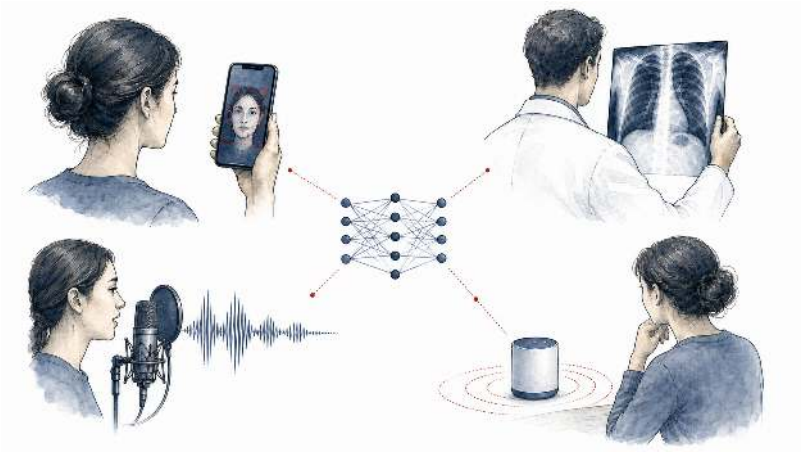
AI and Deep Learning

Day 1: the whole day

Day 1 of 4

Fatih Kansoy
Worcester College, University of Oxford

Where deep learning is already working



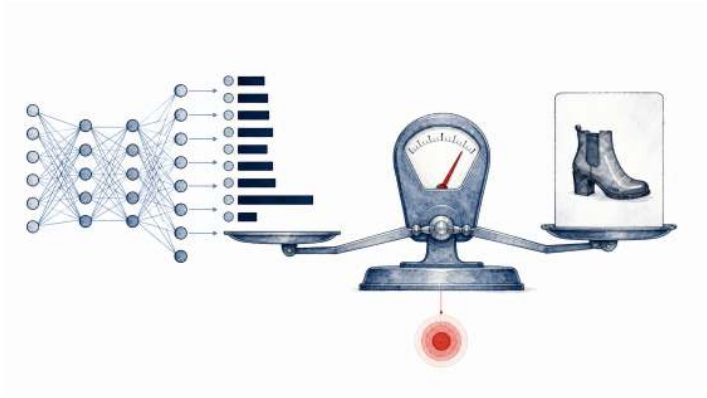
input data $\longrightarrow$ **learned representation** $\longrightarrow$ prediction

Part IV

Forward propagation and loss functions

One example produces one loss value. Training adjusts the parameters to reduce it.

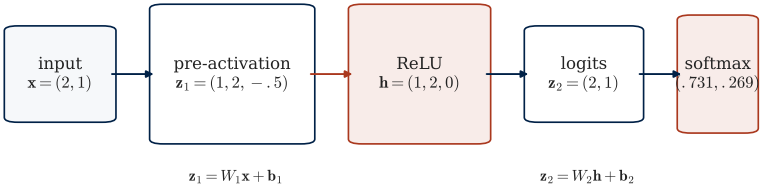
The forward pass ends with a loss



model scores + observed label $\longrightarrow$ **one loss value**

The loss measures the disagreement between the prediction and the observed label. Training adjusts the parameters to reduce it.

The complete forward pass has four stages



1. The first matrix produces hidden scores $\boldsymbol{z}_1$.
2. ReLU produces the hidden representation $\boldsymbol{h}$.
3. The second matrix produces class scores $\boldsymbol{z}_2$.
4. Softmax and cross-entropy turn those scores into one loss value.

Part III explained how the hidden representation is built. Part IV completes the calculation from class scores to a training signal.

Read W_1 one row at a time

$$\boldsymbol{z}_1 = W_1\,\boldsymbol{x} + \boldsymbol{b}_1 = \begin{bmatrix} 1 & -1 \\ 0.5 & 0.5 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix}$$

W: weights $\quad$ $\boldsymbol{x}$: input $\quad$ $\boldsymbol{b}$: bias $\quad$ the result $\boldsymbol{z}_1$

$$z_{1,1} = 1(2) - 1(1) + 0 = 1$$

$$z_{1,2} = 0.5(2) + 0.5(1) + 0.5 = 2$$

$$z_{1,3} = -1(2) + 2(1) - 0.5 = -0.5$$

Each *row* of W_1 is one unit's weights, and each entry of $\boldsymbol{b}_1$ is that unit's bias. The input $\boldsymbol{x}$ is shared by all three. Three rows, three units, three numbers out, all at once.

The shapes check: $(3 \times 2)(2 \times 1) + (3 \times 1) = (3 \times 1)$.

Pre-activation and representation are different objects

$$\underbrace{\boldsymbol{z}_1 = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix}}_{\text{pre-activation}} \xrightarrow{\text{ReLU, entry by entry}} \underbrace{\boldsymbol{h} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}}_{\text{representation}}$$

$\boldsymbol{z}_1$ is what the weights produced. $\boldsymbol{h}$ is what the layer passes on. They differ in exactly the entries ReLU switched off, and Part V will need both of them, separately.

This distinction is essential for backpropagation: $\boldsymbol{z}_1$ determines which ReLU units can pass a gradient.

The second matrix produces one score per class

$$\boldsymbol{z}_2 = W_2\,\boldsymbol{h} + \boldsymbol{b}_2 = \begin{bmatrix} 1 & 0.5 & -1 \\ 0 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

W: weights $\quad$ $\boldsymbol{h}$: hidden features $\quad$ $\boldsymbol{b}$: bias $\quad$ the result $\boldsymbol{z}_2$

$$z_{2,1} = 1(1) + 0.5(2) - 1(0) + 0 = 2$$
$$z_{2,2} = 0(1) + 0.5(2) + 1(0) + 0 = 1$$

Row j of W_2 says how strongly each hidden feature $\boldsymbol{h}$ counts towards class j. The two outputs are the **logits**: scores on no particular scale, not yet probabilities.

the top score 2 is class 1, the bottom score 1 is class 2

Logits rank classes, but they are not probabilities

The final matrix produced two **logits**:

$$z_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

What the logits tell us

- Class 1 has the larger score.
- The model therefore prefers class 1.
- The score gap is $2 - 1 = 1$.

Why they are not probabilities

- Logits can be negative or larger than 1.
- Here they sum to 3, not 1.
- Their scale has no direct probability meaning.

> **Before comparing the prediction with the label, convert the logits into positive numbers that sum to one while preserving their order.**

Softmax converts scores in two steps

Start with any vector of logits $\boldsymbol{z} = (z_1, \ldots, z_K)$.

1. **Exponentiate each score:**

$$a_j = e^{z_j}.$$

   Every a_j is positive, and larger logits remain larger.

2. **Divide by their total:**

$$p_j = \frac{a_j}{\sum_k a_k} = \frac{e^{z_j}}{\sum_k e^{z_k}}.$$

   The resulting probabilities satisfy $p_j > 0$ and $\sum_j p_j = 1$.

> **Softmax does not choose a class by itself. It expresses the model's relative preference for every class as a probability distribution.**

Softmax depends only on differences between logits

These three logit vectors have the same gap between class 1 and class 2:

$$(2, 1), \qquad (102, 101), \qquad (0, -1).$$

Therefore all three produce the same probabilities:

$$\boldsymbol{p} = (0.731,\ 0.269).$$

The algebra shows why. Adding the same constant c to every logit gives

$$\frac{e^{z_j + c}}{\sum_k e^{z_k + c}} = \frac{e^c e^{z_j}}{e^c \sum_k e^{z_k}} = \boxed{\frac{e^{z_j}}{\sum_k e^{z_k}}}.$$

Softmax measures how the classes compare with one another. A common shift changes none of those comparisons.

Apply softmax to the two logits

1. Exponentiate

$$\begin{bmatrix}2\\1\end{bmatrix} \longrightarrow \begin{bmatrix}e^2\\e^1\end{bmatrix} = \begin{bmatrix}7.389\\2.718\end{bmatrix}.$$

The values are positive and keep the same order.

2. Normalize

$$7.389 + 2.718 = 10.107,$$

$$\boldsymbol{p} = \begin{bmatrix}7.389/10.107\\2.718/10.107\end{bmatrix} = \begin{bmatrix}0.731\\0.269\end{bmatrix}.$$

- Both entries are positive and they sum to 1.
- The larger logit remains the larger probability.

A one-point logit advantage becomes a 73% versus 27% probability split.

Prediction and label play different roles

Model prediction

$$\boldsymbol{p} = \begin{bmatrix} 0.731 \\ 0.269 \end{bmatrix}.$$

The largest probability is for class 1, so the model predicts

class 1.

Observed label

$$\boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The 1 marks the observed category, so the correct answer is

class 2.

The prediction is computed from $\boldsymbol{x}$, the weights and the biases.

The label is then used to evaluate that prediction.

> **Softmax tells us what the model believes. The label tells us which probability should have been large.**

The one-hot label selects the correct-class probability

Multiply each predicted probability by the matching entry of the label:

class	predicted p_j	label y_j	$y_j p_j$
1	0.731	0	0
2	0.269	1	0.269

The zero removes the probability for class 1; the one keeps the probability for the observed class:

$$p_{\text{true}} = \sum_j y_j p_j = \boldsymbol{y}^\top \boldsymbol{p} = \boxed{0.269}.$$

The label enters the forward calculation here. During inference there is no label, so the calculation stops at the predicted probabilities.

Cross-entropy turns probability into a penalty

The loss should be small when p_{true} is large and large when p_{true} is small. Cross-entropy uses

$$\ell = -\log p_{\text{true}}.$$

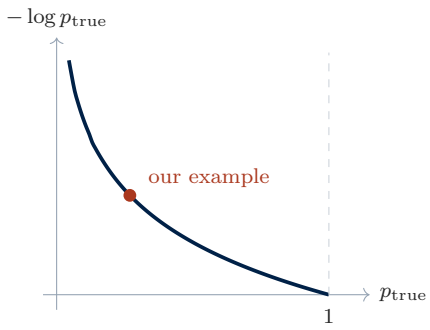
For this example,

$$\ell = -\log(0.269) = \boxed{1.313}.$$

$$\begin{aligned} p_{\text{true}} = 1 \quad &\Rightarrow \quad \ell = 0 \\ p_{\text{true}} = 0.269 \quad &\Rightarrow \quad \ell = 1.313 \\ p_{\text{true}} = 0.01 \quad &\Rightarrow \quad \ell = 4.605 \end{aligned}$$

High probability on the correct class gives a small loss; low probability gives a large loss.

The logarithm gives the loss useful properties



1. The loss is 0 at $p_{\text{true}} = 1$ and grows without bound as $p_{\text{true}} \to 0$. Confident errors receive the largest penalty.

2. Products of probabilities become sums of losses, which can be averaged across a batch.

3. Together with softmax, cross-entropy gives a simple gradient for the output logits.

For the worked example, $p_{\text{true}} = 0.269$ gives $\ell = 1.313$.

Accuracy treats every wrong prediction alike

	logits z_2	p	predicted	correct?	loss
near miss	(1.0, 0.9)	(0.525, 0.475)	class 1	**no**	0.744
our example	(2.0, 1.0)	(0.731, 0.269)	class 1	**no**	1.313
confident error	(3.0, 0.0)	(0.953, 0.047)	class 1	**no**	3.049

- Accuracy assigns all three examples the same result: incorrect.
- Cross-entropy distinguishes a near miss from a confident error.
- Unlike accuracy, cross-entropy changes smoothly as the logits change.

Accuracy summarizes final decisions. Cross-entropy supplies a differentiable signal for improving the parameters.

Average the example losses to obtain the objective

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_i = -\frac{1}{n}\sum_{i=1}^{n} \log p_{i,\,y_i}$$

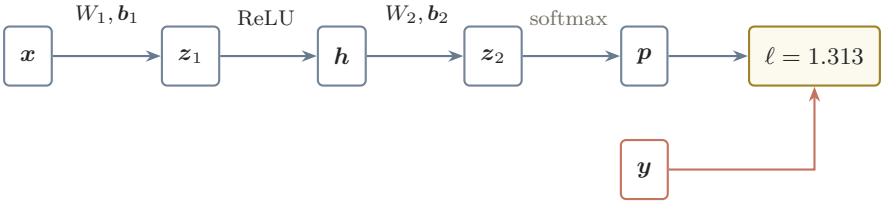
Each example produces one loss ℓ_i. Their average is the objective used to evaluate the current parameters on a batch of n examples.

> θ denotes the network's 17 weights and biases. With the data held fixed, training changes θ in order to reduce $L(\theta)$.

- We average the losses—not the predictions—so one difficult or unusual example does not determine the entire update.
- The average also remains comparable when the batch size changes

Using the mean rather than the sum keeps the scale of the objective comparable across different batch sizes.

The loss completes the forward computation



Every arrow represents a differentiable calculation, and the chain ends at one loss value.

> **Later we follow this graph backwards and computes how the loss changes with each of the 17 parameters.**

Part IV: main lessons

1. **Logits are relative class scores.** Their order indicates the preferred class, but they are not probabilities.

2. **Softmax normalizes the logits.** It produces positive probabilities that sum to one and depend only on score differences.

3. **The label selects the correct-class probability.** In the example, class 2 selects $p_{\text{true}} = 0.269$.

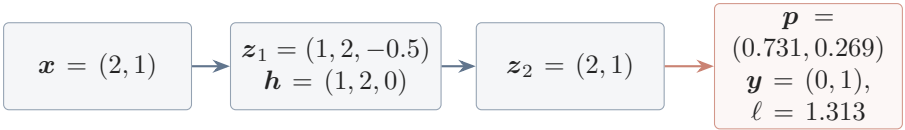
4. **Cross-entropy produces one penalty.** $\ell = -\log p_{\text{true}} = 1.313$ for this example.

5. **Training minimizes the average loss.** This connects the forward computation to parameter updates.

The full chain is now complete: $\boldsymbol{x} \to \boldsymbol{z}_1 \to \boldsymbol{h} \to \boldsymbol{z}_2 \to \boldsymbol{p} \to \ell$.

The forward pass turned one example into one loss

$$\boldsymbol{x} \longrightarrow \boldsymbol{z}_1 \longrightarrow \boldsymbol{h} \longrightarrow \boldsymbol{z}_2 \longrightarrow \boldsymbol{p} \longrightarrow \ell$$



1. The current parameters transformed the input into class probabilities.
2. The observed label was class 2, so the loss used $p_2 = 0.269$.
3. Cross-entropy converted that probability into the loss 1.313.

The forward pass made the model's current error measurable.

The loss evaluates the parameters; it does not change them

$$(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) \xrightarrow{\text{forward pass}} \ell(\boldsymbol{\theta}) = 1.313, \qquad \boldsymbol{\theta} = (\theta_1, \ldots, \theta_{17}).$$

What the loss tells us

- The prediction favoured class 1, but the observed class was 2.
- The observed class received probability 0.269.
- These parameter values produced loss 1.313.

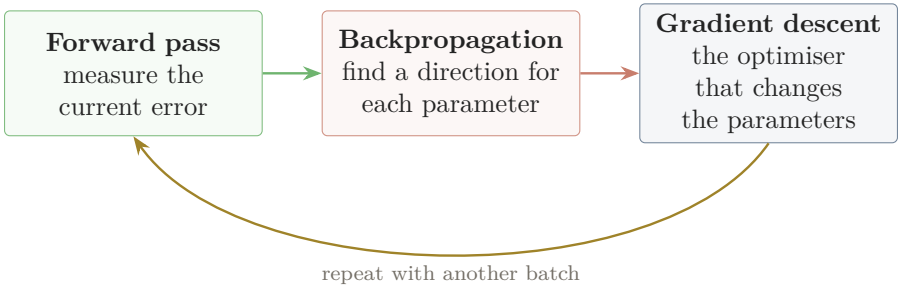
What the loss alone cannot tell us

- Which parameters contributed to the error.
- Whether each parameter should increase or decrease.
- How to coordinate all 17 parameter changes.

Measuring the error is necessary. Learning also requires a direction for changing the parameters.

Training closes the loop around the forward pass

Part IV completed the first stage. A trained network requires the entire loop:



$$\boldsymbol{\theta} \xrightarrow{\text{forward}} \ell \xrightarrow{\text{backpropagation}} \boldsymbol{g} = \nabla_{\theta}\ell \xrightarrow{\text{gradient descent}} \boldsymbol{\theta}^{\text{new}} \longrightarrow \cdots$$

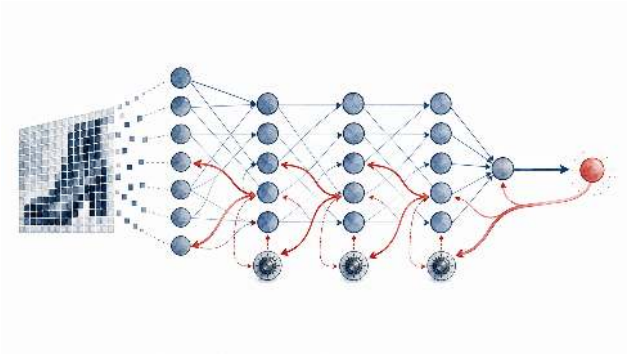
Training is repeated measurement and correction, not a single forward calculation.

Part V

Backpropagation and optimisation

The loss gives one overall error. Learning requires one useful direction for every weight and bias.

Training needs one answer for each parameter



$\ell = 1.313 \quad \longrightarrow \quad$ **17 questions about 17 parameters**

W_1: 6 weights, $\boldsymbol{b}_1$: 3 biases, W_2: 6 weights, $\boldsymbol{b}_2$: 2 biases.

The loss tells us that the prediction is poor. Training needs to know how the loss would change if each parameter moved a little.

One derivative studies one parameter

Choose parameter θ_j. Keep the other parameters at their current values and ask how a small change in θ_j would affect the loss:

$$g_j = \frac{\partial \ell}{\partial \theta_j}.$$

For a small change δ,

$$\ell(\theta_j + \delta) - \ell(\theta_j) \approx g_j \delta.$$

- $g_j > 0$: increasing θ_j raises the loss;
- $g_j < 0$: increasing θ_j lowers the loss;
- $|g_j|$: strength of the local effect.

The derivative measures what a small parameter change would do; it does not perform the change.

▸ Interpret one small change

The gradient collects $g_1, g_2, \ldots, g_{17}$

$$\begin{array}{ccccccccc} \theta_1 & \theta_2 & \theta_3 & \cdots & \theta_{15} & \theta_{16} & \theta_{17} \\ \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow \\ g_1 & g_2 & g_3 & \cdots & g_{15} & g_{16} & g_{17} \end{array}$$

$$\boxed{\nabla_\theta \ell = (g_1, g_2, \ldots, g_{17})}$$

1. g_j answers the same local question for parameter θ_j.
2. The sign tells us which way is uphill for that parameter.
3. The size tells us how strongly the loss responds nearby.

One loss value becomes a map of the local slope in all 17 parameter directions.

Backpropagation computes; gradient descent updates

$$\ell \xrightarrow{\text{backpropagation}} \boxed{\boldsymbol{g} = \nabla_\theta \ell = \left(\frac{\partial \ell}{\partial \theta_1}, \ldots, \frac{\partial \ell}{\partial \theta_{17}} \right)}$$

$$\boldsymbol{g} \xrightarrow{\text{gradient descent}} \boxed{\boldsymbol{\theta}^{\text{new}} = \boldsymbol{\theta}^{\text{old}} - \eta \boldsymbol{g}}$$

Backpropagation

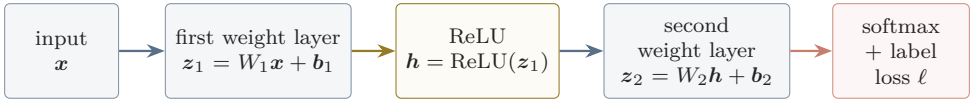
- uses the loss and stored forward values;
- calculates $\boldsymbol{g}$ using the chain rule;
- does not change the parameters.

Gradient descent

- is the optimiser used here;
- receives $\boldsymbol{g}$ and the learning rate η;
- changes the parameters in direction $-\boldsymbol{g}$.

Backpropagation returns $\boldsymbol{g}$; gradient descent uses it to update $\boldsymbol{\theta}$.

The forward pass records how the loss was produced



For the running example,

$$\boldsymbol{z}_1 = (1, 2, -0.5) \xrightarrow{\text{ReLU}} \boldsymbol{h} = (1, 2, 0).$$

1. Weight layers calculate scores.
2. ReLU changes the first-layer scores into hidden values.
3. The loss compares the final probabilities with the observed label.

The network stores these intermediate values because the backward pass will need them.

Backpropagation reuses the forward path in reverse

The forward pass calculated and stored

$$\boldsymbol{x} \longrightarrow \boldsymbol{z}_1 \longrightarrow \boldsymbol{h} \longrightarrow \boldsymbol{z}_2 \longrightarrow \boldsymbol{p} \longrightarrow \ell.$$

Backpropagation starts at the loss and calculates how the loss changes at each earlier stage, in the opposite order:

$$\ell \xrightarrow{\text{softmax and loss}} \frac{\partial \ell}{\partial \boldsymbol{z}_2} \xrightarrow{\text{second weight layer}} \frac{\partial \ell}{\partial \boldsymbol{h}} \xrightarrow{\text{ReLU}} \frac{\partial \ell}{\partial \boldsymbol{z}_1} \xrightarrow{\text{first weight layer}} \nabla_\theta \ell.$$

1. Start from how the final scores affected the loss.
2. Pass that information through the second layer and ReLU.
3. Continue to the first layer and complete every parameter derivative.

The backward pass returns the complete gradient $\boldsymbol{g} = \nabla_\theta \ell = (g_1, \ldots, g_{17})$.

▸ Open the chain-rule calculation

The loss gives the first backward signal: $\boldsymbol{p} - \boldsymbol{y}$

The backward pass first asks how each class score affected the loss. For softmax and cross-entropy, the answer has a simple form:

$$\frac{\partial \ell}{\partial \boldsymbol{z}_2} = \boldsymbol{p} - \boldsymbol{y}.$$

For the running example,

$$\begin{bmatrix} 0.731 \\ 0.269 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.731 \\ -0.731 \end{bmatrix}.$$

$\partial \ell / \partial z_{2,1} = +0.731$
increasing class-1
score raises the loss

$\partial \ell / \partial z_{2,2} = -0.731$
increasing class-2
score lowers the loss

Gradient descent moves in the opposite direction: class 1 down, observed class 2 up.

▸ Derive $\boldsymbol{p} - \boldsymbol{y}$

A weight layer sends backward information to two destinations

For one connection, the forward calculation is

$$z = wa + b.$$

Backpropagation arrives with

$$g_z = \frac{\partial \ell}{\partial z},$$

which describes how the loss responds to a small change in z.

The layer must now answer two different questions:

1. **How did the earlier value a affect the loss?**
   This derivative is sent to the preceding layer so backpropagation can continue.

2. **How did the trainable weight w affect the loss?**
   This derivative is stored so the optimiser can update w.

The first result continues the backward pass; the second changes the current layer during the update.

▸ Follow both backward tasks

The chain rule gives both results from the same equation

From $z = wa + b$, changing a, w, or b changes z by

$$\frac{\partial z}{\partial a} = w, \qquad \frac{\partial z}{\partial w} = a, \qquad \frac{\partial z}{\partial b} = 1.$$

Multiplying each local effect by the arriving value $g_z = \partial \ell / \partial z$ gives

$$\underbrace{\frac{\partial \ell}{\partial a} = g_z w}_{\text{send to the preceding layer}}, \qquad \underbrace{\frac{\partial \ell}{\partial w} = g_z a}_{\text{store for the weight update}}, \qquad \underbrace{\frac{\partial \ell}{\partial b} = g_z}_{\text{store for the bias update}}.$$

In the running example, hidden value $h_2 = 2$ reaches class-1 score $z_{2,1}$ through a weight $w = 0.5$, and the arriving signal is $g_z = 0.731$. Therefore

$$\frac{\partial \ell}{\partial a} = 0.3655, \qquad \frac{\partial \ell}{\partial w} = 1.462, \qquad \frac{\partial \ell}{\partial b} = 0.731.$$

▸ Follow the complete calculation

Backpropagation now reaches the ReLU between the layers

$$\frac{\partial \ell}{\partial \boldsymbol{z}_2} \xrightarrow{W_2} \frac{\partial \ell}{\partial \boldsymbol{h}} \xrightarrow{\text{ReLU}} \frac{\partial \ell}{\partial \boldsymbol{z}_1} \xrightarrow{W_1} \text{first-layer gradients.}$$

The forward pass already stored

$$\boldsymbol{z}_1 = \begin{bmatrix} 1 \\ 2 \\ -0.5 \end{bmatrix}.$$

Backpropagation does not calculate $W_1\boldsymbol{x} + \boldsymbol{b}_1$ again. It uses only the sign of each saved score:

$$z_{1,j} > 0 \Longrightarrow \text{ReLU}'(z_{1,j}) = 1, \qquad z_{1,j} < 0 \Longrightarrow \text{ReLU}'(z_{1,j}) = 0.$$

For this example,

$$\text{ReLU}'(\boldsymbol{z}_1) = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

These three slopes now determine which parts of the arriving hidden-layer signal can continue to W_1.

▸ Derive the ReLU backward rule

The saved ReLU slopes determine what reaches the first layer

The second weight layer gives $\partial \ell / \partial \boldsymbol{h} = W_2^\top (\boldsymbol{p} - \boldsymbol{y})$. ReLU multiplies the resulting vector by the saved slopes entry by entry:

$$\underbrace{\begin{bmatrix} 0.731 \\ 0 \\ -1.462 \end{bmatrix}}_{\text{arriving from } W_2} \odot \underbrace{\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}}_{\text{saved ReLU slopes}} = \underbrace{\begin{bmatrix} 0.731 \\ 0 \\ 0 \end{bmatrix}}_{\text{sent to } W_1}.$$

- Unit 1 receives a nonzero signal and passes it.
- Unit 2 is active, but the signal arriving from W_2 is already zero.
- Unit 3 receives a nonzero signal, but its saved ReLU slope is zero, so the signal stops.

Applying the same weight-layer rule to W_1 now completes the remaining gradients.

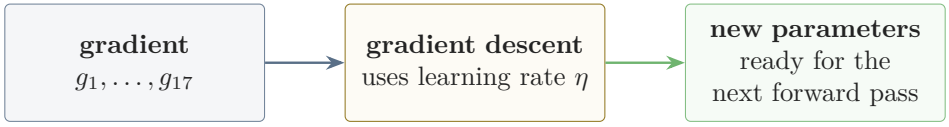
▸ Calculate the ReLU gradients

Gradient descent turns the completed gradient into one update

After the backward pass,

$$\nabla_\theta \ell = (g_1, g_2, \ldots, g_{17})$$

contains one local direction for every weight and bias.



$$\boldsymbol{\theta}^{\text{new}} = \boldsymbol{\theta}^{\text{old}} - \eta \nabla_\theta \ell$$

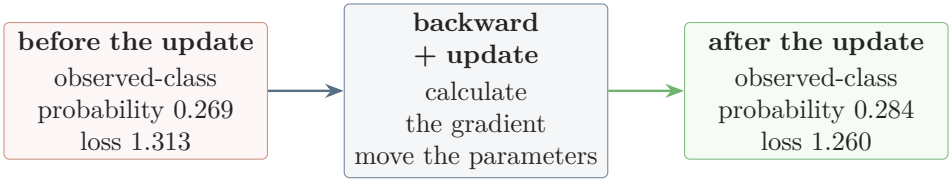
The minus sign moves against the local increase in loss. The learning rate keeps that move controlled.

Backpropagation calculates g. Gradient descent updates θ using $-\eta g$.

▸ Check one numerical update

The next forward pass checks whether the update helped

For a transparent numerical check, use the two output-bias entries of the gradient for one small update and keep every other parameter fixed. Then run the same example forward again:



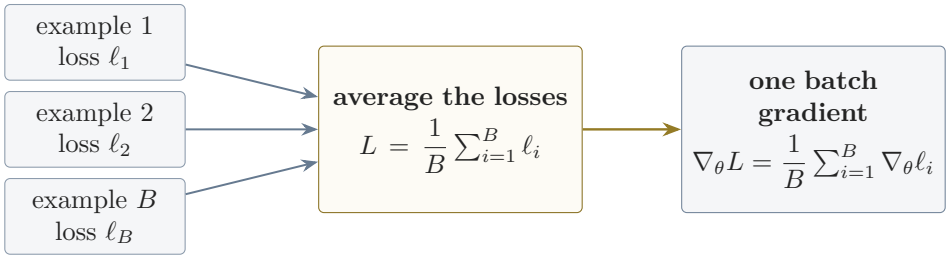
probability of the observed class rises and loss falls.

This is the purpose of learning: use the loss to change the parameters so the next prediction better matches the observed label.

This simplified check isolates one part of the update. Full training updates every parameter using batch gradients.

▸ Check the numerical update

A batch combines several examples before one update



One example gives one loss. A batch contains B different examples, each with its own loss. Their information is averaged before one parameter update.

With 600 training images, batch size 32 gives 19 steps

The experiment contains $N = 600$ training images and uses $B = 32$ images per batch. Each batch produces one parameter update, called one **step**.

$$18 = \left\lfloor \frac{600}{32} \right\rfloor \qquad \text{complete 32-image batches,}$$

$$576 = 18 \times 32 \qquad \text{images used in those complete batches,}$$

$$24 = 600 - 576 \qquad \text{images left for the final smaller batch,}$$

$$19 = 18 + 1 = \left\lceil \frac{600}{32} \right\rceil \qquad \text{steps in one epoch.}$$

- Steps 1–18 use 32 images each; step 19 uses the remaining 24 images.
- After these 19 steps, all 600 images have been used once: this is one **epoch**.
- The number of epochs is separate from the number of steps in each epoch. Validation performance helps decide when to stop.

Training must begin from an initial set of parameters

Gradient descent changes existing parameter values. Before the first forward pass, however, no loss or gradient exists, so the network needs a starting point $\boldsymbol{\theta}_0$:

$$\boldsymbol{\theta}_0 \xrightarrow{\text{first forward pass}} L_0 \xrightarrow{\text{backpropagation}} \nabla_\theta L_0 \xrightarrow{\text{update}} \boldsymbol{\theta}_1.$$

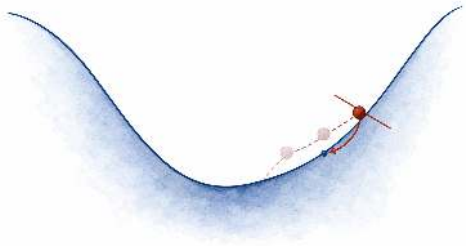
Why not give every hidden unit the same initial weights?

$$\text{same inputs and weights} \implies \text{same outputs} \implies \text{same gradients} \implies \text{same updates.}$$

The units remain identical and learn the same feature. A layer with several identical units therefore wastes its width.

> **Weights start with small random differences so hidden units can follow different learning paths. Biases commonly start at zero.**

The learning rate controls how far each update moves

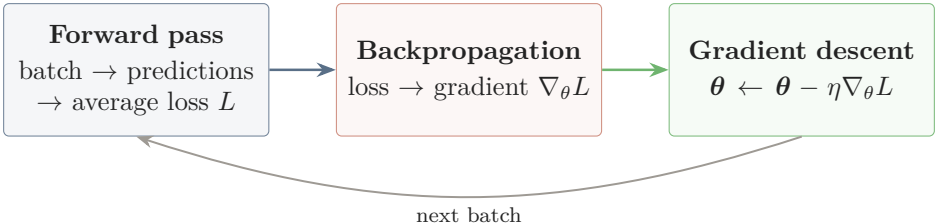


$$\boldsymbol{\theta}^{\text{new}} = \boldsymbol{\theta}^{\text{old}} - \eta \nabla_{\theta} L$$

- **Too small:** correct direction, very slow progress.
- **Suitable:** loss falls steadily.
- **Too large:** the update overshoots and loss may rise.

The gradient is local: it describes the loss near the current parameters. The learning rate η decides how far to trust that local information.

Every training step repeats the same three operations



1. The forward pass measures the current error.
2. Backpropagation calculates how each parameter affects that error.
3. Gradient descent changes the parameters and the next step starts.

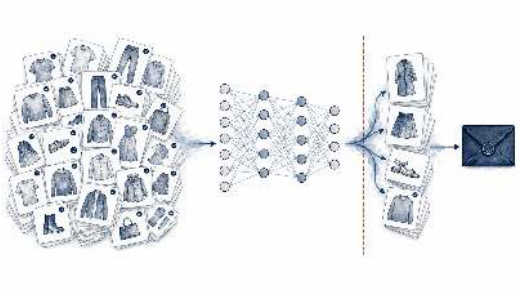
Training can reduce error on the examples it sees. The next question is whether the learned rule also works on new examples.

Part VI

Generalisation and model evaluation

Training improves performance on known examples. Evaluation asks whether that improvement carries to new examples.

Low training loss is not enough



Training performance

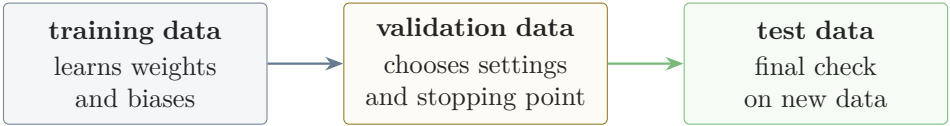
How well the model handles examples that influenced its parameter updates.

Generalisation

How well the trained model handles new examples from the same setting.

> **Overfitting** occurs when training performance improves while performance on held-out examples becomes worse.

Training, validation, and test data have different jobs



1. Training data changes the parameters directly.
2. Validation data changes modelling choices, not the fitted weights directly.
3. Test data should change nothing after its result is seen.

The three sets can have the same format. Their difference is which decisions they are allowed to influence.

Each epoch creates a new checkpoint for validation

The parameter values saved after an epoch form a **checkpoint**.

During one epoch, the optimiser processes the 600 training images in 19 batches:

$$\boldsymbol{\theta}_{t-1} \xrightarrow[19 \text{ updates}]{600 \text{ training images}} \boldsymbol{\theta}_t.$$

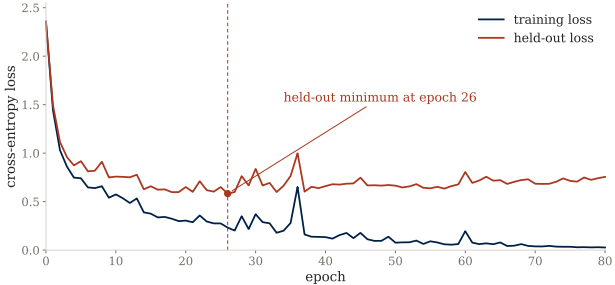
The resulting parameters are then evaluated on 300 different images:

$$\big(300 \text{ validation images}, \boldsymbol{\theta}_t\big) \xrightarrow{\text{forward pass only}} L_{\text{validation}}(\boldsymbol{\theta}_t).$$

1. Training images create the next checkpoint $\boldsymbol{\theta}_t$.
2. Validation images evaluate that checkpoint but do not change it.
3. Repeating this process for 80 epochs produces a sequence of validation losses to compare.

A sequence of checkpoint losses reveals when performance on unseen images stops improving.

This run reaches its lowest validation loss at epoch 26



Run: 600 training and 300 validation (held-out) Fashion-MNIST images; $784 \to 32 \to 10$; $B = 32$; 80 epochs.

At the minimum: epoch 26

Training loss is 0.232; validation loss reaches its minimum of 0.583.

By the final epoch: epoch 80

Training loss falls to 0.027, but validation loss rises to 0.755.

$$t^* = \arg\min_t L_{\text{validation}}(\boldsymbol{\theta}_t) = 26, \qquad 26 \times 19 = 494 \text{ training updates.}$$

The value 26 is observed, not preselected; early stopping keeps $\boldsymbol{\theta}_{26}$.

After epoch 26, more training worsens validation performance

From epoch 26 to epoch 80, the two losses move in opposite directions:

$$\underbrace{0.232 \longrightarrow 0.027}_{\text{training loss falls}} \qquad \text{but} \qquad \underbrace{0.583 \longrightarrow 0.755}_{\text{validation loss rises}}.$$

The optimiser is fitting the training images more closely, but predictions on unseen images are becoming worse.
The validation result determines the next action:

1. **Stop and restore $\boldsymbol{\theta}_{26}$.** Do not keep the parameters from epoch 80.
2. **If validation performance is still insufficient, change the training procedure.** Use more representative data or add regularisation such as weight decay or dropout.
3. **Train again and compare on validation data.** The test set remains untouched while these choices are made.

> **A change is useful only if it improves performance on validation data, not merely on the training examples.**

Use the test set once—after every modelling choice is fixed

Before looking at test performance, validation data must already have selected:

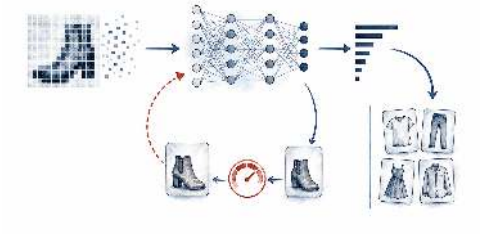
1. the input preparation and network architecture;
2. the optimiser, learning rate, batch size, and regularisation;
3. the stopping rule and saved checkpoint;
4. the evaluation measure and any decision threshold.

$$\underbrace{\text{fixed model and evaluation rule}}_{\text{no further choices}} \xrightarrow{\text{one test evaluation}} \underbrace{\text{final performance estimate}}_{\text{report the result}}.$$

If a test result leads to retraining or another modelling choice, the test set has become validation data. Final evaluation then requires a new untouched test set.

Training fits the parameters; validation chooses the model and checkpoint; the test set reports the final result once.

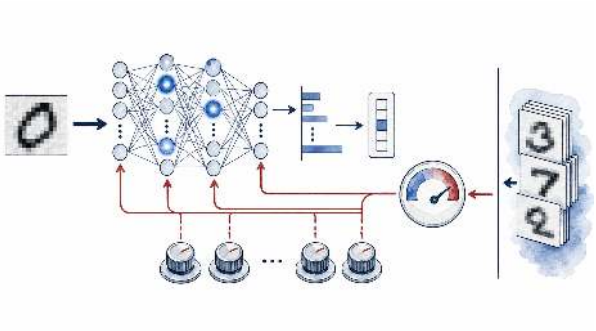
The learning and evaluation system is now complete



1. The forward pass produces probabilities and a loss.
2. Backpropagation computes the gradient; the optimiser updates the parameters.
3. Validation and test data assess whether the fitted procedure generalises beyond the training examples.

We can now explain how a network learns and how its performance is judged. Next: how should the architecture read an image?

Images require a different connection pattern



1. A dense layer reads a flattened list of pixel values.
2. Visual evidence often depends on neighbouring pixels.
3. The same edge or texture can appear at many positions.

> The new problem is architectural: preserve local neighbourhoods and reuse a learned detector across the image. The loss, backpropagation, and parameter updates remain unchanged.

The image task: classify one pet photograph by breed

1. **Input:** one colour photograph.
2. **Output:** one of 37 breed labels.
3. **Available data:** 7,349 labelled photographs.

Pose, crop, lighting, and background may change while the correct breed stays the same.

The classifier must retain evidence that distinguishes breeds while reducing sensitivity to changes that should not alter the label.

Oxford–IIIT Pet dataset; Parkhi et al. (2012).

A network builds one prediction through a forward chain

$$\begin{aligned} \boldsymbol{x} &\longrightarrow \boldsymbol{z}_1 = W_1\boldsymbol{x} + \boldsymbol{b}_1 \longrightarrow \boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z}_1) \\ &\longrightarrow \boldsymbol{z}_2 = W_2\boldsymbol{h} + \boldsymbol{b}_2 \longrightarrow \boldsymbol{p} = \mathrm{softmax}(\boldsymbol{z}_2) \longrightarrow \ell = -\log p_y. \end{aligned}$$

1. The **architecture** specifies the layers, connections, and activation functions. The parameters θ are the weights and biases learned from data.
2. Matrix-plus-bias layers produce scores. A nonlinear activation such as ReLU allows successive layers to build more flexible functions.
3. Softmax turns the final scores into class probabilities. The observed label selects p_y, and cross-entropy converts it into one loss.

One example follows the complete chain $\boldsymbol{x} \to \boldsymbol{p} \to \ell$: input, prediction, then error.

Training reduces average loss; evaluation checks new data

$$\underbrace{L_B(\theta) = \frac{1}{B}\sum_{i=1}^{B}\Big[-\log p_{i,y_i}\Big]}_{\text{average loss for one batch}} \qquad \underbrace{\boldsymbol{g} = \nabla_\theta L_B(\theta)}_{\text{backpropagation}} \qquad \underbrace{\theta \leftarrow \theta - \eta\boldsymbol{g}}_{\text{optimiser update}} .$$

1. The same parameters process every example in a batch; the example losses are averaged before one update.
2. Backpropagation computes how the average loss changes with every parameter. The optimiser uses that gradient to update the parameters.
3. Repeated batches train the network. Validation data selects modelling choices and a checkpoint; the test set measures final performance once.

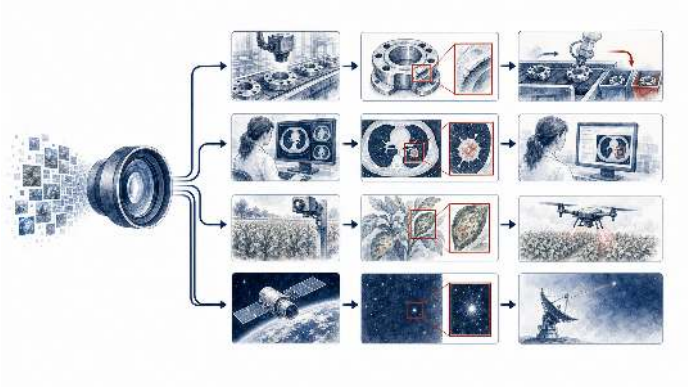
CNNs keep this learning and evaluation procedure. What changes is the way image values are connected to hidden units.

Part I

Why images need convolution

A CNN uses two properties of images: neighbouring pixels form patterns, and the same pattern may occur at different positions.

Computer vision supports decisions in many fields



factory line
inspect every part; remove the flawed one

clinic
flag the region a radiologist should review

field
treat only the plants that show disease

telescope
notice the one light that changed overnight

Each application requires the same basic operation: convert an image into information that supports a decision.

The lecture addresses four questions

I **Why do images require different connections?** Nearby pixels form patterns, and those patterns may appear anywhere.

II **How does a CNN produce a class decision?** Filters produce feature maps; deeper layers combine and summarise them.

III **How can a CNN learn from limited labelled images?** Protect the split, add valid variation, and reuse pretrained weights.

IV **How should its performance be evaluated?** Locate errors, inspect influential regions, and test suspected shortcuts.

The Oxford–IIIT Pet classification task provides one example throughout: the input is a photograph and the output is one of 37 breed labels.

Task and dataset: classify 37 pet breeds

Input: one photograph. Output: one of 37 breed names.

	Fashion-MNIST, Day 1	Oxford–IIIT Pet, today
image	28×28 grey	colour, sizes vary, about 400×400
classes	10, visibly different	37 breeds, several nearly alike
images	70,000	7,349
per class	7,000	about 199

Compared with Fashion-MNIST, the images are larger, the class differences are finer, and substantially fewer labelled examples are available. Parkhi et al. (2012).

CNNs change the architecture, not the training procedure

image $\longrightarrow$ network $\longrightarrow$ scores $\longrightarrow$ loss $\longrightarrow$ parameter update

The training procedure remains the same

- weighted sums, biases, and ReLU;
- softmax and cross-entropy;
- backpropagation and the optimiser;
- validation and final testing.

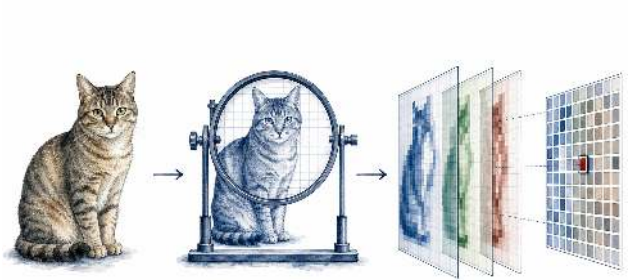
The connection pattern changes

- analyse nearby pixels together;
- detect the same pattern at different positions;
- reuse a detector instead of relearning it at every position.

Forward propagation, loss, backpropagation, and optimisation remain unchanged. The new issue is how image values are connected to units.

A photograph enters the network as an RGB tensor

$$\text{image shape} = H \times W \times C$$



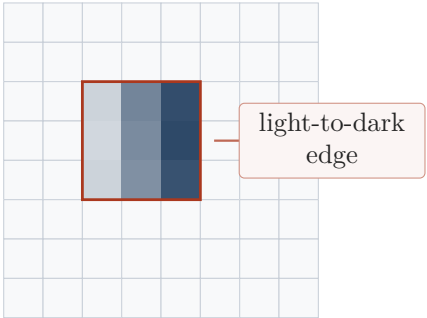
- H rows and W columns locate a position.
- C records the measurements stored there.
- For an RGB photograph, $C = 3$.

$$64 \times 64 \times 3 = 12{,}288$$

input values in one resized photograph

Parts such as ears and muzzles are not supplied as input features. The network must learn useful combinations of pixel values.

Visual patterns are defined by neighbouring pixels



one channel of the photograph

One brightness value says little on its own. A small arrangement of nearby values can describe a contrast, edge, curve, or texture.

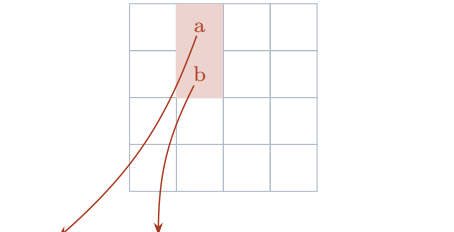
These local patterns form parts of larger structures: fur around an ear, the boundary of a lesion, a crack in a component, or a spot on a leaf.

Neighbourhood
A small group of nearby positions analysed together.

The network architecture should preserve spatial relationships between neighbouring pixels.

Flattening keeps values but removes the neighbourhood rule

in the image, a and b touch



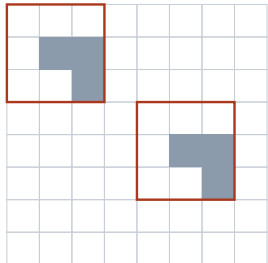
in the list, the same two cells sit four apart
(one channel drawn; colour
makes the list three times longer)

1. Flattening turns the grid into one long list.
2. No number is deleted; the grid could be reconstructed if its shape were retained.
3. A dense unit receives a and b as separate inputs. Its connections do not record that they were adjacent.

A dense network can learn local relationships, but locality is not built into its connection rule.

The values survive, so the layer could learn neighbourhoods from data. It simply starts with no idea that they exist.

Dense layers use different weights at different positions



the same local pattern at two positions

The local patterns are identical. Only their position changes.

After flattening, the two copies occupy different parts of the input list:

$$\text{position A} \longrightarrow \text{weights } \boldsymbol{w}_A,$$
$$\text{position B} \longrightarrow \text{weights } \boldsymbol{w}_B.$$

A dense network can learn both cases, but it must relearn the same pattern for each position.

A defect, lesion, or animal may appear anywhere in an image. Position-specific weights therefore duplicate the task of learning the same detector.

A useful local detector should work across the image

The dense connection rule reveals a more basic problem: moving a pattern changes which weights process it.

Dense connection rule

$$\text{pattern at A} \longrightarrow \boldsymbol{w}_A,$$
$$\text{same pattern at B} \longrightarrow \boldsymbol{w}_B$$

two positions, two unrelated sets of weights

Image-appropriate rule

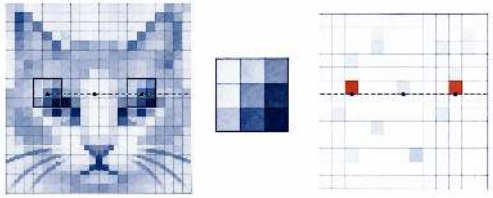
$$\text{pattern at A} \longrightarrow \boldsymbol{K},$$
$$\text{same pattern at B} \longrightarrow \boldsymbol{K}$$

one local detector, reused across positions

This requires two properties: inspect a local neighbourhood and move the same weights across the image. Together, these properties define convolution.

Convolution uses local connections and shared weights

Convolution implements the required connection rule with a learned filter.



Local connections

Each response uses a small window, so nearby measurements are analysed together.

Shared weights

The same learned detector is applied at every position, so the same pattern is found wherever it appears.

> **Local connections combine nearby pixels; shared weights find the same pattern in different parts of the image.**

Weight sharing sharply reduces the parameter count

The saving follows directly from using one learned filter at every position. Compare two first layers on the $64 \times 64 \times 3$ image.

Flatten $\rightarrow$ 128 **dense units**

1,572,992

one separate weight for every input value and every unit

128 **local** 3×3 **RGB filters**

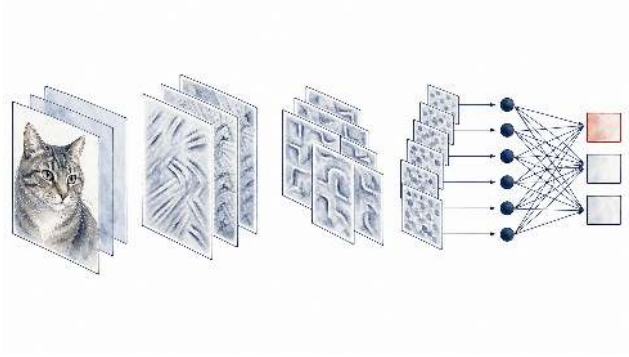
3,584

$(3 \times 3 \times 3 + 1) \times 128$, reused at every position

Convolution examines every position with shared weights. A CNN then stacks such layers to turn local responses into a decision.

▸ Parameter calculation

A CNN converts pixels into class scores in stages



photograph	**local maps**	**richer maps**	**decision**
aligned RGB measurements	where simple patterns match	combinations of local features	one score per breed

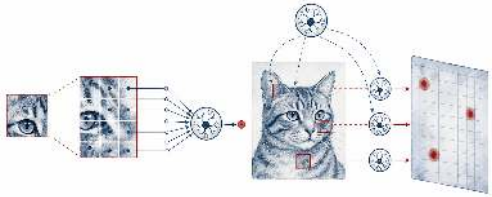
Convolution provides an appropriate connection pattern for images. The next section develops the complete classification pipeline.

Part II

How a CNN produces a class decision

Filters produce feature maps; deeper layers combine them; pooling and the classifier convert them into class probabilities.

A convolutional unit applies a weighted sum to a local patch



It uses the same operations as Day 1:

$$z = \sum_{u,v,c} P_{uvc} K_{uvc} + b, \qquad a = \mathrm{ReLU}(z).$$

- P: the small image patch under the unit
- K: learned weights, called a **filter**
- a: the response at this position

> **A convolutional unit uses the same weighted sum and activation as a dense unit. It differs by reading a local patch and sharing its weights across positions.**

Matching cells are multiplied, then all products are added

Multiply each window value by the filter weight in the same cell.

Window P $\qquad\qquad$ **Filter** K

$$\begin{bmatrix} 0.1 & 0.5 & 0.9 \\ 0.1 & 0.5 & 0.9 \\ 0.1 & 0.5 & 0.9 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

left column	middle column	right column
$3(0.1)(-1) = -0.3$	$3(0.5)(0) = 0$	$3(0.9)(1) = 2.7$

$$\underbrace{-0.3 + 0 + 2.7}_{\text{add all nine products}} = \boxed{2.4}.$$

One full window and one filter produce one weighted sum. Bias and ReLU come after this local comparison.

▸ Open the nine products

A slight edge produces a weak response

The same filter now visits a different window. Its right side is only slightly brighter, so we expect a smaller answer.

New window P_{weak} **Same filter** K

$$\begin{bmatrix} 0.4 & 0.5 & 0.5 \\ 0.4 & 0.5 & 0.5 \\ 0.4 & 0.5 & 0.5 \end{bmatrix} \odot \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

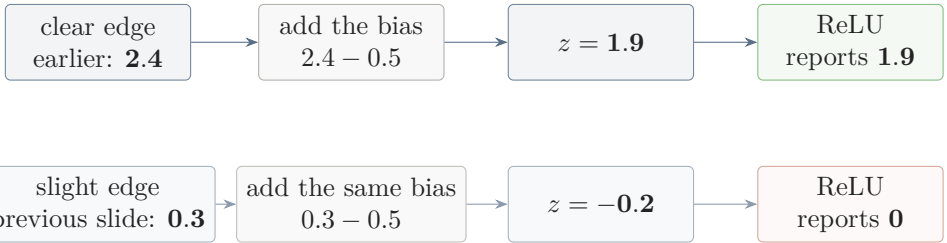
left column	**middle column**	**right column**
$3(0.4)(-1) = -1.2$	$3(0.5)(0) = 0$	$3(0.5)(1) = 1.5$

$$-1.2 + 0 + 1.5 = \boxed{0.3}.$$

This 0.3 **is the weak response used on the next slide. No bias or ReLU has been applied yet.**

The bias sets the threshold; ReLU applies it

The same filter gave 2.4 for the clear edge and 0.3 for the slight edge. Now give it $b = -0.5$.



$\mathrm{ReLU}(z) = \max(0, z)$. With $b = -0.5$, only a weighted sum above 0.5 stays positive. Training learns this cutoff through the bias.

ReLU passes a positive response and suppresses a negative response. The bias changes the response level required for an activation to pass forward.

We choose the connection rule; training learns the values

We choose the design

- the window size;
- where the filter is reused;
- how many filters the layer has.

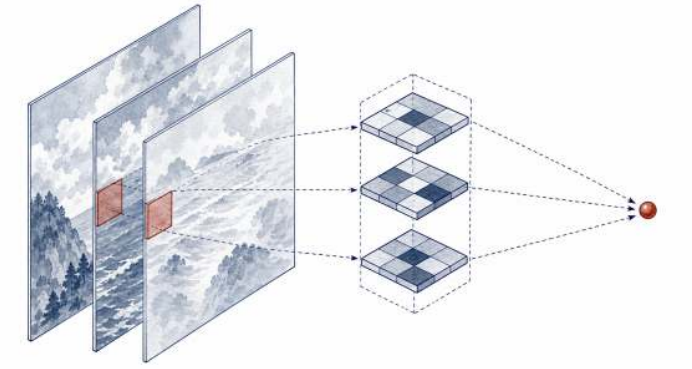
Training learns the numbers

- every filter weight;
- one bias for each filter;
- values that reduce the task loss.

predict $\longrightarrow$ measure error $\longrightarrow$ update weights

The Day 1 learning loop is unchanged. Only the connection pattern is different.

A convolutional filter spans all input channels



A 3×3 RGB patch contains

$$3 \times 3 \times 3 = 27$$

values.
One filter therefore has:

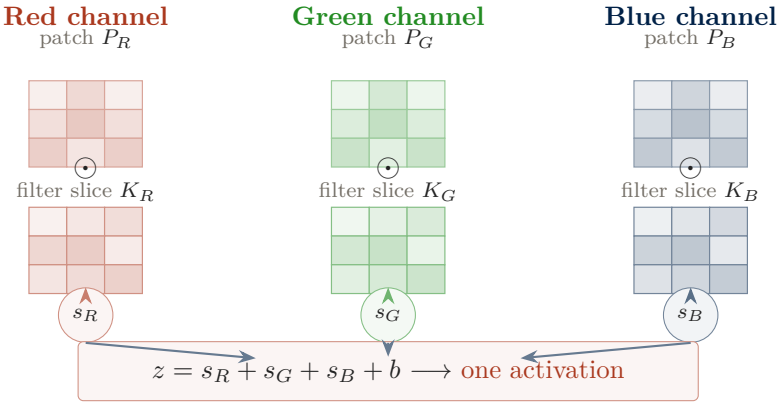
$$\underbrace{27}_{\text{weights}} + \underbrace{1}_{\text{bias}} = 28$$

learned parameters.

The filter is translated across height and width. At each spatial position, it reads all three colour channels together.

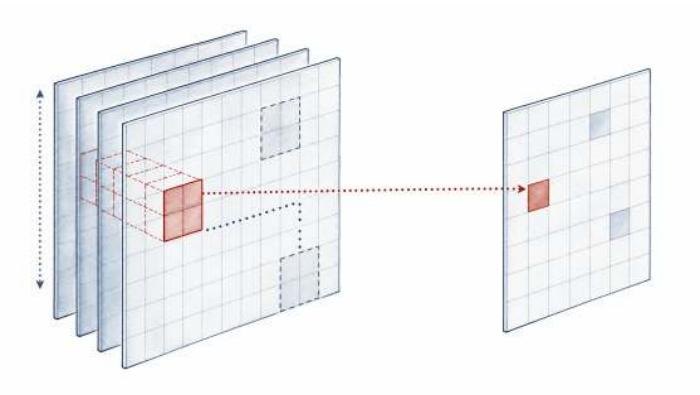
It is one filter with three slices, not three separate filters.

For colour, one filter has one grid per input channel



A 3×3 filter on RGB has $3 \times 3 \times 3 = 27$ weights. It spans all three channels. These are three slices of *one filter*, not three separate filters.

Scanning one filter produces one feature map

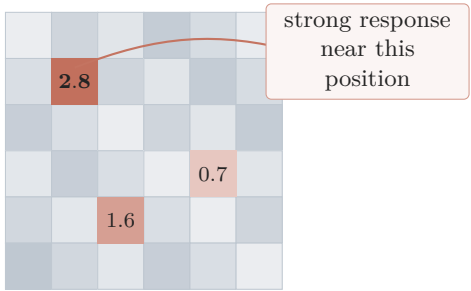


apply
place the same filter on each neighbourhood

compute
compute one response at each position

arrange
arrange responses in a 2D feature map

A feature map records strength and location



one filter → one two-dimensional map

Value: how strongly the local window matched.

Cell position: approximately where it matched.

If the pattern moves, the strong response moves to another cell. The map therefore keeps approximate position as well as response strength.

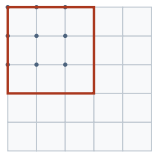
A feature map is a grid of learned responses, not a reconstructed photograph or a set of literal named animal parts.

Padding changes border coverage; stride changes visited positions

We count the positions the filter can visit; each position produces one feature-map cell.

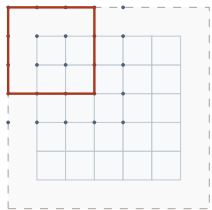
Toy input 5×5, filter 3×3. Blue dots mark valid top-left starts; the red square shows one filter placement.

no padding



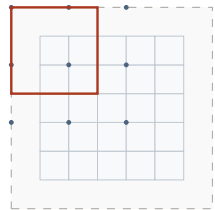
$p = 0,\ s = 1 \Rightarrow 3 \times 3$ map

pad by one cell



$p = 1,\ s = 1 \Rightarrow 5 \times 5$ map

stride two



$p = 1,\ s = 2 \Rightarrow 3 \times 3$ map

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

m: output size, n: input size, k: filter size, p: padding, s: stride.

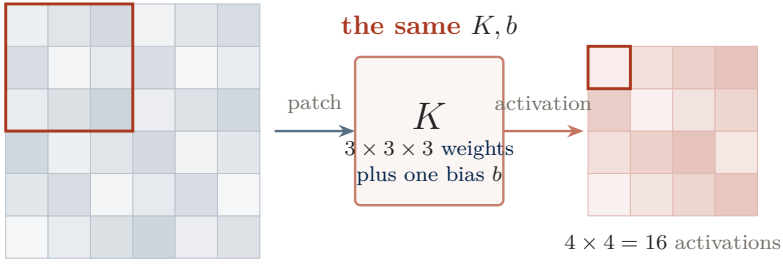
Padding p adds border cells; **stride** s sets the step between filter positions. Apply the formula separately to height and width; $\lfloor \cdot \rfloor$ means round down.

▸ Derive the general count

Moving the filter changes the patch, not the learned weights

Learn one filter once, then reuse it at every valid position.

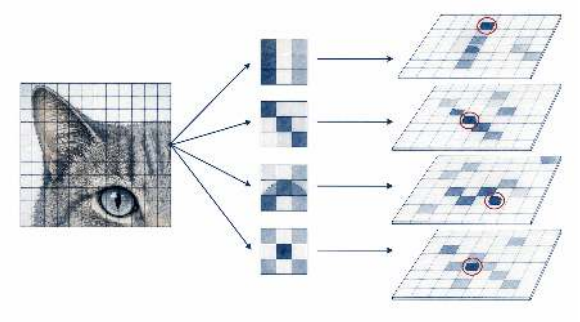
a different patch at each position **one output cell per position**



Spatial size: $m = \lfloor (6 + 2(0) - 3)/1 \rfloor + 1 = 4$ positions per side, so the feature map is 4×4. $(p = 0,\ s = 1)$

$\underbrace{4 \times 4}_{16 \text{ produced activations}}$ all reuse $\underbrace{(3 \times 3 \times 3) + 1}_{28 \text{ stored learned parameters}}$

Different filters make different maps



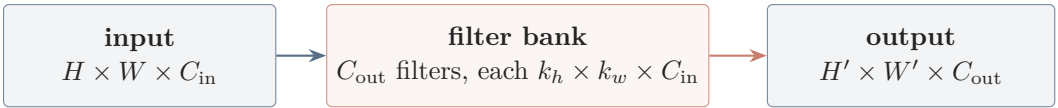
Same input
every filter reads it

Different weights
each filter asks a different question

Different maps
one feature map per filter

> This picture uses four filters, so it produces four feature maps. In general, C_{out} filters make C_{out} output channels.

Spatial size and channel depth have different causes



Spatial size H', W'
input size, filter size, padding, stride

Channel depth C_{out}
number of filters in the layer

C_{in} determines the depth of each filter.

C_{out} determines how many maps are stacked.

▸ Work through one filter bank

Appendix

Calculations behind the mechanism

These slides are optional during the main lecture.

Appendix: the loss depends on every softmax probability

For logits $\boldsymbol{z}$, softmax and cross-entropy are

$$p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}, \qquad \ell = -\sum_j y_j \log p_j.$$

Fix one logit z_i. Changing z_i changes the softmax denominator, so it changes every probability p_j, not only p_i. The chain rule must therefore include every path from z_i to the loss:

$$\frac{\partial \ell}{\partial z_i} = \sum_j \underbrace{\frac{\partial \ell}{\partial p_j}}_{\text{loss response to } p_j} \underbrace{\frac{\partial p_j}{\partial z_i}}_{\text{softmax response to } z_i}.$$

The first factor follows directly from the logarithm:

$$\ell = -\sum_j y_j \log p_j \quad \Longrightarrow \quad \frac{\partial \ell}{\partial p_j} = -y_j \frac{1}{p_j} = -\frac{y_j}{p_j}.$$

Next, calculate how z_i changes p_i and every other p_j.

◂ Back to the first backward signal

▸ Differentiate softmax

Appendix: one logit changes every softmax probability

Write the common denominator as

$$S = \sum_k e^{z_k}, \qquad p_j = \frac{e^{z_j}}{S}, \qquad \frac{\partial S}{\partial z_i} = e^{z_i}.$$

Its own probability: $j = i$

Both the numerator and denominator depend on z_i:

$$\begin{aligned}\frac{\partial p_i}{\partial z_i} &= \frac{e^{z_i} S - e^{z_i} e^{z_i}}{S^2} \\ &= \frac{e^{z_i}}{S}\left(1 - \frac{e^{z_i}}{S}\right) \\ &= p_i(1 - p_i).\end{aligned}$$

Another probability: $j \neq i$

The numerator is fixed; only the denominator depends on z_i:

$$\begin{aligned}\frac{\partial p_j}{\partial z_i} &= -\frac{e^{z_j} e^{z_i}}{S^2} \\ &= -\frac{e^{z_j}}{S}\frac{e^{z_i}}{S} \\ &= -p_j p_i.\end{aligned}$$

Increasing z_i raises p_i and lowers every p_j with $j \neq i$. Softmax couples the class probabilities through their shared denominator.

◂ Back to the chain-rule setup

▸ Combine the derivatives

Appendix: the terms simplify to $p_i - y_i$

Separate the p_i term from the p_j terms and insert the derivatives:

$$\begin{aligned}
\frac{\partial \ell}{\partial z_i} &= \left(-\frac{y_i}{p_i}\right) p_i(1-p_i) + \sum_{j \neq i} \left(-\frac{y_j}{p_j}\right)(-p_j p_i) \\
&= -y_i(1-p_i) + p_i \sum_{j \neq i} y_j \\
&= -y_i(1-p_i) + p_i(1-y_i) \quad \text{since } \sum_{j \neq i} y_j = 1 - y_i \\
&= -y_i + y_i p_i + p_i - p_i y_i \\
&= p_i - y_i.
\end{aligned}$$

Applying this result to every logit gives $\boxed{\partial \ell / \partial \boldsymbol{z} = \boldsymbol{p} - \boldsymbol{y}}$.

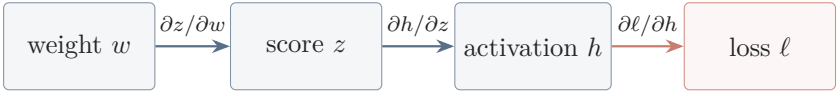
The observed class has gradient $p_i - 1$; every other class has gradient p_i.

◂ Back to the softmax derivative

◂ Return to the first backward signal

Appendix: the chain rule along one path

Consider a first-layer weight w on one path to the loss:



$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial h} \times \frac{\partial h}{\partial z} \times \frac{\partial z}{\partial w}.$$

Along one path

Multiply the local effects.

When paths meet

Add their contributions.

Backpropagation calculates each local effect once and reuses it where paths share the same later calculations.

◂ Back to the backward-path overview

Appendix: one arriving gradient serves two purposes

For $z = wa + b$, backpropagation arrives with $g_z = \partial \ell / \partial z$. This value measures how the loss responds to z; the chain rule combines it with each local effect.

1. Continue the backward pass

Since a affects z through w,

$$\frac{\partial \ell}{\partial a} = \underbrace{\frac{\partial \ell}{\partial z}}_{g_z} \underbrace{\frac{\partial z}{\partial a}}_{w} = g_z w.$$

This result is sent to the preceding layer.

2. Prepare this layer's update

The same arriving value gives

$$\frac{\partial \ell}{\partial w} = g_z a, \qquad \frac{\partial \ell}{\partial b} = g_z.$$

These gradients are stored for the optimiser.

For $a = 2$, $w = 0.5$, and $g_z = 0.731$,

$$\frac{\partial \ell}{\partial a} = 0.3655, \qquad \frac{\partial \ell}{\partial w} = 1.462, \qquad \frac{\partial \ell}{\partial b} = 0.731.$$

The same gradient continues backward and supplies this layer's parameter gradients.

◂ Back to the two backward tasks

▸ Continue to the weight calculation

Appendix: deriving the weight gradient 1.462

Choose weight $w = (W_2)_{1,2}$, connecting hidden value h_2 to class-1 score $z_{2,1}$.

1. The output backward signal was
$$\frac{\partial \ell}{\partial z_{2,1}} = 0.731.$$
This 0.731 came from the first entry of $\boldsymbol{p} - \boldsymbol{y}$.

2. The forward pass stored
$$h_2 = 2.$$
This is the value that crossed weight w.

3. The weight gradient is therefore
$$\boxed{\frac{\partial \ell}{\partial w} = 0.731 \times 2 = 1.462.}$$

Weight gradient = arriving backward signal × stored forward input.

◂ Back to the weight-layer overview

▸ Interpret the value 1.462

Appendix: interpreting the gradient with a small change

The derivative 1.462 describes the local change in loss per unit change in the weight. To interpret it, choose a small hypothetical change:

$$\Delta w = 0.01.$$

> 0.01 is not produced by the network and is not a learned value. It is chosen only as a simple small change for interpreting the derivative.

For a small change,

$$\Delta \ell \approx \frac{\partial \ell}{\partial w} \Delta w = 1.462 \times 0.01 = 0.01462 \approx 0.0146.$$

Because the value is positive, increasing w would raise the loss. Gradient descent therefore moves w in the opposite direction.

Choosing 0.001 instead would predict a loss change of 0.001462. The derivative 1.462 itself would not change.

◂ Back to the derivative slide

▸ Continue to all output weights

Appendix: the complete 2×3 output-weight gradient

For every connection from hidden unit h_j to class score $z_{2,c}$,

$$\frac{\partial \ell}{\partial (W_2)_{c,j}} = \underbrace{\frac{\partial \ell}{\partial z_{2,c}}}_{\text{arriving backward signal}} \times \underbrace{h_j}_{\text{stored forward input}}.$$

	from $h_1 = 1$	from $h_2 = 2$	from $h_3 = 0$
to $z_{2,1}$, signal $+0.731$	$+0.731$	$+1.462$	0
to $z_{2,2}$, signal -0.731	-0.731	-1.462	0

$$\frac{\partial \ell}{\partial W_2} = \begin{bmatrix} 0.731 & 1.462 & 0 \\ -0.731 & -1.462 & 0 \end{bmatrix}.$$

Column 2 is twice column 1 because $h_2 = 2h_1$. Column 3 is zero because this example produced $h_3 = 0$.

◂ Back to the weight-layer overview

▸ Carry the signal to hidden units

Appendix: each hidden unit combines its downstream effects

Hidden unit h_j feeds both class scores, so its two downstream contributions must be added:

$$\frac{\partial \ell}{\partial h_j} = (W_2)_{1,j} \frac{\partial \ell}{\partial z_{2,1}} + (W_2)_{2,j} \frac{\partial \ell}{\partial z_{2,2}}.$$

hidden unit	through $z_{2,1}$	through $z_{2,2}$	total
h_1	$1(+0.731)$	$0(-0.731)$	0.731
h_2	$0.5(+0.731)$	$0.5(-0.731)$	0
h_3	$-1(+0.731)$	$1(-0.731)$	-1.462

$$\frac{\partial \ell}{\partial \boldsymbol{h}} = \begin{bmatrix} 0.731 \\ 0 \\ -1.462 \end{bmatrix} = W_2^\top \begin{bmatrix} 0.731 \\ -0.731 \end{bmatrix}.$$

◂ Back to the weight-layer overview

▸ Explain the zero for h_2

Appendix: hidden unit 2 has cancelling effects

Hidden unit h_2 connects to both class scores with weight 0.5:

$$\frac{\partial \ell}{\partial h_2} = \underbrace{0.5(+0.731)}_{\text{path to } z_{2,1}} + \underbrace{0.5(-0.731)}_{\text{path to } z_{2,2}} = 0.$$

The same result can be read in the forward direction:

$$h_2 \text{ rises by } \delta \quad \Longrightarrow \quad \boldsymbol{z}_2 \text{ rises by } \begin{bmatrix} 0.5\delta \\ 0.5\delta \end{bmatrix}.$$

Softmax depends on the difference between the class scores. Raising both scores by the same amount changes neither probability, so the loss is unchanged.

$h_2 = 2$ means the unit is active, while $\dfrac{\partial \ell}{\partial h_2} = 0$ means its effects cancel.

◂ Back to the weight-layer overview

▸ Continue through ReLU

Appendix: ReLU has slope 1 on one side and 0 on the other

For one unit,

$$h = \mathrm{ReLU}(z) = \max(0, z) = \begin{cases} z, & z > 0, \\ 0, & z < 0. \end{cases}$$

Positive side: $z > 0$

Here $h = z$. If z changes slightly by Δz, then

$$\Delta h = \Delta z \qquad \Longrightarrow \qquad \mathrm{ReLU}'(z) = \frac{\Delta h}{\Delta z} = 1.$$

A change entering ReLU passes through unchanged.

Negative side: $z < 0$

Here $h = 0$. A small change in z still gives $h = 0$, so

$$\Delta h = 0 \qquad \Longrightarrow \qquad \mathrm{ReLU}'(z) = \frac{\Delta h}{\Delta z} = 0.$$

The output does not respond to the change.

At $z = 0$, the two sides meet at a corner; implementations conventionally use a slope of 0.

> **The sign saved during the forward pass determines the local ReLU slope used during backpropagation.**

◂ Back to the ReLU transition

▸ Apply the chain rule

Appendix: the chain rule turns ReLU slopes into gradient gates

For one unit, the forward path is

$$z \longrightarrow h = \mathrm{ReLU}(z) \longrightarrow \ell.$$

Backpropagation arrives with

$$g_h = \frac{\partial \ell}{\partial h},$$

which measures how the loss responds to the ReLU output h.
To continue to the earlier score z, apply the chain rule:

$$\underbrace{\frac{\partial \ell}{\partial z}}_{g_z} = \underbrace{\frac{\partial \ell}{\partial h}}_{g_h} \underbrace{\frac{\partial h}{\partial z}}_{\mathrm{ReLU}'(z)} = g_h \, \mathrm{ReLU}'(z).$$

$z > 0 \quad \Longrightarrow \quad g_z = g_h \times 1 = g_h$

the incoming gradient passes

$z < 0 \quad \Longrightarrow \quad g_z = g_h \times 0 = 0$

the incoming gradient stops

Apply the same chain-rule multiplication independently to each ReLU unit.

◂ Back to the ReLU slopes

▸ Apply the rule to all three units

Appendix: ReLU passes or blocks each hidden signal

The hidden-unit signals must pass through the ReLU decisions stored during the forward pass:

$$\frac{\partial \ell}{\partial \boldsymbol{z}_1} = \frac{\partial \ell}{\partial \boldsymbol{h}} \odot \mathrm{ReLU}'(\boldsymbol{z}_1).$$

unit	saved $z_{1,j}$	incoming $\partial \ell / \partial h_j$	ReLU slope	outgoing $\partial \ell / \partial z_{1,j}$
1	1	0.731	1	0.731
2	2	0	1	0
3	−0.5	−1.462	0	0

$$\frac{\partial \ell}{\partial \boldsymbol{z}_1} = \begin{bmatrix} 0.731 \\ 0 \\ 0 \end{bmatrix}.$$

Unit 2 passes a zero it already received. Unit 3 receives a non-zero signal, but its ReLU gate was closed, so the signal is blocked.

◂ Back to the ReLU overview

▸ Continue to the first layer

Appendix: the complete first-layer gradient

For a first-layer connection from input x_i to score $z_{1,j}$,

$$\frac{\partial \ell}{\partial (W_1)_{j,i}} = \underbrace{\frac{\partial \ell}{\partial z_{1,j}}}_{\text{destination signal}} \times \underbrace{x_i}_{\text{source value}} .$$

Here $\boldsymbol{x} = (2, 1)^\top$ and $\partial \ell / \partial \boldsymbol{z}_1 = (0.731, 0, 0)^\top$.

	from $x_1 = 2$	from $x_2 = 1$
to $z_{1,1}$, signal 0.731	1.462	0.731
to $z_{1,2}$, signal 0	0	0
to $z_{1,3}$, signal 0	0	0

$$\frac{\partial \ell}{\partial W_1} = \begin{bmatrix} 1.462 & 0.731 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \qquad \frac{\partial \ell}{\partial \boldsymbol{b}_1} = \begin{bmatrix} 0.731 \\ 0 \\ 0 \end{bmatrix}.$$

Back to the weight-layer overview

Count the non-zero gradients

Appendix: one example gives nine non-zero gradients

	parameters	non-zero gradients	why the rest are zero
W_1	6	2	units 2 and 3 pass no signal to layer 1
$\boldsymbol{b}_1$	3	1	the same two units
W_2	6	4	the column from $h_3 = 0$ is zero
$\boldsymbol{b}_2$	2	2	both class-score signals are non-zero
total	**17**	**9**	

One example gives a narrow instruction: eight parameters receive zero gradient from this example. Another example may activate different units and produce a different pattern of non-zero gradients.

This is why training uses many examples, batches, and repeated epochs rather than relying on one example and one update.

◂ Back to the weight-layer overview

▸ Active, inactive, and dead ReLUs

Appendix: active, inactive, and dead ReLU units

Active on one example: $z > 0$. The unit passes its value and can receive a gradient from that example.

Inactive on one example: $z < 0$. The unit outputs zero and learns nothing from that particular example. This is normal.

Dead across the dataset: $z < 0$ for every example. The unit never activates and receives no gradient, so its weights cannot recover through ordinary ReLU training.

A very large learning-rate update can push a bias far below zero. Leaky ReLU and GELU retain some slope for negative inputs, which reduces this risk.

◂ Back to the ReLU overview

▸ Check a gradient-descent update

Appendix: isolate one small gradient-descent update

These numbers come from one deliberately simplified gradient-descent update. Only the two output biases change; every weight and every other bias remains fixed.

Before the update,

$$\boldsymbol{b}_2^{\text{old}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} 0.731 \\ 0.269 \end{bmatrix}, \qquad \boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

For softmax with cross-entropy, the output-bias gradient is

$$\frac{\partial \ell}{\partial \boldsymbol{b}_2} = \boldsymbol{p} - \boldsymbol{y} = \begin{bmatrix} 0.731 \\ -0.731 \end{bmatrix}.$$

With learning rate $\eta = 0.05$, gradient descent gives

$$\boldsymbol{b}_2^{\text{new}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.05 \begin{bmatrix} 0.731 \\ -0.731 \end{bmatrix} = \begin{bmatrix} -0.03655 \\ 0.03655 \end{bmatrix}.$$

◂ Back to the forward-pass check

▸ Continue the calculation

Appendix: the new biases produce the new loss

The weights and hidden values are unchanged. Therefore the change in the two output biases is added directly to the old logits:

$$\begin{aligned} \boldsymbol{z}_2^{\text{new}} &= \boldsymbol{z}_2^{\text{old}} + \left(\boldsymbol{b}_2^{\text{new}} - \boldsymbol{b}_2^{\text{old}}\right) \\ &= \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} -0.03655 \\ 0.03655 \end{bmatrix} = \begin{bmatrix} 1.96345 \\ 1.03655 \end{bmatrix}. \end{aligned}$$

The observed label is class 2, so softmax selects the second probability:

$$p_{\text{true}}^{\text{new}} = \frac{e^{1.03655}}{e^{1.96345} + e^{1.03655}} = 0.28355 \approx \boxed{0.284}.$$

Cross-entropy converts that probability into the new loss:

$$\ell^{\text{new}} = -\log(0.28355) = 1.26035 \approx \boxed{1.260}.$$

This calculation is the source of the updated probability 0.284 and loss 1.260 shown on the main slide.

◂ Back one step ◂ Return to the forward-pass check

Selected references

- Goodfellow, Bengio, and Courville (2016), *Deep Learning*, Chapters 6–8.
- Bishop and Bishop (2024), *Deep Learning: Foundations and Concepts*, Chapters 7–8.
- He et al. (2015), "Delving Deep into Rectifiers."
- Prechelt (1998), "Early Stopping – but when?"
- Parkhi et al. (2012), "Cats and Dogs."

The numerical example continues the network and loss calculation from Day 1.

Appendix: the dense and convolutional parameter counts

For a $64 \times 64 \times 3$ input,

$$D = 64 \times 64 \times 3 = 12{,}288$$

input values reach every dense unit. With 128 units:

$$\underbrace{(D+1)}_{\substack{D \text{ weights} \\ +\ 1 \text{ bias}}} \times 128 = (12{,}288 + 1) \times 128 = 1{,}572{,}992.$$

For 128 local 3×3 RGB filters:

$$\underbrace{(3 \times 3 \times 3 + 1)}_{27 \text{ weights} + 1 \text{ bias}} \times 128 = 3{,}584.$$

The convolution still produces many activations because every filter is scanned across the image. The small parameter count refers to the weights learned once and reused.

Return to the comparison

Appendix: calculation of a filter response

$$\underbrace{\begin{bmatrix} 0.1 & 0.4 & 0.9 \\ 0.2 & 0.6 & 0.9 \\ 0.0 & 0.5 & 0.9 \end{bmatrix}}_{P} \odot \underbrace{\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}}_{K} = \begin{bmatrix} -0.1 & 0 & 0.9 \\ -0.2 & 0 & 0.9 \\ 0 & 0 & 0.9 \end{bmatrix}.$$

Sum the nine products:

$$-0.1 + 0 + 0.9 - 0.2 + 0 + 0.9 + 0 + 0 + 0.9 = 2.4.$$

With $b = -0.5$:

$$z = 2.4 - 0.5 = 1.9, \qquad \mathrm{ReLU}(z) = 1.9.$$

◂ Return to the filter

Appendix: convolution output sizes and parameter counts

For a filter of width k, padding p, and stride s, the output along one spatial axis is

$$\text{out} = \left\lfloor \frac{\text{in} + 2p - k}{s} \right\rfloor + 1.$$

same-size convolution

$$\frac{64 + 2(1) - 3}{1} + 1 = 64$$

halving pool

$$\frac{64 + 2(0) - 2}{2} + 1 = 32$$

A layer with C_{in} input channels and C_{out} filters learns

$$(k_h k_w C_{\text{in}} + 1) C_{\text{out}}$$

parameters: one channel-spanning filter and one bias per output map.

Return to the pipeline

Appendix: complete CNN shapes and parameters

step	operation	output shape	learned parameters
input	resized photograph	$64 \times 64 \times 3$	0
1	Conv 3×3, 16, $p = 1, s = 1$, ReLU	$64 \times 64 \times 16$	$(27 + 1)16 = 448$
2	Max pool $2 \times 2, s = 2$	$32 \times 32 \times 16$	0
3	Conv 3×3, 32, $p = 1, s = 1$, ReLU	$32 \times 32 \times 32$	$(144 + 1)32 = 4{,}640$
4	Max pool $2 \times 2, s = 2$	$16 \times 16 \times 32$	0
5	Conv 3×3, 64, $p = 1, s = 1$, ReLU	$16 \times 16 \times 64$	$(288 + 1)64 = 18{,}496$
6	Max pool $2 \times 2, s = 2$	$8 \times 8 \times 64$	0
7	global average pooling	64	0
8	dense $64 \to 37$, softmax	37	$(64 + 1)37 = 2{,}405$
		total	25,989

Shapes count stored activations. Parameter counts include only learned weights and biases; ReLU, pooling, and global averaging add none.

Return to the pipeline

Sources: data, convolution, and transfer


O. Parkhi, A. Vedaldi, A. Zisserman, and C. Jawahar.
Cats and Dogs.
CVPR, 2012.


Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner.
Gradient-Based Learning Applied to Document Recognition.
Proceedings of the IEEE, 1998.


A. Krizhevsky, I. Sutskever, and G. Hinton.
ImageNet Classification with Deep Convolutional Neural Networks.
NeurIPS, 2012.


J. Yosinski et al.
How transferable are features in deep neural networks?
NeurIPS, 2014.