# Quantitative Research Methods

## Chapter 10: Deep Learning

Prof. Dr. Christoph Weisser

HSBI

Summer Semester 2026

# The course at a glance

| | Lecture | Topic |
|---|---|---|
| | 1 | Ch. 1 + 2 — Introduction; what is statistical learning |
| | 2 | Ch. 2 — Accuracy, bias–variance, Bayes classifier, KNN |
| | 3–4 | Ch. 3 — Linear regression (estimation, inference, diagnostics) |
| | 5–6 | Ch. 4 — Classification (logistic, LDA/QDA, naive Bayes, ROC) |
| | 7 | Ch. 5 — Resampling (validation, CV, bootstrap) |
| | 8 | Ch. 6 — Model selection and regularisation (ridge, lasso) |
| | 9 | Ch. 7 — Moving beyond linearity (splines, GAMs) |
| | 10 | Ch. 8 — Tree-based methods (trees, forests, boosting) |
| ▶ | **11** | **Ch. 10 — Deep learning (MLPs, CNNs, training)** |
| | 12 | Ch. 13 — Multiple testing (FWER, FDR) |

Twelve sessions of 180 minutes. ▶ marks today's chapter. Short exercises every ∼20 min; extended exercises every ∼45 min; each chapter has a companion lab notebook.

# Contents



Optional and advanced material is collected in the **appendix** at the end of the deck.

# Notation in this chapter

| Symbol | Meaning |
|---|---|
| $X_j$, $p$ | input feature $j$; number of input features |
| $K$ | number of hidden units in a layer |
| $g(\cdot)$ | activation function (sigmoid, tanh, ReLU) |
| $A_k$ | activation of hidden unit $k$: $A_k = g\left(w_{k0} + \sum_j w_{kj} X_j\right)$ |
| $w_{kj}$, $w_{k0}$ | weight from input $j$ into hidden unit $k$; its bias |
| $\beta_k$, $\beta_0$ | output weight of hidden unit $k$; output bias |
| softmax | output activation $e^{z_m} / \sum_{m'} e^{z_{m'}}$ for multiclass labels |
| $W$, $F$, $P$, $S$ | conv arithmetic: input width, filter size, padding, stride |
| $\theta$ | all weights and biases of the network collectively |
| $\eta$ | learning rate of (stochastic) gradient descent |
| $\ell$ | per-observation loss (squared error, cross-entropy) |

# Sources and references

## Primary textbook

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). *An Introduction to Statistical Learning, with Applications in Python.* Springer.

# Learning objectives

By the end of this chapter you will be able to:

- **Define** a single-layer and multilayer perceptron.
- **Explain** how convolutional networks exploit local structure and translation invariance.
- **Outline** recurrent networks and the basic transformer idea.
- **Train** a small network with stochastic gradient descent + backpropagation.
- **Reason** about when deep learning is worth it.

# Roadmap of this chapter

**A tour of modern deep learning**

1. Single-layer networks (logistic regression in disguise).
2. Multilayer perceptrons.
3. Convolutional networks for images.
4. Recurrent networks and transformers for sequences.
5. How they are trained (SGD, backprop) and regularised.

# Outline

# What is deep learning?

A renaissance of neural networks driven by GPUs, large labelled datasets, and algorithmic improvements (dropout, batch norm, attention).

**Takeaway**

Dominant in image, speech, and natural-language tasks. Less consistently dominant on small to medium tabular data — where gradient-boosted trees often win.

**In industry — Banking and insurance: the tabular reality check**

On the wide customer, policy and transaction tables that dominate a bank's or insurer's analytics stack, gradient boosting (Chapter 8) usually matches or beats a neural network at a fraction of the cost — and is far easier to document for model validation. Deep learning earns its keep on the *unstructured* assets those firms also hold: scanned documents, call recordings, images.

# Where this chapter is used in industry

| Sector | Concrete application | Which decision it drives |
|---|---|---|
| Manufacturing | CNN on line-scan camera images: surface defects, missing components, assembly faults | Scrap, rework or stop the line |
| Healthcare | Radiology screening and triage; many AI imaging devices are cleared by regulators | Which scans the radiologist reads first |
| Insurance, banking | OCR plus sequence models on claims forms, invoices, contracts, KYC documents | Straight-through processing; route exceptions to a human |
| Retail, streaming | Embedding-based recommendation and two-tower retrieval | What appears on the home page and in the ranking |
| Telco, services | Speech recognition and call-centre analytics | Route, summarise and quality-check conversations |
| Retail, energy | One network trained across thousands of related demand series | Purchasing, staffing and dispatch volumes |

**In industry — The common thread**

Every row has an *unstructured* input — pixels, characters, waveforms — where the network learns the features a human would otherwise hand-code.

# Industry case in depth: automated visual inspection

## In industry — Semiconductors and automotive: end-of-line inspection

**Response.** A binary label per image, *defect* / *no defect*, usually with a location (bounding box or heat map) so the operator can see what was flagged.
**Inputs.** Line-scan or area-scan camera images of each unit under fixed lighting; often several views per unit, sometimes with the process metadata (machine, shift, recipe) attached for later diagnosis.

## What the fitted model produces and who acts on it

A defect probability per image. A threshold turns it into an action: pass, divert to **rework**, **scrap**, or — if the flagged rate spikes — **stop the line** and call process engineering. Quality engineering owns the threshold; the production manager signs off on any rule that can halt output. The flag itself lands on the **shift supervisor**, who must act on it and answer for the stopped line.

## Honest caveats

Defects are *rare*, so labelled positives are scarce and badly imbalanced — augmentation, transfer learning and a cost-weighted threshold do the heavy lifting. A false stop is expensive, a missed defect can be worse, and the model must be *revalidated* whenever the product, the fixture or the lighting changes.

# Outline

# Single hidden layer

$$\hat{f}(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left( w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right).$$

## Reading the formula symbol by symbol

- $X_j$: input feature $j$ ($j = 1, \ldots, p$); $g$: nonlinear activation (sigmoid, tanh, ReLU).
- $w_{kj}$: weight from input $j$ into hidden unit $k$; $w_{k0}$: its bias.
- $K$: number of hidden units (activations $A_k = g(\cdot)$); $\beta_k$: unit $k$'s output weight, $\beta_0$ the output bias.

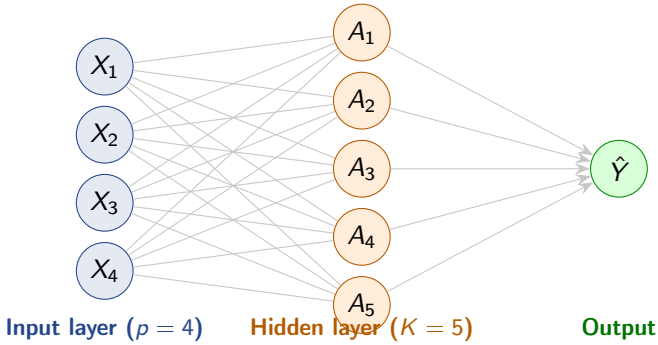
## Takeaway

Read it inside-out: linear combination $\rightarrow$ nonlinearity $g$ $\rightarrow$ linear combination. Logistic regression is the *no hidden layer* case: a sigmoid applied straight to $\beta_0 + \sum_j \beta_j X_j$.

# Single-layer network as a graph


*Source: Figure 10.1 — ISLP, James et al. (2023).*

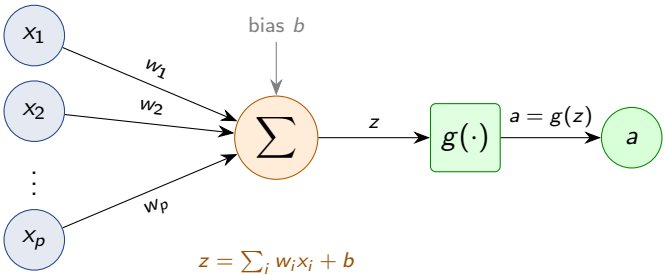
# Feed-forward network as a labelled graph




## Takeaway

Each hidden unit computes $A_k = g\left(w_{k0} + \sum_j w_{kj} X_j\right)$; the output linearly combines them, $\hat{Y} = \beta_0 + \sum_k \beta_k A_k$. "Fully connected" means every input feeds every hidden unit.

# Anatomy of a single neuron




**Takeaway**

A neuron takes a weighted sum of its inputs plus a bias, then applies a nonlinearity $g$. Composing layers of these units builds the network above.

# Activation functions: sigmoid vs. ReLU

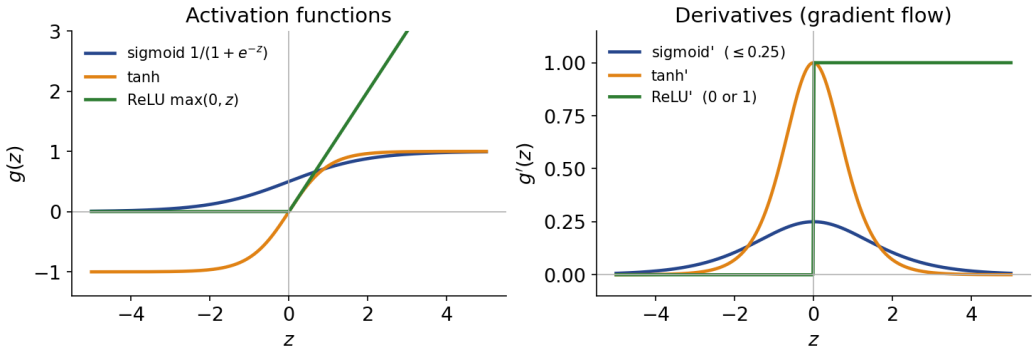
## How to read this

- **Sigmoid**: $g(z) = 1/(1 + e^{-z})$, output in $(0, 1)$. Saturates at $\pm\infty$ — vanishing gradients.
- **Tanh**: $g(z) = \tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$, output in $(-1, 1)$; same S-shape but zero-centred.
- **ReLU**: $g(z) = \max(0, z)$. Today's default — sparser activations, faster training, better gradient flow.

## Why it matters

The activation is the *only* nonlinearity in the network: remove it and stacked layers collapse to one linear map (Exercise 10.2). ReLU is preferred because its gradient stays healthy through deep stacks.

# Activation functions and their gradients




**Takeaway**

ReLU's gradient is 1 on the active region, so signals pass undiminished; the sigmoid derivative never exceeds 0.25 and vanishes in the tails — the root of vanishing gradients in deep nets.

# Exercise 10.1 — Forward pass by hand [Math]

## Exercise

Consider a single hidden-layer network with input $X = (X_1, X_2) = (2, 3)$, $K = 2$ ReLU hidden units, and a linear output. The parameters are

$$\begin{aligned} &\text{Unit 1: } w_1 = (1, 1),\ w_{10} = -4, \\ &\text{Unit 2: } w_2 = (-1, 2),\ w_{20} = 0, \end{aligned} \qquad \beta_0 = 1,\ \beta_1 = 2,\ \beta_2 = -1.$$

1. Compute the two hidden activations $A_1, A_2$ using $g(z) = \max(0, z)$.
2. Compute the network output $\hat{f}(X)$.
3. Treating the output as a binary-classification logit, apply the sigmoid $\sigma(z) = 1/(1 + e^{-z})$ to obtain $\Pr(Y = 1 \mid X)$ and give the predicted class at threshold 0.5.

# Solution — Exercise 10.1 (1/2)

## Solution

**Method.** Work inside-out: each hidden unit first forms its affine pre-activation $z_k = w_{k0} + w_k^\top X$, then applies the nonlinearity $g(z) = \max(0, z)$ element-wise.

**Step 1 — pre-activations.**

$$z_1 = w_{10} + w_1^\top X = -4 + (1)(2) + (1)(3) = -4 + 5 = 1,$$

$$z_2 = w_{20} + w_2^\top X = 0 + (-1)(2) + (2)(3) = -2 + 6 = 4.$$

**Step 2 — ReLU activations.** Both pre-activations are positive, so ReLU passes them through unchanged:

$$A_1 = \max(0, 1) = 1, \qquad A_2 = \max(0, 4) = 4.$$

## Common mistake

Forgetting the bias $w_{10} = -4$ and reporting $z_1 = 5$ — always add the bias before applying $g$.

# Solution — Exercise 10.1 (2/2)

## Solution

**Step 3 — output.**

$$\hat{f}(X) = \beta_0 + \beta_1 A_1 + \beta_2 A_2 = 1 + (2)(1) + (-1)(4) = -1.$$

**Step 4 — sigmoid probability.**

$$\sigma(-1) = \frac{1}{1+e^{1}} = \frac{1}{1+2.718} = 0.269.$$

Since $0.269 < 0.5$, the predicted class is $Y = 0$. *Shortcut check:* $\sigma$ is monotone, so $\sigma(z) \geq 0.5 \iff z \geq 0$ — the negative logit $-1$ already announces class 0 before any arithmetic.

**Take-away.** A forward pass is just alternating affine maps and element-wise nonlinearities. Note that if $z_1$ had been negative, ReLU would have set that unit to exactly 0, "switching it off".

## Common mistake

Squashing the hidden units with the sigmoid too — here $g$ is ReLU; only the output logit is passed through $\sigma$.

# Exercise 10.2 — Why nonlinearity? [Concept]

## Exercise

1. Show algebraically that a two-layer network with *identity* activations, $A^{(1)} = W^{(1)}X + b^{(1)}$ and $\hat{Y} = W^{(2)}A^{(1)} + b^{(2)}$, is equivalent to a single linear model. What does this imply for stacking layers without a nonlinearity?
2. Give two concrete advantages of ReLU over the sigmoid activation.
3. Explain the "vanishing gradient" problem of the sigmoid using its derivative.

# Solution — Exercise 10.2 (1/2)

## Solution

**Part 1 — linear collapse.** Substituting,

$$\hat{Y} = W^{(2)}\left(W^{(1)}X + b^{(1)}\right) + b^{(2)} = \underbrace{\left(W^{(2)}W^{(1)}\right)}_{W'}X + \underbrace{\left(W^{(2)}b^{(1)} + b^{(2)}\right)}_{b'} = W'X + b'.$$

This is a single affine (linear) map. Hence *any* number of linear layers collapses to one linear layer: depth buys nothing without a nonlinearity. Nonlinear $g$ is what lets a network approximate curved decision boundaries and functions. *Common mistake: hoping the biases add flexibility — they collapse too, into the single vector $b'$.*

**Part 2 — ReLU advantages.**

- *No saturation for $z > 0$*: the derivative is exactly 1, so gradients pass through undiminished, enabling deep networks to train.
- *Cheap and sparse*: $\max(0, z)$ is a single comparison; it also sets many units to exactly 0, giving sparse, efficient representations.

# Solution — Exercise 10.2 (2/2)

## Solution

**Part 3 — vanishing gradients.** For $\sigma(z) = 1/(1 + e^{-z})$ the derivative is

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big) \leq 0.25,$$

with its maximum at $z = 0$, where $\sigma'(0) = 0.5 \times 0.5 = 0.25$, and $\sigma'(z) \to 0$ as $|z| \to \infty$ (already at $z = 5$, $\sigma' \approx 0.0066$). In backpropagation the layer gradients *multiply*, so through $L$ sigmoid layers the signal shrinks by up to $0.25^L$ — for $L = 10$ that is $0.25^{10} \approx 10^{-6}$: it vanishes, and early layers barely learn. ReLU's derivative of 1 on the active region avoids this.

**Interpretation.** Vanishing gradients are a property of the *product* of many small local derivatives, not of any one layer: each sigmoid layer is harmless alone, but ten in a row mute the error signal reaching the early weights.

## Common mistake

Evaluating the derivative as $\sigma'(z) = z(1 - z)$ — the formula plugs in the *output* $\sigma(z)$, not the pre-activation $z$ itself.

# Extended Exercise 10.1 — Tiny network by hand [Math]

## Extended exercise (15 min)

A network has two inputs, one hidden layer of **two ReLU units**, and a single **sigmoid** output. Take input $x = (x_1, x_2) = (1, 2)$ and target $y = 0$, with

$$\text{Unit 1: } w_1 = (1,1),\ b_1 = -1; \qquad \text{Unit 2: } w_2 = (0,1),\ b_2 = 0;$$

$$\text{logit } s = v_1 a_1 + v_2 a_2 + c, \quad v = (1, -\tfrac{1}{2}),\ c = 0, \quad \hat{y} = \sigma(s).$$

Use squared-error loss $L = \frac{1}{2}(\hat{y} - y)^2$ and learning rate $\eta = 0.1$.

1. **Forward pass:** find $z_k$, $a_k = \max(0, z_k)$, the logit $s$, $\hat{y} = \sigma(s)$ and the loss $L$.
2. **Backprop:** write the chain rule for $\partial L / \partial w_{11}$ (the $x_1 \to$unit-1 weight) and evaluate each factor.
3. **Update:** take one gradient-descent step on $w_{11}$ and interpret it.

# Solution — Extended Exercise 10.1 (1/3)

## Solution — Forward pass

**Method.** Alternate affine maps and nonlinearities, $z_k \to a_k \to s \to \hat{y} \to L$, and keep every intermediate value — backpropagation will reuse each one.

**Hidden pre-activations and ReLU.**

$$z_1 = (1)(1) + (1)(2) - 1 = 2, \qquad a_1 = \max(0, 2) = 2,$$

$$z_2 = (0)(1) + (1)(2) + 0 = 2, \qquad a_2 = \max(0, 2) = 2.$$

**Output logit and sigmoid.**

$$s = (1)(2) + (-\tfrac{1}{2})(2) + 0 = 1, \qquad \hat{y} = \sigma(1) = \frac{1}{1 + e^{-1}} = \frac{1}{1 + 0.368} = 0.731.$$

**Squared-error loss** (target $y = 0$, so the error is $\hat{y}$ itself):

$$L = \tfrac{1}{2}(\hat{y} - y)^2 = \tfrac{1}{2}(0.731)^2 = 0.267.$$

# Solution — Extended Exercise 10.1 (2/3)

## Solution — Backpropagation: the chain rule

The weight $w_{11}$ reaches the loss along $z_1 \to a_1 \to s \to \hat{y} \to L$:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial s} \frac{\partial s}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_{11}}.$$

Evaluate each factor:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = 0.731, \qquad \frac{\partial \hat{y}}{\partial s} = \sigma(s)\big(1 - \sigma(s)\big) = (0.731)(0.269) = 0.197,$$

$$\frac{\partial s}{\partial a_1} = v_1 = 1, \qquad \frac{\partial a_1}{\partial z_1} = 1\{z_1 > 0\} = 1, \qquad \frac{\partial z_1}{\partial w_{11}} = x_1 = 1.$$

The first two factors combine into the reusable "error signal" $\delta = (\hat{y} - y)\,\sigma'(s) = 0.731 \times 0.197 = 0.144$.

## Common mistake

Evaluating $\sigma'(s)$ by plugging $s = 1$ into $u(1-u)$ — the $u$ in $\sigma'(s) = u(1-u)$ is $u = \sigma(s) = 0.731$.

# Solution — Extended Exercise 10.1 (3/3)

## Solution — Gradient step and interpretation

**Multiply the factors.**

$$\frac{\partial L}{\partial w_{11}} = 0.731 \times 0.197 \times 1 \times 1 \times 1 = 0.144.$$

**Gradient-descent step** ($\eta = 0.1$):

$$w_{11} \leftarrow w_{11} - \eta \frac{\partial L}{\partial w_{11}} = 1 - 0.1(0.144) = 0.986.$$

*Check:* redoing the forward pass with $w_{11} = 0.986$ gives $s = 0.986$, $\hat{y} = 0.728$, $L = 0.265 < 0.267$ — the loss indeed fell.

**Interpretation.** $\hat{y} = 0.731$ overshoots the target 0, so the gradient is positive and the step *lowers* $w_{11}$, pulling $\hat{y}$ down. The shared factor $\delta = (\hat{y} - y)\,\sigma'(s) = 0.144$ is the output "error signal": backprop reuses it for *every* weight, multiplying only by that weight's local input. Had $z_1$ been negative, $\partial a_1/\partial z_1 = 0$ would kill the gradient — a "dead" ReLU unit stops learning.

# Outline

# Multilayer perceptron (MLP)

Stack hidden layers:

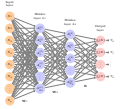
## Forward pass

$$
\begin{aligned}
A^{(1)} &= g(W^{(1)}X + b^{(1)}),\\
A^{(2)} &= g(W^{(2)}A^{(1)} + b^{(2)}),\\
&\vdots\\
\hat{Y} &= h(W^{(L)}A^{(L-1)} + b^{(L)}).
\end{aligned}
$$

### How to read this

$h$ is the output activation: identity for regression, softmax for multiclass classification.

# MLP as a graph: stacking hidden layers



**Takeaway**

Each layer transforms the previous layer's features; stacking them lets the network build progressively more abstract representations from the raw inputs.

*Source: Figure 10.4 — ISLP, James et al. (2023).*

# Where plain MLPs still pay off in business

## In industry — Retail and energy: one network for many series

Retailers and grid operators forecast demand for thousands of related items or nodes. A single "global" network trained across *all* series shares structure — holidays, promotions, weather — that a per-series model cannot borrow, and its output sets purchasing, staffing and dispatch volumes. The error is paid in stock-outs at one end and write-offs at the other.

## In industry — High-cardinality categories: learned embeddings

Business tables are full of identifiers with thousands of levels — SKU, store, postcode, merchant. A network can map each level to a short learned vector (an *embedding*) instead of a wide one-hot block, which is often the one place a neural model clearly beats a tree ensemble on tabular data.

# Exercise 10.3 — Counting parameters [Math]

## Exercise

A fully-connected classifier has the architecture

$$\underbrace{10}_{\text{inputs}} \to 32 \to 16 \to \underbrace{3}_{\text{softmax outputs}},$$

where every hidden and output unit has a bias term.

1. State the general rule for the number of trainable parameters in a dense layer with $n_{\text{in}}$ inputs and $n_{\text{out}}$ units.
2. Count the parameters layer by layer.
3. Give the total.

# Solution — Exercise 10.3 (1/2)

## Solution

**Step 1 — the rule and why.** A dense layer maps $n_{\text{in}} \to n_{\text{out}}$: every input–output pair carries one weight (an $n_{\text{in}} \times n_{\text{out}}$ matrix), and each output unit adds one bias, so

$$\#\text{params} = n_{\text{in}}\, n_{\text{out}} + n_{\text{out}}.$$

The input layer itself holds *no* parameters — it only passes the feature values forward.

**Step 2 — layer by layer.**

$$\begin{aligned}
\text{Layer 1 } (10 \to 32): &\quad 10 \times 32 + 32 = 320 + 32 = 352,\\
\text{Layer 2 } (32 \to 16): &\quad 32 \times 16 + 16 = 512 + 16 = 528,\\
\text{Output } (16 \to 3): &\quad 16 \times 3 + 3 = 48 + 3 = 51.
\end{aligned}$$

# Solution — Exercise 10.3 (2/2)

## Solution

**Step 3 — total.**

$$352 + 528 + 51 = \boxed{931} \text{ trainable parameters.}$$

**Where the parameters live.** The three blocks contribute

$$352/931 \approx 38\%, \qquad 528/931 \approx 57\%, \qquad 51/931 \approx 5\% :$$

the $32 \to 16$ block dominates because its weight *product* $n_{\text{in}} n_{\text{out}}$ is largest, while all biases together ($32 + 16 + 3 = 51$) are almost a rounding error.

**Take-away.** Biases add only $n_{\text{out}}$ each but are easy to forget. Most parameters live in the widest weight matrices, which is why hidden-layer width drives model size — and why parameter counting is a quick sanity check before training.

## Common mistake

Attaching parameters to the 10 input units, or dropping the $+n_{\text{out}}$ bias term — both corrupt the total.

# Outline

# Why convolutions?

Image data have local, translation-invariant structure.

## Takeaway

A cat remains a cat shifted by 10 pixels. Fully-connected networks ignore this and need much more data.

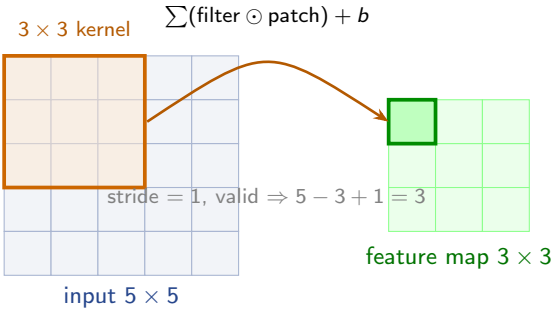
## Two key layer types

- **Convolutional layers**: each unit looks at a small patch of the input; weights are shared across positions.
- **Pooling layers**: down-sample (max or average) for translation tolerance.

## In industry — Healthcare: radiology triage

Many AI-enabled imaging devices hold regulatory clearance, mostly in radiology. They *prioritise or flag* studies rather than decide — a radiologist still signs the report.

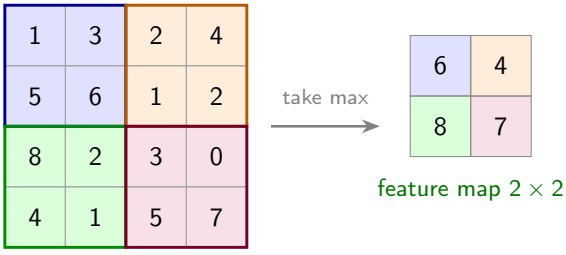
# The convolution operation




**Takeaway**

Each output cell is a dot product of the shared-weight filter with a local input patch. Weight sharing plus locality make convolutional neural networks (CNNs) data-efficient for images.

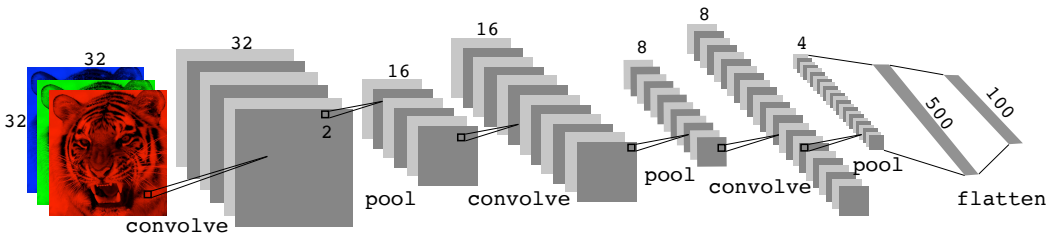
# Max-pooling: downsampling by local maxima




**Takeaway**

Each outlined $2 \times 2$ window is replaced by its maximum, shrinking $H, W$ by the stride while keeping the depth. This buys small-shift tolerance and cuts computation — and pooling has *no* trainable parameters.

# A simple CNN architecture




**Takeaway**

The pattern repeats: convolutions extract local features, pooling shrinks the maps, and a final dense layer classifies. Early layers learn edges; deeper layers assemble them into whole objects.

*Source: Figure 10.8 — ISLP, James et al. (2023).*

# Modern CNN tricks

- **Batch normalisation**: re-centre and scale each layer's inputs per mini-batch — stabilises and speeds up training.
- **Dropout**: randomly switch off units so the net cannot rely on any single one (detailed in the Fitting section).
- **Data augmentation**: random crops, flips, colour jitter — free extra "images" that teach invariance.
- **Skip / residual connections** (ResNet): let gradients bypass layers, enabling very deep networks.
- **Transfer learning**: reuse features pretrained on ImageNet, then fine-tune on your (smaller) task.

**Takeaway**

These tricks turned CNNs from finicky to reliable. Most are cheap to add and mainly improve *generalisation*.

# Exercise 10.4 — CNN layer arithmetic [Math]

## Exercise

A colour image of size $32 \times 32 \times 3$ is fed to a convolutional layer with 16 filters, each $5 \times 5$, stride 1, no padding ("valid"). Recall the spatial-size formula

$$O = \left\lfloor \frac{W - F + 2P}{S} \right\rfloor + 1.$$

1. Compute the output volume (height, width, depth) of the conv layer.
2. Compute the number of trainable parameters in the conv layer (include biases).
3. A $2 \times 2$ max-pooling layer (stride 2) follows. What is its output volume and how many parameters does it add?
4. Redo part (1) if instead padding $P = 2$ ("same") is used.

# Solution — Exercise 10.4 (1/2)

## Solution

**Part 1 — conv output.** With $W = 32$, $F = 5$, $P = 0$, $S = 1$:

$$O = \frac{32 - 5 + 0}{1} + 1 = 28.$$

Each filter slides over the image and produces *one* $28 \times 28$ feature map, so depth = number of filters: the output volume is $28 \times 28 \times 16$.

**Part 2 — conv parameters.** Each filter spans the full input depth: $5 \times 5 \times 3 = 75$ weights, $+1$ bias $= 76$ per filter. With 16 filters:

$$16 \times 76 = \boxed{1216} \text{ parameters.}$$

*Why so few?* A dense layer from the $32 \times 32 \times 3 = 3072$ inputs to the same $28 \times 28 \times 16 = 12\,544$ outputs would need $3072 \times 12\,544 + 12\,544 \approx 38.5$ million parameters — weight sharing cuts that by a factor of about 30 000.

# Solution — Exercise 10.4 (2/2)

## Solution

**Part 3 — pooling.** Max-pooling with $F = 2$, $S = 2$:

$$O = \frac{28 - 2}{2} + 1 = 14,$$

giving $14 \times 14 \times 16$: pooling acts on each of the 16 channels separately, so depth is preserved. It has **no** trainable parameters — it only takes a maximum.

**Part 4 — with padding $P = 2$.**

$$O = \frac{32 - 5 + 2(2)}{1} + 1 = \frac{31}{1} + 1 = 32.$$

Output $32 \times 32 \times 16$: "same" padding preserves the spatial size. In general, stride-1 "same" padding takes $P = (F - 1)/2$ — here $(5 - 1)/2 = 2$ — so conv layers can be stacked without shrinking the maps.

*Common mistake: adding a $+2P$ term for the pooling layer (it uses $P = 0$) or forgetting to divide by the stride before adding 1.*

# Outline

# Recurrent neural networks (RNNs)

Process sequences $x_1, \ldots, x_T$ by carrying a hidden state forward:

$$h_t = g(W_x\, x_t + W_h\, h_{t-1} + b)\,.$$

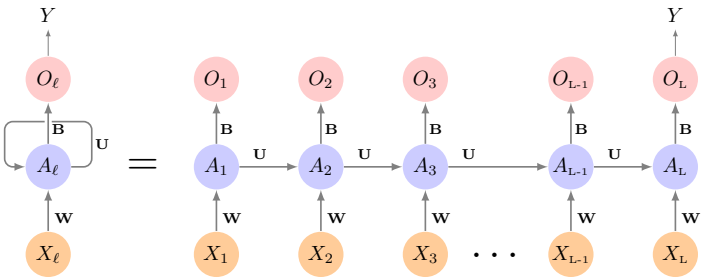
## Reading the recurrence

- $x_t$: input at step $t$; $h_t$: hidden state ("memory") at step $t$.
- $W_x$: weights on the new input; $W_h$: weights on the previous state $h_{t-1}$; $b$: bias.
- The *same* $W_x, W_h, b$ are reused at every step — weight sharing across time.

## Takeaway

Suitable for time series, text, audio. Plain RNNs suffer from vanishing/exploding gradients; LSTM and GRU add gating to carry long-range information.

# An RNN unrolled in time




**Takeaway**

Unrolling shows the recurrence as a deep feed-forward net that reuses the same weights at every step; training uses backpropagation "through time".

*Source: Figure 10.12 — ISLP, James et al. (2023).*

# Sequence models in commercial use

## In industry — Insurance and banking: document and claims processing

OCR plus sequence models read claims forms, invoices, contracts and KYC documents and extract the fields — claim amount, IBAN, policy number, counterparty. This is one of the highest-volume commercial uses of deep learning: it decides which files clear *straight through* and which are routed to a human handler. A silently mis-extracted amount becomes a wrong payment, so confidence thresholds and sampling audits are part of the design, not an afterthought.

## In industry — Telco and services: speech and language

Speech recognition turns call recordings into text for routing, summarising and compliance checks; large language models over retrieved document embeddings answer customer and internal support queries. The business case is handling time, not accuracy for its own sake.

# Outline

# Loss and optimisation

Choose a loss $L(\theta) = \sum_i \ell(y_i, \hat{y}_i; \theta)$ and minimise by stochastic gradient descent:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \ell_{\text{minibatch}}(\theta).$$

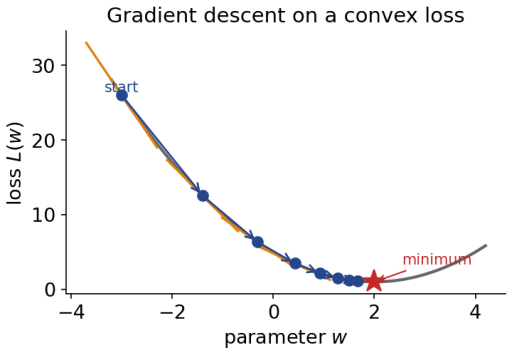
## Reading the update

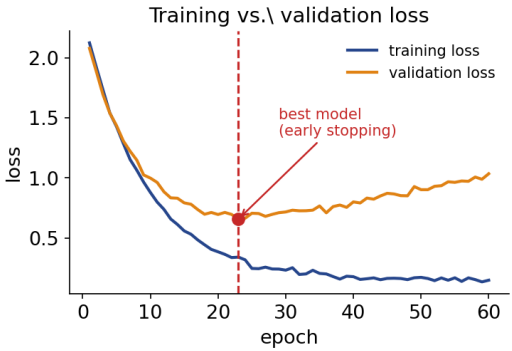
- $\theta$: all weights and biases; $\nabla_\theta \ell$: gradient (steepest-*ascent* direction) of the loss.
- Step *against* the gradient, scaled by the learning rate $\eta$ (too large diverges, too small crawls — see Exercise 10.5).
- "Stochastic": each step uses one random mini-batch (32/64/128) — a cheap, noisy estimate of the full-data gradient.
- Modern optimisers add memory: SGD+momentum, Adam, RMSprop.

## In industry — Operations: what training costs an organisation

Behind this one line sit GPU capacity, large labelled data sets and an MLOps stack: pipelines, versioning, monitoring, retraining. Projects stall on that plumbing far more often than on the optimiser.

# Training a network: descent and early stopping

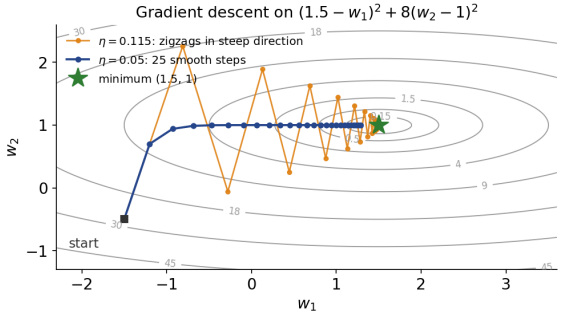



**Takeaway**

Gradient descent steps downhill along the loss surface (left). In practice, watch training *and* validation loss and stop at the validation minimum, before overfitting sets in (right).

# Gradient descent on a loss surface: the real path




## Takeaway

Each step is perpendicular to the local contour: with $\eta = 0.05$ the loss falls from 27 to 0.05 in 25 steps — fast down the steep $w_2$ axis, then a crawl along the flat valley — while the too-large $\eta = 0.115$ overshoots and zigzags in the steep direction.

# Regularisation in deep learning

Modern networks can have more parameters than data points but still generalise. Why?

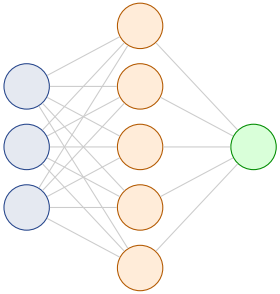
## How to read this

- Implicit regularisation by SGD.
- Dropout, weight decay ($\ell_2$), early stopping.
- Data augmentation.
- Architecture choice is itself a strong inductive bias.

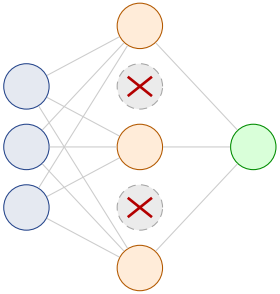
## In industry — Regulated deployment: generalising is not the only bar

Where a network is used to assess creditworthiness, to screen job applicants or for a medical purpose, the EU AI Act places it in the *high-risk* category, with documentation, logging, monitoring and human-oversight obligations. That raises the bar for an opaque model: a validation team will ask not only "does it generalise?" but "can you show why it said this, and who reviewed it?"

# Dropout: randomly switch off units during training



**Full network**



**During training: 2 units dropped**

**Takeaway**

On each mini-batch, every hidden unit is kept with probability $1 - \rho$ and otherwise *zeroed*, so the net cannot lean on any single unit and learns redundant, robust features. At test time all units are used (weights rescaled).

# Exercise 10.5 — One gradient-descent step [Math]

## Exercise

Consider minimising the loss $L(w) = (w - 3)^2$ by gradient descent, $w \leftarrow w - \eta\, L'(w)$, starting from $w = 2$.

1. Write $L'(w)$ and evaluate it at $w = 2$.
2. Perform one update with learning rate $\eta = 0.1$.
3. Show that the update rule for the *error* $e = w - 3$ is $e \leftarrow (1 - 2\eta)\, e$, and deduce the range of $\eta$ for which gradient descent converges.
4. What happens with $\eta = 1.5$? Interpret.

# Solution — Exercise 10.5 (1/2)

## Solution

**Part 1 — gradient.** $L'(w) = 2(w-3)$, so $L'(2) = 2(2-3) = -2$.

**Part 2 — one step ($\eta = 0.1$).**

$$w \leftarrow 2 - 0.1 \times (-2) = 2 + 0.2 = 2.2.$$

We moved from 2 toward the minimiser $w^\star = 3$: good.

**Part 3 — convergence range.** Substituting $L'(w) = 2(w-3)$ into the update,

$$w \leftarrow w - \eta\, 2(w-3) \implies (w-3) \leftarrow (w-3) - 2\eta(w-3) = (1-2\eta)(w-3).$$

So $e_{t+1} = (1-2\eta)\, e_t$, hence $e_t = (1-2\eta)^t\, e_0$: a geometric sequence, which dies out iff its ratio satisfies $|1-2\eta| < 1$, i.e.

$$0 < \eta < 1.$$

With $\eta = 0.1$ the ratio is 0.8 — each step removes 20% of the remaining error.

# Solution — Exercise 10.5 (2/2)

## Solution

**Part 4** — $\eta = 1.5$. Then $1 - 2\eta = -2$, so $|1 - 2\eta| = 2 > 1$: the error *doubles* each step and flips sign. Concretely $w \leftarrow 2 - 1.5(-2) = 5$; now $|5 - 3| = 2 > |2 - 3| = 1$, and iterating,

| $t$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $w_t$ | 2 | 5 | $-1$ | 11 |
| $e_t = w_t - 3$ | $-1$ | 2 | $-4$ | 8 |

The steps overshoot the minimum ever more wildly: gradient descent **diverges**. This is the classic "learning rate too large" failure — $\eta$ trades off speed against stability.

**Interpretation.** The extremes are instructive: $\eta = 0.5$ makes the ratio $1 - 2\eta = 0$ and lands on $w^\star = 3$ in *one* step (possible only because this loss is exactly quadratic), while a tiny $\eta$ converges but crawls.

## Common mistake

Concluding "smaller is always safer, so smaller is better" — at $\eta = 0.01$ the ratio is 0.98 and hundreds of steps are needed here.

# Outline

# Python lab — run it live in the notebook

## Companion notebook: `chapter_10_lab.ipynb`

The code in this section is meant to be **demonstrated live**. Open the companion Jupyter notebook and run it cell by cell:

- §1, *Data* — loading `Default`, the train/test split and standardisation;
- §2, *PyTorch model* — the `MLP` class, seeded with `torch.manual_seed(1)` so the AUC below is reproducible;
- §3, *Training loop* — mini-batches via `DataLoader`, then the test AUC.

Requires `torch`; §4, *Exercises* ends the notebook with hands-on tasks.

# An MLP in PyTorch

```
import torch
from torch import nn, optim

class MLP(nn.Module): # a model = a subclass of nn.Module
    def __init__(self, p, k=64, K=2): # p inputs, k hidden units, K classes
        super().__init__()
        self.net = nn.Sequential( # layers applied in order:
            nn.Linear(p, k), # affine map W x + b (p -> k)
            nn.ReLU(), # nonlinearity max(0, z)
            nn.Dropout(0.2), # randomly zero 20% of units (regularises)
            nn.Linear(k, k), nn.ReLU(), nn.Dropout(0.2), # second hidden layer
            nn.Linear(k, K)) # output layer: K logits (no softmax here)
    def forward(self, x): # forward pass: how inputs become outputs
        return self.net(x)

model = MLP(p=Xtr.shape[1]) # build the network for our p predictors
opt = optim.Adam(model.parameters(), lr=1e-3) # optimiser: Adam, learning rate 0.001
```

# Exercise 10.6 — Softmax and cross-entropy [Python]

## Exercise

A 3-class classifier outputs the logits $z = (2.0,\ 1.0,\ 0.1)$ for one observation whose true label is class 0.

1. By hand, compute the softmax probabilities $p_k = e^{z_k} / \sum_j e^{z_j}$.
2. Compute the cross-entropy loss $\ell = -\log p_y$ for the true class $y = 0$.
3. Write short NumPy code that reproduces both and prints them.

# Solution — Exercise 10.6 (1/3)

## Solution

**Method.** Softmax exponentiates each logit (making it positive) and divides by the sum, turning arbitrary scores into a probability vector; cross-entropy then scores only the probability given to the *true* class.

**Step 1 — exponentiate.**

$$e^{2.0} = 7.389, \quad e^{1.0} = 2.718, \quad e^{0.1} = 1.105, \qquad \sum = 11.212.$$

**Step 2 — softmax probabilities.**

$$p = \left(\tfrac{7.389}{11.212}, \tfrac{2.718}{11.212}, \tfrac{1.105}{11.212}\right) = (0.659,\ 0.242,\ 0.099).$$

*Check:* the three probabilities sum to 1.000.

**Step 3 — cross-entropy for $y = 0$.**

$$\ell = -\log p_0 = -\log(0.659) = 0.417.$$

*Common mistake: using* $\log_{10}$ *instead of the natural log (*$-\ln 0.659 = 0.417$*).*

# Solution — Exercise 10.6 (2/3)

## Solution

```python
import numpy as np # numerics library

z = np.array([2.0, 1.0, 0.1]) # logits from the network
p = np.exp(z) / np.exp(z).sum() # softmax: exponentiate, normalise
print("probs:", np.round(p, 3)) # [0.659 0.242 0.099]

y = 0 # true class label
loss = -np.log(p[y]) # cross-entropy = -log(true-class prob)
print("loss:", round(float(loss), 3)) # 0.417
```

# Solution — Exercise 10.6 (3/3)

## Solution — Interpretation

The model assigns the highest probability (0.659) to the correct class, so the loss is modest. A perfect prediction ($p_0 = 1$) gives loss 0; a confidently wrong prediction sends $-\log p_0 \to \infty$ — cross-entropy punishes confident errors hardest, exactly the gradient signal a classifier needs.

**Why the "log-sum-exp" trick works.** Softmax is shift-invariant: dividing top and bottom by $e^c$ gives $p_k = e^{z_k - c} / \sum_j e^{z_j - c}$, unchanged for any $c$. Choosing $c = \max_k z_k = 2.0$ here,

$$e^0 = 1, \quad e^{-1} = 0.368, \quad e^{-1.9} = 0.150, \qquad p = \tfrac{1}{1.518}\,(1,\ 0.368,\ 0.150) = (0.659,\ 0.242,\ 0.099),$$

the same probabilities — but every exponent is now $\leq 0$, so nothing can overflow no matter how large the logits get.

## Common mistake

Applying softmax yourself and then calling PyTorch's `nn.CrossEntropyLoss` — that loss already includes the softmax and expects raw logits; softmaxing twice silently distorts the loss.

# Extended Exercise 10.3 — Overfitting and regularisation [Python]

## Extended exercise (15 min)

Fit a **PyTorch** multilayer perceptron to the `Default` data, *watch it overfit*, then tame it. Encode `student` as 0/1, standardise `(balance, income, student)`, and predict `default`.

1. Train on a *small* subset ($n_{\text{train}} = 200$) with a wide MLP (two hidden layers of 128, no dropout) and no weight decay, recording the training and validation cross-entropy after every epoch.
2. Where does the validation loss turn upward, and what does the training loss do?
3. Refit with strong L2 regularisation (`weight_decay=0.1` in Adam); compare the validation loss and say what *early stopping* would choose.

## Run this live

The worked solution sits in `chapter_10_lab.ipynb` under *Lecture exercises — worked Python solutions*: the cell headed *Extended Exercise 10.3 — Overfitting and regularisation [Python]*.

# Solution — Extended Exercise 10.3 (1/3)

## Solution — code: load / standardise / convert to tensors

```
import pandas as pd, torch
from torch import nn, optim
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

df = pd.read_csv("Default.csv")
df["y"] = (df["default"] == "Yes").astype(int) # response -> 0/1
df["student"] = (df["student"] == "Yes").astype(int) # dummy-code student
X = df[["balance", "income", "student"]].astype(float).values
Xtr, Xte, ytr, yte = train_test_split(X, df["y"].values,
    train_size=200, random_state=0, stratify=df["y"]) # tiny train set!
sc = StandardScaler().fit(Xtr) # scale using TRAIN stats only
Xtr, Xte = sc.transform(Xtr), sc.transform(Xte)
# PyTorch works on tensors: features float32, labels int64
Xtr, Xte = map(lambda a: torch.tensor(a, dtype=torch.float32), (Xtr, Xte))
ytr, yte = map(lambda a: torch.tensor(a, dtype=torch.int64), (ytr, yte))
```

# Solution — Extended Exercise 10.3 (2/3)

## Solution — code: train and record the loss after every epoch

```
def curve(weight_decay, epochs=400):
    torch.manual_seed(1) # reproducible weights
    net = nn.Sequential(nn.Linear(3,128), nn.ReLU(), # wide MLP
        nn.Linear(128,128), nn.ReLU(), nn.Linear(128,2))
    loss_fn = nn.CrossEntropyLoss() # softmax + log-loss in one
    opt = optim.Adam(net.parameters(), lr=1e-2,
                     weight_decay=weight_decay) # L2 penalty lives here
    tr, va = [], []
    for _ in range(epochs): # 1 epoch = 1 full-batch step here
        opt.zero_grad() # reset gradients
        loss = loss_fn(net(Xtr), ytr) # forward pass on TRAIN
        loss.backward(); opt.step() # backprop + update
        with torch.no_grad(): # eval: no gradients
            tr.append(loss.item()); va.append(loss_fn(net(Xte), yte).item())
    return tr, va
tr0, va0 = curve(0.0) # unregularised -> memorises the 200 points
tr1, va1 = curve(0.1) # strong weight decay (L2)
```

# Solution — Extended Exercise 10.3 (3/3)

## Solution — typical result and interpretation

**Typical values** (exact numbers vary with seed and version):

`unreg: train` $\to$ `0.00  val-min` $\approx$ `0.10 (epoch 16)  val-final` $\approx$ `0.93`

`decay: train` $\approx$ `0.19  val` *`lower and flat`* $\approx$ `0.18`

- *Unregularised:* training loss dives toward 0 (the net **memorises** the 200 points), but validation loss bottoms out within the first few dozen epochs and then **climbs** almost tenfold — textbook overfitting.
- *Weight decay:* training loss stays higher, yet validation loss is *lower and flat*: the L2 penalty trades a little fit for much better generalisation.
- *Early stopping* halts at the validation minimum — a free regulariser needing no extra penalty term.

*Same recipe as the companion notebook* (`chapter_10_lab.ipynb`); there the MLP adds dropout and mini-batches via `DataLoader`.

*Common mistake:* judging the two nets by training loss — the unregularised net "wins" at 0.00 precisely because it memorises. Only the validation curve ranks them honestly.

# Outline

# Chapter 10 in one slide

## What we accomplished

Surveyed the modern deep-learning landscape and connected it to the classical methods of the earlier chapters.

- Single- and multilayer perceptrons.
- Convolutional networks for images.
- Recurrent networks and transformers.
- Optimisation: SGD, momentum, Adam, backpropagation.
- Regularisation: dropout, weight decay, early stopping.

# Key formulas at a glance

**Neuron** $a = g\big(\sum_j w_j x_j + b\big)$, activation $g$.

**Single hidden layer** $f(x) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\big(w_{k0} + \sum_j w_{kj} x_j\big)$.

**Activations** ReLU $g(z) = \max(0, z)$; sigmoid $\sigma(z) = \dfrac{1}{1+e^{-z}}$; softmax $\dfrac{e^{z_m}}{\sum_\ell e^{z_\ell}}$.

**Loss** regression $\sum_i (y_i - f(x_i))^2$; classification $-\sum_i \sum_k y_{ik} \log f_k(x_i)$.

**Gradient descent** $\theta \leftarrow \theta - \eta \, \nabla_\theta L(\theta)$ (learning rate $\eta$).

**Dense params** $\#\text{params} = n_{\text{in}} \, n_{\text{out}} + n_{\text{out}}$ (weight matrix + one bias per unit).

**Conv output** size $= \lfloor (W - F + 2P)/S \rfloor + 1$ (input $W$, filter $F$, pad $P$, stride $S$).

**Conv params** $\#\text{params} = (F^2 d_{\text{in}} + 1) \times (\#\text{filters})$; pooling has none.

# Vocabulary checklist

| Term | Meaning |
|---|---|
| Activation function | Element-wise nonlinearity (ReLU, tanh, sigmoid) |
| Hidden unit | One node in a hidden layer |
| Backpropagation | Reverse-mode automatic differentiation |
| SGD / momentum | Stochastic gradient + velocity term |
| Adam | Adaptive per-parameter learning rates |
| Dropout | Random unit muting during training |
| Weight decay | $\ell_2$ penalty on weights |
| CNN | Conv + pool + dense layers for images |
| RNN / LSTM | Sequential models with hidden state |
| Transformer | Self-attention based sequence model |
| Double descent | Test-error second dip |

# Decision rules of thumb

- Tabular data ($n < 10^5$, $p < 10^3$): try gradient boosting first.
- Images: pretrained CNN (ResNet, EfficientNet) and fine-tune.
- Text: pretrained transformer; fine-tune only the head when $n$ is small.
- Always normalise inputs and labels.
- Track training and validation loss together.
- Save the best-validation checkpoint.

# Common pitfalls

## Don't

1. Compare a deep network to a poorly tuned baseline.
2. Forget to shuffle training data before each epoch.
3. Use too high a learning rate (loss explodes).
4. Trust accuracy without checking calibration.
5. Skip data augmentation for image / audio tasks.
6. Re-use the validation set for many tuning rounds.

# What's next

Chapter 13: *Multiple Testing*.

- The multiple-comparisons problem and the family-wise error rate.
- Bonferroni and Holm corrections.
- The false discovery rate and Benjamini-Hochberg.

# Appendix — what is in here, and why

Nothing in the appendix is needed to follow the chapter: each slide is a formal derivation, a longer worked exercise, or a side topic. Read it when you want the full story, or when a later chapter sends you back here.

## Contents of the appendix

- **CNN architecture arithmetic (Extended Exercise 10.2)** — shapes and parameter counts through a whole network. Exercise 10.4 does the same arithmetic for a single layer.
- **Transformers** — self-attention in one slide. Outside the scope of ISLP Chapter 10; here so that the word is not a mystery.
- **Backpropagation** — the chain rule that computes the gradients. Every framework does this for you; the main deck keeps what you must know — that training is gradient descent on a loss.
- **Double descent** — why very over-parameterised models can improve again past the interpolation point. A genuinely advanced topic.

# Extended Exercise 10.2 — CNN architecture arithmetic [Math]

## Extended exercise (15 min)

A $64 \times 64 \times 3$ RGB image passes through the stack below (convolutions use the padding shown; pooling is max-pooling, no padding). Recall $O = \lfloor (W - F + 2P)/S \rfloor + 1$.

| Layer | Kernel | Stride / Pad | Filters |
|---|---|---|---|
| Conv1 | $3 \times 3$ | $1 / 1$ | 32 |
| Pool1 | $2 \times 2$ | $2 / 0$ | — |
| Conv2 | $3 \times 3$ | $1 / 1$ | 64 |
| Pool2 | $2 \times 2$ | $2 / 0$ | — |
| Dense | softmax | — | 10 |

1. Give the output volume ($H \times W \times$ depth) after each layer.
2. Count the learnable parameters of every layer, and the total.
3. What input region does one unit of the final map, after Pool2, see (receptive field)?

# Solution — Extended Exercise 10.2 (1/3)

## Solution — spatial dimensions

Apply $O = (W - F + 2P)/S + 1$ to $H$ and $W$; depth = number of filters (pooling keeps depth).

$$\begin{aligned}
\text{Conv1} &: (64 - 3 + 2)/1 + 1 = 64 \quad \Rightarrow \; 64 \times 64 \times 32, \\
\text{Pool1} &: (64 - 2)/2 + 1 = 32 \quad \Rightarrow \; 32 \times 32 \times 32, \\
\text{Conv2} &: (32 - 3 + 2)/1 + 1 = 32 \quad \Rightarrow \; 32 \times 32 \times 64, \\
\text{Pool2} &: (32 - 2)/2 + 1 = 16 \quad \Rightarrow \; 16 \times 16 \times 64.
\end{aligned}$$

Flattening the last volume gives $16 \times 16 \times 64 = 16384$ features into the dense layer. "Same" padding preserves $H, W$; each pool halves them.

## Common mistake

Flattening only $16 \times 16 = 256$ features — the depth counts too: $16 \times 16 \times 64 = 16\,384$.

# Solution — Extended Exercise 10.2 (2/3)

## Solution — learnable parameters

A conv filter spans the *full input depth*, so $\# = (F^2 d_{\text{in}} + 1) \times (\#\text{filters})$; pooling has none.

$$\text{Conv1} : (3^2 \cdot 3 + 1) \times 32 = 28 \times 32 = 896,$$

$$\text{Conv2} : (3^2 \cdot 32 + 1) \times 64 = 289 \times 64 = 18\,496,$$

$$\text{Dense} : 16384 \times 10 + 10 = 163\,850,$$

$$\text{Pool1, Pool2} : 0.$$

$$\text{Total} = 896 + 18\,496 + 163\,850 = \boxed{183\,242} \text{ parameters.}$$

# Solution — Extended Exercise 10.2 (3/3)

## Solution — receptive field and interpretation

**Method.** Track the receptive field $r$ and the jump $j$ (input pixels between neighbouring units): a layer with filter $F$ and stride $S$ updates $r \leftarrow r + (F-1)\,j$, then $j \leftarrow j\,S$. Start at $r = 1$, $j = 1$:

$$\begin{aligned}
\text{Conv1 } (F{=}3,\ S{=}1): &\quad r = 1 + 2 \times 1 = 3, &\quad j = 1,\\
\text{Pool1 } (F{=}2,\ S{=}2): &\quad r = 3 + 1 \times 1 = 4, &\quad j = 2,\\
\text{Conv2 } (F{=}3,\ S{=}1): &\quad r = 4 + 2 \times 2 = 8, &\quad j = 2,\\
\text{Pool2 } (F{=}2,\ S{=}2): &\quad r = 8 + 1 \times 2 = 10, &\quad j = 4.
\end{aligned}$$

One unit of the final map therefore sees a $10 \times 10$ patch of the original image. **Take-aways.** (i) Convolutions are parameter-thrifty: the two conv layers use $<$ 20k weights through *weight sharing*, while the single dense layer holds 164k — about 89% of the model. (ii) Depth plus pooling grow the receptive field, letting deep units assemble local edges into global shapes.

# Transformers (briefly)

**Takeaway**

"Attention is all you need" (Vaswani et al., 2017). Self-attention computes weighted combinations of all positions in a sequence. Beats RNNs on most NLP tasks; foundation of modern LLMs.

**Self-attention in one line**

Each position forms a *query* and compares it against every position's *key*; the resulting weights average the *values*. So every output token can look directly at every input token — no matter how far apart, and all positions are processed in parallel (unlike an RNN's step-by-step loop).

# Backpropagation in one slide

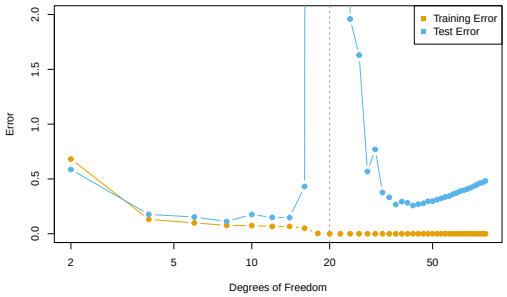
## The idea in two passes

- **Forward pass**: push $x$ through the layers, caching each activation.
- **Backward pass**: send an "error signal" $\delta$ from the loss back to every weight, multiplying local derivatives (the chain rule).
- Each weight's gradient $=$ (error signal at its output) $\times$ (its input); shared factors are reused, so the cost is $\approx$ one forward pass.

## Takeaway

Backprop is just the chain rule applied from output back to input — reverse-mode automatic differentiation. Implemented in PyTorch, JAX, TensorFlow, so you rarely write it by hand.

# Double descent




## Why the second dip

Test error first traces the classical U-shape, then peaks at the *interpolation threshold* (the model *just* fits the data and is unstable), and falls again as extra parameters let SGD pick a smoother interpolant.

*Source: Figure 10.20 — ISLP, James et al. (2023).*