ML - NOTES

Introduction

Machine Learning follows a data-driven approach that differs from the one adopted before.
Before ML, the experts (programmer, engineer...) based the solution of a certain problem P on a well-defined algorithm A. An example of this workflow can be the "sort the elements of a list" problem.
What Machine Learning does is focusing the problem P on a dataset, a collection of data (of any type). Related to the dataset D a machine learning model is chosen and trained. An example of this is distinguishing cat images from dog images.
So machine learning (or data mining) is the task of producing knowledge from data.

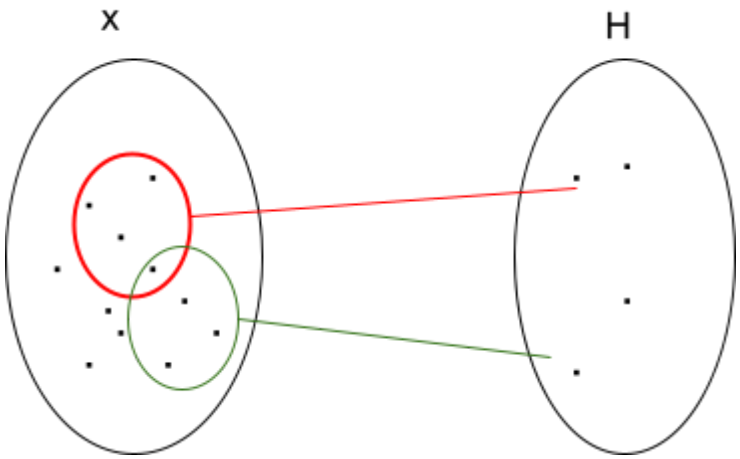
ML has been widely used in lots of applications during the recent years: computer vision, speech Interaction, robot control, finance (...). This is due to technology improvements (GPUs), the increased availability of data and the recent progress in algorithms and theory.
In fact, neural networks on which machine learning models are built, were already introduced in the '60s, but couldn’t be applied.

Learning means improving a performance P with some experience E. To know if P is improving, we need some measurements called metrics. We want to reach good performances and to do that we need (also) a huge amount of data.

A machine learning model works if the target function used to update parameters is computable. The real target function V is unknown, so we use an approximation V’ containing some parameters (weights) that the model will update in order to minimize an *error.* In its generalized formula, the error can be seen as the difference between the predicted label V’(b) and the real label V(b). Maybe this approximation V’ is not optimal, so at a certain step the value of the function given as input a sample b, V’(b), can be different to the real value (label) and so produce an error. To update weights some learning algorithms can be used (such as LSM).
The learning function is an *approximation* of the real function which outputs are as close as possible to the real outputs when the input doesn't belong to the dataset.
ML problems can be distinguished in categories, such as supervised learning, unsupervised learning, reinforcement learning …
Depending on the type of dataset we face a particular problem:

- **Supervised learning area:** When the dataset also contains labels. Both features (X) and labels (Y) can be discrete or continuous. When Y is discrete we are dealing with a classification task; when Y is continuous, we are dealing with a regression task.
- **Unsupervised learning area:** When the dataset doesn't contain labels. Most of the applications are clustering, pattern extraction, image compression, …
- **reinforcement learning area**: When the dataset is made up of traces of execution. Here the main goal is to learn a policy (so a state-action function). We only have sparse and time-delayed rewards. Applications: game playing, robotic tasks,...
- other

Concept learning is a binary classification problem with 2 possible outcomes (symbols). It takes part in a supervised learning area where the input space is the cartesian product of the attributes (features) describing the input and the possible outcomes are Y = {0,1}.
Concept learning is defined as $C: X \rightarrow \{0, 1\}$, where C denotes the target function, unknown. So we want to approximate it with an hypothesis h* from H.
We assume now that we have a perfect knowledge about the dataset, so there is no noise.
Given D={$(x_{i,}, c(x_i))$}, $c(x_i)$ represents the real label of a sample x_i belonging to D.
The hypothesis space H contains all the possible functions we should use to predict the outcome in our model, so all the possible approximations. Each hypothesis in H is a set of *conjunctions of constraints*, as many as the attributes are.
Each constraint can be ? (Any value is accepted) or v_i (a fixed value, belonging to the set of values of the input space). Ex: <v1,...,?,...vn>. When h={?,...,?}, h(x) returns always 1 for any x.
But in general, **h(x)=1 if all the constraints in h are valid with respect to x**, else it would be 0. In concept learning every hypothesis is associated to a set of instances (all the instances that are classified as positive by such hypothesis).



In order to find the best hypothesis, so the best approximation of C, given the dataset D, we need to:

- define H
- define a sort of measurement
- define an appropriate algorithm

An hypothesis is consistent w.r.t. D if:

$h(x_i) = c(x_i)$ for each i={0,...,N} and xi in D. (N is the dimension of the dataset)

The real goal of a machine learning system is to find the best hypothesis h
that predicts correct values of h(x') for instances x' ∉ D with respect to the unknown values c(x').

The performance measure of h consists in evaluating $h(x_i) = c(x_i)$ for each i={0,...,N} and xi in D.

Inductive learning hypothesis:
Any hypothesis that approximates the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

In concept learning every hypothesis is associated to a set of instances.

Definition of version space:
We can now define the version space $VS_{H,D}$ as a subset of the hypothesis space H containing all the consistent hypothesis w.r.t. D. VS can be empty and can be computed with some known algorithms (e.g. list-then-eliminate algorithm).

An hypothesis is specific if:
It is the best hypothesis to pick from VS.

We can also think about another hypothesis space H' which is formed by the *disjunction* of all hypotheses in H. So H' space is *more complex* and has a higher representational power, since it can represent more solutions.
In general, **ML doesn't require an hypothesis space which is too complex just because it can lose generalization**, it may not predict values for each input in the input space. This situation occurs when the solution (the learnt function) is too powerful w.r.t the dataset, but not with other points. This is overfitting.
It can also happen that we pick a hypothesis which is too simple and can not generalize well on other data points. This is underfitting. So we have to find the best parameters able to overcome underfitting and stop before overfitting comes.

Formally, ML prefers h w.r.t h' because:
Given two hypothesis h belonging to H, h' belonging to H, and a point x not in the dataset, it is more likely that

$$|h(x)-t| < |h'(x)-t|$$

Dataset may contain noisy data, so some errors in collecting labels yi that are different from the real ones c(xi).

Classification Evaluation

Performance evaluation in classification is based on accuracy or on error rate.
We want to **measure** the performances on unseen data, the points excluded from the dataset.
A measurement of a performance is good when the estimator, the function we want to learn, is good.
To understand when an estimator is good, statistical analysis is crucial.
As we've already said, the solutions depend on data and data depend on the subset S of points sampled from the input space which depends on the dataset. **So the solutions depend on distributions of data.**
It's important to know that *probability distributions over features in X are NOT uniform*. This means that if I take two subsets S and S' sampled from the same dataset D, then I will have different solutions and results for S and S'.
In order to evaluate performances, an error measurement is needed.
We define as:

- **"true" error:** probability that h will misclassify an instance drawn at random according to D.

$$error_D(h) = Pr_{x \in D}[f(x) != h(x)]$$

We use an approximation of this error, defined as:

- **"sample" error:** proportion of examples in S that h misclassifies.

$$error_S(h) = 1/n \sum_{x \epsilon S} \delta(f(x) != h(x))$$

The **accuracy** can be defined as:

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

If the accuracy on S is high but the one on D is poor, then our model is not learning good.
S is a subset of training data, so the approximation of the error doesn't include all the distribution of data. (But in this way we can compute the error).
S is also obtained from a sampling process, so it depends on it.
Since we're dealing with random variables, then we should use the expected value of the error sample i.e. the weighted average of the error sample over all the possible samples.
This means we should also compute all the possible subsets S of the input space: intractable computation.

Let's define the **estimation bias** as the difference between the expected value of the sample error oh h and the true error of h. *We want this factor close to 0*.
The condition for an estimator to be unbiased is that S, the subset of the dataset D, and h, the estimator, are chosen independently. This happens if we don't use S to compute h.
Also an *analytic method* can be used to understand the true error and the error sample: the confidence intervals. This technique, not often used in ML, defines the interval where the true error lies given the error sample. Its computation is:

$$error_s(h) \mathrel{+}= z_N \sqrt{error_s(h)(1 - error_s(h))/n}$$

z_N is a known factor proportional to the probability of having the true error in that interval.

The procedure to guarantee h and s independent is the following:

1. start from the dataset D
2. Split it into training and testing randomically such that the probability distribution of D is maintained both on training and test sets
3. compute h on the training set
4. evaluate h through the sample error

Trade-off between training and test: typically ⅔ of samples in D will compose to the training set and the remaining ⅓ samples go to the test set.
This choice is made to have a tradeoff among these situations:

- more samples for training few for test -> better model but inaccurate evaluation
- few samples for training more for test -> weak model but accurate evaluation

Let's now define **K-Fold Cross Validation.** It is a technique widely used in ML consisting in 3 main steps:

1. Take the dataset D and split into K disjoint subsets
2. for i=1,...,k do
   - $T_i \leftarrow \{D - S_i\}$
   - $h_i \leftarrow L(T_i)$
   - $\delta_i \leftarrow error_{S_i}(h_i)$
3. return $error = 1/k \sum_{i=1,..,k} \delta_i$

Typically, the higher K is, the better the accuracy.

Note: since K-Fold Cross Validation uses a sampling process, if it is repeated two, or more, times it will produce different results.

Overfitting, formally:
h from H overfits training data if there exists an alternative h' from the same hypothesis space H such that:

$$error_s(h) < error_s(h') \text{ AND } error_D(h) > error_D(h')$$

h' is preferred because it generalizes better on unseen data (D)

Conclusion: avoid solutions that specializes itself on training

Compare two hypothesis h1, h2:
In order to compare two hypothesis h1, h2, we can use the approximations of true errors, defining this metric:

$$d = error_{s_1}(h_1) - error_{s_2}(h_2)$$

it is still valid if s1 = s2 = s

Compare two learning algorithms L1, L2:
In order to compare two learning algorithms L1 and L2 we use:

$$E_S[error_s(L_A(T)) - error_s(L_B(T))]$$

This measure can be approximated again with K-Fold Cross Validation, using:

4. Take the dataset D and split into K disjoint subsets
5. for i=1,...,k do
   - $T_i \leftarrow \{D - S_i\}$
   - $h_1 \leftarrow L_1(T_i),\ h_2 \leftarrow L_2(T_i)$
   - $\delta_i \leftarrow error_{S_i}(h_1) - error_{S_i}(h_2)$
6. return error = $1/k \sum_{i=1,\dots,k} \delta_i$

Pick h1 if δ < 0, h2 otherwise.

In some cases, accuracy is not enough to evaluate a classification method, namely when the dataset is unbalanced.
In classification we use other metrics to evaluate performances, such as:

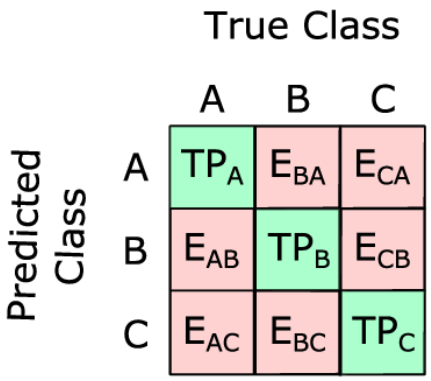
- **TP** positive samples predicted positive
- **TN** negative samples predicted negative
- **FP** negative samples predicted positive
- **FN** positive samples predicted negative
- **error rate** = (FP+FN)/(TP+TN+FP+FN)
- **accuracy** = 1-error
- **recall** = TP/(TP+FN) ability to avoid FN
- **precision** = TN/(TN+FP) ability to avoid FP
- **F1-score** = 2((Pr*Rec)/(Pr+Rec))

These metrics are domain-dependent.

The confusion matrix is used to represent in a matrix form TP, FP, TN, FN

TP	FP
FN	TN

When we have multiple classes, the confusion matrix is extended and the main diagonal represents the correct classifications. An example is:



Often heat maps are used.
In this case:

- True Positives (TP): Diagonal elements from the top-left to bottom-right represent the number of instances where the true class and the predicted class match.
- True Negatives (TN): Elements outside the main diagonal in the bottom right corner. In a multi-class problem, these are all the instances that are correctly predicted as not belonging to the class of interest for each class.
- False Positives (FP): Elements in the columns but not on the diagonal. These are instances where the model incorrectly predicted the class as belonging to the class of interest when it does not.
- False Negatives (FN): Elements in the rows but not on the diagonal. These are instances where the model incorrectly predicted a different class when it should have predicted the class of interest.

In regression tasks, the main metrics used are:

- mean absolute error: 1/n (∑i |h(xi)-ti|)
- mean squared error: 1/n (∑i (h(xi)-ti)^2)
- root mean squared error: √(mse)
- percentage error: |f(xi)-ti|/ti

The choice depends on data and applications.

Also K-Fold Cross Validation can be used in the same way, using the steps above.

Decision Trees (concept learning as search)

Intro

During the course we will discuss models according to the type of H. Let's distinguish models in:

- ad hoc models: decision trees,...
- parametric models
- non parametric models

When dealing with decision trees *we assume* that:

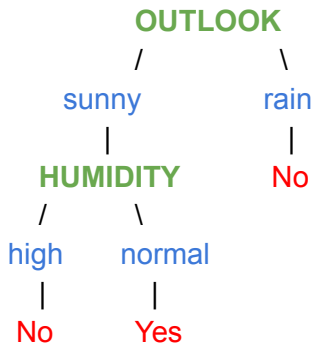
- the input space is the cartesian product of attributes A1x…xAm
- the task is a classification problem f:X->C

Decision trees

The decision tree contains:

- a **root node** containing the "best" attribute
- **internal nodes** containing other attributes
- **edges** (branches of each internal node) consisting of the values corresponding to that internal node (so that particular attribute) from where the edge starts
- **leaves** containing the values of classification classes

Ex. PlayTennis



Red = class
Green = attributes
Blue = values of attributes

Logic Formula of a DT

A tree can be transformed into a logical formula consisting of the disjunctions of conjunctions of paths leading to a particular class.
In the example above, the formula leading to the "NO" leaf is:
[(OUTLOOK AND rain) or (OUTLOOK AND sunny AND high)]
To get the formula of Yes leaf, you just can put a NOT before the formula leading to No (and vice versa).

Decision trees are useful since they can answer the question "why the model produces that particular estimator?". The answer provides the resulting path to a leaf node.

ID3 Algorithm
A very common algorithm used to compute a decision tree is ID3. It consists of 5 main steps and its structure is recursive.
The **input of ID3** are examples, target_attribute, Attributes
The **output of ID3** is the decision tree
The steps are:

1) Create a root node for the tree
2) if all Examples are positive, then return the root with label +
3) if all Examples are negative, then return the root with label -
4) if attributes is empty, then return the root with label = most common value of target_attribute in examples
5) otherwise:
   A <- the “best” attribute for Examples
   Assign A as decision attribute for root
   For each value v_i of A:
      add a new branch from root corresponding to test A = v_i
      examples v_i = subset of examples having v_i as value for that attribute A
      if examples v_i is empty then add a leaf Node with label = most common value of target_attributes
      else add the tree ID3(examples v_i, target_attributes, attributes-{A})

So we can define this model as a *statistical algorithm* since it uses the “most common” target attributes in case the list is empty, so it is robust to noisy data.

Good Properties of ID3:

- hypothesis space is **complete**
- one solution provided which is the **best locally** (search bias): we can’t determine how many DTs are consistent
- **doesn’t use back tracking**
- Statistically-based search choices: **robust to noisy data**
- uses all the training examples at each step: **not incremental**

Bad properties of DT in general:

- determine how deeply to grow the DT
- handling continuous attributes
- choosing appropriate attribute solution measures
- handling training data with missing attributes values
- handling attributes with different costs

How to choose the “best” attribute?
The tree shape depends on this choice. So it is crucial.
In general, **we should use the attribute allowing us to split the tree such that each partition contains all positive or all negative examples.**

To do this in formulas, we use a concept related to the Entropy of the dataset.
We define as E(S) the entropy of the set S as measure of impurity of S and it is equal to:

$$E(S) = - p_+ \log_2(p_+) - p_- \log_2(p_-)$$

where p_+ and p_- are portions of the positive and negative examples.

A balanced dataset, containing the same number of positive and negative examples has the maximum value of entropy (1).
So we want to minimize the entropy by maximizing the **information gain,** defined as:

$$\text{Gain(S,A)} = E(s) - \sum_{v \in Values(A)} |S_v|/|S| \; E(S_v)$$

When adding a new example, the space of solutions, so the space of trees, increases.

Overfitting in DT
One common issue with decision trees is Overfitting. It happens mostly when the size of the tree increases. So we need to stop before having overfitting doing two things:

- stop growing data -> determine the correct tree size in advance
- grow full tree then post prune

To determine the correct tree size

- use a separate set of examples (distinct from the training examples) to evaluate the utility of post-pruning
- apply a statistical test to estimate accuracy of a tree on the entire data distribution
- using an explicit measure of the complexity for encoding the examples and the decision trees.

In the **reduced-error pruning technique**, we use the validation set to check if the performances are increasing, so if there might be occurring overfitting.
Pruning is a method doing a local search to prevent overfitting and can be done in two ways:

- removing sub trees and assign the most common classification starting by those who produce better validation set accuracy
- reducing the set of samples (but when data set is limited it can give bad results)

Else, we can use **post pruning**, which rules are:

- infer the decision tree allowing for overfitting
- convert the learned tree into an equivalent set of rules
- prune each rule independently of others
- sort final rules into desired sequence to use

This method is more efficient, since it maps in a unidirectional way the decision tree to the space of rules. So it's expanding the representational power of solutions.

Other types of DT:
We can extend decision trees also to continuous attributes, or multi-valued attributes, or unknown attributes values.
Also we can add a cost measure of a certain attribute in order to choose the path best.

Applications: The model using decision trees is random forest which is less sensitive to overfitting.

Recap of overfitting in DT
As said, decision trees are sensitive to overfitting. So the produced estimator h can lead to overfitting, this means that there exists another hypothesis h' such that:

$$acc_T(h) > acc_T(h') \; AND \; acc_S(h) < acc_S(h')$$

To prevent this issue we can use:

- **pruning**: we change h with another h' from the same hypothesis space H. We have to split the dataset into training and validation. This is done inside the learner algorithm.

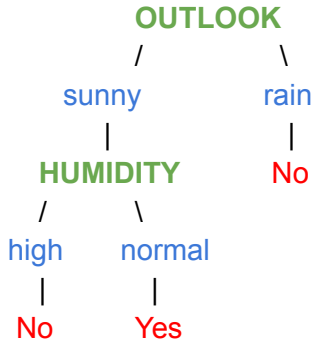
- **rule space**: we change h with h' from another hypothesis space H', which is more powerful w.r.t. H, but the local search on H' allows us to find a more general hypothesis.

If the dataset D contains no noise, ID3 is able to return a solution h with accuracy on the training set = 1, and accuracy on the test set < 1.
If D has some noise, ID3 is capable of building the tree but ACC is < 1 both on training and test set.
When ACC(h) on training set T is 1, then h is consistent with T.

How to check consistency in decision trees? Build the corresponding table of the tree, with as many columns as the attributes (also including the real label column and predicted label column) and fill it with values corresponding to those attributes of paths. If all the elements in the column of real labels and those in the column of predicted labels are the same, then h is consistent with respect to D.

In the case of the dataset D defined with the following samples:
x1 = <Outlook: sunny, Humidity: high> y1 = <N>
x2 = <Outlook: rain> y2 = <N>
x3 = <Outlook: sunny, Humidity: normal> y3 = <Y>

And considering the following solution:



We would have this table:

OUTLOOK	HUMIDITY	Real Label	Predicted Label
sunny	high	N	N
rain	?	N	N
sunny	normal	Y	Y

In this case h is consistent with D.
In the second row humidity could assume any value, it wouldn't affect the results.

Probability (and Bayes Network)

Probability measures in an explicit way the *uncertainty*, a phenomenon occurring when events are not fixed and can influence an action to happen.
We don't use logic formulas since they provide a discrete measure (0/1), instead probabilities use numbers from 0 to 1.

Some definitions:

- The **Sample space** Ω is the set containing all possibilities of an event (intractable).
- A **point ω** belonging to this space is called a sample point.
- **Probability** is a function mapping from the state space to real values such that these values are between 0 and 1 and the sum of the probability over all the sample points is equal to 1.
- **Random process** is a process with different outcomes each time it's done.
- A is an **event** so a subset of the state space.
- A **random variable** is both a variable and a function mapping from the sample space to another space. Instead of considering the sample space, we use this random variable X to return a random value of the sample space.

Some formulas:

$$P(X = x_i) = \sum_{\{\omega \in \Omega | X(\omega) = x_i\}} P(\omega)$$

event a ≡ A = true ≡ {ω ∈ Ω|A(ω) = true}
event ¬a ≡ A = false ≡ {ω ∈ Ω|A(ω) = false}
event a ∧ b = points ω where A(ω) = true and B(ω) = true
event ¬a ∨ b = points ω where A(ω) = false or B(ω) = true

- **Prior or unconditional probabilities** of propositions correspond to belief prior to the arrival of any (new) evidence.

- A **probability distribution** is a function assigning a probability value to all possible assignments of a random variable.

- **Joint probability distribution** for a set of random variables gives the probability of every atomic joint event on those random variables (i.e., every sample point in the joint sample space).

- **Conditional/posterior probability**: Belief after the arrival of some evidence.

- **Conditional probability distributions**: representation of all the values of conditional probabilities of random variables.

Definition of **conditional probability**:

$$P(a|b) \equiv P(a \wedge b)/P(b) \text{ if } P(b) \neq 0$$

Denominator can be viewed as a normalization constant α

Product rule

$$P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$$

Chain rule is derived by successive application of product rule:

$$P(X1, X2) = P(X1)P(X2|X1)$$

A and B are **independent** iff

$$P(A|B) = P(A) \text{ or } P(B|A) = P(B) \text{ or}$$
$$P(A, B) = P(A)P(B)$$

X **conditionally independent** from Y given Z iff

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

$$P(Y1, \ldots , Yn|Z) = P(Y1|Y2, \ldots Yn, Z)P(Y2|Y3 \ldots Yn, Z) \cdots P(Yn|Z)$$

Yi conditionally independent from Yj given Z:

$$P(Y1, \ldots , Yn|Z) = P(Y1|Z)P(Y2|Z) \cdots P(Yn|Z)$$

Chain rule + Conditional independence

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

Bayes' rule $P(a|b) = P(b|a)P(a)/P(b)$ or

$$P(Z|Y1, \ldots , Yn) = \alpha P(Y1, \ldots , Yn|Z) P(Z)$$

Yi, . . . Yn conditionally independent each other given Z

$$P(Z|Y1, \ldots , Yn) =$$
$$\alpha P(Y1|Z) \cdots P(Yn|Z)P(Z)$$

Bayesian Learning

Bayesian Learning is a class of methods modeling the problem as Bayesian optimization.

Classification as Probabilistic estimation

We need a definition of classification as a *probabilistic estimation*:

Given f:X->V, target function, a dataset D, a new instance x' not belonging to D, the best prediction is f'(x') = v* where v* = $argmax_v P(v|x', D)$

The argmax function returns v* which are the values in V (labels) allowing the argument P(v|x',D) reaching the maximum. This operation is invariant to scale. So, in this formulation we're returning labels.

In a more general formulation we only have: P(V|x',D) which provides the probability distribution over V (more information).

Learning as Probabilistic estimation

Let's introduce the hypothesis space.
Given a dataset D and hypothesis space H, compute a probability distribution over H given D, P(H,D):

P(h|D) = P(D|h)P(h) / P(D) [Bayes]

where:

- P(h|D) is the posterior probability
- P(D|h) is the likelihood measuring the probability of having D given a solution h
- P(h) is the prior probability
- P(D) is a normalization factor

We want the best h so the one maximizing P(h|D). This means computing:

h_{MAP}=$argmax_h$ P(D|h)P(h) / P(D)

= $argmax_h$ P(D|h)P(h) *since argmax is scale invariant

If we also assume that the hypothesis are identically distributed, then we arrive to the maximum likelihood hypothesis:

h_{ML} = $argmax_h$ P(D|h) * since P(h) becomes a constant

How to compute it?
Different approaches:

- **Brute force** -> intractable
- **Most Probable Classification of New Instances** -> not accurate; given a new instance x', h(x') may not be the most probable classification
  Ex.
  P(h1|D) = 0.4, P(h2|D) = 0.3, P(h3|D) = 0.3
  Given a new instance x', h1(x') = +, h2(x') = -, h3(x') = -
  So, it would return + since it gives the highest value of probability.
- **Bayes Optimal Classifier** -> Optimal learner: no other classification method using the same hypothesis space and same prior knowledge can outperform this method on average. It maximizes the probability that the new instance x is classified correctly,

Bayes Optimal Classifier
Consider target function f : X → V, V = {v1, ..., vk}, data set D and a new instance x ∉ D:

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

where P(vj |x, hi) is the probability that $h_i(x) = v_j$ is independent from D given hi.

So, P(vj |x, D) is equal to the weighted average of all predictions of all the hypotheses.

The result is V_ob = $argmax_{v_j}$ P(vj | x, D) = $argmax_{v_j} \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$

In the example of before:
P(h1|D) = 0.4, P(h2|D) = 0.3, P(h3|D) = 0.3
Given a new instance x', h1(x') = +, h2(x') = -, h3(x') = -

This technique would have compute:

$\sum_{h_i \in H} P(+ \mid x, h_i) P(h_i \mid D) = 0.4$

$\sum_{h_i \in H} P(- \mid x, h_i) P(h_i \mid D) = 0.6$

So the $v_{OB} =$ − since it's the label which maximizes the argmax.

P(vj |x, D) includes a sum over all the hypotheses in H, then its computation is intractable. Also, v_{OB} provides a more powerful representation wrt the hypothesis, because v_{OB} might not be represented in H.

We compute the posterior probability considering the prior probability of each hypothesis (given) and the likelihood of a value given the hypothesis.

(Examples)

1. I have 5 bags of candies:

- the first one has P(h1) = 0.1 and has 100% cherry, 0% lime
- the second one P(h2) = 0.2 and has 75% cherry, 25% lime
- the third one P(h3) = 0.4 and has 50% cherry, 50% lime
- the fourth one P(h4) = 0.2 and has 25% cherry, 75% lime
- the fifth one P(h5) = 0.1 and has 100% lime, 0% cherry

So we have a symmetric scheme.
The prior probability distribution can be described as a vector containing all the prior probabilities of each hypothesis: < P(h1), …, P(h5)> = <0.1, 0.2, 0.4, 0.2, 0.1>
The likelihood of the lime candy (so the likelihood of extracting a lime given a certain hypothesis) is again a vector containing the likelihood wrt each hypothesis:
<0, 0.25, 0.5, 0.75, 1>
Same for the likelihood of cherry candy: <1, 0.75, 0.5, 0.25, 0>

Suppose the following experiment:

1. First candy is lime: D1 = {l}
   P(hi |{d1}) = αP({d1}|hi)P(hi) = α < 0, 0.25, 0.5, 0.75, 1 > · < 0.1, 0.2, 0.4, 0.2, 0.1 >
   = α< 0, 0.05, 0.2, 0.15, 0.1 > = < 0, 0.1, 0.4, 0.3, 0.2 >
2. Second candy is lime D1 = {l,l}: we exploit the independency
   P(hi |{d1, d2}) = αP({d1, d2}|hi)P(hi) = αP({d2}|hi) **P({d1}|hi)P(hi) known**
   = α < 0, 0.25, 0.5, 0.75, 1 > · **< 0, 0.1, 0.4, 0.3, 0.2 > =**
   = < 0, 0.038, 0.308, 0.346, 0.308 >

where α = 1/(0.1+0.2+0.4+0.2+0.1)
And so on.

2. Consider H continuous space, D = {c cherries, l lime}, N = c + l, and P(c|hθ) = θ
   P(l|hθ) = 1 − θ
h_ML = argmax_hθ P(D|hθ) = argmax_hθ L(D|hθ),
so we use Logaritm (L(D|hθ) = log P(D|hθ))
L(D|hθ) = c log θ + l log(1 − θ)
We can solve it analytically by putting dL/dθ = 0
θ = c/N

General approach

We typically use logarithmic space, so logarithm of probabilities in order to exploit some properties of it. Also, the argmax operation of a monotonic function (logarithm) applied to another function (probability distribution) is equal to the argmax on this other function (probability distribution). So logarithms don't change the results.

Given dataset D = {di} with di ∈ {0, 1}, assuming a probability distribution P(di ; Θ), the maximum likelihood estimation is:

$$\Theta\_ML = argmax_{\Theta} \log P(di \mid \Theta)$$

We should rewrite P in terms of known distribution (Bernoulli, Binomial, …) to directly arrive at the solution.

Naive Bayes Classifier

The assumption of Naive Bayes Classifier is that X is conditionally independent of Y given Z, so:

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

It is a strong assumption (-) but reduces the space of hypothesis (+).

Now, we want to find f:X->V, where each x is described by attributes a1,...,an.
To do this we want to compute:

$$(1)\quad argmax_{v_j} P(vj \mid x, D) = argmax_{v_j} P(vj \mid a1, \ldots, an, D)$$

By doing this *we remove the sum over all the possible hypotheses* which is an intractable computation.
The result of (1) is the value vj thanks to which the argument is maximized and it is:

$$v_{MAP} = argmax_{v_j} P(a1,...,an|vj,D)P(vj|D) / P(a1,...,an|D)$$

$$v_{MAP} = argmax_{v_j} P(a1,...,an|vj,D)P(vj|D)$$

*denominator is a factor, can be removed

Now let's assume $P(a1,...,an|vj,D) = \Pi\ P(ai|vj,D)$, so we're assuming a1,...,an to be independent wrt vj and D. IN GENERAL IT IS NOT TRUE. So we're making an approximation -> we're not guaranteed that the solution gives the optimal.

With this assumption the result of (1) is:

$$V_{NB} = argmax_{v_j} P(vj \mid D)\ \Pi i\ P(ai \mid vj, D)$$

The right term is also easy to compute -> efficiency

Given D, we can estimate $\Pi i\ P(ai \mid vj, D)$, with the following algorithm:

Naive Bayes Learn(A, V, D)

```
for each target value vj ∈ V
    Pˆ(vj|D) ← estimate P(vj|D)
    for each attribute Ak
        for each attribute value ai ∈ Ak
            Pˆ(ai|vj, D) ← estimate P(ai|vj, D)
```

The estimation can be done with some **statistical analysis**:

1) Pˆ(vj|D) = |{<..., vj}| / |{D}|
2) Pˆ(ai|vj, D) = |{..., ai,...,vj}| / |{..., vj}|

In big datasets these are actually good approximations.
The issue is that these two terms can go to zero when (1) no training instances have vj, (2) no training instances have ai, vj. In order to avoid 0, we add some scalar factor, m and p where p is the prior estimate and m is the weight of p.
In this way, 1) and 2) become:

1) Pˆ(vj|D) = |{<..., vj}| + **mp** / |{D}|
2) Pˆ(ai|vj, D) = |{..., ai,...,vj}| + **mp** / |{..., vj}| + **p**

Now, to classify a new instance we will use:

$$V_{NB} = argmax_{v_j} \text{Pˆ(vj|D) } \Pi i \text{ Pˆ(ai|vj, D)}$$

Example: PlayTennis
Consider a new instance <Outlook = sun, Temp = cool, Humid = high, Wind = Strong>
P(PlayTennis = Yes) = P(Y) = 9/14, -> it means that 9 samples over 14 have vj = Yes (so they are positive samples)
P(PlayTennis = No) = P(n) = 5/14 -> it means that 5 samples over 14 have vj = No (so they are negative samples)
P(Wind = Strong | Y) = 3/9 -> 3 of out 9 samples have wind = strong as attribute and vj = Y
P(wind = strong | N) = ⅜ -> 3 of out 5 samples have wind = strong as attribute and vj = N

So we can iterate and compute the result. Since there are only two values for v, then:

1) vj = Y -> P(Y)P(sun|Y)P(cool|Y)P(high|Y)P(strong|Y) = 0.005
2) vj = N -> P(N)P(sun|N)P(cool|N)P(high|N)P(strong|N) = 0.021

So vj = N is the result.
The most important thing to consider here is not to have exactly
P(a1,...,an|vj,D) = Πi P(ai|vj,D), but to have the right value returned by the argmax.
If I know that the right Vj is N and the argmax returns v1 = Y = 0.12 and v2 = N = 0.11, so they are good approximations, but according to these values I'd choose v1 = Y, then I get wrong.
Thus, I should prefer far approximations (e.g. v1 = 0.98 and v2 = 0.999) but leading to right result (v2 = Y).

Learning to classify texts

Input: set of documents (sequences of words) MyDocs ⊂ Docs, each classified as c1, . . . , ck.
Dataset = {<doc, c>i}
We want to learn a target function f : Docs→ {c1, . . . , ck}
Given a new document doc !∈ D, compute:

$$c_{NB} = argmax_{c_j \in C} \text{P(cj|D)P(doc|cj, D)}$$

Using the naive bayes assumption, this formula becomes:

$$c_{NB} = argmax_{c_j \in C} \text{ P(cj|D) } \Pi i \text{ P(pi = wi | cj, D), for i=1,...m (length)}$$

We have also to deal with:

1) the position of the words: we assume it doesn't matter so that we can only compute:

$$c_{NB} = argmax_{c_j \in C} P(cj|D)$$

2) the variable length of the words: we use an embedding system of the document (bag of words representation) consisting of expressing the vocabulary V = {w1,...,wm} in a feature vector of fixed size. We can use different approaches:
   a) boolean feature vector
   b) ordinal feature vector

In case 2.a) we apply:

Estimate $\hat{P}(c_j)$ and $\hat{P}(w_i|c_j)$ using *Bernoulli distribution*.

LEARN_NAIVE_BAYES_TEXT_BE(D, C)
- $V \leftarrow$ all distinct words in D
- for each target value $c_j \in C$ do
  - $docs_j \leftarrow$ subset of D for which the target value is c_j
  - $t_j \leftarrow |docs_j|$: total number of documents in c_j
  - $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
  - for each word w_i in V do
    - $t_{i,j} \leftarrow$ number of documents in c_j containing word w_i
    - $\hat{P}(w_i|c_j) \leftarrow \frac{t_{i,j}+1}{t_j+2}$

In case 2.b) we apply:

Estimate $\hat{P}(c_j)$ and $\hat{P}(w_i|c_j)$ using *Multinomial distribution*.

LEARN_NAIVE_BAYES_TEXT_MU(D, C)
- $V \leftarrow$ all distinct words in D
- for each target value $c_j \in C$ do
  - $docs_j \leftarrow$ subset of D for which the target value is c_j
  - $t_j \leftarrow |docs_j|$: total number of documents in c_j
  - $\hat{P}(c_j) \leftarrow \frac{t_j}{|D|}$
  - $TF_j \leftarrow$ total number of words in $docs_j$ (counting duplicates)
  - for each word w_i in V do
    - $TF_{i,j} \leftarrow$ total number of times word w_i occurs in $docs_j$
    - $\hat{P}(w_i|c_j) \leftarrow \frac{TF_{i,j}+1}{TF_j+|V|}$

This model is used in even difficult applications such as spam classification, and though it uses some strict and strong assumptions it provides good results and efficiency in terms of resources.

Probabilistic Models for classification

input space: continuous
probability functions: continuous

The problem formulation is:
Given f:X->C, (X ⊆ R^d), D = {(xi,ci)i=1,...,N} and x ∉ D, estimate P(ci|X,D).
We can "remove" D from this probability and use the notation:
P(ci | x): posterior probability
P(x | ci): class conditional densities

We can distinguish between two approaches, leading to different results:

- **generative**: P(x | ci) is estimated and we can use it to generate new samples
- **discriminative**: P(ci | x) is estimated directly

Let's consider the first one.

Probabilistic Generative Models

Consider two classes: c1 and c2.

P(c1 | x) = [P(x|c1)P(c1)]/P(x) *for bayes*

= [P(x | c1) P(c1) / [P(x|c1)P(c1) + P(x|c2)P(c2)] *for total probability theorem* =

…

= 1/[1+σ(a)],

where a = ln [P(x|c1)P(c1)]/[P(x|c2)P(c2)]. σ is a sigmoid function, also monotonic.

NOW, *assume* P(x|c1) follows the Gaussian distribution N(x; μi, Σ), where μi is the mean of the gaussian for the class i and Σ is the covariance matrix (equal for each class).

P(x|c1) = N(x; μ1, Σ)
P(x|c2) = N(x; μ2, Σ)

So, a = ln N(x; μ1, Σ)P(c1) / N(x; μ2, Σ)P(c2) which can be rewritten as:
a = … = w^Tx + w0. So we have a linear equation in x.

Where:
w = $\Sigma^{-1}(\mu 1 - \mu 2)$
w0 = - 1/2 $(\mu_1{}^T \Sigma^{-1} \mu_1) + 1/2\,(\mu_2{}^T \Sigma^{-1} \mu_2) + ln\,[P(c1)/P(c2)]$

-> P(c1|x) = σ(w^Tx + w0)
-> P(c2|x) = 1-P(c1|x) we use the previous result to get a new sample

To solve this problem in terms of a parametric models, we define finally:
P(c1) = π
P(c2) = 1-π
P(x|c1) = N(x; μ1, Σ)
P(x|c2) = N(x; μ2, Σ)

So estimating **μ1, μ2, Σ, π** means solving this problem and to do this we use the dataset.

Solution of the parametric model
D = $\{(x_n, t_n)_{i=1,\dots,N}\}$
We define t_n = 1 if x_n belongs to the c1 class, else t_n = 0.

If the number of samples belonging to class c1 is called N1 and the number of samples belonging to class c2 is called N2, then N = N1+N2.

The method we use to solve the parametric model described above we use the **maximum likelihood solution:**

$P(t|\pi, \mu1, \mu2, \Sigma, D) = \Pi\ [\pi\ N(xn; \mu1, \Sigma)]^{t_n}\ [(1 - \pi)\ N(xn; \mu2, \Sigma)]^{(1-t_n)}$

$= \Pi\ [P(c1)P(x|c1)]^{t_n}\ [P(c2)P(x|c2)]^{(1-t_n)}$

So the first term of this multiplication is the contribution of the c1 class, while the second term is the contribution of the c2 class.

From this formula we get:

$$\pi = \frac{N_1}{N}$$

$$\boldsymbol{\mu}_1 = \frac{1}{N_1}\sum_{n=1}^{N} t_n x_n \qquad \boldsymbol{\mu}_2 = \frac{1}{N_2}\sum_{n=1}^{N}(1 - t_n)x_n$$

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

$$\text{with } S_i = \frac{1}{N_i}\sum_{n\in C_i}(x_n - \boldsymbol{\mu}_i)(x_n - \boldsymbol{\mu}_i)^T,\ i = 1, 2$$

The means are the averages of samples of class 1 and 2 respectively.
Also the covariance matrix is the weighted average of the term S which provides the distance between each sample wrt the mean of the distribution.

Prediction of a new sample

To predict another sample x' (not belonging to D) we compute P(c1 | x') = σ(wT x'+ w0)
which returns a value between 0 and 1 (output of a sigmoid). If this value is > 0.5 then x' is classified with class c1 else with class c2.

Maximum likelihood (K classes)
We can extend what said before even for k-classes classification:

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{\sum_j P(x|C_j)P(C_j)} = \frac{exp(a_k)}{\sum_j exp(a_j)}$$

with ak = ln P(x|ck)P(ck)

The term at the left is the softmax function which maps input values (of a vector) between 0 and 1 in a way that their sum is equal to 1.
Then we assume:

$$P(C_k) = \pi_k,\ P(\mathrm{x}|C_k) = \mathcal{N}(\mathrm{x}; \boldsymbol{\mu}_k, \Sigma)$$

$$\text{Data set } D = \{(\mathrm{x}_n, \mathrm{t}_n)_{n=1}^{N}\}, \text{ with } \mathrm{t}_n \text{ 1-of-K encoding}$$

And after solving the maximum likelihood problem, we get:

$$\pi_k = \frac{N_k}{N}$$

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} t_{nk}\, \mathrm{x}_n$$

$$\Sigma = \sum_{k=1}^{K} \frac{N_k}{N} S_k\ , \quad S_k = \frac{1}{N_k} \sum_{n=1}^{N} t_{nk} (\mathrm{x}_n - \boldsymbol{\mu}_k)(\mathrm{x}_n - \boldsymbol{\mu}_k)^T$$

Maximum Likelihood for parametric model (General formulation)

Likelihood for a parametric model MΘ of dataset D = <X,t>: P(t|Θ, X),
Then the maximum likelihood solution is: Θ* = argmax_Θ ln P(t|Θ, X)

Some conclusions and problems.

Let's consider first a generic function f:R^d -> {c1,...,ck}
D = <X,t>
X is a matrix of shape (N,d) where N is the number of samples and d is the dimension of the input space (how many attributes describe X).
t is a matrix of shape (N, k) where N is the number of samples and k is the number of classes.

The method one-out-of-k encoding is used to fill with zeros all the terms of the vector excluding the one at index k if k is the class of the corresponding sample.

Given the parametric model:

- P(x|ci) = N(xi; μi, Σ)
- P(ci) = π

We can define the set of parameters Θ = {π, μi, Σ} and design a parametric model MΘ.

Which is the size of the independent parameters? i.e. how many degrees of freedom are there?

Suppose d = 4 = dimensionality of the input space, k = 3 = classes.

- π is a vector of k elements, but only 2 of them are independent, since we have put the constraint that the sum of the elements of π is 1 -> here 2 degrees of freedom
- μi is a vector of size d = 4 (since it represents the mean of x) but there are k=3 vectors μi each independent. -> here 4x3=12 degrees of freedom
- Σ is a 4x4 matrix and it is equal for each class, so it is symmetric we keep 10 independent parameters (for a symmetric matrix of dimensions nxn the number of independent variables is binomial coefficient of n+1 over 2) out of 16 total parameters -> here 10 degrees of freedom

The total size of the model is 2+12+10=24.

Note that if Σ was different for each class, then we should have picked all the 16 parameters.

What happens if Σ is the same for all classes vs if Σ is different for each class?

First case: Σ is the same
The same gaussian distribution is applied over the classes (with different shapes)

Second case: Σ is different
We should have different gaussian distributions over the data points with different shapes (according to the distribution of each class)

Discriminative models

Problem formulation: Consider a function f:R^d –> $\{c_1, \ldots, c_k\}$ and dataset D in matrix form <X, t>, where X is a matrix of shape nxd and t is a matrix of shape nxk (if k=2 t is a vector of values 0 or 1).
What we want to compute is P($C_k|X$), the posterior probability.

Let's make the assumption: P belongs to a family of known distributions (Gaussians)

In case of k=2 classes, P($C_k|X$) can be written as $\sigma(w^T x)$, where σ is the sigmoid function

In case of k>2 classes, P($C_k|X$) can be written as $softmax(w^T x)$

We can now set a maximum likelihood problem of the form $P(h|D) = P(D|h)P(h) / P(D)$

Let's define as θ the parameters describing the model M_θ equal to the hypothesis h.

The likelihood formula is:

$$P(\theta|D) = P(D|\theta)P(\theta) / P(D)$$

D = <X,t>

$$P(\theta|X,t) = P(X,t|\theta)P(\theta) / P(X,t)$$

X the input doesn't depend on θ

$$P(\theta|X,t) = P(t|\theta,X)P(\theta) / P(X,t)$$

The solution of this problem is: $\theta^* = argmax_\theta P(t|\theta, X) = argmax_\theta P(t|\theta)$ (simplified notation)

θ^* is the best configuration explaining data.

We can also write θ^* = w* to show the dependence of P with w.

w* = $argmax_w P(t|\sigma(w^T x))$

We can solve this with two approaches: generative (find μ, Σ, π, …) or discriminative.

In the discriminative approach:

it is convenient to maximize the log likelihood, instead of the likelihood:

$$\theta^* = argmax_\theta Log\ P(t|\theta, X) = argmax_\theta Log\ P(t|\theta)$$

But in some cases, it is also convenient to minimize it, by simply adding minus to the previous formula:

$$\theta^* = argmin_\theta - Log\ P(t|\theta, X) = argmin_\theta - Log\ P(t|\theta) \quad \text{(i)}$$

$- Log\ P(t|\theta, X)$ is also called negative log-likelihood.

How to compute the solution for (i) ?

Let's first call y = (x;w) the prediction of the model parameterized by w wrt the input x.

In case of two classes (k=2): y(x;w) = $\sigma(w^T x) \in (0,1)$.

In case of k>2 classes: y(w;x) = softmax($w^T x$).

Logistic Regression

Contrary to what the name suggests, it is not a regression model, but a classification one.

It is a model whose input lies in the continuous space and returns values in (0,1) returned by the sigmoid function. The process is done through the discriminative approach, so it uses the log likelihood function to approximate the target function: $y = 1/(1 + e^{-w^T x})$.

Given x, we can write a matrix of y of n rows (as many as the number of samples) and k columns as many as the classes are.

In the case of k=2, y is a vector of shape nx1 where each row can be a number in (0,1) (the result of the sigmoid).

In the case of k>2, y is a matrix of shape nxk where each row is a vector containing values from 0 to 1 and their sum is 1. (result of softmax)

In the case of 2 classes:

$$P(t|w) = \Pi_{i=1,..,N} y_n^{t_n}(1 - y_n)^{1-t_n}$$

The first term is the contribution of samples with t_n = 1, the second one is the contribution of samples with t_n=0.

$$Log\ P(t|w) = \Sigma_{i=1,..,N} t_n log(y_n) + (1 - t_n) log(1 - y_n)$$

$$- Log\ P(t|w) = - \Sigma_{i=1,..,N} t_n log(y_n) + (1 - t_n) log(1 - y_n)$$

This last expression is a function in w and can be seen as an error function.: the binary cross entropy.

In fact, this is the cross-entropy error E(w) = $-\ Log\ P(t|w)$.
The solver is:

$$w^* = argmin_w E(w) = argmin_w [-\ LogP(t|w)] \text{ (ii)}$$

To define what this term is, sometimes we can use a closed form solution (analytical), but most of the time it is not possible. We have to go towards iterative methods using optimization algorithms such as Gradient Descent.

The approach when k>2 is the same and the negative log likelihood is: $-\ \Sigma_{k=1,\dots,K}\Sigma_{n=1,\dots,N} t_{n,k} Logy_{n,k}$

Ex. of iterative method:

Gradient descent

We start with random values of w (let's call them w^0). By fixing w, we can compute (ii) and obtain E(w^0). Then we compute the gradient *around* E(w^0) and move towards the negative direction of it.
The length of the step with which we move is called the learning rate step η. By applying this process, we arrive to another point of w, let's call it w^1. We repeat the same steps until convergence.

We can associate this process to a formula:

$$w_{i+1} <-\ w_i\ -\ \eta \nabla_w E(w_i)$$

The issues of this procedure are:

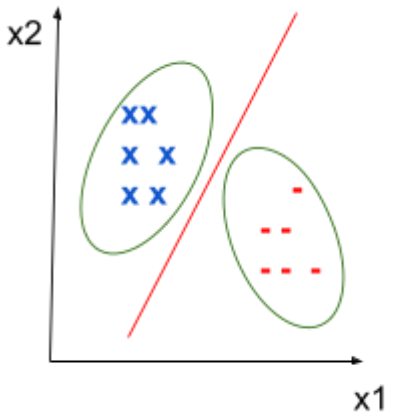
- random initialization of w -> if we are in a flat region we wouldn't move at all (the gradient there is 0)
- value of the learning step -> how to set it? analytical or not?
- no guarantees of global minimum
- termination criterion -> when should we stop?

The termination condition can be either:

1) maximum number of iterations reached
2) no improvements within the current iteration and the previous one wrt some threshold value

Both of these criterions are influenced by hyperparameters: the maximum number of iterations and the threshold value.
In general, it is preferable to end before the process and lose some (very) little improvements instead of consuming time and resources.

Ex: solutions of generative and discriminative models in a 2-dimensional space



circles-> generative model solution with different variance

line -> discriminative model solution

.
As the figure shows, the discriminative methods draw a partition of the space to separate classes.
If we are in 2D: the partition is a line

If we are in 3D: the partition is a plane
If we are in ND: the partition is an hyperplane

When do all these methods fail?
A model fails when the assumptions are not true. In this case, the assumption is that data follows the Gaussian distribution.
An example where the points are not distributed in a Gaussian way is the banana-shape or circle-shape.

Linear Model for Classification

Notation:

Two classes:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 = \tilde{\mathbf{w}}^T\tilde{\mathbf{x}}, \text{ with:}$$

$$\tilde{\mathbf{w}} = \begin{pmatrix} w_0 \\ \mathbf{w} \end{pmatrix}, \tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$$

where w is the matrix of the weights and w0 is the bias, a term representing the systematic error that prevents the model from accurately representing the underlying relationship between input features and the target variable. Usually the *tilde* is omitted.

The assumption here is that **data is linearly separable** i.e. there exists a linear partition (line, plane, hyperplane) separating the space in 2 regions where the first region contains only + values and the second one contains only - values. It may happen that data is not linearly separable so we have to make this assumption. In this kind of problem, the separator is itself a solution. Of course, there may be lots of separators which are consistent with data, so lots of solutions.

So, our goal is to find a function separating regions such that all the classes are correctly separated.
The function is in the form: $y: X -> \{c_1, \ldots, c_k\}$. y is itself a solution, a K-class discriminant.

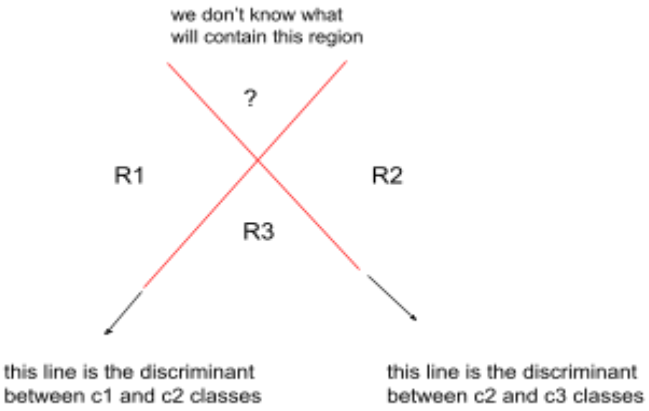
In case of 2 classes: $y(x) = w^T x + w_0$ or in the compact form $y(x) = W^T x$. So the size of W is d+1

In case of k>2 classes: $y_1(x) = w_1{}^T x + w_0$, $y_2(x) = w_2{}^T x + w_0$, …, $y_n(x) = w_n{}^T x + w_0$

So, we have a system of n linear equations. So the size of W is (d+1)xn.

It is not possible to convert a multiclass classification problem into a set of binary classifiers. That's why we use a system of k linear equations to be solved in order to find a hyperplane in R^d.

Ex. One-versus the rest (k-1) classifiers: 3 classes, 2 binary classifiers



How to choose the best solution?
In general, we'd choose the one that satisfies this condition:
given a new sample x' not in D, the likelihood of misclassifying data is less than the one of the other solutions.

How to find the solution? i.e. find W
Three main approaches based on the minimization of the error function through some optimization methods are presented.

1) **Least squares**
   Let's consider D=$\{(x_n, t_n)_{n=1,..,N}\}$
   Find y(x) = $W^T x$
   To do this we use a 1-of-k coding scheme for t: $x \in C_k \rightarrow t_k = 1, t_j = 0 \; \forall j != k$
   We call T the vector containing all $t_i, i = 1,..., N$
   We define the error function as a quadratic form given by the matrix multiplication between the difference of the prediction of the model and the real label. So this error gives a measure of the performance of the model. Its formula is:

$$E(w) = 1/2 \, trace\{(XW - T)^T (XW - T)\}$$

   The trace is the sum of elements of the main diagonal of the resulting matrix, so the error is a real value. If E(w) = 0 then the model predicts every sample correctly.
   This method has a **closed form solution**:

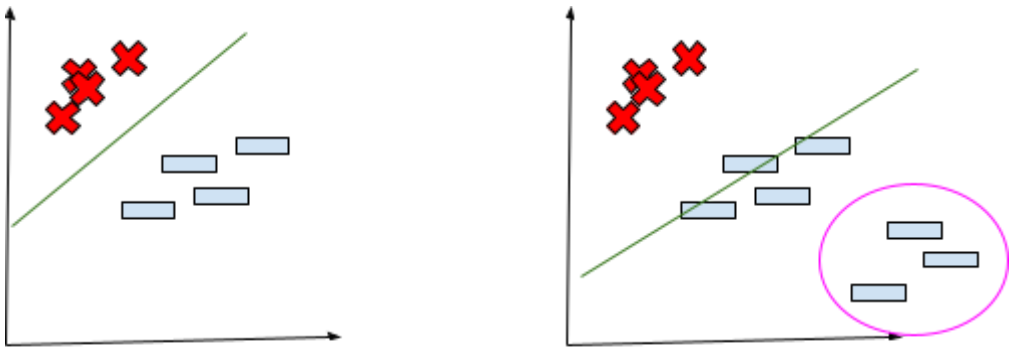
$$W^* = (X^T X)^{-1} X^T T,$$

   $(X^T X)^{-1} X^T$ is the pseudo-inverse matrix.

$$y(x) = T^T ((X^T X)^{-1} X^T)^T X$$

   Least Squares works well in some cases, but **it's not robust to outliers**, since the resulting separator is affected by all points of the dataset. Outliers are points whose distribution is different from the main distribution describing other points.

   Ex.

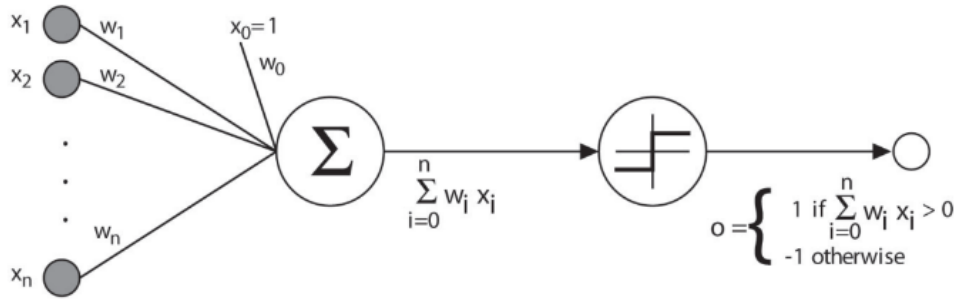


As the picture shows, the line (the solution of LS) is attracted by the outliers and misclassified some samples.

The classification of a new instance x not in D, consists in computing y(x) which is a vector of K y_i., with i = {1,...,K}. From this vector we pick k = argmax_i {y_i(x)}

2) Perceptron

Linear combination of the inputs x plus a sign function



To train it, we need to define an error function:

$$E(w) = 1/2\,\Sigma_{n=1,\dots,N}(t_n - o_n)^2$$

without sign function we have:

$$E(w) = 1/2\,\Sigma_{n=1,\dots,N}(t_n - w^T x_n)^2 \text{ (i)}$$

with sign function we have:

$$E(w) = 1/2\,\Sigma_{n=1,\dots,N}(t_n - sign(w^T x_n))^2 \text{ (ii)}$$

This error has no closed solution, so we'll use an iterative method with Gradient Descent:

$$\partial E/\partial w_i = \Sigma_{n=1,\dots,N}(t_n - w^T x_n)(-x_{i,n}) \text{ (i)}$$

$$\partial E/\partial w_i = \Sigma_{n=1,\dots,N}(t_n - sign(w^T x_n))(-x_{i,n}) \text{ (ii)}$$

With the computation of the gradient we can apply the perceptron algorithm based on GD:

1) initialize w randomly
2) $w_i <- w_i - \eta \partial E/\partial w_i$ until convergence

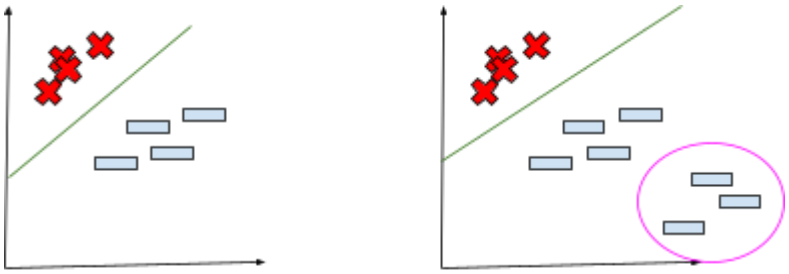
3) output w

This algorithm can be implemented by considering:

- **batch mode**: all dataset is processed to update weights
- **mini-batch mode**: a small subset is processed to update weights (better solution)
- **incremental mode**: only one sample is used

The perceptron is robust to outliers since it just uses the classification of a sample to adjust the weights, without considering the distance from the line to the points.

Ex. considering the two scenarios:



no outliers

outliers

When adding the outliers, the separator (solution) doesn'change, since they are classified correctly and the error is 0 (no update).

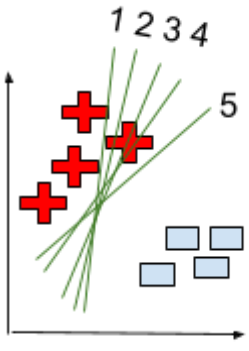
Note: value of learning rate is important

if η is too big, we may end up in a point where the error is higher wrt the previous one: divergence.

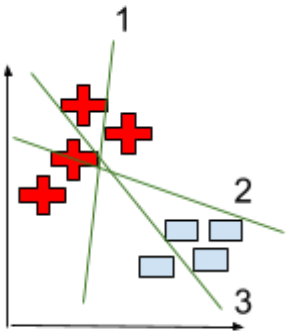
if η is too small, we're almost sure we're going to the minimum, but the number of steps is too high: slow convergence.

To set the right η we should prune this parameter, or make it a variable (not a constant) changing in time through some optimizers.

Evolution of the solutions with η small or big:



small learning rate:
stop at 5 -> maybe it's not the best



big learning rate:
not able to find the solution

The classification of a new instance x not in D, consists in computing k = sign(w* x), using the learnt w*.

3) **Fisher's Linear Discriminant**

It is a geometric approach based on error minimization.
Consider k=2 (two classes):
Determine y = $w^T x$ and classify x ∈ C1 if y ≥ −w0, x ∈ C2 otherwise.

The resulting w is the one maximizing the distance between the 2 classes.
Let's define:

$$N1 = number\ of\ samples\ belonging\ to\ class\ C1$$
$$N2 = number\ of\ samples\ belonging\ to\ class\ C2$$
$$m1 = 1/N1\ \Sigma_{n\in C_1} x_n$$
$$m2 = 1/N2\ \Sigma_{n\in C_2} x_n$$

The first variant is:

Choose w that maximizes J(w) = w^T(m2 − m1), subject to ||w|| = 1.

The second variant (**using the fisher criterion**) is:

Choose w that maximizes J(w) = $w^T S_B w$ / $w^T S_W w$, where

$$S_B = (m2 - m1)^T$$
$$S_w = \Sigma_{n\in C_1}(x_n - m1)(x_n - m1)^T + \Sigma_{n\in C_2}(x_n - m2)(x_n - m2)^T$$

Sb, the between class scatter matrix, represents the spread between the means of different classes.
Sw, the within class scatter matrix, represents the spread of data points within each class.
The solution is obtained by computing dJ(w)/dw = 0.
Summarizing, given a two classes classification problem, Fisher's linear discriminant is given by the function y = $w^T x$ and the classification of new instances is given by y ≥ −w0, where

$$w = S_w^{-1}(m2 - m1)$$
$$w_0 = w^T m,\ \text{m is the global mean of all the dataset.}$$

The fisher's linear discriminant **is not robust to outliers** (but not as Least Square), because it uses means and variances related to all points.

Support Vector Machine

Support Vector Machine also called SVM or maximum margin model is a linear model used for classification.

Problem formulation:
Let's consider k=2 classes.
The dataset is $D = \{(x_n, t_n)_{n=1,..,N}\}$

The desired function f is defined as follows: $f: R^d \rightarrow \{-1, +1\}$ (The values -1,+1 are useful for the mathematical expression of the dataset).

We constraint f to be a linear model, so $y(x; w) = W^T x + w_0$

The *assumption* we make is the linear separability of data:

$$\exists\, w,\ w_0\ such\ that\ y(x_n) > 0\ if\ t_n = 1\ and\ y(x_n) < 0\ if\ t_n = -1\,.$$

The key idea behind SVM is using a goodness' measurement of the hyperplane separating data: the **margin**. We define as margin the shortest distance between the hyperplane h and the closest sample point to h as follows: $margin = min_{n=1,\dots,N} |y(x_n)|/||w||$.

To get a more accurate solution, SVM wants to maximize this margin:

$$w^*, w_0 = argmax_{w,w_0}(margin)\,,$$

subject to $t_n(y_n) >= 1.$

Property: once we get w*, w0 as the optimal solution h, then there will be *at least* a pair of samples, one belonging to the positive class and the other belonging to the negative one such that their margin wrt h is the same.

To get an easier solution we will normalize data such that

$$t_n(w^T x_k + w_0) = 1,\ \forall k = 1,\dots, N\,.$$

In this way, the formula of margin becomes: $margin = min_{n=1,\dots,N}\ 1/||w||$. *Rescaling doesn't affect the solution.*

We can now solve $w^*, w_0 = argmax_{w,w_0}(margin) = argmax_{w,w_0}[min_{n=1,\dots,N} 1/||w||]$.

But we transform it first in:

$$w^*, w_0 = argmax_{w,w_0}[min_{n=1,\dots,N} 1/||w||] = argmin_{w,w_0} ||w||^2 * 0.5.$$

subject to $t_n(y_n) >= 1$

So, we have transformed the problem from a solution space in R^{d+1}, where d is the input space dimensions (number of attributes) to a solution space in R^N where N is the dataset dimension.

This problem has now a quadratic form solution, obtained by applying the Lagrangian method.

Lagrangian solution is computed as:

$w^* = \Sigma_{n=1,\dots,N} a^*_n t_n x_n$, where $t_n x_n$ are known and a^*_n are Lagrangian multipliers

The definition of this solution explains why we pass from d+1 research space to a N research space, which looks strange given that N>>>d, typically.

In fact, we have lots of samples x_n with an associate Lagrange multiplier $a_n = 0$. When this happens, x_n doesn't contribute to the solution. This is better expressed with the KTK condition:

$\forall\, x_n \in D,\ either\ a^*_n = 0\ or\ t_n y(x_n) = 1,\ thus\ t_n y(x_n) > 1\ implies\ a_n = 0.$

Given this formal condition, we define support vectors the set of points s.t. $t_k y(x_k) = 1$ and $a_n > 0$.

This means that SVM is robust to outliers since they are points far away from h and so they have a = 0: they will not contribute to the solution.

We can thus rewrite the solution in terms of support vectors:

$$y(x) = \Sigma_{x_j \in SV} a_j^*\ t_j x_k^T x_j + w_0^* = 0$$

To compute w_0 *:

$$t_n(\Sigma_{x_j \in SV} a_j * \ t_j x_k^T x_j + w_0 *) \ = \ 1$$

$w_0 *= t_k - \Sigma_{x_j \in SV} a_j * \ t_j x_k^T x_j$, can be affected by calculus errors so we use

$w_0 *= 1/|SV| \ \Sigma_{x_k \in SV} (t_k - \Sigma_{x_j \in SV} a_j * \ t_j x_k^T x_j)$, the average is more stable

To classify a new sample x':

$$y(x') \ = \ \Sigma_{x_j \in SV} a_j * \ t_j x'^T x_j + w_0 *$$

SVM is efficient also when d<<|D| (most of ai will be zero) or when d is large or infinite.

How to remove the assumption of linear separability?

We always want to maximize the margin to have more probability that given a new sample it will be classified correctly. Even in situations where there are some points which can not be separated from the rest in a linear way, we still want to use SVM and accept their misclassification. To do that, we need to relax the hypothesis of linearly separability of data, by introducing slack variables ξ.

- $\xi_n = 0 \ if \ x_n \ is \ on \ or \ inside \ the \ correct \ margin \ boundary$
- $\xi_n > 0 \ and \ \xi_n <= 1 \ if \ x_n \ is \ inside \ the \ correct \ margin \ boundary$
- $\xi_n = 1 \ if \ x_n \ is \ on \ the \ wrong \ margin \ boundary$

The assumption of linear separability becomes: $t_n(y_n) >= 1 - \xi_n \ for \ each \ n = 1,..., N$

The optimization problem becomes: $w *, w_0 = \ argmin_{w,w_0} ||w||^2 * 1/2 + C \Sigma_{n=1,..,N} \xi_n$, where C is a constant proportional to the value of ξ_n. It is now subject to $t_n(y_n) >= 1 - \xi_n \ for \ each \ n = 1,..., N$.
The solution can be obtained following the same previous computation.

When data is not linearly separable, we can also use a change of coordinates (e.g. from cartesian to polar).

We have to note that linearity is expressed in terms of the parameters of the model: we can have a model linear wrt w but non linear wrt x.

Linear Models for Regression

Regression: type of supervised learning used for predicting a continuous output variable. The goal is model the relationship between the input features and the output variable.

Linear Regression

Problem formulation:

Learn an approximation of $f: X -> Y$, where $X \subseteq R^d$, $Y \subseteq R$. So, outputs are real values.
$D = \{(x_n, y_n)_{n=1,...,N}\}$. This approximation, y(x,w), should be linear wrt x and w:

$$y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_d x_d$$

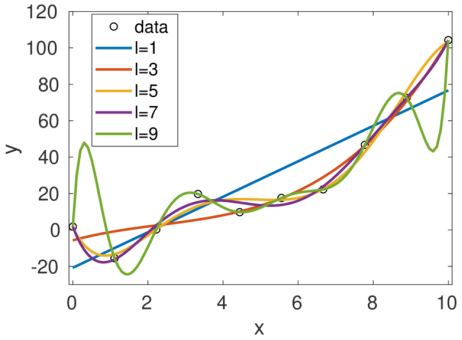
X is a vector of shape d+1 with the first element set to 1, and w is a vector of shape d+1.

We can also use a transformation of the input space Φ(x) which maps x into a new feature space (also with another dimension wrt d, let's call this dimension M). This transformation can be non-linear wrt x, so y(Φ(x),w) is linear wrt w but not wrt x.

$$y = w_0 + w_1 \Phi_1(x) + w_2 \Phi_2(x) + ... + w_d \Phi_d(x)$$

It can be written in matrix form: $y = W^T \Phi(x)$, W and Φ(x) will have M+1 components (typically M != d).

Ex. of transformation: polynomial curve fitting



From the figure above we distinguish:

- data: points follow the real function f but we consider also some noise, otherwise the problem would be trivial
- "l" (we have called it M): it is the order of the polynomial function. If:
  - l = 0 -> the solution would return a constant value v* minimizing the error (i.e. the orthogonal segment measuring the distance between the point and the predicted function). This solution will lead to underfitting (the inability of the model to adapt both to training data and test data).
  - l > 0 -> the solution will be a polynomial function with order l, and for increasing values of l we will have a more complex solution. When the order of the polynomial is high, there will be no error: overfitting. The model has learnt exactly how training data is distributed. It fits data perfectly. This is not desirable in machine learning, because given a new sample x' (not in D) the error the solution produces is high.

So, l (M) is a hyperparameter of the model which needs to be tuned. It determines the complexity of the model and influences underfitting and overfitting.

How do we find the solution?

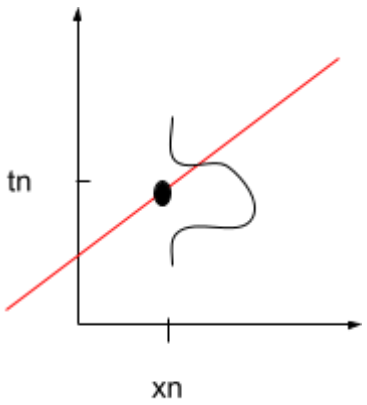
Two methods can be used:

1) **Maximum likelihood**

   $t = y(x; w) + \epsilon$, where t is the target value and ϵ is an error.

   Assuming Gaussian Noise: $P(\epsilon|\beta) = N(\epsilon|0, \beta^{-1})$ where the mean is 0 (unbiased) and β^{-1} is the variance.

   The probability we want to compute is $P(t|x, w, \beta) = N(t|y(x; w), \beta^{-1})$. The gaussian is centered on the prediction of the model and the variance is equal to β^{-1} :



   Now, we want to find the parameters maximizing the likelihood:

   $P(\{t_1, \ldots, t_n\}|\{x_1, \ldots, x_n\}, w, \beta) = \Pi_{n=1,\ldots,N} N(t_n|w^T\Phi(x_n), \beta^{-1})$, because we assume samples iid

   Let's compute the log likelihood:

   $ln\, P(\{t_1, \ldots, t_n\}|\{x_1, \ldots, x_n\}, w, \beta) = \sum_{n=1}^{N} N(t_n|w^T\Phi(x_n), \beta^{-1})$ =

   $-0.5\beta \sum_{n=1}^{N} [t_n - w^T\Phi(x_n)]^2 - N/2\, ln(2\pi\beta^{-1})$, the first term is the MSE, the second term doesn't depend on w.

   This leads us to a conclusion: maximizing the log-likelihood means minimizing the negative log-likelihood but also minimizing the mean squared error.

   The solution is solving a quadratic function with a closed form:

$$E_D(w) = 0.5\,(t - \Phi w)^T(t - \Phi w))$$
$$\nabla E_D(w) = 0$$
$$w^* = (\Phi^T\Phi)^{-1}\Phi^T t$$

2) **Gradient descent:**

   it is also possible to use the iterative algorithm to solve this problem and we are guaranteed to reach a local optimal solution.

$$w <- w - \eta\nabla E_D(w)$$

We can have an issue with this mechanism related to the value of w. If w is too high, then the function is not smooth and we are more likely to have a big error.
To solve this problem, we typically use a regularization term added to the error:
$E = E_D(w) + \lambda E_w(w)$.
This second term, the regularization term, doesn't depend on data and generally it is:
$E_w(w) = 0.5\, w^T w$ or $E_w(w) = \sum_{j=0}^{M} |w_j|^q$.
It is used because it discourages w to be too high.
λ is a hyperparameter of the model.

In case of multiple outputs:
y is a vector of K components and T is a vector of K components. We solve the problem as the previous case.

Adding hyperparameters is either:
- good: we can make the model more flexible
- bad: we have to check if values are adapt (tuning is needed)

However, parameter optimization is better than creating a new model.

Kernel Models

Consider a linear model y(x;w) = $w^T x$ and a dataset $D = \{(x_n, y_n)_{n=1,\dots,N}\}$.
Define an error function:
$J(w) = 0.5 \sum_{n=1}^{N} (w^T x_n - t_n)^2 + \lambda w^T w \; -> \; matrix\ form \; -> \; J(w) = (t - Xw)^T (t - Xw) + \lambda ||w||^2$
Optimal solution is given by:

$$\nabla J(w) = 0$$
$$w = -1/\lambda \sum_{n=1}^{N} (w^T x_n - t_n) x_n$$

Let's call

$$\alpha_n = -1/\lambda\, (w^T x - t_n),\ then\ w^* = \sum_{n=1}^{N} \alpha_n x_n, \text{ (linear combination of x)}$$

So, $y(x; w^*) = w^{*T} x = \sum_{n=1}^{N} \alpha_n x_n x$.
We can put this in matrix form W* = $X^T \alpha$ and $\alpha = (XX^T + \lambda I_N)^{-1} t$
Let's call $K = XX^T$, the "Gram Matrix".
K represents an inner product, and can be interpreted as a similarity measure between two instances.
So, we explicit this function of similarity, also called kernel, as: $K: XxX \; -> \; R$.
Properties of k:
- k(x',x) = k(x,x') typically
- k(x',x) >= 0 typically
- real-valued function

- k(x',x) = $x'^{T}x$ if kernel is linear
- applicable to any x even of infinite size

We can apply this principle, **also called Kernel Trick,** of substituting an inner product of the input vector x with a kernel. This process is known as Kernelized linear models and can be applied to any x (even infinite size).
Also if we have a transformation of the input Φ(x) we can define k(x,x') = $\Phi(x)^{T}\Phi(x')$ even without knowing Φ.
In order to apply kernel methods we typically have to normalize data (with some techniques).

Kernel Families
There exists a lot of kernel families: linear, polynomial, RBF, sigmoid.

- Linear kernel: k(x,x') = $x^{T}x'$
- Polynomial kernel k(x,x') = $(\beta x^{T}x' + \gamma)^{d},\ d \in \{2, 3,...\}$
- Radial Basis Function (RBF): k(x,x') = $exp(-\beta|x - x'|^{2})$
- Sigmoid: k(x,x') = tanh($\beta x^{T}x' + \gamma$)

Kernelized SVM for classification

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

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

$$w_0 = 1/|SV| \sum_{x_i \in SV} (t_i - \sum_{x_j \in S} a_j t_j k(x_i, x_j))$$

Kernelized Linear regression

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

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

$$\alpha = (K + \lambda I_N)^{-1}t \rightarrow the\ inverse\ matrix\ computation\ requires\ N^{2}\ operations\ (K\ has\ shape\ NxN,\ not\ sparse)$$

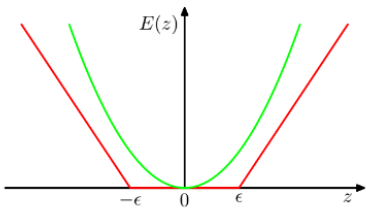
Kernelized SVM for regression

Since the computation of previous case is intractable, we consider another loss function:

$$J(w) = C \sum_{n=1}^{N} E_{\epsilon}(y_n, t_n) + 0.5\,||w||^{2},\ where\ C = \lambda^{-1}.$$

The error is defined as:

$$E_{\epsilon}(y_n, t_n) = 0\ if\ |y - t| < \epsilon,\ else\ E_{\epsilon}(y_n, t_n) = |y - t| - \epsilon\,.$$



But this error is not differentiable so it is difficult to solve: we have to introduce the slack variables ξ_n^+, ξ_n^-.

These variables are equal to zero when the error is zero, so for points inside the $\epsilon - tube$.

$$t_n <= y_n + \epsilon + \xi_n^+$$

$$t_n <= y_n - \epsilon - \xi_n^-$$

The loss function can be rewritten as:

$$J(w) = C \sum_{n=1}^{N} (\xi_n^+ + \xi_n^-) + 0.5\,||w||^2$$

Subject to:

$$t_n <= y(x_n; w) + \epsilon + \xi_n^+$$

$$t_n <= y(x_n; w) - \epsilon - \xi_n^-$$

ξ_n^->= 0

ξ_n^+>= 0

This error penalizes the points whose error is not zero, so points outside the tube.

From Karush-Kuhn-Tucker (KKT) condition, Support vectors contribute to predictions:

aˆn > 0 ⇒ ϵ + ξn + yn − tn = 0, data point lies on or above upper boundary of the ϵ - tube
aˆ'n > 0 ⇒ ϵ + ξn − yn + tn = 0, data point lies on or below lower boundary of the ϵ - tube

All other data points inside the ϵ-tube have ˆan = 0 and ˆa'n = 0 and thus do not contribute to prediction.

Properties:

- Kernel methods overcome difficulties in defining non-linear models
- Kernelized SVM is one of the most effective ML method for classification and regression
- Still requires model selection and hyper-parameters tuning

Instance Based Learning

Instance based learning is a concept related to non-parametric models. These models don't have a fixed number of parameters, instead this number grows with the amount of data. So the model structure is not known a priori. Parametric models have a fixed number of parameters and so fixed size.

K-Nearest Neighbors - Classification problem

Problem formulation: $f: X \rightarrow C,\ D = \{(x_n, y_n)_{n=1,\dots,N}\}$

Solution:

1) find k nearest neighbors of a new instance x
2) Assign to x the most common label among the majority of neighbors

The likelihood of a new instance is: p(c|x, D,K) = $1/K \sum_{x_n} I(t_n = c)$. So this method corresponds to maximize a local likelihood since each term x_n of the summation belongs to the set of neighbors of x (N), so $x_n \in N_k$ (x, D).
I is a function returning 1 if the equation inside is true, 0 otherwise.
K is for instance an hyperparameter of the model.

When K = 1: the classification of x depends on the nearest neighbor (only one!)
When K = 3: the classification of x depends on the 3 nearest neighbors x1, x2, x3. If x2 and x3 are “red” and x1 is “blue”, then x will be classified as red since the majority of its neighbors are red.

Increasing K brings smoother regions.
When K is too small, then there’s more probability of overfitting to occur. **Even when the labels are noisy, we have the risk of overfitting.** When K is too high then there will occur underfitting.

This technique may lead to some **issues**:

- requires storage of all the dataset: prefer KNN when the dataset is small
- depends on a distance function

Moreover, there’s no training phase.

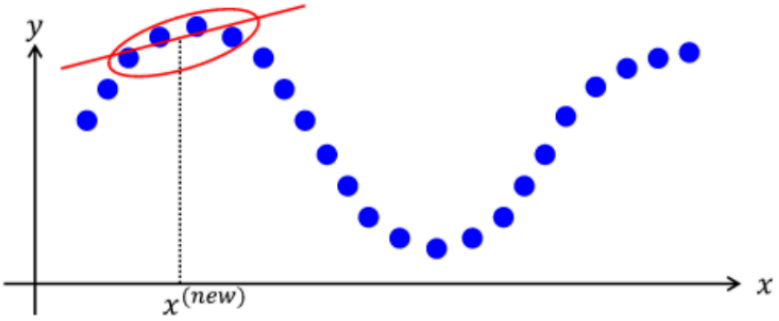
The distance function can be realized as a kernelized version of a distance function $||x - x_n||^2,\ with\ x_n \in N_k(x, D)$.

K-Nearest Neighbors - Regression problem

Problem formulation: $f: X \rightarrow R,\ D = \{(x_n, y_n)_{n=1,..,N}\}$.

Solution:

1) compute N_k (x_q, D) subset of nearest neighbors of x_q
2) Fit a regression model y(x; w) on N_k (x_q, D)
3) Return y(x_q, w)



Multiple Learners

There can be some portions of data better explained with a particular model and other parts better explained with others. So combining different models, learners, is a useful approach.

Instead of putting much effort into training until reaching better results (which often requires a lot of time) then we should train different models both in parallel (each model is independent from the rest) or in a sequential way (each model is dependent from the previous ones).

When the training is done in parallel we talk about voting or bagging, when the training is done in a sequential way then we talk about boosting.

Voting

Given a dataset D,

1. use D to train a set of models ym(x), for m = 1, . . . , M
2. make predictions with:

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

$$y_{voting}(x) = argmax \sum_{m=1}^{M} w_m I(y_m(x) = c)\,,\ for\ classification\ problems$$

with wm ≥ 0, ∑wm = 1 (prior probability of each model) and I(e) = 1 if e is true, 0 otherwise.

The learners are executed:

- in parallel and weights can be:
  - fixed
  - variable depending on some functions (ex. gating, or other learnable functions (stacked))
- in cascade

Bagging

Given a dataset D,

1. generate M bootstrap data sets D1, . . . , DM, with Di ⊂ D
2. use each bootstrap (each intersection pair can be non - empty) data set Dm to train a model ym(x), for m = 1, . . . , M
3. make predictions with a voting scheme

$$y_{bagging}(x) = 1/M \sum_{m=1}^{M} y_m(x)$$

Boosting

The idea behind this approach is that base classifiers are trained in sequence using a weighted data set where weights are based on performance of previous classifiers.

In order to update weights (weights of samples correctly classified are decreased, weights of samples misclassified are increased), we use AdaBoost:

Given $D = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$, where $\mathbf{x}_n \in \mathbf{X}$, $t_n \in \{-1, +1\}$

1. Initialize $w_n^{(1)} = 1/N$, $n = 1, ..., N$.
2. For $m = 1, ..., M$:
   - Train a weak learner $y_m(\mathbf{x})$ by minimizing the weighted error function:

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n), \text{ with } I(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

   - Evaluate: $\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$ and $\alpha_m = \ln\left[\frac{1-\epsilon_m}{\epsilon_m}\right]$
   - Update the data weighting coefficients:

$$w_n^{(m+1)} = w_n^{(m)} \exp[\alpha_m I(y_m(\mathbf{x}_n) \neq t_n)]$$

The final classifier will output:

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

Properties of AdaBoost:

AdaBoost is used because it can minimize an exponential error function.
Advantages:

- fast, simple and easy to program
- no prior knowledge about base learner is required
- no parameters to tune (except for M)
- can be combined with any method for finding base learners
- theoretical guarantees given sufficient data and base learners with
- moderate accuracy

Issues:

- Performance depends on data and the base learners (can fail with insufficient data or when base learners are too weak)
- Sensitive to noise

Artificial Neural Network

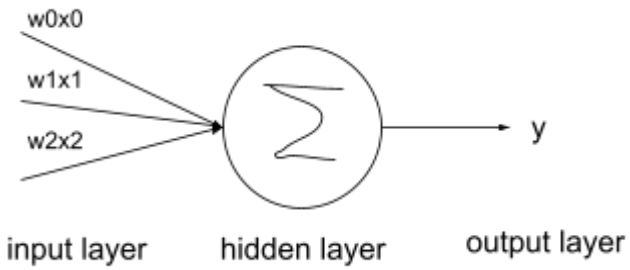
From this moment, we'll face non linear models which have some limitations when dealing with complex problems, since they cannot model interaction between input variables. Also linear models require best choices of hyperparameters and parameters in order to get good results.

Goal: estimate $f: X -> Y$ with $Y = \{C_1, ..., C_k\}$ or $Y = R$. So classification or regression problems.

Data: $D = \{(x_n, t_n)_{n=1,...,N}\}$ such that $t_n \simeq f(x_n)$.

Framework: Define $y = f'(x, \Theta)$ and learn Θ s.t. f' approximates f.

Feedforward network (example):



The input layer is a linear combination of weights w and inputs x. There will be as many input neurons as the number of features. They are passed to the first hidden layer.
The circle represents the unit where the information from the input layer is processed by using this general formula: $f(\sum_i x_i w_i + b)$, where f is an activation (non linear) function and b is a bias term.

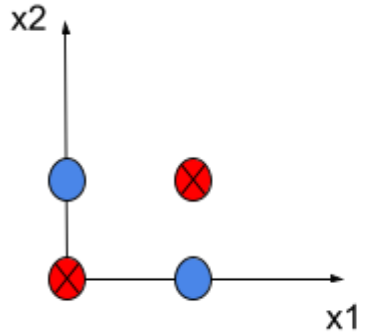
The output takes as input the output of the last hidden layer and processes it with another activation (non linear) function g. If we wouldn't introduce f and g, the model would end up as linear.

The term feedforward refers to the flow of information from the input node to the output node without loops.
The term network refers to the composition (chain) of elementary functions in an acyclic graph.

Example: XOR FUNCTION

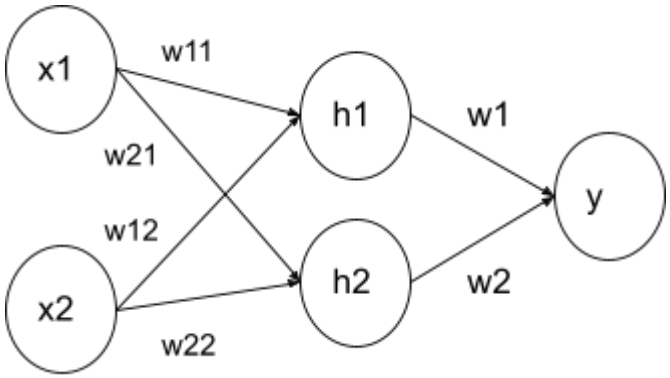
x1	x2	x1 xor x2
0	0	0
1	0	1
0	1	1
1	1	0

So $f\colon R^2 -> \{0,1\}$



not linearly separable!

With a linear model we can compute the solution, but it is not even consistent with the dataset.
So we use the fnn:



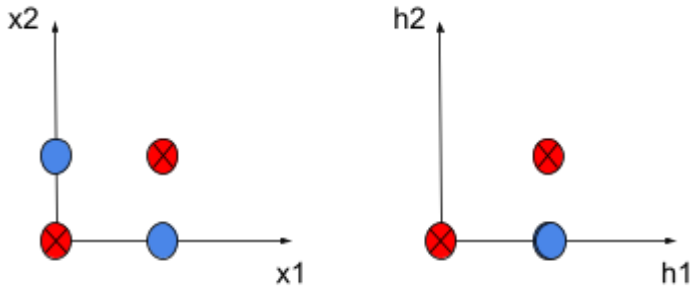
In matrix form:

- X = [x1 x2]T
- H = [h1 h2]T
- W = [w11 w12
  w21 w22]

So, $h1 = f_1(w_{11}x_1 + w_{12}x_2)$, $h2 = f_2(w_{21}x_1 + w_{22}x2)$ *and in matrix form*: $h = f(W^T X)$, $y = g(w^T h)$.
In this case we'll consider f = ReLu and g = identity function.
Finally, $y = w^T h = w^T max(0, W^T X + c) + b$. We have 4 (W) + 2 (c) + 2 (w) + 1 (b) parameters of the model.

By minimizing the MSE Loss function we have:
W = [1 1] c = [0 -1] w = [1 -2] b = 0
1 1

The network automatically transforms the input space in the feature space:



In this way the space is linearly separable, so we can apply the function of the output layer.

- The depth is given by the number of hidden layers −> “universal approximation theorem” says that it is not necessary to have deep neural networks, instead we need wide nn. But in practice, deeper and narrow neural networks are much easier to train and provide better generalization.
- The width is given by the number of units in each layer.
- The activation functions define the type of the units in each layer −> In principle we can use any type of activation function, but there are some considerations to do. We have to prefer activation functions avoiding unit saturations (because of gradient problems). So, for example, the sigmoid activation function is not good in some cases because it has flat regions in the extreme intervals. ReLu activation function is one of the most used because it avoids this problem and even if it is not differentiable in zero, this case doesn’t affect the performances because it never (or almost never) happens due to noise.
- The loss function defines the type of the cost functions −> This choice is related to the activation function choice.

A priori we can’t know which depth, width, activation functions and cost functions to use. We have to try.

In general, when the task is:
- **regression**:
    - **linear units**: identity act function
    - **loss function**: MSE
- **binary classification:**
    - **linear units**: sigmoid function
    - **loss function**: binary cross entropy

- **multi-class classification**:
    - **linear units:** softmax
    - **loss function**: categorical cross entropy

Activation functions

ReLU

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

- easy to optimize
- not differentiable at 0, but no problems for this in practice, (due to noise)

Sigmoid g and hyperbolic tangent t

$$g(z) = \sigma(z)$$
$$t(z) = \tanh(z)$$

- no logarithm at the output: units saturate easily
- gradient based learning is very slow
- hyperbolic tangent gives larger gradients with respect to the sigmoid

Gradient Computaton

We first have to distinguish between the computation / estimation of the gradient (backpropagation) and the training algorithm (which uses the estimation of the gradient) referred to as gradient descent (or sgd), used to minimize the cost or loss function during training of a model.
It consists in 2 steps:

- forward: computations are done from the input to the output, and the prediction is the result. it is then compared with the real target in order to produce the loss (lots of losses can be used)
- backward: the loss is used to update the parameters according to it. The gradient computation is propagated through the network from the output to the input.

Back-propagation is simple and efficient. If not, the repetition of this algorithm many times and with thousands/millions of parameters could be too expensive.

So, the goal is to compute the gradient of the loss wrt the parameters. We know that neural networks are compositions of functions. So, in order to compute the gradient we have to apply the chain rule.

Let: $y = g(x)$ and $z = f(g(x)) = f(y)$
Applying the chain rule we have:

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

For vector functions, $g : \mathbb{R}^m \mapsto \mathbb{R}^n$ and $f : \mathbb{R}^n \mapsto \mathbb{R}$ we have:

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

equivalently in vector notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z,$$

with $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ the $n \times m$ Jacobian matrix of g.

The backpropagation algorithm works as follows:

Forward step

Require: Network depth l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value
$h^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ **do**
$\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
$\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
end for
$\mathbf{y} = \mathbf{h}^{(l)}$
$J = L(\mathbf{t}, \mathbf{y})$

Backward step

$\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$
for $k = l, l-1, \ldots, 1$ **do**
Propagate gradients to the pre-nonlinearity activations:
$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}
$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$
$\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$
Propagate gradients to the next lower-level hidden layer:
$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$
end for

Stochastic Gradient Descent

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

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

Issues with learning rate:
We believe that the random initialization of theta produces a starting solution which is far from the minima. So we set a bigger learning rate at first, in order to move fast towards the negative direction of the gradient (and lose less time). When the number of the iterations increases, we decrease the learning rate accordingly, because we are moving towards the minima and the step we take should be progressively shorter.
A strategy to implement this idea is the following:

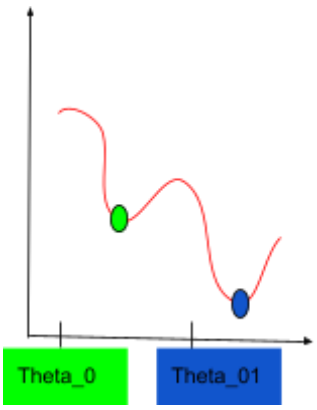
Until iteration τ $(k \leq \tau)$:

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

After iteration τ $(k > \tau)$:

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

Issues with initialization:



Also the initialization of theta is crucial:
If we start with theta_0, then the convergence may end up in the local minima (the green one). If we start with theta_01, then the convergence should end up in the global minima. This is what we want. And to do this, we introduce the concept of "Momentum". The idea is that we want to add a term in the update process due to the "velocity" of the solution. If the velocity is enough to overcome the "hill", then we may converge to the real minima.
This term is: $v^{k+1} = \mu v^{k} - \eta g$, where v^{k+1} is the velocity of next iteration, v^{k} is the velocity of the current one, μ is the "Momentum" hyperparameter, $\mu \in [0, 1]$, η is the learning rate and g is the gradient.

SGD with Momentum

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

Momentum μ might also increase according to some rule through the iterations (e.g. Nestorov Momentum)

There may also be algorithms adapting the learning rate according to the current iteration, these are called “Adaptive Algorithms”: Adam, AdaGrad, RMSprop, …

Regularization
Reaching the global minima doesn’t guarantee anything about overfitting. So, we have to prevent it with some techniques:

1) **Regularization:** Add a regularization term (not depending on the dataset) to the loss function, so that it penalizes the parameters if they are too high.
2) **Data Augmentation:** To avoid the model to specialize itself with data, then we should add some virtual samples obtained by applying some deformation on the real ones. This helps in having a more variable dataset.
3) **Early Stopping:** It is a common practice to keep track of the training and validation loss or accuracy during the epochs. When the validation loss (accuracy) starts growing (decreasing) then the model is going to specialize too much wrt data. So we need to stop.
4) **Parameter Sharing:** Helpful with CNN mostly. It is a technique used to add constraints to the different parameters (which are supposed to be independent without parameter sharing).
5) **Dropout:** It is used to drop out some neurons (randomly) at each iteration, in order to avoid that the model learns too much about the existing connections.

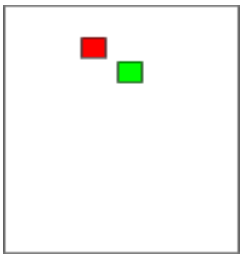
So, in practice:

- choose the architecture and all its parameters.
- choose the optimizer (training) and all its parameters.
- choose the regularization technique.

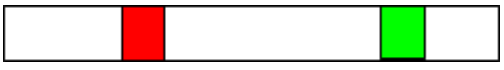
Typically, in order to save time, there’s a pre-compile step where the data structures depending on the architectures and optimizers used are prepared (with no dataset involved).

Convolutional Neural Network

They represent a revolution in computer vision. Theory of convolution is old, but its success comes later due to lack of data and computational power.
Convolutional neural networks are mainly used when dealing with images. Convolution represents an automatic way to extract features, a crucial aspect in computer vision where before CNN handmade techniques were used. So the goal of the convolutional layer is to learn the best parameters of the kernel.
In this case, the convolution operation is like the cross-correlation.



Images have spatial properties to be exploited. So we are interested in capturing extra-information coming from data (e.g. positions of pixels).
Near pixels have a high likelihood to represent the same content. So near pixels are highly correlated.



When the image is passed to a FCL, we flatten the picture and lose all this information.

In general images are tensors (so multi-channel matrices), depending on the nature of the image:

- grayscale: 2d matrix (2d tensor)
- color rgb: 3 channels with a 2d matrix each (3d tensor)
- video stream: 4d tensor

Typically images have **lots of parameters** (e.g. a 640x480 image has 307.200 pixels), so if we pass this input image to a FCL then we would have 307.200 neuron units: high number of parameters.
With convolutions we solve this problem, since in its general form, convolutions is a downsampling operation.

The main component of the convolution is the **kernel**, a filter used to extract features. The (few) parameters of the kernel are learnable (during the CNN training). The dimensions of the kernel are typically 3x3, 5x5 and 7x7.
The depth of the kernel must be equal to the depth of the image! This is a constraint.

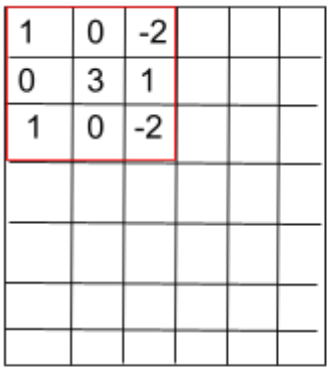
Convolution in formulas:

Cross-correlation

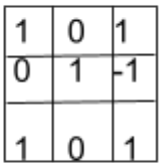
$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

We use this formula which is not the classical convolution because K is learnable, so we can adjust the parameters inside K.

Let's see how the convolution operation works:



input image



kernel

The kernel is passed through all the input dimensions and, at each convolution operation, the result is a scalar obtained from the sum of the products of each element in the image with the corresponding (same indices) element of the kernel. The window of the input image where the convolution is applied is called **receptive field.**
The first output (first sum between the 3x3 window of the image on the top left and the 3x3 kernel) gives:
1*1 + 0*0 + -2*1 + 0*0 + 3*1 + 1*1 + 1*1 + 0*0 + -2*1 = **2.**
Once this operation is done through all the image dimensions (so all the 3x3 window) then we get the final output (a 2d matrix), whose dimensions are typically smaller than the input one.
The output of a convolution is called a **feature map.** Applying once a convolution will produce one feature map.

We can choose two things when applying convolutions:

- **stride**: step (number of pixels) skipped when sliding the kernel over the image. The more the stride is the smaller the output becomes
- **padding**: this technique fills the input matrix with values (typically zeros) in order to have a bigger dimension of the output (because we need to slide over more windows of the input and so we have more output cells).

With stride = 1 and padding = same padding, we have that the output size is equal to the input size.

So, the output size depends on:

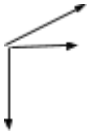
- input width and height
- kernel width and height
- stride
- padding

We can build now the formulas:

$$W_{output} = (W_{input} - W_{kernel} + 2 * Padding) / Stride$$

$$H_{output} = (H_{input} - H_{kernel} + 2 * Padding) / Stride$$

Also the convolution has dimensions which are independent from the input dimensions:

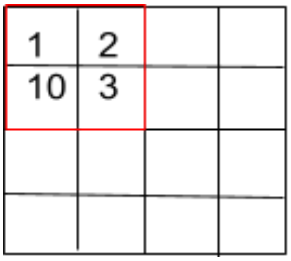
- **1d convolution:** the kernel slides over the input in ONE direction →
- **2d convolution**: the kernel slides over the input in TWO directions 
- **3d convolution**: the kernel slides over the input in THREE directions 

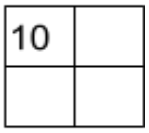
This application is done independently from the size of the input.
Applying a 2d Convolution on a 3d input image, will produce a 2d output.

Most of the time, we need more than one feature map, so we need to apply more than one convolution.
Let's call **d** the number of convolutions. The output of a convolution is now a tensor composed of d feature maps having size W_{output}, H_{output}.

Activation function
The convolution is a linear operation, so in order to introduce non-linearities we need to apply an activation function on the convolution output (which won't change the dimensions).

Pooling layer
After applying a convolution layer, a pooling layer is typically placed, which downsamples the input (which is the output of the convolution). It is an operator (max or average) with a dimension.





Let's see an example:
Here we have 2x2 max pooling.
The operator is applied over the 2x2 windows of the input (and from the 2x2=4 values the max is taken as result).

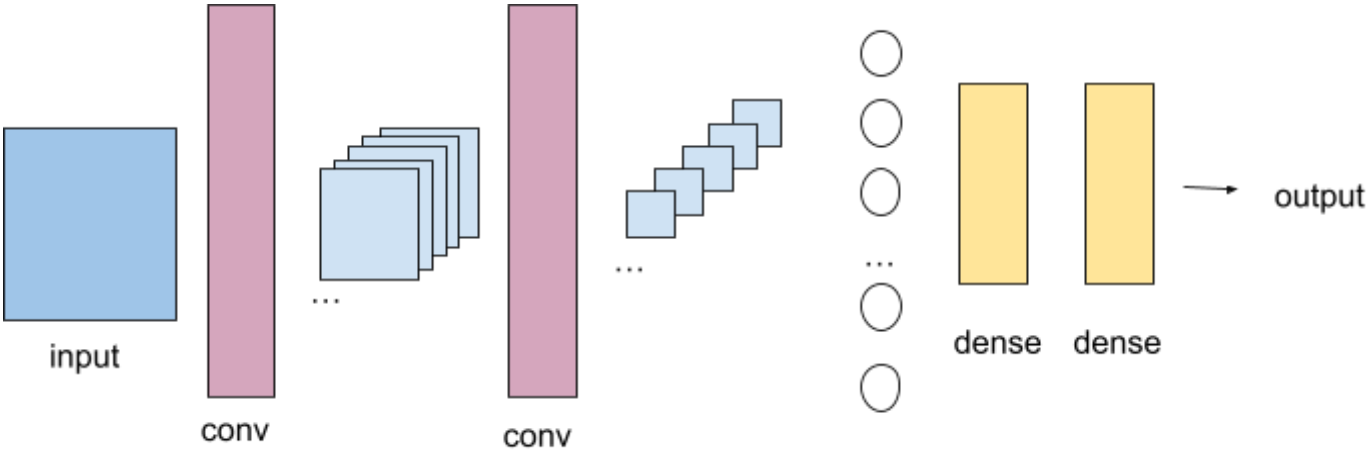
Even in pooling we might have stride and padding, so the output of a pooling layer is:

$$W_{output} = (W_{input} - W_{pooling} + 2 * Padding) / Stride$$

$$H_{output} = (H_{input} - H_{pooling} + 2 * Padding) / Stride$$

We can also add Regularization terms, Dropout, Batch Normalization…

A basic structure example of a CNN might be:



Where each conv layer has hyperparameters as the dimension of the kernel, the depth of the output (number of feature maps)...

Number of parameters of convolution

The total number of parameters for a kernel is:

$$d\ (number\ of\ feature\ maps)\ x\ w_k\ (width\ of\ the\ kernel)\ x\ h_w (height\ of\ the\ kernel)\ x\ d_k (depth\ of\ the\ kernel)$$

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

$d\ x\ w_k\ x\ h_w\ x\ d_k + d$, the last term is due to bias.

Number of parameters of CNN

The number of parameters of this network is the sum of the "convolutional part" and the "network part" with flatten, dense, … *The biggest contribution comes from the second part.*

The first part uses parameter sharing, which also reduces overfitting, because the convolution (at each step) is applied with the same kernel.

Sparse Connectivity
As said, the advantage of CNNs is the reduced number of parameters. We refer to this process as “sparse connectivity” which means that the outputs don’t depend on the entire input (as in FCL) but only on a few inputs (depending on the dimension of the kernel) extracted by the kernel.

Parameter sharing
Another important aspect in cnn is the fact that learnable parameters are shared to all the units: the same small window of the filter with dimensions nxn is applied through all the dimensions of the input. The weights of the filters are shared across the entire input space. This means that the same set of weights is used to compute the dot product between the filter and the input data, regardless of the spatial location. Parameter sharing significantly reduces the number of parameters in the model, making it computationally more efficient and helping to handle translation-invariant features. Parameter sharing allows the network to learn spatial hierarchies of features. Lower layers capture simple features, such as edges and textures, while higher layers combine these features to represent more complex patterns and objects.

Famous CNNs: AlexNet, LeNet, VGG, ResNet,

Applications of CNNs:
- Train a new model on a dataset
- Use pre-trained models (e.g., trained on ImageNet) to predict ImageNet categories for new images extract features to train another model (e.g., SVM)
- Refine pre-trained models on a new dataset (new set of classes)

Transfer Learning

The acquired knowledge on a source dataset is inferred to a new target domain.

This can be done in 2 ways:
- **fine-tuning**:
  (1) we train the network on the source domain and save weights
  (2) we start the training on the target domain with the weights obtained at step (1), so no random initialization

  We can both train all network parameters or freeze the first layers (parameters of the first layers won’t be updated). This is because the first layers are used to extract features and this has already been done on the source domain, so we can just exploit it.

  Pro Full advantage of the CNN!
  Con ‘Heavy’ training

- **CNN feature extractor**:
  1) extract features at a specific layer of CNN, usually: last convolutional layer (flattened), dense layers
  2) collect extracted features x’ of training/validation split and associate corresponding labels t in a new dataset D’
  3) train a new classifier C’ using dataset D’, e.g. ANN (extreme case of fine-tuning), SVM, linear classifier

4) classify extracted features of test set using the classifier C'

Pro No need to train the CNN!
Con Cannot modify features, source and target domains should be as 'compatible' as possible

Unsupervised learning

In the dataset, samples are only from the input space (no output information): $D = \{(x_n)_{n=1...,N}\}$.
We don't have anything about y, so we have to go through different techniques. A typical task of unsupervised learning is clustering.
Unsupervised learning algorithms determine mixed probability distributions from data.

Gaussian mixture model (one of the most famous model for UL)

Assumption: data distributions are from a set of k Gaussians.

The prob distribution is a weighted average of this k distributions following a gaussian (this is why mixture).

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Prior probability

π_k: probability of selecting that particular gaussian
μ, Σ**:** mean and variance of that particular gaussian

We can generate data from a mixture of gaussians. Each sample xn is generated by this process:

- choose gaussian k according to prior probability [π_1, ..., π_k]
- then generate an instance at random following that prior, μ, Σ

when we build an unsupervised dataset, we can't distinguish samples from their labels (because we dont know them) samples will belong to the same "type"/"class": (e.g. all points have the same color or shape in the plot...)

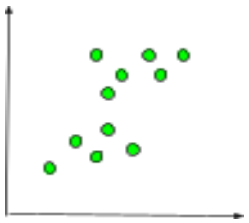
K-Means - one of the most general problem of clustering

We *assume* a gaussian model, and given a dataset we want only to estimate the K means of data generated from K Gaussian distributions.
We don't estimate covariance matrices because we *assume* all the gaussians have the same covariance matrix with a spherical shape. This is the assumption we make for the algorithm to work.

Input: D = {xn} for each n=1,..,N and K
Output: μ_1, …, μ_K

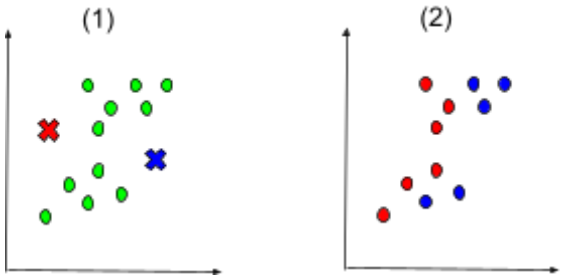
Algorithm:

Ex. of data (as you can see points have the same color), with K=2



1) If we don’t know it, make a decision on the value of K = num of clusters

2) Put an initial partition that classifies the data into k clusters: (e.g. (1) take the first k training samples as a single-element cluster, these points are called representative. (2) Assign each of the remaining (N-k) training samples to the cluster with the nearest centroid by computing the distance. after each assignment recompute the centroid of the new cluster).



3) Take each sample in sequence and compute its distance from the centroid of each of the clusters. If a sample is not currently in the cluster with the closest centroid, switch this sample to that cluster and update the centroid of the two clusters involved in the switch.

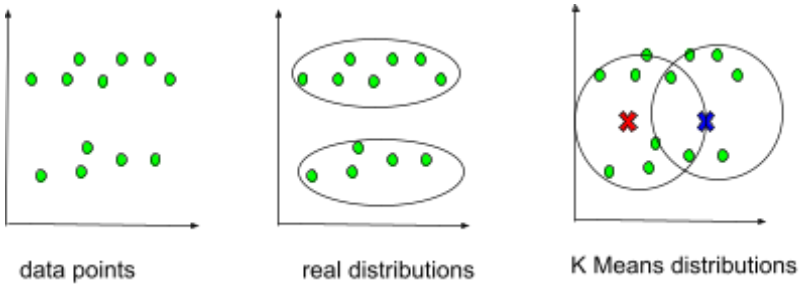
This process is on until we reach convergence: no new assignment (i.e. centroids are equal to the previous ones)

So centroids, which are not necessarily data points, are moving while the data points remain fixed.
In general the distances between points and the centroids decrease over time, so at some point we reach the convergence (the local minimum). The number of partitions into k clusters is finite. Convergence is guaranteed.

Issues of K Means

- everything is based on a distance function:
  when we have features really different from each other (e.g. salary/age),
  how we can define the distance ?
  so kmeans methods are sensitive to the distance function
- the value of k (we might not know k at first)
- sensitive to initial conditions
- not robust to outliers
- The result is a circular cluster shape because it is based on distance.

Due to these problems, we can know when K Means doesn't work:



(also with banana-shape points k means won't work)
The resulting 2 means (red and blue cross) arethe centers of a spherical (here circular because we have only 2 dimensions) shape because variances are assumed to be the same for each cluster since we use the Euclidean distance as proximity metric.

In order to avoid these problems we should use Kmeans clustering only if there are many data available; use median instead of mean and define better distance functions.

GMM

Given the KMeans problems, we want to use the gaussian mixture model:

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Let's introduce to this scheme, a set of variables z_k, called latent variables, one for each cluster. z_k in {0,1}.

Analysis of latent variables allows for a better understanding of input data.
We'll use them to measure the probability of one sample to belong to that particular distribution:

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

so:

$$P(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}$$

If i know that z_i is 1, then I understand that the sample has been generated from the gaussian i, so i can consider only z_i and discard the rest.
The vector z will be in the form: z = [0.02, 0.10, ..., 0.65], where each value z_i is the probability of a sample to be generated from the i-th Gaussian distribution.

By adding new variables, we are introducing something unknown, but we can estimate the parameters of the models, and z, with iterative methods.

When z are variables with 1-out-of-K encoding and P(zk = 1) = πk

$$P(\mathbf{x}) = \sum_{\mathbf{z}} P(\mathbf{z})P(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

GMM distribution P(x) can be seen as the marginalization of a distribution P(x, z) over variables z.
To get z and the parameters of the model, we'll use the posterior.

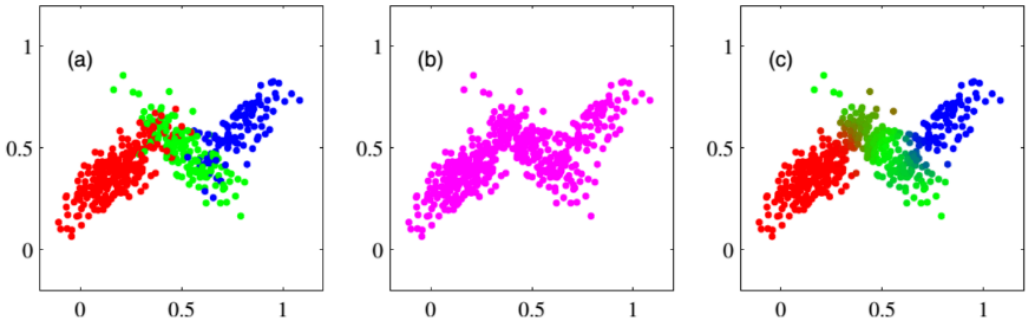
Let's define the posterior

$$\gamma(z_k) \equiv P(z_k = 1|\mathbf{x}) = \frac{P(z_k = 1)\, P(\mathbf{x}|z_k = 1)}{P(\mathbf{x})}$$

$$\gamma(z_k) = \frac{\pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

Note:
π_k: prior probability of z_k
$\gamma(z_k)$: posterior probability after observation of $\mathbf{x}$.

Example of GMM:



(a) The first figure shows the ground truth of the distribution of this particular dataset (unknown): P(x,z)
(b) The second picture shows the dataset (all points have the same color): P(x) "marginalized distribution"
© The third figure represents how data has been separated using the GMM. As it shows, some points, those on the boundaries of the red, green and blue regions, have mixed colors meaning that the vector z has some mixed probabilities (e.g. z_point = [0.4, 0.5, 0.1])

Expectation Maximization
It is an algorithm that overcomes some difficulties of k means
It makes no assumptions over covariances
But, it won't work with banana distributions

Problem formulation

Given data set D = {(xn)} and GMM:

$$P(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

estimate the parameters (π_k, μ_k, Σ_k).

We will use the maximization of the likelihood of the probability of data given the parameters of the model:

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

At maximum we'll have:

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) (\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T$$

$$\pi_k = \frac{N_k}{N}, \quad with \;\; N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

The **expectation step E** estimates the posterior given the parameters of the model
The **maximization step M** estimates the parameters of the model given the posterior

Algorithm

- Initialize $\pi_k^{(0)}, \boldsymbol{\mu}_k^{(0)}, \boldsymbol{\Sigma}_k^{(0)}$
- Repeat until termination condition $t = 0, \ldots, T$
  - **E step**

$$\gamma(z_{nk})^{(t+1)} = \frac{\pi_k^{(t)} \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} \mathcal{N}(\mathbf{x}_k; \boldsymbol{\mu}_j^{(t)}, \boldsymbol{\Sigma}_j^{(t)})}$$

  - **M step**

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)} \mathbf{x}_n$$

$$\boldsymbol{\Sigma}_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)} (\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(t+1)})^T$$

$$\pi_k^{(t+1)} = \frac{N_k}{N}, \quad with \;\; N_k = \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)}$$

We don't have a clear condition of termination, we can compare the parameters from one step to the other, or others. It converges to local maximum likelihood.

It works similar to kmeans:

- Can be generalized to other distributions
- more robust to dataset coming from gaussian with different covariances
- depends on initialization
- number of initial K is an issue again

There's a general formulation of EM: **General EM Problem**

Given:

- Observed data $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
- Unobserved latent values $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$
- Parametrized probability distribution $P(\mathbf{Y}|\boldsymbol{\theta})$, where
  - $\mathbf{Y} = \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$ is the full data $\mathbf{y}_n = \langle \mathbf{x}_n, \mathbf{z}_n \rangle$
  - $\boldsymbol{\theta}$ are the parameters

Determine:

- $\boldsymbol{\theta}^*$ that (locally) maximizes $E[\ln P(\mathbf{Y}|\boldsymbol{\theta})]$

Many uses:

- Unsupervised clustering
- Bayesian Networks
- Hidden Markov Models

Define likelihood function $Q(\boldsymbol{\theta}'|\boldsymbol{\theta})$ defined on variables $\mathbf{Y} = \mathbf{X} \cup \mathbf{Z}$, using observed $\mathbf{X}$ and current parameters $\boldsymbol{\theta}$ to estimate $\mathbf{Z}$

EM Algorithm:

Estimation (E) step: Calculate $Q(\boldsymbol{\theta}'|\boldsymbol{\theta})$ using current hypothesis $\boldsymbol{\theta}$ and observed data $\mathbf{X}$ to estimate probability distribution over $\mathbf{Y}$

$$Q(\boldsymbol{\theta}'|\boldsymbol{\theta}) \leftarrow E[\ln P(\mathbf{Y}|\boldsymbol{\theta}')|\boldsymbol{\theta}, \mathbf{X}]$$

Maximization (M) step: Replace hypothesis $\boldsymbol{\theta}$ by the hypothesis $\boldsymbol{\theta}'$ that maximizes this Q function

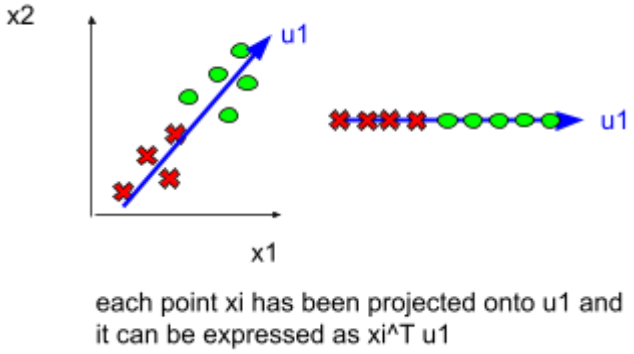
$$\boldsymbol{\theta} \leftarrow \underset{\boldsymbol{\theta}'}{\operatorname{argmax}}\, Q(\boldsymbol{\theta}'|\boldsymbol{\theta})$$

Dimensionality Reduction

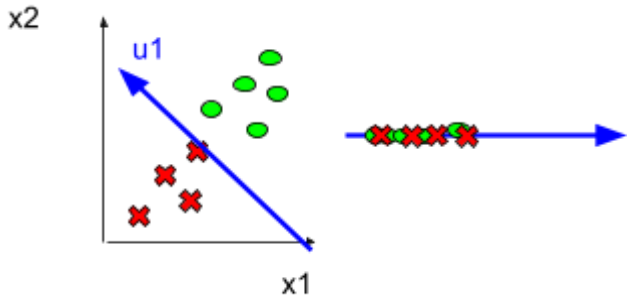
Inputs can have high dimensionality (e.g. images)
Can we characterize data into a less dimensional space better representative for the dataset as well?
Let's consider a transformation over an image like rotations building up a manifold, a surface of the input space.
The variability of the dataset is given by the way we transform data.

Principal Component Analysis
Applications: dimensionality reduction, data compression, data visualization, feature extraction.
Linear transformation from the input space into another space by projecting data into its more relevant dimensions. This linear transformation should preserve the information coming from the data. The transformation should also maximize the variance (so we should choose the right direction for the projection)
E.g. from 2 dimensions to 1 dimension:

This is a good direction on which project data:



This is not a good direction on which project data (this doesn't maximize the variance):



PCA - Variance Maximization

Given $D = \{x_n\} \in R^d$, unsupervised dataset

Goal: Maximize data variance after projection to some direction u_1 (unit vector, so $u_1^T u_1 = 1$)

The projected points will be in the form $x_n^T u_1$.

Mean value of data points: $\bar{x} = 1/N \sum_{n=1}^{N} x_n$

Data-centered matrix X (N × d):

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}_1 - \bar{\mathbf{x}})^T \\ \cdots \\ (\mathbf{x}_n - \bar{\mathbf{x}})^T \\ \cdots \\ (\mathbf{x}_N - \bar{\mathbf{x}})^T \end{bmatrix}$$

Mean of projected points: $\bar{x}^T u_1$

Variance of projected points:

$$\frac{1}{N}\sum_{n=1}^{N}[\mathbf{u}_1^T\mathbf{x}_n - \mathbf{u}_1^T\bar{\mathbf{x}}]^2 = \mathbf{u}_1^T S\mathbf{u}_1$$

with:

$$S = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T = \frac{1}{N}\mathbf{X}^T\mathbf{X}$$

So, Problem definition is **Maximize the projected variance**:

$max_{u_1} u_1^T S u_1$, subject to $u_1^T u_1 = 1$.

This is equivalent to unconstrained maximization with a Lagrange multiplier

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

The solution to this problem is Su1 = λ1u1, so u1 must be an **eigenvector** of S. Variance is maximal when u1 is the eigenvector corresponding to the largest eigenvalue λ1.
This is called the **first principal component.**

We can extend this process to other m dimensions, with m < d

- compute x̄: mean of the data
- compute S: covariance matrix of the dataset
- find m eigenvectors of S corresponding to the m largest eigenvalues

PCA - Error minimization

Consider a complete orthonormal D-dimensional basis such that:

$$\mathbf{u}_i^T\mathbf{u}_j = \delta_{ij}$$

with $\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$

Each data point can be written as

$$\mathbf{x}_n = \sum_{i=1}^{d}\alpha_{ni}\mathbf{u}_i$$

Using the orthonormality property we have $\alpha_{ni} = \mathbf{x}_n^T\mathbf{u}_i$, hence

$$\mathbf{x}_n = \sum_{i=1}^{d}(\mathbf{x}_n^T\mathbf{u}_i)\mathbf{u}_i$$

Goal: Approximate xn using a lower-dimensional representation. We can write:

$$\tilde{\mathbf{x}}_n = \sum_{i=1}^{m} z_{ni}\mathbf{u}_i + \sum_{i=m+1}^{d} b_i\mathbf{u}_i$$

Note: b_i terms do not depend on sample $\mathbf{x}_n$.
Evaluate approximation error (MSE) as

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

Minimizing w.r.t. z_{nj} and b_i, we get

$$z_{ni} = \mathbf{x}_n^T\mathbf{u}_i,\ i = 1, \ldots, m$$

$$b_i = \bar{\mathbf{x}}^T\mathbf{u}_i,\ i = M+1, \ldots, d$$

Using these expression we get:

$$\mathbf{x}_n - \tilde{\mathbf{x}}_n = \sum_{i=m+1}^{d} [(\mathbf{x}_n - \bar{\mathbf{x}})^T\mathbf{u}_i]\mathbf{u}_i$$

The overall approximation error becomes:

$$J = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=m+1}^{d} (\mathbf{x}_n^T\mathbf{u}_i - \bar{\mathbf{x}}^T\mathbf{u}_i)^2 = \sum_{i=m+1}^{d} \mathbf{u}_i^T S\mathbf{u}_i$$

Minimize the approximation error subject to constraint $u_i^T u_i = 1$:

$$\tilde{J} = \sum_{i=m+1}^{d} \mathbf{u}_i^T S\mathbf{u}_i + \lambda_i(1 - \mathbf{u}_i^T\mathbf{u}_i)$$

Setting derivative of a ui to zero we have:

$$S\mathbf{u}_i = \lambda_i\mathbf{u}_i$$

Hence ui is an eigenvector of S with eigenvalue λi. The approximation error is then given by:

$$J = \sum_{i=m+1}^{d} \lambda_i$$

This is minimized by selecting ui as the eigenvectors corresponding to the d − m smallest eigenvalues.
Note: Choosing d − m smallest eigenvalues of S corresponds to finding the m highest eigenvalues of S as in the maximum variance formulation.

So:
if we consider from 1 to m dimensions, then we are maximizing the variance (by finding the highest m eigenvalues of S)
if we consider from m+1 to d, then we are minimizing the reconstruction error (by finding the smallest eigenvalues of S).
We can also notice that the majority of the smallest eigenvalues is close to zero, so the solution of the decomposition is efficient having this sparse structure.

PCA for high-dimensional data
What if number of points is smaller than the dimensionality, i.e. N < d? At least d-N+1 eigenvalues of S are zero. In this case finding eigenvalues of S (d × d matrix) is inefficient.

Solution for N < d:

Let $\mathbf{X}$ be the $N \times d$ centered data matrix (i.e., n-th row is $(\mathbf{x}_n - \bar{\mathbf{x}})^T$)

and the corresponding covariance matrix:

$$S = \frac{1}{N}\mathbf{X}^T\mathbf{X}$$

The corresponding eigenvector equations is

$$\frac{1}{N}\mathbf{X}^T\mathbf{X}\mathbf{u}_i = \lambda_i\mathbf{u}_i$$

By left-multiplying by $\mathbf{X}$ we obtain

$$\frac{1}{N}\mathbf{X}\mathbf{X}^T(\mathbf{X}\mathbf{u}_i) = \lambda_i(\mathbf{X}\mathbf{u}_i)$$

By defining $\mathbf{v}_i = \mathbf{X}\mathbf{u}_i$ we have

$$\frac{1}{N}\mathbf{X}\mathbf{X}^T\mathbf{v}_i = \lambda_i\mathbf{v}_i$$

$\mathbf{X}\mathbf{X}^T$ has the same $N-1$ eigenvalues of $\mathbf{X}^T\mathbf{X}$ (the others are 0).

$\mathbf{X}\mathbf{X}^T$ is an $N \times N$ matrix whose eigenvalues can be computed efficiently.

Given the eigenvalues λ_i of $\mathbf{X}\mathbf{X}^T$, to find the eigenvectors we left-multiply by $\mathbf{X}^T$

$$\left(\frac{1}{N}\mathbf{X}^T\mathbf{X}\right)(\mathbf{X}^T\mathbf{v}_i) = \lambda_i(\mathbf{X}^T\mathbf{v}_i)$$

This makes clear that $(\mathbf{X}^T\mathbf{v}_i)$ is an eigenvector of S with eigenvalue λ_i.

To find $\mathbf{u}_i$ we have to normalize these eigenvectors such that $\mathbf{u}_i^T\mathbf{u}_i = 1$

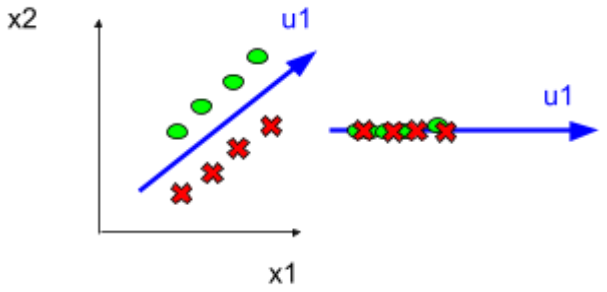
$$\mathbf{u}_i = \frac{1}{\sqrt{N\lambda_i}}\mathbf{X}^T\mathbf{v}_i$$

Summing up, when $N << d$:

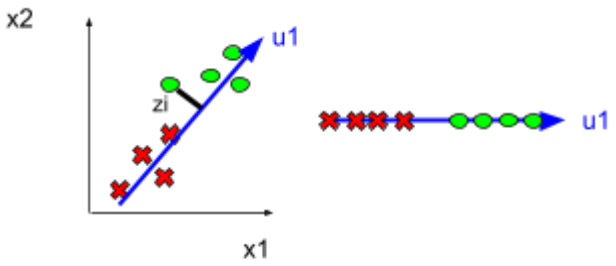
- Consider the centered data matrix $\mathbf{X}$
- Compute (efficiently) the $N-1$ eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of $\mathbf{X}\mathbf{X}^T$
- Let $\mathbf{u}_i = \frac{1}{\sqrt{N\lambda_i}}\mathbf{X}^T\mathbf{v}_i$

The so-obtained $N-1$ vectors $\mathbf{u}_i$ are the eigenvectors of $S = \mathbf{X}^T\mathbf{X}$, with non-null eigenvalue λ_i

WHEN NOT TO APPLY PCA:
In general it is useful in several applications, but there always exists cases where pca would lose all information of data:



It is also important to notice that with PCA we lose some information, even with a good projection:



the distance zi has been lost when projecting on u1

Probabilistic PCA

Linear Latent variable model

- Represent data $\mathbf{x}$ with lower dimensional latent variables $\mathbf{z}$
- Assume linear relationship

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu}$$

- Assume Gaussian distribution of latent variables $\mathbf{z}$

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

- Assume Linear-Gaussian relationship between latent variables and data

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

Marginal distribution

$$P(\mathbf{x}) = \int P(\mathbf{x}|\mathbf{z})P(\mathbf{z})d\mathbf{z} = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \mathbf{C})$$

with

$$\mathbf{C} = \mathbf{W}\mathbf{W}^T + \sigma^2\mathbf{I}$$

Posterior distribution

$$P(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mathbf{M}^{-1}\mathbf{W}^T(\mathbf{x} - \boldsymbol{\mu}), \sigma^2\mathbf{M})$$

with

$$\mathbf{M} = \mathbf{W}^T\mathbf{W} + \sigma^2\mathbf{I}$$

Maximum likelihood: given data $\mathbf{X}$

$$\underset{\mathbf{W},\boldsymbol{\mu},\sigma}{\operatorname{argmax}} \ln P(\mathbf{X}|\mathbf{W}, \boldsymbol{\mu}, \sigma^2) = \sum_{n=1}^{N} \ln P(\mathbf{x}_n|\mathbf{W}, \boldsymbol{\mu}, \sigma^2)$$

Setting derivatives to 0, we have a closed form solution

$$\boldsymbol{\mu}_{ML} = \bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$

$$\mathbf{W}_{ML} = ...$$

$$\sigma^2_{ML} = ...$$

$\mathbf{W}$ depends on the eigenvalues and eigenvectors of S (not trivial proof)

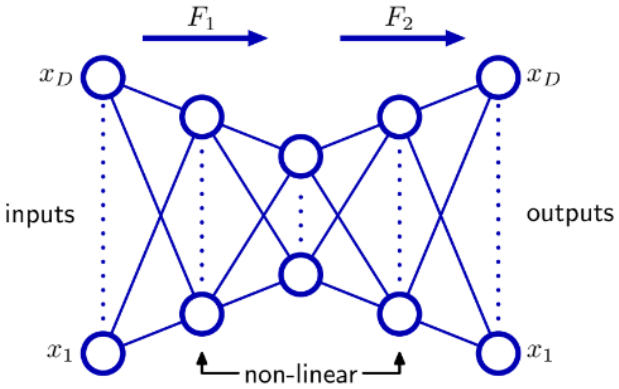
Maximum likelihood solution for the probabilistic PCA model can be obtained also with the EM algorithm.

PCA is a linear and unsupervised method, so it doesn't consider labels. There are cases in which the manifold is too complex to be represented with a linear transformation. So we should move towards techniques (non linear latent variable models) which use non-linear transformation (models based on neural networks).

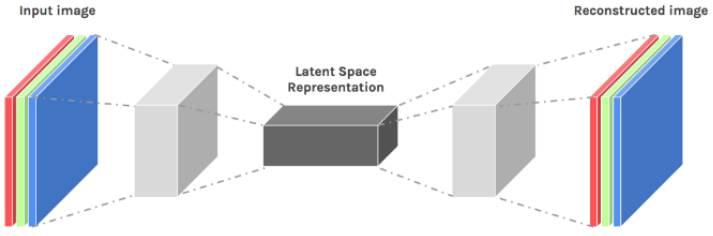
(from the exercise: we have to flatten the input to allow pca working)

Autoassociative Neural Networks (Autoencoders)

Autoencoders can be seen as a method for non-linear principal component analysis. So they are special architectures of NN used for dimensionality reduction. They can also be seen as compression architectures (with loss).
Autoencoders use two types of NN (encoder and decoder) combining the work. Again we don't have labels.
The goal is to reconstruct the input, so the network at the end should learn the identity function.
So, we force the network to pass through a lower representation vector of the input (also called bottleneck) and so to encode the input into another representation and then decode it into the original dimensions. The decoder returns the best approximation of the input, a manifold minimizing the loss function. The loss function is the reconstruction error.



DENSE LAYERS (FCL) architecture:
The input and the output layers are the same (let's say they have D nodes) and the hidden layers are reduced in size. Usually encoder and decoder networks are symmetric. Typically the networks are trained passing the same input to input layers and output layers. F1 can be seen as the function operating the dimensionality reduction.



CONVOLUTIONAL AUTOENCODERS
The autoencoder can also be designed as a CNN: The encoder part uses convolutions, while the decoder part uses inverse convolution. The convolution can be seen as the function operating the dimensionality reduction.

Autoencoders for anomaly detection
Task of detecting anomalies from data. Typically anomalies are rare (unbalanced data), so it is difficult to have a classification problem with anomaly detection. For this reason we consider the one-classification problem.

Problem

learn $f : X \to \{n, a\}$ with data set $D = \{(x_n, n)\}$

Train with only samples from normal class (x_n, n)

Given test $x' \notin D$, predict $\hat{f}(x') \in \{n, a\}$ (normal vs. abnormal)

Solution

The autoencoder learns normal data and computes the train loss. Since anomalies are rare and the AE has not learnt about them, it would not be able to encode them, this would produce a higher loss wrt the standard loss. This means that classification consists in considering a threshold value for the loss thanks to which we can classify a sample as normal (loss_sample < threshold) or abnormal (else).

Note: works even better with ensembles of autoencoders.

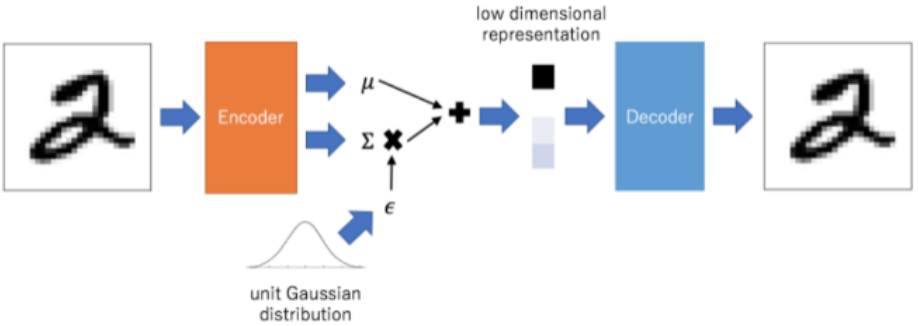
Generative models

Variational Auto Encoder (VAE)

We introduce a particular layer into the latent space which enforces the latent space to have a Gaussian distribution. So the autoencoder will learn a representation of the latent space which is Gaussian.
The idea to exploit is the locality principle: if two points are close (far) in the original space, then they would be close (far) in the latent space. So two distributions that are close (far) in the original space, then they would be close (far) in the latent space. Feed latent vectors and get realistic samples of P(X). Similar vector values should produce similar generated instances. To do this we need the encoder to produce a distribution instead of a vector. The decoder operates on samples from this distribution.

Problems:

1) how to produce a distribution?
   We use a parametric distribution (Gaussian) by adding a layer in the network:

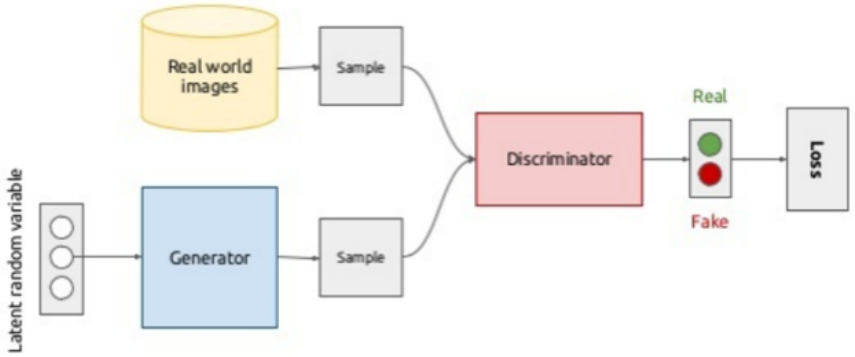


2) how to prevent degeneration?
   Add loss term based on KL-Divergence
3) Sampling operation (decoder) is not differentiable -> reparametrization

Taking a random latent representation z and applying a learnt decoder we can *generate* new samples (both with autoencoder and variational autoencoder).

GANs - Generative Adversarial Networks

We call the parts of Gans generator and discriminator.
The generator generates a new (fake) sample given a random z and passes it to the discriminator which also receives the real sample. The discriminator has to distinguish between real and fake (binary classifier).
The adversarial set consists in having a min-max game for which both discriminator and generator have to improve their performances: the generator wants to fool the discriminator, while the discriminator tries to discriminate as good as possible 'real' from 'fake' samples.



The training which is not so easy is done in an iterative and alternate way:

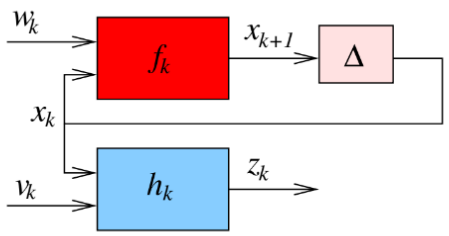
Repeat:

- Train the discriminator with a batch of data $\{(x_n, Real)\} \cup \{(x'_m, Fake)\}$, where x_n comes from the data set, while x'_m are images generated from the generator with random values of the latent variable.
- Train the generator by using the entire model (generator + discriminator) with discriminator layers fixed (i.e., not trainable) with a bacth of data $\{(r_k, Real)\}$, where r_k are random values of the latent variable.

Unfortunately, GANs can be used to attack ML models.

Markov Decision Processes and Reinforcement Learning

Dynamic System



x: state
z: observations
w, v: noise

f: state transition model
h: observation model

A dynamic system evolves over time through the use of a transition function that determines the evolution of the state (the representation of the system, seen as an instantaneous snapshot of the system itself).

The evolution also depends on some external inputs. In practical actions and other conditions.

In many conditions, we can not access the state, so we add sensors to the dynamic system and extract information from them. (observation model).

The problem we consider is that the functions are unknown.
So given a dynamic system evolving in some unknown mode, how can we learn information about it? How can we know when the system reaches a goal? How can we drive the system making the optimal choices to reach it?

we distinguish between:

- reasoning: given the model (f,h) and the current state x_k, predict the future (x_{k+T}, z_{k+T})
- learning: given past experience ($z_{0:k}$), determine the model (f,h).

State of a Dynamic System
The state is very important: it contains all the information we need for our task. It encodes the knowledge used to predict the future, gathered through operation and to pursue the goal.

Observability of states

We first assume states are fully observable: the agent can always look at the red box, it is the identity function. We can observe directly x. At any time we can read x(t).

The decision making problem for an agent is to decide which action must be executed in a given state. So, The agent has to *compute* the function: $\pi: X -> A$. It maps states into actions.

When the model of the system is unknown, then the agent has to *learn* the function π.

Supervised vs Reinforcement

To learn this function, I need input, output pairs (a dataset).
But how to define A, the best action reaching the goal, in the dataset?
In supervised learning it is difficult to collect a dataset D = {(xi, yi)} in which we can also have the optimal move for each configuration i (yi). So we move to a reinforcement learning approach.
In Reinforcement Learning, we consider a particular structure of the dataset in the form:

{< x1, a1, r1, …, xn, an, rn>} for a particular instance i. **This is the difference between reinforcement and supervised dataset.** r1, …, rn are rewards. The final dataset will have lots of these instances.
Here a1, …, an **are not** the best actions: they are random.

Dynamic System Representation

X: set of states

- explicit discrete and finite representation $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$
- continuous representation $\mathbf{X} = F(\ldots)$ (state function)
- probabilistic representation $P(\mathbf{X})$ (probabilistic state function)

A: set of actions

- explicit discrete and finite representation $\mathbf{A} = \{a_1, \ldots, a_m\}$
- continuous representation $\mathbf{A} = U(\ldots)$ (control function)

δ: transition function

- deterministic / non-deterministic / probabilistic

Z: set of observations

- explicit discrete and finite representation $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_k\}$
- continuous representation $\mathbf{Z} = Z(\ldots)$ (observation function)
- probabilistic representation $P(\mathbf{Z})$ (probabilistic observation function)

Markov Property

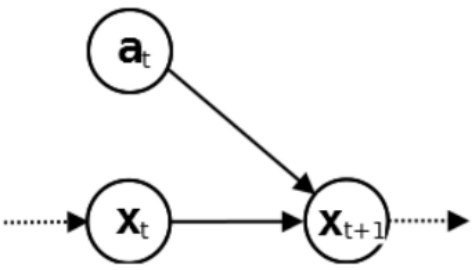
It is an assumption we make on the system: once the current state is known I can forget all the past -> the current state contains all the information I need to predict the future. I can ignore the history. The future action will depend on the current state.
The future is conditionally independent from the past once the present is known:

$$P(x_{t+1}|x_t, x_{t-1}, \ldots) = P(x_{t+1} | x_t)$$

Markov Decision Processes (MDP)

It represents the decision we take for making decisions.

Graphical model



We assume that the states are fully observable.
This can be seen as a partial Markov chain where x_t+1 depends only on x_t and a_t (2 entering arrows), so it doesn't depend on other history.

A MDP is a process or model of a dynamic system evolving over time where we want to make decisions (actions) to reach a goal using the Markov assumption.

Deterministic transitions

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

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \to \mathbf{X}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \to \Re$ is a reward function

Markov property: $\mathbf{x}_{t+1} = \delta(\mathbf{x}_t, a_t)$ and $r_t = r(\mathbf{x}_t, a_t)$
Sometimes, the reward function is defined as $r : \mathbf{X} \to \Re$

Deterministic transition means that the outcome of a particular action in a specific state is completely determined and predictable.

Non-deterministic transitions

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

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $\delta : \mathbf{X} \times \mathbf{A} \to 2^{\mathbf{X}}$ is a transition function
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \to \Re$ is a reward function

As output we have a set of possible outputs.

Stochastic transitions

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

- $\mathbf{X}$ is a finite set of states
- $\mathbf{A}$ is a finite set of actions
- $P(\mathbf{x}'|\mathbf{x}, \mathbf{a})$ is a probability distribution over transitions
- $r : \mathbf{X} \times \mathbf{A} \times \mathbf{X} \to \Re$ is a reward function

Set of finite states, actions, probability distribution over x and a reward computed as a function returning a real value.

X=state, A = action we take, X = next state

we don’t know how the probability evolves over time (and also the reward)

However, since there is fully observability, after executing the action then we can fully observe the state x' coming after having executed the action.

MDP Solution Concept

Now, we want to compute a policy, the behavior of the agent, as a function:
$\pi: X -> A$, we want this function to be optimal so for each state x ∈ X, π(x) ∈ A is the optimal action to be executed in state x.
optimality = maximize the cumulative reward.

Value Function

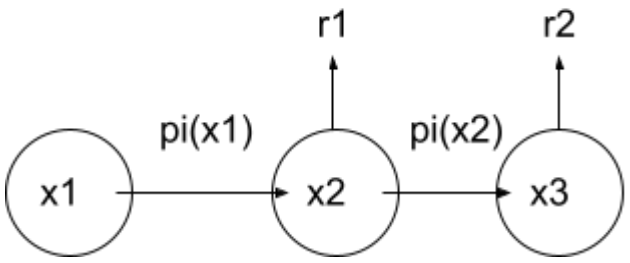
Now we introduce the following concept of discount factor for which we penalize future rewards (because in future there will always be a risk for the reward to happen)
Optimality is defined with respect to maximizing the expected value of the cumulative discounted reward in the non-deterministic / stochastic case: (in case of deterministic transitions we just have the cumulative reward because there's no randomness)

Deterministic case

$$V^{\pi}(\mathbf{x}) \equiv r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots$$

Non-deterministic/stochastic case:

$$V^{\pi}(\mathbf{x}) \equiv E[r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots]$$



We consider the argument of the expected value as a random process.

Optimal Policy

Optimal policy: $\pi^* \equiv \mathrm{argmax}_{\pi}\, V^{\pi}(\mathbf{x})\,,\ \forall \mathbf{x} \in \mathbf{X}$

So, the agent will prefer a shorter path to get the reward.

Optimal policy

π^* is an **optimal policy** iff for any other policy π

$$V^{\pi^*}(\mathbf{x}) \geq V^{\pi}(\mathbf{x}), \forall \mathbf{x}$$

For infinite horizon problems (we don't put a limit on the size of the evolution or we make the system in a way that the agent doesn't know when the episode ends, it will discover it after the execution of the action), a stationary MDP always has an optimal **stationary policy.** (transition and reward functions don't change over time)

Reasoning and Learning in MDP (skipped)

Problem: MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$

Solution: Policy $\pi : \mathbf{X} \rightarrow \mathbf{A}$

If the MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$ is completely known $\rightarrow$ reasoning or planning

If the MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$ is not completely known $\rightarrow$ learning

Note: Simple examples of reasoning in MDP can be modeled as a search problem and solved using standard search algorithm (e.g., A*).

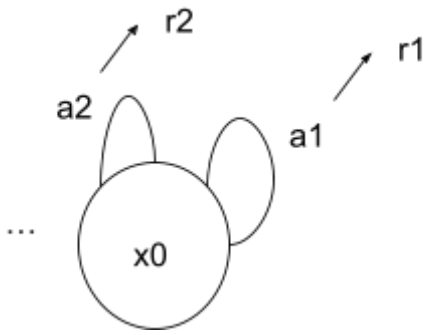
One-state Markov Decision Processes (MDP)

$$MDP = \langle \{\mathbf{x}_0\}, \mathbf{A}, \delta, r \rangle$$

- $\mathbf{x}_0$ unique state
- $\mathbf{A}$ finite set of actions
- $\delta(\mathbf{x}_0, a_i) = \mathbf{x}_0, \ \forall a_i \in \mathbf{A}$ transition function
- $r(\mathbf{x}_0, a_i, \mathbf{x}_0) = r(a_i)$ reward function

Optimal policy: $\pi^*(\mathbf{x}_0) = a_i$

One state and a set of actions, so when executing an action we return always to the same state



Deterministic One-state MDP

If r(ai) is **deterministic and known**, then the reward is always the same.

Optimal policy: $\pi^*(\mathbf{x}_0) = \text{argmax}_{a_i \in \mathbf{A}}\, r(a_i)$

If r(ai) is **deterministic and unknown**, then

Algorithm:

1. for each $a_i \in \mathbf{A}$
   - **execute** a_i and **collect** reward $r_{(i)}$
2. Optimal policy: $\pi^*(\mathbf{x}_0) = a_i$, with $i = \text{argmax}_{i=1\ldots|\mathbf{A}|}\, r_{(i)}$

Note: exactly $|\mathbf{A}|$ iterations are needed.

|A| because since it's deterministic we need only one iteration per action

Non-Deterministic One-state MDP

If r(ai) is **non-deterministic and known**, then

Optimal policy: $\pi^*(\mathbf{x}_0) = \text{argmax}_{a_i \in \mathbf{A}}\, E[r(a_i)]$

If r(ai) is **non-deterministic and unknown**, then

Algorithm:

1. Initialize a data structure Θ
2. For each time $t = 1, \ldots, T$ (until termination condition)
   - **choose** an action $a_{(t)} \in \mathbf{A}$
   - **execute** $a_{(t)}$ and **collect** reward $r_{(t)}$
   - Update the data structure Θ
3. Optimal policy: $\pi^*(\mathbf{x}_0) = \ldots$, according to the data structure Θ

Note: **many** iterations ($T >> |\mathbf{A}|$) are needed.

Since it's non deterministic we need to test an action multiple times.

We assume here rewards to follow a Gaussian distribution, the data structure is used to see the highest mean of Gaussians.

If $r(a_i)$ is **non-deterministic** and **unknown** and $r(a_i) = \mathcal{N}(\mu_i, \sigma_i)$, then

Algorithm:

1. Initialize $\Theta_{(0)}[i] \leftarrow 0$ and $c[i] \leftarrow 0$, $i = 1...|\mathbf{A}|$
2. For each time $t = 1, \ldots, T$ (until termination condition)
   - **choose** an index $\hat{\imath}$ for action $a_{(t)} = a_{\hat{\imath}} \in \mathbf{A}$
   - **execute** $a_{(t)}$ and **collect** reward $r_{(t)}$
   - increment $c[\hat{\imath}]$
   - update $\Theta_{(t)}[\hat{\imath}] \leftarrow \frac{1}{c[\hat{\imath}]}\left(r_{(t)} + (c[\hat{\imath}] - 1)\Theta_{(t-1)}[\hat{\imath}] \right)$
3. Optimal policy: $\pi^*(\mathbf{x}_0) = a_i$, with $i = \text{argmax}_{i=1...|\mathbf{A}|}\, \Theta_{(T)}[i]$

We can see the outcome of the action after its execution, so we get a value distributed over the gaussian followed by the reward The update step tries to compute the mean of the gaussians *online* with the values. The data structure will accumulate an estimate of the means of the corresponding gaussian distribution for each π_i.

Experimentation Strategies
How actions are chosen by the agents?
Exploitation: select action a that maximizes Qˆ(x, a), the one which *is currently better,* in this way we reduce time in testing actions that are likely to be non optimal, but we can also go exploring the “bad side” and we will never reach the optimal.
Exploration: select random action a (with low value of Qˆ(x, a)), we explore bad approximations (far away from optimality) in order to have good approximations, but this process is very slow.
Think about Q as the data structure

So, we want a good trade off between exploitation and exploration.

ϵ-greedy strategy

0 <= ϵ <= 1,
select a random action with probability ϵ
select the best action with probability 1 − ϵ
ϵ can decrease over time -> after several iterations we are more confident with the accumulations and we can increase exploitation, at first we prefer exploration.
So, we are balancing between exploration and exploitation. ϵ is a critical parameter.

soft-max strategy
actions with higher Qˆ values are assigned higher probabilities, but every action is assigned a non-zero probability.

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

k > 0 determines how strongly the selection favors actions with high Qˆ values. k may increase over time (first exploration, then exploitation).

Learning with Markov Decision Processes

Given an agent accomplishing a task according to an MDP $\langle \mathbf{X}, \mathbf{A}, \delta, r \rangle$,
for which functions δ and r are **unknown** to the agent,

determine the optimal policy π^*

Note: This is not a supervised learning approach!

- Target function is $\pi : \mathbf{X} \rightarrow \mathbf{A}$
- but we do not have training examples $\{(\mathbf{x}_{(i)}, \pi(\mathbf{x}_{(i)}))\}$
- training examples are in the form $\langle (\mathbf{x}_{(1)}, a_{(1)}, r_{(1)}), \ldots, (\mathbf{x}_{(t)}, a_{(t)}, r_{(t)}) \rangle$

Example: k-Armed Bandit

One state MDP with k actions: $a_1, \ldots, a_k$.

Stochastic case: $r(a_i) = \mathcal{N}(\mu_i, \sigma_i)$ Gaussian distribution

Choose a_i with ϵ-greedy strategy:
uniform random choice with prob. ϵ (exploration),
best choice with probability $1 - \epsilon$ (exploitation).

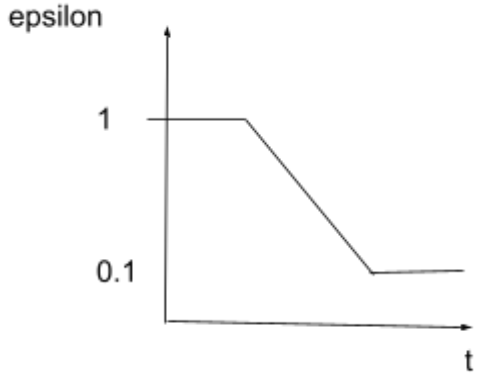
Training rule:

$$Q_n(a_i) \leftarrow Q_{n-1}(a_i) + \alpha[\bar{r} - Q_{n-1}(a_i)]$$

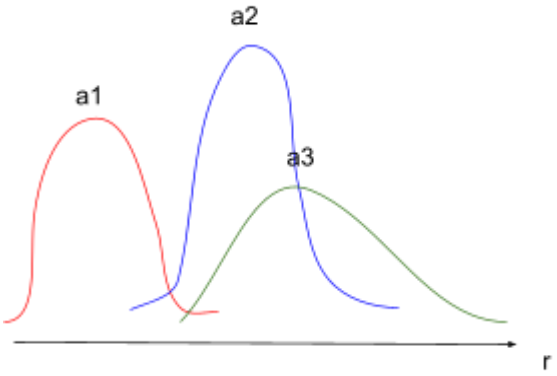
$$\alpha = \frac{1}{1 + v_{n-1}(a_i)}$$

with $v_{n-1}(a_i)$ = number of executions of action a_i up to time $n - 1$.

EX.
Consider having a single state x0 and 3 actions a1, a2, a3. We consider epsilon-strategy with epsilon-decay. $\varepsilon \; -> \; random\; state;\; 1 \; - \; \varepsilon \; -> \; "best"\; current\; action.$



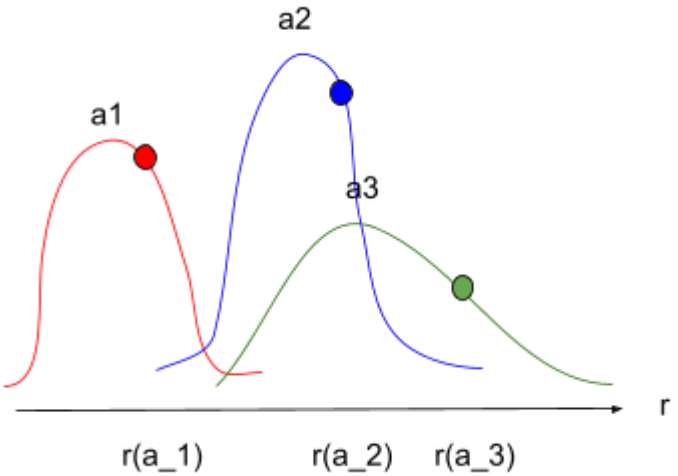
We want to compute the optimal policy. Distributions of a1, a2 and a3 are unknown:



The data structure, let's call it Q' (estimation of the real Q function), is initialized as a table with 3 (as many as the actions) entries filled with zeros. So no information is available at the beginning. Q' is:

0
0
0

At first we randomly choose action a1 and get a corresponding reward r(a1) and then update the first entry of the Q' table with the approximation of the mean of a1. Same thing for actions in a2 and a3.



By collecting more values, we get to good approximations of the means of the distributions. Q' will be:

μ_1
μ_2
μ_3

With exploitation and exploration both we have good approximations even for dynamic domain distributions. Indeed, with a fixed strategy we wouldn't have this flexibility.

MDP Learning Task

same approaches but includes state evolution considerations.
Since δ and r are not known, the agent cannot predict the effect of its actions. But it can execute them and then observe the outcome.

The learning task is thus performed by repeating these steps: (main block of RL algorithm)

- choose an action
- execute the chosen action
- observe the resulting new state
- collect the reward

(Then update the data structure)

Approaches to Learning with MDP

The optimal policy can be computed in 2 ways:

- **Value iteration**
  (estimate the Value function and then compute π)
  The value function contains the information about the expected comulative reward.
  The agent could learn the value function V_π* (x) (written as V*(x))

  From which it could determine the optimal policy:

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

Gamma is the discount factor.
r(x,a) is the immediate reward that I get executing a from x
delta(x,a) is the best I can do in the future

However, this policy cannot be computed in this way because **δ and r are not known**.
Even when I have V*.
So we need to define a Q function:

Q Function (deterministic case)

$Q^{\pi}(x, a)$: expected value when executing a in the state x and then act according to π.

$$Q^{\pi}(\mathbf{x}, a) \equiv r(\mathbf{x}, a) + \gamma V^{\pi}(\delta(\mathbf{x}, a))$$

$$Q(\mathbf{x}, a) \equiv r(\mathbf{x}, a) + \gamma V^*(\delta(\mathbf{x}, a))$$

If the agent learns Q, then it can determine the optimal policy without knowing δ and r.

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

Observe that

$$V^*(\mathbf{x}) = \max_{a \in \mathbf{A}} \{ r(\mathbf{x}, a) + \gamma V^*(\delta(\mathbf{x}, a)) \} = \max_{a \in \mathbf{A}} Q(\mathbf{x}, a)$$

Thus, we can rewrite

$$Q(\mathbf{x}, a) \equiv r(\mathbf{x}, a) + \gamma V^*(\delta(\mathbf{x}, a))$$

as

$$Q(\mathbf{x}, a) = r(\mathbf{x}, a) + \gamma \max_{a' \in \mathbf{A}} Q(\delta(\mathbf{x}, a), a')$$

From this we get the **update rule** used to estimate Q in RL algorithms. We can sample the information from the environment after the execution of the action and observe the actual successor states:

$$\hat{Q}(\mathbf{x}, a) \leftarrow \bar{r} + \gamma \max_{a'} \hat{Q}(\mathbf{x}', a')$$

where r̄ is the immediate reward and x' is the state resulting from applying action a in state x.
With this rule we can build the **"Q Learning Algorithm for Deterministic MDPs"**

1. for each $\mathbf{x}, a$ initialize table entry $\hat{Q}_{(0)}(\mathbf{x}, a) \leftarrow 0$
2. observe current state $\mathbf{x}$
3. for each time $t = 1, \ldots, T$ (until termination condition)
   - **choose** an action a
   - **execute** the action a
   - **observe** the new state $\mathbf{x}'$
   - **collect** the immediate reward $\bar{r}$
   - update the table entry for $\hat{Q}(\mathbf{x}, a)$ as follows:

$$\hat{Q}_{(t)}(\mathbf{x}, a) \leftarrow \bar{r} + \gamma \max_{a' \in \mathbf{A}} \hat{Q}_{(t-1)}(\mathbf{x}', a')$$

   - $\mathbf{x} \leftarrow \mathbf{x}'$
4. Optimal policy: $\pi^*(\mathbf{x}) = \text{argmax}_{a \in \mathbf{A}} \hat{Q}_{(T)}(\mathbf{x}, a)$

Convergence in deterministic MDP

$\hat{Q}_n(\mathbf{x}, a)$ underestimates $Q(\mathbf{x}, a)$

We have: $0 \leq \hat{Q}_n(\mathbf{x}, a) \leq \hat{Q}_{n+1}(\mathbf{x}, a) \leq Q(\mathbf{x}, a)$

Convergence guranteed if all state-action pairs visited infinitely often

This means that with high n (high number of experiments) then we are likely to have a really good approximation of Q.
Convergence is **not** guaranteed when at some point (at iteration n) anymore pairs of state-action are not visited for some reasons. Every action has to be executed at every time.

So for example, when we have an epsilon-greedy strategy with epsilon decay, if epsilon goes to **zero** at some point, then convergence is not guaranteed. epsilon = 0 means that no random action is chosen.

An (optimal) policy is a function mapping every state to one (best) action: so don't confuse it with a trace of execution!
when plotting a policy we need to ensure that from each state we make an action.

this is not valid (from the same state 2 actions are given)

Non-deterministic Case

Transition and reward functions are non-deterministic.

We define V, Q by taking expected values

$$\begin{aligned} V^{\pi}(\mathbf{x}) &\equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots] \\ &\equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}] \end{aligned}$$

Optimal policy

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(\mathbf{x}), (\forall \mathbf{x})$$

Definition of Q

$$\begin{aligned} Q(\mathbf{x}, a) &\equiv E[r(\mathbf{x}, a) + \gamma V^*(\delta(\mathbf{x}, a))] \\ &= E[r(\mathbf{x}, a)] + \gamma E[V^*(\delta(\mathbf{x}, a))] \\ &= E[r(\mathbf{x}, a)] + \gamma \sum_{x'} P(\mathbf{x}'|\mathbf{x}, a) V^*(\mathbf{x}') \\ &= E[r(\mathbf{x}, a)] + \gamma \sum_{x'} P(\mathbf{x}'|\mathbf{x}, a) \max_{a'} Q(\mathbf{x}', a') \end{aligned}$$

Optimal policy

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

Non-deterministic Q-learning

Q learning generalizes to non-deterministic worlds with training rule

$$\hat{Q}_n(\mathbf{x}, a) \leftarrow \hat{Q}_{n-1}(\mathbf{x}, a) + \alpha[r + \gamma \max_{a'} \hat{Q}_{n-1}(\mathbf{x}', a') - \hat{Q}_{n-1}(\mathbf{x}, a)]$$

which is equivalent to

$$\hat{Q}_n(\mathbf{x}, a) \leftarrow (1 - \alpha)\hat{Q}_{n-1}(\mathbf{x}, a) + \alpha[r + \gamma \max_{a'} \hat{Q}_{n-1}(\mathbf{x}', a')]$$

where

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

$visits_n(\mathbf{x}, a)$: total number of times state-action pair $(\mathbf{x}, a)$ has been visited up to n-th iteration

Convergence in non-deterministic MDP

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

Non-deterministic Q-learning also converges when every pair state, action is visited infinitely often

Other algorithms for non-deterministic learning
Temporal Difference (skipped), SARSA

SARSA
Different update (no max) based on “on-policy” method: a’ is chosen by the current policy (a’ is not executed).

SARSA is based on the tuple $< s, a, r, s', a' >$ ($< \mathbf{x}, a, r, \mathbf{x}', a' >$ in our notation).

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

a' is chosen according to a policy based on current estimate of Q.

On-policy method: it evaluates the current policy

Estimate the Q function: $\hat{Q}(\mathbf{x}, a) \approx Q(\mathbf{x}, a)$

However, we just need

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

Evaluating Reinforcement Learning Agents

Cumulative reward plot may be very noisy (due to exploration phases).
A better approach could be:

Repeat until termination condition:
1 Execute k steps of learning
2 Evaluate the current policy πk
(average and stddev of cumulative reward obtained in d runs with no exploration)

Domain-specific performance metrics can also be used.

Remarks on explicit representation of Q

Explicit representation of Qˆ table may not be feasible for large models.
Algorithms perform a kind of rote learning. No generalization on unseen state-action pairs.
Convergence is guaranteed only if every possible state-action pair is visited infinitely often.

Use function approximation:

$$Q_\theta(\mathbf{x}, a) = \theta_0 + \theta_1 F_1(\mathbf{x}, a) + \ldots + \theta_n F_n(\mathbf{x}, a)$$

Use linear regression to learn $Q_\theta(\mathbf{x}, a)$.

Use a neural network as function approximation and learn function Q with Backpropagation.
Implementation options:

- Train a network, using (x, a) as input and Qˆ(x, a) as output
- Train a separate network for each action a, using x as input and Qˆ(x, a) as output
- Train a network, using x as input and one output Qˆ(x, a) for each action

- **Policy iteration**
(estimate directly π instead of Q or V)

Parametric representation of π: $\pi_\theta(\mathbf{x}) = \max_{a \in \mathbf{A}} \hat{Q}_\theta(\mathbf{x}, a)$

Policy value: $\rho(\theta) =$ expected value of executing π_θ.

Policy gradient: $\Delta_\theta \rho(\theta)$

Policy Gradient Algorithm

Policy gradient algorithm for a parametric representation of the policy $\pi_\theta(\mathbf{x})$

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

Policy Gradient Algorithm

(for robot learning)

Estimate optimal parameters of a controller $\pi_\theta = \{\theta_1, ..., \theta_N\}$, given an objective function F.
Method is based on iterating the following steps:
1) generating perturbations of π_θ by modifying the parameters
2) evaluate these perturbations
3) generate a new policy from "best scoring" perturbations

General method

$\pi \leftarrow$ *InitialPolicy*
while *termination condition* **do**
    compute $\{R_1, ..., R_t\}$, *random perturbations of* π
    evaluate $\{R_1, ..., R_t\}$
    $\pi \leftarrow$ *getBestCombinationOf*($\{R_1, ..., R_t\}$)
end while

Note: in the last step we can simply set $\pi \leftarrow \text{argmax}_{R_j} F(R_j)$
(i.e., hill climbing).

Perturbations are generated from π by

$$R_i = \{\theta_1 + \delta_1, ..., \theta_N + \delta_N\}$$

with δ_j randomly chosen in $\{-\epsilon_j, 0, +\epsilon_j\}$, and ϵ_j is a small fixed value relative to θ_j.

Combination of $\{R_1, ..., R_t\}$ is obtained by computing for each parameter j:
- $Avg_{-\epsilon,j}$: average score of all R_i with a negative perturbations
- $Avg_{0,j}$: average score of all R_i with a zero perturbation
- $Avg_{+\epsilon,j}$: average score of all R_i with a positive perturbations

Then define a vector $A = \{A_1, ..., A_N\}$ as follows

$$A_j = \begin{cases} 0 & \text{if } Avg_{0,j} > Avg_{-\epsilon,j} \text{ and } Avg_{0,j} > Avg_{+\epsilon,j} \\ Avg_{+\epsilon,j} - Avg_{-\epsilon,j} & \text{otherwise} \end{cases}$$

and finally

$$\pi \leftarrow \pi + \frac{A}{|A|}\eta$$

Task: optimize AIBO gait for fast and stable locomotion

Objective function F

$$F = 1 - (W_t M_t + W_a M_a + W_d M_d + W_\theta M_\theta)$$

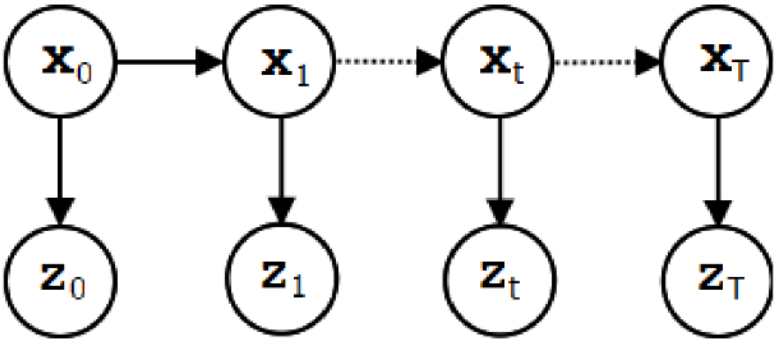
M_t: normalized time to walk between two landmarks
M_a: normalized standard deviation of AIBO's accelerometers
M_d: normalized distance of the centroid of landmark from the image center
M_θ: normalized difference between slope of the landmark and an ideal slope
W_t, W_a, W_d, W_θ: weights

Hidden Markov Models and Partially Observable MDPs

We will drop some of the assumptions taken before and we consider 2 cases:

- not-observable state of the dynamic system
- modeling some observable variables.

A dynamic system is something that evolves over time.
The markov chain represents the states linked in a way that we can consider the evolution of the states over time as a set of random variables x_t. We have considered that a state depends only on the previous state (and not on the past history). This is the *Markovian* assumption.
When we don't consider actions we can not drive the system towards some chosen evolution. So the system is not under our control.
When the states are not directly observable, I can assume the existence of other, random and observable, variables z_t, related to x_t and assume that each z_i depends on x_i. This case is called the hidden markov model (hmm). The variables z_i are called observations.
Graphically:



Definition of HMM:
model with a set of states X (not actions as in MDP, since the evolution is not under our control) and a set of observations Z. We have an initial probability distribution π_0 over the states.

Then we have 2 functions, one for the transition and one for the observations.
So we will also have two probability distributions:

- **P(x_t | x_(t-1))**: state transition probability -> This can be expressed as a discrete probability, so in a matrix form having as # rows = # cols = # states.
  Ex. x = {a,b,c} P(x_t = {a,b,c} | x_t-1 = {a,b,c})

P(x_t = a | x_t-1 = a)	P(x_t = a | x_t-1 = b)	P(x_t = a | x_t-1 = c)
P(x_t = b | x_t-1 = a)	P(x_t = b | x_t-1 = b)	P(x_t = b | x_t-1 = c)
P(x_t = c | x_t-1 = c)	P(x_t = c | x_t-1 = b)	P(x_t = c | x_t-1 = c)

- **P(z_t | x_t):** observation model
  same thing as before: if we have discrete observations and discrete states (so a matrix form with # cols = # states and # rows = # observations)

If we have continuous observations, (e.g. z is a Gaussian Distribution with a mean and a variance), we will have a matrix with one row and # states columns and each element of the matrix represent the gaussian with mean and variance associated to {a,b,c} respectively.

HMM Factorization

$P(x_{0:T}, z_{1:T}) = P(x_0)P(z_0|x_0)P(x_1|x_0)P(z_1|x_1)P(x_2|x_1)P(z_2|x_2)\dots$

HMM Inference

Given HMM = <X, Z, π_0>

We can have:

- filtering: process estimating the current (at time t) state system given the past observations
- smoothing: process estimating the value of one state in the past given the past observations

In the filtering problem we have to find $P(x_T = k \mid z_{1:T}) = \alpha^k{}_T / \sum_j \alpha^j{}_T$, so we have to compute alfa;

In the smoothie problem we have to find $P(x_t = k \mid z_{1:T}) = \alpha^k{}_t \beta^k{}_t / \sum_j \alpha^j{}_T \beta^k{}_t$, so we have to compute alfa and beta.

To compute alfa we use the forward algorithm which proceeds forward and compute all terms alfa; while to compute beta we use the backward algorithm.
(pictures of algorithms)
Both algorithms make the assumption that we know both transition and observation models.

What can we do if this information is not known?

case 1) we assume to observe the states at training time (where the system is open); transition and observation models can be estimated with statistical analysis.

case 2) when states can not be observed at training time, we compute a local maximum likelihood with an Expectation-Maximization method.

POMDP agent = Union of HMM and MDP.

It is a model using the Markov problem and states are partially observable, but here we have actions, so there is a decision to make. The action affects the evolution of the system.

A POMDP is the union of the elements of **MDP** and *HMM*: <X,A,Z, δ, r, o>
So we have X set of states, A set of actions, Z set of observations, δ probability distribution over transitions, r is a reward function and o is the probability distribution over observations.

Solution concept for POMDP
The solution is a policy but we don't know the states. So we have 2 options:

- map from history of observations to actions
- "belief state": probability distribution over the current state

Ex. Tiger Problem (see slides)