Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 2: Preliminaries

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Preliminaries

Tensors and matrices

For the purpose of this course, a **tensor** is an n-dimensional array of elements of the *same type*.[a]

[a] Sidenote: in ML, the word tensor is used informally; 'real' tensors are used to describe multilinear relations between spaces.

Given a tensor X, it can be *indexed* using a tuple of n numbers:

$\underset{(h,w,c)}{X}$	3-dimensional tensor of shape (h, w, c)
$X_{i,j,k}$	element in position (i, j, k) (sometimes X_{ijk})
$[X]_{i,j,k}$	alternative notation for indexing

The argument of the last notation can be an expression, e.g., $[X+Y]_{i,j,k}$.

Tensors are the default data structure in any deep learning framework:

```python
import tensorflow as tf
X = tf.random.normal((64, 64, 3)) # `Random' 3-dimensional tensor
```

NumPy-like indexing is pervasive (with 0-based indexing):

```python
X[0, 0, 0] # Full indexing
X[0] # Partial indexing (slice of the original tensor)
X[:, 0] # Partial indexing on the second axis
```

For homogeneity, we use a similar slicing notation in math:

$X_{:,i}$ 2-dimensional tensor of shape (h, c)

0-dimensional tensors are called **scalars**. Most scalars in this course are real-valued, which can be manipulated in a number of ways:

$$+, -, *, \sin, \cos, \sqrt{}, \exp, |\cdot|, \ldots$$

1-dimensional tensors are **vectors** and are assumed to be *column* vectors (and are written in boldface):

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_m \end{pmatrix} , \quad \mathbf{x}^\top = \begin{pmatrix} x_1 & x_2 & \ldots & x_m \end{pmatrix}$$

Real-valued vectors can be linearly combined to give new vectors:

$$\mathbf{z} = a\mathbf{x} + b\mathbf{y}\,, \;\; [\mathbf{z}]_i = ax_i + by_i\,.$$

The *length* of a vector is given by its Euclidean norm (ℓ_2 norm):

$$\|\mathbf{x}\|^2 = \sum_i x_i^2\,. \tag{1}$$

The (standard) **inner product** between two vectors is:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i x_i y_i = \mathbf{x}^\top \mathbf{y}\,.$$

Geometrically, the inner product can be used to compute the angle θ between the two vectors (**cosine similarity**):

$$\cos(\theta) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|} \,. \tag{2}$$

For two **orthogonal** vectors, $\langle \mathbf{x}, \mathbf{y} \rangle = 0$. Otherwise, the cosine similarity oscillates between -1 (opposite vectors) and $+1$ (aligned vectors).

Euclidean distance can also be defined in terms of inner products:

$$\|\mathbf{x} - \mathbf{y}\|_2^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle \,.$$

2-dimensional tensors are **matrices**:

$$\mathbf{X} = \underbrace{\begin{bmatrix} X_{1,1} & \cdots & X_{1,n} \\ \vdots & \ddots & \vdots \\ X_{m,1} & \cdots & X_{m,n} \end{bmatrix}}_{n \text{ columns},\ m \text{ rows}}$$

Matrices can also be interpreted as a **stack** (**batch**) of vectors:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_m \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{X}_{:,1} & \dots & \mathbf{X}_{:,n} \end{bmatrix}$$

Like vectors, matrices can be linearly combined: $\mathbf{Z} = a\mathbf{X} + b\mathbf{Y}$.

Geometrically, they represent a linear map between two vector spaces:

$$\underset{(m)}{\mathbf{b}} = \underset{(m,n)}{\mathbf{W}} \underset{(n)}{\mathbf{a}} .$$

Matrix multiplication between $\underset{(a,b)}{\mathbf{X}}$ and $\underset{(b,c)}{\mathbf{Y}}$ is defined as:

$$[\mathbf{XY}]_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_{:,j} \rangle = \sum_z X_{iz} Y_{zj} \in \mathbb{R}^{a \times c} .$$

Multiplication is akin to function composition: $f(\mathbf{x}) = (\mathbf{AB})(\mathbf{x})$.

In many cases, writing a batch of operations in terms of matrix multiplications results in an easy and fast implementation (**vectorizing**), e.g.:

$$\mathbf{X}\mathbf{W} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_m \end{bmatrix} \mathbf{W} = \begin{bmatrix} \mathbf{X}_1\mathbf{W} \\ \vdots \\ \mathbf{X}_m\mathbf{W} \end{bmatrix} \tag{3}$$

Using a linear algebra library, we can compute m vector-matrix products in parallel with a single efficient instruction. Compilers (e.g., `tf.function`) can automatically vectorize certain operations.

Another example: $\mathbf{X}\mathbf{X}^\top$ computes all inner products of the form $\langle \mathbf{X}_i, \mathbf{X}_j \rangle$ simultaneously.

A 3-dimensional tensor $X_{(a,b,c)}$ can also be seen as a stack of a matrices of shape (b, c).

Most operations in TensorFlow (and other deep learning frameworks) are optimized for batching operations across leading dimensions, e.g.:

```
X = tf.random.normal((3, 4, 5))
Y = tf.random.normal((3, 5, 10))
Z = tf.linalg.matmul(X, Y) # Result has shape (3, 4, 10)
```

Some scalar operations extend to the matrix case by generalizing their definition, e.g., the **matrix exponential** for squared matrices:

$$\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \,.$$

More commonly, we are interested in applying a scalar operation *element-wise*, i.e., on each element independently:

$$[\exp(\mathbf{X})]_{ij} = \exp(X_{ij}) \tag{4}$$

```
X = tf.math.exp(X) # Element-wise exponential
X = tf.linalg.expm(X) # Matrix exponential
```

Matrix multiplication can also be performed element-wise, in which case we call it the **Hadamard product**:

$$[\mathbf{X} \odot \mathbf{Y}]_{ij} = X_{ij} Y_{ij} \,.$$

Finally, sometimes we write operations that look inconsistent:

$$\underset{(n,m)}{\mathbf{Y}} = \underset{(n,m)}{\mathbf{X}} + \underset{(m)}{\mathbf{a}} \tag{5}$$

This is interpreted as $\mathbf{Y}_i = \mathbf{X}_i + \mathbf{a}$ (**broadcasting**), as popularized by NumPy.

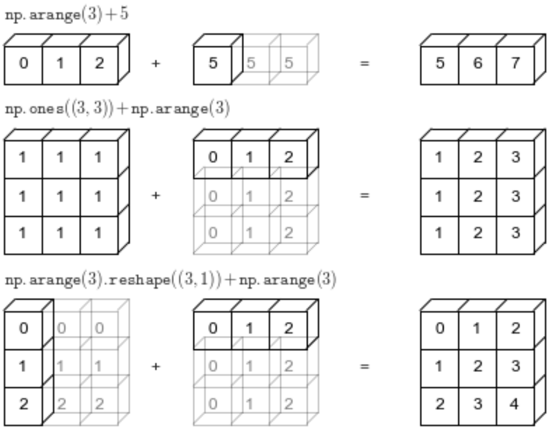

Figure 1: Different examples of broadcasting in NumPy (TF and other frameworks follow similar rules).

Consider the following snippet:

```
a = tf.random.normal((3,))
b = tf.random.normal((3,))

# Sum of errors squared
e = tf.reduce_sum((a - b)**2)

# *WRONG* sum of errors squared
e = tf.reduce_sum((tf.reshape(a, (3,1))
                  - tf.reshape(b, (1,3)))**2)
```

Because of broadcasting, objects of shape (3,), (3,1), or (1,3) are fundamentally different.

Many times, we use **reduction** operations across one or more axes, e.g.:

$$\underset{(b,c)}{\mathbf{H}} = \sum_i \underset{(a,b,c)}{[X]_i} .$$

For example, a generalized dot product between two 3-dimensional tensors X_1 and X_2 can be written as:

$$y = \sum_{i,j,k} [X_1 \odot X_2]_{i,j,k} . \tag{6}$$

For vectors and matrices, we can also write reductions using products:

$$y = \sum_i [\mathbf{x}]_i = \langle \mathbf{x}, \mathbf{1} \rangle . \tag{7}$$

A simplified **Einstein notation** is gaining traction, where repeated indexes are summed over:

$$Z_{ij} = X_{ik}Y_{kj} = \sum_k X_{ik}Y_{kj}$$

And indices not appearing on the left are implicitly summed:

$$z = x_i = \sum_i x_i \tag{8}$$

Einstein notation is implemented in most frameworks with `einsum`, using a string that follows the summing convention:

```
# This is batched matrix multiplication
X = tf.random.normal(shape=[7,5,3])
Y = tf.random.normal(shape=[7,3,2])
Z = tf.einsum('bij,bjk->bik', X, Y)
```

See `https://www.tensorflow.org/api_docs/python/tf/einsum` for more examples and `https://rockt.github.io/2018/04/30/einsum` for a nice introduction.

See einops for a very popular extension of einsum with more functionalities (e.g., patching and more general reductions).

Preliminaries

Derivatives and gradients

Most of this course is funded upon the notion of derivative.

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

$$\partial f(x) = \frac{\partial}{\partial x} f(x) = f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{9}$$

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 $f(x)$.

Derivative of a polynomial:

$$\partial\left[x^p\right] = p x^{p-1}\,.$$

Derivative of exponentials and logarithms:

$$\partial\left[\exp(x)\right] = \exp(x)\,,$$
$$\partial\left[\log(x)\right] = \frac{1}{x}\,.$$

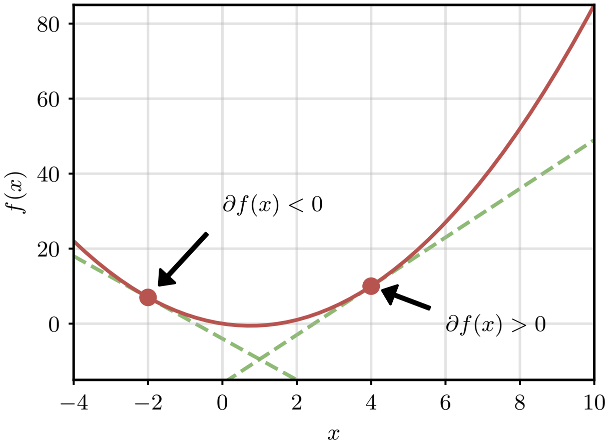

Figure 2: 1D function ($f(x) = x^2 - 1.5x$), showing the derivative at two different locations.

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:

$$[\partial f(\mathbf{x})]_i = \frac{\partial y}{\partial \mathbf{x}} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \tag{10}$$

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:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}, \tag{11}$$

It is easy to prove that:

$$\mathrm{D}_{\mathbf{v}} f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle . \tag{12}$$

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 defined as:

$$\partial f(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix} . \tag{13}$$

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

Derivative of the inner product:

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

Derivative of a linear map:

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

Derivative of a norm:

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

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 $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $g : \mathbb{R}^o \rightarrow \mathbb{R}^m$:

$$\underbrace{\partial [f \circ g]}_{(n,o)} = \underbrace{\partial f}_{(n,m)} \circ \underbrace{\partial g}_{(m,o)} . \tag{14}$$

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

Given a function $f(\mathbf{x}_0)$ evaluated at $\mathbf{x}_0$, then the function:

Slope of the line

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

Displacement from $\mathbf{x}_0$

is the best **linear approximation** of f around $\mathbf{x}_0$ (**Taylor's theorem**). Better approximations can be constructed from higher-order derivatives, but this is enough for building effective optimization algorithms.

A simple example of using the linear approximation:

```python
# Function
f = lambda x: x**2 - 1.5*x

# Derivative (manual)
df = lambda x: 2*x - 1.5

# Linearization at 0.5
x = 0.5
f_linearized = lambda h: f(x) + df(x)*(h - x)

print(f(x + 0.01)) # -0.5049
print(f_linearized(x + 0.01)) # -0.5050
```

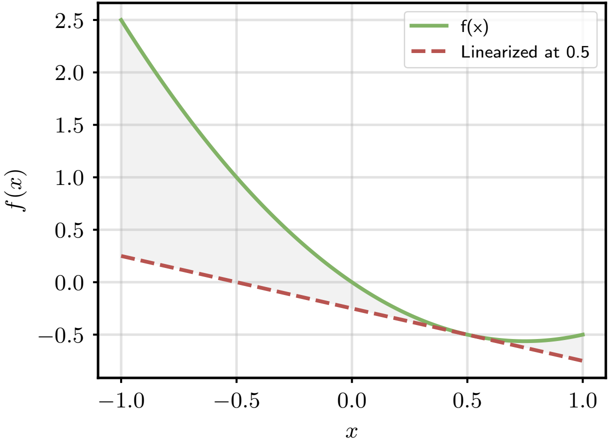

Figure 3: 1D function ($f(x) = x^2 - 1.5x$), linearized at 0.5.

Preliminaries

Numerical optimization

We use gradients to solve generic problems of the form:

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathbb{R}^d} f(\mathbf{x}) \tag{15}$$

This is called **unconstrained optimization** because the domain is $\mathbb{R}^d$. Note that maximizing/minimizing are equivalent in the sense that:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathbb{R}^d} f(\mathbf{x}) = \arg\min_{\mathbf{x} \in \mathbb{R}^d} -f(\mathbf{x}) \tag{16}$$

Also, $f(\mathbf{x}) \in \mathbb{R}$ (**single objective optimization**).

A point $\mathbf{x}$ such that $f(\mathbf{x}) \leq f(\mathbf{x}') \;\; \forall\, \mathbf{x}' \in \mathbb{R}^d$ is called a **global minimum**. If instead (less restrictive):

$$f(\mathbf{x}) \leq f(\mathbf{x}') \;\; \forall\, \mathbf{x}' \in \left\{\mathbf{x}' : \|\mathbf{x}' - \mathbf{x}\|^2 < \varepsilon\right\} \tag{17}$$

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

If $\nabla f(\mathbf{x}) = 0$, $\mathbf{x}$ is called a **stationary point**. Stationary points can be minima, maxima, or inflection points (aka **saddle points**).

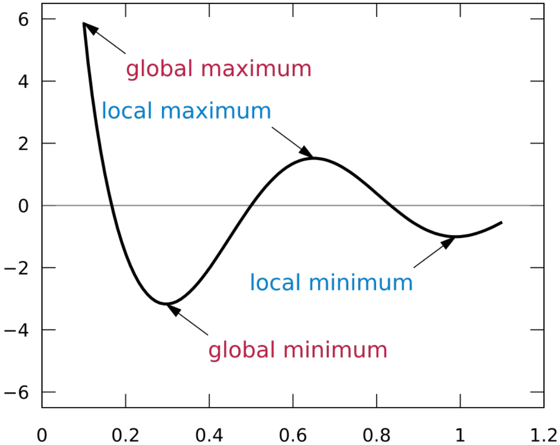

Figure 4: With no additional information, stationary points can be **minima**, **maxima**, and can be **local** or **global** (*Wikimedia, KSmrq*).

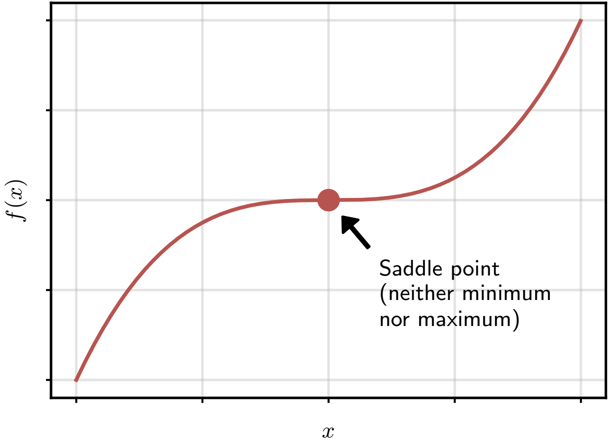

Figure 5: Stationary points can also be **saddle points**, either decreasing or increasing in different directions.

Given a randomly initialized $\mathbf{x}_0$, consider the following iteration:

Guess at iteration t

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \mathbf{p}_t \tag{18}$$

Displacement at iteration t

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

Without lack of generality, we restrict to unit directions ($\|\mathbf{p}_t\| = 1$). The rate of change is given by the directional derivative:

$$\mathrm{D}_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \underbrace{\|\mathbf{p}_t\|}_{=1} \cos(\theta) = \|\nabla f(\mathbf{x}_{t-1})\| \cos(\theta).$$

The above quantity is minimized when $\cos(\theta) = -1$, which happens if $\theta = \pi$, i.e., $\mathbf{p}_t = -\nabla f(\mathbf{x}_{t-1})$. This is the **steepest descent direction**. In general, anything with $\cos(\theta) < 0$ is a descent direction.

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

Gradient descent (GD) finds stationary points by iterating:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})\,. \tag{19}$$

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 $\lambda \in [0, 1]$:

$$f\Big((1-\lambda)\mathbf{x}_1 + \lambda\mathbf{x}_2\Big) \leq (1-\lambda)f(\mathbf{x}_1) + \lambda f(\mathbf{x}_2)\,. \tag{20}$$

If the equality is strict, we say that f is **strictly convex**.

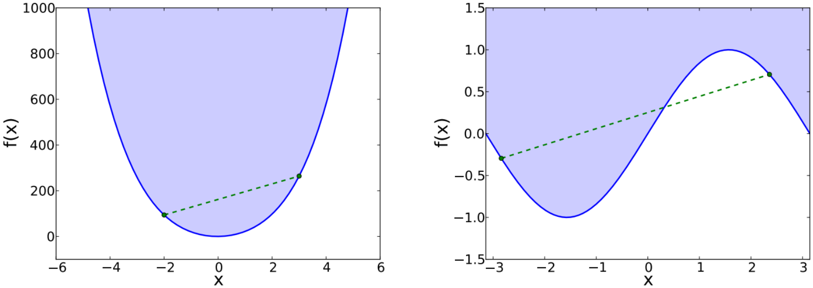

Figure 6: Left: an example of convex function. Right: an example of non-convex function. Taken from "An Introduction to Machine Learning" by Smola and Vishwanathan [unpublished].

Consider a generic $f(\mathbf{x})$, and assume GD converges to a point $\mathbf{x}^*$. Then:

- **Generic non-convex** $f(\mathbf{x})$: The point $\mathbf{x}^*$ is **stationary**.
- **Convex** $f(\mathbf{x})$: The point $\mathbf{x}^*$ is a **global optimum**.
- **Strictly convex** $f(\mathbf{x})$: The point $\mathbf{x}^*$ is **the only** global optimum.

For a non-convex function, unless additional assumptions are made on $f(\mathbf{x})$, this result cannot be improved. Finding a global optimum becomes an **NP-hard** problem, akin to evaluating the entire domain of the function.

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