AI Research Labs

Section 9 of 11 · 4 lessons

Regularisation

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

9.1 L2 and L1 Penalties

9.2 Early Stopping

9.3 Dropout

9.4 Data Augmentation



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.

9

BUILDING AND TRAINING A NETWORK · STEP 9

Regularise

How do I close the gap between training and validation?

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

L2 and L1 Penalties

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

What regularisation is

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

Penalised objective

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

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

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

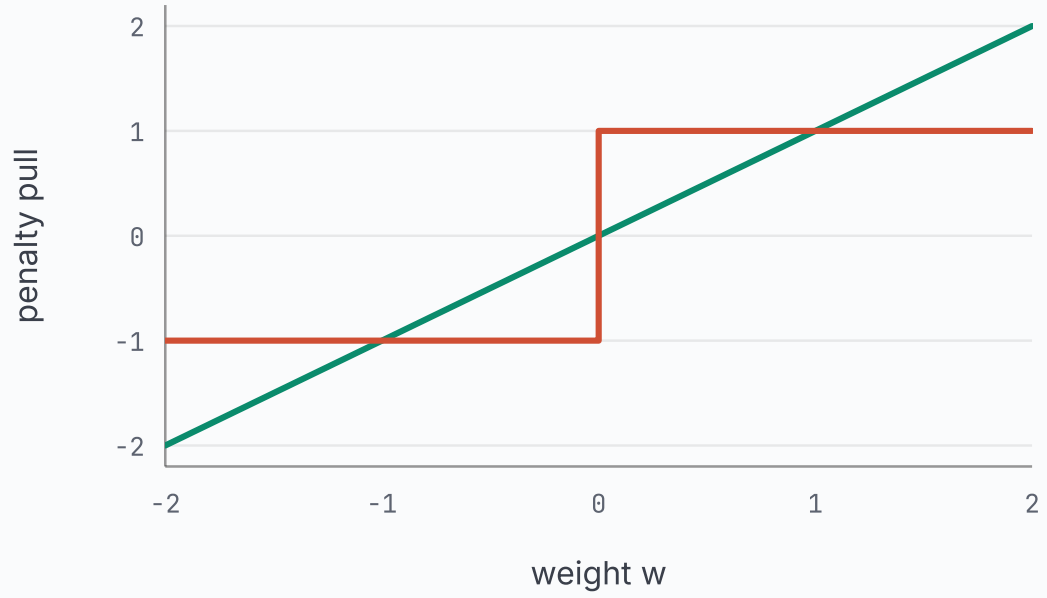
L2 is weight decay

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

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

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

L1 makes weights exactly zero



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

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

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

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

Common doubts

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

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

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

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

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

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

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

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

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

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

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

FIRST PASS

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

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

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

WORKED EXAMPLE

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

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

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

FIRST PASS

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

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

FIRST PASS

Why does L1 reach zero but L2 doesn't?

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

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

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

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

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

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

Thumb rule: "L1 → zeros. L2 → small."

Cheatsheet

L2 weight decay

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

L1 vs L2

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

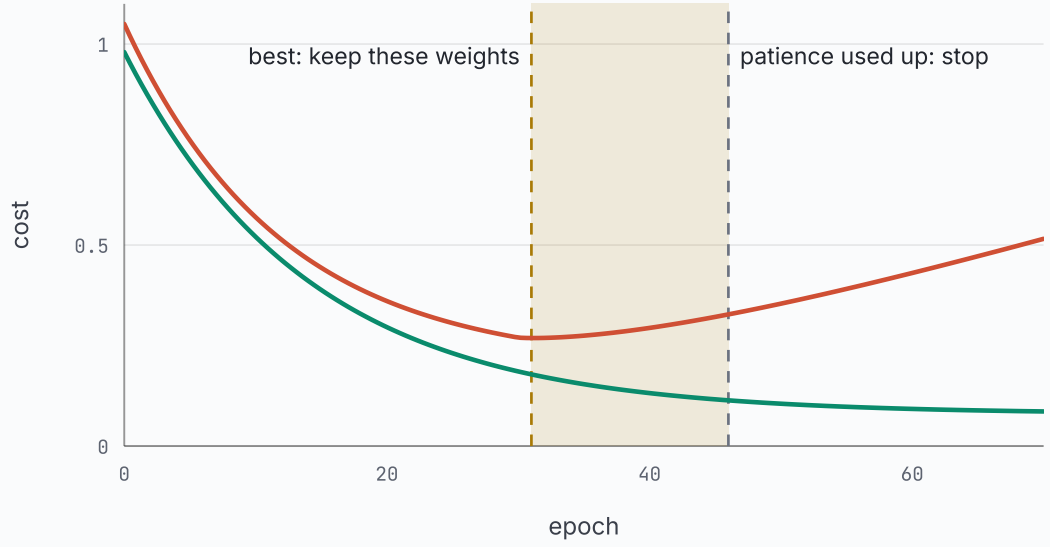
Online version with the interactive quiz: airesearchlabs.si/learn/dl/regularisation/l2-and-l1-penalties

Early Stopping

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

The method

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



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

Why it regularises

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

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

Common doubts

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

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

Cheatsheet

Early stopping

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

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

Dropout

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

Inverted dropout

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

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

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

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

WORKED EXAMPLE

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

Why it works

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

In practice

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

TWO CONVENTIONS: USE THE LECTURE'S

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

Common doubts

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

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

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

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

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

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

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

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

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

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

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

FIRST PASS

Where does dropout go in backpropagation?

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

WORKED EXAMPLE

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

Cheatsheet

Inverted dropout

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

Dropout in PyTorch

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

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

Data Augmentation

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

More data, made from the data you have

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

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

MAKE SURE THE LABEL SURVIVES

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

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

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

Common doubts

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

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

Cheatsheet

Augmentation

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

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