AI Research Labs

Section 7 of 11 · 6 lessons

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

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



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.

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