ML Introduction

The meaning of learning

In order to define what learning means as an ML concept we need three concepts:

- A certain task T that we want to improve on.
- Metrics to measure performance P.
- Experience E.

> *Learning means improving on a certain task T by gaining experience E in order to maximize performances P.*

The thing we want to learn is a ***target function.***

A target function can be defined in many ways and depends on our design choices. For example action functions outputs the next action that an agent should take, while a value function returns a value associated to a certain state.

The way we express a target function is assigning weights to meaningful components of our inputs. If we take for example the configuration of a board b, we can have a function that looks like this:

$$V(b) = w_0 + w_1 bp(b) + w_2 rp(b) + ...$$

In this case $bp(b)$ is the number of black pieces on board b, $rp(b)$ is the number of red pieces on board b etc.

Our problem is about estimating the weights in order to obtain an optimal solution. In order to define what we mean by “finding an optimal solution” we have to define some concepts:

- $V(b)$ is the target function. Basically it represents perfection in our task (that we’ll never reach).
- $V'(b)$ is our function that we will manipulate.
- $V_{train}(b)$ is the function given by the dataset.

We have to structure an algorithm such that given the estimate of the error:

$$error(b) = V'(b) - V(b)$$

will minimize it. This means that we want our function to get as close as possible to the ideal one by iterating the algorithm on the training dataset. The algorithm will work on our function's weights in order to achieve this.

The output of our algorithm will be the function $\hat{V}(b)$ that we will use in practice.

What is Machine Learning?

Machine learning is the set of all the various learning techniques. The ones we will study are:

- Supervised learning
  - Classification
  - Regression
- Unsupervised Learning
- Reinforcement Learning

In general ***a machine learning problem is the task of learning a function given a dataset***.

A function maps elements of a certain input domain to elements of a certain output domain:

$$f : X \to Y$$

A dataset D is a set of samplets that contains informations about the function.

Learning a function means to iteravely generate a function $\hat{f}(x)$ that best approximates the ideal function $f(x)$:

$$\hat{f}(x) \approx f(x) \;\; for\; x \in X - X_d$$

Note that our goal during learning is achieving good prediction *outside* the dataset's domain, which means that we are able to generalize knowledge.

The input dataset will influence the kind of technique we will use. In fact we have:

- *Supervised learning* when the dataset contains information about inputs and their corresponding outputs: $D = \{(x_i, y_i)_{i=1}^n\}$
- *Unsupervised learning* when the dataset contains only the inputs: $D = \{(x_i)_{i=1}^n\}$

Moreover functions can be cathegorized based on the input and output domains.

Inputs can be expressed as a finite set of numbers $A_1 \times ... \times A_m$ (*discrete*) or *continuous* if $X \in \mathbb{R}^m$.

When the output is finite we say that we are facing a problem of *classification.* In the continuous case we have a problem of *regression*.

We call *hypothesis space* H the space of all possible approximation of the optimal function *c*. The learning task consists in finding the best approximation $h^* \in H$ of $c : X \rightarrow Y$.

An hypothesis is called *consistent* only if:

$$h(x) = c(x) \ \ \forall x \in D$$

Performance evaluation

Defining the error

In order to see if our hypothesis h^* is good we want to find a way to compute its error rate and accuracy. In doing so, we must take in account that our hypothesis is trained on an input space which generally follows a certain distribution. A distribution plots every value of our sample together with its probability of finding it inside the population.

That said we define two kinds of error:

- **True error** is the probability that when extracting a sample from the distribution our hypothesis will give an incorrect prediction: $error_D(h) = P_{x \in D}[f(x) \neq h(x)]$
- **Sample error** is the error defined on the testing dataset for which we already know correct solutions. This is not a probability and is computed as: $error_S(h) = \frac{1}{n} \sum_{x \in S} \delta(f(x) \neq h(x))$

Accuracy is defined as:

$$accuracy(h) = 1 - error(h)$$

In order to have a good estimate of the error, we want the sample error to be as close as possible to the true error. This concept is expressed as the *estimation bias*, which shows the difference between the two errors:

$$estimation\ bias = E[error_S(h)] - error_D(h)$$

Note that the sample error is expressed as an expected value, since it is a random variable depending on the sample that has been extracted. We always have to remember that the true expected value of sample error can't be computed, since we would need infinite samples, that corresponds to already having a complete knwoledge of the problem.

The solution in order to have zero estimation bias is making shure that our test and train set will be disjoint. Since we have to work with randomness, the best we can do is keeping a low variance of the estimation of the error. In order to do this we use the concept of confidence intervals:

With approximately $N\%$ probability the $error_D(h)$ falls in to the interval:

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h)}{n}}$$

Where z_N is the coefficient which describes the grade of precision we want.

The only way to keep a good confidence while having a small interval is by increasing n, the number of samples.

The steps of the error computation are:

1. Partition the dataset in train test T and test set S such that: $D = T \cup S, T \cap S = \emptyset, |T| = \frac{2}{3}|S|$
2. Compute an hypothesis using T
3. Evaluate $error_S(h) = \frac{1}{n} \sum_{x \in S} \delta(f(x) \neq h(x))$

Notice that $error_S(h)$ is a random variable since it depends on the split. This error is defined as an unbiased estimator for $error_D(h)$

For a medium sized dataset the size of the split is: $\frac{2}{3}\ train, \frac{1}{3} split$.

Evaluating our solution

Once we have an hypothesis, we want to test it against the possible problems that we can have during evaluation.

A major one can be *Overfitting*. This error happens when our solution gets too specialized on predicting the input space, but results weak when testing. Formally we say that an hypothesis $h \in H$ overfits training data if exists another hypothesis $h' \in H$ such that:

$$error_S(h) < error_S(h') \ \wedge \ error_D(h) > error_D(h')$$

In order to improve our confidence on the avaluation of error we use a technique calles *K-fold cross validation.* It consists on repeating the process of error evaluation multiple times, each time on a different split.

In practice we divide our dataset in k partitions $S_1, S_2, ..., S_k$. At each iteration we use S_i for training and every other for testing, obtaining k different errors. At the end we compute:

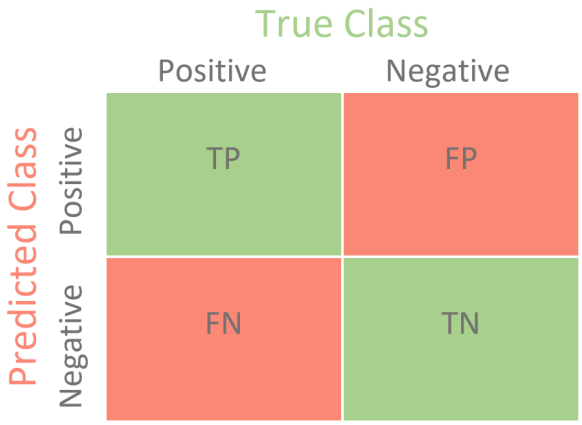
$$error_{L,D} = \frac{1}{K} \sum_{i=1}^{K} \delta_i$$

Notice with K-fold we are evaluating the the learning method. Similarly we can compute accuracy of the learning method as. $accuracy_{L,D} = 1 - error_{L,D}$.

But accuracy not always tells us the whole story. In fact having an unbalanced dataset could still cause us trouble. Take this example:

We have a two classes dataset $D = \{+, -\}$ where negative samples make the 90% of it. From this we train two hypothesis h_1, h_2 obtaining that h_1 is 90% accurate and h_2 is 85% accurate. We might say that the first one is always better, and we'd be wrong. For example h_1 might just be guessing that every sample is negative.

We need to introduce a more accurate metric: *the confusion matrix*. This matrix shows us the number of true guesses vs the number of false guesses. For example if we have two classes, the corresponding confusion matrix is:



The values on the main diagonal represent the correct guesses. From this matrix we can extract other important informations, namely:

$Recall = \frac{|true\ positives|}{|real\ positives|} = \frac{TP}{TP+FN}$

$Precision = \frac{|true\ positives|}{|predicted\ positives|} = \frac{TP}{TP+FP}$

Recall tells us how good is our system in avoiding false negatives, while precision does the same for false positives. Depending on the context, we might be more interested in having a high precision rather than recall and vice versa. In general we want them to be balanced, so we use the F1-score, which accounts for both:

$$F1 - score = \frac{2(Precision \times Recall)}{Precision + Recall}$$

Decision Trees

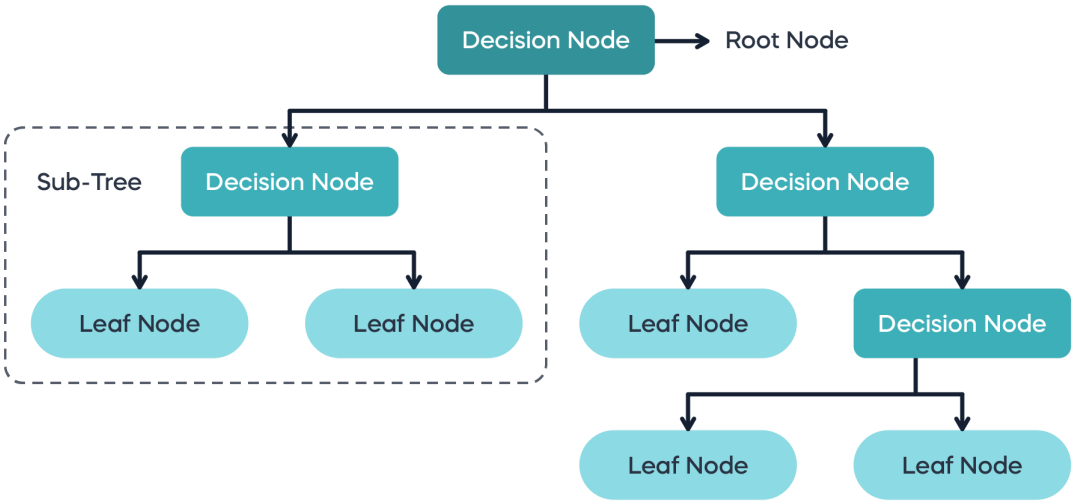
The problem

Given an input dataset consisting in a cartesian product of m attributes $X = A_1 \times A_2 \times ... \times A_m$, and a set of classes C, we want to find a function such that $f : X \to C$.

What is a decision tree

A decision tree is made by:

- **Nodes**, which are the tests for attributes.
- **Branches**, which represent the possibile values for each attribute.
- **Leaves**, which represent the classes.



We define H as the hypothesis space, namely it is the set of all possible decision trees.

A great advantage of decision tree is that they are *expleainable*, meaning that their policy is fully exposed, and can be represented as a formula made by conjunctive rules disjoint by each other.

For example:

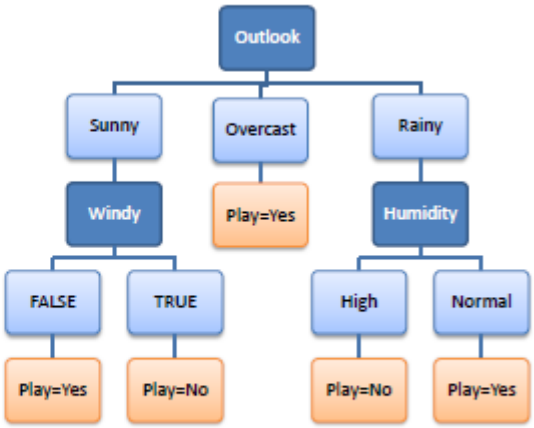
R_1: IF (Outlook=Sunny) AND (Windy=FALSE) THEN Play=Yes

R_2: IF (Outlook=Sunny) AND (Windy=TRUE) THEN Play=No

R_3: IF (Outlook=Overcast) THEN Play=Yes

R_4: IF (Outlook=Rainy) AND (Humidity=High) THEN Play=No

R_5: IF (Outlook=Rain) AND (Humidity=Normal) THEN Play=Yes



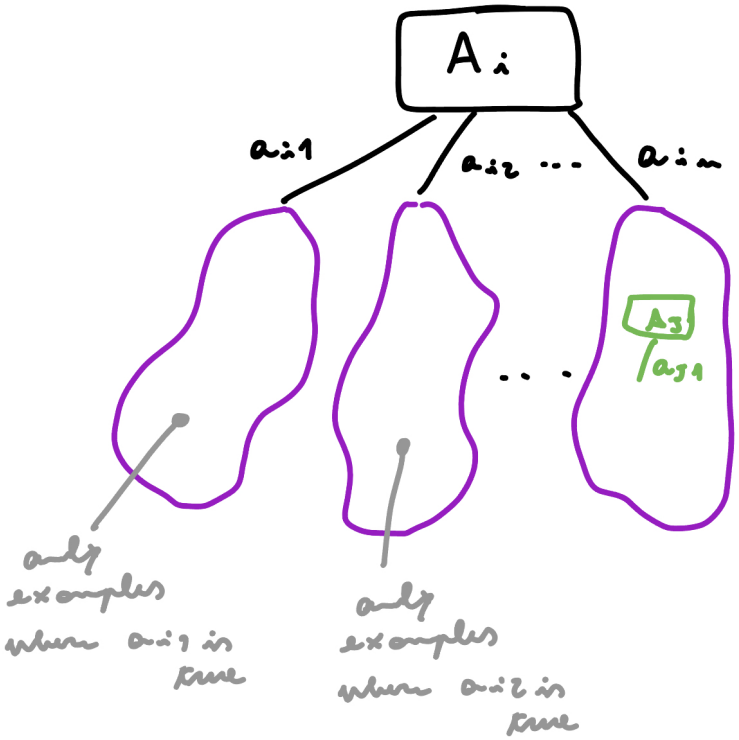
Computing a decision tree: the ID3 algorithm

The ID3 algorithm takes as an *input* three sets: Examples (dataset), Target Attributes (classes), Input Attributes (input) and outputs a decision tree. The algorithm works in a recursive way.

The base cases for this algorithm are.

- All examples are positive → return a leaf node with a positive label.
- All examples are negative → return a lead node with a negative label.
- There are positive and negative examples. but the list of attribute is empty (An example could be the absence of a person among the population with age over 100 years)→ choose the leaf node giving it the label that is most abundant in the dataset (if we have more positives than negatives, then put a positive label, and vice-versa).
- In the case that we have both positives and negatives examples and the list of attributes is not empty, we follow these steps:
  1. Choose the optimal attribute A_i (optimality will be discussed)
  2. Create a node with A_i
  3. Add branches and childrend with that attribute

4. Recursively repeat these steps for its childrens

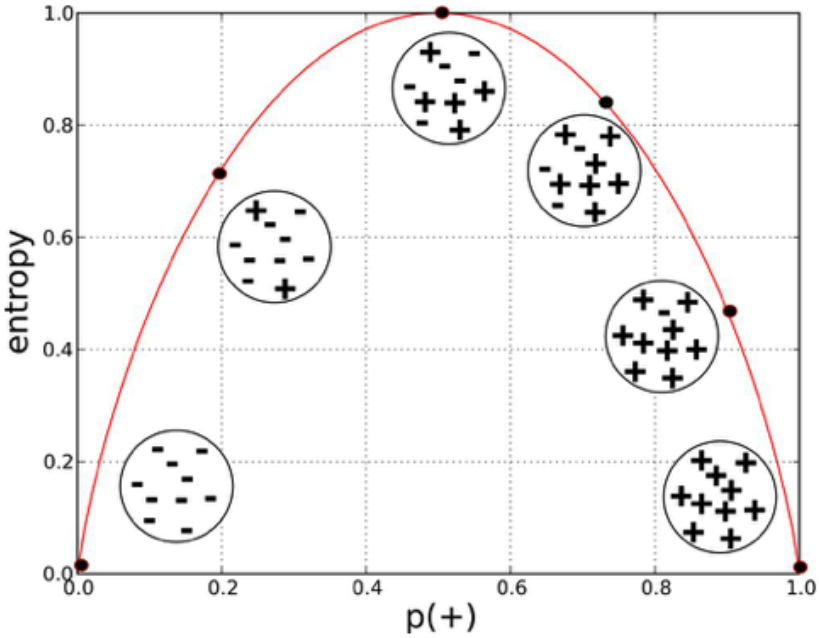


Finding the best split

The problem of choosing each time the best attribute for a split is crucial, since the same decision tree could be shorter or much longer due to this choice. We want to find the optimal one given that our policy is that the best decision tree is the most compact one.

In order to achieve this, we refer to two metrics: entropy and information gain.

Entropy gives an idea of how much examples are balanced inside a certain population. For example if we have two classes, entropy follows this curve:



Notice that entropy is maximum (1) when there is exactly the same number of positive and negative examples, while it is minimum (0), when we have just positives or negatives.

When choosing the best split we seek the minimum entropy, since this will direction us towards the most compact decision tree.

Information gain

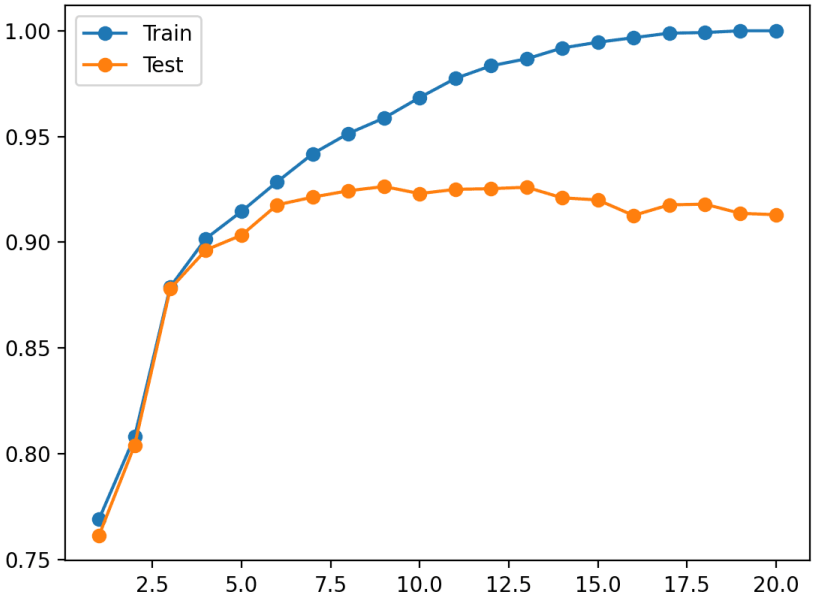
Given a dataset S and an attribute A, the information gain measures the reduction of entropy when splitting S according to A:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

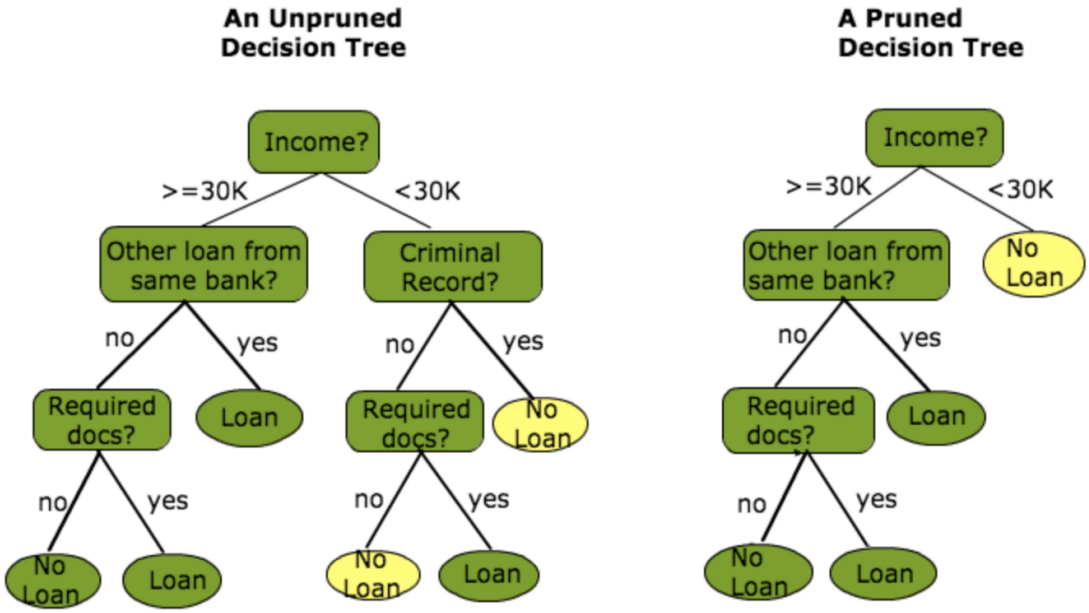
Our algorithm will choose each time the attribute that brings the highest information gain, thus the “best split”.

Evaluating decision trees: the problem of overfitting

When feeding a decision tree, we will always obtain that train accuracy will increase steadily. It happens that there is always a certain point where test accuracy will start dropping, no matter if train accuracy keeps growing. Starting from this point, we happen to be in an overfitting region, meaning that our decision tree will lose its ability to generalize predictions:



A possible approach to correct this phenomena is to perform **post-pruning**. Basically we let the training run, and once finished we start cutting parts of the tree, thus performing backtracking on the error. The logic is that we will keep pruning each time we will obtain a new test accuracy that is higher than the previous one. We will stop only when performance will start dropping once again (we reached a local minimum).



Remember that pruning actually means removing constraints form the rule representation of a decision tree.

Bayesian Learning

Probability notions

Independence

Two events A and B are independent if $P(A|B) = P(A)$ and $P(B|A) = P(B)$.

Conditional independence

There are situations where three or more events may be dependent by each other, but can be expressed as independent if expressed in a certain way.

For example:

$$P(catch|toothache, cavity) = P(catch|cavity)$$

In this case the information about toothache is redundant, thus it can be considered conditionally independent.

In general, we say that X is conditionally independent from Y given Z if:

$$P(X|Y, Z) = P(X|Z)$$

The union of conditional independence and chain rule result in the following formula:

$$P(X, Y, Z) = P(X|Y, Z)P(Y, Z) = P(X|Y, Z)P(Y|Z)P(Z) = P(X|Z)P(Y|Z)P(Z)$$

The application of both rules simplifies a lot the number of computations needed, reducing them from an exponential (2^n) to a linear $(2 \cdot n)$ complexity.

Bayes' rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

It is useful to express this formula as the dependency between cause and effect:

$$P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}$$

What if we want to study the probability of a certain cause to be caused by more effects?

If conditional independence can be applied, we can express it as:

$$P(cause|effect_1, ..., effect_n) = \alpha P(cause) \prod_i P(effect_i|cause)$$

Where α is a normalization factor that can be generalized for this purpose.

Bayesian learning

The foundation of Bayesian learning is that we can express a machine learning problem as a probabilistic one.

In other words, we define our target function $f : X \to V$, with X being our input domain and V our output (let's imagine it as classes). We condider D as the dataset and x' a new instance. The best prediction will be $\hat{f}(x') = v^*$, where:

$$v* = argmax_{v \in V} P(v|x', D)$$

> 💡 Conceptually we are saying that the best prediction is the one that maximizes the probability that it was generated given our prior knowledge of the dataset, and the knowledge of the new instance.

When training, we want to compute the following probability distribution:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

That is the probability that h is the solution given the dataset D, where:

- $P(D|h)$ is the probability that the dataset was generated from the hypothesis.
- $P(h)$ is the prior probability of the hypothesis h.
- $P(D)$ is the prior probability of training data D.

The MAP and ML hypothesis

The foundation of Bayesian learning says that the best the hypothesis is the *Maximum A Posteriori* (MAP) hypothesis, meaning the one that maximizes the probability that it was generated given the training dataset:

$$h_{MAP} = argmax_{h \in H} P(h|D) = argmax_{h \in H} P(D|h)P(h)$$

D will be our input, and we will have to search inside the hypothesis space H the optimal one. With the current formulation of this problem, we would have to know $P(h)$, namely the a-priory probability of h to happen. That is not always the case. In order to address the problem, we could simply assume that *all a-priori probabilities for our hypothesis are the same*, thus eliminating that term.

The problem formulated in this is called finding the *Maximum Likelihood* (ML) hypothesis:

$$h_{ML} = argmax_{h \in H} P(D|h)$$

Making predictions

Once we obtained our hypothesis, how do we make predictions?

We might think that applying $h_{MAP}(x')$ will give the best prediction, but is it actually the most probable one?

Let's consider the following case: we have three hypotesis h_1, h_2, h_3. Upon computation we find that:

$$P(h_1|D) = 0.4 \;\; P(h_2|D) = 0.3 \;\; P(h_3|D) = 0.3$$

Let's suppose that given a new instance x' we obtain the following predictions:

- $h_1(x') = positive$
- $h_2(x') = negative$
- $h_3(x') = negative$

If we just cosidered the MAP hypothesis, we would say that our prediction is positive. But we can observe that h_2 and h_3 give a compound contribution of 0.6, which is heavier than h_1.

We can elaborate a strategy to take in account this problem, by considered a weighted probability of our hypothesis. This approach is called the *Bayes' Optimal Classifier*.

Consider a target function $f : X \to V$, with $V = \{v_1, ..., v_n\}$ (the set of classes), the dataset D and a new instance x'.

The probability for each class will be:

$$P(v_j|x', D) = \sum_{h_i \in H} P(v_j|x', h_i) P(h_i|D)$$

Meaning that the probability for each class is the average of the probability of the class given each hypothesis weighted by the probability of that hypothesis was generated by the training dataset.

The best prediction will be:

$$v_{OB} = argmax_{v_j \in V} \sum_{h_i \in H} P(v_j|x', h_i) P(h_i|D)$$

The Optimal Bayes' Classifier always finds the optimal solution.

Naive Bayes Classifier

If we consideral the general case for a machine learning problem, we'll have to deal with two major problems: the space of hypothesis might be very big and instances can be described by a set of many attributes. In this cases finding a solution with the Bayes Optimal Classifier method might be unfeasable.

In order to deal with the general cases we use the Naive Bayes Classifier, which exploits the notion of conditional independence.

Let's assume that each instance x is described by a set of attributes $< a_1, ..., a_n >$. We want to compute:

$$argmax_{v_j \in V} P(v_j | x, D) = argmax_{v_j \in V} P(v_j | a_1, ..., a_n, D)$$

The MAP prediction will be (using Bayes' rule):

$$v_{MAP} = argmax_{v_j \in V} P(v_j | a_1, ..., a_n, D) = argmax_{v_j \in V} \alpha P(a_1, ..., a_n | v_j, D) P(v_j | D)$$

The naive Bayes assumtion is that all attributes are conditionally independent. Applying this consideration we wil obtain:

$$v_{NB} = argmax_{v_j \in V} P(v_j | D) \prod_i P(a_i | v_j, D)$$

What we obtained is a Naive Bayes Classifier. This kind of classifier does not guarantee to find the optimal solution, but it is still able to have good performances.

Probabilistic models for classification

Probabilistic generative models

Let's consider the usual problem of classification. We want to find a function $f : X \to C$, given a training dataset D, with the goal of being able to classify new instances $x' \notin D$.

> 💡 Generative models have a **two-step approach** to this problem. The first one consists in computing the **class-conditional density** $P(x|C_i)$. This is a description of how the dataset has been generated in terms of a probabilistic model, meaning that given this first model we could be able to generate new data sampling from it. The second step for this approach will be using the class-conditional density in order to compute the **posterior probability** $P(C_i|x)$ **using Bayes' theorem**.

During the following explainations, the term D representing the dataset will be omitted from formulas in order to use a lighter notation.

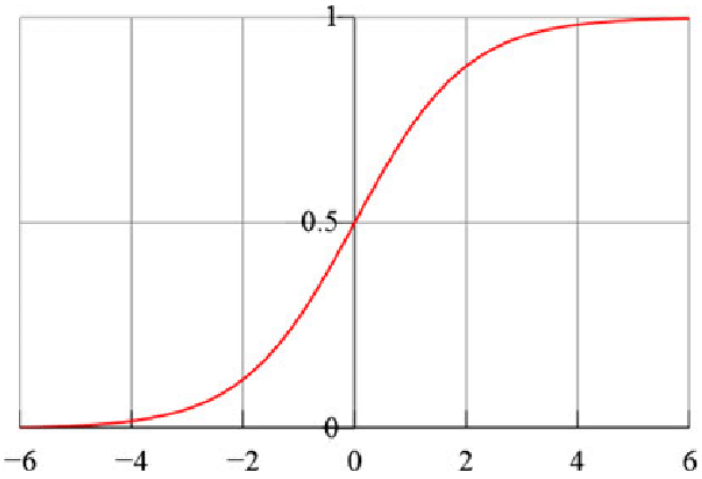
The two classes case

We want to find the posterior probability assuming that we have two classes C_1, C_2:

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x)} = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)} = \frac{1}{1+e^{-a}} = \quad = \sigma(a)$$

This first formula shows us that we can model the computation of posterior probability as a sigmoid function, having as a term $a = ln\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}$, the logaritm of the ratio of the two classes.

A sigmoid function has the interesting probability of squeezing every value $\in (-\infty, +\infty)$ in a range between 0 and 1.



We are now interested in computing the conditional probability of a sample given a class. We will start from the important assumption that samples are generated by an underlying Gaussian distribution. We obtain the following equality:

$$P(x|C_i) = \mathcal{N}(x; \mu_i, \Sigma)$$

Where the Gaussian distribution is parametrized by two terms: the mean for each class μ_i and the covariance matrix Σ, which is the same for every class. That said, we can update our formula for the term a:

$$a = ln\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)} = ln\frac{\mathcal{N}(x; \mu_1, \Sigma)P(C_1)}{\mathcal{N}(x; \mu_2, \Sigma)P(C_2)}$$

Now it comes an important trick. We define the terms:

$$w = \Sigma^{-1}(\mu_1 - \mu_2)$$

$$w_0 = -\frac{1}{2}\mu_1^T\Sigma^{-1} + \frac{1}{2}\mu_2^T\Sigma^{-1}\mu_2 + ln\frac{P(C_1)}{P(C_2)}$$

That after some computation will let us express the term a as a linear combination of terms in x:

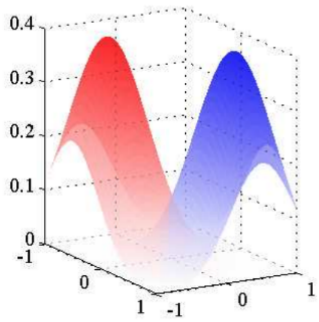
$$a = ln\frac{\mathcal{N}(x; \mu_1, \Sigma)P(C_1)}{\mathcal{N}(x; \mu_2, \Sigma)P(C_2)} = ... = w^Tx + w_0$$

Since a is the only variable in the formulation of the posterior probability as a sigmoid, we reduced the problem of computing the posterior probability as a linear equation:

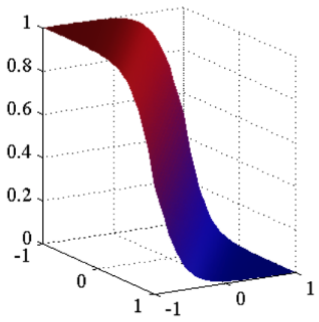
$$P(C_1|x) = \sigma(w^Tx + w_0)$$

Let's now assume that we can compute w^T and w_0. This means that we know the distribution of the classes over the samples, an example of that is the left illustration in the image below. Starting from this information, we will able the compute the posterior probability for every class. The logic

here is that by placing any new instance on the distribution we will be able to know how likely it is that it belongs to a certain class.



$P(\mathbf{x}|C_1), P(\mathbf{x}|C_2)$



$P(C_1|\mathbf{x})$

In the two class cases the decision making policy is simple:

$$P(C_1|x) \begin{cases} > 0.5 & \text{output } C_1 \\ < 0.5 & \text{output } C_2 \end{cases}$$

The k-class case

Let's suppose the general case, where we have three classes. The prior probability for each class is:

$$\begin{gathered} P(C_1) = \pi_1 \\ P(C_2) = \pi_2 \\ \dots \\ P(\pi_k) = 1 - \sum_{i=1}^{k-1} \pi_i \end{gathered}$$

> **Exam question:**
> *How many independent parameters does a model with k classes have?*
> A model with k classes has k-1 independent parameters, since we just need k-1 parameters in order to computed the k-th one.

Computing the solution for 2 classes

We want to find a way to compute the parameters of our problem. Let's take a step back to the two classes case.

Given the previous assumptions we have $P(C_1) = \pi, P(C_2) = 1 - \pi$. The class-conditional distributions are $P(x|C_1) = \mathcal{N}(x|\mu_1, \Sigma), P(x|C_2) = \mathcal{N}(x|\mu_2, \Sigma)$.

We want to find a solution to computing the parameters μ_1, μ_2, Σ.

The first step is to properly encode the training dataset. Considering the two classes case, our dataset will be a set of pairs in the form $D = \{(x_n, t_n)_{n=1}^N\}$, where t_n is a binary variable that is 1 if $x_n \in C_1$, 0 if $x_n \in C_2$.

The result is a vector $t = [t_1\ t_2\ ...\ t_n]^T$ that encodes every class for every sample.

Now we define a likelihood function:

$$P(t|\pi, \mu_1, \mu_2, \Sigma, D) = \prod_{n=1}^{N} [\pi \mathcal{N}(x_n; \mu_1, \Sigma)]^{t_n} [(1-\pi)\mathcal{N}(x_n; \mu_2, \Sigma)]^{1-t_n}$$

Since we only know D, our goal is to determine the parametes $\pi, \mu_1, \mu_2, \Sigma$ that maximize the likelihood:

$$\pi, \mu_1, \mu_2, \Sigma = argmax_{\pi, \mu_1, \mu_2, \Sigma} P(t|\pi, \mu_1, \mu_2, \Sigma)$$

In order for the problem to be easier to solve, we can formulate it as maximizing the log-likelihood, which means literally to find the argmax for the logaritghm of the previous function (the result of this operation will be the same). In order to compute the argmax we compute the derivative of the function, put it equals to 0 and find the solution for each term.

The estimations for each term will be:

$$\pi = P(C_1) \approx \frac{|\{x, C_1\}|}{|D|}$$

$$\mu_1 = \frac{1}{N_1} \sum_{x_i \in C_1} x_i$$

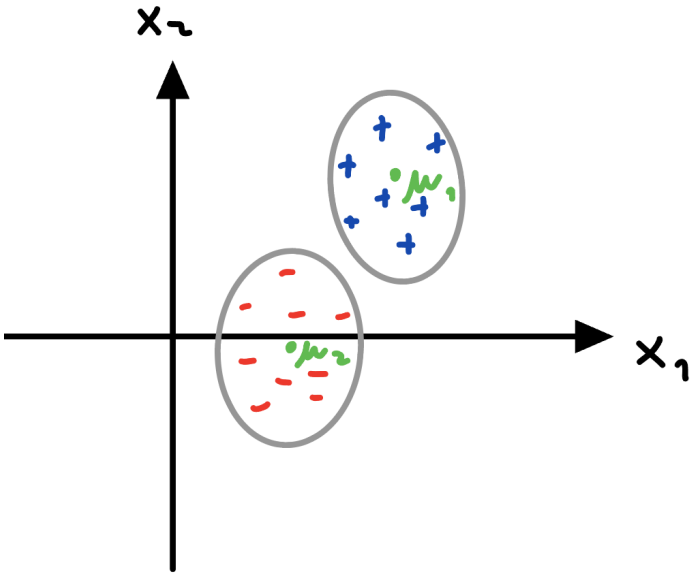
N_1 is the number of samples in class 1 and $\sum_{x_i \in C_1} x_i$ is the sum of samples in class 1.

$$\mu_2 = \frac{1}{N_2} \sum_{x_i \in C_2} x_i$$

$$\Sigma = \frac{N_1}{N} S_1 + \frac{N_2}{N} S_2$$

Σ is the weighted average of the distances of each point from their mean.

The following graph shows the distribution of two samples in 2D:



Notice that the model is robust to noise as long as noisy samples are a minority inside the distribution where they not belong.

> ❗ The model will work as long as samples follow a Gaussian distribution. It will fail in every other case.

A general solution for k classes

In the general case the steps needed to find the solution will be the same, we will just have to adapt the encoding of the training dataset. It will still be represented as a set of pairs $D = \{(x_n, t_n)\}$. But we will use a 1-out-of-k encoding, meaning that for each sample the classes will be encoded as a 1xK vector. There will be a 1 in the position corresponding to the class it belongs, and 0 in every other position:

$$t = [\ ...\ [0\ 0\ 0\ ..\ 0\ 1\ 0..\ 0]\ ...\ ]^T$$

The maximum likelihood will be computed by taking the derivative with respect to each class.

Probabilistic discriminative models

The discriminative approach to probabilistic models is about computing directly the posterior probability of the class given a sample. This means that we won't be able to generate new samples given their distibution, but only to classify them. The reason for this is that we will extend the assumption of studying only Gaussian distributions, but this will make us lose the ability to compute the underlying parameters, making us unable to generate new samples.

This method of classification is called **Logistic Regression.** Notice that even if the name "Regression" is often associated to models in the continuous space, here we will refer to classification problems.

Our first consideration is that posterior probability can be computed not only for Gaussian distribution, but for all exponential ones. In the general case (k>2) the posterior probability will be expressed as a linear combination of the input features:

$$P(C_i|x) = \frac{e^{a_k}}{\sum_j e^{a_j}}$$

With the term $a_k = w^T x + w_0$.

In order to have a more compact notation, we can assign the following terms:

$$w^T x + w_0 = \begin{bmatrix} w_0 & w \end{bmatrix} \begin{bmatrix} 1 \\ x \end{bmatrix}$$

$$\tilde{w} = \begin{bmatrix} w_0 \\ w \end{bmatrix} \quad \tilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$$

So that $a_k = w^T x + w_0 = \tilde{w}^T \tilde{x}$

Since we are not interested in particular parameters, we generically define parametric models as $\mathcal{M}_\Theta := P(t|\Theta, D)$.

The maximum likelihood function will be the set of parameters that will maximize the log-likelihood:

$$\Theta^* = argmax_\Theta \; ln(P(t|\Theta, D))$$

When $\mathcal{M}_\Theta$ belongs to the exponential family, we know that its parameters can be expressed as a linear combination of values, meaning that we can formulate the maximum likelihood as:

$$\tilde{w}^* = argmax_{\tilde{w}} \; ln(P(t|\tilde{w}, x))$$

Two-class logistic regression

In the two classes case we will have $D = \{(x_n, t_n)\}$ with $t_n \in \{0, 1\}$. The likelihood function will be:

$$P(t|\tilde{w}) = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{1 - t_n}$$

Notice that t_n is the actual value of the dataset, while $y_n = P(C_1|\tilde{x}_n) = \sigma(\tilde{w}^T \tilde{x}_n)$ is the posterior prediction of the current model.

The error function for the model is the negative log-likelihood, defined as:

$$E(\tilde{w}) = -ln(P(t|\tilde{w})) = -\sum_{n=1}^{N} [t_n ln(y_n) + (1 - t_n) ln(1 - y_n)]$$

This error function is tipically defined as the *cross-entropy* function.

Our goal will be minimizing the error, namely finding the argmin for the error function:

$$\tilde{w}^* = argmin_{\tilde{w}} E(\tilde{w})$$

> ❗ **Exam question:**
> *What is the relationship between cross-entropy and likelihood?*
> Cross entropy is an error defined as the negative of the log-likelihood.

Solving this minimization problems becomes just a matter of optimization.

A possible approach to the resolution is finding an iterative weightes least squares algorithm. It consists in starting from a random point in space and performing gradient descent until a minimum is found. A main problem with this technique is that we are not guaranteed to find the global minimum, but just a local one.

Multi-class logistic regression

The same concept as before can be extended when we have to predict k classes. We want to find $f : \mathbb{R}^d \rightarrow (C_1, ..., C_k)$.

We will use a 1-out-of-k encoding to represent the output of the dataset. This will result in a matrix made by row vectors.

The procedure will be the same: encode training inputs and outputs, define a likelihood function and solve the resulting optimization problem.

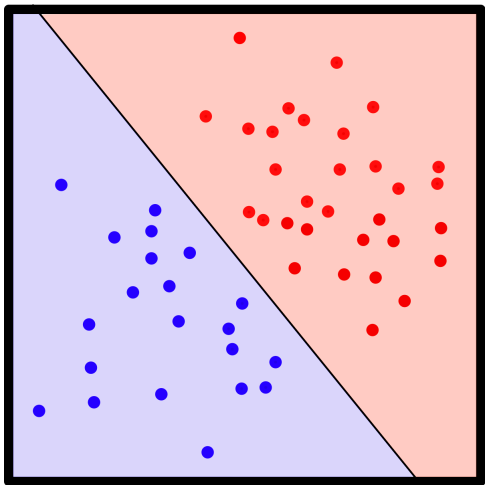
A really useful notion about this method is that it can be applied for any transformed space of the input (feature space).

Linear models for classification

Introduction

The main asssumption when working with linear models is that the samples in the dataset have the property of being *linearly separable:*

We say that a dataset is linerarly separable if exists a linear function that separates the space in two or more region, where each region belongs to a certain class.



An example of a linearly separable dataset

The linear separator function can be a line in a 2-D case, while in generale it is a d-dimensional hyperplane when dealing with d-dimensional spaces.

An ideal resolution for this kind of problem is defining a linear model and train its parameters in order to predict new sample based on the region of space they belong.

The k-classes case

In the k-classes case we will define k linear combinations (one for each class) such that:

$$y_1(x) = w_1^T x + w_{10}$$
$$...$$
$$y_k(x) = w_k^T x + w_{k0}$$

In order to simplify the notation we define $\tilde{w} = \begin{bmatrix} w_0 \\ w \end{bmatrix}$ and $\tilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$.

We can now express the model we want to learn as a column vector made by the k models:

$$y(x) = \begin{bmatrix} y_1(x) \\ ... \\ y_k(x) \end{bmatrix} = \begin{bmatrix} w_1^T x + w_{10} \\ ... \\ w_k^T x + w_{k0} \end{bmatrix} = \begin{bmatrix} \tilde{w}_1^T \\ ... \\ \tilde{w}_k^T \end{bmatrix} \tilde{x} = \tilde{W}^T \tilde{x}$$

Where we $\tilde{W}^T$ is the matrix that contains all the parameters for our model:

"

The goal of machine learning is to find the optimal parameters for $\tilde{W}$.

It is very important to keep in mind that a linear model is valid as long as the parameters are linear, meaning that for example $y(x) = w_1 x_1 + w_2 x_2^2 + w_3 x_3^3 + w_0$ is still a linear model.

Some bad classifiers

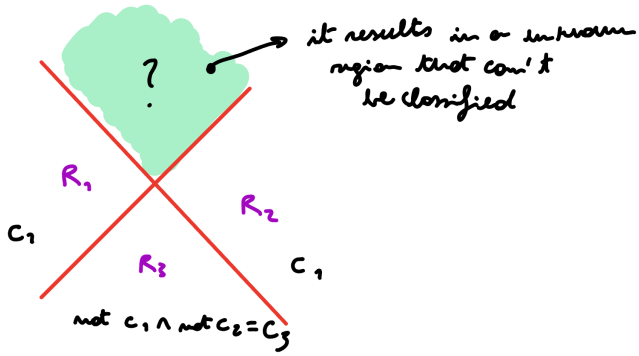
When dealing with this problem we might think about a solution that involves binary classifiers, namely classifiers that tell apart one class from the other. The following examples show why it is a bad idea.

One-versus-the-rest classifier

Here we use k-1 classifiers in order that tell apart the i-th class from all the others. This basically means that for example the classifier for class C1 just knows whether

a sample belongs to class C1 or not.

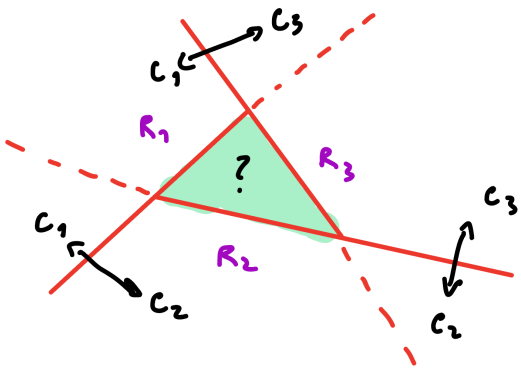
There is a major logic fault with this approach: there is always an unknown region that does not belong to any class, and it will cause conflict during the decision. The following is the example with three classes:



What class is the region that doesn't belong to any class?

One-versus-one classifier

This approach defines a classifier for each possible pair of classes to tell apart each class from the other. We use $\frac{k(k-1)}{2}$ classifiers. This approach still produces an uknown region at the inner intersection of the regions:



The right approach

The right approach to this problem is to define our classifier as a compound of k linear models:

$$y(x) = \begin{bmatrix} y_1(x) \\ ... \\ y_k(x) \end{bmatrix} = \begin{bmatrix} \tilde{w}_1^T \\ ... \\ \tilde{w}_k^T \end{bmatrix} \tilde{x} = \tilde{W}^T \tilde{x}$$

Each new sample will be classified according to the class that will produce the highest value for predictions, namely x is classfied as C_k if $y_k(x) > y_i(x)\ \forall i \neq k$.

The result is that a k-class discriminant will divide the space in a correct way. The following models show different solutions starting from this approach.

Least squares

Our input dataset is defined as $D = \{(x_n, t_n)_{n=1}^N\}$ implementing a 1-out-of-k encoding.

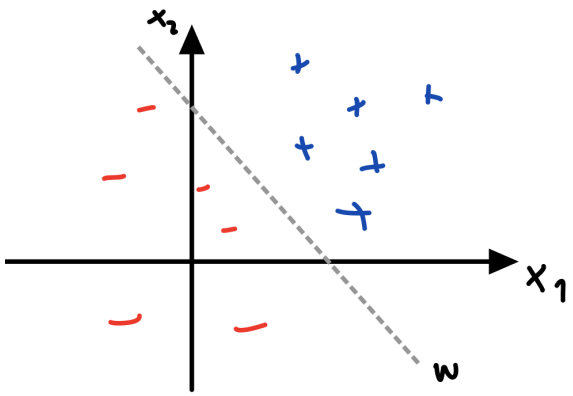
The error function for this model is defined as the sum of the squared errors, where the error is represented as the distance between the prediction $\tilde{X}\tilde{W}$ and the truth T :

$$E(\tilde{W}) = \frac{1}{2} Tr\{(\tilde{X}\tilde{W} - T)^T (\tilde{X}\tilde{W} - T)\}$$

The operator Tr stands for the trace of the matrix, namely the sum of the elements on its main diagonal (thus the sum of the squared errors). The constant 1/2 is there just for the purpose of being simplified when computing the derivative of the squared error.

Our goal is to have a zero distance between the prediction matrix and the true one, meaning that we want them to have the same values.

The result will be a linear separator parametrized by W:



The solution for finding the optimal parameters is:

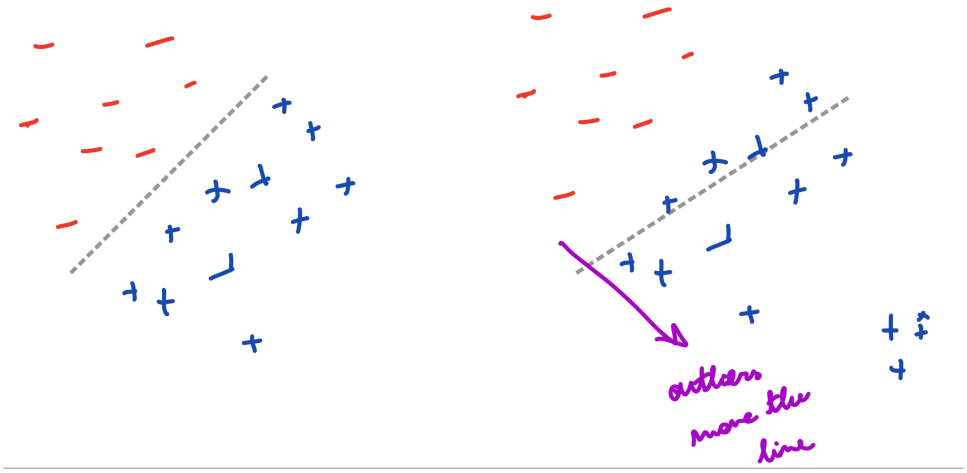
$$\tilde{W} = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T T$$

Where the term $(\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T$ is the pseudo inverse of $\tilde{X}$. Once the solution is computed, we will use the obtained weights inside our model to make predictions:

$$y(x) = \tilde{W}^T \tilde{x}$$

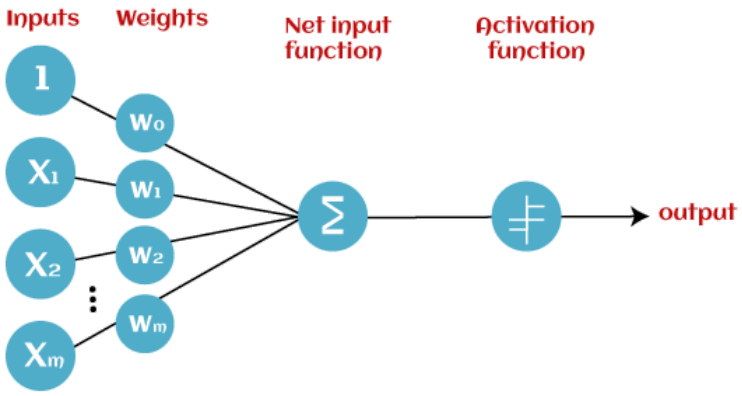
And the predicted class will be $C_k = argmax_{i=1,\dots,k}(y_i(x))$.

The fundamental problem with the least squares method is that the linear separator depends only on the distance error, meaning that outliers will tend to move the line towards them, making the entire model less performant:



Perceptron

A perceptron unit is made by pipeline of two operations. The first one is computing weighted linear combination of the inputs, which is then fed to a sign function:



The first operator is $\Sigma = \sum_i w_i x_i$, while in this case the activation function is the sign function, which takes the weighted sum as an input, and maps its values to -1 if the sum is negative, 1 otherwise.

The whole model can be summarized as $y(x) = sign(W^T X)$.

Let's now take a step back and consider the unthresholded linear unit:

$$o = w_0 + w_1 x_1 + ... + w_n x_n$$

Given a training dataset $D = \{(x_n, t_n)_{n=1}^N\}$, we define the error function as the sum of the squared errors:

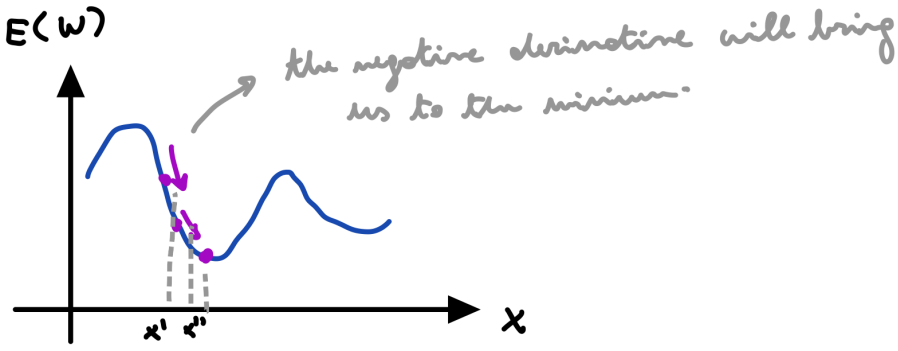
$$E(W) = \frac{1}{2} \sum_{n=1}^{N} (t_n - o_n)^2 = \frac{1}{2} \sum_{n=1}^{N} (t_n - w^T x_n)^2$$

In order to minimize the loss function we want to satisfy $E(w) \iff T = W^T X$.

In order to find the minimum of the error function we start by computing its gradient:

$$\frac{\delta E}{\delta w_i} = ... = \sum_{n=1}^{N} (t_n - w^t x_n)(-x_{i,n})$$

The negative gradient will show us the direction of maximum decrease, that we will follow using an iterative approach in order to reach the minimum.



For each step we update the weights by summing a certain delta, then we compute the new derivative:

$$w_i \leftarrow w_i + \Delta w_i$$
$$w_i = w_i - \eta \frac{\delta E}{\delta w_i}$$

Notice that we introduce the term η, that is a constant called the **Learning Rate**. The choice of the value for the learning rate is fundamental, since it represents by how much we move towards the negative direction. We will see better how this parameter will affect the computation of our model.

Now we have to pay attention to the composition of the delta term:

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i} = -\eta \sum_{n=1}^{N} (t_n - w^T x_n) x_{i,n}$$

The two terms of the subtraction $(t_n - w^T x_n)$ are incompatible, since $t_n \in \{0, 1\}$ while $w^t x_n \in \mathbb{R}$. In order to fix this, we can just use the sign function in order to map the real values of the second term to either 1 or -1. The resulting formula will be:

$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = w_i - \eta \frac{\delta E}{\delta w_i} = \eta \sum_{n=1}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$$

This formulation makes the computation of the model really robust to outliers, since even extreme values will be mapped to 1 or -1. The error will be 0 when the line will perfectly separate the data.

Implementation

When feeding the data to the training phase, we have three possible approaches:

- **Batch mode:** we feed the whole datase $\Delta w_i = \eta \sum_{(x,t) \in D}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$
- **Mini-batch mode:** for each iteration we take a small subset $S \subset D$ $\Delta w_i = \eta \sum_{(x,t) \in S}^{N} (t_n - \text{sign}(w^T x_n)) x_{i,n}$
- **Incremental mode:** we feed one sample per step

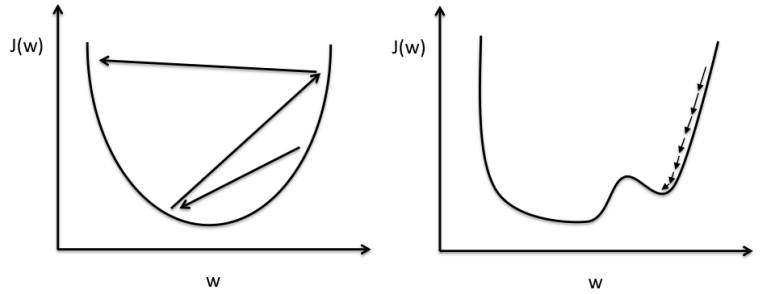
The best approach happend to be the mini-batch mode, since batch mode can be too computationally expensive, while incremental mode becomes slower as the dataset gets larger.

Moreover, we need to consider these two conditions needed for the convergence of the algorithm:

- Data is linearly separable.

- The learning rate is sufficiently small.

The choice of learning rate is in fact crucial for the convergence of our algorithm. In fact if η is too large, we risk make large steps and missing the minimum, often resulting in an oscillation that will lead to divergence:



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

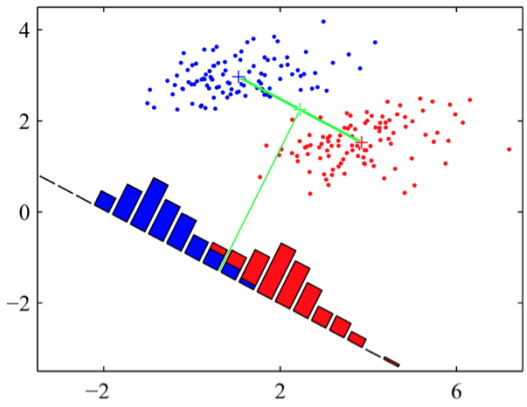
The learning rate will also affect the ability of our model of splitting the regions of our space. In fact for small values of η the linear separator will accumulate next to one of the classes, resulting in overfitting.

The termination conditions for our algorithm will be two:

- When the error is zero: this is the optimal solutions, but not always reachable.
- We define a thresholds for the changes of the loss function. When the following step will produce a change smaller than the threshold, we will stop.

Fisher's linear discriminant

The core idea about Fisher's linear discriminant is to place the linear separator in the middle of the line that connects the centers of mass (means) of the distribution:

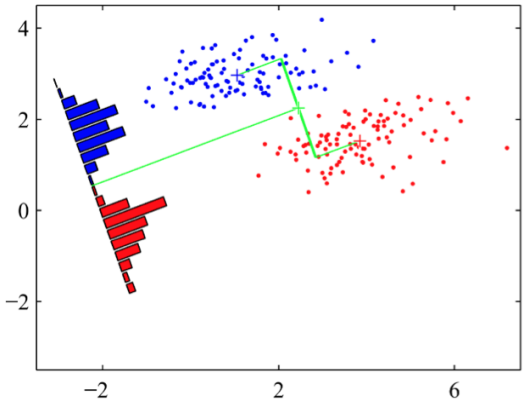


Let's consider a dataset with N_1 elements belonging to class C_1 and N_2 elements of class C_2. Their means will be:

$$m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n \quad m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n$$

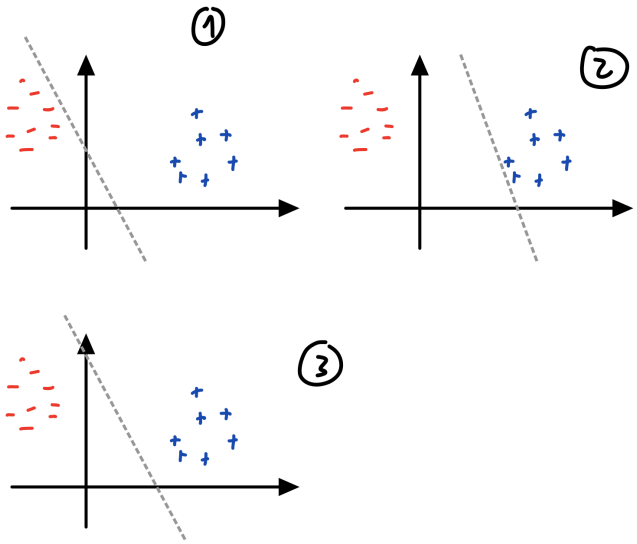
The separation of the two classes will be expressed as $J(W) = W^T(m_2 - m_1)$ and our goal will be to find the parameters for W that maximize separation.

The current solution has the major flaw of not taking into account the covariance of the two distributions. A straight forward solution is to introduce an additional parameters to the optimization problem in the form of a rotation matrix. The result will be a rotation of the linear separator, that will provide a better split for the two regions:



Support Vector Machines (SVMs)

Exam question:
Let's suppose to have the following three solutions: which one is better?
At first sight we might think that they are all equal, since they perfectly partition the dataset. But actually the better solution is the third one, since it will be more likely for it to classify new instances because it divides the space more evenly.



Support Vector Machines were born with the idea of addressing the problem of finding the best division in space in order to provide better accuracy.

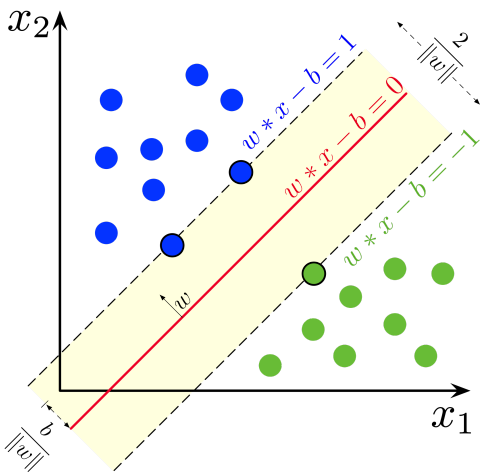
The core concept for this model lies into the definition of margins. A margin is the distance between the closest point of the dataset and the linear separator. Let's now analyze the problem.

We consider a function $f : X \to \{+1, -1\}$ and a dataset $D = \{(x_n, t_n)_{n=1}^N\}$ with $t_n \in \{+1, -1\}$.

We want to find a linear model $y(x) = w^T x + w_0$. Assuming thay D is linearly separable, there exists a linear combination of parameters such the resulting line separates the data:

$$\exists\, w, w_0 \text{ such that } y(x_n) \begin{cases} > 0 \text{ if } t_n = +1 \\ < 0 \text{ if } t_n = -1 \end{cases}$$

We define the margin of a point as $\frac{y(x)}{||w||}$, namely the distance between a cetain point and the separator.



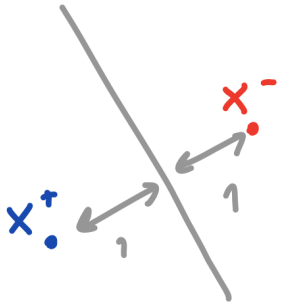
Using the concept of margin, we will define our optimization problem in the terms of maximizing the margin from the closest point to the line, namely:

$$min_{n=1,...,n} \frac{|y(x_n)|}{||W||} = ... = \frac{1}{||W||} min_{n=1,...,N} [t_n(\bar{w}^T x_n + \bar{w}_0)]$$

Minimizing this quantity means finding the hyperplane $h^* = w^{*T} x + w_0$ that will maximize the margins. The parameters that will satisfy this conditions are computes as:

$$w^*, w_0^* = argmax_{w,w_0} \frac{1}{||W||} min_{n=1,..,N} [t_n(w^T x_n + w_0)]$$

A fundamental step before computing the solution is the normalization of the dataset. This means to scale all the points such that the minimum margin for the closest point is greater or equal than 1. Once normalized, we will have that we will have at least two samples (one for C1, the other for C2) having a margin of 1.



We will define these two points as x_k^+ and x_k^- and they will hold the following property:

$$w^{*T}x_k^+ + w_0 = +1$$
$$w^{*T}x_k^- + w_0 = -1$$

The problem of computing the maximum margin can by simplified by computing the result with respect just to these two points. The optimization problem is formulated as:

$$w^*, w_0 = argmax\frac{1}{||W||} = argmin\frac{1}{2}||W||^2$$

The solution to this problem can be computed using the Lagrangian Multipliers method:

$$w^* = \sum_{n=1}^{N} a_n^* t_n x_n$$

The terms a_n^* are named Lagrange multipliers, and they will be our unknown term. In order to find them we apply this function:

"

The formulation of this solution is effective, because it exploits a really important property:

The Karush-Kuhn-Tucker condition states that for each $x_n \in D$, the corresponding Lagrange multiplier will be 1 if the point has margin 1 ($t_n y(x_n) = 1$) or 0 otherwise.

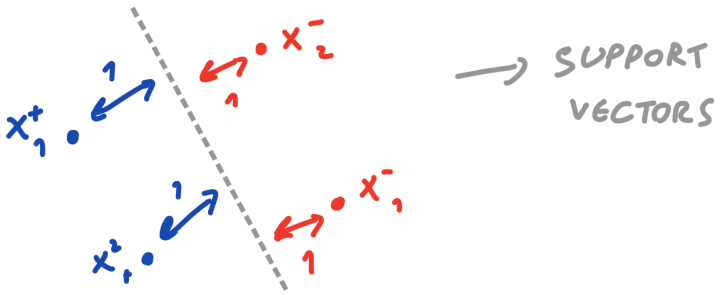
Since the optimization problem is computer as a product of terms, each time the multiplier $a_n^* = 0$, the corresponding term will be 0 and not contribute to the solution. This means that the computation will consider just a small subset of all the training dataset.

The terms that will contribute to the solution are called Support Vectors, and they are formally defined as:

$$SV = \{x_k \in D | t_k y(x_k) = 1\}$$

Therefore, hyperplanes will be describer by support vectors:

$$y(x) = \sum_{x_j \in SV} a_j^* t - jx^T s_k + w_0 = 0$$



In order to compute w_0^* we pick any support vector and solve the equation:

$$w_0^* = t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j$$

Actually, a better solution that is more robust to noisy data is to compute w_0^* with respect to the means of all support vectors:

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} (t_k - \sum_{x_j \in SV} a_j^* t_k x_k^T x_j)$$

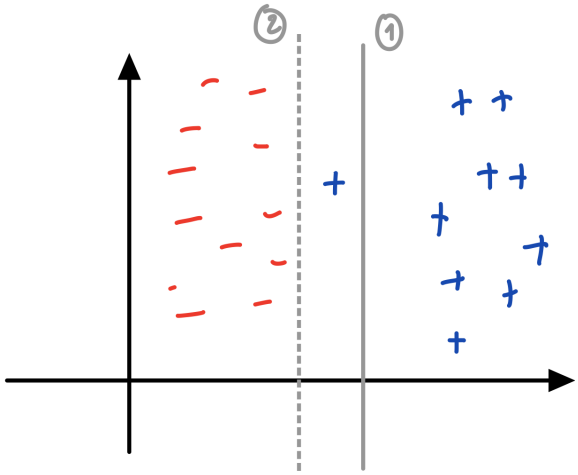
Once we find the maximum margin hyperplante, every new instance will be classified by applying the model

$$y(x') = sign(\sum_{x_k \in SV} a_k^* t_k x'^T x_k + w_0^*)$$

Support Vector Machines with soft margin constraints

We said previously that SVMs are strongly robust to noise produce by outliers, since every point that is outside the margin will give a 0 contribute to the solution of the optimization problem.

Actually, there is another kind of noises that causes problems with SVMs, that is due to points that are too close to the separator with respect to the distribution of their class. Let's see an example:



The result that we would get using the current formulation of the problem will be a separator close to line number 2. But is this the optimal solution? Actually not, because even if line 2 correctly separates the data, a solution like number 1 would be ideal, since it would maximize the probability of correctly classifying new instances by dividing the space more evenly.

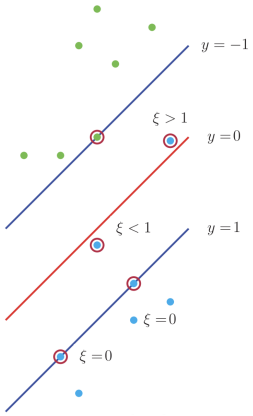
In order to get better performances we can think about relaxing the original constraints of our optimization problem by accepting some exceptions. Remember that our original condition is that every point must have a margin greater or equal that one: $t_n(w^T x + w_0) \geq 1 \ \forall n$.

The idea here is to introduce a slack variable $\xi_n \geq 0$ for each data point, thus allowing for points that have margins less than 1:

$$t_n y(x_n) \geq 1 - \xi_n \ \ \forall n$$

The values for slack variables will follow this logic:

- $\xi_n = 0$ if the point lies on the margin or in the right side of the boundary.
- $0 < \xi_n \leq 1$ if the point lies between the separator and the margin.
- $\xi > 1$ if the point lies on the wrong side of the boundary.



The balance for this will be introducing some penalties to our optimization solutions for points that violate the margin:

$$w^*, w_0* = argmin \frac{1}{2}||W||^2 + C\sum_{n=1}^{N}\xi_n$$

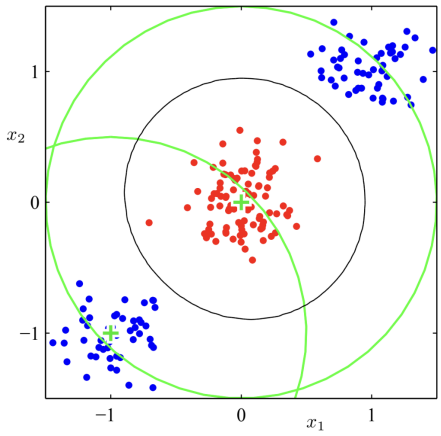
Since we are seeking the argmin, the added term will clearly penalize the terms with higher values for the slack variables. Moreover, we introduced the term C, a coefficient that express how much we want the penalization to impact the result.

Finally, the solution can still be computed using the methods explained before (the lagrangian solution and the KKT condition are still valid).

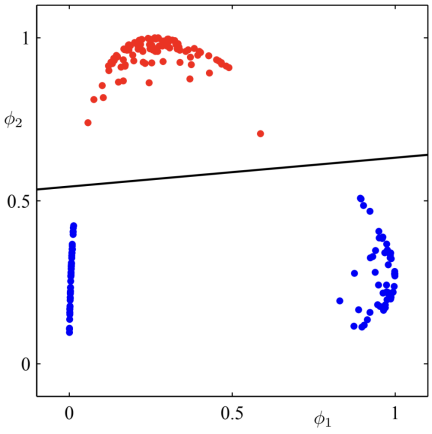
When the dataset is not linearly separable

We already introduced the fact that linear classfication problems works also if the input space is transformed. This means that even if the dataset might not be linearly separable, we can still try to apply some basis transformation in order to make it work.

Let’s assume to have the following dataset:

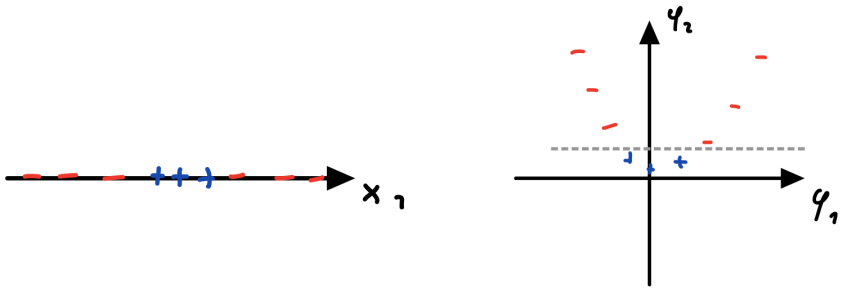


It is clearly non linearly separable. But if we apply a change of basis to its coordinates, expressing it with polar ones, we obtain the following representation:



We obtained a linearly separable representation of the dataset!

Another important property is that our method will still work even if the transformation changes the dimensions of the initial input space, both augmenting or reducing it. For example, we might think to transform data that lies in 1D into a 2D representation of a quadratic curve:



In general, given a dataset $D = \{(x_n, t_n)_{n=1}^N\}$ with $x_n \in \mathbb{R}^d$, we can define a transformation

$$\phi(x) = \begin{bmatrix} \phi_0(x) \\ ... \\ \phi_M(x) \end{bmatrix}$$

with $M \neq d$. This transformation will likely ontroduce nonlinearities. This will result into a transformation of our dataset where $x_n \rightarrow \phi_n = \phi(x_n)$. Therefore we will procude with our newly transformed dataset $D_\phi = \{(\phi_n, t_n)\}$.

For every new instance x', we will proceed transforming it $\phi' = \phi(x')$ and then classifying it with the model.

There are many function for the transformation of basis, the approach is to train them and evaluate which one performs better.

All of those considerations are still valid in the multi-class case, where we just consider the formulation with k linear models, where $y_k(x) = W^T\phi(x)$.

Linear models for regression

Introduction

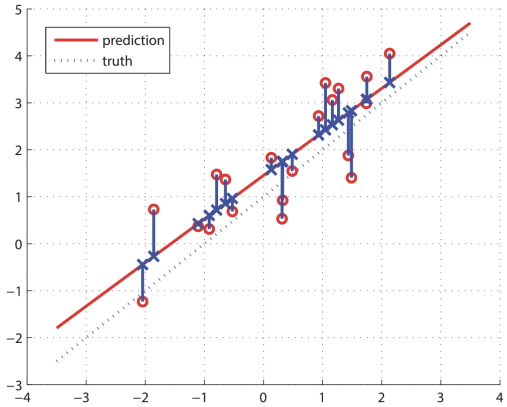
Linear models for regression aim to finding models able to predict real values. Formally a linear model is a function in the form $f : \mathbb{R}^d \rightarrow \mathbb{R}$.

Given a dataset $D = \{(x_n, t_n)_{n=1}^N\}$ $t_n \in \mathbb{R}$, we define a linear model as:

$$y = w_0 + w_1 x_1 + ... + w_d x_d = \begin{bmatrix} w_0 & w_1 & ... & w_d \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ ... \\ x_d \end{bmatrix} = W^T X$$

We can write an explicit formula for the model as $y(x; w) = W^T X$.

When d=1, we will have a one dimensional input space, and a two dimensional representation of the dataset. In this case, the linear model will be a line represented as $y(x; w) = w_0 + w_1 x_1$. The goal of regression is to compute an approximation of the real function that will maximize the prediction (hence, minimize the error).



The following representation shows the data points and the prediction model (the red line). The error is expressed as the distance of each point from the prediction model.

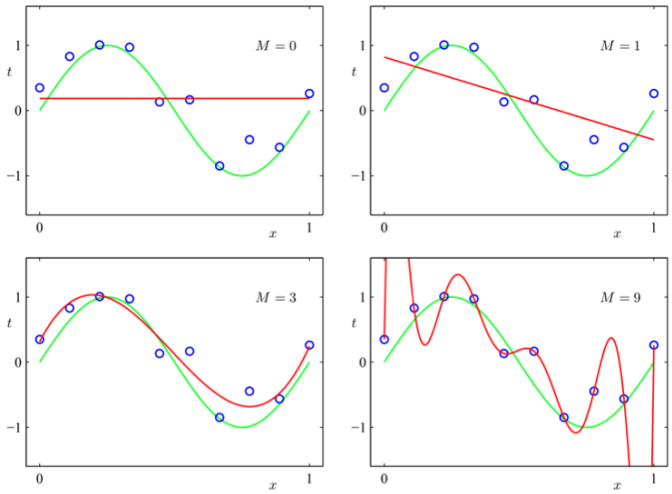
When considering this problem, transformation of the input space are allowed, since we will still keep the model linear with respect to the parameters. Therefore, we can defines a set

of nonlinear transofrmations as:

$$\phi(x) = \begin{bmatrix} \phi_0(x) \\ ... \\ \phi_M(x) \end{bmatrix} \rightarrow y(x;w) = W^T \phi(x)$$

For example, let's consider the transformation $\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ ... \\ x^M \end{bmatrix}$.

Our model will be an M-degree polinomial $y(x;w) = w_0 + w_1 x + w_2 x^2 + ... + w_M x^M$ that we can try tuning in order to get as close as possible to the real function. The following image show an important characteristic of this approach:



As we can see, the more parameter we use, the more we will risk the phenomena of overfitting. In fact we will progressively reduce the error on training dataset, but we will have poor generalization for new predictions.

Computing the model

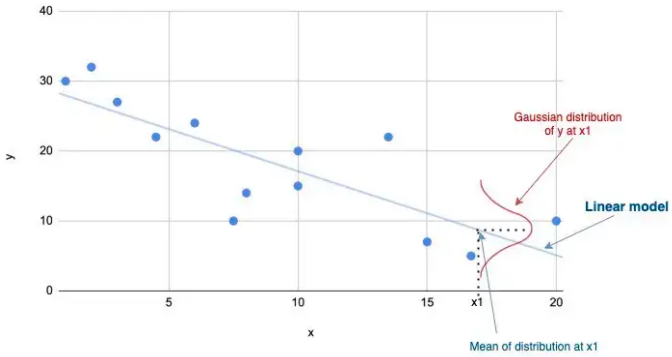
The probabilistic approach

We start by assuming that points of the training dataset come from the true function plus some noise:

$$t = y(x;w) + \epsilon$$

The logic is that we assume that there is a true function, and each point of the dataset correspond to the points of that function plus some noise error generated by a certain

distribution:



Noise will be generated by a zero-mean Gaussian noise distribution:

$$P(\epsilon|\beta) = \mathcal{N}(\epsilon|0, \beta^{-1})$$

Where the term β is called precision (the inverse of the variance). Given our model, we ask ourselves: what is the probability of having that points in the dataset (likelyhood)? This probability is defined as:

$$P(t|x, w, \beta) = \mathcal{N}(t|y(x; w), \beta^{-1})$$

Where $y(x; w)$ will be the mean of the Gaussian. What we want is maximizig the likelihood, aka minimizing the negative log-likelihood. The following is the formulation for this likelihood, obtained assuming that each input is indipendent and identically distributed:

$$P(\{t_1, ..., t_N\}|x_1, ..., x_N, w, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|w^T\phi(x_n); \beta^{-1})$$

The negative log-likelihood will be expressed as

$$lnP(\{t_1, ..., t_N\}|x_1, ..., x_N, w, \beta) = .. = -\beta\frac{1}{2}\sum_{n=1}^{N}[t_n - w^T\phi(x_n)]^2 - \frac{N}{2}ln(2\pi\beta^{-1})$$

Notice that the only term that we care about (the one that contains the parameters) is $\sum_{n=1}^{N}[t_n - w^T\phi(x_n)]^2$, so we can reduce the problem as finding

$$argmaxP(\{t_1, ..., t_N\}|x_1, ..., x_N, w, \beta) = argmin\frac{1}{2}\sum_{n=1}^{N}[t_n - w^T\phi(x_n)]^2$$

That corresponds to minimizing the squared error between the actual data and our model.

Computing the solution: the closed-form approach

The closed-form solutions aims at solving the error by directly computing the solution for the parameters matrix. We start defining the error:

$$E_D(w) = \frac{1}{2}(t - \phi_w)^T(t - \phi_w)$$

We will solve the problem by computing the gradient of the error and putting it to zero:

$$\Delta E_D(w) = 0 \iff \phi^T \phi W = \phi^T t$$

Hence

$$W_{ML} = (\phi^T \phi)^{-1} \phi^T t$$

Computing the solution: the iterative approach

Ideally the closed-form approach already computes the correct solution. The problem with it is that we are trying to solve the problem for the whole ϕ matrix, which means risking to have really bas performances in term of computational time when dealing with huge dimensions.

The iterative approach addresses this problem by computing a step by step solution, by finding each time the negative direction for the gradient and updating the weights. This method is called stochastig gradient descent:

$$\hat{w} \leftarrow \hat{w} + \eta \Delta E_w$$

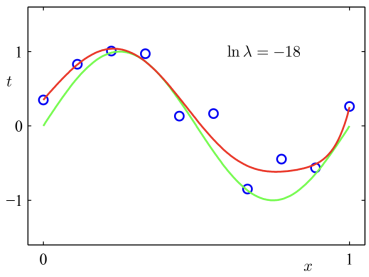
Where η is the learning rate (the algoritm converges when the learning rate is small enough). Therefore

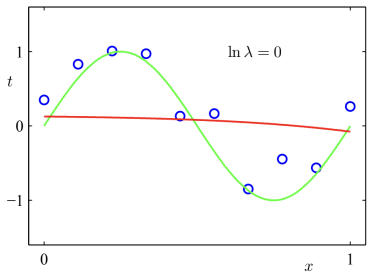
$$\hat{w} \leftarrow \bar{w} + \eta[t_n - \hat{w}^T \phi x_n]\phi(x_n)$$

We still have to address the problem of overfitting. This is why we introduce the regularization term λ. Its function will be to penalize the coefficients that are too high independently from the samples in the dataset. Hence, we are interested in:

$$argmin[E_D(w) + \lambda E_w(w)]$$

A common choice is $E_w = \frac{1}{2} W^T W$. Notice that if the parameter is not well tunes, we risk to reach an underfitting region.





Regression with multiple outputs

Let's assume the case where we have a d-dimensional input and a k-dimensional output $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$. The output will be a vector of size k: $y = [y_1 \ ... \ y_k]^T$ made by k linear models.

Since we are dealing with a k-dimentional output, also the target attributes will be expressed as an NxK matrix in the form:

$$T = \begin{bmatrix} ... \ t_n \ ... \\ ... \end{bmatrix}$$

Similarly to the 1D case, we will obtain the maximum likelihood for:

$$W_{ML} = (\phi^T \phi) \phi^T T$$

Instance based learning

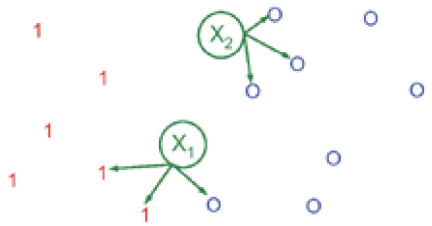
Introduction

Here we introduce non-parametric models. The core idea of instance based learning is to make predictions without an explicit formulation of the output function we are trying to guess.

Fundamentally, we will not work with some models, but with algorithms that directly compute predictions given a dataset and a new instance.

K-Nearest-Neighbours (K-NN)

KNN is one of the most popular algorithms. The idea is simple: given a datasetd D, a parameter k and a new instance x', we individuate the k nearest points to that new instance. The output prediction will correspond to the most frequent class present in those k neighbours.



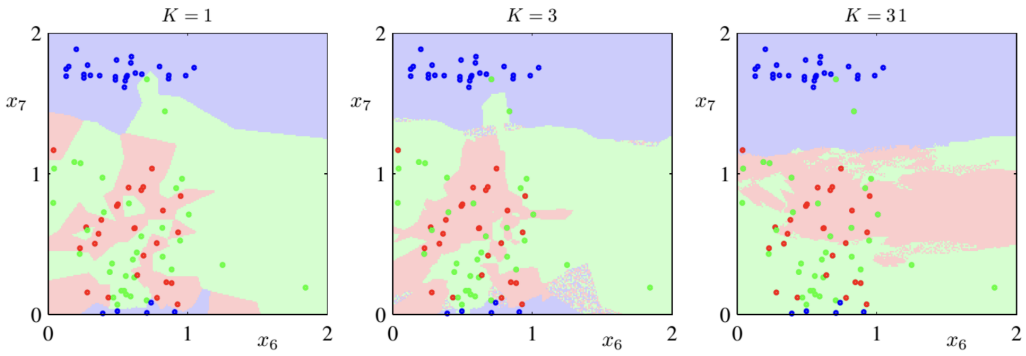
$K = 3$

Given a new instance, the likelihood for class C is:

$$P(C|x', D, k) = \frac{1}{k} \sum_{x_n \in N_k(x_n, D)} I(t_n = C)$$

With $N_k(x_n, D)$ being the k nearest points to x' and $I(t_n = C)$ a function that outputs one if the condition is true, 0 otherwise.

In general, the bigger is k, the smoother the separation surface will be. This will imply that for bigger number of k we will tend to reduce the probability of overfitting.



Despite being so intuitive, KNN has two main problems. The first one is storage: in fact in order to make new predictions, we will always have to consider the whole dataset. Another problem might be finding a proper distance function, that is not always easy to define.

Kernelized KNN

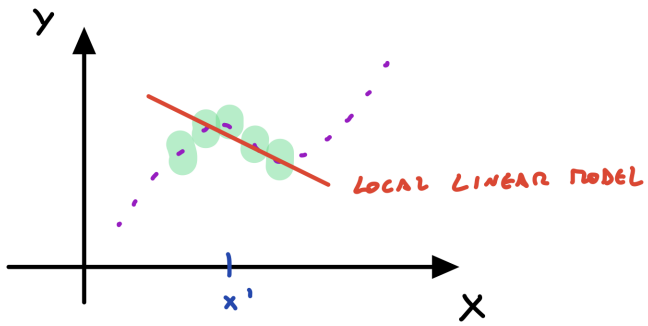
When computing $N_k(x_n, D)$ we can define the distance function as follows:

$$||x - x_n||^2 = x^T x + x_n^T x_n + 2x^T x_n$$

Notice how this form allows the application of the kernel trick.

Locally weighted regression

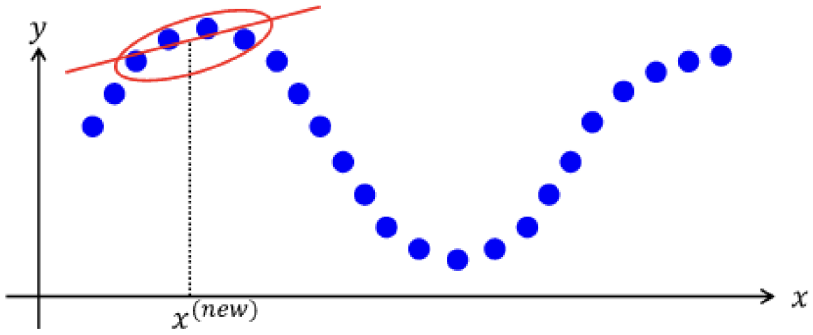
The concept of KNN can be extended also to the problem of regression. Given a dataset and a new instance, we can try to guess it by creating a local linear model that considers only the k-nearest points and make a prediction:



For example with k=5, we consider the 5 closest points to the new instance x' and fit a linear model on these 5 neighbours.

The actual steps are:

1. Compute $N_k(x', D)$
2. Fit a regression model $y(x; w)$ on $N_k(x', D)$
3. Return $y(x', w)$.



Kernel methods

What are kernels?

In the previous chapters we have been dealing with input spaces of known and fixed dimensions such as $X \in R^M$, but in real situation this is not always the case. In face we often have to deal with input spaces of variable length and possibly infinite dimensions like strings, image features etc.

This is when kernel functions come into play. Fundamentally they are functions that given two inputs, return a real value that represents theyr similarity. The following is a formal definition of kernels:

A kernel is a real-valued function $k(x, x') \in \mathbb{R}$ for $x, x' \in \mathcal{X}$ (where $\mathcal{X}$ is some abstract space).

Kernels tipycally satisfy this conditions:

- They are symmetrical: $k(x, x') = k(x', x)$
- They are non-negative: $k(x, x') \geq 0$

The more the two inputs are similiar, the more the output value of the kernel will be close to zero.

The first thing to do when using kernel function is to normalize our dataset. In fact we want our transformations to be independent from the measure units of our inputs. For example the same dataset that uses a measure in centimeters will be scaled differently with respect to the same dataset measured in meters if they are not normalized. Instead, we want to preserve the concept and not care about the scale, thus making our solution independent from the measures.

The most common normalizetion techniques are:

- Min-max: $\bar{x} = \frac{x - min}{max - min}$

- Normalization (or standardization): $\bar{x} = \frac{x-\mu}{\sigma}$, where μ is the mean and σ is the standard deviation of the dataset.

Some common kernel families are:

- Linear: $k(x, x') = x^T x'$
- Polynomial: $k(x, x') = (\beta x^T x' + \gamma)^d \;\; d \in \{1, 2, 3, ...\}$
- Radial basis function: $k(x, x') = e^{-\beta |x - x'|^2}$
- Sigmoid: $k(x, x') = tanh(\beta x^T x' + \gamma)$
- *In general we can define any custom kernel that best fits our data.*

How to use kernels

Let's start by considering the generic definition of a linear model $y(x; w) = W^T X$. Our goal is to minimize the loss function:

$$J(w) = (t - x_w)^T (t - x_w) + \lambda ||W||^2$$

We know that the optimal solution for our weights is:

$$\hat{w} = X^T (X X^T + \lambda I_N)^{-1} t$$

In order to simplify notation, we define $\alpha = (X X^T + \lambda I_N)^{-1} t$. Now the optimal solution can be expressedas a linear combination of terms:

$$\hat{w} = X^T \alpha = \sum_{n=1}^{N} \alpha_n x_n$$

Hence, the solution for our model will be:

$$y(x; \hat{w}) = \hat{W}^T X = \sum_{n=1}^{N} \alpha_n \mathbf{x_n^T x}$$

Notice that we highlited the term $\mathbf{x_n^T x}$ beacuse this can actually be subsituted by the kernel function $k(x, x') = x^T x$. Applying the kernel function, we will obtain the following result:

$$y(x; \hat{w}) = \hat{W}^T X = \sum_{n=1}^{N} \alpha_n k(x_n, x)$$

The huge benefit of using a kernel for our model is that once we computed the solution it will be independent from the dimensions of the input.

The kernel trick

As long as the input vector x appears in our algorithm only in the form of an inner product $x^T x$, we can replace that term with any kernel $k(x, x')$.

Kernelized SVMs

Recalling the definition of Support Vector Machines, we know that our model is formulated as:

$$y(x; \alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n \mathbf{x_n^T x})$$

Having our noticeable term, we know that we can apply the kernel trick:

$$y(x; \alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n k(x_n, x))$$

Subsequently, we can express our Lagrangian optimization problem as:

$$\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{M} a_n a_m t_n t_m \mathbf{k(x_n, x_m)}$$

Hence, the solution will be

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} (t_k - \sum_{x_j \in SV} a_j^* t_k \mathbf{k(x_k, x_j)})$$

Why use kernelized SVMs?

The solution for the classical formulation of the linear model without SVMs implies computing the Graham matrix $K = X^T X$, which grows quadratically with the size of our datasets, since it computes the product for every possible pair of inputs

(grows as $|D|^2$). By contrast, we know that SVMs only require the support vectors of the dataset, that will be a small subset of D.

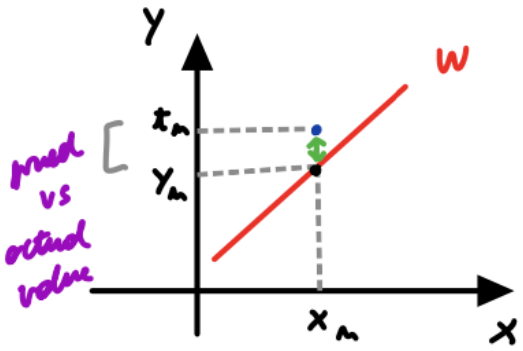
Kernelized regression

Let's start considering the classical formulation of linear regression $y(x) = W^T X$. We define our loss function as:

$$J(w) = \sum_{n=1}^{N} E(y_n, t_n) + \lambda ||W||^2$$

The idea is to achieve a formulation of the model where we can apply the kernel trick.

At the moment the formulation of the error is the quadratic distance between the predicted value and the actual one $E(y_n, t_n) = (y_n - t_n)^2$:



The solution for the current formulation is:

$$\hat{w} = X^T (X^T X + \lambda I_N)^{-1} t = X^T \alpha$$

Our model will be $y(x; \hat{w}) = \sum_{n=1}^{N} \alpha_n \mathbf{x_n^T x}$

Notice that this form admits the application of the kernel trick. Hence, the kernelized linear model is

$$y(x; \hat{w}) = \sum_{n=1}^{N} \alpha_n \mathbf{k}(\mathbf{x_n}, \mathbf{x})$$

Notice that the actual formulation of the problem still suffers the fact that it's computationally heavy due to the Graham matrix computation.

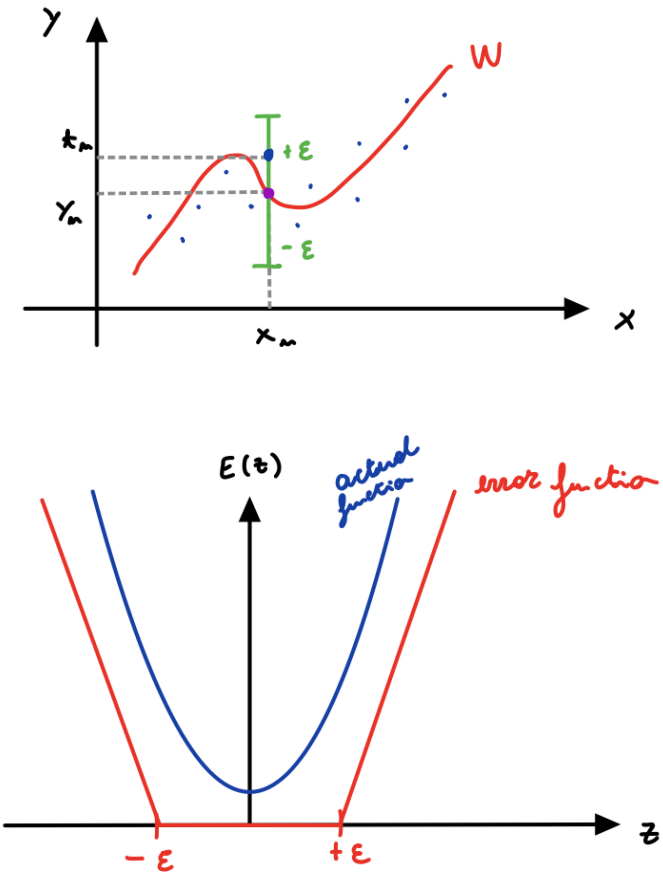
Kernelized SVMs for regression

A less computationally expensive approach would be applyin the concept of SVMs to regression.

Our first step will be defining the error function as an ϵ-insensitive linear function:

$$E_\epsilon(y,t) = \begin{cases} 0 \text{ if } |y-t| < \epsilon \\ |y-t| - \epsilon \text{ otherwise} \end{cases}$$

The idea here is that we define an interval $(y-\epsilon, y+\epsilon)$ such that if the real value falls inside this interval, the error will be zero. Substantially, we are making the error less sensitive to noise.

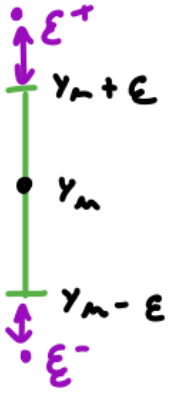


The new formulation for our loss function will be:

$$J(w) = C\sum_{n=1}^{N} E_\epsilon(y_n, t_n) + \frac{1}{2}||W||^2$$

We still have a major problem given by the current formulation of the error. In fact it's easy to notice that the error function is not differentiable, meaning that computing the result would be very difficoult. We have to think about a manipulation aimed at solving this problem.

We introduce two slack variables $\xi_n^+, \xi_n^- \geq 0$. They will measure by how much a point falls outside the admitted interval. In particular their value will be 0 if the points falls inside the interval, otherwise they will return the distance between the point and the edge of the interval.



By using this formulation, we can express the loss function by just taking in account the slack variables:

$$J(w) = C\sum_{n=1}^{N}(\xi^+ + \xi^-) + \frac{1}{2}||W||^2$$

With the constraints:

$$\begin{gathered} t_n \leq y(x_n; w) + \epsilon + \xi^+ \\ t_n \geq y(x_n; w) - \epsilon - \xi^- \\ \xi^+ \geq 0 \\ \xi^- \leq 0 \end{gathered}$$

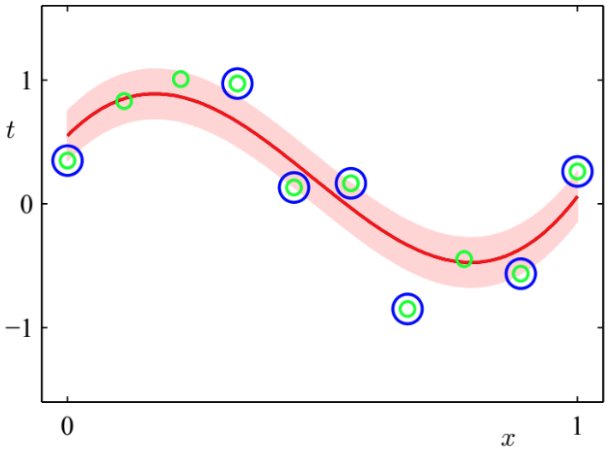
We got to the form of a standard quadratic programming problem.

❗ It is important to notice that the choice for the value of ϵ really important. In fact it will regulate how much noise is acceptable. A good choice is to choose it based on the measure of the precision for our dataset.

Finally, the resulting model after the computation of optimal weights is:

$$y(x) = \sum_{n=1}^{N} (\hat{\alpha}_n - \hat{\alpha}'_n) k(x, x_n) + \hat{w}_0$$

An important property is that the KKT condition is still valid, meaning that the zero terms (every point that falls inside the admitted intervall) will not contribute to the computation.



Artificial Neural Networks

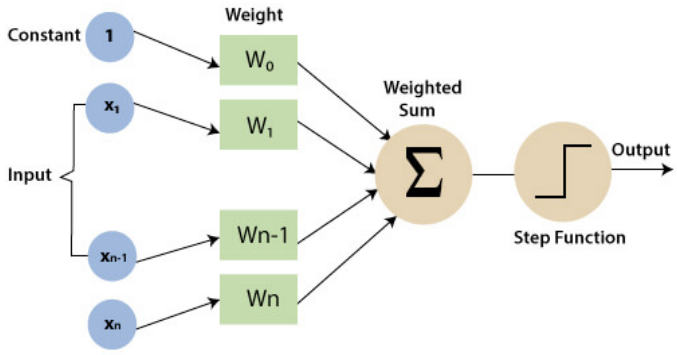
Introduction

As for other Machine Learnig techniques, the goal of Neural Networks is to find a function $\hat{y} : X \rightarrow Y$ that best approximates a target function. The core idea for solving this problem does not differ from the usual: we define a function $f(x, \theta)$ that takes as an input the input of our problems x and the weight of our model. In order to evaluate our model's predictions we will have to define an error function $E(\theta)$ and minimize it by tuning the parameters: $\theta^* = argmin_\theta E(\theta)$.

The main difference between neural networks and other classical machine learning techniques is that neural networks exploit non-linear functions not only with respect to the input space, but also with the parameters of the model itself. We say that the model is non-linear in θ.

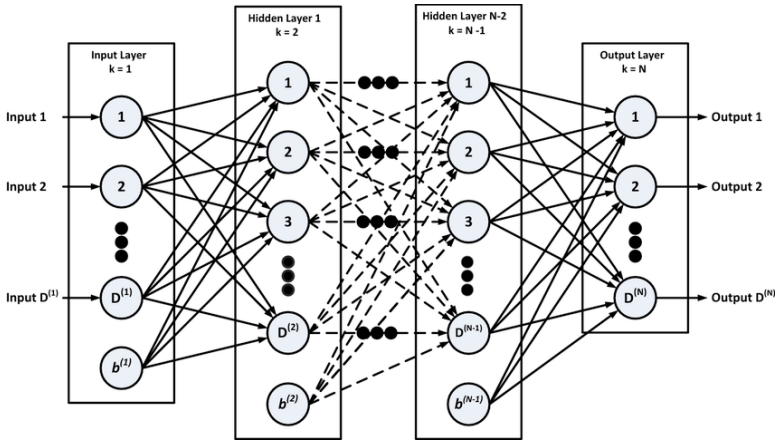
The perceptron

The intuitions for neural networks come from how biological neural networks work. The core unit of a neural network is a matematical abstraction called perceptron. A perceptron is fundamentally a pipeline that given some inputs $\{x_1, ..., x_n\}$ and weights $\{w_1, ..., w_n\}$ first computes a linear combination $\alpha = W^T X$ and then gives it as an input to the activation function that will compute the final output of the neuron.



Neural Networks

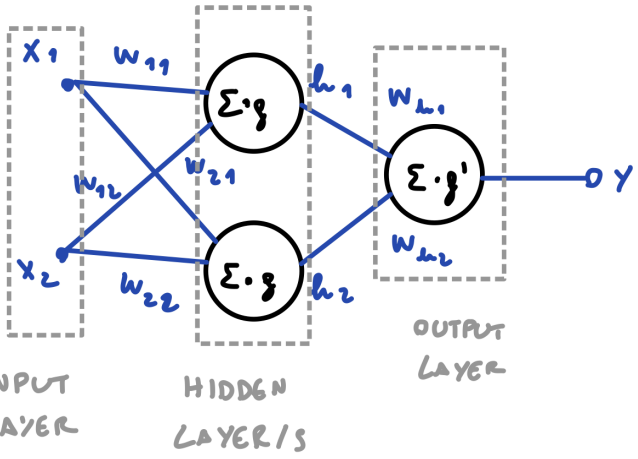
Starting from this fundamental unit, the idea is to realize a networks of interconnected perceptrons:



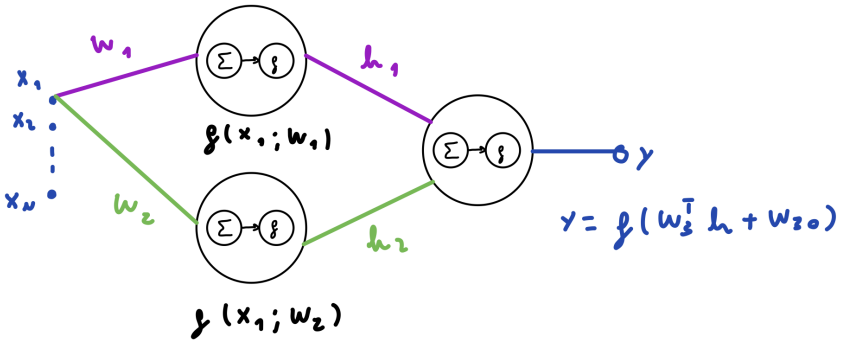
In particular, we will see Feed Forward Neural Neworks (FNNs). In this kind of networks the passage of informations from one layer to the other follows only one direction (from input to output) and does not have loops.

We have some standards denominations for layers of different nature:

- Input layer: it is that corresponds to the input features of our problem.
- Hidden layers: these are all the layers that lie between the input and the output layers.
- Output layer: the final layer of our model that will return the output.



Let's start from a simple layout of network:



Here every input is connected to every unit of the first layer. This means that each of the units will output a linear combination of the same inputs, but with different weights and biases:

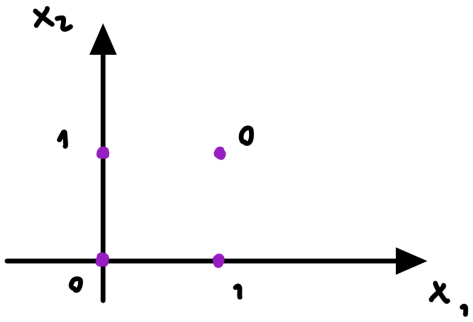
The final output will take as an input h_1, h_2 and subsequently apply a function to them. This means that in general the ouput of a neural network is actually a composition of functions:

$$h_3 = f^{(3)}(f^{(1)}(x, \theta^{(1)}), f^{(2)}(x, \theta^{(2)}), \theta^{(3)})$$

Or in an abbreviated form: $h_3 = f^{(3)}(f^{(2)}(f^{(1)}(x, \theta)))$

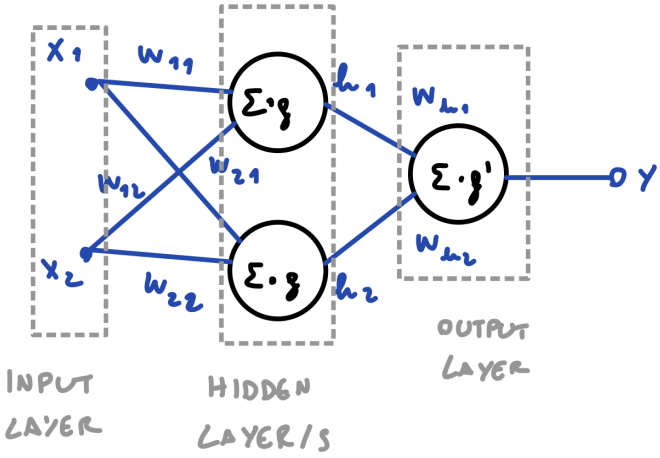
The XOR problem

We want to train a neural network on the representation of the XOR function:

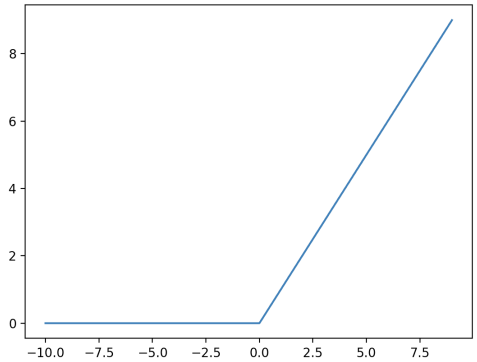


It’s clear that this problem is not solvable by a linear function. It could be solved by applying a kernel, but we will see that we don’t need it if we use a neural network.

We start from a two-layer NN:



The non-linear function denoted as g are called activation functions. As a standard, neurons of the same layer share the same activation function, while the output layer uses a different one. A common choice is using ReLu (Rectified Linear Unit) for the hidden layers and the identity function for the output layer. The ReLu function maps all the negative values to 0, while all of the non negative values stay the same:



The neural network will perform the following computations:

$$h_1 = ReLu(\begin{bmatrix} w_{11} & w_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{10})$$
$$h_2 = ReLu(\begin{bmatrix} w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + w_{20})$$
$$y = \begin{bmatrix} w_{h_1} & w_{h_2} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + b = W^T h + b$$

The expression of the full model is:

$$y(x; \theta) = W^T ReLu(W^T X + c) + b$$

With $\theta =< W, c, w, b >$.

Let's define now an error function. We choose Mean Squared Error (MSE) as a loss function:

$$J(\theta) = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(x_n, \theta))^2$$

This function measures the squared distances between our prediction and the target value.

The formulation that we have studied will be capable of dealing with the XOR problem.

Design choices

When designing a neural network we have to face the following choices:

- How many layers? The **Depth** of the network.
- How many units for each layer? The **Width** of our network.
- Which kind of units? The **Activation Functions**.
- Which kind of cost functions? The **Loss Function**.

The universal approximation theorem

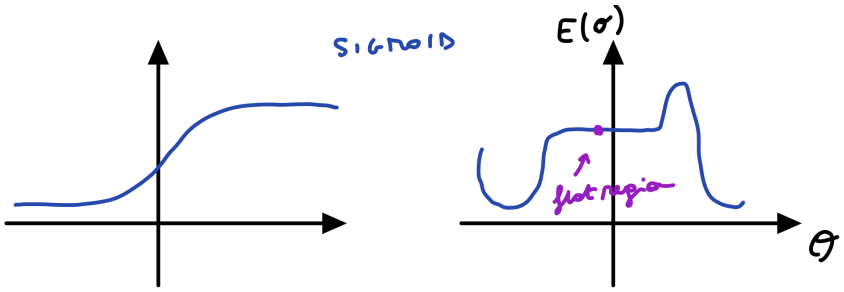
A feed forward neural network with one linear output layer and at least one hidden layer with any "squashing" function is capable of approximating any Borel measurable function with any desired amount of error, provided that enough hidden units are used.

But how many units?

We can't know the perfect amount of hidden units to use a priori. In theory a network with one very wide hidden layer could approximate any function, but in practice this results in a very bad approach. Instead, we normally use many hidden layers to create deep and narrow networks that make computations easier. Hence, the concept of **Deep Learning**.

Combining activation and loss functions

A problem with activation functions that "squish" the input space is that they present flat regions. Therefore, a bad combination of activation and loss functions could result in dealing flat regions with the loss function, which makes performing gradient descent very difficoult:



Thus, we will have to make the proper choices for these functions. We will deal with this in the next chapters.

Defining the cost function

We define the cost function based on the concept of likelihood. In particular, we are interested in the probability of obtaining the correct prediction given the input x and the weights of the neural network: $P(t|x,\theta)$.

The error is defined as the negative log-likelihood:

$$J(\theta) = E_{x,t\in D}\left[-ln(P(T|x,\theta))\right]$$

Similarly to the linear model case, we start by assuming that the likelihood's distribution is modeled by some Gaussian noise:

$$J(\theta) = E_{x,t\in D}\left(\frac{1}{2}||t - f(x;\theta)||^2\right)$$

Where $||t - f(x;\theta)||^2$ is the mean squared error.

Finding the maximum likelihood estimation corresponds to minimizing the mean squared error.

We choose the proper cost function by identifying the nature of our problem:

- Regression
- Binary classification
- Multi-class classification

Regression

When dealing with regression we choose a linear model for our last layer that implements an identity activation function: $y = W^T h + b$. We assume that the likelihood follows a Gaussian noise distribution $P(t|x) = \mathcal{N}(t|\gamma, \beta^{-1})$.

We choose mean squared error for the cost function.

Binary classification

Here we assume that our two classes are encoded as 0 and 1. We can use the Sigmoid for our activation function because of it property of mapping every possible output into values between 0 and 1: $y = \sigma(W^T h + b)$.

Since we are dealing with binary outputs, we can express the likelihood by using a probability distribution. Therefore our loss function $J(\theta) = E_{x,t \in D}\left[-ln(P(T|x,\theta))\right]$ will implement the following probability:

$$-ln(P(t|x)) = -ln[\sigma((\alpha)^t(1-\alpha)^{1-t})] = -ln[\sigma((2t-1)-\alpha)] = \\ = softplus((1-2t)\alpha)$$

Remember that $\alpha = W^T X + b$. Softplus is the name for our error function.

Multi-class classification

In this case we choose the softmax activation function defined as:

$$softmax(\alpha_i) = \frac{e^{\alpha_i}}{\sum_j e^{\alpha_j}}$$

The softmax function normalizes the output for each class. The best prediction will correspond to the class with that will return the closest value to 1.

Here the likelihood is expressed as a multinomial distribution, with J_i corresponding to the error for the i-th class:

$$J_i(\theta) = E_{x,t \in D}[-ln(softmax(\alpha_i)]$$

Activation functions for the hidden units

The premise for the choice of the activation function is that there is no theoretical better one. Empirically, it has been shown that ReLus are the best choice expecially if we don't have any particular information about the dataset. Being non-linear function makes them robust to the phenomenon of flat regions. Their only drawback is that ReLus are not differential in 0, but in practice this won't be a problem.

The formal definition of a ReLu is:

$$g(\alpha) = max(0, \alpha)$$

Computing the gradient

When implementing a neural network it's fundamental to choose an efficient way for computing the gradient. The default algorithm is **Backpropagation**.

The general idea for backpropagation is having an iterative approach, where each iteration has two steps: the first step (*forward step*) is about computing the output value for each layer until we get to the final output y. The second step (*backward step*) computes the loss between the network's output and the corresponding target value and adjusts the values for each layer based on the gradient.

How do we compute the gradient?

In order to get to the actual computation of the gradient we must start by noticing that our final output is the result of a composition of function. Thus, the gradient for the loss function $\nabla_\theta J(\theta)$ will be computed by exploiting the chain rule.

The general formula for computing the derivative of a composition of function is the product of each partial derivative for each function. For example let $y = g(x)$ and $z = f(g(x)) = f(y)$:

$$\frac{dz}{dx} = \frac{df}{dy}\frac{dy}{dx}$$

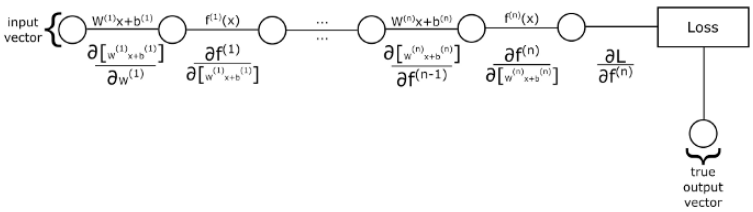
In general we will have to deal with vector functions $g : \mathbb{R}^m \to \mathbb{R}^n$ and functions $f : \mathbb{R}^n \to \mathbb{R}$.

The corresponding gradient will be:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Or, in vector notation: $\nabla_{x^z} = (\frac{\partial y}{\partial x})^T \nabla_{y^z}$.

This concept adapts well to the case of neural networks because the function at layer n is the result of a composition of functions for each layer from 0 to n-1. This means that in order to express its derivative we will have to apply the chain rule and multiply the derivatives for each of the n layers.



Forward step

Forward step

Require: Network depth l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value

$h^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
  $\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
end for
$\mathbf{y} = \mathbf{h}^{(l)}$
$J = L(\mathbf{t}, \mathbf{y})$

For each layer of the function we compute the linear combination $\alpha^{(k)}$ taking as input the output of the previous layer $h^{(k-1)}$. Then we apply the activation function $h^{(k)}$ to $\alpha^{(k)}$ whose result will be the input for the next layer.

The final output y will be the output of the last layer $h^{(l)}$. The final step will be computing the loss between the target and the output.

Backward step

Backward step

$\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$

for $k = l, l-1, \ldots, 1$ **do**

Propagate gradients to the pre-nonlinearity activations:

$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}

$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$

$\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$

Propagate gradients to the next lower-level hidden layer:

$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$

end for

Notice that in this algorithm the variable g represents the gradient and works as an accumulator. Each time that the arrow appears we are updating g.

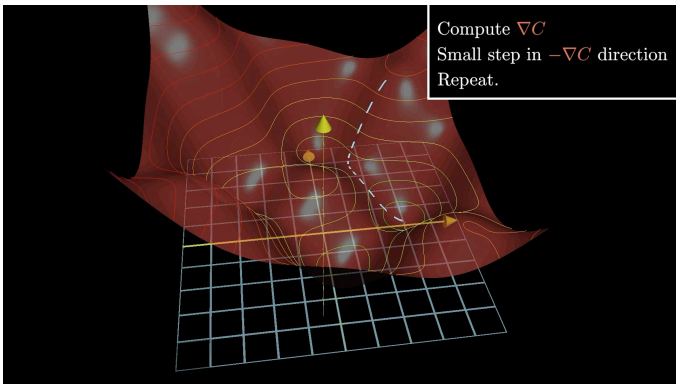
The first step consists in computing the gradient for the loss function $\nabla_y J = \nabla_y L(t, y)$ and assigning it to g.

Then for each previous layer (going from l to 1) we compute the gradient of the loss with respect to the k-th layer $\nabla_{\alpha^{(k)}} J$ as the product between the current value of g (which represents the gradient of the layer k+1) and the derivative of the non-linear function applied to the linear combination of the current layer $f'(\alpha^{(k)})$.

Rember that ultimately our goal is to compute the gradient of the loss function with respect to the weights and biases of each layer, in order to tune them properly and lower the error at the next iteration. In order to do so, we compute the gradient of the bias for layer k with respect to the loss function $\nabla_{b^{(k)}} J = g$ and the gradient of the weights for layer k with respect to the loss function -$\nabla_{W^{(k)}} J = g(h^{k-1})^T$.

The last step is computing the gradient of the loss function with respect to the output of the k-1 layer $\nabla_{h^{k-1}} J = (W^{(k)})^T g$.

This is an example of what gradient descent does in practice:



Notice that the backpropagation algorithm will be repeated many times during the training of our network. This results in a serious problem from a computational cost standpoint, meaning that even if this technique works in theory, in practice we might never reach a result in reasonable time. This is why several tricks have been developed in order to make the computation lighter:

- Use dynamic programming in order to avoid the same computation multiple times.
- Extract the symbolic form of the gradient.

Stochastic Gradient Descent (SGD)

In practice, we have seen that each step of backpropagation can be really demanding as the input size increases. A great workaround technique is stochastic gradient descent. The core concept here is to compute backpropagation with respect to a mini-batch for each iteration. A mini-batch is a small subset of the training set, made of $\{x_1, ..., x_m\}$ random samples (where m is the size of the mini batch).

At each iteration, we will compute the loss function considering only the current mini-batch, then we compute the average for the loss of each mini-batch.

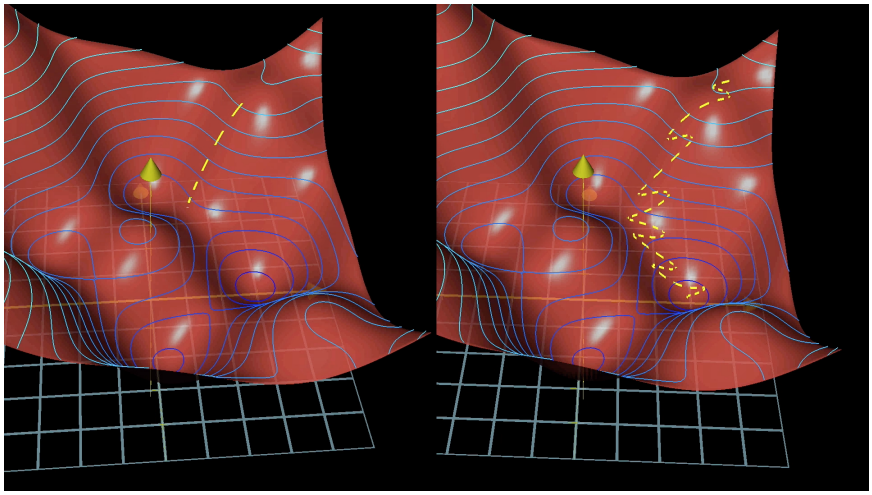
Require: Learning rate $\eta \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of m examples from the dataset D
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \mathbf{g}$
    $k \leftarrow k + 1$
  end while

Observe: $\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{t})$ obtained with backprop

Notice that in the algorithm appears the term η, the learning rate. We already encountered it and, as stated before, this coefficient regulates by how much we move in the direction of the gradient for each step. When deciding the value of η we can encounter two main problems. If its value is too big, we risk to skip the optimum and start oscillating back and forth until divergence. In contrast, if the learning rate is too low, we risk to move too slowly and never reach the local optimum.

The following is a comparison of what happens in classical vs stochastic gradient descent. Notice how we will be able to reach an optimum with both approaches, but

SGD require more steps in order to do it. The important thing is that each of the steps of the SGD takes substantially less time to be performed.



A possible solution to properly tune the learning rate is an adaptive approach. We start from a big value of η and we keep decreasing it at each iteration, until a certain step. This is a clever approach, since it's more likely that we start far from the optimum, and then get closer to the solution as the number of iteration increases. A formal representation of adaptive learning rate is:

Until iteration $\tau(k \leq \tau)$:

$$\eta^{(k)} = (1 - \frac{k}{\tau})\eta^{(k)} + (\frac{k}{\tau})\eta(\tau)$$

After iteration τ:

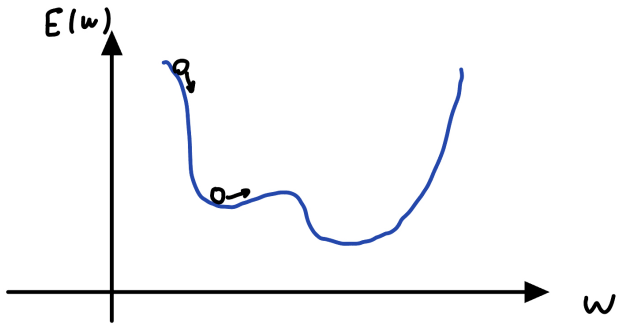
$$\eta^{(k)} = \eta^{(\tau)}$$

The value for τ will be another parameter that we'll have to properly tune.

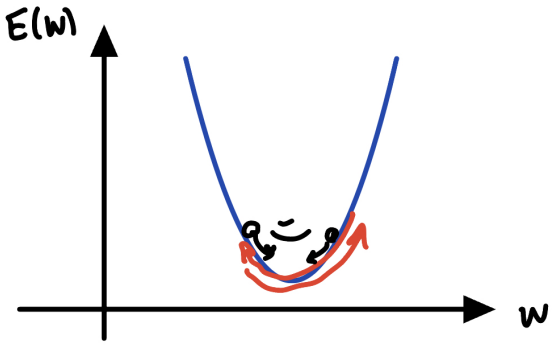
SGD with momentum

The intuition for this approach comes from the physics concept of momentum and potential energy.

When we let a ball roll from a cliff, it will still have a tendency to move forward even when it reached a plain surface due to the accumulation of momentum. The same concept can be applied to the computation of the gradient.



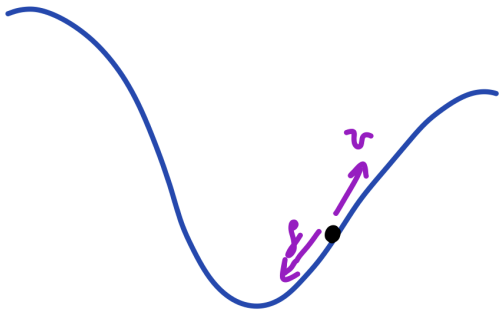
In cases like the image above, it is immediate to understand that applying momentum could make us discover better local optimums. In cases like the image shown below, the ball will keep oscillating until it eventually stabilizes due to friction:



When using SGD with momentum we have to care about two parameters:

- The learning rate $\eta \geq 0$
- The momentum $\mu \geq 0$

The momentum will regulate how much we will keep moving even if we are still. We start form the momentum in order to compute the velocity as $v^{(k+1)} \leftarrow \mu v^{(k)} - \eta g$, meaning that we will keep moving even when gradient g is non-negative. Notice that at a certain point when $v = g$ we will hit a breakpoint where the component of g will be higher than momentum and we will change direction.



Nesterov momentum

This is a technique that uses the same concept of momentum, but chooses to apply momentum before the computation of the gradient (sometimes it works better).

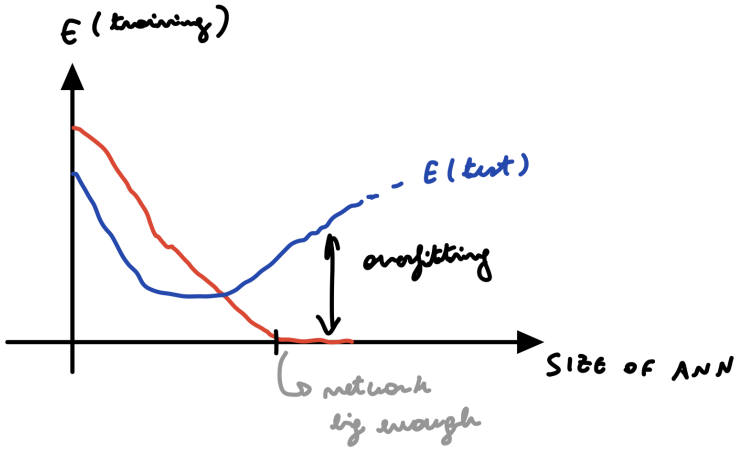
Tuning the parameters

We have seen how to compute the gradient of the loss with respect to each parameter of the network. Optimizers are the specific function that take as an input the loss function and use it to tune the parameters.

Popular optimizers are: AdaGrad, RMSProp, Adam…

Regularization

We have seen how in theory increasing the size of a neural network or the number of training iteration makes our model more performant. In reality we have to take into account that during training we are fitting our network just on a subset of samples. This will cause to hit a certain point where while the error on the training keeps decreasing, the error on the test set will start increasing. When this happens, we are facing overfitting.



We have a set of possible methods to avoid overfitting named regularization methods:

Parameter norm penalties

A common cause for overfitting is having parameters that are too high. In order to face it, we add a regularization term to the cost function that will penalize parameters with high values:

$$E_{reg}(\theta) = \sum_j |\theta_j|^q$$

The resulting cost function will be:

$$\bar{J}(\theta) = J(\theta) + \lambda E_{reg}(\theta)$$

Where λ is named regularization term, and will tune the effect of regularization.

Dataset augmentation

We know that the cause for overfitting is that our model will fit too precisely the test set, resulting in bad performances when generalizing the prediction.

A possible solution is to add noise to the training set, by producing samples that present small variation such as:

- Adding noise
- Image rotation, scaling, illumination etc.

Early stopping

We analyze the performance of the network during training over time by comparing train and test loss. We will stop when the train loss keeps dropping, but test loss starts increasing or remains stable.

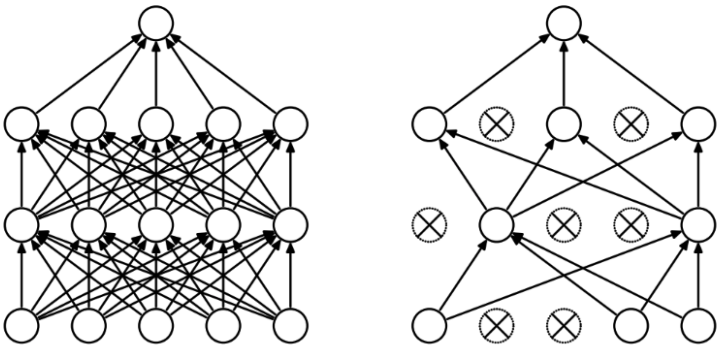
Parameters sharing

We now that a big number of parameters can increase the performance of a network, but this increases also the probability of overfitting. A possible solution here is to keep the numbers of parameters of the architecture, but introduce a constrainy such that some parameters must share the same value. This means that for each iteration we will have a certain number of different parameters (we call them trainable parameters), while all the other will be shared.

Dropout

As well as for parameters sharing, dropout is another method to deal with big numbers of parameters.

The concept here is to keep the same structure, but at each step we randomly choose a certain percentage connection that will be trained, will the others will stay the same.



(a) Standard Neural Net (b) After applying dropout.

Convolutional Neural Networks

Introduction

Convolutional Neural Networks (CNNs) are networks that exploit the operation of convolution in order to process high dimensional data (like images) transforming the dimensionality of the original space.

Convolutions

Convolutions are mathematical operation between two functions, where one function is multiplied by the other one shifted. In the continuous case the operation is expressed as an integral:

$$(x * w)(t) = \int_{a=-\infty}^{\infty} x(a)w(t-a)da$$

The corresponding formula for convolution in the discrete case is based on a summation:

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In the case of CNNs, we will use limited convolutions (we have a limited amount of terms). The two functions involved are conventionally called Input function I and Kernel function K:

$$(I * K)(i,j) = \sum_{m \in S_1} \sum_{n \in S_2} I(m,n) K(i-m, j-n)$$

What we have seen so far is the 2D case, but the operation can be easily extended to n-dimensions by taking into account n indexes. For example, the equivalent in 3D is:

$$(I * K)(i,j,k) = \sum_{m \in S_1} \sum_{n \in S_2} \sum_{u \in S_3} I(m,n,u) K(i-m, j-n, k-u)$$

Convolutions hold the following properties:

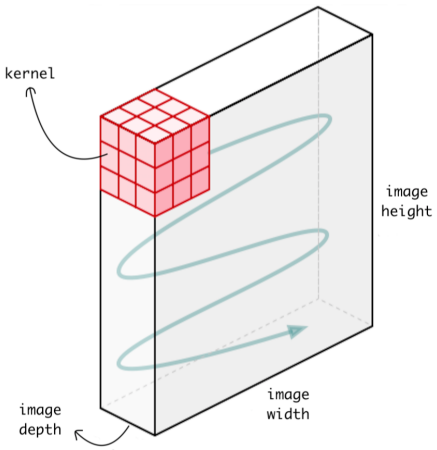
- Commutative: $(I * K)(i,j) = (K * I)(i,j)$
- Cross-correlation: flipping the kernels in the convolution operation produce the same result.

The kernel function

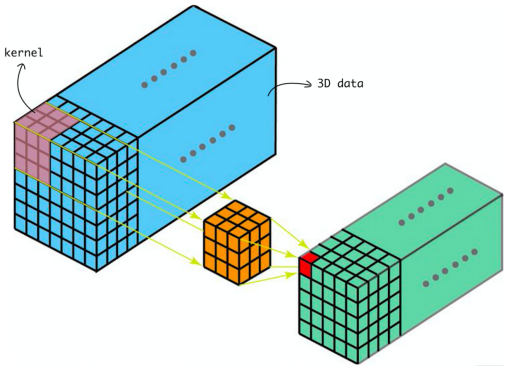
Our goal with convolution will be finding the proper kernels for our problem. We can think about kernels as filters that can be applied to inputs through the operation of convolution.

The dimension of the convolution defines the direction where the kernel will move:

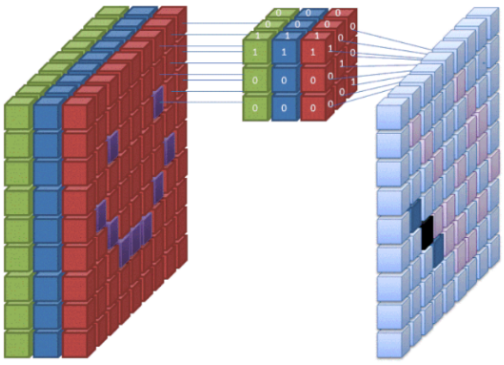
Hence, the dimension of convolution is independent from the size of the input. For example a 1D convolution can be applied to a 2D or 3D input etc.



In practice, when we apply convolution each time we take a slice of the input corresponding to the size of the kernel, we multiply element-wise the two slices and sum each element. Then the kernel is shifted and the operation is repeated at each shift:



We will focus particularly on 2D convolutions. As we already mentioned, this means that our kernel function will be able to move in 2 directions, which implies that the kernel's depth must always be the same size of the input's depth:



Some terminology

A typical architecture for a CNN consists in three layers: an **input layer**, a **convolutional layer** and finally an **output layer**.

We can think about the convolutional layer as a transformation of the input space in some other (tipically reduced) space.

The **input size** is denoted as $w_{in} \times h_{in} \times d_{in}$. The depth of the input usually identifies its number of channels. For example an RGB image will have $d_{in} = 3$.

Similarly, the **kernel size** will be defined bu $w_k \times h_k \times d_k$ and the **output size** $w_{out} \times h_{out} \times d_{out}$

Since we can use multiple kernels, we define d ad the number of kernels of a convolutional layer.

We define the **feature map** (or depth slice) a particular section of the output. In general, we can say that the output of a convolution is a composition of feature maps.

For example the convolution between a $32 \times 32 \times 3$ input and a single $5 \times 5 \times 3$ kernel outputs a $28 \times 28 \times 1$ feature map. If we have $d = 6$ kernels, we will obtain a $28 \times 28 \times 6$ feature map.

The **trainable parameters** of a convolutional layer correspond to the total parameters of its kernels. The i-th convolutional layer will have $w_k^{(i)} \times h_k^{(i)} \times d_k^{(i)} \times d^{(i)}$ trainable parameters.

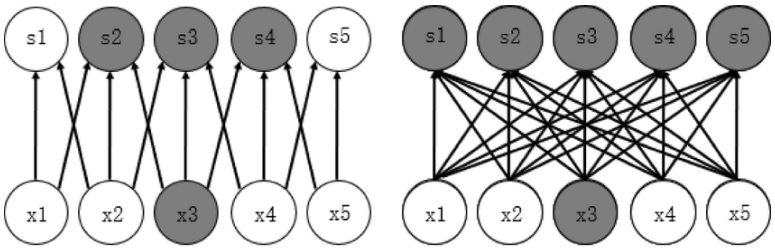
Inside a convolutional layer

Internally, a convolutional layer is composed by three stages. Initially, the **convolution stage** will take the input and convolutes it with a kernel. Then the convololution is passed to a **detector stage**, that applies a non-linear function. Finally the **pooling stage** will transform the output of the detector by applying another filter.

Sparse connectivity

When defining the architecture of a CNN, we have to remember the problem we want to solve, which is usually to work with images or videos. In these cases we care about locality where considering the features of the input. In fact it’s much more likely that a pixel of an image is correlated to it’s neighbour with respect to another pixel far away.

This is why introduce the concept of **sparse connectivity**. Instead of working with fully connected layers, we exploit the concept of **locality**, we connect a pixel just to the adjacent one, thus saving in terms of numbers of parameters needed.



(a) sparse connectivity **(b)** dense connectivity

Kernels work with local operations by computing each time a subset of the input.

Parameter sharing

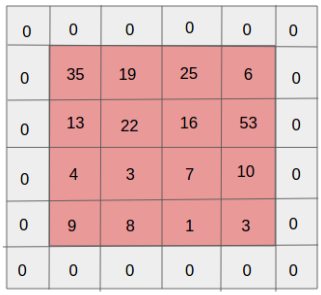
Parameter sharing is a constraint on the weights imposing that some of the weights of the same layer must have the same value. This method allows for a substantial reduction in the number of total trainable parameters.

When we use a kernel, we are basically repeating operations on different slices of the output by using each time the same kernel parameters. This implies a substantial re In fact, when we look at how kernels work, they basically implement both concepts of sparse connectivity and parameter sharing.

Padding

We have seen that in general the operation of convolution between an input and a kernel results in an output with lower dimensions that the input. In particular, if we have an input of sice w_{in} and a kernel of size w_k, the output will have a width of $w_{out} = w_{in} - w_k + 1$. Notice that this happens if the convolution operation starts with the kernel aligned to the border of the input.

Sometimes, we may want our output to have the same size of the input. In this cases we introduce the concept of padding. Basically we compensate the loss of size by extending the origin input using some filling values (usually 0).



The number of this fillings depend on the kernel's size and follows this formula:

$$p = \lfloor \frac{w_k}{2} \rfloor$$

Padding is called:

- Valid if $p = 0$
- Same if $p = \frac{w_k}{2}$

The Pooling stage

We already mentioned that convolutions impliy a change in the size between input and output. Tipically, while the witdth and height become smaller, the depth of the output increases depending on the number of kernels used.

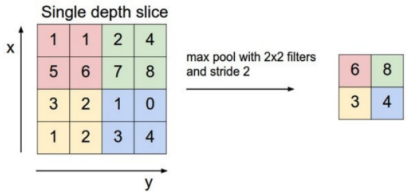
The pooling stage is responsible for further decreasing the size of the final output by applying a filtering operation. This time, the filter consists in a well defined function that has no trainable parameters.

Two common approaches for the pooling stage are:

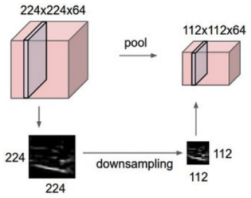
- Max-pooling: we extract the maximum value from the considered region.
- Average-pooling: returns the average value from the considered region.

An important parameter for padding is **stride**. Stride defines by how much the pooling filter moves at each step. In case of more than one dimension, we can define different stride parameters for each dimension. Pooling is considered as a subsampling operation.

Example of **max pooling**



Introduces subsampling:



Computing the size of the output

Considering an input of size $w_{in} \times h_{in} \times d_{in}$ and a kernel $w_k \times h_k \times d_k$ with stride s and padding p, the dimensions of the resulting feature map will be:

$$w_{out} = \frac{w_{in} - w_k + 2p}{s} + 1$$

$$h_{out} = \frac{h_{in} - h_k + 2p}{s} + 1$$

The total number of trainable parameter of a convolutional layer is:

$$|\theta| = w_k \cdot h_k \cdot d_{in} \cdot d_{out} + d_{out}$$

Transfer learning

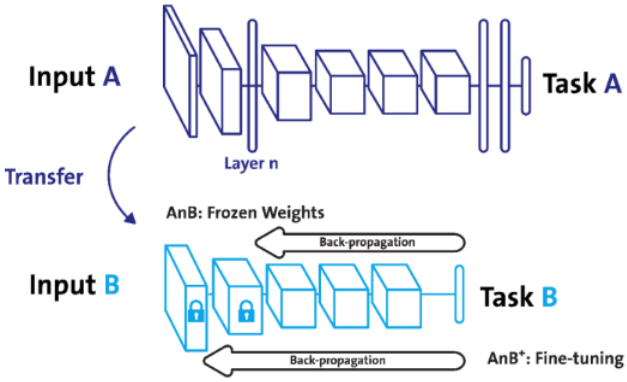
The main idea with transfer learning is to use pre-trained model on new examples belonging to a similar domain. A major advantage of this method is to reduce the computation needed for training from scratch a new model.

Assuming that we have a source domain D_s from where we obtained a function $f_s : X_s \rightarrow Y_s$, with $D(x_s)$ being the distribution that has been used to extract the images from the dataset and $P(y_s|x_s)$ the probability distribution of the output with respect to the input.

Let's now consider a target domain D_t with $f_t : X_t \rightarrow Y_t$, $D(x_t)$ and $P(y_t|x_t)$. Can we exploit the work that has been done in training a model for the source domain and apply it to solve the problem on target domain?

There are many possible techniques.

If both source and target domain are similar (they come from a similar distribution), we can think about keeping the feature extraction part the same and re-training only the last layers. This approach is called **fine-tuning**. Notice that this method still requires a lot of computation, since the final layers of a CNN are dense.



Another intelligent approach is to exploit our CNNs just as feature extractors. In fact, we have seen that CNNs are really good at extracting the fundamental features of the input via the operation of convolution. We could use this property in order to create a pipeline where the CNN extracts the features, that are then fed to another ML model in order to compute a prediction.

Multiple learners

Introduction

The classical approach when working on a machine learning task is to train on a certain dataset and produce a certain output function. Tipically during training the performance in terms of accuracy get better over time, but we will reach a point where accuracy starts to increase at a really lower rate or even start oscillating. Since in practice we might have a limited amount of training time, betting on a single model doesn't feel like a good choice.

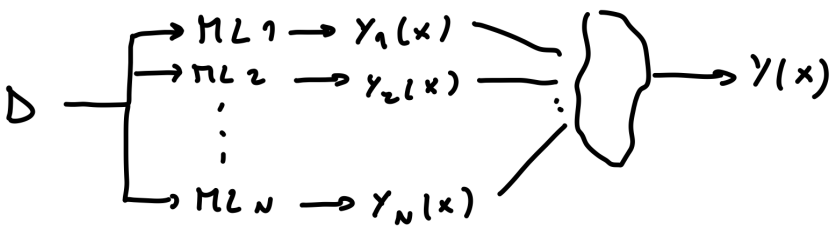
The idea behind multiple learners is to split the computation of training among several modules. This approach almost always leads to better results for the same amount of time with respect to training a single model.

There are three main approaches to multipple learning: voting, bagging and boosting.

Voting

Given a certain dataset D and a certain number of training models $ML_1, ..., ML_N$, we train the every model in parallel (independently) by feeding the same dataset as an input to each of them.

Each of the models will output a certain function $y_i(x)$. The final result will be an ensamble of the results of the different functions.

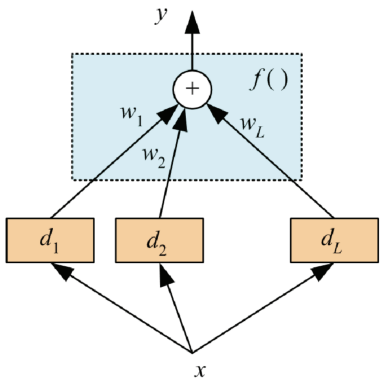


In more detail, when performing voting on **regression**, the result will be the sum of the **weighted average** of the results for each model:

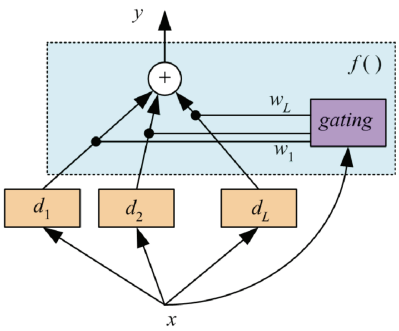
$$y_{voting}(x) = \sum_{m=1}^{M} w_m y_m(x)$$

When dealing with **classification**, the result will be based on the **weighted majority**. This means that we will assign different weights to each prediction, and then computer the argmax of the result:

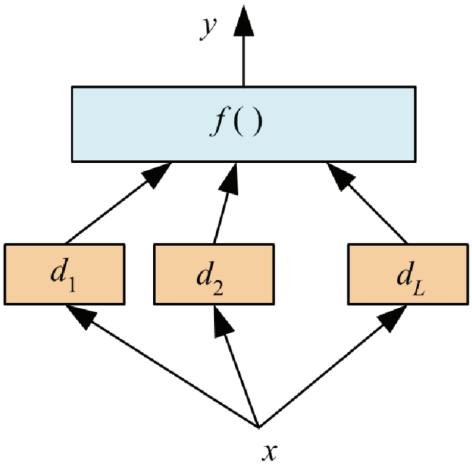
$$y_{voting}(x) = argmax_c \sum_{m=1}^{M} w_m I(y(x_m) = C)$$



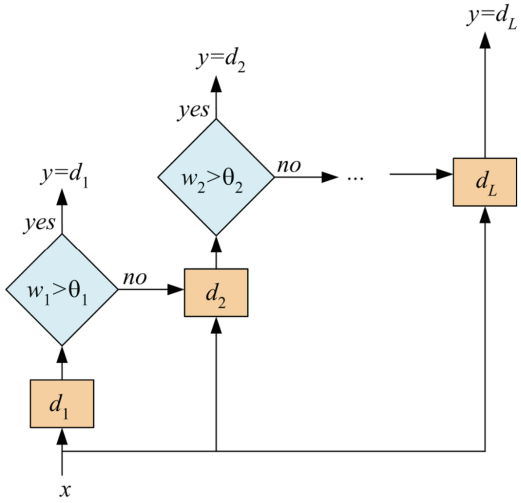
When implementing voting, a finer approach is to introduce **gating**. Basically, we adjust the weights of the voting function based on the kind of input by using a non-linear gating function:



The **scaling** approach is to treat the function that combinates the different outputs as a trainable set of parameters. The function will learn how to assign the proper weights to our outputs:



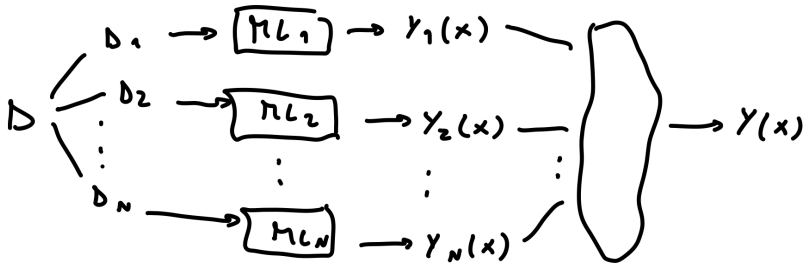
Finally, another approach is **cascading**. We set a certain threshold for confidence and subsequently evaluate the instance on a certain model, until we get an acceptable confidence value:



A general problem of the voting approach is that we are using the same dataset to train every model. This does not produce a good variability on the models obtained, meaning that me might risk of having bad performances for every mode.

Bagging

When implementing bagging, the idea is to pre-process the dataset in order to divide it in different subsets $D_1, ..., D_N$ and feed a different subset to each model:



This method introduces a better approach with respect to voting in terms of variance, but still doesn't implement any interaction between models during training.

Boosting

The main concept of boosting is to train each model on a subset of the entire training dataset, but make the training sequential such that each step will be influenced by the previous one.

Starting from a dataset $D = \{(x_n, y_n)_{n=1}^N\}$, the idea is to assign a weight $w_n^{(m)}$ to each of its samples (where m is the m-th step). At the beginning every weight will be initialized with the same value $w_n^{(1)} = \frac{1}{N}$.

The weights will regulate the importance of each sample during training. In particular the strategy is to evaluate the results and increse the weights when the error between prediction of the sample and the actual value is high, and lower their value when it's low. This will ensure that in the next step training will have produce better results.

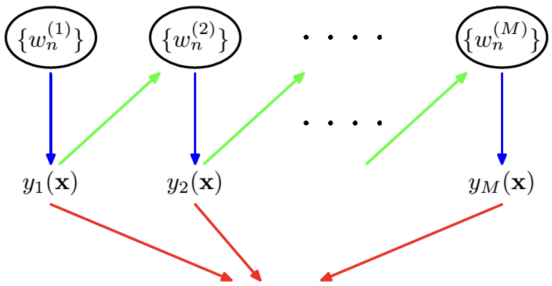
We implement boosting by adding the weights to the computation of the error. The formula for the error will be:

$$E(\theta) = w_n(y(x_n) - t_n)$$

This means that samples with higher weights will produce bigger errors.

A popular implementation of boosting is AdaBoost. Basically we compute the weighted error function, evaluate the results and update the weights accordingly to our strategy, before passing to the next step.

The final classifier will be an ensemble model weighted on the performances of each trained model:



$$Y_M(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right)$$

Unsupervised learning

Introduction

The machine learning techniques that we studied in the previous lessons were about supervised learning, meaning that we assumed that our training datasets were labelled.

When dealing with real problems, many times we have to face situations were we only have unlabelled or partially labelled samples, this problems belong to the discipline of unsupervised learning. Hence, what we want is to find a funcition approximation $f : X \to Y$ by having a dataset that contains unlabelled data $D = \{(x_n)_{n=1}^N\}$.

In the next chapters we will see different techniques for unsupervised learning.

Gaussian Mixture Models

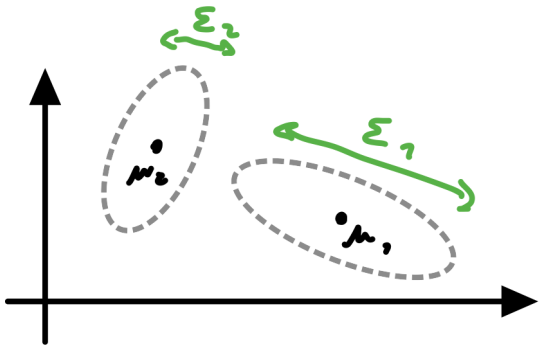
The core idea behind Gaussian Mixture Models (GMM) is that we assume that our data was generated from a certain probability distribution $P(x)$ expressed as a weighted sum of k Gaussians:

$$P(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

Where:

- π_k is the weight of the k-th Gaussian (also called the prior). The higher the weight, the higher is the probability that data was generated from that specific Gaussian.
- μ_k is the mean of the k-th Gaussian.
- Σ_k is the covariance matrix of the k-th Gaussian.

Given all of these parameters, it’s easy to generate a new dataset by sampling data points from the distribution.

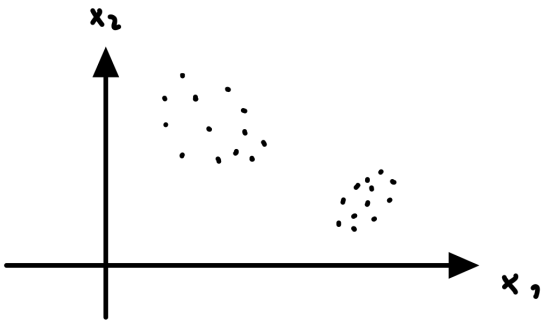


Once we have generated our data, we can forget about their parameters of the original distribution and try to learn them by starting from scratch, given that the number of distributions k is known.

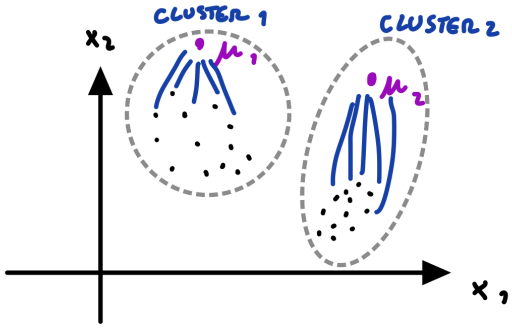
K-Means

We start by assuming that we know that the number of Gaussians k is known and that both covariance and priors are the same for each distribution.

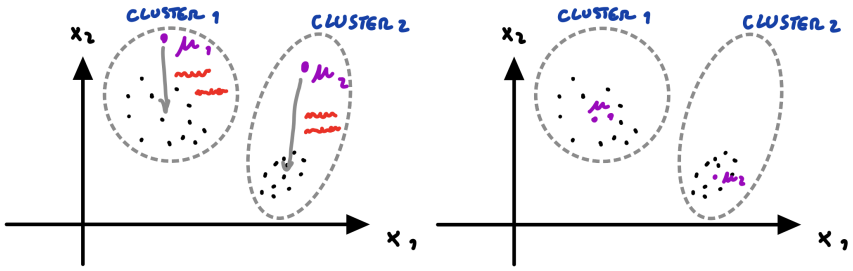
Our goal is to find the $\mu_1, ..., \mu_k$ means given the dataset and k. We will start from a set of unlabelled data like:



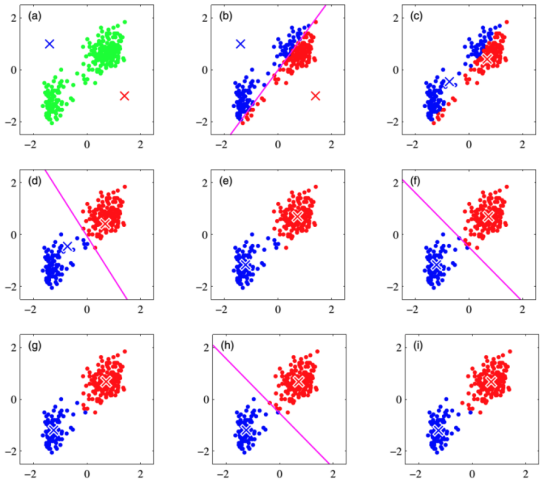
K-means is an iterative algorithm. The first step is to initialize the k means in two possible ways: either we choose a totally random value, or we choose a random point of the dataset. Given these means, we create k clusters based on the minimum distance between every point and the various means. The point will belong to a certain cluster if the distance between it and the cluster’s mean is the closest:



We can now repeat the computation of the mean by computing the center of mass of each cluster, which corresponds to finding the average of the values for each cluster:



After computing the new means, we re-clusterize based on the same criterion of the minimum distance. The algorithm will iteratively repeat these operation until the we won't have any changes in assignment between one step and the previous one.



Example of the execution of k-means.

The output of this algorithm will correspond to the computed means.

Problems of k-means

- We depend on the right guess of k. Having a different number of means with respect to the actual value will result at best only in a partial representation of the dataset.
- Outliers influence negatively the computation by attracting the mean towards them.
- An important design choice is the defintion of a proper distance function, since k-means does not consider the covariances of distribution.

Latent variables

Latent variables are a special kind of variables that are useful to describe our problem, but are not given by the dataset. They help us in better defining the formulation of our problem.

When we analyze samples we don't know from which of the k distributions they come from. Hence, we can think about introducing the boolean variables $z_k \in \{0, 1\}$ that will represent this notion. If $z_k = 1$ means that the sample comes from the k-th distribution, 0 otherwise. The informations about all of these latent variables can be econded using a 1-out-of-k approach: $Z = (z_1, ..., z_k)^T$.

We define the probability of belonging to a certain distribution as the posterior $P(z_k = 1) = \pi_k$

Thus, $P(Z) = \prod_{k=1}^{K} \pi_z^{z_k}$ (the likelihood of that vector is given by the products of the posteriors elevated to the value of the latent variable. Notice that only one latent varable has value 1).

Given Z, we compute the conditional probability of x given Z as:

$$P(x|Z) = \mathcal{N}(x; \mu_k, \Sigma_k)$$

Notice that if we know Z, then the probability only depends on the k-th Gaussian. Given the previous considerations, we can express the posterior as:

$$P(x|Z) = \prod_{k=1}^{N} \mathcal{N}(x; \mu_k, \Sigma_k)^{z_k}$$

The total probability of x will be:

$$P(x) = \sum_{Z} P(Z)P(x|Z) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

We obtained the formulation of a Gaussian Mixture Model!

A fundamental assumption that we can derive is that a Gaussian Mixture Model can be expressed as the marginalization of the joint probabilities $\sum_Z P(Z)P(x|Z)$.

Our problem now is that we don't actually know the value of the latent variables.

We can now define the posterior probability, meaning the probability that a certain data comes from a certain distribution as:

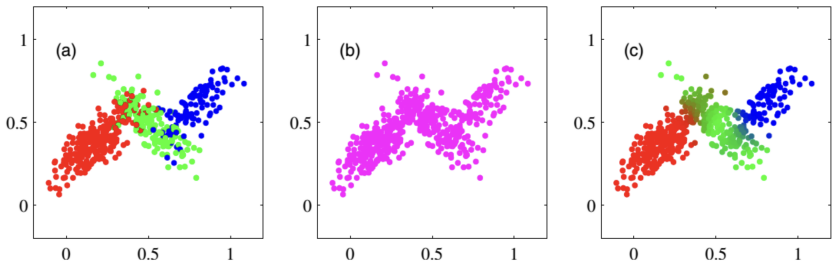
$$\gamma(z_k) = P(z_k = 1|x) = \frac{P(z = 1)P(x|z = 1)}{P(x)}$$

From where we obtain:

$$\gamma(z_k) = \frac{\pi_k \mathcal{N}(x; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_k \mathcal{N}(x; \mu_j, \Sigma_j)}$$

Where the denominator $\sum_{j=1}^{K} \pi_k \mathcal{N}(x; \mu_j, \Sigma_j)$ is called normalization layer.

The following is the representation of what happens when using Gaussian Mixture Models on an unlabelled dataset:



- Image (a) represents the ground truth, that we don't know when dealing with the problem.
- Image (b) shows the unlabelled dataset, that we have as input.
- Image (c) shows the resul of applying Gaussian Mixture Model on the unlabelled dataset. Notice that the colors encode the posterior probability.

Computing the posterior: Expectation Minimization

While before we started with the assumption that both covariance and prior are the same for every distribution, now we are interested in determing all the three parameters μ_k, Σ_k, π_k.

In particular, we aim at computing the maximum likelihood:

$$argmax_{\pi,\mu,\Sigma}\ ln(P(x|\pi,\mu,\Sigma))$$

The computed solution for this problem is:

$$\mu_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(x_{nk})x_n$$
$$\Sigma_k = \frac{1}{N_k}\gamma(z_{nk})(x_n-\mu_k)(x_n-\mu_k)^T$$
$$\pi_k = \frac{N_k}{N}$$

Where $N_k = \sum_{n=1}^{N}\gamma(z_{nk})$.

Notice that the solution depends on the dataset $x_n$on the posterior $\gamma(z_{nk})$. We still have to determine z.

At this point, if we assume to know all the other parameters, we can compute the postirior of z.

Expectation maximization exploits the formulas that we derived, by implementing them into an iterative two-steps algorithm that is repeated until convergence. These two steps are:

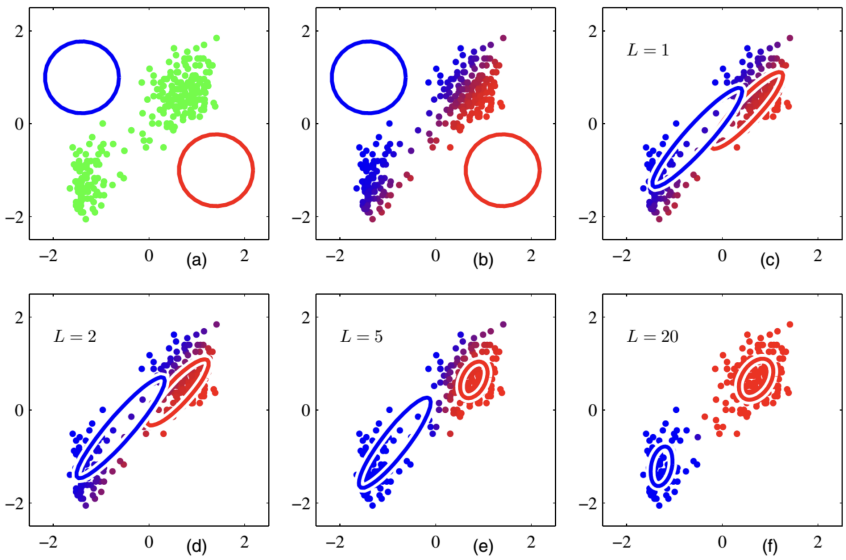
- **Expectation:** given the parameters of the distribution, we computer the posterior of z.
- **Maximization:** given the prediction of the posterior, we compute the parameters of the mode.

The algorithm begins by initializing the parameters $\mu_k^{(0)}, \Sigma_k^{(0)}, \pi_k^{(0)}$ at step 0 with random values.

We start from the expectation step E, where we compute the posterior given the parameters.

The second step is maximization M, where we do the opposite by taking the previously computer posterior and compute the new parameters.

E and M steps are repeated until convergence. This method will output all the parameters.



In order to simplify the notation, we can denote the parameters as $\theta = \mu, \Sigma, \pi$ and the updated parameters as θ'. Now, we can represent the function that updates the parameters as $Q(\theta'|\theta)$.

The EM algorithm can be generically defined as:

E step:

Compute $Q(\theta'|\theta)$ using the current hypothesis on the parameters θ and the observed data X in order to estimate the pobability distribution over $Y = X \cup Z$:

$$Q(\theta'|\theta) \leftarrow E[lnP(Y|\theta')|\theta, X]$$

M step:

Update the current hypothesis to θ' by maximizing the Q function:

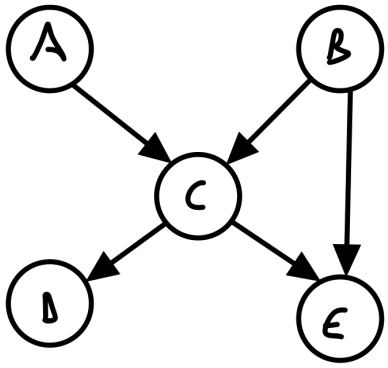
$$\theta \leftarrow argmax_{\theta'} Q(\theta'|\theta)$$

Bayesian Networks

Bayesian networks are basically a graphical model to express the conditional probabilities between two or more variables. For example, we can express $P(X|Z)$ as follows:

Notice that there are two entities in bayesian networks: nodes represent the random variables, while directed edges represent the dependence of a child from its parent\s.

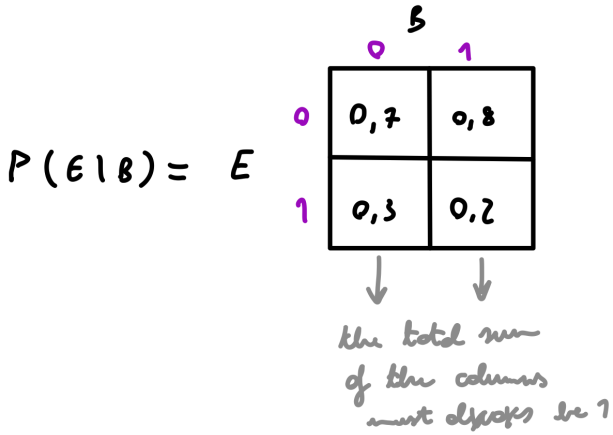
Bayesian networks can grow by representing the relationtip of many variables, like the following example:



Bayesian networks are a particular case of a broader class of graphs called Probabilistic Graphical Models (PGM).

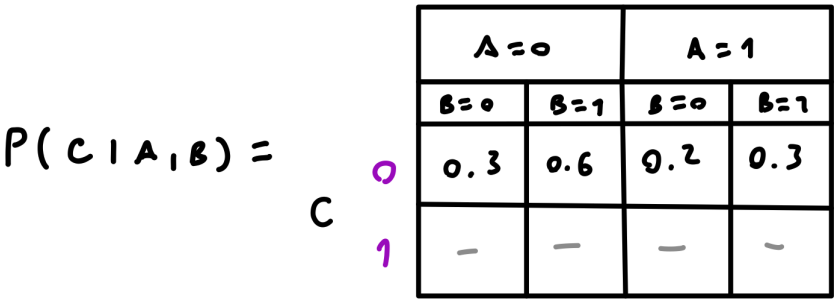
Starting from a graphical representation, we can derive a table of conditional probabilities to represent the dependencies between all the random variables in the form of tabular data.

Let’s suppose that we have $B, C \in \{0, 1\}$. The corresponding tabular representation will be something like this:

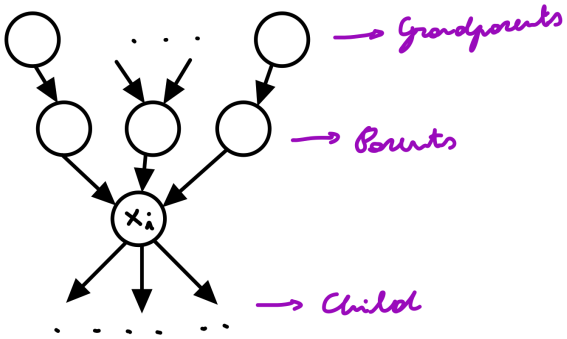


We can express these tables in a more compact way by deleting the last row, since its values can be derived from theprevious ones.

The same can be done with multiple variables, like in this example:



Given a certain node x_i, we call parents all the nodes that are directly connected to it (and with the arrow pointing towards x_i). Bayesian networks have the property that every node is conditionally dependent only from its parents, meaning that a node x_i will be independent from all the nodes that are not his parents. This means that we will express the conditional probability of a node only considering its parents.



We can represent the property of conditional independence for bayesian networks as follows:

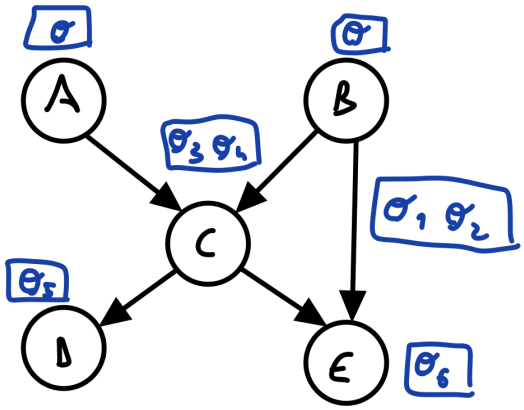
$$P(x_i|Parent(x_i)) = P(P(x_i|Parent(x_i), Z)$$

This property makes us able to represent complex problems in a very compact way.

When computing probabilities involving many variables, we can simpy exploit the chain rule and the probability tables:

$$P(A, B, C, D, E) = P(A|B, C, D, E) \cdot P(B|C, D, E)...$$

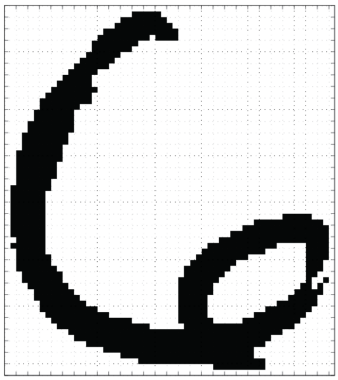
We can express a parametric model of bayesian networks by assigning parameters to each relation:

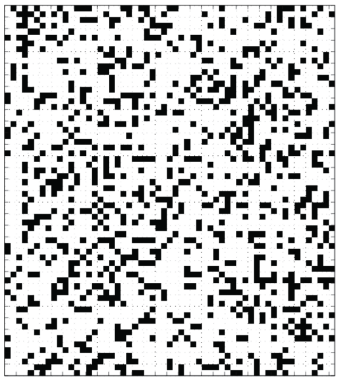


Dimensionality reduction

Introduction

In many cases (like images) we find ourselves to work with very high dimensional inputs. For example, a space $X \in \mathbb{R}^{w \times h \times c}$ has $w \times h \times c$ degrees of freedom. But often it happens that not all the combination of the input are meaningful. In fact actual data may present a much lower variability (for example if we are analyzing a dataset of numbers, only certain combination of pixels make sense in the context of the dataset):





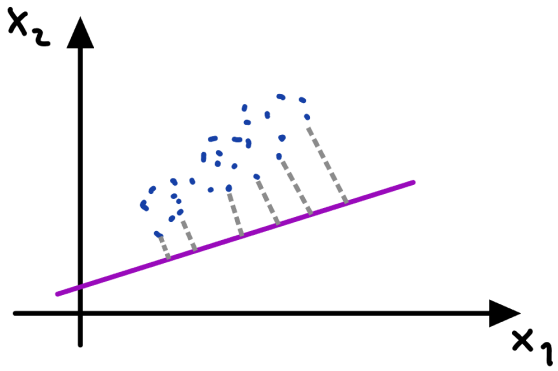
Even if this is a valid combination of pixels, it doesn't make sense in a digits datset.

The field of data reduction studies the possible transformation that can be applied to the input space in order to obtain compact (but still meaningful) representation of it in

lower dimensions. When performing these transformations the risk is to lose information, or obtaining only partial representations of the problem. The following methods aim at finding the best tradeoff between dimensionality reduction and information retention.

Principal component analysis (PCA)

Imagine that we want to apply a transformation of a 2D dataset in $\mathbb{R}^2$ into $\mathbb{R}$. What we do is basically projecting every point in 2 dimensions onto a 1D line:

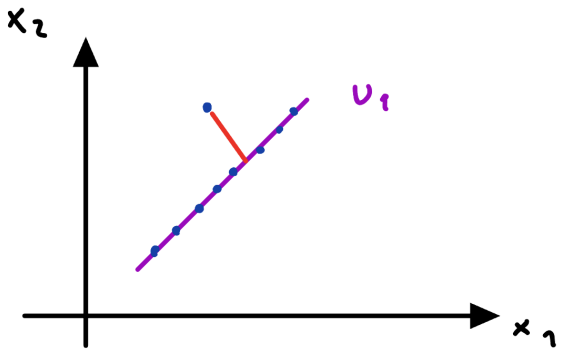


But *what is the best representation?*

What we want is the transformation that guarantees the minimum error (meaning the maximum retention of information).

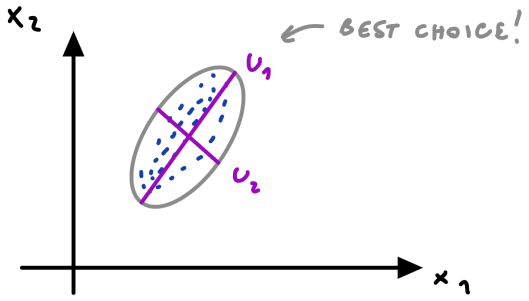
The simplest case is 2D datas that already lie on a linear pattern. Here we have no error in the representation u_1:

If we introduce some variability to the data, we start loosing informations in the representation:



Actually, u_1 still remains the best choice, because despite loosing the information for one point, still retains the informations about the majority of points.

In general, the best choice is the one that follows the direction of maximum variance of the data, like in the following example:



In general, what we want is to produce a transformation $\mathbb{R}^M \rightarrow \mathbb{R}^D$ where D is the number of directions with the maximum variance. We denote the directions of the input space as $x_1, ..., x_M$ and the ones of the transformes space as $u_1, ..., u_D$. The resulting output vector will be:

$$< u_1^T x_n, ..., u_i^T x_n, ..., u_D^T x_n >$$

A fundamental property of the input space is that all of its components are orthogonal, meaning that holds the property that $u_i \cdot u_j = 1$ if they are the same, 0 otherwise. These directions will follow the base vectors that maximize variance of the projected points.

Maximizing the variance

First, we find the mean of the data points $\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n$. Then we find the data-centered matrix, which is the result of rescaling all data in order to have mean 0:

$$X = \begin{bmatrix} (x_1 - \bar{x}^T) \\ ... \\ (x_N - \bar{x})^T \end{bmatrix}$$

This procedure is also called normalization of the input data.

We can now fix a certain direction u_1 with the goal of using it to find the direction of maximum variance.

The mean for the projected points will be $u_1^T \bar{x}$. Their variance will correspond to the sum of the squared distances from the mean:

$$\frac{1}{N} \sum_{n=1}^{N} [u_1^T x_n - u_1^T \bar{x}]^2 = u_1^T S u_1$$

Where $S = \frac{1}{N} \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^T = \frac{1}{N} X^T X$.

What we want is to solve the maximization problem by finding the direction that maximizes the variance:

$$max_{u_1} u_1^T S u_1$$

The solution is: $Su_1 = \lambda_1 u_1$. This means that u_1 is the eigenvector for S and λ_1 is its eigenvalue. Hence, the maximum direction will correspond to the maximum among the eigenvalues of S.

Here is a summary for this approach:

Given a dataset, we compute the data-centered matrix and the covariance matrix S. From S, we extract the λ_i eigenvalues and we order them in a decreasin order. If we are transforming the data into a D-dimensional space, we take the D highest eigenvalues, that correspond to the ones that will be the optimum for the variance maximizaton.

The largest higenvalue of the correlation matrix λ_i is called **principal component**.

Each transformed data point will be represented as:

$$x_n = \sum_{i=1}^{D} \alpha_{ij} u_i$$

Where $\alpha_{ij} = x_n^T u_j$. By exploiting the property of orthonormality of the transformed space, we can reduce the expression as:

$$x_n = \sum_{i=1}^{D} (x_n^T u_i) u_i$$

When reducing the representation to an M dimensional space, we are approximating the point in lower dimensions. Hence, we can express this approximation as the results of two contributions:

$$\tilde{x} = \sum_{i=1}^{M} z_{ni} u_i + \sum_{i=M+1}^{D} b_i u_i$$

Now we can define the error between the original point and the approximated one as the summ of the squared distance between them:

$$J = \frac{1}{N} \sum_{i=1}^{N} ||x_n - \tilde{x}_n||^2$$

How do we reduce this error?

Remember that the approximation is made up by two contributions. The first one is given by the coefficients $z_{ni} = x_n^T u_j \text{ for } i = 1, ..., M$ which is computed from the actual points, while the real error depends on the last $M + 1, ..., D$ contributions, that only depend on the mean of the contributions. Thus, the actual representation of the error is:

$$x_n - \tilde{x}_n = \sum_{i=M+1}^{D} [(x_n - \bar{x})^T u_i] u_i$$

If we sum the contribution of the errors for all the samples in the dataset, then we obtain:

$$J = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=M+1}^{D} (x_n^T u_i - \bar{x}^T u_i)^2 = \sum_{i=M+1}^{D} u_i^T S u_i$$

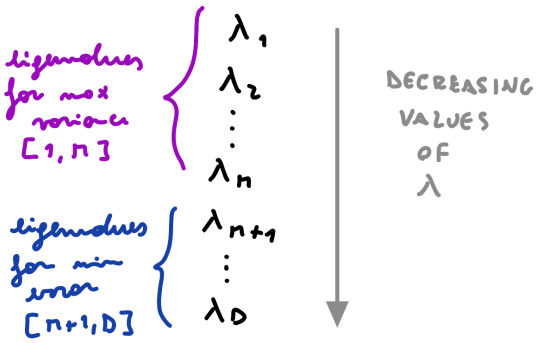
What we have obtaines is a representation of the error based on the covariance!

The formulation we see in particular is the same that we've seen when maximizing the variance, thus, the solutions of the minimization problem for the error will depend on the same relationship as before:

$$Su_i = \lambda_i u_i$$

This means that the minimization of the error will depend on the eigenvalues of S. In order to obtain the minimum error, we will simpy have to choose the $M+1, ..., D$ lowest eigenvalues of S.

This means that the whole problem of maximizing variance and minimizing the approximation error can be boiled down to a single operation. In fact, if we choose the highest M eigenvalues for the maximization of variance, we will automatically obtain the remaining $M+1, ..., D$ eigenvalues that minimize the error:



PCA for high dimensional data

In some cases, it can happen that the number of samples N is smalled than the number of dimensions D of the space they belong to. When this happens, computing the eigenvalues of the covariance matrix is inefficient, since the computation will have to deal with a total of $D \times D$ dimensions.

In cases where $N < D$ we can make a strategic choice for the computation of the eigenvalues by considering the matrix XX^T, which is an $N \times N$ dimensional matrix. Using this trick will allow us to find the same eigenvalues that we need, but searching in a smalled space.

Probabilistic PCA

With probabilistic PCA we define a linear relationship between the actual data $x \in \mathbb{R}^D$ and lower dimentional latent variable $z \in \mathbb{R}^M$. The linear realationship is modeled as:

$$x = Wz + u$$

We make two fundamental assumptions. The first one is that the latent variables follow a normal Gaussian Distribution:

$$P(z) = \mathcal{N}(z; 0, I)$$

The second one is that the conditional probability of x given z follows a linear Gaussian relationship:

$$P(x|z) = \mathcal{N}(x; Wz + \mu, \sigma^2 I)$$

An interesting way of interpreting this probabilistic relationship is that it can be used as a generative model. In fact if we know all the parameters $< W, \mu, \sigma >$, we can sample new data from the obtained distribution based on the values of z.

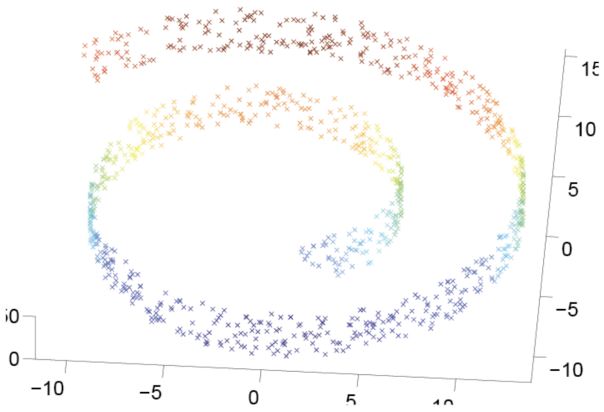
The goal with probabilistic PCA is to find the parameters $< W, \mu, \sigma >$ that maximize the likelihood of the previous relationship:

$$argmax_{W,\mu,\sigma}\ P(x|W, \mu, \sigma^2) = \sum_{n=1}^{N} ln\ P(x_n|W, \mu, \sigma^2)$$

When we set the derivatives to 0, we obtain the solution of the maximization problem in a closed form (here we omit the actual solution).

Non-linear latent variables models

Up until now, we've seen that PCA assumes a linear relationship between the data and the latent variables, but actually this is not always the case. Sometimes there is the need for transforming data that follow a non-linear pattern, like the ones shown in the following image:

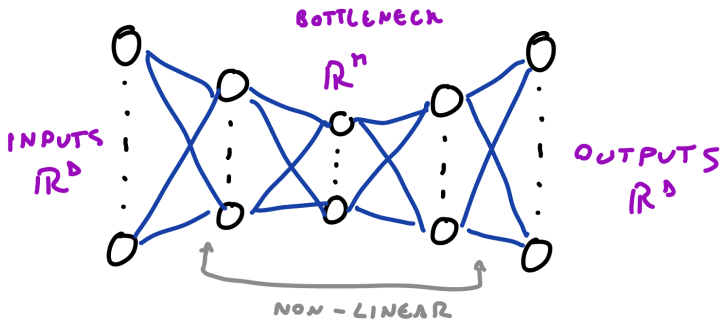


In order to deal with these problem we need to use non-linear models, and neural networks serve well this purpose. The following methods show how we can exploit

them in various cases.

Autoencoders

Autoencoders are a special type of neural networks in terms of architecture. In particular the input and output layer will be of the same dimensions D, while the hidden layer will be of a lower dimension M (called bottleneck):

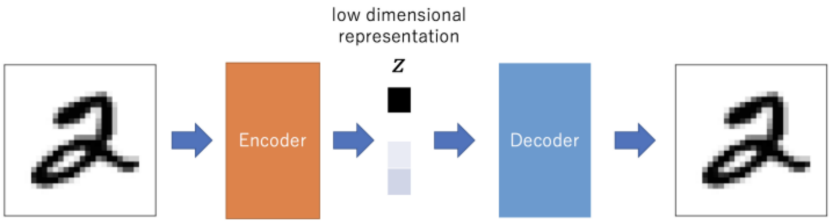


They work by transforming the input x_n into an intermediate representation $z \in \mathbb{R}^M$ and finally reconstructing it into the output x'_n.

An ideal model will perfectly reconstruct the given input such that $x_n = x'_n$.

The great intuition here is that we train the network in order to perform well this operation, the model will have learned how to represent the input space into a reduced space by loosing the minimum amount of information.

The training of an autoencoder is done by giving x_n both as input and as target value. The representation of x_n in the latent space is denoted as z_n.

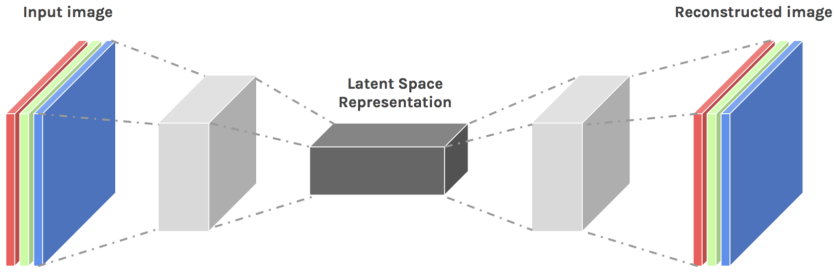


The first part of the network that transforms the input space is called encoder, while the one that reconstructs the output from the latent space is the decoder.

Notice that after training, if we remove the encoder, we can use the decoder as a generative model that outputs new samples starting from the values assigned to the latent space.

Convolutional autoencoders

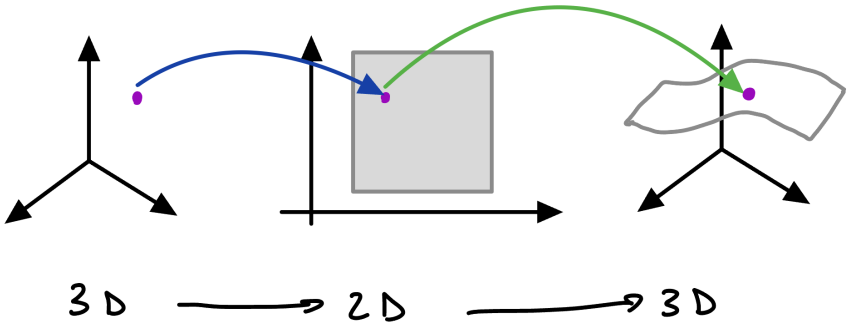
In case of high dimensional data such as images we use convolutional autoencoders, which exploit the concept of convolutional operations to encode the inputs.



The representation in the latent space will then be decoded by a convolutional transposed layer, which in brief does the reverse convolution operation. As with normal autoencoders, we use the same images as input and targets in order to train the network. By doing this, we will be able to use convolutional autoencoders to generate new images.

What is happening behind the scenes?

When using autoencoders we are using a non linear transformation to encode the data in a lower dimensional space. This data will then be re-projected onto the original higher dimensional space, and will tipically lie on a manifold. The following is a representation of what happens when transforming a 3D input in 2D:



Autoencoders for anomaly detection

There are cases when we want to understand whether a certain input is not part of the dataset. This field of study is called anomaly detection and aims at telling apart

abnormal that. Since we want to distinguish normal from abnormal, the problem is also called one-class classiication.

A problem when considering what data to consider is that collecting samples of anomalies does not always make sense, since new anomalies are not predicatble (new anomalies can be very different form pas ones). Thus the approach is based on training only on a dataset that represent normal samples.

Remember that the task is basically a binay classification where we eant to find a function such that $f : X \to \{normal, abnormal\}$. The problem is that we have a dataset that contains only samples from one class $D = \{(x_n normal)_{n=1}^{N}\}$.

How do we deal with that?

A simple but effective solution is the following:

1. Start from a dataset containing unlabelled data $D = \{x_n\}$.
2. Train the autoencoder on the dataset and compute the final training loss.
3. Define a certain threshold based on the loss. For example $\delta = mean(loss) + std(loss)$
4. When we have a new sample x', we can try to use the autoencoder to reconstruct it. We then compute the loss of the autoencoder on this reconstruction. If this loss is higher than the threshold, we consider the new sample as an anomaly, otherwise we consider it as normal.

This method works better when used on an ensemble of autoencoder, where we use the average of the losses for each reconstruction.

Generative models

As we have seen, autoencoders can be used to generate new samples in the input space, but it’s not the purpose they were born to serve. Instead, there are models that were specifically designed for generating new data that is similar to a distribution. The goal of these models is to find a dimensionality reduction that is able to reconstruct an output that is as similar as possible to the input. In particular, we will focus on Variable Autoencoders and Generative adversarial networks.

Variational Autoencoders (VAEs)

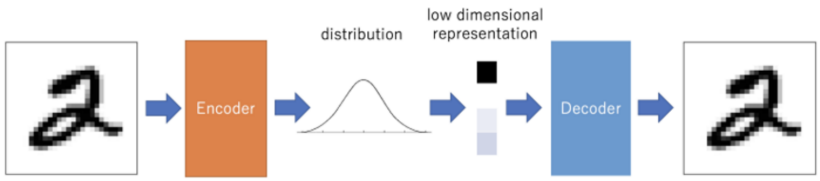
Classical autoencoders are very good at encoding informations in a compact latent space, but despite the latent space can be used to generate new sample, we don’t have proper control on the generation process. In fact since there is no constraint on

how the latent space is organized, two points very closed to each other can produce really different outputs.

We have to find a way to properly characterize regions of the latent space in order to map them to determined outputs in the originalspace. Variational autoencoder serve this purpose by making sure that the distributions in the latent space are well defined in terms of Gaussian distribution.

This means that VAEs will learn the distribution of certain data in terms of Gaussians, such that points that are clos in the latent space are also similar in the latent space. This will give us much more control on the process of generations, since we know that small perturbation in the latent space will produce similar results.

VAEs implement this concept by defining the latent space in terms of probability distributions that dipend on the parameters μ, σ. Therefore, the encoder will produce a distribution instead of a vector in the latent space, while the function of the decoder will be to sample from the computed distributions:



How do we produce these distributions?

The model will still be trained by giving both as taget and input the same value, but what changes is the definition of the loss function, which looks like this:

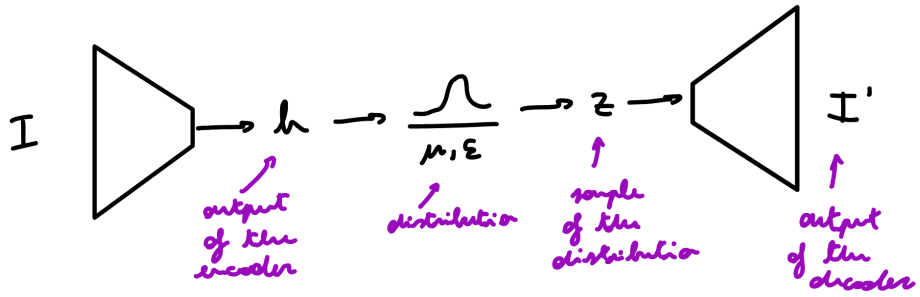
$$loss = MSQE + \lambda \cdot KL(\mathcal{N}(z; \mu, \Sigma), \mathcal{N}(z; 0, I))$$

Notice that while we kept the mean squared error term (that measures the difference from the original and the reconstructed sample), we added a new term KL. This new term is known as the Kullback-Leibler divergence, and is a method to measure the divergence between two distributions. In particular, this term will allow us to shape the latent space to be as close as possible to a normal distribution. The term λ will regulate the importance of this morphing.

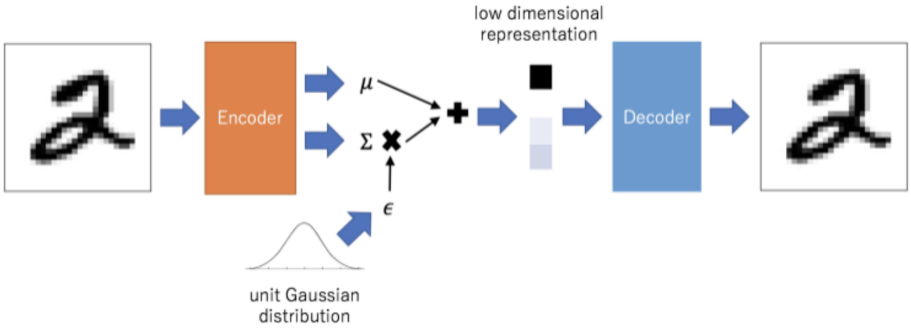
Minimizing the error will allow us to find the best tradeoff between a good reconstruction of the output and a good representation of the latent space.

And now, how we get an output from the obtained distribution?

We will need to sample from this distribution. Basically we want a method that works like this:



While this operation is easy to do in the forward step from input to output, we have a serious problem when implementing backpropagation. In fact we would have to perform some sort of reverse samplimng, but sampling is not differentiable. This problem is solved by substituting the distribution with a single value ϵ to the distribution (re-parametrization) when computing the gradient:

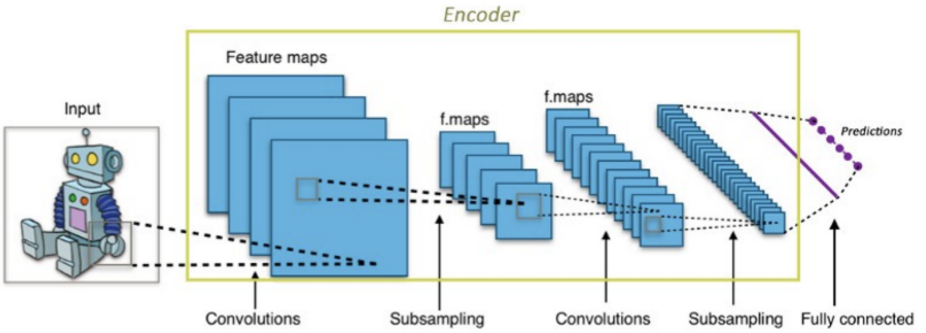


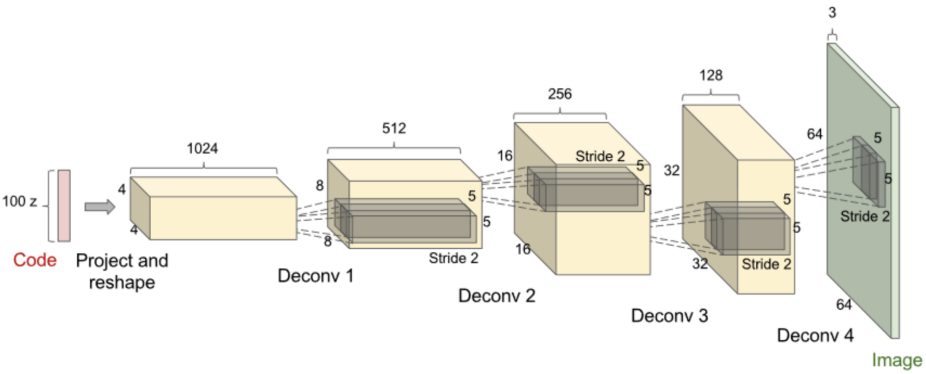
Generative Adversarial Networks (GANs)

Even if VAEs are better at the generation task, we have to remember that autoencoders were born for the task of dimensionality reduction. But when thinking about generative tasks, we don't actually need this feature. What we will study here is a family of models that were born with the specific purpose of generating data.

We will focus on a family of algorithms based on the concept of adversarial training. In particular, the core idea is to make two networks compete on the same task.

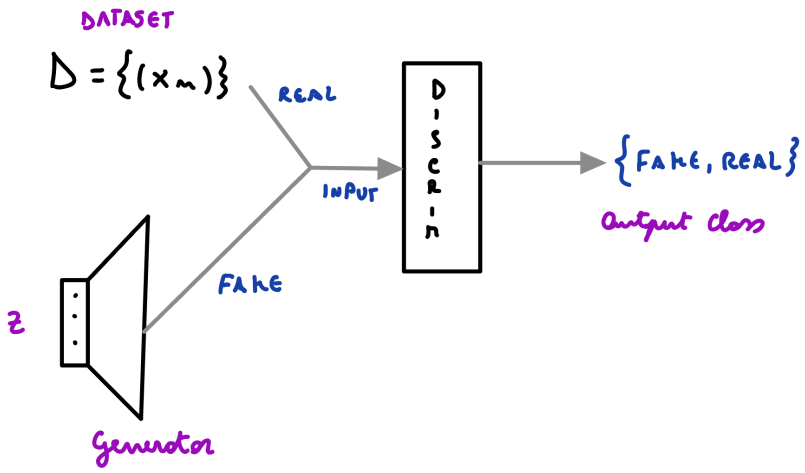
The typical archtecture of Generative Adversarial Networks is to have two components: a **discriminator** and a **generator**.





In particular, the discriminator is actually a decoder that works as usual. WIth respect to autoencoders, we won't need an encoder, but our only task will be to train the generator.

Our input will be an unlabelled dataset $D = \{(x_n)_{n=1}^N\}$. Now we introduce the discriminator, which is a binary classifier, whose task is to tell apart the images coming form the dataset (considered as real) from the ones generated by the generator (considered as fake). Basically, the discriminator has to understand whether an input data is real or fake.



Here, the two dicriminator and generator entities have different point of views. In fact the generator is good if he's able to fool the discriminator, meaning that it can make it classify generated images as real. Instead, the discriminator works well when it's able to properly lable generated data as fake. This is the core idea of the adversarial conept.

The training phase of a GAN has two main phases that are iterated:

1. Train the discriminator while freezing the weights of the generator.
2. Train the generator while freezing the weights of the discriminator.

During the first phase, the images produced by the generator will be labelled as fake such that the discriminator will keep getting better. Vice-versa, during the decond phae the images from the generator will be labelled as real.

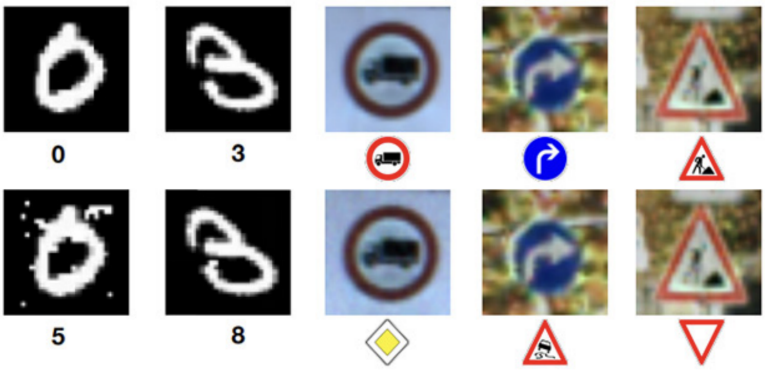
The first time we train the discriminator, the generator's latent space is initialized with random values.

When doing the training that while the loss of one entity increases, the other one decreases. We will stop training when the loss of the two components will converge to a minimum. Notice that we never want the loss of one compomenent to be zero, since this would mean that the other loss will diverge to infinity.

The GAN will work when tipically when the loss of the discirminator will reach about 50%.

Adversarial attacks to ML models

A very dangerous use of GANs is that they can be trained to fool already existing ML models. In particular this can be very impactfuls when using them against anomaly detectors.



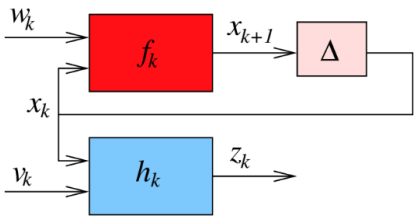
Reinforcement learning

Dynamic systems

A dynamic system can be thought of as the composition of many interconnected units that evolves over time. This evolution is represented by the state of the system, that depends on the dymanics that make the system evolve.

When studying reinforcement learning, dynamic systems are characterized by the following parts:

- State X
- Observations Z
- Noise w,v
- State transition model f
- Observation model h



When studying a dynamic system we have two possible approaches, that depend on the informations that are available to us.

If we assume that all the informations about the system are exposed, then we work with **reasoning**. In fact having all the informations lets us know predict the future states of a system without making any action.

When we don't have a fully observable model of the system, what we can do is **learning** by executing actions in the environment and try to reconstruct the model.

The state of a system

The state of a dynamic system can be considered as a snapshot of the system at a given time (imagine that we can freeze the system for an instant and read its internal state).

A state is fully observable if at any given time an agent can fully read it. In particular, the task of an agent is to choose the best action to perform for each state. In particular, we want to define a **policy** function that maps states to actions:

$$\pi = X \to A$$

Where X is the set of all the possibe states and A the set of all the possible actions. When the model of the system is not known we will have to learn the policy.

The evolution of a system can be encoded as a set of sequences of states, actions and rewards that are updated during learning:

$$D = \{< x_1, a_1, r_1, x_2, ..., x_n, a_n, r_n >^{(i)}\}$$

Rewards are basically signals that the agent receives after having performed an action in a certain state. We use rewards to make the agent learn.

Markov Decision Processes (MDP)

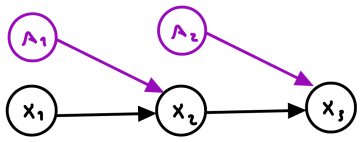
The Markov property

The Markov property is a fundamental property based on conditional independence between states. In particular it says that we just need to know the state and action at time t in order to compute the state at time t+1. In other words, the probability of the future depends only on the current state:

$$P(future|present, past) = P(future|present)$$

The models based on the assumptions made so far are called **Marvov Decision Processes (MDP)**.

We represent graphically these processes using the notation of Bayesian networks:



Here for example $P(x_2|x_1, a_1)$ and $P(x_3|x_2, a_2)$.

The formal definicion of an MDP is:

$$MDP =< Z, A, \delta, r >$$

Where:

- X is a finite set of states
- A is a finite set of actions
- $\delta : X \times A \rightarrow X$ is the transition function
- $r : X \times A \rightarrow \mathbb{R}$ is the reward function

The computations of states and rewards can be defined using the Markov assumption as:

$$x_{t+1} = \delta(x_t, a_t)$$
$$r_t = r(x_t, a_t)$$

In particular transitions can be of two types. **Deterministic transitions** map a state action pair to one possible evolution. Instead **non-deterministic transitions** happen when instead of having a fixed transition, we have a set of possible states that we might reach with a certain probability.

In the non-deterministic case, the transition function will be based on a cerain probability distribution in order to compute the next state. In other words we will have a stochastic representation of the transition function:

$$\delta = P(x_{t+1}|x_t, a_t)$$

In order to obtain an evolution $< x_1, a_1, r_1, x_2 >$, three steps are needed:

- The policy determines the action to execute based on the current state $a_1 = \pi(x_1)$
- The reward function computes the reward $r_1 = r(x_1, a_1)$
- The transition function determines the next state $x_2 = \delta(x_1, a_1)$

Solving an MDP

Solving a Markov Decision Process means finding the optimal policy, namely the best behaviour for the agent. In particular, the best behaviour is the one that maximizes the reward in the shortest time.

How do we define the reward?

In order to have the best reward in the shortest time we introduce the discount facotr $\gamma < 1$. Its role is to penalize the rewards over time, in fact we will consider $r_1 + \gamma r_2 + \gamma^2 r_3 + ...$

If the process that we are studying is non deterministic, we will consider the expected value of all. the rewards $E[r_1 + \gamma r_2 + \gamma^2 r_3 + ...]$.

This is also called **cumulative discounted reward**.

We can use the cumulative discounted reward as a metric for measuring how good is the policy. We define the value function of a policy as:

$$V^{\pi} = E[r_1 + \gamma r_2 + \gamma^2 r_3 + ...]$$

In other words, our goal will be to find the policy that maximizes the value function. We say that π_1 is better than π_2 if:

$$V^{\pi_1}(x_1) > V^{\pi_2}(x_1)$$

The optimal policy is the one with the highest value function (there could be more equivalent optimal policies):

$$V^{\pi^*}(x_1) \geq V^{\pi}(x_1) \;\; \forall x_1 \in X$$

We denote the value of the optimal policy as $V^*(x_1)$.

One state MDP

We start by considered an MDP where we have one state x_0 and many possible actions that always lead to x_0:

This is also defined as the bandit problem, since it remembers the dynamics of a slot machine with n available levers to pull (actions). Each lever will produce a certain reward.

Let's now consider some different situations we could find ourselves in.

The rewards are deterministic and known

In this case we have a table that associates each action to a reward. Here the optimal policy is simply to choose the action that will return the maximum rewards, hence zero trials are needed.

$$\pi(x_0) = argmax_{a_i}\{r_i\}$$

The rewards are deterministic, but not known

Here we just know the available actions. The optimal strategy is to try each action and observe the reward. We are guaranteed to find the optimal policy by performing n trials in total since the reward is still deterministic. Therefore we will have the same formulation for the optimal policy:

$$\pi(x_0) = argmax_{a_i}\{r_i\}$$

The reward is known, but non-deterministic

We can think this situation as having a table that associates each action to a certain probability distribution for the reward in terms of mean and variance. The best strategy will be to choose the distribution with the highest mean:

$$\pi(x_0) = argmax_{a_i}\{\mu_i\}$$

This strategy takes 0 trials.

The reward is non-deterministic and not known

Here the rewards is stochastic and its distribution is not known. A first strategy we can imagine is to repeat each action for a certain amout of trials and compute the means for each action. But this is not really an optimal strategy.

A better approach is to compute the means *online*. This means that we will update the value of the means in real time and update the strategy accordingly. In particular we will perform an action at each step and update the mean by collecting the reward. In practice we will implement an algortithm that will update a data structure at each step like this:

1. Initialize the data structure Θ
2. For each time $t = 1, ..., T$ (until termination condition):
    a. Choose an action $a_t \in A$
    b. Execute a_t and collect the reward r_t
    c. Update the data structure Θ
3. Return the optimal policy $\pi^*(x_0)$ according to Θ

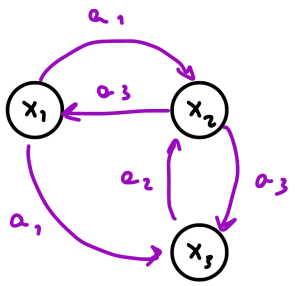
If we assume that the reward follows a Gaussian distribution, we can define the algorithm as follows:

1. Initialize $\Theta_{(0)}[i] \leftarrow 0$ and $c[i] \leftarrow 0$ with $i = 1, ..., |A|$
2. For each time $t = 1, ..., T$:
    a. Choose an index $\hat{i}$ for action $a_t = a_{\hat{i}} \in A$
    b. Execute a_t and collect the reward r_t
    c. Increment $c[\hat{i}]$
    d. Update $\Theta_t[i] \leftarrow \frac{1}{c[\hat{i}]}(r_t + (c[\hat{i}] - 1)\Theta_{t-1}[\hat{i}])$
3. The optimal policy will be $\pi^*(x_0) = a_i$ with $i = argmax_{i=1,...,|A|}\Theta$

This is the skeleton of the algorithm that we will actually implement. An important feature of the online approach is that it can adapt even to situations where the reward changes over time.

The general case

In the general case we consider multiple states, for example:



When we don't know both the transition and reward functions, what we can do is learning through executing actions and collecting data about the reward obtained and the next state.

A first approach to learning is *value iteration*. We start from the notion that the best action to execute in a certain policy depends on the optimal policy:

$$\pi^* = argmax_{a \in A}[r(x, a) + \gamma V^*(\delta(x.a))]$$

Notice that the current formulation depends on functions we don't have. But *do we really need them?*

What we really want is the quantity expressed by these function. We define it using the Q function:

$$Q^*(x, a) = r(x, a) + \gamma V^*(\delta(x.a))$$

If we find the optimal Q function, then the problem is basically solved. What we want is:

$$\pi^*(x) = argmax_a\{Q^*(x,a)\}$$

We will start with an underestimate for the optimal Q function and try get as close as possible to it over time.

Even if at first we don't know it, we have to remember that each state has its own value. The Q function will express the value of that state as the expected reward if we used the optimal policy.

When we start from scratch, the strategy is to assign a value of 0 to every state. As the iterations go on, we will continuosly update the values after executing some action. A fundamental assumption of this strategy is that the rewards are all non-negatives ($r(x,a) \geq 0$). If the system has negative rewards, we simply re-scale them. Each time that an action is performed in a certain state, the corresponding reward will become explicit, and we will exploit this knowledge to learn.

Learning the Q function: the deterministic case

Since we don't know the optimal Q function a priori, we start from an underestimate $\hat{Q}$, that we will evaluate this way:

1. Fix a state x_1 and perform all the possible actions
2. Collect the rewards associated to each state
3. The best action will be the one that produced the highest reward
4. Update $\hat{Q}(x_1,a) \leftarrow r(x_1,a) + \gamma max_{a' \in A}\hat{Q}(x',a')$

At each step the value on the underestimate will keep increasing and tend to the optimal value Q^*.

The notation that we use for the Q function associated to a certain policy π is $Q^{\pi}(x,a)$, while for the optimal function, we only use $Q(x,a)$.

Notice that:

$$V^*(x) = max_{a \in A}\{r(x,a) + \gamma V^*(\delta(x,a))\} = max_{a \in A}Q(x,a)$$

Hence, holds the equality $Q(x,a) = r(x,a) + \gamma V^*(\delta(x,a))$ that lead us to the following formula:

$$Q(x,a) = r(x,a) + \gamma\ max_{a \in A}Q(\delta(x,a),a')$$

Meaning that the optimal Q function depends from the immefiate rewards and the value that maximized the Q function. This helps us to define the training rule as:

$$\hat{Q}(x,a) \leftarrow \bar{r} + \gamma\ max_{a'}\hat{Q}(x',a')$$

The skeleton of the learning algorithms is:

1. For each possible state-action pair (x,a) initialize a table entry $\hat{Q}_{(0)}(x,a) \leftarrow 0$
2. Observe the current state x
3. For each time $t = 1, ..., T$:
    a. Choose an action a
    b. Execute a
    c. Observe the new state x'
    d. Update the entry of the Q underestimate as $Q_{(t)}(x,a) \leftarrow \bar{r} + \gamma\ max_{a' \in A}\hat{Q}_{(t-1)}(x',a')$
    e. $x \leftarrow x'$
4. Return the optimal policy $\pi^*(x) = argmax_{a \in A}\hat{Q}_{(T)}(x,a)$

Since the reward is zero, we will be guaranteed that our estimate will improve over time:

$$0 \leq \hat{Q}_{(n)}(x,a) \leq \hat{Q}_{(n+1)}(x,a) \leq Q(x,a)$$

The convergence will be guaranteed if each state-action paire is visited infinitely often, meaning that the possibility of performing every available action is non zero.

How to choose which action to perform?

Here we have two possible stratiegies:

- **Exploitation**: select each time the action that maximizes the value $\hat{Q}(x,a)$ (if known)

- **Exploration**: select the action randomly

In pratice, we have to find a tradeoff between exploration and exploitation. In facti while exploitation guarantees to seek the maximum value, exploration lets us explore more possible (better) paths.

How to find the right balance?

A possible solution is to implement an ϵ-greedy strategy. Given a parameter $0 \leq \epsilon \leq 1$, we select a random action with probability ϵ, or the best action with probability $1 - \epsilon$.

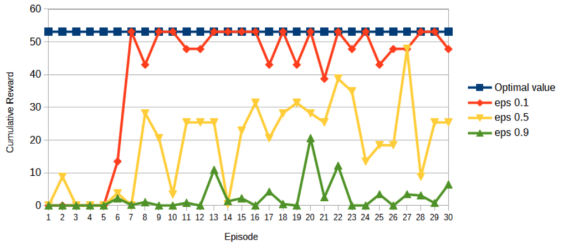
A common aproach is to start with a high value of ϵ in order to favour exploration, and decrease it over time.

Another strategy is softmax, that favours the choice of the probability of choosing the avtion with the higher value.

Evaluating RL agents

Let's suppose that we trained an agent. How do we evaluate the policy he learned?

A possibility is to plot the cumulative reward as the agent learns. Since we have an alternation between exploration and exploitation, the resulting that can be very noisy:



A more useful approach consists in executing k steps of learning an then evaluating the current policy based on the average of the cumulative rewards obtained.

Moreover, other domain specific metrics can be defined.

Learning in the non-deterministic case

In the non-deterministic case actions can lead to different possible states with a certain probability.

In this case it makes sense to consider the expected value of the cumulative discounted reward:

$$V^{\pi}(x) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + ...] = E[\sum_{i=0}^{\infty} \gamma^i r_{t+1}]$$

The optimal policy will be the one that maximizes the expected value:

$$\pi^* = argmax_{\pi} V^{\pi}(x) \; \forall x$$

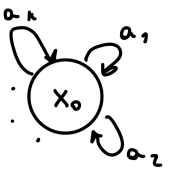
From this observations, we can derive the following equality:

$$Q(x,a) = E[r(x,a) + \gamma V^*(\delta(x,a))] = ... = E[r(x,a)] + \gamma \sum_{x'} P(x'|x,a) V^*(x') = E[r(x,a)] + \gamma \sum_{x'} P(x'|x,a) m$$

The optimal policy will be $\pi^*(x) = argmax_{a \in A} Q(x,a)$.

K-armed bandit

Imagine that we have one state x_0 and k possible levers to pull (actions) $a_1, ..., a_k$:



Let's assume that the reward is stochastic and follows a Gaussian distribution:

$$r(a_i) = \mathcal{N}(\mu_i, \sigma_i)$$

In general we know that it's best to choose the action that corresponds to the reward with the maximum mean. But what if we don't know the parameters of the distributions?

We can approximate the distribution by computing an average reward and update it at each new step. The training rule is:

$$Q_n(a_i) \leftarrow Q_{n-1}(a_i) + \alpha[\bar{r} - Q_{n-1}(a_i)]$$

Where $\alpha = \frac{1}{1+v_{n-1}(a_i)}$ and v_{n-1} the number of execution of action a_i up to time n-1.

Learning in the general case

In the general case we have many actions associated to many possible states. Here we have to adapt the previous logic to the need of taking into account many states:

$$\hat{Q}_n(x,a) \leftarrow \hat{Q}_{n-1}(x,a) + \alpha[r + \gamma\, max_{a'}\hat{Q}_{n-1}(x',a') - \hat{Q}_{n-1}(x,a)]$$

Where the term $r + \gamma\, max_{a'}\hat{Q}_{n-1}(x',a')$ represents the cumulative average over the state.

Whle the approaches we have seen so far are valid, the actual functions used in learning are the parametrized versions of the previous ones. In fact the Q value can be represented as a functions instead of tabular data:

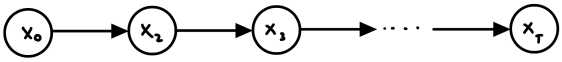
$$Q_\theta(x,a) = \theta_0 + \theta_1 F_1(x,a) + ... + \theta_n F_n(x,a)$$

An important feature of this representation is that it allows us to take in account similarities between states. For example, is a state gives an high reward, it is more likely that states near it can be considered good as well. Morevore, assigning weights to the Q functions allows us to learn by exploiting deep learning (Deep Reinforcement Learning), that allows us to deal with the continuous case as well.

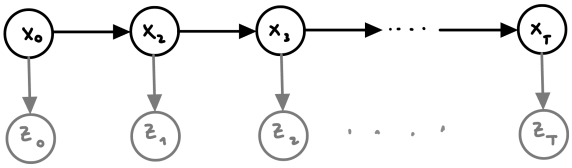
Hidden Markov Models (HMM)

Markov chain

The evolution of a dynamic system can be modeled as a random process based on the markov assumptions. The graphical representation of these states is called markov chain:



In many cases we have to deal with situation where the actuale value of the state is not known (we say that the state is not observable). What the agent can do is observe it from the outside through some obervations, that can be modeled as random variables $z_0, z_1, ..., z_T$:

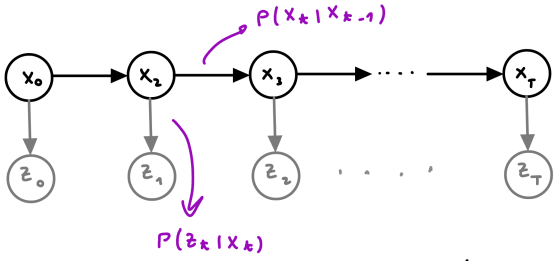


Problems that are modeled in this ways are called Hidden Markov Models. HMMs are formally defined as a set of states X, observations Z and initial distribution π_0:

$$HMM =< X, Z, \pi_0 >$$

Here we can't influence the evolution of the system because we don't have access to actions. What we have is.

- A transition model $P(x_t|x_{t-1})$
- An observation model $P(z_t|x_t)$
- An initial distribution over the states π_0



The probability distribution of being in a certain state given that we were in another state at the previous step can be modeled as a matrix. For example, let's suppose that we have three state $X = \{a, b, c\}$. Here the transition model is represented as $P(x_t = \{a, b, c\} | x_{t-1} = \{a, b, c\})$.

The matrix model is:

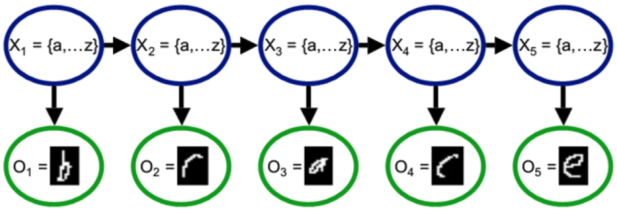
	$x_{t-1}=a$	$x_{t-1}=b$	$x_{t-1}=c$
$x_t=a$	:	0.3	0.1
$x_t=b$	:	0.5	0.2
$x_t=c$	:	0.2	0.1

Let's assume that we have two observations available $Z = \{\alpha, \beta\}$, the relation between them and the actual states can be encoded in an observation matrix:

	$x_t=a$	$x_t=b$	$x_t=c$
$z_t=\alpha$	0.1	0.3	..
$z_t=\beta$	0.4	0.7	..

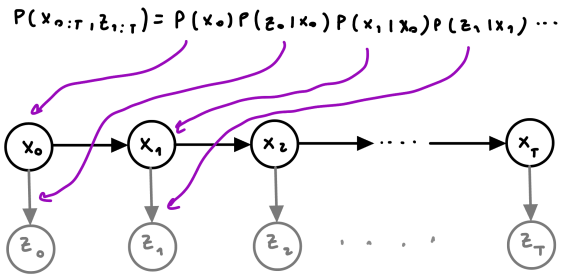
In cases where the state is modelled as a Gaussian distribution, meaning that $x_t = \mathcal{N}(\mu_t, \sigma_t)$, then observations are continuous and modelled as $P(z_t | x_t = \{a, b, c\})$ (the observations will follow a certain gaussian distribution, that will depend on the state).

The power of hidden markov models is that they can model a problem by analyzing the evolution of the system. For example in the following image we see that the caracter $O_2 = r$ and $O_4 = c$ are written in a similar way. But using HMM, we can take into account that given the previous letter, it is more likely to have a c in place 4 (brace is a word that makes sense, brare not).



Chain rule in HMM

Since HMM ar ebased on the Markov's assumption of conditional independence, computing the probability of being at state T given a certain observation can be computing by applying the chain rule as illustrated here:



In particular, when the model of an HMM is known, we can compute all these informations directly.

Two important problems where HMMs are applied are:

- **Filtering**: estimate the state at time T given the previous states $P(x_T = k|z_{1:T})$
- **Smoothing**: given all the observation done until time T, find the value of a certain state in the past.

Learning in HMM

In many cases it happens that we don't know the model of the problem. A first possible case is the following: let's assume that we have an initial phase of the training where we can observe the actual state. Here we can simply estimate the value of all possible cases as:

Transition model computed as i,j transitions vs all the other possible transitions

$$A_{ij} = \frac{|i \to j \text{ transitions}|}{|i \to k \text{ transitions}|}$$

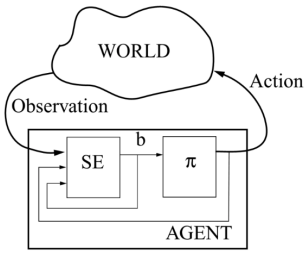
Observation model computed by considering all the time we made an observation v and the state is k, vs all the other observations in that state.

$$b_k(v) = \frac{|\text{observe } v \wedge \text{state } k|}{|\text{observe } k \wedge \text{state } k|}$$

If we can't observe the state during training, then what we can do is learning through observation by using the expectation maximization strategy.

Partially Observable Markov Decision Processes (POMDP)

In the POMDP case we have a dynamic system whose underlying evolution is modeled my a markov model. The state is only partially observable, and we can execute actions in order to drive the evolution of the system. Basically, this problem can be considered as a union between MDP and HMM.



A formal definition of this model is that $\text{POMDP} =< X, A, Z, \delta, r, O >$, where:

- X is the set of all possible states
- A is the set of all possible actions
- Z is the set of all possible observations
- $P(x_0)$ is the probability distribution over the initial state
- $\delta(x, a, x') = P(x'|x, a)$ is the probability distribution of the transition between states
- $r(x, a)$ is the reward function

- $O(x', a, z') = P(z', |x', a)$ is the probability distribution over the possible observations

What is the solution of this model?

Here we want to fung the best policy, namely the function that maps states to actions and maximizes the reward. Our problem is that in POMDP we don't know the state.

But the thing we know is the hystory of actions and observations, thus it makes sense to express the policy as $\pi : \{z_0, z_1, ..., z_{t-1}\} \rightarrow a$. Notice that this function is difficoult to represent, since it would require to store a very high number of observation.

A better approach is to introduce the concept of belief (estimation) of the state. This means that we make an estimate of the current state based on the previous observations, and update it at time t in order to choose the next action. We obtain the function $\pi : b_t \rightarrow a_t$.

Once we defined the belief function, then a POMDP basically becomed an MDP.