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 and gradients

Derivatives and gradients

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

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell\Big(f_{\boldsymbol{\theta}}(x_i), y_i\Big) \,. \tag{1}$$

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

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

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

The **derivative** of a function $f(x)$ is defined as:

$$\partial f(x) = \frac{\partial}{\partial x} f(x) = f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2}$$

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 $f(x)$.

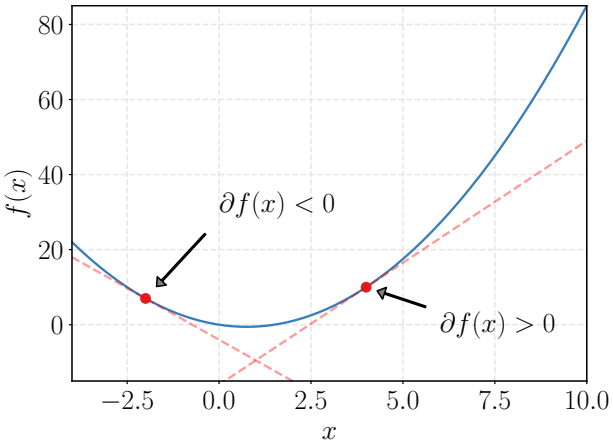


Figure 1: 1D function ($f(x) = x^2 - 1.5x$), showing the derivative at two different locations.

Derivatives possess a number of properties, most notably:

- **Linearity**:
$$\partial\Big[f(x)+g(x)\Big] = f'(x)+g'(x)\,.$$
- **Product rule**:
$$\partial\Big[f(x)g(x)\Big] = f'(x)g(x)+f(x)g'(x)\,,$$
- **Chain rule**
$$\partial\Big[f(g(x))\Big] = f'(g(x))g'(x)\,.$$

For a function $y = f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^m$, the **gradient** $\partial f(\mathbf{x})$ is an m-dimensional vector defined as:

$$[\partial f(\mathbf{x})]_i = \frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \tag{3}$$

where $\mathbf{e}_i$ is the ith standard basis vector:

$$[\mathbf{e}_i]_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Sometimes we use the alternative notation $\nabla f(\mathbf{x})$.

More generally, the **directional derivative** of $f(\mathbf{x})$ in the direction $\mathbf{v}$ is:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}, \tag{4}$$

It is easy to prove that:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle . \tag{5}$$

A partial derivative is a directional derivative in the direction of a standard basis vector.

Everything extends to vector-valued functions $\mathbf{y} = f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$:

The **Jacobian** $\underset{(n,m)}{\partial f(\mathbf{x})}$ of f is a matrix defined as:

$$\partial f(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix}. \tag{6}$$

For $n = 1$, we recover the gradient, while for $m = n = 1$ we recover the standard derivative.

Derivative of the inner product:

$$\frac{\partial}{\partial \mathbf{x}} \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{y} \,.$$

Derivative of a linear map:

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A} \,.$$

Derivative of the squared Euclidean norm:

$$\partial \|\mathbf{x}\|^2 = 2\mathbf{x} \,.$$

See the matrix cookbook for reference:
`https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`.

Jacobians inherit many properties from the scalar case. Importantly, there exists a chain rule for Jacobians. For $f : \mathbb{R}^m \to \mathbb{R}^n$ and $g : \mathbb{R}^o \to \mathbb{R}^m$:

$$\underbrace{\partial[f(g(\mathbf{x}))]}_{(n,o)} = \underbrace{\partial f(g(\mathbf{x}))}_{(n,\boldsymbol{m})} \underbrace{\partial g(\mathbf{x})}_{(\boldsymbol{m},o)}$$

match

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

Given a scalar-valued function f and a reference point $\mathbf{x}_0$, the best **linear approximation** of f around $\mathbf{x}_0$ is given by (Taylor's theorem):

$$\widetilde{f}(\mathbf{x}) = f(\mathbf{x}_0) + \langle \partial f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$

Better approximations can be constructed from higher-order derivatives, but this is enough for building effective optimization algorithms.

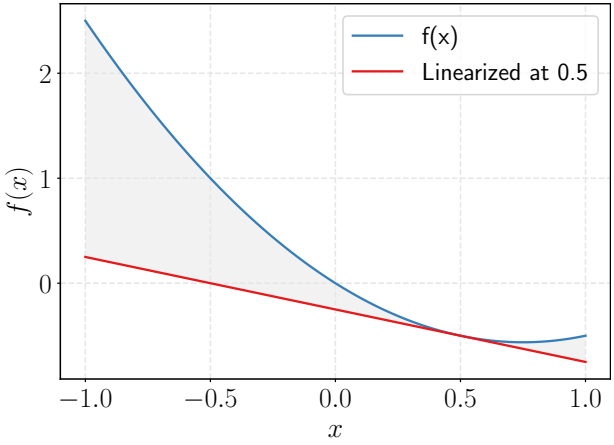


Figure 2: 1D function ($f(x) = x^2 - 1.5x$), linearized at 0.5.

Derivatives and gradients

Numerical optimization

We can use gradients to solve generic problems of the form:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{7}$$

Note that maximizing/minimizing are equivalent in the sense that:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} -\mathcal{L}(\boldsymbol{\theta}) \tag{8}$$

A point $\boldsymbol{\theta}$ such that $\mathcal{L}(\boldsymbol{\theta}) \leq \mathcal{L}(\boldsymbol{\theta}') \ \ \forall\, \boldsymbol{\theta}'$ is called a **global minimum**. If instead (less restrictive):

$$\mathcal{L}(\boldsymbol{\theta}) \leq \mathcal{L}(\boldsymbol{\theta}') \ \ \forall\, \boldsymbol{\theta}' \in \left\{\boldsymbol{\theta}' : \|\boldsymbol{\theta}' - \boldsymbol{\theta}\|^2 < \varepsilon\right\} \tag{9}$$

for some $\varepsilon > 0$, it is called a **local minimum**.

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

First-order optimization algorithms return stationary points, which are either local minima or saddle points.

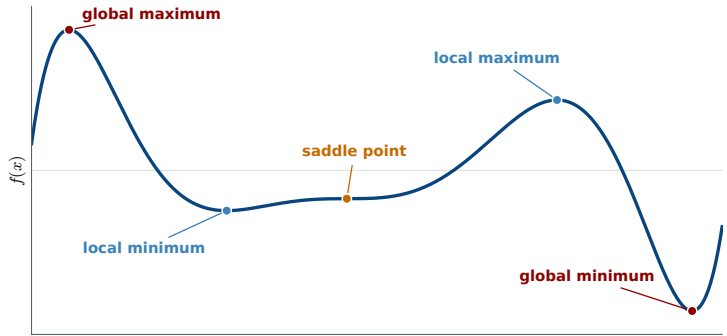


Figure 3: Stationary points can be **local** or **global** extrema, or **saddle points**; here, the saddle is a stationary inflection.

Given a randomly initialized $\boldsymbol{\theta}_0$, an **iterative optimization algorithm** has the form:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \eta_t \mathbf{p}_t \tag{10}$$

$\mathbf{p}_t$ is called a **descent direction** for $\mathcal{L}(\boldsymbol{\theta}_{t-1})$ if $\mathcal{L}(\boldsymbol{\theta}_t) < \mathcal{L}(\boldsymbol{\theta}_{t-1})$ for a sufficiently small η_t. η_t is called **step size** or **learning rate**.

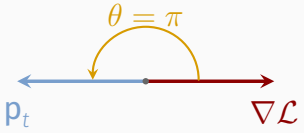
To implement this iterative algorithm we need a way to choose a descent direction (and a step size) for all t.

We restrict to unit directions, $\|\mathbf{p}_t\| = 1$.

Recall: $\langle \mathbf{a}, \mathbf{b} \rangle = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta)$, where θ is the angle between the vectors.

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

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



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

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

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

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

Gradient descent (GD) finds stationary points by iterating:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_{t-1}) \,. \tag{11}$$

This is the basic form of gradient descent (GD, also known as **first-order** optimization): steepest local directions are not necessarily the *fastest* in general and we can accelerate this iteration in many ways.

Derivatives and gradients

Stochastic gradient descent

Consider the steps needed for a single iteration of GD in our specific optimization problem:

1. Computing the output of the model $f(\mathbf{x})$ on *all* examples;
2. Computing the gradient of the cost function with respect to the parameters.

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

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

The key observation is that the training problem in NNs is an average over the full dataset ($\boldsymbol{\theta}$ is the set of weights):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_{\boldsymbol{\theta}}(x_i), y_i)\,. \tag{12}$$

To limit the complexity, we can use only a **mini-batch** $\mathcal{B}_t$ of B examples randomly sampled from the full dataset:

$$\mathcal{L}(\boldsymbol{\theta}) \approx \widetilde{\mathcal{L}}_t(\boldsymbol{\theta}) = \frac{1}{B} \sum_{i \in \mathcal{B}_t} \ell(f_{\boldsymbol{\theta}}(x_i), y_i)\,. \tag{13}$$

The mini-batch gradient is a (unbiased, noisy) estimate of the full gradient.

The previous algorithm is called **stochastic gradient descent** (SGD).

The computational cost of one SGD iteration grows with B (the **batch size**), but does not depend on the full dataset size N once B is fixed.

Under standard regularity assumptions and a suitable learning-rate schedule, the noisy mini-batch gradients can still drive the expected gradient norm towards zero.

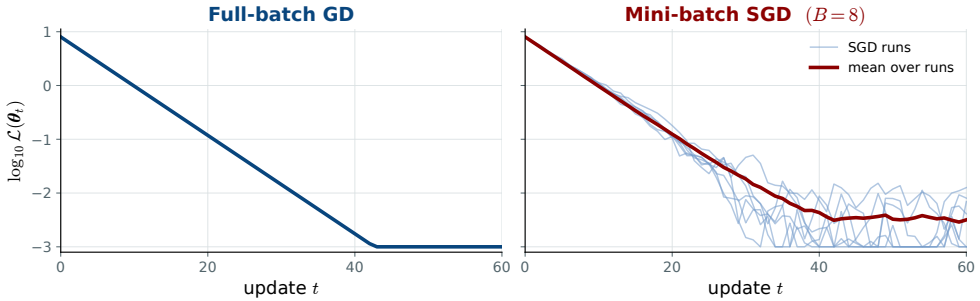


Figure 4: Comparison of standard GD ("full-batch") and SGD: mini-batch sampling introduces noise in each update, but the loss still decreases on average.

Instead of randomly sampling batches from the training dataset at each iteration, we generally apply the following procedure:

1. Shuffle the full dataset;
2. Split the dataset into blocks of B elements and process them sequentially, batch-by-batch;
3. After the last block, return to point (1) and iterate.

A full pass over the dataset is called an **epoch**. This is efficient because the majority of the time we deal with data stored *contiguously* in memory. In practice, even the shuffling operation can be approximated.

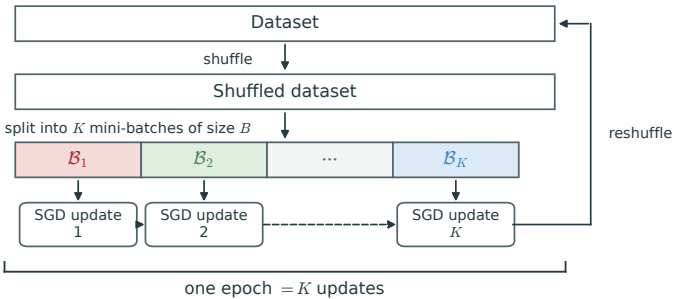


Figure 5: Dividing the optimization process into epochs is also helpful to define metrics and stopping criteria.

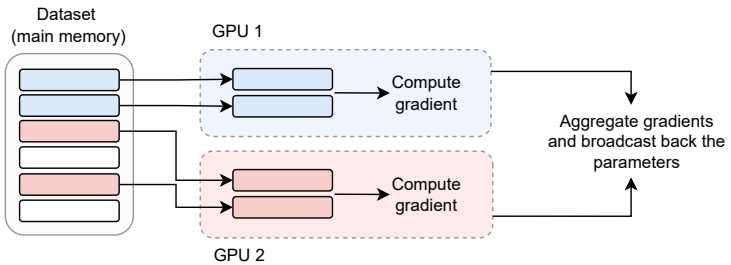


Figure 6: A simple form of distributed stochastic optimization: we process one mini-batch per available machine or GPU (by replicating the weights on each of them) and sum or average the corresponding gradients before broadcasting back the result (which is valid due to the linearity of the gradient operation). This requires a synchronization mechanism across the machines or the GPUs.

Derivatives and gradients

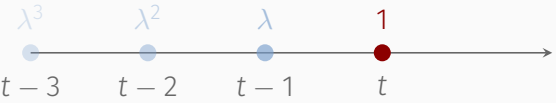
Accelerating gradient descent (optional)

Steepest-descent directions are local. On poorly conditioned objectives they can zig-zag, and noisy gradients make the path more erratic. **Momentum** smooths the path by retaining part of the previous displacement:

$$\mathbf{u}_t = \underbrace{-\eta_t \nabla \widetilde{\mathcal{L}}_t(\boldsymbol{\theta}_{t-1})}_{\text{new gradient}} + \underbrace{\lambda \mathbf{u}_{t-1}}_{\text{retained displacement}},$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \mathbf{u}_t\,, \qquad \mathbf{u}_0 = \mathbf{0}, \quad 0 \le \lambda < 1\,.$$

Past contributions are exponentially damped: a displacement from n iterations ago is weighted by λ^{n-1}.



Let $\mathbf{g}_t = \nabla \widetilde{\mathcal{L}}_t(\boldsymbol{\theta}_{t-1})$. **Adam** maintains two exponential averages:

first moment (direction)

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

second moment (coordinate scale)

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

bias correction (initialization at 0)

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \qquad \widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t} + \varepsilon}$$

We need to introduce two additional hyper-parameters ($\beta_1, \beta_2 \in [0,1)$), but the algorithm is generally robust to their defaults (see **Muon** for more recent research).

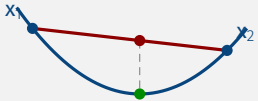
Derivatives and gradients

Convexity and convergence

Convexity plays a pivotal role in optimization. If a function is convex, its optimization is easier with respect to a non-convex one.

f is said to be convex if for any $\lambda \in [0, 1]$:

$$f\Big((1-\lambda)\mathbf{x}_1 + \lambda\mathbf{x}_2\Big) \leq (1-\lambda)f(\mathbf{x}_1) + \lambda f(\mathbf{x}_2)\,. \tag{14}$$



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 $\boldsymbol{\theta}^*$.

More structure in $\mathcal{L}$ $\Longrightarrow$ Stronger conclusion

Generic non-convex	First-order methods generally guarantee only **stationarity**.
Convex	Every stationary point is a **global optimum**.
Strictly convex	If an optimum exists, it is **unique**.

Without additional structure, globally optimizing a general non-convex objective can be **NP-hard**; practical first-order methods therefore usually target stationary points instead.

Chapter 2 from the book, along with the suggested exercises.