Deep learning for embedded engineers

A primer: data, training, CNNs and what they cost

Who this is for. You can program and you know embedded systems, but you have not studied neural networks. Putting a neural network on an embedded device means working with code, tools and people that take a lot of vocabulary for granted. This primer gives that vocabulary and the few ideas behind it: why a network is *trained* rather than programmed, what data it needs, how a loss measures its errors, how a convolutional network turns an image into a decision, what that decision costs in memory and arithmetic, and where techniques such as quantization, pruning and distillation fit. It takes about two hours. The "Go further" lines point to the reading list of Section 12. The companion primer *PyTorch essentials* shows the same ideas in code; it comes with a cookbook of copyable snippets and with scripts to run, listed on its first page.

What you need. Matrix–vector products, derivatives and the chain rule. No statistics beyond averages.

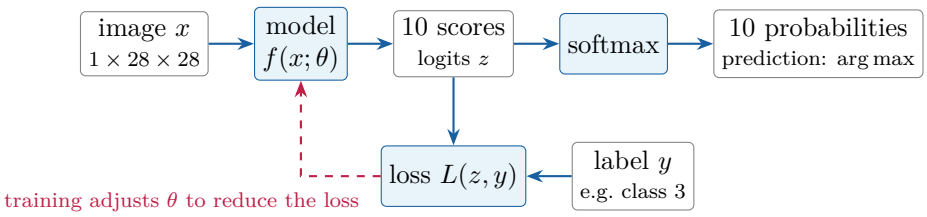
Contents

1 Why train a model?
2 Datasets: training, validation and test sets
3 From a neuron to a network
4 Convolutional neural networks
5 Counting parameters and operations
6 Loss functions
7 How a network learns
8 Hyperparameters and fair comparisons
9 Overfitting and evaluation
10 After training: fitting the model to a device
11 Check your understanding
12 To learn more • 13 Glossary

1 Why train a model?

In ordinary embedded software you write the rules: if the temperature exceeds a threshold, open the valve. For many perception tasks nobody can write the rules. What makes a handwritten 3 a 3, whatever the writer, the pen, the lighting and the position in the image? Instead of writing the rules, we choose a **model**, a function $f(x;\theta)$ with many adjustable numbers θ (its **parameters** or **weights**), and we adjust θ automatically until f gives the right answer on many examples whose answer is known. This adjustment is **training**; the examples replace the specification.

When every training example is a pair (x, y) of an input x and its correct answer y, the **label**, this is **supervised learning**. For image classification, y is one of K classes. (Other settings exist: learning from unlabelled data, or from rewards, but this primer stays with supervised classification.)



The model outputs one score per class. The scores are turned into probabilities (Section 6), and the predicted class is the one with the highest score. During training, a **loss** measures how wrong the scores are for the known label, and an optimisation algorithm changes θ to make the loss smaller (Section 7).

Training and inference. After training, θ is frozen and the model is only evaluated on new inputs: this is **inference**. The two phases have very different costs. Training needs labelled data, many passes over it and a lot of computation, and it is done once, on a workstation or a GPU server. Inference processes one input at a time, for the whole life of the product, and it is what runs on the embedded device. Training is where the accuracy comes from; inference is where the embedded constraints (memory, latency, energy) apply.

Images are tensors. A greyscale image of 28×28 pixels is a grid of 784 numbers; a colour image has three such grids (red, green, blue), called **channels**. We write the shape as $C \times H \times W$ (channels, height, width). A **tensor** is simply a multi-dimensional array; a batch of N images is a tensor of shape $N \times C \times H \times W$. Pixel values are usually scaled to $[0, 1]$ and then **normalised** (shift by a mean, divide by a standard deviation), so that all inputs have a similar range.

Go further: 3Blue1Brown, *Neural networks*, first two videos; Nielsen, chapter 1.

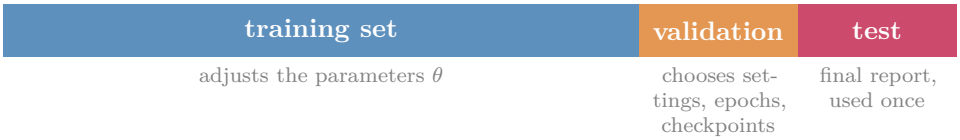
2 Datasets: training, validation and test sets

A **dataset** is a collection of inputs with their labels. A model learns whatever the data shows it, including its accidents, so the data matters as much as the model: labels must be correct, every class needs enough examples, and the examples must look like what the device will see in use (same kind of camera, lighting, background). A model trained on clean, centred digits fails on a tilted, badly lit photo it has never seen anything like.

Dataset	Input	Training + evaluation	Classes	What it is
MNIST	$1 \times 28 \times 28$	$60\,000 + 10\,000$	10	Handwritten digits; the "hello world" of image classification.
FashionMNIST	$1 \times 28 \times 28$	$60\,000 + 10\,000$	10	Clothing items in the MNIST format; harder.
CIFAR-10	$3 \times 32 \times 32$	$50\,000 + 10\,000$	10	Small colour photos of objects and animals.
ImageNet	usually $3 \times 224 \times 224$	about 1.28 million + 50 000 val.	1000	Large photos; the reference benchmark and the source of most pre-trained models.

The ImageNet row gives the ILSVRC training and validation counts; its separate test partition contains 100 000 images. The other rows give training and test counts.

What matters is not how a model behaves on images it has memorised, but on images it has never seen. The labelled data is therefore split into three disjoint sets, each with one role:



- The **training set** is the only data the optimiser sees.
- The **validation set** answers design questions: which learning rate, how many epochs, which of two models, which checkpoint to keep. It is usually held out from the training data, often 10 to 20 %.
- The **test set** gives the final, honest estimate of accuracy. If you look at test results to make a choice, the test set has become a validation set and the reported number is optimistic.

Keep the class proportions similar in each split, and never let the same image, or two near-copies of it, appear in two splits (a *leak*). **Data augmentation** (random crops, flips, small shifts or colour changes of the training images) creates plausible new training examples; it is applied to the training set only.

The rule

Decide on validation, report on test. Every number you report should say which split it was measured on.

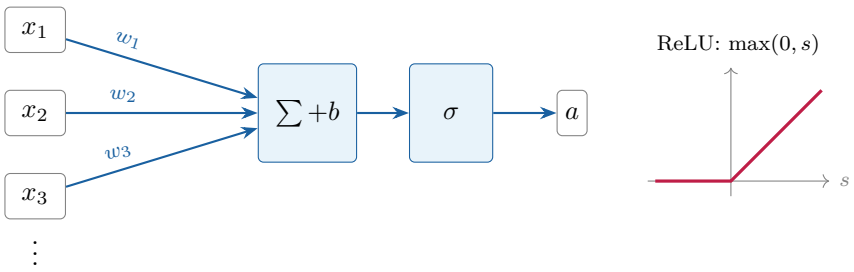
Go further: Google, *Machine Learning Crash Course*, the modules on datasets and generalisation.

3 From a neuron to a network

A **neuron** computes a weighted sum of its inputs, adds a **bias**, and applies a non-linear **activation function** σ:

$$a = \sigma\Big(\sum_{i=1}^{n} w_i x_i + b\Big).$$

A **fully connected** (or *linear*, *dense*) layer is m neurons that read the same n inputs. In matrix form it is $a = \sigma(Wx + b)$ with a weight matrix W of shape $m \times n$ and a bias vector b of length m, so it has $mn + m$ parameters and costs mn multiply–accumulate operations (**MACs**) per input.



The most common activation is the **ReLU**, $\max(0, s)$: it passes positive values and blocks negative ones. Without a non-linearity, stacking layers would be useless: two linear layers in a row are still one linear map. A **multilayer perceptron** (MLP) stacks a few fully connected layers with activations between them; the last layer has one output per class and no activation. Its outputs are the scores z, called **logits**.

Worked example: an MLP for 28×28 images

Flatten the image into a vector of 784 values, then use a hidden layer of 128 neurons and an output layer of 10.

- Hidden layer: $784 \times 128 + 128 = 100\,480$ parameters, $784 \times 128 = 100\,352$ MACs.
- Output layer: $128 \times 10 + 10 = 1\,290$ parameters, $1\,280$ MACs.
- Total: $101\,770$ parameters and $101\,632$ MACs per image. In a fully connected layer every weight is used exactly once per image, so parameters and MACs are almost equal.

Flattening throws away the 2-D structure of the image: the MLP does not know that two pixels are neighbours. That is the motivation for convolutions.

Go further: TensorFlow Playground, to watch a small network learn; Nielsen, chapter 1.

4 Convolutional neural networks

4.1 The convolution

A **convolutional layer** slides a small **kernel** (or *filter*), typically 3×3, over the image. At every position it computes the weighted sum of the pixels under the kernel, plus a bias. The result is a new image, a **feature map**, that is large where the local pattern matches the kernel.

input 5×5

1	2	0	1	3
0	1	3	2	1
2	0	1	0	2
1	3	2	1	0
0	1	0	2	1

$*$

kernel 3×3

1	0	−1
1	0	−1
1	0	−1

$=$

output 3×3

−1	0	−2
−3	1	3
0	1	0

The red window gives the first output value: $(1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1)) + (0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1)) + (2 \cdot 1 + 0 \cdot 0 + 1 \cdot (-1)) = 1 - 3 + 1 = -1$ (bias 0). Moving the window one pixel to the right gives the next value, and so on. This kernel responds to a change of brightness from left to right, a vertical edge. A network does not use hand-made kernels: their values are parameters, learned like any other weight.[1]

Two ideas make convolutions efficient for images. **Locality**: each output looks at a small neighbourhood. **Weight sharing**: the same kernel is used at every position, so a pattern learned in one place is recognised everywhere, with very few parameters.

4.2 Size, padding, stride and channels

For an input of height H, a kernel of size K, **padding** P (zeros added around the border) and **stride** S (step of the window), the output height is

$$H_{\text{out}} = \left\lfloor \frac{H + 2P - K}{S} \right\rfloor + 1,$$

and the same for the width. The example above has $H = 5$, $K = 3$, $P = 0$, $S = 1$, hence $H_{\text{out}} = 3$. With $P = 1$ a 3×3 convolution keeps the size unchanged ("same" padding).

Real layers have several input and output channels. A kernel spans *all* C_{in} input channels, and the layer has one such kernel per output channel. Its weights therefore form a tensor of shape $C_{\text{out}} \times C_{\text{in}} \times K \times K$, plus one bias per output channel, and its output has C_{out} feature maps:

$$\text{parameters} = C_{\text{out}}\, C_{\text{in}}\, K^2 + C_{\text{out}}, \qquad \text{MACs} = H_{\text{out}}\, W_{\text{out}}\, C_{\text{out}}\, C_{\text{in}}\, K^2.$$

4.3 Pooling and the classifier

A **pooling** layer shrinks feature maps. 2×2 **max pooling** with stride 2 keeps the largest value of each 2×2 block, halving height and width; it has no parameters.

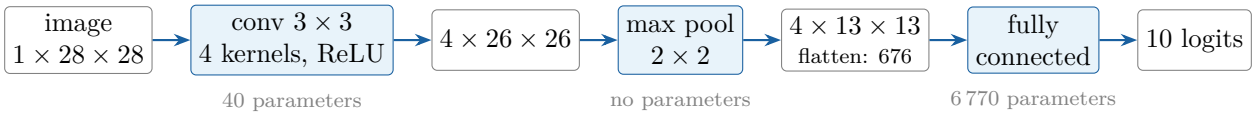
1	3	2	0
4	2	1	5
0	1	3	2
2	6	1	1

max 2×2 →

4	5
6	3

[1] Deep learning libraries compute exactly this sum without flipping the kernel; mathematically it is a cross-correlation, but the name *convolution* is kept.

A typical **CNN** repeats *convolution* → *ReLU* → *pooling* a few times: the feature maps get smaller and more numerous, and they encode more abstract patterns (edges, then corners and textures, then parts of objects). At the end the maps are **flattened** into a vector and one or more fully connected layers produce the class scores. The figure follows the shapes through a very small CNN for 28×28 digit images.



The convolution has no padding, so 28 becomes $28-3+1=26$; pooling halves 26 to 13; the classifier sees $4 \times 13 \times 13 = 676$ values.

Go further: CNN Explainer, an interactive CNN in the browser; CS231n notes, "Convolutional Neural Networks"; Dive into Deep Learning, chapter "Convolutional Neural Networks".

5 Counting parameters and operations

On an embedded device two numbers matter before anything is measured: how many parameters must be stored, and how many MACs must be computed for each image. They are *not* the same thing.

Worked example: the small CNN above

Layer	Output shape	Parameters	MACs per image
Conv 3×3, $1 \to 4$ channels	$4 \times 26 \times 26$	$4 \cdot 1 \cdot 9 + 4 = 40$	$26 \cdot 26 \cdot 4 \cdot 1 \cdot 9 = 24\,336$
Max pool 2×2	$4 \times 13 \times 13$	0	—
Fully connected $676 \to 10$	10	$676 \cdot 10 + 10 = 6\,770$	$676 \cdot 10 = 6\,760$
Total		6 810	31 096

The convolution holds less than 1 % of the parameters but needs 78 % of the MACs: its 36 weights are reused at each of the 26×26 positions. The fully connected layer is the opposite: many weights, each used once.

The contrast grows with image size. A 3×3 convolution from 3 to 16 channels on a $3 \times 32 \times 32$ colour image, with padding 1, has $16 \cdot 3 \cdot 9 + 16 = 448$ parameters but costs $32 \cdot 32 \cdot 16 \cdot 27 = 442\,368$ MACs: each weight is used 1 024 times per image.

What the numbers tell an embedded engineer

- **Parameters** set the storage: 6 810 parameters take 27 240 bytes in 32-bit floating point and 6 810 bytes in 8-bit integers. Storage decides whether the weights fit in on-chip memory or must be read from external DRAM.
- **MACs** set the arithmetic work, a first estimate of compute time and energy.
- **Activations** (the intermediate feature maps, here $4 \times 26 \times 26 = 2\,704$ values after the convolution) set the buffer sizes and the data movement.
- None of them is latency. Latency depends on how the hardware maps the layers: memory traffic, parallelism and utilisation. MACs are an estimate to be checked by measurement on the target.

Go further: Sze et al., *Efficient Processing of Deep Neural Networks*: why data movement, not only the MACs, sets the energy of a network.

6 Loss functions

6.1 Logits, softmax and cross-entropy

The last layer outputs K real numbers $z_1, \dots, z_K$, the **logits**. They can be negative and do not sum to one. The **softmax** turns them into probabilities:

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad 0 < p_k < 1, \qquad \sum_k p_k = 1.$$

The softmax keeps the order of the logits, so the predicted class $\arg\max_k z_k$ is the same with or without it.

To train, we need one number that says how wrong the prediction is: the **loss**. Training is nothing more than making this number small, so the loss defines what the model learns. For classification it is the **cross-entropy loss**: minus the logarithm of the probability given to the correct class y,

$$L = -\ln p_y.$$

It is 0 when the model gives probability 1 to the right class, and it grows without limit as that probability goes to 0. Averaged over many examples, it rewards confident correct answers and punishes confident wrong ones much more than uncertain ones.

Worked example: three classes

Logits $z = (2.0,\ 1.0,\ 0.1)$. Then $e^z \approx (7.389,\ 2.718,\ 1.105)$, their sum is 11.213, and the softmax gives $p \approx (0.659,\ 0.242,\ 0.099)$. The model predicts class 1. (Classes are numbered 1 to K here, as in most mathematical texts; PyTorch numbers them 0 to $K-1$, so class 1 here is index 0 in code.)

- If the label is class 1: $L = -\ln 0.659 \approx 0.417$, a small loss.
- If the label is class 3: $L = -\ln 0.099 \approx 2.317$, a large loss, although the prediction is "only" one place off.
- A model that knows nothing gives every class the same probability $1/K$, so its loss is $\ln K$: 1.099 for 3 classes, 2.303 for 10. A training loss stuck near $\ln K$ means the model has learned nothing yet.

Accuracy, the fraction of correctly classified images, is what we report, but it cannot be used for training: it changes in jumps and gives no direction for improvement. The loss is smooth, so it can be minimised.

6.2 Other losses

The loss follows from the task. The ones you will meet most often:

Task	Model output	Loss (PyTorch name)
One class among K	K logits	Cross-entropy (`nn.CrossEntropyLoss`)
Yes or no, or several independent labels	one logit per label	Binary cross-entropy on a sigmoid (`nn.BCEWithLogitsLoss`)
A number, such as a distance or a temperature	the value itself	Mean squared error (`nn.MSELoss`) or mean absolute error (`nn.L1Loss`)

A loss can also have several terms added together: a penalty on large weights (Section 7), or a term that asks a small network to imitate the outputs of a larger one (distillation, Section 10). The training procedure does not change: it minimises whatever single number the loss returns.

Go further: Nielsen, chapter 3 (cross-entropy); Prince, chapter 5 (loss functions).

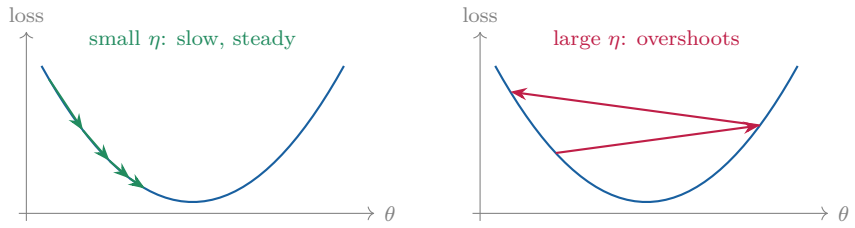
7 How a network learns

7.1 Gradient descent

The loss depends on all parameters θ. Its **gradient** $\nabla_\theta L$ is the vector of partial derivatives: for every parameter, how much the loss changes when that parameter increases slightly. Moving each parameter a small step against its derivative decreases the loss:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L.$$

The step size η is the **learning rate**. Too small, and training is slow; too large, and the steps overshoot and the loss oscillates or explodes.



7.2 Backpropagation

A network is a chain of simple functions, so its gradient follows from the chain rule. One **forward pass** computes the outputs and the loss and keeps the intermediate values; one **backward pass** goes through the layers in reverse and applies the chain rule to get the derivative of the loss with respect to every parameter. This is **backpropagation**. Frameworks such as PyTorch do it automatically: you write the forward computation, and one call computes all gradients. A backward pass costs about twice the forward pass, which is one reason why training happens on a workstation and only inference runs on the embedded device.

7.3 Mini-batches, iterations and epochs

Computing the loss over the whole training set before each step would be slow. Instead each step uses a small random subset, a **mini-batch** of B examples (typically 32 to 256), and averages their losses. This is **stochastic gradient descent** (SGD): each step is noisy but cheap, and the noise averages out. One **iteration** is one step on one mini-batch; one **epoch** is one pass over the whole training set, with the examples reshuffled each time.

Worked example: how many steps?

With 50 000 training images and $B = 128$, an epoch has $\lceil 50\,000/128 \rceil = 391$ iterations (the last batch holds the remaining 80 images). Twenty epochs are 7 820 parameter updates.

The update rule itself is the **optimizer**. Plain SGD often adds **momentum** (it keeps part of the previous step, which smooths the path); **Adam** adapts the step size of each parameter automatically and is a robust default for small networks. **Weight decay** adds a small penalty on large weights, a simple form of regularisation (Section 9).

Training does not always start from random weights. Starting from a model already trained on a large dataset and training it a little more on new data is **fine-tuning**; it needs far less data and time. The same word is used for the short extra training that repairs a model after it has been pruned or quantized (Section 10).

Go further: 3Blue1Brown, *Neural networks*, videos on gradient descent and backpropagation; Nielsen, chapter 2; Karpathy, *Neural Networks: Zero to Hero*, first lecture (backpropagation written from scratch).

8 Hyperparameters and fair comparisons

Parameters are learned; **hyperparameters** are the choices you make before training. They are tuned on the validation set, never on the test set.

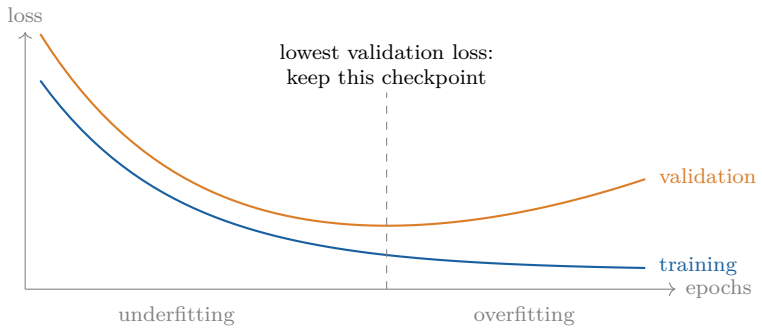
Hyperparameter	Effect
Learning rate η	The most sensitive setting. Too high: the loss diverges. Too low: training stalls. Often reduced during training (a *schedule*).
Batch size B	Larger batches give smoother gradients and better hardware use but fewer updates per epoch; changing B often requires changing η.
Number of epochs	Too few: underfitting. Too many: overfitting and wasted time. Chosen with the validation curve.
Optimizer	SGD with momentum or Adam; each has its own good learning-rate range.
Weight decay	Penalises large weights; reduces overfitting.
Architecture	Depth, width (channels or neurons per layer), kernel sizes. Sets accuracy and cost together.
Augmentation	Random transformations of training images; reduces overfitting.
Random seed	Sets the initial weights and the shuffling. Different seeds give slightly different results.

A fair comparison

To claim that method A beats method B, train both on the same splits, with the same budget (epochs, data, augmentation), select both on validation, and report both on test. Change one thing at a time. Two runs that differ by a few tenths of a percent may differ only by their seed: repeat with several seeds before drawing conclusions.

9 Overfitting and evaluation

A model with enough parameters can memorise its training set. The training loss then keeps falling, while the validation loss stops improving and rises again: the model has learned details of the training images that do not generalise. This is **overfitting**. The opposite, a model too small or trained too briefly to fit even the training data, is **underfitting**.



Schematic shape of the curves, not a measurement.

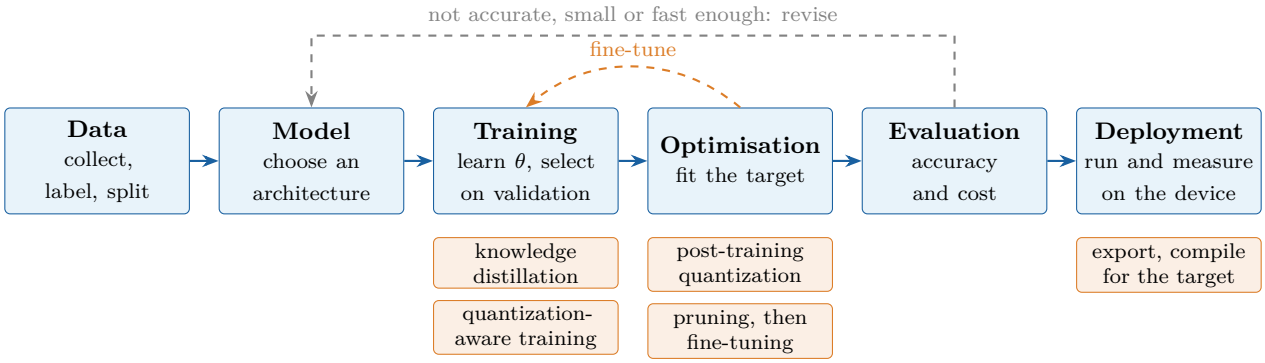
Remedies: more data or augmentation, weight decay, a smaller model, or **early stopping**: keep the checkpoint with the best validation score and stop when it no longer improves for a few epochs (the *patience*).

Measuring the result. **Accuracy** is the fraction of correct predictions. It hides which classes are confused: a **confusion matrix** counts, for each true class, how often each class was predicted. The **recall** of a class is the fraction of its images that were recognised; the **macro recall** averages it over classes, so a rare class counts as much as a frequent one. Always state on which split a number was measured, and report the cost (parameters, MACs, latency) next to the accuracy.

Go further: CS231n notes, "Neural Networks Part 3" (babysitting the learning process); Google, *Machine Learning Crash Course*, overfitting.

10 After training: fitting the model to a device

A trained network is a fixed sequence of tensor operations with fixed weights, usually stored as 32-bit floating-point numbers. It is often too large, too slow or too power-hungry for the device as it is, so the work does not stop at training. The figure places the main techniques in the complete workflow. Each is a subject of its own; here we only say what it does and when it acts.



Orange: where quantization, pruning and distillation act. Dashed arrows: back to an earlier stage.

- **Quantization** stores weights and activations with fewer bits, typically 8-bit integers instead of 32-bit floats: four times less memory, and integer arithmetic that is much cheaper in hardware; many accelerators accept only quantized models. It is applied after training, using a few hundred images to measure the value ranges (*post-training quantization*), or simulated during training when the accuracy loss is too large (*quantization-aware training*).
- **Pruning** removes the weights, channels or whole filters that contribute least, then fine-tunes the network to recover its accuracy. Removing whole channels shrinks the layers and their MACs on any hardware; scattered zero weights only help hardware or software that skips them.
- **Knowledge distillation** trains a small *student* network to match the labels and also the outputs of a large, already trained *teacher*. The teacher's probabilities say more than the label alone (this 3 looks a little like an 8), which often helps the small network. It changes the training loss, not the deployed network: at inference only the student runs.
- **Architecture** is the first lever: a network designed small, by hand or by automated search, may need none of the above.

Every one of these changes the model, so the rules of Sections 2 and 8 still apply: compare before and after on the same test set, and report accuracy together with the cost. On the device, the optimised model is exported to a file, compiled for the target (a processor, a neural-network accelerator or dedicated logic on an FPGA), and its latency and energy are measured there, not estimated.

Go further: MIT 6.5940, *TinyML and Efficient Deep Learning Computing* (lectures on pruning, quantization and distillation); Nagel et al. (quantization); Blalock et al. (pruning); Hinton et al. (distillation); Han et al., *Deep Compression*.

11 Check your understanding

If you can answer these without looking back, you have the basics. Answers are below.

1. Why is a perception task trained rather than programmed? What replaces the specification?

2. What is the difference between a parameter and a hyperparameter? Give two examples of each.
3. Why is the test set used only once, at the end?
4. A batch of 64 colour images of 32×32 pixels is a tensor of which shape?
5. A 3×3 convolution maps 3 input channels to 8 output channels. How many parameters does it have?
6. What is the output size of a 3×3 convolution on a 32×32 input, stride 1, without padding? With padding 1?
7. Why does a convolution have far more MACs than parameters, while a fully connected layer has about as many?
8. What are logits, and what does the softmax do with them? Which loss would you use to predict a temperature?
9. At the start of training on a 10-class problem, the loss is about 2.30. Is that surprising?
10. What happens in one iteration of training? And in one epoch?
11. The training loss keeps decreasing but the validation loss increases. What is happening, and what do you do?
12. Which of quantization, pruning and distillation changes the number format, which removes parts of the network, and which changes only the training?

Answers. (1) Nobody can write rules that cover all the variations of real inputs; labelled examples replace the specification, and training fits the parameters to them. (2) Parameters are learned by training (weights, biases); hyperparameters are chosen before (learning rate, batch size, epochs, architecture). (3) Any decision based on it makes it optimistic; it must stay an unbiased final estimate. (4) $64 \times 3 \times 32 \times 32$. (5) $8 \cdot 3 \cdot 3 \cdot 3 + 8 = 224$. (6) 30×30; 32×32. (7) The convolution reuses each weight at every output position; a fully connected weight is used once per image. (8) The raw class scores of the last layer; the softmax turns them into probabilities that sum to one, without changing their order. A temperature is a number: mean squared (or absolute) error. (9) No: $\ln 10 \approx 2.30$ is the loss of a model that guesses uniformly, as expected before learning. (10) One iteration: forward pass on a mini-batch, loss, backward pass, one parameter update. One epoch: as many iterations as needed to see every training example once. (11) Overfitting: keep the checkpoint with the best validation score (early stopping), add augmentation or weight decay, or use a smaller model. (12) Quantization changes the number format; pruning removes weights, channels or filters; distillation changes only the training (the loss), not the deployed network.

12 To learn more

All free online, grouped by what they are good for. Start with intuition; the last group is for later.

For intuition: pictures and animations, little mathematics

- 3Blue1Brown, *Neural networks*: short animated videos on what a network computes, gradient descent and backpropagation. https://www.3blue1brown.com/topics/neural-networks
- CNN Explainer: an interactive CNN that shows every convolution, activation and pooling step in the browser. https://poloclub.github.io/cnn-explainer/
- TensorFlow Playground: train a tiny network in the browser and watch it learn. https://playground.tensorflow.org/
- Google, *Machine Learning Crash Course*: short modules with exercises on data, splits, overfitting and neural networks. https://developers.google.com/machine-learning/crash-course

For the mathematics: from gentle to complete

- M. Nielsen, *Neural Networks and Deep Learning*: the gentlest derivation of backpropagation and cross-entropy. http://neuralnetworksanddeeplearning.com/
- Stanford CS231n, *Convolutional Neural Networks for Visual Recognition*: lecture notes on linear classifiers, optimisation, backpropagation and CNNs, with practical advice. https://cs231n.github.io/

- S. Prince, *Understanding Deep Learning* (MIT Press, 2023): a modern textbook with clear figures; loss functions, training and CNNs in depth. https://udlbook.github.io/udlbook/
- I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, 2016): the reference for the theory; demanding. https://www.deeplearningbook.org/

For hands-on PyTorch: read and run code (the official PyTorch tutorials are listed in the PyTorch primer)

- A. Zhang, Z. Lipton, M. Li, A. Smola, *Dive into Deep Learning*: every chapter with runnable PyTorch code. https://classic.d2l.ai/chapter_convolutional-neural-networks/index.html
- A. Karpathy, *Neural Networks: Zero to Hero*: videos that build backpropagation and small networks from scratch in Python, then in PyTorch. https://karpathy.ai/zero-to-hero.html

For advanced study: efficient and embedded deep learning

- MIT 6.5940 (S. Han), *TinyML and Efficient Deep Learning Computing*: lectures on pruning, quantization, distillation, architecture search and microcontrollers. https://hanlab.mit.edu/courses/2024-fall-65940
- V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*, 2017: the hardware view, from dataflows to energy. https://arxiv.org/abs/1703.09039
- M. Nagel et al., *A White Paper on Neural Network Quantization*, 2021: post-training quantization and quantization-aware training in practice. https://arxiv.org/abs/2106.08295
- D. Blalock et al., *What is the State of Neural Network Pruning?*, 2020: a survey, and how to compare pruning methods fairly. https://arxiv.org/abs/2003.03033
- G. Hinton, O. Vinyals, J. Dean, *Distilling the Knowledge in a Neural Network*, 2015: the original distillation paper, short and readable. https://arxiv.org/abs/1503.02531
- S. Han, H. Mao, W. Dally, *Deep Compression*, 2015: pruning, quantization and coding combined. https://arxiv.org/abs/1510.00149

13 Glossary

Activation	Non-linear function after a layer, such as ReLU $\max(0, s)$; also a layer's outputs.
Batch, mini-batch	Group of examples processed together; one gradient step per mini-batch.
Checkpoint	Saved copy of the parameters at some point of training.
Channel	One 2-D grid of a tensor: R, G, B for an image, one feature map inside a CNN.
Cross-entropy	Classification loss $-\ln p_y$, computed from the logits through the softmax.
Distillation	Knowledge distillation: a small student network learns from a large teacher and the labels.
Epoch	One pass over the whole training set.
Feature map	Output grid of a convolution kernel: where its pattern appears.
Fine-tuning	Further training of a trained model, on new data or after pruning or quantization.
Gradient	Vector of the derivatives of the loss with respect to every parameter.
Hyperparameter	Setting chosen before training: learning rate, batch size, epochs, architecture.
Inference	Running a trained model on new inputs, without updating it.
Label	The correct answer of a training example, for example its class.
Learning rate	Step size η of gradient descent.
Logits	Raw class scores of the last layer, before the softmax.
Loss	Single number measuring how wrong the model is on a batch; training minimises it.
MAC	Multiply–accumulate, $a \leftarrow a + w\,x$; the unit used to count the work of a network.
Overfitting	Fitting details of the training data that do not generalise; validation gets worse.
Parameters (weights)	Numbers learned by training: kernels, weight matrices, biases.
Pruning	Removing weights, channels or filters that contribute little, then fine-tuning.
Quantization	Representing weights and activations with fewer bits, for example 8-bit integers.
Softmax	Turns logits into probabilities: $p_k = e^{z_k} / \sum_j e^{z_j}$.
Supervised learning	Learning from examples paired with their correct answers.
Tensor	Multi-dimensional array; images in a batch have shape $N \times C \times H \times W$.
Validation set	Held-out data used to make choices; the test set is kept for the final report.