MACHINE LEARNING NOTES 2.0
GIULIANO ABRUZZO, GIULIO PORZI

Table of contents

Cap 1: Intro

Machine learning is an application of AI that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. The goal is to improve performance over time, based on experience. ML is a tool for generating knowledge from data.

- **Data**: is formed by bit of information
- **Knowledge**: is a semantic representation of these data

Model approach is based on modeling a problem and finding a math formulation, instead *ML will not require any model*: it just uses data and information extracted from the system and then build a solution from these. This approach in many cases is more effective and improves the performance.

In this period, we have an exploit of ML. The huge availability of *data* and the increasing of the *computational power* made possible to use specific *algorithms* to find solutions for several problems that were not effective when they were theorized. Data and computational power are not enough by themselves; we must understand how to design a system.

***Learning**: Improving with experience at some task*.

- Improve over task **T**, that is the *task* that we want to learn.
- With respect to performance measure **P**, that is the performance metrics.
- Based on experience **E**, these are the data user for experience.

If you do not specify these 3 elements the learning problems is not well defined, and the solution as well. In a learning problem we must define:

- How to collect experience → **target function**.
- The *structure* of the problem (what should be learned).
- The type of *data structure* we are going to use.
- The *algorithm* to use.

Collecting the experience is the most difficult part; in many cases the data are already available. The function sometimes is quite easy to define. The first thing is to choose the function that you want to learn, then you choose a representation of the function and depending on the other choices you will choose an algorithm and get the solution.

A **generic Machine Learning** (ML) **problem** is learning a function $f : X \rightarrow Y$, given a dataset D containing sampled information about f (D is a set of samples of this function*)*. Learning a function f means computing an approximated function f' that returns values as close to f also on samples x not in the dataset.

What is different in the ML problems categories is how data is represented.

There are 3 main categories for ML problems:

1. **Supervised Learning** $D \subset \{(x, y) | x \in X, y \in Y\}$
   - *Classification*
   - *Regression*
2. **Unsupervised Learning** $D \subset \{x | x \in X\}$
3. **Reinforcement Learning** $D = \{(a_i^1 \ldots a_i^n, r_i | i \in 1 \ldots |S|\}$

Supervised Learning is a set of problems in which the *dataset* is formed by a set of pairs input-output. If I take an element from *X*, I know the corresponding value in *Y*. *X* can be *continuous* or *discrete*, also *Y*. According to the different kind of set we have different names.

Cases in which the aim is to assign each input vector to one of a finite number of discrete categories (finite codomain), are called *classification* problems. The task is called *regression* if the desired output consists of one or more continuous variables (infinite codomain).

$D \subset \{(x, y) | x \in X, y \in Y\}$

- X finite sets: Discrete.
- $X \subset R^n$: Continuos.
- Y finite sets: **Classification**.
- $Y \subset R^k$: **Regression**.

Classification or Pattern Recognition return the class to which a specific instance belongs (given the input, you want to classify it in a predefined class), while Regression tries to approximate real-valued functions.

In the **Unsupervised Learning** we have only the *input* values and we don't know nothing about the corresponding value in the *codomain*. From the input you can still extract knowledge; in fact, you can make clusters and understand if they are significant or we can estimate density and other parameters. Very often this analysis is done in combination with the classification problem.

$$D \subset \{x | x \in X\}$$

The **Reinforcement Learning** is applied to dynamic systems, in which we want to learn a function where the input domain is a set of *all possible states S* with *associated actions* and *rewards* and the learned function is a *transition policy*. The data is a sequence of actions and states and then we have a value that say how good is that "episode". In the typical case you have something that changes over time and you want your agent to be able to learn from it, so the choice is about the right thing to do according to the current situation.

$$D = \{(a_i^1 \dots a_i^n, r_i | i \in 1 \dots |S|\}$$

We also have a special case called **Concept Learning** in which we have a finite input domain and the output is made only by two elements (like Booleans, 0 or 1).

We call c a *target function,* where $c: X \rightarrow Y$ is the function that we want to learn, X is an *instance space* (input set: all the possible inputs of the function) and where $x \in X$ is a *particular instance* in the set. The *dataset* D = {(x_i, c(x_i))} is a set of pairs in which for each instance x_i we have the corresponding value of target function c(x_i) that can be computed only for instances of the dataset (x is in D).

We call **hypothesis space** H the set of all possible functions that I can compute, where $h \in H$ a **hypothesis** and represent an approximation of the target function. We call $h(x)$ an **estimation** of h over x (predicted value) and we want that $h(x)$ is as similar as possible to $c(x)$, especially for values not in dataset.

The real goal is to find the **best hypothesis** that approximates the function c for elements that are **outside** the dataset.

We say that a hypothesis h is **consistent** with the train set D if $c(x) = h(x)\ \forall x \in D$.

Given a training set $D = \{(x_i, c(x_i))\}$ and a hypothesis $h \in H$, a performance measure is based on evaluating $c(x_i) = h(x_i)$ for every $x_i \in D$.

Inductive learning hypothesis: if a hypothesis does well inside the dataset, it will do well also outside the dataset.

The dataset must be representative for the problem that we are considering (large dataset).

We call **Version Space** the subset of H that is **consistent** with the dataset D. Each of these hypotheses is mapped to a subset of X that is also consistent with D, and this means that h(x) = the values of X.

$$VS_{H,D} \coloneqq \{h \in H | h \text{ } is \text{ } consistent \text{ } with \text{ } D\}$$

We call *list-then-eliminate algorithm* an algorithm that compute the version space by iterating on all the hypothesis and checking for any if they are consistent. Most of the time it is impracticable (we can have infinite hypothesis).

If any subset of the instances can be represented in a Version Space and all the solution are computed (search is completed), then the system is not able to classify new instances. In fact, different hypotheses in a version space may return different values for a new instance.

Collecting data is difficult and it’s easy to make some mistakes. **Noisy data** can be into the dataset and so the output is different from the true value of the target function. In this case we can have *no consistent hypothesis.* We then need **statistical methods** to remove the noise.

Cap 2: Classification Evaluation

To introduce the notion of **performance metrics** in ML, we must define a *probability distribution* of all the possible inputs. Let Z be the probability distribution over X, and S are a set of n instances of X sampled with Z. The performance evaluations are based on **accuracy** and **error rate**.

We can have two definitions of error/accuracy (remembering that $accuracy(h) = 1 - error(h)$):

- The **true error** of hypothesis h with respect to target function c and distribution Z is the <u>probability</u> that h will misclassify an instance according to Z.

  The true error represents the probability of making a mistake given an input taken from the *entire distribution*, and this is the error that the system will make at run time. The problem is that this error cannot be computed because we don't know c(x).

  $$error_Z(h) = p_{x \in Z}[c(x) \neq h(x)]$$

- The **sample error** of hypothesis h with respect to target function c and data sample S is the number of mistakes that h makes onS.

  The sample error is the number of mistakes that h makes on a data sample. It can be computed because for the samples of S we know the values of the target function. The problem is that this is not enough because we cannot be sure that this will be the "resulting error" of the entire distribution.

  $$error_S(h) = \frac{|\{x \in S | c(x) \neq h(x)\}|}{|S|}$$

The true error can be just estimated while sample error can be computed. We need to remember that the goal of a learning system is to be accurate in predicting outside the dataset: $h(x), \forall x \notin S$

We call **bias** the difference between the *expected value of the sample error* and the *true error*. We want to obtain a bias of 0, with this value in fact we can approximate the *true error* with the *expected value* of the *sample error*.

For **unbiased estimate** we must have a h and a S chosen independently: $E[error_S(h)] = error_Z(h)$

In order to calculate the *true error,* we need to calculate the **expected value** of the *sample error* by using an **unbiased estimator**. This is done as follow:

- Split the dataset D and train on T:

  $D = T \cup S, \qquad T \cap S = \emptyset, \qquad (|T| = \frac{2}{3}|D|)$

- Compute a hypothesis h using training set T.
- Evaluate the *sample error* to obtain an *unbiased estimator* for the *true error*.

With some probability the *true error* is in an interval defined by the *sample error*, where these 2 values are defined by a coefficient z_N.

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

When we are comparing two different hypotheses on a sample S, we have no guarantee that if a hypothesis is better than the other in S, the same will be also for the entire distribution. In the overfitting we have that the *sample error* of h is lower than the sample error of h' but the *true error* is greater.

We say that a hypothesis h **overfits** the training data if exists an h' such that:

$$error_s(h) < error_s(h') \wedge error_z(h) > error_z(h')$$

K-Fold method: we can evaluate a hypothesis h by estimating the error on different samples S. We *partition* the dataset in **k sets** (1), for all the *k sets* (2) we train the hypothesis with the **training set** (3) and we **compute** the *sample error* (4). Once I have done this k times, I just compute the **average** of all the errors (5) and I get the best approximation of the *true error*.

If k is too small the average will not be good (in general k=30 is good enough).

```
1) $D = S_1 \cup S_2 \cup \ldots \cup S_k$
2) $for\ i = 1\ to\ k$
3) $train\ h_i\ on\ D \setminus S_i$
4) $\delta = \delta + error_{S_i}(h_i)$
5) $error_{k,D} = \frac{\delta}{k}$
```

Accuracy is not a valid metric in some cases. For example, in a binary classification problem with a dataset D with 98% of *true* and 2% of *false*, a dumb hypothesis that returns always *true* has a *very high* accuracy.

	Predicted class	
True Class	Yes	No
Yes	TP: True Positive	FN: False Negative
No	FP: False Positive	TN: True Negative

True positive: instances correctly classified as positive.

True negative: instances correctly classified as negative.

False positive: samples that were negative but classified as positive.

False negative: samples that were positive but classified as negative.

From these values derives other *performance metrics*:

- Accuracy: $(TP + TN)/(TP + FN + TN + FP)$
- Precision: $TP/(TP + FP)$
- Recall: $TP/(TP + FN)$
- False positives rate: $FP/(TN + FP)$
- False negatives rate: $FN/(FN + TP)$
- F-Measure: $2 \cdot Precision \cdot Recall/(Precision + Recall)$

The ROC curve is the plot of (FP, TP) varying some parameters of the algorithm. Another useful value is the ROC Area.

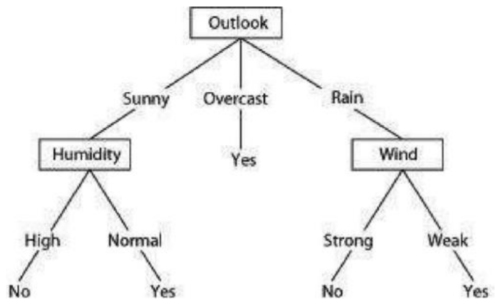
A **confusion matrix** represents in each entry how many instances of class C_i are misclassified as elements of class C_j. The main diagonal contains accuracy for each class.

Cap 3: Decision Tree

Given a *training set* D for a *target function* c, we need to compute a **consistent** hypothesis respecting D. Resolving approach: define an hypotheses space H and implement an algorithm to search an $h \in H$ that is consistent with D.

We are going to consider an hypotheses space made by the set of **decision tree**. This means that every hypothesis is a decision tree. Each decision tree is formed in this way:

- each **internal node** represents an attribute.
- each **branch** (arc towards an internal node) represents a value of an attribute.
- each **leaf** assigns a classification value.



A decision tree can be transformed in a **set of rules**. It represents the disjunction of conjunction of all the paths to the positive leaf nodes. We have an **OR** for each *edge* from the root, and we have an **AND** for each *internal node* in the path.

If we have for example the decision tree in figure, we obtain these rules:

$$(Outlook = Sunny \wedge Humidity = Normal) \vee (Outlook = Overcast) \vee (Outlook = Rain \wedge Wind = Weak)$$

If we want to create a decision tree from a dataset D, we must choose a target attribute.

We will use the **ID3 algorithm** (Iterative Dichotomiser 3).

This algorithm takes in input:

- Examples: data instances.
- Target attribute: what we want to learn.
- Attributes: the set that form the domain of the function.

The tree is built in a recursive way. In the first step it creates the Root node for the tree. Then we can have 3 different steps (all final cases) and a 4th recursive. The firsts two steps check if every element in the subset belongs to the same class, in which case the node is turned into a leaf node and labeled with the class of the examples.

- If all *Examples* are positive (+ , true , 1 , …), then return the node *Root* with a label **+** . This means that we will generate a tree that contains only *one leaf node* that is labelled with **+**.
- If all *Examples* are negative (- , false , 0 , …) then return the node *Root* with a label **-** . This means that we will generate a tree that contains only *one leaf node* that is labeled with **-**.
- If Attributes is **empty**, then return the node *Root* with *a label = most common value of Target attribute* in Examples. This will happen because at some point the recursive will reduce the set of attributes. If it is empty, we are going to create a leaf. This is an important feature of decision tree because it is using statistic, and this happens when we have a noisy dataset.
- **Recursive case**: first, we choose the "**best**" attribute in *Examples* and we generate all branches associated. We then take the subset of *Examples* that contains the branches of the best attribute. If this subset is empty like before we generate a leaf node with a label = most common value of *Target attribute* in *Examples.* Else we call the *recursive* algorithm by passing it, the new subset of *Examples* and the list of the *Attributes* without the *best attribute* used.

If the dataset is **consistent**, the choice based on *most common value* will never happen, in fact there will never be any conflict. This algorithm will produce *consistent data set* whatever is the *choice* of the attribute.

The algorithm can be written as follow:

```
algorithm $ID3(examples, target\_attribute, attributes)$
    $root :=$ new $Node()$
    if $label(e)$ is $+ \ \forall e \in examples$
        $label(root) = +$
        return $root$
    if $label(e)$ is $- \ \forall e \in examples$
        $label(root) = -$
        return $root$
    if $attributes$ is $\emptyset$
        $label(root) =$ most common value of $target\_attribute$ in
            $\hookrightarrow examples$
        return $root$
    $A := best\_decision\_attribute(examples)$
    $label(root) = A$
    foreach $v \in A$
        $branch := add\_branch(root, test(A = v))$
        $E_v := \{e \in examples | e_A = v\}$
        if $E_v$ is $\emptyset$
            $leaf :=$ new $Node()$
            $label(leaf) =$ most common value of $target\_attribute$ in
                $\hookrightarrow examples$
            $add\_node(root, branch, leaf)$
        else
            $add\_subtree(root, branch, ID3(E_v, target\_attribute, attributes \setminus \{A\}))$
```

ID3 does not guarantee an optimal solution. It can converge upon local optima. It uses a greedy strategy by selecting the locally best attribute to split the dataset on each iteration. The tree is built top-down, splitting the problem in smaller parts ("*dividi et impera*" approach).

ID3 considers as **best attribute** the one with the highest **information gain**, which measures how well a given attribute separates the training *examples* according to their target classification.

Let's call p_+ the proportion of positive examples in S and p_- the proportion of negative examples in S.

$$p_+ = \frac{|positive\ examples|}{|examples|} \qquad p_- = 1 - p_+$$

Entropy is defined as:

$$entropy(examples) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

Entropy is max when the dataset contains exactly half positive and half negative ($p_+ = p_- = 0{,}5$), minimum when $p_+ = 0$ or 1.

The information gain is defined as the expected reduction in *entropy* of the examples caused by knowing the value of *attribute* A (reduction of the entropy if we use A to partition the examples/S).

One way to define the *best attribute* for this step it to **compute** the *gain* of each attribute with this method and select the one with the *maximum gain.*

$$gain(examples) = entropy(examples) - \sum_{v \in A} \frac{|E_v|}{|examples|} entropy(E_v)$$

At the end, the ID3:

- Return a hypothesis space that is **complete**.
- In output we have **only one** hypothesis consistent with the dataset (while we cannot determine how many DTs are consistent!).
- **No backtracking** because once a decision is taken, there is no way to change it. This may limit the performance cause in some case you can find out that later this is not a good choice and you cannot go back (stuck on a local minima).
- It's **robust to noisy data** because it uses a *statically based search* choice.
- Uses all training examples at each step. Adding a single sample could change the *best attribute* and therefore also the whole *tree* (it is not incremental!).

Now a question: is it good that the tree *grows* over time to "*accommodate*" all possible instances?

The answer is no, because it can produce **overfitting:** <u>even with a better accuracy computed on the dataset used for training we may have that the accuracy on test dataset decrease</u>.

In order to avoid the overfitting, we can:

- **stop *growing*** the tree when data is not *splitting* in a significant way
- **post-prune**: build the *complete tree* and then **cutting** some part of the tree to reduce his size.

In order to decide which part of the tree to cut, I must estimate if cutting will improve the *accuracy* of the classification, using samples not used for training (**cross validation**...). Before I start the process, I partition the dataset in *training* and *validation set* (the second one used for evaluating the accuracy of the tree). This is called **Reduced-Error Pruning**, in which we replace *subtrees* with the more common values in order to improve *accuracy* on the *validation set*.

If Attributes have *continuous* values, we can use intervals in order to have discrete attributes. Instead, if we have attributes with *many values* or with a *cost,* we can use different performance metrics (not gain) but still based on the reduction of the entropy. If we have *missing values* of an attribute for some samples, there are *s*tatistical methods to generate a solution without losing the information.

There are other methods based on decision trees. **Random Forest** is a method that generates a set of decision trees using *random criteria (*random picking attributes, random subsets of data...) and integrates their values into a final value. The result of the Random Forest is the most common classification of all the trees (*majority vote*). Random Forests are less sensitive to overfitting.

Cap 4: Probability

Sample space

- Ω sample space (set of possibilities)
- $\omega \in \Omega$ is a *sample point/possible world/atomic event/outcome of a random process…*

Probability space (or *probability model*)

- Function $P: \Omega \rightarrow \Re$, such that
  - $0 \leq P(\omega) \leq 1$
  - $\sum_{\omega \rightarrow \Omega} P(\omega) = 1$

An *event* A is any subset of Ω. Probability of an event A is a function assigning to A a value in $[0,1]$:

$$p(A) = \sum_{\omega \in A} p(\omega)$$

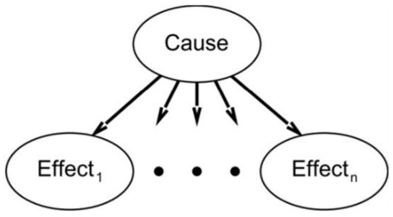
Prior probability is the probability of an event without knowing any other information (e.g. $P(Odd = true) = 0.5$). A *random variable* (outcome of a random phenomenon) is a function from the sample space Ω to some range (e.g., the reals or Booleans) $X: \Omega \rightarrow \Re$.

A **probability distribution** is the set of probability values for all the possible assignment of a random variable (the sum of all values must be 1). For continuous variables, the probability distribution is a continuous function.

Joint probability distribution for a set of random variables gives the probability of every atomic joint event on those random variables.

Conditional probability $P(a|b)$ is the probability of an event given another event happened.

- Conditional probability: $P(a|b) = P(a \wedge b)/P(b)$ if $P(b) \neq 0$.
- Product rule: $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$.
- Sum rule: $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$.
- Total probability: $P(a) = P(a|b)P(b) + P(a|\neg b)P(\neg b)$ and in general $P(X) = \sum_{y \in values(Y)} P(X|Y = y)\, P(Y = y)$ when y are mutually exclusive.
- Independence: X is independent from Y given Z if $P(X, Y|Z) = P(X|Z)$
- Bayes rule $P(a|b) = P(b|a)P(a)/P(b)$



Given a set of *random variables* we can generate a *graph* in which random variables are nodes while the edges represent *dependencies*. This graph is called Bayesian Network.

We can interpret a **classification** problem as a **probabilistic estimation**.

Given a target function $f: X \rightarrow V$, a dataset D and a new instance x', the best prediction $f'(x') = v^*$

$$v^* = \underset{y \in Y}{\operatorname{argmax}} P(y|x', D)$$

The function ***argmax*** returns the value of y for which the function P is maximum (while $maxf(y)$ returns the max value of $f(y)$). We compute the *full distribution* (over the whole codomain Y, $P(y|x', D)$ and we want to apply a decision over a classification result. The output of a probabilistic estimation is the prediction (of the class that we want to assign to a new instance) and the *probability* of this classification.

Cap 5: Bayes

Learning can be seen as a **probabilistic estimation**. In fact, given a dataset D, and a hypothesis space H, we can compute a *probability distribution* over H given D.

From the *Bayes rules* we have that:

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

- $P(h)$ represent the probability a priori (*prior probability*) of a hypothesis h before I get any dataset.
- $P(D)$ represent the prior probability of extracting this dataset from the entire distribution, independently from how the dataset is used in ML.
- $P(h|D)$ represent the probability that h has been generated from the dataset D.
- $P(D|h)$ represent the probability that, given one particular h, I generate the dataset D.

This is the main theorem we will use. We have a typical situation in which we define a hypothesis space, but now we don't want to compute only the best hypothesis, we want a probability distribution over the hypotheses; we want to *assign to each of this hypothesis a probability value* that say how good is the hypothesis.

In general, we want the most probable hypothesis given D, called **maximum a posteriori hypothesis**:

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D) = \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} = \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h)$$

It is defined as the argmax over all possible *hypothesis* in the *hypothesis space*, so it's the value of h for which the probability is maximized. We have eliminated the probability of extracting the database $P(D)$ because it doesn't depend on h.

If we assume that all the *hypotheses* have the same probability we can simplify and choose the **Maximum likelihood** (*ML*) hypothesis:

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} P(D|h)$$

We can consider also the log-likelihood: $h_{ML} = \underset{h \in H}{\operatorname{argmax}} P(D|h) = \underset{h \in H}{\operatorname{argmax}} \log\left(P(D|h)\right)$

We can differentiate it to get the max.

To generate a MAP hypothesis, we can iterate all over the *possible hypotheses*, and for each of this we can compute the *posteriori probability*. We then take the hypothesis h_{MAP} with the highest *posteriori probability.* The problem is that we can have infinite hypothesis. If the space is limited and small,it can be computed, and the learning process can be the **brute force** of all h ∈ H returning h_{MAP} after computing all posteriori probability.

If $h_{MAP}$is the most probable hypothesis given dataset D, given a new instance $x' \notin D$, $h_{MAP}(x')$ may not be the most probable classification. We need to take the weighted average, and then return as value the class for which you have the highest weighted average.

Here we need to introduce the **Bayes Optimal Classifier.**

For a target function $f: X \rightarrow Y$, dataset D and a **new instance** $x' \notin D$, we have that:

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

The value of the *classification* of an instance x', given a dataset D, is given by the *argmax* of the sum of all possible hypothesis:

$$y_{opt} = \underset{y \in Y}{\operatorname{argmax}} \sum_{h \in H} P(y|x', h)P(h|D)$$

There are proof that say no other ML using the same hypothesis space and same prior knowledge can outperform this method on average ("**optimal**"). This is very powerful and returns the *classification* with the *highest probability,* but it is not a practical method when hypothesis space is large.

We cannot use *Optimal Bayes* because we cannot compute it. I can't solve the real problem, so under some *assumptions* the real problem can be simplified and for this **simplified problem** I can find the solution. Whenever I apply some *assumption* to simplify the problem, we have an **approximated solution**. **Naive Bayes Classifier** uses **conditional independence** to approximate the solution.

X is *conditionally independent of* Y *given* Z if:

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

Let's assume a target function $f: X \rightarrow Y$ where each instance x is composed by attributes $(a_1, \dots, a_n)$. Our goal is to compute the **argmax** given a new instance x' and D, where x' is represented as a set of attributes. We obtain, for the ***Bayes rules***:

$$y_{MAP} = \underset{y \in Y}{\operatorname{argmax}} P(y|x', D) = \underset{y \in Y}{\operatorname{argmax}} P(y|a_1, \dots, a_n, D) = \underset{y \in Y}{\operatorname{argmax}} \frac{P(a_1, \dots, a_n|y, D)P(y|D)}{P(a_1, \dots, a_n|D)}$$

$$= \underset{y \in Y}{\operatorname{argmax}} P(a_1, \dots, a_n|y, D)P(y|D)$$

Optimal Bayes solve this problem by applying this to space of all possible hypothesis. We want to avoid this, so we do not use *total probability* or *heuristic space of hypothesis*.

We assume that all the *attributes* are **independent** each other given the dataset and classification value. This is a very strong assumption:

$$P(a_1, \dots, a_n|y, D) = \prod_i P(a_i|y, D)$$

We obtain:

$$y_{NB} = \underset{y \in Y}{\operatorname{argmax}} P(y|D) \prod_i P(a_i|y, D)$$

The value of the classification, using the Naïve Bayes Classifier, is given by the argmax over all the possible classification values of the probability $P(y|D)$ times the product of probabilities of every single values of each attribute of a new instance given y and D.

This formula can be easily *computed*, because there is no iteration or sum over all the possible hypothesis. Using this formula, all we need is just to estimate a very small number of probabilities, and D usually is very small. Also, the number of attributes of an instance are limited.

We can write a **Naive Bayes algorithm**, given a target function f with a *domain* X represented as *cartesian product* of n *attributes* and a *codomain* V that is a set of values, given a *dataset* D and a new instance x made by *attributes* $(a_1, \dots, a_n)$:

Naive Bayes Learn iterates over all classification values and estimate P, for each attribute and each value of the attribute estimates probability of this value given v_i and given D. There are 3 nested loops to estimate the probabilities used in the Bayes theorem.

Target function $f : X \mapsto V$, $X = A_1 \times \ldots \times A_n$, $V = \{v_1, ..., v_k\}$
data set D, new instance $x = \langle a_1, a_2 \ldots a_n \rangle$.

```
Naive_Bayes_Learn($A, V, D$)
    for each target value $v_j \in V$
        $\hat{P}(v_j|D) \leftarrow$ estimate $P(v_j|D)$
        for each attribute $A_k$
            for each attribute value $a_i \in A_k$
                $\hat{P}(a_i|v_j, D) \leftarrow$ estimate $P(a_i|v_j, D)$

Classify_New_Instance($x$)
    $v_{NB} = \underset{v_j \in V}{\text{argmax}}\, \hat{P}(v_j|D) \prod_{a_i \in x} \hat{P}(a_i|v_j, D)$
```

Usually a good way to make estimations is to consider the number of times in which the event happens divided by the total number of events. $P(v_j|D)$ can be estimate as numbers of elements that v_j as output divided by the total number of the elements in the dataset:

$$\hat{P}(v_j|D) = \frac{|\{\langle \ldots, v_j \rangle\}|}{|D|}$$

In the same way, the probability of a particular attribute given classification value and dataset is given by the number of samples for which the attribute happens divided the number of samples for that category in the dataset:

$$\hat{P}(a_i|v_j, D) = \frac{|\{\langle \ldots, a_i, \ldots, v_j \rangle\}|}{|\{\ldots, v_j\}|}$$

Note that if there is a *missing element* (if none of the training instances with target value v_j have an attribute a_i), we have $\hat{P}(a_i|v_j, D) = 0$ and $\hat{P}(v_j|D) \prod_i P(a_i|v_j, D) = 0$

This is not good. We can use a **prior estimate** (virtual samples) so there is always non-zero probability:

$$\hat{P}(a_i|v_j, D) = \frac{|\{\langle \ldots, a_i, \ldots, v_j \rangle\}| + m \cdot p}{|\{\ldots, v_j\}| + m}$$

Where:

- p is a prior estimate for $\hat{P}(a_i|v_j, D)$
- m is a weight given to prior (i.e. number of “virtual” examples)

For example, Bayes Classifiers can be used to **classify text**.

The vocabulary is the set of all the words appearing in any document of the data set (its dimension is n). A text is represented as a n-dimensional feature vector in the Bag of Words representation (but we loose the information about the order of the words). Different options are available for representing features:

- Boolean features: 1 if word appears in the text, 0 otherwise (multivariate Bernoulli distribution).
- Ordinal feature: number of occurrences of the words in the text (multinomial distribution).

Cap 6: Probabilistic Classification

In a probabilistic classification problem, we want to estimate the *posterior probability distribution* of an instance belonging to some class. Our task in classification problems is to *predict* the class to which a new instance belongs, and we can formulate this problem as an **estimation**.

There are two **probabilistic models** for classification:

- **Generative**: estimates $P(C_i|x)$ through $P(x|C_i)$ and Bayes theorem.
- **Discriminative**: estimates $P(C_i|x)$ directly from a model.

These methods are similar, in fact both are based on the idea of maximizing the *likelihood*.

Probabilistic Generative Models

The idea is to compute the probability of a class given an instance by **using *Bayes theorem***.

We turn to a probabilistic view of classification and show how models with linear decision boundaries arise from simple assumptions about the distribution of the data.

Consider first of all the case of two classes (but all the methods can be extended to *multiple classes)*, the **posterior probability** (conditional probability) for class C_1 can be written as:

$$P(C_1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|C_1)P(C_1)}{p(\boldsymbol{x}|C_1)P(C_1) + p(\boldsymbol{x}|C_2)P(C_2)} = \frac{1}{1+e^{-\alpha}} = \sigma(\alpha)$$

With $\alpha = \ln\frac{p(\boldsymbol{x}|C_1)P(C_1)}{p(\boldsymbol{x}|C_2)P(C_2)}$ and σ the **logistic sigmoid.**

Let us assume that the **class-conditional densities are Gaussian (Gaussian Naïve Bayes)** and then explore the resulting form for the posterior probabilities. To start with, we shall assume that all classes share the **same covariance matrix $\boldsymbol{\Sigma}$.**

$$\alpha = \ln\frac{p(\boldsymbol{x}|C_1)P(C_1)}{p(\boldsymbol{x}|C_2)P(C_2)} = \alpha = \ln\frac{\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_1;\boldsymbol{\Sigma})P(C_1)}{\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_2;\boldsymbol{\Sigma})P(C_2)} = \cdots = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + w_0$$

The quadratic terms in x from the exponents of the Gaussian densities have cancelled (due to the assumption of common covariance matrices), leading to a **linear function** of x in the argument of the **logistic sigmoid**.

$$P(C_1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + w_0)$$

We need to find the **maximum likelihood solution**. Our goal is then to find the parameters of this model (prior probabilities, means and covariance of the two gaussian distributions). We just compute likelihood and then we solve the optimization problem for finding max likelihood.

We denote the prior class probability $p(C_1) = \pi$, so that $p(C_2) = 1 - \pi$. For a data point x_n from class C_1, we have $t_n = 0$ and hence

$$p(x_n, C_1) = p(C_1)p(x_n|C_1) = \pi\,\mathcal{N}(x_n|\boldsymbol{\mu}_1;\boldsymbol{\Sigma})$$

Similarly, for class C_2, we have $t_n = 1$ and hence

$$p(x_n, C_2) = p(C_2)p(x_n|C_2) = (1-\pi)\,\mathcal{N}(x_n|\boldsymbol{\mu}_2;\boldsymbol{\Sigma})$$

Thus, the likelihood function is given by

$$p(\boldsymbol{t}|\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) = \prod_{n=1}^{N}[\pi\,\mathcal{N}(x_n|\boldsymbol{\mu}_1;\boldsymbol{\Sigma})]^{t_n}\,[(1-\pi)\,\mathcal{N}(x_n|\boldsymbol{\mu}_2;\boldsymbol{\Sigma})]^{1-t_n}$$

Maximizing the log of the likelihood, we get the needed parameters for computing the probability.

In the linear regression models, the model prediction $y(x, w)$ is given by a linear function of the parameters $\boldsymbol{w}$. In the simplest case, the model is also linear in the input variables and therefore takes the form $y(x) = \boldsymbol{w^T x} + w_0$, so that y is a real number. For **classification** problems, however, we wish to predict discrete class labels, or more generally posterior probabilities that lie in the range $(0,1)$. To achieve this, we consider a generalization of this model in which we transform the linear function of w using a nonlinear function $f(\cdot)$, known as **activation function**, so that $y(x) = f(\boldsymbol{w^T x} + w_0)$

For the case of $K > 2$ classes, we have:

$$P(C_k|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|C_k)P(C_k)}{\sum_j p(\boldsymbol{x}|C_j)P(C_j)}$$

Which is known as the *normalized exponential* and can be regarded as a multiclass generalization of the logistic sigmoid. The normalized exponential is also known as the *softmax function*, as it represents a smoothed version of the 'max' function.

Discriminative

Discriminative models estimate $P(C_i|x)$ modelling them directly, for example by representing them as parametric models and then optimizing the parameters using a training set. One advantage of the discriminative approach is that there will typically be fewer adaptive parameters to be determined. It may also lead to improved predictive performance, particularly when the class-conditional density assumptions give a poor approximation to the true distributions.

For the two-class classification problem, we have seen that the posterior probability of class C_1 can be written as a logistic sigmoid acting on a linear function of $\boldsymbol{x}$, for a wide choice of class-conditional distributions $p(\boldsymbol{x}|C_k)$ $(\text{not only the Gaussian})$. Similarly, for the multiclass case, the posterior probability of class C_k is given by a softmax transformation of a linear function of $\boldsymbol{x}$. For specific choices of the class-conditional densities $p(\boldsymbol{x}|C_k)$, we have used maximum likelihood to determine the parameters of the densities as well as the class priors $p(C_k)$ and then used Bayes' theorem to find the posterior class probabilities. However, an alternative approach is to use the functional form of the generalized linear model explicitly and to determine its parameters directly by using maximum likelihood. We will estimate posterior probability directly without Bayes' theorem.

We consider a dataset in a transformed space, the likelihood function is the probability of receiving the output t given the input n, and the model is given by the **sigmoid function of a linear combination** of the input and the weights.

In the case of the linear regression models, the maximum likelihood solution, on the assumption of a Gaussian noise model, leads to a closed-form solution (see Least Squares). This is a consequence of the quadratic dependence of the log likelihood function on the parameter vector w. For **logistic regression**, there is no longer a closed-form solution, due to the nonlinearity of the logistic sigmoid function. The **Iterative reweighted least squares**, a modified version of least squares which implements an iteration of the Newton-Raphson algorithm to minimize the **cross-entropy error**, can be applied.

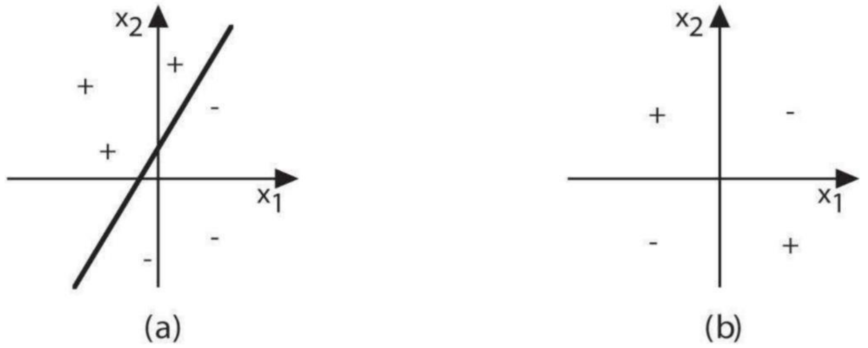
Cap 7: Linear Classification

The goal in classification is to take an input vector $\boldsymbol{x}$ and to assign it to one of K discrete classes C_k where $k = 1, \dots, K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into decision regions whose boundaries are called decision boundaries or decision surfaces. In this chapter, we consider linear models for classification, by which we mean that the *decision surfaces* are linear functions of the input vector x and hence are defined by (D −1)-dimensional hyperplanes within the D-dimensional input space. Data sets whose classes can be separated exactly by linear decision surfaces are said to be linearly separable.

We have a *classification problem* with a **continuous** input space (while in the previous examples the input was formed by finite set of attributes). Output of our *target* is a *set of classes*. This is a classification problem where input are real numbers and we assume that data have a special format so they can be separated by a **linear function**.

We must learn a function $f: X \to Y$ with $X \subset R^d$ and $Y = \{C_1, \dots, C_k\}$ with linearly separable data.

Instances in a *dataset* (*binary*) are **linearly separable** if it exists a **linear function** (hyperplane) that divide the instance space into two regions, such that differently classified instances are separated.



a) Dataset instances are denoted with a symbol. In this case we a two-dimensions space as input and two classes as output $(+ , -)$. This dataset is linearly separable, in fact there exist at least one line that separates the space in 2 regions, one with only positive samples and the other with only negative ones. The line that separates data is used to predict classes of new instances.
b) Dataset is not linearly separable. It's not possible to find any *line* that divides the space in 2 regions, one with only $+$ and the other with only $-$.

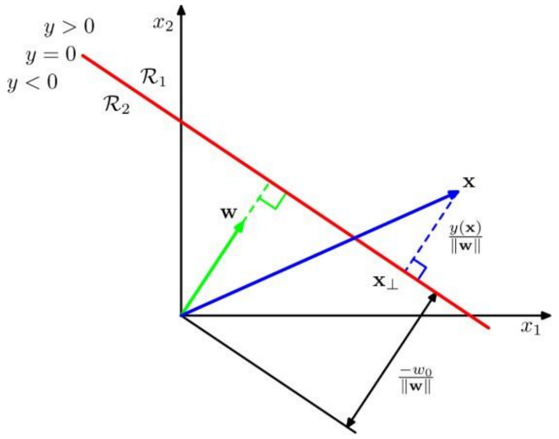
Now we are going to focus on learning a classification function under the assumption that the dataset is linearly separable. We call **Linear Discriminant function** $y: X \to \{C_1, \dots, C_k\}$:

- 2 classes: $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0$.
- K classes: $y_k(\boldsymbol{x}) = \boldsymbol{w}_k^T \boldsymbol{x} + w_{k_0}$, $k = 1, \dots, K$.

In the first case we have 2 classes, then the discriminant function can be given just by a *linear function*, where x is the input, w and w$_0$ are the *coefficient* of this function.

In the second case, we must classify more than 2 classes, so we need to define a set of functions. The model in this case is given by k-1 functions, where k is the *number of classes* of our classification problem.

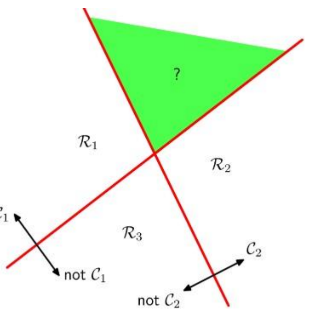
The geometrical interpretation of a **binary classification** problem is: the red line is the function and represents the set of point x_1,x_2 for which y=0, w is the vector perpendicular to the line, and the distance from any point x in the space and the line is: $y(x)/||w||$. The classification is made for $y(x) > 0$ or $y(x) < 0$.

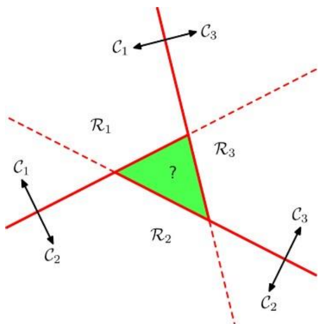


When we have **k-classes** we <u>can't use combination of *binary linear models*</u>.

The One-versus-the-rest classifier (Ovr) strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. This strategy requires the base classifiers to produce a real-valued confidence score for its decision, rather than just a class label; discrete class labels alone can lead to ambiguities, where multiple classes are predicted for a single sample. We have K-1 binary classifiers: C_K vs not-C_K.

In the *One-versus-one (OvO) classifier*, one trains *K* (*K* − 1) / 2 binary classifiers C_K vs C_j for a *K*-way multiclass problem; each receives the samples of a pair of classes from the original training set, and must learn to distinguish these two classes. At prediction time, a voting scheme is applied: all *K* (*K* − 1)/ 2 classifiers are applied to an unseen sample and the class that got the highest number of "+1" predictions gets predicted by the combined classifier.





Attempting to construct a K class discriminant from a set of two class discriminants leads to ambiguous regions, in green. On the left there is an example involving the use of two discriminants designed to distinguish points in class C_k from points not in class C_k (OvR classifier). On the right an example involving three discriminant functions used to separate a pair of classes C_k and C_i (OvO classifier).

For **multiple classes**, we need to define a **set of linear functions.** In order to classify a new instance x, we compute all these functions and we classify the instance with the function K that returns the highest value.

$y_k(\boldsymbol{x}) = \boldsymbol{w}_k^T\boldsymbol{x} + w_{k0}$ assigning $\boldsymbol{x}$ to C_k if $y_k(\boldsymbol{x}) > y_j(\boldsymbol{x})$ for all $j \neq k$

The output of the model is a partition of the space. Using a compact notation with vectors:

$y(\boldsymbol{x}) = \widetilde{\boldsymbol{W}}^{\boldsymbol{T}}\widetilde{\boldsymbol{x}}$ $\quad\quad$ $\widetilde{\boldsymbol{W}} = \begin{pmatrix} w_{1_0} & w_{2_0} & \cdots & w_{k_0} \\ w_1 & w_2 & \cdots & w_k \end{pmatrix}$ $\quad\quad$ $\widetilde{\boldsymbol{x}} = \begin{pmatrix} 1 \\ x \end{pmatrix}$

Given a multi-class classification problem and a dataset D with linearly separable data, we need to **determine W** such that $\boldsymbol{y}(\boldsymbol{x}) = \widetilde{\boldsymbol{W}}^{\boldsymbol{T}}\widetilde{\boldsymbol{x}}$ is the K-class discriminant.

We need to compute the coefficient of the matrix and then classify new instances applying this matrix multiplication and checking the result with the highest value.

There are 4 kinds of solution for this problem:

1. Least squares
2. Fisher's linear discriminant
3. Perceptron
4. Support Vector Machines

Let's consider a *dataset:* $D = \{(x_n, t_n)_{n=1}^{N}\}$

The dataset is a set of pairs, where x_n is a point in our space and x_n is a vector of k components where k is the number of classes and this vector contains one value with 1 and all the other with 0, so it is 1 at position i when $x_n \in C_i$.

Least squares

Our dataset contains n *tuples of values* of our input and n encoding, so we can represent this information in a matrix in which the number of rows is the size of the dataset, and for each row we have one sample encoded as $\tilde{x}_1^T$ and so on.

$$\tilde{X} = \begin{pmatrix} \tilde{x}_1^T \\ \dots \\ \tilde{x}_N^T \end{pmatrix} \quad T = \begin{pmatrix} t_1^T \\ \dots \\ t_N^T \end{pmatrix}$$

This matrix will have all 1 in the first column, in all the rows you have the values in the dataset, and this is called **sign matrix**. T is a matrix where in each row there is the encoding of the value of the *class* for each *sample*.

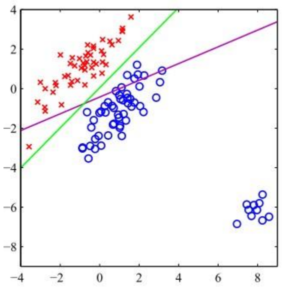
As our goal is to find the matrix W, we need to solve this as an optimization problem, so we need to define an **error function**. This function is defined in term of list square error, and the error is the difference between prediction of the sample in the data and the value that is in the dataset:

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

We want this error to be 0, and we want to find a value of W such that this error is minimum. This is a *simple quadratic problem* that can be solved with a *closed form solution, that is the following:*

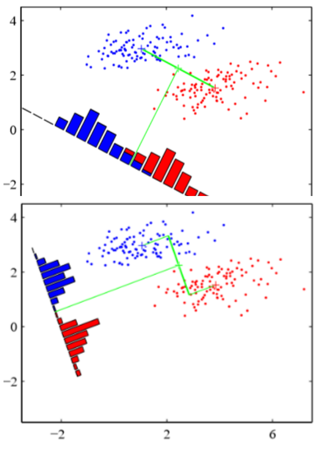
$$\tilde{\mathbf{W}} = \underbrace{(\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^T}_{\tilde{\mathbf{X}}^\dagger}\mathbf{T}$$

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



The problem is that this solution is not robust to **outliners**, because it finds the line that is the *best average distance* from each point.

Fisher's linear discriminant



Considering a two classes case, we must determine $y = w^T x$ in order to classify $x \in C_1$ if $y \geq -w_0$, $x \in C_2$ otherwise.

We want to adjust W in order to find a direction that maximizes class separation.

Consider a data set with N_1 points in C_1 and N_2 points in C_2, so that the **mean vectors** of the two classes are given by:

$$\boldsymbol{m}_1 = \frac{1}{N_1}\sum_{n\in C_1} \boldsymbol{x_n} \qquad \boldsymbol{m}_2 = \frac{1}{N_2}\sum_{n\in C_2} \boldsymbol{x_n}$$

The simplest measure of the separation of the classes, when projected onto w, is the separation of the projected class means. This suggests that we might choose w so as to maximize:

$$m_2 - m_1 = \boldsymbol{w^T}(\boldsymbol{m}_2 - \boldsymbol{m}_1)$$

Finding a line that minimizes the overlap of two projections means choosing $\boldsymbol{w}$ that maximizes $J(\boldsymbol{w}) = \boldsymbol{w^T}(\boldsymbol{m}_2 - \boldsymbol{m}_1)$, constraining $\boldsymbol{w}$ to have unit length $||\boldsymbol{w}|| = 1$ (or we could just increase the magnitude of w).

There is still a problem with this approach, as illustrated in the first figure. This shows two classes that are well separated in the original two-dimensional space (x_1, x_2) but that have considerable overlap when projected onto the line joining their means. This difficulty arises from the strongly nondiagonal covariances of the class distributions. The idea proposed by Fisher is to maximize a function that will give a large separation between the projected class means while also giving a small variance within each class, thereby minimizing the class overlap.

The projection formula $y = \boldsymbol{w^T x}$ transforms the set of labelled data points in x into a labelled set in the one-dimensional space y.

The *within-class variance* of the transformed data from class C_k is therefore given by:

$$s_k^2 = \sum_{n\in C_k} (y_n - m_k)^2$$

Where $y_n = \boldsymbol{w^T x_n}$ and $m_k = \boldsymbol{w^T m_k}$.

We can define the *total within-class variance* for the whole data set to be simply ${s_1}^2 + {s_2}^2$.

The Fisher criterion is defined to be the ratio of the *between-class variance* to the *within-class variance* and is given by:

$$J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}$$

Maximizing this error function, we get a solution based on maximum separation between projections of the means of the 2 distributions rotated by a proper rotation matrix.

We can make the dependence on w explicit to rewrite the Fisher criterion in the form:

$$J(\boldsymbol{w}) = \frac{\boldsymbol{w^T S_B w}}{\boldsymbol{w^T S_W w}}$$

Where $\boldsymbol{S_B}$ is the *between-class* covariance matrix and is given by:

$$\boldsymbol{S_B} = (\boldsymbol{m}_2 - \boldsymbol{m}_1)(\boldsymbol{m}_2 - \boldsymbol{m}_1)^{\boldsymbol{T}}$$

And $\boldsymbol{S_W}$ is the total *within-class* covariance matrix, given by:

$$\boldsymbol{S_W} = \sum_{n \in C_1} (\boldsymbol{x_n} - \boldsymbol{m}_1)(\boldsymbol{x_n} - \boldsymbol{m}_1)^{\boldsymbol{T}} + \sum_{n \in C_2} (\boldsymbol{x_n} - \boldsymbol{m}_2)(\boldsymbol{x_n} - \boldsymbol{m}_2)^{\boldsymbol{T}}$$

Differentiating $J(\boldsymbol{w})$ with respect to $\boldsymbol{w}$, we find that $J(\boldsymbol{w})$ is maximized when:

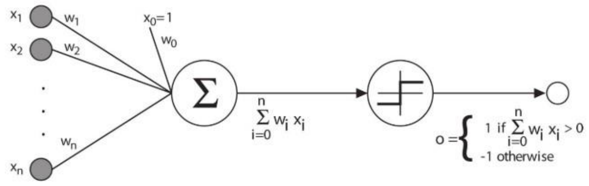
$$(\boldsymbol{w^T S_B w})\boldsymbol{S_w w} = (\boldsymbol{w^T S_w w})\boldsymbol{S_B w}$$

Considering that $m_2 - m_1 = \boldsymbol{w^T}(\boldsymbol{m}_2 - \boldsymbol{m}_1)$ is a scalar, the term $\boldsymbol{S_B w} = (\boldsymbol{m}_2 - \boldsymbol{m}_1)(\boldsymbol{m}_2 - \boldsymbol{m}_1)^{\boldsymbol{T}}\boldsymbol{w}$ on the right-hand side of the equation is always in the direction of (**m2−m1**). Furthermore, we do not care about the magnitude of $\boldsymbol{w}$, only its direction, and so we can drop the scalar factors $(\boldsymbol{w^T S_B w})$ and $(\boldsymbol{w^T S_w w})$. Multiplying both sides by $\boldsymbol{S_w^{-1}}$, we then obtain the solution:

$$\boldsymbol{w}^* \propto \boldsymbol{S_w^{-1}}(\boldsymbol{m}_2 - \boldsymbol{m}_1)$$

Perceptron

The perceptron is a simplified model of a biological **neuron**. While the complexity of biological neuron models is often required to fully understand neural behavior, research suggests a perceptron-like linear model can produce some behavior seen in real neurons.



x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

We have a linear combination of input dimensions with some weights, plus a constant value w_0.

In the modern sense, the perceptron is an algorithm for learning a binary classifier called a *threshold function*. The output will be a function that depends on the sign of the linear combination:

$o(\boldsymbol{x}) = \text{sign}(\boldsymbol{w^T x})$.

Considering the *unthresholded linear unit*: $o(\boldsymbol{x}) = w_0 + w_1 x_1 + \cdots + w_d x_d = \boldsymbol{w^T x}$.

Let's learn w_i from training examples $D = \{(x_n, t_n)_{n=1}^{N}\}$ that minimize the squared error (loss function). The derivate of the error function is done with respect to each w_i.

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{n=1}^{N}(t_n - \boldsymbol{w_n}^{\boldsymbol{T}}\boldsymbol{x_n})^2 = \frac{1}{2}\sum_{n=1}^{N}\frac{\partial}{\partial w_i}(t_n - \boldsymbol{w_n}^{\boldsymbol{T}}\boldsymbol{x_n})^2 = \sum_{n=1}^{N}(t_n - \boldsymbol{w_n}^{\boldsymbol{T}}\boldsymbol{x_n})(-x_{i,n})$$

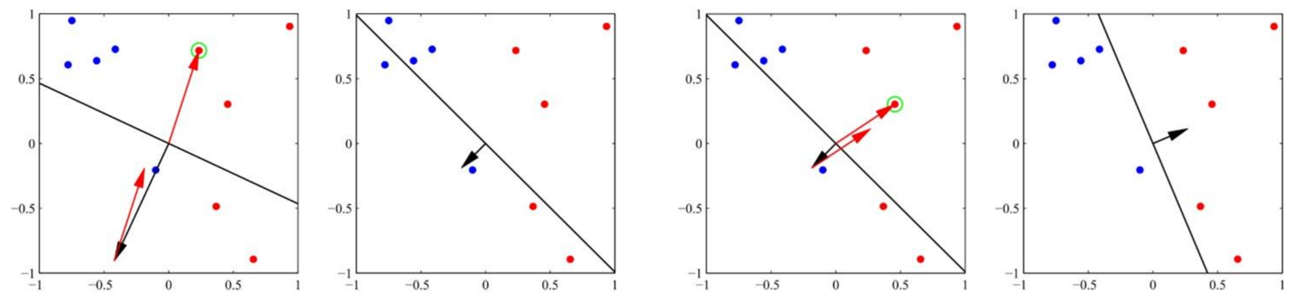
It is an **iterative method** (learn iteratively) with a gradient checking. The algorithm will just start to move to the **negative gradient** in order to minimize the error. The algorithm can update weights in order to adjust the classifier and this is done in an iterative way until we divide all the samples of the dataset.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_{(\boldsymbol{x},t) \epsilon D} (t_n - o(\boldsymbol{x}))\, x_i$$

Batch mode: Consider all dataset D

Incremental (choose one sample) and mini-batch modes (considers a small subset) speed up convergence and are less sensitive to local minima.

The perceptron **convergence** theorem states that if there exists an exact solution (in other words, if the training data set is linearly separable), then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.



The first plot shows the initial parameter vector w shown as a black arrow together with the corresponding decision boundary (black line), in which the arrow points towards the decision region which classified as belonging to the red class. The data point circled in green is misclassified and so its feature vector is added to the current weight vector, giving the new decision boundary shown in the second plot. The third plot shows the next misclassified point to be considered, indicated by the green circle, and its feature vector is again added to the weight vector giving the decision boundary shown in the fourth plot for which all data points are correctly classified.

Exam question

The perceptron is an iterative algorithm to reach the optimization problem and it works with the linear combination of the input dimensions with some weights plus a constant term w_0.

The algorithm starts with a random hyperplane and it tries to minimize the loss function (square error). This function is given by the difference between what I have in the dataset and the prediction that w will return on the input. To minimize the loss function, I have to derivate the error respect to each w_0, so I can compute the gradient and I can see that the algorithm will move in the direction of negative gradient to minimize the error. The weights will be updated each time in an iterative way until some termination condition.

Support Vector Machine

SVMs constructs a **maximum margin separator**, a decision boundary with the largest possible distance to example points. This helps them generalize well.

The maximum margin separator is expressed as a linear discriminant function, which for a two-class classification problem with a fixed feature-space transformation $\boldsymbol{\varphi}(\boldsymbol{x})$ is:

$$y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}) + b$$

The perpendicular **distance** of a point $\boldsymbol{x}$ from a hyperplane defined by $y(\boldsymbol{x}) = 0$ is given by

$$|y(\boldsymbol{x})|/||\boldsymbol{w}||$$

Furthermore, we are only interested in solutions for which all data points are correctly classified, so that $t_n y(\boldsymbol{x_n}) > 0$ for all n (which leads to $|y(\boldsymbol{x})| = t_n y(\boldsymbol{x_n})$).

The margin is given by the perpendicular distance of a point $\boldsymbol{x_n}$ from the data set, and we wish to optimize $\boldsymbol{w}$ and b in order to maximize this distance. Thus, the maximum margin solution is found by solving:

$$\boldsymbol{w}^*, b^* = \underset{w,b}{\operatorname{argmax}} \left\{ \frac{1}{||\boldsymbol{w}||} \min_n [t_n(\boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}) + b)] \right\}$$

We **rescale** all these points by multiply all points by a constant term, not affecting the solution, in such a way that the closest point $\boldsymbol{x_k}$ has a distance from the optimal hyperplane of 1 (*canonical* *representation*)

$$t_n(\boldsymbol{w}^T \boldsymbol{\varphi}(\boldsymbol{x_k}) + b) = 1$$

When the maxim margin hyperplane $\boldsymbol{w}^*, b^*$ is found, there will be at least two closest points, one for each class. In the canonical representation of the problem the maxim margin hyperplane can be found by solving the optimization problem:

$$\max \frac{1}{||\boldsymbol{w}||} = \min \frac{1}{2} ||\boldsymbol{w}||^2$$

subject to: $t_n(\boldsymbol{w}^T \boldsymbol{\varphi}(\boldsymbol{x_k}) + b) > 1 \; \forall n = 1, \dots, N$

It's a quadratic programming problem solved with **Lagrangian method** and the solution is expressed in terms of Lagrange multipliers.

We obtain the following two conditions:

$$\boldsymbol{w} = \sum_{n=1}^{N} a_n t_n \varphi(\boldsymbol{x_k})$$

$$0 = \sum_{n=1}^{N} a_n t_n$$

A constrained optimization of this form satisfies the *Karesh-Kuhn-Tucker* (KKT) conditions. Any data points for which $a_n = 0$ will not appear in the sum in the classification function, and hence plays no role in making predictions for new data points. The remaining data points are called *support vectors* and they correspond to data points that lie on the maximum margin hyperplanes in feature space.

This property is central to the practical applicability of SVMs since, once the model is trained, a significant proportion of the data points can be discarded and only the support vectors retained.

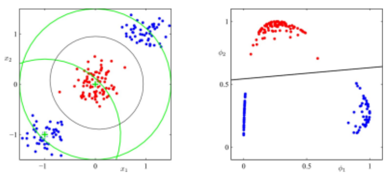
SVMs can be extended to soft margin constraints, introducing slack variables $\xi_n \geq 0$, one for each training data point:

- $\xi_n = 0$ if the point is on or inside the correct margin boundary.
- $0 < \xi_n \leq 1$ if the point is inside the margin on the correct side.
- $\xi_n \geq 1$ if the point is on the wrong side of the boundary (misclassified).

Our goal is now to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary. We therefore minimize:

$$\min \left[\frac{1}{2} ||\boldsymbol{w}||^2 + C \sum_{n=1}^{N} \xi_n \right]$$

Exam question



If the dataset is **not perfectly separable,** we can apply a **non-linear transformation** of the inputs, we apply a basis functions $\phi(x)$ and with this function we obtain a dataset that is separable in the feature space ϕ.

Cap 8: Linear Regression

Now we consider the problem of learning a function which **output** is **continuous**. The *output domain* of our *target function* is given by the set of *real numbers*, or a subset of it.

We are going to learn a function $f: X \to Y$ where $X \subseteq R^d$ and $Y = R$ from dataset $D = \{(x_n, t_n)_{n=1}^N\}$.

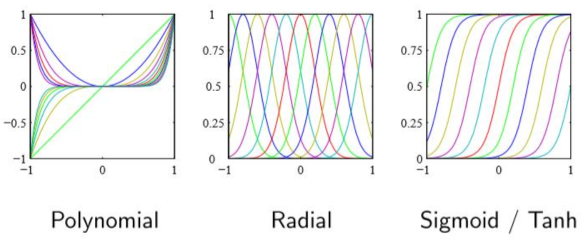
We have a dataset that is a *set of pairs,* where t_n is a real value. We want to find a *function* such that returns a real value to an input x (*not a class already defined* as we have seen before). We need to define a *model* $y(x; w)$ with parameters w to approximate the target function f. The linear model for linear function is:

$$y(x; w) = w_0 + w_1 x_1 + \cdots + w_n x_n = w^T x$$

We can use the linear **basis function** model to generate linear models of **non-linear functions**. We can generate a new model, as the linear combination of the coefficients w times the values that are obtained by transforming input space by any non-linear function $\phi(x)$:

$$y(x; w) = w^T \phi(x)$$

Now we have a non-linear function in x, but we still have a linear function in w. Since we are interested in computing w, this is still a linear model and we can apply all the methods we saw for linear models, in fact the non-linear function in x *does not affect the solution*.



To *minimize the error* in our model, we will *maximize* the likelihood.

It can be demonstrated that the condition of maximum likelihood corresponds, under some assumptions, to least square error minimization. Assumptions:

- The target variable t is given by a deterministic function $y(x, w)$ with additive Gaussian noise (*error model*) so that $t = y(x, w) + \varepsilon$
- Observations are independent and identically distributed (i.i.d).

$$\min E_D(\boldsymbol{w}) = \min \frac{1}{2} \sum_{n=1}^{N} [t_n - \boldsymbol{w}^T \phi(\boldsymbol{x}_n)]^2$$

Batch techniques, such as the maximum likelihood solution, which involve processing the entire training set in one go, can be computationally costly for large datasets. If the dataset is sufficiently large or in real time applications, it may be worthwhile to use **sequential algorithms**, also known as on-line algorithms, in which the data points are considered one at a time, and the model parameters updated after each such presentation.

We can obtain a sequential learning algorithm by applying the technique of stochastic gradient descent, also known as sequential gradient descent, as follows. If the error function comprises a sum over data points $E = \sum_n E_n$, then after presentation of pattern n, the stochastic gradient descent algorithm updates the parameter vector w using $w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n$

where τ denotes the iteration number, and η is a learning rate parameter.

For the case of the sum-of-squares error function, this gives:

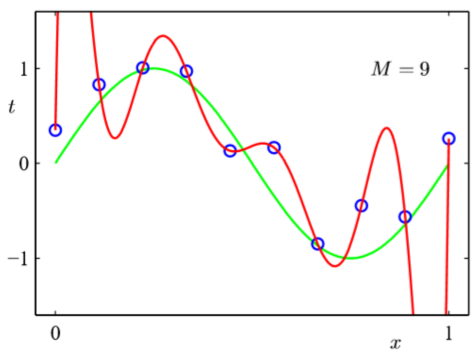
$$w^{(\tau+1)} = w^{(\tau)} + \eta \left(t_n - w^{(\tau)^T} \varphi_n\right) \varphi_n$$

Regularization is a technique to control over-fitting.

$$\min E_D(w) + \lambda E_W(w)$$

Where $\lambda > 0$ is the regularization factor

A common choice is to use: $E_W(w) = \frac{1}{2} w^T w$



Exam question

The parameters can be estimated in batch mode and in sequential mode.

The difference is that in batch mode we take the whole dataset in order to minimize a squared error function, instead in the sequential one we update the parameters with an SGD algorithm, similar to the *perceptron* one, but with continuous values as t_n.

Cap 9: Kernels methods

So far, the object where represented as fixed-length feature-vector $X \in R^m$ or $\phi(x)$. How can I deal with objects with variable length or infinite dimension?

Kernel methods are useful to introduce *non-linear function* in linear models.

The 'kernel approach' offers an alternative solution to increase the computational power of linear learning machines by mapping data into a high dimensional feature space.

A kernel function is a real-valued function that maps a pair of instances into real numbers.

$k(\boldsymbol{x}, \boldsymbol{x}') \in \mathbb{R}$, for $\boldsymbol{x}, \boldsymbol{x}' \in X$ where X is some abstract space.

Typically, k is symmetric $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$ and non-negative $k(\boldsymbol{x}, \boldsymbol{x}') \geq 0$

- **Linear** $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{x}'$
- **Polynomial** $k(\boldsymbol{x}, \boldsymbol{x}') = (\beta \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{x}' + \gamma)^d$
- **Radial Basis Function (RBF)** $k(\boldsymbol{x}, \boldsymbol{x}') = e^{-\beta|x-x'|^2)}$
- **Sigmoid** $k(\boldsymbol{x}, \boldsymbol{x}') = \tanh(\beta \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{x}' + \gamma)$

For any linear model with a kernel k: $y(x, \widehat{w}) = \sum_{i=1}^{N} \alpha_i k(\boldsymbol{x}, \boldsymbol{x}')$

The solution is: $\alpha = (K + \lambda I_N)^{-1} t$

where $K = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix}$ is the **gram matrix**.

This solution is general for any kernel. We obtain a linear model of a non-linear function.

This is known as a *dual formulation* because, by noting that the solution for a can be expressed as a linear combination of the elements of $\varphi(x)$, were cover the original formulation in terms of the parameter vector w. Note that the prediction at $\boldsymbol{x}$ is given by a linear combination of the target values from the training set. The advantage of the dual formulation, as we shall see, is that it is expressed entirely in terms of the kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$. We can therefore work directly in terms of kernels and avoid the explicit introduction of the feature vector $\varphi(x)$, which allows us implicitly to use feature spaces of high, even infinite, dimensionality. In this way it is possible to extend linear models to infinite dimension objects.

The concept of a kernel formulated as an inner product in a feature space allows us to build interesting extensions of many well-known algorithms by making use of the **kernel trick**, also known as kernel substitution. The general idea is that, if we have an algorithm formulated in such a way that the input vector $\boldsymbol{x}$ enters only in the form of **scalar products**, then we can replace that scalar product with some other choice of kernel. For instance, the technique of kernel substitution can be applied to principal component analysis in order to develop a nonlinear variant of PCA. Kernelized SVM is one of the most effective ML method for classification and regression.

Kernel methods overcome difficulties in defining non-linear models. Using a not-linear kernel (like RBF) to a **not linearly separable dataset,** I can obtain, with a coordinate transformation in the input, a linearly separable dataset on which I can then apply various algorithms.

Kernel methods require the choice of suitable kernels and hyper-parameters tuning.

Cap 10: Instance-based Learning

The idea is to introduce a set of methods that allows to define non-parametric models. So far, we only considered *parametric models*, and the solution of our problem was given by finding a set of parameters for a model.

In a **parametric model**, a model has a fixed number of parameters. Remember that the size of the model does not depend on the size of the dataset. Instead in the **non-parametric model** the number of parameters grows with amount of data. For this kind of models, I cannot say in advance how many parameters I have.

Instance based Learning is an example of non-parametric model.

K-NEAREST NEIGHBORS is a type of instance-based learning, or lazy learning, where the function is only approximated locally, and all computation is deferred until function evaluation. There is no explicit model derived from the data.

In K-NN we have a fixed dataset and I want to classify a new point (new instance) as follows:

1. Find K nearest neighbors of new instance **x**
2. Assign to **x** the most common label among the majority of neighbors

The *likelihood* of a class c for a new instance $\boldsymbol{x}$ is:

$$p(c|\boldsymbol{x}, D, K) = \frac{1}{K} \sum_{i \in N_k(\boldsymbol{x}, D)} \mathbb{I}\,(y_i = c)$$

K-NN requires the entire training data set to be **stored**, leading to expensive computation if the data set is large. Increasing k brings to smoother regions (reducing overfitting).

The k-nearest neighbors' problem can be **kernelized** using **radial basis functions**, which depend only on the magnitude of the distance between the arguments. We can see that this is a valid kernel by expanding the square:

$$||\boldsymbol{x} - \boldsymbol{x}_i||^2 = \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}_i^T\boldsymbol{x}_i - 2\boldsymbol{x}^T\boldsymbol{x}_i$$

Since this algorithm relies on distance for classification, normalizing the training data can improve its accuracy dramatically. Instead of using a distance function, we can use a kernel function to define what are the instances that should influence the prediction of this instance.

Let's assume now to have a **regression problem** $f: X \to \mathfrak{R}$ with dataset $D = \{(x_n, t_n)_{n=1}^N\}$

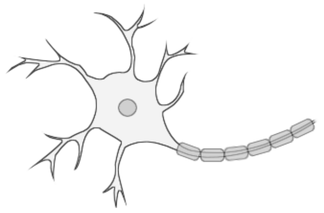
In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors. Given a new instance x we have that:

We look in the dataset the closest instances with the new one we want to predict and then we do some computation to return a prediction based on local samples.

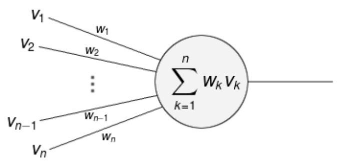
$$\hat{f}(\boldsymbol{x}) = \sum_{i \in N_k(\boldsymbol{x}, D)} y_i k(\boldsymbol{x}, \boldsymbol{x}')$$

In K-NN we take the k samples in the dataset closer to the instance that you want to classify, in regression we fit a local regression model around the query sample **x**.

Cap 11: Artificial Neural Networks



Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.



The goal of a feedforward network is to approximate some function f^*. For example, a classifier $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation.

Neural networks are parametric models in which we define an error function and we find the minimum of this function. There are many terms that can create some confusion. ANN represent all possible ways in which we create a network of units that are connected, so it's a family of networks. A Feedforward Neural Network (FNN) is a NN with a special constrain on the connection between nodes of the network. These models are called **feedforward** because information flows through the function being evaluated from **x**, through the intermediate computations used to define f, and finally to the output **y**. There are **no feedback** connections in which outputs of the model are fed back into itself. The signal proceeds only in one direction (without loops). It is a **layered architecture** in which each unit is connected only with the next layer and there are no connections between non-adjacent layer. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**.

NN are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.

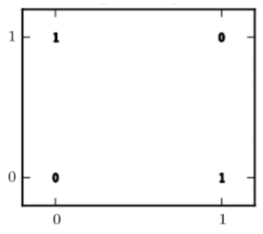
All these families are **parametric models** used for *classification* and *regression*. NN are complex combination of many parametric functions (non-convex problem). In general, NN are used for approximating a function and for this they are called *function approximation*.

The goal is to estimate some function $f: X \to Y$ (with $Y = \{C_1, \dots, C_k\}$ or $Y = \Re$), considering the dataset $D = \{(x_n, t_n)_{n=1}^{N}\}$ such that $t_n \approx f(x_n)$.

We want to define $y = \hat{f}(\boldsymbol{x};\boldsymbol{\theta})$ and learn the **parameters** $\boldsymbol{\theta}$ so that $\hat{f}$ approximates f.

The NN take inspiration from **neurons** of the brain. There is a **processing unit** that receives some information from other units and then generate an output that will be sent to other units.

FNNs are **chain structures** and the length of the chain is the **depth** of the network. The final layer is called **output layer**.



NN can automatically do something that we do manually in other tasks.

For example, in the XOR problem, a classification problem, we have in input 2 Boolean variable (0,1) and in output a single Boolean variable that is 1 only if there is exactly one element with 1. This is not a *linearly separable dataset* so it can't be solved with *linear models* while it can be solved through NNs.

Universal approximation theorem: an FFN with a linear output layer and at least one hidden layer wide enough (so with enough hidden units) with any "squashing" activation function (e.g., sigmoid) can approximate any Borel measurable function with arbitrary precision.

In theory, a short and very wide network can approximate any function. In practice, a deep and narrow network is easier to train and provides better results in generalization.

Increasing of parameters does not always lead to a better NN. A deep FFN has at least 3 hidden layers.

An important aspect of the design of a deep neural network is the choice of the cost function. Choice of network output units and cost function are related. We will distinguish 3 cases, and for each case we have a different **activation function** for the output layer and a different **cost function**:

1. Regression -> **linear** output unit -> mean squared error cost function
2. Binary classification -> **sigmoid** output unit -> binary cross-entropy (likelihood is a Bernoulli)
3. Multi-class classification -> **softmax** output unit -> categorical cross entropy (likelihood is a Multinomial)

Exam question

Linear regression squared error: $E(\theta) = \frac{1}{2}\sum_n^N (t_n - \theta^T x_n)$

Sigmoid error: $E(\theta) = -\ln P(t|x) = \text{softplus}((1-2t)\alpha)$

Softmax error: $E(\theta)_i = ln\sum_j e^{\alpha_j} - \alpha_i$

For the *hidden units,* there are no rigid criteria. Only experiments and intuitions can help us. Predicting which **activation function** will work best is usually impossible.

ReLU (Rectified Linear Unit) is a non-linear activation function defined as the positive part of its argument. It is very easy to compute and to optimize and can be a good choice for hidden units. A smooth approximation to the rectifier is the analytic softplus function $f(x) = ln(1 + e^x)$.

The sigmoid and the hyperbolic tangent activation functions are closely related because $tanh(z) = 2\sigma(2z) - 1$.

Exam question

Hidden activation functions:

- **ReLU** activation function: $g(\alpha) = \max(0, \alpha)$
- **Sigmoid** activation function: $g(\alpha) = \sigma(\alpha)$
- **Tangent** activation function: $g(\alpha) = \tanh(\alpha)$

Output activation function:

- **Linear regression** activation function: $y = \boldsymbol{w^T h} + \boldsymbol{b}$
- **Sigmoid** activation function: $y = \sigma(\boldsymbol{w^T h} + \boldsymbol{b})$
- **Softmax** activation function: $y_i = \text{softmax}(\alpha)_\text{i}$

Linear units do not saturate. Units in **sigmoid** saturates only when it gives the correct answer. Units in **softmax** saturates only when there are minimal errors.

Gradient-Based Learning

The largest difference between the linear models we have seen so far and neural networks is that the **nonlinearity** of a neural network causes most interesting **loss functions** to become **nonconvex**. This means that neural networks are usually trained by **using iterative, gradient-based optimizers** that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Stochastic gradient descent applied to nonconvex loss functions has no convergence guarantee and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values.

In a feedforward neural network, the information flows from input to output through the network. To train the network we need to compute the gradients with respect to the network parameters, and this is done by the **back-propagate algorithm** that is a simple and efficient algorithm that computes the gradient of a network (it is not a training algorithm as statistic gradient descent and it is not specific to FNNs).

The back-propagation algorithm allows the information from the cost to then flow backward through the network in order to compute the gradient.

The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the **chain rule**, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming.

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors x in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians, traveling backward through the graph in this way until we reach x. For any node that may be reached by going backward from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

In the 1st phase, the **forward phase,** it computes the *prediction* of the *current network* with respect to some input and compares the *output* of the *network* with the *output* of the *training set* and compute the *loss function*. In the 2nd phase, the backward phase, the cost computed is **backpropagated** from the output to the input to make an estimation.

Once we have the *gradient*, we want to find some local minimum. This is the crucial part of the learning, and there exists several *learning algorithms* as:

Stochastic Gradient Descent

In stochastic (or "on-line") gradient descent, the true gradient of the cost function is approximated by a gradient at a single example. A compromise between computing the true gradient and the gradient at a single example is to compute the gradient against more than one training example (called a "mini-batch") at each step. This can perform significantly better than "true" stochastic gradient descent.

SGD with Momentum

Stochastic gradient descent with momentum remembers the update Δ w at each iteration and determines the next update as a linear combination of the gradient and the previous update. Momentum can accelerate learning. The name momentum stems from an analogy to momentum in physics: the weight vector w, thought of as a particle traveling through parameter space, incurs acceleration from the gradient of the loss ("force"). Unlike in classical stochastic gradient descent, it tends to keep traveling in the same direction, preventing oscillations.

Algorithms with adaptive learning rates
Based on analysis of the gradient of the loss function it is possible to determine, at any step of the algorithm, whether the learning rate should be increased or decreased.

Regularization is an important feature to reduce overfitting (generalization error). It can be done in several methods:

1. **Parameter norm penalties:** it adds a regularization term E_{reg} to the cost function
2. **Data augmentation**: *generating additional data* and including them in the dataset (we can add noise or apply data transformation such as image rotation, scaling ecc...).
3. **Early Stopping:** *stopping* iterations early avoid overfitting to the training set of data.
4. **Parameter sharing:** it constrains some parameters to be the same. In this way, the NN has to learn only one set of parameters shared to all the units.
5. **Dropout:** it randomly removes network units with some probability α.

Exam questions:

If we have a regression problem with an ANN with a target function $R^m \rightarrow R^n$ and we have x hidden units, the **number of hidden parameters** is:

$$mx + x + nx + n$$

There are mx connections from input to hidden units, x hidden units, nx connections from hidden to output and finally n output units.

Back-propagation is affected by **local minima** and this can be resolved with the momentum. In fact, with this the momentum, you avoid "finding" a local minimum where the error function can no longer be differentiable.

Back-propagation is not affected by **overfitting**. In fact, it is not a training algorithm. It's only a method to compute the gradients. The model can be affected by overfitting and there are several methods to avoid it like: early stopping, in which you stop iterations early, parameter sharing, in which you constrain on having subset of model parameters to be equal, or dropout in which you randomly remove network units with some probability alpha.

The stochastic gradient descent differs from the backpropagation because it uses mini-batch of the dataset, and modify the parameters based on the gradient and the learning rate

FOCUS ON BACK-PROPAGATION

EXAMPLE FROM WIKIPEDIA:

Consider a simple neural network with two input units, one output unit and no hidden units, and in which each neuron uses a linear output that is the weighted sum of its input.

Initially, before training, the weights will be set randomly. Then the neuron learns from training examples, which in this case consist of a set of tuples ***(x_1,x_2,t)*** where x_1 and x_2 are the inputs to the network and t is the correct output (the output the network should produce given those inputs, when it has been trained). The initial network, given x_1 and x_2, will compute an output y that likely differs from t (given random weights). A loss function L(y,t) is used for measuring the discrepancy between the target output t and the computed output y.

As an example, consider a regression problem using the square error as a loss:

$$L(t, y) = (t - y)^2 = E$$

where E is the discrepancy or error.

Consider the network on a single training case: (1,1,0). Thus, the input and are 1 and 1 respectively and the correct output, t is 0. Now if the relation is plotted between the network's output y on the horizontal axis and the error E on the vertical axis, the result is a parabola. The minimum of the parabola corresponds to the output y which minimizes the error E. For a single training case, the minimum also touches the horizontal axis, which means the error will be zero and the network can produce an output y that exactly matches the target output t. Therefore, the problem of mapping inputs to outputs can be reduced to an optimization problem of finding a function that will produce the minimal error.

However, the output of a neuron depends on the weighted sum of all its inputs:

$$y = x_1 w_1 + x_2 w_2$$

Where w_1 and w_2 are the weights on the connection from the input units to the output unit. Therefore, the error also depends on the incoming weights to the neuron, which is ultimately what needs to be changed in the network to enable learning. If each weight is plotted on a separate horizontal axis and the error on the vertical axis, the result is a parabolic bowl. For a neuron with k weights, the same plot would require an elliptic paraboloid of dimensions. One commonly used **algorithm to find the set of weights that minimizes the error is gradient descent**. **Backpropagation is then used to calculate the steepest descent direction in an efficient way**.

A BETTER EXPLANATION HERE:

https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQO-OWTQDNU6R1_67000Dx_ZCJB-3pi&index=3

Cap 12: CNN

We will see now how to apply neural networks to input signals like video, images, audio (sometimes pre-processed). Signals are numerical representations of physical quantities.

Convolution or Cross-correlation is an operation used in these types of networks and consist of multiplying images with some *kernels* in order to do some *transformation*.

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x(t), the position of the spaceship at time t. Both x and t are real -valued, that is, we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w(a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called convolution. In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the input, and the second argument (in this example, the function w) as the **kernel.** The output is sometimes referred to as the **feature map**.

There is at least one convolutional layer, a layer that uses convolution as operation. We want to learn the parameters of the *kernels* in order to transform the image in another image, and this operation will contribute to classify correctly out dataset.

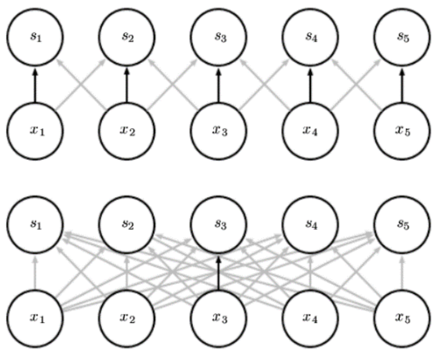
In a convolutional layer we usually have 3 components:

1. **Convolutional stage**, in which we apply a *convolutional filter* (one or more filters).
2. **Detector stage**, in which an **activation function** is applied (usually *non-linear function*).
3. **Pooling stage**:

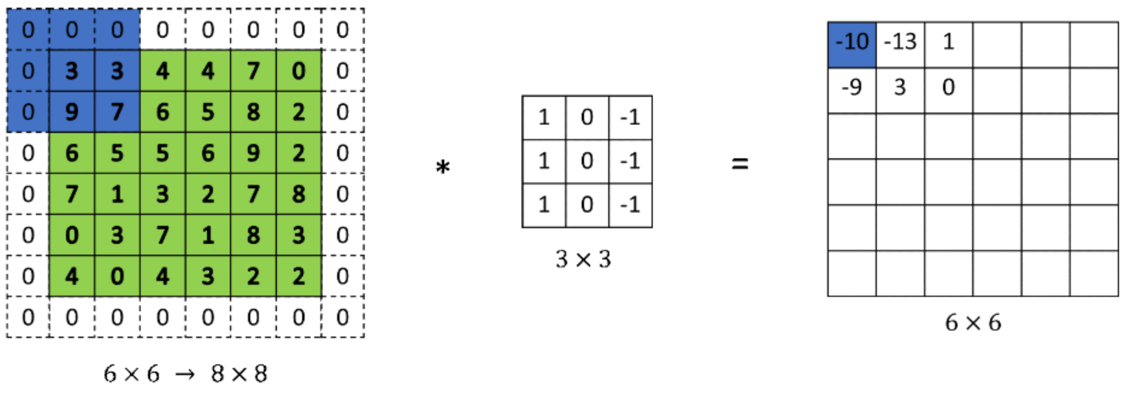
Convolutional networks typically have **sparse connectivity** (also referred to as sparse interactions or sparse weights). Sparse connectivity means that only a subset of units is connected to the input. This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency.

The **parameter sharing** used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. The values of the *kernel* will be shared through all the instances of the input. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. The particular form of parameter sharing causes the layer to have a property called **equivariance to translation**.

In the figure, black arrows indicate the connections that use a particular parameter in two different models. In the one on the top, the black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Because of parameter sharing, this single parameter is used at all input locations in the top one. In the second one, the single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing, so the parameter is used only once.



Padding works by extending the area of which a convolutional neural network processes an image. The kernel is the neural networks filter which moves across the image, scanning each pixel and converting the data into a smaller, or sometimes larger, format. Every time we apply a convolutional operator our image shrinks. Usually we don't want our image to shrink every time we detect the edges or other features in it. Pixels on the corners around the image border are used much less in the output so we're throwing away a lot of the information near the border of the image. In order to assist the kernel with processing the image, padding is added to the frame of the image to allow for more space for the kernel to cover the image. Adding padding to an image processed by a CNN allows for more accurate analysis of images. In the following figure there is an example of padding of 1 before convolving with a 3x3 matrix.



Stride controls how the filter convolves around the input volume. The amount by which the filter shifts is the stride. Stride is normally set in a way so that the output volume is an integer and not a fraction.

Once we have done convolution, we have the **Detector Stage** in which we apply some *non-linear transformation* to the output of the convolution stage like *ReLU* or *tanh*.

We then have the **Pooling Stage**, in which we apply the *pooling operation*. This is generally used to reduce size of the layer or to introduce some invariants with respect to translation or also for scaling and rotation. In fact, the system must be *robust* to translation or small transformation. It is an operation similar to a kernel operation, but the values won't be trained (in *pooling step* we just want to perform a fixed operation). A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood.

The **width** and the **height** of the output of each layer of a **CNN** are calculated as:

$$w_{output} = \frac{w_{input} - w_{kernel} + 2\ padding}{stride} + 1$$

$$h_{output} = \frac{h_{input} - h_{kernel} + 2\ padding}{stride} + 1$$

The **third dimension** is the number of the **feature maps** of the layer.

The number of parameters for a convolutional layer (in the pooling layer is 0 because there are no parameters you can learn in pooling), is calculated as:

$$n_{parameters\_layer} = (n \cdot m \cdot\ l + 1) \cdot\ k$$

$n \cdot m$ is the Kernel size (ex. 5x5, 3x3)

l number of feature maps in input in the layer

k number of feature maps in output of the layer

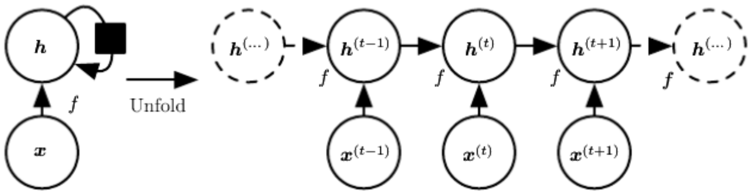
Cap 13: Complements of NN

Recurrent neural networks (RNNs) are a family of NN for processing **sequential data**.

Dealing with sequences, we have to use NNs that take into account the fact that data points are related and their order is important.

In a recurrent network, each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.

Cycles represent the influence of the present value of a variable on its own value at a future time step.



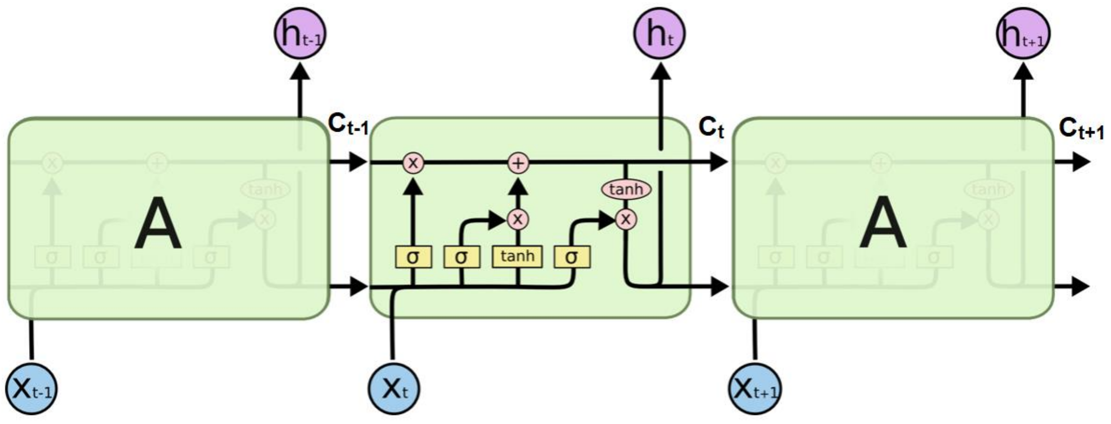
Unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events, results in the sharing of parameters across a deep network structure. Each cell is then dedicated to a different data sample.

Long Short Term Memories (LSTMs) are RNNs with a special structure, designed to capture long-term dependencies. LSTMs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode.

LSTM recurrent networks have "LSTM cells" that have an **internal recurrence** (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information.

Leaky units allow the network to accumulate information over a long duration. Once that information has been used, however, it might be useful for the neural network to forget the old state. For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it.

The main idea is to separate cell output h_t and cell state C_t.



1st gate: *forget* mechanism.

The first step is to decide what information we're going to throw away from the cell state. A sigmoid layer drops elements of C_{t-1} based on values of h_{t-1} and input x_t.

2nd & 3rd gate: *update* mechanism.

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C'_t, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

Last step: compute *output.*

Finally, we need to decide what we're going to output. This output will be based on our cell state but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

Transfer Learning

Transfer learning focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks. Talking about image classification, there are several CNN pre-trained on millions of images with thousands of output classes that can be used for a different domain and/or learning task. There are two main approaches to transfer learning of CNN.

Fine-tuning

It uses same network architecture with pre-trained model (with network parameters 'copied' from the pre-trained model). It freezes the parameters of some layers (usually the first ones) and it trains of all network parameters with back-propagation.

CNN as feature extractor

It extracts features at a specific layer of CNN**.** It then collects extracted features x_0 of training/validation split and associate corresponding labels t in a new dataset D' which is used to train a new classifier. In this way there is no need to train the CNN, but features, source and target domains should be as 'compatible' as possible because they cannot be modified.

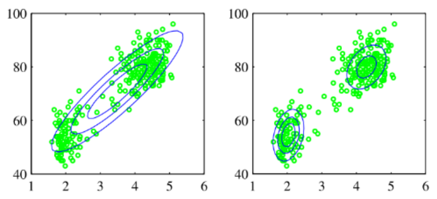
Cap 14: Unsupervised Learning

In many situations we want to find some features in the data in analysis. Unsupervised Learning (UL) is **learning without a teacher**, meaning that we have data without labels. We just have input dataset available $D = \{x_n\}$ but we don't have any *target values*.

We want to cluster data depending on their characteristic, finding multiple classes from data. Unsupervised learning algorithm determine ***mixed probability distribution*** from data.

Sometimes, UL can be merged with supervised learning to provide better results.

Gaussians Mixture Model (GMM)

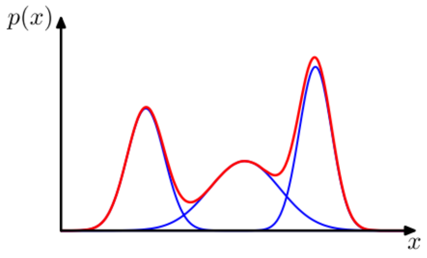


While the Gaussian distribution has some important analytical properties, it suffers from significant limitations when it comes to modelling real data sets. Many times, we see many clumps in data, and a simple Gaussian distribution is unable to capture this structure, whereas a linear superposition of different Gaussians gives a better characterization of the data set. Such superpositions, formed by taking linear combinations of more basic distributions such as Gaussians, can be formulated as probabilistic models known as *mixture distributions.*

A linear combination of Gaussians can give rise to very complex densities. By using a sufficient number of Gaussians, and by adjusting their means and covariances as well as the coefficients in the linear combination, almost any continuous density can be approximated with arbitrary accuracy.

We therefore consider a superposition of K Gaussian densities of the form:

$$P(\boldsymbol{x}) = \sum_{k01}^{K} \pi_k \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})$$



Each Gaussian density $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})$ is called a component of the mixture and has its own mean vectors μ_k and covariance matrix Σ_k.

K-means Clustering

k-means has the goal of computing K means of the gaussian distribution, given a dataset.

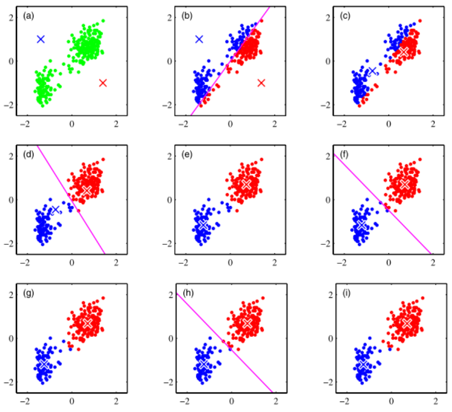
The input is a set of samples and the value k, so we have to tell the system how many gaussian are present or how many clusters we want to form and the output of the algorithm will be the means of the distributions (without considering weights and covariance).

K-means is an iterative process with several steps:

1. Begin with a decision of the *value of k* (number of cluster).
2. Put any **initial partition** that classify the data in k cluster. You can take random training samples or you can assign the first k elements of the dataset as training samples, and you assign the remaining (N-k) training samples to cluster with the nearest **centroid** (means of the distribution) and after each assign recompute the centroid.
3. Take each sample in sequence and compute its distance from the centroid of each cluster, if a sample is not in the closest centroid you switch it and you update the centroid of the two clusters involved.
4. Repeat step 3 until we obtain the convergence (the centroid has stabilized).

The **convergence** will always occur if:

- For each switch in step 2, the sum of distances from each training sample to that training sample's group centroid is decreased.
- There are only finitely many partitions of the training examples into k clusters.



This method has several drawbacks:

▪ The number of **clusters K** must be determined beforehand.

▪ Sensitive to **initial condition** (local optimum) when a few data are available.

▪ It's not robust to **outliners**. Very far data from the centroid may pull the centroid away from the real one.

▪ The result is a **circular cluster shape** because it's based on distance.

Gaussian mixtures with latent variables

We now turn to a formulation of Gaussian mixtures in terms of discrete **latent variables**. This will provide us with a deeper insight into this important distribution and will also serve to motivate the expectation-maximization algorithm.

Let us introduce a <u>K-dimensional binary random variable z</u> having a 1-of-K representation in which a particular element z_k is equal to 1 and all other elements are equal to 0. The values of z_k therefore satisfy $z_k \in \{0,1\}$ and $\sum z_k = 1$, and we see that there are K possible states for the vector z according to which element is nonzero.

The **marginal distribution over z** is specified in terms of the mixing coefficient π_k, such that:

$$p(z_k = 1) = \pi_k$$

Thus:

$$p(\boldsymbol{z}) = \prod_{k=1}^{K} \pi_k^{z_k}$$

Similarly, the **conditional distribution** of x given a particular value for z is a Gaussian:

$$p(\boldsymbol{x}|z_k = 1) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})$$

Thus:

$$p(\boldsymbol{x}|\boldsymbol{z}) = \prod_{k=1}^{K} \mathcal{N}\,(\boldsymbol{x}|\boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})^{z_k}$$

From the chain rule, the joint distribution $p(\boldsymbol{x}, \boldsymbol{z})$ is given by $p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$, and the **marginal distribution of $\boldsymbol{x}$** is then obtained by summing the joint distribution over all possible states of z to give:

$$p(\boldsymbol{x}) = \sum_{z} p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})$$

We have therefore found an <u>equivalent formulation of the Gaussian mixture involving an explicit latent variable</u>. We are now able to <u>work with the joint distribution $p(\boldsymbol{x}, \boldsymbol{z})$</u> instead of the marginal distribution $p(\boldsymbol{x})$, and this will lead to significant simplifications, most notably through the introduction of the expectation-maximization (EM) algorithm.

GMM distribution $p(\boldsymbol{x})$ can be seen as the marginalization of a distribution $p(\boldsymbol{x}, \boldsymbol{z})$ over variables z.

Another quantity that will play an important role is the **conditional probability of z given x**. We shall use $\gamma(z_k)$ to denote $p(z_k = 1|\boldsymbol{x})$, whose value can be found using Bayes' theorem:

$$\gamma(z_k) = p(z_k = 1|\boldsymbol{x}) = \frac{P(z_k = 1)P(\boldsymbol{x}|z_k = 1)}{P(\boldsymbol{x})} = \frac{\pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}; \boldsymbol{\Sigma_k})}{\sum_{j=1}^{J} \pi_j \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_j}; \boldsymbol{\Sigma_j})}$$

We shall view π_k as the prior probability of $z_k = 1$, and the quantity $\gamma(z_k)$ as the corresponding posterior probability once we have observed x.

The output of the Gaussian Mixture Model is a probability distribution that is the weighted sum of k-gaussian distributions.

Expectation Maximization (EM)

An elegant and powerful method for finding maximum likelihood solutions for models with latent variables is called the expectation-maximization algorithm. EM algorithm is a general method to estimate likelihood for mixed distributions including observed and latent variables.

Expectation Maximization is a generalization of the k-means algorithm, because it can estimate all the *parameters* (means $\boldsymbol{\mu_k}$, covariances $\boldsymbol{\Sigma_k}$ and mixing coefficients $\pi_k$while k-means estimates only the means. It uses the **maximum likelihood** formulation.

$$\underset{\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}}{\operatorname{argmax}} \ln p(\boldsymbol{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})$$

It is worth emphasizing that there is not a closed-form solution for the parameters of the mixture model because the posterior probabilities $\gamma(z_{nk})$ depend on those parameters in a complex way. However, a simple **iterative scheme** for finding a solution to the maximum likelihood problem can be applied. We first choose some initial values for the means, covariances, and mixing coefficients. Then we alternate between the following two updates that we shall call the E step and the M step.

In the expectation step, or **E step**, we use the current values for the parameters to evaluate the posterior probabilities. We then use these probabilities in the maximization step, or **M step**, to re-estimate the means, covariances, and mixing coefficients.

The big difference from the k-means is that in the k-means we assign each point to a cluster while EM **assigns a probability distribution** to each point (a point can be 75% blue and 25% red).

EM can also be generalized to other distributions (not only Gaussians).

EM is a greedy approach. We have no guarantee that the solution will converge to a global maximum likelihood. It depends on initialization phase.

Cap 15: Dimensionality Reduction

We discussed probabilistic models having discrete latent variables, such as the mixture of Gaussians. We now explore models in which some, or all, of the latent variables are continuous. An important motivation for such models is that many data sets have the property that the data points all lie close to a manifold of much lower dimensionality than that of the original data space.

To see why this might arise, consider an artificial data set constructed by taking one of the off-line digits, represented by a 64 x 64 pixel grey-level image, and embedding it in a larger image of size 100 x 100 by padding with pixels having the value zero (corresponding to white pixels) in which the location and orientation of the digit is varied at random. Across a data set of such images, there are only three degrees of freedom of variability, corresponding to the vertical and horizontal translations and the rotations. The data points will therefore live on a subspace of the data space whose intrinsic dimensionality is three. The manifold will be nonlinear because, for instance, if we translate the digit past a particular pixel, that pixel value will go from zero (white) to one (black) and back to zero again. which is clearly a nonlinear function of the digit position. In this example, the translation and rotation parameters are latent variables because we observe only the image vectors and are not told which values of the translation or rotation variables were used to create them. For real digit image data, there will be a further degree of freedom arising from scaling. Moreover, there will be multiple additional degrees of freedom associated with more complex deformations due to the variability in an individual's writing as well as the differences in writing styles between individuals. Nevertheless, the number of such degrees of freedom will be small compared to the dimensionality of the data set.

Any images dataset has this issue. The number of dimensions is given by the size of the images. If you take a dataset of digits of an image 64x57 (rows, columns) the input space has a **dimensionality** given by the product of these two values (around 4000). If we consider all possible combinations that we will have in a dataset with 4000 dimensions we will have many images that are no meaningful. If we want to classify only images of a cat or a dog, there are much less possible combination of color pixels in an image. The idea is to try to analyze the input to understand the real dimensions that make the variability of our dataset.

Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables.

It can be divided into **feature selection** and **feature extraction**.

Principal Component Analysis (PCA)

Principal Component Analysis is a technique for various tasks as: dimensionality reduction, data compression, data visualization and feature extraction.

It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.

Given data $\{x_n\} \epsilon \mathbb{R}^D$, PCA maximizes data **variance after projection** to some direction **u₁**.

Projected point: $\boldsymbol{u}_1^{\boldsymbol{T}} \boldsymbol{x_n}$

Variance of projected points:

$$\frac{1}{N}\sum_{n=1}^{N}[\boldsymbol{u}_1^{\boldsymbol{T}}\boldsymbol{x_n} - \boldsymbol{u}_1^{\boldsymbol{T}}\overline{\boldsymbol{x}}]^2 = \boldsymbol{u}_1^{\boldsymbol{T}} S \boldsymbol{u}_1$$

To maximize the projected variance subject to constrain $\boldsymbol{u}_1^T\boldsymbol{u}_1$ =1 is equivalent to an unconstrained maximization with a Lagrange multiplier:

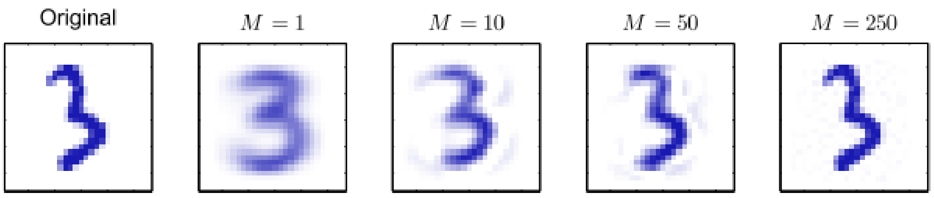
$$\max_{\boldsymbol{u}_1} \boldsymbol{u}_1^T S \boldsymbol{u}_1 + \lambda_1(1 - \boldsymbol{u}_1^T\boldsymbol{u}_1)$$

Setting the derivative with respect to $\boldsymbol{u}_1$to zero we notice that $\boldsymbol{u}_1$**must be an eigenvector of the covariance matrix** S.

$$S\boldsymbol{u}_1 = \lambda_1\boldsymbol{u}_1$$

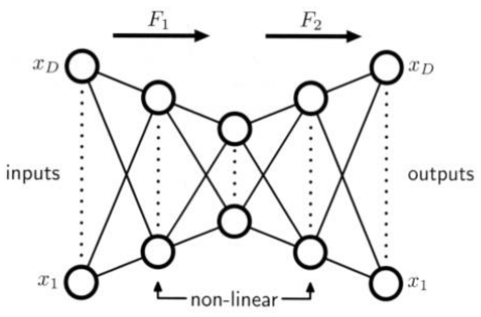
Repeat to find other directions which maximize variance of projected data and that are **orthogonal** to the previous directions. The eigenvectors that correspond to the largest eigenvalues (the principal components) can now be used to reconstruct a large fraction of the variance of the original data. Moreover, the first few eigenvectors can often be interpreted in terms of the large-scale physical behavior of the system.

In the following figure there is an example of a reconstruction with a limited number of components.



Autoassociative neural networks (Autoencoders)

Previously, we considered neural networks in the context of supervised learning, where the role of the network is to predict the output variables given values for the input variables. However, neural networks have also been applied to unsupervised learning where they have been used for dimensionality reduction. This is achieved by using a network having the same number of outputs as inputs, hidden layers with reduced sized (**bottleneck**) and optimizing the weights so as to minimize some measure of the reconstruction error between inputs and outputs with respect to a set of training data.

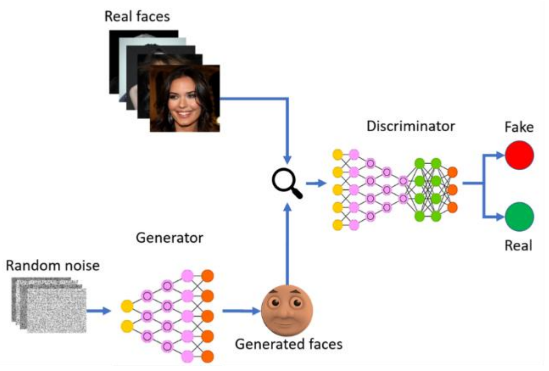


Addition of extra hidden layers of **nonlinear units** gives an autoassociative network the ability to perform a **nonlinear dimensionality reduction**.

Generative Models

Generative Adversarial Networks (GANs) focus on learning a distribution, while **Variational Auto-Encoders (VAEs)** focus on learning latent space structure.

Given a training set, **GAN** learns to generate new data with the same statistics as the training set. For example, a GAN trained on photographs can generate new photographs that look at least superficially authentic to human observers. It uses adversarial training, opposing two networks, a **generator (decoder)** and a **discriminator** (critic), to push both of them to improve iteration after iteration.



The generator network is made by a **decoder**, which receives a code (random vector) and produces an image (the opposite of an encoder as a CNN, which processes an image and produces a vector/code).

A **VAE** is an autoencoder whose encodings distribution is regularized during the training in order to ensure that its latent space has good properties allowing us to generate some new data.

Its goal is to modify data in specific directions (change expression or add glasses in a photo), identifying meaningful directions in latent space.

Exam questions

In order to represent x on the M components I need to project each point on $u_1, u_2, \ldots, u_m$ using the formula $\boldsymbol{u}_1^{\boldsymbol{T}}\boldsymbol{u}_1$, $\boldsymbol{u}_2^{\boldsymbol{T}}\boldsymbol{u}_2$ and so on. In order to be coherent with what **PCA,** these vectors must maximize the variance. After computing the mean of the x and the covariance matrix S on the whole dataset, and I will take the **M highest eigenvalues**:

$$\boldsymbol{u}_{1..\boldsymbol{m}}^{\boldsymbol{T}} S \boldsymbol{u}_{1\ldots\boldsymbol{m}} = \lambda_{1..m}$$

Autoencoders are neural networks with *hidden layers* smaller in size than other layers (bottleneck) and they are mainly used in **non-linear models**. The bottleneck will therefore be forced, due to its limited numbers of units, to re- transform/reconstruct the input values into output values smaller in size.

Cap 16: Multiple Learners

Instead of trying to get the best from a single model, we try to train different models (maybe they are not optimally trained) and after that we combine them. The result has a better *accuracy* and is more *efficient* than any individual one.

Models can be trained in **parallel** (*voting* or *bagging*) or in **sequence** (*boosting*).

In the **voting** we use the same dataset to train different models and the *prediction* can be done by a simple *weighted average* of the *prediction* of each single model. We use the weights because if you think that a model is better than another, you can improve its influence on the weighted average. There are several types of voting system like:

- **Mixture of experts**: a function that depends on input instance decides the different distribution of the weights.
- **Stacking**: function that decides the weights is not given a priori but the system adapts it.
- **Cascading**: we query one learner at time, and there is a selection on the result based on a threshold. At the end, the result will not be a combination of the output. It will be the output of the one with higher confidence.

In the **bagging**, each model is trained with a different subset of the dataset; the generation of subsets typically is made with random sampling with replacement.

In the **boosting** instead we train all the learner sequentially, so each time the dataset depends on training of the previous model. This approach is very effective because when you generate dataset for the next learner you give more importance to the samples that are not being correctly classified by the previous one. Later models tend to focus on harder-to-classify examples. Weights depend on performances of previous classifiers. Most common algorithm for boosting is **AdaBoost**, in which the algorithm will compute the *weights* for each model (samples not correctly classified with a bigger weight) and this process is repeated M times, where M is the number of *learner* that we want to train. Once the M learners are trained, we consider a *weighted average* of the output of all the learners. AdaBoost can be explained as the sequential minimization of an exponential error function. The problem of this algorithm is that the performance depends on *data* (fail if there are insufficient data) and is sensitive to *noise*.

Instead of designing a learning algorithm that is accurate over the entire space, one can focus on finding base learning algorithms that only need to be better than random. Combined learners theoretically outperform any individual learner. AdaBoost practically outperforms many other base learners in many problems.

To minimize the error, **AdaBoost,** instead of computing the *global error,* uses a **sequential method**. Starting from the last model (the last trained), it considers the remaining models as constant values. The error can be derived by considering only the last model, and the algorithm does this backwards up to the first model so the minimum value for the error is obtained.

Cap 17: Markov Decision Processes and Reinforcement Learning

We address now the computational issues involved in making decisions in a stochastic environment.

Reinforcement learning is the ability of an agent to **learn a behavior**. It is a branch of AI that tries to develop a computational approach to solving the problem of **learning through interaction**.

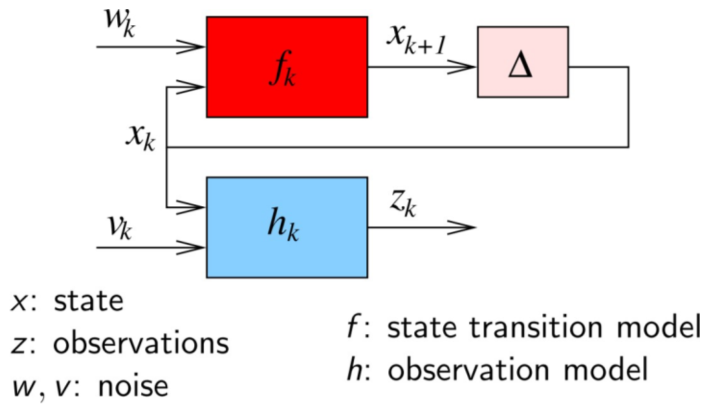
Reinforcement learning is learning what to do—**how to map situations to actions**—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning is different from unsupervised learning because reinforcement learning is trying to **maximize a reward signal** instead of trying to find hidden structure. RL explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment.

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of **states**, **actions**, and **rewards**.

The basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment.

A **dynamic system** is a system whose **state** evolves over time. State is a representation of the *information needed to model the system*.



Notation:

- X set of states
- A set of actions
- Δ transition function
- Z set of observations

The system evolves because there is a **dynamic transition function f**, responsible to model how *states* evolve over times (how to compute *next state* given the current). This evolution is affected by some **noises** (w, v), coming from *environment* (not modelled in the state). Sometimes the state can be observed. We usually have an **observation model,** a component able to extract *information* from the state and its outcome is denoted with z.

When the state is fully observable, the decision-making problem for an agent is to decide which action must be executed in a given state.

We want to define an agent able to *drive the evolution of the state*. For example, if we want an agent that is able to play chess, we need an agent able to find a sequence of states that has some advantages, so we need an <u>agent able to control the sequence of states to achieve some goal</u>.

We have 2 possibilities for design such an AI: reasoning and learning.

- **Reasoning:** knowing every information about the model (f, h) and the current state, I can compute the next state from my current state, generating my behavior. I can just *predict* without executing any action.
- **Learning**: I don't have *any information* about the transition function and the observation model. I don't know how the system evolves but I can make tests. I can try, executing some actions, and moving to the next state. Based on this knowledge, I can learn how to do better the next time. Given *past experience* I determinate the *model*.

The state x contains all information needed in order to take decisions about the future; we can imagine that as a *snapshot* at a given time that you can take on the system you are analyzing.

The learner and decision maker is called the agent. The goal for the agent is to define a function that allows him to make decision about what to do in a given state. The agent has to compute the **policy/behavior function**, that maps states into actions $\pi: \boldsymbol{X} \rightarrow A$. Policy is a mapping of an action to every possible state in the system.

When the model is not known the agent must learn the function π.

One difference between Supervised Learning and Reinforcement Learning is in the dataset. In supervised learning we have a dataset composed by set of pairs (input, output) of the function we want to learn, while in RL we have a dataset in which for each sequence of states I collect rewards, numerical values that determine how good is the action.

Supervised learning: learning a function $f: X \rightarrow Y$, given $D = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$

Unsupervised learning: learning a behavior function $\pi: X \rightarrow A$, given $D = \{\langle \boldsymbol{x}_1, a_1, r_1, \dots, \boldsymbol{x}_n, a_n, r_n \rangle^{(i)}\}$

RL algorithms are **domain independents**. In order to learn how to play chess, we don't need to encode in any way the rules of the game so they can learn any policy from any game.

An important property that we are going to use is the **Markov property**, which allows a significant reduction of complexity.

"The future is independent of the past given the present."

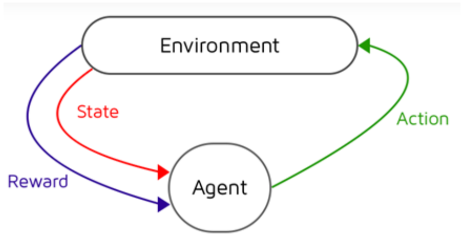
The knowledge of current state is all we need to make predictions. Current state contains all the information needed to choose the best action. We don't need to know what the history has been to reach current state. A Markov process is a process that has the Markov property. In some dynamic system the history of actions and states may be relevant, so this is an important simplification.

Markov Decision Processes (MDP)

Sequential decision problems in uncertain environments, also called Markov decision processes, are defined by a transition model specifying the probabilistic outcomes of actions and a reward function specifying the reward in each state.

Markov Decision Processes is a tuple that contains the set of states that we denote with X (finite and discrete), the set of actions denoted with A (finite and discrete), the transition function Δ and the reward function r.

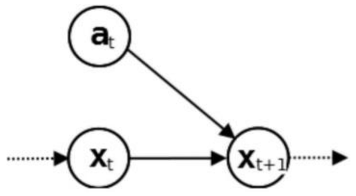
$$MDP = \langle \boldsymbol{X}, \boldsymbol{A}, \delta, r \rangle$$



An MDP is a model of an agent interacting synchronously with a world. The agent takes as input the state of the world and generates as output actions, which themselves affect the state of the world.

In the MDP framework, it is assumed that, although there may be a great deal of uncertainty about the effects of an agent's actions, there is never any uncertainty about the agent's current state—it has complete and perfect perceptual abilities. <u>MDPs are **fully observable**</u>. Even in presence of non-deterministic or stochastic actions, the resulting state is not known before the execution of the action, but it can be fully observed after its execution.

Thanks to Markov property, once I know the current state X_t, all the information about the past are useless. In a Bayes network there is an arrow from X_t to X_{t+1} meaning that there is a probability distribution between the two states but there won't be arrows coming directly from any previous point. Instead, if you don't have any Markov assumption, the probability distribution for reaching the next state depends on previous states.



$\delta: \boldsymbol{X} \times \boldsymbol{A} \to \boldsymbol{X}$ is a transition function

$r: \boldsymbol{X} \times \boldsymbol{A} \to \Re$ is a reward function (sometimes it is defined as $r: \boldsymbol{X} \to \Re$)

Markov property: $\boldsymbol{x_{t+1}} = \delta(\boldsymbol{x_t}, a_t)$ and $r_t = r(\boldsymbol{x_t}, a_t)$

- In a **deterministic MDP**, the transition function is a function that given a pair <current-state, current-action> return exactly one next state. $\delta: \boldsymbol{X} \times \boldsymbol{A} \to \boldsymbol{X}$
- In a **non-deterministic MDP**, whatever is the action the agent decides to execute, we have no guarantees about the outcome. Actions are unreliable. The transition function codomain is now 2^x, the set of all possible sets of x. Given a state and an action, the transition function returns a set of possible states, but we will be able to observe the next state only after the execution of the action. $\delta: \boldsymbol{X} \times \boldsymbol{A} \to 2^{\boldsymbol{X}}$. Also, the reward function is extended. We have to consider the outcome of an action. Reward depends on the current state, the action and the next state and it is given when I execute an action and this action brings to the next state. $r: \boldsymbol{X} \times \boldsymbol{A} \times \boldsymbol{X} \to \Re$. An example can be a robot with noisy sensors and effectors. It is appropriate in that case to model actions and rewards as nondeterministic.
- In a ***Stochastic MDP*** there is a *probabilistic transition* $p(\boldsymbol{x}'|\boldsymbol{x}, \boldsymbol{a})$, the conditional probability of the successor state, given the current state and the current action. It's an extension of a non-deterministic MDP. An example can be the game of Backgammon, where the action outcome is probabilistic since move involves roll of dice.

Given an MDP, we want to find the **optimal policy** π^*, the best action to be executed at any given time. Optimality is defined in terms of the <u>*cumulative (discounted) reward.*</u>

Policy is a function $\pi: \boldsymbol{X} \to A$ and, for each state, $\pi(x) \epsilon A$ is the optimal action to be executed in such state. We can define the optimization function as the *sum of all rewards*. Giving the same value to rewards obtained soon or in the future is not the best idea since it can lead to non-optimal choices.

Instead, we use a **discount factor $\boldsymbol{\gamma}$**, an exponential term that penalizes rewards that are in the future, in the expression of the **value function V**.

In a deterministic case:

$$V^{\pi}(x_1) = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots$$

In a non-deterministic/stochastic case:

$$V^{\pi}(x_1) = E[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots]$$

The optimal policy is then:

$$\pi^* = \operatorname*{argmax}_{\pi} V^{\pi}(x) \ \forall x \in X$$

Where $r_t = r(x_t, a_t, x_{t+1})$, $a_t = \pi(x_t)$ and $\gamma \in [0,1]$

For infinite horizon problems, a stationary MDP always has an optimal stationary policy. Stationary policy means that the action that is chosen from the policy function is always the same.

The general problem is given by a $MDP = \langle \boldsymbol{X}, \boldsymbol{A}, \delta, r \rangle$ and the solution is a policy $\pi: \boldsymbol{X} \to A$.

If the $MDP = \langle \boldsymbol{X}, \boldsymbol{A}, \delta, r \rangle$ is completely known, it is a *reasoning* or *planning* problem. Otherwise, it is a *learning* problem.

In a simple **one-state MDP** $MDP = \langle \{\boldsymbol{x}_0\}, \boldsymbol{A}, \delta, r \rangle$, there is only one state x_0 and there is a set of possible actions. Whenever we execute an action, we come back to state x_0. The *transition function* just pushes it back to the unique state $\delta(\boldsymbol{x}_0, a_i) = \boldsymbol{x}_0, \forall a_i \in \boldsymbol{A}$. The *reward* function assigns a reward to any execution of the action. The *optimal policy* is just one particular action among all the possible actions.

- If r is deterministic and known $\pi^*(x_0) = \underset{a_i \in A}{\operatorname{argmax}}\, \mathrm{r}(a_i)$
- If r is deterministic and unknown then execute each a_i ($|A|$ executions), collect the rewards and return the best a_i
- If r is not deterministic and known $\pi^*(x_0) = \underset{a_i \in A}{\operatorname{argmax}}\, \mathrm{E}[\mathrm{r}(a_i)]$
- If r is not deterministic and unknown collect many times T ($T \gg |A|$ executions), the rewards for actions and update a data structure at entry a_i with the average of all rewards of a_i.

When we have **more than one state**, the system will evolve and change its state after the execution of an action. If transition function and reward function are not known the agent cannot predict the effects of its action, but it can execute them and then observe the outcome. The **learning** task is thus performed by repeating these steps:

- Choose an action
- Execute it
- Observe the resulting state
- Collect reward

Until this point we considered the problem of evaluating the performance of a specific policy π applied to an MDP. If we want to find the **optimal policy $\boldsymbol{\pi}^*$** (the one maximizing the value in each state) for a given MDP we may consider as solution tools:

- Brute force: enumerate all the possible policies, evaluate their values and consider the one having the maximum value.
- ***Value iteration*** based on estimating the value function $V^{\pi}(x)$ and then computing the policy.
- ***Policy iteration*** based on estimating directly the policy.

The utility of a state is the expected utility of the state sequences encountered when an optimal policy is executed, starting in that state. **The value iteration** algorithm for solving MDPs works by iteratively solving the equations relating the utility of each state to those of its neighbors. It estimates the value function $V^{\pi^*}(x)$ and then it determines the optimal policy:

$$\pi^* = \underset{a \in A}{\operatorname{argmax}}[r(x,a) + \gamma V^*(\delta(x,a))]$$

Where $r(x, a)$ represents the reward obtained by executing the action.

Many times, the policy cannot be computed in this way because δ and r are unknown. If the probabilities or rewards are unknown, the problem is one of **reinforcement learning**.

For this purpose, it is useful to define a further function, which corresponds to taking the action a and then continuing optimally. In order to compute the policy without knowing δ and r, a **state-action function** named **Q-function** is introduced. If the agent learns Q, then π^* can be computed without knowing the two functions. Q-function gives the expected utility of taking a given action in a given state (sometimes referred as quality of the action). In other words, Q is the expected return the agent will get if it takes action a at time t, given state x, and thereafter follows policy π. Q-function is defined as the current reward plus gamma times the value of the policy from the next state:

$$Q^\pi(x, a) = r(x, a) + \gamma V^*\big(\delta(x, a)\big)$$

In case of **stochastic models**, the definition is extended as a weighted average, where the weights are given by the probability to reach the next state:

$$Q^\pi(x, a) = \sum_{x'} p(x'|x, a)\,[r(x, a, x') + \gamma V^{\pi'}(x')]$$

In the **deterministic** case:

$$Q(x_t, a_t) = r(x_t, a_t) + \gamma \max_{a' \in A} Q\,(x_{t+1}, a')$$

Deterministic Q-learning ($\hat{Q}$ denote learner's current approximation to Q):

1. For each $\boldsymbol{x}, a$ initialize table entry $\hat{Q}_0(\boldsymbol{x}, \boldsymbol{a}) \leftarrow 0$
2. Observe current state $\boldsymbol{x}$
3. For each time $t = 1, \dots, T$ (until termination condition)
    - Choose and execute an action a
    - Observe the new state $\boldsymbol{x}'$ and collect the immediate reward $\bar{r}$
    - Update the table entry for $\hat{Q}(\boldsymbol{x}, \boldsymbol{a})$ and $\boldsymbol{x}$ with $\boldsymbol{x}'$

    $$\hat{Q}_t(\boldsymbol{x}, \boldsymbol{a}) \leftarrow \bar{r} + \gamma \max_{a' \in A} \hat{Q}_{t-1}\,(\boldsymbol{x}', a')$$
4. Find the optimal policy as the argmax of

$$\pi^* = \operatorname*{argmax}_{a \in A} \hat{Q}_T(\boldsymbol{x}, \boldsymbol{a})$$

With the Q-function, we are not using δ and r. We are just observing the new state $\boldsymbol{x}'$ and immediate reward $\bar{r}$ after the execution of the chosen action.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between **exploration** and **exploitation**. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future.

- Exploitation: select action a that maximizes $\hat{Q}(\boldsymbol{x}, \boldsymbol{a})$.
- Exploration: select random action a.

Actions can be chosen in various way.

In ***ϵ*-greedy strategy**, a random action with probability ϵ and the best action with probability $1-\epsilon$ are selected. In order to have *exploration* at the beginning and then *exploitation,* ϵ decrease over time. One problem with this strategy is that it treats all of the actions, apart from the best action, equivalently. If there are a few seemingly good actions and other actions that look much less promising, it may be more sensible to select among the good actions: putting more effort toward determining which of the promising actions is best, and less effort to explore the others.

The **soft-max strategy** consists in assigning higher probabilities to action with higher $\hat{Q}$, but every action has a non-zero probability. K determines how strongly the selection favors actions with high $\hat{Q}$. As before, typically k increase with time, in order to have exploration first and then exploitation.

$$P(a_i|x) = \frac{k^{\hat{Q}(x,a_i)}}{\sum_j k^{\hat{Q}(x,a_j)}}$$

Cap 18: Reinforcement Learning (non-deterministic)

When we are moving from deterministic to **non-deterministic** case the algorithm does not change, except for the update of the Q-table. In fact, the agent will choose an action with some strategy and then it will update the Q- function with a different formula (non-deterministic Q-learning).

Considering a Markov Decision Process $MDP = \langle \boldsymbol{X}, \boldsymbol{A}, \delta, r \rangle$, where:

- $\boldsymbol{X}$ is a finite set of states.
- $\boldsymbol{A}$ is a finite set of actions.
- $\delta = p(\boldsymbol{x}'|\boldsymbol{x}, a')$ is a probability distribution over transitions.
- $r(\boldsymbol{x}, a, \boldsymbol{x}')$ is a reward function.

Rewards depends on current state and successor state, which is not determinate because whenever an action is executed, we are not sure about successor state having only a probability distribution. Transition and rewards function are **non-deterministic.**

The **value function** in a non-deterministic case is defined as the expected value of the *cumulative discounted reward*:

$$V^{\pi}(x_1) = E[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots]$$

The **optimal policy** is:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^{\pi}(x) \ \forall x \in X$$

In the deterministic case Q is given by reward function plus gamma times "the best I can do in the future", while here Q is defined as the **expected value** of the same quantity. Q is given by the best I can do from any state with the probability of reaching that state:

$$Q(x,a) = E\left[r(x,a) + \gamma V^*\big(\delta(x,a)\big)\right] = E[r(\boldsymbol{x},a)] + \gamma \sum_{x'} p(\boldsymbol{x}'|\boldsymbol{x},a)\, V^*(x')$$

$$= E[r(\boldsymbol{x},a)] + \gamma \sum_{x'} p(\boldsymbol{x}'|\boldsymbol{x},a) \max_{a' \in A} \mathrm{Q}\,(\boldsymbol{x}', a')$$

The optimal policy is then:

$$\pi^*(\boldsymbol{x}) = \underset{a \in A}{\operatorname{argmax}}\, Q^*(\boldsymbol{x}, a)$$

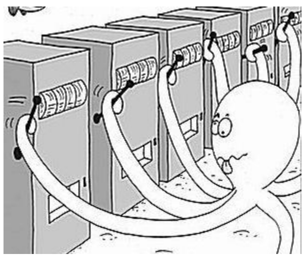
Non-deterministic Q-learning:

$$\hat{Q}_n(\boldsymbol{x},\boldsymbol{a}) = \hat{Q}_{n-1}(\boldsymbol{x},\boldsymbol{a}) + \alpha\left[r + \gamma \max_{a' \in A} \hat{Q}\,(\boldsymbol{x}', a') - \hat{Q}_{n-1}(\boldsymbol{x},\boldsymbol{a})\right]$$

With $\alpha(\boldsymbol{x},\boldsymbol{a}) = \alpha_{n-1}(\boldsymbol{x},\boldsymbol{a}) = \frac{1}{1+visits_{n-1}(\boldsymbol{x},\boldsymbol{a})}$

- $visits_n(\boldsymbol{x},\boldsymbol{a})$ is the total number of times state-action pair $(\boldsymbol{x},\boldsymbol{a})$ has been visited up to iteration n.
- $r + \gamma \max_{a' \in A} \hat{Q}\,(\boldsymbol{x}', a')$ is the **estimation** of what I can get: the actual reward plus gamma times the reward I could get in the future.
- $\hat{Q}_{n-1}(\boldsymbol{x},\boldsymbol{a})$ instead is what I believed before executing an action on x. The difference of the two is the error between my estimation and what I observe after the execution of the action.

Deterministic Q-learning does not converge in non-deterministic worlds! ($\hat{Q}_{n+1}(\boldsymbol{x},\boldsymbol{a}) \geq \hat{Q}_n(\boldsymbol{x},\boldsymbol{a})$ is not guaranteed).



An example of a One-State stochastic MDP is the **k-armed bandit problem**. The name "multi-armed bandits" comes from a scenario in which a gambler faces several slot machines ("one-armed bandits") that look identical at first but produce different expected winnings. We only have one state and pulling a different lever will bring a different reward. The goal is to find the action that maximizes the cumulative reward.

There are other algorithms for non-deterministic learning like **Temporal Difference $\boldsymbol{TD(\lambda)}$,** where instead of only looking of what happens in the next state, I can **look in more than one state**. Instead of computing one step time difference I compute n-step time difference. Sometimes TD converges faster than Q learning.

Another variant of Q-learning is **SARSA** that computes the best I can do with **current policy.** Instead of considering max of all possible actions on Q-table, I just choose an action (without using the max operator). A SARSA agent interacts with the environment and updates the policy based on actions taken. This is known as an *on-policy learning* algorithm because it evaluates the current policy.

$$\hat{Q}_n(\boldsymbol{x},\boldsymbol{a}) = \hat{Q}_{n-1}(\boldsymbol{x},\boldsymbol{a}) + \alpha\big[r + \gamma(\boldsymbol{x}',a') - \hat{Q}_{n-1}(\boldsymbol{x},\boldsymbol{a})\big]$$

This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent "S_1", the action the agent chooses "A_1", the reward "R" the agent gets for choosing this action, the state "S_2" that the agent enters after taking that action, and finally the next action "A_2" the agent chooses in its new state. SARSA is based on the tuple $\langle s, a, r, s', a'\rangle$(in our notation would be $\langle x, a, r, x', a'\rangle$).

It is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the magnitude of the utilities on the states involved doesn't need to be precise. This insight suggests an alternative way to find optimal policies, without estimating $V(x)$ or $Q(x,a)$. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π_0:

- *Policy evaluation E*: given a policy π_i, calculate the corresponding utilities of the states $U_i = U^{\pi_i}$, if π_i were to be executed.
- *Policy improvement I*: calculate a new optimal policy π_{i+1}, using one-step look-ahead based on U_i.

We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. Its size depends on the number of states (generally very high).

The policy is usually modeled with a parameterized function respect to ϑ. $\pi_\vartheta(a|\boldsymbol{x})$ will be a **parametric representation** of the policy function while $\rho(\vartheta)$ will be the policy value, the expected value of executing π_ϑ. The value of the reward (objective) function depends on this policy and then various algorithms can be applied to optimize ϑ for best reward.

Policy Gradient methods is a technique that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. A policy gradient algorithm computes the gradient of the policy in a local interval with respect to the current point and then move in order to improve value of this policy.

```
choose $\theta$
while termination condition do
    estimate $\Delta_\theta \rho(\theta)$ (through experiments)
    $\theta \leftarrow \theta + \eta \Delta_\theta \rho(\theta)$
end while
```

Cap 19: Hidden Markov Models

In a Hidden Markov Model (HMM) the system being modeled is assumed to be a Markov process with unobservable ("hidden") states.

We still assume the presence of a **Markov chain;** the current state contains all information needed to understand, but now we don't make any assumptions about the *fully observability* of the states.

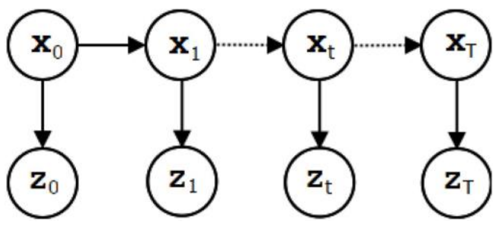
The agent cannot look into the states and cannot understand exactly the *current configuration*.

For each state we have a dependency with the observation. States x_t are discrete and **non-observable** and the observations z_t can be discrete or continuous. The evolution is not controlled by our system (there are no controls u_t). The interest in this model is to understand the state in which we are.

We want to solve the **state estimation problem**.

$HMM = \langle \boldsymbol{X}, \boldsymbol{Z}, \pi_0 \rangle$

- Transition model: $P(\boldsymbol{x_t}|\boldsymbol{x_{t-1}})$
- Observation model: $P(\boldsymbol{z_t}|\boldsymbol{x_t})$
- Initial distribution $\pi_0 = P(\boldsymbol{x}_0)$



We have set of **states** X, a set of **observations** Z, and an **initial probability distribution** for the initial state. Then we have a transition model that is a probability distribution that denote dynamic evolution of the system (like MDP but without actions). The observation model is the probability of an observation given a state.

When we consider a finite set of states, we can consider the transition model like a $n * n$ matrix called **transition matrix of the HMM** $\boldsymbol{A} = \{A_{ij}\}$:

$$A_{ij} \equiv P(\boldsymbol{x_t} = j|\boldsymbol{x_{t-1}} = i)$$

where n is the number of states and each component contains the probability distribution of moving from state i to state j.

Most of the solution of this problem are based on the solution of the **chain rule,** that says that the *joint probability* of a model is given by the product of all the terms computed considering probability of one random variable condition only to the direct parents:

$$P(\boldsymbol{x}_{0:\boldsymbol{T}}, \boldsymbol{z}_{0:\boldsymbol{T}}) = P(\boldsymbol{x}_0)P(\boldsymbol{z_0}|\boldsymbol{x}_0)P(\boldsymbol{x}_1|\boldsymbol{x}_0)P(\boldsymbol{z}_1|\boldsymbol{x}_1)P(\boldsymbol{x}_2|\boldsymbol{x}_1)P(\boldsymbol{z}_2|\boldsymbol{x}_2) \dots$$

Many related tasks ask about the probability of one or more of the **latent variables**, given the model's parameters and a sequence of observations $\boldsymbol{z}_{1:T}$.

- **Filtering** is the estimation of current state given all the observations we have so far.

  The task is to compute, given the model's parameters and a sequence of observations, the distribution over hidden states of the **last latent variable** $\boldsymbol{x}_T$ at the end of the sequence:

  $$P(\boldsymbol{x}_T = k|\boldsymbol{z}_{1:T}) = \frac{\alpha_T^k}{\sum_j \alpha_T^j}$$

  This task is normally used when the sequence of latent variables is thought of as the underlying states that a process moves through at a sequence of points of time, with corresponding observations at each point in time. We then want to know the **state of the process at the end**.

- **Smoothing** is the estimation of **past states given other observations**.

  This is similar to filtering but asks about the distribution of a latent variable somewhere in the middle of a sequence, i.e. to compute, for some $t < T$:

$$P(\boldsymbol{x}_t = k|\boldsymbol{z}_{1:T}) = \frac{\alpha_T^k \beta_T^k}{\sum_j \alpha_T^j \beta_T^j}$$

Alpha and *beta* are *forward* and *backward* steps in time.

In the HMM, we need to estimate *transition function* and *observation model*.

- **States can be observed at training time:**

$$A_{ij} = \frac{|\{i \to j\ transitions\}|}{|\{i \to *\ \ transitions\}|}$$

$$b_k(v) = \frac{|observe\ v \wedge state\ k|}{|observe * \wedge state\ k|}$$

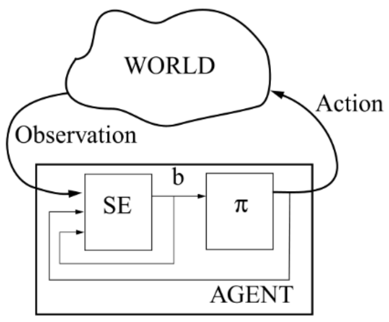
  Transition and observation model can be estimated with statistical analysis. We look the transition from i to j divided by the number of times we look to others transitions from i. b_k is the same but for the observation v.

- **States cannot be observed at training time:**

  The system is not observable but it's still possible to estimate parameters with methods based on **expectation maximization EM**, computing a local maximum likelihood.

Partially Observable Markov Decision Processes (POMDPs)

Consider the problem of a robot navigating in a large office building. The robot can move from hallway intersection to intersection and can make local observations of its world. Its actions are not completely reliable, however. Sometimes, when it intends to move, it stays where it is or goes too far; sometimes, when it intends to turn, it overshoots. It has similar problems with observation. Sometimes a corridor looks like a corner; sometimes a T-junction looks like an L-junction. How can such an error-plagued robot navigate, even given a map of the corridors? In general, the robot will have to remember something about its history of actions and observations and use this information, together with its knowledge of the underlying dynamics of the world (the map and other information), to maintain an estimate of its location. Given an uncertain estimate of its location, the robot has to decide what actions to take. In some cases, it might be sufficient to ignore its uncertainty and take actions that would be appropriate for the most likely location. In other cases, it might be better for the robot to take actions for the purpose of gathering information, such as searching for a landmark or reading signs on the wall. In general, it will take actions that fulfill both purposes simultaneously.

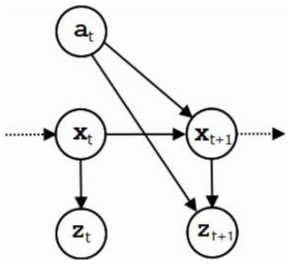


Problems like the one described above can be modeled as **partially observable Markov decision processes (POMDPs).**

One important facet of the POMDP approach is that there is no distinction drawn between actions taken to change the state of the world and actions taken to gain information. This is important because, in general, every action has both types of effect.

The agent chooses between actions based on the amount of information they provide, the amount of reward they produce, and how they change the state of the world.

We previously studied MDPs and HMMs. Each of these models made some simplification: **MDP** say that states are fully observable, **HMM** just avoid controlling system. If we want to put together all this so controlling the evolution of the system and estimate the state without making assumption on full observability, we can use a more realistic agent that combines MDP and HMM, called **POMDP**.



Now we have **actions** that controls the evolution of the state, but states are not *fully observable* so we can observe them through **observations**.

A partially observable Markov decision process can be described as a tuple:

$$POMDP = \langle \boldsymbol{X}, \boldsymbol{A}, \boldsymbol{Z}, \delta, r, o \rangle$$

- $\boldsymbol{X}, \boldsymbol{A}, \delta, r$ describe a Markov decision process.
  - $\boldsymbol{X}$ is a set of states
  - $\boldsymbol{A}$ is a set of actions
  - $\delta(\boldsymbol{x}', a, \boldsymbol{x}') = P(\boldsymbol{x}'|\boldsymbol{x}', a)$ is a *probability distribution over transitions*
  - $r(\mathbf{x}, a)$ is a reward function
- $\boldsymbol{Z}$ is a set of observations
- $o(\boldsymbol{x}', a, \boldsymbol{z}') = P(\boldsymbol{z}'|\boldsymbol{x}', a)$ is a *probability distribution over observations*.

It's a combination of items of the two models (**observation** over the state and **action**).

POMDP is the only model that can have actions used by the agent to get knowledge.

A solution is still the *policy*, but we can't use the definition used for the MDP. A policy in MDP is a m*apping* from states to actions, but here we only have an estimation of the state. Considering a policy that maps from history of observations to actions is not feasible as histories are too long. We can make a process from which we can compute an estimation of the state and use it as an input to our decision-making approach.

The **belief state** is the *probability distribution over the current state* ("where I think I am").

We let $b(\boldsymbol{x})$ denote the probability assigned to a world state x by belief state b.

The policy is now a mapping from a set of belief states $\boldsymbol{B}$ to a set of actions A, $\pi: \boldsymbol{B} \rightarrow A$

Given current belief state b, an action a, and an observation $\boldsymbol{z}$' observed after the execution of a we can compute the **next belief state**:

$$b'(\boldsymbol{x}') = \frac{P(\boldsymbol{z}'|\boldsymbol{x}', a)\ \sum_{x \in \boldsymbol{X}} P(\boldsymbol{x}'|b, a, \boldsymbol{x})\, P(\boldsymbol{x}|\boldsymbol{b}, \boldsymbol{a})}{P(\boldsymbol{z}'|b, a)} = \frac{o(\boldsymbol{x}', a, \boldsymbol{z}')\ \sum_{x \in \boldsymbol{X}} \delta(\boldsymbol{x}, a, \boldsymbol{x}')\, b(\boldsymbol{x})}{P(\boldsymbol{z}'|b, a)}$$

We can define the **value function** and from this we can define the **optimal policy** in terms of optimal value function:

$$V(b) = \max_{a \in \boldsymbol{A}} [\sum_{x \in \boldsymbol{X}} b(\boldsymbol{x}) r(\boldsymbol{x}, a) + \gamma \sum_{z \in \boldsymbol{Z}} P(\boldsymbol{z}|b, a) V(b_{\boldsymbol{z}}^{a})]$$

POMDP can be described as an MDP in the belief states, but belief states are infinite!

We need to **discretize** them. There are methods to approximate it with a linear function, or other methods that make a partition of the interval and for each interval they define a linear function. Discretization is reached solving multiple linear regression problems.

When observations are discrete, we can use a **policy tree**. Instead of *mapping* states, the *history* of observations can be represented in a *tree*. At each level we have an action and the set of all possible observation in the branches. We can go from the root to a leaf, each time choosing the action to make. At the end, the trace will be given by the path of actions and observations.