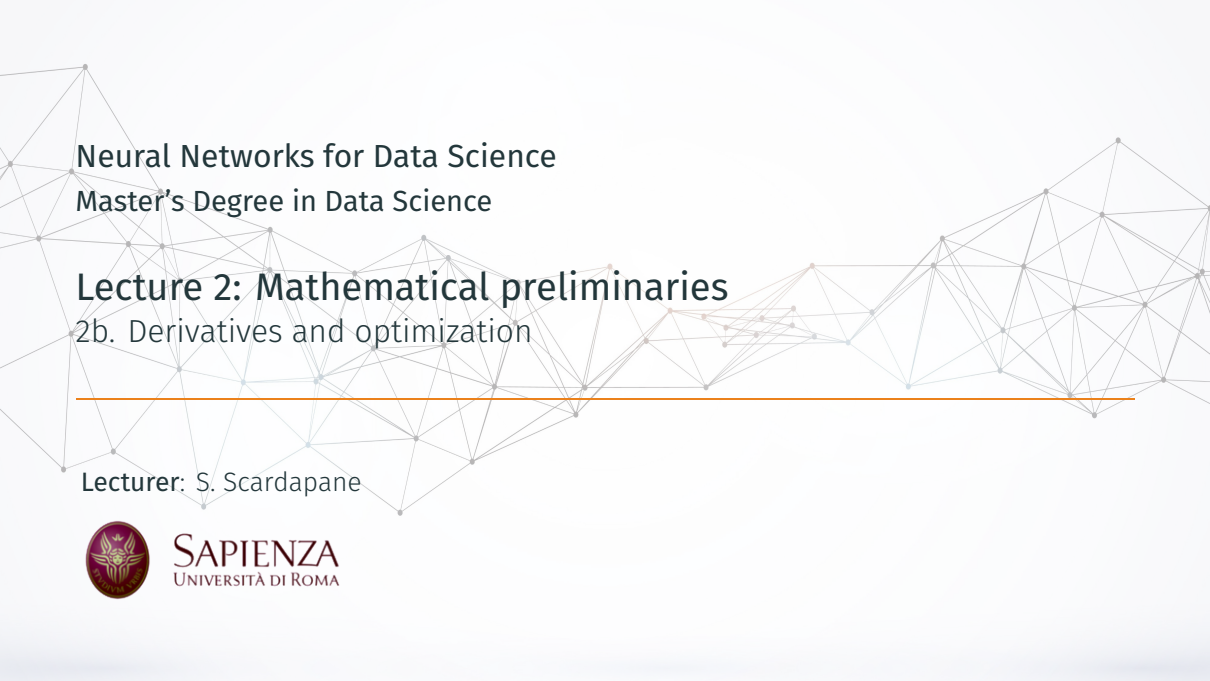




Neural Networks for Data Science
Master's Degree in Data Science

Lecture 2: Mathematical preliminaries

2b. Derivatives and optimization

Lecturer: S. Scardapane



SAPIENZA
UNIVERSITÀ DI ROMA

Derivatives

Derivatives and gradients

For a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ (more on this next lecture), let $\boldsymbol{\theta}$ denote the trainable parameters of a model $f_{\boldsymbol{\theta}}(\mathbf{x})$ (e.g., for a linear layer $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$) and define:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i),$$

where ℓ is some per-example loss. Training is the optimization problem:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}).$$

Maximizing is the same as minimizing $-\mathcal{L}$. How does $\mathcal{L}$ change when we perturb $\boldsymbol{\theta}$? This is what derivatives tell us.

Most of this course is based on the notion of derivative.

For $f : \mathbb{R} \rightarrow \mathbb{R}$, the **derivative** is:

$$\partial f(x) = \frac{\partial}{\partial x} f(x) = f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

Even for a continuous function, $\partial f(x)$ might not be defined everywhere.

Informally, the derivative expresses the rate of change of f around an infinitesimal displacement from x , or the slope of the line tangent to $f(x)$.

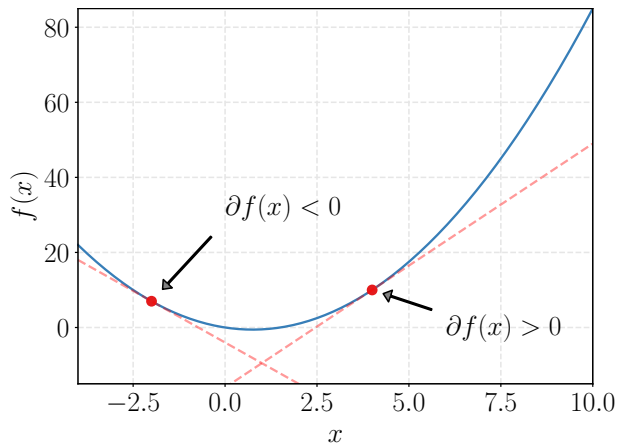


Figure 1: 1D function ($f(x) = x^2 - 1.5x$), showing the derivative at two different locations.

Derivatives possess a number of properties, most notably:

Linearity $\partial [f(x) + g(x)] = f'(x) + g'(x)$

Product rule $\partial [f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$

Chain rule $\partial [f(g(x))] = f'(g(x))g'(x)$

For $f : \mathbb{R}^D \rightarrow \mathbb{R}$, the **gradient** $\nabla f(\mathbf{x})$ is a D -dimensional vector defined as:

$$[\nabla f(\mathbf{x})]_i = \frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h},$$

where $\mathbf{e}_i$ is the i th standard basis vector, with a 1 in position i and 0 elsewhere.

Each entry is the derivative along one coordinate, keeping the others fixed.

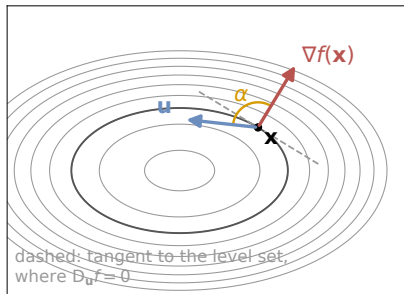
More generally, the **directional derivative** of $f : \mathbb{R}^D \rightarrow \mathbb{R}$ in the direction $\mathbf{u}$ is:

$$D_{\mathbf{u}}f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}.$$

It is easy to prove that:

$$D_{\mathbf{u}}f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{u} \rangle.$$

It is largest along the gradient and zero along the level set. A partial derivative is a directional derivative in the direction of a standard basis vector.



Everything extends to vector-valued functions $f : \mathbb{R}^D \rightarrow \mathbb{R}^H$, $\mathbf{y} = f(\mathbf{x})$:

The **Jacobian** $\partial f(\mathbf{x}) \sim (H, D)$ of f is the matrix:

$$\partial f(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_H}{\partial x_1} & \cdots & \frac{\partial y_H}{\partial x_D} \end{pmatrix}.$$

For $H = 1$, its single row is the gradient; for $D = H = 1$, we recover the standard derivative.

Three derivatives we will use often:

Inner product

$$f : \mathbb{R}^D \rightarrow \mathbb{R}$$

$$\nabla_{\mathbf{x}} \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{y}$$

Linear map

$$f : \mathbb{R}^D \rightarrow \mathbb{R}^H$$

$$\partial_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}$$

Squared Euclidean norm

$$f : \mathbb{R}^D \rightarrow \mathbb{R}$$

$$\nabla \|\mathbf{x}\|^2 = 2\mathbf{x}$$

Jacobians inherit many properties from the scalar case. Importantly, there exists a chain rule for Jacobians. For $g : \mathbb{R}^D \rightarrow \mathbb{R}^H$ and $f : \mathbb{R}^H \rightarrow \mathbb{R}^O$:

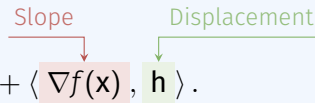
$$\begin{array}{ccc}
 \boxed{\partial[f(g(\mathbf{x}))]} & = & \boxed{\partial f(g(\mathbf{x}))} \boxed{\partial g(\mathbf{x})} \\
 (O, D) & & (O, H) \quad (H, D) \\
 & & \xleftrightarrow{\text{match}}
 \end{array}$$

In words: the Jacobian of the composition of two functions is the product of their Jacobian matrices.

For $f : \mathbb{R} \rightarrow \mathbb{R}$ and a small displacement h , the best **linear approximation** of f around x is (**Taylor's theorem**):

$$f(x + h) \approx f(x) + \partial f(x) h .$$

For $f : \mathbb{R}^D \rightarrow \mathbb{R}$ and a displacement $\mathbf{h}$:


$$f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \langle \nabla f(\mathbf{x}), \mathbf{h} \rangle .$$

Better approximations use higher-order derivatives, but this is enough for building effective optimization algorithms.

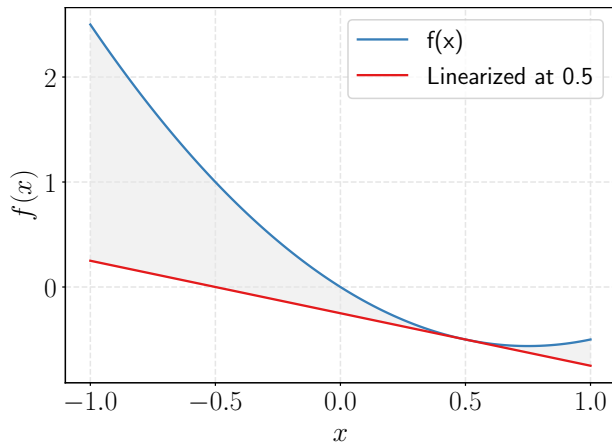


Figure 2: 1D function ($f(x) = x^2 - 1.5x$), linearized at 0.5.

Optimization

Gradient descent and convexity

A point θ such that $\mathcal{L}(\theta) \leq \mathcal{L}(\theta')$ for all θ' is a **global minimum**. It is a **local minimum** if this holds only nearby:

$$\mathcal{L}(\theta) \leq \mathcal{L}(\theta') \quad \text{for all } \theta' \text{ with } \|\theta' - \theta\| < \varepsilon,$$

for some $\varepsilon > 0$.

If $\nabla \mathcal{L}(\theta) = 0$, θ is a **stationary point**. Stationary points can be minima, maxima, or **saddle points**.

First-order optimization algorithms return stationary points, in practice local minima or saddle points.

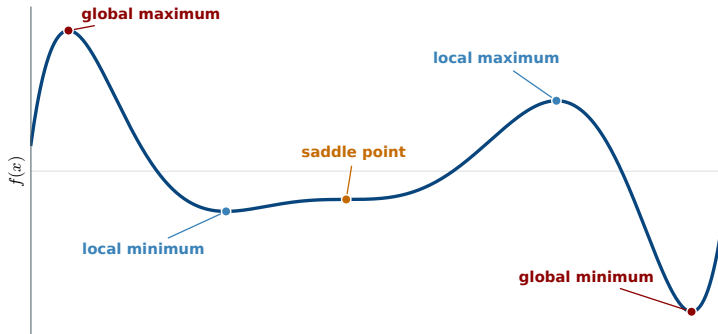
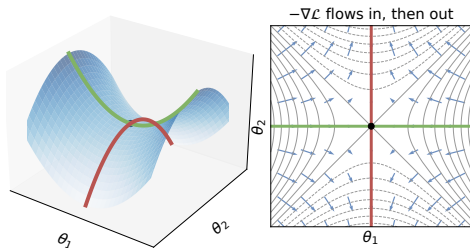


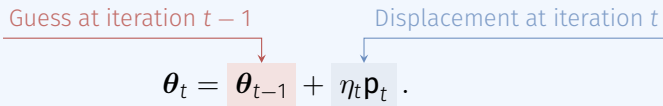
Figure 3: Stationary points can be **local** or **global** x extrema, or **saddle points**; here, the saddle is a stationary inflection.

For $\mathcal{L}(\boldsymbol{\theta}) = \theta_1^2 - \theta_2^2$, the gradient vanishes at $\mathbf{0}$, a **minimum** along θ_1 and a **maximum** along θ_2 :



A minimum must curve upwards in *every* direction, so with millions of parameters most stationary points are saddles. GD slows down near them; gradient noise helps to escape.

Given a randomly initialized θ_0 , an **iterative optimization algorithm** has the form:



The diagram shows the equation $\theta_t = \theta_{t-1} + \eta_t \mathbf{p}_t$. Above the term θ_{t-1} , the text "Guess at iteration $t - 1$ " is written in red, with a red line and arrow pointing to the θ_{t-1} term. Above the term $\eta_t \mathbf{p}_t$, the text "Displacement at iteration t " is written in blue, with a blue line and arrow pointing to the $\eta_t \mathbf{p}_t$ term. The θ_{t-1} term is highlighted with a light red background, and the $\eta_t \mathbf{p}_t$ term is highlighted with a light blue background.

$$\theta_t = \theta_{t-1} + \eta_t \mathbf{p}_t .$$

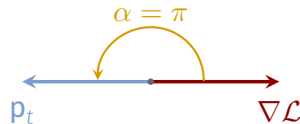
$\mathbf{p}_t$ is a **descent direction** if $\mathcal{L}(\theta_t) < \mathcal{L}(\theta_{t-1})$ for a sufficiently small $\eta_t > 0$, called the **step size** or **learning rate**.

To implement this algorithm, we need a way to choose a descent direction (and a step size) at every iteration.

We restrict to unit directions, $\|\mathbf{p}_t\| = 1$, and recall that $\langle \mathbf{a}, \mathbf{b} \rangle = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\alpha)$, where α is the angle between the two vectors.

The rate of change along $\mathbf{p}_t$ is:

$$\begin{aligned} D_{\mathbf{p}_t} \mathcal{L}(\boldsymbol{\theta}_{t-1}) &= \langle \nabla \mathcal{L}(\boldsymbol{\theta}_{t-1}), \mathbf{p}_t \rangle \\ &= \|\nabla \mathcal{L}(\boldsymbol{\theta}_{t-1})\| \underbrace{\|\mathbf{p}_t\|}_{=1} \cos(\alpha). \end{aligned}$$



Minimum at $\cos(\alpha) = -1$:

$$\mathbf{p}_t \propto -\nabla \mathcal{L}(\boldsymbol{\theta}_{t-1}).$$

This is the **steepest descent direction**. More generally, any direction with $\cos(\alpha) < 0$ is a descent direction.

The resulting algorithm is called **gradient descent**.

Gradient descent (GD) looks for stationary points by iterating:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_{t-1}).$$

This is the basic **first-order** method. The steepest local direction is not necessarily the *fastest* way to the minimum, and we will see how to accelerate it at the end of this lecture.

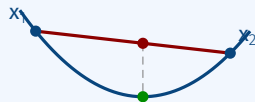
In the interactive part (2c), we see how η_t decides between slow, oscillating and diverging runs.

Convexity plays a pivotal role in optimization: convex functions are much easier to optimize than non-convex ones.

f is **convex** if, for any $\mathbf{x}_1, \mathbf{x}_2$ and $\lambda \in [0, 1]$:

$$f\left((1 - \lambda)\mathbf{x}_1 + \lambda\mathbf{x}_2\right) \leq (1 - \lambda)f(\mathbf{x}_1) + \lambda f(\mathbf{x}_2).$$

The graph lies **below** the **chord** joining any two of its points.



If the inequality is strict for $\mathbf{x}_1 \neq \mathbf{x}_2$ and $\lambda \in (0, 1)$, then f is **strictly convex**.

Under standard regularity and step-size assumptions, suppose (S)GD converges to a stationary point θ^* . More structure in $\mathcal{L}$ gives a stronger conclusion:

Non-convex

Only **stationarity**:
 θ^* may be a local minimum or a saddle point.

Convex

Every stationary point is a **global minimum**.

Strictly convex

If a minimum exists, it is **unique**.

Neural network losses are non-convex. Globally optimizing a general non-convex objective can be NP-hard, so in practice we settle for good stationary points.

Optimization

Stochastic gradient descent

Consider the steps needed for a single iteration of GD on our training problem:

1. Computing the output of the model $f_{\theta}(\mathbf{x}_i)$ on *all* examples;
2. Computing the gradient of the loss with respect to the parameters.

Both operations scale *linearly* in the number of examples N , which is unfeasible when we have 10^5 examples or more.

In practice, to train neural networks we use **stochastic** versions of GD, which approximate the gradient from smaller amounts of data.

The loss is an average over all N examples. We approximate it with a **mini-batch** $\mathcal{B}_t$ of B examples, sampled at random:

$$\tilde{\mathcal{L}}_t(\boldsymbol{\theta}) = \frac{1}{B} \sum_{i \in \mathcal{B}_t} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i).$$

The mini-batch gradient $\nabla \tilde{\mathcal{L}}_t$ is an unbiased, noisy estimate of the full gradient $\nabla \mathcal{L}$.

Stochastic gradient descent (SGD) uses $\nabla \tilde{\mathcal{L}}_t$ in place of $\nabla \mathcal{L}$. One iteration costs $O(B)$, regardless of N . With a suitable learning-rate schedule, it still drives the expected gradient norm towards zero.

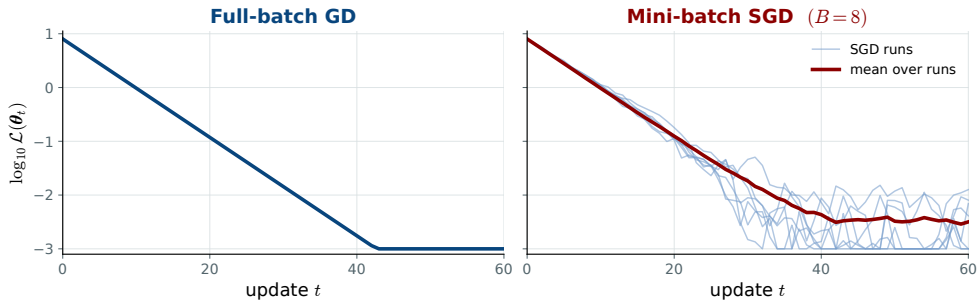
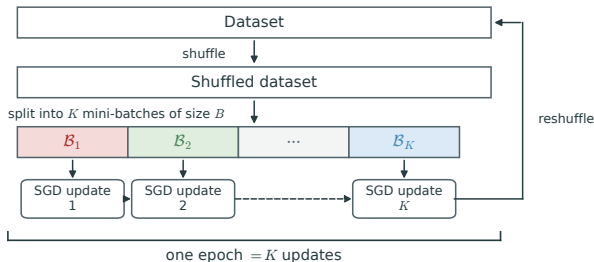


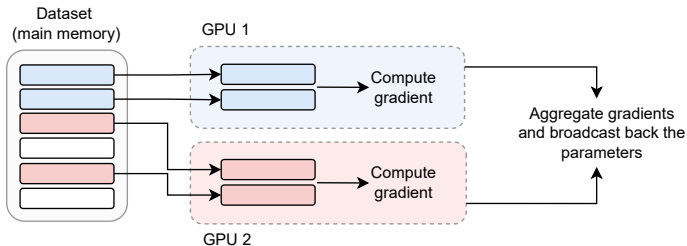
Figure 4: Comparison of standard GD (“full-batch”) and SGD: mini-batch sampling introduces noise in each update, but the loss still decreases on average.

Instead of sampling a new batch at every iteration, we usually **shuffle** the dataset, **split** it into blocks of B examples processed one after the other, and **repeat**.



A full pass over the dataset is an **epoch**. Reading contiguous blocks is efficient, and epochs are convenient to define metrics and stopping criteria.

With several GPUs, we replicate the weights and give each GPU its own mini-batch. Since the gradient is linear, we can average the per-GPU gradients and broadcast the result, which needs synchronization.



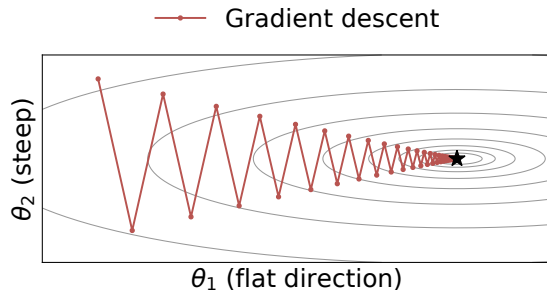
Optimization

Momentum and Adam (optional)

Consider an elongated quadratic:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2} (\theta_1^2 + \kappa \theta_2^2), \quad \kappa = 20.$$

The gradient points mostly across the valley. GD oscillates along the steep direction and moves slowly along the flat one.



To stay stable, the step size must satisfy $\eta < 2/\kappa$ for the steep direction. The flat direction then shrinks by only $1 - \eta$ per step.

Momentum smooths the path; Adam rescales each coordinate.

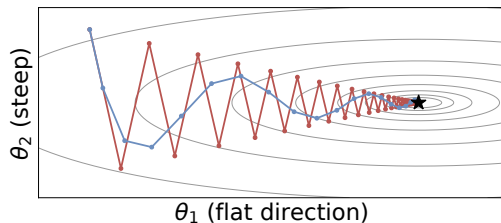
Momentum keeps a running displacement $\mathbf{u}_t$, with $\mathbf{u}_0 = \mathbf{0}$ and $0 \leq \lambda < 1$:

$$\mathbf{u}_t = \lambda \mathbf{u}_{t-1} - \eta_t \nabla \tilde{\mathcal{L}}_t(\boldsymbol{\theta}_{t-1}),$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \mathbf{u}_t.$$

Each step adds the **new gradient** to a **fraction of the previous step**.

— Gradient descent — Momentum



Unrolling, $\mathbf{u}_t$ sums all past gradients with weights $1, \lambda, \lambda^2, \dots$: directions that agree accumulate, oscillations cancel. A common choice is $\lambda = 0.9$.

Adam keeps two averages of $\mathbf{g}_t = \nabla \tilde{\mathcal{L}}_t(\boldsymbol{\theta}_{t-1})$:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \quad \text{average direction}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t \quad \text{average squared size}$$

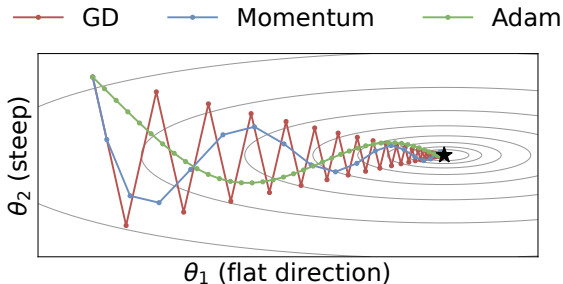
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon} \quad \text{per-coordinate step}$$

Both averages start at zero, so we correct their bias:

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t}.$$

Each coordinate is divided by its typical gradient size, so steep and flat directions take comparable steps. Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$.

On the same quadratic, running each method until it reaches the minimum:



Adam's steps have similar size in both coordinates, so it heads along the valley. Adam (with weight decay) is the default optimizer for most networks.

This lecture follows **Chapter 2** of the book, along with the suggested exercises.

- ▶ In the interactive part (2c), we explore step sizes, batch sizes and learning-rate schedules.
- ▶ Next, we use these tools to train our first models: linear models for regression and classification.