Principles of Deep Learning

Mattia Bergomi and Pietro Vertechi

mattiagbergomi@gmail.com, pietro.vertechi@protonmail.com

Table of contents

Part 1

- Minimal requirements, programming basics, and good practices
- Global Information Tracker (GIT): how to and setup

Part 2

- A (very brief) historical perspective.
- A hands-on analysis of the mathematical principles underlying deep learning.
- How to build a complete data analysis pipeline.
- From equations to code.

Python setup

- Python installation
- PIP
- Virtual environments
- Project blueprint

Good practices

- Single-responsibility principle

```
def function_11(a_in, opt_in):
  return a_in if opt_in is None else 0

...

def function1(a_in, b_in, opt_in = None):
  c_t = function_11(a_in, opt_in)
  return c_t + function_12(a_in, b_in)
```

- Unit test

```
def test_function_11():
  function_11(_, None) == 0
```

- Error management

```
class ACustomException(Exception):
  def __init__(self, my_arg):
    self.message = "A meaningful comment)
    super().__init__(self.message)
```

Good practices

- Write clear documentation

```
ss InvalidTravelType(Exception):
 """Exception raised for errors in travel type setting.

 Attributes:
     travel_type (int) -- input travel_type
 """
 def __init__(self, travel_type):
     self.message = "Travel type {} is invalid. Choose a value in
     super().__init__(self.message)
```

- Seopcify types and ouputs

```
def __init__(self, travel_type: int) -> None:
    self.message = "Travel type {} is invalid.
    "Choose a value in {}".format(travel_type, TRAVEL_TYPES)
    super().__init__(self.message)
```

- Provide examples and sample calls

```
if __name__ == "__main__":
  ...
```

GIT setup

- Install GIT
- Create an account on GitHub
- Clone a repository

```
git clone
```

- Create a repository
- Push a commit

The overarching purpose of deep learning

The primary objective of deep learning methods is to find a function that maps an input to an output in an "optimal" way (i.e., minimizing some cost function).

The overarching purpose of deep learning

The primary objective of deep learning methods is to find a function that maps an input to an output in an "optimal" way (i.e., minimizing some cost function).

Deep learning methods are particularly useful when the structure of the optimal solution is not known, but there is an ample dataset of "correct" examples.

The overarching purpose of deep learning

The primary objective of deep learning methods is to find a function that maps an input to an output in an "optimal" way (i.e., minimizing some cost function).

Deep learning methods are particularly useful when the structure of the optimal solution is not known, but there is an ample dataset of "correct" examples.

Example problems

- Classification: What category does a given input belong to?
- Regression: What is the expected value of an unknown variable given a set of predictors?
- Control theory: What are the optimal control parameters given sensor data?
- ...

The overarching strategy of deep learning

Despite the wide variety of applications of deep learning, the general strategy is always the same.

The overarching strategy of deep learning

Despite the wide variety of applications of deep learning, the general strategy is always the same.

- Choose a differentiable parametric function $\hat{y} = f(x, p)$, where x is the input and p the parameters.

The overarching strategy of deep learning

Despite the wide variety of applications of deep learning, the general strategy is always the same.

- Choose a differentiable parametric function $\hat{y} = f(x, p)$, where x is the input and p the parameters.
- Define a differentiable loss function $\mathcal{L}(\hat{y})$ and minimize it with respect to p (using its derivatives).

The overarching strategy of deep learning

Despite the wide variety of applications of deep learning, the general strategy is always the same.

- Choose a differentiable parametric function $\hat{y} = f(x, p)$, where x is the input and p the parameters.
- Define a differentiable loss function $\mathcal{L}(\hat{y})$ and minimize it with respect to p (using its derivatives).

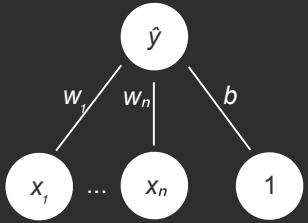
The key difficulty is the choice of the parametric function f.

Early attempts

- The first neurally-inspired artificial decision-maker was Rosenblatt's *perceptron* (1958).
- A single unit applies a squashing, non-decreasing, nonlinear transformation to a weighted sum of input units.

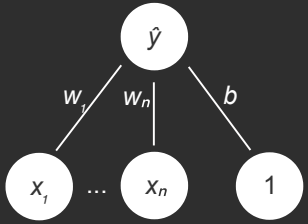
Early attempts

- The first neurally-inspired artificial decision-maker was Rosenblatt's *perceptron* (1958).
- A single unit applies a squashing, non-decreasing, nonlinear transformation to a weighted sum of input units.



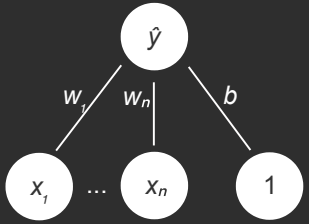
Early attempts

- The first neurally-inspired artificial decision-maker was Rosenblatt's *perceptron* (1958).
- A single unit applies a squashing, non-decreasing, nonlinear transformation to a weighted sum of input units.
- In matrix notation, $\hat{y} = \sigma(Wx + b)$.



Early attempts

- The first neurally-inspired artificial decision-maker was Rosenblatt's *perceptron* (1958).
- A single unit applies a squashing, non-decreasing, nonlinear transformation to a weighted sum of input units.
- In matrix notation, $\hat{y} = \sigma(Wx + b)$.
- Limitations of the perceptron were discovered quickly.
- For instance, this network cannot learn to implement the XOR function for binary inputs. That is to say, for inputs taking values either 0 or 1, return 1 if exactly one of the inputs is 1, return 0 otherwise. **Exercise.** Why?



Solution to XOR problem

Let us assume that there exists a squashing, non-decreasing, nonlinear transformation σ, as well as weights $\{w_i\}_{i=1}^N$ and bias b, such that the associated perceptron implements the XOR function on boolean inputs.

Solution to XOR problem

Let us assume that there exists a squashing, non-decreasing, nonlinear transformation σ, as well as weights $\{w_i\}_{i=1}^N$ and bias b, such that the associated perceptron implements the XOR function on boolean inputs.

We can also assume $N = 2$, as otherwise we could simply set $x_i = 0$ for $i > 2$.

Solution to XOR problem

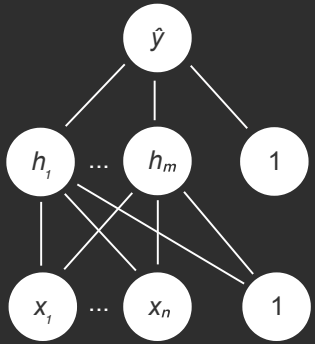
Let us assume that there exists a squashing, non-decreasing, nonlinear transformation σ, as well as weights $\{w_i\}_{i=1}^N$ and bias b, such that the associated perceptron implements the XOR function on boolean inputs.

We can also assume $N = 2$, as otherwise we could simply set $x_i = 0$ for $i > 2$.

$1 = \sigma(w_1 + b) > \sigma(b) = 0$ and $1 = \sigma(w_2 + b) > \sigma(w_1 + w_2 + b) = 0$, hence $w_1 > 0$ and $w_1 < 0$, which is a contradiction.

The multilayer perceptron

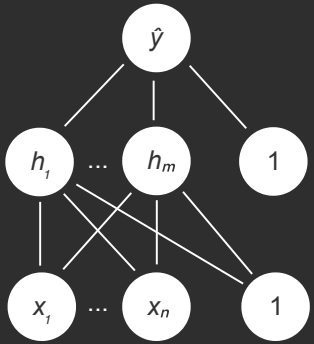
The proposed solution was to "stack" perceptrons:



It can be shown that a single hidden layer perceptron with a linear readout can approximate any function, given sufficiently many hidden units (Hornik, 1991).

The multilayer perceptron

The proposed solution was to "stack" perceptrons:



In formulas:

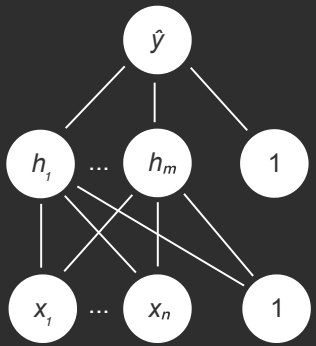
$$h = \sigma(W_1 x + b_1)$$

$$\hat{y} = W_2 h + b_2$$

It can be shown that a single hidden layer perceptron with a linear readout can approximate any function, given sufficiently many hidden units (Hornik, 1991).

The multilayer perceptron

The proposed solution was to "stack" perceptrons:



It can be shown that a single hidden layer perceptron with a linear readout can approximate any function, given sufficiently many hidden units (Hornik, 1991).

In formulas:

$$h = \sigma(W_1 x + b_1)$$

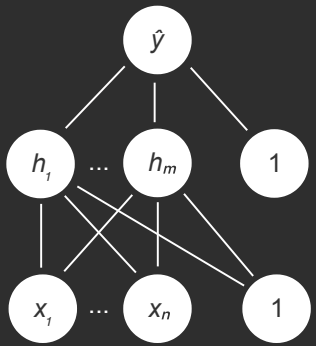
$$\hat{y} = W_2 h + b_2$$

Exercise. How can one compute XOR with this architecture? For simplicity, take as squashing nonlinearity the step function

$$\sigma(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The multilayer perceptron

The proposed solution was to "stack" perceptrons:



It can be shown that a single hidden layer perceptron with a linear readout can approximate any function, given sufficiently many hidden units (Hornik, 1991).

In formulas:

$$h = \sigma(W_1 x + b_1)$$

$$\hat{y} = W_2 h + b_2$$

Exercise. How can one compute XOR with this architecture? For simplicity, take as squashing nonlinearity the step function

$$\sigma(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The matrices W_1, W_2 and the vectors b_1, b_2 are our parameters. How can we optimize $\mathcal{L}(\hat{y})$ as a function of the parameters?

A concrete example

Let x, y be vectors. Let us define

$$\tilde{h} = W_1 x + b_1$$
$$h = \sigma(\tilde{h})$$
$$\hat{y} = W_2 h + b_2$$

A concrete example

Let x, y be vectors. Let us define

$$\begin{aligned}\tilde{h} &= W_1 x + b_1 \\ h &= \sigma(\tilde{h}) \\ \hat{y} &= W_2 h + b_2\end{aligned}$$

We wish to find parameters such that $\hat{y}$ is as close as possible to y.

We can consider a simple square norm loss, that is to say, $\mathcal{L} = \|y - \hat{y}\|^2$.

A concrete example

Let x, y be vectors. Let us define

$$\begin{aligned}\tilde{h} &= W_1 x + b_1 \\ h &= \sigma(\tilde{h}) \\ \hat{y} &= W_2 h + b_2\end{aligned}$$

We wish to find parameters such that $\hat{y}$ is as close as possible to y.

We can consider a simple square norm loss, that is to say, $\mathcal{L} = \|y - \hat{y}\|^2$.

Computing derivatives is a simple, but laborious, application of the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)^*$$

A concrete example

Let x, y be vectors. Let us define

$$\begin{aligned}\tilde{h} &= W_1 x + b_1 \\ h &= \sigma(\tilde{h}) \\ \hat{y} &= W_2 h + b_2\end{aligned}$$

We wish to find parameters such that $\hat{y}$ is as close as possible to y.

We can consider a simple square norm loss, that is to say, $\mathcal{L} = \|y - \hat{y}\|^2$.

Computing derivatives is a simple, but laborious, application of the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)^*$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = h \frac{\partial \mathcal{L}}{\partial \hat{y}} \\ \frac{\partial \mathcal{L}}{\partial b_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \\ \frac{\partial \mathcal{L}}{\partial h} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} = \frac{\partial \mathcal{L}}{\partial \hat{y}} W_2\end{aligned}$$

A concrete example

Let x, y be vectors. Let us define

$$\begin{aligned}\tilde{h} &= W_1 x + b_1 \\ h &= \sigma(\tilde{h}) \\ \hat{y} &= W_2 h + b_2\end{aligned}$$

We wish to find parameters such that $\hat{y}$ is as close as possible to y.

We can consider a simple square norm loss, that is to say, $\mathcal{L} = \|y - \hat{y}\|^2$.

Computing derivatives is a simple, but laborious, application of the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)^*$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = h \frac{\partial \mathcal{L}}{\partial \hat{y}} \\ \frac{\partial \mathcal{L}}{\partial b_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \\ \frac{\partial \mathcal{L}}{\partial h} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} = \frac{\partial \mathcal{L}}{\partial \hat{y}} W_2\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial \tilde{h}} = \frac{\partial \mathcal{L}}{\partial h} \frac{\partial h}{\partial \tilde{h}} = \frac{\partial \mathcal{L}}{\partial h} \odot \sigma'(\tilde{h})^*$$

A concrete example

Let x, y be vectors. Let us define

$$\begin{aligned}\tilde{h} &= W_1 x + b_1 \\ h &= \sigma(\tilde{h}) \\ \hat{y} &= W_2 h + b_2\end{aligned}$$

We wish to find parameters such that $\hat{y}$ is as close as possible to y.

We can consider a simple square norm loss, that is to say, $\mathcal{L} = \|y - \hat{y}\|^2$.

Computing derivatives is a simple, but laborious, application of the chain rule.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)^*$$

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = h \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

$$\frac{\partial \mathcal{L}}{\partial h} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} = \frac{\partial \mathcal{L}}{\partial \hat{y}} W_2$$

$$\frac{\partial \mathcal{L}}{\partial \tilde{h}} = \frac{\partial \mathcal{L}}{\partial h} \frac{\partial h}{\partial \tilde{h}} = \frac{\partial \mathcal{L}}{\partial h} \odot \sigma'(\tilde{h})^*$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \tilde{h}} \frac{\partial \tilde{h}}{\partial W_1} = x \frac{\partial \mathcal{L}}{\partial \tilde{h}}$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial \tilde{h}} \frac{\partial \tilde{h}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial \tilde{h}}$$

A general recipe: backpropagation

Backpropagation is the generalization of the above technique.

- During the forward pass, we compute *and store* all intermediate values from x to $\hat{y}$.
- For each one of these values, we compute the derivatives of the loss with respect to it, in *reversed order*.
- In the practical session, we will implement it.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.

- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u}$.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.
- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u}$.
- Equivalently, $\left(\frac{\partial \mathcal{L}}{\partial u}\right)^* = \left(\frac{\partial v}{\partial u}\right)^* \left(\frac{\partial \mathcal{L}}{\partial v}\right)^*$.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.
- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u}$.
- Equivalently, $\left(\frac{\partial \mathcal{L}}{\partial u}\right)^* = \left(\frac{\partial v}{\partial u}\right)^* \left(\frac{\partial \mathcal{L}}{\partial v}\right)^*$.
- When computing the *primitive* g, we also compute and store the adjoint operator $\left(\frac{\partial u}{\partial v}\right)^*$.

Automatic differentiation

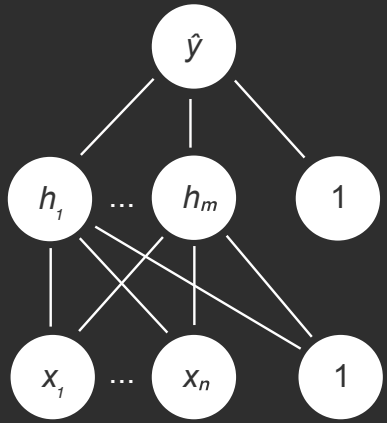
- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.
- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u}$.
- Equivalently, $\left(\frac{\partial \mathcal{L}}{\partial u}\right)^* = \left(\frac{\partial v}{\partial u}\right)^* \left(\frac{\partial \mathcal{L}}{\partial v}\right)^*$.
- When computing the *primitive* g, we also compute and store the adjoint operator $\left(\frac{\partial u}{\partial v}\right)^*$.
- During the reverse pass, we compose all adjoint operators.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.
- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial u}$.
- Equivalently, $\left(\frac{\partial \mathcal{L}}{\partial u}\right)^* = \left(\frac{\partial v}{\partial u}\right)^* \left(\frac{\partial \mathcal{L}}{\partial v}\right)^*$.
- When computing the *primitive* g, we also compute and store the adjoint operator $\left(\frac{\partial u}{\partial v}\right)^*$.
- During the reverse pass, we compose all adjoint operators.
- Automatic differentiation libraries perform this procedure automatically on a *directed acyclic graph* representing our computation.

Automatic differentiation

- In the above example, given a *primitive* $v = g(u)$ (matrix multiplication, addition, or pointwise nonlinearity), we wish to compute $\frac{\partial \mathcal{L}}{\partial u}$ as a function of $\frac{\partial \mathcal{L}}{\partial v}$.
- This is done by composing differentials, i.e. $\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v}\frac{\partial v}{\partial u}$.
- Equivalently, $\left(\frac{\partial \mathcal{L}}{\partial u}\right)^* = \left(\frac{\partial v}{\partial u}\right)^* \left(\frac{\partial \mathcal{L}}{\partial v}\right)^*$.
- When computing the *primitive* g, we also compute and store the adjoint operator $\left(\frac{\partial u}{\partial v}\right)^*$.
- During the reverse pass, we compose all adjoint operators.
- Automatic differentiation libraries perform this procedure automatically on a *directed acyclic graph* representing our computation.



The road so far

- We have constructed a parametric function $\hat{y} = f(x, p)$: the *multilayer perceptron*.
- The parameters p contain all the weight matrices W and bias vectors b.
- Given a loss $\mathcal{L}(y, \hat{y})$ we can compute its derivatives with respect to p via backpropagation.

Stochastic (i.e., batched) gradient descent

- Let us now consider a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.
- We wish to minimize $\mathcal{L}(y_i, \hat{y}_i)$, across all $(x_i, y_i) \in \mathcal{D}$, where $\hat{y}_i = f(x_i, p)$.
- Intuitive approach: compute $\frac{\partial \mathcal{L}}{\partial p}$ directly and apply gradient descent.

Stochastic (i.e., batched) gradient descent

- Let us now consider a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.
- We wish to minimize $\mathcal{L}(y_i, \hat{y}_i)$, across all $(x_i, y_i) \in \mathcal{D}$, where $\hat{y}_i = f(x_i, p)$.
- Intuitive approach: compute $\frac{\partial \mathcal{L}}{\partial p}$ directly and apply gradient descent.

Unfortunately, computing $\frac{\partial \mathcal{L}}{\partial p}$ requires averaging over the whole dataset, which can be very expensive.

A more practical approach (batched optimization) is to the following.

- Compute $\frac{\partial \mathcal{L}}{\partial p}$ for a given subset (batch) of data.
- Apply a step of gradient descent.
- Select a novel batch of data and repeat the procedure.

Stochastic (i.e., batched) gradient descent

- Let us now consider a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$.
- We wish to minimize $\mathcal{L}(y_i, \hat{y}_i)$, across all $(x_i, y_i) \in \mathcal{D}$, where $\hat{y}_i = f(x_i, p)$.
- Intuitive approach: compute $\frac{\partial \mathcal{L}}{\partial p}$ directly and apply gradient descent.

Unfortunately, computing $\frac{\partial \mathcal{L}}{\partial p}$ requires averaging over the whole dataset, which can be very expensive.

A more practical approach (batched optimization) is to the following.

- Compute $\frac{\partial \mathcal{L}}{\partial p}$ for a given subset (batch) of data.
- Apply a step of gradient descent.
- Select a novel batch of data and repeat the procedure.

Remark

In the above procedure, gradient descent on a batch of data is not the only option.

Many other batched optimizers can be used.

The overall procedure

- Start with a dataset of samples $\{x_i\}_{i=1}^N$ and correct outcomes $\{y_i\}_{i=1}^N$.

The overall procedure

- Start with a dataset of samples $\{x_i\}_{i=1}^N$ and correct outcomes $\{y_i\}_{i=1}^N$.
- Define a parametric function $\hat{y} = f(x, p)$ (multilayer perceptron, for today).

The overall procedure

- Start with a dataset of samples $\{x_i\}_{i=1}^N$ and correct outcomes $\{y_i\}_{i=1}^N$.
- Define a parametric function $\hat{y} = f(x, p)$ (multilayer perceptron, for today).
- Define a loss $\mathcal{L}(y, \hat{y})$.

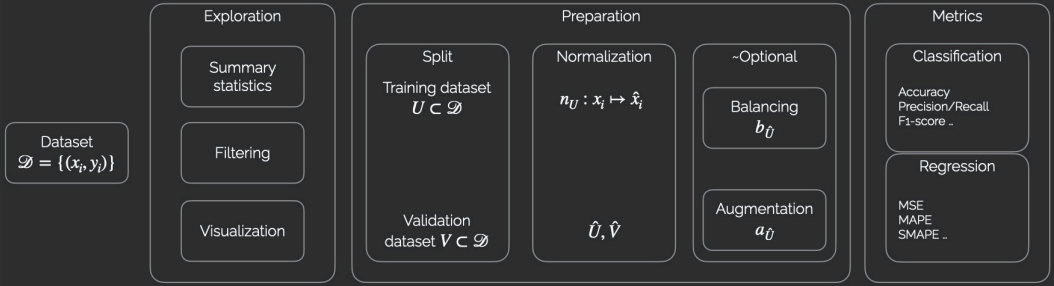
The overall procedure

- Start with a dataset of samples $\{x_i\}_{i=1}^N$ and correct outcomes $\{y_i\}_{i=1}^N$.
- Define a parametric function $\hat{y} = f(x, p)$ (multilayer perceptron, for today).
- Define a loss $\mathcal{L}(y, \hat{y})$.
- Find optimal parameters using a batched optimizer and backpropagation (training).

The overall procedure

- Start with a dataset of samples $\{x_i\}_{i=1}^N$ and correct outcomes $\{y_i\}_{i=1}^N$.
- Define a parametric function $\hat{y} = f(x, p)$ (multilayer perceptron, for today).
- Define a loss $\mathcal{L}(y, \hat{y})$.
- Find optimal parameters using a batched optimizer and backpropagation (training).
- Test the trained multilayer perceptron on novel samples.

From data to prediction



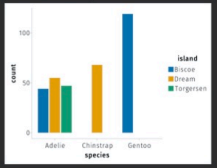
From data to prediction

Exploration

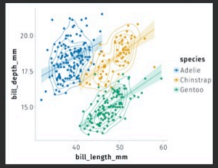
Summary statistics

Visualization

Frequency, distribution,...



Density, linear trend,...



Dimensionality reduction

PCA
t-SNE
UMAP
...

From data to prediction

Split

Training dataset $U \subset \mathscr{D}$

Validation dataset $V \subset \mathscr{D}$

Aim: gain insight on the performance of the model on actual data points

Hard-split

- Permute the dataset (if possible).
- Choose a ratio (e.g., 66% training and 33% test)

Folding

From data to prediction

Normalization

$n_U : x_i \mapsto \hat{x}_i$

$\hat{U}, \hat{V}$

Aim: scale the data uniformly.

Scaling

Let $I = [a, b] \subset \mathbb{R}$, $M = \max U$, and $m = \min U$

$$n_U = \frac{\square - m}{M - m}(b - a) + b$$

Standardization

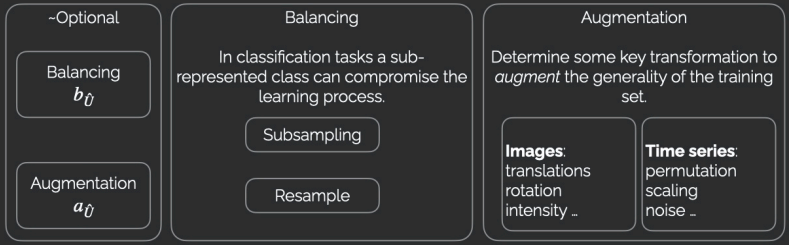
Let $\mu_U = \text{mean}(U)$ and $\sigma_U = \text{std}(U)$

$$n_U = \frac{\square - \mu_U}{\sigma_U}$$

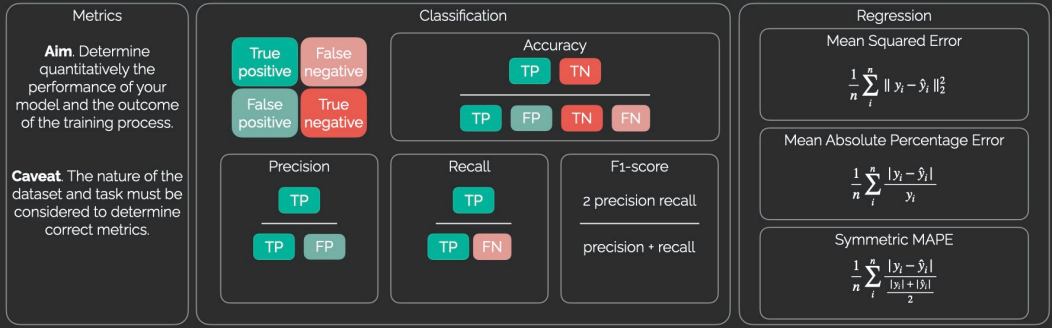
Caveat:

1. n_U should be either agnostic or dependent only on the training data
2. n_U should preferably be invertible

From data to prediction



From data to prediction



From data to prediction

Model
$M(\vartheta) = ?$

Equivariance

Main topic of lecture 2.

Loss

The loss function must be continuous and differentiable.
Beyond these assumptions it has to be tuned to one's problem.

Try to imagine possible loss functions to:
Classify images
Forecast time series
Generate images

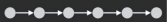
Architecture

There is no ultimate answer in establishing the optimal architecture for a given (dataset, task, model) triplet.

Heuristics and differentiable algorithms can learn architectures. Their optimality is still unclear.

Optimizer

Exercise. Let



be a minimization process through Gradient Descent. How the same process could be represented for SGD and SGD with momentum?

Monitoring

During training, we usually store the following quantities for the training and validation dataset:

- Running loss
- Running metrics

This allows us to spot overfitting and in case add early-stopping to our pipeline

mattiagbergomi@gmail.it

pietro.vertechi@protonmail.com