Convolutional Networks: Building Blocks and Training

Week 2 · Day 1: From Learned Filters to a Working CNN

Yusuf Brima, Ph.D.

Introduction to Computer Vision · AIMS Cameroon, 2026–2027

Today

1. **From learned filters to a network**
2. **The building blocks**: convolution, activation, pooling, the classifier head
3. **The loss**: softmax and cross-entropy
4. **Backpropagation**: how every weight learns its share of the blame
5. **Optimisation**: SGD, momentum, Adam, learning rates
6. **Training a real CNN**, and reading what it learned
7. **Generalisation**: overfitting, regularisation, batch norm

By the end of today you should be able to

- trace tensor shapes and count parameters through a CNN;
- explain cross-entropy and backpropagation with a worked example;
- choose an optimiser and diagnose a bad learning rate;
- recognise overfitting and pick a remedy;
- read a training curve and a confusion matrix.

Where we are

Last Friday

- Classical vision: *designed* features + a learned classifier.
- We *learned* a 3×3 filter by gradient descent; it became Sobel.
- A neuron = a filter at one location; a conv layer = many shared filters.

Today

- Stack those layers into a network, and train *all* of it, end to end.
- Every number on today's slides comes from networks trained for this lecture, on a laptop CPU.

Week 2: the deep learning era

Day 1: CNN blocks and training

Day 2: architectures, transfer learning

Day 3: object detection

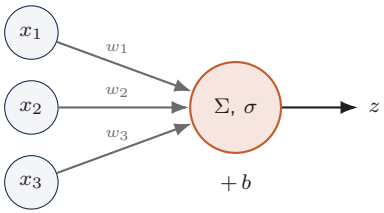
Day 4: segmentation

Day 5: vision transformers, captioning

Part 1

From learned filters to a network

One neuron, revisited



A neuron

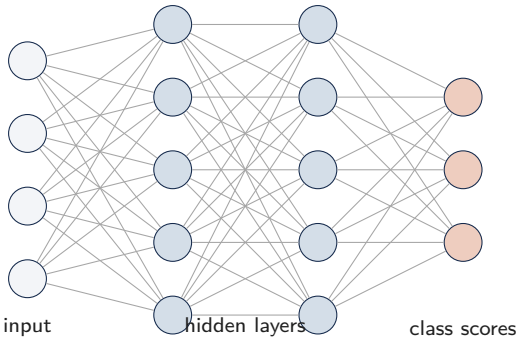
$$z = \sigma\big(w_1x_1+w_2x_2+w_3x_3+b\big) = \sigma(\mathbf{w}^\top\mathbf{x}+b)$$

- Weighted sum: a dot product, the same as one output pixel of a convolution.
- σ: a non-linear "activation", e.g. ReLU.
- Learning = finding $\mathbf{w}$ and b from examples.

Intuition

A neuron asks one question of its input: "how much do you look like my template $\mathbf{w}$?", and answers only if the match is strong enough (the activation).

Layers of neurons: a neural network

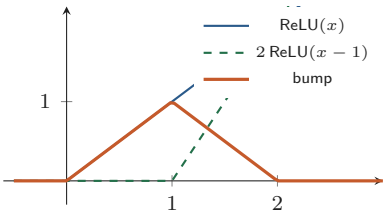


- Each layer's outputs are the next layer's inputs.
- Early layers detect simple patterns; later layers combine them into complex ones (Week 1, Day 1: retina $\rightarrow$ V1 $\rightarrow$ IT).
- The last layer produces one score per class.

Intuition

Depth means *composition*: an ear detector built from edge detectors, a cat detector built from ear, eye and fur detectors.

Why depth? Building shapes from ReLUs



Three ReLUs make a bump

$f(x) = \mathrm{ReLU}(x) - 2\,\mathrm{ReLU}(x-1) + \mathrm{ReLU}(x-2)$

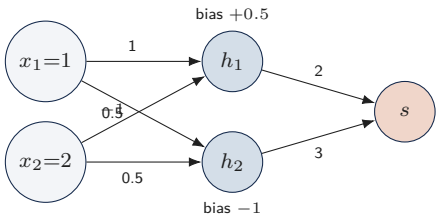
$x \le 0$: 0. $x = 1$: 1. $x \ge 2$: $2 - 2 + 0 = 0$.

A triangle on $[0, 2]$, built from three straight-line pieces.

Intuition

Enough bumps can approximate any curve. Stack layers and the bumps themselves get combined, so complex shapes need far fewer neurons: depth is *composition*.

A tiny network, computed by hand



Forward pass

$h_1 = \mathrm{ReLU}(1 \cdot 1 + (-1) \cdot 2 + 0.5) = \mathrm{ReLU}(-0.5) = 0$

$h_2 = \mathrm{ReLU}(0.5 \cdot 1 + 0.5 \cdot 2 - 1) = \mathrm{ReLU}(0.5) = 0.5$

$s = 2 \cdot 0 + 3 \cdot 0.5 = 1.5$

Intuition

h_1 is "off" for this input: its switch is closed, and during backpropagation no gradient will flow through it. Different inputs switch different neurons on: the network is a different linear function in each region of input space.

Why not connect every pixel to every neuron?

A fully connected network (MLP) on images

- Flatten the image into one long vector and connect every pixel to every neuron.
- Works on tiny images; scales terribly.
- Knows nothing about *where* pixels are: shuffle all pixels the same way in every image and it learns equally well.

Counting weights

A 28×28 grey image into 256 neurons: $784 \times 256 + 256 = 200\,960$ weights.

A 224×224 colour photo into 1000 neurons: $150\,528 \times 1000 \approx 150$ million weights, for one layer.

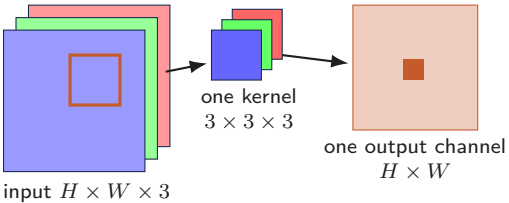
Intuition

Convolution builds in two facts about images: **nearby pixels belong together** (locality) and **a cat is a cat wherever it is** (weight sharing).

Part 2

The building blocks of a CNN

The convolutional layer, with channels

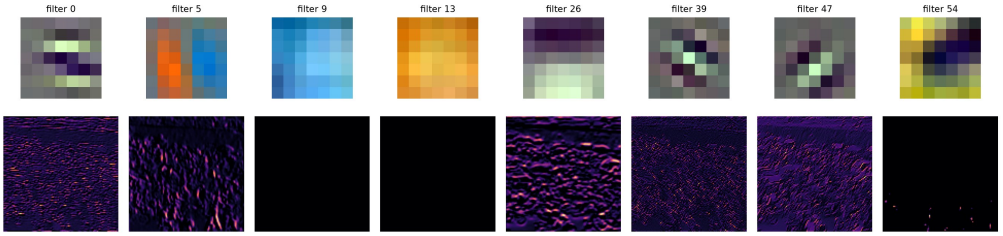


- A colour image has 3 channels. A kernel spans **all input channels**: $K \times K \times C_{\text{in}}$ weights.
- Each kernel produces **one** output channel (a feature map).
- A layer has C_{out} kernels, so its output has C_{out} channels.

One output value

$$y_o[m,n] = b_o + \sum_{c=1}^{C_{\text{in}}} \sum_{i,j} w_{o,c}[i,j]\; x_c[m+i,\, n+j] \qquad \text{parameters: } (K^2 C_{\text{in}} + 1)\, C_{\text{out}}$$

What a first layer computes on a real photograph



Lontsie, CC BY-SA 4.0

Minette

Top: eight of the 64 learned $7 \times 7 \times 3$ kernels in the first layer of a trained ResNet-18. Bottom: the feature map each produces on the Douala crossroads, after ReLU.

Intuition

Oriented edge detectors light up on poles, umbrellas and road markings; colour-blob detectors light up on clothes and taxis. Week 1's toolbox, learned rather than designed.

Worked example: shapes and parameters of a conv layer

The first layer of VGG

Input $224 \times 224 \times 3$. Layer: 64 kernels of size 3×3, stride 1, padding 1.

Output size: $\lfloor (224 + 2 - 3)/1 \rfloor + 1 = 224$, so $224 \times 224 \times 64$.

Parameters: $(3 \cdot 3 \cdot 3 + 1) \cdot 64 = 1792$.

Multiplications: $224^2 \times 64 \times 27 \approx 87$ million.

A 1×1 convolution

Input $56 \times 56 \times 256$; 64 kernels of size 1×1.

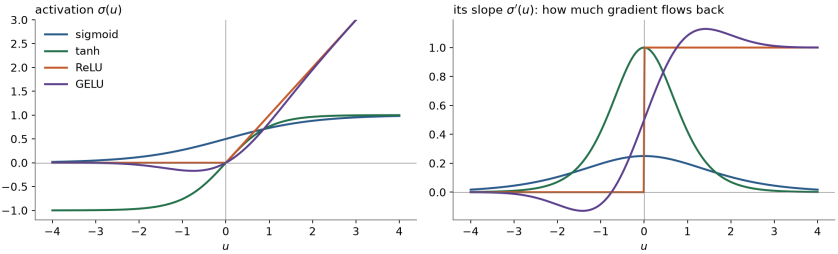
Output: $56 \times 56 \times 64$. Parameters: $(256 + 1) \cdot 64 = 16\,448$.

It looks at one pixel at a time but mixes all 256 channels: a cheap way to change the number of channels.

Intuition

Few parameters, many multiplications: convolutions are cheap to store and expensive to compute.

Activation functions

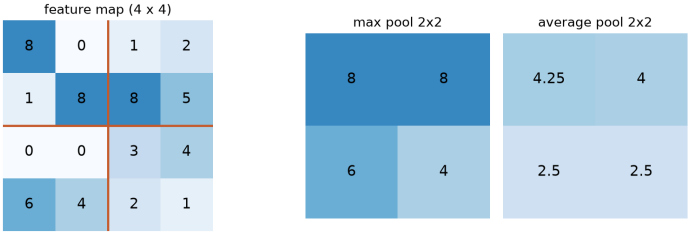


- **Sigmoid**, **tanh**: smooth, but flat at both ends: slope near 0, so gradients die.
- **ReLU** $\max(0, u)$: slope exactly 1 for positive inputs. Cheap and trains fast. The default since AlexNet (2012).
- **GELU**: a smooth ReLU, used in transformers (Day 5).

Intuition

Without an activation, stacked layers collapse into one linear layer (Week 1, Day 5). With ReLU, each neuron acts like a switch: on (pass the signal) or off (block it). Many switches together can carve out very complex decision boundaries.

Pooling: summarise a neighbourhood



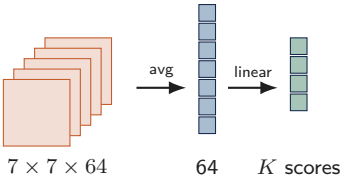
- **Max pooling** keeps the strongest response in each 2×2 block: "was the pattern anywhere here?"
- **Average pooling** keeps the mean.
- Both halve height and width, with **no parameters**.

Intuition

Pooling buys tolerance to small shifts (V1 complex cells; HOG's cells) and shrinks the maps so that later layers see a larger part of the image for the same cost.

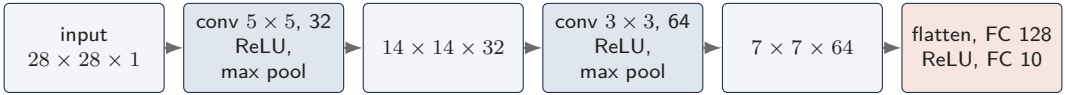
The head: from feature maps to class scores

- After several conv + pool stages we have a small stack of feature maps, e.g. $7 \times 7 \times 64$.
- **Flatten** it to a vector (3136 numbers) and apply fully connected layers.
- Or use **global average pooling**: average each map to one number, giving 64 numbers whatever the image size.
- A final linear layer gives K scores; softmax turns them into probabilities.



Intuition: the conv layers are the **feature extractor** (learned ϕ); the head is the **classifier** (learned g_θ): the Week 1 pipeline, with both parts trained together.

Our network for today



Counting all parameters

conv1: $(5 \cdot 5 \cdot 1 + 1) \cdot 32 = 832$
conv2: $(3 \cdot 3 \cdot 32 + 1) \cdot 64 = 18\,496$
FC 1: $(7 \cdot 7 \cdot 64) \cdot 128 + 128 = 401\,536$
FC 2: $128 \cdot 10 + 10 = 1290$

Total: 422 154, and 95% of them sit in the first fully connected layer.

Intuition

The two conv layers, which do all the "seeing", need fewer than 20 000 weights. The big FC layer is why modern networks replace flattening with global average pooling (Day 2).

How much computation, and what does each unit see?

Multiply-adds for one image

conv1: $28 \cdot 28$ positions $\times$ 32 maps $\times$ 25 $= 627\,200$
conv2: $14 \cdot 14 \times 64 \times 288 = 3\,612\,672$
FC 1: $3136 \times 128 = 401\,408$; FC 2: 1280

Total: about 4.6 million. The conv layers hold 5% of the parameters but do 92% of the work.

Receptive field, layer by layer

conv 5×5: each unit sees 5×5.
pool (stride 2): 6×6; steps now 2 pixels apart.
conv 3×3: $6 + 2 \cdot 2 = 10$.
pool: $10 + 1 \cdot 2 = 12$.

Each of the final 7×7 cells sees a 12×12 window of the 28×28 image: about a quarter of a sleeve or collar.

Intuition

Parameters decide memory and overfitting risk; multiply-adds decide speed; receptive fields decide what context a unit can use. Three different questions about the same network.

The same network in PyTorch

```
import torch.nn as nn, torch.nn.functional as F

class SmallCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.c1  = nn.Conv2d(1, 32, 5, padding=2)
        self.c2  = nn.Conv2d(32, 64, 3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):                          # (B
     , 1, 28, 28)
        x = F.max_pool2d(F.relu(self.c1(x)), 2)    #
     (B, 32, 14, 14)
        x = F.max_pool2d(F.relu(self.c2(x)), 2)    #
     (B, 64, 7, 7)
        x = F.relu(self.fc1(x.flatten(1)))         #
     (B, 128)
        return self.fc2(x)                         #
     (B, 10) scores
```

- `B` is the batch size: many images at once.
- Channels come *first* (Week 1, Day 2: CHW).
- The network outputs raw scores ("logits"); the loss applies softmax.

Practitioner's Note

Write the shape after every line as a comment. Most bugs in your first CNN are shape bugs.

Part 3

The loss: how wrong is the network?

Today's data: Fashion-MNIST



70 000 grey 28×28 images of clothing in 10 classes, from the Zalando online shop (2017). A harder drop-in replacement for handwritten digits.

Intuition

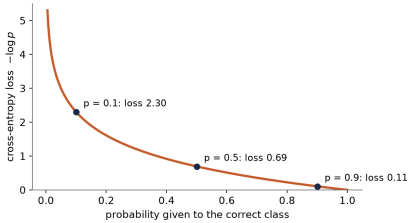
Small enough to train on a laptop in minutes, hard enough to show every phenomenon of today: look at how alike shirts, T-shirts, pullovers and coats are.

From scores to probabilities, and a loss

Softmax and cross-entropy

$$p_k = \frac{e^{s_k}}{\sum_j e^{s_j}}, \qquad \mathcal{L} = -\log p_y$$

s_k: score for class k; y: the correct class.



- The loss only looks at the probability given to the **correct** class.
- Confident and right: loss near 0. Confident and wrong: loss very large.

One image of a shirt

Scores: shirt 2.0, T-shirt 1.0, coat 0.1 $\Rightarrow p = (0.66, 0.24, 0.10)$; loss $= -\log 0.66 = 0.42$. With $p_{\text{shirt}} = 0.10$ instead: loss $= 2.30$.

Why not simply maximise accuracy?

- Accuracy changes only when a prediction *flips*. Nudge a weight slightly and accuracy almost never changes: its gradient is zero almost everywhere.
- Cross-entropy changes smoothly with every weight: it says "you were 66% sure, aim for 90%".
- So we *train* on cross-entropy and *report* accuracy.

Intuition

Accuracy is a pass/fail exam; cross-entropy is a detailed mark. You learn much more from "you got 62 out of 100" than from "fail".

Practitioner's Note

At the start of training, with 10 classes and random weights, every $p_k \approx 0.1$, so the loss should be $-\log 0.1 = 2.30$. If your first loss is far from that, something is wrong.

Part 4

Backpropagation: assigning the blame

The question backpropagation answers

- Our network has 422 154 weights and produces one loss number.
- For gradient descent we need $\partial \mathcal{L} / \partial w$ for **every** weight: how much would the loss change if I nudged this weight?
- Backpropagation computes all of them in one backward pass, at about the cost of one forward pass.

Intuition

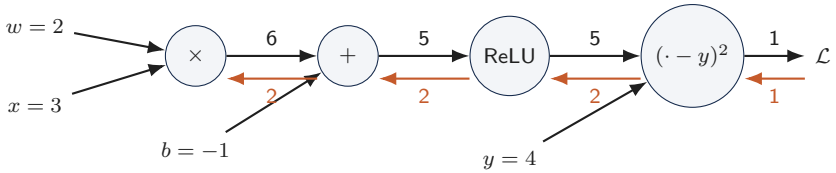
A football team concedes a goal. Who is to blame? The goalkeeper a lot, the defender who lost the ball somewhat, the striker who lost it earlier a little. Backpropagation distributes the blame backwards, in proportion to each player's influence on the outcome.

The chain rule

$$\text{if } \mathcal{L} = f(g(w)) \text{ then } \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial w}$$

Influences multiply along a path.

A computational graph, forward and backward



Forward (black), backward (orange)

Forward: $wx = 6$, $+b = 5$, ReLU $= 5$, $\mathcal{L} = (5-4)^2 = 1$.

Backward: $\partial\mathcal{L}/\partial(\text{out}) = 2(5-4) = 2$. ReLU passes it (input > 0). The sum passes it to both inputs. The product sends $2 \cdot x = 6$ to w.

- So $\partial\mathcal{L}/\partial w = 6$ and $\partial\mathcal{L}/\partial b = 2$.
- Gradient step with $\eta = 0.05$: $w \to 2 - 0.3 = 1.7$, $b \to -1.1$.
- New output: $\max(0, 1.7 \cdot 3 - 1.1) = 4.0$. Loss: 0. One step fixed it.

How gradients flow through each building block

- **Addition**: copies the gradient to both inputs.
- **Multiplication by a weight**: gradient to the weight = incoming gradient $\times$ the input.
- **ReLU**: passes the gradient where the input was positive, blocks it where negative.
- **Max pooling**: routes the gradient only to the position that was the maximum.
- **Convolution**: gradient to the kernel = correlation of the error map with the input (Week 1, Day 5).

Intuition

Every block only needs to know its *local* derivative. Backpropagation chains these local rules together. That is why you can build networks from any combination of blocks and they still train.

Practitioner's Note

You never write backpropagation by hand: PyTorch's *autograd* records the forward computation and runs the chain rule backwards when you call `loss.backward()`.

Backpropagation through a convolution, by hand

A 1D convolution with two weights

Input $x = (1, 2, 3)$, kernel (w_1, w_2), no padding:

$y_1 = w_1 \cdot 1 + w_2 \cdot 2, \quad y_2 = w_1 \cdot 2 + w_2 \cdot 3.$

Suppose the loss sends back $\partial \mathcal{L}/\partial y = (1, -1)$.

$$\frac{\partial \mathcal{L}}{\partial w_1} = 1 \cdot 1 + (-1) \cdot 2 = -1$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = 1 \cdot 2 + (-1) \cdot 3 = -1$$

- Each weight was used at *every* position, so its gradient is a **sum over positions**: input value $\times$ error at that position.
- That sum is a correlation of the input with the error map: exactly the rule we derived on Week 1, Day 5 when we learned Sobel.
- Weight sharing means every position teaches the same filter.

Checking gradients numerically

Central difference

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\mathcal{L}(w+\epsilon) - \mathcal{L}(w-\epsilon)}{2\epsilon}$$

$\mathcal{L}(w) = w^2$ at $w = 3$

$\epsilon = 0.001$: $\frac{9.006001 - 8.994001}{0.002} = 6.000$.
The chain rule gives $2w = 6$. They agree.

- Two forward passes per weight: far too slow for training (422 154 weights), but perfect for *testing* a hand-written layer on a few weights.
- PyTorch provides `torch.autograd.gradcheck` for exactly this.

Practitioner's Note

If you ever write your own layer or loss, check its gradient numerically before you trust a single training run.

Vanishing gradients: why ReLU won

- Going backwards, the gradient is multiplied by each activation's slope.
- A sigmoid's slope is at most 0.25.
- ReLU's slope is exactly 1 wherever it is active.

Ten layers deep

Sigmoid, best case: $0.25^{10} \approx 9.5 \times 10^{-7}$. The first layer receives a millionth of the signal: it barely learns.

ReLU (active units): $1^{10} = 1$. The signal arrives intact.

Intuition

Passing a message along a line of ten people, each of whom whispers at a quarter of the volume they heard: the last person hears nothing. ReLU lets every active person speak at full volume.

Practitioner's Note

Deep networks still need help: careful initialisation, batch normalisation (later today) and residual connections (Day 2).

Part 5

Optimisation: walking downhill

Mini-batch stochastic gradient descent

One SGD step

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{1}{B} \sum_{i \in \text{batch}} \nabla_{\mathbf{w}} \mathcal{L}_i$$

η: learning rate. B: batch size.

- The full-data gradient is expensive; a random **mini-batch** of 32–256 images gives a noisy but cheap estimate.
- **Epoch**: one pass over all training images.

Our training run

12 000 images, batch size 64: $\lceil 12\,000/64 \rceil = 188$ steps per epoch. 8 epochs: 1504 weight updates.

Intuition

The noise from mini-batches is not only tolerated, it helps: it shakes the weights out of sharp, narrow valleys towards broad ones that generalise better.

Batch size and why GPUs matter

Batch size

- Larger batches: smoother gradient estimates, better use of parallel hardware, more memory.
- Smaller batches: noisier steps, often generalise slightly better, fit on small GPUs.
- Doubling the batch size? Roughly double the learning rate too.

Memory for activations

conv1 output: $28 \cdot 28 \cdot 32 = 25\,088$ floats per image; a batch of 64 kept for backprop: ≈ 6.4 MB, for one layer.

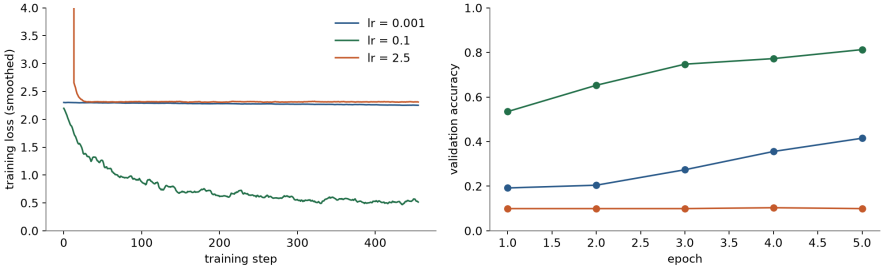
Why GPUs

- Training is millions of independent multiply-adds: the same operation on different numbers.
- A CPU has a few powerful cores; a GPU has thousands of simple ones working in parallel.
- Our baseline took 224 seconds on a 2-core laptop CPU. A free Colab GPU is typically 10–50 times faster on networks like this.

Practitioner's Note

Put model and data on the same device: `.to("cuda")`. Mixed devices: the second most common PyTorch error.

The learning rate is the most important knob

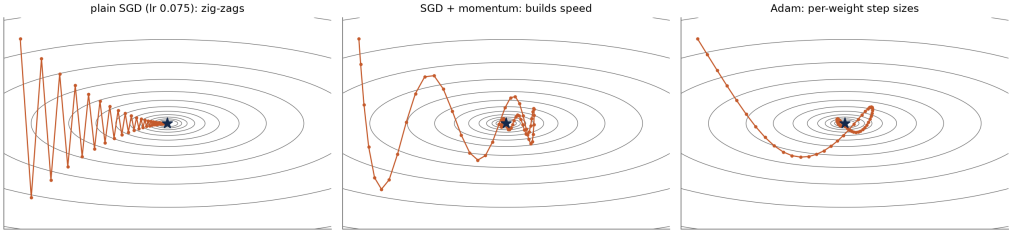


Same network, same data, plain SGD, three learning rates. **0.001**: barely moves in five epochs (validation 41.5%). **0.1**: steady progress (81.3%). **2.5**: the loss explodes and the network never recovers (9.9%, i.e. chance).

Intuition

Too small: tiny steps, endless walk. Too large: you overshoot the valley floor and bounce up the other side. A good learning rate is the largest one that still goes steadily downhill.

Momentum and Adam

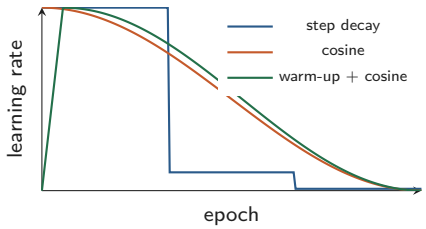


- **Momentum**: keep a running velocity, $v \leftarrow 0.9\,v + \nabla\mathcal{L}$, and step along it. Zig-zags cancel; consistent directions accelerate.
- **Adam**: momentum + a separate step size for every weight, scaled down where gradients are large and up where small.

Intuition

A ball rolling down a narrow valley (momentum) does not bounce wall to wall like a nervous walker (plain SGD). Adam additionally gives each direction its own stride length: the narrow valley of Week 1, Day 5, made round.

Learning-rate schedules and initialisation



Big steps early to make progress, small steps late to settle into the minimum.

Initialisation

- Weights start random, small, and different (identical weights would learn identical features).
- **He initialisation** for ReLU: standard deviation $\sqrt{2/\text{fan-in}}$, so signals neither shrink nor explode from layer to layer.

conv2 of our network

fan-in $= 3 \cdot 3 \cdot 32 = 288$: std $= \sqrt{2/288} \approx 0.083$.

Part 6

Training a real CNN

The training loop

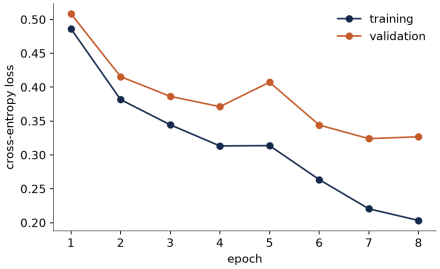
```
model = SmallCNN()
opt = torch.optim.Adam(model.parameters(), lr=1e
    -3)
for epoch in range(8):
    model.train()
    for xb, yb in train_loader:          # mini-
     batches of 64
        logits = model(xb)               # 1.
     forward
        loss = F.cross_entropy(logits, yb)  # 2.
     loss
        opt.zero_grad()                  # 3. clear
      old grads
        loss.backward()                  # 4.
     backprop
        opt.step()                       # 5.
     update weights
    model.eval()
    with torch.no_grad():
        val_acc = accuracy(model, val_loader)
```

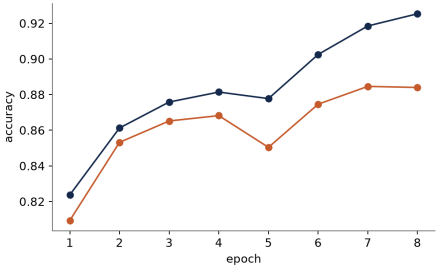
- Five lines do all the learning: forward, loss, zero, backward, step.
- `model.train()` / `model.eval()` switch dropout and batch norm between modes.
- `torch.no_grad()` skips gradient bookkeeping during evaluation.

Practitioner's Note

Forgetting `zero_grad()` makes gradients accumulate across steps, a classic silent bug.

Our CNN learns



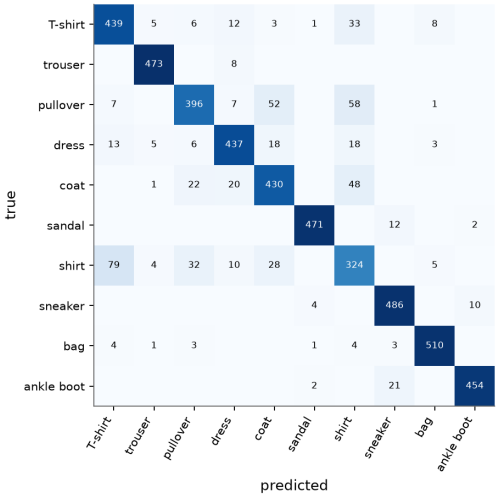


12 000 training images, Adam, 8 epochs, about 224 seconds on a laptop CPU. Validation accuracy: **88.4%**; training accuracy: 92.5%.

Intuition

Most learning happens in the first epoch or two; then gains slow down. The small gap between training and validation curves is the first sign of *overfitting*, the topic of the last part.

Where does it go wrong? The confusion matrix

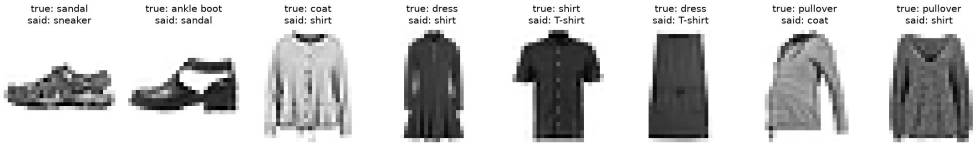


- Row = true class, column = prediction. The diagonal holds the correct answers.
- Trousers, bags, sandals and sneakers: almost perfect.
- **Shirts**: only 324 of 482 correct; 79 called T-shirt, 32 pullover, 28 coat.
- Sneakers and ankle boots are confused in both directions.

Practitioner's Note

Always look at the confusion matrix, not just accuracy. In a medical setting, *which* mistakes happen matters more than how many.

Looking at the mistakes

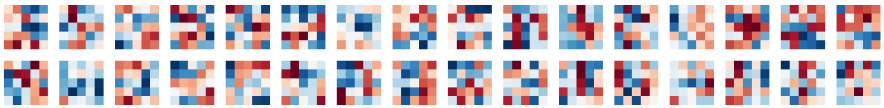


- Many errors are genuinely ambiguous at 28×28: a shirt and a T-shirt differ by a collar and buttons a few pixels wide.
- Some labels are arguably wrong: datasets contain label noise.

Intuition

Looking at mistakes tells you what to fix: more resolution? more data of a confusing class? cleaner labels? It is the cheapest, most informative debugging step there is.

What did the first layer learn?



The 32 first-layer 5×5 kernels of our trained network (red positive, blue negative). Started from random noise; shaped only by the goal of classifying clothes.

Intuition

You can see horizontal and vertical edge detectors and blob-like detectors, as in ResNet trained on ImageNet: the same building blocks emerge from very different data.

Can we trust the probabilities?

- A model is **calibrated** if, among all images where it says "90% shirt", about 90% really are shirts.
- Modern deep networks are often **over-confident**: they say 99% and are right far less often (Guo et al., 2017).
- Cross-entropy keeps pushing confidence up on training images, long after they are classified correctly.

Temperature scaling

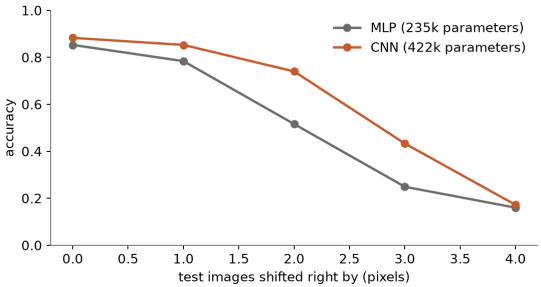
$$p_k = \frac{e^{s_k/T}}{\sum_j e^{s_j/T}}$$

Fit one number $T > 1$ on the validation set; it softens over-confident outputs without changing which class wins.

Practitioner's Note

Whenever a probability drives a decision (refer this patient? spray this field?), check calibration on held-out data before using a threshold.

CNN vs. fully connected network



Same data, same training. Test images shifted to the right by 0–4 pixels.

- Unshifted: MLP 85.3%, CNN 88.3%.
- 2-pixel shift: MLP 51.6%, CNN 74.0%.
- 4-pixel shift: both collapse (16.0%, 17.2%).

Intuition

Convolution + pooling give the CNN *some* tolerance to position, like HOG last week. But only a little: beyond a few pixels it must *see* shifted examples in training, which is what augmentation does.

Part 7

Generalisation and regularisation

The real goal: unseen images

- Training loss measures how well we fit the *training* images. We care about *new* images.
- **Overfitting**: the network memorises training images, including their noise, instead of the general pattern.
- Three datasets:
  - **training**: to fit the weights;
  - **validation**: to choose settings and when to stop;
  - **test**: touched once, at the end.

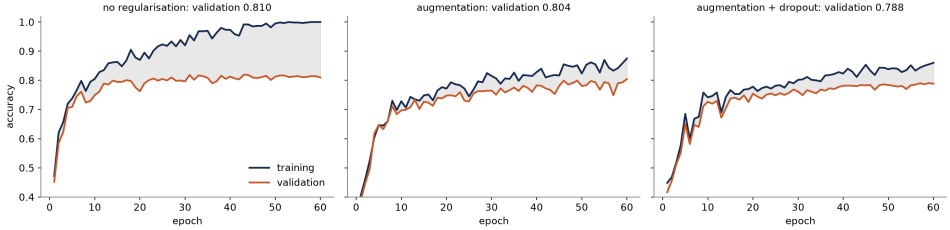
Intuition

A student who memorises past exam answers scores 100% on past papers and fails the real exam. A network with 422 000 weights and 600 training images can memorise every one of them.

Practitioner's Note

If you tune settings on the test set, it silently becomes a validation set, and your reported accuracy is optimistic.

Overfitting, live: 600 training images



No regularisation: training accuracy reaches 100.0%; validation stays at 81.0%. The grey gap is memorisation. **Augmentation** (shifts, flips): training 87.5%, validation 80.4%. **Augmentation + dropout**: training 86.0%, validation 78.8%.

Intuition

Regularisation did its first job: it closed the gap, so the network memorised far less. But within 60 epochs it did *not* raise validation accuracy: regularised networks learn more slowly and need longer training, and these particular augmentations add little for centred clothing images.

What the overfitting experiment teaches

- A **gap** between training and validation accuracy is the symptom of memorisation; regularisation reliably shrinks it.
- Whether *validation* accuracy improves depends on the data, the amount of training, and whether the augmentations match real variation.
- The remedy that always helps is **more, and more varied, data**.

Practitioner's Note

Never assume a technique helps: measure it on validation data, with the same training budget, and ideally over a few random seeds. Keep it only if it earns its place. This is what we will do with every design choice this week.

Intuition

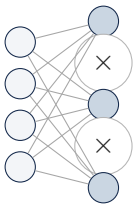
Regularisation makes training *harder* on purpose, so that the network can only succeed by learning patterns that also hold on new images. Harder training needs more time.

Three regularisers

Weight decay

$$\mathcal{L}' = \mathcal{L} + \lambda \sum w^2$$

Penalise large weights. Prefers simple, smooth functions that do not depend heavily on any single input.



Dropout During training, randomly switch off half the neurons at each step. No neuron can rely on a particular partner.

Data augmentation Show random flips, crops, shifts, colour changes of each training image (Week 1, Day 5). The label stays the same.

The network learns invariances from examples instead of memorising exact pixels.

Practitioner's Note

Early stopping is the fourth, simplest regulariser: keep the weights from the epoch with the best validation score.

Weight decay and early stopping, worked

What weight decay does to one weight

With penalty λw^2, each SGD step becomes

$$w \leftarrow (1 - 2\eta\lambda)\, w - \eta\, \nabla \mathcal{L}$$

$\eta = 0.1$, $\lambda = 0.01$: every step multiplies w by 0.998. A weight the data does not support decays to $0.998^{1000} \approx 0.14$ of its size in 1000 steps.

Early stopping

Train while watching validation accuracy; save the weights at the best epoch; stop when it has not improved for, say, 5 epochs.

In our unregularised 600-image run, the best validation accuracy was 81.9%; the final one was 81.0%.

Intuition

Weight decay: "keep weights small unless the data insists". Early stopping: "stop before memorising". Both pull the network towards simpler functions.

Augmentation: which transformations are safe?



RyanDannyChacko, CC BY-SA 4.0

A transformation is safe if it **does not change the label**. A flipped cat is still a cat. A rotated "6" becomes a "9"; a flipped road sign or text is wrong; a colour shift can turn a ripe banana into an unripe one.

Practitioner's Note

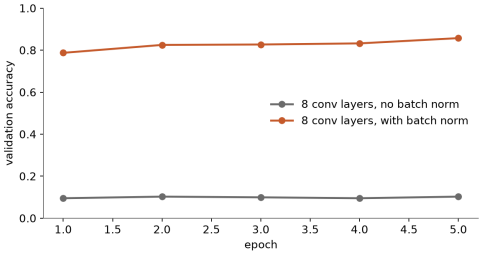
Choose augmentations from the variations your deployment will really see: phone models, lighting, angles, blur. Augmentation is how you tell the network which of Day 1's "nuisance" changes to ignore.

Batch normalisation

Batch norm, per channel

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}, \qquad y = \gamma\,\hat{x} + \beta$$

γ, β are learned. At test time, running averages of μ, σ are used.



An 8-layer plain CNN, SGD with momentum, same learning rate: without batch norm 10.3%, with batch norm 85.7% after five epochs.

- Keeps each layer's inputs at a steady scale while earlier layers change.
- Allows larger learning rates; deep networks train much more reliably.

Intuition: the input normalisation of Week 1, Day 2, applied inside the network at every layer.

A recipe for training a network

1. **Look at your data.** Classes, image sizes, label quality.
2. **Check the first loss** ($\approx \log K$).
3. **Overfit one small batch**: a model that cannot reach 100% on 32 images has a bug.
4. **Find a learning rate**: the largest that decreases the loss steadily.
5. **Train, watching both curves.**
6. **Regularise** only once you see a gap.
7. **Read the confusion matrix and the mistakes.**

Intuition

Training networks is an experimental science: change one thing at a time, keep notes, compare curves.

Practitioner's Note

Fix the random seed while debugging, then report results averaged over several seeds: our runs vary by about half a percent from seed to seed.

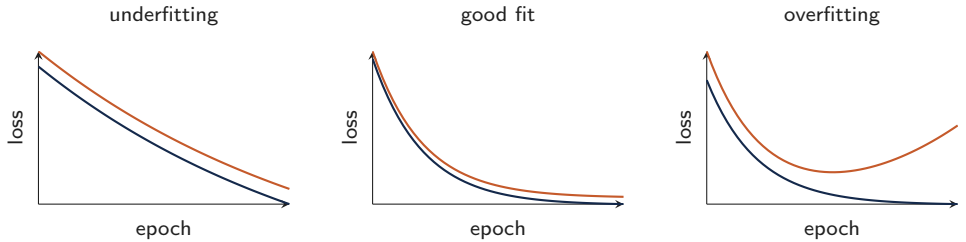
Symptoms and likely causes

Symptom	Likely causes
First loss far from $\log K$	wrong labels or loss; inputs not normalised; softmax applied twice
Loss is `nan` or explodes	learning rate too high; division by zero; inputs with `inf`
Loss does not move at all	learning rate too low; frozen layers; `zero_grad` or `step` missing; dead ReLUs
Training good, validation bad	overfitting; train/validation from different sources; leakage the other way
Validation better than training	dropout or augmentation active only in training (normal); evaluation in train mode
Accuracy stuck at one class	class imbalance; labels shuffled independently of images

Practitioner's Note

Keep this table next to you during the tutorial. Almost every failed run you will see this week is one of these rows.

Reading training curves



Underfitting: both losses high. Bigger model, train longer, higher learning rate.

Good fit: both low, small gap. Ship it (after the test set).

Overfitting: training keeps falling, validation turns up. More data, augmentation, regularisation, early stopping.

Part 8

Wrapping up

Key takeaways

1. A CNN stacks conv $\rightarrow$ ReLU $\rightarrow$ pool blocks, then a classifier head. Track shapes; count parameters $(K^2 C_{\text{in}} + 1) C_{\text{out}}$.
2. Train with **cross-entropy**: it rewards confidence in the right class.
3. **Backpropagation** = chain rule; each block only needs its local derivative.
4. ReLU avoids vanishing gradients.
5. The **learning rate** matters most; momentum and Adam smooth the path.
6. Watch training *and* validation curves; read the confusion matrix.
7. Fight overfitting with augmentation, dropout, weight decay, early stopping; batch norm makes deep training reliable.

Before the next lecture

Practise (tutorial with the tutors)

- Train today's CNN on Fashion-MNIST yourself; reproduce the curves.
- Break it on purpose: learning rate too high, no `zero_grad`, wrong shapes.
- Add batch norm and dropout; compare.
- Run on a Colab GPU and compare the time per epoch with your CPU.

Read

- Goodfellow et al., *Deep Learning*, Ch. 6 (§6.5) and Ch. 9
- Zhang et al., *Dive into Deep Learning*, Ch. 7

Next lecture: architectures and transfer learning

- LeNet, AlexNet, VGG, Inception, ResNet
- Why residual connections let networks go deep
- Efficient networks for phones
- Seeing what a network looks at (Grad-CAM)
- Transfer learning on real bean-leaf disease photos from Uganda

Credits

Fashion-MNIST: H. Xiao, K. Rasul, R. Vollgraf (Zalando Research), MIT licence. ResNet-18 weights: torchvision (ImageNet). All other networks were trained for this lecture with `Week2/tools/make_week2_figures.py`.

Photographs from Wikimedia Commons:
`douala_ndokoti`: Minette Lontsie, CC BY-SA 4.0
`cat_leaves`: RyanDannyChacko, CC BY-SA 4.0