Lecture 4: Fitting Models

Block 4.2 Advanced AI

Eoin O’Brien

Fitting as optimisation

Recap

- A neural network is a **family of functions** indexed by its parameters
- A loss turns the quality of one particular parameter setting into **one scalar number**
- Training asks a new question:

> **How should we change the parameters so that this number gets smaller?**

derivative → **partial derivative** → **gradient** → **curvature** → **optimiser**.

Learning goals

By the end of this lecture, you should be able to:

- explain training as **optimisation in parameter space**
- interpret a **partial derivative** as a local sensitivity
- read $\nabla_\phi \mathcal{L}$ and explain why the **gradient** gives a downhill update direction
- use the **chain rule** to derive a simple loss gradient
- compute the gradient of least squares in both **scalar** and **vector** form
- interpret the **Jacobian** as a local linear map
- interpret Hessian **eigenvectors and eigenvalues** as principal curvature directions and curvatures
- compare **gradient descent, SGD, momentum, and Adam** conceptually

The fitting problem

Suppose a model predicts

$$\hat{\mathbf{y}} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] \,,$$

where $\boldsymbol{\phi}$ contains all trainable parameters.

A loss function assigns one scalar to the model's performance on the training data:

$$\mathcal{L}[\boldsymbol{\phi}].$$

Training means searching for

$$\hat{\boldsymbol{\phi}} = \arg\min_{\boldsymbol{\phi}} \mathcal{L}[\boldsymbol{\phi}].$$

In optimisation texts, $\mathcal{L}$ is often called the **objective function**. In machine learning, **loss** and **cost** are also common. Here, the objective is to minimise the loss.

What does $\arg\min$ mean?

The expression

$$\hat{\phi} = \arg\min_{\phi} \mathcal{L}[\phi]$$

is easier to read from the inside out.

- $\mathcal{L}[\phi]$: **how bad is this parameter setting?**
- $\min$: find the smallest achievable loss
- $\arg\min$: return the **argument** — the parameter values — that achieve it
- $\hat{\phi}$: our fitted estimate of those parameters

The optimiser changes **parameters**. Those parameters determine the predictions, and the predictions determine the loss.

Parameter vectors

For a model with two parameters,

$$\boldsymbol{\phi} = \begin{bmatrix} \phi_0 \\ \\ \phi_1 \end{bmatrix},$$

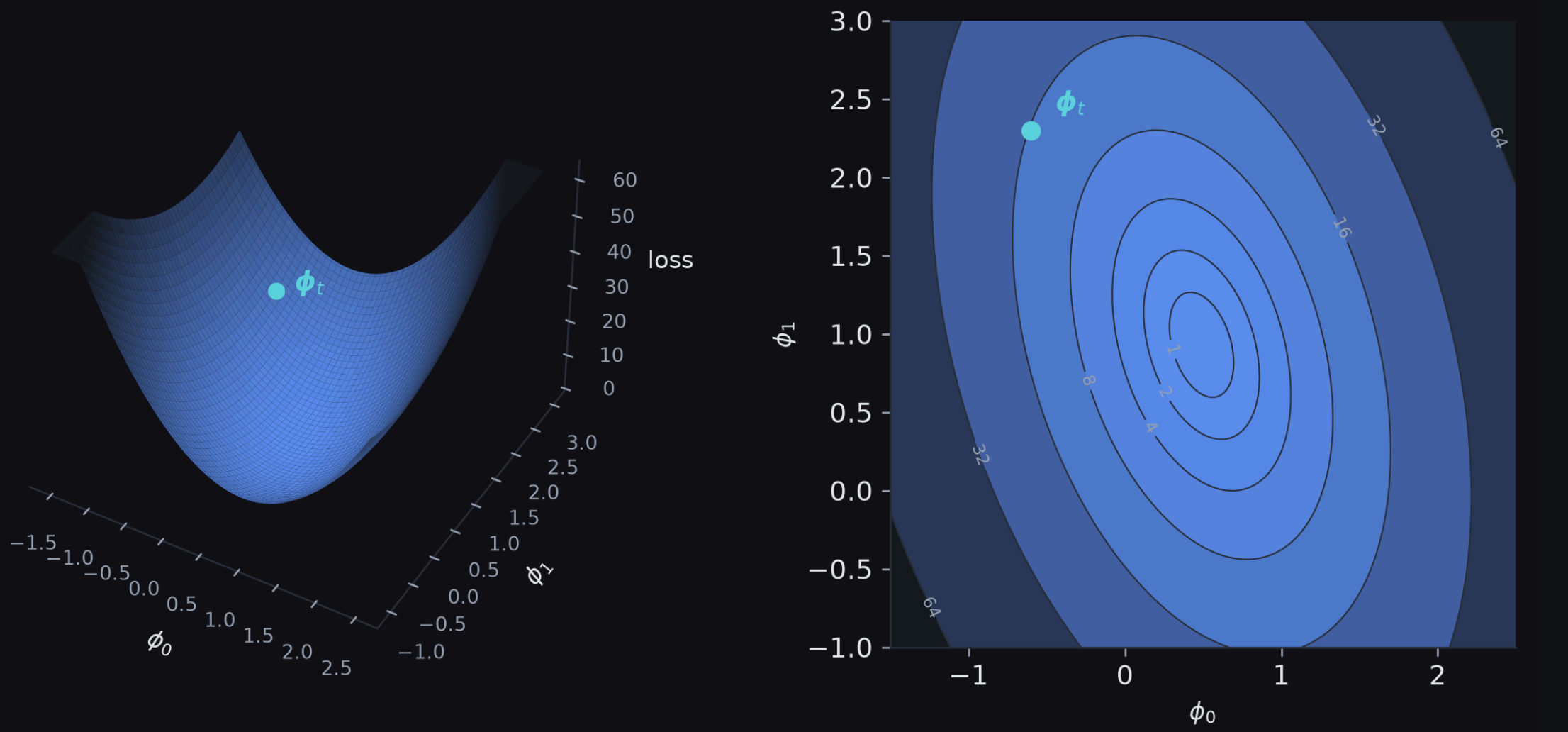
one parameter setting is one point in a two-dimensional **parameter space**.

The loss assigns a height to every point:

$$(\phi_0, \phi_1) \longmapsto \mathcal{L}[\phi_0, \phi_1].$$

Training can now be viewed geometrically: **move through parameter space toward lower loss**.

Parameter space and the loss surface



Each parameter setting is a point in parameter space; the loss assigns a height to that point.

Calculus for optimisation

Average slope

Suppose we move from z_0 to $z_0 + \Delta z$.

The function changes by

$$\Delta g = g(z_0 + \Delta z) - g(z_0).$$

Over that interval, the average slope is

$$\frac{\Delta g}{\Delta z} = \frac{g(z_0 + \Delta z) - g(z_0)}{\Delta z}.$$

As the interval shrinks, the secant slope approaches the **local slope** at z_0.

Local slope

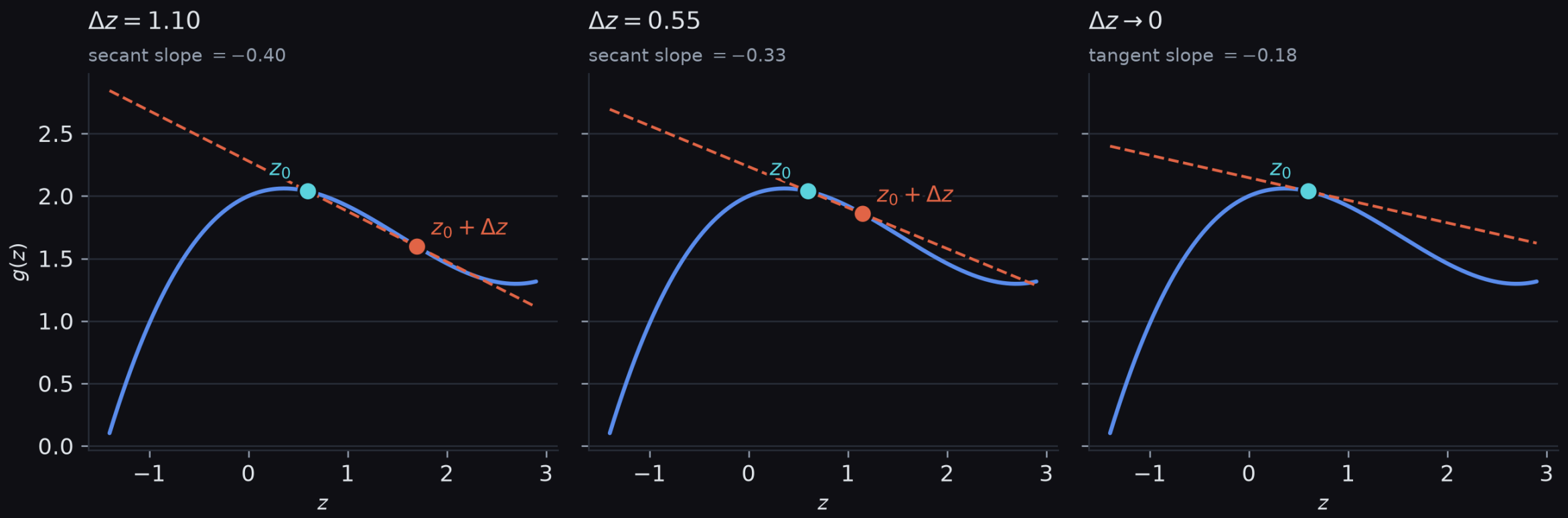
For a scalar function $g(z)$,

$$\left.\frac{dg}{dz}\right|_{z_0} = \lim_{\Delta z \to 0} \frac{g(z_0 + \Delta z) - g(z_0)}{\Delta z}.$$

The limit says: measure the slope over smaller and smaller intervals around z_0.

For this lecture, the important result is the value of that local slope — not the formal machinery of limits.

Secant to tangent



As the second point moves toward z_0, the secant slope approaches the tangent slope: the derivative at z_0.

First-order prediction

For a sufficiently small step Δz,

$$\frac{\Delta g}{\Delta z} \approx \left.\frac{dg}{dz}\right|_{z_0}.$$

Multiply by Δz:

$$\underbrace{\Delta g}_{\text{predicted change}} \approx \underbrace{\left.\frac{dg}{dz}\right|_{z_0}}_{\text{change per unit}} \underbrace{\Delta z}_{\text{proposed step}}.$$

Using the local slope

For example, if the local slope is 3 and $\Delta z = 0.1$,

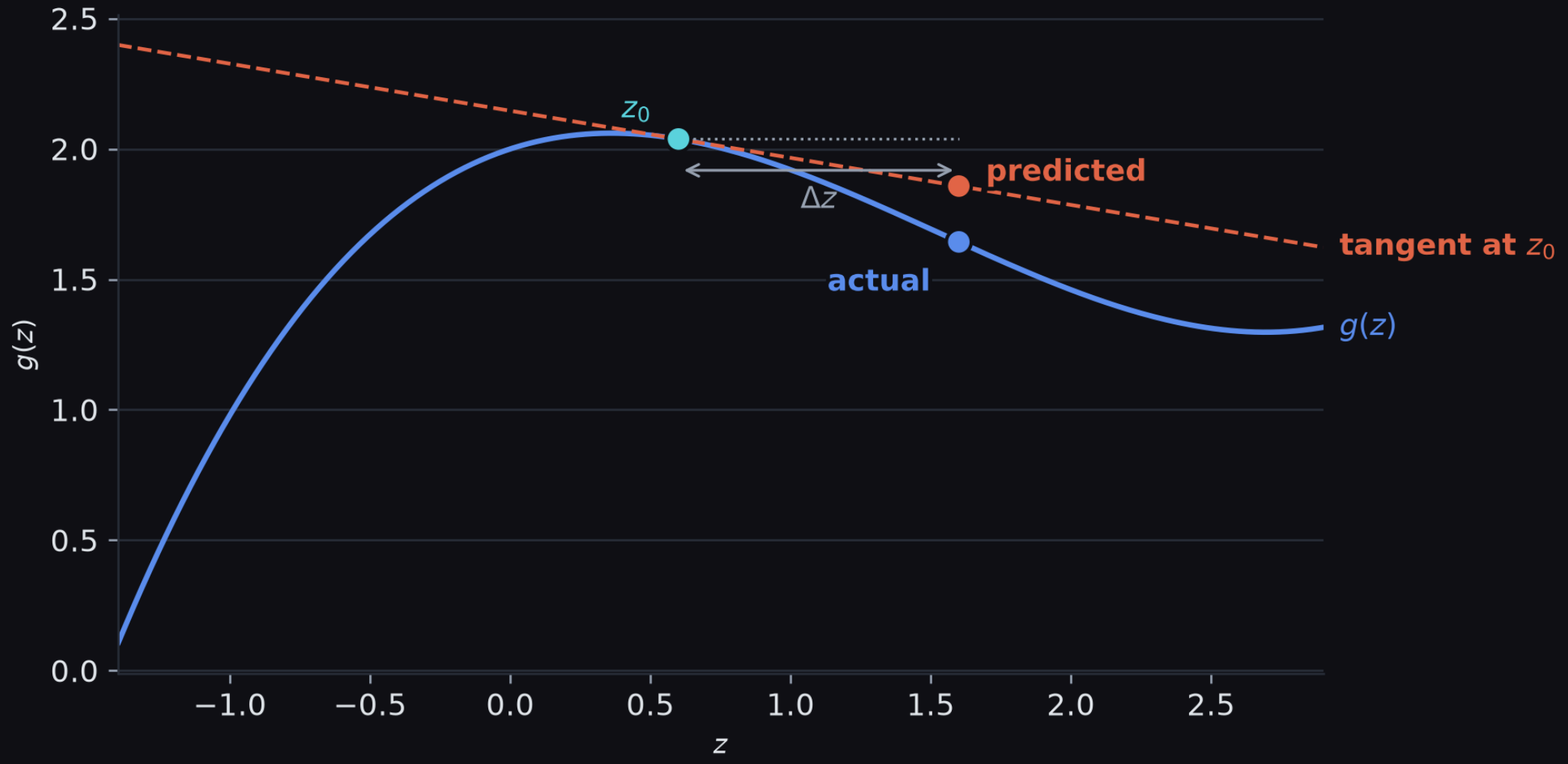
$$\Delta g \approx 3 \times 0.1 = 0.3.$$

Since $\Delta g = g(z_0 + \Delta z) - g(z_0)$,

$$\boxed{g(z_0 + \Delta z) \approx g(z_0) + \left.\frac{dg}{dz}\right|_{z_0} \Delta z}$$

The derivative gives the **local slope** and, for a small step, predicts the change.

Tangent-line approximation



Near z_0, the tangent line uses the derivative to predict the effect of a small input change Δz.

Many parameters

A neural network does not have one parameter.

It may have millions or billions:

$$\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^\top.$$

With many parameters, ask the derivative question **one coordinate at a time**:

> What happens to the loss if I nudge this parameter while temporarily holding the others fixed?

This is a **partial derivative**.

Partial derivatives

For a loss $\mathcal{L}[\phi_0, \phi_1]$,

$$\frac{\partial \mathcal{L}}{\partial \phi_0}$$

means:

- vary ϕ_0 slightly
- hold ϕ_1 fixed
- measure the local rate of change of the loss

Likewise,

$$\frac{\partial \mathcal{L}}{\partial \phi_1}$$

asks the same question along the other coordinate direction.

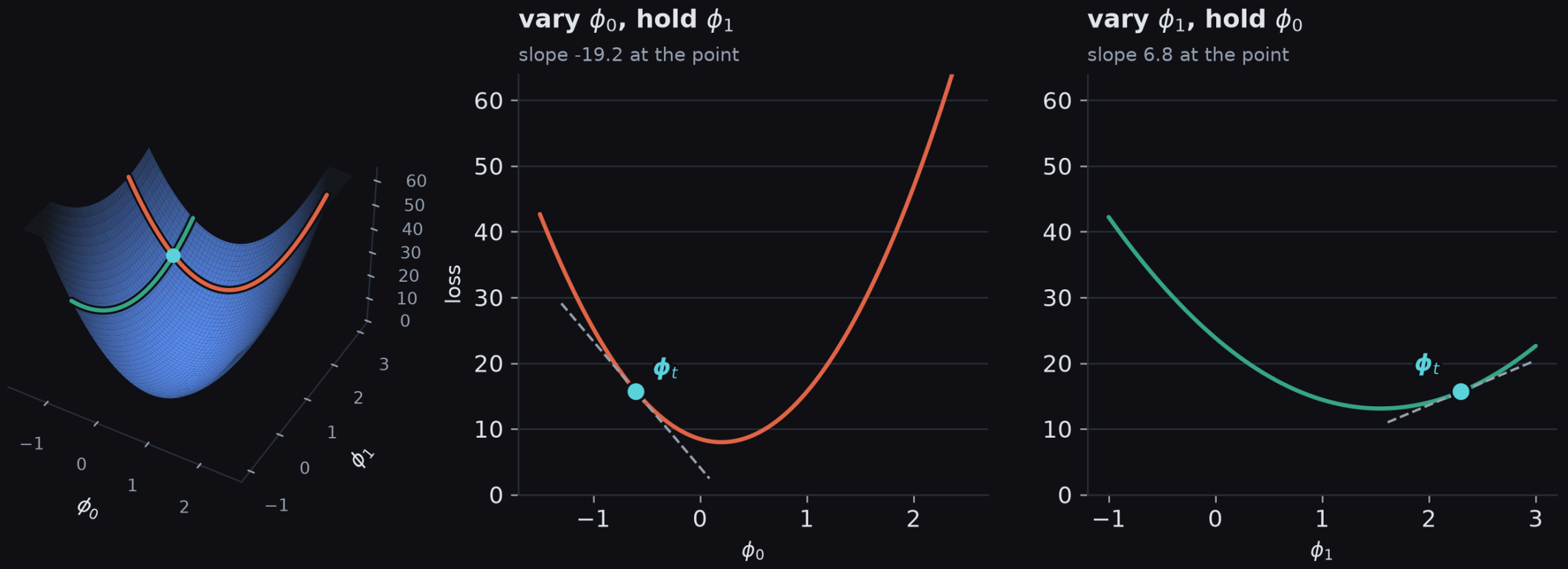
Partial-derivative notation

Notation:

- d is used above for a function with one independent variable
- ∂ reminds us that $\mathcal{L}$ depends on **several** parameters and we are varying only one

Read $\partial\mathcal{L}/\partial\phi_0$ as **“partial L with respect to phi-zero.”**

Partial derivatives as slices



Each partial derivative measures the local slope along one parameter axis while the other parameters are held fixed.

The gradient

One partial derivative tells us about one coordinate.

The **gradient** collects all coordinate sensitivities in one vector:

$$\nabla_{\phi}\mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \phi_0} \\ \frac{\partial \mathcal{L}}{\partial \phi_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \phi_N} \end{bmatrix}$$

∇ is **nabla**.

Read $\nabla_{\phi}\mathcal{L}$ as **“the gradient of the loss with respect to the parameters.”**

Its shape matches ϕ: one sensitivity for every parameter.

First-order change in parameter space

Suppose we change all parameters by a small vector $\Delta\boldsymbol{\phi}$.

The scalar prediction rule becomes

$$\boxed{\mathcal{L}[\boldsymbol{\phi}+\Delta\boldsymbol{\phi}] \approx \mathcal{L}[\boldsymbol{\phi}] + \nabla_{\boldsymbol{\phi}}\mathcal{L}^{\top}\Delta\boldsymbol{\phi}}$$

The dot product adds the predicted contribution from every coordinate:

$$\nabla_{\boldsymbol{\phi}}\mathcal{L}^{\top}\Delta\boldsymbol{\phi} = \frac{\partial\mathcal{L}}{\partial\phi_0}\Delta\phi_0 + \frac{\partial\mathcal{L}}{\partial\phi_1}\Delta\phi_1 + \cdots + \frac{\partial\mathcal{L}}{\partial\phi_N}\Delta\phi_N.$$

Steepest ascent and descent

For parameter moves with the same **Euclidean length**,

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}^{\top} \Delta \boldsymbol{\phi}$$

is largest when $\Delta \boldsymbol{\phi}$ points in the **same direction** as the gradient.

Using the angle form of the dot product from week 1:

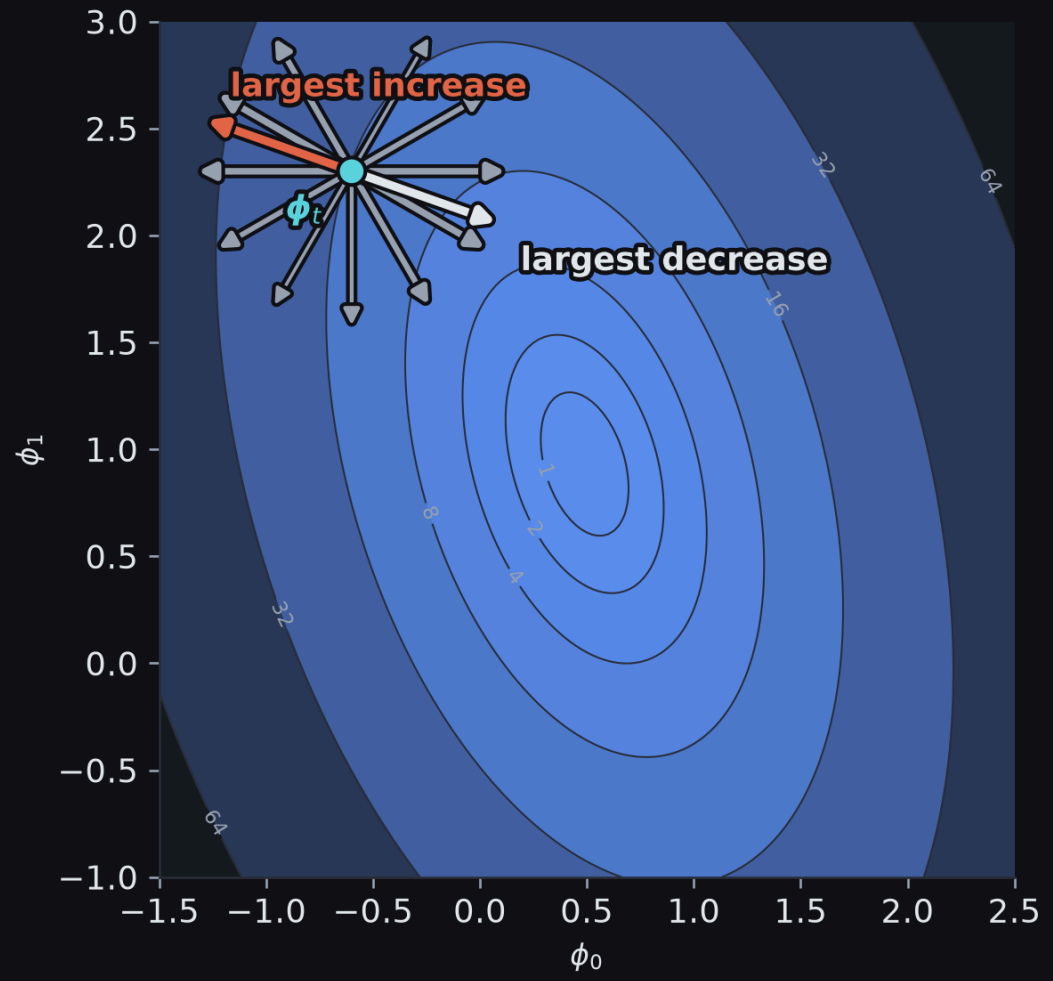
$$\nabla_{\boldsymbol{\phi}} \mathcal{L}^{\top} \Delta \boldsymbol{\phi} = \|\nabla_{\boldsymbol{\phi}} \mathcal{L}\| \; \|\Delta \boldsymbol{\phi}\| \cos\theta.$$

- at the current point, $\|\nabla_{\boldsymbol{\phi}} \mathcal{L}\|$ is fixed
- comparing equal-length moves fixes $\|\Delta \boldsymbol{\phi}\|$
- only $\cos\theta$ varies: it is largest at $\theta = 0$ and smallest at $\theta = \pi$

So, under the ordinary Euclidean notion of step size, and wherever the gradient isn't $\mathbf{0}$:

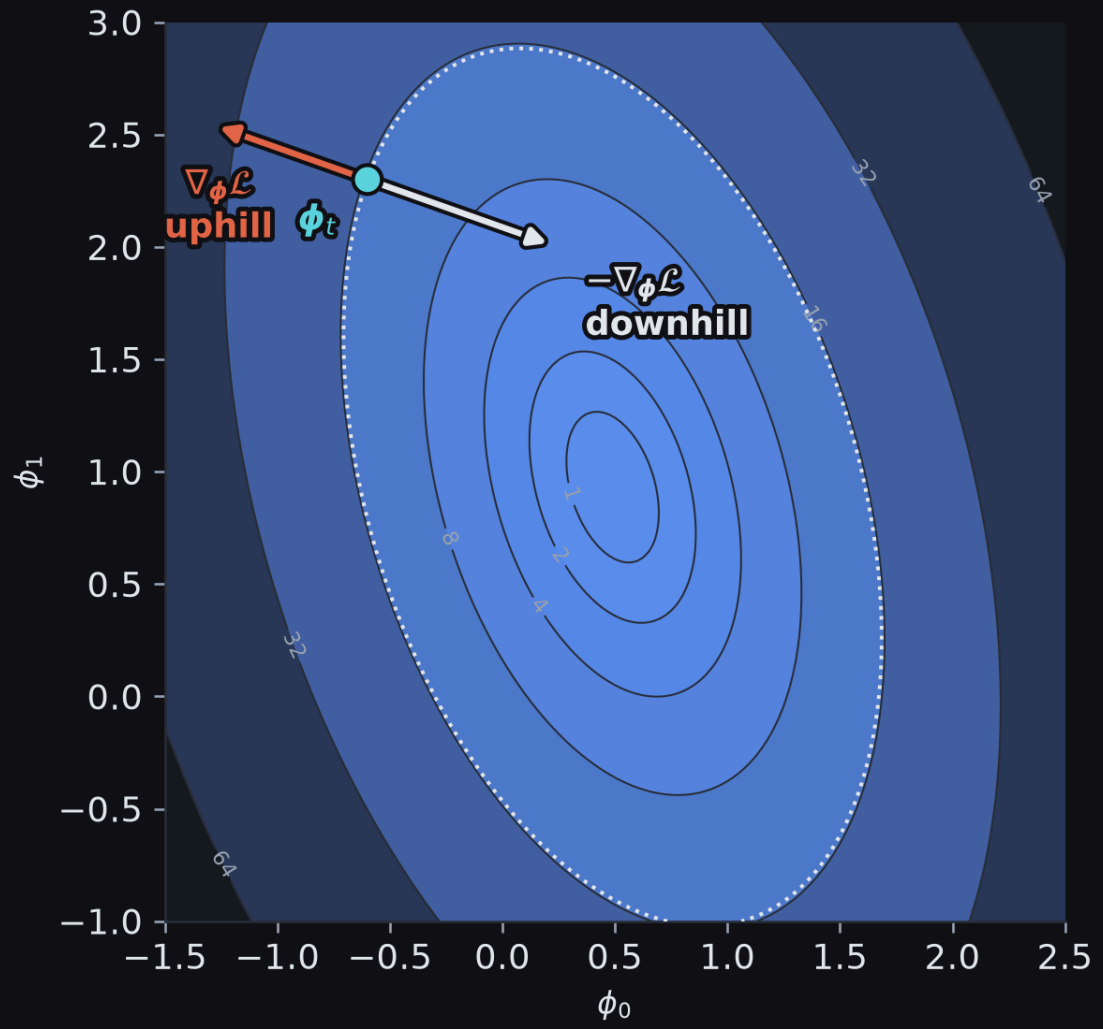
- $\nabla_{\boldsymbol{\phi}} \mathcal{L}$ is the locally steepest uphill direction
- $-\nabla_{\boldsymbol{\phi}} \mathcal{L}$ is the locally steepest downhill direction

Equal-length moves



Among equal-length parameter moves, the move aligned with the gradient gives the largest predicted increase in loss; the opposite move gives the largest predicted decrease.

The gradient on a contour map



On a contour map, $\nabla_\phi \mathcal{L}$ points in the direction of steepest local increase; $-\nabla_\phi \mathcal{L}$ gives the steepest local decrease.

Reading the contour map

- **The dotted curve** is the contour through ϕ_t: every point on it has the same loss
- **The gradient is perpendicular to the contour locally**, giving the steepest first-order increase
- **The minus sign** reverses that direction to give the steepest first-order decrease

First-order optimisation

Gradient descent

Gradient descent uses this direction directly:

$$\underbrace{\boldsymbol{\phi}_{t+1}}_{\text{new parameters}} = \underbrace{\boldsymbol{\phi}_{t}}_{\text{current parameters}} - \underbrace{\alpha}_{\text{step size}} \underbrace{\nabla_{\boldsymbol{\phi}}\mathcal{L}[\boldsymbol{\phi}_{t}]}_{\text{uphill direction}}.$$

- the gradient gives the local uphill direction
- the minus sign reverses it
- $\alpha > 0$ sets the step size
- the calculation is repeated at the new parameter values

Learning rate

The learning rate α sets the size of each update.

Too small:

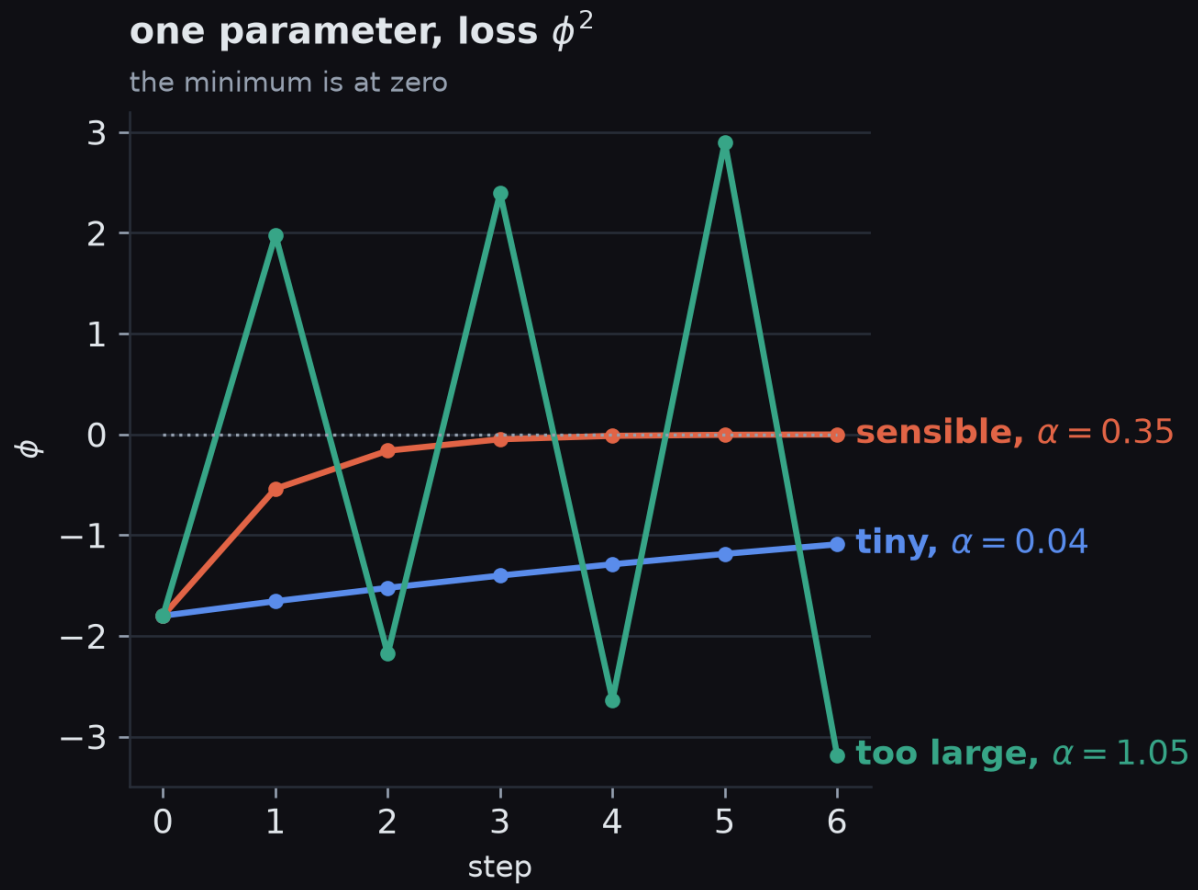
- stable updates
- slow progress

Too large:

- faster movement
- risk of overshoot, oscillation, or divergence

A classical way to choose the step length is a **line search**: first choose a downhill direction, then search along that line for a useful value of α.

Step direction and step length





The gradient chooses a descent direction; the learning rate or line search determines how far to move along it.

Differentiating a model

Worked example: linear regression

Start with the familiar model

$$\hat{y}_i = \mathrm{f}[x_i, \boldsymbol{\phi}] = \phi_0 + \phi_1 x_i.$$

For one training example (x_i, y_i), define the residual

$$r_i = \underbrace{\phi_0 + \phi_1 x_i}_{\text{prediction}} - \underbrace{y_i}_{\text{target}}.$$

Week 1 wrote the residual as target minus prediction. This lecture follows Prince's order, so the **residual sign flips** but the squared loss is unchanged.

With squared error,

$$\ell_i = r_i^2.$$

Numerical check

Take one example with

$$x_i = 2, \qquad y_i = 5, \qquad \phi_0 = 1, \qquad \phi_1 = 1.$$

The model predicts 3, so

$$r_i = 3 - 5 = -2, \qquad \ell_i = (-2)^2 = 4.$$

Now nudge only the intercept by $\Delta\phi_0 = 0.1$. The prediction rises from 3 to 3.1, so the residual becomes less negative and the squared error should **fall**.

The derivative predicts the size of that local change.

Chain-rule graph



multiply the edge sensitivities along the path

The chain rule multiplies local sensitivities along a dependency path from a parameter to the loss.

The chain rule

Learning problems are built from dependent calculations:

$$\phi_0 \longrightarrow r_i \longrightarrow \ell_i.$$

The chain rule multiplies the **local sensitivities** along that path:

$$\frac{\partial \ell_i}{\partial \phi_0} = \underbrace{\frac{\partial \ell_i}{\partial r_i}}_{\text{loss sensitivity}} \underbrace{\frac{\partial r_i}{\partial \phi_0}}_{\text{model sensitivity}}$$

More generally, if changing a changes b, and changing b changes c,

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}.$$

A deep network uses the same rule repeatedly through a much larger dependency graph.

Derivative with respect to ϕ_0

We want

$$\frac{\partial \ell_i}{\partial \phi_0}.$$

Start from

$$\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2.$$

Use the chain rule:

$$\frac{\partial \ell_i}{\partial \phi_0} = \underbrace{2r_i}_{\text{how loss changes with residual}} \underbrace{\frac{\partial r_i}{\partial \phi_0}}_{=1} = 2r_i.$$

The intercept derivative is proportional to the signed residual.

Derivative with respect to ϕ_1

For the slope parameter:

$$\frac{\partial \ell_i}{\partial \phi_1} = \underbrace{2r_i}_{\text{error signal}} \underbrace{\frac{\partial r_i}{\partial \phi_1}}_{=x_i} = 2x_i r_i.$$

Why does x_i appear?

Because changing the slope affects the prediction by an amount proportional to the input:

$$\frac{\partial}{\partial \phi_1}(\phi_1 x_i) = x_i.$$

Checking the derivative prediction

The example from earlier: $x_i = 2$, $r_i = -2$, and a nudge of $\Delta\phi_0 = 0.1$.

- **Sensitivity**: $\partial \ell_i / \partial \phi_0 = 2 r_i = -4$
- **Predicted change**: $-4 \times 0.1 = -0.4$
- **Actual change**: $(-1.9)^2 - (-2)^2 = -0.39$

The derivative predicts -0.4; the actual change is -0.39.

Their difference is second order: halve the step and it quarters.

Per-example gradient

Stack the two partial derivatives to form the per-example gradient:

$$\nabla_{\phi} \ell_i = \begin{bmatrix} 2r_i \\ \\ 2x_i r_i \end{bmatrix} = 2r_i \begin{bmatrix} 1 \\ \\ x_i \end{bmatrix}.$$

The two factors have different roles:

- $2r_i$: **how wrong** the model is
- $[1, x_i]^\top$: **how this example couples to the parameters**

Dataset gradient

If

$$\mathcal{L}[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i,$$

then differentiation distributes across the sum:

$$\nabla_{\boldsymbol{\phi}} \mathcal{L} = \sum_{i=1}^{I} \nabla_{\boldsymbol{\phi}} \ell_i.$$

The full gradient is the sum of the per-example gradients.

Sum or mean?

We use a **sum** here because it follows directly from the per-example derivation. Many libraries report the **mean** loss instead:

$$\bar{\mathcal{L}} = \frac{1}{I}\sum_{i=1}^{I} \ell_i.$$

The minimiser is unchanged; the gradient is simply scaled by $1/I$. That constant scale can be accounted for in the learning rate.

Matrix calculus

Writing one derivative per parameter is manageable for two parameters.

With millions of parameters, coordinate-by-coordinate notation is no longer practical.

Matrix calculus expresses the same derivative relationships compactly for **vectors and matrices**, while keeping the structure of the computation visible.

The design matrix

x_i	y_i
-0.84	-0.27
-0.65	-0.05
-0.25	-0.05
$\vdots$	
0.95	1.43

examples

stack

$$\mathbf{X} = \begin{bmatrix} 1 & -0.84 \\ 1 & -0.65 \\ 1 & -0.25 \\ \vdots & \\ 1 & 0.95 \end{bmatrix}$$

design matrix

multiply

$$\mathbf{X}\hat{\phi} = \begin{bmatrix} -0.29 \\ -0.12 \\ 0.26 \\ \vdots \\ 1.38 \end{bmatrix} = \hat{\mathbf{y}}$$

predictions

The design matrix collects all examples so that the complete prediction vector is computed as $\mathbf{X}\phi$.

Linear regression in vector form

For all I examples, create the design matrix

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_I \end{bmatrix}, \qquad \boldsymbol{\phi} = \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix}.$$

Stacking the examples gives all predictions in one matrix multiplication:

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\phi}.$$

The residual vector

The residual vector is

$$\mathbf{r} = \mathbf{X}\boldsymbol{\phi} - \mathbf{y}.$$

- **Same order as the scalar above**: prediction minus target, so this is the negative of week 1's residual vector
- the squared loss is unchanged
- the **parameter gradient is unchanged** when the convention is used consistently: both $\mathbf{r}$ and $\partial \mathbf{r} / \partial \boldsymbol{\phi}$ change sign

The loss in vector form

The sum of squared residuals is

$$\mathcal{L}[\boldsymbol{\phi}] = \sum_{i=1}^{I} r_i^2 = \mathbf{r}^\top \mathbf{r}.$$

Substituting $\mathbf{r} = \mathbf{X}\boldsymbol{\phi} - \mathbf{y}$:

$$\mathcal{L}[\boldsymbol{\phi}] = (\mathbf{X}\boldsymbol{\phi} - \mathbf{y})^\top (\mathbf{X}\boldsymbol{\phi} - \mathbf{y}).$$

The vector expression is identical to the scalar least-squares loss.

Matrix gradient I: residual sensitivity

We already have

$$\mathcal{L} = \mathbf{r}^\top \mathbf{r}.$$

So the loss sensitivity with respect to the residual vector is

$$\nabla_{\mathbf{r}} \mathcal{L} = 2\mathbf{r}.$$

This is the vector version of $d(r^2)/dr = 2r$.

And because

$$\mathbf{r} = \mathbf{X}\boldsymbol{\phi} - \mathbf{y},$$

a small parameter move produces

$$\Delta\mathbf{r} = \mathbf{X}\Delta\boldsymbol{\phi}.$$

Matrix gradient II: parameter sensitivity

The first-order change in loss is

$$\Delta\mathcal{L} \approx 2\mathbf{r}^\top \Delta\mathbf{r} = (2\mathbf{r})^\top \mathbf{X}\Delta\boldsymbol{\phi}.$$

Rearrange the scalar product:

$$\Delta\mathcal{L} \approx \left(2\mathbf{X}^\top \mathbf{r}\right)^\top \Delta\boldsymbol{\phi}.$$

Compare this with

$$\Delta\mathcal{L} \approx (\nabla_{\boldsymbol{\phi}}\mathcal{L})^\top \Delta\boldsymbol{\phi}.$$

Both expressions must agree for **every** small $\Delta\boldsymbol{\phi}$, so their coefficient vectors are equal:

$$\boxed{\nabla_{\boldsymbol{\phi}}\mathcal{L} = 2\mathbf{X}^\top \mathbf{r} = 2\mathbf{X}^\top(\mathbf{X}\boldsymbol{\phi} - \mathbf{y})}.$$

Interpreting the matrix gradient

1. $\mathbf{X}\boldsymbol{\phi} - \mathbf{y}$: compute the **residuals**.
2. $2(\cdot)$: convert each residual into a **loss sensitivity**.
3. $\mathbf{X}^\top(\cdot)$: map those sensitivities back onto the **parameter coordinates**.

This is the same chain rule as the one-example derivation, written for the whole dataset.

The Jacobian

Suppose a vector-valued function produces

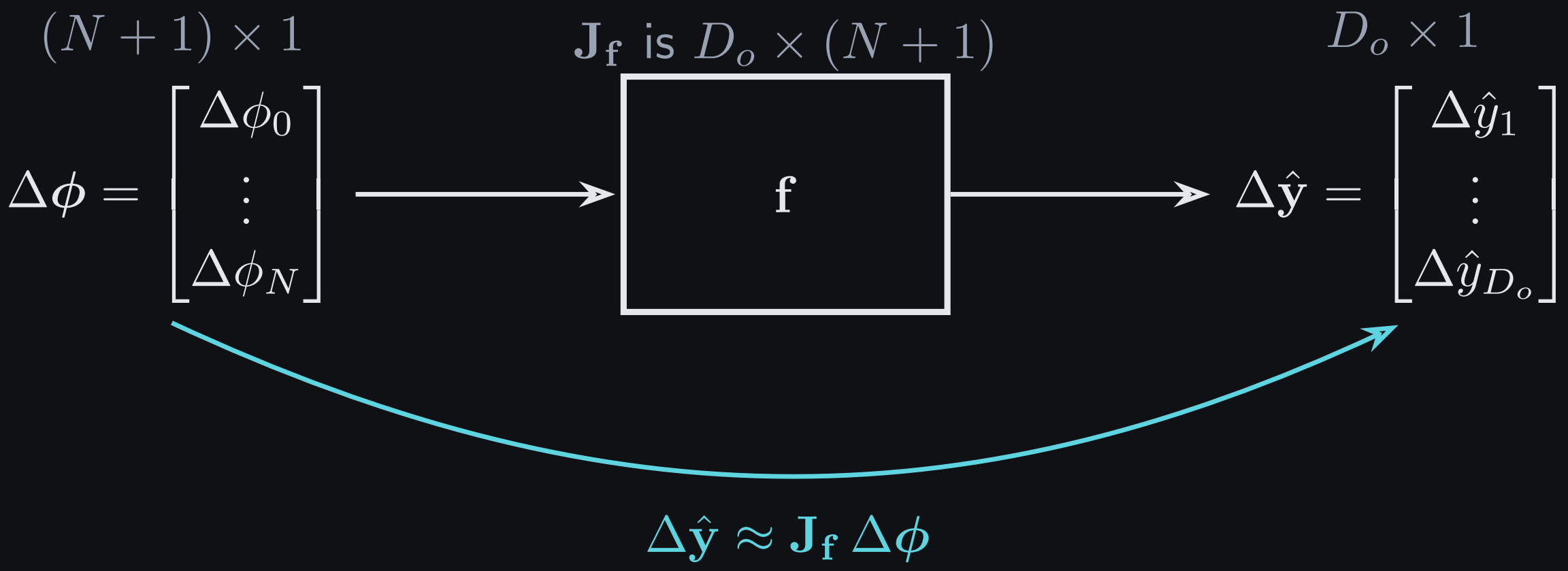
$$\mathbf{z} = \mathbf{f}(\boldsymbol{\phi}) \in \mathbb{R}^{D_o}, \qquad \boldsymbol{\phi} \in \mathbb{R}^{N+1}.$$

Its **Jacobian** collects every first-order sensitivity:

$$\mathbf{J_f} = \frac{\partial \mathbf{z}}{\partial \boldsymbol{\phi}^\top} = \begin{bmatrix} \frac{\partial z_1}{\partial \phi_0} & \cdots & \frac{\partial z_1}{\partial \phi_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_{D_o}}{\partial \phi_0} & \cdots & \frac{\partial z_{D_o}}{\partial \phi_N} \end{bmatrix}.$$

Our convention is: **rows correspond to outputs; columns correspond to parameters**. With this convention, $\mathbf{J_f}$ has shape $D_o \times (N+1)$.

The Jacobian as a local linear map



The Jacobian maps a small parameter perturbation to the corresponding first-order change in the model outputs.

The Jacobian in the chain rule

A loss often has the structure

$$\boldsymbol{\phi} \longrightarrow \hat{\mathbf{y}} \longrightarrow \mathcal{L}.$$

The multivariable chain rule says

$$\nabla_{\boldsymbol{\phi}} \mathcal{L} = \mathbf{J}_{\mathbf{f}}^{\top} \nabla_{\hat{\mathbf{y}}} \mathcal{L}.$$

- $\nabla_{\hat{\mathbf{y}}} \mathcal{L}$ gives the sensitivity of the scalar loss to each output
- $\mathbf{J}_{\mathbf{f}}^{\top}$ maps those output sensitivities back to the parameters

Check the dimensions:

$$\underbrace{\mathbf{J}_{\mathbf{f}}^{\top}}_{(N+1)\times D_o} \underbrace{\nabla_{\hat{\mathbf{y}}} \mathcal{L}}_{D_o \times 1} = \underbrace{\nabla_{\boldsymbol{\phi}} \mathcal{L}}_{(N+1)\times 1}.$$

The same chain-rule pattern propagates loss sensitivities through a deep network.

Curvature and loss geometry

The convex case

Linear least squares has a convex loss landscape.

For a convex function, every local minimum is also global.

Geometrically, every chord joining two points on the graph lies on or above the function.

Convexity removes **suboptimal local minima** from the landscape.

Gradient descent still needs a suitable step-size strategy to converge.

Second derivatives

For a scalar function $g(z)$,

$$g'(z) = \frac{dg}{dz}$$

gives the local slope.

Differentiate the slope itself:

$$g''(z) = \frac{d^2 g}{dz^2}.$$

The second derivative measures how the slope changes as we move.

Locally:

- $g''(z) > 0$: the graph curves upward
- $g''(z) < 0$: the graph curves downward
- $g''(z) = 0$: second-order information alone may be inconclusive

The Hessian

For a scalar loss with many parameters, the gradient is itself a vector-valued function:

$$\boldsymbol{\phi} \longmapsto \nabla_{\boldsymbol{\phi}} \mathcal{L}.$$

Differentiate that vector with respect to the parameters:

$$\boxed{\mathbf{H} = \frac{\partial (\nabla_{\boldsymbol{\phi}} \mathcal{L})}{\partial \boldsymbol{\phi}^{\top}} = \nabla^2_{\boldsymbol{\phi}} \mathcal{L}}$$

This matrix is the **Hessian**.

Hessian entries

Expanded,

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 \mathcal{L}}{\partial \phi_0^2} & \dfrac{\partial^2 \mathcal{L}}{\partial \phi_0 \partial \phi_1} & \cdots \\ \dfrac{\partial^2 \mathcal{L}}{\partial \phi_1 \partial \phi_0} & \dfrac{\partial^2 \mathcal{L}}{\partial \phi_1^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

- diagonal terms: curvature along individual coordinates
- off-diagonal terms: how changing one parameter changes the slope with respect to another

To interpret the Hessian geometrically, recall how a matrix acts on directions.

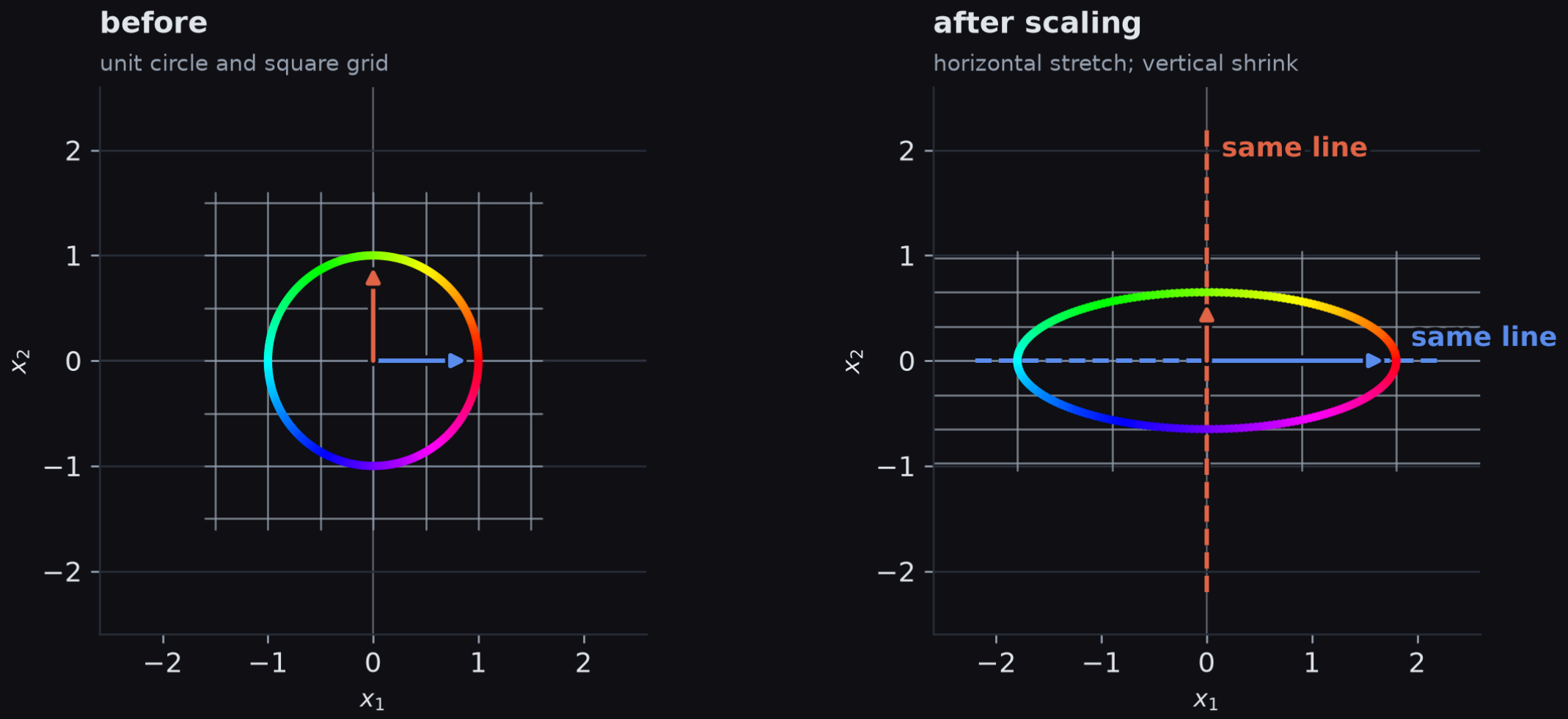
Scaling

Recall that matrix multiplication can be read geometrically as a **linear transformation**.

Here the matrix stretches one coordinate and shrinks the other:

$$\mathbf{A} = \begin{bmatrix} 1.8 & 0 \\ 0 & 0.65 \end{bmatrix}.$$

Scaling the plane



Under non-uniform scaling, the grid and unit circle stretch into a rectangle-like grid and an ellipse. The hue tracks each point from the original circle to its transformed position.

Invariant lines under scaling

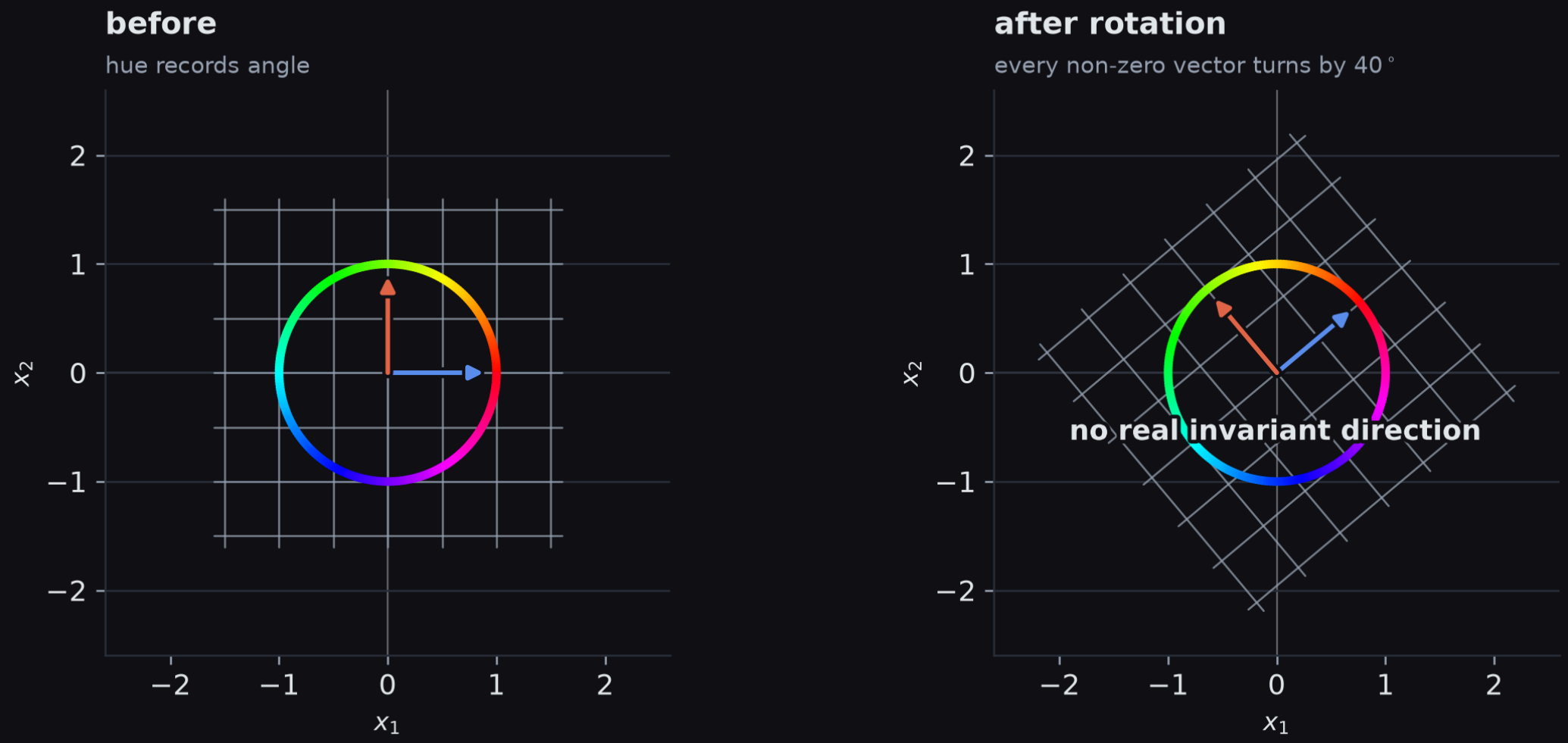
For this scaling matrix, horizontal and vertical vectors change length but remain on their original lines.

Rotation

Now rotate every vector by 40°:

$$\mathbf{R} = \begin{bmatrix} \cos 40^\circ & -\sin 40^\circ \\ \sin 40^\circ & \cos 40^\circ \end{bmatrix}.$$

Rotating the plane



A rotation preserves the shape of the unit circle, so hue is important: every coloured point moves around the circle. No non-zero real vector remains on its original line for this angle.

No real invariant direction

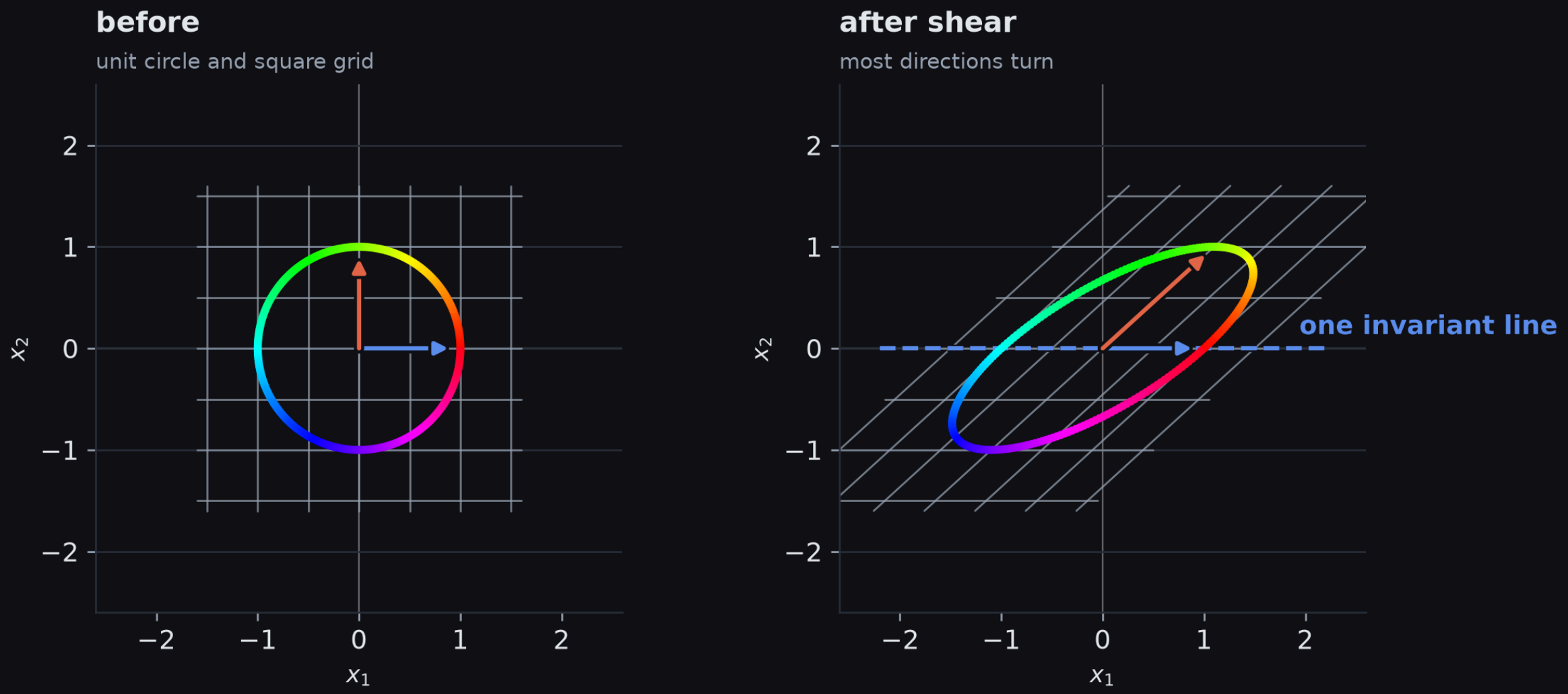
The circle still looks like a circle, but the hue shows that **every non-zero real vector has moved to a different line**.

Shear

A shear mixes the coordinates:

$$\mathbf{S} = \begin{bmatrix} 1 & 1.1 \\ 0 & 1 \end{bmatrix}.$$

Shearing the plane



A shear tilts the grid and turns most vectors, but the horizontal axis remains on the same line.

One invariant line

Most directions turn, but vectors on the horizontal axis remain on that same line.

A real 2×2 matrix therefore need not have two distinct real invariant directions.

Eigenvectors and eigenvalues

Those invariant directions are **eigenvector directions**.

A non-zero vector $\mathbf{v}$ is an **eigenvector** of $\mathbf{A}$ when

$$\boxed{\mathbf{A}\mathbf{v} = \lambda\mathbf{v}}.$$

Geometrically:

- $\mathbf{A}\mathbf{v}$ is a scalar multiple of $\mathbf{v}$
- for $\lambda \neq 0$, the transformed vector stays on the **same line through the origin**
- $|\lambda|$ gives the scale factor; the sign gives the orientation
- any non-zero multiple of an eigenvector lies on the same **eigendirection**

Interpreting eigenvalues

The sign and magnitude of λ tell us what happened:

- $\lambda > 1$: stretch
- $0 < \lambda < 1$: shrink
- $\lambda < 0$: scale and reverse orientation
- $\lambda = 0$: collapse that direction to the origin

For the scaling matrix above,

$$\mathbf{A}\begin{bmatrix}1\\0\end{bmatrix} = 1.8\begin{bmatrix}1\\0\end{bmatrix}, \qquad \mathbf{A}\begin{bmatrix}0\\1\end{bmatrix} = 0.65\begin{bmatrix}0\\1\end{bmatrix}.$$

The coordinate axes are eigenvector directions; 1.8 and 0.65 are their eigenvalues.

Eigenvalues as polynomial roots

To find an eigenvector, we need a non-zero solution of

$$(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = \mathbf{0}.$$

A non-zero solution exists only when $\mathbf{A} - \lambda \mathbf{I}$ is **singular**: it must map some non-zero vector to zero.

For a square matrix, singular means determinant zero:

$$\boxed{\det(\mathbf{A} - \lambda \mathbf{I}) = 0}.$$

This **characteristic equation** is a polynomial in λ.

The eigenvalues are the **roots of this polynomial**.

As with ordinary polynomial roots, they need not all be real.

Rotation and complex eigenvalues

For a 90° rotation,

$$\mathbf{R}_{90} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \qquad \det(\mathbf{R}_{90} - \lambda \mathbf{I}) = \lambda^2 + 1.$$

Its roots are $\lambda = \pm i$, so there is **no real eigendirection**. That matches the geometry: every real vector turns.

Repeated eigenvalues

A repeated root need not provide the same number of independent eigenvectors.

For the shear matrix

$$\mathbf{S} = \begin{bmatrix} 1 & 1.1 \\ 0 & 1 \end{bmatrix},$$

we get

$$\det(\mathbf{S} - \lambda \mathbf{I}) = (1 - \lambda)^2.$$

So $\lambda = 1$ is a **repeated eigenvalue**, but the horizontal axis supplies only **one independent real eigendirection**.

Symmetric matrices

Symmetric Hessians

For the sufficiently smooth losses considered here, the mixed second partials agree, so the Hessian is **symmetric**.

Real symmetric matrices have:

- real eigenvalues
- a full set of mutually orthogonal eigenvector directions

For a Hessian, the eigenvectors therefore give a complete set of real principal directions.

Local change in the gradient

For a small parameter move $\Delta\boldsymbol{\phi}$, the Hessian gives the first-order change in the **gradient**:

$$\boxed{\nabla_{\boldsymbol{\phi}}\mathcal{L}[\boldsymbol{\phi}+\Delta\boldsymbol{\phi}] \approx \nabla_{\boldsymbol{\phi}}\mathcal{L}[\boldsymbol{\phi}] + \mathbf{H}\Delta\boldsymbol{\phi}}$$

$\mathbf{H}\Delta\boldsymbol{\phi}$ is the first-order predicted change in the gradient caused by the parameter move $\Delta\boldsymbol{\phi}$.

Second-order loss approximation

Near the current parameters,

$$\mathcal{L}[\boldsymbol{\phi} + \Delta\boldsymbol{\phi}] \approx \mathcal{L}[\boldsymbol{\phi}] + \underbrace{\nabla_{\boldsymbol{\phi}}\mathcal{L}^{\top}\Delta\boldsymbol{\phi}}_{\text{first-order change}} + \frac{1}{2}\underbrace{\Delta\boldsymbol{\phi}^{\top}\mathbf{H}\Delta\boldsymbol{\phi}}_{\text{curvature correction}} .$$

The linear term contains the local slope information; the quadratic term adds the curvature correction.

The least-squares Hessian

Recall

$$\nabla_{\boldsymbol{\phi}} \mathcal{L} = 2\mathbf{X}^\top(\mathbf{X}\boldsymbol{\phi} - \mathbf{y}).$$

Differentiate once more:

$$\mathbf{H} = 2\mathbf{X}^\top\mathbf{X}.$$

The Hessian no longer depends on $\boldsymbol{\phi}$: ordinary least squares has the same curvature everywhere in parameter space.

Because $\mathbf{X}^\top\mathbf{X}$ is positive semidefinite, the least-squares objective is convex.

- $\mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v} = \|\mathbf{X}\mathbf{v}\|^2 \geq 0$ for every $\mathbf{v}$
- full column rank makes the Hessian positive definite and the minimiser unique
- dependent columns create zero-curvature directions and can give multiple optimal minimisers

Isotropy: equal turning

The terminology comes from Greek:

- *isos* (ἴσος): **equal**
- *tropos* (τρόπος): a **turn**, **way**, or **direction**

So **isotropic** is literally the idea of **equal turning**:

> turn in a different direction, and the behaviour stays the same.

Anisotropic adds a negative prefix: the behaviour is **not the same in every direction**.

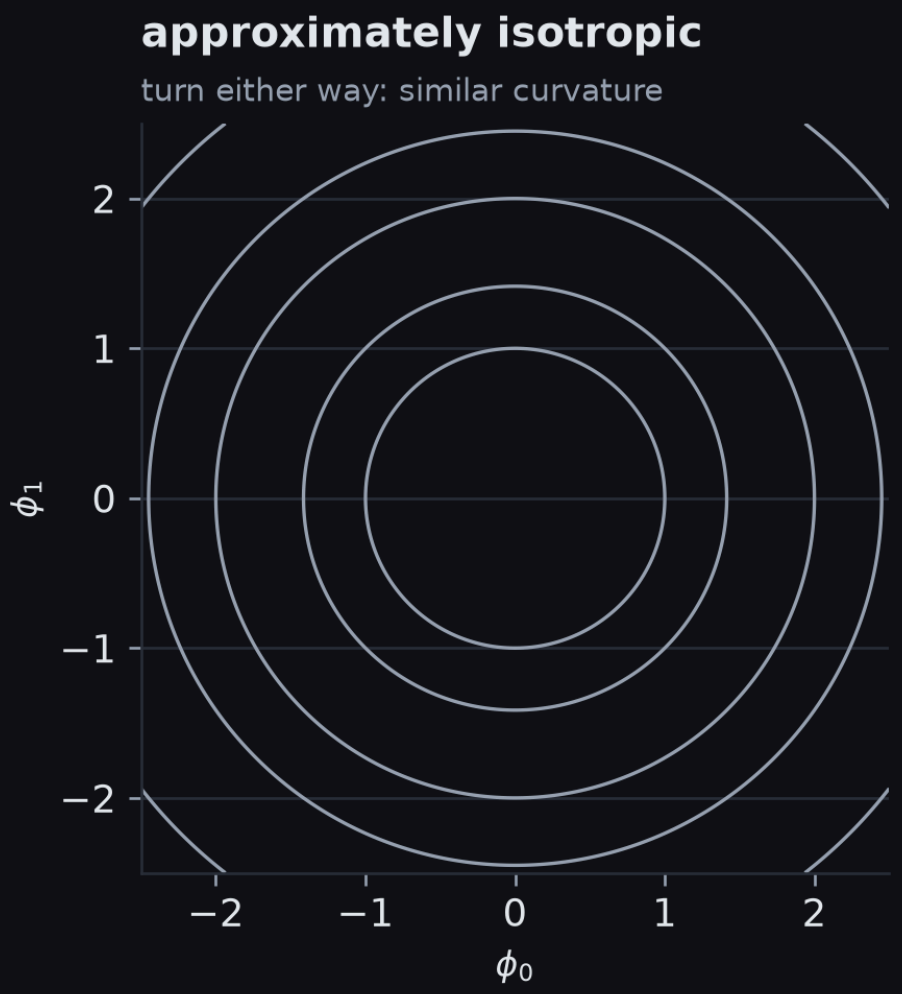
A familiar tropos

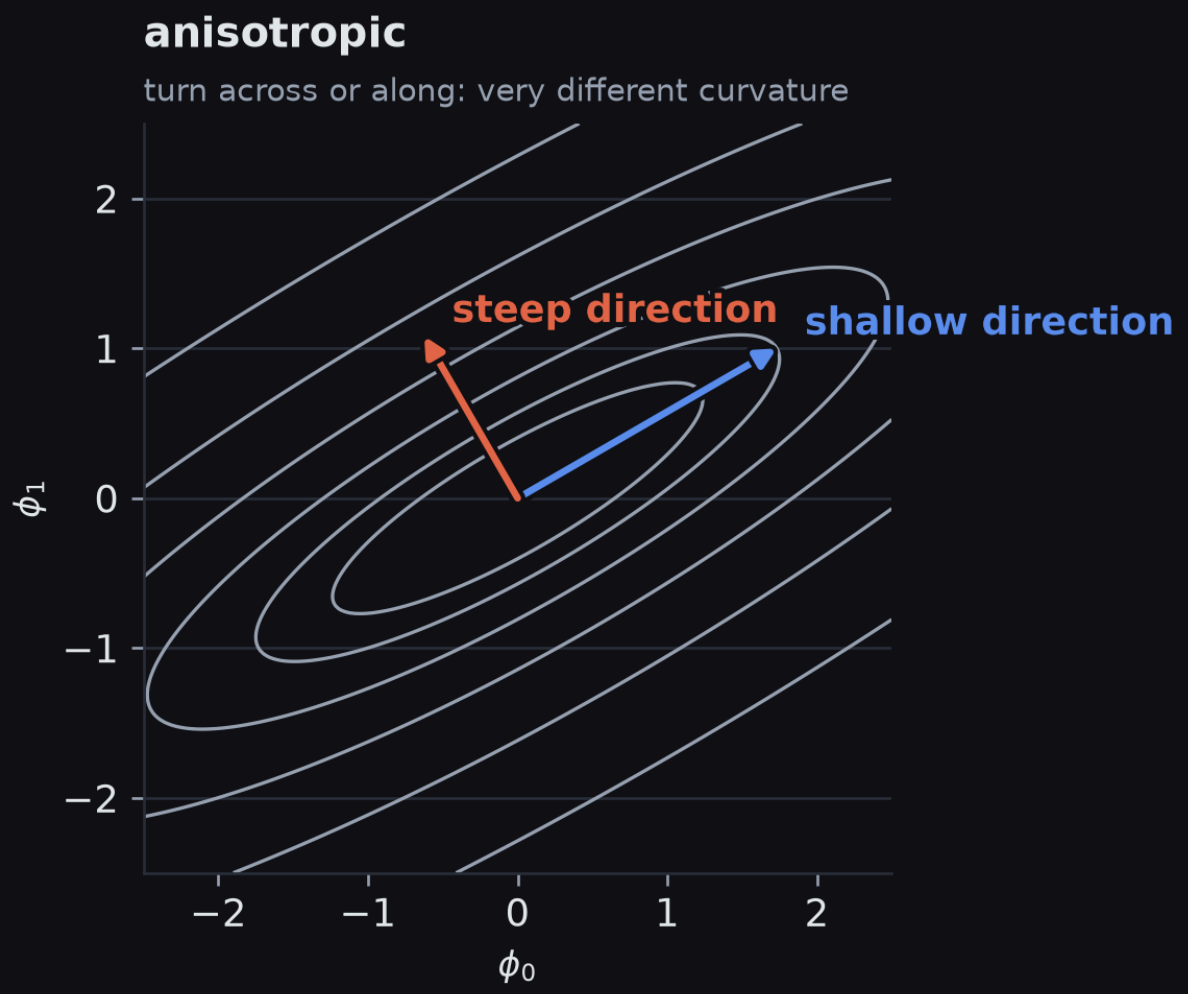
Homer describes Odysseus as *polytropos* (πολύτροπος): *poly*, **many**, + *tropos*, **turns / ways**.

In Homer the epithet has a broader sense — resourceful, wily, "of many turns" — but the shared root is useful here:

polytropos → **many-turning;** ***isotropic*** → **equal-turning.**

Isotropy in parameter space





Isotropic means that turning to a different direction changes little; anisotropic means that curvature depends strongly on the direction.

Isotropy and eigenvalues

For the positive-curvature landscapes shown here:

$$\lambda_1 \approx \lambda_2 \quad \Rightarrow \quad \text{roughly circular contours,}$$

whereas

$$\lambda_{\text{steep}} \gg \lambda_{\text{shallow}} \quad \Rightarrow \quad \text{elongated contours.}$$

Curvature along an eigendirection

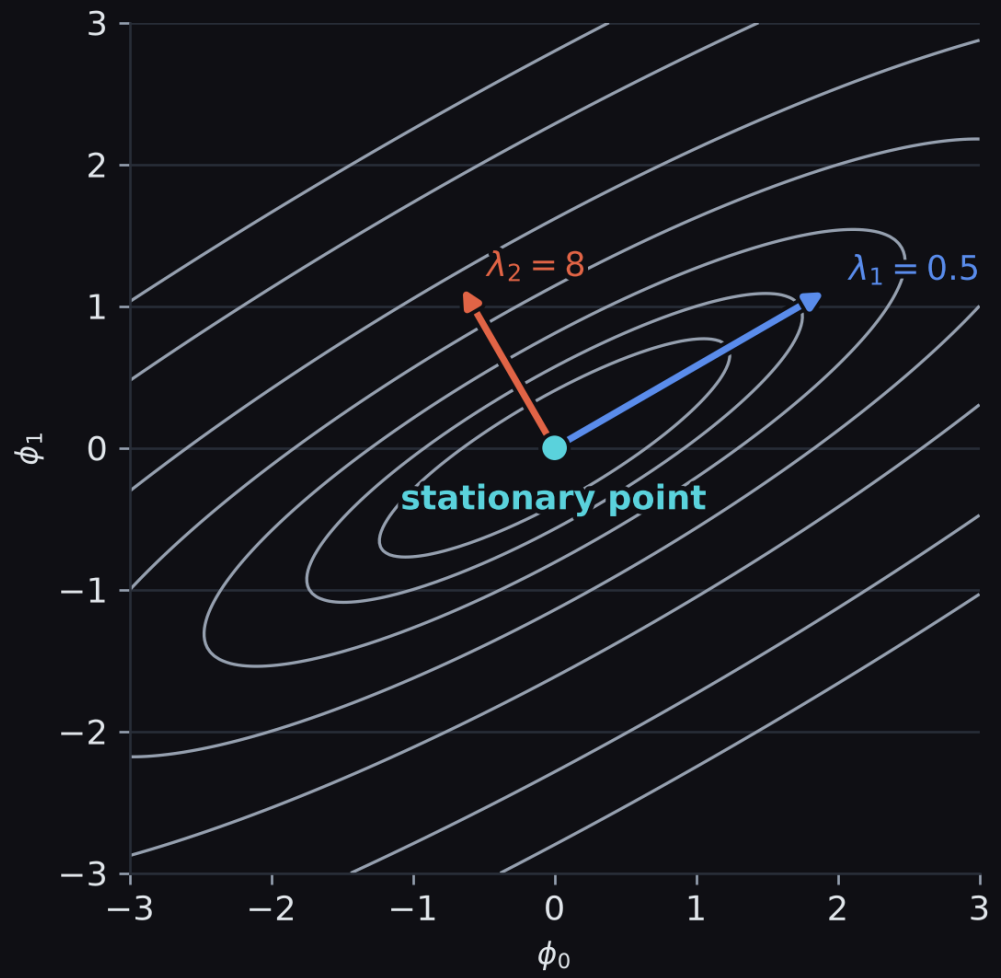
The Hessian eigenvectors identify **principal curvature directions** in parameter space.

Let $\mathbf{v}_k$ be a **unit eigenvector** of the Hessian and move along that direction, $\Delta\boldsymbol{\phi} = c\,\mathbf{v}_k$. Then

$$\Delta\boldsymbol{\phi}^\top \mathbf{H} \Delta\boldsymbol{\phi} = \lambda_k c^2 .$$

- **Large** $|\lambda_k|$: strong curvature along that eigendirection
- **Small** $|\lambda_k|$: weak curvature or a nearly flat direction
- **Sign of** λ_k: positive bends upward; negative bends downward
- in the scaling example, eigenvalues were scale factors; for a Hessian, they measure signed local curvature

Principal curvature directions

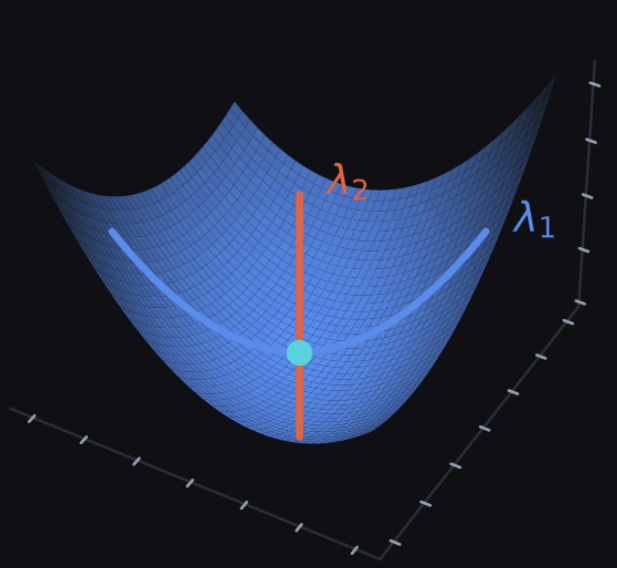


The Hessian eigenvectors give the principal curvature directions. One direction can be shallow while another is steep, producing an elongated valley.

Hessian geometry

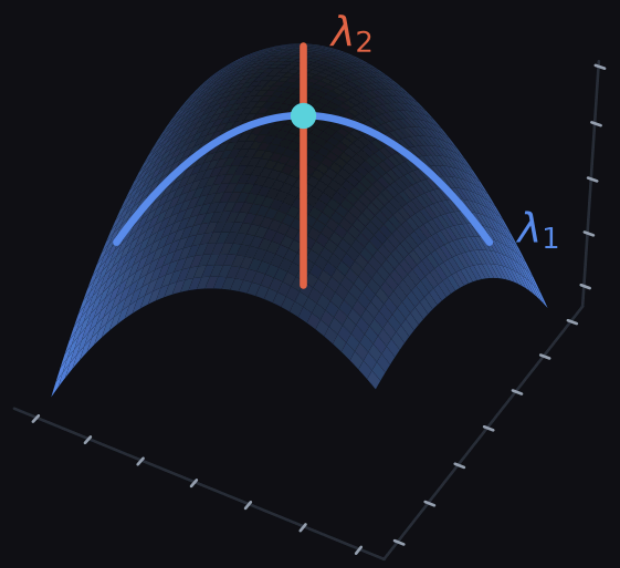
bowl

$\lambda_1 > 0,\ \lambda_2 > 0$



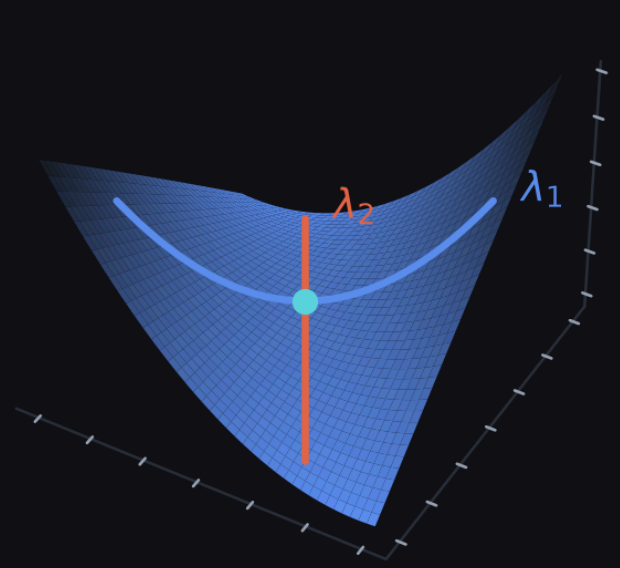
hill

$\lambda_1 < 0,\ \lambda_2 < 0$



saddle

$\lambda_1 > 0,\ \lambda_2 < 0$



The signs of the Hessian eigenvalues distinguish locally bowl-shaped, hill-shaped, and saddle-like geometry.

Classifying stationary points

At a stationary point where $\nabla_\phi \mathcal{L} = \mathbf{0}$:

- all eigenvalues > 0 $\rightarrow$ locally bowl-shaped $\rightarrow$ strict local minimum
- all eigenvalues < 0 $\rightarrow$ locally hill-shaped $\rightarrow$ strict local maximum
- mixed signs $\rightarrow$ some directions up, some down $\rightarrow$ saddle point

Zero eigenvalues require more care: second-order information alone may not decide the case.

Convexity and the Hessian

For a twice-differentiable function on a convex domain:

- **positive semidefinite**, $\mathbf{H} \succeq 0$: every eigenvalue is non-negative, so no direction curves downward
- $\mathbf{H}(\phi) \succeq 0$ everywhere is sufficient for **convexity**
- $\mathbf{H}(\phi) \succ 0$ everywhere gives **strict convexity**

Non-convex neural-network losses

For models nonlinear in their **parameters**, the loss landscape is generally non-convex.

- adding nonlinear input features such as x^2 does not destroy convexity if the model remains linear in ϕ
- placing parameters inside nonlinear functions can produce a non-convex loss surface

Non-convex landscape features

Possible features include:

- many valleys
- local minima
- saddle points
- long flat directions
- directions with very different curvature

The gradient still gives a local descent direction, but non-convexity removes the guarantee that following it will reach the global minimum.

Non-convex does not mean untrainable

Non-convexity removes the global-minimum guarantee. It does **not** imply that gradient-based training fails.

The Gabor model

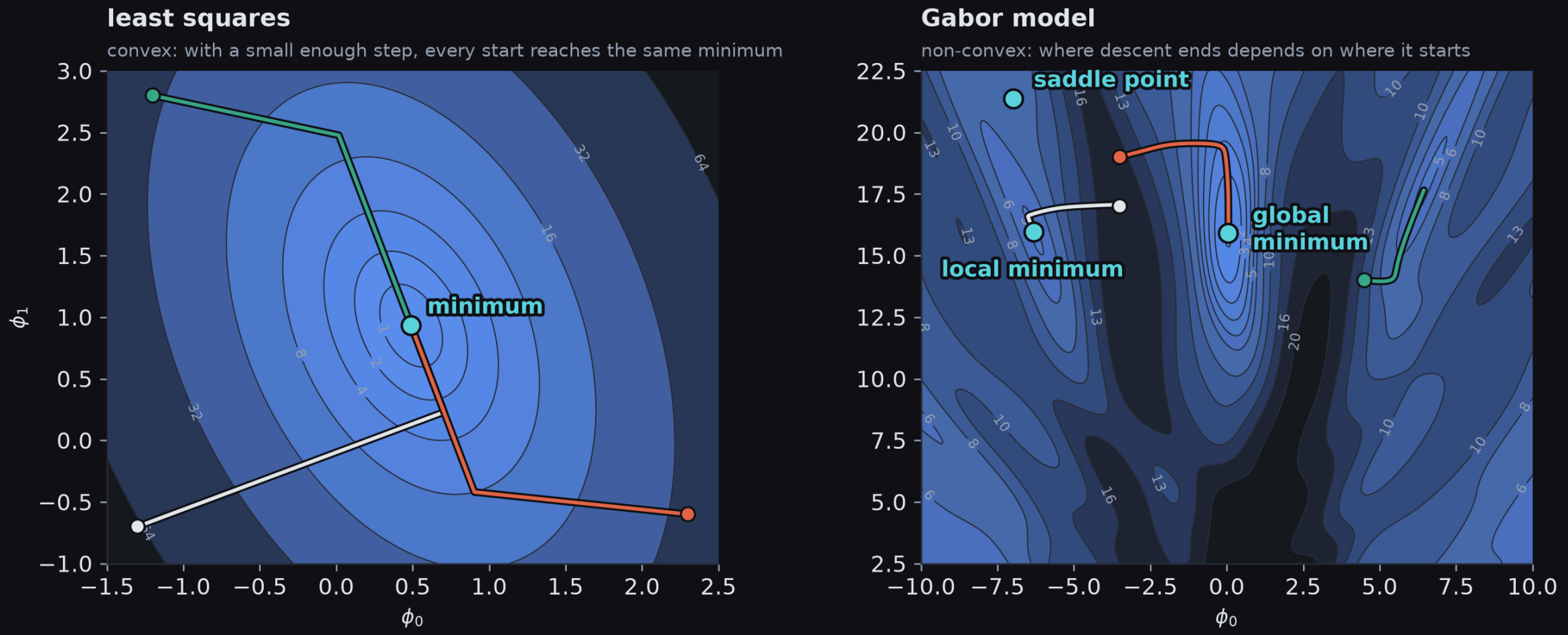
Prince's Gabor model (equation 6.8):

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin(z)\, \exp\left(-\frac{z^2}{32}\right), \qquad z = \phi_0 + 0.06\, \phi_1 x.$$

- ϕ_0 shifts the wave along the input axis
- ϕ_1 squeezes or stretches it
- **The data**: 40 noisy samples from one true setting of the pair

With two parameters we can still plot the complete loss surface; unlike least squares, it is non-convex.

Convex and non-convex loss landscapes



Convex objectives have no suboptimal local minima; non-convex objectives can contain multiple valleys and saddle points.

Local minima and saddle points

A **local minimum** has no nearby downhill direction, but a better solution may exist elsewhere.

A **saddle point** need not be a minimum even when the gradient is zero. At a **strict saddle**, curvature is upward in some directions and downward in others.

Therefore

$$\nabla_\phi \mathcal{L} \approx \mathbf{0}$$

shows that the local slope is small; it does not prove that the global minimum has been found.

Second-order methods

Near a well-behaved local minimum, second-order methods such as Newton's method use curvature to rescale the gradient:

$$\Delta\boldsymbol{\phi} \approx -\mathbf{H}^{-1}\nabla_{\boldsymbol{\phi}}\mathcal{L}.$$

Curvature adjusts the scale of the update:

- steep curvature $\rightarrow$ move cautiously
- shallow curvature $\rightarrow$ move further

But a neural network with millions of parameters would have an enormous Hessian. Forming and inverting it directly is usually impractical.

Stochastic and adaptive optimisation

Cost of full-batch updates

The full gradient uses every training example:

$$\nabla_{\phi}\mathcal{L} = \sum_{i=1}^{I} \nabla_{\phi}\ell_i.$$

For a large dataset, computing that exact sum before every update is expensive. A minibatch lets us update the parameters after examining only part of the dataset.

Stochastic gradient descent

Instead of using the whole dataset, choose a minibatch $\mathcal{B}_t$ at step t.

Define the minibatch gradient

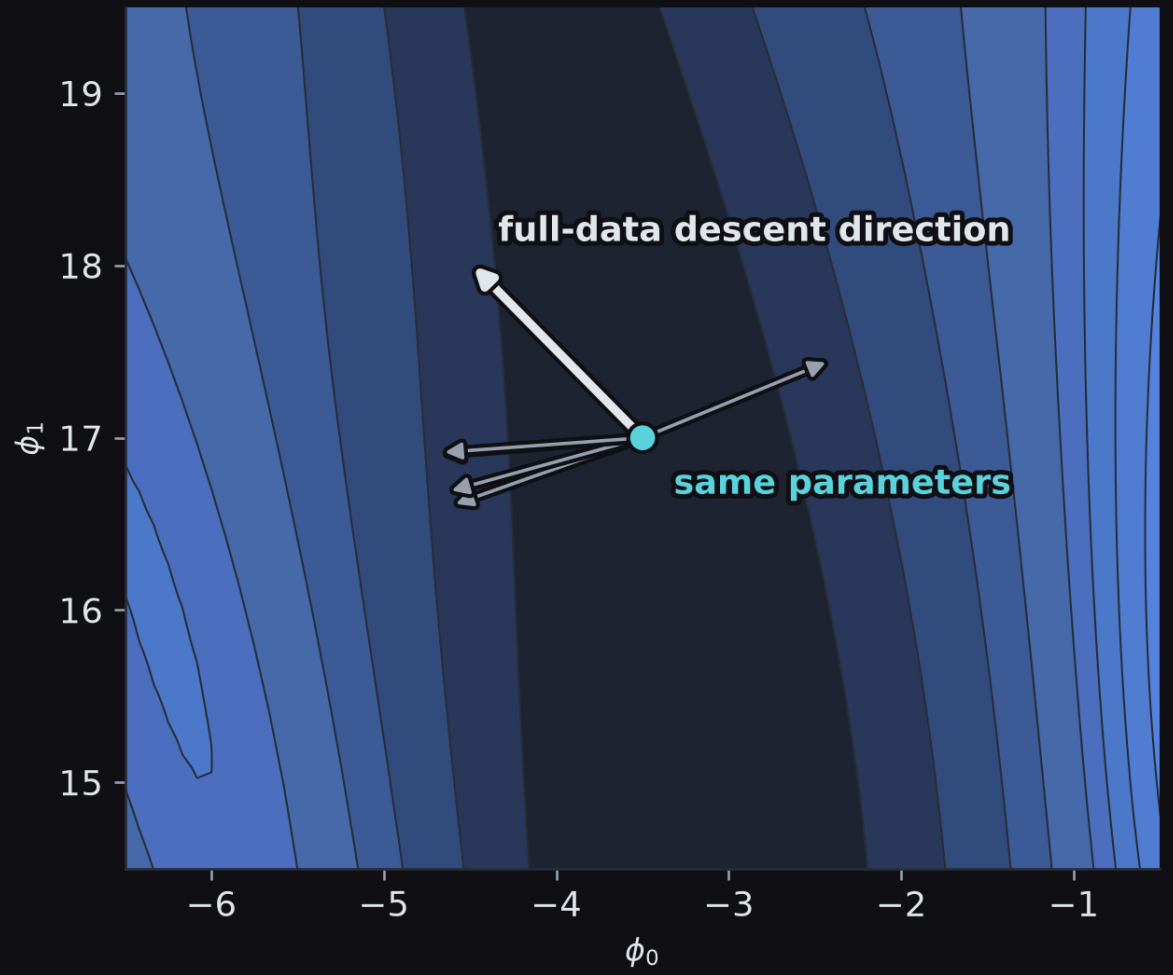
$$\mathbf{g}_t = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \nabla_{\boldsymbol{\phi}} \ell_i(\boldsymbol{\phi}_t).$$

Then update

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \mathbf{g}_t.$$

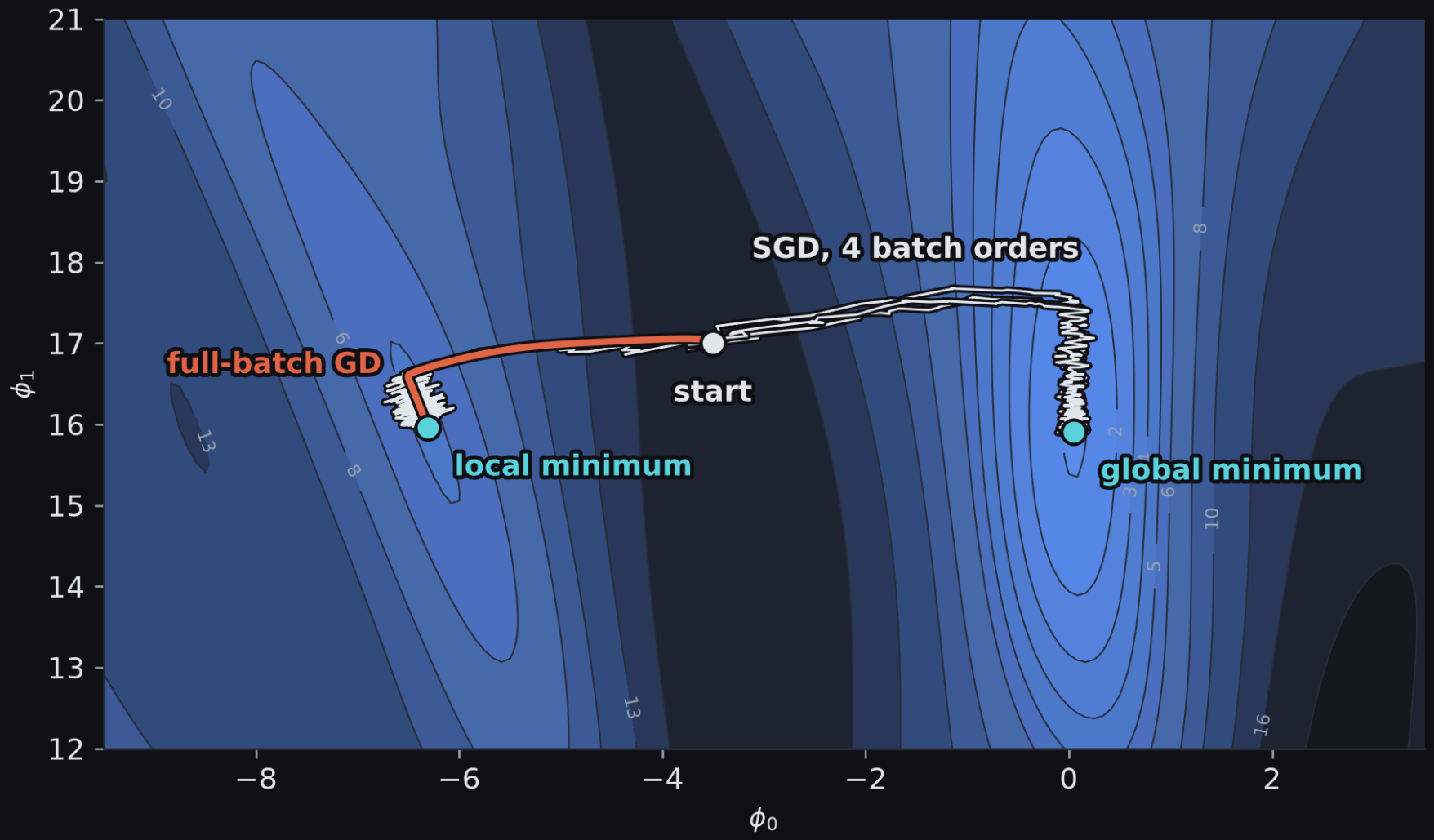
Averaging makes the gradient scale less dependent on batch size. Relative to the earlier **sum-loss** convention, this estimates $\frac{1}{I}\nabla_{\boldsymbol{\phi}}\mathcal{L}$; the constant factor I can be absorbed into the learning rate.

Minibatch gradient variation



At the same parameter setting, different minibatches give different descent directions. The highlighted arrow is the full-data direction; all arrows are drawn at a fixed length, so only their direction carries meaning.

Full-batch and stochastic descent paths



Full-batch gradient descent follows one deterministic gradient field; minibatch sampling perturbs the gradient and therefore the trajectory.

Effects of minibatching

A minibatch gradient is an estimate of the full-dataset **mean** gradient.

The sampling noise changes the optimisation trajectory.

Each batch defines a slightly different local estimate of “downhill.”

Minibatching introduces **sampling noise** into the trajectory:

- each step is cheaper
- updates begin before scanning the whole dataset
- the perturbed trajectory can sometimes move away from flat or suboptimal regions
- under uniform sampling, the expected minibatch gradient equals the full-dataset **mean** gradient

Batch size and learning rate interact

Batch size and learning rate jointly determine how noisy the parameter trajectory is.

Batch, iteration, epoch

These terms describe different units of training.

- **Batch / minibatch:** the examples used to compute one gradient estimate
- **Iteration / step:** one parameter update
- **Epoch:** one pass through the training dataset

Special cases:

- batch size 1 $\rightarrow$ classical stochastic update
- batch size = dataset size $\rightarrow$ full-batch gradient descent

Shuffling without replacement

A common training loop shuffles the examples and then forms minibatches without replacement.

This ensures that:

- within an epoch, each training example contributes once
- between steps, the optimiser sees different subsets and therefore different gradient estimates

The randomness comes from **which examples are grouped together and when they are seen**.

Stochastic trajectories

With full-batch gradient descent, each point in parameter space has one deterministic gradient.

With SGD, the gradient also depends on the sampled batch:

$$\mathbf{g}_t = \mathbf{g}(\boldsymbol{\phi}_t, \mathcal{B}_t).$$

SGD therefore follows the gradient of a **slightly different minibatch loss at each step**.

The path may temporarily move uphill on the full training loss even when the current minibatch gradient points downhill for its own batch.

Learning-rate schedules

Many training schedules use larger steps early and reduce them later.

Early training:

- parameters may still be far from a low-loss region
- larger steps can move through parameter space quickly

Later training:

- parameters may be nearer a low-loss region
- smaller steps allow finer adjustment

A **learning-rate schedule** reduces α during training.

Some training recipes also use **warm-up**: begin with a very small learning rate and gradually increase it, limiting early update sizes while activations, gradients, and optimiser state may not yet be well behaved.

Oscillation in narrow valleys

Imagine a long narrow valley:

- very steep from side to side
- shallow along its length

Plain gradient descent can zig-zag across the steep direction while making slow progress along the shallow direction.

Momentum averages gradients over successive steps: persistent directions reinforce, while directions that keep reversing tend to cancel.

Momentum

Let $\mathbf{g}_t$ be the current minibatch gradient.

Momentum maintains an exponentially smoothed gradient:

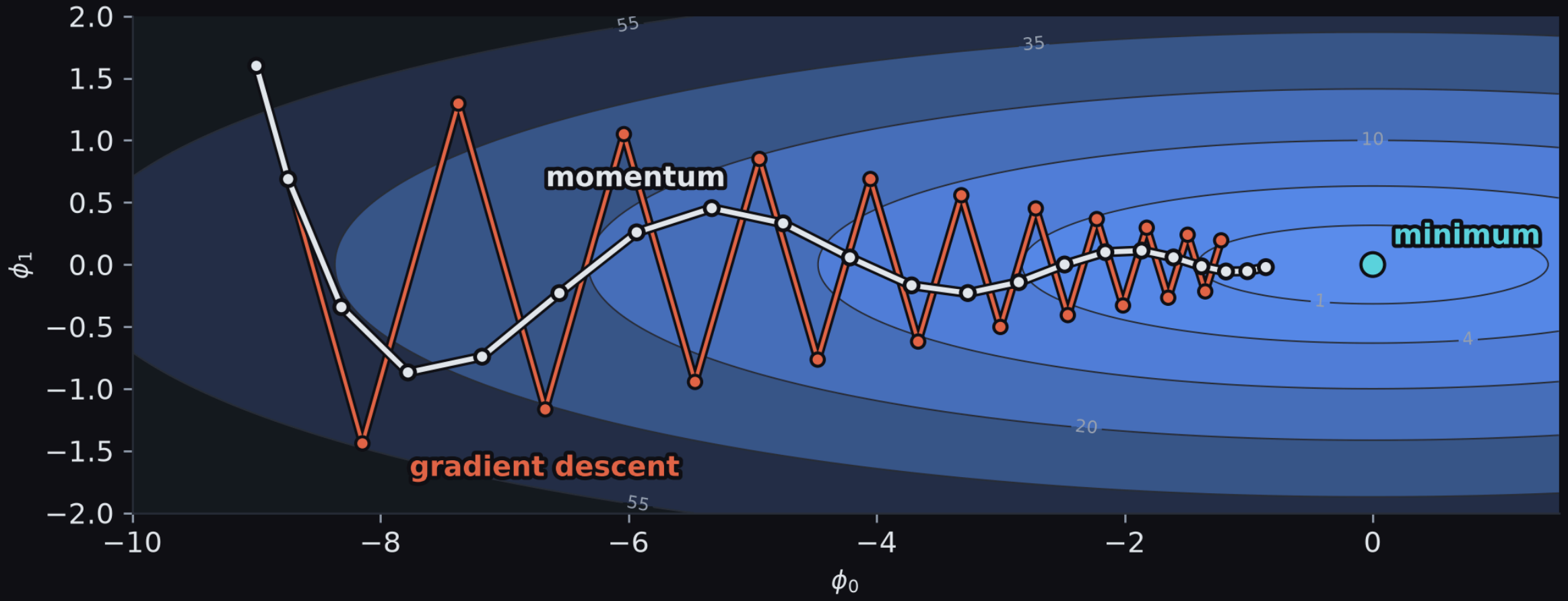
$$\underbrace{\mathbf{m}_{t+1}}_{\text{smoothed gradient}} = \underbrace{\beta \mathbf{m}_t}_{\text{keep some history}} + \underbrace{(1-\beta)\mathbf{g}_t}_{\text{add current evidence}}.$$

Then update

$$\boldsymbol{\phi}_{t+1} = \boldsymbol{\phi}_t - \alpha \mathbf{m}_{t+1}.$$

Under a constant gradient, this exponential moving average settles at $\mathbf{m} = \mathbf{g}$.

Momentum in a narrow valley



Averaging gradients over time damps oscillation across steep directions while preserving progress along persistent directions.

Persistent and reversing gradients

Repeatedly aligned gradients reinforce one another.

Repeatedly reversing gradients cancel one another.

Momentum tends to:

- reinforce persistent directions
- damp oscillation in reversing directions
- smooth minibatch noise

β controls memory:

- small $\beta \rightarrow$ react strongly to the current batch
- large $\beta \rightarrow$ average over a longer history

Nesterov momentum

Ordinary momentum measures the gradient **where we are now** and then combines it with momentum.

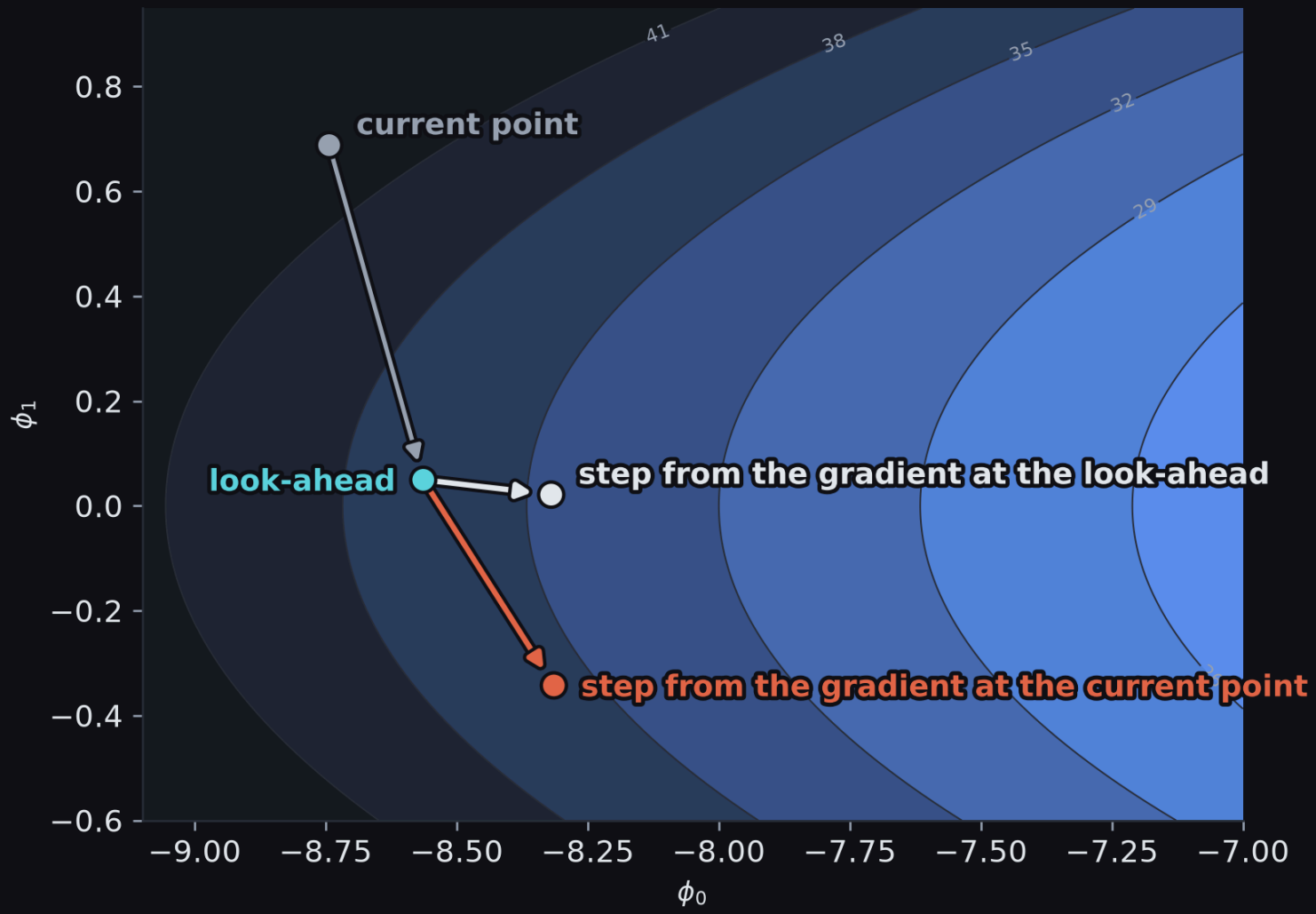
Nesterov momentum first moves to the position predicted by the momentum term.

$$\text{look-ahead point} = \boldsymbol{\phi}_t - \alpha\beta\mathbf{m}_t.$$

Then measure the gradient at that predicted position.

The gradient at the look-ahead point corrects the momentum-predicted trajectory.

The Nesterov look-ahead



Nesterov momentum first follows the momentum direction, then evaluates the gradient at the predicted look-ahead point.

Coordinatewise step scales

A fixed global learning rate assumes the same step scale is sensible in every coordinate.

- **Gradient magnitudes differ** across layers and parameters, often by orders of magnitude
- one global α is then too large in some coordinates and too small in others

Adaptive methods instead scale each coordinate using its own gradient history.

- coordinatewise scaling can compensate for different gradient magnitudes across coordinates
- it cannot rotate the coordinate system to align with an oblique steep direction

AdaGrad introduced this general idea; RMSProp and AdaDelta modified how squared-gradient history is accumulated. Adam combines this adaptive scaling with momentum-like averaging of the gradient itself.

Coordinatewise gradient scaling

Suppose the gradient is $\mathbf{g}_t$.

As a thought experiment, divide each coordinate by its own magnitude:

$$\frac{\mathbf{g}_t}{\sqrt{\mathbf{g}_t \odot \mathbf{g}_t} + \epsilon},$$

then for $|g_{t,j}| \gg \epsilon$, each coordinate is approximately $\operatorname{sign}(g_{t,j})$.

This is **coordinatewise scaling**, not vector normalisation by $\|\mathbf{g}_t\|$.

- $\odot$ means elementwise multiplication
- ϵ prevents division by zero

This prevents steep coordinates from automatically receiving the largest parameter changes, but using only the sign discards too much information and can oscillate around the minimum.

Adam: direction and scale

Adam maintains two moving averages.

First moment — a smoothed gradient:

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1)\mathbf{g}_t.$$

Second raw moment — a smoothed squared gradient:

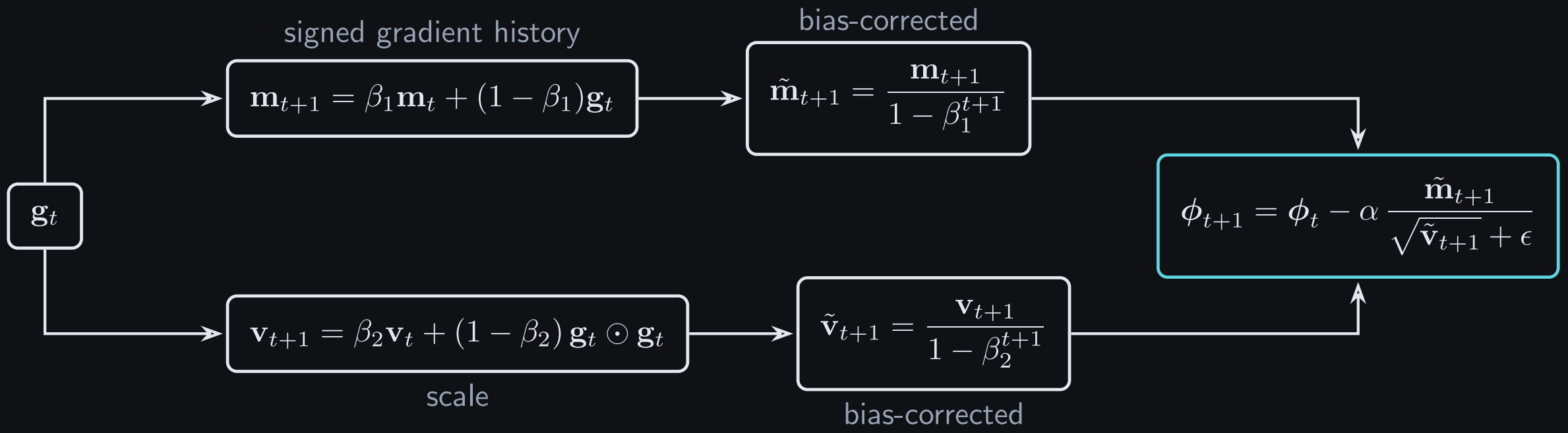
$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)(\mathbf{g}_t \odot \mathbf{g}_t).$$

$\mathbf{m}_t$ tracks the **recent signed gradient**; $\mathbf{v}_t$ tracks **typical squared magnitude per coordinate**.

The terminology “first moment” and “second raw moment” is mean-like language. In particular, $\mathbf{v}_t$ is **not the variance** of the gradient.

In PyTorch, `Adam(betas=...)` takes the pair (β_1, β_2).

Adam's two moving averages



Adam combines a moving average of gradients with a moving average of squared gradients to produce an adaptively scaled update.

Adam bias correction

At the start of training,

$$\mathbf{m}_0 = \mathbf{0}, \qquad \mathbf{v}_0 = \mathbf{0}.$$

Because both moving averages start at zero, their early values are biased toward zero.

Adam compensates with

$$\tilde{\mathbf{m}}_{t+1} = \frac{\mathbf{m}_{t+1}}{1 - \beta_1^{t+1}}, \qquad \tilde{\mathbf{v}}_{t+1} = \frac{\mathbf{v}_{t+1}}{1 - \beta_2^{t+1}}.$$

As t grows, β_1^{t+1} and β_2^{t+1} shrink and the bias-correction factors approach one.

The Adam update

The parameter update is

$$\phi_{t+1} = \phi_t - \underbrace{\alpha}_{\text{global scale}} \frac{\overbrace{\tilde{\mathbf{m}}_{t+1}}^{\text{smoothed gradient}}}{\underbrace{\sqrt{\tilde{\mathbf{v}}_{t+1}}}_{\text{coordinatewise scale}} + \underbrace{\epsilon}_{\text{numerical stability}}}.$$

Adam combines **momentum-like gradient smoothing** with **per-coordinate adaptive scaling**. The square root and division are applied elementwise.

Effect of adaptive scaling

A single global step size is difficult to tune when gradient magnitudes differ substantially across coordinates.

Adam reduces the effective step in coordinates with persistently large squared gradients and allows relatively larger movement where gradients have been smaller.

Adaptive scaling reduces sensitivity to differences in gradient scale across coordinates, but the global learning rate still matters.

Adam versus SGD

The relative behaviour of adaptive methods and SGD depends on the objective and training setup.

In practice:

- Adam often makes rapid initial progress
- SGD with momentum can be highly competitive when carefully tuned
- optimiser comparisons are entangled with learning-rate schedules and other hyperparameters

Neither optimiser dominates across architectures, datasets, and training recipes.

Optimisers and backpropagation

Parameters vs hyperparameters

Model parameters are learned from data:

$$\phi.$$

Training hyperparameters control the learning procedure, for example:

- optimiser choice
- learning rate and schedule
- batch size
- momentum coefficients
- number of epochs

Parameters are optimised **inside one training run**.

Hyperparameters are compared **across training runs**.

Information used by each optimiser

The optimisers differ in the information used to construct each update.

Gradient descent

- current gradient

Momentum

- current gradient + recent gradient history

Adam

- smoothed gradient history + coordinatewise squared-gradient history

Newton-style methods

- current gradient + explicit Hessian curvature

Where the optimiser acts

Training repeatedly follows this sequence:

$$\text{parameters} \to \text{predictions} \to \text{loss} \to \text{gradient} \to \text{parameter update}.$$

Once the gradient is known, the optimisation methods in this lecture perform the final step:

$$\nabla_{\phi}\mathcal{L} \quad \longrightarrow \quad \Delta\phi.$$

The optimiser decides how to turn that gradient into a parameter update.

For a deep network, the remaining problem is to compute that gradient efficiently.

Deep networks as function compositions

A deep network does not map parameters to loss in one step.

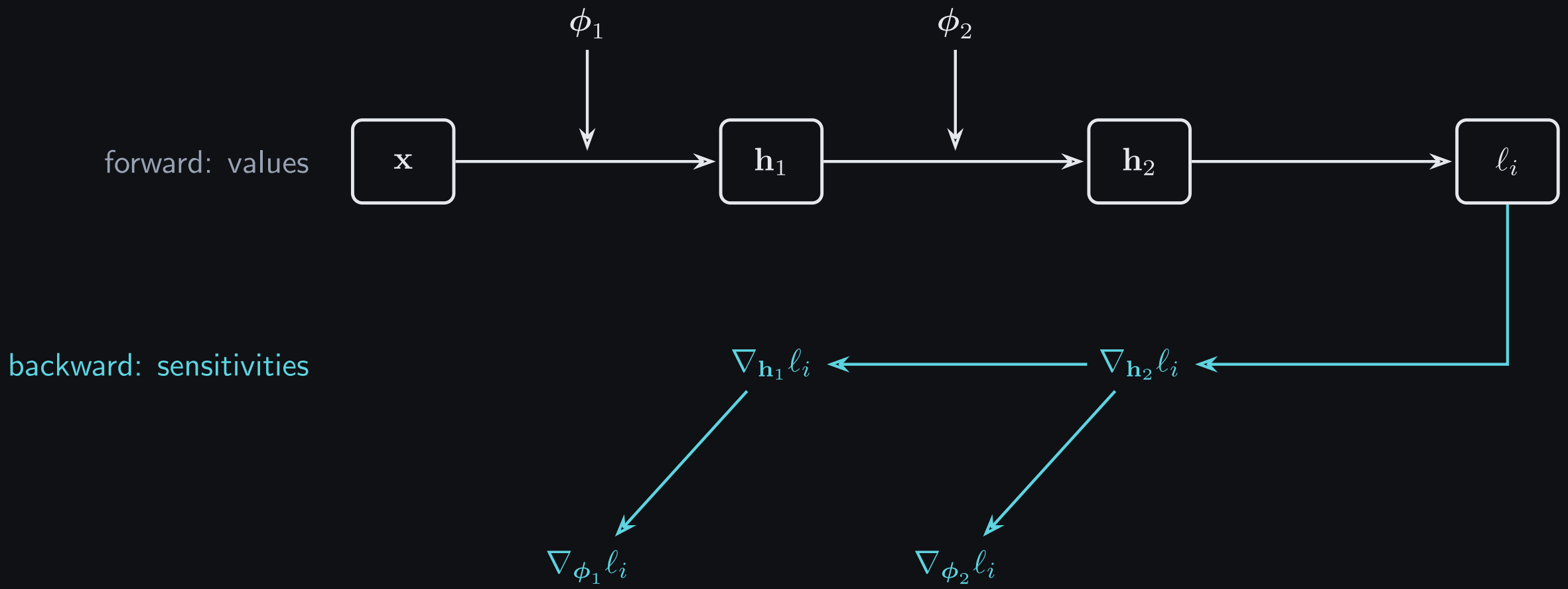
A simplified computation looks like

$$\mathbf{x} \to \mathbf{h}_1 \to \mathbf{h}_2 \to \cdots \to \hat{\mathbf{y}} \to \mathcal{L}.$$

Each intermediate quantity depends on the one before it, so changing an early parameter can affect the loss through many successive operations.

The chain rule tells us how to compose these local sensitivities.

Computation graph: forward and backward



The forward computation produces activations and a loss; the reverse computation asks how that loss is sensitive to earlier quantities and parameters.

The gradient problem in deep networks

Suppose the network is written as a composition

$$\mathbf{h}_1 = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}_1], \qquad \mathbf{h}_2 = \mathbf{f}[\mathbf{h}_1, \boldsymbol{\phi}_2], \qquad \ell_i = \ell(\mathbf{h}_2).$$

- **One example at a time** here: dataset or minibatch gradients aggregate these per-example gradients

The optimiser ultimately needs sensitivities such as

$$\nabla_{\boldsymbol{\phi}_1} \ell_i, \qquad \nabla_{\boldsymbol{\phi}_2} \ell_i.$$

Computing those gradients independently, parameter by parameter, would be impractical in a deep network.

Backpropagation is the efficient reverse computation used to obtain them from the computation graph.

Initialisation affects gradient scale

Backpropagation repeatedly combines derivatives from successive layers.

For an early layer, the gradient contains products of many local derivative terms.

If those terms are repeatedly small, the gradient can shrink as it moves backward.

If they are repeatedly large, the gradient can grow rapidly.

The initial parameter values therefore affect not only the initial predictions, but also the **scale of the gradients available for learning**.

Two issues become central in deep networks:

- **gradient computation**: obtain all parameter sensitivities efficiently
- **initialisation**: choose starting parameters that preserve useful signal and gradient scales

Backpropagation addresses the first; the choice of initial parameter distribution addresses the second.

Summary

- Training is optimisation in **parameter space**
- A partial derivative measures sensitivity to one parameter; $\nabla_\phi \mathcal{L}$ stacks all parameter sensitivities into the gradient
- Gradient descent uses the locally steepest **first-order** downhill direction
- The Jacobian organises first derivatives of vector-valued functions
- The Hessian organises second derivatives; its eigenvectors give principal curvature directions and its eigenvalues give signed curvature
- Non-convex losses can contain multiple minima, saddle points, and flat directions
- Minibatches trade exact gradients for cheaper, noisy estimates
- Momentum smooths gradient history; Adam also rescales coordinates adaptively
- Deep networks require efficient computation of parameter gradients; **backpropagation** addresses that problem

Source and reading

Primary source for this lecture:

- Simon J. D. Prince, *Understanding Deep Learning*, Chapter 6, **Fitting Models**
- Appendix B.5 for the matrix-calculus notation used in the gradient and Hessian sections