Lecture 7: Regularisation

Week 4 · principles

Eoin O’Brien

Regularisation and inductive bias

Solutions with zero training loss

- From last Friday:
  - an over-parameterised model can interpolate every training example
  - many parameter settings then reach the same, or nearly the same, training loss
  - those settings can behave very differently between the observed examples
- Training loss says which solutions fit the observed data
- It needn’t say which fitted solution behaves sensibly elsewhere

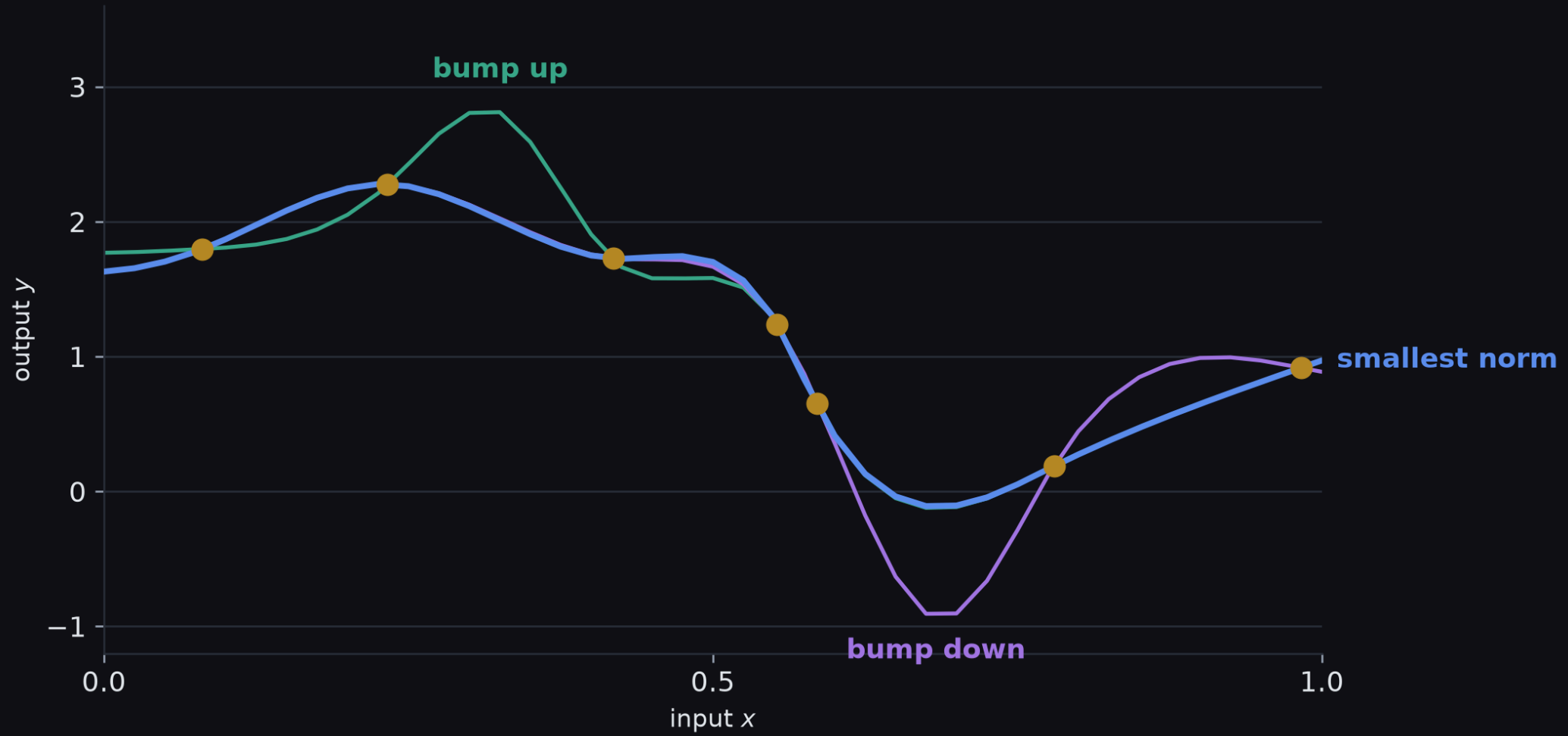
Regularisation as solution selection

- Several fitted models explain the training data equally well
- They behave differently between the observations
- Something other than training loss has to choose

Definition 1 **Regularisation** is any mechanism that biases learning toward a subset of the fitting solutions, to improve generalisation.

- **Typical preferences**: smoother functions, or less sensitivity to perturbation

Interpolants of the same points



Three functions that pass exactly through the same seven training points, so all three have zero training loss. Blue is the smallest-norm fit; the other two add a bump between two observations, where no training point can see it. Training loss cannot choose between them.

Strict and broad senses of regularisation

- **Strict sense**: an explicit term added to the loss
  - the objective itself changes
- **Broad sense**: any strategy intended to improve generalisation
  - the objective may be unchanged
- **This session**: explicit regularisation formally first, then the broad sense

Explicit regularisation

The data-fitting objective

$$\widehat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{arg\,min}} \sum_{i=1}^{I} \ell_i[\mathbf{x}_i, y_i].$$

- $(\mathbf{x}_i, y_i)$: training pair i
- ℓ_i: mismatch between prediction and target for that one example
- **The only question asked**: which parameters best fit the observed training data?

A scalar penalty over parameters

- **Penalty**: a scalar $g[\boldsymbol{\phi}]$, larger for less-preferred parameter choices
- **Strength**: $\lambda > 0$ sets how strongly it competes with data fit

$$\widehat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{arg\,min}} \left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, y_i] + \lambda g[\boldsymbol{\phi}] \right].$$

A scalar penalty makes preferences optimisable

- An optimiser needs **one scalar objective** to compare candidate parameter settings
- The data loss already gives one score: how badly the model fits the observations
- $g[\phi]$ gives a second score: how strongly a parameter setting violates the preference we want to impose
- λ sets the exchange rate between those two scores

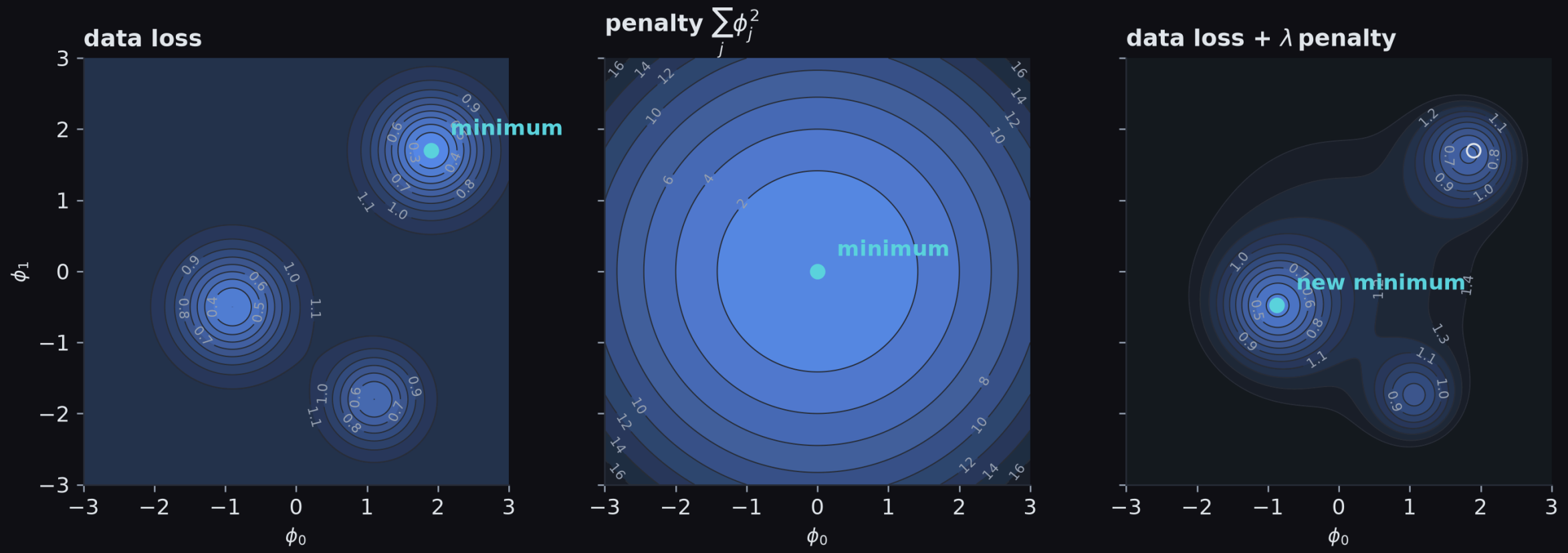
The penalty is therefore **not another measurement from the data**

It is a modelling choice about which fitting solutions we would rather select when data fit alone does not decide

Regularisation and the optimum

- *The regularised objective is a different surface from the data-fitting loss.*

Loss, penalty and their sum



Left: a data-fitting loss over two parameters with three basins; the lowest is marked. Centre: an L2 penalty, smallest at the origin. Right: their sum. The penalty raises the far basin more than the near one, so the global minimum of the sum moves to a different basin.

Parameter magnitude gives a simple generic preference

- The interpolating models from the opening all fit the observed points
- They can nevertheless require very different parameter values
- A simple generic preference is: **when fit is comparable, favour parameters closer to zero**
- In one dimension, distance from zero is just $|\phi|$
- With many parameters, we need one number that measures their **combined magnitude**

Small parameter magnitude is a **proxy**, not a theorem about simple functions

Neural networks can be reparameterised, so the same function can sometimes be represented with different weight scales

L2 is Euclidean length in parameter space

Remember how Pythagoras combines movement along perpendicular axes?

For a parameter vector

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_P \end{bmatrix}^\top,$$

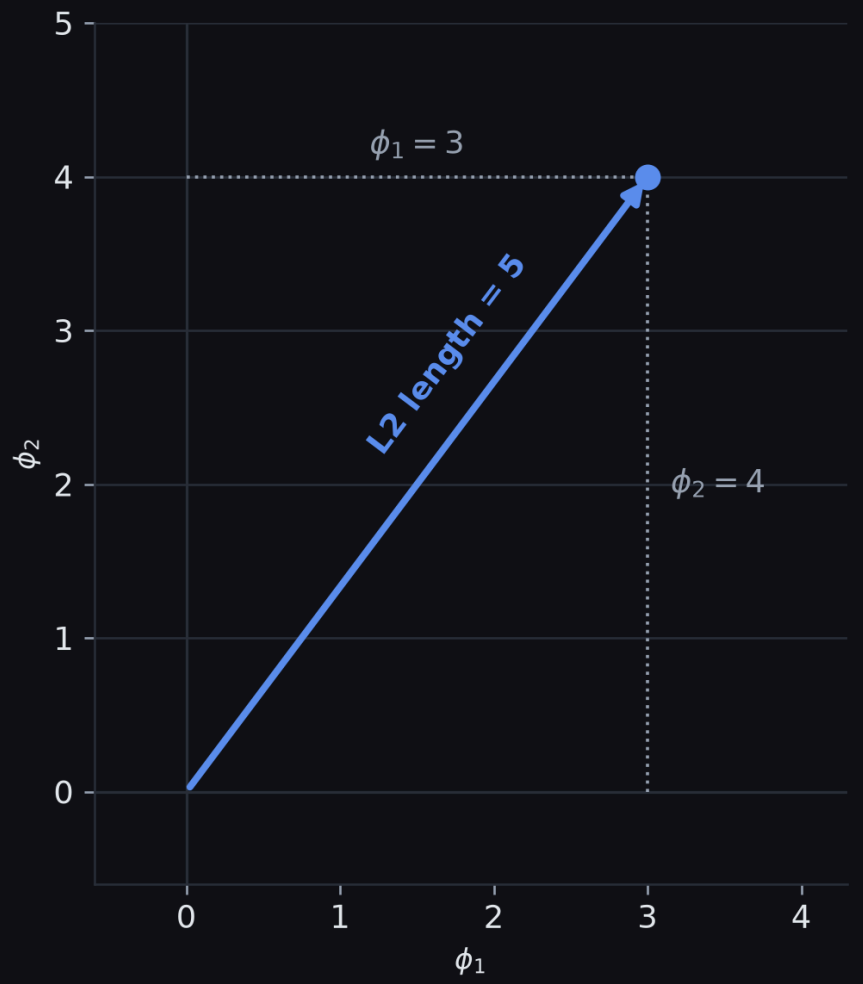
a **norm** is a rule that reduces a whole vector to one non-negative measure of its size

The **L2 norm** is the familiar Euclidean choice: distance from the origin

$$\|\boldsymbol{\phi}\|_2 = \sqrt{\phi_1^2 + \phi_2^2 + \cdots + \phi_P^2}.$$

- Example: $\boldsymbol{\phi} = [3, 4]^\top$ has $\|\boldsymbol{\phi}\|_2 = 5$
- Every coordinate contributes to the overall length
- Moving farther from zero in any direction increases the norm
- Only the distance matters: parameter vectors at the same radius receive the same L2 magnitude

L2 is distance from the origin



In two parameter dimensions, the L2 norm is ordinary Euclidean distance from the origin. The vector [3,4] has length 5 by Pythagoras. An L2 penalty therefore charges according to how far the whole parameter vector lies from zero, not according to any one coordinate alone.

Why prefer smaller weights?

- A neural-network layer forms **weighted sums** of its inputs or activations
- A large weight acts like a large gain: a change upstream can produce a larger change downstream
- Large gains across several layers can make the fitted function highly sensitive to small changes
- L2 gives a **soft preference** against relying on very large gains
  - it does not forbid them when the data strongly support them

Intuition: smaller weights can make the network less eager to produce sharp changes from small upstream differences

- **Caveat**: parameter magnitude is only a proxy for function behaviour
  - neural networks can sometimes represent the same function with different weight scales

Squared L2 is convenient for optimisation

Regularisation normally uses

$$\|\boldsymbol{\phi}\|_2^2 = \sum_{j=1}^{P} \phi_j^2$$

rather than the square root itself

- squaring preserves the ordering: the vector with smaller L2 norm also has smaller squared L2 norm
- the derivative is simple: $\nabla_{\boldsymbol{\phi}} \|\boldsymbol{\phi}\|_2^2 = 2\boldsymbol{\phi}$
- large coordinates become disproportionately expensive
- the penalty is smooth everywhere

So L2 regularisation says: **fit the data, but pay an increasing cost for moving far from the origin in parameter space**

Squared L2 in one parameter

Data term alone, minimised at $\phi = 3$:

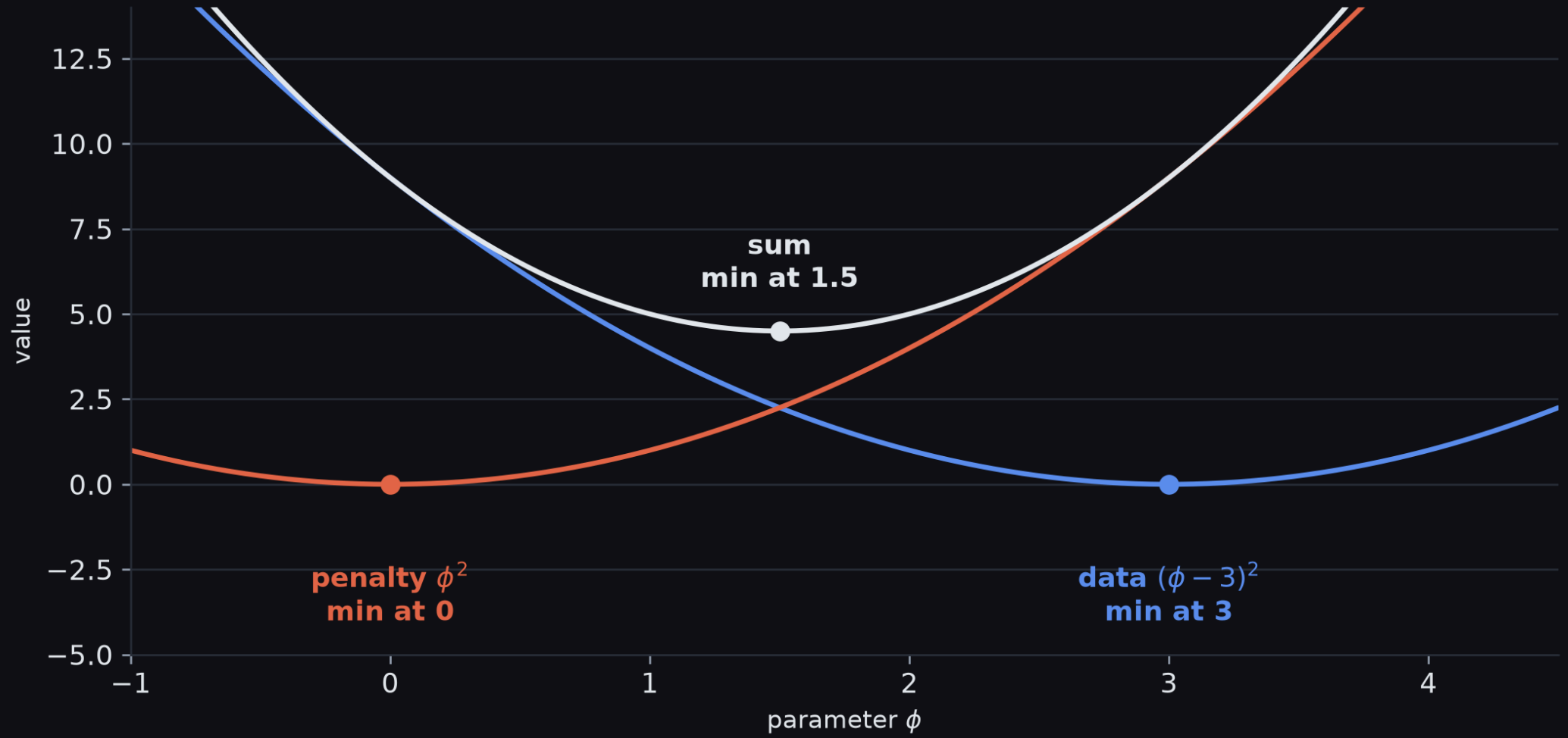
$$\mathcal{L}_{\text{data}}(\phi) = (\phi - 3)^2.$$

Add an L2 penalty with $\lambda = 1$:

$$\mathcal{L}_{\text{reg}}(\phi) = (\phi - 3)^2 + \phi^2.$$

- **Data term**: minimised at 3
- **Penalty**: minimised at 0
- **Sum**: a compromise between them

The one-parameter example



The data-fitting loss is minimised at φ = 3, the L2 penalty at φ = 0, and their sum at φ = 1.5: the regularised optimum gives up some data fit for a smaller parameter.

The regularised optimum

- Commit first: which ϕ minimises $(\phi - 3)^2 + \phi^2$?
- Differentiate, then set the derivative to zero

$$\frac{d\mathcal{L}_{\text{reg}}}{d\phi} = 2(\phi - 3) + 2\phi.$$

$$2(\phi - 3) + 2\phi = 0 \quad \Longrightarrow \quad \widehat{\phi} = 1.5.$$

- **Trade-off**: a worse data fit, accepted for a smaller parameter

The penalty’s gradient

Regularised loss:

$$\mathcal{L}_{\text{reg}}(\boldsymbol{\phi}) = \mathcal{L}(\boldsymbol{\phi}) + \lambda g[\boldsymbol{\phi}]$$

Its gradient, which the optimiser follows:

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}_{\text{reg}} = \nabla_{\boldsymbol{\phi}} \mathcal{L} + \lambda \nabla_{\boldsymbol{\phi}} g[\boldsymbol{\phi}].$$

- Every update combines two signals:
  - improve fit to the observed data
  - move toward parameters the regulariser favours

What squared L2 does during an update

For the penalty

$$\lambda \|\boldsymbol{\phi}\|_2^2,$$

the penalty gradient is

$$2\lambda\boldsymbol{\phi}.$$

- a positive weight contributes a positive penalty gradient
  - gradient descent subtracts it, so the weight moves downward toward zero
- a negative weight contributes a negative penalty gradient
  - subtracting it moves the weight upward toward zero
- a larger-magnitude weight feels a proportionally larger pull

Geometric reading: the L2 penalty adds a smooth bowl centred at the origin, and its gradient pulls the parameter vector inward

L2 regularisation

Definition 2 **L2 regularisation** adds λ times the sum of squared parameters to the data loss:

$$\widehat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\arg\min}\left[\sum_{i=1}^{I} \ell_i[\mathbf{x}_i, y_i] + \lambda \sum_j \phi_j^2\right].$$

- **Quadratic**: doubling $|\phi_j|$ quadruples that parameter's penalty

- Match the two penalty terms, with each ℓ_i the negative log-likelihood:

$$\lambda \sum_j \phi_j^2 = \frac{1}{2\sigma_\phi^2} \sum_j \phi_j^2 \quad \Longrightarrow \quad \lambda = \frac{1}{2\sigma_\phi^2}$$

- Smaller prior variance $\rightarrow$ larger λ $\rightarrow$ stronger pull toward zero

From vector L2 to a weight matrix

A weight matrix is still just a collection of scalar weights arranged in rows and columns

- for a vector, squared L2 adds the square of every coordinate
- for a matrix, do exactly the same thing to every entry
- equivalently: **flatten the matrix into one long vector, then take its L2 norm**

The resulting matrix norm is the **Frobenius norm**:

$$\|\boldsymbol{\Omega}_k\|_F = \|\mathrm{vec}(\boldsymbol{\Omega}_k)\|_2.$$

Frobenius is L2 after flattening

For $\boldsymbol{\Omega}_k \in \mathbb{R}^{D_{k+1} \times D_k}$,

$$\|\boldsymbol{\Omega}_k\|_F^2 = \sum_{r=1}^{D_{k+1}} \sum_{c=1}^{D_k} (\Omega_{k,rc})^2.$$

For example,

$$\boldsymbol{\Omega} = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} \quad \Longrightarrow \quad \|\boldsymbol{\Omega}\|_F = \sqrt{3^2 + 4^2} = 5.$$

- Nothing conceptually new happened: the notation changed because the weights are stored as a matrix
- **Weight-only network penalty**: $\lambda \sum_k \|\boldsymbol{\Omega}_k\|_F^2$

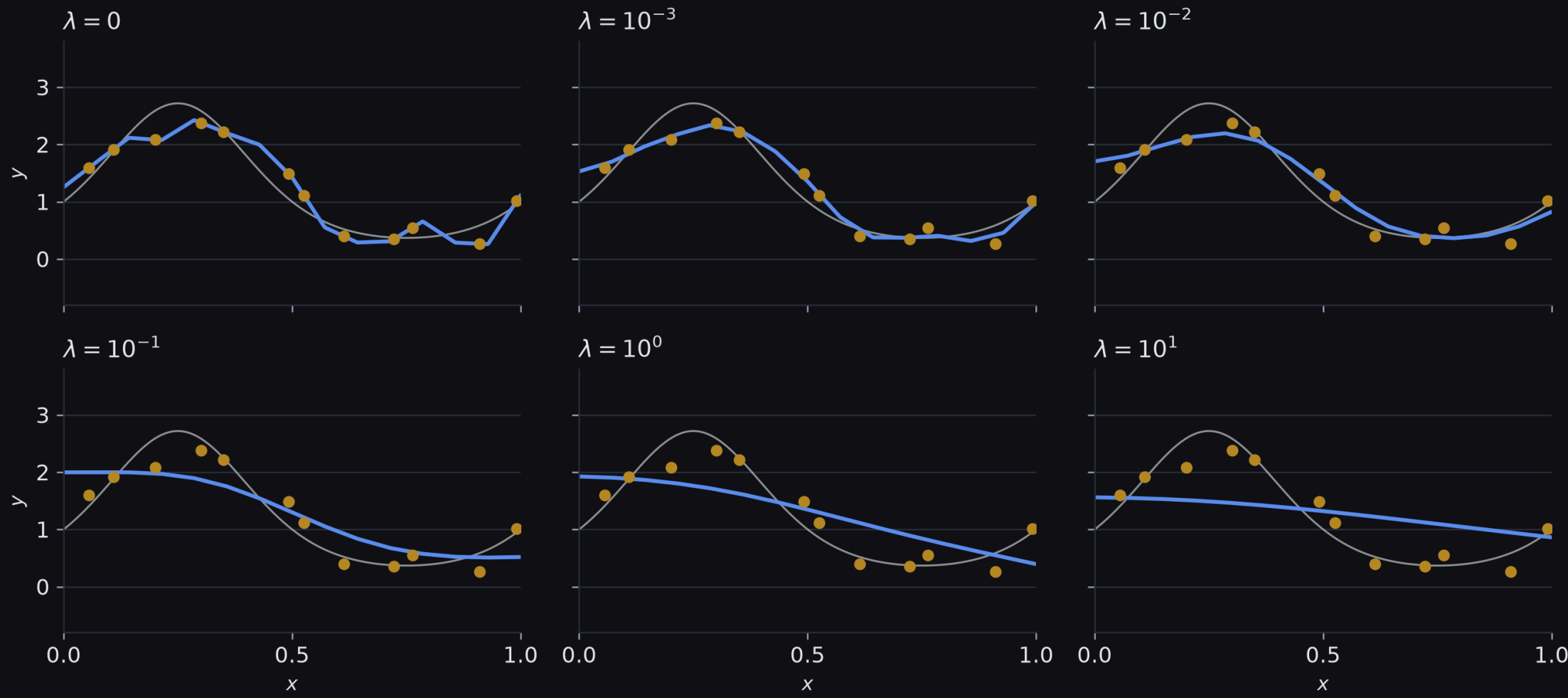
Biases and the penalty

- In neural networks, L2 is usually applied to the weights, not the biases
- So $\lambda \sum_k \|\Omega_k\|_F^2$ doesn't penalise every scalar in ϕ
- **Probabilistic reading**: no L2 prior penalty is imposed on the biases in this formulation
- **In code**: biases go in their own optimiser parameter group, with zero decay

Small weights in the toy model

- In the simplified piecewise-linear network:
  - each layer forms weighted sums
  - large weights allow larger changes in downstream pre-activations
  - penalising weight magnitude discourages those changes
- **Limit**: with every weight zero, the output becomes a constant: the final bias
- **Scope**: explains this toy model's smoothing
  - not a theorem that smaller weights make every network smoother

L2 strength and the fitted function



The same twelve training points and the same fourteen-joint model, fitted with increasing L2 strength λ. With no penalty the fit chases the noise; moderate λ removes the sample-specific bends; large λ removes the underlying shape as well. The thin grey curve is the true mean.

Regularisation strength, bias and variance

- As λ increases:
  - training fit generally worsens
  - sensitivity to sample-specific variation can fall
  - the fitted function becomes more constrained
  - too much constraint underfits
- ***One mechanism can reduce variance while increasing bias.***
- **Choosing λ**: a hyperparameter, set on validation evidence

Loss reduction and the coefficient

The L2 objective above used a **summed** data loss:

$$\mathcal{L}_{\text{sum}} = \sum_{i=1}^{I} \ell_i + \lambda_{\text{sum}}\, g[\boldsymbol{\phi}].$$

- PyTorch `CrossEntropyLoss` takes the mean by default:

$$\mathcal{L}_{\text{mean}} = \frac{1}{I}\sum_{i=1}^{I} \ell_i + \lambda_{\text{mean}}\, g[\boldsymbol{\phi}].$$

- Scaling an objective by a positive constant doesn't move its minimiser
- So the two share an optimum when:

$$\lambda_{\text{mean}} = \frac{\lambda_{\text{sum}}}{I}.$$

- **Consequence**: a coefficient's value only makes sense with its loss reduction

L2 regularisation vs weight decay

- **L2 regularisation**: a penalty inside the objective

$$\mathcal{L}_{\text{reg}}(\boldsymbol{\phi}) = \mathcal{L}(\boldsymbol{\phi}) + \lambda \|\boldsymbol{\phi}\|^2$$

Definition 3 **Weight decay** shrinks the parameters by a factor $1 - \lambda'$ in the update itself:

$$\boldsymbol{\phi} \leftarrow (1 - \lambda')\boldsymbol{\phi} - \alpha \nabla_{\boldsymbol{\phi}} \mathcal{L}.$$

- Commit first: are these the same definition?
- **Answer**: no; one changes the objective, the other the update rule
  - they coincide under plain SGD

L2 and weight decay under SGD

Gradient of the squared penalty:

$$\nabla_{\phi}\left(\lambda\|\phi\|^{2}\right)=2\lambda\phi.$$

- An SGD step becomes:

$$\begin{aligned}\phi_{t+1} &= \phi_t-\alpha\left(\nabla_{\phi}\mathcal{L}+2\lambda\phi_t\right)\\ &= (1-2\alpha\lambda)\phi_t-\alpha\nabla_{\phi}\mathcal{L}.\end{aligned}$$

The coefficient mapping

- **L2 under SGD**: $(1 - 2\alpha\lambda)\boldsymbol{\phi}_t - \alpha\nabla_{\boldsymbol{\phi}}\mathcal{L}$
- **Weight decay**: $(1 - \lambda')\boldsymbol{\phi}_t - \alpha\nabla_{\boldsymbol{\phi}}\mathcal{L}$
- They match when:

$$\lambda' = 2\alpha\lambda.$$

- **Scope**: the equivalence is tied to the optimiser's update rule

PyTorch's SGD coefficient

With the convention $\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \|\boldsymbol{\phi}\|^2$:

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}_{\text{reg}} = \nabla_{\boldsymbol{\phi}} \mathcal{L} + 2\lambda \boldsymbol{\phi}.$$

- PyTorch SGD's `weight_decay=wd` adds $wd\,\boldsymbol{\phi}$ to the gradient
- For the same data-loss reduction, the coefficients correspond as:

$$wd = 2\lambda.$$

- **Plain SGD, no momentum**: the direct shrinkage coefficient at step t is

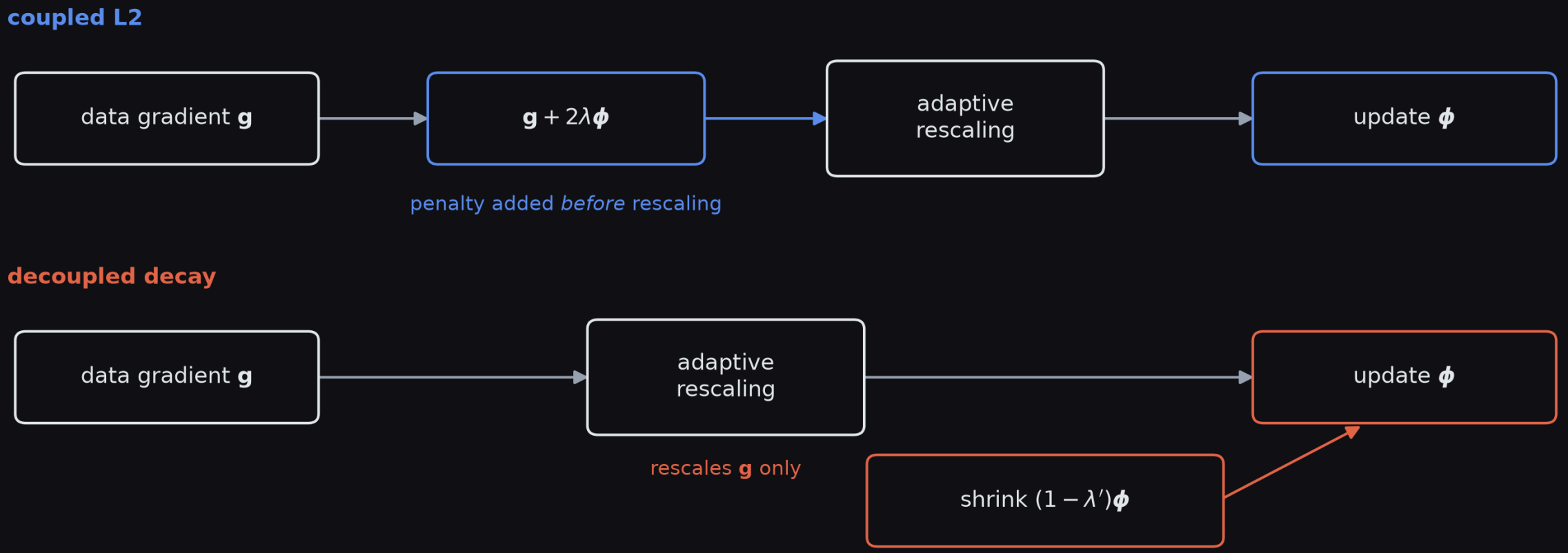
$$\lambda'_t = \alpha_t \, wd$$

- **Momentum or adaptive optimisers**: derive the effect from the actual update rule
- **Sum to mean**: rescale the L2 coefficient first

Adaptive scaling

- Adaptive optimisers rescale each gradient coordinate using its history
- **Coupled L2**: $2\lambda\phi$ joins the gradient before the rescaling
  - so it's transformed along with the data gradient
- **Direct shrinkage**: skips that transformation
- The motivation for **decoupled weight decay** in AdamW

Coupled L2 and decoupled decay



Two update rules for an adaptive optimiser. Top, coupled L2: the penalty's gradient 2λφ joins the data gradient g before the adaptive rescaling, so each coordinate's penalty is rescaled too. Bottom, decoupled weight decay (AdamW): only g is rescaled, and the shrinkage is applied to φ on a separate branch. Under plain SGD there is no rescaling and the two coincide.

Coupled and decoupled decay in PyTorch

```
# SGD adds weight_decay * parameter to the gradient
sgd = torch.optim.SGD(
    weight_params,
    lr=lr,
    weight_decay=wd,
)

# Decoupled weight decay for Adam-style optimisation
adamw = torch.optim.AdamW(
    weight_params,
    lr=lr,
    weight_decay=wd,
)
```

- Give biases a separate parameter group with zero decay

Different norms encode different preferences

The penalty defines what counts as an expensive parameter vector

- **L2**: $\sum_j \phi_j^2$
  - increasingly expensive large coordinates
  - smooth shrinkage toward zero
- **L1**: $\sum_j |\phi_j|$
  - cost grows linearly with magnitude
  - favours sparse solutions; suitable optimisation can set some coordinates exactly to zero
- **L0-style**: number of non-zero parameters
  - ignores magnitude and asks only how many parameters are active
- **Elastic net**: combines L1 and L2

Choosing a norm means choosing a **geometry of preference**: different penalties favour different kinds of solutions

Sparsity

- A hidden unit whose incoming weights are all zero contributes nothing
  - it can potentially be removed
- Links regularisation to:
  - model compression
  - pruning
  - faster inference
- **Cost**: the L0 count is discontinuous at zero
  - gradient-based optimisation can’t differentiate it the way it does L2
  - practical methods use relaxations or specialised procedures

Limits of explicit regularisation

- Evidence on how much ordinary L2 itself explains generalisation is mixed
- Individual weight magnitude is only an indirect proxy for function smoothness
- Other norms and constraints can target more specific properties
- **Limit**: a controlled modelling bias, not a universal explanation of why deep networks generalise

Implicit regularisation

Optimisation as solution selection

- **Setting**: several parameter vectors reach the same training minimum
- The objective has no explicit $g[\phi]$ term
- The training algorithm can still reach some of those minima more readily than others

Definition 4 Implicit regularisation is a systematic preference among equally good minima that comes from the training algorithm rather than from any term in the loss.

- **Consequence**: the optimiser is part of the learning system's inductive bias

Gradient flow isolates finite-step effects

We want to isolate an effect that is easy to miss:

the optimiser can bias the solution even when we never write a regularisation term

- continuous gradient flow is the idealised limit of infinitesimally small steps
- it gives us a reference path determined by the local gradient at every instant
- ordinary gradient descent takes finite jumps instead
- comparing the two isolates the bias introduced by **discretising** the optimisation

Gradient flow is not a better training algorithm here

It is a mathematical baseline for asking what finite step size changes

Continuous gradient flow

- Imagine gradient-descent steps that are infinitesimally small
- The parameters then move continuously in time t

$$\frac{d\boldsymbol{\phi}}{dt} = -\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\phi}).$$

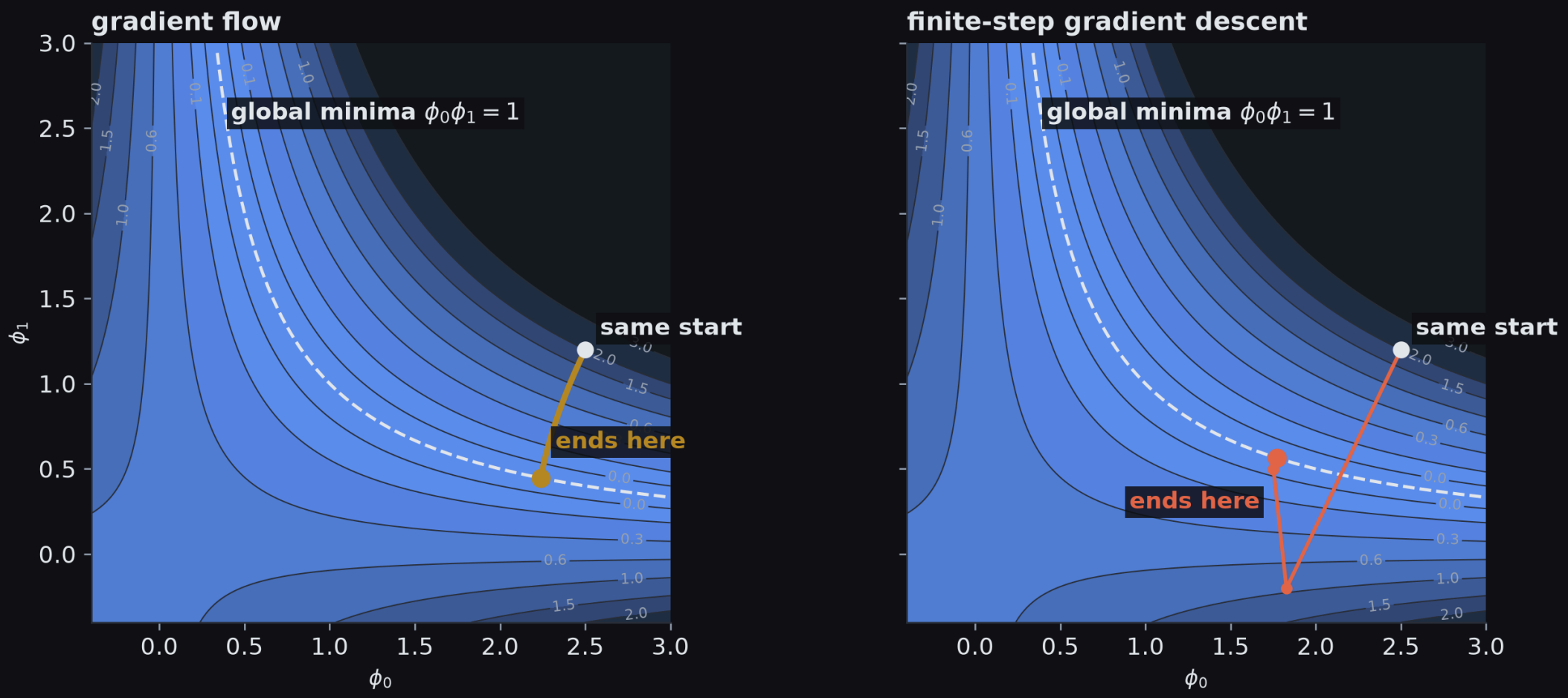
- **Read it as**: the velocity through parameter space points straight down the local gradient

Finite-step gradient descent

$$\phi_{t+1} = \phi_t - \alpha \nabla_\phi \mathcal{L}(\phi_t).$$

- **Notation**: from here t counts steps, not continuous time
  - α plays the role of the time elapsed per step
- The gradient is measured at ϕ_t, then held fixed for the whole step
- So the discrete path need not follow the continuous one

Gradient flow vs gradient descent



The same loss surface and the same starting point, shown separately so the trajectories are easy to compare. Left: continuous gradient flow follows a smooth curved path to one point on the valley of global minima. Right: finite-step gradient descent follows straight chords based on the gradient at the start of each step and reaches a different point on the same zero-loss valley, where the valley is wider.

The modified loss for gradient descent

- **Question**: how far does discrete descent deviate from the continuous flow?
- **Reframe**: which slightly modified loss would make the continuous flow track the discrete path?

$$\widetilde{\mathcal{L}}_{\mathrm{GD}}(\boldsymbol{\phi}) = \mathcal{L}(\boldsymbol{\phi}) + \frac{\alpha}{4}\|\nabla_{\boldsymbol{\phi}}\mathcal{L}(\boldsymbol{\phi})\|^2.$$

- **Status**: stated here, not derived
  - it comes from backward-error analysis, keeping the leading order in α

The gradient-norm term

$$\frac{\alpha}{4}\|\nabla_\phi \mathcal{L}\|^2$$

- **Remember how the gradient collects one partial derivative for each parameter?**
- It points in the direction of steepest local increase
- The L2 norm combines those many partial derivatives into **one overall steepness measure**:

$$\|\nabla_\phi \mathcal{L}\|_2 = \sqrt{\sum_j \left(\frac{\partial \mathcal{L}}{\partial \phi_j}\right)^2}$$

- Its square is large where the surface is steep in parameter space
- The factor $\alpha/4$ makes the effect stronger for larger steps
- So in the modified-loss view, discrete descent is repelled from regions of large gradient magnitude

Minima under the modified loss

- At an exact differentiable minimum, $\nabla_\phi \mathcal{L} = \mathbf{0}$
- So the added gradient-norm term is zero there too
- Every original differentiable minimum therefore remains **unpenalised by this added term**
- **The route**: $\|\nabla_\phi \mathcal{L}\|^2$ is large on steep walls
  - the modified flow is pushed towards gently curved regions before it arrives
  - this is why finite steps can favour wider minima
- ***What changes is the route, and so which member of a family of minima is reached.***

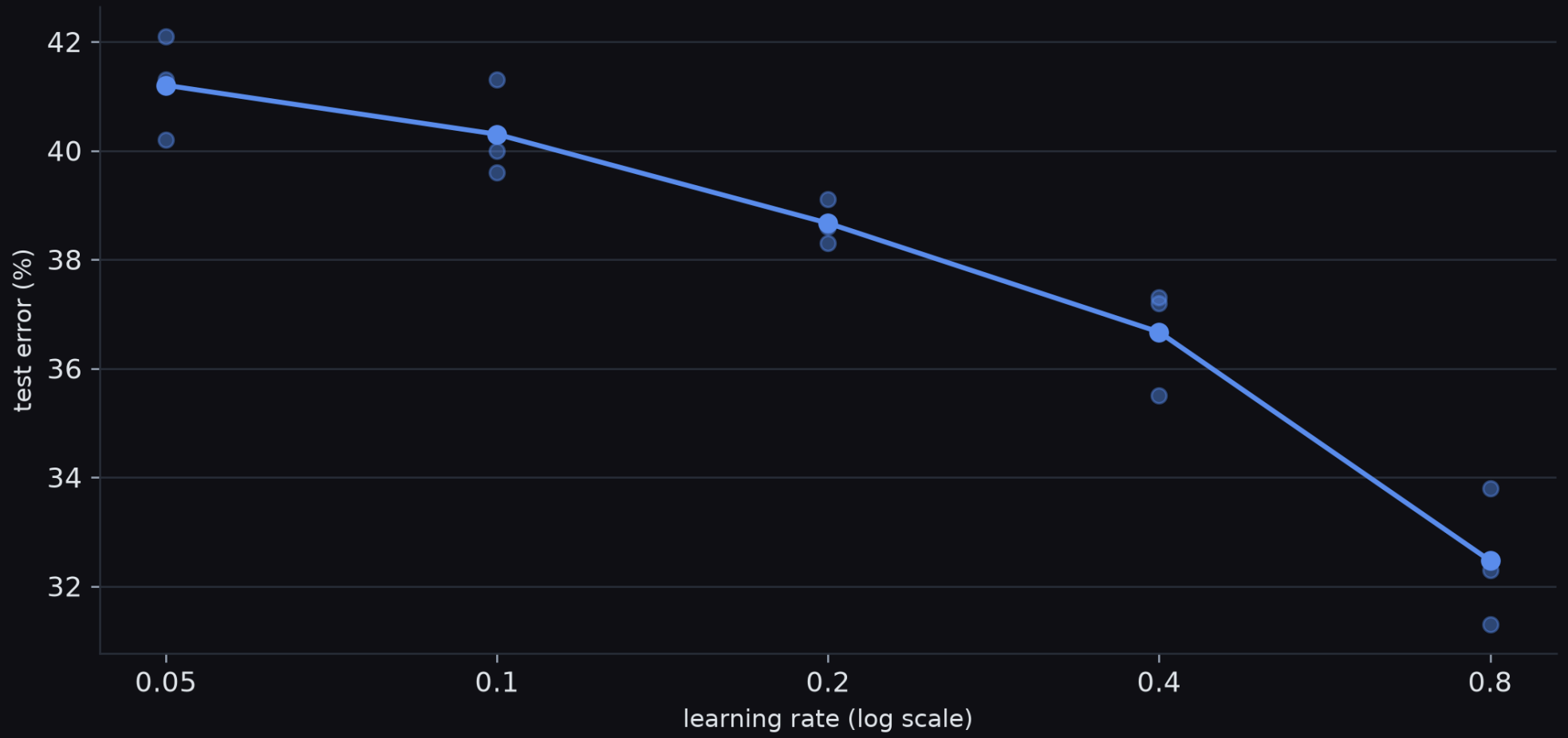
Limits of the modified-loss approximation

- The modified-loss approximation comes from a truncated backward-error expansion
  - only the leading-order term in α is kept
- **Read it as**: an approximate description of finite-step dynamics
- **Not**: an exact identity for arbitrary learning rates and arbitrary training trajectories

Learning rate and implicit regularisation

- In this analysis a larger α does two things:
  - makes each step longer
  - strengthens the implicit gradient-norm penalty

Learning rate and test error



Our rerun of the chapter's experiment: MNIST-1D, a network with two hidden layers of 100 units, SGD with batches of 100. Each run takes 6000/LR steps, so every run has the same opportunity to travel; every run fits the training set exactly. Dots are single runs from three seeds, the line their mean. Larger learning rates reach lower test error.

Reading the learning-rate result

- **Our rerun**: the chapter's MNIST-1D experiment, with every run given matched travel
- Larger learning rates give lower test error
  - 41% at the smallest rate, 32% at the largest
- The chapter reports the same trend
- **Confound**: these runs are minibatch SGD
  - so this experiment cannot isolate the full-batch finite-step mechanism from minibatch-SGD effects

- **What it doesn’t show**:
  - that the largest stable learning rate is always best
  - that learning rate can be raised without regard to optimisation
  - that the gradient-norm approximation fully explains the observation
- ***Learning rate can change which solution training reaches, as well as how fast parameters move.***

Minibatch gradients

- **Convention**: from here $\mathcal{L}$ is the **mean** over examples
  - earlier in this lecture the data-fitting objective used a sum
- Batch b has its own mean loss $\mathcal{L}_b$ over its examples $\mathcal{B}_b$

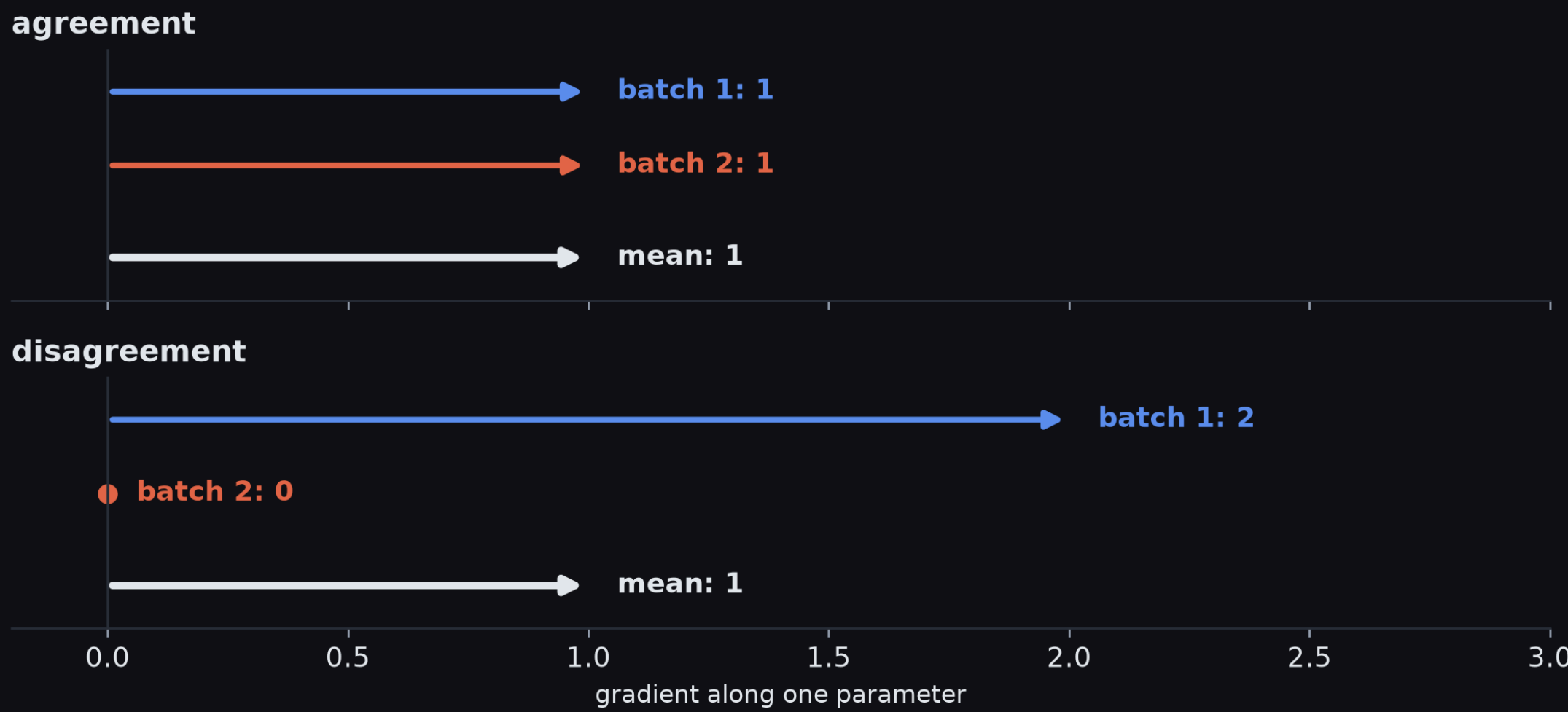
$$\mathcal{L} = \frac{1}{I}\sum_{i=1}^{I} \ell_i, \qquad \mathcal{L}_b = \frac{1}{|\mathcal{B}_b|}\sum_{i \in \mathcal{B}_b} \ell_i.$$

- The full gradient is an aggregate
- Individual batch gradients can disagree around it

Two batch-gradient cases

- Two scalar batch gradients, g_1 and g_2
- **Case A**: $g_1 = 1, g_2 = 1$
- **Case B**: $g_1 = 2, g_2 = 0$
- Both have mean gradient 1

Batch agreement



Two pairs of minibatch gradients along one parameter. Top: both batches say 1, so they agree exactly. Bottom: one says 2 and the other 0. Both pairs average to the same full gradient, 1 (white), so the mean alone cannot tell the two cases apart.

Gradient variance measures batch disagreement

The full-data gradient keeps only the **average direction**

That average can hide whether minibatches:

- all agree closely
- disagree strongly and merely cancel in the mean

Variance answers the missing question:

how far do the individual batch gradients spread around their mean?

A small variance means the batches tell a similar local story; a large variance means different subsets of the data are pulling the parameters differently

Batch-gradient variance

- **Batch-gradient variance**: the mean squared distance of the batch gradients from their mean
- Commit first: which case has the larger variance? Compute both.

$$\text{A: } \frac{(1-1)^2+(1-1)^2}{2} = 0, \qquad \text{B: } \frac{(2-1)^2+(0-1)^2}{2} = 1.$$

- The mean can’t separate the two cases; the variance can

Averaging minibatch gradients

- **Assumption**: the dataset is split into B equal-sized minibatches
- Then the full gradient is the mean of the batch gradients:

$$\nabla \mathcal{L} = \frac{1}{B} \sum_{b=1}^{B} \nabla \mathcal{L}_b.$$

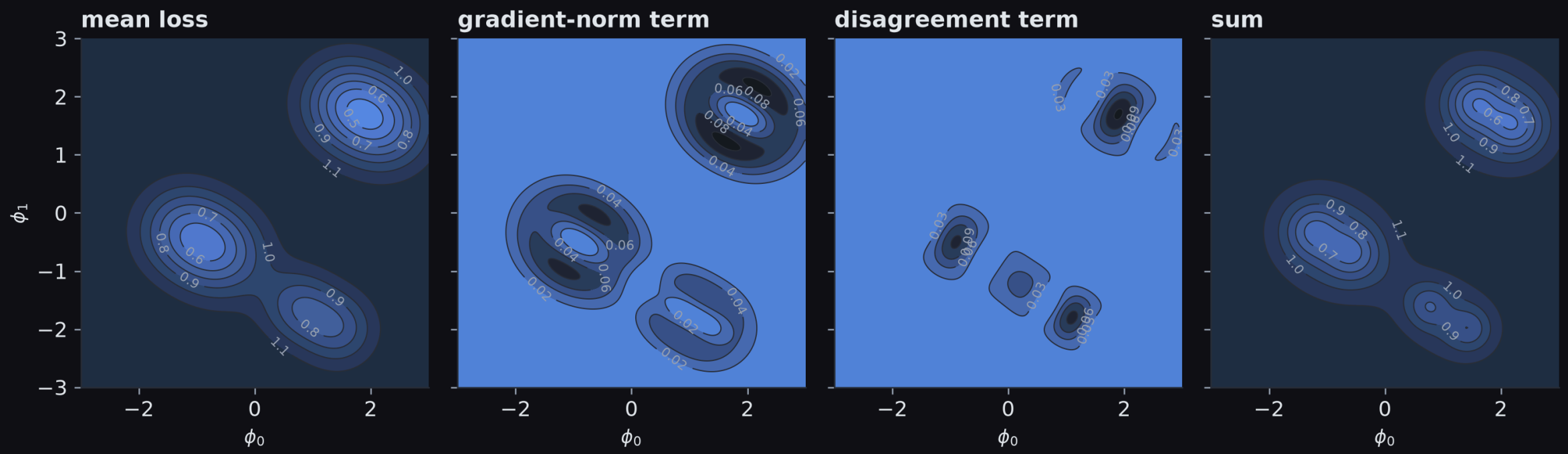
- With unequal sizes, weight each batch gradient by its fraction of the dataset

Three terms of the modified SGD objective

$$\underbrace{\mathcal{L}}_{\text{fit the data}} + \underbrace{\frac{\alpha}{4}\|\nabla\mathcal{L}\|^2}_{\text{avoid steep regions}} + \underbrace{\frac{\alpha}{4B}\sum_b \|\nabla\mathcal{L}_b - \nabla\mathcal{L}\|^2}_{\text{prefer batch agreement}}$$

- **Interpretation**: this is the leading-order modified objective for the average SGD trajectory
- The last term is absent from full-batch gradient descent
- It is $\alpha/4$ times the batch-gradient variance
  - Case A: $\alpha/4$ times 0
  - Case B: $\alpha/4$ times 1
- ***Unlike the gradient-norm term, the disagreement term need not vanish at a minimum of $\mathcal{L}$, so it can move the minimum itself.***

The modified SGD surface

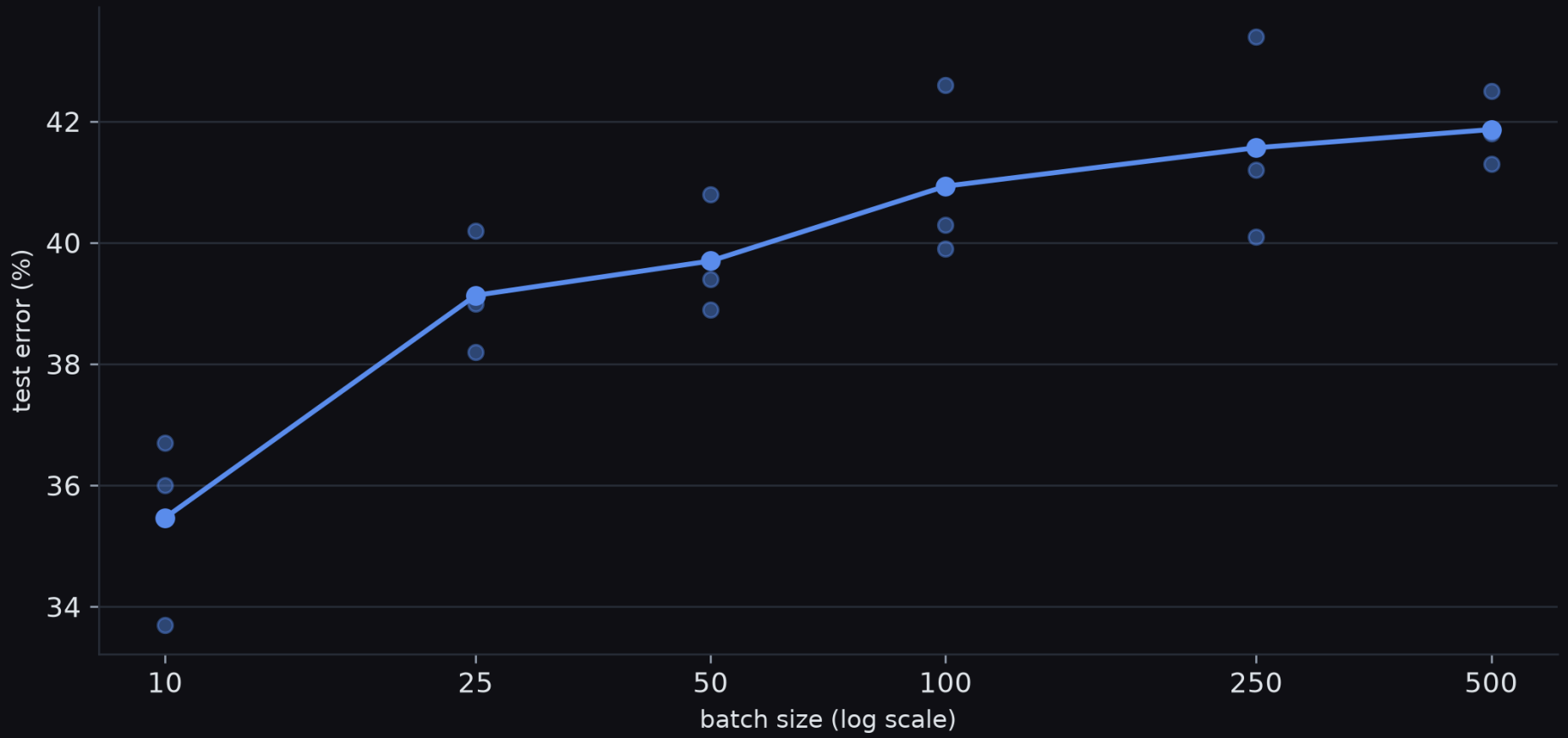


The pieces of the modified SGD objective, over the same two parameters, for two minibatches whose losses are the same three basins shifted slightly apart. The mean loss; the gradient-norm term $(\alpha/4)\|\nabla L\|^2$; the disagreement term, largest on the basin floors where the curvature is highest and the two shifted batches pull furthest apart; and their sum. Unlike the gradient-norm term, the disagreement term is not zero at a minimum, so the minimum of the sum moves.

Batch size and gradient disagreement

- Smaller batches average over fewer examples
  - so individual batch gradients can vary more
- **Our rerun**: MNIST-1D again, each run stopped at the epoch it first fits the training set
- Smaller batches reach lower test error
  - 35% at batch size 10, 42% at 500
- The chapter reports the same trend

Batch size and test error



Our rerun of the chapter's experiment: the same network and data, SGD at learning rate 0.05, with each run stopped at the first epoch where it fits the training set exactly, so every run is compared at the point of memorisation. Dots are single runs from three seeds, the line their mean. The smallest batch reaches the lowest test error; from 25 upwards, the steps between means are smaller than the spread between seeds.

Reading the batch-size result

- Implicit regularisation is one possible explanation
- Batch size also changes:
  - gradient noise
  - number of updates per epoch
  - computational efficiency
  - optimisation stability
- **Status**: “smaller batches generalise better” is an empirical tendency under some conditions, not a law

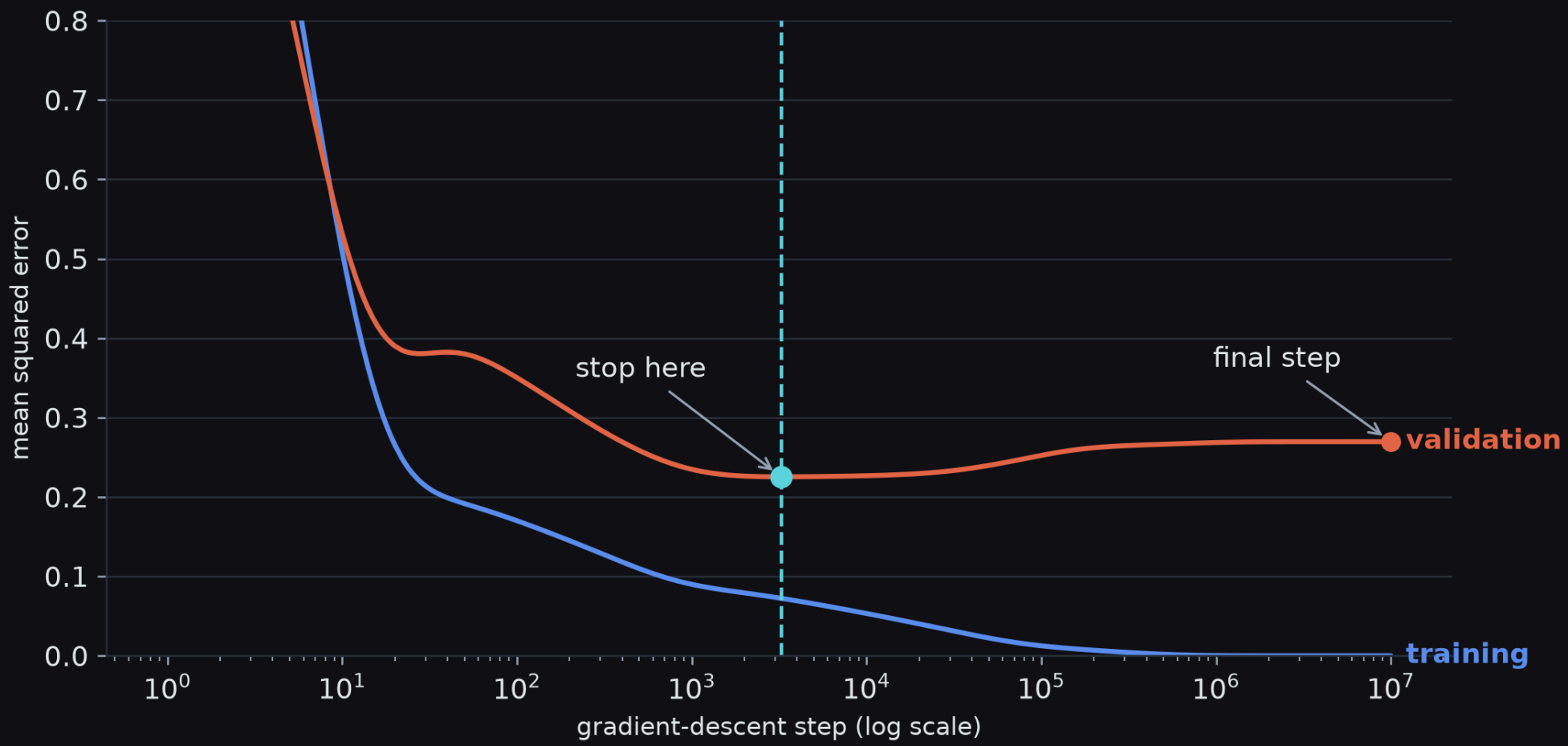
Early stopping

Stopping on validation loss

- Training loss often keeps falling after validation performance has stopped improving
- A flexible model can learn broad structure first, then sample-specific detail

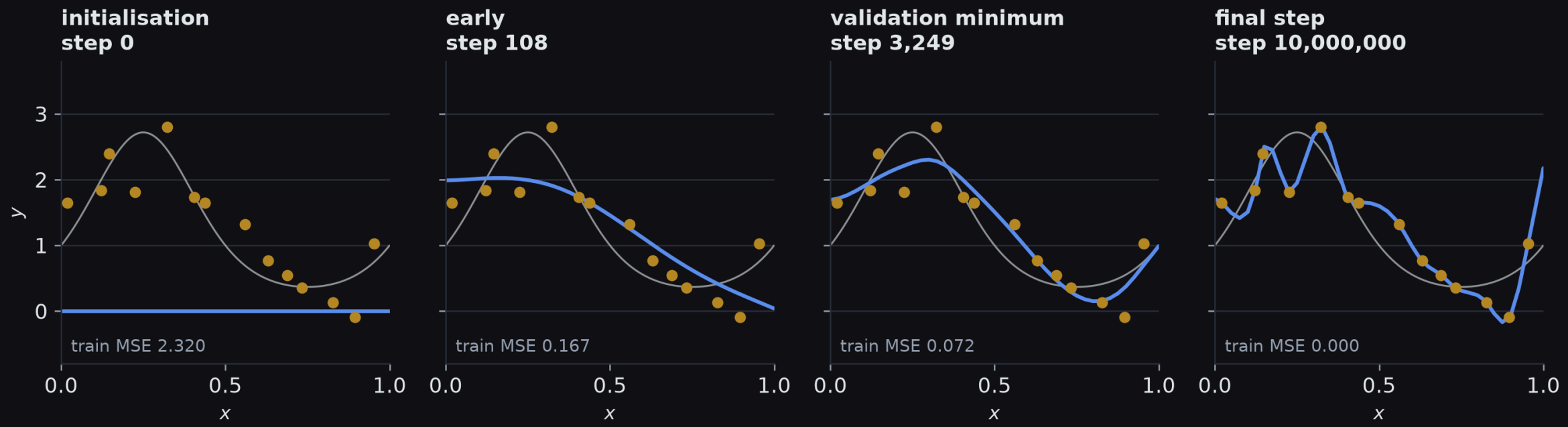
Definition 5 **Early stopping** ends optimisation before full convergence and keeps the parameters from the checkpoint with the best validation performance.

Training and validation loss



Full-batch gradient descent on the fixed-joint model from last Friday, forty joints and fourteen training points, started at zero. Training loss keeps falling towards zero; validation loss, on fresh points from the same process, reaches its minimum and then rises. Early stopping keeps the checkpoint at that minimum.

The fit during training



The same run at four moments: at initialisation, early, at the validation minimum, and at the final step. The coarse shape of the true mean (thin grey) arrives first; the late steps bend the fit through individual noisy points while training loss keeps falling.

The early-stopping hyperparameter

- **Hyperparameter**: the effective stopping time
- Chosen from a single training run:
  - evaluate validation performance periodically
  - store checkpoints
  - keep the checkpoint with the best validation performance
- The test set stays untouched until model selection is complete

Early stopping and L2

- **Two intuitions**:
  - parameters start small and have less time to grow
  - stopping early limits the effective complexity optimisation explores
- **Under a quadratic approximation with zero initialisation**: early stopping is approximately equivalent to L2 regularisation in gradient descent
  - assuming a quadratic approximation to the loss
  - and parameters initialised to zero
  - effective weight $\lambda \approx 1/(\tau\alpha)$, with τ the stopping time
- Outside those assumptions, treat them as related biases rather than identical algorithms

Validation frequency

- Suppose validation is checked every T optimisation steps
- The selected stopping point is then quantised to those checkpoints
- **Large T**: can miss the best region
- **Small T**: adds evaluation cost
- Select checkpoints on the metric that actually matters

Combining predictions

Averaging three predictions

- **Setup**: three regression models, one target whose underlying value is near 5
- **Predictions**:

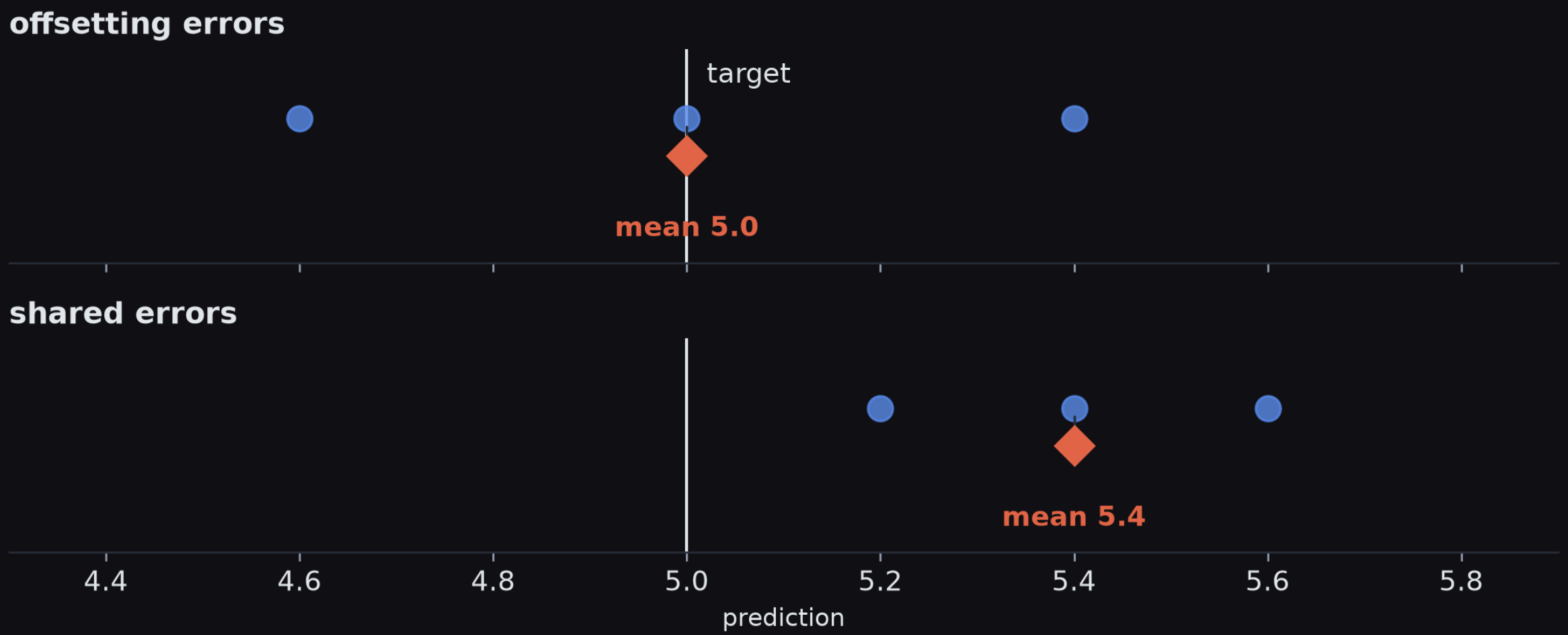
$$4.6, \qquad 5.4, \qquad 5.0$$

- **Mean prediction**:

$$\frac{4.6 + 5.4 + 5.0}{3} = 5.0.$$

- averaging can cancel errors when the models don’t fail in the same way

Offsetting and shared errors



Three predictions of a target of 5. Top: the errors fall on both sides and the mean lands on the target. Bottom: every error has the same sign, and the mean inherits it.

Ensembles

Definition 6 An **ensemble** is a group of models whose predictions are combined into one prediction.

- **Regression**: mean or median of the outputs
- **Classification**: mean of the pre-softmax activations, or the most frequent predicted class
- ***The gain comes from diversity among the models' errors.***
  - storing the same model several times gains nothing

Correlated errors

- **Shared error**: every model errs the same way, and averaging preserves it
- **Sources of diversity**:
  - different random initialisations
  - different sampled datasets
  - different hyperparameters
  - different model families
- **Idealised case**: independent errors, where the cancellation argument is strongest

Diversity from initialisation

- networks started from different random parameters can converge to different fitted functions
- **Where it matters**: regions the training data barely constrain
  - e.g. far from the training examples
- averaging several independently trained networks moderates their arbitrary behaviour there

Bagging

Definition 7 Bootstrap aggregating, or **bagging**, trains each model of an ensemble on its own resample of the training data, drawn with replacement, and combines their predictions.

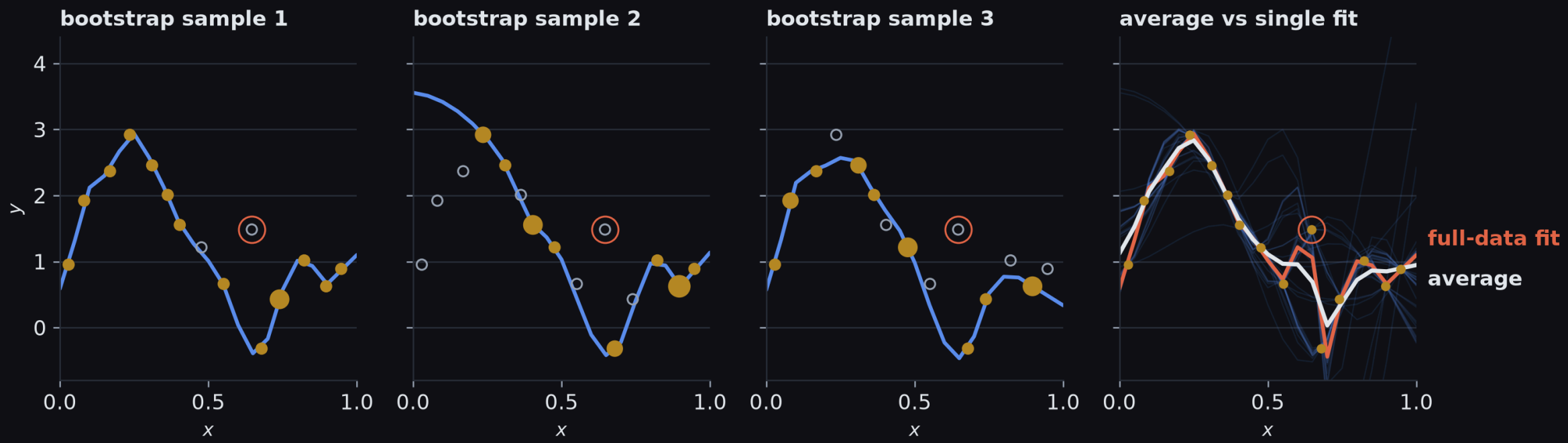
1. sample a new training set with replacement from the original data
2. train a separate model on each resample
3. combine their predictions

- **With replacement**:
  - some examples appear several times
  - some examples are absent from a particular resample

Bagging and outliers

- **Omitted point**: a model whose resample lacks an unusual point infers that region from the remaining observations
- **Averaged**: the ensemble is less sensitive to that one observation

Bagging on the toy regression



Bagging on a toy regression with one unusual point (circled). The first three panels are bootstrap samples: marker size shows how many times each point was drawn, and hollow markers were not drawn at all. The last panel overlays all forty bootstrap fits, their average, and the single fit to the full data; at the unusual point the average is pulled less far from the true mean.

The cost of ensembles

- multiple training runs
- multiple stored parameter sets
- multiple inference passes

Dropout, perturbations, and soft targets

Changing subnetworks break fragile dependencies

A network can rely on fragile partnerships between units:

- one unit creates a feature
- another cancels or corrects it
- the combination fits the observed data
- none of the units needs to work robustly on its own

A **fixed** smaller network would reduce capacity, but it would not create dropout's changing-subnetwork pressure

A **changing random** subnetwork repeatedly breaks those partnerships, forcing useful behaviour to survive under many combinations of active units

Dropout

Definition 8 Dropout clamps a random subset of hidden units to zero at each training iteration, drawing a fresh subset every time.

- **Tempting reading**: dropping half the units just trains a smaller network
- **What it misses**: the dropped subset changes every iteration
  - the active subnetwork keeps changing
  - all the subnetworks share the same underlying parameters

The dropout mask

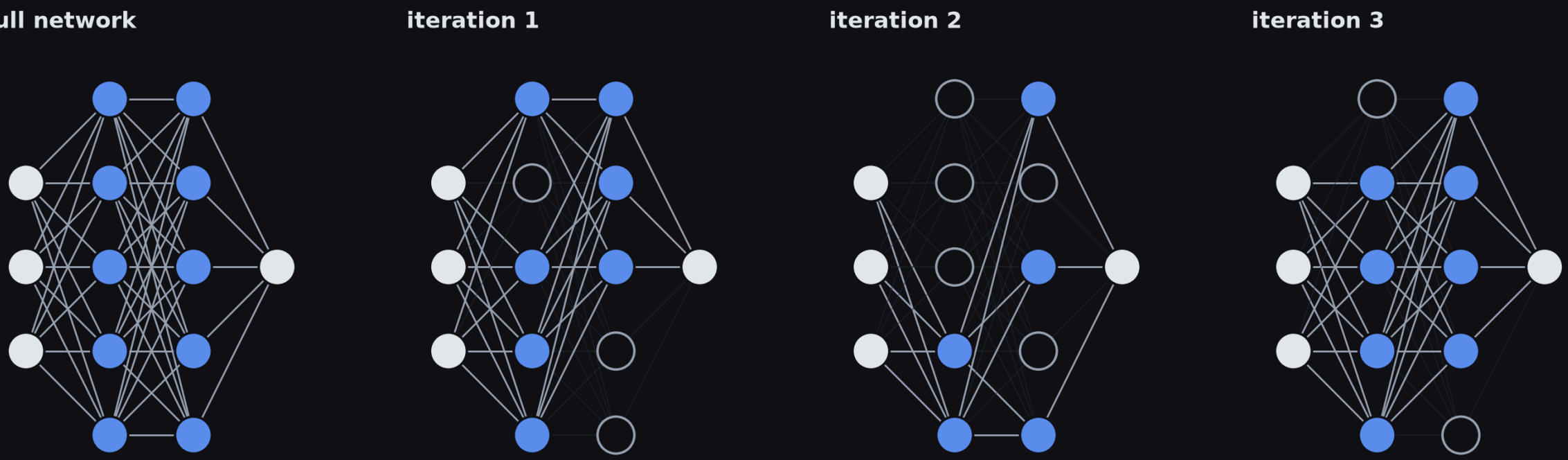
- **Drop probability** p: the probability that a unit is dropped
  - PyTorch's `p` means the same
- **Mask** $\mathbf{m}$: one entry per unit, the same shape as $\mathbf{h}$
  - $m_j = 1$ (kept) with probability $1 - p$, and $m_j = 0$ (dropped) otherwise
  - i.e. $m_j \sim \text{Bernoulli}(1 - p)$
  - drawn independently for each unit, afresh at each iteration
- **Masked activation**: $\widetilde{\mathbf{h}} = \mathbf{m} \odot \mathbf{h}$, the elementwise product

$$\mathbf{h} = \begin{bmatrix} 1.2 \\ 0.4 \\ 2.0 \\ 0.7 \end{bmatrix}, \qquad \mathbf{m} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\widetilde{\mathbf{h}} = \mathbf{m} \odot \mathbf{h} = \begin{bmatrix} 1.2 \\ 0 \\ 2.0 \\ 0 \end{bmatrix}.$$

- a dropped unit’s incoming and outgoing contributions vanish for that iteration

Masks across iterations

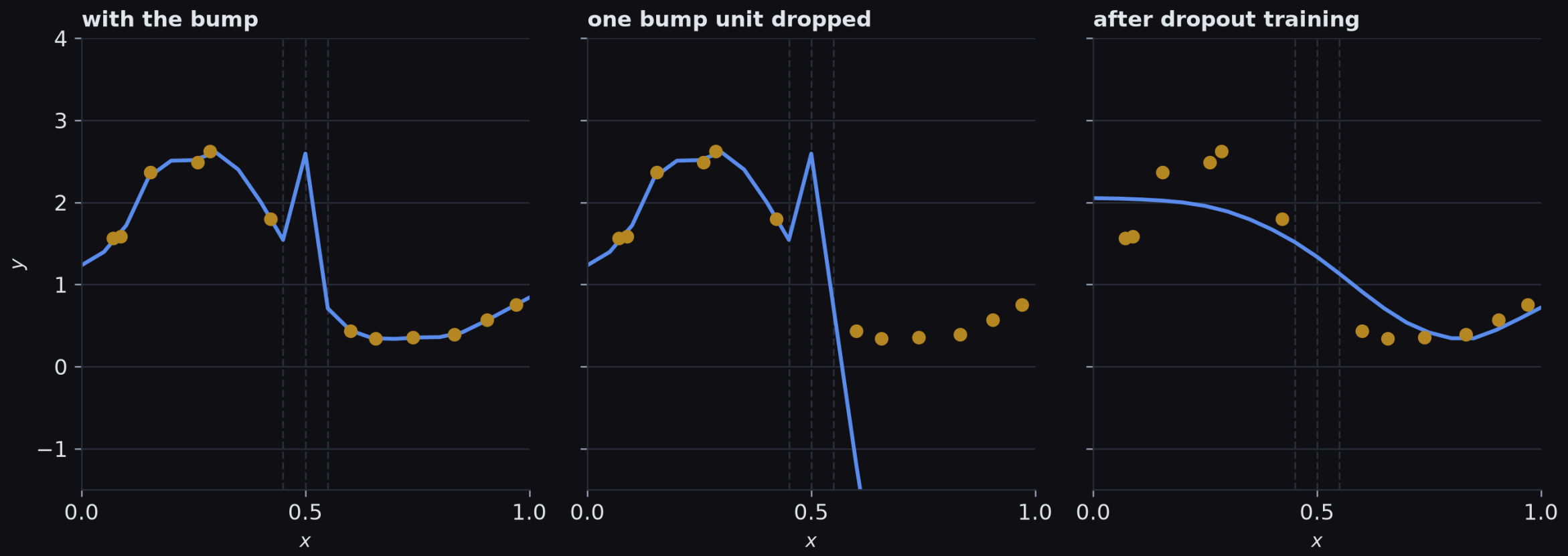


One network, four training iterations. Each iteration samples a fresh mask; dropped hidden units (hollow) take their incoming and outgoing connections with them. Every subnetwork uses the same underlying weights, so each update trains part of one shared model.

Brittle cancellation

- **Setup**: several hidden units cooperate to build a local hinge
  - in a region with no training observations
  - so the training loss doesn’t change
- **Why no gradient**: the hinge costs nothing, so gradient descent has no reason to remove it
- **What dropout does**: drops one of the cooperating units
  - the cancellation fails, exposing the dependence
  - the resulting gradient pushes toward a less brittle solution

Dropout and a hidden bump



Left: a fit whose three hidden units at the dashed joints build a bump in a gap between the training points; they cancel exactly everywhere else, so the bump costs no training loss. Centre: drop one of the three and the cancellation fails, and the function moves far from the data. Right: after training with dropout, the bump's weights have shrunk and the gap is spanned smoothly, at the cost of a looser fit to the data everywhere. The same training without dropout leaves the bump in place.

Dropout scaling conventions

- **Inference scaling**: the chapter's convention
  - training masks only, with no rescaling

$$\tilde{\mathbf{h}} = \mathbf{m} \odot \mathbf{h}, \qquad \mathbb{E}\left[\tilde{h}_j\right] = (1-p)h_j$$

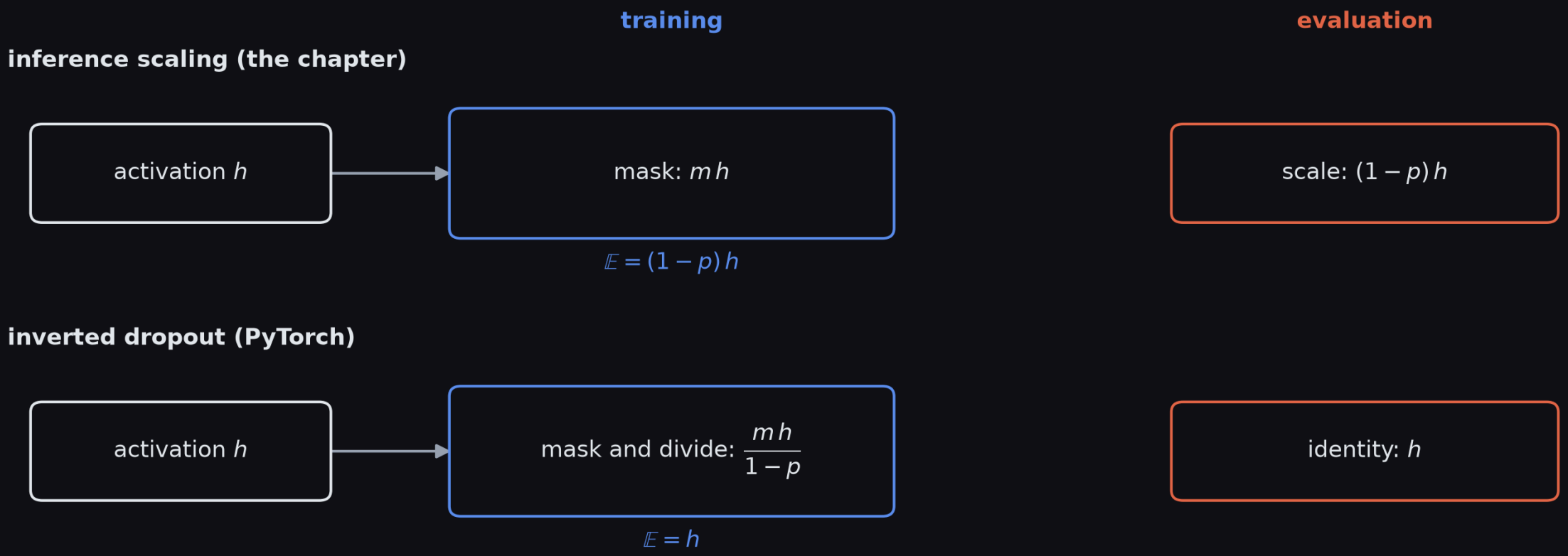
- **At inference**: multiply the outgoing weights by $1 - p$
  - this is the **weight scaling inference rule**
- **Exactness**: matches the expected next-layer pre-activation exactly
  - through later nonlinearities, it only approximates averaging over masks

Definition 9 **Inverted dropout** divides the kept activations by $1 - p$ during training, so evaluation needs no rescaling.

$$\widetilde{\mathbf{h}} = \frac{\mathbf{m} \odot \mathbf{h}}{1 - p}, \qquad \mathbb{E}\left[\widetilde{\mathbf{h}}\right] = \mathbf{h}$$

- **PyTorch**: uses this convention
- **At evaluation**: `nn.Dropout` is the identity operation

Two dropout conventions



The same activation h and drop probability p under both conventions. Top, inference scaling: training only masks, so a unit's expected output is (1 – p)h, and evaluation multiplies by 1 – p to match. Bottom, inverted dropout, as PyTorch implements it: training masks and divides by 1 – p, so the expectation is already h and evaluation is the identity.

Inverted dropout's expected scale

- **Setup**: $h = 2$, $p = 0.5$, inverted dropout
- Try it: what is $\mathbb{E}\left[\tilde{h}\right]$?
- **Dropped**: value 0, with probability 0.5
- **Kept**: value $2/(1 - 0.5) = 4$, with probability 0.5

$$\mathbb{E}\left[\tilde{h}\right] = 0.5(0) + 0.5(4) = 2 = h.$$

- the training activation has the same expected scale as the evaluation activation

Monte Carlo dropout

- **Standard inference**: dropout's randomness is switched off
- **Monte Carlo dropout** keeps it on:
  - sample a dropout mask and predict
  - repeat several times
  - combine the predictions
- **Relation to ensembles**: an ensemble-like prediction without storing separately trained networks

Dropout as multiplicative noise

- a Bernoulli mask is **multiplicative noise** on the hidden activations
- **Other places to perturb**:
  - inputs
  - weights
  - hidden activations
  - labels
- **Common question**: should the desired prediction stay stable under that perturbation?

Input noise

- **Perturbed input**: at each step, replace $\mathbf{x}$ with

$$\widetilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\epsilon}.$$

- **Assumption**: nearby inputs should have similar outputs
- **Effect**: fitting many perturbed copies discourages sharp local changes in the learned function

Input noise and the input Jacobian

- **Remember how the Jacobian collected all the first-order sensitivities of one vector to another?**
- In week 2 we differentiated network outputs with respect to the parameters
- Here we ask a different sensitivity question: **how much does the output move when the input moves?**
- So take the derivatives with respect to the input
- **Shapes**: $\mathbf{x} \in \mathbb{R}^{D_i}$, $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] \in \mathbb{R}^{D_o}$
- **Input Jacobian**:

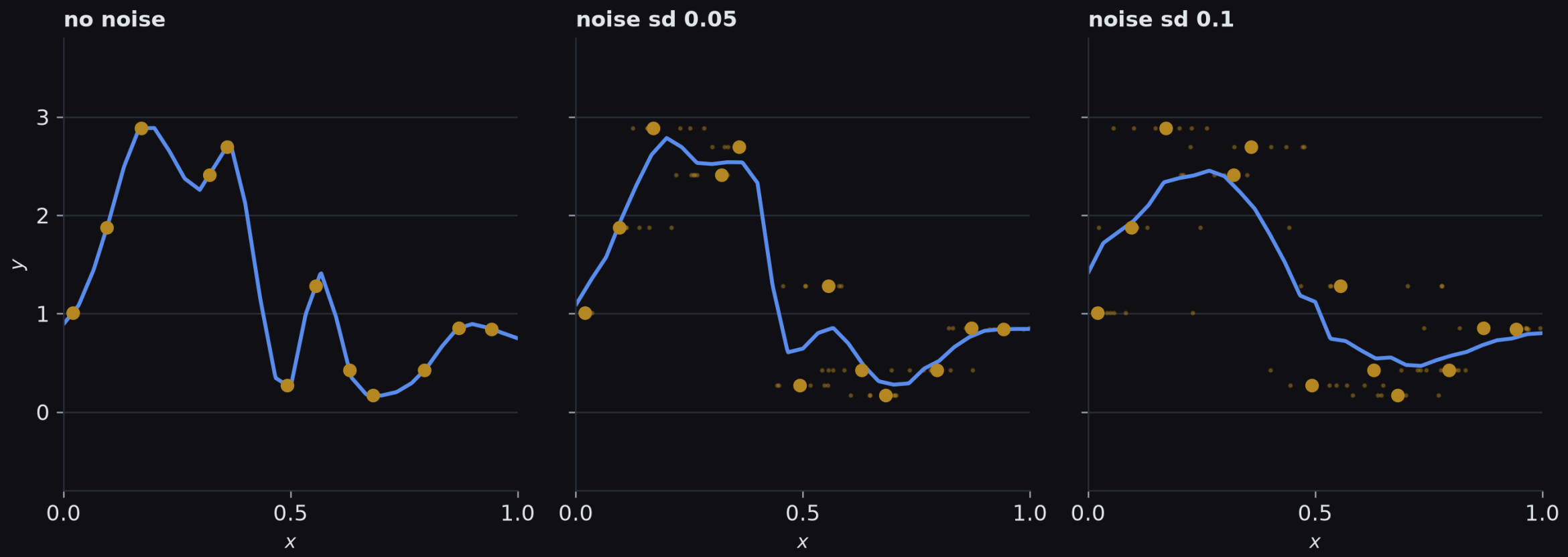
$$\mathbf{J}_{\mathbf{x}} = \frac{\partial \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]}{\partial \mathbf{x}} \in \mathbb{R}^{D_o \times D_i}.$$

- **First-order change**: for a small perturbation $\delta\mathbf{x}$,

$$\mathbf{f}[\mathbf{x} + \delta\mathbf{x}, \boldsymbol{\phi}] \approx \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] + \mathbf{J}_{\mathbf{x}}\, \delta\mathbf{x}.$$

- **Regression**: input-noise training can be related to a regularisation term that penalises these derivatives
  - large input Jacobian: a small perturbation can change the prediction substantially

Input noise on the toy regression



The same twelve observations and the same thirty-joint model, fitted to many perturbed copies of each input. The faint dots are some of those copies. With no noise the fit interpolates; as the noise standard deviation grows, the fitted function gets smoother.

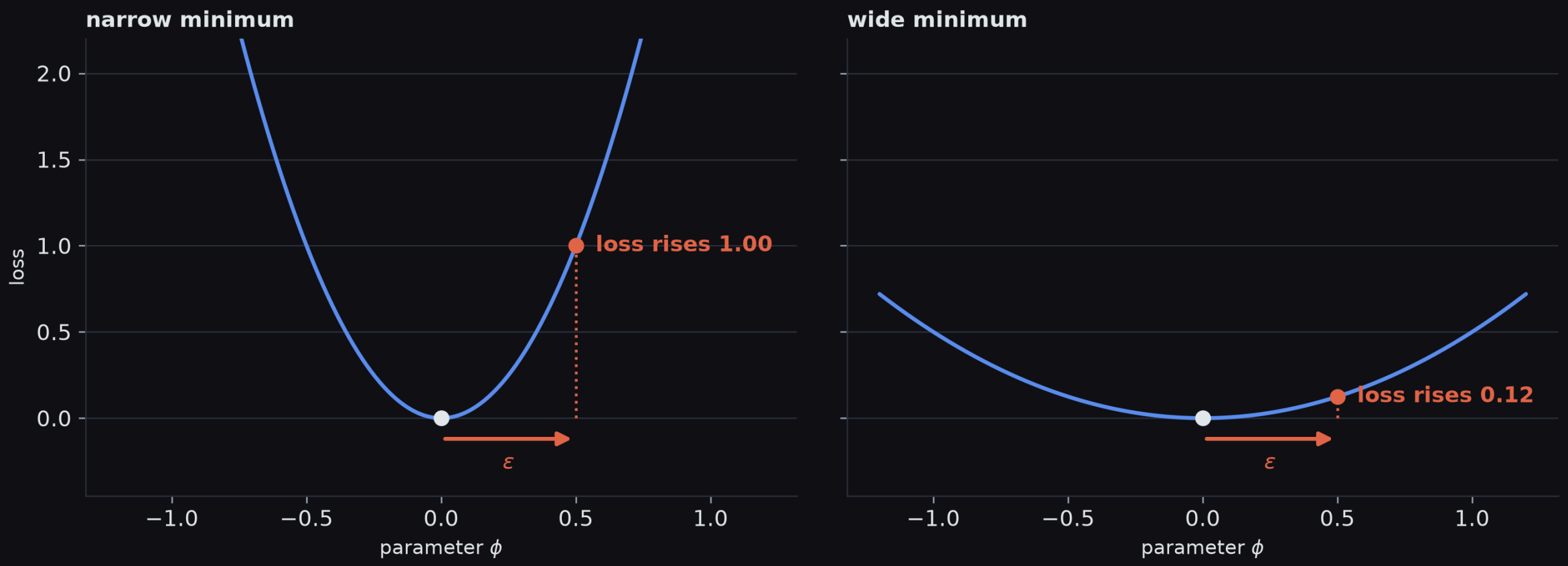
Weight noise

- **Perturbed parameters**: during training, use

$$\widetilde{\phi} = \phi + \epsilon.$$

- a useful solution must keep working for nearby parameter values
- optimisation is pushed toward regions where small parameter errors have little effect
  - the intuition behind seeking **wide** or **flat minima**
- **Limit**: a better-generalisation claim for wide minima is a hypothesis with supporting evidence, not a theorem
  - flatness depends on how the parameters are scaled, one reason the link is not a theorem

Narrow and wide minima



Two minima with the same loss. The same parameter perturbation ε raises the loss sharply in the narrow basin and only a little in the wide one. The link from wide minima to generalisation is an intuition with supporting evidence, not a theorem.

Adversarial training

- **Input noise**: samples a perturbation from a fixed distribution
- **Adversarial training**: searches for a small perturbation that causes a large change in model behaviour
  - a worst-case local perturbation
- an extreme form of training for stability under input perturbation

Overconfident classification

- **One-hot target**: the likelihood objective rewards pushing the labelled class's probability toward 1
- **Consequence**: the logits are pushed toward ever more extreme values

Definition 10 **Label smoothing** replaces the one-hot target with a distribution that keeps most of the probability mass on the labelled class and spreads the rest over the other classes.

Soft targets stop rewarding unlimited logit separation

A one-hot target says

$$[0, 0, 1, 0, 0]$$

and cross-entropy keeps rewarding a model for pushing the labelled class closer and closer to probability 1

- once the class is already correct, ever larger logit gaps can still reduce the loss
- a one-hot target assigns **zero target mass to every other class**, so cross-entropy continues to reward increasing separation
- label smoothing changes the training signal so extreme concentration is no longer the only way to improve it

The aim is not to make the model uncertain at random

It is to stop the target from demanding ever larger logit separation after the class is already correct

Label smoothing

- **Setup**: K classes, true class c, smoothing parameter ρ

$$\tilde{y}_c = 1 - \rho, \qquad \tilde{y}_k = \frac{\rho}{K-1} \quad (k \neq c).$$

- Try it: write the smoothed target for $K = 5$ and $\rho = 0.20$

$$[0, 0, 1, 0, 0] \longrightarrow [0.05, 0.05, 0.80, 0.05, 0.05].$$

Cross-entropy with a soft target

- **Predicted probabilities**: $\hat{y}_1, \ldots, \hat{y}_K$
- **Soft target**: $\tilde{y}_1, \ldots, \tilde{y}_K$

$$\ell_i = -\sum_{k=1}^{K} \tilde{y}_k \log \hat{y}_k.$$

- **Correct class**: still rewarded strongly
- **Concentration**: a distribution arbitrarily concentrated on it is now penalised
- **What changes**: the training target; the network architecture is untouched

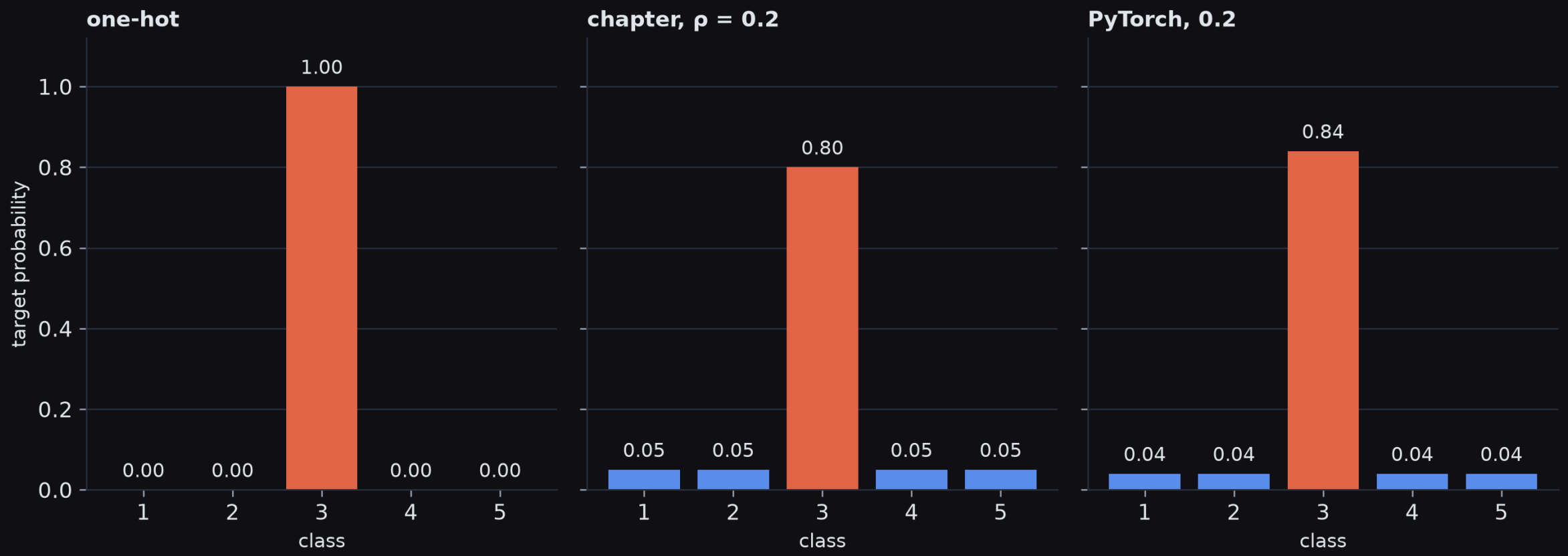
PyTorch's smoothing convention

- **PyTorch**: `CrossEntropyLoss(label_smoothing=eps)`, with ε = `eps`
  - mixes the one-hot target with a uniform distribution over all K classes
- For $K = 5$ and $\varepsilon = 0.20$:

$$[0, 0, 1, 0, 0] \longrightarrow [0.04, 0.04, 0.84, 0.04, 0.04].$$

- **Chapter,** $\rho = 0.20$: $[0.05, 0.05, 0.80, 0.05, 0.05]$
- **Same idea, different parameterisation**: equal coefficients don't mean equal targets

Three target distributions



The target for a five-class example whose true class is 3. Left: one-hot. Centre: the chapter's smoothing with ρ = 0.2, which spreads ρ over the four wrong classes. Right: PyTorch's label_smoothing = 0.2, which mixes in a uniform distribution over all five, so the true class keeps more mass.

Parameter uncertainty

One fitted parameter vector hides alternatives

- Standard training usually returns one estimate $\widehat{\phi}$
- But several parameter vectors may explain the observed data almost equally well
- Those alternatives can make different predictions away from the training examples
- A point estimate hides that remaining uncertainty

Bayesian inference keeps a distribution over plausible parameter vectors instead of collapsing immediately to one.

- This is the same motivation that made ensembles useful earlier: different plausible models can disagree

Remember maximum likelihood?

From the loss-function session, the likelihood asks

$$p(\mathcal{D} \mid \phi) :$$

> if these were the parameters, how compatible would the observed data be?

- **Maximum likelihood** picks the single ϕ that makes the observed data most likely
- It uses the data, but it does not represent uncertainty over which parameter values remain plausible

The likelihood answers a **forward** question: parameters $\rightarrow$ possible data

The Bayesian question reverses the conditioning

For inference, the question we actually want is

$$p(\phi \mid \mathcal{D}) :$$

> after seeing these data, how plausible is each parameter vector?

- $p(\mathcal{D} \mid \phi)$ and $p(\phi \mid \mathcal{D})$ are different quantities
- We cannot simply swap the two sides of the conditioning bar

Bayes' rule tells us how to reverse that conditioning while accounting for what was plausible before the data arrived.

Bayes as reweighting

Suppose only two candidate models are possible

candidate	prior	likelihood of observed data	prior $\times$ likelihood
A	0.75	0.20	0.15
B	0.25	0.80	0.20

- **Before the data**: candidate A is three times as plausible as B
- **Evidence from the data**: the observations are four times as likely under B
- **Combine them**: multiply prior by likelihood
- **Normalise**: divide by $0.15 + 0.20 = 0.35$

$$\Pr(A \mid \mathcal{D}) \approx 0.43, \qquad \Pr(B \mid \mathcal{D}) \approx 0.57.$$

Bayes updates relative plausibility: **prior $\times$ likelihood, then renormalise**

Bayes' rule

Remember conditional probability:

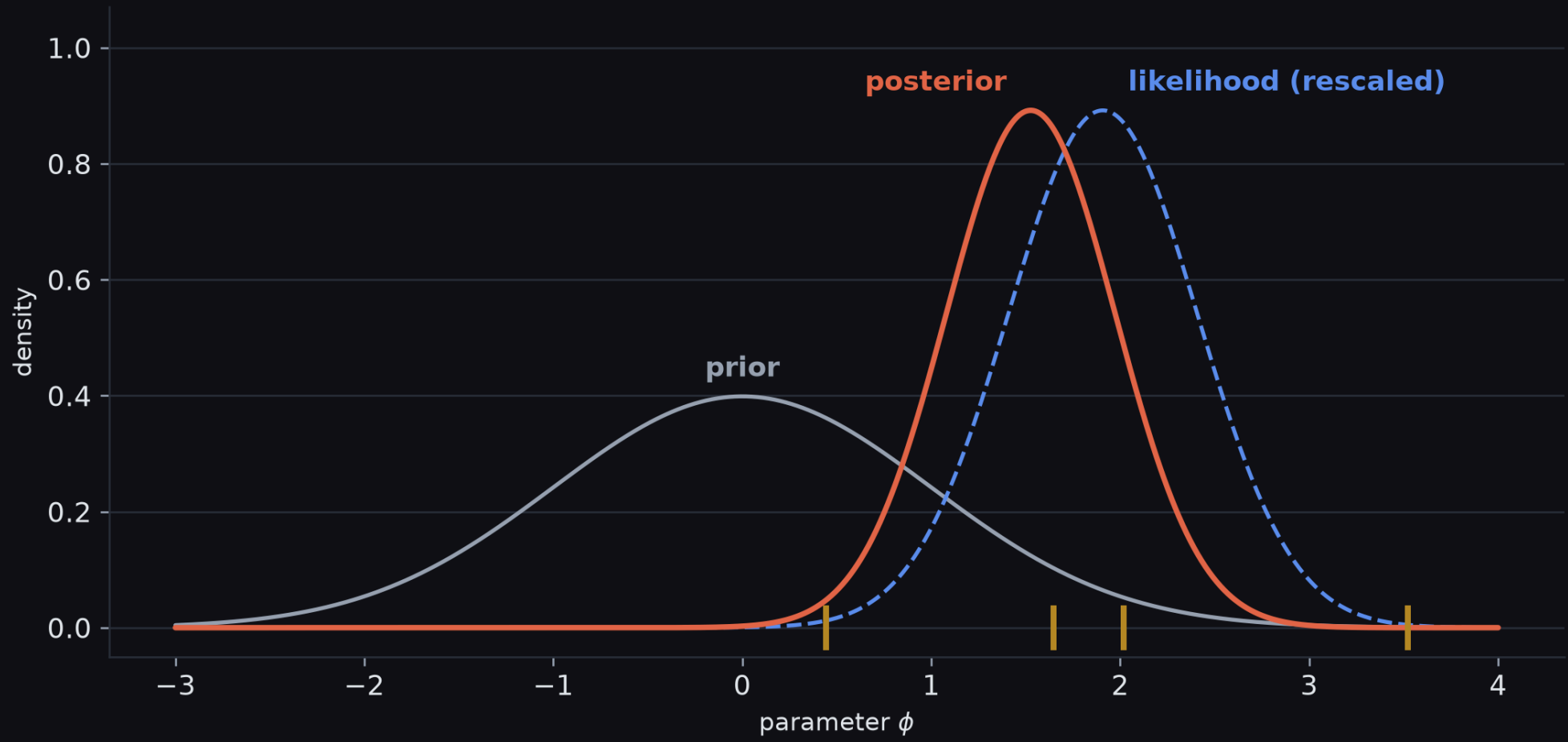
$$\Pr(A \mid B) = \frac{\Pr(B \mid A)\Pr(A)}{\Pr(B)}.$$

For parameters and data, read the same structure as

$$\underbrace{p(\boldsymbol{\phi} \mid \mathcal{D})}_{\text{posterior}} \propto \underbrace{p(\mathcal{D} \mid \boldsymbol{\phi})}_{\text{likelihood}}\underbrace{p(\boldsymbol{\phi})}_{\text{prior}}.$$

- **Prior**: plausibility before these observations
- **Likelihood**: how well a candidate explains the observations
- **Posterior**: updated plausibility after combining both
- The missing proportionality constant is the **evidence** that makes the posterior integrate to one

Prior, likelihood and posterior



One scalar parameter φ: the mean of four noisy observations (ticks). The prior is a Gaussian centred on 0; the likelihood peaks near the observations and is rescaled here to share the axis, since it is not a density in φ. Their product, normalised, is the posterior: between the two, and narrower than the prior.

From one parameter to a neural network

For the full parameter vector $\boldsymbol{\phi}$ and training data

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{I},$$

the posterior is

$$p(\boldsymbol{\phi} \mid \mathcal{D}) = \frac{\left[\prod_{i=1}^{I} \Pr(y_i \mid \mathbf{x}_i, \boldsymbol{\phi})\right] p(\boldsymbol{\phi})}{\int \left[\prod_{i=1}^{I} \Pr(y_i \mid \mathbf{x}_i, \boldsymbol{\phi}')\right] p(\boldsymbol{\phi}')\, d\boldsymbol{\phi}'}.$$

- The numerator is **likelihood** $\times$ **prior** for one candidate parameter vector
- The denominator sums or integrates those weights over every possible candidate
- That denominator is the **evidence**: it normalises the posterior

MAP collapses the posterior back to one vector

Sometimes we still want one parameter vector rather than a full distribution

Definition 11 The **maximum a posteriori (MAP)** estimate is the parameter vector at the mode of the posterior.

$$\hat{\boldsymbol{\phi}}_{\text{MAP}} = \arg\max_{\boldsymbol{\phi}} \left[p(\boldsymbol{\phi}) \prod_{i=1}^{I} \Pr(y_i \mid \mathbf{x}_i, \boldsymbol{\phi}) \right].$$

- The evidence disappears from the $\arg\max$ because it does not depend on $\boldsymbol{\phi}$
- **MAP keeps one Bayesian idea**: the prior can influence which fitting solution is selected
- **MAP discards another**: uncertainty represented by the rest of the posterior

The L2 penalty has a Bayesian interpretation

Earlier, L2 was introduced geometrically:

> prefer a shorter weight vector when data fit is comparable

Now ask the Bayesian version of the same modelling choice:

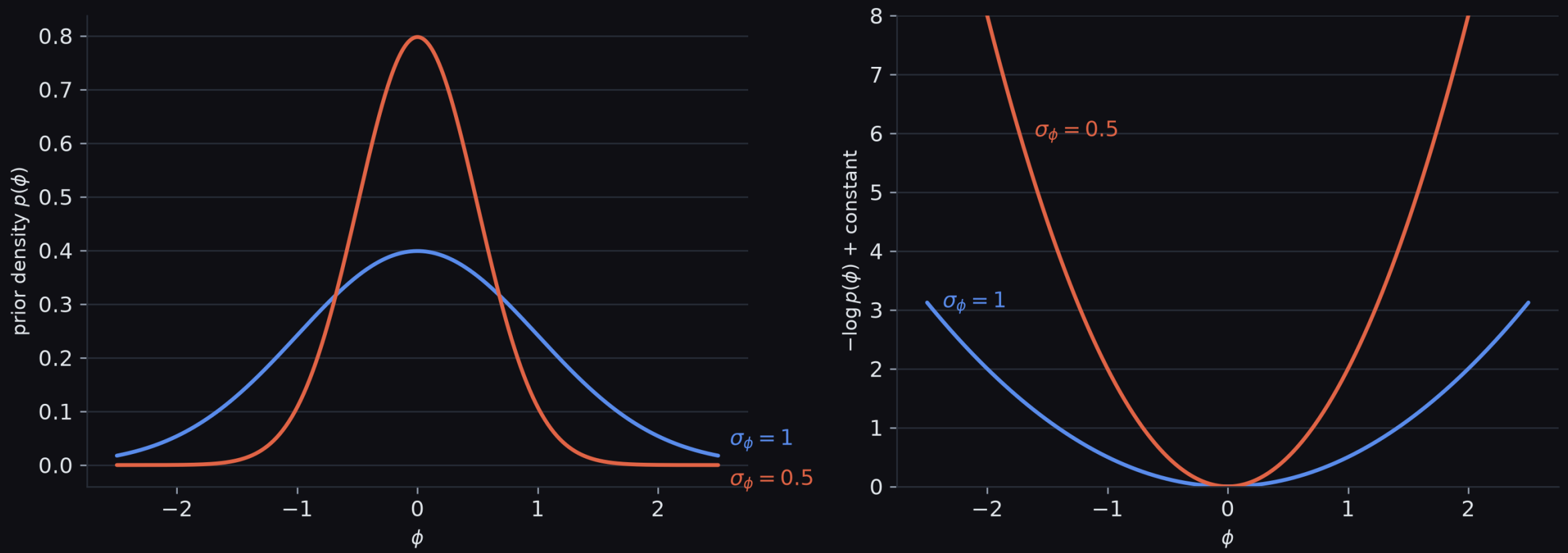
> what prior would express a soft, symmetric preference for weights near zero?

An independent zero-centred Gaussian prior does exactly that

$$\phi_j \sim \mathcal{N}(0, \sigma_\phi^2).$$

- zero is the centre of the preference
- positive and negative weights of equal magnitude are treated symmetrically
- large weights remain possible, but become progressively less plausible
- this is a **modelling choice**, not a claim that trained weights are literally sampled from a Gaussian

A Gaussian prior and its negative log



A zero-centred Gaussian prior and the penalty it implies. Left: smaller prior variance concentrates more density near zero. Right: taking the negative log turns each Gaussian into a quadratic bowl; the narrower prior gives the steeper penalty.

MAP with a Gaussian prior recovers L2

For independent Gaussian weight priors,

$$p(\boldsymbol{\phi}) \propto \exp\left[-\frac{1}{2\sigma_\phi^2}\sum_j \phi_j^2\right].$$

Take the negative log of the MAP objective:

$$-\log p(\mathcal{D} \mid \boldsymbol{\phi}) + \frac{1}{2\sigma_\phi^2}\sum_j \phi_j^2 + \text{constant}.$$

Compare with the L2 objective from earlier:

$$\mathcal{L}(\boldsymbol{\phi}) + \lambda \sum_j \phi_j^2.$$

$$\boxed{\lambda = \frac{1}{2\sigma_\phi^2}}$$

Two readings of the same L2 penalty

Geometric reading

- penalise distance from the origin in parameter space
- gradient descent gets an inward pull proportional to each weight

Probabilistic reading

- use an independent zero-centred Gaussian prior over the penalised weights
- MAP trades data fit against prior plausibility

A smaller prior variance means a stronger belief in weights near zero, and therefore a larger L2 coefficient λ

This connection **reinterprets** L2; it is not needed in order to use L2 regularisation

Posterior prediction keeps the uncertainty MAP throws away

Suppose two parameter settings both remain plausible after training, but they disagree on a new input

- MAP keeps one of them and discards the other
- the full posterior keeps both possibilities
- prediction should therefore reflect the remaining parameter uncertainty

The Bayesian rule is: ask every plausible parameter vector for its prediction, then weight that prediction by its posterior plausibility

The posterior predictive distribution

For a new input $\mathbf{x}$,

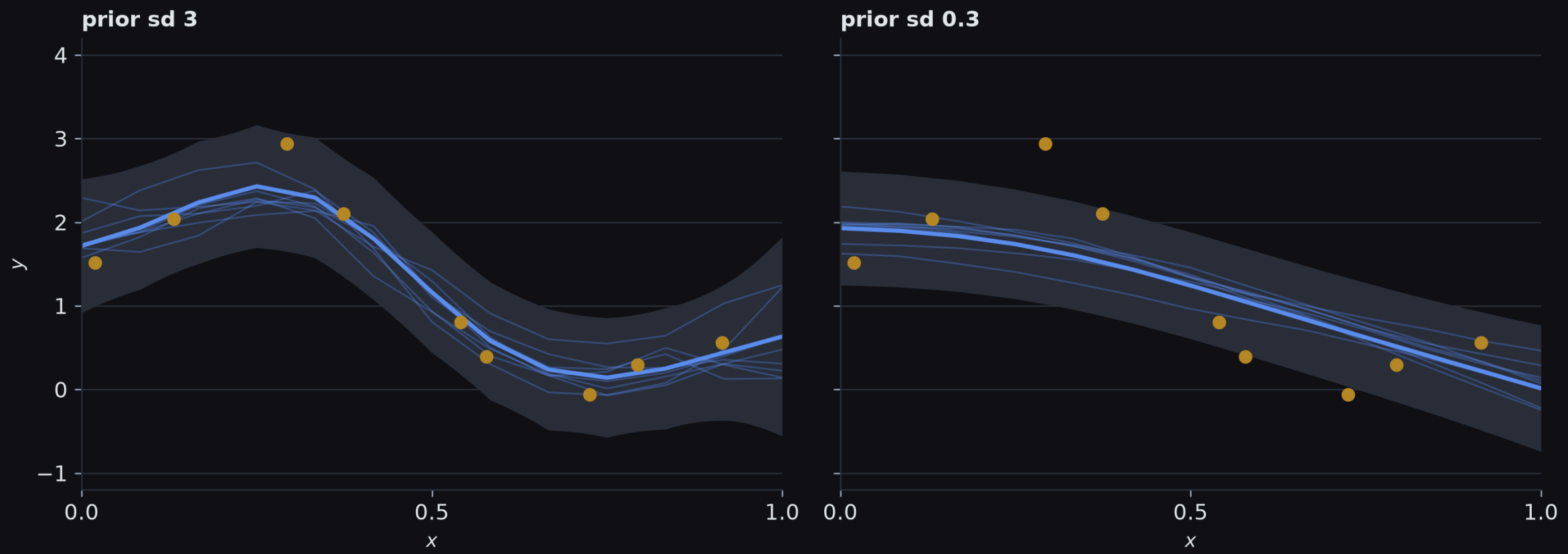
$$\Pr(y \mid \mathbf{x}, \mathcal{D}) = \int \Pr(y \mid \mathbf{x}, \boldsymbol{\phi})\, p(\boldsymbol{\phi} \mid \mathcal{D})\, d\boldsymbol{\phi}.$$

Definition 12 The **posterior predictive distribution** averages the prediction from every parameter vector, weighted by its posterior density.

Bayesian prediction as an ensemble

- **Members**: every candidate $\boldsymbol{\phi}$ defines a model prediction $\Pr(y \mid \mathbf{x}, \boldsymbol{\phi})$
- **Weights**: posterior density $p(\boldsymbol{\phi} \mid \mathcal{D})$
- **Combination**: the integral is an infinite weighted ensemble
- **Contrast**: the ensembles earlier today average a finite collection of separately fitted networks

Posterior samples and the predictive band



Bayesian inference for the fixed-joint model with a Gaussian prior on its weights, computed in closed form. Thin lines are the functions given by weight vectors drawn from the posterior; the thick line is the predictive mean and the band two predictive standard deviations. Left, a broad prior: the functions follow the data and spread furthest at the right-hand end. Right, a narrow prior: the functions are held to a smoother trend, agree with one another, and miss the peak.

Bayesian neural networks in practice

- For a complex neural network:
  - the posterior over all parameters has no general closed form
  - the posterior predictive integral cannot generally be evaluated exactly
- Practical approximations include:
  - variational Bayes
  - Markov chain Monte Carlo
  - stochastic-gradient MCMC variants
- **Trade-off**: explicit parameter uncertainty, at substantial computational and modelling cost

Additional training signal

Borrowing signal from other data

- **Problem**: the target task has little labelled data
- **Source**: other data supply a representation, before or alongside target-task fitting
- **Three routes**:
  - transfer learning
  - multi-task learning
  - self-supervised learning

Transfer learning

1. pre-train on a related task with more data
2. remove or replace the task-specific output layer
3. attach a head for the target task
4. train only the new head, or fine-tune some or all of the network
    - **Representation view**: reuse features learned on the related task
    - **Optimisation view**: start from a parameter region already shaped by that task

Freezing vs fine-tuning

- **Freeze the backbone**: fit only the new output layers
  - assumes the learned representation already transfers
- **Fine-tune**: let the representation itself adapt to the target task
- **Deciding factors**:
  - volume of target data
  - similarity of the two tasks
  - cost of changing the pre-trained representation

Multi-task learning

- **Setting**: one network learns several related objectives at once
- **Example**, from one image:
  - segmentation
  - pixel-wise depth
  - a caption
- **Shared representation**: every task needs some understanding of the same input
  - so each objective may benefit from the others

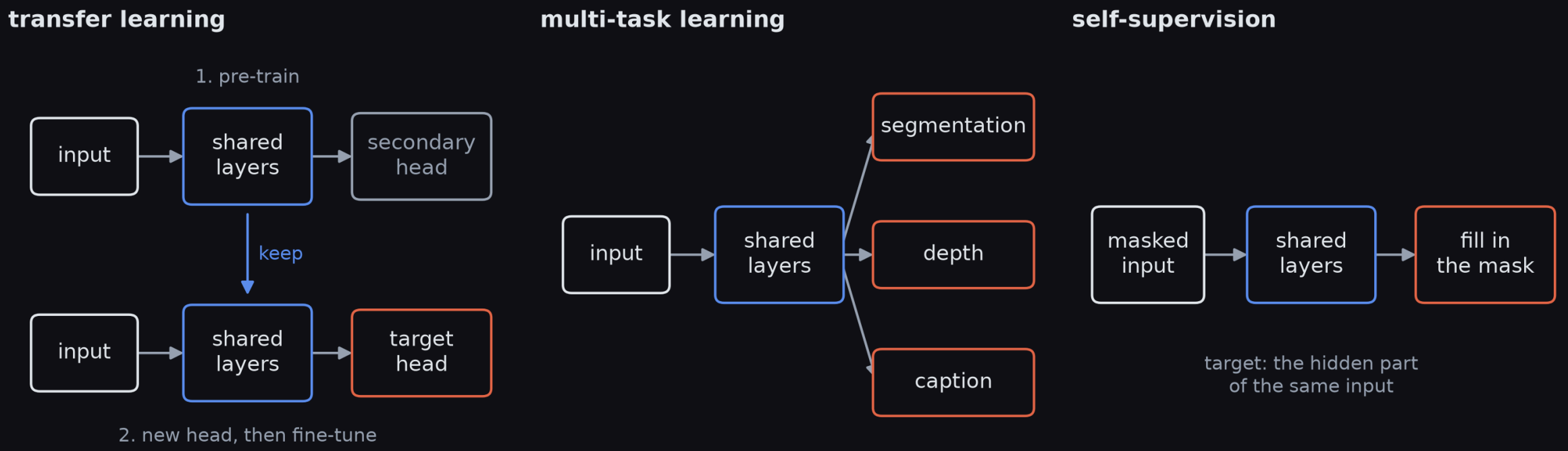
Generative self-supervision

- **Task**: hide part of an example, predict the missing content
- **Examples**:
  - masked image region: inpaint it
  - masked text token: predict it
- **Target source**: the input itself
  - so large unlabelled datasets become training material

Contrastive learning

- **Task**: compare examples that should be related with examples that should not
  - no reconstruction of missing content
- **Examples**:
  - transformed versions of one image vs unrelated images
  - consecutive sentences vs unrelated sentences
  - spatially related patches of one image
- **Signal**: which examples should share structure, with no human class label

Transfer, multi-task and self-supervised learning



One visual grammar for three ways to borrow training signal: a shared representation (blue) with task-specific heads. Transfer learning trains on a secondary task, then replaces the head. Multi-task learning trains several heads at once. Generative self-supervision masks part of an unlabelled input and trains the network to fill it in.

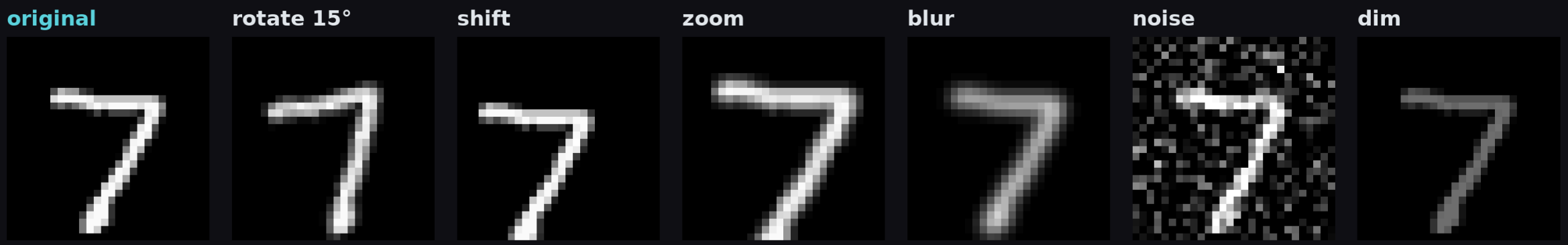
Data augmentation

- **Setting**: an image labelled *bird*, for ordinary bird classification
- **Label-preserving transformations**:
  - modest rotation
  - horizontal flip
  - blur
  - colour changes
- **Effect**: the model learns insensitivity to variation declared irrelevant to the task

Augmentation as an assumption

- **Condition**: useful only when the correct target stays unchanged
- Pick a task: name one transformation that would break its label
- **Failure cases**:
  - horizontal flip for text or asymmetric symbols
  - large crop that removes the labelled object
  - colour change when colour determines the class
  - word substitution that changes sentence meaning
- ***Augmentation asserts an invariance; it does not create truth.***

Augmented copies of one digit



label: 7, for every one

One MNIST digit and six transformed copies. Each copy is a new training input with the same label, 7; together they tell the network which changes should not move its prediction. MNIST © Yann LeCun and Corinna Cortes, CC BY-SA 3.0.

Supervision without new target labels

- **Transfer learning**: another labelled dataset
- **Multi-task learning**: additional labels on related tasks
- **Self-supervision**: targets generated from unlabelled data
- **Augmentation**: task-preserving transformations of existing examples
- **Shared mechanism**: constrain the representations and predictions to fit a wider body of evidence

Choosing a regularisation strategy

Mechanisms of regularisation

- The methods can be grouped by how they improve generalisation
- **Smoother functions**: bias the fit towards a function that changes slowly
- **More effective data**: add examples, or information, the fit can use
- **Combining models**: average over several fits to reduce fitting uncertainty
- **Wider minima**: prefer a minimum where small parameter errors matter less

Methods by mechanism

smoother functions

- L2
- early stopping
- input noise
- label smoothing
- ensembling
- Bayesian approach
- dropout

more effective data

- augmentation
- transfer learning
- multi-task learning
- input noise
- label smoothing

combine models

- ensembling
- Bayesian approach
- dropout

wider minima

- implicit regularisation
- noise on weights
- dropout

Four broad regularisation mechanisms and representative methods. Methods highlighted in cyan act through more than one mechanism: input noise, label smoothing, ensembling and the Bayesian approach through two, dropout through three.

Methods with several mechanisms

- **Dropout**: smoother functions, combining models and wider minima
- **Input noise, label smoothing**: smoother functions and more effective data
- **Ensembling, Bayesian approach**: smoother functions and combining models
- **Use**: a reasoning aid; the groups overlap

Failure modes and interventions

observed concern	plausible intervention
highly sample-sensitive fit	L2, early stopping, bagging
brittle hidden-unit dependencies	dropout
sensitivity to small input changes	input noise, adversarial training
overconfident class probabilities	label smoothing
limited labelled target data	transfer, self-supervision, augmentation
uncertainty across plausible fitted models	ensemble, Bayesian treatment

- **Status**: each row is a hypothesis to validate

Choosing the strength

- Each method brings a hyperparameter
  - L2 coefficient λ
  - dropout probability p
  - stopping time
  - noise scale
  - label-smoothing strength
  - augmentation strength and frequency
- More regularisation is not monotonically better
- **Right amount depends on**: data, architecture, optimiser and task

Framework calls

```
# L2-style penalty under SGD
optimizer = torch.optim.SGD(
    weight_params,
    lr=lr,
    weight_decay=wd,
)

# Decoupled weight decay
optimizer = torch.optim.AdamW(
    weight_params,
    lr=lr,
    weight_decay=wd,
)
```

- **Check**: know the update or target each call implements

What each method changes

method	what changes?
L2	objective / parameter penalty
finite-step GD / SGD	optimisation trajectory
early stopping	optimisation duration
ensemble	prediction rule across fitted models
dropout	stochastic hidden activations
input / weight noise	training computation
label smoothing	target distribution
Bayesian inference	treatment of parameter uncertainty
transfer / self-supervision	information available to fitting
augmentation	training examples and assumed invariances

Revision

Distinctions to keep straight

distinction	do not conflate
explicit vs implicit	a written penalty in the objective vs solution-selection bias introduced by optimisation
L2 vs weight decay	squared-parameter penalty vs direct parameter shrinkage; equivalent only for particular update rules
training vs inference dropout scaling	compensate at inference vs inverted dropout during training; use one convention, not both
label-smoothing coefficients	the same nominal coefficient can define different target distributions
regularisation vs validation	regularisation supplies candidate biases; validation chooses among their strengths and variants

- Regularisation can modify
  - the objective
  - the optimisation path
  - the training duration
  - a stochastic computation
  - the targets
  - the prediction rule
  - the information available to fitting

Regularisation: revision summary

- explicit regularisation adds a penalty term to the objective
- MAP estimation interprets the negative log-prior as such a term
- a zero-centred Gaussian prior yields an L2 penalty
- L2 regularisation and weight decay coincide only under particular update rules
- finite-step GD and minibatch SGD can bias the optimisation trajectory even when the written loss is unchanged
- early stopping chooses a validation-supported point along the optimisation path
- ensembles combine diverse fitted solutions; bagging creates diversity by bootstrap resampling
- dropout trains changing subnetworks and requires a clearly understood scaling convention
- noise, label smoothing, Bayesian inference, transfer learning and augmentation regularise through different mechanisms
- regularisation strength and method remain empirical model-selection choices

Source and revision locators

- Simon J. D. Prince, *Understanding Deep Learning*, Chapter 9: “Regularisation” (Prince 2023)
- **Sections**: 9.1 explicit regularisation; 9.2 implicit regularisation; 9.3 heuristics to improve performance; 9.4 summary
- **Core equations**: 9.1–9.13
- **Notebooks**: 9.1 L2 regularisation; 9.2 implicit regularisation; 9.3 ensembling; 9.4 Bayesian approach; 9.5 augmentation
- **Figures worth revisiting**: 9.1–9.14, especially the matched L2, implicit-SGD, early-stopping, dropout and augmentation sequences
- **Problems**: 9.1–9.6 for explicit regularisation, noise, label smoothing, weight decay and other norms

Prince, Simon J. D. 2023. *Understanding Deep Learning*. MIT Press. https://udlbook.github.io/udlbook/.