AI Research Labs

Complete course · 44 lessons

Deep learning, *from the neuron up.*

From a single neuron to a trained network. Why one neuron cannot solve XOR, why depth and non-linearity matter, how backpropagation computes every gradient, and how to make a network generalise. Each lesson builds the intuition, then the maths, then a calculation you can do by hand.

1 Foundations of Deep Learning
2 The Neuron and Linear Separability
3 Multi-layer Perceptrons
4 Activation Functions
5 The Universal Approximation Theorem
6 Loss Functions and Output Layers
7 Backpropagation
8 Generalisation and Data
9 Regularisation
10 Convolutional Neural Networks
11 Network Design and PyTorch



Free course with quizzes, progress tracking and Scoopy, an AI tutor:
airesearchlabs.si/learn/dl

Deep Learning Foundations · Edition November 2026 · © 2026 AI Research Labs. Free for personal study.

Contents

Section 1 · Foundations of Deep Learning

1.1 Machine Learning and Deep Learning
1.2 Types of Learning
1.3 Gradient Descent and Momentum
1.4 Notation, Probability and Jargon

Section 2 · The Neuron and Linear Separability

2.1 The Artificial Neuron
2.2 Decision Boundaries and Linear Separability
2.3 The XOR Problem

Section 3 · Multi-layer Perceptrons

3.1 Layers, Matrices and Shapes
3.2 Solving XOR with a Hidden Layer
3.3 Why Depth Helps: Composing Regions
3.4 Why Non-linearity Is Essential

Section 4 · Activation Functions

4.1 Sigmoid, tanh, ReLU and Friends
4.2 Vanishing Gradients
4.3 Dying ReLU, Leaky ReLU and ELU
4.4 Choosing Activation Functions

Section 5 · The Universal Approximation Theorem

5.1 The Universal Approximation Theorem
5.2 Building Functions from Bumps
5.3 What the Theorem Does Not Promise

Section 6 · Loss Functions and Output Layers

6.1 Training as Minimising a Loss
6.2 MSE, MAE and Asymmetric Losses
6.3 Softmax and Cross-entropy
6.4 Binary, Multi-class and Logits in PyTorch

Section 7 · Backpropagation

7.1 The Chain Rule and Computational Graphs
7.2 Backpropagation for a Single Sigmoid Neuron
7.3 One Hidden Layer: Deriving Every Gradient
7.4 Backpropagation in Matrix Form
7.5 Multiple Paths and Deeper Networks
7.6 Mini-batches and Gradient Checking

Section 8 · Generalisation and Data

8.1 Fitting Is Not Learning
8.2 Training, Validation and Test Sets
8.3 Underfitting, Overfitting and Learning Curves
8.4 Preparing the Data
8.5 Hyperparameters and a Training Recipe

Section 9 · Regularisation

9.1 L2 and L1 Penalties
9.2 Early Stopping
9.3 Dropout
9.4 Data Augmentation

Section 10 · Convolutional Neural Networks

10.1 From Fully Connected to Convolutional
10.2 The Convolution Operation
10.3 Pooling
10.4 CNN Components and Feature Hierarchies

Section 11 · Network Design and PyTorch

11.1 Depth versus Width

11.2 Designing a Network for Tabular Data

11.3 Neural Networks in PyTorch

Section 1

Foundations of Deep Learning

What deep learning is, the types of learning, the notation, and how gradient descent minimises an objective.

1.1 Machine Learning and Deep Learning · 6 min

1.2 Types of Learning · 6 min

1.3 Gradient Descent and Momentum · 9 min

1.4 Notation, Probability and Jargon · 7 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

1

BUILDING AND TRAINING A NETWORK · STEP 1

Frame the task

What should the network predict, and from what?

Lessons here: 1.1, 1.2 · feeds into Choose the output and loss

7

BUILDING AND TRAINING A NETWORK · STEP 7

Update the weights

How do I turn gradients into a better network?

Lessons here: 1.3 · feeds into Evaluate honestly

◆

UNDERNEATH THE MAP · FOUNDATION

Notation, probability and jargon

Lessons here: 1.4 · feeds into Compute the gradients

Machine Learning and Deep Learning

Machine learning builds a model and learns its parameters from data. Deep learning is the part of machine learning that uses many-layered neural networks. What that means, the learning loop, and a short history.

A model, and learning its parameters

Machine learning (ML) is model building. A **model** is a compact mathematical description of how inputs relate to outputs. **Learning** is the procedure that finds good values for the model's **parameters** from examples, instead of a person writing every rule by hand.

$$\hat{y} = mx + c$$

Here x is the input, $\hat{y}$ the prediction, and m and c are the parameters. If learning finds $m = 5$ and $c = 30$, the model predicts $\hat{y} = 5 \cdot 6 + 30 = 60$ for $x = 6$. The *form* of the model (a straight line) is chosen in advance; the data decide its parameters.

The learning loop

Building a model is a loop, not a single step:

Data → Model → Learning → Evaluation → back to the start until performance is acceptable.

Evaluation is part of the loop because a model that fits its training examples can still fail on new ones (Section 8).

Where deep learning fits

Deep learning (DL) is the part of ML that uses **neural networks with many layers**. Each layer transforms its input into a new representation, and later layers build on earlier ones, so the network learns useful features itself instead of relying on hand-made ones.

KEY DISTINCTION

ML is the broad field. DL is ML with multi-layered neural networks. Not every ML model is a neural network (decision trees and linear regression aren't), and not every neural network is deep.

APPLICATION AREA	EXAMPLES
Prediction	Weather, demand, prices
Pattern recognition	Recognising and segmenting objects in images
Sequential decision-making	Games, robotics, control
Generative AI	Generating text and images

A short history

WHEN	MILESTONE	WHY IT MATTERS
1943	McCulloch and Pitts model a neuron as a logic circuit	The first mathematical abstraction of a neuron
1959	ADALINE (Widrow and Hoff)	An early real application: cancelling echoes on phone lines
1986	Backpropagation popularised	Made training multi-layer networks practical
2000s	GPUs	Enough computation for much larger networks
2010	ReLU popularised	Eased training of deep networks (Section 4)
2012	A deep CNN wins ImageNet by a wide margin	Convolutional networks take over computer vision (Section 10)

Common doubts

FIRST PASS **Is a model the same thing as the learning algorithm?**

No. The **model** is the mathematical mapping, such as $\hat{y} = mx + c$ or a neural network with its layers. The **learning algorithm** (for example gradient descent) is the procedure that adjusts the model's parameters to fit the data. The same model can be trained by different algorithms.

GOING DEEPER **If deep learning is just machine learning, why does it get its own name?**

Because of what the layers do. Classic ML often needs people to design the input features. A deep network learns a **sequence of representations** itself: early layers find simple patterns and later layers combine them. That ability to learn features is what made DL so effective on images, audio and text.

Cheatsheet

ML vs DL

ML: any model whose parameters are learned from data. DL: ML with multi-layered neural networks that learn their own representations.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/foundations/machine-learning-and-deep-learning

Types of Learning

Supervised, unsupervised, semi-supervised and reinforcement learning: what training information each uses, what it aims for, and the notation for training examples.

Four ways to learn

TYPE	TRAINING INFORMATION	GOAL AND EXAMPLES
Supervised	Labelled pairs $(x^{(i)}, y^{(i)})$	Learn the mapping $x \to y$. **Classification** predicts a category; **regression** predicts a number.
Unsupervised	Inputs only, no labels	Find structure: clustering, anomaly detection, density estimation.
Semi-supervised	A few labelled examples, many unlabelled	Use the unlabelled data to help: name one face in a photo cluster and the rest follow.
Reinforcement	An agent acts and receives rewards or penalties	Learn a policy that maximises reward: games, robotics, control.

Notation for training data

A supervised training set has N examples, written $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$. The superscript in brackets, (i), is an **index** that picks out example i, not a power. $x^{(i)}$ is usually a vector of features and $y^{(i)}$ its label.

Classification or regression?

Look at the target, not the inputs:

- "Will this customer leave? (yes/no)" and "which digit is this?" are **classification**: a finite set of labels.
- "What grade will a student get from their hours of study?" is **regression**: a number.

The type of target decides the output layer and the loss of a network (Section 6).

> SCOPE
>
> This course covers supervised learning with neural networks. Reinforcement learning needs extra machinery (Markov decision processes) and is outside its scope; know the high-level distinction.

Common doubts

FIRST PASS **Does $x^{(2)}$ mean x squared?**

No. In ML notation $x^{(2)}$ means "the second training example". A power would be written without brackets, x^2. Mixing the two up is a common source of mistakes in derivations, so it's worth writing a small legend of what each index means.

Cheatsheet

Learning types

Supervised: labelled (x, y) → classification (category) or regression (number). Unsupervised: no labels → structure. Semi-supervised: few labels. Reinforcement: rewards → policy.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/foundations/types-of-learning

Gradient Descent and Momentum

Training means minimising an objective. Gradient descent repeatedly steps against the gradient. A worked example, what the learning rate does, why a zero gradient is not proof of a minimum, and how momentum helps.

Training is optimisation

Learning means choosing parameters that make an objective (a **cost** or **loss**) as small as possible:

$$\min_{x \in \mathbb{R}^d} f(x)$$

Here x is the vector of parameters and f the objective. At an interior minimum of a differentiable function, the gradient is zero: $\nabla f(\hat{x}) = 0$. That condition is **necessary but not sufficient**: the gradient is also zero at a maximum, at a **saddle point** and on a flat plateau.

The gradient descent update

The gradient ∇f points in the direction of steepest local *increase*, so step the other way:

$$x_{k+1} = x_k - \eta \, \nabla f(x_k)$$

η (eta) is the **learning rate**, a hyperparameter you choose.

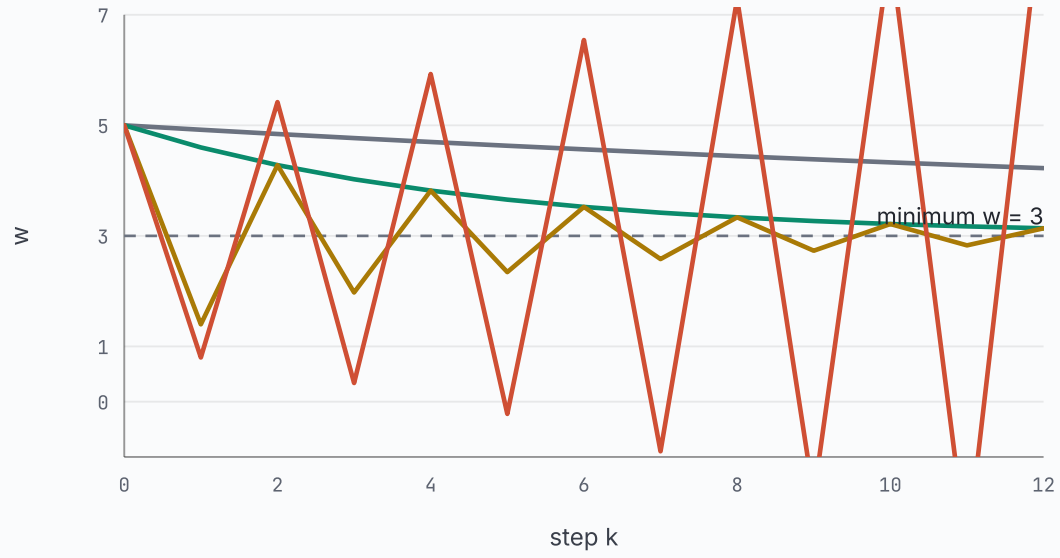
WORKED EXAMPLE

Minimise $f(w) = (w-3)^2$, whose minimum is at $w = 3$. The derivative is $f'(w) = 2(w-3)$. Start at $w_0 = 5$ with $\eta = 0.1$:

- $w_1 = 5 - 0.1 \cdot 2(5-3) = 5 - 0.4 = 4.6$
- $w_2 = 4.6 - 0.1 \cdot 2(1.6) = 4.6 - 0.32 = 4.28$
- $w_3 = 4.28 - 0.1 \cdot 2(1.28) = 4.024$

Each step multiplies the distance to the minimum by $1 - 2\eta = 0.8$: $w_k - 3 = 2 \cdot 0.8^k$.

What the learning rate does



Gradient descent on $f(w) = (w - 3)^2$ from $w = 5$. The distance to the minimum is multiplied by $(1 - 2\eta)$ each step: shrinking for $0 < \eta < 1$, oscillating when the factor is negative, and growing when $\eta > 1$.

- **Too small:** progress is very slow.
- **Too large:** updates overshoot, oscillate or diverge.
- **Narrow valleys** (steep one way, shallow the other) make plain gradient descent zig-zag.
- **Local minima, saddle points and plateaus** can stall it.

Momentum

Momentum adds a fraction α of the previous step:

$$x_{k+1} = x_k - \eta \nabla f(x_k) + \alpha \Delta x_k, \qquad \Delta x_k = x_k - x_{k-1}$$

Steps in a consistent direction build up speed, while zig-zag components cancel out. This helps on badly conditioned objectives. **RMSprop** and **Adam** go further by adapting the step size for each parameter; Adam is a common default in practice.

Common doubts

FIRST PASS **Why do we subtract the gradient instead of adding it?**

The gradient points uphill: it's the direction in which f increases fastest. We want f to **decrease**, so we move the opposite way. In the example, at $w = 5$ the derivative is $+4$ (the function rises to the right), so we move left, towards 3.

GOING DEEPER **If the gradient is zero, have we found the best parameters?**

Not necessarily. A zero gradient only says the surface is locally flat. It could be a minimum, a maximum, a **saddle point** (up in one direction, down in another) or a plateau. In high-dimensional neural network losses, saddle points and plateaus are common, which is one reason momentum and Adam help.

FIRST PASS **What exactly is an objective function?**

It is the single number that training tries to make as small as possible, written as a function of the parameters. The data is held fixed; only the parameters change. In deep learning it is usually the average loss over the training examples, sometimes plus a penalty on the weights.

In the worked example, $f(w) = (w - 3)^2$ is the objective and w is the parameter. At $w = 5$ its value is $f(5) = 4$; at $w = 3$ it is 0, the best possible.

FIRST PASS **What is a gradient?**

With one parameter, it is the derivative: the slope of the objective at the current point, meaning how much f changes for a small change in w. At $w = 5$, $f'(5) = 2(5 - 3) = 4$, so nudging w up by 0.01 raises f by about 0.04.

With many parameters, the gradient ∇f is the vector of partial derivatives, one per parameter. Together they point in the direction of steepest increase, and the length of the vector says how steep the surface is there. A network with a million weights has a gradient with a million entries.

FIRST PASS **What is the difference between the objective's value and its gradient?**

The **value** says how bad the parameters are right now: the height of the surface. The **gradient** says which way is downhill and how steep it is: the slope. For $f(w) = (w - 3)^2$:

- $w = 4$: value $f(4) = 1$, gradient $f'(4) = 2$.
- $w = 1$: value $f(1) = 4$, gradient $f'(1) = -4$.
- $w = 3$: value 0, gradient 0.

$w = 1$ and $w = 5$ have the same value, 4, but opposite gradients, because they sit on opposite sides of the minimum. The update uses only the gradient; the value is what you watch to see whether training is working.

FIRST PASS **What do a positive and a negative gradient tell me?**

The sign says which way the objective rises. **Positive**: f goes up as w goes up, so decrease w. **Negative**: f goes down as w goes up, so increase w. Subtracting $\eta \nabla f$ does the right thing automatically.

At $w = 1$: $f'(1) = -4$, so $w_1 = 1 - 0.1 \times (-4) = 1.4$, a step to the right, towards 3. The size of the gradient sets the size of the step, which is why steps shrink as you approach the minimum.

GOING DEEPER **What is the difference between a minimum, a maximum, a saddle point and a plateau?**

At all four the gradient is zero, or nearly zero, so the gradient alone can't tell them apart. The shape around the point can:

- **Minimum**: every direction goes up. $f(w) = (w-3)^2$ at $w = 3$.
- **Maximum**: every direction goes down. $-(w-3)^2$ at $w = 3$. Gradient descent moves away from it at the slightest push.
- **Saddle point**: up in some directions, down in others, like a horse's saddle. $f(x, y) = x^2 - y^2$ at $(0, 0)$. Common in high-dimensional network losses.
- **Plateau**: a wide, almost flat region. The gradient is tiny everywhere, so steps are tiny and training looks stuck, for example when sigmoid units saturate.

The curvature (second derivative) separates them: positive in every direction means a minimum, negative in every direction a maximum, mixed signs a saddle.

GOING DEEPER **Why does $eta = 1$ make the example oscillate?**

Each update multiplies the distance to the minimum by $1 - 2\eta$: $w_{k+1} - 3 = (1 - 2\eta)(w_k - 3)$. With $\eta = 1$ the factor is -1: the distance stays the same and only the side flips.

From $w_0 = 5$: $w_1 = 5 - 1 \times 4 = 1$, then $w_2 = 1 - 1 \times (-4) = 5$, and so on. It bounces between 5 and 1 forever and never converges.

- $0 < \eta < 0.5$: approaches 3 steadily from one side.
- $\eta = 0.5$: lands on 3 in one step.
- $0.5 < \eta < 1$: overshoots each time but closes in (the $\eta = 0.9$ curve in the chart).
- $\eta > 1$: overshoots further each time and diverges.

The 2 is the curvature, $f''(w) = 2$. In general a step size above $2/\text{curvature}$ diverges.

GOING DEEPER **How do GD, SGD, momentum, RMSprop and Adam differ?**

All of them use the same gradient from backpropagation. They differ only in how they turn it into a step.

- **Gradient descent** (batch): the gradient over the whole training set each step. Exact, but every step is expensive.
- **SGD** (stochastic or mini-batch): the gradient on one example or a small batch. A noisy estimate, but many cheap steps, and the noise can help escape saddle points.
- **Momentum**: adds a fraction of the previous step. Speeds up along a consistent direction and damps zig-zag.
- **RMSprop**: divides each parameter's step by a running average of its recent squared gradients. Parameters with large gradients take smaller steps; flat directions take larger ones.
- **Adam**: momentum and RMSprop together, with a correction for the first few steps. The usual default, often with $\eta = 10^{-3}$.

FIRST PASS

What exactly is a point on the objective's graph?

One choice of parameters and how good it is. The horizontal position is the parameter value, w; the height is the objective at that value, $f(w)$. The point $(5, 4)$ on $f(w) = (w - 3)^2$ means "with $w = 5$, the objective is 4".

Training starts at one point and moves along the curve, one update at a time, towards the lowest point. With two parameters the graph is a surface and a point is a spot on it; with a million parameters you can't draw it, but the idea is the same.

FIRST PASS

How does momentum actually help?

Plain gradient descent only looks at the slope where it stands now. In a long narrow valley the slope points mostly across the valley, so it zig-zags from wall to wall and creeps along the floor.

Momentum keeps a running memory of past steps. The side-to-side parts point in opposite directions on alternate steps and cancel out; the part along the valley points the same way every time and builds up. The result: fewer wasted steps, faster progress along gentle slopes, and enough speed to roll through small bumps and flat stretches.

FIRST PASS

How do I know when gradient descent has converged, and when to stop?

Plot the objective against the iteration number: the **convergence curve**. A healthy curve falls steeply at first, then flattens. It has converged when it stops falling: the change per step, or the size of the gradient, drops below a small tolerance such as 10^{-6}. A curve that zig-zags or climbs means the learning rate is too large.

For neural networks, don't wait for the training loss to flatten. Watch the **validation** loss and stop when it stops improving: that is **early stopping** (lesson 9.2). In practice you also cap the number of epochs.

Cheatsheet

Gradient descent

$x_{k+1} = x_k - \eta \nabla f(x_k)$. Momentum: $+ \alpha \Delta x_k$ with $\Delta x_k = x_k - x_{k-1}$.

Learning rate

Too small: slow. Too large: oscillates or diverges. Zero gradient ≠ minimum (could be a maximum, saddle or plateau).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/foundations/gradient-descent-and-momentum

Notation, Probability and Jargon

The maths toolkit: vectors, matrices and index notation, expectation and variance with a worked example, and the ML vocabulary you will meet throughout the course.

Notation

SYMBOL	MEANING
x, w	Lower case: vectors (column vectors by default)
W, X	Upper case: matrices
$w^\top$	Transpose: the column vector w written as a row
w_i	Subscript: component i of w
$x^{(i)}$	Training example i (an index, not a power)
$\partial J / \partial w$	Partial derivative of the objective J with respect to w
∇J	The gradient: all partial derivatives of J collected in a vector

The four tools you need are **linear algebra** (layers are matrix products), **calculus** (especially the chain rule, for backpropagation), **optimisation** (gradient descent) and **probability** (outputs as probabilities, losses as likelihoods).

Expectation and variance

A discrete random variable has a probability mass function whose probabilities sum to 1; a continuous one has a density, and probabilities are areas under it.

$$\mathbb{E}[X] = \sum_i x_i \, p(x_i), \qquad \mathrm{Var}(X) = \mathbb{E}\big[(X-\mu)^2\big], \; \mu = \mathbb{E}[X]$$

WORKED EXAMPLE

X is 0 with probability 0.75 and 10 with probability 0.25.

- $\mathbb{E}[X] = 0(0.75) + 10(0.25) = 2.5$
- $\mathrm{Var}(X) = 0.75(0-2.5)^2 + 0.25(10-2.5)^2 = 4.6875 + 14.0625 = 18.75$
- Standard deviation $= \sqrt{18.75} \approx 4.33$, in the same units as X.

Vocabulary

TERM	MEANING
Parametric model	Described by a fixed set of parameters, such as $y = mx + c$ or a network's weights
Hyperparameter	A setting chosen outside gradient-based learning: learning rate, number of layers, width, dropout rate
Bias	Two meanings: the bias term b of a neuron, or systematic error (the bias–variance trade-off). A third, social bias, is about unfair outcomes for groups.
Overfitting	Fitting the training examples, noise included, so closely that new examples are predicted badly
Regularisation	Any change to learning meant to improve generalisation rather than training error (Section 9)

Common doubts

FIRST PASS **What is the difference between a parameter and a hyperparameter?**

Parameters (weights and biases) are learned by gradient descent. **Hyperparameters** are choices made around that process: the learning rate, how many layers and units, the batch size, the dropout rate. They are chosen using validation data (Section 8), not by gradient descent.

FIRST PASS **What is the difference between parametric and non-parametric models?**

A **parametric** model has a fixed shape described by a fixed number of parameters, however much data you have. Linear regression is always $y = \beta_0 + \beta_1 x_1 + \ldots$: more data improves the βs but doesn't add new ones. A neural network is parametric too: its size is fixed by the architecture.

A **non-parametric** model's complexity grows with the data. **K-nearest neighbours** keeps every training example and predicts from the closest ones, so the "model" is the dataset itself.

GOING DEEPER **Does non-parametric mean the model has no parameters?**

No. It means there is no **fixed, finite** set of parameters that defines the model. K-nearest neighbours still has settings such as K and the distance measure, but those are hyperparameters you choose, not values learned from data. What grows with the data is everything the model must store: all n training points.

Cheatsheet

Expectation and variance

$\mathbb{E}[X] = \sum x_i p(x_i)$; $\mathrm{Var}(X) = \mathbb{E}[(X-\mu)^2]$. Std dev $= \sqrt{\mathrm{Var}}$.

Index, not power

$x^{(i)}$ is training example i; x^2 is a square. Write a legend of indices before long derivations.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/foundations/notation-probability-and-jargon

Section 2

The Neuron and Linear Separability

What a single artificial neuron computes, what its decision boundary looks like, and why it cannot solve XOR.

2.1 The Artificial Neuron · 6 min

2.2 Decision Boundaries and Linear Separability · 7 min

2.3 The XOR Problem · 8 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

M1

BUILDING AND TRAINING A NETWORK · DENSE LANE · STEP M1

Understand the neuron

What can one neuron do, and what can it never do?

Lessons here: 2.1, 2.2, 2.3 · feeds into Stack layers

The Artificial Neuron

A neuron computes a weighted sum of its inputs, adds a bias and applies an activation function. What each part does, a worked example, and why the bias is just a weight on a constant input.

What a neuron computes

$$y = f\left(w^\top x + b\right) = f\left(\sum_i w_i x_i + b\right)$$

- x: the input vector (the features).
- w: the **weights**, one per input.
- b: the **bias**.
- $z = w^\top x + b$: the **pre-activation**, a single number.
- f: the **activation function**, which turns z into the output y.

So a neuron does two things: a weighted vote of its inputs, then a (usually non-linear) squashing of that vote.

What weights and bias mean

A **positive weight** means "more of this input pushes the output up"; a **negative weight** pushes it down; a weight near zero means the input barely matters. The **bias** shifts the threshold independently of the inputs: a large negative bias makes the neuron hard to switch on.

The bias can be treated as one more weight, $w_0 = b$, on a constant input $x_0 = 1$. Then $z = \sum_{i=0}^{n} w_i x_i$, which is why some books write the neuron without a separate bias.

WORKED EXAMPLE

$x = (2, -1)$, $w = (0.5, 1.5)$, $b = -0.2$.

$z = 0.5(2) + 1.5(-1) - 0.2 = 1 - 1.5 - 0.2 = -0.7$.

- Sigmoid: $y = \sigma(-0.7) = 1/(1 + e^{0.7}) \approx 0.332$.
- ReLU: $y = \max(0, -0.7) = 0$.
- Sign: $y = -1$.

Same pre-activation, three different outputs: the activation function decides how the neuron responds.

Common doubts

FIRST PASS **Why do we need a bias at all?**

Without a bias, $z = w^\top x$ is always 0 when $x = 0$, so the decision boundary $w^\top x = 0$ must pass through the origin. The bias lets the boundary sit anywhere. In one dimension it's the difference between a line through the origin, $y = mx$, and any line, $y = mx + c$.

FIRST PASS **Is it still a neuron if it has no activation function?**

Yes. A neuron with no activation is said to use the **identity** (or linear) activation, $f(z) = z$: it outputs its weighted sum plus bias unchanged. Output neurons for regression are usually like this. Linear regression is exactly one such neuron.

GOING DEEPER **Can a neuron have more than one activation function?**

In the standard model, no: a neuron computes one weighted sum and applies one activation to it. If you applied two functions one after the other, that composition is just a single, different activation function. Different neurons can use different activations, though, and a layer's output can go through extra operations such as softmax or normalisation.

GOING DEEPER **Can a neuron contain another neuron inside it?**

No. A neuron is the smallest unit: a weighted sum, a bias and an activation. Structure comes from **connecting** neurons: the outputs of one layer become the inputs of the next. When a "neuron" seems to hide something more complex, it is really a small network of neurons.

Cheatsheet

The neuron

$y = f(w^\top x + b)$. Pre-activation $z = w^\top x + b$; bias = weight on a constant input 1.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/the-neuron/the-artificial-neuron

Decision Boundaries and Linear Separability

A neuron with a threshold activation splits the input space with a straight line (a hyperplane). Which problems that can solve: AND and OR worked through, and the definition of linear separability.

The decision boundary

With a sign activation, the neuron outputs $+1$ when $z = w^\top x + b > 0$ and -1 when $z < 0$. The **decision boundary** is where $z = 0$:

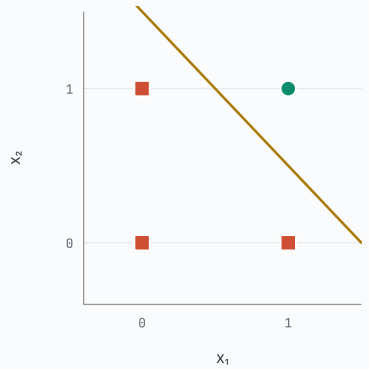
$$w^\top x + b = 0$$

In 2 dimensions this is a line, in 3 a plane, in d a **hyperplane**. The weight vector w is perpendicular (normal) to it and points towards the side classified $+1$; the bias moves it away from the origin.

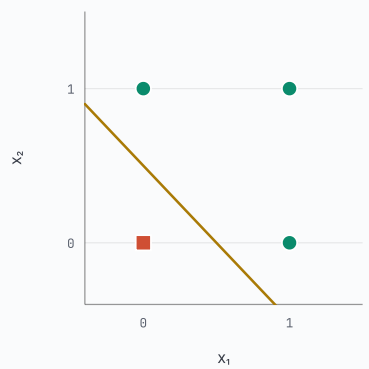
Linear separability

Two classes are **linearly separable** if some w and b put every positive example on one side and every negative example on the other: $w^\top x + b > 0$ for all positives and < 0 for all negatives. A single neuron can only solve linearly separable problems.

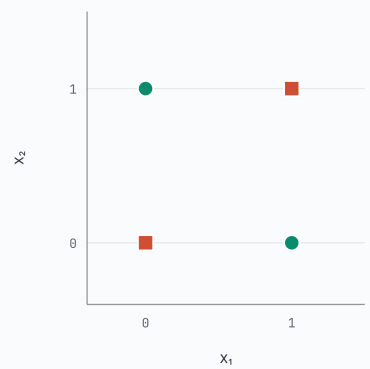
AND and OR are separable



AND: w = (1, 1), b = −1.5



OR: w = (1, 1), b = −0.5



XOR: no single line works

(X_1, X_2)	AND: $X_1 + X_2 - 1.5$	OR: $X_1 + X_2 - 0.5$
(0, 0)	−1.5 → −1	−0.5 → −1
(1, 0)	−0.5 → −1	0.5 → +1
(0, 1)	−0.5 → −1	0.5 → +1
(1, 1)	0.5 → +1	1.5 → +1

Same weights, different bias: the bias slides the line, turning AND into OR. XOR (true when exactly one input is 1) puts the two positives on opposite corners, and no line separates them. The next lesson proves it.

EXAM PHRASING

"A dataset is linearly separable if a hyperplane can separate the classes without error. AND and OR are linearly separable; XOR is not."

Common doubts

FIRST PASS **Why is w perpendicular to the boundary?**

Take two points x_a and x_b on the boundary: $w^\top x_a + b = 0$ and $w^\top x_b + b = 0$. Subtracting gives $w^\top(x_a - x_b) = 0$, so w is at right angles to every direction along the boundary. Moving in the direction of w increases z fastest, which is why w points to the positive side.

Cheatsheet

Decision boundary

$w^\top x + b = 0$: a line (2-D), plane (3-D) or hyperplane. w is normal to it and points to the positive side.

Separable or not

AND, OR: linearly separable (one neuron works).
XOR: not separable (needs a hidden layer).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/the-neuron/decision-boundaries-and-linear-separability

The XOR Problem

A proof that no single neuron can compute XOR, what happens when you try to train one anyway, and two ways out: hand-made features or a hidden layer.

The impossibility proof

XOR is positive for $(1, 0)$ and $(0, 1)$ and negative for $(0, 0)$ and $(1, 1)$. Suppose one neuron with weights w_1, w_2 and bias b classified all four correctly. Then:

INPUT	CLASS	REQUIREMENT
(0, 0)	negative	$b < 0$
(1, 1)	negative	$w_1 + w_2 + b < 0$
(1, 0)	positive	$w_1 + b > 0$
(0, 1)	positive	$w_2 + b > 0$

- Adding the two **positive** rows: $w_1 + w_2 + 2b > 0$.
- Adding the two **negative** rows: $w_1 + w_2 + 2b < 0$.

The same number can't be both positive and negative, so no such w_1, w_2, b exist. **No single neuron with a linear decision boundary can compute XOR.**

> WHY IT MATTERS
>
> This is a proof of impossibility, not a failed training run. No amount of training, data or learning-rate tuning will make one neuron solve XOR.

What training does anyway

Train a single sigmoid neuron on XOR with mean squared error and it gets stuck: every output settles near 0.5 and the loss at 0.25. With targets 0 and 1, predicting 0.5 everywhere gives $\frac{1}{4}\sum(0.5 - y)^2 = 0.25$: the best a straight boundary can do is refuse to decide.

Way out 1: hand-made features

XOR can be written as $x_1 + x_2 - 2x_1x_2$. If you add the feature x_1x_2 as a third input, XOR is **linear** in (x_1, x_2, x_1x_2) and one neuron can solve it. But this needs someone to design the right feature.

Way out 2: a hidden layer

A network with a **hidden layer** can learn a useful representation itself. Section 3 builds an explicit 2–2–1 network for XOR: its hidden units compute OR and AND, and the output combines them. That is the core idea of deep learning: layers learn the features.

Common doubts

FIRST PASS **Is XOR unsolvable because we did not train long enough?**

No. The inequalities show that **no** weights work, so this is about what one neuron can represent, not about training. A failed training run could be bad luck; a contradiction in the inequalities is a proof.

GOING DEEPER **Why does the trained neuron output exactly 0.5 for every input?**

With MSE and targets 0 and 1, each pair of opposite corners pulls the boundary in opposite directions and they cancel. The point that minimises the squared error when no boundary can separate the classes is to predict the mean target, 0.5, everywhere, giving loss 0.25.

Cheatsheet

XOR proof in one line

Positives give $w_1 + w_2 + 2b > 0$; negatives give $w_1 + w_2 + 2b < 0$. Contradiction, so one neuron cannot solve XOR.

Two ways to solve XOR

Add the feature $x_1 x_2$ (XOR $= x_1 + x_2 - 2x_1 x_2$), or add a hidden layer that learns the representation.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/the-neuron/the-xor-problem

Section 3

Multi-layer Perceptrons

How layers of neurons are written as matrices, how a hidden layer solves XOR, and why depth and non-linearity give networks their power.

3.1 Layers, Matrices and Shapes · 11 min

3.2 Solving XOR with a Hidden Layer · 8 min

3.3 Why Depth Helps: Composing Regions · 7 min

3.4 Why Non-linearity Is Essential · 6 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

M2

BUILDING AND TRAINING A NETWORK · DENSE LANE · STEP M2

Stack layers

How do hidden layers create new representations?

Lessons here: 3.1, 3.2, 3.3, 3.4 · feeds into Choose the activations

Layers, Matrices and Shapes

A multi-layer perceptron stacks layers of neurons. How one layer is written as a matrix product, how to read the shapes, how to count parameters, and the two conventions (lecture vs code) you must not mix.

One layer as a matrix product

A layer applies many neurons to the same input at once. Stack each neuron's weights as a **row** of a matrix W and its bias into a vector b:

$$y^{(0)} = x, \qquad y^{(k)} = f_k\left(W_k\, y^{(k-1)} + b_k\right), \quad k = 1, \ldots, L$$

The output of one layer is the input of the next. Computing all layers in order is the **forward pass**.

Reading the shapes

For a layer with n inputs and m neurons:

- W is $m \times n$ (one row per neuron, one column per input);
- b has m entries;
- the layer has $mn + m$ trainable parameters.

WORKED EXAMPLES: COUNTING PARAMETERS

NETWORK	LAYER 1	LAYER 2	TOTAL
2 → 3 → 1	$3 \times 2 + 3 = 9$	$1 \times 3 + 1 = 4$	13
100 → 50 → 10	$50 \times 100 + 50 = 5{,}050$	$10 \times 50 + 10 = 510$	5,560
784 → 128 → 10	$128 \times 784 + 128 = 100{,}480$	$10 \times 128 + 10 = 1{,}290$	101,770

Almost all the parameters sit in the widest connections. A 28×28 image flattened to 784 inputs already needs 100,352 weights for one modest hidden layer, which is part of why images use convolutions (Section 10).

Worked example: a 2–3–1 network, neuron by neuron

Two inputs, three hidden neurons with ReLU, and one output neuron with the identity activation (so it can output any number, as in regression). Each neuron has its **own** weights and bias.

THE FORWARD PASS FOR $X = (1, 2)$

NEURON	WEIGHTS	BIAS	WEIGHTED SUM z	OUTPUT
h_1	$(0.5,\ -0.2)$	0.1	$0.5(1) - 0.2(2) + 0.1 = 0.2$	$\mathrm{ReLU}(0.2) = 0.2$
h_2	$(-0.3,\ 0.4)$	0	$-0.3(1) + 0.4(2) + 0 = 0.5$	$\mathrm{ReLU}(0.5) = 0.5$
h_3	$(0.2,\ -0.5)$	0.2	$0.2(1) - 0.5(2) + 0.2 = -0.6$	$\mathrm{ReLU}(-0.6) = 0$
$\hat{y}$	$(1.0,\ -0.5,\ 0.8)$	0.3	$1.0(0.2) - 0.5(0.5) + 0.8(0) + 0.3 = 0.25$	identity: 0.25

- Information flows one way: the inputs feed every hidden neuron, and the three hidden outputs feed the output neuron.
- h_3's weighted sum is negative, so ReLU outputs 0: for this input, h_3 contributes nothing.
- Parameters: $3 \times 2 + 3 = 9$ in the hidden layer, $1 \times 3 + 1 = 4$ at the output: **13** in all.

Two conventions: don't mix them

	LECTURE (MATHS)	CODE (NUMPY, PYTORCH)
Input	One example, a column vector x	A batch, matrix X with one example per **row**
Weight shape	W: outputs × inputs	W: inputs × outputs
Layer	$Wx + b$	$XW + b$

Both are correct; they are transposes of each other. State which one you use and check that every product has matching inner dimensions.

Common doubts

FIRST PASS **Why is the weight matrix outputs × inputs and not the other way round?**

Because each **row** holds one neuron's weights, and a matrix–vector product takes the dot product of each row with x. With m neurons and n inputs, that's m rows of length n. Code that stores one example per row uses the transpose, inputs × outputs, so that XW works.

FIRST PASS **Do all neurons in a layer have the same weights and biases?**

No. Each neuron has its own weights and bias, and that's the point: each one learns to detect something different. If they all started identical, they would get identical gradients and stay identical forever, so the layer would act like one neuron. That's why weights start **random**.

FIRST PASS **How do the weights and the bias change a neuron's output?**

Each weight scales how much one input matters, and its sign says whether that input pushes the output up or down. The bias shifts the weighted sum up or down regardless of the inputs, which moves the point where ReLU switches on. In the example, raising h_3's bias from 0.2 to 1.0 would make its sum $+0.2$, so it would switch on.

FIRST PASS **How do I multiply a 1×2 by a 2×1, and a 2×1 by a 1×2?**

Write the shapes side by side. The **inner** numbers must match, and the result has the **outer** numbers.

PRODUCT	SHAPES	RESULT	EXAMPLE
Row × column	(1×**2**)(**2**×1)	1×1: one number (a **dot product**)	$\begin{bmatrix}1 & 2\end{bmatrix}\begin{bmatrix}3\\4\end{bmatrix} = 1\cdot 3 + 2\cdot 4 = 11$
Column × row	(2×**1**)(**1**×2)	2×2: a full table (an **outer product**)	$\begin{bmatrix}3\\4\end{bmatrix}\begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}3 & 6\\4 & 8\end{bmatrix}$

Both appear in a network. The forward pass uses dot products: $z = w^\top x + b$. Backpropagation uses an outer product for a layer's weight gradient, $\partial L/\partial W = \delta\, a^\top$: every entry is (error at that output) × (input on that connection).

Thumb rule: "row times column gives a number; column times row gives a table."

Cheatsheet

Layer and parameter count

$y^{(k)} = f_k(W_k y^{(k-1)} + b_k)$. n inputs, m neurons: W is $m \times n$, $mn + m$ parameters.

Shape conventions

Lecture: $Wx + b$, W is outputs×inputs. Code: $XW + b$, rows are examples, W is inputs×outputs.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/multi-layer-perceptrons/layers-matrices-and-shapes

Solving XOR with a Hidden Layer

An explicit 2–2–1 network that computes XOR, checked for all four inputs. The hidden layer changes the representation so that one output neuron can separate the classes.

The network

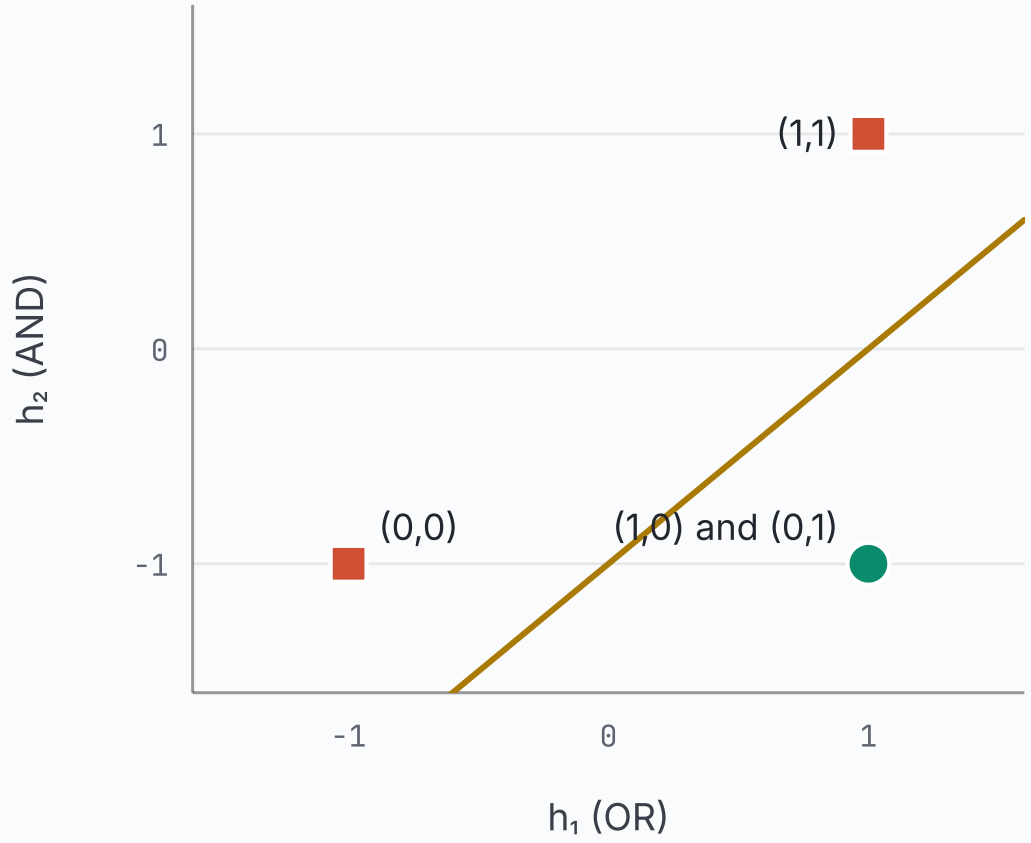
Two inputs, two hidden neurons, one output, all with the sign activation:

$$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} -0.5 \\ -1.5 \end{bmatrix}, \qquad W_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad b_2 = -1$$

Hidden unit 1 computes $x_1 + x_2 - 0.5$: that's **OR**. Hidden unit 2 computes $x_1 + x_2 - 1.5$: that's **AND**. The output computes $h_1 - h_2 - 1$: "OR and not AND", which is XOR.

INPUT	HIDDEN PRE-ACTIVATIONS	HIDDEN OUTPUTS (H_1, H_2)	OUTPUT $H_1 - H_2 - 1$	XOR
(0, 0)	(−0.5, −1.5)	(−1, −1)	−1 + 1 − 1 = −1	−1 ✓
(1, 0)	(0.5, −0.5)	(+1, −1)	1 + 1 − 1 = +1	+1 ✓
(0, 1)	(0.5, −0.5)	(+1, −1)	1 + 1 − 1 = +1	+1 ✓
(1, 1)	(1.5, 0.5)	(+1, +1)	1 − 1 − 1 = −1	−1 ✓

What the hidden layer did



In the hidden layer's coordinates, the two positive inputs land on the same point and the classes are linearly separable, so one output neuron finishes the job.

The hidden layer maps the four inputs to new coordinates in which the classes **are** linearly separable. Both positive inputs even land on the same point.

A COMMON MISSTATEMENT

"The hidden layer works by mapping to a higher dimension." Not here: the hidden layer has 2 units, the same as the input. It's the **non-linear** mapping that makes the classes separable, not extra dimensions.

Common doubts

FIRST PASS **Would gradient descent find these exact weights?**

Not with sign activations: their derivative is zero almost everywhere, so there is no gradient to follow (Section 4). With sigmoid units, training finds *some* solution, usually a different one, and not always: with only 2 hidden units, a fair share of random starts get stuck. With 3 or more hidden units training succeeds far more reliably.

FIRST PASS **Where do the thresholds 0.5 and 1.5 come from?**

Look at the sum $x_1 + x_2$ for the four inputs: it's 0, 1, 1 or 2.

- **OR** should fire when the sum is 1 or more, so put its threshold **between 0 and 1**: 0.5.
- **AND** should fire only when the sum is 2, so put its threshold **between 1 and 2**: 1.5.

Any value inside those gaps works; the middle of the gap is the safest choice. The output works the same way: $h_1 - h_2$ is 0 for the negatives and 2 for the positives, so its threshold sits in the middle, at 1 (the bias -1).

FIRST PASS **Is there an easy way to remember this network?**

XOR = OR and not AND. The hidden layer computes OR and AND; the output says "yes" when OR is on and AND is off.

In the input square, OR and AND are two parallel lines. The two positive points, (1,0) and (0,1), lie in the strip between them. A single neuron can draw only one line, which is why it needs the hidden layer to draw the second.

Cheatsheet

XOR with a hidden layer

Hidden units compute OR ($x_1 + x_2 - 0.5$) and AND ($x_1 + x_2 - 1.5$); output $h_1 - h_2 - 1$. The non-linear hidden representation makes XOR separable.

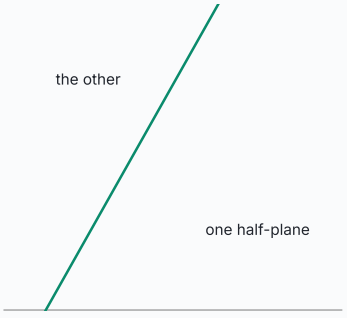
Online version with the interactive quiz: airesearchlabs.si/learn/dl/multi-layer-perceptrons/solving-xor-with-a-hidden-layer

Why Depth Helps: Composing Regions

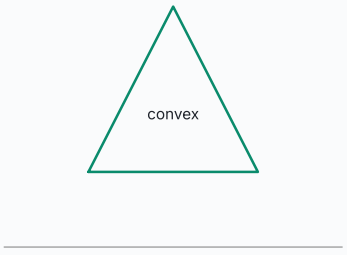
With threshold neurons, the first layer cuts the space into half-planes, the next combines them into convex regions, and the next into unions of regions. That is the geometric picture of what depth adds.

Layer by layer

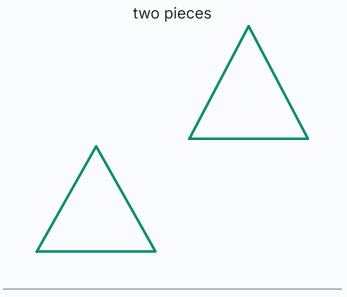
LAYER	EACH UNIT COMPUTES	REGION IT CAN DESCRIBE
1st hidden layer	A line (hyperplane) test	A **half-plane**
2nd layer, AND-like unit	"All of these half-plane tests are true"	An **intersection** of half-planes: a **convex** region (triangle, polygon)
3rd layer, OR-like unit	"Any of these convex regions"	A **union** of convex regions: non-convex or even disconnected shapes



Layer 1: each unit draws one line.



Layer 2: AND of three lines gives a triangle.



Layer 3: OR of two triangles gives a disconnected region.

The general lesson: composition

Later layers build on what earlier layers detect. In images the same idea appears as edges → textures and parts → objects (Section 10). Depth lets a network express structured, complicated functions by combining simple pieces, often with far fewer units than a single wide layer would need.

DON'T OVERSTATE IT

This geometric picture uses idealised threshold units. Real networks use smooth activations, and depth also makes training harder (vanishing gradients, Section 4). Section 11 weighs depth against width.

Common doubts

GOING DEEPER **Can a single hidden layer not also describe a triangle?**

A single hidden layer of threshold units followed by one output unit can combine half-planes with a weighted vote, so with the right weights it can carve out a convex region. What needs another layer is **combining regions**, such as the union of two separate triangles. With smooth activations and enough units, one hidden layer can approximate any continuous function (Section 5), but it may need vastly more units than a deeper network.

Cheatsheet

What depth adds

Layer 1: half-planes. Layer 2 (AND): convex regions. Layer 3 (OR): unions, so non-convex or disconnected regions.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/multi-layer-perceptrons/why-depth-helps

Why Non-linearity Is Essential

Stacking layers without non-linear activations gives nothing new: any stack of linear layers collapses to a single linear layer. The algebra, a numerical check, and the consequence for XOR.

The collapse

Suppose both layers use the identity activation $f(z) = z$:

$$W_2(W_1 x + b_1) + b_2 = \underbrace{(W_2 W_1)}_{W} x + \underbrace{(W_2 b_1 + b_2)}_{b}$$

That is a single affine layer $Wx + b$. Repeating the argument, **any number of linear layers collapses to one**. Its decision boundary is still a hyperplane, so it still can't solve XOR.

NUMERICAL CHECK

$W_1 = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$, $b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $W_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}$, $b_2 = 2$.

- Collapsed: $W_2 W_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$ and $W_2 b_1 + b_2 = 1 + 2 = 3$, so $y = x_1 + x_2 + 3$.
- Check with $x = (1, 1)$: layer 1 gives $(1 + 2 + 1,\ 0 + 1 + 0) = (4, 1)$; layer 2 gives $4 - 1 + 2 = 5$. The collapsed formula gives $1 + 1 + 3 = 5$. ✓

What non-linearity buys

Put a non-linear f between the layers and the composition no longer simplifies: each layer can bend and fold the space, which is how the hidden layer in the XOR network made the classes separable. Non-linear activations are what give depth its power.

EXAM PHRASING

"Without non-linear activations, stacked layers compose to a single affine map $(W_2 W_1)x + (W_2 b_1 + b_2)$, so a deep linear network is no more expressive than one linear layer and cannot solve XOR."

Common doubts

FIRST PASS **So is a linear activation ever useful?**

Yes, at the **output** of a regression network, where you want an unbounded real number (Section 6). What you shouldn't do is use only linear activations in the **hidden** layers, because then the hidden layers add nothing.

FIRST PASS **Does linear regression need a non-linear activation function?**

No. Linear regression is one neuron with the identity activation: $\hat{y} = w^\top x + b$. That is exactly what it should be, because it models a straight-line (linear) relationship. Non-linear activations matter when you stack **hidden** layers: without them, the layers collapse into one linear map, as shown above.

Cheatsheet

Linear layers collapse

$W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (W_2 b_1 + b_2)$: one affine layer. Depth needs non-linear activations.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/multi-layer-perceptrons/why-non-linearity-is-essential

Section 4

Activation Functions

The common activations and their derivatives, the vanishing-gradient and dying-ReLU problems, and how to choose activations for hidden and output layers.

4.1 Sigmoid, tanh, ReLU and Friends · 9 min

4.2 Vanishing Gradients · 7 min

4.3 Dying ReLU, Leaky ReLU and ELU · 6 min

4.4 Choosing Activation Functions · 5 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

4

BUILDING AND TRAINING A NETWORK · STEP 4

Choose the activations

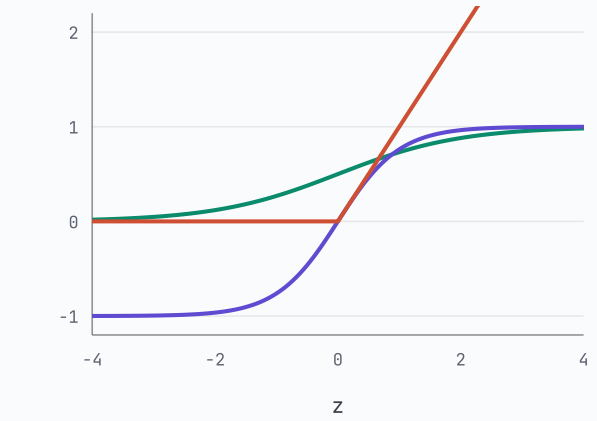
Which non-linearity in hidden layers, and which at the output?

Lessons here: 4.1, 4.2, 4.3, 4.4 · feeds into Compute the gradients

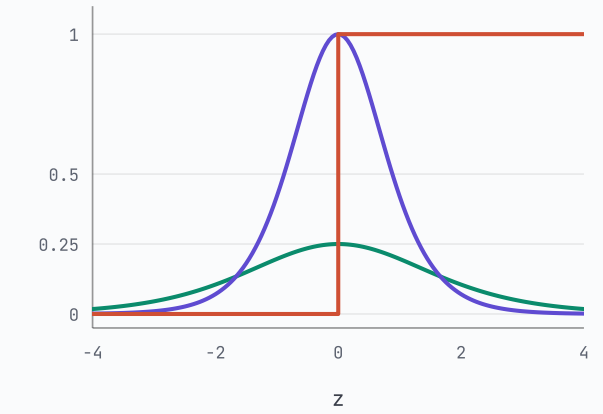
Sigmoid, tanh, ReLU and Friends

The common activation functions side by side: their formulas, ranges and derivatives, where each is used, and why the sign function cannot be trained by gradient descent.

The functions



Sigmoid squashes to (0, 1), tanh to (−1, 1); ReLU is zero for negative inputs and the identity for positive ones.



Derivatives. Sigmoid and tanh fade to zero for large |z| (saturation). ReLU's derivative is exactly 1 on the positive side.

ACTIVATION	FORMULA	RANGE	DERIVATIVE	TYPICAL USE AND LIMITATION
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	(0, 1)	$\sigma(1-\sigma)$, at most 0.25	Binary output probability; saturates
tanh	$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	(−1, 1)	$1 - \tanh^2 z$, at most 1	Zero-centred hidden units; still saturates
ReLU	$\max(0, z)$	[0, ∞)	0 for $z < 0$, 1 for $z > 0$	Default for hidden layers; can "die"
Sign	$+1$ if $z > 0$, -1 if $z < 0$	{−1, +1}	0 almost everywhere	Threshold constructions; untrainable by gradients
Linear	z	(−∞, ∞)	1	Regression output
Softmax	$o_j = \frac{e^{s_j}}{\sum_k e^{s_k}}$	(0, 1), sums to 1	Section 6	Multi-class output probabilities

Two derivatives to know by heart

Sigmoid. $\sigma'(z) = \sigma(z)\,\big(1 - \sigma(z)\big)$. It peaks at $z = 0$, where $\sigma = 0.5$ and $\sigma' = 0.25$. At $z = 2$: $\sigma = 0.881$, $\sigma' = 0.105$. At $z = 5$: $\sigma = 0.993$, $\sigma' = 0.0066$. Large inputs **saturate** the unit and kill its gradient.

tanh. $\tanh'(z) = 1 - \tanh^2(z)$, which peaks at 1. tanh is a rescaled sigmoid, $\tanh z = 2\sigma(2z) - 1$, and being zero-centred it often trains a little better in hidden layers.

Why the sign function can't be trained

Away from $z = 0$, a small change in the weights doesn't change a sign unit's output at all, so its derivative is **zero almost everywhere** (and undefined at 0). Gradient descent gets no signal about which way to move. That's why trainable networks use smooth or piecewise-linear activations instead.

Common doubts

FIRST PASS **How do I remember the sigmoid derivative?**

Write $\sigma = (1 + e^{-z})^{-1}$ and differentiate: $\sigma' = e^{-z}(1 + e^{-z})^{-2}$. Since $e^{-z}/(1 + e^{-z}) = 1 - \sigma$, this is $\sigma(1 - \sigma)$. In backpropagation you already have σ from the forward pass, so the derivative costs one multiplication: $a(1 - a)$.

GOING DEEPER **How do I derive σ′ = σ(1 − σ)? Why isn't the derivative of e^(−z) something like −e^(−2z)?**

Write $\sigma(z) = (1 + e^{-z})^{-1}$ and use the chain rule, outside then inside:

1. **Outside, power rule:** $\frac{d}{du}u^{-1} = -u^{-2}$, so you get $-(1 + e^{-z})^{-2}$.
2. **Inside, exponential rule:** $\frac{d}{dz}(1 + e^{-z}) = -e^{-z}$. The 1 is a constant, so it vanishes. The exponent stays $-z$; you only multiply by its derivative, -1.
3. **Multiply:** $-(1 + e^{-z})^{-2} \cdot (-e^{-z}) = \dfrac{e^{-z}}{(1 + e^{-z})^2} = \dfrac{1}{1 + e^{-z}} \cdot \dfrac{e^{-z}}{1 + e^{-z}} = \sigma\,(1 - \sigma)$.

Thumb rule: "The power rule lowers the power. The exponential rule keeps the exponent and multiplies by its derivative." So $\frac{d}{dz}e^{-z} = -e^{-z}$, never $-e^{-2z}$.

Cheatsheet

Sigmoid

$\sigma(z) = 1/(1 + e^{-z}) \in (0, 1)$; $\sigma' = \sigma(1 - \sigma) \leq 0.25$ (at $z = 0$). Saturates for large $|z|$.

tanh and ReLU

$\tanh \in (-1, 1)$, $\tanh' = 1 - \tanh^2 \leq 1$. ReLU $= \max(0, z)$, derivative 0 or 1.

Sign is untrainable

Its derivative is 0 almost everywhere, so gradient descent gets no signal. Use sigmoid, tanh or ReLU.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/activation-functions/sigmoid-tanh-relu-and-friends

Vanishing Gradients

Why early layers of a deep sigmoid network barely learn: backpropagation multiplies one activation derivative per layer, and sigmoid derivatives are at most 0.25. The numbers, the mechanism and the fixes.

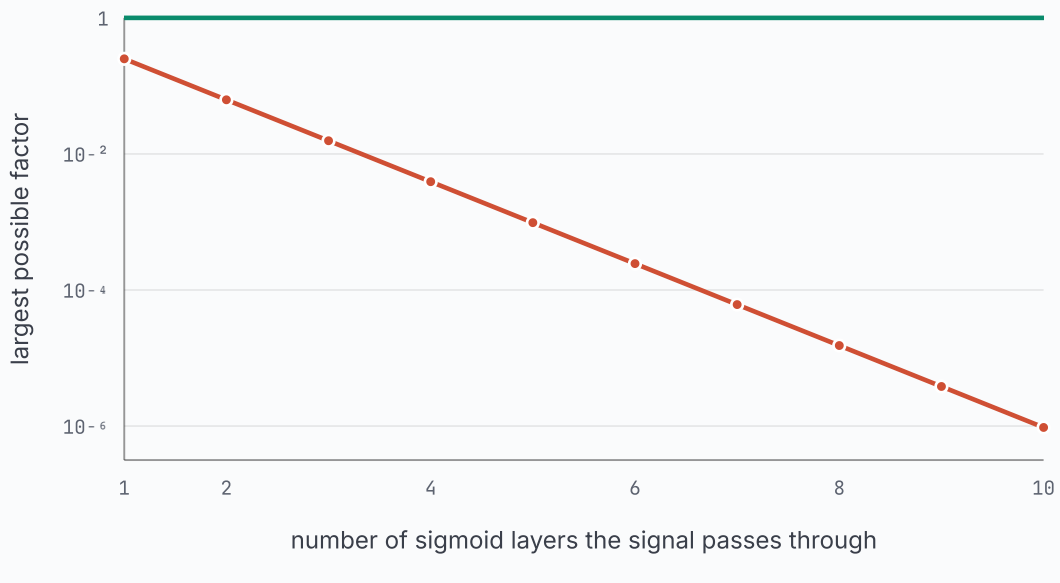
The mechanism

Backpropagation (Section 7) carries the error signal from the output back to earlier layers. At each layer it is multiplied by a transposed weight matrix and by the **activation derivative**. Written compactly, with $D_k = \operatorname{diag}\big(f'(z^{(k)})\big)$:

$$\delta^{(k)} = D_k\, W_{k+1}^{\top} D_{k+1} W_{k+2}^{\top} \cdots \delta^{(L)}$$

For sigmoid units, every factor f' is **at most 0.25**. Ten such factors give at most

$$0.25^{10} = 4^{-10} \approx 9.5 \times 10^{-7}$$



The activation-derivative part of the gradient shrinks geometrically with depth for sigmoid units (log scale). Actual gradients also depend on the weights.

So the output layers keep learning while the first layers receive almost no update. **Saturation** makes it worse: a unit with a large $|z|$ has σ' far below 0.25.

Remedies

- **ReLU** in hidden layers: its derivative is exactly 1 for active units, so the activation doesn't shrink the signal.

- **Careful initialisation** (for example He initialisation for ReLU) and **normalised inputs**, so units start out of saturation.
- Architectural fixes for very deep networks, such as batch normalisation and residual connections (beyond this course).

EXAM PHRASING

"The sigmoid derivative is at most 0.25, so backpropagating through many sigmoid layers multiplies many small factors and the gradients reaching early layers shrink towards zero. ReLU's derivative is 1 for positive inputs, so it doesn't shrink the signal."

Common doubts

GOING DEEPER **Can gradients also become too large?**

Yes. If the weights are large, the product of weight matrices can grow with depth instead of shrinking: **exploding gradients**, which make training unstable. Sensible initialisation keeps the products near 1 in both directions, and gradient clipping is a common safety net.

FIRST PASS **If the earliest layers get tiny gradients, what about the later layers?**

The later layers are fine. A layer's gradient is a product of one slope per layer **between it and the output**, so layers near the output multiply only a few small factors, and layers near the input multiply many.

With sigmoid slopes of at most 0.25: one layer back, the gradient is scaled by 0.25; five layers back, by $0.25^5 \approx 0.001$. So the last layers keep learning while the first layers barely move from their random starting weights, and the network behaves as if it were much shallower.

ReLU helps because its slope is exactly 1 for positive inputs, so the product doesn't shrink.

Cheatsheet

Vanishing gradients

Backprop multiplies one f' per layer. Sigmoid $f' \leq 0.25$, so 10 layers give $\leq 0.25^{10} \approx 10^{-6}$. ReLU ($f' = 1$ when active) avoids the shrinkage.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/activation-functions/vanishing-gradients

Dying ReLU, Leaky ReLU and ELU

ReLU solves vanishing gradients for active units but creates a new failure: a unit stuck on the negative side gets zero gradient forever. How it happens and the remedies.

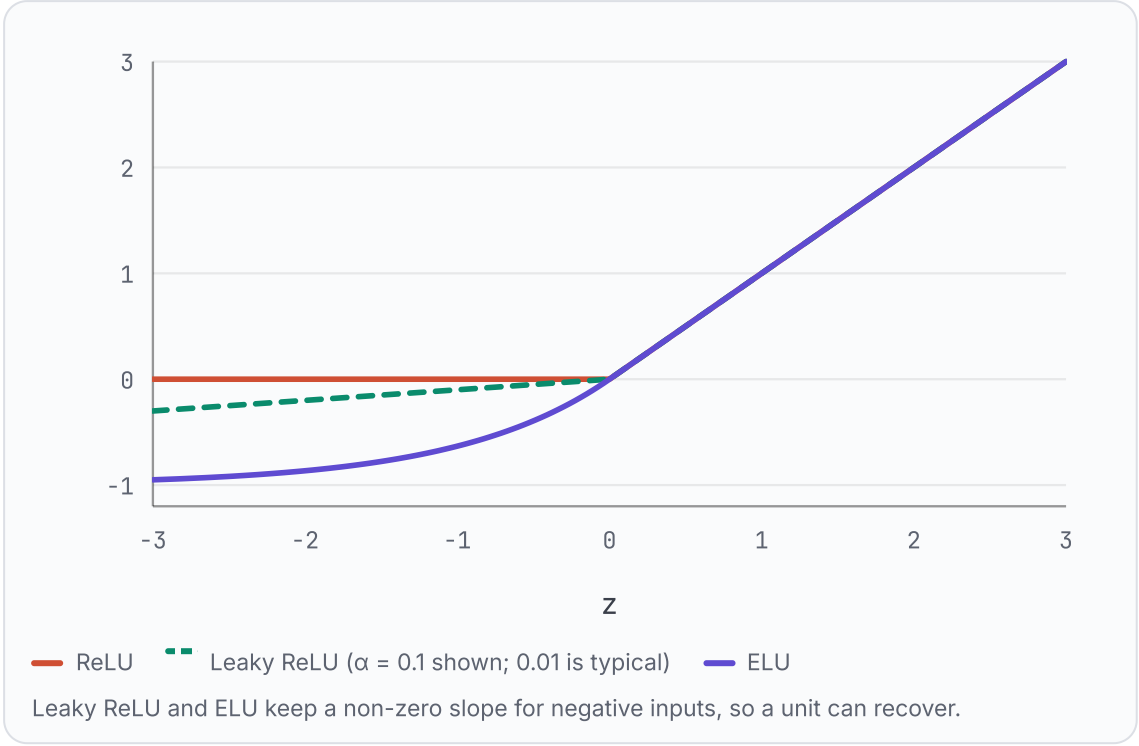
Why ReLU is the default

ReLU, $\max(0, z)$, has derivative 1 on its positive side, so active units pass the error signal back undiminished. It is also very cheap to compute and gives **sparse** activations (many units output exactly 0). It's the recommended default for hidden layers of deep networks.

Dying ReLU

If a unit's pre-activation is negative for **every** training example, its output is always 0 and so is its gradient. Gradient descent never changes its incoming weights, so it stays dead. A large learning-rate step that pushes the bias far negative is a typical cause.

Remedies



Leaky ReLU and ELU keep a non-zero slope for negative inputs, so a unit can recover.

- **Leaky ReLU:** $f(z) = z$ for $z \geq 0$ and αz for $z < 0$, with $\alpha \approx 0.01$. The small negative slope keeps a gradient flowing.
- **ELU:** a smooth curve $e^z - 1$ on the negative side, also with a non-zero gradient.
- A **smaller learning rate** and **He initialisation** make dying less likely in the first place.

DON'T CONFUSE THE TWO PROBLEMS

Vanishing gradients: many small derivative factors multiplied across layers (typical of sigmoid). **Dying ReLU**: one unit stuck at zero output and zero gradient. Different mechanisms, different fixes.

Common doubts

FIRST PASS **Is a dead ReLU unit harmful or just useless?**

Mostly useless: it always outputs 0, so it contributes nothing and wastes capacity. If many units die, the network effectively becomes much smaller than designed and may underfit.

FIRST PASS **What's the difference between "dead neurons" and "not zero-centred"?**

They are two different problems of two different activations.

	DEAD NEURONS	NOT ZERO-CENTRED
Which activation	**ReLU**	**Sigmoid** (outputs 0 to 1)
What happens	$z < 0$ for every input, so the output is 0 and the slope is 0. The neuron never updates again.	Outputs are always positive, so all weight gradients into the next layer share one sign. Updates zig-zag and training is slower.
How bad	The neuron is useless for good	Training is slower, but it still works
Fix	Leaky ReLU or ELU (a small slope for $z < 0$)	tanh (outputs −1 to 1)

Thumb rule: "ReLU can die. Sigmoid is lopsided, and it also saturates, which causes vanishing gradients."

Cheatsheet

Leaky ReLU

$f(z) = z$ for $z \geq 0$, αz for $z < 0$ ($\alpha \approx 0.01$). Keeps a gradient on the negative side; fixes dying ReLU.

Vanishing vs dying

Vanishing: small f' factors multiply across layers (sigmoid). Dying ReLU: a unit stuck at $z < 0$, output and gradient 0.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/activation-functions/dying-relu-leaky-relu-and-elu

Choosing Activation Functions

Which activation to use where: ReLU for deep hidden layers, sigmoid for shallow networks and binary outputs, linear for regression outputs, softmax for multi-class outputs.

Hidden layers

- **Deep networks:** ReLU (or Leaky ReLU / ELU if units die).
- **Shallow networks** (one or two hidden layers): sigmoid or tanh work fine; vanishing gradients only bite with depth.
- Never purely linear: the layers would collapse into one (Section 3).

Output layer: decided by the task

TASK	OUTPUT ACTIVATION	WHY
Regression (a number)	Linear	Predictions can be any real value; a sigmoid would trap them in (0, 1)
Binary classification	One sigmoid unit	Outputs $P(y = 1 \mid x)$
Multi-class classification	Softmax over the classes	Positive outputs that sum to 1, one probability per class

In PyTorch, the sigmoid or softmax is usually folded into the loss function instead of the model (Section 6.4).

RULE OF THUMB

Hidden layers: ReLU. Output: whatever makes the outputs mean what the task needs: a real number, a probability, or a probability per class.

Common doubts

FIRST PASS **Can I use a sigmoid output for regression if my targets are between 0 and 1?**

You can, if the targets genuinely live in (0, 1), such as proportions. But for general regression targets (prices, temperatures) a sigmoid caps every prediction between 0 and 1, which is wrong, and it also saturates. Use a linear output and standardise the targets instead.

FIRST PASS **Why might the output neuron have no activation function?**

For regression the target can be any real number, like a price or a temperature. ReLU would forbid negative outputs, and sigmoid or tanh would squash the output into a fixed range. The identity lets the output take any value, and the hidden layers still supply the non-linearity.

FIRST PASS **Do all neurons in a layer use the same activation function?**

In practice, yes: a layer uses one activation for all its neurons, because that is simpler and works well. Nothing in the maths forbids mixing. Different **layers** often differ, though: ReLU in the hidden layers and identity, sigmoid or softmax at the output.

Cheatsheet

Output activation by task

Regression: linear. Binary: one sigmoid. Multi-class: softmax. Hidden layers: ReLU (deep) or sigmoid/tanh (shallow).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/activation-functions/choosing-activation-functions

Section 5

The Universal Approximation Theorem

What the theorem says, the intuition behind it, and what it does not promise about training, size or generalisation.

5.1 The Universal Approximation Theorem · 7 min

5.2 Building Functions from Bumps · 7 min

5.3 What the Theorem Does Not Promise · 6 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

UNDERNEATH THE MAP · FOUNDATION

Universal approximation

Lessons here: 5.1, 5.2, 5.3 · feeds into Stack layers, Size the network

The Universal Approximation Theorem

One hidden layer with enough neurons can approximate any continuous function on a bounded domain as closely as you like. The precise statement, what each condition means, and the extension that covers ReLU.

The statement

Let σ be a continuous sigmoidal activation, $K \subset \mathbb{R}^d$ a compact (closed and bounded) set, and $h : K \to \mathbb{R}$ any continuous function. Then for every $\varepsilon > 0$ there is a network with **one hidden layer** of finitely many units,

$$\mathrm{NN}(x) = \sum_{j=1}^{n} v_j \, \sigma\big(w_j^\top x + b_j\big) \,,$$

such that

$$\sup_{x \in K} \big|h(x) - \mathrm{NN}(x)\big| < \varepsilon.$$

This is Cybenko's theorem (1989). Leshno and colleagues (1993) extended it to any continuous activation that is **not a polynomial**, which includes ReLU.

Reading each condition

CONDITION	WHAT IT MEANS
Compact K	Inputs come from a closed, bounded region, such as $[0,1]^d$. No promise about behaviour far outside it.
Continuous h	No jumps in the target. (Classification outputs are handled through probabilities; see lesson 5.3.)
Any $\varepsilon > 0$	You can make the error as small as you want, by using enough units.
$\sup$, the worst case	The error bound holds **everywhere** on K at once (uniform approximation), not just on average.
Non-polynomial activation	A polynomial activation only produces polynomials of bounded degree, which can't approximate everything.

PLAIN-LANGUAGE VERSION

"On a closed and bounded input domain, a neural network with a single hidden layer, enough neurons and a suitable non-linear activation can approximate any continuous function to any desired accuracy."

Common doubts

FIRST PASS **If one hidden layer is enough, why do we build deep networks?**

Because "enough neurons" can mean an impractically huge number, and the theorem says nothing about finding the weights or about generalising from limited data. Deep networks can often represent the same function with far fewer units, by composing simple pieces (Section 3). Lesson 5.3 lists what the theorem doesn't promise.

FIRST PASS **What do "continuous function", "approximate" and "compact set" mean in plain words?**

TERM	PLAIN MEANING	EXAMPLE
Continuous function	You can draw it without lifting your pen: no jumps	$\sin x$ is continuous; a step from 0 to 1 is not
Approximate	Get as close as you want to the target function, everywhere on the domain (within ε)	Error below 0.01 at every input in $[0, 1]$
Compact set	A closed, bounded range of inputs	$[0, 1]$ or a box $[0, 1]^d$; not "all real numbers"

"Approximate" here is about **representing** a function that's already known. It isn't about predicting new data: generalisation is a separate question the theorem doesn't answer.

FIRST PASS **Does the theorem say that in practice we need more than one hidden layer?**

No. It says one hidden layer is **enough to exist**. The reason we build deep networks in practice comes from elsewhere: the one-layer network may need an impractically large number of neurons, while a deeper network can represent the same function with far fewer (lesson 11.1).

Cheatsheet

Universal approximation

Continuous h on compact K, any $\varepsilon > 0$: a one-hidden-layer network with enough units has $\sup_{x \in K} |h(x) - \mathrm{NN}(x)| < \varepsilon$. Non-polynomial activations (incl. ReLU).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/universal-approximation/the-universal-approximation-theorem

Building Functions from Bumps

The intuition behind universal approximation in one dimension: two steep sigmoids make a bump, many bumps make a staircase, and narrower steps track any continuous curve.

Two sigmoids make a bump

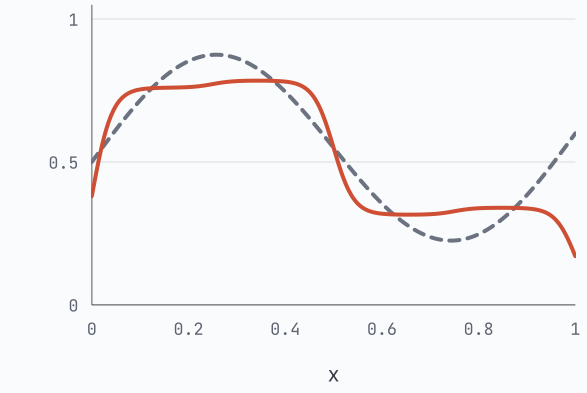
A very steep sigmoid $\sigma(k(x-a))$ with large k is almost a step that switches on at $x = a$. Subtract a second step that switches on at b:

$$\text{bump}_{[a,b]}(x) = \sigma\big(k(x-a)\big) - \sigma\big(k(x-b)\big)$$

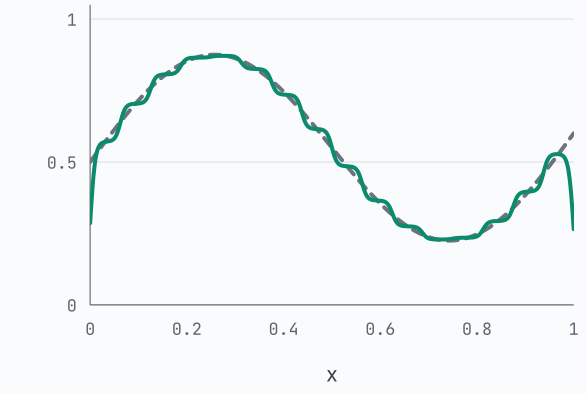
This is about 1 between a and b and about 0 elsewhere. It needs just two hidden units.

Many bumps make any curve

Split the domain into N small intervals. Give each one a bump scaled to the target's height there, and add them all up in the output unit. The result is a staircase that follows the target. Narrower intervals mean smaller steps and less error, at the cost of more units ($2N$ hidden units for N bumps).



Four bumps: a coarse staircase.



Sixteen bumps: already close everywhere.

This is not how a trained network actually works, but it shows why a single hidden layer has, in principle, enough expressive power.

Common doubts

GOING DEEPER **How does this extend to more than one input?**

In two or more dimensions you can build localised "towers" by combining several steep units in different directions, and then sum towers the same way. The number of units needed grows very quickly with the input dimension, which is one reason the theorem's networks can be impractically large.

Cheatsheet

Bump intuition

$\sigma(k(x-a)) - \sigma(k(x-b)) \approx 1$ on $[a, b]$. Sum scaled bumps → staircase approximation; more bumps, smaller error, more units.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/universal-approximation/building-functions-from-bumps

What the Theorem Does Not Promise

The universal approximation theorem is an existence result. It says nothing about how many neurons, how to find the weights, or whether the network will generalise. Plus: how it applies to classifiers with discrete outputs.

Existence is not attainability

The theorem guarantees that a suitable network **exists**. It does not say:

- **How many neurons** you need: the number can be astronomically large.
- **How to find the weights**: it gives no training algorithm.
- **That gradient descent will find them**: training can get stuck even when a solution exists.
- **Anything about generalisation**: fitting a function on the training data is not the same as predicting new data well.
- **How much data** you need.

FROM THE LAB

A 2–2–1 sigmoid network *can* represent XOR, yet in a lab experiment it failed to train in about 20% of random starts: the two hidden units ended up computing the same function, or one saturated. With 3 or more hidden units, every run succeeded. Sufficient capacity is not the same as easy training.

Is one hidden layer enough for every task?

In theory yes, in practice usually not. Deep networks can represent many functions far more efficiently, with fewer units, by building **hierarchical features**, and they often train and generalise better for structured data such as images. Section 11 compares depth and width.

Classifiers have discrete outputs

The theorem is about continuous functions, but a class label is discrete. The way round it: a classifier doesn't output the label directly. It outputs **probabilities** (through a sigmoid or softmax), which are continuous functions of the input, and the network can approximate those. A decision rule, such as picking the most probable class, then turns them into a label.

EXAM-READY WORDING

"The universal approximation theorem is an existence result about approximation. It does not guarantee a practical network size, an efficient training procedure, good generalisation or modest data requirements."

Common doubts

FIRST PASS **Does the theorem mean overfitting is impossible?**

The opposite, if anything. A network powerful enough to approximate any function is also powerful enough to memorise noise in the training data. The theorem is about what can be represented, not about what will be learned from a finite sample.

Cheatsheet

UAT limits

Existence only: no bound on size, no training method, no guarantee GD finds it, nothing about generalisation or data needed.

UAT and classifiers

Approximate the continuous class probabilities (sigmoid/softmax), then apply a decision rule (argmax / threshold).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/universal-approximation/what-the-theorem-does-not-promise

Section 6

Loss Functions and Output Layers

How training minimises a loss, how MSE, MAE and cross-entropy behave, and how to pair output layers with losses.

6.1 Training as Minimising a Loss · 8 min

6.2 MSE, MAE and Asymmetric Losses · 8 min

6.3 Softmax and Cross-entropy · 10 min

6.4 Binary, Multi-class and Logits in PyTorch · 8 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

5

BUILDING AND TRAINING A NETWORK · STEP 5

Choose the output and loss

How do I measure how wrong the network is?

Lessons here: 6.1, 6.2, 6.3, 6.4 · feeds into Compute the gradients

Training as Minimising a Loss

Training has two steps: choose a loss that measures how wrong the model is, then minimise it. Worked through for a straight line with mean squared error: the gradients, one update, and the loss going down.

Two decisions

1. **Choose a loss** $J(\theta)$ that measures how badly the model with parameters θ does on the data.
2. **Minimise it**, usually with gradient descent: $\theta \leftarrow \theta - \eta \nabla J(\theta)$.

Mean squared error for a line

For $\hat{y}^{(i)} = mx^{(i)} + c$ and N examples:

$$J(m, c) = \frac{1}{N} \sum_{i=1}^{N} \left(mx^{(i)} + c - y^{(i)} \right)^2$$

Differentiate with the chain rule:

$$\frac{\partial J}{\partial m} = -\frac{2}{N} \sum_i x^{(i)} \left(y^{(i)} - mx^{(i)} - c \right), \qquad \frac{\partial J}{\partial c} = -\frac{2}{N} \sum_i \left(y^{(i)} - mx^{(i)} - c \right)$$

WORKED EXAMPLE: ONE GRADIENT STEP

Data $(1, 2), (2, 4), (3, 5)$. Start at $m = 1$, $c = 0$; residuals $y - mx - c$ are $1, 2, 2$.

- $J = (1 + 4 + 4)/3 = 3$.
- $\partial J/\partial m = -\frac{2}{3}(1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2) = -\frac{22}{3} \approx -7.33$.
- $\partial J/\partial c = -\frac{2}{3}(1 + 2 + 2) = -\frac{10}{3} \approx -3.33$.
- With $\eta = 0.05$: $m = 1 + 0.367 = 1.367$, $c = 0 + 0.167 = 0.167$.
- New predictions $1.53, 2.90, 4.27$; new $J \approx 0.66$, down from 3.

Neural networks do exactly this, with millions of parameters and backpropagation (Section 7) to get the gradients.

Common doubts

FIRST PASS **Why are the gradients negative when the line is too low?**

The residuals $y - \hat{y}$ are positive (the line is below the points), so $\partial J/\partial m$ and $\partial J/\partial c$ come out negative. Gradient descent subtracts them, so m and c **increase** and the line moves up towards the data, which is what we want.

FIRST PASS **What is the difference between a loss function and an objective function?**

Strictly:

- **Loss** $\ell(\hat{y}_i, y_i)$: the error on **one** example.
- **Cost**, or empirical risk: the average loss over the training set, $\frac{1}{n}\sum_i \ell(\hat{y}_i, y_i)$.
- **Objective**: whatever training actually minimises. That is the cost, plus a penalty if you regularise: $J(\theta) = \frac{1}{n}\sum_i \ell + \lambda\,\Omega(\theta)$.

Books and frameworks often use the words loosely (PyTorch calls everything a "loss"), and this lesson calls J the loss. Here J is the average squared error with no penalty, so it is both the cost and the objective. In an exam, define the terms you use.

GOING DEEPER **Does the objective function represent the error?**

Mostly. With no penalty, the objective is the average error on the **training** data. Two caveats:

- With a regularisation penalty, it is error plus a cost for large weights, so a lower objective doesn't always mean a lower training error.
- It is training error, not the error on new data that you actually care about. That is why a validation set is kept aside (Section 8).

Cheatsheet

MSE gradients for a line

$J = \frac{1}{N}\sum(mx + c - y)^2$; $\partial_m J = -\frac{2}{N}\sum x(y - mx - c)$; $\partial_c J = -\frac{2}{N}\sum(y - mx - c)$.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/losses-and-outputs/training-as-minimising-a-loss

MSE, MAE and Asymmetric Losses

Regression losses compared: MSE punishes big errors hard and estimates the mean; MAE is robust to outliers and estimates the median; pinball losses price over- and under-prediction differently.

The regression losses

LOSS	FORMULA (AVERAGE OVER N)	PROPERTY
MSE (L2)	$\frac{1}{N}\sum(\hat{y}_i - y_i)^2$	Estimates the **mean**; large errors dominate
MAE (L1)	$\frac{1}{N}\sum\lvert\hat{y}_i - y_i\rvert$	Estimates the **median**; robust to outliers
Pinball (asymmetric)	αr if $r \geq 0$, $-\beta r$ if $r < 0$, with $r = \hat{y} - y$	Different prices for over- and under-prediction

Why MAE resists outliers



The loss for one prediction.



Its derivative: MSE's grows with the error.

MSE's gradient is proportional to the error, so one huge residual can dominate the whole update. MAE's gradient has the same size (±1) however big the error, so a few outliers have limited pull. MAE isn't differentiable at exactly zero; a valid **subgradient** (any value in $[-1, 1]$, usually 0) is used there.

WORKED EXAMPLE: MEAN VS MEDIAN

Targets $1, 2, 3, 4, 100$. The single number that minimises MSE is the mean, 22. The one that minimises MAE is the median, 3. One outlier drags the MSE answer far from the typical value.

Asymmetric losses

When over-predicting and under-predicting cost different amounts (running out of stock versus holding extra), the pinball loss weights them differently: with $\alpha = 1$ and $\beta = 3$, under-predicting costs three times as much as over-predicting, so the model learns to aim high.

For any regression loss, use a **linear output**: a sigmoid would restrict the predictions to (0, 1).

Common doubts

GOING DEEPER **If MAE is more robust, why is MSE so common?**

MSE is smooth everywhere, its gradient shrinks near the optimum (which helps convergence), and it matches the assumption of Gaussian noise. MAE is preferable when the data has outliers you don't want to chase. Many practitioners use a hybrid (the Huber loss) that is quadratic near zero and linear far away.

Cheatsheet

MSE vs MAE

MSE: estimates the mean; gradient $2e$, outliers dominate. MAE: estimates the median; gradient ± 1, robust; subgradient at 0.

Pinball loss

$r = \hat{y} - y$: αr if $r \geq 0$, $-\beta r$ if $r < 0$. Different costs for over- and under-prediction.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/losses-and-outputs/mse-mae-and-asymmetric-losses

Softmax and Cross-entropy

Softmax turns class scores into probabilities; cross-entropy scores those probabilities. A worked example, the numerical-stability trick, and the derivation of the famously simple gradient: predicted minus target.

Softmax: scores to probabilities

The last layer of a K-class classifier produces K raw scores s_j, called **logits**. Softmax turns them into probabilities:

$$o_j = \frac{e^{s_j}}{\sum_{k=1}^{K} e^{s_k}}$$

Every o_j is positive and they sum to 1. The target z is **one-hot**: 1 for the correct class, 0 for the rest.

Cross-entropy

$$H = -\sum_j z_j \log o_j$$

With a one-hot target only one term survives: $H = -\log(\text{probability given to the correct class})$.

WORKED EXAMPLE

Logits $s = (2, 1, 0)$ for classes A, B, C; the correct class is A.

- $e^2 = 7.389$, $e^1 = 2.718$, $e^0 = 1$; sum $= 11.107$.
- Softmax: $o = (0.665,\ 0.245,\ 0.090)$.
- Loss: $H = -\ln 0.665 \approx 0.408$.
- Gradient (below): $o - z = (-0.335,\ 0.245,\ 0.090)$. It pushes A's score up and the others down.

For comparison, a probability of 0.5 on the correct class costs $-\ln 0.5 \approx 0.693$, 0.95 costs 0.051, and 0.01 costs 4.6. Confident mistakes are punished hard.

Deriving the gradient

First the softmax derivative. For $j = k$: $\partial o_j / \partial s_j = o_j(1 - o_j)$; for $j \neq k$: $\partial o_j / \partial s_k = -o_j o_k$. Both cases: $\partial o_j / \partial s_k = o_j(\delta_{jk} - o_k)$. Then:

$$\frac{\partial H}{\partial s_k} = -\sum_j \frac{z_j}{o_j} o_j(\delta_{jk} - o_k) = -z_k + o_k \sum_j z_j = o_k - z_k$$

using $\sum_j z_j = 1$. So

$$\boxed{\frac{\partial H}{\partial s_j} = o_j - z_j}$$

Predicted probability minus target. The softmax derivative is already folded in; there is no extra activation-derivative factor. Sigmoid with binary cross-entropy gives the same result, $\hat{y} - y$.

Numerical stability

e^s overflows for large logits. Subtract the largest logit first: $o_j = e^{s_j - \max s} / \sum_k e^{s_k - \max s}$. The common factor $e^{-\max s}$ cancels, so the probabilities are unchanged: $(2, 1, 0)$ and $(0, -1, -2)$ give the same softmax.

Common doubts

FIRST PASS **Why is cross-entropy better than MSE for classification?**

Two reasons. It punishes confident wrong answers heavily ($-\ln 0.01 \approx 4.6$), and with a softmax or sigmoid output its gradient is simply $o - z$, so it doesn't vanish when the output unit saturates. MSE through a sigmoid has an extra $\hat{y}(1 - \hat{y})$ factor that goes to zero exactly when the model is confidently wrong.

GOING DEEPER **Where did the $1/o_j$ go in the derivation?**

$\partial(-z_j \log o_j)/\partial o_j = -z_j/o_j$, and the softmax derivative contributes a factor o_j, so they cancel. That cancellation is why the softmax–cross-entropy pair is so convenient, and why frameworks compute them together.

Cheatsheet

Softmax and cross-entropy

$o_j = e^{s_j} / \sum_k e^{s_k}$; $H = -\sum_j z_j \log o_j$; gradient $\partial H / \partial s_j = o_j - z_j$.

Stable softmax

Subtract $\max_k s_k$ from every logit before exponentiating. Same probabilities, no overflow.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/losses-and-outputs/softmax-and-cross-entropy

Binary, Multi-class and Logits in PyTorch

Which output layer and loss go with which task, binary cross-entropy worked out, why accuracy can't be the training loss, the best accuracy any classifier can reach, and the logits rule that trips people up in PyTorch.

Output layer and loss by task

TASK	OUTPUT LAYER	LOSS	PYTORCH
Regression	Linear	MSE	`nn.MSELoss`
Binary classification	One sigmoid unit	Binary cross-entropy	`nn.BCEWithLogitsLoss`
Multi-class	Softmax over K classes	Categorical cross-entropy	`nn.CrossEntropyLoss`

Binary cross-entropy

$$\ell(\hat{y}, y) = -\big[y \log \hat{y} + (1-y)\log(1-\hat{y})\big], \qquad \hat{y} = \sigma(z) = P(y=1 \mid x)$$

For $y = 1$ it's $-\log \hat{y}$; for $y = 0$, $-\log(1-\hat{y})$. With $y = 1$: $\hat{y} = 0.9$ costs 0.105 and $\hat{y} = 0.1$ costs 2.303. A threshold (0.5 by default) turns $\hat{y}$ into a class.

The logits rule in PyTorch

RETURN LOGITS, NOT PROBABILITIES

`nn.CrossEntropyLoss` and `nn.BCEWithLogitsLoss` take **raw logits** and apply log-softmax or sigmoid internally, in a numerically stable way. If your model also applies a softmax at the end, the loss applies it a second time: training still runs but the gradients are wrong and learning is poor. Leave the final layer linear; apply softmax only when you want probabilities for reporting.

Why not train on accuracy?

Accuracy only changes when a prediction crosses the threshold, so as a function of the weights it's flat almost everywhere: its gradient is zero. Train on cross-entropy, which changes smoothly, and **report** accuracy.

The best any classifier can do

WORKED EXAMPLE: OVERLAPPING CLASSES

Class A is uniform on the square $[0, 5]^2$; class B uniform on $[0, 10]^2$; equal priors. Inside the small square the densities are $1/25$ (A) and $1/100$ (B), so $P(A \mid x) = \frac{1/25}{1/25+1/100} = 0.8$. Outside it, only B is possible.

- A point lands in the small square with probability $0.5 \cdot 1 + 0.5 \cdot 0.25 = 0.625$. The best guess there is A, right 80% of the time.
- Outside (probability 0.375) the best guess is B, right every time.
- Best possible accuracy: $0.625 \times 0.8 + 0.375 \times 1 = 0.875$.
- Lowest possible cross-entropy: $0.625 \times (-0.8 \ln 0.8 - 0.2 \ln 0.2) \approx 0.313$ nats.

No network, however large, can beat this **Bayes-optimal** limit, because the classes genuinely overlap.

Common doubts

FIRST PASS **Do I need a sigmoid layer before BCEWithLogitsLoss?**

No, and you shouldn't add one. As the name says, it expects logits and applies the sigmoid itself. If you want probabilities at prediction time, apply `torch.sigmoid` to the logits then.

GOING DEEPER **Why can't my model reach 100% accuracy on overlapping classes?**

If the same input can belong to either class, any rule that picks one class for that input must be wrong some of the time. The Bayes-optimal classifier picks the more probable class everywhere, and its accuracy (0.875 in the example) is a ceiling for every model on that data distribution.

Cheatsheet

Task → output → loss

Regression: linear + MSE. Binary: sigmoid + BCE (BCEWithLogitsLoss). Multi-class: softmax + CE (CrossEntropyLoss).

Logits rule

CrossEntropyLoss / BCEWithLogitsLoss take raw logits. Don't put softmax/sigmoid in the model when using them.

Accuracy as a loss

Accuracy is piecewise constant: zero gradient. Train on cross-entropy; report accuracy.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/losses-and-outputs/binary-multiclass-and-logits

Section 7

Backpropagation

How the chain rule gives every gradient, derived by hand for a single neuron and a hidden layer, then in matrix form, with mini-batches and gradient checking.

7.1 The Chain Rule and Computational Graphs · 10 min

7.2 Backpropagation for a Single Sigmoid Neuron · 9 min

7.3 One Hidden Layer: Deriving Every Gradient · 13 min

7.4 Backpropagation in Matrix Form · 9 min

7.5 Multiple Paths and Deeper Networks · 8 min

7.6 Mini-batches and Gradient Checking · 7 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

6

BUILDING AND TRAINING A NETWORK · STEP 6

Compute the gradients

How does each weight learn its share of the blame?

Lessons here: 7.1, 7.2, 7.3, 7.4, 7.5 · feeds into Update the weights

7

BUILDING AND TRAINING A NETWORK · STEP 7

Update the weights

How do I turn gradients into a better network?

Lessons here: 7.6 · feeds into Evaluate honestly

The Chain Rule and Computational Graphs

Backpropagation is the chain rule applied backwards through the network, reusing every intermediate result. The chain rule on a small graph, with numbers, and the two rules: multiply along a path, add across paths.

The chain rule

If E depends on y and y depends on x:

$$\frac{dE}{dx} = \frac{dE}{dy}\frac{dy}{dx}$$

For a longer chain $x \to s \to \hat{y} \to E$, multiply the **local derivatives** along the path: $\frac{dE}{dx} = \frac{dE}{d\hat{y}}\frac{d\hat{y}}{ds}\frac{ds}{dx}$.

Two rules for graphs

- **Multiply** local derivatives along a path.
- **Add** the contributions of all paths when a value feeds several downstream nodes.

WORKED EXAMPLE: ONE WEIGHT, WITH NUMBERS

$s = wx$, $\hat{y} = \sigma(s)$, $E = (\hat{y} - t)^2$, with $x = 1$, $w = 0.5$, $t = 1$.

- Forward: $s = 0.5$, $\hat{y} = \sigma(0.5) = 0.6225$, $E = (0.6225 - 1)^2 = 0.1425$.
- Local derivatives: $\frac{dE}{d\hat{y}} = 2(\hat{y} - t) = -0.755$; $\frac{d\hat{y}}{ds} = \hat{y}(1 - \hat{y}) = 0.2350$; $\frac{ds}{dw} = x = 1$.
- Multiply: $\frac{dE}{dw} = -0.755 \times 0.2350 \times 1 \approx -0.177$.

Negative, so increasing w reduces the error, as expected when the output is below the target.

Why "backwards" and why it's efficient

Backpropagation is **reverse-mode automatic differentiation**. One forward pass computes and stores every intermediate value; one backward pass, from the loss towards the inputs, computes the derivative of the loss with respect to **every** parameter, reusing shared terms instead of recomputing them. The cost is about the same as one more forward pass, however many parameters there are. That reuse is what makes training large networks feasible.

One training iteration, end to end

Backpropagation is one step of a loop. Every term from the earlier sections has a single job in it. For n training examples:

- **Loss** $\ell(\hat{y}_i, y_i)$: the error on one example.
- **Empirical risk** $\hat{R}(\theta) = \frac{1}{n}\sum_i \ell\big(f_\theta(x_i), y_i\big)$: the average loss on the training data, our stand-in for the error on all future data.

- **Objective** $J(\theta) = \hat{R}(\theta) + \lambda\,\Omega(\theta)$: what training actually minimises. With $\lambda = 0$ it is just the empirical risk.
- **ERM** (empirical risk minimisation) is the principle: pick the θ that minimises $\hat{R}$. With the penalty, it is **regularised ERM**.
- **Optimisation** is how we minimise J: gradient descent or a variant. It needs $\nabla_\theta J$, and **backpropagation** is how we get it.

STEP	WHAT HAPPENS	RESULT
1. Forward pass	Inputs flow through the layers; every activation is stored for later	Predictions $\hat{y}$
2. Loss and objective	Compare $\hat{y}$ with y, average over the batch, add any penalty	J, one number
3. Backward pass	Chain rule from J back through the stored values to every weight	$\nabla_\theta J$, one number per weight
4. Optimiser step	$\theta \leftarrow \theta - \eta\,\nabla_\theta J$ (or momentum, Adam)	New weights
5. Repeat	Next mini-batch; after each epoch, check the validation loss	A trained network

So the objective is calculated at the end of every forward pass, the gradient straight after it in the backward pass, and the weights change only in step 4, after the backward pass has finished. With mini-batches, J and its gradient are computed on the batch, so both are estimates of the full-data values. In PyTorch: `optimizer.zero_grad()` clears the old gradients, `loss = loss_fn(model(x), y)` is steps 1 and 2, `loss.backward()` is step 3, and `optimizer.step()` is step 4.

Click through one full iteration on the 2–3–1 network from lesson 3.1: forward, loss, backward, update, and forward again.

WHY COMPUTE THE OBJECTIVE AGAIN AFTER THE UPDATE?

A gradient is only valid at the point where it was computed. Once the weights change, the old J and the old gradient are out of date. The next forward pass measures how good the **new** weights are and gives the gradient at the new point.

Continue the worked example above. The gradient was $\frac{dE}{dw} \approx -0.177$. Take one step with $\eta = 1$: $w = 0.5 + 0.177 = 0.677$. Forward again: $\hat{y} = \sigma(0.677) = 0.6631$ and $E = (0.6631 - 1)^2 = 0.1135$, down from 0.1425. The error fell, so the step helped. If E had gone up, that would be a sign that η is too large.

Common doubts

FIRST PASS **Is backpropagation the same as gradient descent?**

No. **Backpropagation** computes the gradient. **Gradient descent** uses it to update the weights. One computes the direction; the other takes the step. PyTorch mirrors this: `loss.backward()` is backpropagation and `optimizer.step()` is the update.

FIRST PASS **Is ERM an optimisation algorithm? How is it different from gradient descent?**

No. **ERM** is a principle: it says **what** to minimise, the average training loss (plus a penalty if regularised). **Gradient descent** is an algorithm: it says **how** to minimise it, step by step. ERM defines the target; gradient descent, Adam or a closed-form formula like OLS's gets you there.

FIRST PASS **Do both ordinary and regularised ERM update the weights?**

Yes. Both are minimised with the same loop: forward pass, objective, gradient, update. The only difference is the objective. With an L2 penalty, the gradient gains an extra term λW, so every update also shrinks the weights a little: weight decay (lesson 9.1).

FIRST PASS **Why does a neural network have so many gradients?**

Because it has many parameters, and the gradient has one entry per parameter: how much the objective changes if you nudge **that one** weight or bias. The 2–3–1 network in lesson 3.1 has 13 parameters, so 13 partial derivatives; a network with a million weights has a million. Backpropagation computes all of them in one backward pass, and each one updates its own parameter.

FIRST PASS **How do I say ∂L/∂w in words, and how do I write the different gradients?**

$\partial L/\partial w$ is **how much the loss changes when this weight changes by a tiny amount**. If $\partial L/\partial w = -1.5$, raising w by 0.01 lowers the loss by about 0.015, so gradient descent raises w.

SYMBOL	READ IT AS
$\partial L/\partial w$	Gradient for one weight
$\partial L/\partial W$ or $\nabla_W L$	A matrix of gradients, the same shape as W
$\delta = \partial L/\partial z$	The error signal at a neuron's pre-activation
$\partial L/\partial W^{(2)}$	Gradient for the weights of layer 2 (the superscript is the layer)

Cheatsheet

Chain rule on graphs

Multiply local derivatives along a path; add contributions over all paths from a node to the loss.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/chain-rule-and-computational-graphs

Backpropagation for a Single Sigmoid Neuron

The derivation you are most likely to be asked for: the gradient of mean squared error for one sigmoid neuron over a batch of examples, step by step, with a numerical example and the matrix form.

Set-up

N examples $x^{(i)}$ with targets y_i. For each example:

$$s_i = w^\top x^{(i)} + b, \qquad \hat{y}_i = \sigma(s_i), \qquad E = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

The derivation

Step 1, the error signal. Define $\delta_i = \partial E / \partial s_i$. By the chain rule through $\hat{y}_i$:

$$\delta_i = \frac{\partial E}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial s_i} = \frac{2}{N}(\hat{y}_i - y_i)\cdot \hat{y}_i(1-\hat{y}_i)$$

The first factor comes from the squared error, the second is the sigmoid derivative $\sigma' = \sigma(1-\sigma)$.

Step 2, the weights. s_i depends on w through $w^\top x^{(i)}$, so $\partial s_i / \partial w = x^{(i)}$. Every example contributes, so sum:

$$\frac{\partial E}{\partial w} = \sum_i \delta_i \, x^{(i)} = X^\top \delta$$

Step 3, the bias. $\partial s_i / \partial b = 1$:

$$\frac{\partial E}{\partial b} = \sum_i \delta_i$$

WORKED EXAMPLE: ONE EXAMPLE, ONE UPDATE

$x = (1, 2)$, $w = (0.1, -0.2)$, $b = 0$, target $y = 1$, $N = 1$.

- Forward: $s = 0.1 - 0.4 = -0.3$; $\hat{y} = \sigma(-0.3) = 0.4256$.
- Error signal: $\delta = 2(0.4256 - 1) \times 0.4256 \times 0.5744 = -1.1489 \times 0.2445 = -0.2808$.
- Gradients: $\partial E/\partial w = \delta x = (-0.2808, -0.5617)$; $\partial E/\partial b = -0.2808$.
- Update with $\eta = 1$: $w = (0.381, 0.362)$, $b = 0.281$.
- Check: new $s = 0.381 + 0.724 + 0.281 = 1.385$, $\hat{y} = 0.80$. Much closer to the target 1.

In code (one example per row)

```
s = X @ w + b
y_hat = 1 / (1 + np.exp(-s))
delta = 2.0 / N * (y_hat - y) * y_hat * (1 - y_hat)
grad_w = X.T @ delta
grad_b = delta.sum()
```

WHAT EXAMINERS LOOK FOR

Name each chain-rule factor: the loss derivative $\frac{2}{N}(\hat{y} - y)$, the activation derivative $\hat{y}(1 - \hat{y})$, and the input x (or 1 for the bias). Then sum over examples.

Common doubts

FIRST PASS **Why is there a sum over examples in the weight gradient?**

Because the loss E is an average over all N examples, and every example's prediction depends on the same w. The derivative of a sum is the sum of the derivatives, so each example adds its own $\delta_i x^{(i)}$. The $1/N$ is already inside δ_i.

GOING DEEPER **Why does the loss never reach exactly zero for a sigmoid neuron on AND?**

$\sigma(s)$ only reaches 0 or 1 as $s \to \pm\infty$, so hitting the targets exactly would need infinite weights. Training keeps growing the weights and the loss keeps shrinking towards 0 without reaching it. Thresholding at 0.5 gives exact 0/1 answers for reporting accuracy, but a step function can't be trained.

Cheatsheet

Single sigmoid neuron (MSE)

$\delta_i = \frac{2}{N}(\hat{y}_i - y_i)\hat{y}_i(1 - \hat{y}_i)$; $\partial E/\partial w = X^\top \delta$; $\partial E/\partial b = \sum_i \delta_i$.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/backpropagation-for-a-single-neuron

One Hidden Layer: Deriving Every Gradient

The classic exam derivation: a network with one sigmoid hidden layer and a linear output, trained with squared error. Output-layer gradients, the hidden error signal, input-layer gradients, and a full numerical example.

The network

Inputs $x_1, \dots, x_m$; n sigmoid hidden units; one linear output; loss $L = \frac{1}{2}(y - t)^2$ for target t.

$$z_j^{(1)} = \sum_i w_{ji}^{(1)} x_i + b_j^{(1)}, \qquad a_j^{(1)} = \sigma\big(z_j^{(1)}\big), \qquad y = \sum_j w_j^{(2)} a_j^{(1)} + b^{(2)}$$

Step 1: the output error

$$\delta^{(2)} = \frac{\partial L}{\partial y} = y - t$$

(The $\frac{1}{2}$ cancels the 2 from the square. The output is linear, so there is no activation-derivative factor.)

Step 2: output-layer gradients

$$\frac{\partial L}{\partial w_j^{(2)}} = \delta^{(2)}\, a_j^{(1)}, \qquad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}$$

Step 3: the hidden error

Hidden unit j affects the loss only through y. Chain rule:

$$\delta_j^{(1)} = \frac{\partial L}{\partial z_j^{(1)}} = \underbrace{\frac{\partial L}{\partial y}}_{\delta^{(2)}} \underbrace{\frac{\partial y}{\partial a_j^{(1)}}}_{w_j^{(2)}} \underbrace{\frac{\partial a_j^{(1)}}{\partial z_j^{(1)}}}_{a_j^{(1)}(1-a_j^{(1)})} = \delta^{(2)}\, w_j^{(2)}\, a_j^{(1)}\big(1 - a_j^{(1)}\big)$$

Why each factor: the output weight $w_j^{(2)}$ scales how much a change in a_j moves y; the sigmoid derivative scales how much a change in z_j moves a_j.

Step 4: input-layer gradients

$$\frac{\partial L}{\partial w_{ji}^{(1)}} = \delta_j^{(1)}\, x_i = (y - t)\, w_j^{(2)}\, a_j^{(1)}\big(1 - a_j^{(1)}\big)\, x_i, \qquad \frac{\partial L}{\partial b_j^{(1)}} = \delta_j^{(1)}$$

The pattern in both layers: **gradient of a weight = error signal at its output end × the activation at its input end.**

WORKED EXAMPLE

$x = (1, 1)$, target $t = 1$. Hidden unit 1: weights $(0.5, 0.5)$, bias 0. Hidden unit 2: weights $(-0.5, 0.5)$, bias 0. Output weights $w^{(2)} = (1, -1)$, bias $b^{(2)} = 0.5$.

- Forward: $z^{(1)} = (1, 0)$, $a^{(1)} = (\sigma(1), \sigma(0)) = (0.7311, 0.5)$; $y = 0.7311 - 0.5 + 0.5 = 0.7311$; $L = \frac{1}{2}(0.2689)^2 = 0.0362$.
- Output error: $\delta^{(2)} = 0.7311 - 1 = -0.2689$.
- Output gradients: $\partial L/\partial w^{(2)} = (-0.2689 \times 0.7311,\ -0.2689 \times 0.5) = (-0.1966, -0.1345)$; $\partial L/\partial b^{(2)} = -0.2689$.
- Hidden errors: $\delta_1^{(1)} = -0.2689 \times 1 \times 0.7311 \times 0.2689 = -0.0529$; $\delta_2^{(1)} = -0.2689 \times (-1) \times 0.25 = +0.0672$.
- Input gradients ($x_i = 1$): unit 1's weights both -0.0529; unit 2's both $+0.0672$; biases the same.

Unit 2's gradient is positive because its output weight is negative: making a_2 smaller raises y towards the target.

Click through the same example step by step, with each neuron drawn as its sum and its activation so you can see where each error signal sits:

Common doubts

FIRST PASS **Why does the hidden error contain the output weight?**

Because a hidden unit's only influence on the loss is through the output, and its effect on the output is multiplied by the weight connecting them. A hidden unit wired to the output with weight 0 can't change the loss at all, so its error signal is 0.

GOING DEEPER **What changes with several outputs or a sigmoid output?**

With several outputs, a hidden unit affects each one, so its error is a **sum** over outputs: $\delta_j^{(1)} = \left(\sum_k \delta_k^{(2)} w_{kj}^{(2)}\right) a_j(1 - a_j)$ (lesson 7.5). With a sigmoid output and squared error, the output error gains the factor $y(1-y)$: $\delta^{(2)} = (y - t)\, y(1 - y)$. The rest of the derivation is unchanged.

FIRST PASS **Is the derivative with respect to the bias always 1?**

Only the local one. In $z = wx + b$, raising b by 0.1 raises z by exactly 0.1, so $\partial z/\partial b = 1$ always. But the gradient you update with is the **loss** gradient, and it goes through the chain rule:

GRADIENT	CHAIN RULE	RESULT
$\partial L/\partial w$	$\delta \cdot \partial z/\partial w = \delta \cdot x$	δ × input
$\partial L/\partial b$	$\delta \cdot \partial z/\partial b = \delta \cdot 1$	just δ

Example: $\delta = -0.5$ and $x = 3$ give $\partial L/\partial w = -1.5$ and $\partial L/\partial b = -0.5$, not 1. That's why a bias behaves like a weight on an input that is always 1.

Cheatsheet

One hidden layer (sigmoid hidden, linear out, ½ squared error)

$\delta^{(2)} = y - t$; $\partial w_j^{(2)} = \delta^{(2)} a_j$; $\delta_j^{(1)} = \delta^{(2)} w_j^{(2)} a_j (1 - a_j)$; $\partial w_{ji}^{(1)} = \delta_j^{(1)} x_i$.

The pattern

Weight gradient = (error signal at its output end) × (activation at its input end). Bias gradient = error signal.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/one-hidden-layer-every-gradient

Backpropagation in Matrix Form

The same algorithm for a whole batch at once: four steps, the shape of every matrix, the element-wise product, how the output error changes with the loss, and a complete NumPy implementation.

Forward pass for a batch

One example per row of X ($N \times d$), h sigmoid hidden units, k outputs:

$$S^{(1)} = XW^{(1)} + b^{(1)}, \quad H = \sigma\big(S^{(1)}\big), \quad S^{(2)} = HW^{(2)} + b^{(2)}, \quad \hat{Y} = \sigma\big(S^{(2)}\big)$$

The four steps

1. **Output error** $\delta^{(2)} = \partial E / \partial S^{(2)}$. For sigmoid output + MSE: $\delta^{(2)} = \frac{2}{N}(\hat{Y} - Y) \odot \hat{Y} \odot (1 - \hat{Y})$.
2. **Output-layer gradients:** $\dfrac{\partial E}{\partial W^{(2)}} = H^{\top}\delta^{(2)}$, and $\dfrac{\partial E}{\partial b^{(2)}}$ = the column sums of $\delta^{(2)}$.
3. **Propagate back:** $\delta^{(1)} = \big(\delta^{(2)} W^{(2)\top}\big) \odot H \odot (1 - H)$.
4. **Hidden-layer gradients:** $\dfrac{\partial E}{\partial W^{(1)}} = X^{\top}\delta^{(1)}$, and $\dfrac{\partial E}{\partial b^{(1)}}$ = the column sums of $\delta^{(1)}$.

$\odot$ is **element-wise** multiplication. The general rule: **weight gradient = (layer input)ᵀ × (error at the layer's output)**, and the error travels backwards through the transposed weights and the activation derivative.

Check the shapes

QUANTITY	SHAPE	QUANTITY	SHAPE
X	$N \times d$	$\delta^{(2)}$	$N \times k$
$W^{(1)}$	$d \times h$	$H^{\top}\delta^{(2)}$	$h \times k$, matches $W^{(2)}$ ✓
H	$N \times h$	$\delta^{(2)} W^{(2)\top}$	$N \times h$, matches H ✓
$W^{(2)}$	$h \times k$	$X^{\top}\delta^{(1)}$	$d \times h$, matches $W^{(1)}$ ✓

A gradient always has the same shape as the thing it's the gradient of. If a product doesn't fit, a transpose is missing.

Changing the loss changes only step 1

OUTPUT AND LOSS	OUTPUT ERROR $\Delta^{(2)}$
Sigmoid + MSE	$\frac{2}{N}(\hat{Y}-Y)\odot\hat{Y}\odot(1-\hat{Y})$
Linear + MSE	$\frac{2}{N}(\hat{Y}-Y)$
Softmax + cross-entropy	$(O-Z)/N$
Linear + MAE	$\operatorname{sign}(\hat{Y}-Y)/N$

In NumPy

```
def forward(X, p):
    H = sigmoid(X @ p["W1"] + p["b1"])
    y_hat = sigmoid(H @ p["W2"] + p["b2"])
    return y_hat, H

def backward(X, y, p):          # MSE, sigmoid hidden and output
    y_hat, H = forward(X, p)
    N = len(y)
    d2 = 2.0 / N * (y_hat - y) * y_hat * (1 - y_hat)
    d1 = (d2 @ p["W2"].T) * H * (1 - H)
    return {"W2": H.T @ d2, "b2": d2.sum(0),
            "W1": X.T @ d1, "b1": d1.sum(0)}
```

Common doubts

FIRST PASS **Why do the biases use a sum over rows?**

Every example uses the same bias, and $\partial S/\partial b = 1$ for each one, so the bias gradient is each column of δ added up over the N examples. In code that's `d.sum(0)`.

Cheatsheet

Matrix backprop (4 steps)

$\delta^{(2)}$ from the loss; $\partial W^{(2)} = H^{\top}\delta^{(2)}$; $\delta^{(1)} = (\delta^{(2)}W^{(2)\top})\odot f'(S^{(1)})$; $\partial W^{(1)} = X^{\top}\delta^{(1)}$. Biases: column sums.

Output error by loss

Sigmoid+MSE: $\frac{2}{N}(\hat{Y}-Y)\odot\hat{Y}\odot(1-\hat{Y})$. Linear+MSE: $\frac{2}{N}(\hat{Y}-Y)$. Softmax+CE: $(O-Z)/N$.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/backpropagation-in-matrix-form

Multiple Paths and Deeper Networks

When a hidden unit feeds several outputs, its gradient is a sum over paths. A two-hidden-layer, two-output example, the general backward recursion, and how it explains vanishing gradients.

A network with fan-out

Two hidden layers (outputs v, then z) and two outputs y_1, y_2. A bar marks a pre-activation, so $\bar{y}_1 = w_{3,11} z_1 + w_{3,21} z_2 + b_{3,1}$ and $y_1 = f(\bar{y}_1)$. The cost for one example is

$$J = \tfrac{1}{2}\big(y_1 - y_1^{(i)}\big)^2 + \tfrac{1}{2}\big(y_2 - y_2^{(i)}\big)^2$$

Last layer: one path each

$$\frac{\partial J}{\partial w_{3,11}} = \big(y_1 - y_1^{(i)}\big)\, f'(\bar{y}_1)\, z_1, \qquad \frac{\partial J}{\partial b_{3,1}} = \big(y_1 - y_1^{(i)}\big)\, f'(\bar{y}_1)$$

Second layer: sum over both outputs

z_1 feeds **both** y_1 and y_2, so a weight into z_1 affects the cost along two paths, and the contributions add:

$$\frac{\partial J}{\partial w_{2,11}} = \Big[\big(y_1 - y_1^{(i)}\big) f'(\bar{y}_1)\, w_{3,11} + \big(y_2 - y_2^{(i)}\big) f'(\bar{y}_2)\, w_{3,12}\Big]\, f'(\bar{z}_1)\, v_1$$

The first layer follows the same way and **reuses the bracketed terms** already computed. That reuse is what backpropagation is.

> COMMON ERROR
>
> Following only one path when a node fans out to several outputs. The derivative with respect to a shared node is the **sum** over every downstream path. Draw the network and trace each path.

The general recursion

For any depth, with $D_k = \operatorname{diag}\big(f'(z^{(k)})\big)$:

$$\delta^{(k)} = D_k\, W_{k+1}^{\top}\, \delta^{(k+1)} = D_k W_{k+1}^{\top} D_{k+1} W_{k+2}^{\top} \cdots \delta^{(L)}$$

The matrix product $W^{\top}\delta$ does the summing over paths automatically. Unrolled, it shows one activation-derivative factor per layer, which is exactly why deep sigmoid networks suffer vanishing gradients (Section 4).

Common doubts

FIRST PASS **Where does the sum appear in the matrix formula?**

In $W^\top \delta$: row j of $W^\top$ times δ is $\sum_k w_{kj}\,\delta_k$, the sum over every unit k that unit j feeds into, each weighted by its connecting weight. So the matrix form and the "sum over paths" rule are the same thing.

Cheatsheet

Fan-out

A node feeding several downstream nodes: its gradient is the sum over all paths. In matrix form, $W^\top \delta$ does the sum.

Backward recursion

$\delta^{(k)} = D_k W_{k+1}^\top \delta^{(k+1)}$, $D_k = \mathrm{diag}(f'(z^{(k)}))$.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/multiple-paths-and-deeper-networks

Mini-batches and Gradient Checking

How often to update: online, full-batch and mini-batch gradient descent compared. And how to test a hand-written gradient with central finite differences before trusting it.

When to update

METHOD	UPDATE AFTER	TRADE-OFF
Online / stochastic	Each single example	Many cheap, noisy updates
Full batch	The whole dataset	Stable average gradient; one update per full pass
Mini-batch	A small group (e.g. 32–256)	The usual compromise: frequent updates, reasonable noise, efficient on GPUs

One full pass through the training data is an **epoch**. With 10,000 examples and batches of 100, an epoch is 100 updates.

Gradient checking

Hand-derived gradients are easy to get subtly wrong. Check them against a numerical estimate, one parameter at a time:

$$\frac{\partial E}{\partial \theta_k} \approx \frac{E(\theta + \epsilon e_k) - E(\theta - \epsilon e_k)}{2\epsilon}, \qquad \epsilon \approx 10^{-6}$$

e_k nudges only parameter k. The **central** difference (both directions) is far more accurate than a one-sided one. Compare with the analytical gradient: a relative error above about 10^{-6} suggests a bug.

WORKED EXAMPLE

$E(\theta) = \theta^2$ at $\theta = 3$: $\frac{(3+10^{-6})^2 - (3-10^{-6})^2}{2\times 10^{-6}} = \frac{12\times 10^{-6}}{2\times 10^{-6}} = 6$, matching $E' = 2\theta = 6$.

- It costs two loss evaluations **per parameter**, so it's a test, never a training method.
- A network that trains is **not** proof that its gradients are right: in one lab, a sign error in a single bias still let the network learn XOR in about half the random starts.

Common doubts

FIRST PASS **Is a mini-batch gradient the "true" gradient?**

It's an estimate of it: the average gradient over a random sample of examples. On average it points the right way, with some noise. That noise is cheap to accept, because you get many more updates per epoch, and it can even help escape flat regions.

Cheatsheet

Online, batch, mini-batch

Online: update per example (noisy). Batch: per full dataset (stable, slow). Mini-batch: small groups, the usual choice. Epoch = one pass.

Gradient check

$\partial E/\partial \theta_k \approx [E(\theta + \epsilon e_k) - E(\theta - \epsilon e_k)]/2\epsilon$, $\epsilon \approx 10^{-6}$. A debug test, 2 evaluations per parameter.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/backpropagation/mini-batches-and-gradient-checking

Section 8

Generalisation and Data

Why fitting is not learning, how to split and prepare data, how to read learning curves, and a reliable training procedure.

8.1 Fitting Is Not Learning · 7 min

8.2 Training, Validation and Test Sets · 7 min

8.3 Underfitting, Overfitting and Learning Curves · 8 min

8.4 Preparing the Data · 7 min

8.5 Hyperparameters and a Training Recipe · 8 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

2

BUILDING AND TRAINING A NETWORK · STEP 2

Prepare the data

How do I split, scale and encode the data?

Lessons here: 8.2, 8.4 · feeds into Evaluate honestly, Choose the activations

7

BUILDING AND TRAINING A NETWORK · STEP 7

Update the weights

How do I turn gradients into a better network?

Lessons here: 8.5 · feeds into Evaluate honestly

8

BUILDING AND TRAINING A NETWORK · STEP 8

Evaluate honestly

Is it learning, or memorising?

Lessons here: 8.1, 8.3 · feeds into Regularise, Size the network

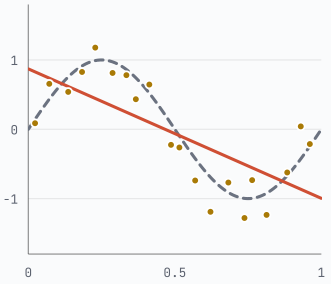
Fitting Is Not Learning

The goal of training is to predict new data well, not to match the training data. Polynomials of degree 1, 4 and 15 fitted to the same 20 noisy points show the difference.

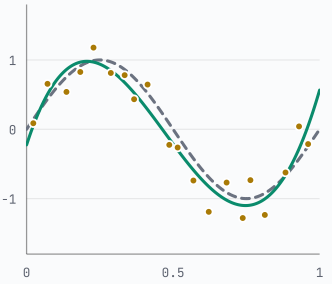
The real goal

Training minimises the loss on the training examples, but that loss is only a **proxy**. What we want is good predictions on **new** inputs from the same distribution: **generalisation**. A flexible enough network can fit any training set, even one whose labels were shuffled at random, without learning anything useful.

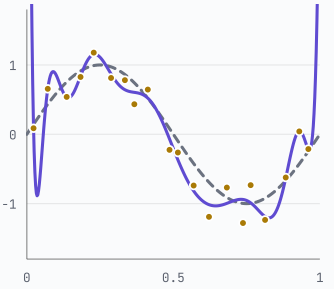
One dataset, three models



Degree 1: too simple. Misses the pattern (underfits).



Degree 4: follows the pattern, ignores the noise.



Degree 15: chases every point, wild in between (overfits).

Twenty noisy samples of $\sin 2\pi x$ (dashed). The degree-15 polynomial has the **lowest** error on these 20 points and the **worst** predictions between them. A lower training error does not mean a better model.

THE RULE

Whether a model generalises can only be measured on data it was **not** trained on. That's what validation and test sets are for (next lesson).

Common doubts

FIRST PASS **If the degree-15 fit passes near every point, why is it worse?**

Because the points contain noise. A model flexible enough to pass through every point is also fitting the noise, which won't be there in new data. Between the training points, nothing constrains it, so it swings wildly. The degree-4 fit captures the repeatable pattern and ignores the noise.

FIRST PASS **What are risk, empirical risk, ERM and regularised ERM, in plain words?**

New names for things you already do:

NAME	MEANING	YOU KNOW IT AS
Risk	Average loss on **all possible** data	What you care about, but can't compute
Empirical risk	Average loss on the **training** data	The training cost J
ERM	Choose the weights that minimise empirical risk	Ordinary training
Regularised ERM	Minimise empirical risk + a penalty	Training with L1 or L2 (lesson 9.1)

Example: training losses 0.2, 0.4 and 0.6 give empirical risk $J = 0.4$. With an L2 penalty of $\lambda \cdot \frac{1}{2} \sum w^2 = 0.1 \times 0.5 = 0.05$, regularised ERM minimises 0.45 instead.

Memory line: "Empirical risk = training loss. ERM = training. Regularised ERM = training with a penalty."

Cheatsheet

Generalisation

Training loss is a proxy. The goal is low error on unseen data from the same distribution; measure it on held-out data.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/generalisation/fitting-is-not-learning

Training, Validation and Test Sets

What each data split is for, typical split sizes, the forms of data leakage that quietly ruin an evaluation, and k-fold cross-validation when data is scarce.

Three disjoint sets

SET	USED FOR	NEVER
Training	Fitting the weights by gradient descent	Judging the final model
Validation	Every other choice: architecture, learning rate, regularisation, number of epochs	Computing training gradients
Test	One final estimate of generalisation, at the very end	Tuning anything: look at it once

Typical splits are 80/10/10 or 60/20/20. Shuffle before splitting so all three sets have the same distribution.

Data leakage

Leakage is any route by which information from the evaluation data gets into training, making scores look better than they will be in use:

- **The same object in two sets:** several images of one patient, or consecutive video frames, split across train and test. Split by patient or by clip instead.
- **Time:** for time series, train on the past and test on the future, never a random split.
- **Preprocessing statistics** (means, standard deviations, min/max) computed on all the data. Compute them on the training set only.
- **Tuning on the test set**, even informally by trying a few models and keeping the best test score.

Scarce data: k-fold cross-validation

Split the training data into k folds. Train k times, each time validating on a different fold, and average the k validation scores. Every example is used for validation exactly once.

Common doubts

FIRST PASS **Why do I need both a validation set and a test set?**

Because every decision you make using the validation score slightly fits the model to the validation set. After choosing among many options, the best validation score is optimistic. The test set, untouched until the end, gives an honest estimate. Reusing it for decisions turns it into another validation set.

GOING DEEPER **If my training set already contains every possible input, do I still need a test set?**

No. In the XOR lab all four possible inputs are in the training set, so there is nothing new to generalise to and a test set adds nothing. A test set matters whenever the model must work on inputs it hasn't seen, which is almost always.

FIRST PASS **To judge generalisation, do I compare training with validation or with test?**

With **validation** while you're still making choices (model size, λ, when to stop), and with **test** only once at the end, to report the final number. If you look at the test score while choosing, the choice adapts to the test set and the reported number becomes too optimistic.

Thumb rule: "Validation is for choosing, test is for reporting."

FIRST PASS **What information "wouldn't be available at prediction time"? Examples of leakage?**

Ask of every feature: "At the moment I make the prediction, would I really have this value?" If not, it leaks the answer.

TASK	LEAKY FEATURE	WHY IT LEAKS
Does the patient have pneumonia?	"Took antibiotics"	Given **after** the diagnosis
Will this customer cancel?	"Cancellation reason"	Filled in only once they've cancelled
Will this loan default?	"Sent to debt collector"	Happens only after a default
Tomorrow's price	Any data from tomorrow onwards	That's the future

The second kind is test data leaking into training: normalising with statistics from the whole dataset (compute them on the training set only), the same patient or photo in two sets (split by person), or a random split of a time series (train on the past, test on the future).

Thumb rule: "If it comes from the future or from the test set, it's leakage." The symptom is a score that looks too good, then falls apart in real use.

Cheatsheet

Three sets

Train: fit weights. Validation: every other choice, detect overfitting. Test: once, at the end. Typical 80/10/10 or 60/20/20.

Leakage

Same patient/clip in two sets; random split of time series; normalisation stats from all data; tuning on the test set.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/generalisation/training-validation-and-test-sets

Underfitting, Overfitting and Learning Curves

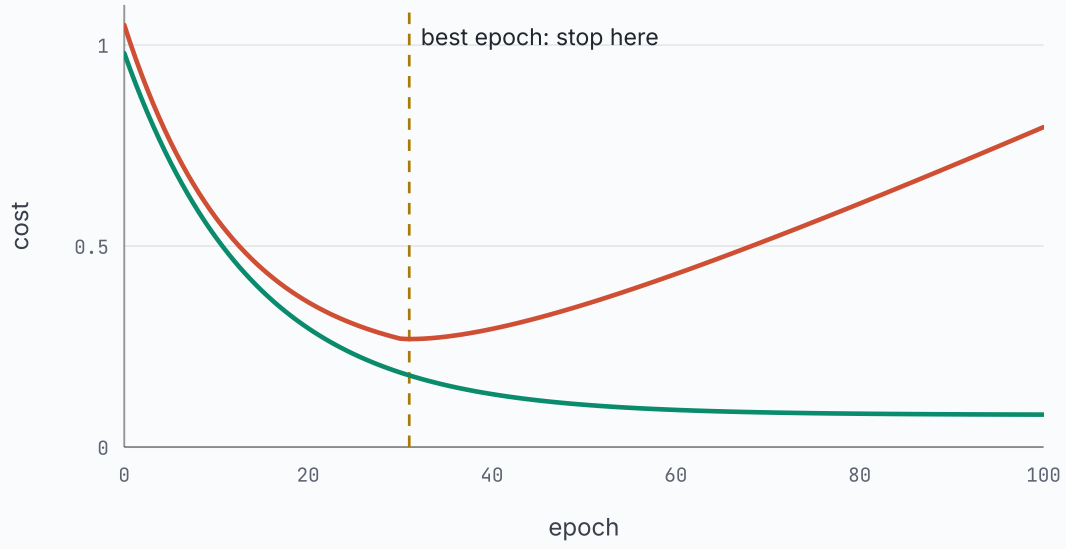
How to tell underfitting from overfitting with training and validation errors, how to read learning curves, and the bias–variance decomposition that explains both.

Diagnosing from the two errors

TRAINING ERROR	VALIDATION ERROR	DIAGNOSIS
High	High	**Underfitting**: too little capacity, or not trained enough
Low	Much higher	**Overfitting**: the gap measures it
Low	Low, close to training	Good fit

Training error alone never tells you: it keeps falling as capacity grows.

Learning curves



Training cost falls throughout. Validation cost reaches a minimum and then rises as the network starts fitting noise. Record both after every epoch.

The epoch with the lowest validation cost is where to stop: **early stopping** (Section 9).

Bias–variance decomposition

Model the target as $y = f(x) + \varepsilon$ with noise variance σ^2. For a model $\hat{y}(x)$ trained on a random training set:

$$\mathbb{E}\big[(\hat{y} - y)^2\big] = \underbrace{\big(\mathbb{E}[\hat{y}] - f(x)\big)^2}_{\text{bias}^2} + \underbrace{\mathbb{E}\big[(\hat{y} - \mathbb{E}[\hat{y}])^2\big]}_{\text{variance}} + \underbrace{\sigma^2}_{\text{noise}}$$

- **Bias:** systematic error of the model class. Low capacity (degree 1) has high bias: underfitting.
- **Variance:** how much the fitted model changes with the training sample. High capacity (degree 15) has high variance: overfitting.
- **Noise:** irreducible; no model can remove it.

More capacity trades bias for variance. Regularisation lowers variance at the cost of a little bias.

Common doubts

FIRST PASS **Why does the validation cost go up again if the network is still improving on training data?**

Early in training the network learns the general pattern, which helps both sets. Later it starts fitting quirks and noise specific to the training examples. That keeps lowering the training cost but makes predictions on new data worse, so validation cost rises.

FIRST PASS **What is the difference between convergence and overfitting?**

Convergence is about the optimiser: the training loss has stopped going down. **Overfitting** is about generalisation: the training loss keeps falling while the validation loss rises. They can happen together or apart. A network can overfit long before training converges, which is why you watch validation loss and stop early, and it can converge without overfitting if the model and data suit each other.

FIRST PASS **When do I check for bias and variance, and how?**

After every training run, by comparing training loss with validation loss, before you ever touch the test set.

TRAINING LOSS	GAP (VALIDATION - TRAINING)	PROBLEM	FIX
High	Small	High bias (underfitting)	Bigger model, train longer, lower λ
Low	**Large**	High variance (overfitting)	The fixes in the next doubt
Low	Small	Good fit	Stop and test

The trade-off appears when you fix things: a bigger model lowers bias but raises variance, and more regularisation does the opposite. Keep whichever setting gives the lowest **validation** loss.

Thumb rule: "Training loss shows bias. The gap shows variance."

FIRST PASS **What can I do to stop overfitting?**

Every fix falls into one of three groups:

GROUP	METHOD	HOW IT HELPS	LESSON
1. More or better data	More data	Harder to memorise many examples	8.1
	Data augmentation	"New" examples made by flipping, cropping or adding noise	9.4
2. Limit the model	Smaller model	Fewer neurons, less room to memorise noise	11.1
	L1 / L2	A penalty on big weights keeps the function smooth	9.1
	Dropout	Randomly switching off neurons stops co-adaptation	9.3
3. Limit the training	Early stopping	Stop when validation loss rises; keep the best epoch	9.2

All of them lower variance at the cost of a little bias, and their settings (λ, p, size, patience) are chosen on the validation set. Too much of any of them causes underfitting.

Memory line: "Feed it more, restrict the model, or stop it early."

Cheatsheet

Diagnose

Both errors high: underfitting. Train low, validation much higher: overfitting (the gap). Watch learning curves every epoch.

Bias–variance

$\mathbb{E}[(\hat{y} - y)^2] = \text{bias}^2 + \text{variance} + \sigma^2$. Capacity ↑: bias ↓, variance ↑.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/generalisation/underfitting-overfitting-and-learning-curves

Preparing the Data

Scaling features, handling missing values and categories, checking labels and dealing with unbalanced classes. Each with the reason it matters for a neural network.

Scale the features

If one feature lies in $[0.1, 0.7]$ and another in $[-10^4, 2 \times 10^4]$, the loss surface is badly conditioned (gradient descent zig-zags) and large inputs push sigmoid units into saturation. Rescale each feature j:

$$\text{min–max: } \tilde{x}_j = \frac{x_j - \min_j}{\max_j - \min_j} \in [0, 1], \qquad \text{standardisation: } \tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

Compute $\min_j, \max_j, \mu_j, \sigma_j$ on the **training set only** and apply the same transformation to validation, test and future inputs. Images: divide pixel values by 255. Standardise regression targets too.

> FROM THE LAB
>
> Inputs in $[0, 10]$ saturated the sigmoid hidden units and training stalled. Standardising them (training mean and standard deviation) fixed it. Always do this.

Other checks

ISSUE	WHAT TO DO
Missing features	Few affected examples: drop them. Otherwise impute with the training mean or median and add a "was missing" indicator
Categorical inputs	One-hot encode. An integer code (red = 1, green = 2, blue = 3) invents an order and spacing
Noisy labels	Labels are expensive and often wrong; inspect a sample by hand
Unbalanced classes	If 99% of examples are one class, predicting it always scores 99% accuracy. Use a confusion matrix, precision and recall; re-weight the loss or over-sample the rare class

Common doubts

FIRST PASS **Min–max or standardisation: which should I use?**

Both work for most networks. Standardisation is the common default and copes better with outliers (one extreme value squashes every other value under min–max). Min–max is natural when the range is known and fixed, such as pixel values from 0 to 255.

FIRST PASS

When do I use one-hot encoding?

When a feature is a category with **no natural order**: colour, city, blood type. Coding red = 1, green = 2, blue = 3 would tell the network that blue > green > red, an order that isn't true. One-hot gives each category its own 0/1 column, so all categories are equally far apart.

KIND OF CATEGORY	EXAMPLE	USE
No order	colour, city	One-hot
Real order	small < medium < large	Plain numbers 0, 1, 2
Thousands of categories	product IDs	Embeddings (lesson 11.2)

Class labels for multi-class outputs are one-hot too: cat = (1, 0, 0), compared with the softmax output by cross-entropy.

Thumb rule: "No order → one-hot. Real order → numbers."

FIRST PASS

What are precision, recall and F1, and how do I remember them?

- **Precision → Predicted:** of everything the model *predicted* positive, how many really are? $\frac{TP}{TP+FP}$
- **Recall → Real:** of everything that *really* is positive, how many did the model find? $\frac{TP}{TP+FN}$
- **F1** combines them: $F_1 = \frac{2PR}{P+R}$. It's low if **either** is low, so a model can't score well by being good at only one.

Example: there are 10 real positives; the model predicts 8 positives, of which 6 are right. Precision $= 6/8 = 0.75$, recall $= 6/10 = 0.6$, $F_1 = \frac{2\times0.75\times0.6}{1.35} \approx 0.67$.

Use them instead of accuracy when the classes are unbalanced.

Cheatsheet

Scaling

Min–max $(x - \min)/(\max - \min)$; standardise $(x - \mu)/\sigma$. Statistics from training data only. Pixels ÷ 255.

Data checks

Missing: drop or impute + indicator. Categories: one-hot. Labels: inspect. Imbalance: confusion matrix, precision/recall, re-weight.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/generalisation/preparing-the-data

Hyperparameters and a Training Recipe

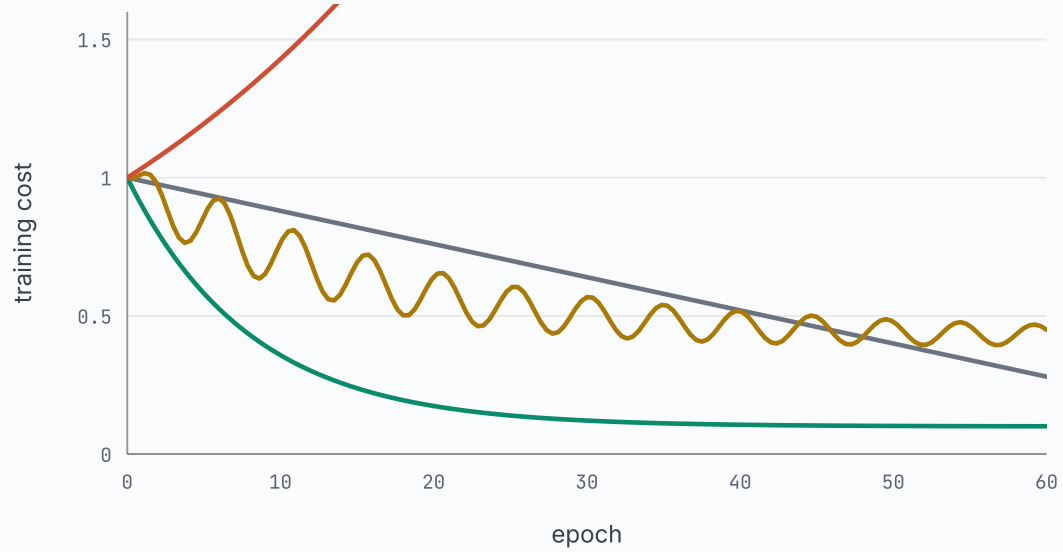
What hyperparameters are, how to search them, how to read a learning rate from the loss curve, and a seven-step procedure for training a network that works.

Hyperparameters

Everything not learned by gradient descent: the number of layers and units, activations, learning rate, batch size, number of epochs, regularisation strength λ, dropout rate and initialisation. Choose them on the **validation** set.

- **Grid search**: every combination; cost grows exponentially with the number of hyperparameters.
- **Random search**: usually better once you have more than two or three hyperparameters.
- **Bayesian optimisation**, or by hand guided by learning curves.
- Search η and λ on a **log scale** (0.001, 0.01, 0.1, ...).

Reading the learning rate from the loss curve



The learning rate matters most. A slowly falling cost: raise η. An oscillating or exploding cost: lower it.

A training recipe

1. **Data:** shuffle; split train/validation/test; normalise with training statistics; encode categories.
2. **Task:** pick the output layer and loss (Section 6).
3. **Sanity check:** a reasonably sized network should drive the training cost near zero on about 100 examples. If it can't, there's a bug in the code or data.

4. **Train** with mini-batch gradient descent or Adam, recording training and validation curves every epoch.
5. **Underfitting** (both errors high): bigger network, longer training, higher η; check normalisation and activations.
6. **Overfitting** (large gap): more data or augmentation, weight decay, dropout, early stopping, or a smaller network.
7. **Select** hyperparameters on validation, evaluate **once** on the test set, and report that number.

Common doubts

FIRST PASS **Why should a network be able to overfit 100 examples?**

Because memorising a tiny dataset is the easiest thing a network can do. If it can't even do that, the problem isn't generalisation: it's a bug (wrong labels, a broken gradient, a learning rate far off). It's a quick test that saves hours of tuning.

GOING DEEPER **What did the lab show matters most for training a small XOR network?**

Width had the largest effect: 2 hidden units converged in about 80% of random starts, 3 or more in all of them. The learning rate barely mattered within 0.3–3, but a huge one (300) pushed every unit into saturation on the first step and training stalled at loss 0.25. Very small initial weights (0.1) left all hidden units identical, on a plateau.

Cheatsheet

Training recipe

Split + normalise → output/loss → overfit 100 examples → train, log curves → underfit? bigger/longer → overfit? regularise → test once.

Learning-rate diagnosis

Slow decrease: η too small. Oscillating or diverging cost: η too large. Search η and λ on a log scale.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/generalisation/hyperparameters-and-a-training-recipe

Section 9

Regularisation

How L2 and L1 penalties, early stopping, dropout and data augmentation reduce overfitting.

9.1 L2 and L1 Penalties · 9 min

9.2 Early Stopping · 5 min

9.3 Dropout · 8 min

9.4 Data Augmentation · 5 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

9

BUILDING AND TRAINING A NETWORK · STEP 9

Regularise

How do I close the gap between training and validation?

Lessons here: 9.1, 9.2, 9.3, 9.4 · feeds into Prepare the data, Update the weights

L2 and L1 Penalties

Add a penalty on the size of the weights to the loss. L2 gives weight decay: every step shrinks the weights a little. L1 pushes small weights to exactly zero. The derivation and the difference.

What regularisation is

Regularisation is any change to learning meant to reduce the **generalisation** error rather than the training error. It's the answer to overfitting when you can't get more data. The main methods: penalty terms, early stopping, dropout and data augmentation.

Penalised objective

$$\min_{\theta} \; J(\theta) + \lambda\, \Omega(\theta), \qquad \lambda \geq 0$$

$$\Omega_2 = \tfrac{1}{2} \sum_{k} \|W_k\|_F^2 = \tfrac{1}{2} \sum_{k,i,j} \left(w_{ij}^{(k)}\right)^2, \qquad \Omega_1 = \sum_{k,i,j} \left|w_{ij}^{(k)}\right|$$

$\lambda = 0$ is the original problem; $\lambda \to \infty$ sends every weight to zero. Biases are not penalised.

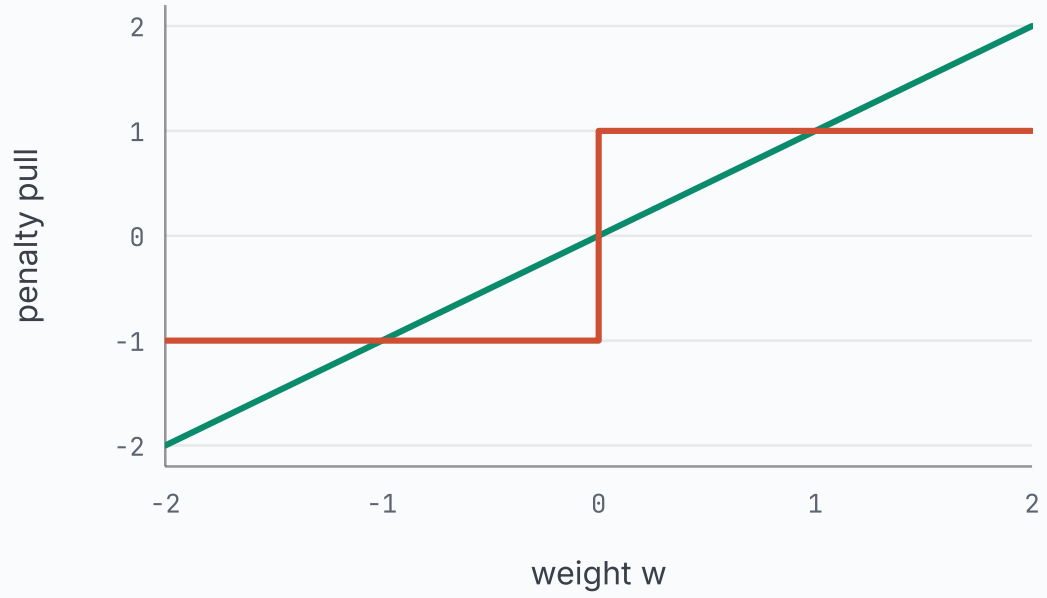
L2 is weight decay

$\partial \Omega_2 / \partial W = W$, so a gradient step becomes

$$W \leftarrow W - \eta \left(\frac{\partial J}{\partial W} + \lambda W \right) = (1 - \eta\lambda)\, W - \eta\, \frac{\partial J}{\partial W}$$

Every step first **shrinks** the weights by the factor $(1 - \eta\lambda)$, then takes the usual step. With $\eta = 0.1$ and $\lambda = 0.01$, that's ×0.999 per step. Weights the data don't need decay towards zero; weights the data support are kept. Typical λ: 10^{-5} to 10^{-2} on a log grid. In PyTorch, the optimiser's `weight_decay` argument does this.

Ll makes weights exactly zero



The gradient of each penalty (λ = 1). L2's pull shrinks with the weight, so small weights shrink but never reach 0. L1 pulls with the same force however small the weight, so it drives small weights all the way to 0.

$\partial\Omega_1/\partial w = \text{sign}(w)$: a constant-size pull towards zero. Small weights end up **exactly** zero, so connections are removed and unhelpful inputs ignored: **sparsity**, a form of feature selection.

	L1	L2
Penalty	Sum of absolute weights	Sum of squared weights
Effect	Exact zeros (sparse)	Many small weights, none exactly zero
Use	When you want sparsity	The usual choice in deep learning

Small weights give a function that changes slowly with the input, like the degree-4 curve rather than the degree-15 one in lesson 8.1.

Common doubts

FIRST PASS **Why aren't the biases penalised?**

A bias only shifts a unit's threshold; it doesn't make the function more wiggly or sensitive to the inputs, so it doesn't drive overfitting. Penalising it would just stop units from positioning their thresholds where the data needs them.

GOING DEEPER **Isn't a smaller network a simpler way to regularise?**

It is a form of regularisation, but in practice a large network with a penalty usually works better: λ is a single continuous knob that's easy to tune on validation data, while architecture choices are discrete and expensive to search.

GOING DEEPER **Is the penalised objective the same as regularised ERM?**

Yes. **Empirical risk minimisation** (ERM) chooses the parameters that minimise the average training loss, $\hat{R}(\theta) = \frac{1}{n}\sum_i \ell\big(f_\theta(x_i), y_i\big)$. **Regularised ERM** minimises $\hat{R}(\theta) + \lambda\,\Omega(\theta)$ instead: exactly the penalised objective above.

Ordinary ERM trusts the training data completely, so a flexible network can fit its noise. The penalty gives up a little training error in exchange for smaller weights, which generalise better. λ sets the trade-off and is chosen on the validation set. ERM is introduced in Empirical Risk and Population Risk in the machine learning course.

FIRST PASS **Why do we need both λ and Ω?**

They answer different questions. Ω **measures** how big the weights are; λ says **how much that matters**. Think of Ω as counting the crimes and λ as how seriously they're punished. The same Ω with $\lambda = 0$ is ignored, and with a huge λ dominates the loss. You choose λ on the validation set; too large and the model underfits.

FIRST PASS **Why add the penalty to the loss at all?**

Training only minimises what's in the loss. On its own, the loss is happy to use huge weights if they fit the training data a little better, noise included. Adding $\lambda\Omega$ makes "keep the weights small" part of the goal, so the gradient gets an extra pull towards zero, and the network only keeps a large weight when the data really pay for it.

FIRST PASS **Where does regularisation go in backpropagation? Does it change the forward pass?**

No. The forward pass, the error signals δ and the bias gradients are unchanged. Only the **weight gradients** get one extra term before the update:

	WEIGHT GRADIENT	UPDATE
None	$\partial J/\partial w$	$w - \eta\,\partial J/\partial w$
L1	$\partial J/\partial w + \lambda\,\mathrm{sign}(w)$	$w - \eta\lambda\,\mathrm{sign}(w) - \eta\,\partial J/\partial w$
L2	$\partial J/\partial w + \lambda w$	$(1 - \eta\lambda)\,w - \eta\,\partial J/\partial w$

WORKED EXAMPLE

$w = 0.5$, $\partial J/\partial w = -1.5$, $\lambda = 0.2$, $\eta = 0.1$.

- None: $0.5 - 0.1(-1.5) = 0.65$
- L1: gradient $-1.5 + 0.2 = -1.3$, so $w = 0.5 + 0.13 = 0.63$
- L2: gradient $-1.5 + 0.2 \times 0.5 = -1.4$, so $w = 0.5 + 0.14 = 0.64$

Both penalties hold the weight back a little from where the data alone would push it.

FIRST PASS **In L1, how do I know whether to add +λ or −λ?**

Follow the sign of the weight: $\mathrm{sign}(w)$ is $+1$ for a positive weight and -1 for a negative one. A positive weight gets $+\lambda$ in its gradient, and since the update **subtracts** the gradient, the weight goes down. A negative weight gets $-\lambda$ and goes up. Either way, L1 always pulls towards zero.

FIRST PASS **Why does L1 reach zero but L2 doesn't?**

L1 takes a **fixed** amount off each step; L2 takes a **percentage**. Start at $w = 0.5$ with a pull of 0.1 per step (ignoring the data):

- L1: $0.5 \to 0.4 \to 0.3 \to 0.2 \to 0.1 \to 0$. It gets there.
- L2 (×0.9 each step): $0.5 \to 0.45 \to 0.405 \to 0.36 \to \ldots$ It keeps shrinking but never reaches 0.

Like eating a cake: a fixed slice each day finishes it; 10% of what's left each day never does. For large weights ($|w| > 1$), L2's pull λw is the stronger one; it's near zero that L1 wins.

FIRST PASS **When do I use L1 and when L2? Can I use both?**

	L1	L2
Push	Fixed	Percentage
Effect	Exact zeros (sparsity)	Small weights, none zero
Use when	You suspect many inputs are useless and want them switched off (feature selection)	The default in deep learning

Yes, you can use both together; that's called **elastic net**.

Thumb rule: "L1 → zeros. L2 → small."

Cheatsheet

L2 weight decay

$J + \frac{\lambda}{2}\sum w^2 \to W \leftarrow (1-\eta\lambda)W - \eta\,\partial J/\partial W$. Shrinks weights, none exactly 0. Biases not penalised.

L1 vs L2

L1: pull $\lambda\,\mathrm{sign}(w)$, exact zeros, sparsity. L2: pull λw, small weights, the DL default.

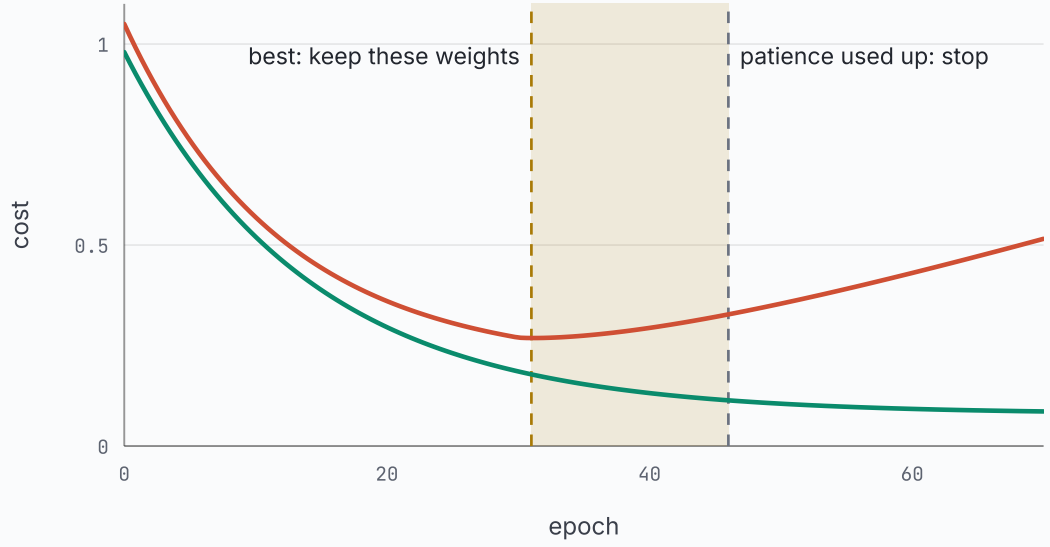
Online version with the interactive quiz: airesearchlabs.si/learn/dl/regularisation/l2-and-l1-penalties

Early Stopping

Watch the validation cost every epoch, keep the best weights, and stop when it stops improving. Why it regularises, and the patience setting.

The method

1. After every epoch, compute the validation cost.
2. Whenever it's the best so far, save a copy of the weights.
3. Stop when it hasn't improved for a set number of epochs, the **patience**.
4. Restore the saved best weights.



With a patience of 15 epochs, training stops at epoch 46 and the weights from epoch 31 are restored.

Why it regularises

Weights start small and grow as training continues. Stopping early limits how far they can move from their initial values, which limits how wiggly the learned function can become. For a linear model, early stopping is equivalent to L2 regularisation.

It costs one validation pass per epoch, it also chooses the number of epochs for you, and it's used in almost every training run.

Common doubts

FIRST PASS **Why wait for a patience period instead of stopping at the first increase?**

Validation cost is noisy, especially with mini-batches; it often ticks up for an epoch or two and then improves again. Patience avoids stopping on a blip. Because you restore the best weights, waiting a little longer costs only time.

Cheatsheet

Early stopping

Track validation cost each epoch, save the best weights, stop after "patience" epochs without improvement, restore the best. ≈ L2 for linear models.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/regularisation/early-stopping

Dropout

Randomly switch off hidden units during training. Inverted dropout as lectured: drop with probability p, scale the survivors by 1/(1−p), change nothing at test time. Why it prevents co-adaptation and how to use it in PyTorch.

Inverted dropout

Training: for each mini-batch, drop each hidden unit independently with probability p (set its output to 0). With a mask $m_i \sim \text{Bernoulli}(1-p)$, that is 1 with probability $1-p$:

$$\tilde{h} = \frac{1}{1-p}\, m \odot h$$

A new mask is drawn for every mini-batch. **Test:** nothing is dropped and nothing is rescaled; the full network is used.

The $1/(1-p)$ factor keeps the expected value of each activation unchanged: $\mathbb{E}[\tilde{h}_i] = \frac{1}{1-p}(1-p)\, h_i = h_i$. That's why no correction is needed at test time.

WORKED EXAMPLE

$p = 0.5$, activations $h = (2, 4, 1, 3)$, mask $m = (1, 0, 1, 0)$. Then $\tilde{h} = 2 \cdot (2, 0, 1, 0) = (4, 0, 2, 0)$. At test time the layer outputs $(2, 4, 1, 3)$ unchanged.

Why it works

- **No co-adaptation:** a unit can't rely on particular other units being present, so each must learn something useful on its own instead of jointly fitting the noise with a few partners.
- **An ensemble:** each mask picks out a sub-network, and all of them share weights. With n droppable units there are 2^n sub-networks; using the full network at test time approximately averages them, which lowers variance.

In practice

- Put `nn.Dropout(p)` **after** the activation.
- $p = 0.5$ for large fully connected layers, 0.1–0.2 elsewhere.
- Call `model.train()` while training and `model.eval()` before evaluating; forgetting `model.eval()` is a classic bug (dropout stays on at test time).
- Never apply dropout to the output layer.

TWO CONVENTIONS: USE THE LECTURE'S

Older sources define p as the probability of **keeping** a unit and scale the activations at **test** time instead. The lectured version defines p as the probability of **dropping** a unit and scales by $1/(1-p)$ during **training**, with no change at test time. Both give the same expected activations; state which you use.

Common doubts

FIRST PASS **Why not drop units at test time too?**

At test time you want the best, most stable prediction, which is the average over all the sub-networks. Using the full network (with training-time scaling already done) approximates that average in one pass. Random dropping at test time would make predictions noisy.

GOING DEEPER **Does PyTorch's nn.Dropout(p) use p as the drop or keep probability?**

The **drop** probability, with inverted scaling during training, exactly as lectured. `nn.Dropout(0.2)` zeroes 20% of activations in training mode and multiplies the rest by 1/0.8 = 1.25; in evaluation mode it does nothing.

FIRST PASS **What does "drop each unit with probability p" actually mean?**

For every hidden neuron, flip a biased coin **each mini-batch**. With probability p the neuron is switched off: its output is set to 0 for that mini-batch. With $p = 0.5$, about half the neurons are off, and a different half next time.

MINI-BATCH	NEURON 1	NEURON 2	NEURON 3
1	on	off	on
2	off	on	on
3	on	on	off

FIRST PASS **Why scale the survivors up by 1/(1 – p), and is that why dropout works?**

With $p = 0.5$, half the neurons are off, so the next layer would receive about half its usual input. Doubling the survivors ($1/(1-0.5) = 2$) keeps the total the same **on average**, so at test time, when every neuron is on, nothing needs changing.

The boost is only bookkeeping; it isn't why dropout reduces overfitting. That comes from the dropping: neurons can't rely on particular partners (no co-adaptation), and training many thinned networks acts like an ensemble.

Memory line: "Drop and boost in training, use everything at test."

FIRST PASS

Where does dropout go in backpropagation?

Unlike L1 and L2, dropout changes the **forward pass**: multiply each hidden output by its mask and by $1/(1-p)$. Backpropagation then follows automatically. A dropped neuron output 0, so its δ is 0 and the weights into and out of it get no update in that mini-batch. The survivors' gradients carry the same $1/(1-p)$ factor.

WORKED EXAMPLE

$p = 0.5$, $h = (0.6, 0.2, 1.0)$, mask $m = (1, 0, 1)$: $\tilde{h} = 2 \cdot (0.6, 0, 1.0) = (1.2, 0, 2.0)$. At test time the layer passes $(0.6, 0.2, 1.0)$ unchanged.

Cheatsheet

Inverted dropout

Train: drop each unit with prob. p, $\tilde{h} = \frac{1}{1-p} m \odot h$, new mask per batch. Test: no dropping, no scaling.

Dropout in PyTorch

`nn.Dropout(p)` after the activation; `model.train()` / `model.eval()`; never on the output layer. $p \approx 0.5$ for big FC layers, 0.1–0.2 otherwise.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/regularisation/dropout

Data Augmentation

When you can't collect more data, make more from what you have: transformed copies that keep the right label. Which transformations suit images, audio and tabular data, and which ones break the label.

More data, made from the data you have

More training data is the most reliable cure for overfitting. When you can't collect it, add **transformed copies** of the examples, using only transformations that **don't change the correct answer**.

DATA	TYPICAL AUGMENTATIONS
Images	Small translations, rotations, crops, brightness changes, horizontal flips
Audio	Time shifts, added background noise
Tabular	Small additive noise on the inputs

MAKE SURE THE LABEL SURVIVES

Flipping a photo of a cat still gives a cat. Flipping a "6" or a word does not give the same digit or text, so don't flip digits or text. A 180° rotation turns a 6 into a 9.

- Apply augmentations **at random to each mini-batch during training**, so the network rarely sees the exact same example twice.
- **Never** at test time: evaluate on the real inputs.

The effect is to teach the network to be **invariant** to those transformations: the label shouldn't depend on where the object sits or how bright the photo is.

Common doubts

FIRST PASS **Is augmentation really regularisation if it adds data?**

Yes: by the definition, it changes training to reduce generalisation error, not training error (training error usually goes up a little, because the task got harder). It encodes prior knowledge, "these changes don't matter", that the raw data alone doesn't contain.

Cheatsheet

Augmentation

Label-preserving transforms, random per mini-batch, training only. Images: shift, rotate, crop, brightness, flip (not digits/text). Audio: shift, noise.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/regularisation/data-augmentation

Section 10

Convolutional Neural Networks

Why images need a different layer, how convolution and pooling work, and how CNNs build features from edges to objects.

10.1 From Fully Connected to Convolutional · 7 min

10.2 The Convolution Operation · 9 min

10.3 Pooling · 5 min

10.4 CNN Components and Feature Hierarchies · 6 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

?

BUILDING AND TRAINING A NETWORK · THE FORK

Does the input have spatial structure?

Dense layers, or convolutions?

Lessons here: 10.1 · feeds into Understand the neuron, Convolve

C1

BUILDING AND TRAINING A NETWORK · CONVOLUTIONAL LANE · STEP C1

Convolve

How does a filter find a pattern anywhere in the image?

Lessons here: 10.2

C2

BUILDING AND TRAINING A NETWORK · CONVOLUTIONAL LANE · STEP C2

Pool

How do feature maps shrink while keeping what matters?

Lessons here: 10.3

C3

BUILDING AND TRAINING A NETWORK · CONVOLUTIONAL LANE · STEP C3

Stack into a hierarchy

How do layers build up from edges to objects?

Lessons here: 10.4 · feeds into Choose the output and loss, Size the network

From Fully Connected to Convolutional

Why a fully connected layer is a poor fit for images, and the two ideas that fix it: local connectivity and weight sharing. With the parameter count that shows the difference.

The problem with fully connected layers on images

In a fully connected (FC) layer every output unit connects to every input value. For an image that ignores two facts:

- **Locality:** nearby pixels are related; a pixel in the top-left corner says little about one in the bottom-right.
- **Translation:** the same pattern (an edge, an eye) can appear anywhere, and should be detected the same way wherever it is.

Two ideas

IDEA	WHAT IT MEANS	EFFECT
Local connectivity	Each output looks only at a small patch of the input (e.g. 3×3)	Far fewer connections
Weight sharing	The same small filter is applied at every position	Parameters don't depend on image size; a pattern is detected anywhere

WORKED EXAMPLE: PARAMETER COUNT

A 32×32 colour image has $32 \times 32 \times 3 = 3{,}072$ values.

- **Fully connected** to 100 units: $3{,}072 \times 100 + 100 = 307{,}300$ parameters.
- **Convolutional**, 16 filters of size 3×3×3: $16 \times 27 + 16 = 448$ parameters.

The convolutional count depends on the filter size and the number of filters, not on the number of positions the filters slide over.

Rule: a conv layer with C_{out} filters of size $k \times k$ over C_{in} input channels has $k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}} + C_{\text{out}}$ parameters.

Common doubts

FIRST PASS **Do fewer parameters make a CNN less powerful?**

Not for images. The parameters it gives up encode connections that don't help (distant pixels, position-specific copies of the same detector). Building locality and translation into the architecture makes learning far more data-efficient, which is why CNNs beat FC networks on images while being much smaller.

Cheatsheet

Why convolutions

Local connectivity (small patches) + weight sharing (same filter everywhere). 32×32×3 → 100 FC units: 307,300 params; 16 filters 3×3×3: 448.

Conv layer parameters

$k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}} + C_{\text{out}}$ (weights + one bias per filter).

Online version with the interactive quiz: airesearchlabs.si/learn/dl/convolutional-networks/from-fully-connected-to-convolutional

The Convolution Operation

Slide a small filter over the input, multiply and add at each position: that produces a feature map. A full worked example by hand, what the filter detects, channels, and the output-size formula.

The operation

A **filter** (kernel) is a small grid of learned weights. Place it over a patch of the input, multiply matching values, add them up (plus a bias): that's one output value. Slide it one step and repeat. The grid of outputs is a **feature map**.

WORKED EXAMPLE: 4×4 INPUT, 3×3 FILTER, STRIDE 1, NO PADDING

Input

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

$*$

Filter (vertical edges)

1	0	-1
1	0	-1
1	0	-1

$=$

Feature map

0	2
-1	-1

Top-left output: the filter covers the highlighted 3×3 patch. Column sums of the patch are $1+0+2=3$, $2+1+1=4$ and $0+3+0=3$; the filter weights the columns $+1, 0, -1$, so the output is $3-3=0$.

Top-right: columns 2–4 of the top three rows give sums $4, 3, 2$, so $4-2=2$. Bottom row (rows 2–4): $3-4=-1$ and $2-3=-1$.

This filter responds to a **vertical edge**: brighter on the left than the right gives a positive value. A CNN learns its filter values; early layers typically end up with edge- and colour-detectors like this one.

Channels

A colour image has 3 channels, so each filter is $k \times k \times 3$ and sums over all three. A layer with 16 filters produces 16 feature maps, which become 16 input channels for the next layer.

Output size

The standard formula, for input size n, filter size k, padding p and stride s:

$$\text{output size} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$

Here $n = 4$, $k = 3$, $p = 0$, $s = 1$: $(4 - 3)/1 + 1 = 2$. **Padding** adds a border (usually zeros) so the output can keep the input's size; **stride** is the step between positions. Check your course's notation for these before relying on them in an exam.

A NAMING DETAIL

Deep learning "convolution" is, strictly, cross-correlation: the filter isn't flipped. Since the filter is learned, the difference doesn't matter in practice.

Common doubts

FIRST PASS **Is the same filter really used at every position?**

Yes. That's weight sharing: the 9 weights of a 3×3 filter (times the channels) are the only weights it has, reused at every position. Each different filter in a layer has its own weights and produces its own feature map.

Cheatsheet

Convolution

Slide filter, multiply-add per position (+ bias) → feature map. Output size $\lfloor (n + 2p - k)/s \rfloor + 1$.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/convolutional-networks/the-convolution-operation

Pooling

Pooling shrinks feature maps by summarising small regions. Max and average pooling worked out by hand, and what pooling buys: less computation and tolerance to small shifts.

Max and average pooling

Pooling slides a window (often 2×2, stride 2) over a feature map and keeps one number per window: the **maximum** or the **average**. It has no learned weights.

WORKED EXAMPLE: 2×2 POOLING, STRIDE 2

Feature map

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

→ Max pool

4	5
6	3

or Average pool

2.5	2
2.25	1.75

Top-left window $\{1, 3, 4, 2\}$: max 4, average 2.5. Top-right $\{2, 0, 1, 5\}$: max 5, average 2. Bottom-left $\{0, 1, 2, 6\}$: max 6, average 2.25. Bottom-right $\{3, 2, 1, 1\}$: max 3, average 1.75. The highlighted cells are the maxima.

What it buys

- **Smaller maps:** 2×2 pooling quarters the number of values, cutting computation in later layers.
- **Tolerance to small shifts:** if a feature moves by a pixel within a window, the max is unchanged. This gives a degree of translation invariance.
- Later units see a **larger region** of the original image, which helps them detect bigger structures.

Common doubts

FIRST PASS **Does max pooling throw information away?**

Yes, deliberately: it keeps whether a feature was detected strongly somewhere in the window and discards exactly where. For recognising objects that's usually what you want. For tasks needing precise positions (such as segmentation), architectures use less pooling or recover the detail later.

Cheatsheet

Pooling

Max or average over windows (e.g. 2×2, stride 2). No weights. Shrinks maps, adds tolerance to small shifts.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/convolutional-networks/pooling

CNN Components and Feature Hierarchies

The role of each part of a CNN (convolutional layers, activations, pooling and fully connected layers) and how stacking them builds features from edges to textures to whole objects.

The parts and their roles

COMPONENT	ROLE
Convolutional layers	Apply learned filters to extract local features into feature maps
Activation functions (ReLU)	Add non-linearity after each convolution, so stacked layers don't collapse into one linear map
Pooling layers	Shrink the maps, add tolerance to small shifts, cut computation
Fully connected layers	Near the output: combine the extracted features into the final prediction (with softmax for classification)

A typical stack: [convolution → ReLU → pooling] repeated a few times, then flatten, then one or two fully connected layers, then the output.

The feature hierarchy

DEPTH	TYPICAL LEARNED FEATURES
Early layers	Edges, corners, colour blobs
Middle layers	Textures, simple shapes, parts (an eye, a wheel)
Deep layers	Whole objects and faces

Each layer combines the features of the layer below, over a larger area of the image. This mirrors how images are actually built (objects from parts, parts from edges), which is why CNNs generalise well on images and why they no longer need hand-crafted features.

EXAM-READY SUMMARY

"CNNs exploit spatial locality through local connectivity and weight sharing, which cuts parameters dramatically. Stacked convolution, non-linearity and pooling layers learn a hierarchy from edges to parts to objects, and fully connected layers at the end make the prediction."

Common doubts

GOING DEEPER **How does a deep unit "see" a large part of the image if each filter is only 3×3?**

Through stacking. A unit in layer 2 sees a 3×3 patch of layer-1 maps, and each of those saw a 3×3 patch of the image, so it depends on a 5×5 region of the image. Pooling enlarges this region further. This region is called the receptive field, and it grows with depth.

Cheatsheet

CNN components

Conv: local features. ReLU: non-linearity. Pooling: shrink + shift tolerance. FC: combine features → prediction.

Feature hierarchy

Early: edges, corners. Middle: textures, parts. Deep: objects, faces. Each layer combines the one below over a larger area.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/convolutional-networks/cnn-components-and-feature-hierarchies

Section 11

Network Design and PyTorch

Depth versus width, designing a network for a real task, and implementing and training a network in PyTorch.

11.1 Depth versus Width · 7 min

11.2 Designing a Network for Tabular Data · 8 min

11.3 Neural Networks in PyTorch · 10 min

Where this section fits

Every lesson is a step in building a model. These are the steps this section teaches; the Big picture PDF and the online map show how they connect to the rest.

M3

BUILDING AND TRAINING A NETWORK · DENSE LANE · STEP M3

Size the network

How deep, how wide, and for which task?

Lessons here: 11.1, 11.2 · feeds into Regularise

7

BUILDING AND TRAINING A NETWORK · STEP 7

Update the weights

How do I turn gradients into a better network?

Lessons here: 11.3 · feeds into Evaluate honestly

Depth versus Width

More layers or more units per layer? How depth and width differ in expressive power, parameter count and trainability, and why most successful networks are deep rather than very wide.

Two ways to add capacity

	MORE DEPTH (LAYERS)	MORE WIDTH (UNITS PER LAYER)
Expressive power	Builds **hierarchical, compositional** features; some functions need far fewer units when built deep	More features at the **same level** of abstraction; a shallow net may need exponentially many units for the same function
Parameters	Grow roughly linearly with the number of layers	Grow with the **product** of adjacent widths: doubling two adjacent layers quadruples their weights
Training	Harder: vanishing or exploding gradients, more sequential computation	Easier to optimise, highly parallel
Main risk	Optimisation difficulty	Overfitting from sheer parameter count

Why depth usually wins

The universal approximation theorem says one wide hidden layer *can* represent any continuous function, but possibly with an enormous number of units. Composing layers lets a network reuse intermediate features (lesson 3.3), so it represents structured functions such as images, speech and language much more efficiently. In practice, depth often contributes more to useful expressive power than width.

Making depth trainable

- **ReLU** activations, to avoid vanishing gradients (Section 4).
- **Careful initialisation** and normalised inputs.
- **Batch normalisation** and **residual (skip) connections**, which let very deep networks train well (beyond this course).

A BALANCED ANSWER

"Depth gives hierarchical features and can be exponentially more efficient than width, but is harder to optimise. Width is easier to train but adds parameters fast and risks overfitting. Modern networks are deep, with ReLU, normalisation and residual connections to keep them trainable."

Common doubts

FIRST PASS **So should I always make my network as deep as possible?**

No. Extra depth only helps if the problem has structure to exploit and you can train it. For small tabular datasets, two or three hidden layers are usually plenty; deeper networks mostly add training trouble and overfitting risk. Let validation performance decide.

FIRST PASS **How do I answer a "trade-offs between depth and width" question?**

Use the layout **two options × two criteria (power, cost) + a verdict**.

	DEEPER (MORE LAYERS)	WIDER (MORE NEURONS PER LAYER)
Power	Features of features: edges → shapes → objects	More features at the same level
Efficiency	The same function with far fewer neurons	May need exponentially more neurons
Training	Harder: vanishing gradients (eased by ReLU, batch norm, residual connections)	Easier and more stable
Speed	Slower: layers run one after another	Faster on GPUs: a layer's neurons run in parallel
Risk	Both add parameters, so **both** can overfit	

Link it to the universal approximation theorem: one wide layer is enough in theory, but the theorem says nothing about how many neurons that takes. **Verdict:** depth gives more power per parameter, so modern networks go deep and use ReLU, batch norm and residual connections to keep them trainable.

Memory line: "Deep is efficient but hard to train. Wide is easy to train but wasteful."

Cheatsheet

Depth vs width

Depth: hierarchical features, efficient, harder to train (vanishing gradients). Width: easy to train, parallel, but may need far more neurons. Both add capacity, so both can overfit.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/depth-versus-width

Designing a Network for Tabular Data

A worked design for a regression problem: predicting house prices from features such as size, bedrooms and location. Architecture, activations, loss, regularisation, categorical features and the challenges to expect.

The task

Predict a house's price (a number) from features: floor area, number of bedrooms, location, property type, and so on. This is **regression** on **tabular** data.

A sensible starting design

CHOICE	PROPOSAL	WHY
Layers	2–3 fully connected hidden layers, e.g. 64 → 32 → 16 units	Tabular features have no spatial structure, so dense layers suit; shrinking widths compress the features
Hidden activation	ReLU	Non-linear, cheap, avoids vanishing gradients
Output	1 unit, **linear**	A price is any positive real number; a sigmoid would cap it
Loss	MSE (MAE if outliers matter)	Standard for regression; MAE resists extreme prices
Regularisation	L2 weight decay; optionally dropout; early stopping	Tabular datasets are often small, so overfitting is the main risk

It's a starting point, not a recipe: the best architecture depends on the data and is chosen on validation performance.

Preparing the features

- **Numeric** features: standardise using training statistics (also the target, then undo it on the predictions).
- **Categorical** features (location, property type): **one-hot encode**. With many categories (hundreds of postcodes), use an **embedding layer** that learns a short dense vector per category.
- **Missing values:** impute with training medians and add "was missing" indicators.
- **Outliers:** cap or remove clear errors; consider a log of the price as the target.

Challenges to expect

- **Overfitting** on a small dataset: regularise, use cross-validation, collect more data.
- **Mixed feature types and scales**: encode and scale consistently.
- **Irrelevant features**: L1 or simply checking which features matter.
- **Leakage:** a feature known only after the sale (such as the final agent fee) must be excluded.

Common doubts

FIRST PASS **Why not encode location as a number, like 1 = north, 2 = south?**

Because the network would treat it as a quantity: "south" would be "twice north", and locations 1 and 3 would look further apart than 1 and 2. One-hot encoding gives each location its own input with no false ordering; embeddings go further and learn which locations behave similarly.

Cheatsheet

Tabular regression design

Dense ReLU layers (64→32→16), 1 linear output, MSE, L2 + early stopping (+ dropout). Standardise numerics, one-hot or embed categories.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/designing-a-network-for-tabular-data

Neural Networks in PyTorch

Define a network, choose a loss and optimiser, and write the training loop. A 100 → 50 → 10 classifier line by line, the logits rule, train and eval modes, and a custom activation function.

Defining the network

A classifier with 100 input features, 50 ReLU hidden units and 10 classes:

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(100, 50)   # 100*50 + 50 = 5,050 parameters
        self.fc2 = nn.Linear(50, 10)    # 50*10 + 10  =   510 parameters

    def forward(self, x):
        return self.fc2(torch.relu(self.fc1(x)))   # logits: no softmax here
```

THE LOGITS RULE

`nn.CrossEntropyLoss` applies log-softmax itself, so the model must return **raw logits**. A question may ask for "an output layer with softmax": for training with `CrossEntropyLoss`, leave it out and say why, then apply `torch.softmax(logits, dim=1)` only when you need probabilities.

Training

```
model = MLP()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)   # or torch.optim.Adam

for epoch in range(num_epochs):
    model.train()                       # training mode (dropout on)
    for xb, yb in train_loader:         # mini-batches
        optimizer.zero_grad()           # 1. clear old gradients
        loss = loss_fn(model(xb), yb)   # 2. forward pass and loss
        loss.backward()                 # 3. backpropagation
        optimizer.step()                # 4. gradient descent update

    model.eval()                        # evaluation mode (dropout off)
    with torch.no_grad():               # no gradients needed
        val_loss = sum(loss_fn(model(xb), yb).item() for xb, yb in val_loader)
```

- `zero_grad()` : PyTorch **adds** gradients by default, so clear them each step.
- `backward()` is backpropagation; `step()` is the update. L2 weight decay: `torch.optim.SGD(..., weight_decay=1e-4)` .
- Record training and validation loss every epoch to watch for overfitting.

A custom activation function

Activations are just modules. For example "LeakySigmoid": $\sigma(x)$ for $x \geq 0$ and $0.01\,\sigma(x)$ for $x < 0$:

```
class LeakySigmoid(nn.Module):
    def forward(self, x):
        s = torch.sigmoid(x)
        return torch.where(x ⩾ 0, s, 0.01 * s)
```

Use it like any layer, e.g. `self.act = LeakySigmoid()` and `self.fc2(self.act(self.fc1(x)))` . Autograd differentiates `torch.where` and `torch.sigmoid` automatically, so no hand-written backward is needed.

Common doubts

FIRST PASS **What happens if I forget optimizer.zero_grad()?**

Gradients from every previous batch keep adding up, so each step uses a growing sum of old gradients instead of the current one. Training behaves erratically or diverges. Clear them at the start of every iteration.

GOING DEEPER **Does the LeakySigmoid fix the vanishing-gradient problem?**

Not really. For negative inputs it's $0.01\,\sigma(x)$, so its gradient there is even **smaller** than the sigmoid's, and for large positive inputs it saturates just like a sigmoid. It's a coding exercise more than a recommended activation; ReLU or Leaky ReLU are the practical choices.

Cheatsheet

Training loop

`zero_grad()` → forward + loss → `loss.backward()` → `optimizer.step()`. `model.train()` / `model.eval()` + `torch.no_grad()` for validation.

Classifier output

Return logits from the last `nn.Linear`; `nn.CrossEntropyLoss` applies log-softmax. Softmax only for reporting probabilities.

Online version with the interactive quiz: airesearchlabs.si/learn/dl/design-and-pytorch/neural-networks-in-pytorch