Part 02 - Machine Learning Foundations

Maura Pintor (maura.pintor@unica.it)

Pattern classification

Goal: assign a label to a pattern

A **pattern** is a description of an object through a set of measurement called **features**

Example

Seagull or flamingo?

How can ML recognize a flamingo?

Let's use one single feature for now, and we decide to use the length of the legs.

We will call this unknown quantity x

In general, flamingos' legs are longer than the seagulls'. Then, we can set a decision rule:

Let's use one single feature for now, and we decide to use the length of the legs.

We will call this unknown quantity x

In general, flamingos' legs are longer than the seagulls'. Then, we can set a decision rule:

if $x > x^*$, it's a flamingo

Let's use one single feature for now, and we decide to use the length of the legs.

We will call this unknown quantity x

In general, flamingos' legs are longer than the seagulls'. Then, we can set a decision rule:

if $x > x^*$, it's a flamingo

if $x <= x^*$, it's a seagull

Let's use one single feature for now, and we decide to use the length of the legs.

We will call this unknown quantity x

In general, flamingos' legs are longer than the seagulls'. Then, we can set a decision rule:

if $x > x^*$, it's a flamingo

if $x <= x^*$, it's a seagull

But how can we estimate x^*?

Training dataset

We need a set of **labeled** examples to compute statistics on the two classes

For example, we can estimate the average lenght of legs of all the flamingos and all the seagulls

$D = [x_1, x_2, x_3, \ldots, x_n]$

Training dataset

Training dataset

However, our classifier still makes mistakes with one single feature

Training dataset

However, our classifier still makes mistakes with one single feature

How can we improve the performances?

We can use an additional feature. Let's use the wingspan. Now we can represent each subject with two values:

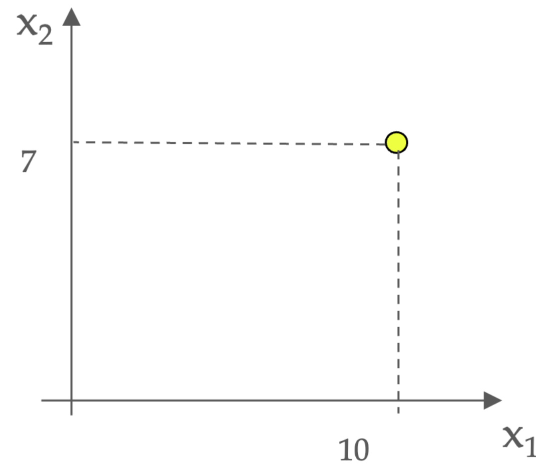
$$\mathbf{x_i} = [x_{i,1}, x_{i,2}]$$

$$\mathbf{x_1} = [10, 7]$$

We can use an additional feature. Let's use the wingspan. Now we can represent each subject with two values:

$\mathbf{x_i} = [x_{i,1}, x_{i,2}]$

$\mathbf{x_1} = [10, 7]$

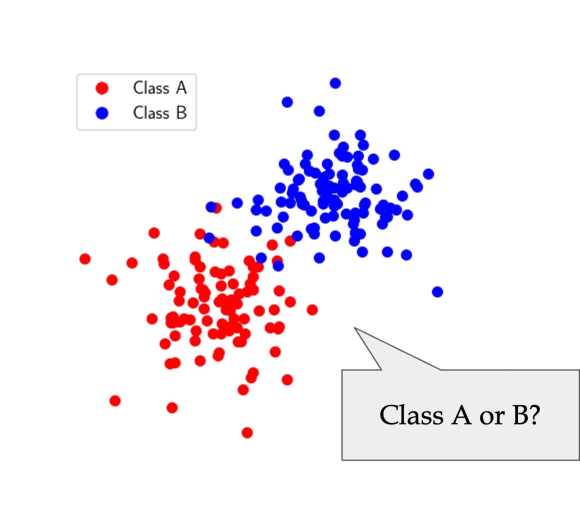


More in general, we can represent patterns as d-dimensional vectors

$$\mathbf{x_i} = [x_{i,1}, x_{i,2}, \dots, x_{i,n}]$$

Now that we have the representation, how can we assign any pattern to a class?

Now that we have the representation, how can we assign any pattern to a class?



How to pick the right classifier?

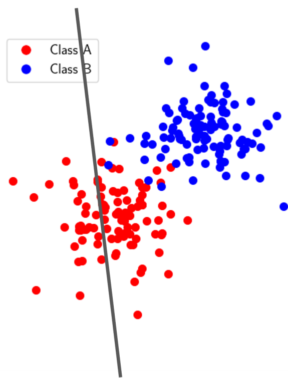
We assume that there is an unknown underlying function that can map inputs (samples) to outputs (classes)

We can start with a simple model to approximate the function

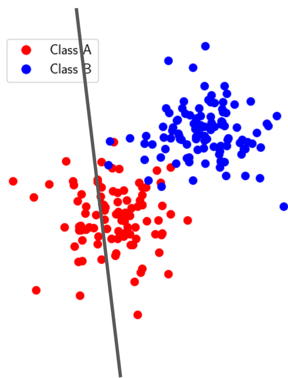
Which one is a good model for separating the two classes?

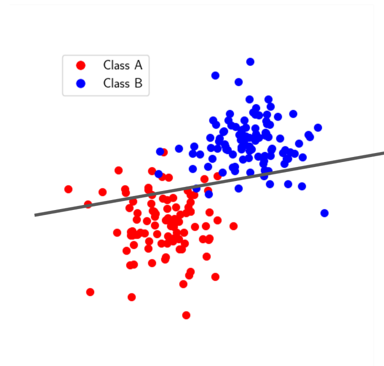
How to pick the right classifier?

How to pick the right classifier?

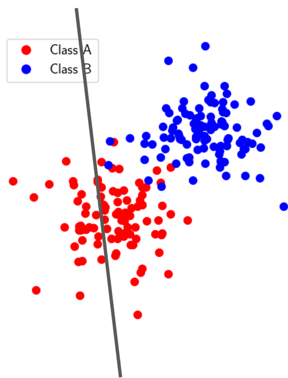


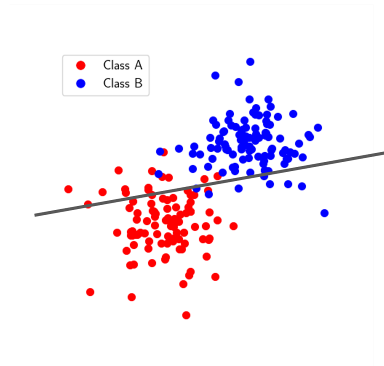
How to pick the right classifier?

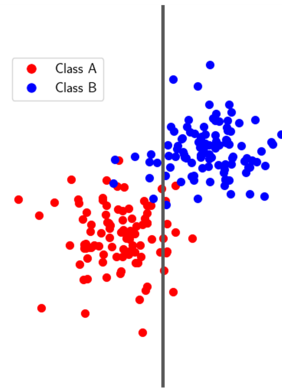




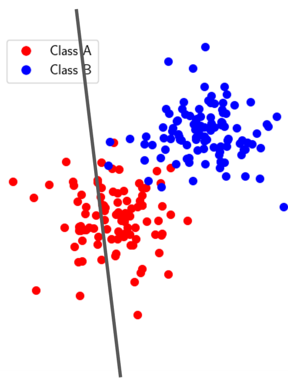
How to pick the right classifier?

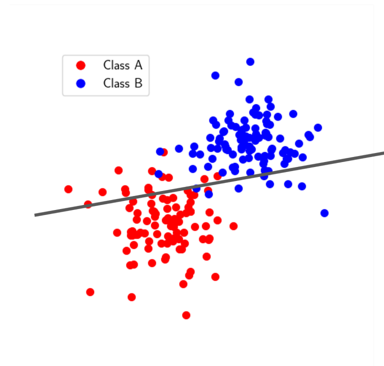


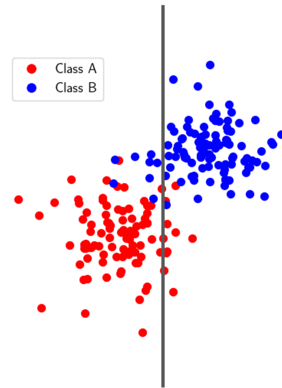


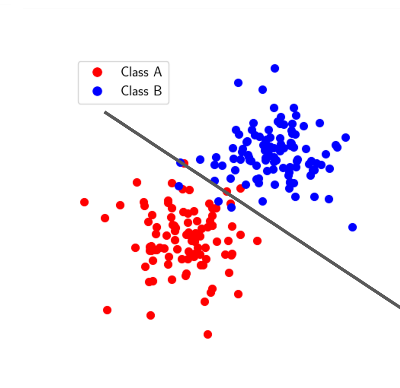


How to pick the right classifier?



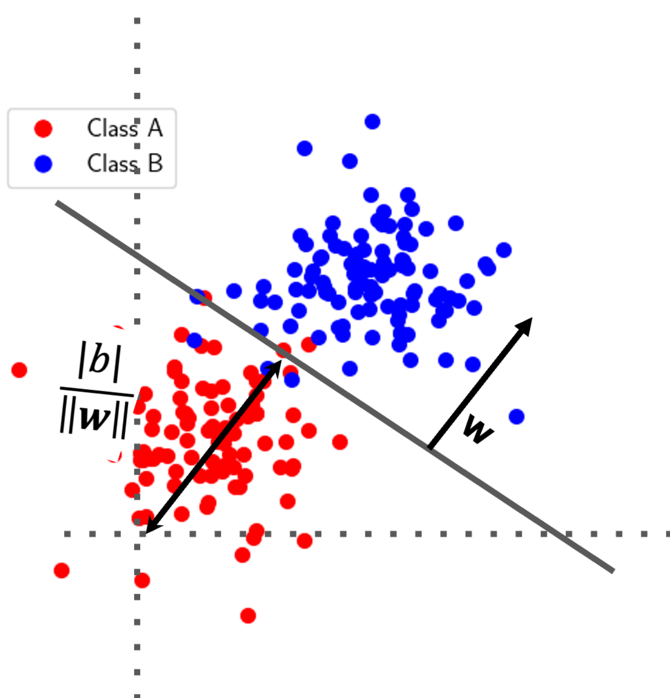






More formally ...

We can write the equation of the separating line (a.k.a., **linear classifier**)



$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{j=1}^{d} w_j x_j + b$$

More formally ...

$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{j=1}^{d} w_j x_j + b$

Then we use the decision rule

if $f(\mathbf{x} < 0)$, it's a *seagull*

if $f(\mathbf{x} >= 0)$, it's a *flamingo*

Let's see with an example

$\mathbf{w} = [1, 1], b = -15$

$\mathbf{x} = [10, 7]$

Let's see with an example

$\mathbf{w} = [1, 1], b = -15$

$\mathbf{x} = [10, 7]$

$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b =$

Let's see with an example

$\mathbf{w} = [1, 1], b = -15$

$\mathbf{x} = [10, 7]$

$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b =$

$= 1 \cdot 10 + 1 \cdot 7 - 15 = 17 - 15 = 2 > 0$

Let's see with an example

$\mathbf{w} = [1, 1], b = -15$

$\mathbf{x} = [10, 7]$

$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b =$

$= 1 \cdot 10 + 1 \cdot 7 - 15 = 17 - 15 = 2 > 0$

$f(\mathbf{x}) > 0$ --> flamingo!

Let's see with an example

$\mathbf{w} = [1, 1], b = -15$

$\mathbf{x} = [10, 7]$

$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b =$

$= 1 \cdot 10 + 1 \cdot 7 - 15 = 17 - 15 = 2 > 0$

$f(\mathbf{x}) > 0$ --> flamingo!

What about the sample $\mathbf{x} = [7, 1]$?

Multi-class classification

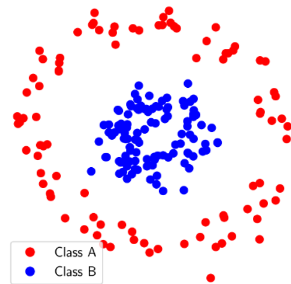
What happens when we have more than two classes?

- one vs. all (OVA)
    - each classifier separates one class from all others
- one vs. one (OVO)
    - each classifier separates one class from one other

In deep learning, in general, OVA is the most used.

Non-linear classifiers

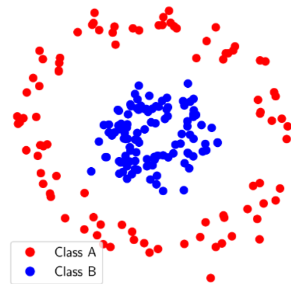
What if the data is not **linearly separable**?



Non-linear classifiers

What if the data is not **linearly separable**?

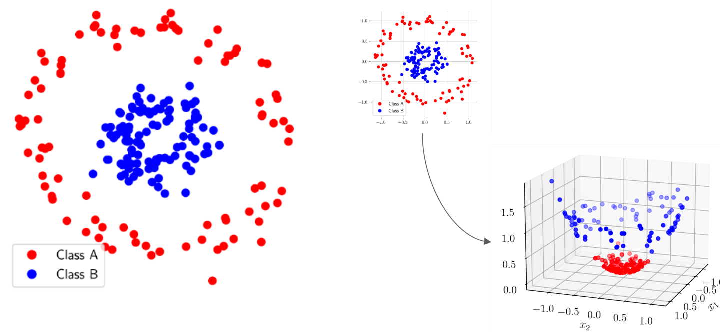
We use tricks to find a separating function that has low error



Non-linear classifiers

What if the data is not **linearly separable**?

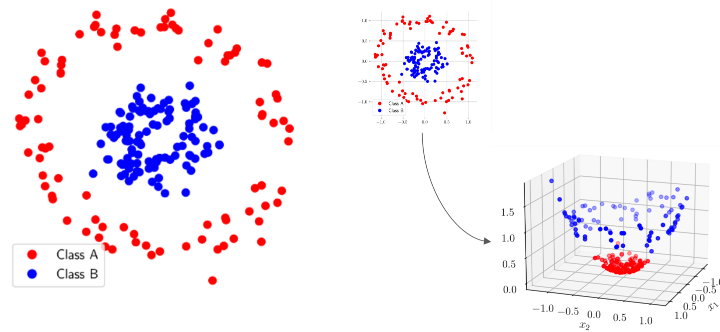
We use tricks to find a separating function that has low error



Non-linear classifiers

What if the data is not **linearly separable**?

We use tricks to find a separating function that has low error

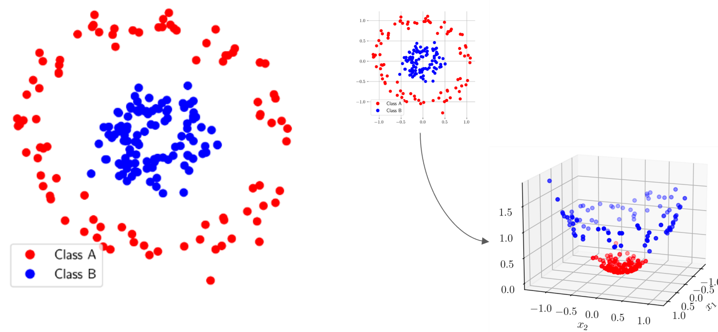


We apply a non-linear transformation to the data points

Non-linear classifiers

What if the data is not **linearly separable**?

We use tricks to find a separating function that has low error

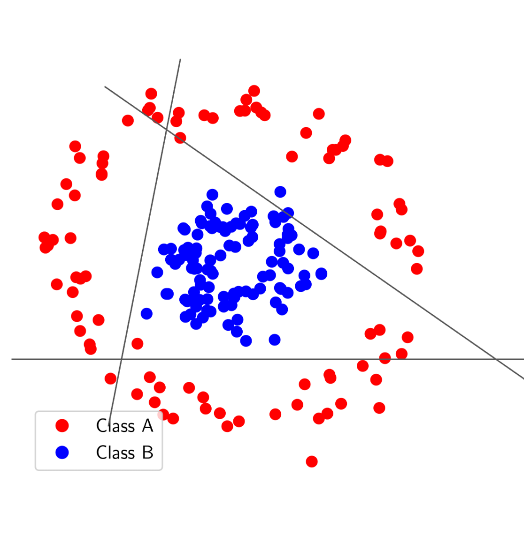


We apply a non-linear transformation to the data points

Then we can find a linear separation in this new parametrization

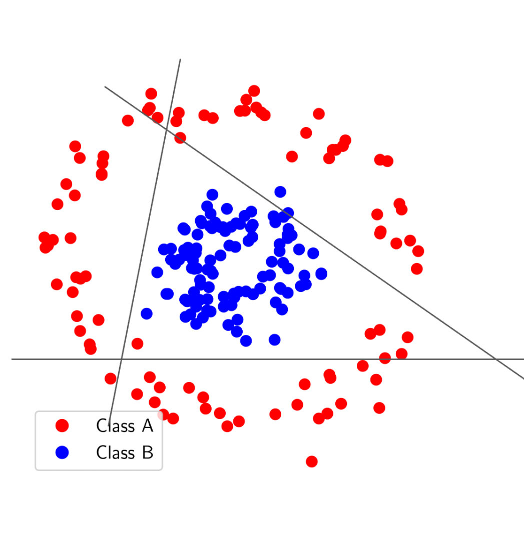
There is another solution

We can also use separate classifiers for different regions of the decision space



There is another solution

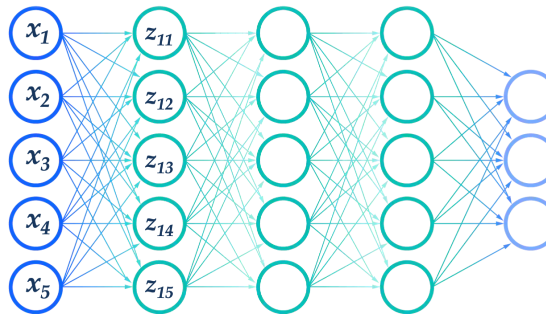
We can also use separate classifiers for different regions of the decision space



or ...

Deep learning

Deep Neural Networks (DNNs) stack **layers** of linear classifiers and non-linear activation functions



The output of each layer l is computed as a function of the product of $\mathbf{w}_l$ and the output of the previous layer $\mathbf{z}_{l-1}$

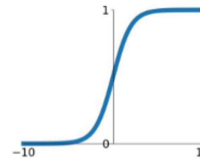
The function is called **activation function** and it is usually non-linear

$$\mathbf{z}_l = a(\mathbf{w}_l \mathbf{z}_{l-1})$$

Choices of the function a are in general:

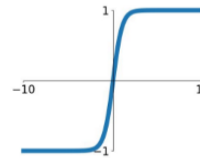
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



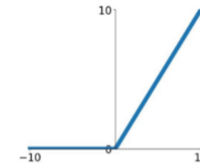
tanh

$\tanh(x)$



ReLU

$\max(0, x)$



Introducing non-linearities ensures learning non-linear decision functions.

How to find the best separator?

To evaluate the goodness of each candidate function (each value of w and b), we define a **loss function** ℓ

$$L(D, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

Where:

- y_i is the true label of sample $\mathbf{x}_i$
- $f(\mathbf{x}_i; \boldsymbol{\theta})$ is the decision function
- $\frac{1}{n} \sum_{i=1}^{n} (\ldots)$ computed over the training set

By computing the loss over the whole training set, we get an estimate of the quality of the predictor

How to get the minimum loss

We define the learning problem as an optimization problem

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} \frac{1}{n} \sum_{1=1}^{n} \ell(y_i, f(\mathbf{x}_i; \theta))$$

Some simple problems have a closed form solution (you can find $\mathbf{w}^*$ and b^* directly by solving the problem)

For others we have to rely on **solvers** that find an approximate solution

How does a loss look like?

One example is the 0-1 loss

Recall the decision rule:

If $f(\mathbf{x}) < 0$, decision is -1

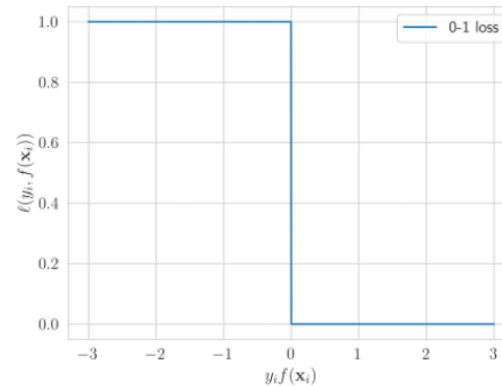
If $f(\mathbf{x}) >= 0$, decision is 1

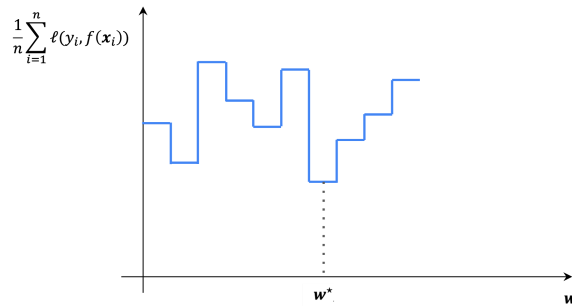
0-1 loss

- y = -1, f(x) < 0 --> classification is correct
- y = 1, f(x) > 0 --> classification is correct
- y = -1, f(x) > 0 --> classification is wrong
- y = 1, f(x) < 0 --> classification is wrong

Which we can write as

$$\ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta})) = 0 \text{ if } f(\mathbf{x}_i) y_i > 0 \text{ else } 1$$





The 0-1 loss is discontinuous and hard to optimize

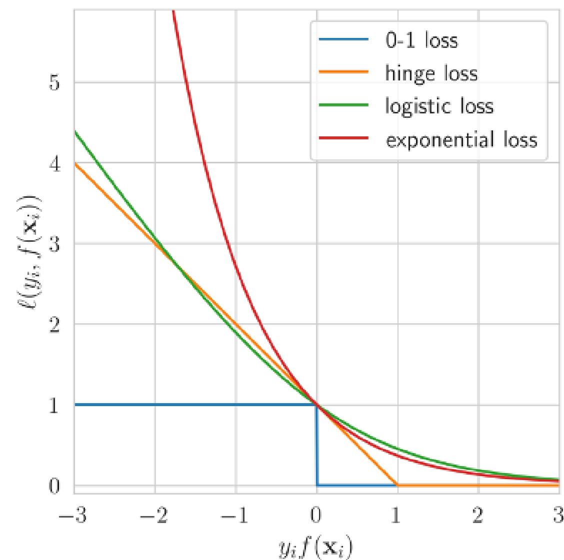
$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

Finding $\mathbf{w}^*$ (and b) requires trying all values of $\mathbf{w}$ (and b)

It's called a **NP-Hard problem**

We can work better with smoother and convex bounds of the 0-1 loss Minimizing these will also minimize the 0-1 loss

- Hinge loss $\ell(y, \mathbf{x}) = \max(0, 1 - yf)$
- Exponential loss $\ell(y, \mathbf{x}) = e^{-yf}$
- Logistic loss $\ell(y, \mathbf{x}) = \log_2(1 + e^{-yf})$



Convexity helps optimization (helps finding solutions more efficiently) and provides guarantees on the optimality of the solution

We can now solve the problem with a **solver**

The most commonly-used solver is **gradient descent**

Gradient descent

The most popular algorithm for solving the optimization problem is Gradient Descent

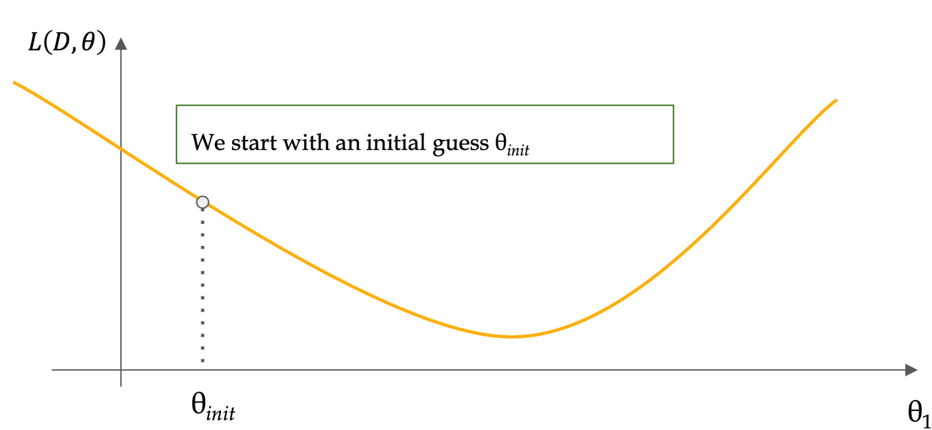
We use the **gradient** of the loss function

$$L(D, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

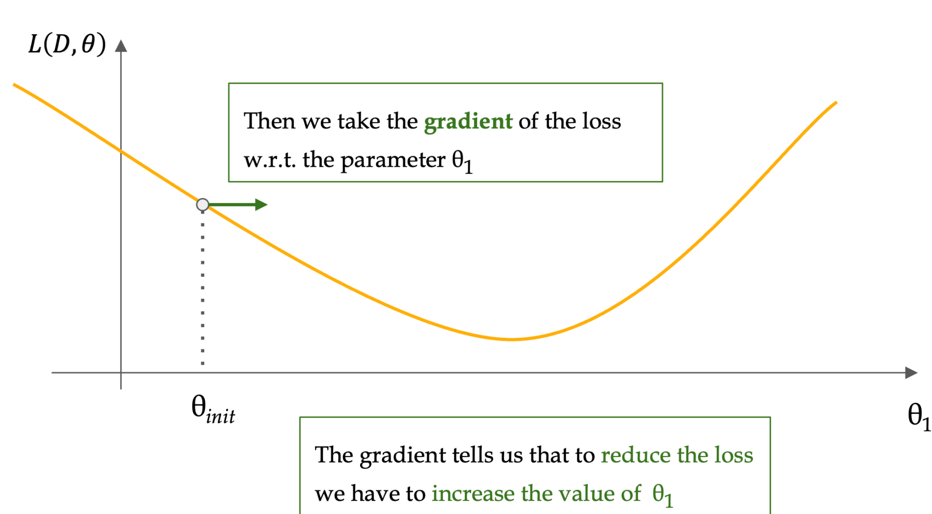
$$\nabla_{\boldsymbol{\theta}} L = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta}))$$

And **iteratively** update the parameters to find the optimal ones

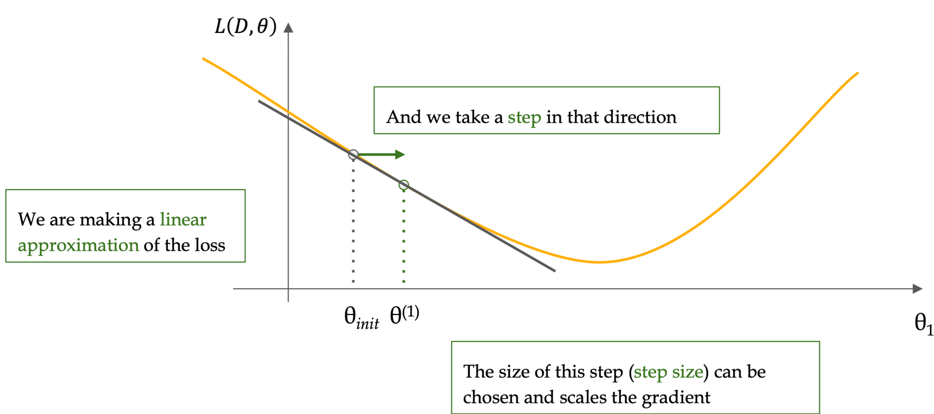
Let's assume that we fix all parameters but one (e.g., θ_1)



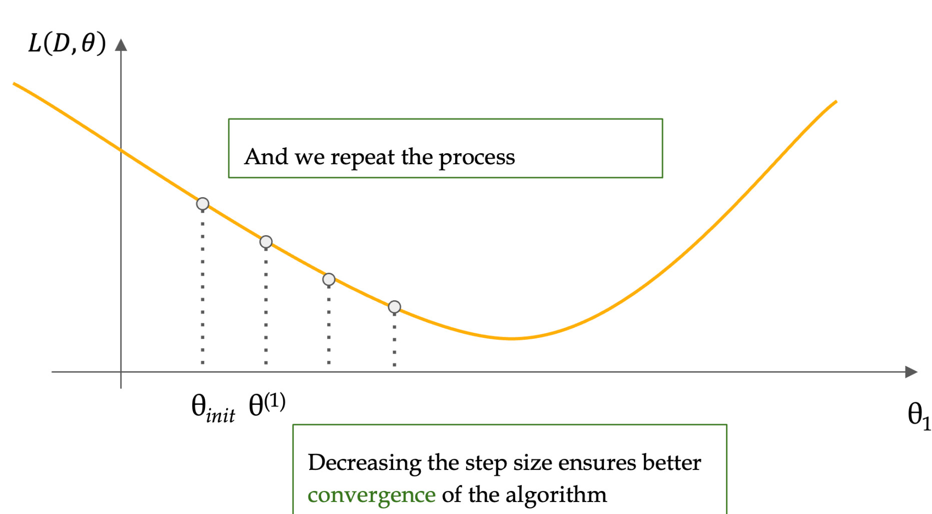
Let's assume that we fix all parameters but one (e.g., θ_1)



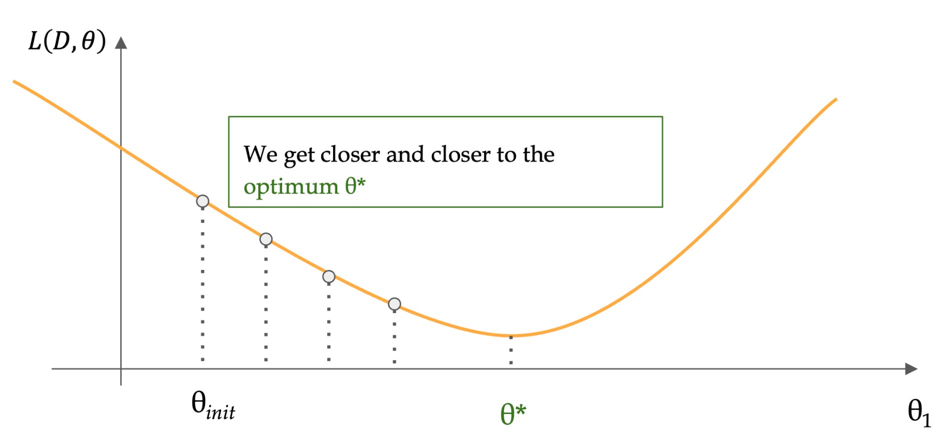
Let's assume that we fix all parameters but one (e.g., θ_1)



Let's assume that we fix all parameters but one (e.g., θ_1)

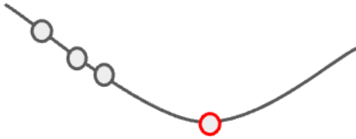


Let's assume that we fix all parameters but one (e.g., θ_1)



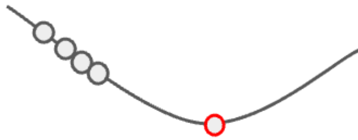
What influences the progress (and results) of the optimization?

- number of steps
  - if we don't take enough steps we can stop too early and far from the optimum



What influences the progress (and results) of the optimization?

- step size
  - if the step size is too small, we need many steps to reach convergence

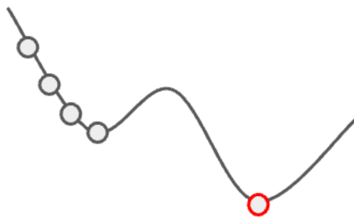


  - if the step size is too big, we might overshoot the optimum

  - the decay of the step size is also important (how much we reduce the step size)

What influences the progress (and results) of the optimization?

- function that we are optimizing
    - there might be local minima and our optimization can get stuck in them



Try out gradient descent with this online tool:
https://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html

In general, we can compute the gradient of the loss w.r.t. multiple parameters and update the parameters simultaneously

The gradient is a vector with one component for each parameter

We can optimize the parameters of a DNN all simultaneously, by computing the gradient of the loss w.r.t. each single parameter

Chain rule and backpropagation

Let's introduce briefly the chain rule, as it will be useful later to find out how to automatically update all the parameters of a DNN with a few lines of code.

Consider two functions of a single independent variable $f(x)$ and $g(x)$. The composite function is defined as $h = g(f(x))$.

You can think of f and g as functions applied in cascade, as the output of $f(x)$ goes as the argument of $g(x)$

Chain rule and backpropagation

The chain rule is useful to find the derivative of the composite function

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x}$$

If $u = f(x)$ is the output of $f(x)$, then we can compute the derivative of the composed function as the product of the derivative of the "external" function w.r.t. u and the derivative of the "internal" function w.r.t. x.

Let's see it with an example:

$$f(x) = 2x - 1 = u$$

$$g(x) = x^2$$

$$h(x) = g(f(x)) = (2x - 1)^2$$

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x} = 2u \cdot 2 = 2(2x - 1) \cdot 2 = 4(2x - 1)$$

Application in machine learning

A neural network can be represented as a nested composite function

$$y = f_K(f_{K-1}(\cdots(f_1(\mathbf{x}))))$$

Here, $\mathbf{x}$ are the inputs to the neural network, whereas y are the outputs

Every function, f_i, for $i = 1, \ldots, K,$ is characterized by its own weights

Application in machine learning

Applying the chain rule to such a composite function allows us to work backwards through all of the hidden layers and efficiently calculate the **error gradient** of the loss function **w.r.t. each weight**, w_i, of the network until we arrive at the input

$$\nabla_{x_i} L = \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial w_1} \frac{\partial w_1}{\partial x_i} + \cdots + \frac{\partial L}{\partial w_M} \frac{\partial w_M}{\partial x_i}$$

Thus, we can update each weight so that the loss decreases

This is the basic parameter update for gradient descent. By re-iterating multiple times and using a small enough learning rate, the model will converge to better values of parameters (lower loss)

$$w_i \leftarrow w_i - \alpha \nabla L_{w_i}$$

Autograd

We can get the gradient of all parameters of our models by propagating derivatives backward using the chain rule

The basic requirement is that all functions can be differentiated analytically

If this is the case, we can compute the gradient with respect to the parameters in one sweep (even with highly-complicated models!)

Computing the gradient of the loss with respect to the parameters amounts to writing the analytical expression for the derivatives and evaluating them once.

However, this composition might be complicated to write and compute analytically

This is when PyTorch come to the rescue, with a PyTorch component called **autograd**

PyTorch creates operations that can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs

This means we won’t need to derive our model by hand

Given a "forward" expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

Autograd

We will see in detail how to apply autograd in the next lessons.

Elements of performance evaluation

Once we train our model, we should evaluate its performance on a separate dataset (unseen in training)

We can represent the predictions with the **confusion matrix**

		Predicted	
		Positive	Negative
True Class	Positive	TP	FN
	Negative	FP	TN

- TP = True Positive = Predicted positive and actual positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

We can represent the predictions with the **confusion matrix**

		Predicted	
		Positive	Negative
True Class	Positive	TP	FN
	Negative	FP	TN

- Correct = TP + TN
- Errors = FP + FN

We can define metrics that measure the error (or the correct predictions), or characterize different kinds of errors

Accuracy $= \dfrac{(TP+TN)}{(TP+TN+FP+FN)} = \dfrac{\text{correct}}{\text{all samples}}$

Precision $= \dfrac{TP}{(TP+FP)} = \dfrac{\text{how many predicted and really positive}}{\text{all predicted positive}}$

Recall $= \dfrac{TP}{(TP+FN)} = \dfrac{\text{how many predicted and really positive}}{\text{all really positive}}$

The problem with achieving zero error

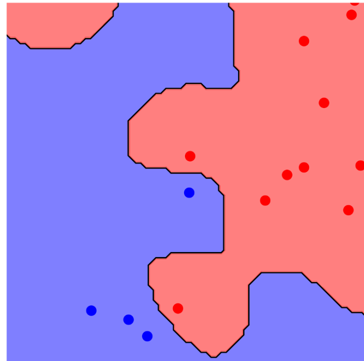
Learning the best parameters for the given dataset might lead to poor generalization

It means that the classifier is overly-specialized on the training set, but it does not work well on unseen data

The problem with achieving zero error

Learning the best parameters for the given dataset might lead to poor generalization

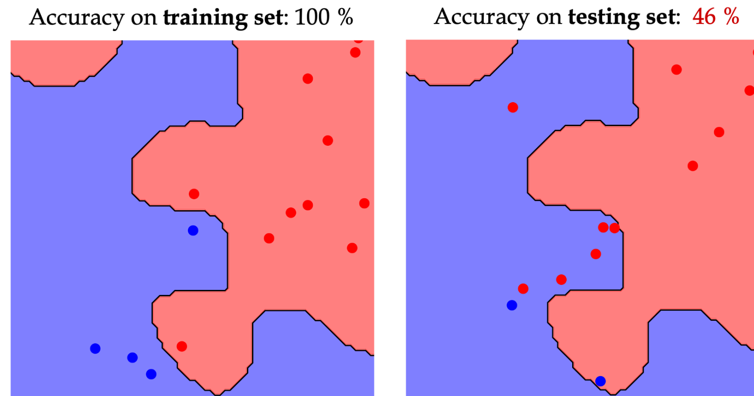
It means that the classifier is overly-specialized on the training set, but it does not work well on unseen data



The problem with achieving zero error

Learning the best parameters for the given dataset might lead to poor generalization

It means that the classifier is overly-specialized on the training set, but it does not work well on unseen data



Overcoming overfitting

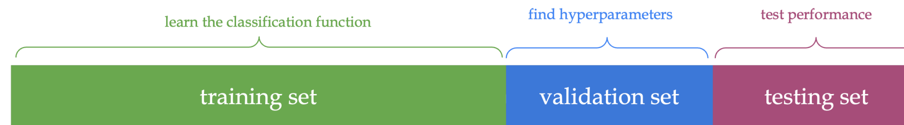
One technique to reduce overfitting is to add a penalty term to prevent the model from giving too much weight to specific samples

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(\mathbf{x}_i; \boldsymbol{\theta})) + \lambda \Omega(\mathbf{w})$$

- The term $\Omega(\mathbf{w})$ enforces a penalty on the magnitude of the classifier's parameters to promote smoother functions across the feature space
- The hyperparameter λ tunes the tradeoff between the loss and the penalty
  - larger λ: more regularized, at the cost of increasing the error
  - smaller λ: reduces training error but learns more complex functions

Overcoming overfitting

The parameters (e.g., λ) should be picked separately with a different set called validation set



The Cross-Validation strategy allows validation on different runs with the same dataset

End of part 3

Summary:

- Loss function and optimization
- Optimization problems
- Solvers - gradient descent
- Performance evaluation, overfitting, validation

End of part 2

In the next chapter:

- Tensors basics

Maura Pintor (maura.pintor@unica.it)