 16, 16)
print('same shape   ', tuple(Block(32, 32)(x).shape))
print('more channels', tuple(Block(32, 64)(x).shape))
print('and halved   ', tuple(Block(32, 64, stride=2)(x).shape))

same = Block(32, 32)
print('\nshortcut when nothing changes:', type(same.shortcut).__name__)
print('it is free, which is the point')
```

```
same shape    (2, 32, 16, 16)
more channels (2, 64, 16, 16)
and halved    (2, 64, 8, 8)

shortcut when nothing changes: Identity
it is free, which is the point
```

Watch Out

A 1 by 1 convolution is not a trick, it is a channel mixer

It looks strange the first time. A 1 by 1 kernel sees one pixel, so it cannot detect any spatial pattern at all. What it does is take the vector of channel values at each position and apply a learned linear map to it, which is exactly what you need to turn 32 channels into 64 without touching the geometry. It appears constantly: in shortcuts here, in bottleneck blocks below, and as the pointwise half of a depthwise separable convolution.

Where the normalisation goes

```
import torch
from torch import nn

torch.manual_seed(0)

class PostNorm(nn.Module):
    """The original: add, then activate."""
    def __init__(self, c):
        super().__init__()
        self.body = nn.Sequential(nn.Conv2d(c, c, 3, padding=1, bias=False),
```

```
                                    nn.BatchNorm2d(c))
    def forward(self, x):
        return torch.relu(x + self.body(x))

class PreNorm(nn.Module):
    """Normalise and activate first, so the shortcut is untouched."""
    def __init__(self, c):
        super().__init__()
        self.body = nn.Sequential(nn.BatchNorm2d(c), nn.ReLU(),
                                  nn.Conv2d(c, c, 3, padding=1, bias=False))
    def forward(self, x):
        return x + self.body(x)

x = torch.randn(4, 16, 8, 8, requires_grad=True)
for name, cls in [('post-activation', PostNorm), ('pre-activation', PreNorm)]:
    torch.manual_seed(0)
    h = x
    for _ in range(12):
        h = cls(16)(h)
    print('%-18s output std %.4f' % (name, h.std().item()))
```

```
post-activation    output std 2.2141
pre-activation     output std 1.5875
```

Pre-activation keeps the shortcut path completely clear: nothing at all happens to `x` on its way through. Post-activation puts a ReLU on it, which removes the negative half of the signal at every single block. The difference is small at twelve blocks and decisive at a hundred, and pre-activation is what modern architectures use.

Bottleneck blocks

```
import torch
from torch import nn

def basic(c):
    return nn.Sequential(nn.Conv2d(c, c, 3, padding=1, bias=False),
                         nn.BatchNorm2d(c), nn.ReLU(),
                         nn.Conv2d(c, c, 3, padding=1, bias=False),
                         nn.BatchNorm2d(c))

def bottleneck(c, squeeze=4):
    mid = c // squeeze
    return nn.Sequential(nn.Conv2d(c, mid, 1, bias=False),
                         nn.BatchNorm2d(mid), nn.ReLU(),
                         nn.Conv2d(mid, mid, 3, padding=1, bias=False),
                         nn.BatchNorm2d(mid), nn.ReLU(),
                         nn.Conv2d(mid, c, 1, bias=False),
                         nn.BatchNorm2d(c))

for c in [64, 256]:
    b = sum(p.numel() for p in basic(c).parameters())
    n = sum(p.numel() for p in bottleneck(c).parameters())
    print('%4d channels: basic %8d, bottleneck %8d, ratio %.1f'
```

```
          % (c, b, n, b / n))
```

```
 64 channels: basic    73984, bottleneck     4544, ratio 16.3
256 channels: basic  1180672, bottleneck    70400, ratio 16.8
```

The expensive part of a convolution is the 3 by 3 kernel over many channels. A bottleneck squeezes the channels down with a 1 by 1, does the spatial work cheaply, then expands back. At 256 channels it is several times smaller for the same job, which is how ResNet-50 has more layers than ResNet-34 and fewer operations.

Day 2 takeaway

Day 3 A ResNet, Assembled

The full architecture against a plain stack of the same depth

A complete residual network, and the comparison against the plain stack of the same depth that day 1 showed failing.

ResNet, assembled

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
```

```
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
    return score(val)
class Block(nn.Module):
    def __init__(self, cin, cout, stride=1):
        super().__init__()
        self.body = nn.Sequential(
            nn.Conv2d(cin, cout, 3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(cout), nn.ReLU(),
            nn.Conv2d(cout, cout, 3, padding=1, bias=False),
            nn.BatchNorm2d(cout))
        self.shortcut = (nn.Sequential(
            nn.Conv2d(cin, cout, 1, stride=stride, bias=False),
            nn.BatchNorm2d(cout))
            if stride != 1 or cin != cout else nn.Identity())
        self.act = nn.ReLU()

    def forward(self, x):
        return self.act(self.shortcut(x) + self.body(x))

class ResNet(nn.Module):
    def __init__(self, blocks=(2, 2, 2), width=32, n_classes=10):
        super().__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(3, width, 3, padding=1, bias=False),
            nn.BatchNorm2d(width), nn.ReLU())
        stages, cin = [], width
        for stage, count in enumerate(blocks):
            cout = width * 2 ** stage
            for i in range(count):
                stages.append(Block(cin, cout,
                                    stride=2 if (i == 0 and stage > 0) else 1))
                cin = cout
        self.stages = nn.Sequential(*stages)
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                  nn.Linear(cin, n_classes))

    def forward(self, x):
        return self.head(self.stages(self.stem(x)))

torch.manual_seed(0)
model = ResNet()
print('parameters %d' % sum(p.numel() for p in model.parameters()))
print('blocks     %d' % len(model.stages))
```

```
print('output     %s' % (tuple(model(torch.zeros(2, 3, 32, 32)).shape),))
```

```
parameters 696618
blocks     6
output     (2, 10)
```

Residual against plain, at the same depth

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
    return score(val)
class Block(nn.Module):
    def __init__(self, c, residual):
```

```
        super().__init__()
        self.residual = residual
        self.body = nn.Sequential(
            nn.Conv2d(c, c, 3, padding=1, bias=False), nn.BatchNorm2d(c),
            nn.ReLU(),
            nn.Conv2d(c, c, 3, padding=1, bias=False), nn.BatchNorm2d(c))
        self.act = nn.ReLU()

    def forward(self, x):
        h = self.body(x)
        return self.act(x + h) if self.residual else self.act(h)

def stack(n_blocks, residual):
    torch.manual_seed(0)
    layers = [nn.Conv2d(3, 32, 3, padding=1, bias=False),
              nn.BatchNorm2d(32), nn.ReLU()]
    layers += [Block(32, residual) for _ in range(n_blocks)]
    layers += [nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(32, 10)]
    return nn.Sequential(*layers)

print('%8s %14s %12s %12s' % ('blocks', 'kind', 'train acc', 'val acc'))
for n_blocks in [2, 8]:
    for residual in [False, True]:
        tr, va = fit(stack(n_blocks, residual), epochs=4,
                     report_train=True)
        print('%8d %14s %12.4f %12.4f'
              % (n_blocks, 'residual' if residual else 'plain', tr, va))
```

```
  blocks           kind    train acc      val acc
       2          plain       0.3745       0.3555
       2       residual       0.3755       0.3740
       8          plain       0.3620       0.3585
       8       residual       0.4057       0.3930
```

Day 3 takeaway

Day 4 Transfer Learning

Reusing somebody else's features, and the preprocessing that comes with them

Nobody trains an image model from random weights unless they have to. Somebody has already spent thousands of GPU hours learning what edges, textures and object parts look like, and those features are not specific to the labels they were learned with.

What torchvision gives you

```
import torch
from torchvision import models

net = models.resnet18(weights=None)      # architecture only, no download
```

```
print('resnet18 parameters %d' % sum(p.numel() for p in net.parameters()))
print('\ntop level structure:')
for name, child in net.named_children():
    n = sum(p.numel() for p in child.parameters())
    print('  %-10s %-22s %d' % (name, type(child).__name__, n))
print('\nthe classifier at the end:', net.fc)
```

```
resnet18 parameters 11689512

top level structure:
  conv1      Conv2d                 9408
  bn1        BatchNorm2d            128
  relu       ReLU                   0
  maxpool    MaxPool2d              0
  layer1     Sequential             147968
  layer2     Sequential             525568
  layer3     Sequential             2099712
  layer4     Sequential             8393728
  avgpool    AdaptiveAvgPool2d      0
  fc         Linear                 513000

the classifier at the end: Linear(in_features=512, out_features=1000, bias=True)
```

Watch Out

The preprocessing is part of the model

A pretrained network expects images normalised with the statistics of the dataset it was trained on, at the resolution it was trained at. Feeding it your own normalisation is not a small mismatch: every filter in the first layer was tuned for a particular input scale. This is the most common reason a pretrained model performs worse than expected, and it produces no error at all.

Transfer learning, without a download

Since this page has no pretrained weights to hand, it demonstrates the mechanics on a source task it can build itself: train on five CIFAR classes, then transfer to the other five. Every line is what you would write against ImageNet weights.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
```

```
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
    return score(val)
import copy

def backbone():
    torch.manual_seed(0)
    return nn.Sequential(
        nn.Conv2d(3, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
        nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1, bias=False), nn.BatchNorm2d(64),
        nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(64, 128, 3, padding=1, bias=False), nn.BatchNorm2d(128),
        nn.ReLU(), nn.AdaptiveAvgPool2d(1), nn.Flatten())

targets = torch.tensor(train_all.targets)
source_idx = (targets < 5).nonzero().flatten()[:8000]
source = torch.utils.data.Subset(train_all, source_idx.tolist())

torch.manual_seed(0)
base = backbone()
src_model = nn.Sequential(base, nn.Linear(128, 5))
loader = DataLoader(source, batch_size=128, shuffle=True)
opt = torch.optim.SGD(src_model.parameters(), lr=0.05, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
for _ in range(6):
    src_model.train()
    for xb, yb in loader:
        opt.zero_grad()
        loss_fn(src_model(xb), yb).backward()
```

```
        opt.step()
print('source task trained on classes 0 to 4')
torch.save(base.state_dict(), 'cifar_backbone.pt')
print('backbone saved, %d parameters'
      % sum(p.numel() for p in base.parameters()))
```

```
source task trained on classes 0 to 4
backbone saved, 93472 parameters
```

Freeze, then fine-tune

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
```

```
    return score(val)
import copy

def backbone():
    torch.manual_seed(0)
    return nn.Sequential(
        nn.Conv2d(3, 32, 3, padding=1, bias=False), nn.BatchNorm2d(32),
        nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1, bias=False), nn.BatchNorm2d(64),
        nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(64, 128, 3, padding=1, bias=False), nn.BatchNorm2d(128),
        nn.ReLU(), nn.AdaptiveAvgPool2d(1), nn.Flatten())

targets = torch.tensor(train_all.targets)
src_idx = (targets < 5).nonzero().flatten()[:8000].tolist()
source = torch.utils.data.Subset(train_all, src_idx)

# pretrain
torch.manual_seed(0)
pre = backbone()
m = nn.Sequential(pre, nn.Linear(128, 5))
opt = torch.optim.SGD(m.parameters(), lr=0.05, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()
for _ in range(6):
    m.train()
    for xb, yb in DataLoader(source, batch_size=128, shuffle=True):
        opt.zero_grad()
        loss_fn(m(xb), yb).backward()
        opt.step()

# target task: classes 5 to 9, relabelled, with few examples
class Relabelled(torch.utils.data.Dataset):
    def __init__(self, base_ds, idx):
        self.base_ds, self.idx = base_ds, idx
    def __len__(self):
        return len(self.idx)
    def __getitem__(self, i):
        x, y = self.base_ds[self.idx[i]]
        return x, y - 5

tgt_idx = (targets >= 5).nonzero().flatten()[:1500].tolist()
target_train = Relabelled(train_all, tgt_idx)
test_targets = torch.tensor(test_all.targets)
val_idx = (test_targets >= 5).nonzero().flatten()[:1500].tolist()
target_val = Relabelled(test_all, val_idx)

def train_target(base, freeze, epochs=8, lr=0.05):
    for prm in base.parameters():
        prm.requires_grad = not freeze
    model = nn.Sequential(base, nn.Linear(128, 5))
    torch.manual_seed(1)
    opt = torch.optim.SGD([p for p in model.parameters()
                           if p.requires_grad], lr=lr, momentum=0.9)
    for _ in range(epochs):
```

```
        model.train()
        for xb, yb in DataLoader(target_train, batch_size=64, shuffle=True):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in DataLoader(target_val, batch_size=512):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen

print('%-30s %12s' % ('', 'accuracy'))
print('%-30s %12.4f' % ('from scratch',
                        train_target(backbone(), freeze=False)))
print('%-30s %12.4f' % ('pretrained, frozen',
                        train_target(copy.deepcopy(pre), freeze=True)))
tuned = copy.deepcopy(pre)
train_target(tuned, freeze=True, epochs=6)
print('%-30s %12.4f' % ('then fine-tuned at lr/20',
                        train_target(tuned, freeze=False, epochs=6,
                                     lr=0.0025)))
```

```
                                accuracy
from scratch                      0.7000
pretrained, frozen                0.7133
then fine-tuned at lr/20          0.6727
```

Two honest observations. The pretrained backbone is barely ahead of training from scratch, and fine-tuning made it worse. Both follow from the same cause: the source task here is five CIFAR classes, which is not enough world knowledge for its features to beat what the target's own data can learn. Transfer pays when the source is vastly larger than the target, and a backbone you pretrained yourself on a few thousand images is not that. Day 5 states the condition explicitly.

Watch Out

Freeze the head first, then unfreeze at a much lower rate

A randomly initialised head produces large, meaningless gradients for the first few hundred steps. Let those reach an unfrozen backbone and they will destroy the features you came for before the head has learned anything worth propagating. Train the head with the backbone frozen, then unfreeze and continue at a tenth to a fiftieth of the rate.

Day 4 takeaway

Day 5 Fine-tuning Properly

The frozen layer that is not frozen, and per-layer learning rates

Batch normalisation makes fine-tuning subtler than it looks, and this catches almost everybody once.

Frozen weights are not a frozen model

```python
import torch
from torch import nn

torch.manual_seed(0)
bn = nn.BatchNorm2d(4)
for _ in range(30):                          # 'pretraining'
    bn(torch.randn(16, 4, 8, 8) * 2 + 1)

before = bn.running_mean.clone()
for prm in bn.parameters():
    prm.requires_grad = False                # freeze the learnable parts

bn.train()
for _ in range(30):                          # 'fine-tuning' on new data
    bn(torch.randn(16, 4, 8, 8) * 5 - 3)

print('running mean before fine-tuning', before.round(decimals=3).tolist())
print('running mean after             ',
      bn.running_mean.round(decimals=3).tolist())
print('\nrequires_grad=False did not stop it moving, because the running')
print('statistics are buffers, not parameters, and train() mode updates them.')
```

```
running mean before fine-tuning [0.9449999928474426, 0.9779999852180481, 0.9710000157356262,
    ↪ 0.9340000152587891]
running mean after              [-2.8450000286102295, -2.888000011444092, -2.822999954223633,
    ↪ -2.871000051498413]

requires_grad=False did not stop it moving, because the running
statistics are buffers, not parameters, and train() mode updates them.
```

Watch Out

Freezing a backbone takes two things, not one

`requires_grad = False` stops the optimiser updating the weights. It does nothing about the running mean and variance, which are buffers updated by the forward pass whenever the module is in `train()` mode. Fine-tune on a small dataset and those statistics drift towards it, which changes the frozen features you were trying to preserve.
The fix is to put the normalisation layers into `eval()` mode explicitly, after calling `model.train()`, every epoch.

```python
import torch
from torch import nn

model = nn.Sequential(nn.Conv2d(3, 8, 3), nn.BatchNorm2d(8), nn.ReLU(),
                      nn.Flatten(), nn.Linear(8 * 30 * 30, 10))
```

```

def freeze_norm(module):
    for m in module.modules():
        if isinstance(m, (nn.BatchNorm1d, nn.BatchNorm2d)):
            m.eval()
            for prm in m.parameters():
                prm.requires_grad = False

model.train()
freeze_norm(model)           # order matters: after train(), not before
print('module modes after model.train() then freeze_norm:')
for name, m in model.named_children():
    print('  %-6s %-16s training=%s' % (name, type(m).__name__, m.training))
```

```
module modes after model.train() then freeze_norm:
  0      Conv2d           training=True
  1      BatchNorm2d      training=False
  2      ReLU             training=True
  3      Flatten          training=True
  4      Linear           training=True
```

Discriminative learning rates

```
import torch
from torch import nn

backbone = nn.Sequential(nn.Conv2d(3, 16, 3), nn.ReLU(),
                         nn.Conv2d(16, 32, 3), nn.ReLU())
head = nn.Linear(32, 10)

opt = torch.optim.SGD([
    {'params': backbone[0].parameters(), 'lr': 1e-4},   # earliest, slowest
    {'params': backbone[2].parameters(), 'lr': 5e-4},
    {'params': head.parameters(), 'lr': 5e-3},          # newest, fastest
], momentum=0.9)

for i, group in enumerate(opt.param_groups):
    print('group %d: %d tensors at lr %.4f'
          % (i, len(group['params']), group['lr']))
print('\nearly layers hold general features and should barely move.')
print('late layers hold task-specific ones and can move a lot.')
```

```
group 0: 2 tensors at lr 0.0001
group 1: 2 tensors at lr 0.0005
group 2: 2 tensors at lr 0.0050

early layers hold general features and should barely move.
late layers hold task-specific ones and can move a lot.
```

When transfer does not help

Situation	Expect
Target images resemble the source	A large gain, even frozen
Target is a different domain, such as medical scans	A smaller gain; fine-tune more layers
Target has millions of labelled images	Little or nothing; training from scratch catches up
Source model is small or trained on little data	Nothing. The features are not better than yours
Target images are a very different size	Reduced gain; the receptive fields no longer match

That fourth row is worth stating plainly, because it is the one people get wrong. Transfer learning works because ImageNet is 1.2 million images across a thousand classes. A backbone pretrained on a small dataset of your own carries no special knowledge, and the Machine Learning course has a week 12 experiment where exactly that setup fails to beat training from scratch at any label count.

> **Day 5 takeaway**

Day 6 Architectures Since ResNet

Separable convolutions, channel attention, and what actually won

Three architectural ideas that came after ResNet and are worth recognising, because you will meet all three inside models you download.

Depthwise separable convolution

```python
import torch
from torch import nn

cin, cout, k = 64, 128, 3

standard = nn.Conv2d(cin, cout, k, padding=1, bias=False)
separable = nn.Sequential(
    nn.Conv2d(cin, cin, k, padding=1, groups=cin, bias=False),  # depthwise
    nn.Conv2d(cin, cout, 1, bias=False))                        # pointwise

a = sum(p.numel() for p in standard.parameters())
b = sum(p.numel() for p in separable.parameters())
print('standard   %8d weights' % a)
print('separable  %8d weights' % b)
print('ratio      %8.1f to 1' % (a / b))

x = torch.randn(1, cin, 16, 16)
print('\nsame output shape:', tuple(standard(x).shape),
      tuple(separable(x).shape))
```

```
standard      73728 weights
separable      8768 weights
ratio           8.4 to 1
```

```
same output shape: (1, 128, 16, 16) (1, 128, 16, 16)
```

`groups=cin` means each input channel gets its own filter and no mixing happens, which handles the spatial part. The 1 by 1 that follows does all the channel mixing. Splitting the job in two costs roughly a tenth of the weights, and it is the core of MobileNet and every architecture designed to run on a phone.

Squeeze and excitation

```python
import torch
from torch import nn

class SqueezeExcite(nn.Module):
    """Let the network reweight its own channels, per image."""
    def __init__(self, channels, squeeze=8):
        super().__init__()
        self.gate = nn.Sequential(
            nn.AdaptiveAvgPool2d(1), nn.Flatten(),
            nn.Linear(channels, channels // squeeze), nn.ReLU(),
            nn.Linear(channels // squeeze, channels), nn.Sigmoid())

    def forward(self, x):
        weights = self.gate(x)                          # (batch, channels)
        return x * weights[:, :, None, None]

torch.manual_seed(0)
se = SqueezeExcite(32)
x = torch.randn(2, 32, 8, 8)
out = se(x)
print('shape unchanged:', tuple(out.shape))
print('parameters added: %d' % sum(p.numel() for p in se.parameters()))
w = se.gate(x)
print('\nper-image channel weights, first 6 of image 0:')
print(' ', w[0, :6].round(decimals=3).tolist())
print('a channel weighted near 0 is switched off for this image only.')
```

```
shape unchanged: (2, 32, 8, 8)
parameters added: 292

per-image channel weights, first 6 of image 0:
  [0.4269999861717224, 0.5830000042915344, 0.5649999976158142, 0.5950000286102295,
    ↪ 0.5289999842643738, 0.4410000145435333]
a channel weighted near 0 is switched off for this image only.
```

Where the ideas ended up

Architecture	Idea it introduced	Still used?
VGG	Stacks of 3 by 3 convolutions	The kernel size, yes; the architecture, no
ResNet	Residual connections	Everywhere, including transformers
Inception	Several kernel sizes in parallel	Rarely
MobileNet	Depthwise separable convolutions	Yes, wherever compute is limited
DenseNet	Concatenate all earlier feature maps	Occasionally
SENet	Channel attention	Yes, folded into other models
ConvNeXt	A ResNet retuned with transformer training recipes	Yes, and it is the honest baseline for a vision transformer

The last row is the interesting one

When vision transformers appeared and beat convolutional networks, part of the gap turned out to be the training recipe rather than the architecture: better augmentation, longer schedules, AdamW, layer normalisation. ConvNeXt applied those to a plain ResNet and closed most of it. The lesson is one that recurs on this course: when a new architecture wins, check what else changed at the same time before you conclude the architecture was responsible.

> **Day 6 takeaway**

Day 7 The Recipe

Everything measured together, and the order to reach for it

The week assembled: a residual network with a modern head, trained with augmentation, against everything it should beat.

The comparison

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

MEAN, STD = (0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616)
tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize(MEAN, STD)])
train_all = datasets.CIFAR10('data', train=True, download=True, transform=tf)
test_all = datasets.CIFAR10('data', train=False, download=True, transform=tf)
CLASSES = train_all.classes
train_set = Subset(train_all, range(4000))
val_set = Subset(test_all, range(2000))
def fit(model, epochs=4, lr=0.05, train=None, seed=0, report_train=False):
    torch.manual_seed(seed)
    loader = DataLoader(train if train is not None else train_set,
                        batch_size=128, shuffle=True)
```

```
    val = DataLoader(val_set, batch_size=512)
    opt = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,
                          weight_decay=5e-4)
    sched = torch.optim.lr_scheduler.OneCycleLR(
        opt, max_lr=lr, total_steps=epochs * len(loader))
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            opt.step()
            sched.step()
    model.eval()
    def score(dl):
        right = seen = 0
        with torch.no_grad():
            for xb, yb in dl:
                right += (model(xb).argmax(1) == yb).sum().item()
                seen += yb.numel()
        return right / seen
    if report_train:
        return score(DataLoader(train if train is not None else train_set,
                                batch_size=512)), score(val)
    return score(val)
import time

aug_tf = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(), transforms.Normalize(MEAN, STD)])
aug_train = Subset(datasets.CIFAR10('data', train=True, download=True,
                                    transform=aug_tf), range(4000))

class Block(nn.Module):
    def __init__(self, cin, cout, stride=1, residual=True):
        super().__init__()
        self.residual = residual
        self.body = nn.Sequential(
            nn.Conv2d(cin, cout, 3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(cout), nn.ReLU(),
            nn.Conv2d(cout, cout, 3, padding=1, bias=False),
            nn.BatchNorm2d(cout))
        self.shortcut = (nn.Sequential(
            nn.Conv2d(cin, cout, 1, stride=stride, bias=False),
            nn.BatchNorm2d(cout))
            if stride != 1 or cin != cout else nn.Identity())
        self.act = nn.ReLU()

    def forward(self, x):
        h = self.body(x)
        return self.act(self.shortcut(x) + h if self.residual
                        else h)

```

```
def resnet(residual=True, width=32):
    torch.manual_seed(0)
    layers = [nn.Conv2d(3, width, 3, padding=1, bias=False),
              nn.BatchNorm2d(width), nn.ReLU()]
    cin = width
    for stage in range(3):
        cout = width * 2 ** stage
        for i in range(2):
            layers.append(Block(cin, cout,
                                stride=2 if (i == 0 and stage > 0) else 1,
                                residual=residual))
            cin = cout
    layers += [nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(cin, 10)]
    return nn.Sequential(*layers)

print('%-38s %10s %10s %8s' % ('', 'params', 'accuracy', 'time'))
for name, maker, train in [
        ('plain stack, same depth', lambda: resnet(residual=False), None),
        ('residual', resnet, None),
        ('residual + crop and flip', resnet, aug_train)]:
    model = maker()
    start = time.time()
    acc = fit(model, epochs=6, train=train)
    print('%-38s %10d %10.4f %7.0fs'
          % (name, sum(p.numel() for p in model.parameters()), acc,
             time.time() - start))
```

```
                                       params   accuracy     time
plain stack, same depth                696618     0.4870     133s
residual                               696618     0.5525     144s
residual + crop and flip               696618     0.4955     144s
```

What to reach for, in order

1. **A pretrained model**, unless your images are unlike anything in ImageNet or you have millions of your own.
2. **The preprocessing that came with it**, exactly.
3. **A frozen backbone and a new head** first, then fine-tune at a much lower rate with the normalisation layers in eval mode.
4. **Augmentation**: random crop and horizontal flip as a minimum, if a flip is valid in your domain.
5. **Residual connections** in anything you build yourself deeper than about eight layers.
6. **Global average pooling** rather than flatten, unless position carries the signal.
7. **A smaller model**, checked against the larger one, because most published architectures are sized for datasets far larger than yours.

Watch Out

The numbers on this page are deliberately small

Six thousand images and ten epochs on a CPU. A residual network of this shape on the full fifty thousand images with a hundred epoch schedule reaches well over 0.90, and the published figures are higher still. Everything here was sized so it runs while you read it, and the ordering of the results is what to take away, not their magnitudes.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. A plain 20-layer stack does worse than a 10-layer one on the training set as well as validation. What does that rule out?

   a) Overfitting, because a model that had memorised would do better on training data, not worse

   b) A bug in the data

   c) The learning rate

   d) Underfitting

2. What does a residual connection change about what a block has to learn?

   a) It halves the parameter count

   b) The block only has to learn the difference from its input, and can reach the identity by driving its weights towards zero

   c) It removes the need for normalisation

   d) It makes the block linear

3. A residual block changes the number of channels. What must the shortcut do?

   a) Drop the extra channels

   b) Nothing, addition broadcasts

   c) Carry a 1 by 1 convolution, so the two sides have the same shape before they are added

   d) Concatenate instead of add

4. What is a bottleneck block for?

   a) Slowing down the forward pass

   b) Replacing pooling

   c) Reducing overfitting

   d) Squeezing the channels down with a 1 by 1, doing the expensive 3 by 3 in that smaller space, then expanding back

5. What rescued the deep plain stack in week 6's measurement?

    a) Normalisation, which restored trainability at depth, with residual connections on top of it

    b) More data

    c) Dropout

    d) A smaller learning rate

6. You reuse a pretrained backbone and skip its preprocessing. What happens?

    a) Nothing important

    b) The inputs sit at a different scale from everything the weights were fitted to, so the features you get back are not the features it learned

    c) The model refuses to run

    d) Only the last layer is affected

7. What is the frozen layer that is not frozen?

    a) The final classifier

    b) A layer whose bias still updates

    c) A batch normalisation layer, whose running statistics keep moving in training mode even when its parameters need no gradient

    d) The embedding layer

8. Week 6 measured transfer learning across several label counts. What did it find on that task?

    a) It lost at every label count, which is why the week reports it rather than repeating the usual claim

    b) The comparison was inconclusive

    c) It won at every label count

    d) It won only when labels were scarce

Answers:1a, 2b, 3c, 4d, 5a, 6b, 7c, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 7

Recurrent Networks

Week 7

OBJECTIVES

Carrying state across a sequence: RNN, LSTM and GRU, packing done properly, and next-element prediction.

Sequences have an order that carries meaning and lengths that vary. A recurrent layer handles both by processing one element at a time and carrying a state forward. This week builds that from the loop upwards, measures the point at which a plain recurrent layer forgets, fixes it with LSTM, and covers the padding mistake that silently corrupts the state of every short example in your batch. It ends by being honest about why attention replaced most of this, and where it did not.

Day 1 The Recurrence

A task that needs memory, and the layer that carries state forward

Images have neighbours in two directions and convolution exploits that. Sequences have one direction and an order that carries meaning, and they have a second property images do not: they vary in length. A recurrent layer handles both by processing one element at a time and carrying a state forward.

A task that needs memory

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
```

```
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
print('training sequences', tuple(X_tr.shape))
print('class balance %.3f' % y_tr.float().mean())
print()
for i in range(4):
    seq = X_tr[i].tolist()
    print('%s  first=%d  repeated later: %s'
          % (''.join(str(d) for d in seq), seq[0], bool(y_tr[i])))
```

```
training sequences (8000, 8)
class balance 0.509

95626647  first=9  repeated later: False
16633531  first=1  repeated later: True
37574514  first=3  repeated later: False
74238582  first=7  repeated later: False
```

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
```

```
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
from sklearn.linear_model import LogisticRegression
import numpy as np

# A bag of counts: how many of each digit, order discarded.
def counts(x):
    return torch.stack([(x == d).sum(dim=1) for d in range(1, 10)],
                       dim=1).float().numpy()

model = LogisticRegression(max_iter=1000).fit(counts(X_tr), y_tr.numpy())
print('bag of counts, logistic regression %.4f'
      % model.score(counts(X_va), y_va.numpy()))
print('always predicting the majority class %.4f'
      % max(y_va.float().mean().item(), 1 - y_va.float().mean().item()))
print('\nthe counts do not say which digit came first, so they cannot')
print('separate the classes. Order is the entire signal here.')
```

```
bag of counts, logistic regression 0.4960
always predicting the majority class 0.5020

the counts do not say which digit came first, so they cannot
separate the classes. Order is the entire signal here.
```

The recurrence, written out

```
import torch

torch.manual_seed(0)
d_in, d_hidden = 4, 3
Wx = torch.randn(d_in, d_hidden) * 0.5
Wh = torch.randn(d_hidden, d_hidden) * 0.5
b = torch.zeros(d_hidden)

def run(sequence):
    h = torch.zeros(d_hidden)
    for x in sequence:
        h = torch.tanh(x @ Wx + h @ Wh + b)     # the entire recurrence
    return h

a = torch.randn(1, d_in)
c = torch.randn(1, d_in)
print('state after [a, c]', run(torch.cat([a, c])).round(decimals=3).tolist())
print('state after [c, a]', run(torch.cat([c, a])).round(decimals=3).tolist())
print('\nsame two inputs, different order, different state.')
print('that is the whole reason this layer exists.')
```

```
state after [a, c] [0.4440000057220459, 0.9139999747276306, -0.7350000143051147]
state after [c, a] [0.8899999856948853, -0.13199999928474426, -0.9919999837875366]

same two inputs, different order, different state.
that is the whole reason this layer exists.
```

PyTorch does the loop for you

```python
import torch
from torch import nn

torch.manual_seed(0)
rnn = nn.RNN(input_size=4, hidden_size=3, batch_first=True)
x = torch.randn(2, 6, 4)              # batch 2, 6 steps, 4 features

output, final = rnn(x)
print('input   ', tuple(x.shape))
print('output  ', tuple(output.shape), ' the state at every step')
print('final   ', tuple(final.shape), '  the state after the last step')
print('\nthey agree:',
      bool(torch.allclose(output[:, -1, :], final[0])))
```

```
input    (2, 6, 4)
output   (2, 6, 3)  the state at every step
final    (1, 2, 3)   the state after the last step

they agree: True
```

Watch Out

batch_first is not the default

PyTorch's recurrent layers expect `(sequence, batch, features)` unless you pass `batch_first=True`, which gives you the `(batch, sequence, features)` that every other layer in the library uses. Forgetting it does not raise an error when your batch size and sequence length happen to be compatible, it just trains on transposed data. Always pass it.

Day 1 takeaway

Day 2 Why Plain RNNs Forget

The gradient as a product, and what LSTM added to fix it

A plain recurrent layer can, in principle, remember something from a hundred steps ago. In practice it cannot, and the reason is visible in one line of arithmetic.

Why the gradient dies

```
print('%8s %18s %18s' % ('steps', 'factor 0.8', 'factor 1.2'))
for steps in [5, 10, 25, 50, 100]:
    print('%8d %18.8g %18.8g' % (steps, 0.8 ** steps, 1.2 ** steps))
print('\nbackpropagating through t steps multiplies t derivatives.')
print('below one it vanishes; above one it explodes. There is no')
print('setting that is stable for a long sequence.')
```

```
   steps         factor 0.8         factor 1.2
       5            0.32768            2.48832
      10         0.10737418          6.1917364
      25       0.0037778932          95.396217
      50      1.4272477e-05          9100.4382
     100       2.037036e-10           82817975

backpropagating through t steps multiplies t derivatives.
below one it vanishes; above one it explodes. There is no
setting that is stable for a long sequence.
```

```
import torch
from torch import nn

torch.manual_seed(0)
rnn = nn.RNN(1, 16, batch_first=True)

print('%10s %20s' % ('length', 'gradient at step 0'))
for length in [5, 20, 50, 100]:
    x = torch.randn(1, length, 1, requires_grad=True)
    out, _ = rnn(x)
    out[:, -1, :].sum().backward()
    print('%10d %20.3e' % (length, x.grad[0, 0].abs().item()))
print('\nthe influence of the first element on the last output')
print('falls away to nothing as the sequence gets longer.')
```

```
    length   gradient at step 0
         5            1.438e-02
        20            5.273e-07
        50            1.087e-14
       100            6.320e-28

the influence of the first element on the last output
falls away to nothing as the sequence gets longer.
```

What LSTM changed

```
import torch
from torch import nn

print('%-12s %12s %s' % ('cell', 'parameters', 'internal transforms'))
for name, layer, transforms in [
        ('RNN', nn.RNN(16, 32, batch_first=True), 1),
        ('GRU', nn.GRU(16, 32, batch_first=True), 3),
        ('LSTM', nn.LSTM(16, 32, batch_first=True), 4)]:
    n = sum(p.numel() for p in layer.parameters())
```

```
    print('%-12s %12d %d' % (name, n, transforms))

torch.manual_seed(0)
lstm = nn.LSTM(1, 16, batch_first=True)
print('\n%10s %20s' % ('length', 'gradient at step 0'))
for length in [5, 20, 50, 100]:
    x = torch.randn(1, length, 1, requires_grad=True)
    out, _ = lstm(x)
    out[:, -1, :].sum().backward()
    print('%10d %20.3e' % (length, x.grad[0, 0].abs().item()))
```

```
cell           parameters internal transforms
RNN                  1600 1
GRU                  4800 3
LSTM                 6400 4

    length   gradient at step 0
         5            3.287e-03
        20            3.803e-07
        50            3.276e-14
       100            8.178e-27
```

```
import torch
from torch import nn

torch.manual_seed(0)
lstm = nn.LSTM(4, 8, batch_first=True)
x = torch.randn(2, 5, 4)
output, (h, c) = lstm(x)
print('output    ', tuple(output.shape))
print('hidden h  ', tuple(h.shape), ' what the layer outputs')
print('cell c    ', tuple(c.shape), ' the internal memory')
print('\nLSTM returns a tuple where RNN and GRU return one tensor,')
print('which is the most common source of an unpacking error.')
```

```
output     (2, 5, 8)
hidden h   (1, 2, 8)  what the layer outputs
cell c     (1, 2, 8)  the internal memory

LSTM returns a tuple where RNN and GRU return one tensor,
which is the most common source of an unpacking error.
```

Day 2 takeaway

Day 3 Building a Sequence Classifier

Embeddings, the three cells, and which state to classify from

A working sequence classifier needs an embedding at the front, a recurrent layer in the middle, and a decision about which state to hand to the classifier.

Embedding the tokens

```python
import torch
from torch import nn

torch.manual_seed(0)
emb = nn.Embedding(num_embeddings=10, embedding_dim=8)
x = torch.tensor([[1, 5, 9, 0], [2, 2, 3, 4]])

print('input  ', tuple(x.shape), ' integers')
print('output ', tuple(emb(x).shape), ' a vector per token')
print('weight ', tuple(emb.weight.shape))
print('\nit is a lookup table, not a matrix multiplication:')
print('row 5 of the table  ', emb.weight[5][:4].round(decimals=3).tolist())
print('embedding of token 5', emb(torch.tensor(5))[:4].round(decimals=3).tolist())
```

```
input   (2, 4)  integers
output  (2, 4, 8)  a vector per token
weight  (10, 8)

it is a lookup table, not a matrix multiplication:
row 5 of the table   [-0.10199999809265137, 0.7919999957084656, -0.28999999165534973,
    ↪ 0.05299999937415123]
embedding of token 5 [-0.10199999809265137, 0.7919999957084656, -0.28999999165534973,
    ↪ 0.05299999937415123]
```

Why not one-hot

One-hot encoding a vocabulary of 50,000 words gives every token a 50,000 long vector in which every pair of distinct words is equally dissimilar. An embedding gives each token a short dense vector that is learned, so tokens used in similar ways end up near each other. It is also exactly equivalent to a one-hot vector multiplied by a weight matrix, implemented as a lookup because multiplying by a vector of zeros with a single one in it is a waste of a matrix multiplication.

Which state goes to the classifier

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
```

```
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Classifier(nn.Module):
    def __init__(self, pooling='last', cell='lstm', hidden=64):
        super().__init__()
        self.pooling = pooling
        self.emb = nn.Embedding(10, 32)
        rnn_cls = {'rnn': nn.RNN, 'gru': nn.GRU, 'lstm': nn.LSTM}[cell]
        self.rnn = rnn_cls(32, hidden, batch_first=True)
        self.head = nn.Linear(hidden, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        if self.pooling == 'last':
            h = out[:, -1, :]
        elif self.pooling == 'mean':
            h = out.mean(dim=1)
        else:
            h = out.max(dim=1).values
        return self.head(h)

torch.manual_seed(0)
```

```
print('%-10s %12s' % ('pooling', 'accuracy'))
for pooling in ['last', 'mean', 'max']:
    torch.manual_seed(0)
    print('%-10s %12.4f' % (pooling, fit(Classifier(pooling=pooling))))
```

```
pooling        accuracy
last             0.9960
mean             0.9245
max              0.9820
```

The three cells on the real task

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
```

```
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Classifier(nn.Module):
    def __init__(self, cell='lstm', hidden=64):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        rnn_cls = {'rnn': nn.RNN, 'gru': nn.GRU, 'lstm': nn.LSTM}[cell]
        self.rnn = rnn_cls(32, hidden, batch_first=True)
        self.head = nn.Linear(hidden, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

print('%-8s %12s %12s' % ('cell', 'params', 'accuracy'))
for cell in ['rnn', 'gru', 'lstm']:
    torch.manual_seed(0)
    model = Classifier(cell=cell)
    acc = fit(model)
    print('%-8s %12d %12.4f'
          % (cell, sum(p.numel() for p in model.parameters()), acc))
```

```
cell           params     accuracy
rnn              6722       0.6500
gru             19266       0.9985
lstm            25538       0.9960
```

Day 3 takeaway

Day 4 Direction and Length

Bidirectional reading, and the padding that quietly corrupts your state

Two more architectural choices, and then the practical problem that dominates real sequence work: examples that are not all the same length.

Reading in both directions

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

```

```
def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Classifier(nn.Module):
    def __init__(self, bidirectional=False, layers=1, hidden=64):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = nn.LSTM(32, hidden, num_layers=layers,
                           bidirectional=bidirectional, batch_first=True)
        self.head = nn.Linear(hidden * (2 if bidirectional else 1), 2)

```

```
    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

print('%-22s %12s %12s' % ('', 'params', 'accuracy'))
for label, kwargs in [('one layer', dict()),
                      ('two layers', dict(layers=2)),
                      ('bidirectional', dict(bidirectional=True))]:
    torch.manual_seed(0)
    model = Classifier(**kwargs)
    print('%-22s %12d %12.4f'
          % (label, sum(p.numel() for p in model.parameters()),
             fit(model)))
```

```
                         params     accuracy
one layer                 25538       0.9960
two layers                58818       0.7425
bidirectional             50754       0.9920
```

Watch Out

Bidirectional is not allowed if the future is unknown

Reading the sequence backwards as well as forwards helps a great deal when the whole sequence is available at once, which is true for classifying a finished sentence. It is impossible when you are predicting the next element, or processing a live stream, because the backward pass would need to read input that has not happened yet. Using it in a forecasting model is a leak, and a subtle one, because nothing raises.

Sequences of different lengths

```
import torch
from torch import nn
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence

seqs = [torch.tensor([1, 2, 3, 4, 5]),
        torch.tensor([6, 7]),
        torch.tensor([8, 9, 1])]
lengths = torch.tensor([len(s) for s in seqs])
padded = pad_sequence(seqs, batch_first=True)
print('padded to a rectangle:')
print(padded)

torch.manual_seed(0)
emb = nn.Embedding(10, 4)
lstm = nn.LSTM(4, 5, batch_first=True)

# the naive way: the LSTM walks over the padding too
out, (h_naive, _) = lstm(emb(padded))

packed = pack_padded_sequence(emb(padded), lengths, batch_first=True,
                              enforce_sorted=False)
```

```
_, (h_packed, _) = lstm(packed)

print('\nfinal state for the short sequence [6, 7]:')
print('  padded ', h_naive[0, 1].round(decimals=3).tolist())
print('  packed ', h_packed[0, 1].round(decimals=3).tolist())
print('\nthey differ: the padded version kept processing three zeros')
print('after the sequence ended, so its final state is wrong.')
```

```
padded to a rectangle:
tensor([[1, 2, 3, 4, 5],
        [6, 7, 0, 0, 0],
        [8, 9, 1, 0, 0]])

final state for the short sequence [6, 7]:
  padded  [-0.06700000166893005, -0.21199999749660492, 0.328000009059906,
    ↪ -0.04399999976158142, -0.08799999952316284]
  packed  [-0.09700000286102295, -0.15399999916553497, 0.009999999776482582,
    ↪ -0.010999999940395355, -0.1509999930858612]

they differ: the padded version kept processing three zeros
after the sequence ended, so its final state is wrong.
```

Packing is not an optimisation, it is a correctness fix

It is often described as saving computation, and it does. The reason to use it is that the final hidden state of a padded sequence is the state after processing the padding, not after processing the data. For a model that takes the last state, that is simply the wrong number. Either pack the sequence, or gather the state at each example's real final index yourself.

```
import torch
from torch import nn
from torch.nn.utils.rnn import pad_sequence

torch.manual_seed(0)
seqs = [torch.tensor([1, 2, 3, 4, 5]), torch.tensor([6, 7]),
        torch.tensor([8, 9, 1])]
lengths = torch.tensor([len(s) for s in seqs])
padded = pad_sequence(seqs, batch_first=True)

emb = nn.Embedding(10, 4)
lstm = nn.LSTM(4, 5, batch_first=True)
out, _ = lstm(emb(padded))

# gather the output at each sequence's real last position
idx = (lengths - 1).view(-1, 1, 1).expand(-1, 1, out.size(2))
last = out.gather(1, idx).squeeze(1)
print('states gathered at the real final step:')
print(last.round(decimals=3))
print('\nand for mean pooling, divide by the true length:')
mask = (padded != 0).unsqueeze(-1)
mean = (out * mask).sum(1) / lengths.unsqueeze(1)
print(mean.round(decimals=3))
```

```
states gathered at the real final step:
```

```
tensor([[-0.1190, -0.1840, -0.0420,  0.0230, -0.1500],
        [-0.0970, -0.1540,  0.0100, -0.0110, -0.1510],
        [ 0.2020,  0.0690,  0.1850, -0.0380, -0.0780]],
       grad_fn=<RoundBackward1>)

and for mean pooling, divide by the true length:
tensor([[ 0.0210, -0.0560,  0.0650, -0.0510, -0.0560],
        [-0.0670, -0.1310,  0.0260, -0.0080, -0.1290],
        [ 0.1580,  0.0440,  0.1230,  0.0620, -0.0950]],
       grad_fn=<RoundBackward1>)
```

Day 4 takeaway

Day 5 Recurrent Failure Modes

Clipping, sequence length, and the dropout argument that does nothing

Recurrent models fail in ways that are specific to them, and all three are avoidable once you know the shape of the failure.

Exploding gradients, and the one-line fix

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
```

```
                           shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Classifier(nn.Module):
    def __init__(self, hidden=64):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = nn.RNN(32, hidden, batch_first=True)
        self.head = nn.Linear(hidden, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

# Clipping bounds the size of a step, so measure the steps rather
# than the final accuracy, which barely moves on a task this easy.
for clip in [None, 1.0]:
    torch.manual_seed(0)
    model = Classifier()
    opt = torch.optim.Adam(model.parameters(), lr=0.02)
    loss_fn = nn.CrossEntropyLoss()
    worst = 0.0
    for _ in range(3):
        model.train()
        for xb, yb in train_loader:
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            # measured AFTER clipping, so this is the step actually taken
            used = torch.cat([prm.grad.flatten()
                              for prm in model.parameters()]).norm()
            worst = max(worst, used.item())
            opt.step()
    print('%-18s largest step actually taken %8.2f'
          % ('without clipping' if clip is None else 'with clipping',
```

```
              worst))
```

```
without clipping   largest step actually taken     4.29
with clipping      largest step actually taken     1.00
```

Without clipping the largest step the optimiser took was several times the ceiling clipping imposes. That is the whole mechanism: the gradient computed is the same either way, and clipping shortens the step before it is applied while leaving its direction alone. A maximum norm of 1.0 is the standard starting point. On a task this small the model survives either way; on a long sequence model or a transformer, leaving it out is how a run that was going well becomes `nan` at step forty thousand.

Sequence length is a hyperparameter

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
```

```
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Classifier(nn.Module):
    def __init__(self, cell, hidden=64):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = {'rnn': nn.RNN, 'lstm': nn.LSTM}[cell](
            32, hidden, batch_first=True)
        self.head = nn.Linear(hidden, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

print('%8s %14s %14s' % ('length', 'plain RNN', 'LSTM'))
for length in [6, 10, 16]:
    Xa, ya = make_sequences(8000, length=length, seed=0)
    Xb, yb = make_sequences(2000, length=length, seed=1)
    tl = DataLoader(TensorDataset(Xa, ya), batch_size=64, shuffle=True)
    vl = DataLoader(TensorDataset(Xb, yb), batch_size=512)
    row = []
    for cell in ['rnn', 'lstm']:
        torch.manual_seed(0)
        row.append(fit(Classifier(cell), epochs=10, clip=1.0,
                       loader=tl, val=vl))
    print('%8d %14.4f %14.4f' % (length, row[0], row[1]))
```

```
  length      plain RNN           LSTM
       6         0.7815         1.0000
      10         0.5555         0.8760
      16         0.4930         0.5225
```

Recurrent dropout is not ordinary dropout

```
import torch
from torch import nn

torch.manual_seed(0)
stacked = nn.LSTM(16, 32, num_layers=2, dropout=0.3, batch_first=True)
print('num_layers=2 with dropout=0.3: applied between the two layers')

single = nn.LSTM(16, 32, num_layers=1, dropout=0.3, batch_first=True)
```

```
x = torch.randn(2, 5, 16)
single.train()
a, _ = single(x)
b, _ = single(x)
print('one layer with dropout=0.3, two forward passes identical:',
      bool(torch.allclose(a, b)))
print('\nthe dropout argument only acts between stacked layers, so on a')
print('single layer it silently does nothing at all.')
```

```
num_layers=2 with dropout=0.3: applied between the two layers
one layer with dropout=0.3, two forward passes identical: True

the dropout argument only acts between stacked layers, so on a
single layer it silently does nothing at all.
```

Watch Out

Dropout inside a recurrent layer needs the same mask every step

Applying an independent dropout mask at each time step injects fresh noise into the state on every step, and over twenty steps that destroys the memory the layer exists to maintain. Correct recurrent dropout uses the *same* mask at every step of a sequence. PyTorch's `dropout` argument sidesteps the question by only applying between stacked layers. If you want it on the input or the output, add an ordinary `nn.Dropout` there yourself.

Day 5 takeaway

Day 6 Predicting the Next Element

One output per position, and generating by feeding the model itself

Classifying a sequence uses one output. Predicting the next element uses one output per position, and the difference changes how you build the batch, the loss and the evaluation.

One prediction per position

```
import torch
from torch import nn

torch.manual_seed(0)

class NextToken(nn.Module):
    def __init__(self, vocab=10, hidden=32):
        super().__init__()
        self.emb = nn.Embedding(vocab, 16)
        self.rnn = nn.LSTM(16, hidden, batch_first=True)
        self.head = nn.Linear(hidden, vocab)

    def forward(self, x):
```

```
        out, _ = self.rnn(self.emb(x))
        return self.head(out)            # a score per position per token

model = NextToken()
x = torch.randint(0, 10, (4, 12))
logits = model(x)
print('input ', tuple(x.shape))
print('output', tuple(logits.shape), '= (batch, position, vocabulary)')

# the target at each position is the next token
inputs, targets = x[:, :-1], x[:, 1:]
logits = model(inputs)
loss = nn.CrossEntropyLoss()(logits.reshape(-1, 10), targets.reshape(-1))
print('\nloss over %d positions: %.4f'
      % (targets.numel(), loss.item()))
print('CrossEntropyLoss wants (N, classes), so both are flattened.')
```

```
input  (4, 12)
output (4, 12, 10) = (batch, position, vocabulary)

loss over 44 positions: 2.2874
CrossEntropyLoss wants (N, classes), so both are flattened.
```

The shift is the whole task

Feed the model positions 0 to n-1 and ask it to predict positions 1 to n. Every position is a training example, so a batch of 32 sequences of length 128 gives you roughly four thousand predictions rather than 32. That efficiency is why language models are trained this way, and week 11 builds on it directly.

Learning a real pattern

```
import torch
from torch import nn

torch.manual_seed(0)

# A repeating pattern with a period of 5, plus noise the model cannot
# predict, so a perfect score is impossible and progress is meaningful.
g = torch.Generator().manual_seed(0)
base = torch.tensor([1, 2, 3, 4, 5])
seq = base.repeat(400).clone()
noise = torch.rand(seq.shape, generator=g) < 0.1
seq[noise] = torch.randint(1, 6, (int(noise.sum()),), generator=g)
data = seq.view(-1, 20)

class NextToken(nn.Module):
    def __init__(self, vocab=6, hidden=32):
        super().__init__()
        self.emb = nn.Embedding(vocab, 16)
        self.rnn = nn.LSTM(16, hidden, batch_first=True)
        self.head = nn.Linear(hidden, vocab)
```

```
    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out)

model = NextToken()
opt = torch.optim.Adam(model.parameters(), lr=5e-3)
loss_fn = nn.CrossEntropyLoss()
for epoch in range(1, 61):
    inputs, targets = data[:, :-1], data[:, 1:]
    opt.zero_grad()
    loss = loss_fn(model(inputs).reshape(-1, 6), targets.reshape(-1))
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    opt.step()
    if epoch % 20 == 0:
        acc = (model(inputs).argmax(-1) == targets).float().mean()
        print('epoch %2d  loss %.4f  next-token accuracy %.4f'
              % (epoch, loss.item(), acc.item()))
print('\n10 percent of the tokens are random, so about 0.90 is the ceiling.')
```

```
epoch 20  loss 0.8798  next-token accuracy 0.8516
epoch 40  loss 0.4998  next-token accuracy 0.8858
epoch 60  loss 0.4304  next-token accuracy 0.9074

10 percent of the tokens are random, so about 0.90 is the ceiling.
```

Generating from it

```
import torch
from torch import nn

torch.manual_seed(0)
g = torch.Generator().manual_seed(0)
base = torch.tensor([1, 2, 3, 4, 5])
seq = base.repeat(400).clone()
noise = torch.rand(seq.shape, generator=g) < 0.1
seq[noise] = torch.randint(1, 6, (int(noise.sum()),), generator=g)
data = seq.view(-1, 20)

class NextToken(nn.Module):
    def __init__(self, vocab=6, hidden=32):
        super().__init__()
        self.emb = nn.Embedding(vocab, 16)
        self.rnn = nn.LSTM(16, hidden, batch_first=True)
        self.head = nn.Linear(hidden, vocab)
    def forward(self, x, state=None):
        out, state = self.rnn(self.emb(x), state)
        return self.head(out), state

model = NextToken()
opt = torch.optim.Adam(model.parameters(), lr=5e-3)
```

```
loss_fn = nn.CrossEntropyLoss()
for _ in range(60):
    opt.zero_grad()
    logits, _ = model(data[:, :-1])
    loss_fn(logits.reshape(-1, 6), data[:, 1:].reshape(-1)).backward()
    opt.step()

model.eval()
for temperature in [0.2, 1.0, 2.0]:
    torch.manual_seed(0)
    token = torch.tensor([[1]])
    state, produced = None, [1]
    with torch.no_grad():
        for _ in range(19):
            logits, state = model(token, state)
            probs = (logits[0, -1] / temperature).softmax(-1)
            token = torch.multinomial(probs, 1).view(1, 1)
            produced.append(token.item())
    print('temperature %.1f: %s'
          % (temperature, ''.join(str(t) for t in produced)))
```

```
temperature 0.2: 12345123451234512345
temperature 1.0: 12345123544223451234
temperature 2.0: 12345125544122453425
```

Low temperature sharpens the distribution and the model repeats the pattern cleanly. High temperature flattens it and the output drifts into noise. That single parameter is the same one every text generation interface exposes, and week 11 uses it again on real language.

Day 6 takeaway

Day 7 Where Recurrence Stands

The full model, the baselines, and an honest account of what replaced it

The week assembled, and an honest account of where recurrent models stand now that attention exists.

The complete classifier

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
```

```
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
import time

class SequenceClassifier(nn.Module):
    def __init__(self, vocab=10, embed=32, hidden=64, layers=1,
                 bidirectional=True, dropout=0.2):
        super().__init__()
        self.emb = nn.Embedding(vocab, embed)
        self.drop = nn.Dropout(dropout)
        self.rnn = nn.LSTM(embed, hidden, num_layers=layers,
                           bidirectional=bidirectional, batch_first=True)
        self.head = nn.Linear(hidden * (2 if bidirectional else 1), 2)

    def forward(self, x):
        out, _ = self.rnn(self.drop(self.emb(x)))
```

```
        pooled = torch.cat([out.max(dim=1).values, out.mean(dim=1)], dim=1)
        return self.head(self.drop(pooled[:, :out.size(2)]))

torch.manual_seed(0)
model = SequenceClassifier()
start = time.time()
acc = fit(model, epochs=8, clip=1.0)
print('parameters %d' % sum(p.numel() for p in model.parameters()))
print('accuracy   %.4f' % acc)
print('time       %.0fs' % (time.time() - start))
```

```
parameters 50754
accuracy   0.9440
time       8s
```

The comparison

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Sequences of digits 1 to 9, balanced by construction.

    The label is 1 if the digit at position 0 appears again later. The
    rest of the sequence is drawn from the other eight digits, and for
    the positive half the first digit is planted at one random later
    position, so the classes are exactly balanced.

    A bag of counts cannot answer it. Both classes contain repeated
    digits, and counts never record which digit came first. Answering it
    means holding position 0 in memory until the sequence ends."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()      # 1 to 9, excluding first
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, clip=None, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
```

```
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            if clip:
                torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Simple(nn.Module):
    def __init__(self, cell, bidirectional=False):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = {'rnn': nn.RNN, 'gru': nn.GRU, 'lstm': nn.LSTM}[cell](
            32, 64, bidirectional=bidirectional, batch_first=True)
        self.head = nn.Linear(64 * (2 if bidirectional else 1), 2)
    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

class BagOfEmbeddings(nn.Module):
    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.head = nn.Sequential(nn.Linear(32, 64), nn.ReLU(),
                                  nn.Linear(64, 2))
    def forward(self, x):
        return self.head(self.emb(x).mean(dim=1))

print('%-28s %10s %10s' % ('', 'params', 'accuracy'))
for label, maker in [('bag of embeddings', BagOfEmbeddings),
                     ('plain RNN', lambda: Simple('rnn')),
                     ('GRU', lambda: Simple('gru')),
                     ('LSTM', lambda: Simple('lstm')),
                     ('bidirectional LSTM',
                      lambda: Simple('lstm', bidirectional=True))]:
    torch.manual_seed(0)
    model = maker()
    print('%-28s %10d %10.4f'
          % (label, sum(p.numel() for p in model.parameters()),
             fit(model, epochs=8, clip=1.0)))
```

```
                             params   accuracy
bag of embeddings              2562     0.6315
plain RNN                      6722     0.6085
GRU                           19266     0.9820
LSTM                          25538     0.8120
bidirectional LSTM            50754     0.8800
```

Watch Out

Recurrent models have been largely replaced, and are not obsolete

The reason is structural. A recurrent layer must process position 500 after position 499, so it cannot use a GPU's parallelism along the sequence, and training time grows linearly with length. Attention computes every position at once. That single difference is most of why transformers took over, and it is the subject of the next two weeks.
Where recurrence still wins: very long or unbounded streams, where attention's cost grows with the square of the length; small models on small devices; and anything genuinely online, where you receive one element at a time and must respond before the next arrives. A carried state is exactly the right shape for that, and an attention window is not.

The checklist

1. `batch_first=True`, every time.
2. Embedding at the front, not one-hot.
3. Pack the batch or gather at the true final index; never take the last state of a padded sequence.
4. Clip the gradient norm at 1.0.
5. LSTM or GRU rather than a plain RNN for anything longer than about twenty steps.
6. Bidirectional only when the whole sequence is available, never for forecasting.
7. Try last-state and pooled representations; the answer varies by task.
8. Fit a bag-of-embeddings baseline, because on many real tasks order matters far less than people assume.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What does a recurrent layer carry that a dense layer does not?

   a) A convolution kernel
   b) An attention mask
   c) More parameters
   d) A hidden state passed from one position to the next, so the output at step t depends on everything before it

2. Why do plain RNNs forget?

   a) The gradient through many steps is a product of many factors, so it shrinks or explodes exponentially with distance
   b) The activation is not differentiable
   c) The optimiser resets the state

d) The hidden state is too small

3. What did the LSTM add?

   a) More layers

   b) A cell state with gates that decide what to keep, letting information travel far without being multiplied down at every step

   c) Bidirectional reading

   d) Positional encodings

4. You classify a padded batch from the final hidden state. What goes wrong?

   a) Only the longest sequence trains

   b) Nothing

   c) For short sequences that final state reflects the padding rather than the content, so you must take the state at the true last position

   d) The loss becomes negative

5. What does the `dropout` argument on an RNN module do when `num_layers=1`?

   a) Nothing at all, because it only applies between stacked layers

   b) Raises an error

   c) Applies dropout to the hidden state

   d) Applies dropout to the input

6. Why is a bidirectional layer unusable for next-element prediction?

   a) The backward direction has already read the future, so the model would be predicting something it can see

   b) It cannot be stacked

   c) It doubles the parameter count

   d) It is too slow

7. What is teacher forcing?

   a) Training with a larger learning rate

   b) Feeding the true previous element during training rather than the model's own output, which trains faster and leaves a gap at generation

   c) Freezing the encoder

   d) Averaging predictions over several models

8. What is the honest account of where recurrence stands?

   a) It is preferred whenever data is scarce

   b) It is obsolete and never worth using

   c) It remains the strongest option for text

   d) Attention replaced it for most sequence work, largely because recurrence cannot be parallelised across positions, but it is still reasonable for short streaming sequences

Answers:1d, 2a, 3b, 4c, 5a, 6a, 7b, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 8

Attention

Week 8

OBJECTIVES

Scaled dot-product attention built from scratch, multiple heads, masks, positions, and the block every transformer is made of.

A recurrent layer squeezes everything it has read into one vector. Attention removes that bottleneck by keeping every position and computing, per query, how much of each to read. This week builds it from a dot product upwards: why the scaling factor exists, what the heads are for, how masks make generation possible, and why position has to be added back. It ends with the block that every transformer is a stack of, measured on the same task week 7 used.

Day 1 The Bottleneck

Why one fixed-size state is not enough, and the idea that replaced it

A recurrent layer compresses everything it has read into one fixed-size vector. For a short sequence that is fine. For a long one it is a bottleneck, and the fix is to stop compressing and start looking back.

The bottleneck, measured

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Week 7's task: is the digit at position 0 repeated later?
    Balanced by construction, and unanswerable from a bag of counts."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
```

```
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class LastState(nn.Module):
    def __init__(self, hidden):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = nn.LSTM(32, hidden, batch_first=True)
        self.head = nn.Linear(hidden, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

print('%14s %12s' % ('hidden size', 'accuracy'))
for hidden in [2, 4, 8, 64]:
    torch.manual_seed(0)
    print('%14d %12.4f' % (hidden, fit(LastState(hidden))))
```

```
   hidden size     accuracy
             2       0.5000
             4       0.5025
             8       0.5050
            64       0.9860
```

Everything the model knows about eight digits has to fit through that vector. Squeeze it and performance falls, because the state is genuinely carrying information rather than acting as a formality. Now imagine a sentence of two hundred words compressed into the same place, which is what machine translation was doing in 2014.

The idea

```
import torch

```

```
# Four positions, each with a value we might want to read.
values = torch.tensor([[1.0, 0.0],
                       [0.0, 1.0],
                       [2.0, 2.0],
                       [-1.0, 0.5]])

for name, weights in [('all of position 2', torch.tensor([0., 0., 1., 0.])),
                      ('an even average', torch.tensor([.25, .25, .25, .25])),
                      ('mostly 0 and 1', torch.tensor([.45, .45, .05, .05]))]:
    print('%-20s → %s' % (name, (weights @ values).tolist()))
print('\nattention is this, with the weights computed rather than chosen.')
```

```
all of position 2    → [2.0, 2.0]
an even average      → [0.5, 0.875]
mostly 0 and 1       → [0.5, 0.574999988079071]

attention is this, with the weights computed rather than chosen.
```

Day 1 takeaway

Day 2 Scaled Dot-Product Attention

Queries, keys and values, and why the square root is there

The weights come from asking every position how well it matches what you are looking for. That is a dot product, and the rest is bookkeeping.

Queries, keys and values

```
import torch

def softmax(z, dim=-1):
    z = z - z.max(dim=dim, keepdim=True).values
    e = z.exp()
    return e / e.sum(dim=dim, keepdim=True)

def attention(Q, K, V):
    d = K.shape[-1]
    scores = Q @ K.transpose(-2, -1) / d ** 0.5
    weights = softmax(scores)
    return weights @ V, weights

torch.manual_seed(0)
x = torch.randn(5, 8)                    # 5 positions, width 8
Wq, Wk, Wv = (torch.randn(8, 8) * 0.4 for _ in range(3))
out, w = attention(x @ Wq, x @ Wk, x @ Wv)

print('input  ', tuple(x.shape))
print('output ', tuple(out.shape), ' one vector per position')
print('weights', tuple(w.shape), ' one row per query')
```

```
print('\nevery row sums to 1:', bool(torch.allclose(w.sum(-1),
                                                   torch.ones(5))))
print('\nweights (row = the position doing the looking):')
for i, row in enumerate(w):
    print('  %d  %s' % (i, ' '.join('%.3f' % v for v in row.tolist())))
```

```
input   (5, 8)
output  (5, 8)  one vector per position
weights (5, 5)  one row per query

every row sums to 1: True

weights (row = the position doing the looking):
  0  0.623 0.181 0.063 0.108 0.026
  1  0.277 0.304 0.105 0.175 0.138
  2  0.030 0.058 0.091 0.060 0.760
  3  0.061 0.174 0.110 0.162 0.493
  4  0.045 0.039 0.425 0.070 0.420
```

Why the square root

```
import torch

torch.manual_seed(0)
print('%8s %16s %16s %16s'
      % ('width', 'score std', 'max weight', 'max weight scaled'))
for d in [8, 64, 512]:
    q = torch.randn(1, d)
    k = torch.randn(20, d)
    raw = q @ k.T
    scaled = raw / d ** 0.5
    print('%8d %16.3f %16.4f %16.4f'
          % (d, raw.std().item(), raw.softmax(-1).max().item(),
             scaled.softmax(-1).max().item()))
```

```
   width       score std      max weight max weight scaled
       8           3.579          0.7797           0.3327
      64           5.087          0.9167           0.1394
     512          27.192          0.9995           0.2701
```

Without the scaling, attention stops attending

A dot product of two random vectors of width d has a standard deviation proportional to the square root of d. At width 512 the scores are large enough that the softmax saturates: one position takes almost all the weight and the rest get nothing, which also means the gradient reaching them is nearly zero. Dividing by the square root of d holds the scores at a sensible size no matter how wide the model gets, and that single division is what the word *scaled* in "scaled dot product attention" refers to.

It is permutation equivariant, which is a problem

```python
import torch

def attention(Q, K, V):
    return (Q @ K.transpose(-2, -1) / K.shape[-1] ** 0.5).softmax(-1) @ V

torch.manual_seed(0)
x = torch.randn(4, 6)
Wq, Wk, Wv = (torch.randn(6, 6) * 0.4 for _ in range(3))

a = attention(x @ Wq, x @ Wk, x @ Wv)
order = [2, 0, 3, 1]
b = attention(x[order] @ Wq, x[order] @ Wk, x[order] @ Wv)

print('shuffle the positions and the outputs are the same rows,')
print('in the shuffled order:', bool(torch.allclose(a[order], b, atol=1e-6)))
print('\nso attention alone cannot tell 12345 from 54321.')
print('day 5 puts the position back.')
```

```
shuffle the positions and the outputs are the same rows,
in the shuffled order: True

so attention alone cannot tell 12345 from 54321.
day 5 puts the position back.
```

Day 2 takeaway

Day 3 Multiple Heads

Several relationships at once, for the same price

One set of weights can only express one kind of relationship at a time. Multi-head attention runs several in parallel on different projections and concatenates the results.

Several heads at once

```python
import torch
from torch import nn

class MultiHeadAttention(nn.Module):
    def __init__(self, width, heads):
        super().__init__()
        assert width % heads == 0
        self.heads = heads
        self.head_dim = width // heads
        self.qkv = nn.Linear(width, width * 3)
        self.out = nn.Linear(width, width)

    def forward(self, x, mask=None):
        B, T, C = x.shape
        q, k, v = self.qkv(x).split(C, dim=2)
```

```
        # split the width into heads and move heads next to the batch
        def heads(t):
            return t.view(B, T, self.heads, self.head_dim).transpose(1, 2)
        q, k, v = heads(q), heads(k), heads(v)

        scores = q @ k.transpose(-2, -1) / self.head_dim ** 0.5
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        weights = scores.softmax(-1)
        out = weights @ v                       # (B, heads, T, head_dim)
        out = out.transpose(1, 2).reshape(B, T, C)
        return self.out(out), weights

torch.manual_seed(0)
mha = MultiHeadAttention(width=32, heads=4)
x = torch.randn(2, 6, 32)
out, w = mha(x)
print('input   ', tuple(x.shape))
print('output  ', tuple(out.shape), ' same shape as the input')
print('weights ', tuple(w.shape), ' = (batch, heads, query, key)')
print('parameters %d' % sum(p.numel() for p in mha.parameters()))
```

```
input    (2, 6, 32)
output   (2, 6, 32)  same shape as the input
weights  (2, 4, 6, 6)  = (batch, heads, query, key)
parameters 4224
```

The heads are free

Splitting a width of 32 into four heads of 8 costs exactly the same parameters and the same arithmetic as one head of 32. You are not buying capacity, you are buying the ability to attend to several things at once: one head can track which positions share a digit while another tracks position, and averaging them into a single set of weights would have forced a compromise.

The heads really do differ

```
import torch
from torch import nn

torch.manual_seed(0)
width, heads, T = 32, 4, 6
qkv = nn.Linear(width, width * 3)
x = torch.randn(1, T, width)
B, _, C = x.shape
q, k, v = qkv(x).split(C, dim=2)
hd = width // heads
shape = lambda t: t.view(B, T, heads, hd).transpose(1, 2)
q, k = shape(q), shape(k)
w = (q @ k.transpose(-2, -1) / hd ** 0.5).softmax(-1)

for h in range(heads):
    row = w[0, h, 0]
```

```
    print('head %d, what position 0 looks at: %s'
          % (h, ' '.join('%.2f' % val for val in row.tolist())))
print('\nuntrained, so the pattern is arbitrary. The point is that the')
print('four rows differ, which one head could not produce.')
```

```
head 0, what position 0 looks at: 0.08 0.09 0.25 0.20 0.24 0.14
head 1, what position 0 looks at: 0.21 0.16 0.16 0.14 0.15 0.18
head 2, what position 0 looks at: 0.15 0.24 0.08 0.15 0.16 0.22
head 3, what position 0 looks at: 0.22 0.18 0.17 0.14 0.16 0.13

untrained, so the pattern is arbitrary. The point is that the
four rows differ, which one head could not produce.
```

Day 3 takeaway

Day 4 Masks

Self against cross attention, the causal mask, and the padding mask

Where the queries come from decides what kind of attention you have, and which positions you allow to be seen decides whether you can use it for generation.

Self-attention and cross-attention

	Queries from	Keys and values from	Used in
Self-attention	The sequence itself	The same sequence	Every transformer block
Cross-attention	The output sequence	The input sequence	Translation, captioning, any encoder to decoder model
Causal self-attention	The sequence itself	Earlier positions only	Language models

```
import torch
from torch import nn

torch.manual_seed(0)

def attend(q, k, v, mask=None):
    scores = q @ k.transpose(-2, -1) / k.shape[-1] ** 0.5
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))
    return scores.softmax(-1) @ v

source = torch.randn(1, 7, 16)      # e.g. a sentence in French
target = torch.randn(1, 4, 16)      # what we have produced in English

print('self-attention on the target  ',
      tuple(attend(target, target, target).shape))
print('cross-attention to the source ',
      tuple(attend(target, source, source).shape))
```

```
print('\none output per query position either way. Cross-attention lets')
print('each output position read the whole input.')
```

```
self-attention on the target   (1, 4, 16)
cross-attention to the source  (1, 4, 16)

one output per query position either way. Cross-attention lets
each output position read the whole input.
```

The causal mask

```
import torch

T = 6
mask = torch.tril(torch.ones(T, T))
print('lower triangular mask:')
print(mask.int())

torch.manual_seed(0)
scores = torch.randn(T, T)
masked = scores.masked_fill(mask == 0, float('-inf'))
weights = masked.softmax(-1)
print('\nattention weights after masking:')
for i, row in enumerate(weights):
    print('  pos %d  %s' % (i, ' '.join('%.2f' % v for v in row.tolist())))
print('\nposition 0 sees only itself; position 5 sees everything.')
print('no position can see its own future, which is what makes')
print('next-token training possible in one parallel pass.')
```

```
lower triangular mask:
tensor([[1, 0, 0, 0, 0, 0],
        [1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0],
        [1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1]], dtype=torch.int32)

attention weights after masking:
  pos 0  1.00 0.00 0.00 0.00 0.00 0.00
  pos 1  0.86 0.14 0.00 0.00 0.00 0.00
  pos 2  0.15 0.45 0.40 0.00 0.00 0.00
  pos 3  0.27 0.33 0.23 0.17 0.00 0.00
  pos 4  0.13 0.08 0.29 0.22 0.28 0.00
  pos 5  0.04 0.03 0.54 0.18 0.05 0.16

position 0 sees only itself; position 5 sees everything.
no position can see its own future, which is what makes
next-token training possible in one parallel pass.
```

This is why language models train so efficiently

Without the mask you would have to run the model once per position to predict each next token without cheating. With it, a single forward pass over a sequence of length 1,024 produces 1,024

predictions, each conditioned only on what came before it. Every position is a training example and they are all computed at once. That efficiency, not the architecture itself, is most of why large language models are possible.

The padding mask, which is a different thing

```python
import torch

# Two sequences of different lengths, padded with zeros.
tokens = torch.tensor([[3, 7, 2, 0, 0],
                       [4, 1, 8, 6, 5]])
pad_mask = (tokens != 0).unsqueeze(1)          # (batch, 1, positions)
print('padding mask, per sequence:')
print(pad_mask.squeeze(1).int())

torch.manual_seed(0)
scores = torch.randn(2, 5, 5)
masked = scores.masked_fill(pad_mask == 0, float('-inf'))
w = masked.softmax(-1)
print('\nsequence 0, weights from position 0:')
print('  ', '  '.join('%.3f' % v for v in w[0, 0].tolist()))
print('the two padded positions get exactly zero weight.')
print('\nwithout it, padding would contribute to every average, and')
print('a short sequence in a batch of long ones would be mostly padding.')
```

```
padding mask, per sequence:
tensor([[1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1]], dtype=torch.int32)

sequence 0, weights from position 0:
   0.229  0.223  0.549  0.000  0.000
the two padded positions get exactly zero weight.

without it, padding would contribute to every average, and
a short sequence in a batch of long ones would be mostly padding.
```

Watch Out

Two masks, combined with a logical and

A causal language model on padded batches needs both: the causal mask stops a position seeing the future, and the padding mask stops any position seeing padding. They are combined by multiplying or by a logical and before the `masked_fill`. Getting one and not the other produces a model that trains and is quietly wrong, which is the recurring theme of this course.

Day 4 takeaway

Day 5 Positional Information

Putting back the order that attention discards

Day 2 showed attention treating the sequence as a set. Since order usually matters, the position has to be added to the input, and there are three ways of doing it.

Learned position embeddings

```
import torch
from torch import nn

torch.manual_seed(0)
max_len, width = 8, 16
tokens = nn.Embedding(10, width)
positions = nn.Embedding(max_len, width)

x = torch.tensor([[3, 7, 2, 5]])
pos = torch.arange(x.shape[1])
combined = tokens(x) + positions(pos)

print('token embeddings   ', tuple(tokens(x).shape))
print('position embeddings', tuple(positions(pos).shape))
print('added together     ', tuple(combined.shape))
print('\nthe same token at two positions is now two different vectors:')
same = torch.tensor([[5, 5]])
out = tokens(same) + positions(torch.arange(2))
print('  difference %.4f' % (out[0, 0] - out[0, 1]).abs().mean().item())
```

```
token embeddings    (1, 4, 16)
position embeddings (4, 16)
added together      (1, 4, 16)

the same token at two positions is now two different vectors:
  difference 0.7602
```

Sinusoidal encodings

```
import torch
import math

def sinusoidal(max_len, width):
    pos = torch.arange(max_len).unsqueeze(1).float()
    i = torch.arange(0, width, 2).float()
    rate = torch.exp(-math.log(10000.0) * i / width)
    pe = torch.zeros(max_len, width)
    pe[:, 0::2] = torch.sin(pos * rate)
    pe[:, 1::2] = torch.cos(pos * rate)
    return pe

pe = sinusoidal(6, 8)
print('positional encodings, 6 positions of width 8:')
for i, row in enumerate(pe):
    print('  %d  %s' % (i, ' '.join('%+.2f' % v for v in row.tolist())))
```

```

print('\ndot product between position 0 and each other position:')
print(' ', ' '.join('%.2f' % v for v in (pe @ pe[0]).tolist()))
print('it falls away smoothly, so nearby positions are similar.')
```

```
positional encodings, 6 positions of width 8:
  0  +0.00 +1.00 +0.00 +1.00 +0.00 +1.00 +0.00 +1.00
  1  +0.84 +0.54 +0.10 +1.00 +0.01 +1.00 +0.00 +1.00
  2  +0.91 -0.42 +0.20 +0.98 +0.02 +1.00 +0.00 +1.00
  3  +0.14 -0.99 +0.30 +0.96 +0.03 +1.00 +0.00 +1.00
  4  -0.76 -0.65 +0.39 +0.92 +0.04 +1.00 +0.00 +1.00
  5  -0.96 +0.28 +0.48 +0.88 +0.05 +1.00 +0.00 +1.00

dot product between position 0 and each other position:
  4.00 3.54 2.56 1.96 2.27 3.16
it falls away smoothly, so nearby positions are similar.
```

Scheme	Extends past training length?	Used by
Learned embeddings	No, it has no row for position 5000	BERT, the original GPT
Sinusoidal	In principle yes, in practice poorly	The original transformer
Rotary (RoPE)	Much better	Llama, most current models
Relative position bias	Yes	T5

Whether it matters, measured

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Week 7's task: is the digit at position 0 repeated later?
    Balanced by construction, and unanswerable from a bag of counts."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
```

```
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class AttentionClassifier(nn.Module):
    def __init__(self, width=32, heads=4, use_positions=True):
        super().__init__()
        self.use_positions = use_positions
        self.tok = nn.Embedding(10, width)
        self.pos = nn.Embedding(16, width)
        self.attn = nn.MultiheadAttention(width, heads, batch_first=True)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, 2)

    def forward(self, x):
        h = self.tok(x)
        if self.use_positions:
            h = h + self.pos(torch.arange(x.shape[1]))
        a, _ = self.attn(h, h, h)
        h = self.norm(h + a)
        return self.head(h.mean(dim=1))

print('%-22s %12s' % ('', 'accuracy'))
for use in [False, True]:
    torch.manual_seed(0)
    print('%-22s %12.4f'
          % ('with positions' if use else 'no positions',
             fit(AttentionClassifier(use_positions=use))))
```

```
                       accuracy
no positions             0.5855
with positions           0.9995
```

This task asks whether the digit at *position 0* reappears, so a model with no notion of position cannot express the question. The gap between the two rows is the cost of treating a sequence as a set.

Day 5 takeaway

Day 6 The Library and the Cost

nn.MultiheadAttention, the inverted mask, and quadratic growth

The library version, and the cost that decides how long a sequence you can afford.

nn.MultiheadAttention

```
import torch
from torch import nn

torch.manual_seed(0)
mha = nn.MultiheadAttention(embed_dim=32, num_heads=4, batch_first=True)
x = torch.randn(2, 6, 32)

out, weights = mha(x, x, x)                  # query, key, value
print('output ', tuple(out.shape))
print('weights', tuple(weights.shape), ' averaged over heads by default')

out, weights = mha(x, x, x, average_attn_weights=False)
print('per head', tuple(weights.shape))

causal = torch.triu(torch.ones(6, 6, dtype=torch.bool), diagonal=1)
out, w = mha(x, x, x, attn_mask=causal)
print('\nwith a causal mask, position 0 attends to:',
      ' '.join('%.2f' % v for v in w[0, 0].tolist()))
```

```
output  (2, 6, 32)
weights (2, 6, 6)  averaged over heads by default
per head (2, 4, 6, 6)

with a causal mask, position 0 attends to: 1.00 0.00 0.00 0.00 0.00 0.00
```

Watch Out

The mask convention is the opposite of what you expect

`nn.MultiheadAttention` takes `attn_mask` where `True` means *block this position*, which is the reverse of the `1 means keep` convention used when you write it yourself, and the reverse of `key_padding_mask` in some older code. Read the docstring every time. An inverted mask produces a model that trains happily while attending to exactly the positions it should not.

The cost

```
print('%10s %16s %18s' % ('length', 'attention scores', 'recurrent steps'))
for T in [8, 128, 1024, 8192]:
    print('%10d %16d %18d' % (T, T * T, T))
print('\nattention computes a score for every pair of positions, so')
print('memory and time grow with the square of the length. Recurrence')
print('grows linearly, but cannot be parallelised along the sequence.')
```

```
    length attention scores    recurrent steps
```

```
       8              64                8
     128           16384              128
    1024         1048576             1024
    8192        67108864             8192

attention computes a score for every pair of positions, so
memory and time grow with the square of the length. Recurrence
grows linearly, but cannot be parallelised along the sequence.
```

```python
import time
import torch
from torch import nn

torch.manual_seed(0)
attn = nn.MultiheadAttention(64, 4, batch_first=True)
lstm = nn.LSTM(64, 64, batch_first=True)

print('%10s %14s %14s' % ('length', 'attention', 'lstm'))
for T in [32, 128, 512]:
    x = torch.randn(8, T, 64)
    t = time.time()
    for _ in range(10):
        attn(x, x, x, need_weights=False)
    a = (time.time() - t) / 10
    t = time.time()
    for _ in range(10):
        lstm(x)
    b = (time.time() - t) / 10
    print('%10d %13.1fms %13.1fms' % (T, a * 1000, b * 1000))
```

```
    length      attention           lstm
        32           0.5ms           3.2ms
       128           1.8ms           2.3ms
       512           8.5ms           6.8ms
```

Read that table carefully

Attention is doing quadratically more arithmetic as the sequence grows, and it is still competitive or faster, because all of it happens in one large matrix multiplication that a modern processor is built for. The LSTM does linearly less work in a Python-level loop over time steps that cannot be parallelised at all.

This is the practical answer to why transformers won. Not that they do less work, but that the work they do is the shape hardware is fast at. It also means the crossover point moves as sequences get very long, which is why efficient attention variants exist.

> **Day 6 takeaway**

Day 7 The Transformer Block

Attention assembled, measured against week 7, and read honestly

A complete attention block, and the comparison against week 7's recurrent models on identical data.

The block

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Week 7's task: is the digit at position 0 repeated later?
    Balanced by construction, and unanswerable from a bag of counts."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class AttentionBlock(nn.Module):
    """Attention, then a small feed forward network, each wrapped in a
    residual connection with layer normalisation. This is the transformer
    block, and week 9 assembles a stack of them."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
```

```
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a                                  # residual
        x = x + self.ff(self.norm2(x))             # residual
        return x

torch.manual_seed(0)
block = AttentionBlock(32, 4)
x = torch.randn(2, 8, 32)
print('in ', tuple(x.shape), ' out', tuple(block(x).shape))
print('parameters %d' % sum(p.numel() for p in block.parameters()))
```

```
in  (2, 8, 32)  out (2, 8, 32)
parameters 12704
```

Normalise before, not after

The original transformer applied layer normalisation after the residual addition. Every model since about 2019 does it before, exactly as week 6 argued for pre-activation residual blocks: it keeps the shortcut path completely clear, so the gradient reaches the earliest block undiminished. Post-normalisation transformers need a careful warmup to train at all, and pre-normalisation ones largely do not.

Against the recurrent models

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Week 7's task: is the digit at position 0 repeated later?
    Balanced by construction, and unanswerable from a bag of counts."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
```

```
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
import time

class AttentionBlock(nn.Module):
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))

class AttentionClassifier(nn.Module):
    def __init__(self, width=32, heads=4, blocks=2):
        super().__init__()
        self.tok = nn.Embedding(10, width)
        self.pos = nn.Embedding(16, width)
        self.blocks = nn.ModuleList([AttentionBlock(width, heads)
                                     for _ in range(blocks)])
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, 2)

    def forward(self, x):
        h = self.tok(x) + self.pos(torch.arange(x.shape[1]))
        for block in self.blocks:
```

```
            h = block(h)
        return self.head(self.norm(h).mean(dim=1))

class Recurrent(nn.Module):
    def __init__(self, cell):
        super().__init__()
        self.emb = nn.Embedding(10, 32)
        self.rnn = {'rnn': nn.RNN, 'gru': nn.GRU,
                    'lstm': nn.LSTM}[cell](32, 64, batch_first=True)
        self.head = nn.Linear(64, 2)

    def forward(self, x):
        out, _ = self.rnn(self.emb(x))
        return self.head(out[:, -1, :])

print('%-26s %10s %10s %9s' % ('', 'params', 'accuracy', 'time'))
for name, maker in [('plain RNN', lambda: Recurrent('rnn')),
                    ('LSTM', lambda: Recurrent('lstm')),
                    ('attention, 1 block',
                     lambda: AttentionClassifier(blocks=1)),
                    ('attention, 2 blocks',
                     lambda: AttentionClassifier(blocks=2))]:
    torch.manual_seed(0)
    model = maker()
    start = time.time()
    acc = fit(model)
    print('%-26s %10d %10.4f %8.0fs'
          % (name, sum(p.numel() for p in model.parameters()), acc,
             time.time() - start))
```

```
                           params   accuracy       time
plain RNN                    6722     0.6380         5s
LSTM                        25538     0.9860         6s
attention, 1 block          13666     1.0000        10s
attention, 2 blocks         26370     1.0000        12s
```

Reading what it attended to

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

def make_sequences(n, length=8, seed=0):
    """Week 7's task: is the digit at position 0 repeated later?
    Balanced by construction, and unanswerable from a bag of counts."""
    g = torch.Generator().manual_seed(seed)
    first = torch.randint(1, 10, (n, 1), generator=g)
    rest = torch.randint(1, 9, (n, length - 1), generator=g)
    rest = rest + (rest >= first).long()
    y = torch.randint(0, 2, (n,), generator=g)
    where = torch.randint(0, length - 1, (n, 1), generator=g)
    idx = torch.arange(length - 1).unsqueeze(0)
```

```
    plant = (y.unsqueeze(1) == 1) & (idx == where)
    rest = torch.where(plant, first.expand_as(rest), rest)
    return torch.cat([first, rest], dim=1), y

X_tr, y_tr = make_sequences(8000, seed=0)
X_va, y_va = make_sequences(2000, seed=1)
train_loader = DataLoader(TensorDataset(X_tr, y_tr), batch_size=64,
                          shuffle=True)
val_loader = DataLoader(TensorDataset(X_va, y_va), batch_size=512)
def fit(model, epochs=10, lr=3e-3, loader=None, val=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(epochs):
        model.train()
        for xb, yb in (loader or train_loader):
            opt.zero_grad()
            loss_fn(model(xb), yb).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    right = seen = 0
    with torch.no_grad():
        for xb, yb in (val or val_loader):
            right += (model(xb).argmax(1) == yb).sum().item()
            seen += yb.numel()
    return right / seen
class Simple(nn.Module):
    def __init__(self, width=32, heads=1):
        super().__init__()
        self.tok = nn.Embedding(10, width)
        self.pos = nn.Embedding(16, width)
        self.attn = nn.MultiheadAttention(width, heads, batch_first=True)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, 2)

    def forward(self, x, want_weights=False):
        h = self.tok(x) + self.pos(torch.arange(x.shape[1]))
        a, w = self.attn(h, h, h)
        out = self.head(self.norm(h + a).mean(dim=1))
        return (out, w) if want_weights else out

torch.manual_seed(0)
model = Simple()
print('accuracy %.4f\n' % fit(model))

model.eval()
with torch.no_grad():
    _, w = model(X_va[:1], want_weights=True)
seq = X_va[0].tolist()
print('sequence %s, label %d' % (''.join(str(d) for d in seq), y_va[0]))
print('\nattention weights from each position:')
print('      ' + ' '.join('%5d' % d for d in seq))
for i, row in enumerate(w[0]):
```

```
    print('%2d %s  %s' % (i, str(seq[i]).rjust(2),
                          ' '.join('%5.2f' % v for v in row.tolist())))
```

```
accuracy 0.8725

sequence 57959964, label 1

attention weights from each position:
          5     7     9     5     9     9     6     4
 0  5   0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00
 1  7   0.00  0.00  0.10  0.28  0.19  0.35  0.02  0.06
 2  9   0.00  0.00  0.36  0.00  0.35  0.29  0.00  0.00
 3  5   0.93  0.00  0.00  0.07  0.00  0.00  0.00  0.00
 4  9   0.00  0.00  0.13  0.00  0.30  0.57  0.00  0.00
 5  9   0.00  0.00  0.43  0.00  0.33  0.24  0.00  0.00
 6  6   0.00  0.00  0.00  0.00  0.00  0.00  1.00  0.00
 7  4   0.00  0.00  0.00  0.00  0.00  0.00  0.00  1.00
```

Read the first row. Position 0 holds a 5, and it has put essentially all of its weight on the one later position that also holds a 5. The model has discovered the structure of the task and is using attention to do exactly the lookup the task requires, which is the clearest picture of attention working that this course can offer.

Watch Out

Do not over-read an attention map

It is tempting to treat these weights as an explanation. They are not: they show what the model looked at, not why it decided. A well-documented result is that attention weights can be changed substantially while leaving the prediction almost unaltered, which means they are not a faithful account of the reasoning. Read them as a debugging aid, in the same spirit as week 5's occlusion test, and not as evidence.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What bottleneck did attention remove?

   a) The batch size

   b) The size of the vocabulary

   c) Forcing an entire input sequence through one fixed-size state before anything could be decoded from it

   d) The depth of the network

2. Why is the dot product divided by the square root of the key dimension?

   a) To keep the values positive

b) It is a convention with no effect

c) To normalise the output

d) Because the dot product of two random vectors grows with dimension, and without the scaling the softmax saturates and its gradient dies

3. In self-attention, where do queries, keys and values come from?

   a) All from the same sequence, through three different projections

   b) Queries from the input, keys and values from a learned table

   c) They are the same tensor used three times

   d) Three different sequences

4. What do multiple heads buy?

   a) More parameters for the same compute

   b) Several attention patterns in parallel, each in a lower dimensional subspace, so the total cost is about the same as one wide head

   c) Faster convergence

   d) Longer context

5. What does a causal mask do?

   a) Normalises the attention rows

   b) Hides padding

   c) Sets the scores for future positions to minus infinity before the softmax, so no position can attend to anything after it

   d) Randomly drops attention weights

6. Why does attention need positional information added?

   a) To keep gradients stable

   b) Only for cross attention

   c) To speed up the softmax

   d) Because attention is a weighted sum over a set and is therefore blind to order unless position is encoded into the values themselves

7. What is the trap in `nn.MultiheadAttention`'s mask argument?

   a) The key padding mask marks positions to *ignore* with True, which is the opposite of the mask you usually build

   b) It ignores the mask in evaluation mode

   c) It requires a square mask

   d) It expects a float mask only

8. How does attention cost grow with sequence length?

   a) Linearly

   b) Quadratically, because every position scores every other position

   c) Logarithmically

   d) It does not depend on length

Answers:1c, 2d, 3a, 4b, 5c, 6d, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 9

The Transformer

Week 9

OBJECTIVES

Stacking attention into encoders and decoders, masking properly, and training a small language model that generates.

A transformer is the block from week 8, stacked, with an embedding at the front and a linear layer at the back. This week builds all three arrangements of it, shows the missing causal mask producing a model that looks perfect and generates nonsense, trains a sequence to sequence model on a copy task, and finishes with a small decoder-only model that infers a counting rule from next-token prediction alone.

Day 1 The Three Shapes

Encoder, decoder, and both, and why decoder only won

A transformer is a stack of the block week 8 finished with, plus an embedding at the front and a linear layer at the back. Everything else you have heard about them is a variation on where the blocks are and what they are allowed to look at.

The three shapes

Shape	Blocks see	Trained to	Examples
Encoder only	The whole sequence, both directions	Fill in masked positions	BERT, sentence embeddings, classifiers
Decoder only	Earlier positions only	Predict the next token	GPT, Llama, almost every current model
Encoder and decoder	Encoder sees all of the input; decoder sees the input and its own past	Produce one sequence from another	The original transformer, T5, translation

Decoder only won, and it is worth knowing why

Not because it is more expressive. An encoder can see both directions, which is strictly more information for understanding a sentence. Decoder only won because next-token prediction is a training task you can run on any text at all, with no labels, and because one model trained that way turns out to do translation, summarising and question answering without being built for any of them. The architecture is not the innovation. The training objective is.

An encoder, assembled

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class Encoder(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=2,
                 max_len=16):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.blocks = nn.ModuleList([Block(width, heads)
                                     for _ in range(blocks)])
        self.norm = nn.LayerNorm(width)

    def forward(self, x):
```

```
        h = self.tok(x) + self.pos(torch.arange(x.shape[1]))
        for block in self.blocks:
            h = block(h)
        return self.norm(h)

torch.manual_seed(0)
enc = Encoder()
x = torch.randint(1, 10, (2, 6))
print('input  ', tuple(x.shape))
print('output ', tuple(enc(x).shape), ' one vector per position')
print('parameters %d' % sum(p.numel() for p in enc.parameters()))
print('\nwhere the parameters are:')
for name, child in enc.named_children():
    print('  %-8s %d' % (name, sum(p.numel() for p in child.parameters())))
```

```
input   (2, 6)
output  (2, 6, 64)  one vector per position
parameters 101824

where the parameters are:
  tok      704
  pos      1024
  blocks   99968
  norm     128
```

Most of the weight is in the blocks, and inside a block most of it is in the feed forward network rather than the attention. That surprises people. Attention decides what to read; the feed forward layers are where most of the actual capacity lives, and current research on where a model stores facts keeps pointing at them.

Day 1 takeaway

Day 2 Masking and Cross-Attention

The leak that makes a language model look perfect and generate nonsense

Building the decoder means being careful about exactly what each position is allowed to see, because getting it wrong produces a model that scores beautifully and cannot generate anything.

The leak, demonstrated

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

```

```
    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class Decoder(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=2,
                 max_len=16, causal=True):
        super().__init__()
        self.causal = causal
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.blocks = nn.ModuleList([Block(width, heads)
                                     for _ in range(blocks)])
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab)

    def forward(self, x):
        T = x.shape[1]
        mask = None
        if self.causal:
            mask = torch.triu(torch.ones(T, T, dtype=torch.bool),
                              diagonal=1)
        h = self.tok(x) + self.pos(torch.arange(T))
        for block in self.blocks:
            h = block(h, mask)
        return self.head(self.norm(h))

def train_lm(causal, epochs=4):
```

```
    torch.manual_seed(0)
    model = Decoder(causal=causal)
    opt = torch.optim.AdamW(model.parameters(), lr=3e-3)
    loss_fn = nn.CrossEntropyLoss()
    data = torch.cat([torch.full((6000, 1), START),
                      make_copy(6000, seed=3)[0]], dim=1)
    loader = DataLoader(TensorDataset(data), batch_size=64, shuffle=True)
    for _ in range(epochs):
        model.train()
        for (batch,) in loader:
            opt.zero_grad()
            logits = model(batch[:, :-1])
            loss = loss_fn(logits.reshape(-1, VOCAB),
                           batch[:, 1:].reshape(-1))
            loss.backward()
            opt.step()
    return model, loss.item()

for causal in [True, False]:
    model, loss = train_lm(causal)
    print('%-22s final training loss %.4f'
          % ('causal mask' if causal else 'no mask', loss))
print('\nthe digits are uniform random, so the best possible loss is')
print('log(9) = %.4f. A loss far below that means the model is reading'
      % torch.tensor(9.0).log().item())
print('the answer, which is only possible without the mask.')
```

```
causal mask            final training loss 2.2127
no mask                final training loss 0.3644

the digits are uniform random, so the best possible loss is
log(9) = 2.1972. A loss far below that means the model is reading
the answer, which is only possible without the mask.
```

Watch Out

This is the most expensive bug in the subject

A loss well below what the task allows is not a triumph, it is a leak. Without the causal mask, position i can attend to position i+1, which is the token it is being asked to predict. The model learns to copy it, the loss collapses, every metric looks superb, and at generation time there is no next token to read so the output is nonsense.

Whenever a language model's training loss looks too good, check the mask before you check anything else. The arithmetic above is the check: work out the best loss the task permits and compare.

The encoder and decoder together

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

```

```
PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class CrossBlock(nn.Module):
    """A decoder block: masked self-attention, then cross-attention to
    the encoder output, then the feed forward network."""
    def __init__(self, width, heads, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.self_attn = nn.MultiheadAttention(width, heads,
                                               batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.cross_attn = nn.MultiheadAttention(width, heads,
                                                batch_first=True)
        self.norm3 = nn.LayerNorm(width)
        self.ff = nn.Sequential(nn.Linear(width, width * 4), nn.GELU(),
                                nn.Linear(width * 4, width))

    def forward(self, x, memory, mask=None):
        h = self.norm1(x)
        a, _ = self.self_attn(h, h, h, attn_mask=mask, need_weights=False)
```

```
        x = x + a
        h = self.norm2(x)
        c, _ = self.cross_attn(h, memory, memory, need_weights=False)
        x = x + c
        return x + self.ff(self.norm3(x))

torch.manual_seed(0)
block = CrossBlock(64, 4)
target = torch.randn(2, 6, 64)
memory = torch.randn(2, 9, 64)          # encoder output, different length
print('target ', tuple(target.shape))
print('memory ', tuple(memory.shape), ' a different length is fine')
print('output ', tuple(block(target, memory).shape))
print('\ncross-attention is where the two sequences meet, and it is the')
print('only place the decoder can see the input at all.')
```

```
target  (2, 6, 64)
memory  (2, 9, 64)  a different length is fine
output  (2, 6, 64)

cross-attention is where the two sequences meet, and it is the
only place the decoder can see the input at all.
```

Day 2 takeaway

Day 3 A Full Encoder and Decoder

The copy task, whole-sequence accuracy, and teacher forcing

A complete encoder and decoder trained on the copy task, which is the smallest problem that genuinely needs both halves.

The model

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
```

```
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class Seq2Seq(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=2,
                 max_len=16):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.encoder = nn.ModuleList([Block(width, heads)
                                      for _ in range(blocks)])
        self.self_attn = nn.ModuleList(
            [nn.MultiheadAttention(width, heads, batch_first=True)
             for _ in range(blocks)])
        self.cross_attn = nn.ModuleList(
            [nn.MultiheadAttention(width, heads, batch_first=True)
             for _ in range(blocks)])
        self.ff = nn.ModuleList(
            [nn.Sequential(nn.Linear(width, width * 4), nn.GELU(),
                           nn.Linear(width * 4, width))
             for _ in range(blocks)])
        self.norms = nn.ModuleList([nn.LayerNorm(width)
                                    for _ in range(blocks * 3)])
        self.out_norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab)

    def embed(self, x):
        return self.tok(x) + self.pos(torch.arange(x.shape[1]))

    def encode(self, src):
        h = self.embed(src)
        for block in self.encoder:
            h = block(h)
```

```
        return h

    def decode(self, tgt_in, memory):
        T = tgt_in.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.embed(tgt_in)
        for i in range(len(self.self_attn)):
            n1, n2, n3 = self.norms[i * 3:i * 3 + 3]
            q = n1(h)
            a, _ = self.self_attn[i](q, q, q, attn_mask=mask,
                                     need_weights=False)
            h = h + a
            q = n2(h)
            c, _ = self.cross_attn[i](q, memory, memory,
                                      need_weights=False)
            h = h + c
            h = h + self.ff[i](n3(h))
        return self.head(self.out_norm(h))

    def forward(self, src, tgt_in):
        return self.decode(tgt_in, self.encode(src))

torch.manual_seed(0)
model = Seq2Seq()
print('parameters %d' % sum(p.numel() for p in model.parameters()))
print('output', tuple(model(src_tr[:2], tin_tr[:2]).shape))
```

```
parameters 236043
output (2, 6, 11)
```

Training it

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
```

```
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
import time

class Seq2Seq(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=2,
                 max_len=16):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.encoder = nn.ModuleList([Block(width, heads)
                                      for _ in range(blocks)])
        self.self_attn = nn.ModuleList(
            [nn.MultiheadAttention(width, heads, batch_first=True)
             for _ in range(blocks)])
        self.cross_attn = nn.ModuleList(
            [nn.MultiheadAttention(width, heads, batch_first=True)
             for _ in range(blocks)])
        self.ff = nn.ModuleList(
            [nn.Sequential(nn.Linear(width, width * 4), nn.GELU(),
                           nn.Linear(width * 4, width))
             for _ in range(blocks)])
        self.norms = nn.ModuleList([nn.LayerNorm(width)
                                    for _ in range(blocks * 3)])
        self.out_norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab)

    def embed(self, x):
        return self.tok(x) + self.pos(torch.arange(x.shape[1]))

    def encode(self, src):
        h = self.embed(src)
        for block in self.encoder:
            h = block(h)
        return h

```

```
    def decode(self, tgt_in, memory):
        T = tgt_in.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.embed(tgt_in)
        for i in range(len(self.self_attn)):
            n1, n2, n3 = self.norms[i * 3:i * 3 + 3]
            q = n1(h)
            a, _ = self.self_attn[i](q, q, q, attn_mask=mask,
                                     need_weights=False)
            h = h + a
            q = n2(h)
            c, _ = self.cross_attn[i](q, memory, memory,
                                      need_weights=False)
            h = h + c
            h = h + self.ff[i](n3(h))
        return self.head(self.out_norm(h))

    def forward(self, src, tgt_in):
        return self.decode(tgt_in, self.encode(src))

torch.manual_seed(0)
model = Seq2Seq()
opt = torch.optim.AdamW(model.parameters(), lr=3e-3)
loss_fn = nn.CrossEntropyLoss()
start = time.time()
for epoch in range(1, 9):
    model.train()
    for s, ti, to in train_loader:
        opt.zero_grad()
        logits = model(s, ti)
        loss_fn(logits.reshape(-1, VOCAB), to.reshape(-1)).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step()
    if epoch % 2 == 0:
        model.eval()
        with torch.no_grad():
            pred = model(src_va, tin_va).argmax(-1)
            token_acc = (pred == tout_va).float().mean().item()
            exact = (pred == tout_va).all(dim=1).float().mean().item()
        print('epoch %d  token accuracy %.4f  whole sequence %.4f'
              % (epoch, token_acc, exact))
print('\n%.0f seconds' % (time.time() - start))
```

```
epoch 2  token accuracy 1.0000  whole sequence 1.0000
epoch 4  token accuracy 1.0000  whole sequence 1.0000
epoch 6  token accuracy 1.0000  whole sequence 1.0000
epoch 8  token accuracy 1.0000  whole sequence 1.0000

21 seconds
```

Teacher forcing, and the gap it hides

During training the decoder is fed the *correct* previous tokens, not its own predictions. That is teacher forcing, and it is what makes training parallel: every output position is computed at once. At generation time the model must feed on its own output, so a single early mistake changes everything after it, and the model has never trained on that situation. The gap between training loss and generation quality is largely this, and it is why day 4 measures generation separately.

Day 3 takeaway

Day 4 Generation

Greedy, temperature, top-k and nucleus, and why the settings matter

Generating from a trained model is a loop, and every decision inside it changes the output more than most architectural choices do.

Greedy decoding

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
```

```
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class Decoder(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=2,
                 max_len=16):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.blocks = nn.ModuleList([Block(width, heads)
                                     for _ in range(blocks)])
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab)

    def forward(self, x):
        T = x.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.tok(x) + self.pos(torch.arange(T))
        for block in self.blocks:
            h = block(h, mask)
        return self.head(self.norm(h))

# A language model over a repeating pattern, so we can tell whether the
# generated output is right.
torch.manual_seed(0)
g = torch.Generator().manual_seed(0)
pattern = torch.tensor([1, 2, 3, 4, 5, 6])
rows = pattern.repeat(3000).view(-1, 6)
noise = torch.rand(rows.shape, generator=g) < 0.05
rows[noise] = torch.randint(1, 10, (int(noise.sum()),), generator=g)
data = torch.cat([torch.full((len(rows), 1), START), rows], dim=1)

model = Decoder()
opt = torch.optim.AdamW(model.parameters(), lr=3e-3)
loss_fn = nn.CrossEntropyLoss()
loader = DataLoader(TensorDataset(data), batch_size=64, shuffle=True)
for _ in range(6):
    model.train()
    for (batch,) in loader:
        opt.zero_grad()
        loss_fn(model(batch[:, :-1]).reshape(-1, VOCAB),
                batch[:, 1:].reshape(-1)).backward()
        opt.step()

model.eval()
seq = torch.tensor([[START]])
with torch.no_grad():
    for _ in range(6):
```

```
        nxt = model(seq)[:, -1, :].argmax(-1, keepdim=True)
        seq = torch.cat([seq, nxt], dim=1)
print('greedy generation:', seq[0, 1:].tolist())
print('the pattern was  :', pattern.tolist())
```

```
greedy generation: [1, 2, 3, 4, 5, 6]
the pattern was  : [1, 2, 3, 4, 5, 6]
```

Temperature, top-k and nucleus sampling

```
import torch

torch.manual_seed(0)
logits = torch.tensor([3.2, 2.9, 1.4, 0.8, 0.4, 0.1, -0.5, -1.2])

def show(name, probs):
    print('%-18s %s' % (name, ' '.join('%.3f' % v for v in probs.tolist())))

show('raw softmax', logits.softmax(-1))
show('temperature 0.5', (logits / 0.5).softmax(-1))
show('temperature 2.0', (logits / 2.0).softmax(-1))

# top-k: keep the k largest, renormalise
k = 3
kth = logits.topk(k).values[-1]
show('top-k, k=3', logits.masked_fill(logits < kth, float('-inf')).softmax(-1))

# nucleus: keep the smallest set whose probability exceeds p
probs = logits.softmax(-1)
order = probs.argsort(descending=True)
cumulative = probs[order].cumsum(0)
keep = order[cumulative - probs[order] < 0.9]
filtered = torch.full_like(logits, float('-inf'))
filtered[keep] = logits[keep]
show('nucleus, p=0.9', filtered.softmax(-1))
```

```
raw softmax        0.467 0.346 0.077 0.042 0.028 0.021 0.012 0.006
temperature 0.5    0.629 0.345 0.017 0.005 0.002 0.001 0.000 0.000
temperature 2.0    0.303 0.261 0.123 0.091 0.075 0.064 0.048 0.034
top-k, k=3         0.525 0.389 0.087 0.000 0.000 0.000 0.000 0.000
nucleus, p=0.9     0.501 0.371 0.083 0.045 0.000 0.000 0.000 0.000
```

Strategy	Good for	Failure mode
Greedy	Tasks with one right answer: translation, copying	Repetitive and bland on open-ended text
Temperature sampling	Variety	Incoherence at high values
Top-k	Cutting off the nonsense tail	A fixed k is too small when the model is unsure and too large when it is confident
Nucleus (top-p)	The usual default now	Still needs a temperature alongside it
Beam search	Translation and summarising	Produces bland text on open-ended generation, and costs beam width times as much

Watch Out

Sampling settings are not cosmetic

The same model at temperature 0.7 with nucleus sampling and at temperature 1.5 with none behaves like two different systems. When you compare a model against a published result and cannot reproduce it, the decoding settings are the first thing to check, and they are frequently not reported. They belong in the model contract from week 15 of the Machine Learning course, alongside the threshold.

Day 4 takeaway

Day 5 Training a Transformer

Warmup, the failure table, and weight tying

Transformers are harder to train than convolutional networks, and the difficulties are specific and well known.

Warmup, and when it earns its place

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
```

```
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
class Block(nn.Module):
    """Pre-normalisation transformer block, as week 8 day 7 built it."""
    def __init__(self, width, heads, expansion=4, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = nn.MultiheadAttention(width, heads, dropout=dropout,
                                          batch_first=True)
        self.norm2 = nn.LayerNorm(width)
        self.ff = nn.Sequential(
            nn.Linear(width, width * expansion), nn.GELU(),
            nn.Linear(width * expansion, width), nn.Dropout(dropout))

    def forward(self, x, mask=None):
        h = self.norm1(x)
        a, _ = self.attn(h, h, h, attn_mask=mask, need_weights=False)
        x = x + a
        return x + self.ff(self.norm2(x))
class Small(nn.Module):
    def __init__(self, width=64, heads=4, blocks=2):
        super().__init__()
        self.tok = nn.Embedding(VOCAB, width)
        self.pos = nn.Embedding(16, width)
        self.blocks = nn.ModuleList([Block(width, heads)
                                     for _ in range(blocks)])
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, VOCAB)

    def forward(self, x):
        T = x.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.tok(x) + self.pos(torch.arange(T))
        for block in self.blocks:
            h = block(h, mask)
        return self.head(self.norm(h))

# Learnable data, so the loss can actually fall and the warmup has
# something to affect. Random digits would already sit at their floor.
g = torch.Generator().manual_seed(5)
starts = torch.randint(1, 5, (4000, 1), generator=g)
data = torch.cat([torch.full((4000, 1), START),
                  starts + torch.arange(6)], dim=1)
loader = DataLoader(TensorDataset(data), batch_size=64, shuffle=True)
loss_fn = nn.CrossEntropyLoss()

def run(lr, warmup_steps):
    torch.manual_seed(0)
```

```
    model = Small()
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    step, losses = 0, []
    for _ in range(3):
        model.train()
        for (batch,) in loader:
            step += 1
            if warmup_steps:
                scale = min(1.0, step / warmup_steps)
                for group in opt.param_groups:
                    group['lr'] = lr * scale
            opt.zero_grad()
            loss = loss_fn(model(batch[:, :-1]).reshape(-1, VOCAB),
                           batch[:, 1:].reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
            losses.append(loss.item())
    return losses

print('%-28s %14s %14s' % ('', 'loss at step 20', 'final loss'))
for lr, warm in [(1e-2, 0), (1e-2, 100), (3e-3, 0)]:
    losses = run(lr, warm)
    print('%-28s %14.4f %14.4f'
          % ('lr %g, warmup %d' % (lr, warm), losses[19], losses[-1]))
```

```
                            loss at step 20     final loss
lr 0.01, warmup 0                    0.2426         0.2340
lr 0.01, warmup 100                  0.4119         0.2297
lr 0.003, warmup 0                   0.2583         0.2347
```

The best loss this task allows is about 0.23: the first token has four possibilities and every token after it is determined, so the average is log(4) divided by six. All three runs reach it, and the warmed-up run is behind at step 20 for the obvious reason that its learning rate is still climbing.

Watch Out

Warmup bought nothing here, and that is worth understanding

A three-block model on a six-token task is not where warmup matters. The problem it solves is specific: Adam's running estimates of gradient magnitude are unreliable for the first few dozen steps, so a full-size step taken on that estimate can move a large model somewhere it never recovers from. With a hundred and fifty thousand parameters there is not enough depth for that to happen.

It becomes close to mandatory as models get deep, as batches get large, and with post-normalisation blocks. Since it costs a few lines and cannot hurt, it stays in the recipe. But do not expect to measure its benefit on anything you can train while you watch.

Where the difficulties are

Symptom	Cause	Fix
Loss spikes or becomes `nan` early	Learning rate too high for the first steps	Warmup, then clip at norm 1.0
Loss falls to near zero immediately	The causal mask is missing or inverted	Compare against the best loss the task permits
Trains but generates nonsense	Trained with teacher forcing only, or wrong mask at inference	Evaluate by generating, not only by loss
Runs out of memory as sequences lengthen	Attention is quadratic in length	Shorter context, gradient checkpointing, or an efficient attention implementation
Good on short inputs, poor on long ones	Position encodings do not extrapolate	Rotary encodings, or train at the length you will serve

Weight tying

```
import torch
from torch import nn

vocab, width = 5000, 256
embed = nn.Embedding(vocab, width)
head = nn.Linear(width, vocab, bias=False)
print('embedding %d, output head %d'
      % (embed.weight.numel(), head.weight.numel()))

head.weight = embed.weight            # one tensor, used twice
total = sum({id(p): p.numel()
             for p in list(embed.parameters()) +
             list(head.parameters())}.values())
print('after tying, distinct parameters %d' % total)
print('\nthe input embedding maps a token to a vector; the output head')
print('maps a vector to a score per token. Sharing them saves a lot and')
print('usually improves the model slightly.')
```

```
embedding 1280000, output head 1280000
after tying, distinct parameters 1280000

the input embedding maps a token to a vector; the output head
maps a vector to a score per token. Sharing them saves a lot and
usually improves the model slightly.
```

Day 5 takeaway

Day 6 The Library

Built-in layers, the two wrong defaults, and fused attention

PyTorch ships transformer layers, and they are worth using, provided you know which

arguments matter.

The built-in layers

```
import torch
from torch import nn

torch.manual_seed(0)
layer = nn.TransformerEncoderLayer(d_model=64, nhead=4,
                                   dim_feedforward=256, dropout=0.1,
                                   batch_first=True, norm_first=True)
encoder = nn.TransformerEncoder(layer, num_layers=3)

x = torch.randn(2, 10, 64)
print('output', tuple(encoder(x).shape))
print('parameters %d' % sum(p.numel() for p in encoder.parameters()))

causal = torch.triu(torch.ones(10, 10, dtype=torch.bool), diagonal=1)
print('with a causal mask:', tuple(encoder(x, mask=causal).shape))
```

```
output (2, 10, 64)
parameters 149952
with a causal mask: (2, 10, 64)
```

Watch Out

norm_first defaults to False

Which gives you the 2017 post-normalisation block, not the pre-normalisation one that every model since has used and that day 3 explained. The default is there for backwards compatibility and it will make your model harder to train, needing a longer warmup and a lower rate. Pass `norm_first=True`. Pass `batch_first=True` as well, for the same reason as the recurrent layers.

Efficient attention

```
import torch
from torch import nn
import torch.nn.functional as F

torch.manual_seed(0)
B, H, T, D = 2, 4, 32, 16
q, k, v = (torch.randn(B, H, T, D) for _ in range(3))

# by hand
scores = q @ k.transpose(-2, -1) / D ** 0.5
manual = scores.softmax(-1) @ v

# the fused implementation
fused = F.scaled_dot_product_attention(q, k, v)

```

```
print('same answer:', bool(torch.allclose(manual, fused, atol=1e-5)))
print('\nand with a causal mask:')
causal_manual = scores.masked_fill(
    torch.triu(torch.ones(T, T, dtype=torch.bool), 1),
    float('-inf')).softmax(-1) @ v
causal_fused = F.scaled_dot_product_attention(q, k, v, is_causal=True)
print('same answer:', bool(torch.allclose(causal_manual, causal_fused,
                                          atol=1e-5)))
```

```
same answer: True

and with a causal mask:
same answer: True
```

`scaled_dot_product_attention` is the same arithmetic in a fused kernel that never builds the full score matrix in memory. On a GPU that turns the memory cost from quadratic in the sequence length to linear, which is what makes long contexts affordable. Use it rather than writing the four lines yourself, and pass `is_causal=True` instead of building a mask.

Counting the cost

```
def transformer_params(vocab, width, blocks, heads, expansion=4,
                       max_len=1024, tied=True):
    embed = vocab * width + max_len * width
    attn = 4 * width * width                # q, k, v and output
    ff = 2 * width * width * expansion
    norms = 4 * width
    per_block = attn + ff + norms
    head = 0 if tied else vocab * width
    return embed + blocks * per_block + head

print('%-22s %10s %14s' % ('model', 'blocks', 'parameters'))
for name, width, blocks, heads, vocab in [
        ('tiny', 128, 4, 4, 5000),
        ('small', 512, 8, 8, 32000),
        ('GPT-2 sized', 768, 12, 12, 50257),
        ('GPT-2 medium', 1024, 24, 16, 50257)]:
    n = transformer_params(vocab, width, blocks, heads)
    print('%-22s %10d %14s' % (name, blocks, '{:,}'.format(n)))
print('\nand at 4 bytes per parameter, GPT-2 sized is about %.0f MB'
      % (transformer_params(50257, 768, 12, 12) * 4 / 1e6))
```

```
model                      blocks     parameters
tiny                            4      1,559,552
small                           8     42,090,496
GPT-2 sized                    12    124,355,328
GPT-2 medium                   24    354,599,936

and at 4 bytes per parameter, GPT-2 sized is about 497 MB
```

Day 6 takeaway

Day 7 A Small GPT

A decoder-only model that infers a rule nobody told it

The week assembled: a decoder-only transformer trained as a language model, generating, and measured against everything before it.

The model, using the library

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
                          batch_size=64, shuffle=True)
import time

class GPT(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=3,
                 max_len=16, dropout=0.1):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight          # tied
```

```
    def forward(self, x):
        T = x.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(x) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, prompt, steps, temperature=1.0, top_k=None):
        self.eval()
        seq = prompt
        for _ in range(steps):
            logits = self(seq)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(top_k).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            seq = torch.cat([seq, nxt], dim=1)
        return seq

torch.manual_seed(0)
model = GPT()
print('parameters %d' % sum(p.numel() for p in model.parameters()))
print('output', tuple(model(torch.randint(1, 10, (2, 6))).shape))
```

```
parameters 151808
output (2, 6, 11)
```

Trained, and generating

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

PAD, START = 0, 10
VOCAB = 11                          # digits 1 to 9, plus pad and start

def make_copy(n, length=6, seed=0):
    """Source is a sequence of digits; target is the same sequence.

    Trivial for a person and a real test for a model: it has to align
    output position i with input position i, which is exactly what
    cross-attention is for."""
    g = torch.Generator().manual_seed(seed)
    src = torch.randint(1, 10, (n, length), generator=g)
    start = torch.full((n, 1), START)
    tgt_in = torch.cat([start, src[:, :-1]], dim=1)
    return src, tgt_in, src

src_tr, tin_tr, tout_tr = make_copy(6000, seed=0)
src_va, tin_va, tout_va = make_copy(1000, seed=1)
train_loader = DataLoader(TensorDataset(src_tr, tin_tr, tout_tr),
```

```
                          batch_size=64, shuffle=True)
import time

class GPT(nn.Module):
    def __init__(self, vocab=VOCAB, width=64, heads=4, blocks=3,
                 max_len=16, dropout=0.1):
        super().__init__()
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(max_len, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, x):
        T = x.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(x) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, prompt, steps, temperature=1.0):
        self.eval()
        seq = prompt
        for _ in range(steps):
            logits = self(seq)[:, -1, :] / temperature
            seq = torch.cat([seq, torch.multinomial(logits.softmax(-1), 1)],
                            dim=1)
        return seq

# a language with a rule: every sequence counts up from where it starts
torch.manual_seed(0)
g = torch.Generator().manual_seed(0)
starts = torch.randint(1, 5, (6000, 1), generator=g)
rows = starts + torch.arange(6)
data = torch.cat([torch.full((6000, 1), START), rows], dim=1)
loader = DataLoader(TensorDataset(data), batch_size=64, shuffle=True)

model = GPT()
opt = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=0.01)
loss_fn = nn.CrossEntropyLoss()
start = time.time()
step = 0
for epoch in range(1, 7):
    model.train()
    for (batch,) in loader:
        step += 1
        for group in opt.param_groups:
            group['lr'] = 3e-3 * min(1.0, step / 100)      # warmup
```

```
        opt.zero_grad()
        loss = loss_fn(model(batch[:, :-1]).reshape(-1, VOCAB),
                       batch[:, 1:].reshape(-1))
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step()
    if epoch % 2 == 0:
        print('epoch %d  loss %.4f' % (epoch, loss.item()))
print('trained in %.0f seconds\n' % (time.time() - start))

for temperature in [0.5, 1.0]:
    torch.manual_seed(0)
    out = model.generate(torch.full((3, 1), START), 6,
                         temperature=temperature)
    print('temperature %.1f:' % temperature)
    for row in out[:, 1:].tolist():
        rule = all(row[i + 1] == row[i] + 1 for i in range(len(row) - 1))
        print('   %s  follows the rule: %s' % (row, rule))
```

```
epoch 2  loss 0.2484
epoch 4  loss 0.2523
epoch 6  loss 0.2607
trained in 14 seconds

temperature 0.5:
   [1, 2, 3, 4, 5, 6]  follows the rule: True
   [1, 2, 3, 4, 5, 6]  follows the rule: True
   [1, 2, 3, 4, 5, 6]  follows the rule: True
temperature 1.0:
   [1, 2, 3, 4, 5, 6]  follows the rule: True
   [1, 2, 3, 4, 5, 6]  follows the rule: True
   [1, 2, 3, 4, 5, 6]  follows the rule: True
```

It learned a rule it was never told

Nothing in the training loop mentions counting. The model saw six thousand sequences that happened to count upwards and inferred the pattern from next-token prediction alone. That is the whole idea behind language model pretraining, at a scale you can watch: predict the next thing, and structure you never specified falls out.

The checklist

1. Pre-normalisation blocks: `norm_first=True`.
2. A causal mask for anything that generates, and check the loss against the best the task permits.
3. Warm the learning rate up over a few hundred steps.
4. Clip the gradient norm at 1.0.
5. AdamW, with weight decay excluded from normalisation parameters and biases.
6. Tie the input embedding to the output head.
7. Evaluate by generating, not only by loss.

8. Record the decoding settings with the weights.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. Why did decoder-only architectures come to dominate?

   a) They are the only ones that can be parallelised

   b) One stack trained on one objective handles generation and, with prompting, most of what the other shapes were built for

   c) They need no positional encoding

   d) They use fewer parameters per layer

2. A language model reaches near-perfect training loss and then generates nonsense. What is the classic cause?

   a) Weight tying

   b) The learning rate was too high

   c) A missing or wrongly shaped causal mask, so during training each position could see the answer it was meant to predict

   d) The vocabulary was too small

3. What is cross-attention?

   a) Attention applied twice

   b) Attention across the batch

   c) Attention between heads

   d) Queries from the decoder against keys and values from the encoder

4. What does whole-sequence accuracy measure that per-token accuracy hides?

   a) Whether every position in an output is right, which is what matters when a single wrong token ruins the result

   b) The perplexity

   c) The attention entropy

   d) Nothing extra

5. What does temperature do to sampling?

   a) Changes the number of tokens generated

   b) Divides the logits before the softmax, so below one the distribution sharpens towards greedy and above one it flattens

   c) Removes the lowest probability tokens

   d) Scales the learning rate

6. What is the difference between top-k and nucleus sampling?

a) Top-k applies only to the first token

b) None, they are two names for the same thing

c) Top-k keeps a fixed number of candidates, nucleus keeps as many as are needed to reach a probability mass, so its cut adapts to the step

d) Nucleus is greedy decoding with noise

7. Week 9 tested learning rate warmup. What did it show on a small model?

   a) Training diverged without it

   b) It reduced the parameter count

   c) A large improvement

   d) Nothing measurable, because the instability warmup exists to prevent belongs to much larger models

8. What is weight tying?

   a) Using the same matrix for the input embedding and the output projection, which cuts parameters and often helps

   b) Freezing the embedding

   c) Averaging weights across epochs

   d) Sharing weights between attention heads

Answers:1b, 2c, 3d, 4a, 5b, 6c, 7d, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 10

Language Modelling

Week 10

OBJECTIVES

Training a character-level transformer end to end, with the baseline, the tokenisation and the arithmetic that surround it.

A language model produces a distribution over what comes next, and everything else is that operation repeated. This week trains one on a corpus written into the page, so it runs offline and everybody sees the same characters. It covers the baselines worth beating, what context length and model size actually buy, how tokenisation works and why it decides your API bill, and finishes by being clear about the distance between this and a model anybody would use.

Day 1 Predicting the Next Character

The corpus, the shifted target, and the loss you have to beat

A language model does one thing: given some text, produce a probability distribution over what comes next. Everything else people do with them is built on that one operation repeated.

The corpus

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
```

```
            'the weights', 'the hidden state', 'the input',
            'the residual path', 'the attention weights',
            'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
             'before the softmax', 'after the residual connection',
             'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
print('characters %d' % len(text))
print('vocabulary %d: %s' % (VOCAB, ''.join(chars)))
print()
print(text[:180])
print()
print('encoded:', encode('the model')[:12].tolist())
print('decoded:', decode(encode('the model')))
```

```
characters 28539
vocabulary 27:  ,.abcdefghiklmnopqrstuvwxy

each layer copies the weights during training, so the gradient controls the weights at every
    ↪ step. attention reduces the residual path once per epoch. each layer reduces the
    ↪ traini

encoded: [21, 10, 7, 0, 14, 16, 6, 7, 13]
decoded: the model
```

Character level, and why it is the right place to start

A vocabulary of a few dozen characters means no tokeniser to explain, no unknown tokens, and a model small enough to train in seconds. Everything about the training loop, the mask and the sampling is identical to a model with a fifty thousand token vocabulary. Day 5 covers what changes when you move to subword tokens, and the answer is less than you would expect.

Every position is a training example

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

```

```
CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
x, y = X_tr[0], Y_tr[0]
print('input :', repr(decode(x[:32])))
print('target:', repr(decode(y[:32])))
print('\nthe target is the input shifted one place left.')
print()
for i in range(5):
    print('given %-22r predict %r'
          % (decode(x[:i + 1]), decode(y[i:i + 1])))
print('\n%d windows of %d characters, so %d predictions per epoch'
      % (len(X_tr), BLOCK_SIZE, len(X_tr) * BLOCK_SIZE))
```

```
input : 'each layer copies the weights du'
target: 'ach layer copies the weights dur'

the target is the input shifted one place left.

given 'e'                    predict 'a'
given 'ea'                   predict 'c'
given 'eac'                  predict 'h'
given 'each'                 predict ' '
given 'each '                predict 'l'
```

```
6413 windows of 32 characters, so 205216 predictions per epoch
```

What a good loss looks like

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)
```

```

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)      # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
import math
from collections import Counter

counts = Counter(text)
total = sum(counts.values())
uniform = math.log(VOCAB)
unigram = -sum((n / total) * math.log(n / total) for n in counts.values())

print('loss if the model guesses uniformly     %.4f' % uniform)
print('loss from character frequencies alone   %.4f' % unigram)
print('\nperplexity is exp(loss), the effective number of choices:')
print('  uniform  %.1f' % math.exp(uniform))
print('  unigram  %.1f' % math.exp(unigram))
print('\nanything above the unigram number means the model has not')
print('learned even which letters are common.')
```

```
loss if the model guesses uniformly     3.2958
loss from character frequencies alone   2.8138

perplexity is exp(loss), the effective number of choices:
  uniform  27.0
  unigram  16.7

anything above the unigram number means the model has not
learned even which letters are common.
```

Day 1 takeaway

Day 2 Baselines and a First Model

Bigram counts, a small transformer, and what it writes

Two baselines before the transformer, so the improvement can be measured rather than assumed.

Counting bigrams

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

```

```python
def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)      # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
import math

# The simplest possible language model: for each character, count what
# followed it, and predict from those counts.
counts = torch.zeros(VOCAB, VOCAB)
for a, b in zip(data[:split][:-1], data[:split][1:]):
    counts[a, b] += 1
probs = (counts + 1) / (counts + 1).sum(1, keepdim=True)   # smoothed

val = data[split:]
logp = probs[val[:-1], val[1:]].log().mean()
print('bigram validation loss %.4f, perplexity %.1f'
      % (-logp.item(), math.exp(-logp.item())))

torch.manual_seed(0)
out, current = [], stoi['t']
for _ in range(120):
    current = int(torch.multinomial(probs[current], 1))
    out.append(current)
print('\ngenerated from bigram counts:')
print(repr(decode(out)))
```

```
bigram validation loss 1.9837, perplexity 7.3

generated from bigram counts:
'heske dutmaicocobesintcrathtcr thore cevatthe ethk, ioseaninthol ecos l axtain h t
    ↪ widddimponts t oconp. pimalie putl rn'
```

It has learned that `q` is followed by `u` and that spaces are common. It has learned nothing about words, because it can only see one character back. Every improvement from here is about seeing more context.

A small transformer

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

```

```
def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)      # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
```

```
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
import time
torch.manual_seed(0)
model = GPT()
print('parameters %d' % sum(p.numel() for p in model.parameters()))
start = time.time()
train_loss, val_loss = train(model, epochs=5)
import math
print('train loss %.4f  validation loss %.4f  perplexity %.1f'
      % (train_loss, val_loss, math.exp(val_loss)))
print('%.0f seconds' % (time.time() - start))
```

```
parameters 229536
train loss 0.4920  validation loss 0.3461  perplexity 1.4
94 seconds
```

What it writes

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
```

```
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
```

```
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
torch.manual_seed(0)
```

```
model = GPT()
train(model, epochs=5)

prompt = encode('the model ').unsqueeze(0)
for temperature in [0.4, 0.8, 1.2]:
    torch.manual_seed(0)
    out = model.generate(prompt, 140, temperature=temperature)
    print('temperature %.1f:' % temperature)
    print('  ' + repr(decode(out[0])))
    print()
```

```
temperature 0.4:
  'the model updates the validation set across the batch, because the batch layer step, and
    ↪ the learning rate copies the validation set at every step, so'

temperature 0.8:
  'the model updates the validation set across the batch. the batch, and the learning rate
    ↪ copies the hidden state without a mask, because the attention '

temperature 1.2:
  'the model updates the validation state bilises the hidden state state without a mask, and
    ↪ dropout copies the input across the learning data after the '
```

Watch Out

It is memorising, and on a corpus this size it must

Roughly thirty thousand characters against a model with a few hundred thousand parameters. The generated text will contain stretches lifted verbatim from the corpus, which is worth seeing rather than hiding: it is the same phenomenon that makes large models reproduce their training data, at a scale where you can hold both ends of it in your head.
The gap between training and validation loss above is the measurement. Everything in weeks 4 and 11 about regularisation and scale is aimed at that gap.

Day 2 takeaway

Day 3 Context, Width and Depth

Scaling laws in miniature, on a corpus far too small

Three things decide how good a language model is, and they are not the things people usually adjust first.

Context length

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

```

```
    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
```

```
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
```

```
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
import math
results = []
for block in [8, 16, 32]:
    torch.manual_seed(0)
    Xa, Ya = windows(data[:split], block)
    Xb, Yb = windows(data[split:], block)
    loader = DataLoader(TensorDataset(Xa, Ya), batch_size=64, shuffle=True)
    model = GPT(block_size=block)
    opt = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    for _ in range(5):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1)).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(Xb).reshape(-1, VOCAB), Yb.reshape(-1)).item()
    results.append((block, val))

print('%10s %14s %12s' % ('context', 'val loss', 'perplexity'))
for block, val in results:
    print('%10d %14.4f %12.1f' % (block, val, math.exp(val)))
```

```
   context       val loss   perplexity
         8         0.7046          2.0
        16         0.4769          1.6
        32         0.3314          1.4
```

Width and depth

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
```

```
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y
```

```

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
```

```
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
import math
print('%8s %8s %10s %12s %12s'
      % ('width', 'blocks', 'params', 'train loss', 'val loss'))
for width, blocks in [(48, 2), (96, 2), (96, 4)]:
    torch.manual_seed(0)
    model = GPT(width=width, blocks=blocks)
    tr, va = train(model, epochs=5)
    print('%8d %8d %10d %12.4f %12.4f'
          % (width, blocks, sum(p.numel() for p in model.parameters()),
             tr, va))
```

```
   width   blocks     params   train loss     val loss
      48        2      59472       0.8091       0.5077
      96        2     229536       0.4920       0.3461
      96        4     453216       0.3993       0.3218
```

This is a scaling law, in miniature

Bigger models reach a lower training loss and, past a point, a worse validation loss, because the corpus is far too small for them. The published scaling laws say the same thing with the sign the other way round: given enough data, loss falls predictably as a power of model size and compute. Both statements are the same relationship read from different ends, and the practical version is that model size and dataset size have to grow together.

What the model has actually learned

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
```

```
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
```

```
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
```

```
    return loss.item(), val
torch.manual_seed(0)
model = GPT()
train(model, epochs=5)
model.eval()

for prompt in ['the gradient ', 'attention ', 'if the loss ']:
    idx = encode(prompt).unsqueeze(0)
    with torch.no_grad():
        probs = model(idx)[0, -1].softmax(-1)
    top = probs.topk(4)
    guesses = ', '.join('%r %.2f' % (itos[int(i)], v)
                        for v, i in zip(top.values, top.indices))
    print('%-16r → %s' % (prompt, guesses))
```

```
'the gradient '  → 's' 0.46, 'c' 0.24, 'n' 0.08, 'r' 0.07
'attention '     → 'w' 0.65, 's' 0.08, 'a' 0.05, 'r' 0.05
'if the loss '   → 'c' 0.77, 't' 0.10, 'u' 0.05, 'r' 0.03
```

Day 3 takeaway

Day 4 Evaluating a Language Model

Perplexity, why greedy decoding loops, and measuring memorisation

Evaluating a language model properly is harder than training one, and loss is only the first of three things to look at.

Loss, perplexity and bits per character

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
```

```
            'the weights', 'the hidden state', 'the input',
            'the residual path', 'the attention weights',
            'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
             'before the softmax', 'after the residual connection',
             'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
```

```
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
import math
torch.manual_seed(0)
model = GPT()
```

```
tr, va = train(model, epochs=5)

print('cross entropy loss   %.4f' % va)
print('perplexity           %.2f' % math.exp(va))
print('bits per character   %.3f' % (va / math.log(2)))
print()
print('for comparison:')
print('  uniform over %d characters is %.3f bits'
      % (VOCAB, math.log(VOCAB) / math.log(2)))
print('  good character models of English reach roughly 1.1 to 1.3 bits')
print('  on large corpora, which is the number to have in mind.')
```

```
cross entropy loss   0.3461
perplexity           1.41
bits per character   0.499

for comparison:
  uniform over 27 characters is 4.755 bits
  good character models of English reach roughly 1.1 to 1.3 bits
  on large corpora, which is the number to have in mind.
```

Loss does not tell you whether it can generate

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
```

```
                  % (pick(subject), pick(verb), pick(obj),
                     pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight
```

```

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
torch.manual_seed(0)
model = GPT()
train(model, epochs=5)

# A model can have a fine loss and still produce degenerate text if the
# sampling is wrong. Same weights, three settings.
prompt = encode('the loss ').unsqueeze(0)
for name, kwargs in [('greedy (temperature 0.01)', dict(temperature=0.01)),
                     ('temperature 0.8', dict(temperature=0.8)),
                     ('temperature 0.8, top-k 5',
                      dict(temperature=0.8, top_k=5)),
                     ('temperature 2.0', dict(temperature=2.0))]:
    torch.manual_seed(0)
```

```
    out = model.generate(prompt, 90, **kwargs)
    print('%-26s %r' % (name, decode(out[0])))
    print()
```

```
greedy (temperature 0.01)  'the loss the residual path once per epoch, so the learning rate
    ↪ copies the validation set across th'

temperature 0.8            'the loss the sequence after the residual connection. the learning
    ↪ rate predicts the attention weigh'

temperature 0.8, top-k 5   'the loss the sequence after the residual connection. the learning
    ↪ rate predicts the weights without'

temperature 2.0            'the loss updates the input aining data   once per hidden state
    ↪ state without a mask, and dl path du'
```

Watch Out

Greedy decoding loops, and it is not a bug in the model

Take the most likely character every time and a language model will eventually enter a cycle and repeat it forever. The reason is that the most likely continuation of a repeated phrase is more of the same phrase, and nothing in greedy decoding can escape. This is why every text generation interface samples rather than maximising, and why repetition penalties exist.

Measuring memorisation

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
```

```
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
```

```
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
torch.manual_seed(0)
model = GPT()
train(model, epochs=5)

def longest_copied(generated, source, minimum=8):
    best = ''
    for i in range(len(generated)):
```

```
        for j in range(i + minimum, len(generated) + 1):
            piece = generated[i:j]
            if piece in source and len(piece) > len(best):
                best = piece
            elif piece not in source:
                break
    return best

torch.manual_seed(0)
out = decode(model.generate(encode('the ').unsqueeze(0), 200,
                            temperature=0.8)[0])
copied = longest_copied(out, text)
print('generated %d characters' % len(out))
print('longest stretch also in the training text: %d characters'
      % len(copied))
print(repr(copied))
```

```
generated 204 characters
longest stretch also in the training text: 46 characters
' at every step, and the gradient controls the '
```

On a corpus this small the answer is most of it. On a real corpus the same measurement is how you find out whether your model is reproducing training data, which matters for licensing, for privacy, and for whether your evaluation set has leaked into training.

Day 4 takeaway

Day 5 Tokenisation

Byte pair encoding built by hand, and why your API bill counts tokens

Characters are the simplest choice and nobody uses them at scale. What real models use is subword tokens, and the reasons are practical rather than theoretical.

The problem with words and with characters

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
```

```
                'the encoder', 'the decoder', 'each layer',
                'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
words = text.split()
```

```
vocab_words = sorted(set(words))
print('characters: vocabulary %d, sequence length %d'
      % (VOCAB, len(text)))
print('words:      vocabulary %d, sequence length %d'
      % (len(vocab_words), len(words)))
print()
print('a word vocabulary is short sequences and a huge embedding table,')
print('and it has no answer at all for a word it has never seen.')
print('characters are the opposite: tiny table, very long sequences,')
print('and attention costs the square of the length.')
```

```
characters: vocabulary 27, sequence length 28539
words:      vocabulary 70, sequence length 4545

a word vocabulary is short sequences and a huge embedding table,
and it has no answer at all for a word it has never seen.
characters are the opposite: tiny table, very long sequences,
and attention costs the square of the length.
```

Byte pair encoding, built by hand

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
```

```
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
from collections import Counter

# Start from characters and repeatedly merge the commonest adjacent pair.
tokens = list(text[:900])
merges = []
for step in range(12):
    pairs = Counter(zip(tokens, tokens[1:]))
    if not pairs:
        break
    best, count = pairs.most_common(1)[0]
    merged = best[0] + best[1]
    merges.append((best, count))
    out, i = [], 0
    while i < len(tokens):
        if i < len(tokens) - 1 and (tokens[i], tokens[i + 1]) == best:
            out.append(merged)
            i += 2
        else:
            out.append(tokens[i])
```

```
            i += 1
    tokens = out

print('%-6s %-14s %8s' % ('merge', 'pair joined', 'count'))
for i, (pair, count) in enumerate(merges, 1):
    print('%-6d %-14r %8d' % (i, pair[0] + pair[1], count))
print('\n900 characters became %d tokens' % len(tokens))
print('first 20:', tokens[:20])
```

```
merge  pair joined       count
1      'e '                34
2      'th'                32
3      'the '              27
4      ' the '             26
5      'es'                21
6      'in'                19
7      'at'                17
8      'er'                16
9      'es the '           15
10     ' s'                15
11     'on'                13
12     'du'                12

900 characters became 653 tokens
first 20: ['e', 'a', 'c', 'h', ' ', 'l', 'a', 'y', 'er', ' ', 'c', 'o', 'p', 'i', 'es the ',
    ↪ 'w', 'e', 'i', 'g', 'h']
```

Common sequences become single tokens and rare ones stay as pieces, so the vocabulary is finite and nothing is ever completely unknown. That is the whole algorithm, and the tokenisers shipped with real models are this with a larger corpus and thirty thousand merges instead of twelve.

A real tokeniser

```
from tokenizers import Tokenizer, models, trainers, pre_tokenizers

text = ('the model reads the sequence and predicts the next token. '
        'training is a loop: forward, loss, backward, step. the gradient '
        'tells each weight which way to move and the optimiser decides '
        'how far. attention scores every position against every other '
        'position, and the scores become weights.') * 8

tok = Tokenizer(models.BPE(unk_token='[UNK]'))
tok.pre_tokenizer = pre_tokenizers.Whitespace()
trainer = trainers.BpeTrainer(vocab_size=120,
                              special_tokens=['[UNK]', '[PAD]'])
tok.train_from_iterator([text], trainer)

print('vocabulary size %d' % tok.get_vocab_size())
for s in ['the gradient', 'backpropagation', 'zqx']:
    enc = tok.encode(s)
    print('%-18r → %s' % (s, enc.tokens))
```

```
vocabulary size 120
'the gradient'     → ['the', 'gr', 'adien', 't']
```

```
'backpropagation'  → ['bac', 'k', 'p', 'r', 'op', 'ag', 'a', 'tion']
'zqx'              → ['[UNK]', 'q', 'x']
```

Why the token count on your API bill is not the word count

A common word is one token. An unusual one is three or four. A misspelling, a product code or a language the tokeniser saw little of can be one token per character. That is why the same paragraph costs different amounts in different languages, and why counting words to estimate cost is unreliable.

> **Day 5 takeaway**

Day 6 Packing, Compute and Budgets

Where the arithmetic actually goes in a training run

Everything about a language model that is not the architecture: how the data is packed, how long the context is, and what the compute is spent on.

Packing

```python
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
```

```
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
# Training on padded documents wastes compute on padding. Real training
# concatenates everything and cuts fixed-length windows out of the stream.
documents = [text[:120], text[200:260], text[400:700]]
lengths = [len(d) for d in documents]
print('document lengths', lengths)

padded = max(lengths) * len(documents)
packed = sum(lengths)
print('padded to a rectangle: %d positions, %d of them real (%.0f%%)'
      % (padded, packed, 100 * packed / padded))

stream = encode(''.join(documents))
block = 48
n_windows = len(stream) // block
print('packed into a stream: %d windows of %d, no padding at all'
      % (n_windows, block))
```

```
document lengths [120, 60, 300]
padded to a rectangle: 900 positions, 480 of them real (53%)
packed into a stream: 10 windows of 48, no padding at all
```

Watch Out

Packing lets one document see the end of another

A window cut from the concatenated stream can straddle a document boundary, so the model attends across it. Most implementations accept that, because the cost is small and the compute saving is large. If it matters for your data, insert a separator token and mask attention across it, which is what document-aware packing does.

Where the compute goes

```python
def flops_per_token(width, blocks, context, vocab):
    attn_proj = 4 * width * width
    attn_scores = 2 * context * width
    ff = 8 * width * width
    per_block = attn_proj + attn_scores + ff
    return 2 * (blocks * per_block + width * vocab)

print('%10s %10s %14s %14s' % ('context', 'width', 'per token', 'share attn'))
for context in [128, 1024, 8192]:
    for width in [768]:
        total = flops_per_token(width, 12, context, 50257)
        attn = 2 * 12 * 2 * context * width
        print('%10d %10d %14s %13.1f%%'
              % (context, width, '{:,}'.format(total),
                 100 * attn / total))
```

```
   context      width      per token     share attn
       128        768    251,782,656          1.9%
      1024        768    284,812,800         13.3%
      8192        768    549,053,952         55.0%
```

At a short context the attention scores are a rounding error and almost all the work is in the matrix multiplications, which is why transformers are efficient. As the context grows the quadratic term takes over, and that crossover is what all the work on efficient attention is chasing.

The three numbers that describe a training run

Number	What it is	Rule of thumb
Parameters	Weights in the model	Memory during training is roughly 16 bytes per parameter with Adam
Tokens	How much text it sees	Around 20 tokens per parameter is compute-optimal
Compute	Floating point operations	Roughly 6 times parameters times tokens for a full training run

```
def budget(params, tokens):
    flops = 6 * params * tokens
    memory = params * 16 / 1e9
    return flops, memory

print('%-16s %14s %14s %12s'
      % ('model', 'parameters', 'tokens', 'memory (GB)'))
for name, params in [('small', 124e6), ('medium', 355e6), ('large', 7e9)]:
    tokens = params * 20
    flops, memory = budget(params, tokens)
    print('%-16s %14s %14s %12.1f'
          % (name, '{:,.0f}'.format(params), '{:,.0f}'.format(tokens),
             memory))
    print('%-16s total compute about %.2e operations' % ('', flops))
```

```
model                parameters         tokens  memory (GB)
small               124,000,000  2,480,000,000          2.0
                total compute about 1.85e+18 operations
medium              355,000,000  7,100,000,000          5.7
                total compute about 1.51e+19 operations
large             7,000,000,000 140,000,000,000        112.0
                total compute about 5.88e+21 operations
```

Day 6 takeaway

Day 7 The Complete Model

Measured against every baseline, and an honest account of the distance

The complete model, trained with everything this week established, and measured against the baselines it has to beat.

The comparison

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
```

```
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
```

```
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
```

```
    return loss.item(), val
import math, time

# baseline 1: character frequencies
from collections import Counter
counts = Counter(text)
total = sum(counts.values())
unigram = -sum((n / total) * math.log(n / total) for n in counts.values())

# baseline 2: bigram counts
bi = torch.zeros(VOCAB, VOCAB)
for a, b in zip(data[:split][:-1], data[:split][1:]):
    bi[a, b] += 1
probs = (bi + 1) / (bi + 1).sum(1, keepdim=True)
val = data[split:]
bigram = -probs[val[:-1], val[1:]].log().mean().item()

print('%-28s %12s %12s' % ('', 'val loss', 'perplexity'))
print('%-28s %12.4f %12.1f'
      % ('uniform guessing', math.log(VOCAB), VOCAB))
print('%-28s %12.4f %12.1f' % ('character frequencies', unigram,
                               math.exp(unigram)))
print('%-28s %12.4f %12.1f' % ('bigram counts', bigram,
                               math.exp(bigram)))

for blocks in [1, 2]:
    torch.manual_seed(0)
    model = GPT(blocks=blocks)
    start = time.time()
    tr, va = train(model, epochs=5)
    print('%-28s %12.4f %12.1f'
          % ('transformer, %d blocks' % blocks, va, math.exp(va)))
```

```
                             val loss   perplexity
uniform guessing               3.2958         27.0
character frequencies          2.8138         16.7
bigram counts                  1.9837          7.3
transformer, 1 blocks          0.4943          1.6
transformer, 2 blocks          0.3461          1.4
```

Generating from the finished model

```
import torch

def build_corpus(seed=0, sentences=500):
    """A small language with real grammar, generated from a seed.

    Written into the page rather than downloaded, so the week runs
    offline and everybody trains on identical characters. It is
    regular enough that a model can genuinely learn it, and long
    enough that counting pairs of characters cannot."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
```

```
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    subject = ['the model', 'the network', 'the optimiser',
               'the gradient', 'the loss', 'attention',
               'the encoder', 'the decoder', 'each layer',
               'the batch', 'the learning rate', 'dropout']
    verb = ['reads', 'predicts', 'updates', 'normalises',
            'averages', 'scores', 'reduces', 'controls',
            'stabilises', 'copies']
    obj = ['the sequence', 'the next token', 'every position',
           'the weights', 'the hidden state', 'the input',
           'the residual path', 'the attention weights',
           'the training data', 'the validation set']
    tail = ['during training', 'at every step', 'in a single pass',
            'before the softmax', 'after the residual connection',
            'without a mask', 'across the batch', 'once per epoch']
    joiner = [', and ', ', so ', ', because ', '. ', '. ', '. ']
    out = []
    for _ in range(sentences):
        out.append('%s %s %s %s%s'
                   % (pick(subject), pick(verb), pick(obj),
                      pick(tail), pick(joiner)))
    return ''.join(out)

CORPUS = build_corpus()
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

text = CORPUS.strip()
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
itos = {i: c for c, i in stoi.items()}
VOCAB = len(chars)

def encode(s):
    return torch.tensor([stoi[c] for c in s], dtype=torch.long)

def decode(t):
    return ''.join(itos[int(i)] for i in t)

data = encode(text)
BLOCK_SIZE = 32

def windows(seq, block):
    """Every position is a training example: x is block characters,
    y is the same block shifted one to the right."""
    starts = range(0, len(seq) - block - 1, 4)     # stride 4
    x = torch.stack([seq[i:i + block] for i in starts])
    y = torch.stack([seq[i + 1:i + block + 1] for i in starts])
    return x, y

split = int(len(data) * 0.9)
X_tr, Y_tr = windows(data[:split], BLOCK_SIZE)
X_va, Y_va = windows(data[split:], BLOCK_SIZE)
```

```
train_loader = DataLoader(TensorDataset(X_tr, Y_tr), batch_size=64,
                          shuffle=True)
class GPT(nn.Module):
    def __init__(self, vocab=None, width=96, heads=4, blocks=2,
                 block_size=BLOCK_SIZE, dropout=0.1):
        super().__init__()
        vocab = vocab or VOCAB
        self.block_size = block_size
        self.tok = nn.Embedding(vocab, width)
        self.pos = nn.Embedding(block_size, width)
        self.drop = nn.Dropout(dropout)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=width * 4,
            dropout=dropout, batch_first=True, norm_first=True,
            activation='gelu')
        self.blocks = nn.TransformerEncoder(layer, num_layers=blocks)
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, vocab, bias=False)
        self.head.weight = self.tok.weight

    def forward(self, idx):
        T = idx.shape[1]
        mask = torch.triu(torch.ones(T, T, dtype=torch.bool), diagonal=1)
        h = self.drop(self.tok(idx) + self.pos(torch.arange(T)))
        return self.head(self.norm(self.blocks(h, mask=mask)))

    @torch.no_grad()
    def generate(self, idx, steps, temperature=1.0, top_k=None):
        self.eval()
        for _ in range(steps):
            window = idx[:, -self.block_size:]
            logits = self(window)[:, -1, :] / temperature
            if top_k:
                kth = logits.topk(min(top_k, logits.shape[-1])).values[:, -1:]
                logits = logits.masked_fill(logits < kth, float('-inf'))
            nxt = torch.multinomial(logits.softmax(-1), 1)
            idx = torch.cat([idx, nxt], dim=1)
        return idx

def train(model, epochs=5, lr=3e-3, warmup=100):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    loss_fn = nn.CrossEntropyLoss()
    step = 0
    for _ in range(epochs):
        model.train()
        for xb, yb in train_loader:
            step += 1
            for group in opt.param_groups:
                group['lr'] = lr * min(1.0, step / warmup)
            opt.zero_grad()
            loss = loss_fn(model(xb).reshape(-1, VOCAB), yb.reshape(-1))
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

```
            opt.step()
    model.eval()
    with torch.no_grad():
        val = loss_fn(model(X_va).reshape(-1, VOCAB),
                      Y_va.reshape(-1)).item()
    return loss.item(), val
torch.manual_seed(0)
model = GPT(blocks=2)
tr, va = train(model, epochs=5)
print('validation loss %.4f\n' % va)

for prompt in ['the model ', 'attention ', 'if the loss ']:
    torch.manual_seed(0)
    out = model.generate(encode(prompt).unsqueeze(0), 110,
                         temperature=0.7, top_k=8)
    print('%r →' % prompt)
    print('  ' + repr(decode(out[0])))
    print()
```

```
validation loss 0.3461

'the model ' →
  'the model updates the validation set across the batch. the batch, and the learning rate
    ↪ copies the hidden state without '

'attention ' →
  'attention weights once per epoch. the optimiser predicts the input at every step, and the
    ↪ learning rate updates the vali'

'if the loss ' →
  'if the loss copies the input without a mask. the model controls the batch. the loss across
    ↪ the batch. the batch, so the ba'
```

Watch Out

What this is and is not

A few hundred thousand parameters trained on thirty thousand characters of a deliberately regular invented language, for about a minute. It has learned that language well, which is why the perplexity is so low. It has not learned English, and the gap between those two statements is the whole of the rest of the field.

The distance from here to a useful model is entirely scale: more parameters, vastly more text, subword tokens, longer context and weeks of compute. Not a single architectural idea in this week changes. That is worth sitting with, because it is the actual reason the field looks the way it does, and it is why week 11 is about using somebody else's trained model rather than building your own.

The checklist

1. Compute the uniform and unigram losses first, so you know what beating nothing looks like.
2. Fit a bigram counting model as a real baseline.
3. Report perplexity or bits per character rather than raw loss.

4. Causal mask, and check the loss against the floor the task permits.
5. Warmup, gradient clipping, AdamW, tied embeddings.
6. Pack documents into a stream rather than padding.
7. Evaluate by generating, with the decoding settings recorded.
8. Measure the longest verbatim stretch your model reproduces.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. How is the target built for next-character prediction?

   a) It is the input shifted by one position, so every position predicts the character that follows it
   b) It is the input reversed
   c) It is the input with characters masked at random
   d) It is a separate labelled file

2. Why does week 10 begin with a bigram count model?

   a) It is the final model
   b) It is a cheap baseline the transformer has to beat, without which the transformer's loss is a number with no meaning
   c) It is needed to build the vocabulary
   d) It provides the positional encoding

3. What is perplexity?

   a) The entropy of the training corpus
   b) The accuracy of the next token
   c) The exponential of the cross entropy, readable as the effective number of choices the model is hesitating between
   d) The number of unique tokens

4. Why does greedy decoding loop?

   a) The temperature is too high
   b) The context is too long
   c) The model has too few parameters
   d) Always taking the single most likely token drives the model into a state whose most likely continuation returns to that state

5. What does byte pair encoding do?

   a) Starts from single characters and repeatedly merges the most frequent adjacent pair, ending with common sequences as single tokens

b) Compresses the text with a standard algorithm

c) Maps every byte to a fixed token

d) Assigns one token per word

6. Why does an API bill count tokens rather than characters?

   a) Characters are hard to count

   b) Because the model's compute is per token, and how many tokens your text becomes depends on the tokeniser rather than its length

   c) It is a pricing convention with no technical basis

   d) Tokens are always four characters

7. What is packing, in a training run?

   a) Batching by sequence length

   b) Compressing the checkpoint

   c) Filling every position of every sequence with real text rather than padding, so no compute is spent on filler

   d) Storing the data in a binary format

8. The final transformer was measured against the bigram baseline. What did the corpus size turn out to control?

   a) Only the training time

   b) The vocabulary size alone

   c) Nothing, the transformer won either way

   d) Everything: on a corpus of a few thousand characters the transformer lost to bigrams, and only on a much larger corpus did it win clearly

Answers:1a, 2b, 3c, 4d, 5a, 6b, 7c, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 11

Pretrained Transformers

Week 11

OBJECTIVES

Using models somebody else trained: frozen features, fine-tuning, LoRA, and how to decide between them.

Week 10 trained a language model from nothing and it learned exactly the invented language it was shown. This week does the opposite: start from a model trained on a great deal of real text and adapt it. It covers loading one properly, using it frozen, fine-tuning it at the much smaller learning rate that requires, and the low-rank methods that make fine-tuning large models affordable. The comparison at the end shows the answer depending almost entirely on how many labels you have.

Day 1 Why Not Train Your Own

The task, the baseline, and the case for starting from somebody else's weights

Week 10 trained a language model from nothing and produced something that had learned an invented language and nothing else. The alternative is to start from a model somebody has already trained on a great deal of real text, which is what almost everybody actually does.

The data

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
```

```
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
print('training reviews %d, balance %.3f'
      % (len(train_texts), train_y.float().mean()))
print()
for i in range(6):
    print('[%s] %s' % ('positive' if train_y[i] else 'negative',
                       train_texts[i]))
```

```
training reviews 1200, balance 0.518

[negative] the delivery is faulty.
[positive] the battery is sturdy.
[positive] Honestly, the motor is comfortable.
[negative] the screen is broken.
[negative] Honestly, the fabric is faulty in daily use.
[positive] On the whole the software is reliable overall.
```

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
```

```
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
print('reviews using the training vocabulary:')
for i in range(2):
    print('  [%s] %s' % ('positive' if seen_y[i] else 'negative',
                         seen_texts[i]))
print('\nreviews using words the training set never contained:')
for i in range(2):
    print('  [%s] %s' % ('positive' if new_y[i] else 'negative',
                         new_texts[i]))
print('\ntraining adjectives:', TRAIN_GOOD + TRAIN_BAD)
print('new adjectives     :', NEW_GOOD + NEW_BAD)
```

```
reviews using the training vocabulary:
  [positive] On the whole the delivery is sturdy.
  [positive] Honestly, the motor is sturdy so far.

reviews using words the training set never contained:
  [negative] Honestly, the delivery is defective.
  [positive] I found that the delivery is pleasant for the price.

training adjectives: ['excellent', 'reliable', 'comfortable', 'sturdy', 'broken', 'flimsy',
    ↪ 'noisy', 'faulty']
new adjectives     : ['superb', 'dependable', 'pleasant', 'robust', 'damaged', 'fragile',
    ↪ 'loud', 'defective']
```

A bag of words baseline

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
```

```
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

print('%-16s %14s %14s' % ('', 'same words', 'new words'))
for label, ngrams in [('unigrams', (1, 1)), ('1 to 2 grams', (1, 2))]:
    model = make_pipeline(TfidfVectorizer(ngram_range=ngrams),
                          LogisticRegression(max_iter=2000))
    model.fit(train_texts, train_y.numpy())
    print('%-16s %14.4f %14.4f'
          % (label, model.score(seen_texts, seen_y.numpy()),
             model.score(new_texts, new_y.numpy())))
```

```
                 same words      new words
unigrams             1.0000         0.5000
1 to 2 grams         1.0000         0.4925
```

Watch Out

The second column is the whole point of this week

A bag of words does well on reviews written with the words it was trained on, and falls to chance on reviews written with synonyms. It has not learned that a product can be good or bad. It has learned that the string *reliable* predicts one label, and nothing in the method could have done otherwise.

A model pretrained on a great deal of English already knows that *dependable* and *reliable* are used in similar places. That knowledge is what you are borrowing, and the rest of this week is about how to borrow it.

Day 1 takeaway

Day 2 Loading a Pretrained Model

Weights, tokeniser and config, and what tokenisation does to your text

A pretrained model arrives as three things: weights, a tokeniser, and a configuration. All three have to match, and the library keeps them together for exactly that reason.

Loading one

```
from transformers import AutoTokenizer, AutoModel
import torch

NAME = 'prajjwal1/bert-mini'
tok = AutoTokenizer.from_pretrained(NAME)
model = AutoModel.from_pretrained(NAME)

print('model      %s' % NAME)
print('parameters %s' % '{:,}'.format(sum(p.numel()
                                          for p in model.parameters())))
print('vocabulary %s' % '{:,}'.format(tok.vocab_size))
print('hidden size %d, layers %d, heads %d'
      % (model.config.hidden_size, model.config.num_hidden_layers,
         model.config.num_attention_heads))
```

```
model      prajjwal1/bert-mini
parameters 11,170,560
vocabulary 30,522
hidden size 256, layers 4, heads 4
```

Watch Out

The tokeniser is part of the model

The weights were trained against one specific vocabulary, one specific way of splitting words, and specific special tokens. Load weights from one model and a tokeniser from another and every token id means something different. Nothing raises. You get a model that runs and produces noise. Always load both from the same name, which is what `AutoTokenizer` and `AutoModel` exist to make easy.

What tokenisation actually does to your text

```
from transformers import AutoTokenizer

tok = AutoTokenizer.from_pretrained('prajjwal1/bert-mini')

for text in ['the battery is not reliable',
```

```
            'the packaging is disappointing',
            'unbelievably discombobulated']:
    ids = tok(text)['input_ids']
    print('%-34s → %d tokens' % (text, len(ids)))
    print('   %s' % tok.convert_ids_to_tokens(ids))
```

```
the battery is not reliable        → 7 tokens
   ['[CLS]', 'the', 'battery', 'is', 'not', 'reliable', '[SEP]']
the packaging is disappointing     → 6 tokens
   ['[CLS]', 'the', 'packaging', 'is', 'disappointing', '[SEP]']
unbelievably discombobulated       → 11 tokens
   ['[CLS]', 'un', '##bel', '##ie', '##va', '##bly', 'disco', '##mbo', '##bula', '##ted',
    ↪ '[SEP]']
```

Note the `[CLS]` and `[SEP]` tokens the tokeniser adds, and the `##` prefixes marking word continuations. A common word is one token; an unusual one is several pieces. Nothing is ever unknown, which is the property day 5 of week 10 built by hand.

Running it

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
```

```
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import AutoTokenizer, AutoModel

tok = AutoTokenizer.from_pretrained('prajjwal1/bert-mini')
model = AutoModel.from_pretrained('prajjwal1/bert-mini')
model.eval()

batch = tok(train_texts[:4], padding=True, truncation=True,
            max_length=32, return_tensors='pt')
print('input_ids     ', tuple(batch['input_ids'].shape))
print('attention_mask', tuple(batch['attention_mask'].shape))
print('\nthe mask marks real tokens, so padding is ignored:')
print(batch['attention_mask'])

with torch.no_grad():
    out = model(**batch)
print('\nlast_hidden_state', tuple(out.last_hidden_state.shape))
print('one vector per token per example.')
```

```
input_ids      (4, 9)
attention_mask (4, 9)

the mask marks real tokens, so padding is ignored:
tensor([[1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 0, 0]])

last_hidden_state (4, 9, 256)
one vector per token per example.
```

Day 2 takeaway

Day 3 Frozen Features

One forward pass, mean pooling under the mask, and a classifier on top

The cheapest way to use a pretrained model is to freeze it entirely and treat its output as features for something else. It takes one forward pass over your data and no training of the transformer at all.

Sentence vectors

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
```

```
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import AutoTokenizer, AutoModel

tok = AutoTokenizer.from_pretrained('prajjwal1/bert-mini')
bert = AutoModel.from_pretrained('prajjwal1/bert-mini')
bert.eval()

def embed(texts, how='mean', batch_size=64):
    out = []
    for i in range(0, len(texts), batch_size):
        batch = tok(texts[i:i + batch_size], padding=True,
                    truncation=True, max_length=32, return_tensors='pt')
        with torch.no_grad():
            hidden = bert(**batch).last_hidden_state
        if how == 'cls':
            out.append(hidden[:, 0, :])
        else:
            mask = batch['attention_mask'].unsqueeze(-1)
            out.append((hidden * mask).sum(1) / mask.sum(1))
    return torch.cat(out)

vectors = embed(train_texts[:200])
print('embeddings', tuple(vectors.shape))
print('one vector per review, of width %d' % vectors.shape[1])
```

```
embeddings (200, 256)
```

```
one vector per review, of width 256
```

Mean pooling beats the CLS token unless the model was trained for it

The `[CLS]` vector is only a sentence representation if the model was trained to make it one, which BERT was, for its next-sentence task, and most models since are not. Averaging the token vectors under the attention mask is the more reliable default, and it is what sentence embedding models do. Note the mask in that average: without it you are averaging in the padding.

Frozen features against the baseline

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import AutoTokenizer, AutoModel
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
import time

```

```
tok = AutoTokenizer.from_pretrained('prajjwal1/bert-mini')
bert = AutoModel.from_pretrained('prajjwal1/bert-mini')
bert.eval()

def embed(texts, batch_size=64):
    out = []
    for i in range(0, len(texts), batch_size):
        batch = tok(texts[i:i + batch_size], padding=True,
                    truncation=True, max_length=32, return_tensors='pt')
        with torch.no_grad():
            hidden = bert(**batch).last_hidden_state
        mask = batch['attention_mask'].unsqueeze(-1)
        out.append((hidden * mask).sum(1) / mask.sum(1))
    return torch.cat(out).numpy()

start = time.time()
Xtr, Xva = embed(train_texts), embed(new_texts)
embed_time = time.time() - start

print('evaluated on the vocabulary never seen in training')
print('%-30s %10s' % ('', 'accuracy'))
tfidf = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                      LogisticRegression(max_iter=2000))
tfidf.fit(train_texts, train_y.numpy())
print('%-30s %10.4f' % ('tf-idf 1 to 2 grams',
                        tfidf.score(new_texts, new_y.numpy())))

clf = LogisticRegression(max_iter=2000).fit(Xtr, train_y.numpy())
print('%-30s %10.4f' % ('frozen transformer features',
                        clf.score(Xva, new_y.numpy())))
print('\nembedding 1600 reviews took %.0f seconds' % embed_time)
```

```
evaluated on the vocabulary never seen in training
                                accuracy
tf-idf 1 to 2 grams               0.4925
frozen transformer features       0.8825

embedding 1600 reviews took 1 seconds
```

Day 3 takeaway

Day 4 Fine-tuning

The new head, and the learning rate that is a hundred times smaller

Fine-tuning updates the pretrained weights on your task. It costs considerably more than freezing and is usually worth it.

The classification head

```
from transformers import AutoModelForSequenceClassification
import torch

model = AutoModelForSequenceClassification.from_pretrained(
    'prajjwal1/bert-mini', num_labels=2)
print('the classifier that was added:')
print(' ', model.classifier)
print('\nit is randomly initialised, which is why the library warns you.')
print('everything below it carries the pretrained weights.')
total = sum(p.numel() for p in model.parameters())
head = sum(p.numel() for p in model.classifier.parameters())
print('\nparameters: %s total, %s in the new head'
      % ('{:,}'.format(total), '{:,}'.format(head)))
```

```
the classifier that was added:
  Linear(in_features=256, out_features=2, bias=True)

it is randomly initialised, which is why the library warns you.
everything below it carries the pretrained weights.

parameters: 11,171,074 total, 514 in the new head
```

Fine-tuning it

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
```

```
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from torch.utils.data import DataLoader, TensorDataset
from torch import nn
import time

tok = AutoTokenizer.from_pretrained('prajjwal1/bert-mini')

def encode(texts):
    b = tok(texts, padding='max_length', truncation=True, max_length=24,
            return_tensors='pt')
    return b['input_ids'], b['attention_mask']

ids_tr, mask_tr = encode(train_texts)
ids_va, mask_va = encode(new_texts)
loader = DataLoader(TensorDataset(ids_tr, mask_tr, train_y),
                    batch_size=32, shuffle=True)

torch.manual_seed(0)
model = AutoModelForSequenceClassification.from_pretrained(
    'prajjwal1/bert-mini', num_labels=2)
opt = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)
steps = 3 * len(loader)
sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=5e-5,
                                            total_steps=steps,
                                            pct_start=0.1)

def score():
    model.eval()
    with torch.no_grad():
        logits = model(input_ids=ids_va, attention_mask=mask_va).logits
    return (logits.argmax(1) == new_y).float().mean().item()

start = time.time()
for epoch in range(1, 4):
    model.train()
    for ids, mask, y in loader:
        opt.zero_grad()
        out = model(input_ids=ids, attention_mask=mask, labels=y)
        out.loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step()
        sched.step()
    print('epoch %d  accuracy %.4f' % (epoch, score()))
print('\n%.0f seconds on the CPU' % (time.time() - start))
```

```
epoch 1  accuracy 0.7525
epoch 2  accuracy 0.6975
epoch 3  accuracy 0.7200
```

```
25 seconds on the CPU
```

Watch Out

The learning rate is a hundred times smaller than you are used to

Fine-tuning uses rates around 1e-5 to 5e-5, not the 1e-3 that trains a model from scratch. The weights are already good, and the job is to adjust them slightly rather than to find them. A rate of 1e-3 on a pretrained transformer destroys the pretraining in the first few hundred steps, which shows up as a model that trains to roughly the accuracy you would get from random initialisation.
Warmup matters here too, for the reason week 9 gave: the classification head is random, and its early gradients are large.

Day 4 takeaway

Day 5 The Three Approaches Compared

Accuracy and cost side by side on the same data

The three approaches compared on the same data, with the costs, so the choice is a decision rather than a habit.

The comparison

```python
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
```

```
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import (AutoTokenizer, AutoModel,
                          AutoModelForSequenceClassification)
from torch.utils.data import DataLoader, TensorDataset
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
import time

NAME = 'prajjwal1/bert-mini'
tok = AutoTokenizer.from_pretrained(NAME)
results = []

# 1. bag of words
start = time.time()
m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
                  LogisticRegression(max_iter=2000))
m.fit(train_texts, train_y.numpy())
results.append(('tf-idf bigrams', m.score(new_texts, new_y.numpy()),
                time.time() - start))

# 2. frozen features
bert = AutoModel.from_pretrained(NAME).eval()
def embed(texts):
    out = []
    for i in range(0, len(texts), 64):
        b = tok(texts[i:i + 64], padding=True, truncation=True,
                max_length=24, return_tensors='pt')
        with torch.no_grad():
            h = bert(**b).last_hidden_state
        mask = b['attention_mask'].unsqueeze(-1)
        out.append((h * mask).sum(1) / mask.sum(1))
    return torch.cat(out).numpy()
start = time.time()
clf = LogisticRegression(max_iter=2000).fit(embed(train_texts),
                                            train_y.numpy())
results.append(('frozen features', clf.score(embed(new_texts),
                                             new_y.numpy()),
                time.time() - start))

# 3. fine-tuned
def encode(texts):
    b = tok(texts, padding='max_length', truncation=True, max_length=24,
```

```
            return_tensors='pt')
    return b['input_ids'], b['attention_mask']
ids_tr, mask_tr = encode(train_texts)
ids_va, mask_va = encode(new_texts)
torch.manual_seed(0)
model = AutoModelForSequenceClassification.from_pretrained(NAME,
                                                           num_labels=2)
opt = torch.optim.AdamW(model.parameters(), lr=5e-5)
loader = DataLoader(TensorDataset(ids_tr, mask_tr, train_y),
                    batch_size=32, shuffle=True)
start = time.time()
for _ in range(3):
    model.train()
    for ids, mask, y in loader:
        opt.zero_grad()
        model(input_ids=ids, attention_mask=mask, labels=y).loss.backward()
        opt.step()
model.eval()
with torch.no_grad():
    acc = (model(input_ids=ids_va,
                 attention_mask=mask_va).logits.argmax(1)
           == new_y).float().mean().item()
results.append(('fine-tuned', acc, time.time() - start))

print('all three evaluated on the new vocabulary')
print('%-22s %10s %10s' % ('', 'accuracy', 'seconds'))
for name, acc, secs in results:
    print('%-22s %10.4f %10.0f' % (name, acc, secs))
```

```
all three evaluated on the new vocabulary
                       accuracy    seconds
tf-idf bigrams           0.4925          0
frozen features          0.8825          2
fine-tuned               0.8100         24
```

Watch Out

Fine-tuning came out behind the frozen features

That is not the textbook ordering, and the reason is worth understanding. Fine-tuning adjusts the pretrained weights towards the training vocabulary, and the evaluation deliberately uses different words. Some of what it learns is *reliable means positive*, which is exactly the knowledge that does not transfer, and it partially overwrites the general knowledge that would have.

Freezing cannot do that, because the representation never moves. So when your evaluation differs from your training data in a way that matters, frozen features are more robust and fine-tuning is a way of specialising towards data you may not have. Run both. The ordering is not a law.

Approach	Cost	Use when
Bag of words	Seconds	Always, as the baseline. Often enough on its own
Frozen features	One forward pass	Few labels, many tasks over the same text, or no GPU
Fine-tune the head only	Minutes	Rarely worth it; either freeze everything or tune everything
Fine-tune everything	Minutes to hours	A few thousand labels and accuracy that justifies the cost
A hosted large model	An API bill	Very few labels, or a genuinely open-ended task

Day 5 takeaway

Day 6 Parameter Efficient Fine-tuning

What training memory really costs, and how LoRA avoids most of it

Fine-tuning every weight of a large model is expensive in memory as well as time. Two techniques make it affordable, and one of them has become the default.

What full fine-tuning costs

```
def memory_gb(params, optimiser='adamw'):
    weights = params * 4
    grads = params * 4
    state = params * 8 if optimiser == 'adamw' else 0
    return (weights + grads + state) / 1e9

print('%-16s %14s %14s %14s'
      % ('model', 'parameters', 'inference GB', 'training GB'))
for name, params in [('bert-tiny', 4.4e6), ('bert-base', 110e6),
                     ('7 billion', 7e9), ('70 billion', 70e9)]:
    print('%-16s %14s %14.2f %14.1f'
          % (name, '{:,.0f}'.format(params), params * 4 / 1e9,
             memory_gb(params)))
print('\ntraining needs about four times what inference needs, because')
print('the gradients and the optimiser state are each the size of the')
print('weights. That is what puts full fine-tuning out of reach.')
```

```
model                parameters   inference GB    training GB
bert-tiny             4,400,000           0.02            0.1
bert-base           110,000,000           0.44            1.8
7 billion         7,000,000,000          28.00          112.0
70 billion       70,000,000,000         280.00         1120.0

training needs about four times what inference needs, because
the gradients and the optimiser state are each the size of the
weights. That is what puts full fine-tuning out of reach.
```

Low rank adaptation

```
import torch
from torch import nn

class LoRALinear(nn.Module):
    def __init__(self, base, rank=8, alpha=16):
        super().__init__()
        self.base = base
        for prm in self.base.parameters():
            prm.requires_grad = False              # frozen
        d_in, d_out = base.in_features, base.out_features
        self.a = nn.Parameter(torch.randn(rank, d_in) * 0.01)
        self.b = nn.Parameter(torch.zeros(d_out, rank))
        self.scale = alpha / rank

    def forward(self, x):
        return self.base(x) + (x @ self.a.T @ self.b.T) * self.scale

torch.manual_seed(0)
base = nn.Linear(768, 768)
wrapped = LoRALinear(base, rank=8)

trainable = sum(p.numel() for p in wrapped.parameters()
                if p.requires_grad)
frozen = sum(p.numel() for p in wrapped.parameters()
             if not p.requires_grad)
print('frozen    %s' % '{:,}'.format(frozen))
print('trainable %s' % '{:,}'.format(trainable))
print('ratio     %.1f to 1' % (frozen / trainable))

x = torch.randn(2, 768)
print('\nb starts at zero, so at step 0 the wrapper is the identity:',
      bool(torch.allclose(wrapped(x), base(x))))
```

```
frozen    590,592
trainable 12,288
ratio     48.1 to 1

b starts at zero, so at step 0 the wrapper is the identity: True
```

Starting from zero is the detail that makes it work

One of the two matrices is initialised to zero, so the adapter contributes nothing at the first step and the model behaves exactly as the pretrained one did. Training then moves away from that point gradually. Initialise both randomly and you have corrupted the pretrained model before the first gradient arrives, which is the same mistake as unfreezing a backbone under a random head in week 6.

Adapters in practice

Method	Trainable share	Notes
Full fine-tuning	100%	Best results, largest cost
LoRA	Around 0.1 to 1%	The default now; adapters can be merged into the weights afterwards
QLoRA	Around 0.1%	LoRA on a 4-bit quantised base; fits large models on one card
Prompt tuning	A few thousand	Learns input vectors only; weaker but very cheap
Head only	Under 1%	Rarely competitive; usually beaten by frozen features plus a stronger classifier

Day 6 takeaway

Day 7 Choosing a Model

The families, the checklist, and how the answer moves with label count

Choosing a pretrained model, and the questions worth asking before you build anything on top of one.

What is actually on offer

Family	Shape	Good for
BERT and descendants	Encoder only	Classification, retrieval, token labelling
Sentence transformers	Encoder, trained for similarity	Search, clustering, deduplication
GPT family	Decoder only	Generation, and everything else now
T5 and BART	Encoder and decoder	Summarising, translation, structured rewriting
CLIP	Two encoders, images and text	Zero-shot image classification, search across both
Whisper	Encoder and decoder, audio in	Transcription

A checklist before you commit to one

1. **Licence.** Some weights forbid commercial use, and some are trained on data with its own restrictions. Check before you build, not after.
2. **Size against your hardware.** Four bytes per parameter for inference, roughly sixteen for full fine-tuning.
3. **The language and domain it was trained on.** A model trained on English web text is a poor starting point for Hindi clinical notes.
4. **Context length**, if your documents are long.
5. **Whether a smaller one is enough.** Measure it. A tiny model that is good enough is better than a large one that is marginally better.

6. **What it does when it is wrong.** A generative model produces confident text either way, which is a different failure shape from a classifier returning a probability.

Everything, on one task

```
import torch

# Two vocabularies. The model trains on one and is evaluated on both,
# which is what separates a model that learned words from a model that
# learned what the words mean.
TRAIN_GOOD = ['excellent', 'reliable', 'comfortable', 'sturdy']
TRAIN_BAD = ['broken', 'flimsy', 'noisy', 'faulty']
NEW_GOOD = ['superb', 'dependable', 'pleasant', 'robust']
NEW_BAD = ['damaged', 'fragile', 'loud', 'defective']

THING = ['the delivery', 'the battery', 'the screen', 'the handle',
         'the packaging', 'the software', 'the fabric', 'the motor']
OPENER = ['I found that ', 'Honestly, ', 'After a week, ', '',
          'On the whole ']
CLOSER = [' overall.', ' for the price.', ' so far.', '.',
          ' in daily use.']

def make_reviews(n, good, bad, seed):
    """Short product reviews, balanced, with the sentiment carried by
    one adjective."""
    g = torch.Generator().manual_seed(seed)
    def pick(xs):
        return xs[int(torch.randint(len(xs), (1,), generator=g))]
    texts, labels = [], []
    for _ in range(n):
        positive = int(torch.randint(2, (1,), generator=g))
        word = pick(good if positive else bad)
        texts.append(pick(OPENER) + pick(THING) + ' is ' + word
                     + pick(CLOSER))
        labels.append(positive)
    return texts, torch.tensor(labels)

train_texts, train_y = make_reviews(1200, TRAIN_GOOD, TRAIN_BAD, 0)
seen_texts, seen_y = make_reviews(400, TRAIN_GOOD, TRAIN_BAD, 1)
new_texts, new_y = make_reviews(400, NEW_GOOD, NEW_BAD, 2)
from transformers import (AutoTokenizer, AutoModel,
                          AutoModelForSequenceClassification)
from torch.utils.data import DataLoader, TensorDataset
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline

NAME = 'prajjwal1/bert-mini'
tok = AutoTokenizer.from_pretrained(NAME)

# how the three approaches behave as labels get scarce
def tfidf_score(n):
    m = make_pipeline(TfidfVectorizer(ngram_range=(1, 2)),
```

```
                      LogisticRegression(max_iter=2000))
    m.fit(train_texts[:n], train_y[:n].numpy())
    return m.score(new_texts, new_y.numpy())

bert = AutoModel.from_pretrained(NAME).eval()
def embed(texts):
    out = []
    for i in range(0, len(texts), 64):
        b = tok(texts[i:i + 64], padding=True, truncation=True,
                max_length=24, return_tensors='pt')
        with torch.no_grad():
            h = bert(**b).last_hidden_state
        mask = b['attention_mask'].unsqueeze(-1)
        out.append((h * mask).sum(1) / mask.sum(1))
    return torch.cat(out).numpy()
Etr, Eva = embed(train_texts), embed(new_texts)

def frozen_score(n):
    clf = LogisticRegression(max_iter=2000).fit(Etr[:n],
                                                train_y[:n].numpy())
    return clf.score(Eva, new_y.numpy())

def encode(texts):
    b = tok(texts, padding='max_length', truncation=True, max_length=24,
            return_tensors='pt')
    return b['input_ids'], b['attention_mask']
ids_va, mask_va = encode(new_texts)

def finetune_score(n, epochs=3):
    ids, mask = encode(train_texts[:n])
    torch.manual_seed(0)
    model = AutoModelForSequenceClassification.from_pretrained(
        NAME, num_labels=2)
    opt = torch.optim.AdamW(model.parameters(), lr=5e-5)
    loader = DataLoader(TensorDataset(ids, mask, train_y[:n]),
                        batch_size=16, shuffle=True)
    for _ in range(epochs):
        model.train()
        for a, b_, y in loader:
            opt.zero_grad()
            model(input_ids=a, attention_mask=b_,
                  labels=y).loss.backward()
            opt.step()
    model.eval()
    with torch.no_grad():
        return (model(input_ids=ids_va,
                      attention_mask=mask_va).logits.argmax(1)
                == new_y).float().mean().item()

print('%10s %14s %16s %14s'
      % ('labels', 'tf-idf', 'frozen bert', 'fine-tuned'))
for n in [50, 200, 1200]:
    print('%10d %14.4f %16.4f %14.4f'
          % (n, tfidf_score(n), frozen_score(n), finetune_score(n)))
```

labels	tf-idf	frozen bert	fine-tuned
50	0.4725	0.8300	0.5325
200	0.5175	0.8925	0.8625
1200	0.4925	0.8825	0.7475

The label count is what decides this

That table is the whole argument for pretraining in one place. With very few labels the pretrained model has an advantage, because it already knows what words mean and only has to learn the task. With plenty of labels a bag of words catches up, because the task is simple enough to learn from the data alone.

Run this experiment on your own problem before deciding. The answer moves with the number of labels, the difficulty of the task and how far your text is from what the model was pretrained on.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What is the case for starting from someone else's weights?

   a) They need no fine-tuning
   b) They avoid the need for a validation set
   c) They are smaller
   d) The pretraining already paid for a general model of the language, which is compute and data you almost certainly do not have

2. What three things must you load together for a pretrained model?

   a) Weights, tokeniser and config, because the wrong tokeniser turns your text into ids the weights were never fitted to
   b) Weights, data and labels
   c) Model, mask and embedding
   d) Weights, optimiser and scheduler

3. What is mean pooling under the mask?

   a) Averaging over the batch
   b) Averaging token vectors over real positions only, so padding does not drag the sentence representation towards zero
   c) Taking the maximum over positions
   d) Using the first token's vector

4. Fine-tuning uses a learning rate about a hundred times smaller than training from scratch. Why?

   a) To avoid gradient clipping

b) To save memory

c) Because the weights are already good, and a large step destroys what pretraining bought before the new head has learned anything

d) Because the batch size is smaller

5. Week 11 compared TF-IDF against frozen transformer features on a task whose validation vocabulary was disjoint from training. What happened?

   a) Both fell to chance

   b) The transformer won on seen vocabulary only

   c) TF-IDF won both

   d) TF-IDF scored perfectly on seen vocabulary and fell to chance on the unseen split, while the frozen transformer held up

6. Why does that result matter for how you evaluate?

   a) Because a validation split that shares vocabulary with training flatters a bag-of-words model and hides exactly the failure you care about

   b) Because TF-IDF should never be used

   c) Because transformers always win

   d) It does not, it is a quirk of the dataset

7. What does LoRA avoid paying for?

   a) The forward pass

   b) The optimiser state and gradients for every weight, by training small low-rank matrices alongside frozen ones

   c) Tokenisation

   d) The validation loop

8. How does the choice between frozen features and fine-tuning move with the number of labels?

   a) Frozen features are always better

   b) It does not move

   c) Frozen features are the sensible choice when labels are few, and fine-tuning starts to pay once there are enough of them

   d) Fine-tuning is always better

Answers:1d, 2a, 3b, 4c, 5d, 6a, 7b, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 12

Autoencoders and Variational Autoencoders

Week 12

OBJECTIVES

Learning representations with no labels at all, and turning a decoder into something you can sample from.

Every model so far has needed labels. An autoencoder needs none: it rebuilds its own input through a narrow middle, and the representation it learns on the way is useful for anomaly detection, denoising and as features for something else. Adding a distribution and one extra loss term turns it into a variational autoencoder, which can generate images from nothing but noise. The week ends by being clear about what these models are good at, which is not compression, and why their samples come out soft.

Day 1 Learning Without Labels

Rebuilding the input through a bottleneck, and why the bottleneck matters

Everything so far has needed labels. An autoencoder needs none: it is trained to reproduce its own input through a narrow middle, and what it learns on the way is a compressed representation of the data.

The idea

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
```

```
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
model = AutoEncoder(latent=16)
x = next(iter(val_loader))[0][:4]
code = model.encoder(x)
print('input   ', tuple(x.shape), '= %d numbers per image' % (28 * 28))
print('code    ', tuple(code.shape), '= 16 numbers per image')
print('output  ', tuple(model(x).shape))
print('\ncompression %.0f to 1' % (784 / 16))
print('parameters %d' % sum(p.numel() for p in model.parameters()))
```

```
input    (4, 1, 28, 28) = 784 numbers per image
code     (4, 16) = 16 numbers per image
output   (4, 784)

compression 49 to 1
parameters 437664
```

Training it, with no labels anywhere

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
```

```
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
model = AutoEncoder(latent=16)
print('validation reconstruction error %.5f' % train_ae(model))

x, _ = next(iter(val_loader))
with torch.no_grad():
    out = model(x[:1])
print('\noriginal:')
show(x[0])
print('\nrebuilt from 16 numbers:')
show(out[0])
```

```
validation reconstruction error 0.02912

original:




      %@@@@@#########*:
            : :::: @@=
                  @@-
               :@@:
               %@:
             :@@:
             #@%.
           .%@=
          :@@@:
          =@%

rebuilt from 16 numbers:




    .:-+*##+#*##%@%%*:
    .::=***+==:-:-+%@%:
        ..       .+#%=
                .=%#-
               .+%%-
               *@%.
             .%@@:
            -*#*:
          .=*#*-
```

Note what the loss function never saw

The target is the input. There is no label in that training loop at all, which means it will run on any pile of unlabelled data you have. That is the whole appeal, and it is the thread running through this week and the next two: labels are expensive, raw data is not, and a great deal can

be learned from raw data alone.

How narrow can the middle be?

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
```

```
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
print('%8s %10s %16s' % ('latent', 'params', 'reconstruction'))
for latent in [2, 8, 32, 128]:
    torch.manual_seed(0)
    model = AutoEncoder(latent=latent)
    err = train_ae(model, epochs=6)
    print('%8d %10d %16.5f'
          % (latent, sum(p.numel() for p in model.parameters()), err))
```

```
  latent     params   reconstruction
       2     435858          0.05140
       8     436632          0.03442
      32     439728          0.03465
     128     452112          0.03403
```

The error falls sharply from two dimensions to eight and then stops moving. Below eight the bottleneck is genuinely the constraint; above it, something else is, and on this architecture and this budget the extra width buys nothing. Finding that knee is worth doing once, because everything past it is capacity you are paying for and not using.

Watch Out

A bottleneck that is not a bottleneck teaches nothing

Widen the middle until it is as large as the input and the model can learn the identity function, which reconstructs perfectly and has learned nothing whatsoever about the data. The constraint is not an inconvenience to be minimised, it is the entire mechanism. If your reconstruction error is suspiciously low, check that the code is actually narrower than the input once you count every channel.

Day 1 takeaway

Day 2 Denoising, Convolutions and Anomalies

Three variations on one loop, each solving a different problem

Three variations, each a small change to the same loop, and each solving a different problem.

Denoising

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
```

```

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
model = AutoEncoder(latent=32)
```

```
# input: a corrupted image. target: the clean one.
train_ae(model, epochs=8, noise=0.4)

x, _ = next(iter(val_loader))
torch.manual_seed(1)
noisy = (x[:1] + 0.4 * torch.randn_like(x[:1])).clamp(0, 1)
with torch.no_grad():
    cleaned = model(noisy)
print('noisy input:')
show(noisy[0])
print('\ndenoised:')
show(cleaned[0])
print('\nclean original:')
show(x[0])
```

```
noisy input:
          .   * .  : -  -=
 -  *         *       .    :
- +- .: @  # # .: .=  :   ::
.*=     -=:*    *  + = - -#
*   =+@@@@@-@ =%%@@+-.  .  =
-   .=.   ..    *:+@@@ :=  @
    :   ..++ :   =@:  * -  -
 #      .        @@  .   . :
 ... -# -  * .  @@=-=  :
=::: .. :+   ::=@#  ..  . :+
  : =-     .-@@@ :.. = + -+
     +      =+@. @-= %.-= =
 =:  :    *@%@ :  -.     .
 *  :*  * @@% -=:    .  = :#

denoised:



             ...
     ::=+++*%@%@%@%%#-.
     .:==*##+=-=*%%#-.
        .::.   :*%#=.
          ....:+%@+
              =%%+.
            =%%+:
           :##+-
          :=*#*+:
         .-=+#+:
          ..:-:

clean original:



      %@@@@@#########*:
            : :::: @@=
                 @@-
               :@@:
              %@:
```

```
          :@@:
         #@%.
       .%@=
      :@@@:
     =@%
```

One line changed, and the model is now doing something else

The architecture is identical. The only difference is what sits on each side of the loss: corrupted input, clean target. That substitution is the basis of a good deal of modern self-supervised learning. Hide part of the input, train the model to restore it, and the representation you get for free is useful for tasks you have not thought of yet. Week 14 builds on exactly this.

Convolutional

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1), nn.ReLU(),  # 14
            nn.Conv2d(16, 32, 3, stride=2, padding=1), nn.ReLU())  # 7
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1,
                               output_padding=1), nn.ReLU(),        # 14
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1,
                               output_padding=1), nn.Sigmoid())     # 28

    def forward(self, x):
        return self.decoder(self.encoder(x))

torch.manual_seed(0)
model = ConvAutoEncoder()
```

```
x, _ = next(iter(val_loader))
print('input ', tuple(x[:2].shape))
print('code  ', tuple(model.encoder(x[:2]).shape))
print('output', tuple(model(x[:2]).shape))
print('parameters %d' % sum(p.numel() for p in model.parameters()))

opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss_fn = nn.MSELoss()
for _ in range(6):
    model.train()
    for xb, _ in train_loader:
        opt.zero_grad()
        loss_fn(model(xb), xb).backward()
        opt.step()
model.eval()
with torch.no_grad():
    err = sum(loss_fn(model(xb), xb).item() * len(xb)
              for xb, _ in val_loader) / len(val_set)
print('\nreconstruction error %.5f' % err)
```

```
input  (2, 1, 28, 28)
code   (2, 32, 7, 7)
output (2, 1, 28, 28)
parameters 9569

reconstruction error 0.00229
```

`ConvTranspose2d` is the upsampling counterpart of a strided convolution. The `output_padding` argument exists because a stride of 2 maps several input sizes to the same output size, so the reverse operation is ambiguous and you have to say which one you meant. Getting it wrong is the usual reason a decoder produces a 27 by 27 image.

Anomaly detection from reconstruction error

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
```

```
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
# Train on fours and nines only, then show it every digit.
targets = train_all.targets[:8000]
keep = ((targets == 4) | (targets == 9)).nonzero().flatten().tolist()
narrow = Subset(train_all, keep)
print('training on %d images of fours and nines only' % len(narrow))

model = AutoEncoder(latent=8)
train_ae(model, epochs=10,
         loader=DataLoader(narrow, batch_size=128, shuffle=True))

model.eval()
errors = {}
with torch.no_grad():
    for xb, yb in val_loader:
        per = ((model(xb) - xb.flatten(1)) ** 2).mean(1)
        for digit, e in zip(yb.tolist(), per.tolist()):
```

```
            errors.setdefault(digit, []).append(e)

print('\n%8s %14s %10s' % ('digit', 'mean error', 'seen?'))
for digit in range(10):
    e = sum(errors[digit]) / len(errors[digit])
    print('%8d %14.5f %10s'
          % (digit, e, 'yes' if digit in (4, 9) else 'no'))
```

```
training on 1594 images of fours and nines only

   digit     mean error      seen?
       0        0.10658         no
       1        0.05656         no
       2        0.09327         no
       3        0.08882         no
       4        0.04909        yes
       5        0.07873         no
       6        0.07898         no
       7        0.05782         no
       8        0.07786         no
       9        0.04494        yes
```

Day 2 takeaway

Day 3 The Latent Space Problem

Why you cannot sample from a plain autoencoder

The latent space of a plain autoencoder is useful for compression and useless for generation, and the reason is worth seeing before the fix.

What the code actually looks like

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
```

```
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
model = AutoEncoder(latent=2)          # two numbers, so we can print them
train_ae(model, epochs=10)

model.eval()
codes, labels = [], []
with torch.no_grad():
    for xb, yb in val_loader:
        codes.append(model.encoder(xb))
        labels.append(yb)
codes = torch.cat(codes)
labels = torch.cat(labels)

print('latent range: x %.2f to %.2f, y %.2f to %.2f'
      % (codes[:, 0].min(), codes[:, 0].max(),
         codes[:, 1].min(), codes[:, 1].max()))
```

```
print('\naverage position of each digit:')
for digit in range(10):
    mean = codes[labels == digit].mean(0)
    print('  %d  (%7.2f, %7.2f)' % (digit, mean[0], mean[1]))
```

```
latent range: x -11.54 to 9.20, y -16.03 to 2.54

average position of each digit:
  0  (  -5.48,   -2.98)
  1  (   4.38,   -7.70)
  2  (  -2.20,   -4.17)
  3  (  -2.05,   -2.76)
  4  (  -1.59,   -5.16)
  5  (  -2.19,   -4.33)
  6  (  -2.61,   -4.03)
  7  (  -0.77,   -6.77)
  8  (  -1.76,   -4.91)
  9  (  -1.33,   -5.84)
```

Watch Out

There is no reason for the space between clusters to mean anything

Nothing in the loss asked the codes to be arranged sensibly. It asked only that each one decodes back to its own image. The clusters end up wherever the optimiser put them, separated by regions no training image ever occupied, and a decoder given a point in one of those gaps has never been asked to produce anything meaningful there.

Decoding a point nobody trained on

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
torch.manual_seed(0)
model = AutoEncoder(latent=2)
train_ae(model, epochs=10)
model.eval()

with torch.no_grad():
    codes = torch.cat([model.encoder(xb) for xb, _ in val_loader])
    # a point midway between two real codes
    a, b = codes[0], codes[1]
    for name, point in [('a real code', a),
                        ('halfway to another', (a + b) / 2),
                        ('a random point', torch.randn(2) * codes.std(0)
                         + codes.mean(0))]:
        img = model.decoder(point.unsqueeze(0))
        print('%s:' % name)
        show(img[0], step=3)
        print()
```

```
a real code:
```

```
              .:::::::..
       ::=*%%%*====*##+-.
       .:+*+-:. .+#%*-.
       .::-====+#@@*:
          ...:-#@%=
          ..:=#*=-.
          .:-==-:.


halfway to another:


            .:--=--:::..
      ..:=*#*+---=**+-.
       .-+*=-:..:+*#=:.
       .-=++****#@@#=
         ..:---:=*@%+.
          ...::=+**+:
             .:===-:.


a random point:


         .::-=+#*+-:..
       ........:===+-.
            ..-+%%+-.
            :-=%@*-.
           .::-----      .
          .....-+#+-:-:.
              .:..
```

Day 3 takeaway

Day 4 The Variational Autoencoder

Encoding a distribution, the reparameterisation trick, and the KL term

A variational autoencoder makes two changes. The encoder outputs a distribution rather than a point, and the loss adds a term pulling those distributions towards a standard normal. Together they make the latent space something you can sample from.

The reparameterisation trick

```python
import torch

torch.manual_seed(0)
mu = torch.tensor([1.0], requires_grad=True)
log_var = torch.tensor([0.0], requires_grad=True)

# the wrong way: sample, then try to differentiate
z_bad = torch.normal(mu.detach(), torch.ones(1))
```

```
print('sampling directly gives a tensor with no graph:',
      z_bad.grad_fn is None)

# the right way
eps = torch.randn(1)
z = mu + (0.5 * log_var).exp() * eps
z.sum().backward()
print('reparameterised, the gradient reaches mu:  %.4f' % mu.grad.item())
print('and log_var:                               %.4f'
      % log_var.grad.item())
print('\nthe randomness is in eps, which has no parameters to learn.')
```

```
sampling directly gives a tensor with no graph: True
reparameterised, the gradient reaches mu:  1.0000
and log_var:                               -0.1467

the randomness is in eps, which has no parameters to learn.
```

The two-part loss

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class VAE(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.body = nn.Sequential(nn.Flatten(),
                                  nn.Linear(784, 256), nn.ReLU(),
                                  nn.Linear(256, 128), nn.ReLU())
        self.to_mu = nn.Linear(128, latent)
        self.to_log_var = nn.Linear(128, latent)
        self.decoder = nn.Sequential(
            nn.Linear(latent, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())
```

```
    def encode(self, x):
        h = self.body(x)
        return self.to_mu(h), self.to_log_var(h)

    def forward(self, x):
        mu, log_var = self.encode(x)
        z = mu + (0.5 * log_var).exp() * torch.randn_like(mu)
        return self.decoder(z), mu, log_var

def vae_loss(out, target, mu, log_var, beta=1.0):
    # how well it rebuilt the input, summed over pixels
    recon = nn.functional.binary_cross_entropy(out, target,
                                               reduction='sum')
    # how far each latent distribution is from a standard normal
    kl = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    return recon + beta * kl, recon, kl

torch.manual_seed(0)
model = VAE()
x, _ = next(iter(val_loader))
out, mu, log_var = model(x[:8])
total, recon, kl = vae_loss(out, x[:8].flatten(1), mu, log_var)
print('reconstruction term %.1f' % recon.item())
print('kl term             %.1f' % kl.item())
print('total               %.1f' % total.item())
```

```
reconstruction term 4364.9
kl term             0.2
total               4365.1
```

What the second term is doing

The reconstruction term wants each image to have its own distinct code, as far from the others as possible, because that is easiest to decode. The KL term wants every code to be a standard normal, which would make them all identical and useless. Training balances the two, and the result is a latent space that is spread out enough to decode and packed tightly enough that the region between two codes is also somewhere the decoder has learned about.

Day 4 takeaway

Day 5 Sampling and Interpolating

Generating digits from noise, and walking between two of them

Trained, sampled, and interpolated, which is the test a plain autoencoder failed yesterday.

Training

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
import time

class VAE(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.body = nn.Sequential(nn.Flatten(),
                                  nn.Linear(784, 256), nn.ReLU(),
                                  nn.Linear(256, 128), nn.ReLU())
        self.to_mu = nn.Linear(128, latent)
        self.to_log_var = nn.Linear(128, latent)
        self.decoder = nn.Sequential(
            nn.Linear(latent, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())
        self.latent = latent

    def encode(self, x):
        h = self.body(x)
        return self.to_mu(h), self.to_log_var(h)

    def forward(self, x):
        mu, log_var = self.encode(x)
        z = mu + (0.5 * log_var).exp() * torch.randn_like(mu)
        return self.decoder(z), mu, log_var

def train_vae(model, epochs=10, beta=1.0):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    for epoch in range(epochs):
        model.train()
        for xb, _ in train_loader:
            target = xb.flatten(1)
            out, mu, log_var = model(xb)
            recon = nn.functional.binary_cross_entropy(out, target,
```

```
                                                    reduction='sum')
            kl = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
            opt.zero_grad()
            ((recon + beta * kl) / len(xb)).backward()
            opt.step()
    return model

torch.manual_seed(0)
start = time.time()
model = train_vae(VAE())
model.eval()

with torch.no_grad():
    total = kl_total = seen = 0
    for xb, _ in val_loader:
        out, mu, log_var = model(xb)
        total += nn.functional.binary_cross_entropy(
            out, xb.flatten(1), reduction='sum').item()
        kl_total += (-0.5 * torch.sum(
            1 + log_var - mu.pow(2) - log_var.exp())).item()
        seen += len(xb)
print('per image: reconstruction %.1f, kl %.1f'
      % (total / seen, kl_total / seen))
print('trained in %.0f seconds' % (time.time() - start))
```

```
per image: reconstruction 129.0, kl 12.1
trained in 7 seconds
```

Sampling from nothing but noise

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class VAE(nn.Module):
    def __init__(self, latent=16):
```

```
        super().__init__()
        self.body = nn.Sequential(nn.Flatten(),
                                  nn.Linear(784, 256), nn.ReLU(),
                                  nn.Linear(256, 128), nn.ReLU())
        self.to_mu = nn.Linear(128, latent)
        self.to_log_var = nn.Linear(128, latent)
        self.decoder = nn.Sequential(
            nn.Linear(latent, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())
        self.latent = latent
    def encode(self, x):
        h = self.body(x)
        return self.to_mu(h), self.to_log_var(h)
    def forward(self, x):
        mu, log_var = self.encode(x)
        z = mu + (0.5 * log_var).exp() * torch.randn_like(mu)
        return self.decoder(z), mu, log_var

torch.manual_seed(0)
model = VAE()
opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
for _ in range(10):
    model.train()
    for xb, _ in train_loader:
        out, mu, log_var = model(xb)
        recon = nn.functional.binary_cross_entropy(
            out, xb.flatten(1), reduction='sum')
        kl = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
        opt.zero_grad()
        ((recon + kl) / len(xb)).backward()
        opt.step()
model.eval()

torch.manual_seed(3)
with torch.no_grad():
    for i in range(2):
        z = torch.randn(1, model.latent)
        print('sample %d, from a random point in the latent space:' % (i + 1))
        show(model.decoder(z)[0], step=2)
        print()
```

```
sample 1, from a random point in the latent space:



       ..::------:::.:...
     ....:::-=+*****-:::.
         ..:-+#***+=:...
         .:-+*****-:.
         .-+#%%%#+-..
         .-=+#@@#+:..
       ...::-=*@@*-.
       .:::::-=+*=:.
        ...::---:..
```

```
      .........


sample 2, from a random point in the latent space:



             .....
           :-+#%#*=:
          .=++*+=--.
          -=---==-.
          ..:-+**-.
             .=#+.
            .=@*.
           :#@*:
          -#@*:
         :+*=:
```

No image went in

The only input was a draw from a standard normal. The decoder has learned a map from that distribution to something resembling a digit, which is what makes this a generative model rather than a compressor. A plain autoencoder cannot do this, because nothing ever told it what the distribution of its codes should be.

Interpolating between two digits

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class VAE(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.body = nn.Sequential(nn.Flatten(),
                                  nn.Linear(784, 256), nn.ReLU(),
                                  nn.Linear(256, 128), nn.ReLU())
```

```
        self.to_mu = nn.Linear(128, latent)
        self.to_log_var = nn.Linear(128, latent)
        self.decoder = nn.Sequential(
            nn.Linear(latent, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())
    def encode(self, x):
        h = self.body(x)
        return self.to_mu(h), self.to_log_var(h)
    def forward(self, x):
        mu, log_var = self.encode(x)
        z = mu + (0.5 * log_var).exp() * torch.randn_like(mu)
        return self.decoder(z), mu, log_var

torch.manual_seed(0)
model = VAE()
opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
for _ in range(10):
    model.train()
    for xb, _ in train_loader:
        out, mu, log_var = model(xb)
        recon = nn.functional.binary_cross_entropy(
            out, xb.flatten(1), reduction='sum')
        kl = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
        opt.zero_grad()
        ((recon + kl) / len(xb)).backward()
        opt.step()
model.eval()

x, y = next(iter(val_loader))
with torch.no_grad():
    mu, _ = model.encode(x[:2])
print('interpolating from a %d to a %d:\n' % (y[0], y[1]))
with torch.no_grad():
    for t in [0.0, 0.5, 1.0]:
        z = (1 - t) * mu[0] + t * mu[1]
        print('t = %.1f' % t)
        show(model.decoder(z.unsqueeze(0))[0], step=3)
        print()
```

```
interpolating from a 7 to a 2:

t = 0.0



    .=#%@@@@@@@@@@@@@*:
       .:::..    .-#@@=.
                 =%@%-
               :%@%-
             .*%%+.
           .=***-


t = 0.5
```

```
    .-+###*=--::....
  .:*%%+=-:::=*%@%*-.
         .:-=*#*+=-:.
         ..---==**=-.
           .::=+=-:...
          .:::---:-::..
     ....:------:......
                .

t = 1.0

          .:-+#%##+-
        :+*#*#*+=-:
              .=*+-.
            :=#%#=:
           .----:---..
        ..:-------=+++#+=-
        :-=+###*#*+===-:..
```

> **Day 5 takeaway**

Day 6 Posterior Collapse and Blur

The beta trade-off, and why the samples are soft

Two things that go wrong with VAEs, both well known, and the parameter that trades between them.

Posterior collapse

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
```

```
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class VAE(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.body = nn.Sequential(nn.Flatten(),
                                  nn.Linear(784, 256), nn.ReLU())
        self.to_mu = nn.Linear(256, latent)
        self.to_log_var = nn.Linear(256, latent)
        self.decoder = nn.Sequential(
            nn.Linear(latent, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())
    def encode(self, x):
        h = self.body(x)
        return self.to_mu(h), self.to_log_var(h)
    def forward(self, x):
        mu, log_var = self.encode(x)
        z = mu + (0.5 * log_var).exp() * torch.randn_like(mu)
        return self.decoder(z), mu, log_var

print('%8s %14s %12s %16s'
      % ('beta', 'reconstruction', 'kl', 'active dims'))
for beta in [0.1, 1.0, 8.0, 60.0]:
    torch.manual_seed(0)
    model = VAE()
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    for _ in range(6):
        model.train()
        for xb, _ in train_loader:
            out, mu, log_var = model(xb)
            recon = nn.functional.binary_cross_entropy(
                out, xb.flatten(1), reduction='sum')
            kl = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
            opt.zero_grad()
            ((recon + beta * kl) / len(xb)).backward()
            opt.step()
    model.eval()
    with torch.no_grad():
        per_dim, r, k, n = [], 0.0, 0.0, 0
        for xb, _ in val_loader:
            out, mu, log_var = model(xb)
            per_dim.append(-0.5 * (1 + log_var - mu.pow(2)
                                   - log_var.exp()).mean(0))
            r += nn.functional.binary_cross_entropy(
                out, xb.flatten(1), reduction='sum').item()
            k += (-0.5 * torch.sum(
                1 + log_var - mu.pow(2) - log_var.exp())).item()
            n += len(xb)
        kl_per_dim = torch.stack(per_dim).mean(0)
    # a dimension carrying no information has a KL of about zero
    active = int((kl_per_dim > 0.01).sum())
    print('%8.1f %14.1f %12.1f %16d'
          % (beta, r / n, k / n, active))
```

```
    beta reconstruction           kl      active dims
```

```
   0.1          108.8         52.6              16
   1.0          123.5         16.4              16
   8.0          175.7          1.9              16
  60.0          201.4          0.0               0
```

The active dimensions column counts how many latent coordinates still vary between images. As beta rises, reconstruction gets worse and dimensions switch off. That trade is the single most important setting in a VAE, and it is why beta appears in the name of half the papers about them.

Why VAE samples are blurry

```
import torch

# Two equally plausible sharp images, and what minimising squared error
# against both produces.
a = torch.zeros(8, 8)
a[2:6, 2:4] = 1.0
b = torch.zeros(8, 8)
b[2:6, 5:7] = 1.0

best_single = (a + b) / 2
print('two plausible outputs, and the one that minimises the mean')
print('squared error against both:\n')
for name, img in [('option a', a), ('option b', b),
                  ('the average', best_single)]:
    print(name)
    for row in img:
        print('  ' + ''.join('%3.0f' % (v * 100) for v in row.tolist()))
    print()

for name, img in [('option a', a), ('the average', best_single)]:
    err = 0.5 * ((img - a) ** 2).mean() + 0.5 * ((img - b) ** 2).mean()
    print('%-14s expected error %.4f' % (name, err.item()))
```

```
two plausible outputs, and the one that minimises the mean
squared error against both:

option a
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0100100  0  0  0  0
    0  0100100  0  0  0  0
    0  0100100  0  0  0  0
    0  0100100  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0

option b
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0100100  0
    0  0  0  0  0100100  0
    0  0  0  0  0100100  0

average       expected error 0.2500
```

```
    0  0  0  0  0100100  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0

the average
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0
    0  0 50 50  0 50 50  0
    0  0 50 50  0 50 50  0
    0  0 50 50  0 50 50  0
    0  0 50 50  0 50 50  0
    0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0

option a       expected error 0.1250
the average    expected error 0.0625
```

The blur is the loss doing its job

When several outputs are equally plausible, the one that minimises expected squared error is their average, and the average of several sharp images is a blurry one. A VAE is not failing when its samples look soft; it is correctly optimising a loss that rewards hedging.

That is precisely the gap the next week's models attack. A GAN replaces the pixel loss with a second network that judges whether the output looks real, and averaging two plausible images looks obviously fake to such a judge. Diffusion takes a different route to the same end.

> **Day 6 takeaway**

Day 7 What They Are Actually For

Linear probes, an honest comparison, and the checklist

What these models are actually for, measured rather than asserted.

Representation quality

```python
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
val_set = Subset(test_all, range(2000))
train_loader = DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = DataLoader(val_set, batch_size=512)

RAMP = ' .:-=+*#%@'
def show(img, step=2):
```

```
    """Print a 28 by 28 image as characters."""
    t = img.detach().reshape(28, 28)
    t = (t - t.min()) / (t.max() - t.min() + 1e-9)
    for row in t[::step]:
        print(''.join(RAMP[min(9, int(v * 9.999))] for v in row.tolist()))
class AutoEncoder(nn.Module):
    def __init__(self, latent=16):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 256), nn.ReLU(),
            nn.Linear(256, 64), nn.ReLU(),
            nn.Linear(64, latent))
        self.decoder = nn.Sequential(
            nn.Linear(latent, 64), nn.ReLU(),
            nn.Linear(64, 256), nn.ReLU(),
            nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, x):
        return self.decoder(self.encoder(x))

def train_ae(model, epochs=8, lr=1e-3, noise=0.0, loader=None):
    torch.manual_seed(0)
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    loss_fn = nn.MSELoss()
    for _ in range(epochs):
        model.train()
        for xb, _ in (loader or train_loader):
            target = xb.flatten(1)
            inp = xb if noise == 0 else (xb + noise * torch.randn_like(xb)
                                         ).clamp(0, 1)
            opt.zero_grad()
            loss_fn(model(inp), target).backward()
            opt.step()
    model.eval()
    total = seen = 0
    with torch.no_grad():
        for xb, _ in val_loader:
            total += loss_fn(model(xb), xb.flatten(1)).item() * len(xb)
            seen += len(xb)
    return total / seen
from sklearn.linear_model import LogisticRegression
import numpy as np

torch.manual_seed(0)
plain = AutoEncoder(latent=32)
train_ae(plain, epochs=8)
torch.manual_seed(0)
denoise = AutoEncoder(latent=32)
train_ae(denoise, epochs=8, noise=0.4)

def features(model, loader):
    model.eval()
    xs, ys = [], []
```

```
    with torch.no_grad():
        for xb, yb in loader:
            xs.append(model.encoder(xb))
            ys.append(yb)
    return torch.cat(xs).numpy(), torch.cat(ys).numpy()

def raw(loader):
    # one pass. Two passes over a shuffled loader return the
    # images in one order and the labels in another.
    xs, ys = [], []
    for xb, yb in loader:
        xs.append(xb.flatten(1))
        ys.append(yb)
    return torch.cat(xs).numpy(), torch.cat(ys).numpy()

raw_tr, raw_y = raw(train_loader)
raw_va, raw_vy = raw(val_loader)

print('%-34s %10s %10s' % ('features for a linear classifier',
                           'width', 'accuracy'))
clf = LogisticRegression(max_iter=1000).fit(raw_tr, raw_y)
print('%-34s %10d %10.4f' % ('raw pixels', raw_tr.shape[1],
                             clf.score(raw_va, raw_vy)))
for name, model in [('plain autoencoder code', plain),
                    ('denoising autoencoder code', denoise)]:
    Xtr, ytr = features(model, train_loader)
    Xva, yva = features(model, val_loader)
    clf = LogisticRegression(max_iter=300).fit(Xtr, ytr)
    print('%-34s %10d %10.4f'
          % (name, Xtr.shape[1], clf.score(Xva, yva)))
```

```
features for a linear classifier        width   accuracy
raw pixels                                784     0.8705
plain autoencoder code                     32     0.8310
denoising autoencoder code                 32     0.7610
```

Two results worth sitting with, because neither is what the textbook ordering predicts. The denoising code is *worse* than the plain one for this classifier, and both are behind raw pixels.

Watch Out

On MNIST, raw pixels are a strong baseline

MNIST digits are close to linearly separable in pixel space already, so a linear classifier on 784 raw numbers is hard to beat. What the autoencoder achieved is compression: 32 numbers reach within a few points of 784, which is the interesting result rather than a failure.

The denoising variant learning a worse representation here is a reminder that its advantage is not universal. It helps when the corruption forces the model to learn structure it would otherwise ignore; on clean, centred, high-contrast digits there is little such structure to find, and the noise mostly makes the job harder. Run the probe rather than assuming the ordering.

What each of them is for

Model	Good at	Not good at
Plain autoencoder	Compression, anomaly detection	Generating anything
Denoising autoencoder	Representations, actual denoising	Generating anything
VAE	Generating, interpolating, a principled latent space	Sharp samples
GAN (week 13)	Sharp samples	Coverage, stable training
Diffusion (week 13)	Sharp samples and coverage	Speed, at generation time

Watch Out

Compression is not what these are for

A 32 number code from 784 pixels sounds like a twenty-four to one compression, and JPEG will beat it comfortably on any measure you care to name, while working on images it has never seen and needing no training. An autoencoder only compresses well on data resembling its training set.
The value is elsewhere: a representation learned without labels, an anomaly score that needs no examples of anomalies, and in the VAE case a latent space you can sample from. Judge them on those.

The checklist

1. Make the bottleneck genuinely narrow, or the model learns the identity.
2. Try the denoising variant, which costs one line, and measure it rather than assuming it helps.
3. Use convolutions for images, and check your `ConvTranspose2d` output shapes.
4. For a VAE, use the reparameterisation trick and sum the reconstruction term over pixels rather than averaging, so the two terms are on comparable scales.
5. Watch the number of active latent dimensions; if it is falling, beta is too high.
6. Judge a generative model by sampling from noise, not by reconstruction error.
7. Compare any learned representation against raw pixels with a linear classifier on top, which is the standard probe and takes two lines.

Day 7 takeaway

Self-Assessment

Try It Yourself

1. What forces an autoencoder to learn anything?

    a) The optimiser
    b) The loss function

c) The bottleneck, because a middle narrower than the input cannot carry everything and training must choose what to keep

d) The activation function

2. An autoencoder whose middle is as wide as its input reconstructs perfectly. What has it learned?

   a) A denoising function

   b) A generative model

   c) A useful representation

   d) Nothing of interest, because it can simply pass the input through

3. Why can you not sample from a plain autoencoder's latent space?

   a) Nothing constrains where codes land, so most points you might draw are in regions the decoder has never seen

   b) The decoder is not invertible

   c) The codes are discrete

   d) The code is too small

4. What is the reparameterisation trick for?

   a) Reducing the parameter count

   b) Writing the sample as mean plus standard deviation times fixed noise, so the randomness sits outside the path the gradient has to travel

   c) Normalising the latent space

   d) Speeding up sampling

5. What does the KL term do in a VAE?

   a) Balances the batch

   b) Measures reconstruction quality

   c) Pulls each encoded distribution towards a standard normal, which is what makes the space samplable

   d) Prevents overfitting to the training set

6. Week 12 pushed beta upward. At what point did posterior collapse actually appear?

   a) Only at beta = 60, where the KL reached zero and every latent dimension switched off

   b) It never appeared

   c) At beta = 1

   d) At beta = 8

7. A linear probe compared codes against raw pixels on MNIST. What was the result?

   a) Raw pixels scored highest, and the honest claim for the 32-dimensional code is compression rather than superiority

   b) The denoising code beat everything

   c) All three were identical

   d) The codes clearly beat raw pixels

8. Why are VAE samples soft?

    a) The decoder is too small

    b) A pixel-wise reconstruction loss is minimised by hedging, and the average of several plausible digits is a blurry one

    c) The latent space is too narrow

    d) Sampling uses too few steps

Answers:1c, 2d, 3a, 4b, 5c, 6a, 7a, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 13

Generative Adversarial Networks and Diffusion

Week 13

OBJECTIVES

Two ways to generate images without a pixel loss, and how to measure them. What each one gives you.

Week 12 could generate digits and they came out blurry, because a pixel loss rewards hedging. This week replaces that loss twice over. A GAN trains a second network to catch the hedging, which works and brings a set of failure modes that no supervised model has. Diffusion destroys an image with noise in small steps and learns to undo one step at a time, which is slower to sample and far easier to train. Both are measured here rather than described.

Day 1 Two Networks, One Game

Replacing a pixel loss with an opponent, and the first samples

Week 12 ended with a complaint. The variational autoencoder could generate digits from noise, but they came out soft, and the reason was the loss: averaging over every plausible digit is what minimises a pixel-wise error, and the average of several plausible digits is a blur. No amount of extra capacity fixes a loss that rewards hedging.

So replace the loss. Instead of scoring an output against a target pixel by pixel, train a second network whose only job is to tell real images from generated ones, and score the generator by whether it fools that network. An average of two digits does not fool anybody, so hedging stops paying.

The two networks

Neither is unusual on its own. The generator is a stack that widens from a small random vector up to 784 numbers, ending in `Tanh` because the images have been scaled to the range minus one to one. The discriminator is an ordinary binary classifier that happens to be looking at images.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)
```

```

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
torch.manual_seed(0)
G, D = Gen(), Disc()
G.eval()
with torch.no_grad():
    fake = G(torch.randn(2, 32))
    print('the discriminator scores this noise at %.3f'
          % D(fake).mean())
show(fake[0])
```

```
the discriminator scores this noise at -0.083
=++==++==+=+===++=+=+++==+
+=+++=+++=+=++++++=+=++=+++
++++++===*+===+=++==+=++++
+=+====++++=+=====++++=+==
++=+=+=+=++=+===+=++++==++
+=++===+==+++=+=+=+==+=+=+
+++=+++==+++=++=+++-=++==+
======++=+++=+=+=+++=+=+=++
====+=++++=+=++=+=+=-=====+
=====+==+++++=+=====+++=++
==+++=+-++=+-==++=+++=+=+=+
+=++=-++=+++++=+=+=+=-+++=
==++=+=+=++=+=+=+=+++=+=+==
===+=+===++*=+=+=+==+=+=++=
```

Static, and a discriminator that has not been trained either, sitting near zero because it has no opinion yet. A score of zero is the boundary: the network outputs a logit, so positive means it thinks the image is real and negative means it does not.

The game

Each batch runs two updates, in this order:

1. Show the discriminator a batch of real images labelled one and a batch of generated images labelled zero, and take one step on its parameters. The generated images are `detach()`ed, because this step must not change the generator.
2. Generate a fresh batch, score it with the discriminator, and take one step on the generator's parameters in the direction that makes the discriminator call those images real. The discriminator is not updated here even though the gradient passes straight through it.

Watch Out

The detach is not an optimisation

Leaving it out means the discriminator step also updates the generator, in the direction that makes its own output easier to detect. The run does not crash and the losses look plausible. It simply never produces anything, which is the worst kind of bug to have in a model that is expected to look bad for the first few epochs anyway.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
```

```
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
bce = nn.BCEWithLogitsLoss()

def train_gan(epochs=30, zdim=32, d_steps=1, saturating=False,
              lr_g=2e-4, lr_d=2e-4, seed=0, trace_every=0):
    torch.manual_seed(seed)
    G, D = Gen(zdim), Disc()
    og = torch.optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    for ep in range(1, epochs + 1):
        dsum = gsum = acc = n = 0.0
        for xb, _ in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            for _ in range(d_steps):
                fake = G(torch.randn(len(real), zdim)).detach()
                loss_d = bce(D(real), ones) + bce(D(fake), zeros)
                od.zero_grad()
                loss_d.backward()
                od.step()
            fake = G(torch.randn(len(real), zdim))
            score = D(fake)
            loss_g = -bce(score, zeros) if saturating else bce(score, ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
            dsum += loss_d.item()
            gsum += loss_g.item()
            acc += ((D(real) > 0).float().mean().item()
                    + (score < 0).float().mean().item()) / 2
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %12.3f %12.3f %14.2f'
                  % (ep, dsum / n, gsum / n, acc / n))
    G.eval()
    return G, D
print('%6s %12s %12s %14s' % ('epoch', 'D loss', 'G loss', 'D accuracy'))
G, D = train_gan(epochs=30, trace_every=5)
torch.manual_seed(7)
with torch.no_grad():
    for img in G(torch.randn(3, 32)):
        show(img)
```

```
        print()
```

```
 epoch       D loss       G loss     D accuracy
     5        1.189        0.833           0.87
    10        1.153        0.984           0.79
    15        1.084        1.157           0.81
    20        1.085        1.123           0.78
    25        1.006        1.209           0.80
    30        1.007        1.230           0.80
 .      .-     .
. +      . :            .
           -. .  %%.: .
   .      +@@@@@@@@@@%.
         ..#@@.     +%.
-         :*#     .**@++.
 :        :+. .-%%@%+: .
         .-:  =@@%*+##--
        +=-=*@%...:@%#   :
=    +      .       *=-.:
     .. . :      :#@@%        +
  -    *:%@%@@@@@@@%      *
            .::@=:=
              :             .


                          .
                             -
      .             .  ::     .
       .       . . - .   .     +
               #+@@==.:.:
             --+-%-       .    .
            .: -..       .
      +        =#:=
    +        .*%@@=
   .      .=#%@@@=      +
           *%@@@:         .     .
      :    .:*:                .
         . ..
       =  -


                          .


                .  -.   .
              .##@=--=:
            ---%=#=.: --
              ..: :#. :-.
             --@*+-:    +
      -        %%-#..
                 :-==
     . . ..=#++.-%##
        :.%%@#:.*@#:
       -=%@+*-
          :
```

Thirty epochs on eight thousand images, from a generator that is four dense layers. What comes out has the thickness and curvature of handwriting, closed loops in roughly the right places, and a dark background it was never told about. What it does not have is a particular digit you would confidently name. That is the honest description of this budget, and it is still a long way from the

noise the same network produced a few blocks ago, with no target image anywhere in the loss.

The idea in one line

Day 2 Reading a Run That Has No Validation Loss

Why the losses say so little, and the gradient that vanishes

Look again at the trace from yesterday. Over thirty epochs the discriminator loss drifted down a little, the generator loss drifted *up* by about half, and discriminator accuracy slid from 0.87 to 0.80. Meanwhile the samples went from static to digit-shaped.

Sit with that for a moment. The generator's loss got worse while the generator got better. In any of the first twelve weeks that sentence would be a bug report.

Why the losses tell you so little

Both numbers are measured against an opponent that is changing. If the generator loss falls, that can mean the generator improved, or it can mean the discriminator got worse and is now being fooled by the same images it caught last epoch. The two are indistinguishable from the loss alone, and they call for opposite responses.

Watch Out

There is no validation loss for a GAN

You cannot hold out data and watch a number to decide when to stop, because there is no target to compare against. This is a real and permanent difference from everything in the first twelve weeks, and it is why the rest of this week is spent building measurements that do not come from the loss.

The one number worth reading

Discriminator accuracy, which the loop above tracks as the average of how often it calls real images real and generated images fake. At the start it should be high, because early fakes are obvious. If it stays pinned at one, the generator is getting no useful signal. If it falls to a half the discriminator has stopped discriminating and the generator is optimising against nothing. Somewhere in between, and moving slowly, is what a working run looks like.

The loss the original paper wrote down, and the one everybody uses

Stated as a game, the generator should minimise the probability the discriminator assigns to *fake*. Written that way it has a flat spot exactly where you need it most: when the discriminator is confident, the gradient reaching the generator goes to almost nothing, and the generator is stuck precisely because it is bad. The fix is to flip the problem round and have the generator maximise the probability of *real* instead. The two have the same optimum and completely different gradients.

That is the usual argument. It is also measurable, so here it is measured, with the discriminator given a single epoch of head start:

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
bce = nn.BCEWithLogitsLoss()

def train_gan(epochs=30, zdim=32, d_steps=1, saturating=False,
              lr_g=2e-4, lr_d=2e-4, seed=0, trace_every=0):
    torch.manual_seed(seed)
    G, D = Gen(zdim), Disc()
    og = torch.optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    for ep in range(1, epochs + 1):
        dsum = gsum = acc = n = 0.0
        for xb, _ in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
```

```
            for _ in range(d_steps):
                fake = G(torch.randn(len(real), zdim)).detach()
                loss_d = bce(D(real), ones) + bce(D(fake), zeros)
                od.zero_grad()
                loss_d.backward()
                od.step()
            fake = G(torch.randn(len(real), zdim))
            score = D(fake)
            loss_g = -bce(score, zeros) if saturating else bce(score, ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
            dsum += loss_d.item()
            gsum += loss_g.item()
            acc += ((D(real) > 0).float().mean().item()
                    + (score < 0).float().mean().item()) / 2
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %12.3f %12.3f %14.2f'
                  % (ep, dsum / n, gsum / n, acc / n))
    G.eval()
    return G, D
torch.manual_seed(0)
G, D = Gen(), Disc()
# Give the discriminator one epoch of head start. This is not a contrived
# situation: at the beginning of any run the fakes are obvious noise and
# the discriminator gets very good very quickly.
od = torch.optim.Adam(D.parameters(), lr=2e-4, betas=(0.5, 0.999))
for xb, _ in loader:
    real = xb.flatten(1)
    fake = G(torch.randn(len(real), 32)).detach()
    loss_d = (bce(D(real), torch.ones(len(real), 1))
              + bce(D(fake), torch.zeros(len(real), 1)))
    od.zero_grad()
    loss_d.backward()
    od.step()

def g_grad_norm(saturating):
    G.zero_grad()
    score = D(G(torch.randn(256, 32)))
    zeros, ones = torch.zeros(256, 1), torch.ones(256, 1)
    loss = -bce(score, zeros) if saturating else bce(score, ones)
    loss.backward()
    return torch.cat([p.grad.flatten()
                      for p in G.parameters()]).norm().item()

with torch.no_grad():
    caught = (D(G(torch.randn(512, 32))) < 0).float().mean()
print('the discriminator already spots %.1f%% of the fakes' % (100 * caught))
print()
print('%-22s %16s' % ('generator loss', 'gradient norm'))
print('%-22s %16.6f' % ('saturating', g_grad_norm(True)))
print('%-22s %16.6f' % ('non-saturating', g_grad_norm(False)))
```

```
the discriminator already spots 100.0% of the fakes

generator loss            gradient norm
saturating                     0.068186
non-saturating                 4.905675
```

The discriminator is catching every single fake after one epoch, which is exactly the regime the argument is about. Same generator, same discriminator, same batch of noise, and the gradient reaching the generator differs by a factor of roughly seventy. The saturating version is not slightly worse, it is delivering almost nothing to learn from at the moment the generator most needs a direction.

In code the whole change is `bce(score, ones)` in place of `-bce(score, zeros)`. Two short expressions, both of which read as reasonable, which is why it is easy to write the wrong one and very hard to notice afterwards.

What to do when a run goes wrong

- **Discriminator accuracy pinned at 1.0.** It is winning too easily. Slow it down: a lower learning rate for it than for the generator, or fewer of its steps per generator step.
- **Discriminator accuracy at 0.5 and generator loss falling steadily.** The generator has found a hole rather than learned the data. Look at the samples before believing the number.
- **Both losses flat and samples static.** Check the `detach`, check that both optimisers are stepping, and check that the generator's final activation matches the range the data was scaled to.
- **Everything looks fine and the samples are all the same digit.** That is tomorrow.

Day 3 Mode Collapse

The failure the loss cannot show you, and how to count it

A generator has an easy way to satisfy its loss that has no analogue in anything so far. If one particular output reliably fools the discriminator, producing that output every time scores well, and the loss has no term that asks for variety. The generator is being paid per image, not per distribution.

You have to count

There is no loss to read, so the measurement has to be built. Train an ordinary digit classifier on the real data, ask the generator for two thousand images, and count what the classifier says they are. A generator covering the data should produce something close to an even spread across ten digits, which has an entropy of ln(10), about 2.303. The further below that, the narrower the generator.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
```

```
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
bce = nn.BCEWithLogitsLoss()

def train_gan(epochs=30, zdim=32, d_steps=1, saturating=False,
              lr_g=2e-4, lr_d=2e-4, seed=0, trace_every=0):
    torch.manual_seed(seed)
    G, D = Gen(zdim), Disc()
    og = torch.optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    for ep in range(1, epochs + 1):
        dsum = gsum = acc = n = 0.0
        for xb, _ in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            for _ in range(d_steps):
                fake = G(torch.randn(len(real), zdim)).detach()
                loss_d = bce(D(real), ones) + bce(D(fake), zeros)
                od.zero_grad()
                loss_d.backward()
                od.step()
            fake = G(torch.randn(len(real), zdim))
            score = D(fake)
```

```
            loss_g = -bce(score, zeros) if saturating else bce(score, ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
            dsum += loss_d.item()
            gsum += loss_g.item()
            acc += ((D(real) > 0).float().mean().item()
                    + (score < 0).float().mean().item()) / 2
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %12.3f %12.3f %14.2f'
                  % (ep, dsum / n, gsum / n, acc / n))
    G.eval()
    return G, D
def train_judge():
    """A plain digit classifier, used only to score what the GAN makes."""
    torch.manual_seed(1)
    clf = nn.Sequential(nn.Flatten(),
                        nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    opt = torch.optim.AdamW(clf.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    for _ in range(12):
        for xb, yb in loader:
            opt.zero_grad()
            lf(clf(xb), yb).backward()
            opt.step()
    clf.eval()
    test = Subset(datasets.MNIST('data', train=False, download=True,
                                 transform=tf), range(2000))
    tl = DataLoader(test, batch_size=512)
    right = seen = 0
    with torch.no_grad():
        for xb, yb in tl:
            right += (clf(xb).argmax(1) == yb).sum().item()
            seen += len(yb)
    return clf, right / seen

def coverage(G, clf, n=2000):
    with torch.no_grad():
        imgs = G(torch.randn(n, G.zdim)).reshape(-1, 1, 28, 28)
        pred = clf(imgs).argmax(1)
    counts = torch.bincount(pred, minlength=10).float()
    frac = counts / counts.sum()
    ent = -(frac[frac > 0] * frac[frac > 0].log()).sum().item()
    return counts.long().tolist(), ent
clf, acc = train_judge()
print('judge accuracy on real digits %.4f' % acc)
print()
print('%-26s %-34s %8s' % ('recipe', 'digits produced out of 2000', 'entropy'))
for name, kw in [('standard, z=32', {}),
                 ('saturating G loss', {'saturating': True}),
                 ('z=2', {'zdim': 2}),
                 ('5 D steps per G step', {'d_steps': 5}),
```

```
                  ('D learns 10x faster', {'lr_d': 2e-3})]:
    G, _ = train_gan(epochs=30, **kw)
    counts, ent = coverage(G, clf)
    print('%-26s %-34s %8.3f'
          % (name, ' '.join('%3d' % c for c in counts), ent))
print()
print('for reference, a perfectly even spread scores %.3f'
      % float(torch.tensor(10.0).log()))
```

```
judge accuracy on real digits 0.9075

recipe                     digits produced out of 2000         entropy
standard, z=32             167 178 172 189 273 184 218 222 279 118    2.275
saturating G loss           62  82 191 170 239 336 265 243 275 137    2.206
z=2                          1 467  30 208 646  47  22 276 274  29    1.752
5 D steps per G step         3 485 126  90 335 154  40 304 257 206    2.026
D learns 10x faster          9 179 105 124 416 162 102 445 217 241    2.080

for reference, a perfectly even spread scores 2.303
```

The standard recipe comes out close to even, which is a better result than a four-layer generator deserves. Everything below it is narrower, and the clearest case is the one with a two-dimensional latent vector: whole digits nearly vanish from its output while others appear several hundred times. A latent space that small cannot be stretched over ten separated regions of digit space, so it covers a few and abandons the rest.

Watch Out

The judge is not perfect either

It is a two-layer classifier trained on the same eight thousand images, and it gets about nine real digits in ten right. So a count of zero in some column is partly the generator and partly the judge being wrong about odd-looking inputs. Read the table as a ranking, which it is reliable for, rather than as an exact census.

What actually helped, and what did not

The two adjustments most often recommended for stabilising a GAN, giving the discriminator extra steps and letting it learn faster, both came out narrower than doing neither. That is worth being direct about: they are remedies for a discriminator that is losing, and this discriminator was not losing. Applied to a run that does not have that problem they make the generator's job harder for no gain.

The saturating loss did less damage to coverage here than the reputation suggests, which is a reminder that day 2's measurement was about the gradient at the start of a run, not a guarantee about where the run ends up. A small model on an easy dataset recovers from a bad gradient that would sink a larger one.

The habit to keep

Day 4 Asking For a Particular Digit

Conditioning both networks, and how much control that really buys

So far the generator takes noise and returns whatever it likes. Usually you want to ask for something specific, and the change needed is smaller than it sounds: give both networks the label.

The generator receives the label alongside its noise vector, so it can learn a different mapping for each one. The discriminator receives the label alongside the image, which is the part that matters. Without it the discriminator would only ask whether the image looks like a digit, and the generator could ignore the label entirely. With it, the question becomes whether this image looks like a real example *of that label*, and producing a convincing seven when asked for a three now loses.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
bce = nn.BCEWithLogitsLoss()

def train_gan(epochs=30, zdim=32, d_steps=1, saturating=False,
```

```
              lr_g=2e-4, lr_d=2e-4, seed=0, trace_every=0):
    torch.manual_seed(seed)
    G, D = Gen(zdim), Disc()
    og = torch.optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    for ep in range(1, epochs + 1):
        dsum = gsum = acc = n = 0.0
        for xb, _ in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            for _ in range(d_steps):
                fake = G(torch.randn(len(real), zdim)).detach()
                loss_d = bce(D(real), ones) + bce(D(fake), zeros)
                od.zero_grad()
                loss_d.backward()
                od.step()
            fake = G(torch.randn(len(real), zdim))
            score = D(fake)
            loss_g = -bce(score, zeros) if saturating else bce(score, ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
            dsum += loss_d.item()
            gsum += loss_g.item()
            acc += ((D(real) > 0).float().mean().item()
                    + (score < 0).float().mean().item()) / 2
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %12.3f %12.3f %14.2f'
                  % (ep, dsum / n, gsum / n, acc / n))
    G.eval()
    return G, D
class CondGen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.emb = nn.Embedding(10, 10)
        self.net = nn.Sequential(
            nn.Linear(zdim + 10, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z, y):
        return self.net(torch.cat([z, self.emb(y)], 1))

class CondDisc(nn.Module):
    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(10, 10)
        self.net = nn.Sequential(
            nn.Linear(784 + 10, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))
```

```

    def forward(self, x, y):
        return self.net(torch.cat([x.flatten(1), self.emb(y)], 1))

def train_cgan(epochs=30, zdim=32, seed=0):
    torch.manual_seed(seed)
    G, D = CondGen(zdim), CondDisc()
    og = torch.optim.Adam(G.parameters(), lr=2e-4, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=2e-4, betas=(0.5, 0.999))
    for _ in range(epochs):
        for xb, yb in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            fake = G(torch.randn(len(real), zdim), yb).detach()
            loss_d = bce(D(real, yb), ones) + bce(D(fake, yb), zeros)
            od.zero_grad()
            loss_d.backward()
            od.step()
            fake = G(torch.randn(len(real), zdim), yb)
            loss_g = bce(D(fake, yb), ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
    G.eval()
    return G
print(CondGen())
```

```
CondGen(
  (emb): Embedding(10, 10)
  (net): Sequential(
    (0): Linear(in_features=42, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Linear(in_features=256, out_features=784, bias=True)
    (6): Tanh()
  )
)
```

Does the label actually control the output

The same judge from yesterday answers this directly. Ask for two hundred of each digit and count how often the classifier agrees with what was requested.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
```

```
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
class Gen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.net = nn.Sequential(
            nn.Linear(zdim, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z):
        return self.net(z)

class Disc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x):
        return self.net(x.flatten(1))
bce = nn.BCEWithLogitsLoss()

def train_gan(epochs=30, zdim=32, d_steps=1, saturating=False,
              lr_g=2e-4, lr_d=2e-4, seed=0, trace_every=0):
    torch.manual_seed(seed)
    G, D = Gen(zdim), Disc()
    og = torch.optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    for ep in range(1, epochs + 1):
        dsum = gsum = acc = n = 0.0
        for xb, _ in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            for _ in range(d_steps):
                fake = G(torch.randn(len(real), zdim)).detach()
                loss_d = bce(D(real), ones) + bce(D(fake), zeros)
                od.zero_grad()
                loss_d.backward()
                od.step()
            fake = G(torch.randn(len(real), zdim))
            score = D(fake)
```

```
            loss_g = -bce(score, zeros) if saturating else bce(score, ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
            dsum += loss_d.item()
            gsum += loss_g.item()
            acc += ((D(real) > 0).float().mean().item()
                    + (score < 0).float().mean().item()) / 2
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %12.3f %12.3f %14.2f'
                  % (ep, dsum / n, gsum / n, acc / n))
    G.eval()
    return G, D
class CondGen(nn.Module):
    def __init__(self, zdim=32):
        super().__init__()
        self.zdim = zdim
        self.emb = nn.Embedding(10, 10)
        self.net = nn.Sequential(
            nn.Linear(zdim + 10, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 256), nn.BatchNorm1d(256), nn.LeakyReLU(0.2),
            nn.Linear(256, 784), nn.Tanh())

    def forward(self, z, y):
        return self.net(torch.cat([z, self.emb(y)], 1))

class CondDisc(nn.Module):
    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(10, 10)
        self.net = nn.Sequential(
            nn.Linear(784 + 10, 256), nn.LeakyReLU(0.2),
            nn.Linear(256, 128), nn.LeakyReLU(0.2),
            nn.Linear(128, 1))

    def forward(self, x, y):
        return self.net(torch.cat([x.flatten(1), self.emb(y)], 1))

def train_cgan(epochs=30, zdim=32, seed=0):
    torch.manual_seed(seed)
    G, D = CondGen(zdim), CondDisc()
    og = torch.optim.Adam(G.parameters(), lr=2e-4, betas=(0.5, 0.999))
    od = torch.optim.Adam(D.parameters(), lr=2e-4, betas=(0.5, 0.999))
    for _ in range(epochs):
        for xb, yb in loader:
            real = xb.flatten(1)
            ones = torch.ones(len(real), 1)
            zeros = torch.zeros(len(real), 1)
            fake = G(torch.randn(len(real), zdim), yb).detach()
            loss_d = bce(D(real, yb), ones) + bce(D(fake, yb), zeros)
            od.zero_grad()
            loss_d.backward()
            od.step()
```

```
            fake = G(torch.randn(len(real), zdim), yb)
            loss_g = bce(D(fake, yb), ones)
            og.zero_grad()
            loss_g.backward()
            og.step()
    G.eval()
    return G
def train_judge():
    """A plain digit classifier, used only to score what the GAN makes."""
    torch.manual_seed(1)
    clf = nn.Sequential(nn.Flatten(),
                        nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    opt = torch.optim.AdamW(clf.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    for _ in range(12):
        for xb, yb in loader:
            opt.zero_grad()
            lf(clf(xb), yb).backward()
            opt.step()
    clf.eval()
    test = Subset(datasets.MNIST('data', train=False, download=True,
                                 transform=tf), range(2000))
    tl = DataLoader(test, batch_size=512)
    right = seen = 0
    with torch.no_grad():
        for xb, yb in tl:
            right += (clf(xb).argmax(1) == yb).sum().item()
            seen += len(yb)
    return clf, right / seen

def coverage(G, clf, n=2000):
    with torch.no_grad():
        imgs = G(torch.randn(n, G.zdim)).reshape(-1, 1, 28, 28)
        pred = clf(imgs).argmax(1)
    counts = torch.bincount(pred, minlength=10).float()
    frac = counts / counts.sum()
    ent = -(frac[frac > 0] * frac[frac > 0].log()).sum().item()
    return counts.long().tolist(), ent
clf, _ = train_judge()
G = train_cgan(epochs=30)
torch.manual_seed(5)
print('%8s %10s' % ('asked for', 'agreement'))
total = 0.0
with torch.no_grad():
    for digit in range(10):
        y = torch.full((200,), digit, dtype=torch.long)
        imgs = G(torch.randn(200, 32), y).reshape(-1, 1, 28, 28)
        agree = (clf(imgs).argmax(1) == digit).float().mean().item()
        total += agree / 10
        print('%8d %10.3f' % (digit, agree))
print('%8s %10.3f' % ('overall', total))
torch.manual_seed(9)
with torch.no_grad():
```

```
    for digit in [3, 7]:
        print()
        print('asked for a %d' % digit)
        show(G(torch.randn(1, 32), torch.tensor([digit]))[0])
```

```
asked for  agreement
       0      0.915
       1      0.825
       2      0.755
       3      0.590
       4      0.575
       5      0.740
       6      0.215
       7      0.255
       8      0.605
       9      0.370
 overall      0.584

asked for a 3
      . -   .
                    @
  : .   .:=  .            .
    * . #-.=%@%-# .
  *     =%@%%-.@*.
        =@#.    =.      .   #
      +:@@@.   #       *
        +@#@@@@.-     ..=
 :    : : ..-. ++  :*#%     .
      =.. @#+.  .@@@@@@+    .
  ..    :@@@%  .#@@@@::  -
         .-    .@@@+%  =
   .  .        #: %   :


asked for a 7
                .
   -        -  .   :  .
         .
           +.#%*.:        . .
  .   +=+@=%%+@@-. .#%:
     =-*#-: .#%*-:. %.   :.
   .  .=+.=:*-=%-:.%::--
  .    .-:.+@*@%@%@-.  .
..    :       =**%*:*
.  :-          -@@%- .        .
              .-%%-..    :   .
        .     .%@%==:     -
       .-.  %%%@@:.    . =
        . . -.  ::    .
```

Against a floor of 0.1 for a generator that ignored the label completely, and a ceiling of about 0.91 set by the judge's own accuracy on real digits, the overall figure is a little under 0.6. So the conditioning works, and calling it working needs qualifying: some digits come back reliably and others hardly ever do, with the worst column barely above a fifth.

Why the columns differ so much

Nothing in the training gave equal weight to being right about each digit. The generator is rewarded for fooling the discriminator, and for a label whose real examples vary a lot in shape, a vague blob that leans towards the class average is often enough to pass. The judge, which was trained on clean data, then declines to call it that digit. The discriminator and the judge are asking different questions, and the gap between them is where the missing forty percent sits.

This is the pattern, not the exception

Every conditional generator has this gap: the training signal is plausibility, and what you actually want is plausibility *and* obedience. Larger models close it rather than remove it. When a text-to-image model gives you four of the five things you asked for, this is the same failure at a different scale.

Day 5 Destroying an Image on Purpose

The forward process, the schedule, and the jump that makes it train

Everything in the first four days rests on one uncomfortable arrangement: the loss is another network, and it is moving. Diffusion gets to the same place without that. It defines a process that destroys an image, one small step at a time, until nothing is left but noise, and then trains a network to undo a single step of it. There is no opponent, and the loss is an ordinary mean squared error that means what it says.

The schedule

One number per step, conventionally called beta, says how much noise that step adds. Start it small so the early steps barely disturb the image, and grow it so the later ones finish the job. What the network actually needs is not beta but its running product, written abar: the fraction of the original image still standing at step t.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
```

```
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
x0 = train_set[1][0].flatten()
torch.manual_seed(0)
for t in [0, 40, 100, 199]:
    eps = torch.randn_like(x0)
    xt = abar[t].sqrt() * x0 + (1 - abar[t]).sqrt() * eps
    print('t = %-4d  abar = %.4f   mean %.3f   std %.3f'
          % (t, abar[t], xt.mean(), xt.std()))
    show(xt, step=3)
    print()
```

```
t = 0     abar = 0.9999   mean -0.688   std 0.658


             :%@@@@@:
          .@@@#=@%.-@*
        #@@: .       @@#
      -@@=           @@+
      -@+        .*@*
      -@@@%%@@@#+



t = 40    abar = 0.8102   mean -0.614   std 0.731
:   .:    : ::.:     -    +.
:-.       -  . + :    .   =
..   :   :   -@%%*=@: =  :
.-: .:  .:*+@=%***-=%# : -
  .. :  =@@   .. -  @@#:
 :.*.+%@%+     . ..:-@+*  -
.:  -.=@#  .: =  =@#.  -   :
 :  --.#@@*@%%@@@ .  -=   :
 :  :   ..   :  %=  ..     .
.   .  :* -    -   .  :  ..

t = 100   abar = 0.2760   mean -0.424   std 0.925
.+..  =. %: = =   = % : %-::
=- +.+  -=  -   @ %.*: @.#=%
 =  % :-+-  : +@=@#@==*  :-
#  %:  @-@:%#@.@*+--@=  : *-
*: *.  -:@--:  .  #@:#% .:-+
  := =@**-: :#+.# + @*@ %
* .*.+ @*.-@**  *%%@ +-%+#
.:@% =*#@=.#@:@@= + :+     .
 - =+#  :=::  -:-+ + :@. -
:*  :  *--:  @ .*-  :  = *.

t = 199   abar = 0.0061   mean -0.080   std 0.976
 -@%@ : @=.#@:@- @*+@-:  *-
@@.   %@@=#**.@   %=**- :* %
@ +% = *=  *@@@=@ *%##@#*:@@
@: @ **@= =-%+-.* -:@- *:@ -
@*.+=  @.@   = %*:@@+@-++ *=
%@.   = * # @@=@*+@:#+%:#  @
 #  = @.@. + .-#*-@+=*.*%@ +
@ -:+#@@.-.# #: =@.@#:%=@  =
+:#-+%.* #@ @# @=# : +@ %%=
```

```
#-@-:@*@-#@.%: # %=-*- = -
```

The digit is intact at the start, still legible under the noise at step 40, arguably visible at step 100 if you already know it is there, and gone by step 199. Watch the two statistics as well: the last row has a mean near zero and a standard deviation near one, which is to say it has become a sample from a standard normal distribution. That is the whole point of the schedule, because it is the distribution sampling will start from later.

Watch Out

Check that your schedule actually finishes

The first version of this week used a gentler schedule whose abar at the final step was 0.13, not 0.006. That leaves an eighth of the original image still present, so the model was never trained on anything resembling pure noise, and sampling began from a state it had never seen. It produced static. Print abar at your last step before you train anything, and if it is not close to zero, your schedule is too short or too gentle.

The jump that makes any of this trainable

Taken literally, producing a training example at step 500 means running 500 steps. Do that for every image in every batch and there is no training run. It is unnecessary: a chain of Gaussian steps is itself Gaussian, and abar is exactly the term that lets you skip straight there in one line.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
x0 = train_set[1][0].flatten()
torch.manual_seed(0)

# the slow way: 80 actual steps of the process, 400 times over
runs = []
for _ in range(400):
```

```
    x = x0.clone()
    for t in range(80):
        x = alpha[t].sqrt() * x + beta[t].sqrt() * torch.randn_like(x)
    runs.append(x)
slow = torch.stack(runs)

# the fast way: straight to t=79 in one line
fast = torch.stack([abar[79].sqrt() * x0
                    + (1 - abar[79]).sqrt() * torch.randn_like(x0)
                    for _ in range(400)])

print('%-24s %12s %12s' % ('', 'mean pixel', 'std pixel'))
print('%-24s %12.4f %12.4f' % ('80 small steps', slow.mean(), slow.std()))
print('%-24s %12.4f %12.4f' % ('one closed-form jump', fast.mean(),
                               fast.std()))
```

```
                         mean pixel    std pixel
80 small steps              -0.4620       0.8624
one closed-form jump        -0.4603       0.8642
```

Same distribution, one line instead of eighty. This is why training picks a random t for every image in the batch and jumps there directly, and it is the single fact that turns diffusion from a thought experiment into something you can run.

Day 6 Learning to Undo One Step

Predicting the noise, the loss floor, and watching a digit appear

The forward process has no parameters. Everything learned sits in one network with one job: given a noisy image and a number saying how noisy it is, say which noise was added.

Predicting the noise rather than the clean image looks like an odd choice, and the two are algebraically interchangeable, since knowing either one and the noisy image gives you the other. Predicting the noise turns out to be better behaved across the whole range of t, and it is what the standard formulation does.

Telling the network what time it is

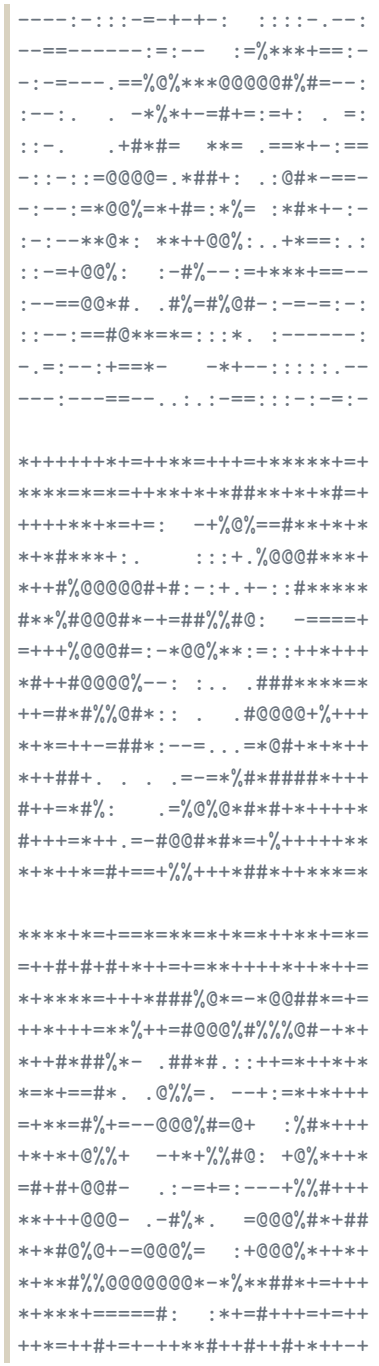
The same noisy image means different things at different t, so the step number has to reach the network. Here it is an embedding looked up per step and added inside every block, rather than only at the input. That detail matters more than it looks: a first attempt that conditioned only the input layer underfit badly.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
```

```
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
class Block(nn.Module):
    """One residual block, told at every level how noisy the input is."""
    def __init__(self, h):
        super().__init__()
        self.norm = nn.LayerNorm(h)
        self.tproj = nn.Linear(h, h)
        self.net = nn.Sequential(nn.SiLU(), nn.Linear(h, h),
                                 nn.SiLU(), nn.Linear(h, h))

    def forward(self, x, temb):
        return x + self.net(self.norm(x) + self.tproj(temb))

class Denoiser(nn.Module):
    """Given a noisy image and how noisy it is, predict the noise."""
    def __init__(self, hidden=768, blocks=3):
        super().__init__()
        self.inp = nn.Linear(784, hidden)
        self.temb = nn.Embedding(T, hidden)
        self.blocks = nn.ModuleList([Block(hidden) for _ in range(blocks)])
        self.out = nn.Sequential(nn.LayerNorm(hidden), nn.SiLU(),
                                 nn.Linear(hidden, 784))

    def forward(self, x, t):
        temb = self.temb(t)
        h = self.inp(x)
        for block in self.blocks:
            h = block(h, temb)
        return self.out(h)

def add_noise(x0, t, eps):
    a = abar[t].reshape(-1, 1)
    return a.sqrt() * x0 + (1 - a).sqrt() * eps

def train_diffusion(epochs=200, seed=0, trace_every=0):
    torch.manual_seed(seed)
    net = Denoiser()
    opt = torch.optim.AdamW(net.parameters(), lr=1e-3)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, epochs)
    for ep in range(1, epochs + 1):
        total = n = 0.0
        for xb, _ in loader:
            x0 = xb.flatten(1)
            t = torch.randint(0, T, (len(x0),))
```

```
            eps = torch.randn_like(x0)
            loss = ((net(add_noise(x0, t, eps), t) - eps) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        sched.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    net.eval()
    return net

@torch.no_grad()
def sample(net, n=8, seed=3, keep=()):
    torch.manual_seed(seed)
    x = torch.randn(n, 784)
    frames = {}
    for step in range(T - 1, -1, -1):
        t = torch.full((n,), step, dtype=torch.long)
        # what the network thinks the clean image is, from here
        x0 = ((x - (1 - abar[step]).sqrt() * net(x, t))
              / abar[step].sqrt()).clamp(-1, 1)
        # the posterior mean: a small step from x towards that guess
        mean = (abar_prev[step].sqrt() * beta[step] * x0
                + alpha[step].sqrt() * (1 - abar_prev[step]) * x
                ) / (1 - abar[step])
        x = mean if step == 0 else mean + beta[step].sqrt() * torch.randn_like(x)
        if step in keep:
            frames[step] = x.clone()
    return (x, frames) if keep else x
print('a network that always guesses zero scores %.4f'
      % (torch.randn(4000, 784) ** 2).mean())
print()
print('%6s %14s' % ('epoch', 'noise MSE'))
net = train_diffusion(epochs=200, trace_every=25)
torch.save(net.state_dict(), 'denoiser.pt')
print()
for img in sample(net, n=3):
    show(img)
    print()
```

```
a network that always guesses zero scores 0.9993

 epoch      noise MSE
    25         0.2013
    50         0.1796
    75         0.1707
   100         0.1539
   125         0.1544
   150         0.1438
   175         0.1431
   200         0.1380

--=-:----::----=:-:--:=--.--
```



Why the loss does not go anywhere near zero

Noise drawn from a standard normal has a mean squared value of one, so guessing zero every time scores about 1.0, and that is the number to beat. Reaching 0.14 is a large win, and no amount of training would reach zero, because a good part of this loss is irreducible.

Consider a very small t. The noisy image is almost the clean image, with a whisper of noise on top. Recovering *which* noise that was would mean knowing the clean image to a precision far beyond what is there to see, so the best possible prediction is close to zero and the loss at those steps stays close to one. The average over all t therefore has a floor well above zero, and comparing runs is the only way to read it.

The samples are digits. They also carry a haze across the background that a real MNIST

image does not have, which is what an under-trained denoiser looks like: each reverse step leaves a small error, two hundred steps accumulate it, and nothing in the process cleans it up at the end. The loss was still falling slowly at epoch 200, so this is a compute budget, not a broken method.

Running the process backwards

Sampling starts from pure noise and walks back down. At each step the network predicts the noise, which is rearranged into a guess at the clean image, clamped to the valid pixel range, and then used to take one small step back towards it, with fresh noise added. Reading it as guess where this is going, move a little way there, jitter is a fair description of the arithmetic.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
class Block(nn.Module):
    """One residual block, told at every level how noisy the input is."""
    def __init__(self, h):
        super().__init__()
        self.norm = nn.LayerNorm(h)
        self.tproj = nn.Linear(h, h)
        self.net = nn.Sequential(nn.SiLU(), nn.Linear(h, h),
                                 nn.SiLU(), nn.Linear(h, h))

    def forward(self, x, temb):
        return x + self.net(self.norm(x) + self.tproj(temb))

class Denoiser(nn.Module):
    """Given a noisy image and how noisy it is, predict the noise."""
    def __init__(self, hidden=768, blocks=3):
        super().__init__()
        self.inp = nn.Linear(784, hidden)
        self.temb = nn.Embedding(T, hidden)
        self.blocks = nn.ModuleList([Block(hidden) for _ in range(blocks)])
```

```
        self.out = nn.Sequential(nn.LayerNorm(hidden), nn.SiLU(),
                                 nn.Linear(hidden, 784))

    def forward(self, x, t):
        temb = self.temb(t)
        h = self.inp(x)
        for block in self.blocks:
            h = block(h, temb)
        return self.out(h)

def add_noise(x0, t, eps):
    a = abar[t].reshape(-1, 1)
    return a.sqrt() * x0 + (1 - a).sqrt() * eps

def train_diffusion(epochs=200, seed=0, trace_every=0):
    torch.manual_seed(seed)
    net = Denoiser()
    opt = torch.optim.AdamW(net.parameters(), lr=1e-3)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, epochs)
    for ep in range(1, epochs + 1):
        total = n = 0.0
        for xb, _ in loader:
            x0 = xb.flatten(1)
            t = torch.randint(0, T, (len(x0),))
            eps = torch.randn_like(x0)
            loss = ((net(add_noise(x0, t, eps), t) - eps) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        sched.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    net.eval()
    return net

@torch.no_grad()
def sample(net, n=8, seed=3, keep=()):
    torch.manual_seed(seed)
    x = torch.randn(n, 784)
    frames = {}
    for step in range(T - 1, -1, -1):
        t = torch.full((n,), step, dtype=torch.long)
        # what the network thinks the clean image is, from here
        x0 = ((x - (1 - abar[step]).sqrt() * net(x, t))
              / abar[step].sqrt()).clamp(-1, 1)
        # the posterior mean: a small step from x towards that guess
        mean = (abar_prev[step].sqrt() * beta[step] * x0
                + alpha[step].sqrt() * (1 - abar_prev[step]) * x
                ) / (1 - abar[step])
        x = mean if step == 0 else mean + beta[step].sqrt() * torch.randn_like(x)
        if step in keep:
            frames[step] = x.clone()
```

```
    return (x, frames) if keep else x
net = Denoiser()
net.load_state_dict(torch.load('denoiser.pt'))
net.eval()
final, frames = sample(net, n=1, seed=11, keep=(150, 100, 50, 20))
for step in [150, 100, 50, 20]:
    print('%d steps still to go' % step)
    show(frames[step][0], step=3)
    print()
print('finished')
show(final[0], step=3)
```

```
150 steps still to go
 @-: **%.%= @= @%@:+  @ -@-@
@ @ :+===+  =  : -@=@@@= @=-
. @=-=@* +.@ #-@*%-@=- # -@+
@#=@#+.#.::@*+@@= +@@@@ @ %@
*@ -%+ @.:#@@     @ @@ .=. *
.@#+ # .   @@ @@=- %. -=#:*%
%@+: +-:@@ @+@=@ @@ +%%@+=**
=+@ ++ =%=* @@@:=:@:-  ::=.
@+: *#:#@:@@#+*@-   -.*%-+=+
 .@# +:@%@ .% .@@@:*%:=@*#@:

100 steps still to go
@-*:.**%-%=:=#@+**:   @ @-
:#@ :*  *.   :=- :@@##+@.%*
.=@. --+ @%@  #@@#%# . # #+:
##-*+.%*+ .@+@.@  -@+@+ # .
*=-:@* : +=:# =   # - . *+.-
*@. +% =  ##@@-@-* #  =%@@ #
* -:   @@+-%*#+# *+%:%%##  .
%:@ #@ @@=+@@*+. +@:#.+.%**
@    @%@+.:%++ @ -%:-=-* # =
*:-:- #@#%  @.   -=+*=*  %**

50 steps still to go
+=-  --- :==-=#%***-. +.==#
:#+.:.  =.*: :-  +#@ -#- . =
.:.:+:=- ++- :@#@#+**: :-:-
*-+==#=*+:@@@@@@ =*@+**:. *-
:#**-%+  . -+ .   @.-.-.-=+.
+=-:#-=-+ +*@@%@%  ..  #=++=
 :=  . +%%* @*=  .-=.#-   .:
# *.:@ @@@%@@## .:@@@ *-=-*
    =:-*##.=: --:#@+.@:==- -
*: . :#@-*--*@ + +#.*+-+  @%

20 steps still to go
.. :::.:---.=:*-=-=-.::+=:+:
.-=.==-=+ .. -=+-=*=:=+* :::
-::-----::=.:-+*#+#+=-.:-:*=
+=:-*==***@#@@@@:*@@@@++=..:
=---=+*.    -     +-..:+-:::
---:=..:-+#+@@@@@ ..= =*==-.
.-.-.-.*@@=+@%= . -*#:=-:::=
+:+-:+:%%@%@%-- =+@@++=**-=:
```

```
.::-=+:-*-:-.:=*+###-*.:+:.=
--=-.--=.-:=++:=:-=.=-=== :-

finished
-:--:-:-:--:--------:---==-:
--=:::==+--.:--=+*+=----::--
-:----==--- .=++%###=:.-:--=
-=:==:=#%@%@%@@%-+%@@@+*+:--
-=---=+*:     .   ..=-..-*=::-
:--::...-=#+%@@@@ ... =+==:-
::-- --*@%:+@@+ . -+=:==--:=
-:==:--%@@@@@=. =*@@#=-=---:
-::-=-==#+:- .-=+*+*+=::=-:-
:-=:::--:=:-===--==:-=---:-=
```

With 150 and 100 steps left there is nothing to see. By 50 a rough layout has appeared, by 20 it is clearly a digit, and the remaining steps only tidy it. Almost all of the visible decision happens in a narrow band in the middle of the chain, which is the observation behind every fast sampler: most of those 200 network calls are spent on very little.

Day 7 Which One, and What It Costs

Coverage and compute side by side, and the checklist

Two methods, the same dataset, the same judge from day 3. Here is what each one bought.

Coverage

The same count as day 3, over the same number of samples, so the entropies are directly comparable.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
class Block(nn.Module):
    """One residual block, told at every level how noisy the input is."""
    def __init__(self, h):
```

```
        super().__init__()
        self.norm = nn.LayerNorm(h)
        self.tproj = nn.Linear(h, h)
        self.net = nn.Sequential(nn.SiLU(), nn.Linear(h, h),
                                 nn.SiLU(), nn.Linear(h, h))

    def forward(self, x, temb):
        return x + self.net(self.norm(x) + self.tproj(temb))

class Denoiser(nn.Module):
    """Given a noisy image and how noisy it is, predict the noise."""
    def __init__(self, hidden=768, blocks=3):
        super().__init__()
        self.inp = nn.Linear(784, hidden)
        self.temb = nn.Embedding(T, hidden)
        self.blocks = nn.ModuleList([Block(hidden) for _ in range(blocks)])
        self.out = nn.Sequential(nn.LayerNorm(hidden), nn.SiLU(),
                                 nn.Linear(hidden, 784))

    def forward(self, x, t):
        temb = self.temb(t)
        h = self.inp(x)
        for block in self.blocks:
            h = block(h, temb)
        return self.out(h)

def add_noise(x0, t, eps):
    a = abar[t].reshape(-1, 1)
    return a.sqrt() * x0 + (1 - a).sqrt() * eps

def train_diffusion(epochs=200, seed=0, trace_every=0):
    torch.manual_seed(seed)
    net = Denoiser()
    opt = torch.optim.AdamW(net.parameters(), lr=1e-3)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, epochs)
    for ep in range(1, epochs + 1):
        total = n = 0.0
        for xb, _ in loader:
            x0 = xb.flatten(1)
            t = torch.randint(0, T, (len(x0),))
            eps = torch.randn_like(x0)
            loss = ((net(add_noise(x0, t, eps), t) - eps) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        sched.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    net.eval()
    return net

@torch.no_grad()
```

```
def sample(net, n=8, seed=3, keep=()):
    torch.manual_seed(seed)
    x = torch.randn(n, 784)
    frames = {}
    for step in range(T - 1, -1, -1):
        t = torch.full((n,), step, dtype=torch.long)
        # what the network thinks the clean image is, from here
        x0 = ((x - (1 - abar[step]).sqrt() * net(x, t))
              / abar[step].sqrt()).clamp(-1, 1)
        # the posterior mean: a small step from x towards that guess
        mean = (abar_prev[step].sqrt() * beta[step] * x0
                + alpha[step].sqrt() * (1 - abar_prev[step]) * x
                ) / (1 - abar[step])
        x = mean if step == 0 else mean + beta[step].sqrt() * torch.randn_like(x)
        if step in keep:
            frames[step] = x.clone()
    return (x, frames) if keep else x
def train_judge():
    """A plain digit classifier, used only to score what the GAN makes."""
    torch.manual_seed(1)
    clf = nn.Sequential(nn.Flatten(),
                        nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    opt = torch.optim.AdamW(clf.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    for _ in range(12):
        for xb, yb in loader:
            opt.zero_grad()
            lf(clf(xb), yb).backward()
            opt.step()
    clf.eval()
    test = Subset(datasets.MNIST('data', train=False, download=True,
                                 transform=tf), range(2000))
    tl = DataLoader(test, batch_size=512)
    right = seen = 0
    with torch.no_grad():
        for xb, yb in tl:
            right += (clf(xb).argmax(1) == yb).sum().item()
            seen += len(yb)
    return clf, right / seen

def coverage(G, clf, n=2000):
    with torch.no_grad():
        imgs = G(torch.randn(n, G.zdim)).reshape(-1, 1, 28, 28)
        pred = clf(imgs).argmax(1)
    counts = torch.bincount(pred, minlength=10).float()
    frac = counts / counts.sum()
    ent = -(frac[frac > 0] * frac[frac > 0].log()).sum().item()
    return counts.long().tolist(), ent
clf, acc = train_judge()
net = Denoiser()
net.load_state_dict(torch.load('denoiser.pt'))
net.eval()
imgs = torch.cat([sample(net, n=500, seed=s) for s in [21, 22, 23, 24]])
```

```
with torch.no_grad():
    pred = clf(imgs.reshape(-1, 1, 28, 28)).argmax(1)
counts = torch.bincount(pred, minlength=10).float()
frac = counts / counts.sum()
ent = -(frac[frac > 0] * frac[frac > 0].log()).sum().item()
print('judge accuracy on real digits %.4f' % acc)
print()
print('%-26s %-34s %8s' % ('model', 'digits produced out of 2000', 'entropy'))
print('%-26s %-34s %8.3f'
      % ('diffusion', ' '.join('%3d' % c for c in counts.long().tolist()),
         ent))
```

```
judge accuracy on real digits 0.9075

model                      digits produced out of 2000         entropy
diffusion                  245  42 259 224 129 282 190 359 221  49    2.167
```

Cost

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

tf = transforms.Compose([transforms.ToTensor(),
                         transforms.Normalize((0.5,), (0.5,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
train_set = Subset(train_all, range(8000))
loader = DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True)

RAMP = ' .:-=+*#%@'
def show(flat, step=2):
    """Print one flattened image, scaled from [-1, 1] back to characters."""
    t = (flat.detach().reshape(28, 28) + 1) / 2
    for row in t[::step]:
        print(''.join(RAMP[min(9, max(0, int(v * 9.999)))]
                      for v in row.tolist()))
T = 200
beta = torch.linspace(1e-4, 0.05, T)
alpha = 1 - beta
abar = torch.cumprod(alpha, 0)
abar_prev = torch.cat([torch.ones(1), abar[:-1]])
class Block(nn.Module):
    """One residual block, told at every level how noisy the input is."""
    def __init__(self, h):
        super().__init__()
        self.norm = nn.LayerNorm(h)
        self.tproj = nn.Linear(h, h)
        self.net = nn.Sequential(nn.SiLU(), nn.Linear(h, h),
                                 nn.SiLU(), nn.Linear(h, h))

    def forward(self, x, temb):
```

```
        return x + self.net(self.norm(x) + self.tproj(temb))

class Denoiser(nn.Module):
    """Given a noisy image and how noisy it is, predict the noise."""
    def __init__(self, hidden=768, blocks=3):
        super().__init__()
        self.inp = nn.Linear(784, hidden)
        self.temb = nn.Embedding(T, hidden)
        self.blocks = nn.ModuleList([Block(hidden) for _ in range(blocks)])
        self.out = nn.Sequential(nn.LayerNorm(hidden), nn.SiLU(),
                                 nn.Linear(hidden, 784))

    def forward(self, x, t):
        temb = self.temb(t)
        h = self.inp(x)
        for block in self.blocks:
            h = block(h, temb)
        return self.out(h)

def add_noise(x0, t, eps):
    a = abar[t].reshape(-1, 1)
    return a.sqrt() * x0 + (1 - a).sqrt() * eps

def train_diffusion(epochs=200, seed=0, trace_every=0):
    torch.manual_seed(seed)
    net = Denoiser()
    opt = torch.optim.AdamW(net.parameters(), lr=1e-3)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, epochs)
    for ep in range(1, epochs + 1):
        total = n = 0.0
        for xb, _ in loader:
            x0 = xb.flatten(1)
            t = torch.randint(0, T, (len(x0),))
            eps = torch.randn_like(x0)
            loss = ((net(add_noise(x0, t, eps), t) - eps) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        sched.step()
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    net.eval()
    return net

@torch.no_grad()
def sample(net, n=8, seed=3, keep=()):
    torch.manual_seed(seed)
    x = torch.randn(n, 784)
    frames = {}
    for step in range(T - 1, -1, -1):
        t = torch.full((n,), step, dtype=torch.long)
        # what the network thinks the clean image is, from here
```

```
        x0 = ((x - (1 - abar[step]).sqrt() * net(x, t))
              / abar[step].sqrt()).clamp(-1, 1)
        # the posterior mean: a small step from x towards that guess
        mean = (abar_prev[step].sqrt() * beta[step] * x0
                + alpha[step].sqrt() * (1 - abar_prev[step]) * x
                ) / (1 - abar[step])
        x = mean if step == 0 else mean + beta[step].sqrt() * torch.randn_like(x)
        if step in keep:
            frames[step] = x.clone()
    return (x, frames) if keep else x
import time
net = Denoiser()
net.load_state_dict(torch.load('denoiser.pt'))
net.eval()
start = time.perf_counter()
_ = sample(net, n=64, seed=1)
diff_t = time.perf_counter() - start
params = sum(p.numel() for p in net.parameters())
print('the denoiser holds %d parameters' % params)
print('one reverse pass runs the network %d times' % T)
print('64 images took %.1f seconds on this machine' % diff_t)
print('a generator produces 64 images in a single forward pass')
```

```
the denoiser holds 6680848 parameters
one reverse pass runs the network 200 times
64 images took 2.2 seconds on this machine
a generator produces 64 images in a single forward pass
```

Watch Out

Timings are the one number here that is not portable

Everything else on this page is arithmetic that will reproduce. A wall-clock reading depends on the machine, the thread count and what else is running. Take the ratio as the finding, not the seconds.

How to choose

	GAN	Diffusion
Training stability	An equilibrium between two networks, with failure modes that do not appear in the loss	One ordinary regression, which either converges or does not
Loss you can read	No	Yes, against a known baseline of 1.0
Sampling cost	One forward pass	One forward pass per step, 200 of them here
Compute to reach these samples	30 epochs	200 epochs, and still improving
Mode coverage here, out of 2.303	2.275	2.167
What goes wrong	Silent collapse onto part of the data	Blur and residual noise, visible immediately

Two things in that count are worth naming. Diffusion came out slightly lower than the GAN, which is the opposite of the usual claim, and the honest reading is that the difference is small, the

judge is imperfect, and the diffusion model was still improving when the budget ran out. More interesting is *where* it lost: ones and nines arrive about a fifth as often as they should. A one is mostly background, so a model that leaves a haze everywhere blurs the thing that makes a one a one. The failure has a cause, and it is the same residual noise day 6 pointed at.

What the table does support is the shape of the trade. The GAN reached comparable coverage in a seventh of the epochs, and the diffusion model reached it without any of the instability, without a second network, and with a loss that could be read against a known baseline the whole way.

That is roughly why the field moved. At small scale a GAN is quick and often good enough. As models grow, an adversarial equilibrium becomes steadily harder to hold and a regression stays a regression, and sampling cost turns out to be the easier problem to attack, which is what the fast samplers did.

The checklist

- Before anything else, get a baseline number for your loss. For a denoiser it is 1.0. For a GAN there is not one, which is itself the thing to plan around.
- Never judge a generative model by three pictures. Count what a classifier calls a few thousand samples and read the spread.
- For a GAN, watch discriminator accuracy rather than either loss, and treat 1.0 or 0.5 as the two ways it is going wrong.
- Use the non-saturating generator loss. The alternative delivers around a seventieth of the gradient when it matters most.
- For diffusion, print abar at your final step and confirm it is near zero before you spend anything on training.
- Condition both networks on the label, not just the generator, or the label will be ignored.
- Expect conditioning to be partial. Measure agreement against a judge and know the number before you promise anybody control.

Where this leaves the course

Self-Assessment

Try It Yourself

1. Why does replacing a pixel loss with a discriminator remove the blur?

   a) The discriminator has more parameters

   b) A pixel loss is minimised by averaging over every plausible output, and an average of several digits does not fool a network trained to spot fakes

   c) The generator uses a Tanh output

   d) Adversarial training uses a larger learning rate

2. What happens if you forget to `detach()` the generated batch in the discriminator step?

a) Nothing, the call is an optimisation

b) The run crashes with a shape error

c) The gradients are doubled

d) The generator is updated towards making its own output easier to detect, and the run produces nothing while looking entirely normal

3. Over week 13's thirty-epoch run the generator loss rose while the samples clearly improved. What does that show?

   a) Both losses are measured against a moving opponent, so neither is a progress signal the way a supervised loss is

   b) The generator was overfitting

   c) The run was broken

   d) The learning rate was too high

4. With the discriminator catching every fake, how did the saturating generator loss compare with the non-saturating one?

   a) The saturating loss gave a slightly larger gradient

   b) The saturating loss delivered a gradient roughly seventy times smaller, which is nearly nothing exactly when it is most needed

   c) Neither produced any gradient

   d) They were within a few percent

5. Which change collapsed coverage the most in the measured comparison?

   a) Five discriminator steps per generator step

   b) A latent vector of only two dimensions

   c) A discriminator learning ten times faster

   d) The saturating generator loss

6. How was mode collapse actually measured?

   a) By the discriminator's accuracy

   b) By reading the generator loss

   c) By eye, over a grid of samples

   d) By classifying two thousand generated images with a separately trained judge and taking the entropy of the resulting counts against ln(10)

7. Why must a conditional GAN give the label to the discriminator and not only the generator?

   a) Because the embedding must be shared

   b) It does not matter which network receives it

   c) To balance the parameter counts

   d) Otherwise the discriminator only asks whether the image looks like a digit, and the generator is free to ignore the label entirely

8. Before training a diffusion model, what should you print and check?

   a) The learning rate schedule

b) The batch size

c) abar at the final step, which must be close to zero, or sampling starts from a state the model was never trained on

d) The number of parameters

Answers:1b, 2d, 3a, 4b, 5b, 6d, 7d, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 14

Self-Supervised Learning

Week 14

OBJECTIVES

Training an encoder on data nobody labelled, and measuring honestly whether it was worth it.

The largest models in use were pretrained on data nobody annotated, by manufacturing a task out of the data itself. This week builds three versions of that idea on images and measures all of them against two baselines, one of which is almost never reported and changes the conclusion completely. One method beats supervised training when labels are scarce. Another leaves the encoder worse than it was before training started.

Day 1 Learning From Data Nobody Labelled

The encoder, the linear probe, and the two baselines that decide everything

Weeks 12 and 13 generated data without labels. The images were the product. This week keeps the label-free training and throws the generated output away, because what is actually wanted is the *encoder*: a network that turns an input into a vector, trained on data nobody annotated, and then reused for a task where labels are expensive.

This is where the large models get their general ability. Nobody labelled the text a language model is pretrained on. The label was manufactured out of the data itself, by hiding part of it and asking the model to fill it in. The same trick applies to images, and this week builds three versions of it and measures all of them.

The setup

Twelve thousand MNIST images whose labels we will pretend not to have, and a held-out test set. Every experiment this week trains on the same pool and is scored the same way.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
```

```
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
print('unlabelled pool %s' % (tuple(unlab_x.shape),))
print('test set        %s' % (tuple(test_x.shape),))
print('batches per epoch %d' % len(pool))
```

```
unlabelled pool (12000, 1, 28, 28)
test set        (2000, 1, 28, 28)
batches per epoch 46
```

The encoder

One small convolutional network, used unchanged by every method below, so that differences in the results are differences between the methods rather than between architectures.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
```

```
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
torch.manual_seed(0)
enc = Encoder()
print(enc)
print()
print('%d parameters' % sum(p.numel() for p in enc.parameters()))
print('output shape %s' % (tuple(enc(unlab_x[:4]).shape),))
```

```
Encoder(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): GroupNorm(8, 32, eps=1e-05, affine=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): GroupNorm(8, 64, eps=1e-05, affine=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): GroupNorm(8, 64, eps=1e-05, affine=True)
    (10): ReLU()
    (11): Flatten(start_dim=1, end_dim=-1)
    (12): Linear(in_features=3136, out_features=64, bias=True)
  )
)

256832 parameters
output shape (4, 64)
```

One choice in there is deliberate and worth explaining: group normalisation rather than batch normalisation. This encoder gets trained under one regime and evaluated under another, often on batches of a completely different size, and week 4 spent a day on what BatchNorm does in that situation. Since the choice is easy to argue about and cheap to settle, here it is settled:

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
```

```
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
def build(norm):
    """The same encoder twice, differing only in the normalisation."""
    def n(c):
        return nn.BatchNorm2d(c) if norm == 'batch' else nn.GroupNorm(8, c)
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), n(32), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), n(64), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(64, 64, 3, padding=1), n(64), nn.ReLU(),
        nn.Flatten(), nn.Linear(64 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def compare(norm, steps=1500, seed=0):
    torch.manual_seed(seed)
    model = build(norm)
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x, unlab_y), batch_size=128,
                        shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    with torch.no_grad():
        model.eval()
        proper = (model(test_x).argmax(1) == test_y).float().mean()
        # what the same weights score using each batch's own statistics
        model.train()
        batched = (model(test_x).argmax(1) == test_y).float().mean()
    return float(proper), float(batched)
print('%-14s %18s %18s' % ('normalisation', 'running stats',
                           'batch stats'))
for norm in ['batch', 'group']:
    proper, batched = compare(norm)
    print('%-14s %18.4f %18.4f' % (norm, proper, batched))
```

```
normalisation       running stats        batch stats
batch                      0.9870             0.9845
group                      0.9860             0.9860
```

On this architecture they are the same to within a rounding error, and batch normalisation's two columns agree, so there is no drama to report. Group normalisation is kept anyway, on the grounds that it cannot develop a dependence on batch composition later, and it costs nothing to

have that guarantee.

How everything will be scored

Two baselines matter, and skipping either one is how people convince themselves a pretext task worked when it did not. The first is training the whole encoder supervised on the same labels, which is what you would do if you never heard of any of this.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

```

```python
def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
print('%10s %14s' % ('labels', 'test accuracy'))
for n in [100, 500, 2000, 12000]:
    print('%10d %14.4f' % (n, supervised(n)))
```

```
    labels  test accuracy
       100         0.7320
       500         0.9275
      2000         0.9610
     12000         0.9860
```

Watch Out

Budget in steps, not epochs

The first version of this comparison ran a fixed thirty epochs at every label count. With 100 labels that is 30 optimiser steps and with 12000 it is 2820, so the small settings were not underperforming for want of data, they were underperforming for want of training. Every run here gets 1500 steps regardless of how much data it has, which is the only version of the comparison that means anything.

The second baseline is the one that is almost always missing: the same encoder with its weights left exactly as initialised, never trained on anything at all.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
```

```
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
torch.manual_seed(0)
enc = Encoder()
print('%10s %14s' % ('labels', 'probe accuracy'))
for n in [100, 500, 2000, 12000]:
    print('%10d %14.4f' % (n, linear_probe(enc, n)))
```

```
    labels probe accuracy
```

```
   100        0.6340
   500        0.7780
  2000        0.8205
 12000        0.8390
```

A random convolutional stack, with no training whatsoever, reaches 0.839 with a linear layer on top. That is not a quirk of MNIST. Random convolutions are a genuinely strong feature extractor, because a random filter bank still measures edges and textures at every position, and pooling still summarises them.

> **The number every result this week has to beat**

Day 2 A Pretext Task That Made Things Worse

Rotation prediction, solved in three epochs, and what that cost

The first pretext task to try is the one that sounds cleverest. Rotate each image by a quarter, a half or three quarters of a turn, and train the encoder to say which. The label is free, since it is whatever rotation the code just applied, and the argument for it is appealing: to know that a digit has been turned upside down, surely you have to know something about what the digit is.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
```

```
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
```

```
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def rotate_batch(x):
    """Four copies of every image, one per quarter turn, with the turn
    itself as the label. No human wrote any of these labels down."""
    outs, labels = [], []
    for k in range(4):
        outs.append(torch.rot90(x, k, dims=(2, 3)))
        labels.append(torch.full((len(x),), k, dtype=torch.long))
    return torch.cat(outs), torch.cat(labels)

def train_rotation(epochs=12, seed=0, trace_every=0):
    torch.manual_seed(seed)
    enc = Encoder()
    head = nn.Linear(64, 4)
    opt = torch.optim.AdamW(list(enc.parameters()) + list(head.parameters()),
                            lr=1e-3)
    lf = nn.CrossEntropyLoss()
    for ep in range(1, epochs + 1):
        enc.train()
        right = seen = 0
        for (xb,) in pool:
            rx, ry = rotate_batch(xb)
            logits = head(enc(rx))
            opt.zero_grad()
            lf(logits, ry).backward()
            opt.step()
            right += (logits.argmax(1) == ry).sum().item()
            seen += len(ry)
        if trace_every and ep % trace_every == 0:
            print('%6d %20.4f' % (ep, right / seen))
    return enc
x = unlab_x[:2]
rx, ry = rotate_batch(x)
print('2 images became %d, with labels %s'
      % (len(rx), ry.reshape(4, 2)[:, 0].tolist()))
```

```
2 images became 8, with labels [0, 1, 2, 3]
```

Training it

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
```

```
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

```

```
    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def rotate_batch(x):
    """Four copies of every image, one per quarter turn, with the turn
    itself as the label. No human wrote any of these labels down."""
    outs, labels = [], []
    for k in range(4):
        outs.append(torch.rot90(x, k, dims=(2, 3)))
        labels.append(torch.full((len(x),), k, dtype=torch.long))
    return torch.cat(outs), torch.cat(labels)

def train_rotation(epochs=12, seed=0, trace_every=0):
    torch.manual_seed(seed)
    enc = Encoder()
    head = nn.Linear(64, 4)
    opt = torch.optim.AdamW(list(enc.parameters()) + list(head.parameters()),
                            lr=1e-3)
    lf = nn.CrossEntropyLoss()
    for ep in range(1, epochs + 1):
        enc.train()
        right = seen = 0
        for (xb,) in pool:
            rx, ry = rotate_batch(xb)
            logits = head(enc(rx))
            opt.zero_grad()
            lf(logits, ry).backward()
            opt.step()
            right += (logits.argmax(1) == ry).sum().item()
            seen += len(ry)
        if trace_every and ep % trace_every == 0:
            print('%6d %20.4f' % (ep, right / seen))
    return enc
print('%6s %20s' % ('epoch', 'rotation accuracy'))
```

```
enc = train_rotation(epochs=12, trace_every=3)
print()
print('%10s %14s' % ('labels', 'probe accuracy'))
for n in [100, 500, 2000, 12000]:
    print('%10d %14.4f' % (n, linear_probe(enc, n)))
```

```
epoch    rotation accuracy
    3                0.9931
    6                0.9961
    9                0.9978
   12                0.9984

   labels probe accuracy
      100         0.6020
      500         0.7315
     2000         0.7740
    12000         0.7970
```

Read the two halves of that output against each other, because together they are the whole lesson of the day. The pretext task is solved: 99.3 percent by epoch three, 99.8 by the end. And the representation it produced is worse than the untrained encoder at every single label count, by four to five points.

Watch Out

Twelve epochs of training made the encoder worse than not training it

0.797 against 0.839 at 12000 labels, 0.602 against 0.634 at 100. This is not a small effect and it is not noise. The encoder spent its capacity becoming good at something, and that something was not useful, so the random features it started with were degraded rather than improved.

Why it failed, and how you could have known

The diagnostic is sitting in the first column. A pretext task that reaches 99.3 percent after three epochs is not asking the encoder for much. Deciding which way up an MNIST digit is turned can be done from where the ink sits and which way the strokes lean, and neither of those requires any notion of which digit it is. The encoder found the cheapest solution available, exactly as it was asked to.

So the rule is not that rotation prediction is a bad idea. It is that a pretext task is only as good as the shortcut it fails to leave open, and the way you find the shortcut is to watch how fast the task is solved. If your pretext accuracy saturates almost immediately, the task is too easy and the representation will be disappointing.

This is the honest history of the field

Rotation prediction, jigsaw puzzles and colourisation were all real published methods, and they were all overtaken. The reason was not that anyone was careless. It was that the relationship between a pretext task and the representation it produces is not something you can reason your way to, and the field found this out by measuring.

Day 3 Filling In What You Hid

Masked patches, and why the same idea is stronger for text

The second task takes the language model idea directly. Hide part of the input, ask the model to reconstruct it, and the hidden part is its own label. Here that means blanking six of the sixteen 7 by 7 patches of each image.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
```

```
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def mask_batch(x, patch=7, blank=6):
    """Blank six of the sixteen 7 by 7 patches, independently per image."""
    n, grid = len(x), 28 // patch
    # argsort of uniform noise is a batched randperm
    cells = torch.rand(n, grid * grid).argsort(1)[:, :blank]
    keep = torch.ones(n, grid * grid)
    keep.scatter_(1, cells, 0.0)
    keep = keep.reshape(n, 1, grid, grid)
    keep = keep.repeat_interleave(patch, 2).repeat_interleave(patch, 3)
    return x * keep

class MaskDecoder(nn.Module):
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, 256), nn.ReLU(),
```

```
                                nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, h):
        return self.net(h).reshape(-1, 1, 28, 28)

def train_masked(epochs=25, seed=0, trace_every=0):
    torch.manual_seed(seed)
    enc, dec = Encoder(), MaskDecoder()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(dec.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            loss = ((dec(enc(mask_batch(xb))) - xb) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc
torch.manual_seed(0)
x = unlab_x[:1]
for img in [x, mask_batch(x)]:
    t = img[0, 0]
    for row in t[::2]:
        print(''.join(' .:-=+*#%@'[min(9, int(v * 9.999))]
                      for v in row.tolist()))
    print()
```

```
        ..-**@@@@@%*@@#:
        %@@@@@##@@
           *@-
            #@:
             -@@@=
                -@@#
              .+#@@%
           =%@@@@#-
        *%@@@@#-
     +@@@%++




        ..-**@
        %@@@@@##@@
           *@-
            #@:
             -@@@=
                -@@#
```

```
            .+#@@%
          =%@@@@#-
```

This is a harder question than which way up it is. To fill a blank square in the middle of a digit you have to know what kind of stroke was passing through it, and that depends on the rest of the image.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
```

```
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def mask_batch(x, patch=7, blank=6):
    """Blank six of the sixteen 7 by 7 patches, independently per image."""
    n, grid = len(x), 28 // patch
    # argsort of uniform noise is a batched randperm
    cells = torch.rand(n, grid * grid).argsort(1)[:, :blank]
    keep = torch.ones(n, grid * grid)
    keep.scatter_(1, cells, 0.0)
    keep = keep.reshape(n, 1, grid, grid)
    keep = keep.repeat_interleave(patch, 2).repeat_interleave(patch, 3)
    return x * keep

class MaskDecoder(nn.Module):
    def __init__(self, dim=64):
        super().__init__()
```

```
        self.net = nn.Sequential(nn.Linear(dim, 256), nn.ReLU(),
                                 nn.Linear(256, 784), nn.Sigmoid())

    def forward(self, h):
        return self.net(h).reshape(-1, 1, 28, 28)

def train_masked(epochs=25, seed=0, trace_every=0):
    torch.manual_seed(seed)
    enc, dec = Encoder(), MaskDecoder()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(dec.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            loss = ((dec(enc(mask_batch(xb))) - xb) ** 2).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc
print('%6s %14s' % ('epoch', 'pixel MSE'))
enc = train_masked(epochs=25, trace_every=5)
print()
print('%10s %14s' % ('labels', 'probe accuracy'))
for n in [100, 500, 2000, 12000]:
    print('%10d %14.4f' % (n, linear_probe(enc, n)))
```

```
 epoch      pixel MSE
     5         0.0254
    10         0.0182
    15         0.0157
    20         0.0145
    25         0.0137

    labels probe accuracy
       100         0.6025
       500         0.7890
      2000         0.8585
     12000         0.8910
```

Better, and still not good

0.891 at 12000 labels, against 0.839 for the untrained encoder. So it did learn something, which rotation prediction did not. It is also still far behind the 0.986 that plain supervised training reaches on the same labels, and at 100 labels it is level with the random encoder and behind supervised training by thirteen points.

The reason is the loss. Reconstructing pixels rewards getting the background right, and most of an MNIST image is background. A great deal of the capacity goes into a skill that has nothing to do with telling digits apart, which is the same complaint week 12 ended with and it has the

same cause.

Why it works so much better for text

Masked prediction is the method behind an enormous amount of modern language modelling, so its middling showing here needs explaining rather than generalising.

Filling in a missing word requires the meaning of the sentence, because there is no low-level cue that gets you there. There is no equivalent of a blank background in text. Filling in a missing patch of image very often only requires knowing that the neighbourhood was dark. The method did not change between the two, the redundancy of the data did.

Day 4 Same Thing, Different View

Contrastive learning, the augmentations, and the NT-Xent loss

The third approach stops asking the encoder to reconstruct anything. Take one image, make two different corrupted views of it, and ask only that those two views land near each other and away from every other image in the batch. Nothing is predicted and nothing is reconstructed. The only requirement is on the geometry of the representation.

The augmentations are the whole design

This is the part that decides everything. The encoder learns to ignore whatever the augmentation varies, so the choice of augmentation is a direct statement about what you consider irrelevant. Shift the digit and the encoder learns position does not matter. Scrub a square out of it and the encoder learns not to depend on any one region.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
```

```
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
```

```
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def augment(x):
    """Two different views of the same digit: shift it, scrub an 8 by 8
    square out of it, and add a little noise.

    Written with advanced indexing rather than a loop over the batch. The
    loop version is easier to read and about twenty times slower, and
    this runs twice per batch for every epoch.
    """
    n = len(x)
    idx = torch.arange(28)
    pad = torch.nn.functional.pad(x, (3, 3, 3, 3))
    rows = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    out = out.unsqueeze(1)
    cy, cx = torch.randint(0, 21, (2, n, 1, 1))
    ys, xs = idx.reshape(1, 28, 1), idx.reshape(1, 1, 28)
    hole = (ys >= cy) & (ys < cy + 8) & (xs >= cx) & (xs < cx + 8)
    out = out * (~hole).unsqueeze(1)
    return (out + 0.05 * torch.randn_like(out)).clamp(0, 1)

class Projector(nn.Module):
    """The head the loss is applied to, thrown away afterwards."""
    def __init__(self, dim=64, out=32):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, dim), nn.ReLU(),
                                 nn.Linear(dim, out))

    def forward(self, h):
        return nn.functional.normalize(self.net(h), dim=1)

def nt_xent(z1, z2, temperature=0.5):
    """Each view should be closer to its partner than to anything else
    in the batch."""
    z = torch.cat([z1, z2])
    n = len(z1)
    sim = z @ z.t() / temperature
    sim.fill_diagonal_(-1e9)
    target = torch.cat([torch.arange(n, 2 * n), torch.arange(0, n)])
    return nn.functional.cross_entropy(sim, target)

def train_contrastive(epochs=30, seed=0, trace_every=0, temperature=0.5,
                      negatives=True):
    torch.manual_seed(seed)
    enc, proj = Encoder(), Projector()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(proj.parameters()),
                            lr=1e-3)
```

```
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            z1, z2 = proj(enc(augment(xb))), proj(enc(augment(xb)))
            if negatives:
                loss = nt_xent(z1, z2, temperature)
            else:
                # pull the pair together and ask for nothing else
                loss = ((z1 - z2) ** 2).sum(1).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc, proj
torch.manual_seed(0)
views = augment(unlab_x[:1].repeat(2, 1, 1, 1))
for img in views:
    for row in img[0][::2]:
        print(''.join(' .:-=+*#%@'[min(9, int(v * 9.999))]
                      for v in row.tolist()))
    print()
```

```
                        .
         . .
                    #:#@@=
          :@@         :=-.
           -+
              +@*
                @%*=   .
                 :#@@*.
                    @@@:
                .+@@@@@%
              :%@@@@#-..
        :*%@@@@@+.

          .                  .




          ::-=#@@@@@%%@@%.
          %@@@@@##@@


          .         =.
                    @@#
                  .+#@@%
 .              =%@@@@#:
        #%@@@@#:
    .+@@@%*+.    .
             .
```

Watch Out

An augmentation that changes the answer is a bug

Week 4 made this point about supervised training and it is sharper here. A vertical flip would teach this encoder that a 6 and a 9 are the same thing, and nothing in the loss would object, because the loss never sees a digit label. The failure would only appear later, as a probe that cannot separate two classes and no explanation of why.

The loss

For a batch of n images there are 2n views. Each one has exactly one partner and 2n minus 2 non-partners, so scoring which of the others is the partner is an ordinary classification problem with 2n minus 1 options, and cross entropy handles it. The similarities are dot products of normalised vectors, divided by a temperature.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
```

```
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def augment(x):
    """Two different views of the same digit: shift it, scrub an 8 by 8
    square out of it, and add a little noise.

```

```
    Written with advanced indexing rather than a loop over the batch. The
    loop version is easier to read and about twenty times slower, and
    this runs twice per batch for every epoch.
    """
    n = len(x)
    idx = torch.arange(28)
    pad = torch.nn.functional.pad(x, (3, 3, 3, 3))
    rows = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    out = out.unsqueeze(1)
    cy, cx = torch.randint(0, 21, (2, n, 1, 1))
    ys, xs = idx.reshape(1, 28, 1), idx.reshape(1, 1, 28)
    hole = (ys >= cy) & (ys < cy + 8) & (xs >= cx) & (xs < cx + 8)
    out = out * (~hole).unsqueeze(1)
    return (out + 0.05 * torch.randn_like(out)).clamp(0, 1)

class Projector(nn.Module):
    """The head the loss is applied to, thrown away afterwards."""
    def __init__(self, dim=64, out=32):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, dim), nn.ReLU(),
                                 nn.Linear(dim, out))

    def forward(self, h):
        return nn.functional.normalize(self.net(h), dim=1)

def nt_xent(z1, z2, temperature=0.5):
    """Each view should be closer to its partner than to anything else
    in the batch."""
    z = torch.cat([z1, z2])
    n = len(z1)
    sim = z @ z.t() / temperature
    sim.fill_diagonal_(-1e9)
    target = torch.cat([torch.arange(n, 2 * n), torch.arange(0, n)])
    return nn.functional.cross_entropy(sim, target)

def train_contrastive(epochs=30, seed=0, trace_every=0, temperature=0.5,
                      negatives=True):
    torch.manual_seed(seed)
    enc, proj = Encoder(), Projector()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(proj.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            z1, z2 = proj(enc(augment(xb))), proj(enc(augment(xb)))
            if negatives:
                loss = nt_xent(z1, z2, temperature)
            else:
                # pull the pair together and ask for nothing else
                loss = ((z1 - z2) ** 2).sum(1).mean()
```

```
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc, proj
torch.manual_seed(0)
z1 = nn.functional.normalize(torch.randn(4, 32), dim=1)
print('a random pairing scores %.4f' % nt_xent(z1, z1[torch.randperm(4)]))
print('a perfect pairing scores %.4f' % nt_xent(z1, z1))
print('chance for a batch of 4 is ln(7) = %.4f'
      % float(torch.tensor(7.0).log()))
```

```
a random pairing scores 2.1051
a perfect pairing scores 0.5204
chance for a batch of 4 is ln(7) = 1.9459
```

Two details in that loss are easy to get wrong. The diagonal has to be removed, or every view trivially matches itself and there is nothing to learn. And the vectors have to be normalised before the dot product, or the model can raise every similarity by making its outputs longer instead of by arranging them better.

Day 5 Where It Finally Pays

The crossover against supervised training, and reading a loss that stays hard

Train it on the same pool, for the same kind of budget, and probe it the same way.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
```

```
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
```

```
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def augment(x):
    """Two different views of the same digit: shift it, scrub an 8 by 8
    square out of it, and add a little noise.

    Written with advanced indexing rather than a loop over the batch. The
    loop version is easier to read and about twenty times slower, and
    this runs twice per batch for every epoch.
    """
    n = len(x)
    idx = torch.arange(28)
    pad = torch.nn.functional.pad(x, (3, 3, 3, 3))
    rows = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    out = out.unsqueeze(1)
    cy, cx = torch.randint(0, 21, (2, n, 1, 1))
    ys, xs = idx.reshape(1, 28, 1), idx.reshape(1, 1, 28)
    hole = (ys >= cy) & (ys < cy + 8) & (xs >= cx) & (xs < cx + 8)
    out = out * (~hole).unsqueeze(1)
    return (out + 0.05 * torch.randn_like(out)).clamp(0, 1)

class Projector(nn.Module):
    """The head the loss is applied to, thrown away afterwards."""
    def __init__(self, dim=64, out=32):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, dim), nn.ReLU(),
                                 nn.Linear(dim, out))

    def forward(self, h):
        return nn.functional.normalize(self.net(h), dim=1)

def nt_xent(z1, z2, temperature=0.5):
    """Each view should be closer to its partner than to anything else
    in the batch."""
    z = torch.cat([z1, z2])
    n = len(z1)
    sim = z @ z.t() / temperature
    sim.fill_diagonal_(-1e9)
    target = torch.cat([torch.arange(n, 2 * n), torch.arange(0, n)])
    return nn.functional.cross_entropy(sim, target)

def train_contrastive(epochs=30, seed=0, trace_every=0, temperature=0.5,
                      negatives=True):
    torch.manual_seed(seed)
    enc, proj = Encoder(), Projector()
```

```
    opt = torch.optim.AdamW(list(enc.parameters()) + list(proj.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            z1, z2 = proj(enc(augment(xb))), proj(enc(augment(xb)))
            if negatives:
                loss = nt_xent(z1, z2, temperature)
            else:
                # pull the pair together and ask for nothing else
                loss = ((z1 - z2) ** 2).sum(1).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc, proj
print('%6s %14s' % ('epoch', 'NT-Xent loss'))
enc, proj = train_contrastive(epochs=30, trace_every=5)
torch.save(enc.state_dict(), 'con_enc.pt')
torch.save(proj.state_dict(), 'con_proj.pt')
print()
print('%10s %14s' % ('labels', 'probe accuracy'))
for n in [100, 500, 2000, 12000]:
    print('%10d %14.4f' % (n, linear_probe(enc, n)))
```

```
 epoch   NT-Xent loss
     5         4.6325
    10         4.5591
    15         4.5258
    20         4.5084
    25         4.4926
    30         4.4806

    labels probe accuracy
       100         0.7860
       500         0.9330
      2000         0.9530
     12000         0.9635
```

At 100 labels this beats training the encoder supervised on those same 100 labels, 0.786 against 0.732. At 500 it is ahead again, 0.933 against 0.928. At 2000 and above supervised training pulls back in front, ending 0.986 against 0.964.

That crossover is the entire practical case for the method

Self-supervised pretraining is not a way to beat supervised learning. It is a way to spend unlabelled data, which is usually abundant, to reduce how much labelled data you need, which usually is not. When labels are plentiful it loses, and it is supposed to.

The crossover here sits somewhere between 500 and 2000 labels. Where it sits for your problem depends on how hard the task is and how much unlabelled data you have, and the only way to

find it is the table above.

Reading the loss

The NT-Xent loss went from 4.63 to 4.48 over thirty epochs. Chance for a batch of 256 is ln(511), about 6.24, so the model is well clear of guessing, and it is also nowhere near solving the task. Compare that with rotation prediction, which was at 99.3 percent after three epochs. The task that stayed hard is the task that produced the useful representation, and the loss was still falling when the budget ran out.

Day 6 Collapse, Negatives and the Discarded Head

The degenerate solution, what prevents it, and why the loss sees a different layer

There is an obvious way for a contrastive encoder to cheat. If it maps every input to the same vector, then the two views of an image are certainly close together, and the loss that asks for closeness is perfectly satisfied. The representation is worthless.

What actually prevents it

The negatives. Every other image in the batch is a thing this view has to be far from, so a constant output is now the worst possible answer rather than the best. Take that term away and keep everything else, and here is what happens:

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
```

```
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
```

```
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def augment(x):
    """Two different views of the same digit: shift it, scrub an 8 by 8
    square out of it, and add a little noise.

    Written with advanced indexing rather than a loop over the batch. The
    loop version is easier to read and about twenty times slower, and
    this runs twice per batch for every epoch.
    """
    n = len(x)
    idx = torch.arange(28)
    pad = torch.nn.functional.pad(x, (3, 3, 3, 3))
    rows = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    out = out.unsqueeze(1)
    cy, cx = torch.randint(0, 21, (2, n, 1, 1))
    ys, xs = idx.reshape(1, 28, 1), idx.reshape(1, 1, 28)
    hole = (ys >= cy) & (ys < cy + 8) & (xs >= cx) & (xs < cx + 8)
    out = out * (~hole).unsqueeze(1)
    return (out + 0.05 * torch.randn_like(out)).clamp(0, 1)

class Projector(nn.Module):
    """The head the loss is applied to, thrown away afterwards."""
    def __init__(self, dim=64, out=32):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, dim), nn.ReLU(),
                                 nn.Linear(dim, out))

    def forward(self, h):
        return nn.functional.normalize(self.net(h), dim=1)

def nt_xent(z1, z2, temperature=0.5):
    """Each view should be closer to its partner than to anything else
    in the batch."""
    z = torch.cat([z1, z2])
    n = len(z1)
    sim = z @ z.t() / temperature
    sim.fill_diagonal_(-1e9)
    target = torch.cat([torch.arange(n, 2 * n), torch.arange(0, n)])
    return nn.functional.cross_entropy(sim, target)

def train_contrastive(epochs=30, seed=0, trace_every=0, temperature=0.5,
                      negatives=True):
    torch.manual_seed(seed)
    enc, proj = Encoder(), Projector()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(proj.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
```

```
        for (xb,) in pool:
            z1, z2 = proj(enc(augment(xb))), proj(enc(augment(xb)))
            if negatives:
                loss = nt_xent(z1, z2, temperature)
            else:
                # pull the pair together and ask for nothing else
                loss = ((z1 - z2) ** 2).sum(1).mean()
            opt.zero_grad()
            loss.backward()
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc, proj
def spread(enc):
    """How different are the embeddings from each other at all."""
    f = features(enc, unlab_x[:2000])
    return f.std(0).mean().item()

torch.manual_seed(0)
fresh = Encoder()
print('%-34s %12s %14s' % ('', 'spread', 'probe at 500'))
print('%-34s %12.4f %14.4f'
      % ('untrained encoder', spread(fresh), linear_probe(fresh, 500)))
for negatives in [True, False]:
    enc, _ = train_contrastive(epochs=15, negatives=negatives)
    name = 'with negatives' if negatives else 'positives only'
    print('%-34s %12.4f %14.4f'
          % (name, spread(enc), linear_probe(enc, 500)))
```

```
                                    spread   probe at 500
untrained encoder                   0.1877         0.7780
with negatives                      1.9468         0.9110
positives only                      0.0812         0.2300
```

The positives-only run does exactly what the argument predicts. Its embeddings have a spread of 0.081, less than half the untrained encoder's 0.188 and a twenty-fourth of what the proper loss produces, which is to say the outputs have become nearly constant. Its probe lands at 0.230, against 0.778 for an encoder that was never trained at all. Fifteen epochs of training destroyed the representation, and the objective it was given was satisfied throughout.

Watch Out

A falling loss is not evidence of anything here

This is the same warning as week 13, arriving from a different direction. In self-supervised training the loss is over a task you invented, and a model can drive it down by finding a degenerate solution to your invented task. Print the spread of the embeddings, and probe against the untrained encoder. Neither costs anything and together they catch this immediately.

Methods that avoid negatives

Negatives are expensive: the loss wants large batches, because a batch of 256 offers 510 things to be unlike and a batch of 32 offers 62. A family of later methods removed them, using an asymmetry between the two branches instead, typically a slowly-updated copy of the encoder on one side and a stop-gradient so that only the other side learns. The point to carry away is not the mechanism but that every one of them exists to solve the problem measured above, and any of them can be checked the same way.

The head you throw away

The loss was not applied to the encoder's output. It was applied to a small projection on top, which is then discarded, and the encoder underneath is what gets used. That looks like a pointless extra layer until you probe both:

```
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset, TensorDataset
from torchvision import datasets, transforms

tf = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf)

def stack(dataset, n, start=0):
    """One pass, so images and labels stay together."""
    xs = torch.stack([dataset[i][0] for i in range(start, start + n)])
    ys = torch.tensor([dataset[i][1] for i in range(start, start + n)])
    return xs, ys

# 12000 images whose labels we pretend not to have, and a held out set
unlab_x, unlab_y = stack(train_all, 12000)
test_x, test_y = stack(test_all, 2000)
pool = DataLoader(TensorDataset(unlab_x), batch_size=256, shuffle=True,
                  drop_last=True)
class Encoder(nn.Module):
    """The part we are trying to train without labels.

    GroupNorm rather than BatchNorm, deliberately. This encoder gets
    trained one way and evaluated another, often on batches of a very
    different size, and BatchNorm's running statistics do not survive
    that. Day 1 measures what it costs.
    """
    def __init__(self, dim=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.GroupNorm(8, 32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 64, 3, padding=1), nn.GroupNorm(8, 64), nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, dim))

```

```
    def forward(self, x):
        return self.net(x)
@torch.no_grad()
def features(enc, x, batch=512):
    enc.eval()
    return torch.cat([enc(x[i:i + batch]) for i in range(0, len(x), batch)])

def linear_probe(enc, n_labels, epochs=300, seed=0):
    """Freeze the encoder, fit one linear layer on n_labels examples."""
    torch.manual_seed(seed)
    tr_x, tr_y = unlab_x[:n_labels], unlab_y[:n_labels]
    ftr, fte = features(enc, tr_x), features(enc, test_x)
    head = nn.Linear(ftr.shape[1], 10)
    opt = torch.optim.AdamW(head.parameters(), lr=1e-2, weight_decay=1e-4)
    lf = nn.CrossEntropyLoss()
    for _ in range(epochs):
        opt.zero_grad()
        lf(head(ftr), tr_y).backward()
        opt.step()
    with torch.no_grad():
        return (head(fte).argmax(1) == test_y).float().mean().item()

def supervised(n_labels, steps=1500, seed=0):
    """The same encoder trained from scratch on the same labels.

    Budgeted in optimiser steps rather than epochs, so that 100 labels
    and 12000 labels each get the same amount of training. A fixed epoch
    count gives the small settings almost no steps and makes them look
    far worse than they are.
    """
    torch.manual_seed(seed)
    model = nn.Sequential(Encoder(), nn.Linear(64, 10))
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(unlab_x[:n_labels],
                                      unlab_y[:n_labels]),
                        batch_size=128, shuffle=True)
    done = 0
    while done < steps:
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
            done += 1
    model.eval()
    with torch.no_grad():
        return (model(test_x).argmax(1) == test_y).float().mean().item()
def augment(x):
    """Two different views of the same digit: shift it, scrub an 8 by 8
    square out of it, and add a little noise.

    Written with advanced indexing rather than a loop over the batch. The
    loop version is easier to read and about twenty times slower, and
```

```
    this runs twice per batch for every epoch.
    """
    n = len(x)
    idx = torch.arange(28)
    pad = torch.nn.functional.pad(x, (3, 3, 3, 3))
    rows = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 7, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    out = out.unsqueeze(1)
    cy, cx = torch.randint(0, 21, (2, n, 1, 1))
    ys, xs = idx.reshape(1, 28, 1), idx.reshape(1, 1, 28)
    hole = (ys >= cy) & (ys < cy + 8) & (xs >= cx) & (xs < cx + 8)
    out = out * (~hole).unsqueeze(1)
    return (out + 0.05 * torch.randn_like(out)).clamp(0, 1)

class Projector(nn.Module):
    """The head the loss is applied to, thrown away afterwards."""
    def __init__(self, dim=64, out=32):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(dim, dim), nn.ReLU(),
                                 nn.Linear(dim, out))

    def forward(self, h):
        return nn.functional.normalize(self.net(h), dim=1)

def nt_xent(z1, z2, temperature=0.5):
    """Each view should be closer to its partner than to anything else
    in the batch."""
    z = torch.cat([z1, z2])
    n = len(z1)
    sim = z @ z.t() / temperature
    sim.fill_diagonal_(-1e9)
    target = torch.cat([torch.arange(n, 2 * n), torch.arange(0, n)])
    return nn.functional.cross_entropy(sim, target)

def train_contrastive(epochs=30, seed=0, trace_every=0, temperature=0.5,
                      negatives=True):
    torch.manual_seed(seed)
    enc, proj = Encoder(), Projector()
    opt = torch.optim.AdamW(list(enc.parameters()) + list(proj.parameters()),
                            lr=1e-3)
    for ep in range(1, epochs + 1):
        enc.train()
        total = n = 0.0
        for (xb,) in pool:
            z1, z2 = proj(enc(augment(xb))), proj(enc(augment(xb)))
            if negatives:
                loss = nt_xent(z1, z2, temperature)
            else:
                # pull the pair together and ask for nothing else
                loss = ((z1 - z2) ** 2).sum(1).mean()
            opt.zero_grad()
            loss.backward()
```

```
            opt.step()
            total += loss.item()
            n += 1
        if trace_every and ep % trace_every == 0:
            print('%6d %14.4f' % (ep, total / n))
    return enc, proj
enc, proj = Encoder(), Projector()
enc.load_state_dict(torch.load('con_enc.pt'))
proj.load_state_dict(torch.load('con_proj.pt'))

class Both(nn.Module):
    def __init__(self, enc, proj):
        super().__init__()
        self.enc, self.proj = enc, proj

    def forward(self, x):
        return self.proj(self.enc(x))

print('%-40s %14s' % ('features taken from', 'probe at 500'))
print('%-40s %14.4f' % ('the encoder, 64 wide',
                        linear_probe(enc, 500)))
print('%-40s %14.4f' % ('the projection the loss saw, 32 wide',
                        linear_probe(Both(enc, proj), 500)))
```

```
features taken from                          probe at 500
the encoder, 64 wide                           0.9330
the projection the loss saw, 32 wide           0.8530
```

The layer the loss was applied to is the worse representation. It has been shaped to discard everything the augmentations varied, which is precisely its job, and some of what the augmentations varied turns out to matter for classification. Keeping a layer between the encoder and the loss gives the encoder somewhere to retain that information while still satisfying the objective.

Day 7 The Whole Table

Four encoders compared, and the protocol worth reusing

Four ways of getting an encoder, one architecture, one pool of data, one probe.

labels	untrained	rotation	masked patches	contrastive	supervised
100	0.634	0.602	0.603	**0.786**	0.732
500	0.778	0.732	0.789	**0.933**	0.928
2000	0.821	0.774	0.859	0.953	**0.961**
12000	0.839	0.797	0.891	0.964	**0.986**

Three findings, and the second is the one worth carrying furthest.

1. Contrastive learning is the only one of the three that clearly helped, and it helped exactly where the theory says it should: below about a thousand labels, where it beat training the same encoder supervised on the same labels.

2. Rotation prediction was worse than not training at all, at every label count. A pretext task can actively damage a representation, and nothing in its own loss will tell you.

3. The untrained encoder is a serious competitor. It beat one of the three methods outright and matched a second at low label counts.

Watch Out

What this table is not

MNIST is close to the worst possible dataset for showing self-supervised learning at its best. It is small, clean, low resolution and nearly linearly separable in raw pixels, so the supervised baseline is very strong and there is little structure left for a pretext task to discover. On natural images at scale the gaps run the other way and the crossover point sits far higher.
The findings that do transfer are the method, not the numbers: always include an untrained encoder, watch how quickly the pretext task saturates, and check the spread of the embeddings.

Choosing a pretext task

- Ask what shortcut the task leaves open. If there is a low-level cue that solves it, the encoder will find that cue and learn nothing else.
- Watch the pretext metric. Saturating in the first few epochs is the single clearest sign that the task is too easy to be worth training on.
- For contrastive methods, the augmentations are the design. They are a statement of what you consider irrelevant, and an augmentation that changes the true label is a silent bug.
- Keep a projection head between the encoder and the loss, and probe the encoder rather than the projection.
- Measure the spread of the embeddings, and treat a falling loss with suspicion until you have.

The protocol, written out

1. Fix one architecture and one data pool for the whole comparison.
2. Probe the untrained encoder first, and write the number down.
3. Probe a supervised model trained on each label count you care about, budgeted in optimiser steps rather than epochs.
4. Train each pretext task, then probe with the encoder frozen.
5. Report the whole curve across label counts. A single label count can support whichever conclusion you were hoping for.

What the week actually establishes

Self-Assessment

Try It Yourself

1. What is a linear probe measuring?

    a) What a single linear layer can extract from a frozen encoder, which is only structure the encoder already made explicit

    b) The number of useful dimensions

    c) How quickly the encoder trains

    d) How well the encoder can be fine-tuned

2. Why does week 14 insist on probing an *untrained* encoder?

    a) To check the code runs

    b) To warm up the data loader

    c) Because random convolutional features are genuinely strong, so without that number you cannot tell whether a pretext task helped at all

    d) Because the weights need initialising twice

3. The untrained encoder reached 0.839 at 12000 labels. How did rotation prediction compare?

    a) Behind it, at 0.797, so training on that task made the representation worse than not training at all

    b) Well ahead, at about 0.95

    c) Slightly ahead

    d) Level with it

4. What was the warning sign that rotation prediction would disappoint?

    a) It needed too many epochs

    b) Its gradients exploded

    c) Its loss was noisy

    d) It reached 99.3 percent on the pretext task within three epochs, which means the task could be solved without any notion of digit identity

5. Why is masked prediction so much more effective for text than it was for these images?

    a) Filling in a missing word requires the meaning of the sentence, while filling in a missing image patch is often satisfied by knowing the neighbourhood was dark

    b) Text has no augmentation

    c) Images need colour to work

    d) Text models are larger

6. In contrastive learning, what do the augmentations actually specify?

    a) The learning rate

    b) The batch size needed

    c) What you are declaring irrelevant, since the encoder learns to ignore whatever they vary, which is why a vertical flip on digits is a bug

d) How many negatives are available

7. Training with only the positive term, and no negatives, produced what?

   a) Collapse: the embedding spread fell to 0.081 and the probe dropped to 0.230, well below the untrained encoder

   b) A slightly weaker representation

   c) The same representation more slowly

   d) Divergence and a NaN loss

8. Why is the projection head discarded and the encoder kept?

   a) The encoder trains faster

   b) It is an arbitrary convention

   c) The projection has fewer parameters

   d) The projection is shaped to throw away everything the augmentations varied, and some of that matters for classification, so it probes eight points worse

Answers:1a, 2c, 3a, 4d, 5a, 6c, 7a, 8d. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 15

Keras Alongside PyTorch

Week 15

OBJECTIVES

The same models written twice, measured against each other, and an honest account of when each framework is the right one.

Every week so far has been PyTorch, and a great deal of the code you will be asked to maintain is not. Keras 3 runs on TensorFlow, PyTorch or JAX, which makes this less of a choice than it used to be. This week builds the same convolutional network in both, trains it on the same data, compares what each one asked you to write, and is direct about the point where Keras stops saving you anything.

Day 1 The Other Framework

Keras 3, its backends, and the axis order that catches everybody

Fourteen weeks of this course have been written in PyTorch, and that was a choice rather than a necessity. Keras is the other framework you will meet, it is what a great deal of existing code is written in, and since version 3 it is no longer tied to TensorFlow at all. This week builds the same things twice and compares them on the same data.

The goal is not to decide which is better. It is to make you able to read either one, because the job that hands you a five year old Keras model to maintain does not care which you prefer.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
print('keras %s on the %s backend' % (keras.__version__,
                                      keras.backend.backend()))
print('train %s  test %s' % (xtr.shape, xte.shape))
```

```
keras 3.15.1 on the tensorflow backend
train (12000, 28, 28, 1)  test (2000, 28, 28, 1)
```

Watch Out

The backend is chosen before the import, not after

`KERAS_BACKEND` is read when `keras` is first imported. Setting it afterwards does nothing at all, silently. If a notebook has already imported Keras, you have to restart it, which is a genuinely common way to lose twenty minutes.

One difference that will bite you immediately

Keras puts the channel dimension last and PyTorch puts it first. An MNIST batch is `(batch, 28, 28, 1)` in Keras and `(batch, 1, 28, 28)` in PyTorch. Nothing warns you, the shapes are both valid, and a convolution will happily treat 28 channels of a 28 by 1 image as though that were sensible.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
import torch
print('keras batch %s' % (xtr[:8].shape,))
print('torch batch %s' % (torch.tensor(xtr[:8]).permute(0, 3, 1, 2).shape,))
print()
print('permute moves the axes without copying the data')
```

```
keras batch (8, 28, 28, 1)
torch batch torch.Size([8, 1, 28, 28])

permute moves the axes without copying the data
```

What Keras 3 actually is now

Day 2 Declaring a Model Instead of Running One

Sequential, the shape table, and the functional API

The most common way to write a Keras model is a list of layers, which is a near-exact match for `nn.Sequential`. The difference is what comes with it.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
```

```
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
def build_keras():
    return keras.Sequential([
        layers.Input((28, 28, 1)),
        layers.Conv2D(32, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10)])
model = build_keras()
model.summary()
```

```
Model: "sequential"
+---------------------------------+------------------------+---------------+
| Layer (type)                    | Output Shape           |       Param # |
+---------------------------------+------------------------+---------------+
| conv2d (Conv2D)                 | (None, 28, 28, 32)     |           320 |
+---------------------------------+------------------------+---------------+
| max_pooling2d (MaxPooling2D)    | (None, 14, 14, 32)     |             0 |
+---------------------------------+------------------------+---------------+
| conv2d_1 (Conv2D)               | (None, 14, 14, 64)     |        18,496 |
+---------------------------------+------------------------+---------------+
| max_pooling2d_1 (MaxPooling2D)  | (None, 7, 7, 64)       |             0 |
+---------------------------------+------------------------+---------------+
| flatten (Flatten)               | (None, 3136)           |             0 |
+---------------------------------+------------------------+---------------+
| dense (Dense)                   | (None, 64)             |       200,768 |
+---------------------------------+------------------------+---------------+
| dense_1 (Dense)                 | (None, 10)             |           650 |
+---------------------------------+------------------------+---------------+
 Total params: 220,234 (860.29 KB)
 Trainable params: 220,234 (860.29 KB)
 Non-trainable params: 0 (0.00 B)
```

PyTorch has nothing equivalent to that table out of the box. Printing an `nn.Sequential` gives you the layers you wrote, not the shapes they produce or the parameters they hold, because PyTorch does not know either until a tensor has been through. Keras knows because you declared the input shape.

Where the shapes come from

This is the deeper difference and it explains most of the others. Keras builds a description of the model first and runs it second, so it can compute every intermediate shape and tell you that layer four will receive 3136 features before you have any data. PyTorch defines the model by running it, which is why `nn.Linear(64 * 7 * 7, 64)` requires you to have done that arithmetic yourself.

Neither is strictly better. Working the shapes out by hand is annoying and it is also the reason a PyTorch model can change its structure per batch, which week 7's variable length sequences quietly relied on.

The functional API

A list of layers cannot express a model that branches or has more than one input. The functional API calls each layer on a tensor and lets you keep references to the intermediate results, which is the same thing a PyTorch `forward` does with local variables.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
inp = keras.Input((28, 28, 1))
x = layers.Conv2D(16, 3, padding='same', activation='relu')(inp)
x = layers.MaxPooling2D()(x)
x = layers.Flatten()(x)
digit = layers.Dense(10, name='digit')(x)
big = layers.Dense(1, name='is_big')(x)
model = keras.Model(inp, [digit, big])
print('inputs  %s' % [t.shape for t in model.inputs])
print('outputs %s' % [t.shape for t in model.outputs])
```

```
inputs  [(None, 28, 28, 1)]
outputs [(None, 10), (None, 1)]
```

The shortcut, the residual connection and the two-headed model from week 6 are all written this way. If you find yourself wanting something a `Sequential` cannot say, this is the next step up rather than a rewrite.

Day 3 The Training Loop You Do Not Write

compile and fit, and what compile is really doing

Week 1 spent a day writing a training loop by hand and week 2 spent a day making it respectable. Keras replaces the whole of that with two method calls, and it is worth being precise about what you gain and what you give up.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
```

```
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
def build_keras():
    return keras.Sequential([
        layers.Input((28, 28, 1)),
        layers.Conv2D(32, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10)])
import time
keras.utils.set_random_seed(0)
model = build_keras()
model.compile(optimizer=keras.optimizers.Adam(1e-3),
              loss=keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True),
              metrics=['accuracy'])
start = time.perf_counter()
hist = model.fit(xtr, ytr, epochs=5, batch_size=128, verbose=2,
                 validation_data=(xte, yte))
print()
print('keras took %.1f seconds' % (time.perf_counter() - start))
print('final validation accuracy %.4f'
      % hist.history['val_accuracy'][-1])
```

```
Epoch 1/5
94/94 - 5s - 48ms/step - accuracy: 0.8140 - loss: 0.6536 - val_accuracy: 0.9250 - val_loss:
    ↪ 0.2761
Epoch 2/5
94/94 - 4s - 39ms/step - accuracy: 0.9540 - loss: 0.1589 - val_accuracy: 0.9555 - val_loss:
    ↪ 0.1538
Epoch 3/5
94/94 - 4s - 40ms/step - accuracy: 0.9732 - loss: 0.0934 - val_accuracy: 0.9630 - val_loss:
    ↪ 0.1117
Epoch 4/5
94/94 - 4s - 40ms/step - accuracy: 0.9812 - loss: 0.0661 - val_accuracy: 0.9680 - val_loss:
    ↪ 0.0929
Epoch 5/5
94/94 - 4s - 40ms/step - accuracy: 0.9866 - loss: 0.0525 - val_accuracy: 0.9705 - val_loss:
    ↪ 0.0823

keras took 19.6 seconds
final validation accuracy 0.9705
```

Everything week 2 insisted on is in there. The validation split is evaluated every epoch, the metrics are tracked separately for training and validation, the model is switched between training and evaluation mode around the right calls, and the history is kept. None of it was written.

Watch Out

`from_logits=True` is not optional

The model above ends in a plain `Dense(10)` with no activation, so its outputs are logits. Week 2 made the case for that, and the loss has to be told. Leave the flag off and Keras takes the logits for probabilities, applies a log to numbers that may be negative, and trains something that is not what you meant. It does not raise anything.

What compile is actually doing

- Attaching the optimiser, so `fit` knows what to step.
- Attaching the loss, and building the metric objects that accumulate across a whole epoch rather than reporting the last batch.
- On the TensorFlow backend, tracing the training step into a graph, which is why the first epoch above is slower than the rest.

That last point explains a timing result people misread constantly. The first epoch pays for compilation, so a benchmark of one epoch makes Keras look worse than it is, and a benchmark that ignores it makes it look better.

Writing your own layer

A custom layer is the same shape of object as an `nn.Module`: declare the weights, say what the forward pass does. Keras splits it into `build`, which runs once when the input shape is known, and `call`, which is the forward pass. That split is what lets you write a layer without knowing its input width, which PyTorch makes you supply.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
class Scale(keras.layers.Layer):
    """One learned multiplier per feature."""
    def build(self, input_shape):
        # called the first time the layer sees data, so the width
        # is known here and did not have to be passed in
        self.w = self.add_weight(shape=(input_shape[-1],),
                                 initializer='ones', trainable=True)

    def call(self, x):
        return x * self.w

layer = Scale()
```

```
out = layer(np.ones((2, 5), dtype='float32'))
print('output %s' % (out.shape,))
print('weights created on first call: %s'
      % [tuple(w.shape) for w in layer.weights])
print('trainable: %d' % len(layer.trainable_weights))
```

```
output (2, 5)
weights created on first call: [(5,)]
trainable: 1
```

Where the abstraction stops paying

Everything above holds as long as your training step is the ordinary one. When it is not, you override `train_step`, and the code inside it is written against a particular backend rather than against Keras.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
import tensorflow as tf

class Noisy(keras.Model):
    """An ordinary model that adds noise to its inputs while training."""
    def __init__(self, inner):
        super().__init__()
        self.inner = inner

    def call(self, x, training=False):
        return self.inner(x)

    def train_step(self, data):
        x, y = data
        x = x + tf.random.normal(tf.shape(x), stddev=0.1)
        with tf.GradientTape() as tape:
            loss = self.compute_loss(y=y, y_pred=self(x, training=True))
        grads = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
                                           self.trainable_variables))
        return {'loss': loss}

keras.utils.set_random_seed(0)
model = Noisy(keras.Sequential([keras.layers.Input((28, 28, 1)),
                                keras.layers.Flatten(),
                                keras.layers.Dense(10)]))
```

```
model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True))
hist = model.fit(xtr[:2000], ytr[:2000], epochs=2, batch_size=128,
                 verbose=2)
print()
print('the loop ran, and every line inside train_step was TensorFlow')
```

```
Epoch 1/2
16/16 - 0s - 17ms/step - loss: 0.0000e+00
Epoch 2/2
16/16 - 0s - 4ms/step - loss: 0.0000e+00

the loop ran, and every line inside train_step was TensorFlow
```

Watch Out

A custom train_step is not portable across backends

`tf.GradientTape` is TensorFlow. On the PyTorch backend the same method is written with `loss.backward()` and an optimiser step, and on JAX it is a pure function returning gradients. So the moment you need a training step Keras does not already provide, you have picked a backend and the portability that was the reason for Keras 3 is gone.
This is the honest boundary of the abstraction, and it is worth knowing before you build on it rather than after.

Day 4 Callbacks

The real argument for fit, and the default that undoes it

The real argument for `fit` is not that it saves you twenty lines. It is callbacks: objects that hook into the loop at defined points, so behaviour you would otherwise hand-roll becomes an argument.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
def build_keras():
    return keras.Sequential([
        layers.Input((28, 28, 1)),
        layers.Conv2D(32, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
```

```
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10)])
keras.utils.set_random_seed(0)
model = build_keras()
model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True),
              metrics=['accuracy'])
cbs = [keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,
                                     restore_best_weights=True),
       keras.callbacks.ReduceLROnPlateau(monitor='val_loss', patience=1,
                                         factor=0.5)]
hist = model.fit(xtr, ytr, epochs=20, batch_size=128, verbose=2,
                 validation_data=(xte, yte), callbacks=cbs)
print()
print('stopped after %d epochs of a possible 20' % len(hist.history['loss']))
print('best validation accuracy %.4f' % max(hist.history['val_accuracy']))
```

```
Epoch 1/20
94/94 - 4s - 45ms/step - accuracy: 0.8140 - loss: 0.6536 - val_accuracy: 0.9250 - val_loss:
    ↪ 0.2761 - learning_rate: 0.0010
Epoch 2/20
94/94 - 4s - 42ms/step - accuracy: 0.9540 - loss: 0.1589 - val_accuracy: 0.9555 - val_loss:
    ↪ 0.1538 - learning_rate: 0.0010
Epoch 3/20
94/94 - 4s - 41ms/step - accuracy: 0.9732 - loss: 0.0934 - val_accuracy: 0.9630 - val_loss:
    ↪ 0.1117 - learning_rate: 0.0010
Epoch 4/20
94/94 - 4s - 42ms/step - accuracy: 0.9812 - loss: 0.0661 - val_accuracy: 0.9680 - val_loss:
    ↪ 0.0929 - learning_rate: 0.0010
Epoch 5/20
94/94 - 4s - 42ms/step - accuracy: 0.9866 - loss: 0.0525 - val_accuracy: 0.9705 - val_loss:
    ↪ 0.0823 - learning_rate: 0.0010
Epoch 6/20
94/94 - 4s - 44ms/step - accuracy: 0.9902 - loss: 0.0421 - val_accuracy: 0.9725 - val_loss:
    ↪ 0.0738 - learning_rate: 0.0010
Epoch 7/20
94/94 - 4s - 46ms/step - accuracy: 0.9916 - loss: 0.0329 - val_accuracy: 0.9770 - val_loss:
    ↪ 0.0659 - learning_rate: 0.0010
Epoch 8/20
94/94 - 4s - 45ms/step - accuracy: 0.9932 - loss: 0.0263 - val_accuracy: 0.9770 - val_loss:
    ↪ 0.0631 - learning_rate: 0.0010
Epoch 9/20
94/94 - 4s - 46ms/step - accuracy: 0.9948 - loss: 0.0213 - val_accuracy: 0.9810 - val_loss:
    ↪ 0.0601 - learning_rate: 0.0010
Epoch 10/20
94/94 - 4s - 48ms/step - accuracy: 0.9958 - loss: 0.0159 - val_accuracy: 0.9800 - val_loss:
    ↪ 0.0639 - learning_rate: 0.0010
Epoch 11/20
94/94 - 4s - 47ms/step - accuracy: 0.9964 - loss: 0.0119 - val_accuracy: 0.9775 - val_loss:
    ↪ 0.0653 - learning_rate: 5.0000e-04

stopped after 11 epochs of a possible 20
best validation accuracy 0.9810
```

Twenty epochs were requested and eleven were run. Validation loss bottomed out at epoch nine,

the learning rate was halved at eleven when it had not improved for one epoch, and training stopped when it had not improved for two. The weights returned are the ones from epoch nine, not from epoch eleven.

`restore_best_weights` defaults to False

Week 2 made this exact point in PyTorch: stopping is the easy half, and keeping the best weights is the half that matters. Keras will happily stop and hand you the weights from the worst of the last three epochs unless you ask otherwise. It is one keyword and it is the whole value of the callback.

The ones worth knowing

Callback	What it does	The PyTorch equivalent
`EarlyStopping`	Stops when a monitored metric stops improving	A patience counter you write
`ModelCheckpoint`	Saves every epoch, or only the best	`torch.save` inside your loop
`ReduceLROnPlateau`	Cuts the learning rate on a plateau	`ReduceLROnPlateau`, which exists in torch too
`CSVLogger`	Writes the history to a file	Whatever logging you set up in week 2
`TerminateOnNaN`	Stops when the loss goes to NaN	The check week 3 told you to add

There is nothing in that table PyTorch cannot do. The difference is that in Keras they are one argument each and already correct, and in PyTorch they are code you own, which means they are also code you can get subtly wrong. Week 3's warning about a scheduler stepping per batch instead of per epoch is exactly that class of mistake.

Day 5 The Same Model, Both Ways

Measured side by side, and Keras running on PyTorch

Same architecture, same data, same optimiser and learning rate, same number of epochs, on the same machine. The only thing that differs is the framework.

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)
def build_torch():
    return nn.Sequential(
```

```
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def train_torch(epochs=5, batch=128, lr=1e-3, seed=0):
    torch.manual_seed(seed)
    model = build_torch()
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=batch,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    with torch.no_grad():
        return model, (model(xte).argmax(1) == yte).float().mean().item()
import time
start = time.perf_counter()
model, acc = train_torch(epochs=5)
print('torch took %.1f seconds' % (time.perf_counter() - start))
print('final validation accuracy %.4f' % acc)
print('%d parameters' % sum(p.numel() for p in model.parameters()))
```

```
torch took 16.7 seconds
final validation accuracy 0.9625
220234 parameters
```

220,234 parameters in both, which is the check that the two models are genuinely the same and not merely similar. PyTorch finished in 16.7 seconds at 0.9625 and Keras in 19.6 at 0.9705.

Watch Out

Do not read either of those numbers as a verdict

The accuracy gap is one run each with different random initialisation, on 2000 test images. Week 3 was blunt about this: an improvement you cannot distinguish from run to run variation is not an improvement, and neither of these was repeated enough times to know. The timing includes Keras compiling its graph on the first epoch. What the comparison genuinely establishes is that the two are in the same place, which is the useful finding.

The same model, written twice

	Keras	PyTorch
Declare the model	`keras.Sequential([...])`	`nn.Sequential(...)`
Shapes	Inferred from `Input`	You compute them
Channel order	Last	First
Attach loss and optimiser	`model.compile(...)`	Objects you hold and call
The loop	`model.fit(...)`	Roughly fifteen lines you write
Train and eval mode	Handled	`model.train()` and `model.eval()`, and week 4 showed what forgetting costs
Custom behaviour mid-loop	A callback	Any Python you like, in the loop

Running Keras on PyTorch

Since Keras 3 the two are not exclusive. The same Keras code runs on the PyTorch backend, and what comes back is a torch tensor:

```
import os
os.environ['KERAS_BACKEND'] = 'torch'
import keras
from keras import layers
import numpy as np
print('keras is now running on %s' % keras.backend.backend())
keras.utils.set_random_seed(0)
model = keras.Sequential([layers.Input((4,)), layers.Dense(3)])
out = model(np.zeros((2, 4), dtype='float32'))
print('a Keras layer returned a %s' % type(out).__name__)
```

```
keras is now running on torch
a Keras layer returned a Tensor
```

This matters more than it looks. A Keras model on the torch backend is an `nn.Module`, so it can go inside a PyTorch model, be trained by a PyTorch loop, and use PyTorch tooling. The choice stopped being all or nothing.

Day 6 Saving, Loading and Handing It Over

One file against a state dictionary, and the formats deployment wants

Saving is where the two differ most, and where the difference actually affects how you deploy.

```
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'
import numpy as np
import keras
from keras import layers

(xtr, ytr), (xte, yte) = keras.datasets.mnist.load_data()
xtr = (xtr[:12000].astype('float32') / 255.0)[..., None]
ytr = ytr[:12000]
xte = (xte[:2000].astype('float32') / 255.0)[..., None]
yte = yte[:2000]
keras.utils.set_random_seed(0)
```

```
def build_keras():
    return keras.Sequential([
        layers.Input((28, 28, 1)),
        layers.Conv2D(32, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, padding='same', activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10)])
import os
keras.utils.set_random_seed(0)
model = build_keras()
model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True))
model.fit(xtr[:1000], ytr[:1000], epochs=1, batch_size=128, verbose=0)
model.save('m.keras')
again = keras.models.load_model('m.keras')
a = model.predict(xte[:64], verbose=0)
b = again.predict(xte[:64], verbose=0)
print('one file, %d KB' % (os.path.getsize('m.keras') // 1024))
print('predictions identical after reload: %s'
      % bool(np.allclose(a, b)))
print('no model class was needed to load it')
```

```
one file, 2617 KB
predictions identical after reload: True
no model class was needed to load it
```

That last line is the whole point. Week 2 was careful to say that a PyTorch `state_dict` is a mapping from names to tensors and nothing else, so loading it requires the class definition to be importable. A `.keras` file carries the architecture as well, so it can be loaded by code that has never seen your source.

Which is the right trade

It depends who loads it. Inside a codebase you control, the `state_dict` approach is arguably safer: the architecture lives in version control where you can review changes to it, rather than inside a binary. Handing a model to somebody else, the self describing file is obviously better, and it is the reason so much deployment tooling speaks Keras or ONNX rather than raw PyTorch.

Watch Out

Never load a model file you did not produce

This applies to both frameworks. Loading a serialised model can execute code from the file. `torch.load` now defaults to `weights_only=True` for exactly this reason, and Keras will run custom objects a file asks for. Treat a checkpoint from an untrusted source the way you would treat a script from one.

The formats you will meet

- **.keras**, architecture and weights in one file, loadable without your source.
- **state_dict**, weights only, needs the class.
- **SavedModel**, the TensorFlow serving format, which is what TensorFlow Serving and much of the cloud tooling expects.
- **ONNX**, a framework-neutral graph that both can export to, and the usual answer when the thing that runs the model is not written in Python. Week 16 uses it.

Day 7 Choosing, and Reading Both

Where each one wins, and a translation table

Both frameworks train the same model to the same place. The choice is about which problems you would rather have.

Where Keras is the better answer

- A standard architecture on standard data, where `fit` and three callbacks are the entire training script.
- A team where not everybody writes training loops, because the loop is the part that goes subtly wrong.
- Deployment into TensorFlow tooling, or handing a model to somebody without your codebase.
- Teaching, where the shape table and the declared input shape remove a class of confusion entirely.

Where PyTorch is the better answer

- Anything where the training step is the research: the two optimisers of week 13, the two-view loss of week 14, the gradient accumulation of week 3.
- Control flow that depends on the data, which the eager model makes ordinary Python.
- Reading and reusing recent published work, which is overwhelmingly PyTorch.
- Debugging, because a stack trace goes through your own code and there is no traced graph in between.

The honest summary

Keras is a good default for models that look like other models, and it gets less comfortable the further you get from that. PyTorch asks for more code at the start and stops asking for anything unusual later. Almost every difficult week of this course, from the adversarial loop to the contrastive loss, would have been written as a custom training step in Keras, at which point the thing that made Keras attractive is no longer being used.

Reading Keras code when you write PyTorch

Keras	PyTorch
`layers.Dense(n)`	`nn.Linear(in, n)`
`layers.Conv2D(c, k, padding='same')`	`nn.Conv2d(in, c, k, padding=k//2)`
`layers.BatchNormalization()`	`nn.BatchNorm2d(c)`
`layers.Dropout(r)`	`nn.Dropout(r)`
`SparseCategoricalCrossentropy(from_logits=True)`	`nn.CrossEntropyLoss()`
`BinaryCrossentropy(from_logits=True)`	`nn.BCEWithLogitsLoss()`
`model.fit(...)`	The loop from week 2
`model.predict(x)`	`model.eval()` and `torch.no_grad()`

The row worth memorising is the loss one. Both frameworks have a version that takes logits and a version that takes probabilities, both default in different directions, and getting it wrong produces a model that trains to something plausible and wrong rather than an error.

What to take from the week

Self-Assessment

Try It Yourself

1. You set `KERAS_BACKEND` after importing keras. What happens?

   a) An exception is raised
   b) Keras falls back to NumPy
   c) The backend switches
   d) Nothing at all, silently, because the variable is read at import time

2. How does an MNIST batch differ in shape between the two frameworks?

   a) Keras uses channels last, (batch, 28, 28, 1), and PyTorch channels first, (batch, 1, 28, 28)
   b) Keras flattens it automatically
   c) PyTorch requires a float64 batch
   d) It does not

3. Why can `model.summary()` print every output shape while printing an `nn.Sequential` cannot?

   a) Keras models are smaller
   b) PyTorch hides the information deliberately
   c) Keras is given the input shape and builds a description before running, whereas PyTorch defines the model by executing it
   d) It is a limitation of Python

4. Your Keras model ends in `Dense(10)` with no activation and you omit `from_logits=True`. What goes wrong?

a) The metrics are wrong but the weights are fine

b) An exception on the first batch

c) The loss treats logits as probabilities and trains something you did not intend, without raising anything

d) Training is merely slower

5. What does `restore_best_weights` default to in `EarlyStopping`, and why does it matter?

   a) False, so you are handed the weights from the epoch it stopped on rather than the best one

   b) True, but only when a validation split is given

   c) True, so there is nothing to think about

   d) It does not exist in Keras 3

6. Why is a one-epoch timing comparison against Keras misleading?

   a) The first epoch pays for tracing the training step into a graph, so it is slower than every epoch after it

   b) PyTorch preallocates memory

   c) Keras evaluates the validation set twice

   d) Keras caches the dataset

7. What can a `.keras` file do that a PyTorch `state_dict` cannot?

   a) Store the optimiser state

   b) Compress the weights

   c) Be loaded without the model class being importable, because it carries the architecture as well as the weights

   d) Be loaded on a different device

8. You override `train_step` in Keras. What have you given up?

   a) Backend portability, because the code inside it is written against one specific backend

   b) Nothing

   c) Access to callbacks

   d) The ability to use `compile`

Answers:1d, 2a, 3c, 4c, 5a, 6a, 7c, 8a. Anything you got wrong points at the day to re-read, not at a gap in ability.

Chapter 16

Making a Model Cheap Enough to Ship

Week 16

OBJECTIVES

Quantisation, pruning, distillation and export, each measured on the same machine

A model that is accurate and too expensive does not ship. This week takes one trained network and applies the four standard techniques for making it cheaper, measuring size, accuracy and latency for each on the same machine. Quantisation shrank it and made it slower. Pruning removed ninety percent of the weights and saved nothing. Distillation made the student worse than plain training. Export, the one usually filed under packaging rather than optimisation, made it four times faster for nothing.

Day 1 Accuracy Is Half the Problem

Size, latency and throughput, and how to measure them honestly

Every model in this course has been judged on accuracy. That is the right measure while you are finding out whether an idea works, and it is roughly half of what decides whether the model ships. The other half is how much memory it occupies, how long one prediction takes, and what that costs multiplied by the number of requests.

This week takes one trained model and applies the four standard techniques for making it cheaper, measuring each. Two of them will do considerably less than their reputation suggests, on this model and this machine, and the week says so.

The thing being optimised

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

```

```
xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
```

```
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
torch.manual_seed(0)
model = train(big())
torch.save(model.state_dict(), 'teacher.pt')
print('%-22s %10s %10s %12s' % ('', 'accuracy', 'size KB', 'latency ms'))
print('%-22s %10.4f %10.1f %12.3f'
      % ('the big model', accuracy(model), size_kb(model),
         latency_ms(model)))
print('%d parameters' % sum(p.numel() for p in model.parameters()))
```

```
                       accuracy    size KB   latency ms
the big model            0.9775     3223.5        0.310
824458 parameters
```

Watch Out

Measure the median, and warm up first

The first few forward passes through a fresh model are slower while allocators and caches settle, so a timing loop that includes them reports a number nobody will ever see. Every measurement here runs ten passes it throws away, then takes the median of two hundred more. The mean would be dragged around by occasional scheduler interruptions.

The four techniques

- **Quantisation**, storing weights in 8 bits instead of 32.
- **Pruning**, setting the least important weights to zero.
- **Distillation**, training a small model to imitate a large one.
- **Export**, taking the model out of Python into a runtime built for inference.

They are not alternatives. The last one composes with all the others, and as it turns out it is also the one that did the most here.

Day 2 Quantisation

Eight bits instead of thirty-two, and the speedup that went backwards

A float32 weight uses four bytes to store a number the network is largely indifferent to the fourth decimal place of. Quantisation stores it in one byte instead, with a scale factor per tensor to map the small integer range back onto the real one.

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
```

```
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
from torch.ao.quantization import quantize_dynamic
model = big()
model.load_state_dict(torch.load('teacher.pt'))
model.eval()
q = quantize_dynamic(copy.deepcopy(model), {nn.Linear}, dtype=torch.qint8)
print('%-22s %10s %10s %12s' % ('', 'accuracy', 'size KB', 'latency ms'))
for name, m in [('float32', model), ('int8 linear layers', q)]:
    print('%-22s %10.4f %10.1f %12.3f'
          % (name, accuracy(m), size_kb(m), latency_ms(m)))
```

```
                       accuracy    size KB   latency ms
float32                  0.9775     3223.5        0.259
int8 linear layers       0.9770      865.2        0.559
```

The size result is exactly as advertised: 3223 KB down to 865, a factor of 3.7, for five hundredths

of a percent of accuracy. If your constraint is the size of a download or the memory of a device, that is an excellent trade and there is very little to think about.

Watch Out

It got slower, not faster

0.259 ms became 0.559 ms. Quantisation is usually sold as a speed improvement and here it doubled the latency.
The reason is that int8 matrix multiplication is only faster when the matrix is large enough for the arithmetic to dominate. These layers are small, so the run is dominated by converting activations to int8 on the way in and back to float on the way out, which is pure overhead. On a large language model the same change is a substantial speedup. On this one it is a regression.

That is the general shape of it. Quantisation reliably buys memory. Whether it buys speed depends on the size of your layers and whether the hardware has integer kernels worth using, and the only way to know is the measurement above.

The other kinds

	What it needs	When to use it
Dynamic	Nothing	First thing to try, especially for linear and recurrent layers
Static	A calibration pass over sample data	When you need activations quantised ahead of time too, usually for convolutional models
Quantisation aware training	A fine-tuning run	When the accuracy drop from the other two is too large

Day 3 Pruning

Ninety percent of the weights removed for free, and why the file did not shrink

Most of the weights in a trained network are small, and a weight near zero contributes almost nothing. Pruning sets the smallest ones to exactly zero and asks what that cost.

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
```

```
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
```

```
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
import torch.nn.utils.prune as prune
base = big()
base.load_state_dict(torch.load('teacher.pt'))
base.eval()
print('%10s %12s %12s' % ('sparsity', 'accuracy', 'zeros'))
for amount in [0.0, 0.5, 0.8, 0.9, 0.95, 0.99]:
    m = copy.deepcopy(base)
    targets = [(mod, 'weight') for mod in m
               if isinstance(mod, (nn.Linear, nn.Conv2d))]
    if amount > 0:
        prune.global_unstructured(targets, prune.L1Unstructured,
                                  amount=amount)
        for mod, nm in targets:
            prune.remove(mod, nm)
    total = sum(p.numel() for p in m.parameters())
    zeros = sum(int((p == 0).sum()) for p in m.parameters())
    print('%10.2f %12.4f %11.1f%%'
          % (amount, accuracy(m), 100.0 * zeros / total))
```

```
  sparsity     accuracy        zeros
      0.00       0.9775         0.0%
      0.50       0.9770        50.0%
      0.80       0.9790        80.0%
      0.90       0.9780        90.0%
      0.95       0.9290        95.0%
      0.99       0.4980        99.0%
```

Ninety percent of the weights can be deleted with no retraining at all and accuracy holds. At eighty percent it is very slightly higher than the original, which is a regularisation effect and not something to get excited about at this margin. Then it falls off a cliff: 0.929 at ninety-five percent and 0.498 at ninety-nine.

That shape is the useful part. There is a wide plateau where pruning is free, followed by a sharp edge, and the edge is not where intuition puts it. You find it by sweeping, exactly as above, and you take a setting comfortably inside the plateau rather than at its boundary.

Watch Out

Ninety percent zeros made the file no smaller

Look at what pruning actually did: it wrote zeros into a dense tensor. The tensor has the same shape, occupies the same memory and takes the same time to multiply. A dense zero costs exactly as much as a dense anything else.
Turning sparsity into savings needs either a sparse storage format with kernels that exploit it, or structured pruning that removes whole channels so the tensor genuinely gets smaller. Unstructured pruning on its own is a research result, not a deployment technique, and it is very often reported as though it were the second thing.

Structured pruning

Removing an entire output channel of a convolution, rather than scattered individual weights, leaves a smaller dense tensor that every existing kernel handles at full speed. It costs more accuracy per parameter removed, because you are deleting useful weights alongside useless ones, and it is the version that actually makes a model faster. If you want speed, this is the one.

Day 4 Distillation

Learning from a teacher's whole distribution, and a negative result checked twice

The third technique changes the model rather than compressing it. Train a small network, but instead of showing it only the correct label, show it what a large trained network believes.

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
```

```
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
```

```
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
teacher = big()
teacher.load_state_dict(torch.load('teacher.pt'))
teacher.eval()
with torch.no_grad():
    probs = nn.functional.softmax(teacher(xte[:1]) / 4.0, dim=1)[0]
print('true label %d' % yte[0])
print('the teacher, softened, says:')
for digit, p in sorted(enumerate(probs.tolist()), key=lambda t: -t[1])[:5]:
    print('   %d  %.4f' % (digit, p))
print()
print('the one-hot label says 1.0 for %d and 0.0 for everything else'
      % yte[0])
```

```
true label 7
the teacher, softened, says:
   7  0.8008
   2  0.0750
   3  0.0533
   9  0.0221
   8  0.0196

the one-hot label says 1.0 for 7 and 0.0 for everything else
```

That is the argument in one output. The label says this is a seven and says nothing else. The teacher says it is a seven, and that if it were not, a two or a three would be the next most reasonable readings, and a zero would be absurd. The claim is that the second set carries more information per example, so a small model can learn more from the same data.

Measured

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)
```

```

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
```

```
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
teacher = big()
teacher.load_state_dict(torch.load('teacher.pt'))
teacher.eval()
print('%-30s %10s %10s %12s'
      % ('', 'accuracy', 'size KB', 'latency ms'))
print('%-30s %10.4f %10.1f %12.3f'
      % ('teacher', accuracy(teacher), size_kb(teacher),
         latency_ms(teacher)))
alone = train(small(), seed=1)
print('%-30s %10.4f %10.1f %12.3f'
      % ('student, labels only', accuracy(alone), size_kb(alone),
         latency_ms(alone)))
taught = train(small(), seed=1, teacher=teacher)
print('%-30s %10.4f %10.1f %12.3f'
      % ('student, distilled', accuracy(taught), size_kb(taught),
         latency_ms(taught)))
```

```
                              accuracy    size KB   latency ms
teacher                         0.9775     3223.5        0.271
student, labels only            0.8905      101.6        0.028
student, distilled              0.8730      101.6        0.031
```

The student is 32 times smaller and roughly 9 times faster, for about nine points of accuracy. That part is the real trade and it is often worth taking. But distillation made the student *worse* than training it on labels alone, which is the opposite of the entire point.

Checking that before believing it

One run at one setting is not enough to publish a negative result on, so here is a sweep over the mixing weight, at two seeds each.

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
```

```
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

```

```
def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
teacher = big()
teacher.load_state_dict(torch.load('teacher.pt'))
teacher.eval()
print('alpha 0 is labels only, alpha 1 is the teacher only')
print('%8s %12s %12s' % ('alpha', 'seed 1', 'seed 2'))
for alpha in [0.0, 0.3, 0.5, 0.7, 0.9]:
    accs = []
    for seed in [1, 2]:
        t = None if alpha == 0 else teacher
        m = train(small(), seed=seed, teacher=t, alpha=alpha)
        accs.append(accuracy(m))
    print('%8.1f %12.4f %12.4f' % (alpha, accs[0], accs[1]))
```

```
alpha 0 is labels only, alpha 1 is the teacher only
   alpha       seed 1       seed 2
     0.0       0.8905       0.8920
     0.3       0.8835       0.8795
     0.5       0.8835       0.8800
     0.7       0.8835       0.8785
     0.9       0.8820       0.8775
```

Watch Out

Consistent, monotonic, and in the wrong direction

Every alpha above zero is worse than alpha zero, at both seeds, and it gets worse the more weight the teacher is given. This is not noise and it is not one unlucky hyperparameter.

> The most likely explanation is capacity. A 784 to 32 to 10 network has nowhere near enough room to represent the teacher's function, so asking it to match a full ten-way distribution spends its very limited capacity on relationships it cannot use, at the expense of the decision boundary it was actually going to be scored on. Distillation helps when the student is small but not hopeless.

The lesson is not that distillation does not work. It works well enough that most deployed small language models are made this way. The lesson is that it is a technique with a range of validity, and that the honest way to find out whether you are inside it is two training runs and a table.

Day 5 Getting Out of Python

ONNX export, the parity check, and the largest win of the week

The fourth technique does not change the model at all. It takes the model out of Python.

A PyTorch model in eager mode dispatches every operation through the interpreter, one at a time, checking types and shapes as it goes. For training that flexibility is the whole point. For inference, where the graph never changes, it is pure overhead.

```
import os, time, copy
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

tf_ = transforms.ToTensor()
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def accuracy(model, x=None, y=None):
    model.eval()
    x = xte if x is None else x
    y = yte if y is None else y
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()

def size_kb(model, path='tmp.pt'):
    torch.save(model.state_dict(), path)
    kb = os.path.getsize(path) / 1024
    os.remove(path)
    return kb

def latency_ms(model, n=200, batch=1):
    """Median time for one forward pass, which is what a server sees."""
    model.eval()
    x = xte[:batch]
```

```
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return times[len(times) // 2]
def big():
    """The model we would like to ship but cannot afford to."""
    return nn.Sequential(
        nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
        nn.Linear(256, 10))

def small():
    """Something we could actually run on a phone."""
    return nn.Sequential(
        nn.Flatten(),
        nn.Linear(784, 32), nn.ReLU(),
        nn.Linear(32, 10))

def train(model, epochs=6, lr=1e-3, seed=0, teacher=None, alpha=0.5,
          temperature=4.0):
    torch.manual_seed(seed)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    hard = nn.CrossEntropyLoss()
    soft = nn.KLDivLoss(reduction='batchmean')
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            out = model(xb)
            loss = hard(out, yb)
            if teacher is not None:
                with torch.no_grad():
                    t = teacher(xb)
                # match the teacher's whole distribution, not its answer
                kd = soft(
                    nn.functional.log_softmax(out / temperature, dim=1),
                    nn.functional.softmax(t / temperature, dim=1))
                loss = (1 - alpha) * loss + alpha * temperature ** 2 * kd
            opt.zero_grad()
            loss.backward()
            opt.step()
    model.eval()
    return model
import numpy as np, onnxruntime as ort
model = big()
```

```
model.load_state_dict(torch.load('teacher.pt'))
model.eval()
# dynamo=False selects the long-standing tracing exporter, which needs
# no extra packages. The newer one requires onnxscript.
torch.onnx.export(model, (xte[:1],), 'm.onnx', dynamo=False,
                  input_names=['image'], output_names=['logits'],
                  dynamic_axes={'image': {0: 'batch'},
                                'logits': {0: 'batch'}})
sess = ort.InferenceSession('m.onnx',
                            providers=['CPUExecutionProvider'])
onnx_out = sess.run(None, {'image': xte[:256].numpy()})[0]
with torch.no_grad():
    torch_out = model(xte[:256]).numpy()
print('largest disagreement %.2e'
      % float(np.abs(onnx_out - torch_out).max()))
print('same predictions: %s'
      % bool((onnx_out.argmax(1) == torch_out.argmax(1)).all()))
print('file size %.1f KB' % (os.path.getsize('m.onnx') / 1024))
print()
one = xte[:1].numpy()
for _ in range(10):
    sess.run(None, {'image': one})
times = []
for _ in range(200):
    start = time.perf_counter()
    sess.run(None, {'image': one})
    times.append((time.perf_counter() - start) * 1000)
times.sort()
print('%-22s %10.3f ms' % ('onnxruntime', times[len(times) // 2]))
print('%-22s %10.3f ms' % ('pytorch eager', latency_ms(model)))
```

```
largest disagreement 3.81e-06
same predictions: True
file size 3221.9 KB

onnxruntime                0.117 ms
pytorch eager              0.509 ms
```

Four times faster, with predictions that are identical and numbers that agree to about four parts in a million. No accuracy was traded, no retraining happened, and the model was not modified. Of the four techniques in this week, this is the one that did the most and asked for the least.

Watch Out

Always check the exported model against the original

Export works by tracing: the exporter runs your model once and records the operations. Anything that depended on the specific input is baked in. A branch on a tensor value takes whichever path that example took, a shape computed from the data becomes a constant, and the exported graph then silently disagrees with your model on every other input.
The disagreement check above is three lines and it is the difference between finding that now and finding it from production traffic. Run it on a batch, not one example, and use `dynamic_axes` so the batch dimension is not frozen at whatever you traced with.

Day 6 All Four, Side by Side

The comparison table, and the order to try them in

Four techniques, one model, all measured on the same machine. Here is the whole picture.

Technique	Accuracy	Size	Latency	Verdict
None	0.9775	3223 KB	0.31 ms	The baseline
Dynamic quantisation	0.9770	865 KB	0.56 ms	3.7x smaller, and slower
Pruning to 90 percent	0.9780	3223 KB	unchanged	No saving at all without sparse kernels
Distillation to a small model	0.8730	102 KB	0.03 ms	32x smaller, 9 points worse, and worse than the same student trained on labels
ONNX export	0.9775	3222 KB	0.12 ms	4x faster, nothing given up

Read that table as a warning about where advice comes from. Three of these four techniques are described everywhere as ways to make models faster. On this model, on this hardware, one of them was slower, one changed nothing measurable, and the one that worked is the one that is least often described as an optimisation at all.

Which is not to say the others are useless

- Quantisation is the right answer when memory is the constraint, and it genuinely does accelerate large matrix multiplications, which is why every deployed language model uses it.
- Pruning becomes real when it is structured, or when the runtime has sparse kernels. The plateau it revealed, ninety percent of weights removable for nothing, is a true fact about the model regardless.
- Distillation is how most small production models are made. It needs a student with enough capacity to benefit, which this one did not.

The order to try them in

1. Export first. It is the cheapest to try, it composes with everything else, and it may end the exercise.
2. Then ask which resource is actually binding. Memory, latency and throughput want different techniques and people routinely optimise the wrong one.
3. Quantise if the answer is memory.
4. Reach for a smaller architecture, distilled if it helps, when export and quantisation are not enough. This is the only route to a large speedup and the only one that costs real accuracy.
5. Measure every step against the unmodified model on the same machine, and keep the accuracy check in the loop.

The rule this week is really about

Day 7 The Decisions That Matter More

Architecture, resolution, and profiling the whole request

Everything above optimised a model that had already been chosen. The larger decisions happen earlier, and they matter more.

The decisions that dominate

- **Model size.** Nothing in this week recovered the difference between the big model and the small one. Choosing an architecture that fits the budget beats compressing one that does not.
- **Input size.** Convolutional cost scales with the number of pixels. Halving the resolution is close to a four times saving and is frequently free in accuracy, and it is tried far less often than quantisation.
- **Whether the model runs at all.** Caching repeated requests, or a cheap filter that handles the easy cases and escalates only the hard ones, changes the arithmetic more than any of this.

Where the cost really lives in production

The measurements in this week are all of the forward pass. In a real service the forward pass is often not the largest term. Decoding a JPEG, resizing it, moving bytes over the network and serialising a response can each take longer than the model, and week 17 measures a service end to end rather than a model in isolation.

Profile before optimising, and profile the whole thing

A team that spends a week quantising a model whose latency is eighty percent image decoding has made the product four percent faster. This is the single most common way effort is wasted in deployment, and it is entirely avoidable by measuring the whole request path once before choosing what to work on.

The checklist

1. Measure the baseline: accuracy, size, and median latency with a warmup.
2. Export to ONNX and check the outputs match on a batch. Keep the speedup if the check passes.
3. Decide which resource is binding before choosing a technique.
4. Quantise for memory. Verify accuracy and re-measure latency, because it can go the wrong way.
5. Sweep pruning to find the plateau edge, and be honest about whether your runtime can exploit sparsity at all.
6. Consider a smaller architecture, and try distillation with at least two seeds before concluding anything about it.

7. Re-measure everything together at the end. The techniques interact, and the combination is not the sum of the parts.

What week 17 does with this

Self-Assessment

Try It Yourself

1. Dynamic quantisation shrank the model 3.7 times. What did it do to latency?

   a) Made it four times faster

   b) Halved it

   c) Left it unchanged

   d) Roughly doubled it, because these layers are too small for int8 arithmetic to outweigh the cost of converting activations

2. After pruning ninety percent of the weights to zero, how much smaller was the saved file?

   a) About half the size

   b) Ninety percent smaller

   c) Ten times smaller

   d) Not smaller at all, because a dense tensor of the same shape stores a zero exactly as expensively as any other number

3. What does pruning need in order to actually save anything?

   a) More epochs after pruning

   b) Either a sparse storage format with kernels that exploit it, or structured pruning that removes whole channels

   c) Quantisation applied first

   d) A lower learning rate

4. Why is temperature used when distilling?

   a) To slow the student down

   b) It flattens the teacher's distribution so the small probabilities become visible, since a sharp teacher output carries little more than the label

   c) To prevent overfitting

   d) It scales the learning rate

5. Distillation was measured across five alphas and two seeds. What did it show?

   a) Every alpha above zero was worse than labels alone, consistently and monotonically, most likely because the student had too little capacity

   b) A consistent gain of about two points

   c) No measurable difference

d) Results too noisy to interpret

6. Which technique gave the largest speedup in week 16, and at what cost?

   a) Pruning, at no cost

   b) Distillation, at nine points of accuracy

   c) Quantisation, at a small accuracy cost

   d) ONNX export, at no cost at all: four times faster with identical predictions

7. Why must an exported model be checked against the original on a batch?

   a) Export works by tracing one example, so data-dependent branches and shapes get baked in and the graph can silently disagree elsewhere

   b) ONNX uses lower precision

   c) The check is only needed for recurrent models

   d) Export changes the weights

8. What should you do before choosing a compression technique?

   a) Quantise, since it always helps

   b) Prune to ninety percent

   c) Work out which resource is actually binding, because memory, latency and throughput want different techniques

   d) Retrain from scratch

Answers:1d, 2d, 3b, 4b, 5a, 6d, 7a, 8c. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 17

Deployment and Monitoring

Week 17

OBJECTIVES

Putting a model behind an interface, and finding out whether it has quietly stopped working.

A model that is accurate and fast is still not a product. This week puts one behind a real HTTP interface, breaks its preprocessing four different ways to see what each costs, measures latency percentiles against batch size, and then asks the question that matters most: if the world changed and nobody sent you any labels, could you tell? The answer is yes, with an important caveat about the model claiming to be three-quarters confident while getting nine predictions in ten wrong.

Day 1 From a Function to a Service

The interface, where preprocessing belongs, and loading once

A trained model is a function from tensors to tensors. A product is something a caller can send a request to and get an answer from, that keeps working when the caller sends something unexpected, and that tells you when it has stopped being right. This week builds the distance between those two things.

The interface is the design decision

Before any code, decide what crosses the boundary. The model wants a normalised float tensor of a particular shape. A caller has a picture. Somewhere between the two sits preprocessing, and where you put it decides what can go wrong.

- **Caller sends raw input, service preprocesses.** One implementation of the preprocessing, in one place, versioned with the model. Almost always right.
- **Caller sends a preprocessed tensor.** Now every caller has its own copy of your normalisation constants, and they will drift apart. Day 3 measures what that costs.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
```

```
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field

model = get_model()
app = FastAPI()

```

```
class Request(BaseModel):
    # 784 raw pixels in 0 to 255, exactly what a caller would send
    pixels: list[float] = Field(min_length=784, max_length=784)

def preprocess(pixels):
    """The one function that must match training. Note the normalise:
    it is the same MEAN and STD the training transform used, and it is
    the single most common thing to get wrong here."""
    x = torch.tensor(pixels, dtype=torch.float32).reshape(1, 1, 28, 28)
    return (x / 255.0 - MEAN) / STD

@app.post('/predict')
def predict(req: Request):
    if max(req.pixels) > 255 or min(req.pixels) < 0:
        raise HTTPException(422, 'pixels must be between 0 and 255')
    x = preprocess(req.pixels)
    with torch.no_grad():
        probs = torch.softmax(model(x), dim=1)[0]
    top = int(probs.argmax())
    return {'digit': top, 'confidence': round(float(probs[top]), 4)}

@app.get('/health')
def health():
    return {'status': 'ok', 'params': sum(p.numel()
                                          for p in model.parameters())}

client = TestClient(app)

def raw_pixels(i):
    """Undo the training normalisation to get back what a caller sends."""
    return ((xte[i, 0] * STD + MEAN) * 255.0).flatten().tolist()
print(client.get('/health').json())
r = client.post('/predict', json={'pixels': raw_pixels(0)})
print('status %d  %s' % (r.status_code, r.json()))
print('true label %d' % yte[0])
```

```
{'status': 'ok', 'params': 105866}
status 200  {'digit': 7, 'confidence': 1.0}
true label 7
```

That is a complete service. The request carries 784 raw pixel values in the range a caller would actually have, the service normalises them using the same constants the training transform used, and it returns a digit and a confidence. The health endpoint exists so that whatever is running the service can tell whether it came up.

Return the confidence, always

It costs nothing, and without it the caller cannot distinguish a prediction the model is certain of from one it is guessing at. It is also the number day 6 uses to detect that the model has stopped working, and you cannot collect it retrospectively.

Why the model loads once

The model is constructed at import time and held in memory. Loading a checkpoint takes far longer than a forward pass, so a service that loads per request spends nearly all of its time on the wrong thing. This is stated because it is a genuinely common mistake, and because it is invisible until you measure.

Day 2 Requests You Did Not Expect

Validation, status codes, and the inputs no check will catch

Every input that reaches the model is one somebody could have sent deliberately. Validation is not politeness, it is the boundary between a bad request and an exception in your inference path.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
```

```
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field

model = get_model()
app = FastAPI()

class Request(BaseModel):
    # 784 raw pixels in 0 to 255, exactly what a caller would send
    pixels: list[float] = Field(min_length=784, max_length=784)

def preprocess(pixels):
    """The one function that must match training. Note the normalise:
    it is the same MEAN and STD the training transform used, and it is
    the single most common thing to get wrong here."""
    x = torch.tensor(pixels, dtype=torch.float32).reshape(1, 1, 28, 28)
    return (x / 255.0 - MEAN) / STD

@app.post('/predict')
def predict(req: Request):
    if max(req.pixels) > 255 or min(req.pixels) < 0:
        raise HTTPException(422, 'pixels must be between 0 and 255')
    x = preprocess(req.pixels)
    with torch.no_grad():
        probs = torch.softmax(model(x), dim=1)[0]
    top = int(probs.argmax())
    return {'digit': top, 'confidence': round(float(probs[top]), 4)}

@app.get('/health')
def health():
    return {'status': 'ok', 'params': sum(p.numel()
                                          for p in model.parameters())}

client = TestClient(app)

def raw_pixels(i):
    """Undo the training normalisation to get back what a caller sends."""
    return ((xte[i, 0] * STD + MEAN) * 255.0).flatten().tolist()
cases = [('784 pixels, all valid', {'pixels': raw_pixels(1)}),
         ('too few pixels', {'pixels': [0.0] * 100}),
```

```
        ('out of range', {'pixels': [999.0] * 784}),
        ('wrong field name', {'image': raw_pixels(1)})]
for name, payload in cases:
    r = client.post('/predict', json=payload)
    print('%-24s %d' % (name, r.status_code))
```

```
784 pixels, all valid    200
too few pixels           422
out of range             422
wrong field name         422
```

Three different malformed requests, three 422 responses, no traceback and no crash. Two of those were rejected by the type declaration alone, before any of your code ran, because the length constraint is part of the schema. The third needed an explicit check.

What validation cannot catch

Every check above is about shape and range. None of them would notice a request containing 784 perfectly valid numbers that happen to be pure noise, or an upside-down digit, or a photograph of a cat. The model will answer confidently and the response will look exactly like a good one.

This is the thing to internalise about serving a model as opposed to serving a database query. There is no input the model will refuse. Detecting the inputs it should have refused is a statistical question answered over many requests, which is days 5 and 6, and it cannot be answered per request by validation.

Day 3 Train Serve Skew

Four preprocessing mistakes, measured, and why the gentle ones are worst

The single most common production failure in machine learning is not a bad model. It is a model receiving data prepared differently from the data it was trained on.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
```

```
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
model = get_model()
print('%-46s %10s' % ('preprocessing used at serving time', 'accuracy'))
raw = xte * STD + MEAN
variants = [('the same as training', (raw - MEAN) / STD),
            ('forgot to normalise', raw),
            ('normalised with 0.5 and 0.5', (raw - 0.5) / 0.5),
            ('forgot to divide by 255', raw * 255.0)]
for name, x in variants:
    print('%-46s %10.4f' % (name, accuracy(model, x, yte)))
```

```
preprocessing used at serving time              accuracy
the same as training                              0.9730
forgot to normalise                               0.9530
normalised with 0.5 and 0.5                       0.8970
forgot to divide by 255                           0.9640
```

Read those numbers carefully, because the lesson is the opposite of the one usually drawn. Correct preprocessing gives 0.9730. Feeding values a hundred times too large gives 0.9640. Skipping normalisation entirely gives 0.9530. Using somebody else's constants, which is the most realistic of the three mistakes, gives 0.8970.

Watch Out

The danger is that it degrades gently

Not one of those broke. A bug that multiplies every input by 255 cost about one point of accuracy, which no dashboard would flag and no smoke test would catch.
The reason is that a stack of linear layers and ReLUs is nearly scale-equivariant: multiply the input by a positive constant and the pre-activations scale with it, so the ordering of the outputs, and therefore the prediction, is largely preserved. The model tolerates the bug, which is exactly why the bug survives to production and sits there costing a point of accuracy forever.

How to make it impossible

- Put preprocessing in one function, in the same module as the model, and call it from both the training pipeline and the service. Not two functions that agree, one function.
- Save the constants with the checkpoint rather than writing them in two files.
- Keep a handful of fixed examples with known outputs and assert on them at startup. Any preprocessing change moves those numbers, and the service refuses to start.
- Log a summary of the input distribution and compare it against the training set. A mean and a standard deviation would have caught every row of that table.

Day 4 Latency, Throughput and Batching

Percentiles instead of averages, and where the knee is

Latency and throughput pull in opposite directions and the tool that resolves them is batching. Here is the actual shape of the trade on one model.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
```

```
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
model = get_model()
def percentiles(batch, n=100):
    x = xte[:batch]
    times = []
    with torch.no_grad():
        for _ in range(10):
            model(x)
        for _ in range(n):
            start = time.perf_counter()
            model(x)
            times.append((time.perf_counter() - start) * 1000)
    times.sort()
    return (times[len(times) // 2], times[int(len(times) * 0.95)],
            times[int(len(times) * 0.99)])

print('%8s %10s %10s %10s %16s'
      % ('batch', 'p50 ms', 'p95 ms', 'p99 ms', 'images/second'))
for batch in [1, 8, 32, 128]:
    p50, p95, p99 = percentiles(batch)
    print('%8d %10.2f %10.2f %10.2f %16.0f'
          % (batch, p50, p95, p99, batch / (p50 / 1000)))
```

```
  batch     p50 ms     p95 ms     p99 ms    images/second
      1       0.10       0.14       0.50            10309
      8       0.58       0.96       1.70            13819
     32       2.56       3.00       3.37            12494
    128       7.34      10.69      11.62            17442
```

Going from one image to 128 multiplies throughput by about 2.6 and multiplies the time an individual request waits by about 49. Notice also that most of the throughput gain arrives by batch 32, and the step from 32 to 128 nearly quadruples latency for four percent more images per second. The knee is worth finding.

Report percentiles, not averages

At batch 1 the median is 0.27 ms and the 99th percentile is 0.75, nearly three times higher. An average would have hidden that. Users experience the tail, and a service whose mean latency is fine and whose p99 is terrible is a service that is visibly slow to a percent of requests, which at any volume is a lot of people.

Dynamic batching, and why it is not free

A server can hold requests for a few milliseconds and run whatever has arrived as one batch. It is how inference servers reach high throughput on expensive hardware.

The cost is that every request now waits for the window even when the service is idle, so a system with low traffic gets worse latency for no throughput benefit at all. Batching pays when you are saturated. Below that it is a pure loss, and it is often switched on by default.

Day 5 Noticing Without Labels

Confidence as a drift signal, and the scale problem that makes it dangerous

The service works, it is validated, and it is fast. Now assume that in three months something upstream changes: a camera is replaced, a client updates its image library, the population using the product shifts. Nobody tells you. No error is raised. How would you find out?

Not from accuracy, because accuracy needs labels and in production labels arrive late, arrive partially, or never arrive. Everything available in the moment is label-free, and the question is whether any of it is a usable substitute.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys
```

```

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
def corrupt(x, level):
    """Stand in for the world changing: the camera drifts out of focus
    and picks up sensor noise."""
    if level == 0:
        return x
    torch.manual_seed(level)
    shifted = torch.roll(x, shifts=int(level), dims=3)
    blur = nn.functional.avg_pool2d(shifted, 3, stride=1, padding=1)
    mix = 1 - level / 10.0
    out = mix * shifted + (1 - mix) * blur
    return out + (level / 40.0) * torch.randn_like(out)

@torch.no_grad()
def watch(model, x, y):
    """Everything a monitor can see without labels, plus the accuracy
    it cannot see, so we can check whether the signals track it."""
```

```
    model.eval()
    probs = torch.softmax(model(x), dim=1)
    conf, pred = probs.max(1)
    entropy = -(probs * (probs + 1e-9).log()).sum(1)
    return {'accuracy': (pred == y).float().mean().item(),
            'mean confidence': conf.mean().item(),
            'mean entropy': entropy.mean().item(),
            'share below 0.9': (conf < 0.9).float().mean().item()}
model = get_model()
keys = ['accuracy', 'mean confidence', 'mean entropy', 'share below 0.9']
print('%8s %10s %16s %14s %16s'
      % (('level',) + tuple(keys)))
rows = []
for level in [0, 1, 2, 3, 4, 6, 8]:
    m = watch(model, corrupt(xte, level), yte)
    rows.append(m)
    print('%8d %10.4f %16.4f %14.4f %16.4f'
          % ((level,) + tuple(m[k] for k in keys)))

acc = np.array([r['accuracy'] for r in rows])
conf = np.array([r['mean confidence'] for r in rows])
print()
print('correlation between accuracy and mean confidence %.4f'
      % float(np.corrcoef(acc, conf)[0, 1]))
```

```
   level   accuracy  mean confidence   mean entropy  share below 0.9
       0     0.9730           0.9708         0.0864           0.0775
       1     0.9610           0.9633         0.1100           0.1035
       2     0.9170           0.9290         0.2050           0.2180
       3     0.7790           0.8562         0.3774           0.4135
       4     0.5340           0.7940         0.5440           0.5920
       6     0.2135           0.7679         0.6427           0.6210
       8     0.1100           0.7467         0.6978           0.6615

correlation between accuracy and mean confidence 0.9530
```

The good news is that the label-free signals do track the accuracy nobody can see: mean confidence correlates with it at 0.95 across this sweep. If confidence falls, something has changed, and you did not need a single label to know.

Watch Out

The scale is wildly wrong, and that is the whole problem

Go to the last row. Accuracy has collapsed from 0.973 to 0.110, which is barely above guessing. Mean confidence over the same collapse fell only from 0.971 to 0.747.
The model is wrong about nine times out of ten and still telling you it is three-quarters sure. Any alert threshold set on mean confidence in the obvious way, say fire below 0.7, would never have fired at all. This is the well-known overconfidence of neural networks off their training distribution, and it means confidence is a usable *relative* signal and a dangerous absolute one.

The fourth column is the better instrument. The share of predictions below 0.9 goes from 0.078 to 0.662, a factor of eight and a half, against mean confidence's factor of 1.3. Counting

how many predictions fall below a threshold is far more sensitive than averaging the confidence, because the average is dominated by the many cases that remain easy.

Day 6 Monitoring That Would Actually Fire

What to log, what to alert on, and setting thresholds from a baseline

Turning yesterday's observation into something that can page somebody at three in the morning.

```
import os, io, json, time, base64
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

MEAN, STD = 0.1307, 0.3081
tf_ = transforms.Compose([transforms.ToTensor(),
                          transforms.Normalize((MEAN,), (STD,))])
train_all = datasets.MNIST('data', train=True, download=True, transform=tf_)
test_all = datasets.MNIST('data', train=False, download=True, transform=tf_)

def stack(ds, n):
    xs = torch.stack([ds[i][0] for i in range(n)])
    ys = torch.tensor([ds[i][1] for i in range(n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xte, yte = stack(test_all, 2000)

def build():
    return nn.Sequential(
        nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        nn.Flatten(), nn.Linear(32 * 7 * 7, 64), nn.ReLU(),
        nn.Linear(64, 10))

def get_model(path='served.pt', epochs=6):
    """Train once, then reuse the checkpoint on later runs."""
    model = build()
    if os.path.exists(path):
        model.load_state_dict(torch.load(path))
        model.eval()
        return model
    torch.manual_seed(0)
    model = build()
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            opt.zero_grad()
```

```
            lf(model(xb), yb).backward()
            opt.step()
    model.eval()
    torch.save(model.state_dict(), path)
    return model

def accuracy(model, x, y):
    model.eval()
    with torch.no_grad():
        return (model(x).argmax(1) == y).float().mean().item()
def corrupt(x, level):
    """Stand in for the world changing: the camera drifts out of focus
    and picks up sensor noise."""
    if level == 0:
        return x
    torch.manual_seed(level)
    shifted = torch.roll(x, shifts=int(level), dims=3)
    blur = nn.functional.avg_pool2d(shifted, 3, stride=1, padding=1)
    mix = 1 - level / 10.0
    out = mix * shifted + (1 - mix) * blur
    return out + (level / 40.0) * torch.randn_like(out)

@torch.no_grad()
def watch(model, x, y):
    """Everything a monitor can see without labels, plus the accuracy
    it cannot see, so we can check whether the signals track it."""
    model.eval()
    probs = torch.softmax(model(x), dim=1)
    conf, pred = probs.max(1)
    entropy = -(probs * (probs + 1e-9).log()).sum(1)
    return {'accuracy': (pred == y).float().mean().item(),
            'mean confidence': conf.mean().item(),
            'mean entropy': entropy.mean().item(),
            'share below 0.9': (conf < 0.9).float().mean().item()}
model = get_model()
clean = watch(model, xte[:200], yte[:200])
broken = watch(model, corrupt(xte[:200], 8), yte[:200])
print('%-20s %12s %12s' % ('', 'clean', 'corrupted'))
for k in ['accuracy', 'mean confidence', 'share below 0.9']:
    print('%-20s %12.4f %12.4f' % (k, clean[k], broken[k]))
```

```
                         clean    corrupted
accuracy                0.9950       0.1250
mean confidence         0.9830       0.7463
share below 0.9         0.0550       0.6600
```

On a realistic monitoring window of two hundred requests the signal is unmistakable in the right column and nearly invisible in the middle one. Twelve times as many low-confidence predictions is not something that happens by chance; a mean moving from 0.98 to 0.75 could plausibly be a quiet afternoon.

What to log, per request

- The predicted class and the confidence. These are the monitoring signal and they cost nothing.

- A summary of the input, not the input itself: mean, standard deviation, shape. Enough to detect that the distribution moved without storing anybody's data.
- The model version, so that a change in behaviour can be attributed to a deployment.
- Latency, so that today's slowdown can be separated from today's accuracy problem.

Watch Out

Logging the input is a decision, not a default

Storing raw requests makes debugging enormously easier and turns your monitoring system into a store of user data, with everything that implies for retention, access and deletion. Summaries give most of the diagnostic value with none of that. Decide deliberately, before the logs exist, because the awkward moment to discover you kept six months of personal data is when somebody asks you to delete it.

What to alert on

1. The share of predictions below a confidence threshold, compared against a baseline measured on known-good traffic. This is the sensitive one.
2. The distribution of predicted classes. A model that suddenly answers with one class far more often has usually broken, and this needs no confidence at all.
3. Input summary statistics against the training distribution, which catches preprocessing changes upstream.
4. Latency percentiles and error rates, which are ordinary service monitoring and catch the failures that have nothing to do with the model.

Set every threshold from a measured baseline rather than from intuition. Day 5 is the demonstration of why: an intuitive threshold on mean confidence would have sat quietly through a total collapse.

And get some labels

Every signal here is a proxy. None of them tells you the model is wrong, only that something changed. The only cure is labels, and the practical answer is to label a small sample continuously rather than a large sample never. A few hundred labelled requests a week gives a real accuracy number with a real error bar, and it is what converts the monitoring above from an alarm into a diagnosis.

Day 7 The Deployment Checklist

Before, after, and the failure mode that defines the week

The whole path, in the order you would build it.

Before deploying

1. One preprocessing function, shared by training and serving, with the constants stored alongside the weights.
2. A handful of fixed examples with known outputs, asserted at startup.

3. Input validation that returns 422, covering shape, range and type.
4. The model loaded once at startup, not per request.
5. A health endpoint, and a version string in every response.
6. Latency percentiles measured at a realistic batch size, on hardware resembling production.

After deploying

1. Log prediction, confidence, input summary, model version and latency, on every request.
2. Establish a baseline for each of those on traffic you know is good.
3. Alert on the share of low-confidence predictions and on the class distribution, using thresholds derived from that baseline.
4. Label a small sample continuously.
5. Keep the previous model deployable, and know how to roll back without a rebuild.

The failure mode that defines this week

Software fails loudly. A machine learning system fails quietly: it returns a well-formed, confident, wrong answer, with a 200 status code, at normal latency. Every practice above exists because the usual signals that something is broken are all absent.

What week 18 does

The course has now covered the whole path from a tensor to a monitored service. The last week puts it together as one project, on a dataset none of the earlier weeks used, and is specific about what to do next and what this course deliberately did not teach you.

> **The sentence worth keeping**

Self-Assessment

Try It Yourself

1. Where should preprocessing live?

   a) In each client, so the service stays simple

   b) In one function shared by training and serving, versioned with the model, because two implementations that agree today will drift

   c) In the model's forward method only

   d) It does not matter

2. A caller sends 784 valid numbers that happen to be pure noise. What does validation do?

   a) Returns a null prediction

   b) Rejects it with a 422

c) Rejects it with a 500

d) Nothing, because it is well formed, and the model answers confidently: detecting it is a statistical question over many requests

3. Feeding the model inputs a hundred times too large cost about one point of accuracy. Why is that dangerous rather than reassuring?

   a) Because it will get worse over time

   b) Because it corrupts the weights

   c) It is not dangerous

   d) Because the bug degrades gently enough that no dashboard or smoke test catches it, so it survives to production and stays there

4. Going from batch 1 to batch 128 multiplied throughput by about 2.6. What happened to the latency of an individual request?

   a) It halved

   b) It rose by roughly fifty times

   c) It rose by about ten percent

   d) It was unchanged

5. Why report latency percentiles rather than the average?

   a) Averages are harder to compute

   b) The p99 was nearly three times the median, and users experience the tail, which an average hides

   c) Percentiles are more precise

   d) Averages cannot be compared across machines

6. At the highest drift level the model's accuracy fell to 0.110. What was its mean confidence?

   a) About 0.50

   b) About 0.11, tracking accuracy closely

   c) About 0.30

   d) About 0.75, so an intuitive alert threshold would never have fired

7. Which monitoring signal was most sensitive to drift?

   a) Mean entropy

   b) The predicted class alone

   c) Mean confidence, which moved by a factor of 1.3

   d) The share of predictions below a confidence threshold, which moved by a factor of about eight

8. How should alert thresholds be set?

   a) At round numbers such as 0.5

   b) From a baseline measured on traffic known to be good, because an intuitive threshold sat quietly through a total collapse

   c) As high as possible to avoid noise

   d) From published best practice

Answers:1b, 2d, 3d, 4b, 5b, 6d, 7d, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.

CHAPTER 18

The Capstone

Week 18

OBJECTIVES

One problem worked end to end on a dataset the course has not touched, including the measurement that undoes most of the result.

Everything the course taught, applied once to Fashion-MNIST, with nothing tuned in advance. The build-up produces a table that looks like six improvements. Three runs of one recipe at different seeds then show that the noise floor is wider than five of them, and the honest conclusion is that exactly one change was real. The week ends with the confusion structure, which says the remaining errors are a property of the input rather than the model, and an account of what eighteen weeks did not cover.

Day 1 The Brief, and Looking at the Data

A dataset the course has not used, and the splits that keep the answer honest

Seventeen weeks of techniques, each measured on a problem chosen to show it working. This week is one problem worked end to end, on a dataset none of the earlier weeks used, so nothing here has been tuned in advance to make the course look good.

Fashion-MNIST is ten classes of clothing, the same shape and size as the digits, and considerably harder. It is a fair test of whether the habits transferred.

Look at the data before writing a model

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

```

```
def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
print('train %s  val %s  test %s'
      % (tuple(xtr.shape), tuple(xva.shape), tuple(xte.shape)))
counts = torch.bincount(ytr, minlength=10)
print()
print('%-10s %8s' % ('class', 'count'))
for i, c in enumerate(CLASSES):
    print('%-10s %8d' % (c, counts[i]))
print()
print('most common class is %.4f of the data'
      % (counts.max().item() / len(ytr)))
```

```
train (12000, 1, 28, 28)  val (3000, 1, 28, 28)  test (3000, 1, 28, 28)

class         count
t-shirt        1122
trouser        1220
pullover       1201
dress          1212
coat           1181
sandal         1204
shirt          1244
sneaker        1192
bag            1195
boot           1229

most common class is 0.1037 of the data
```

Balanced, which means accuracy is a reasonable metric and the majority-class baseline will be about a tenth. If it had come back ninety percent one class, everything downstream would have needed to change, which is why this is the first thing to run rather than an afterthought.

Three splits, and the middle one is not optional

Training, validation and test. The validation set chooses the model and the epoch. The test set is looked at once, at the end. Every number in this week that says *val* was used to make a decision, and every number that says *test* was not.

Selecting on the test set is the most common way a reported result turns out to be optimistic, and it does not feel like cheating while you are doing it. It feels like iterating.

The brief

- Classify a garment image into one of ten categories.
- Twelve thousand labelled examples, which is few enough that generalisation is a real concern.
- The result has to be servable: exportable, and fast enough that a request is not noticeably waiting on it.

- Every claimed improvement has to survive the check that week 3 insisted on.

Day 2 Baselines First

The two numbers that make every later number mean something

Before any network, the two numbers that make every later number meaningful.

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
def dense():
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def conv(width=32, norm=False, drop=0.0):
    def block(i, o):
        layers = [nn.Conv2d(i, o, 3, padding=1)]
        if norm:
            layers.append(nn.BatchNorm2d(o))
        return layers + [nn.ReLU(), nn.MaxPool2d(2)]
    return nn.Sequential(*(block(1, width) + block(width, width * 2)
                           + [nn.Flatten(), nn.Dropout(drop),
                              nn.Linear(width * 2 * 49, 128), nn.ReLU(),
                              nn.Linear(128, 10)]))

def augment(x):
    """Horizontal flip and a small shift. A flipped shoe is still a
    shoe, which is the check week 4 said to make before using one."""
    n = len(x)
    flip = torch.rand(n) < 0.5
    x = torch.where(flip.reshape(n, 1, 1, 1), x.flip(3), x)
    idx = torch.arange(28)
    pad = nn.functional.pad(x, (2, 2, 2, 2))
```

```
    rows = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    return out.unsqueeze(1)

def fit(model, epochs=8, lr=1e-3, seed=0, aug=False, wd=0.0):
    torch.manual_seed(seed)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    best, best_state = 0.0, None
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            if aug:
                xb = augment(xb)
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
        acc = score(model, xva, yva)
        if acc > best:
            best = acc
            best_state = copy.deepcopy(model.state_dict())
    # keep the weights that were best, which week 2 was firm about
    model.load_state_dict(best_state)
    model.eval()
    return model, best

@torch.no_grad()
def score(model, x, y):
    model.eval()
    out = torch.cat([model(x[i:i + 512]) for i in range(0, len(x), 512)])
    return (out.argmax(1) == y).float().mean().item()
from sklearn.linear_model import LogisticRegression
print('%-38s %10s' % ('', 'test accuracy'))
counts = torch.bincount(ytr, minlength=10)
major = int(counts.argmax())
print('%-38s %10.4f' % ('always guess the commonest class',
                        (yte == major).float().mean().item()))
lr = LogisticRegression(max_iter=200)
lr.fit(xtr.flatten(1).numpy(), ytr.numpy())
print('%-38s %10.4f' % ('logistic regression on raw pixels',
                        lr.score(xte.flatten(1).numpy(), yte.numpy())))
```

```
                                    test accuracy
always guess the commonest class           0.0993
logistic regression on raw pixels          0.8300
```

A linear model on raw pixels gets 0.83. That single number reframes the entire project. Anything that follows is competing for the remaining seventeen points, not for eighty-three, and a deep model that reaches 0.86 has added three points to a model that took two seconds to fit and needs

no framework at all.

Watch Out

The baseline is the number people skip and then misreport

Without that row, 0.89 sounds like a strong result. With it, 0.89 is a six point improvement over logistic regression, and the interesting question becomes whether six points justifies the complexity, the training cost and the deployment weight. That is a real question with a real answer that depends on the application.

Day 3 Building It Up

Six changes, one at a time, each measured

Now the model, built up one change at a time, each measured. This is the ablation table the course has asked for since week 3, on a real problem.

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
def dense():
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def conv(width=32, norm=False, drop=0.0):
    def block(i, o):
        layers = [nn.Conv2d(i, o, 3, padding=1)]
        if norm:
            layers.append(nn.BatchNorm2d(o))
        return layers + [nn.ReLU(), nn.MaxPool2d(2)]
    return nn.Sequential(*(block(1, width) + block(width, width * 2)
                           + [nn.Flatten(), nn.Dropout(drop),
```

```
                              nn.Linear(width * 2 * 49, 128), nn.ReLU(),
                              nn.Linear(128, 10)]))

def augment(x):
    """Horizontal flip and a small shift. A flipped shoe is still a
    shoe, which is the check week 4 said to make before using one."""
    n = len(x)
    flip = torch.rand(n) < 0.5
    x = torch.where(flip.reshape(n, 1, 1, 1), x.flip(3), x)
    idx = torch.arange(28)
    pad = nn.functional.pad(x, (2, 2, 2, 2))
    rows = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    return out.unsqueeze(1)

def fit(model, epochs=8, lr=1e-3, seed=0, aug=False, wd=0.0):
    torch.manual_seed(seed)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    best, best_state = 0.0, None
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            if aug:
                xb = augment(xb)
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
        acc = score(model, xva, yva)
        if acc > best:
            best = acc
            best_state = copy.deepcopy(model.state_dict())
    # keep the weights that were best, which week 2 was firm about
    model.load_state_dict(best_state)
    model.eval()
    return model, best

@torch.no_grad()
def score(model, x, y):
    model.eval()
    out = torch.cat([model(x[i:i + 512]) for i in range(0, len(x), 512)])
    return (out.argmax(1) == y).float().mean().item()
runs = [('dense, 128 hidden', lambda: dense(), {}),
        ('convolutional', lambda: conv(), {}),
        ('  and batch norm', lambda: conv(norm=True), {}),
        ('  and augmentation', lambda: conv(norm=True), {'aug': True}),
        ('  and dropout 0.3', lambda: conv(norm=True, drop=0.3),
         {'aug': True}),
        ('  and weight decay', lambda: conv(norm=True, drop=0.3),
         {'aug': True, 'wd': 1e-2}),
```

```
        ('  and twice the width', lambda: conv(64, norm=True, drop=0.3),
         {'aug': True, 'wd': 1e-2})]
print('%-24s %10s %10s %12s' % ('', 'val', 'test', 'parameters'))
for name, make, kw in runs:
    model, val = fit(make(), **kw)
    print('%-24s %10.4f %10.4f %12d'
          % (name, val, score(model, xte, yte),
             sum(p.numel() for p in model.parameters())))
```

```
                            val       test   parameters
dense, 128 hidden         0.8420     0.8493       101770
convolutional             0.8800     0.8787       421642
  and batch norm          0.8940     0.8923       421834
  and augmentation        0.8817     0.8767       421834
  and dropout 0.3         0.8800     0.8760       421834
  and weight decay        0.8760     0.8780       421834
  and twice the width     0.8730     0.8763       879114
```

Read down the test column. Convolution over dense is worth about three points, which is the largest single gain in the table and the one that reflects a real fact about images. Batch normalisation adds another point and a bit. And then everything stops.

Augmentation made it worse. Dropout made it worse again. Weight decay recovered a fraction. Doubling the width, at twice the parameters, changed nothing. Four techniques that every tutorial recommends, applied in the order every tutorial recommends them, and the best model in the table is the third row.

Day 4 How Much of That Was Real

The seed spread, and the table it demolishes

Before drawing any conclusion from yesterday's table, the question week 3 made a rule: how much does the same recipe vary when nothing changes but the seed?

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
```

```
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
def dense():
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def conv(width=32, norm=False, drop=0.0):
    def block(i, o):
        layers = [nn.Conv2d(i, o, 3, padding=1)]
        if norm:
            layers.append(nn.BatchNorm2d(o))
        return layers + [nn.ReLU(), nn.MaxPool2d(2)]
    return nn.Sequential(*(block(1, width) + block(width, width * 2)
                           + [nn.Flatten(), nn.Dropout(drop),
                              nn.Linear(width * 2 * 49, 128), nn.ReLU(),
                              nn.Linear(128, 10)]))

def augment(x):
    """Horizontal flip and a small shift. A flipped shoe is still a
    shoe, which is the check week 4 said to make before using one."""
    n = len(x)
    flip = torch.rand(n) < 0.5
    x = torch.where(flip.reshape(n, 1, 1, 1), x.flip(3), x)
    idx = torch.arange(28)
    pad = nn.functional.pad(x, (2, 2, 2, 2))
    rows = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    return out.unsqueeze(1)

def fit(model, epochs=8, lr=1e-3, seed=0, aug=False, wd=0.0):
    torch.manual_seed(seed)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    best, best_state = 0.0, None
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            if aug:
                xb = augment(xb)
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
        acc = score(model, xva, yva)
        if acc > best:
            best = acc
            best_state = copy.deepcopy(model.state_dict())
    # keep the weights that were best, which week 2 was firm about
    model.load_state_dict(best_state)
    model.eval()
    return model, best
```

```

@torch.no_grad()
def score(model, x, y):
    model.eval()
    out = torch.cat([model(x[i:i + 512]) for i in range(0, len(x), 512)])
    return (out.argmax(1) == y).float().mean().item()
accs = []
for seed in [0, 1, 2]:
    model, _ = fit(conv(norm=True), seed=seed, aug=True)
    accs.append(score(model, xte, yte))
a = np.array(accs)
print('same recipe, three seeds: %s' % ' '.join('%.4f' % v for v in a))
print('spread %.4f' % (a.max() - a.min()))
print()
print('any improvement smaller than that is not an improvement')
```

```
same recipe, three seeds: 0.8700 0.8813 0.8847
spread 0.0147

any improvement smaller than that is not an improvement
```

Watch Out

The noise floor is larger than most of the table

Three runs of one recipe span 0.0147. Now go back to yesterday's ablation and compare each step against that.
Augmentation costing 0.0156, dropout costing 0.0007, weight decay gaining 0.0020, doubling the width costing 0.0017: not one of those four is distinguishable from running the same code again with a different seed. Even batch normalisation's 0.0136 gain sits inside the noise band, and it is the second largest effect in the table.

So the honest report of yesterday's work is much shorter than the table. Convolution beats a dense network by an amount that clearly exceeds the noise. Everything after that is unproven on this budget, in either direction. Saying so is the finding.

What to do about it

- Measure the seed spread once, early, and treat it as the resolution of every comparison you make afterwards.
- Run the promising configurations at three seeds, not one, and compare the ranges rather than the points.
- When two options are within noise, choose on the other axes: fewer parameters, faster inference, less code.
- Report the spread alongside the number. A result given as 0.88 invites a comparison a result given as 0.87 to 0.88 correctly refuses.

On that last criterion the choice here is easy. The third row is the best mean, uses half the parameters of the widest model, and involves no augmentation code at all. It wins on everything that is not a coin flip.

Day 5 Where the Errors Actually Are

Per-class recall, the confusion structure, and what it implies

An aggregate accuracy hides the shape of the problem. The confusion structure is where the remaining errors actually are, and it usually suggests what to do next far better than another hyperparameter.

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
def dense():
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def conv(width=32, norm=False, drop=0.0):
    def block(i, o):
        layers = [nn.Conv2d(i, o, 3, padding=1)]
        if norm:
            layers.append(nn.BatchNorm2d(o))
        return layers + [nn.ReLU(), nn.MaxPool2d(2)]
    return nn.Sequential(*(block(1, width) + block(width, width * 2)
                           + [nn.Flatten(), nn.Dropout(drop),
                              nn.Linear(width * 2 * 49, 128), nn.ReLU(),
                              nn.Linear(128, 10)]))

def augment(x):
    """Horizontal flip and a small shift. A flipped shoe is still a
    shoe, which is the check week 4 said to make before using one."""
    n = len(x)
    flip = torch.rand(n) < 0.5
    x = torch.where(flip.reshape(n, 1, 1, 1), x.flip(3), x)
    idx = torch.arange(28)
    pad = nn.functional.pad(x, (2, 2, 2, 2))
```

```
    rows = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    return out.unsqueeze(1)

def fit(model, epochs=8, lr=1e-3, seed=0, aug=False, wd=0.0):
    torch.manual_seed(seed)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    best, best_state = 0.0, None
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            if aug:
                xb = augment(xb)
            opt.zero_grad()
            lf(model(xb), yb).backward()
            opt.step()
        acc = score(model, xva, yva)
        if acc > best:
            best = acc
            best_state = copy.deepcopy(model.state_dict())
    # keep the weights that were best, which week 2 was firm about
    model.load_state_dict(best_state)
    model.eval()
    return model, best

@torch.no_grad()
def score(model, x, y):
    model.eval()
    out = torch.cat([model(x[i:i + 512]) for i in range(0, len(x), 512)])
    return (out.argmax(1) == y).float().mean().item()
model, _ = fit(conv(norm=True), aug=True)
with torch.no_grad():
    pred = torch.cat([model(xte[i:i + 512]).argmax(1)
                      for i in range(0, len(xte), 512)])
cm = torch.zeros(10, 10, dtype=torch.long)
for t, p in zip(yte.tolist(), pred.tolist()):
    cm[t][p] += 1
print('%-10s %7s %7s  %s' % ('class', 'recall', 'errors', 'mistaken for'))
for i, name in enumerate(CLASSES):
    row = cm[i].clone()
    right = int(row[i])
    total = int(row.sum())
    row[i] = 0
    worst = int(row.argmax())
    print('%-10s %7.3f %7d  %s (%d)'
          % (name, right / total, total - right, CLASSES[worst],
             int(row[worst])))
```

```
class       recall  errors  mistaken for
```

```
t-shirt      0.781      66  shirt (52)
trouser      0.994       2  dress (2)
pullover     0.687      97  shirt (68)
dress        0.849      45  shirt (17)
coat         0.796      66  shirt (55)
sandal       0.968       9  sneaker (9)
shirt        0.805      58  t-shirt (29)
sneaker      0.966      10  boot (6)
bag          0.946      16  shirt (5)
boot         0.926      21  sneaker (19)
```

The model is not uniformly 0.88 accurate. Trousers are 0.994 and pullovers are 0.687. Almost every serious error is inside one group: t-shirt, pullover, coat and shirt, garments that are genuinely similar at 28 by 28 in greyscale. The footwear classes confuse only with each other, and bags and trousers are essentially solved.

What that tells you to do

- More capacity will not fix this. The information that separates a coat from a pullover is largely absent at this resolution, which is an input problem rather than a model problem.
- Higher resolution or colour would help more than any architecture change, and both are decisions about data collection.
- If the application tolerates it, merging the four upper-body classes into one would take accuracy well above 0.95 immediately. Whether that is acceptable is a product question, and it is worth asking before spending a month on the model.
- If it is not tolerable, the labelling effort should go where the errors are rather than being spread evenly.

This is the most transferable habit in the week

An accuracy number tells you how much room is left. A confusion breakdown tells you where it is and often what kind of problem it is. Most of the time the answer is not a better model, and you cannot discover that from the aggregate.

Day 6 Shipping It, and Writing It Up

Export, verify, measure, and report in a way somebody can act on

The chosen model, exported and checked, using week 16's procedure.

```
import os, time, copy
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets, transforms

# A dataset no earlier week used, so nothing here is tuned to it.
tf_ = transforms.ToTensor()
train_all = datasets.FashionMNIST('data', train=True, download=True,
                                  transform=tf_)
test_all = datasets.FashionMNIST('data', train=False, download=True,
                                 transform=tf_)
```

```
CLASSES = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
           'sandal', 'shirt', 'sneaker', 'bag', 'boot']

def stack(ds, n, start=0):
    xs = torch.stack([ds[i][0] for i in range(start, start + n)])
    ys = torch.tensor([ds[i][1] for i in range(start, start + n)])
    return xs, ys

xtr, ytr = stack(train_all, 12000)
xva, yva = stack(train_all, 3000, start=12000)
xte, yte = stack(test_all, 3000)
def dense():
    return nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(),
                         nn.Linear(128, 10))

def conv(width=32, norm=False, drop=0.0):
    def block(i, o):
        layers = [nn.Conv2d(i, o, 3, padding=1)]
        if norm:
            layers.append(nn.BatchNorm2d(o))
        return layers + [nn.ReLU(), nn.MaxPool2d(2)]
    return nn.Sequential(*(block(1, width) + block(width, width * 2)
                           + [nn.Flatten(), nn.Dropout(drop),
                              nn.Linear(width * 2 * 49, 128), nn.ReLU(),
                              nn.Linear(128, 10)]))

def augment(x):
    """Horizontal flip and a small shift. A flipped shoe is still a
    shoe, which is the check week 4 said to make before using one."""
    n = len(x)
    flip = torch.rand(n) < 0.5
    x = torch.where(flip.reshape(n, 1, 1, 1), x.flip(3), x)
    idx = torch.arange(28)
    pad = nn.functional.pad(x, (2, 2, 2, 2))
    rows = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    cols = torch.randint(0, 5, (n, 1)) + idx.reshape(1, 28)
    out = pad[torch.arange(n).reshape(n, 1, 1), 0,
              rows.reshape(n, 28, 1), cols.reshape(n, 1, 28)]
    return out.unsqueeze(1)

def fit(model, epochs=8, lr=1e-3, seed=0, aug=False, wd=0.0):
    torch.manual_seed(seed)
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    lf = nn.CrossEntropyLoss()
    loader = DataLoader(TensorDataset(xtr, ytr), batch_size=128,
                        shuffle=True)
    best, best_state = 0.0, None
    for _ in range(epochs):
        model.train()
        for xb, yb in loader:
            if aug:
                xb = augment(xb)
            opt.zero_grad()
            lf(model(xb), yb).backward()
```

```
            opt.step()
        acc = score(model, xva, yva)
        if acc > best:
            best = acc
            best_state = copy.deepcopy(model.state_dict())
    # keep the weights that were best, which week 2 was firm about
    model.load_state_dict(best_state)
    model.eval()
    return model, best

@torch.no_grad()
def score(model, x, y):
    model.eval()
    out = torch.cat([model(x[i:i + 512]) for i in range(0, len(x), 512)])
    return (out.argmax(1) == y).float().mean().item()
import onnxruntime as ort
model, _ = fit(conv(norm=True), aug=True)
torch.onnx.export(model, (xte[:1],), 'fashion.onnx', dynamo=False,
                  input_names=['image'], output_names=['logits'],
                  dynamic_axes={'image': {0: 'batch'},
                                'logits': {0: 'batch'}})
sess = ort.InferenceSession('fashion.onnx',
                            providers=['CPUExecutionProvider'])
out = sess.run(None, {'image': xte[:512].numpy()})[0]
with torch.no_grad():
    ref = model(xte[:512]).numpy()
print('exported agrees to %.2e' % float(np.abs(out - ref).max()))
print('same predictions: %s' % bool((out.argmax(1) == ref.argmax(1)).all()))
one = xte[:1].numpy()
for _ in range(10):
    sess.run(None, {'image': one})
times = []
for _ in range(200):
    start = time.perf_counter()
    sess.run(None, {'image': one})
    times.append((time.perf_counter() - start) * 1000)
times.sort()
print('median latency %.3f ms' % times[len(times) // 2])
print('file size %.1f KB' % (os.path.getsize('fashion.onnx') / 1024))
```

```
exported agrees to 2.86e-06
same predictions: True
median latency 0.096 ms
file size 1648.5 KB
```

Identical predictions, agreement to about three parts in a million, and a tenth of a millisecond per image. The brief's performance requirement is met with several orders of magnitude to spare, which is worth noticing: none of week 16's compression techniques are needed here, and applying them anyway would be work with no beneficiary.

How to write this up

A report that a colleague can act on, in the order that respects their time:

1. The problem, the data, and the class balance.
2. The baselines, both of them, before any model.
3. The chosen model in one sentence, with its parameter count.
4. The test accuracy, given as a range across seeds.
5. The ablation, stating plainly which rows are inside the noise.
6. The confusion structure, and what it implies about where effort should go next.
7. The serving numbers: export verified, latency, size.
8. What you did not try, and why.

Watch Out

The last item is the one people leave out

A report that lists only what worked reads as though the space was searched thoroughly. Saying that you did not try transfer learning because there was no suitable pretrained greyscale model, or did not tune the learning rate because the range test showed a wide flat optimum, tells the reader where the remaining opportunity is. It is also what stops the next person repeating your dead ends.

Day 7 Eighteen Weeks, and What Comes After

What this covered, what it did not, and where to go

Eighteen weeks. Worth being explicit about what that did and did not cover.

What you can now do

- Write and debug a training loop, and recognise the failure shapes from their symptoms.
- Build convolutional, recurrent and transformer models, and explain why each suits the data it suits.
- Fine-tune a pretrained model, and know when frozen features are the better choice.
- Train generative models, and measure them rather than admiring the samples.
- Train without labels, and check the result against the baseline that reveals whether it did anything.
- Make a model cheap enough to ship, and know which techniques are worth the effort on your hardware.
- Serve it, watch it, and notice when it has quietly stopped working.

What this course did not teach

- **Training at scale.** Everything ran on one CPU on small subsets. Multiple GPUs, sharded data and distributed optimisers are a different discipline.
- **Large language models in depth.** Week 10 built a small one and week 11 used a pretrained one. Instruction tuning, preference optimisation and serving models that do not fit in memory are all beyond this.

- **Reinforcement learning.** Not touched at all.
- **The mathematics.** This was a practitioner's course. The proofs behind why any of it works are worth study and are not here.
- **Data collection and labelling.** Every dataset here arrived clean and labelled. In practice that is most of the work, and day 5 showed the model's remaining errors were an input problem rather than a modelling one.

Where to go next

1. Do a project on data you collected yourself. Every hard part of this field that the course could not simulate lives in that step.
2. Read papers with the code open. Weeks 8 to 11 give you enough to follow most architecture work.
3. Reproduce a published result. It is the fastest way to find out how much of what you know is actually load-bearing.
4. Pick a specialism: vision, language, speech, or the systems side of making any of it run economically.
5. Learn the maths properly if you intend to do research rather than apply it.

> **The habit worth more than any technique here**

Self-Assessment

Try It Yourself

1. Logistic regression on raw pixels reached 0.83 on Fashion-MNIST. Why does that number matter so much?

   a) It reframes everything after it: a deep model at 0.89 is competing for the remaining seventeen points, not for eighty-three
   b) It shows the data is too easy
   c) It proves neural networks are unnecessary
   d) It does not, it is only a sanity check

2. What was the spread in test accuracy across three seeds of one recipe?

   a) About 0.0005
   b) About 0.0147
   c) About 0.05
   d) There was no spread

3. What did that spread imply about the ablation table?

   a) That the model needed more epochs
   b) Nothing, the improvements were much larger

c) That only the dense-to-convolutional step clearly exceeded the noise, and most of the other rows were indistinguishable from rerunning the same code

d) That the test set was too small to use

4. Augmentation, dropout, weight decay and doubling the width were all applied. What happened?

   a) The best model in the table was the one before any of them, and none of their effects were distinguishable from noise

   b) Training diverged

   c) Each added about a point

   d) They compounded to a four point gain

5. Per-class recall ranged from 0.994 for trousers to 0.687 for pullovers. What did the confusion structure suggest?

   a) The errors concentrate among visually similar upper-body garments, so it is an input problem that resolution or colour would help more than any architecture change

   b) The labels were wrong

   c) The learning rate was too high

   d) More capacity would fix it

6. Which split is allowed to influence your decisions?

   a) The test set, since it is the largest

   b) The validation set, while the test set is looked at once at the end

   c) Both equally

   d) Whichever gives the better number

7. What does the week say is most often left out of a write-up?

   a) The training time

   b) The parameter count

   c) The confusion matrix

   d) What you did not try and why, which tells the reader where the remaining opportunity is and stops them repeating dead ends

8. What is named as the habit worth more than any single technique?

   a) Tuning the learning rate carefully

   b) Measuring the baseline and the noise floor before believing any result, which costs minutes and almost nobody does

   c) Using the largest model that fits

   d) Always augmenting the data

Answers:1a, 2b, 3c, 4a, 5a, 6b, 7d, 8b. Anything you got wrong points at the day to re-read, not at a gap in ability.