Machine Learning

1. Introduction

ML can be seen as learning function from samples or produce knowledge from data. Learning as search requires definition of **hypothesis space** and an algorithm to search solutions in this space.

ML problem is to learning a function (**target function**) $f : X \rightarrow Y$, given a dataset $D = \{(x, y)\}$ in such a way to find some approximation $\boxed{f(x)\,'\approx f(x)}$, $\forall\, x' \notin D$.

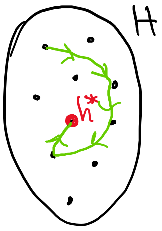
Depending on the dataset we have different problems:

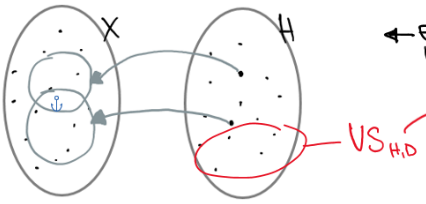
- **Supervised learning** when we have an output y for each sample y for the dataset $D = \{(x_i,y_i)\}$.
    - Classification: return the class to which a specific instance belong to.
    - Regression: approximate real-valued function

- Unsupervised if we do not have y_i, so we have $D = \{x_i\}$
    - Clustering

- **Reinforcement Learning** Having a triple of elements $D = (s_i,\ a_i,\ r_i)$ (state, action, reward), RL is the practice of learning a policy π (learn what is the action in the next state)

Call $\boldsymbol{H} = \{h_1,h_2,...,h_n\}$, the **hypothesis space** = the set of all possible approximation of the problem.
Given a target function $c(x)$ that we want to learn and a set of $H = \{h_1,h_2,...,h_n\}$, the goal is to find the best $\boldsymbol{h}^*$ so that $h^*(x) \approx c(x)$.
Any hypothesis that week approximates the target function over a sufficient large set of training examples will also approximate the target function well over other unobserved examples → h^* will predicts correct values of h(x') for instances x' with respect to the unknown values c(x').





← EACH h IN H GENERATE A SUBSPACE IN X

$VS_{H,D} = \{ h \in H \mid \text{CONSISTENT}(h, D) \}$

2. Evaluation and Estimators

TARGET

- **True Error** of *h* w.r.t. target function *f* and distribution *D* is the probability that *h* will misclassify an instance drawn at random according to *D*:

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

It cannot be computed.

WE CAN COMPUTE

- **Sample Error** of *h* w.r.t. the target function *f* and data sample *S* is the proportion of examples *h* misclassifies:

$$\boxed{error_s(h) = \frac{1}{|S|}\sum_{x \in S} \delta(f(x) \neq h(x))} \rightarrow \begin{cases} 1 & \text{IF } f(x) \neq h(x) \\ 0 & \text{OTHERWISE} \end{cases}$$

Accuracy: $accuracy(h) = 1 - error(h)$
The goal of a learning system is to be accurate in $h(x)$ $\forall x \notin S$.
If the $accuracy_s(h)$ is very high but the $accuracy_D(h)$ is low our system will not be useful.

→ We want $\boxed{error_S(h) \approx error_D(h)}$

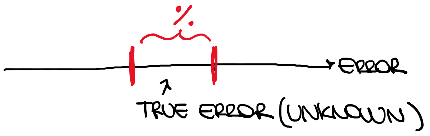
To guarantee this we consider:

- **Estimation Bias:** $bias \equiv E[error_S(h) - error_D(h)]$ BIAS → 0

$error_S(h)$ is random, how to compute?

1) Statistical method: **Confidence Intervals:**
Compute an interval that guarantee that the TrueError is inside it with a certain probability.

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



2) Build some methods to compute (unbias) estimation: **Estimators**
   1) Partition the data set $D = T \cup S$, $T \cap S = \emptyset$
   2) Compute a hypothesis h using training T
   3) Evaluate $error_S(h) = \frac{1}{n}\sum_{x \in S} \delta(f(x) \neq h(x))$

In general:

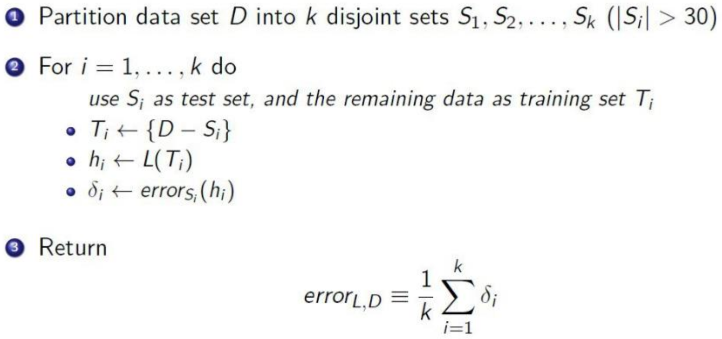
- Having more samples for training and less for testing improves performance of the model: potentially better model, but $error_S(h)$ $NOT \approx$ $error_D(h)$,
- Having more samples for evaluation and less for training reduces variance of estimation: $error_S(h) \approx error_D(h)$, but this value may be not satisfactory.

→ Trade off for medium sized datasets: 2/3 for training, 1/3 for testing.

Overfitting *h* overfits training data if, given another hypo *h'* we have that:

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

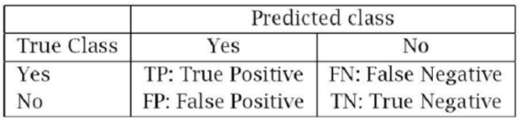
K-fold Cross Validation: we can use it to compare solutions and learning algorithm:



```
1. Partition data set $D$ into $k$ disjoint sets $S_1, S_2, \dots, S_k$ ($|S_i| > 30$)
2. For $i = 1, \dots, k$ do
      use $S_i$ as test set, and the remaining data as training set $T_i$
      • $T_i \leftarrow \{D - S_i\}$
      • $h_i \leftarrow L(T_i)$
      • $\delta_i \leftarrow errors_{S_i}(h_i)$
3. Return
      $error_{L,D} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i$
```

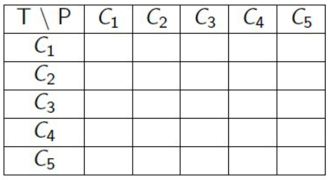
Other performance metrics:

- **Error rate** $\frac{|errors|}{|instances|} \quad = \frac{FN+FP}{TP+TN+FP+FN}$
- **Accuracy**= $1 - \ error\ rate$
- **Recall** $\quad = \frac{TP}{TP+FP}$
- **Precision** $=\frac{TP}{TP+FP}$
- **F1-score** $=\frac{2 \cdot Precision * Recall}{Precision+Recall}$



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

- **Confusion Matrix**: Report how many times an instance of class Ci is classified in class Cj Main diagonal contains accuracy for each class.
  Outside the diagonal: which classes are more often confused.

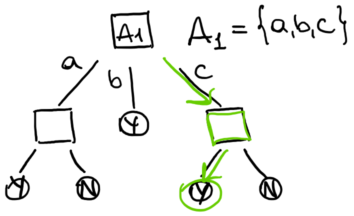


T \ P	C_1	C_2	C_3	C_4	C_5
C_1					
C_2					
C_3					
C_4					
C_5					

3. Decision Trees

DT can represent classification function by making decisions explicit. Given an instance space X coming from a set of attributes a DT has:

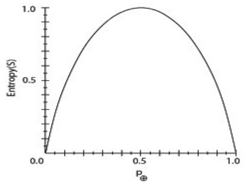
- an attribute for each internal node test
- a branch for each value of an attribute $a_{i,j} \in A_i$
- a leaf to which assigns a classification value $\{yes, no\}$



DT represent a disjunction of conjunctions of constraints on the attribute value of instances. In this way you can transform the tree into a rule for each path to a leaf node.

$example$: $\textit{if}\,(o = sunny) \wedge (H = high)$ $\textit{then}$ $PlayTennis = yes$

Entropy: measure the impurity of the set of samples S:

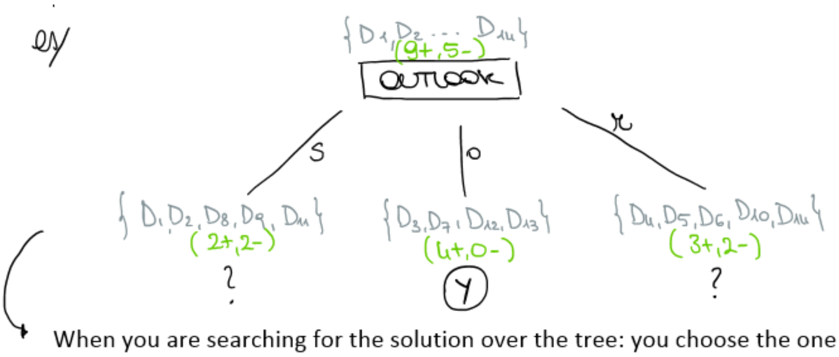


$$\boxed{Entropy(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-}$$

Where p_+ proportion of positive samples and $p_- = 1 - p_+$ the negative ones.

Information gain measures how well a given attribute separates the training examples according to their target classification. It is measured as the expected reduction in entropy of S caused by knowing the value of attribute A.

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



When you are searching for the solution over the tree: you choose the one with the highest information gain at each step.

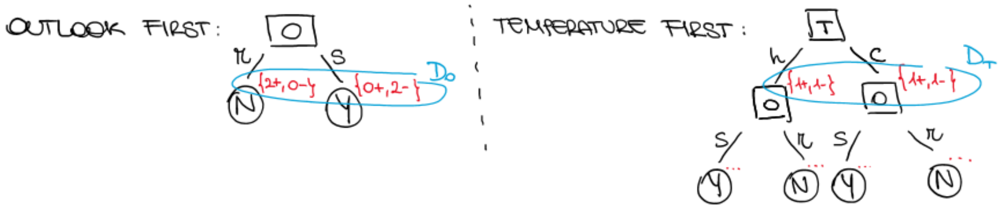
(NB: it is not greedy! Not the optimal)

↳ takes decision only on LOCAL information but anyway it build consistent tree (but it will be longer)

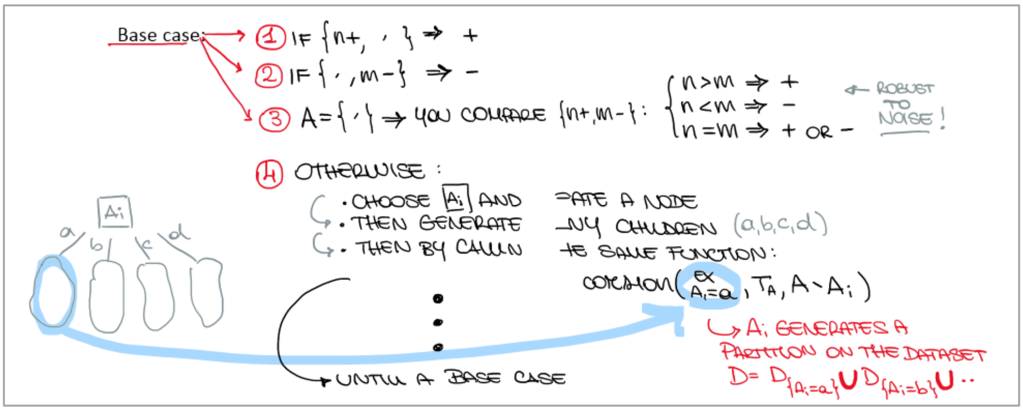
G1
G11
G2
G3
h*
...

ID3 Algorithm is an algorithm used to generate a DT from a dataset, knowing the *examples* (sample dataset), Target_*attribute* and the *Attributes*.

- Output DT depends on the attribute order!



- Optimality = the one that first reach the destination, so that guarantee that the tree is short → choose an attribute with 0 – or 0+ that guarantee the end of the tree
- ID3 algorithm selects the attribute that include highest information gain.


Base case:
① IF {n+, · } ⇒ +
② IF { · , m-} ⇒ -
③ A={ · } ⇒ YOU COMPARE {n+, m-}: n>m ⇒ + ; n<m ⇒ - ; n=m ⇒ + OR -
← ROBUST TO NOISE!
④ OTHERWISE:
· CHOOSE Ai AND =ATE A NODE
· THEN GENERATE _NY CHILDREN (a,b,c,d)
· THEN BY CALLIN HE SAME FUNCTION:
Ai
a
b
c
d
ID3(ex Ai=a, TA, A∖Ai)
↳ Ai GENERATES A PARTITION ON THE DATASET
D= D{Ai=a} ∪ D{Ai=b} ∪ ..
UNTIL A BASE CASE

Hypothesis Space Search by ID3:

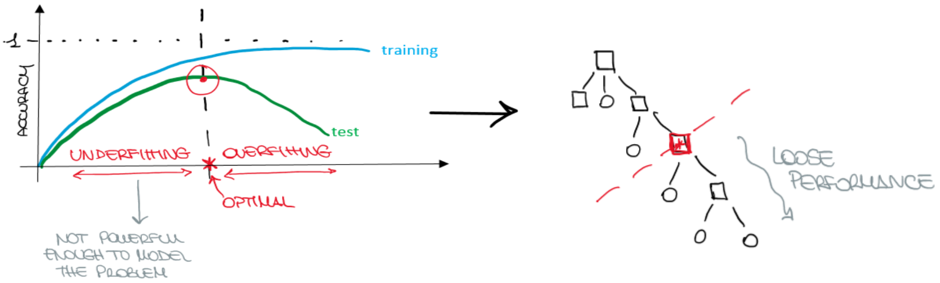
- Hypothesis space is complete (target concept is there!)
- Outputs a single hypothesis (cannot determine how many DTs are consistent!)
- No back tracking (local minima!)
- Statistically-based search choices (robust to noisy data!)
- Uses all the training examples at each step (not incremental!)

Issues in Decision Tree Learning

- Determining how deeply to grow the DT
- Handling continuous attributes
- Choosing appropriate attribute selection measures
- Handling training data with missing attribute values
- Handling attributes with different costs

Overfitting condition when the model completely fits the training data but fails to generalize the testing unseen data. Could happen that we continue the development of the tree only for one sample and get a deeper tree, with a deep branch only for that sample.

We must evaluate at each step the tree on a test set to see at which step we have a jeak of accuracy → after this step accuracy drop and we are overfitting.



To avoid tree overfitting:

- Stop growing when data split is not statistically significant
- grow a full tree and then **post-prune** (replace nodes not important with leafs).

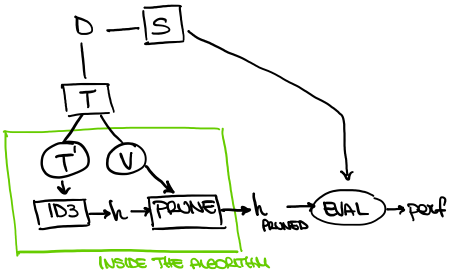
To determine correct tree size:

- 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.

Reduced-Error Pruning it produces smallest version of most accurate subtree

Split data into **Training** and **Validation** set. Do until further pruning is harmful (decreases accuracy):

1. Evaluate impact on validation set of pruning each possible node (remove all the subtree and assign the most common classification)
2. Greedily remove the one that most improves validation set accuracy

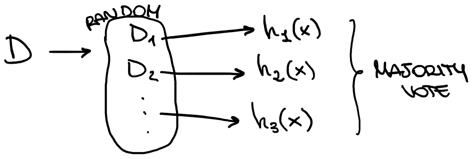


Rule Post-Pruning

Infer tree as well as possible (allowing for overfitting). Convert tree to equivalent set of rules. Prune each rule by removing any preconditions that result in improving its estimated accuracy. Sort final rules by their estimated accuracy and consider them in this sequence when classifying.

- greedy! So not optimal

Random Forest ensemble method that generates a set of DTs with some random criteria (bagging, feature selection, …) and integrates their values into a final result. Integration of results: majority vote (most common class returned by all the trees). Random Forests are less sensitive to overfitting.



4. Probability and Bayes

Representation of uncertainty with probabilities.

Prior Probability: it is the probability before collect any experience (Dataset is empty). Correspond to belief prior to arrival of any (new) evidence.

Posterior/Conditional probability: probability arise after the arrival of some evidence. If I know the outcome of a random variable, how will this affect probability of other random variables?

$$\boxed{P(a|b) \equiv \frac{P(a \wedge b)}{P(b)}}, \qquad if\ P(b) \neq 0$$

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

- A and B are **independent** iff one does not affect the other:

  $P(A|B) = P(A)$ or $P(B|A) = P(B)$ or $P(A,B) = P(A)P(B)$

  - if $X_1, \dots, X_n$ independent → $P(X_1, \dots, X_n) = P(X_1)P(X_2) \cdots P(X_n)$ reducing the size of the distribution from exponential to linear.
  - X is **Conditionally independent** from Y, given Z iff: $P(X|Y,Z) = P(X|Y)$

Chain rule: $P(X,Y,Z) = P(X|Y,Z)P(Y,Z) = P(X|Y,Z)P(Y,Z)P(Z) = P(X|z)P(Y|Z)P(Z)$

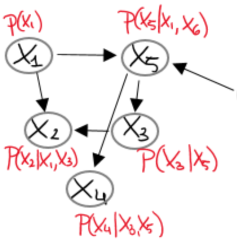
Bayes Rule

$$\boxed{P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}} \Leftrightarrow P(Y|X) = \frac{P(X|Y)P(Y)}{\underbrace{P(X)}_{='\alpha'}}$$

- For multivariable we can write: $P(Z|Y_1, \dots, Y_n) = \alpha\, P(Y_1, \dots Y_n|Z)P(Z)$
- If $Y_1, \dots . Y_n$ conditionally independent: $P(Z|Y_1, \dots, Y_n) = \alpha P(Y_1|Z) \cdots P(Y_n|Z)P(Z)$

We can represent **Bayesian Networks** for graphical notation for conditional independence assertions and hence for specification of full joint distribution:

- a set of nodes, one per variable
- a directed, acyclic graph (link "directly influences")
- a conditional distribution for each node given its parents: $P\big(X_i\big|Parents(X_i)\big)$



→ FOR X_6 I CONSIDER ONLY THE PRIOR

- $P(X_2|X_1,X_3) = P(X_2|X_1,X_3,\underbrace{X_4,X_5,X_6}_{\text{DOES NOT AFFECT PROBABILITY!}})$

→ BY BAYESIAN NETWORK I CAN COMPUTE $P(Z|Y_1 \dots Y_n)$

5. Bayes Learning

Bayesian learning uses Bayes' theorem to determine the conditional probability of a hypotheses given some evidence or observations.

- Provides practical learning algorithms:
  - **Naive Bayes learning** (examples affect prob. that a hypothesis is correct)
  - Combine prior knowledge (probabilities) with observed data
  - Make probabilistic predictions (new instances classified by weighted combination of multiple hypotheses)
  - Requires prior probabilities (often estimated from available data)
- Provides useful conceptual framework for evaluating other learning algorithms

Bayes Theorem: given $P(h)$ the prior probability of the hypothesis h, and $P(D)$ the prior probability of the training data D. The Bayes rule is:

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

Maximum a Posteriori probability (MAP)

When classifying new data we want to assign it the most probable hypothesis. To do so we can use **Maximum a posteriori** hypothesis h_{MAP}:

$$\boxed{h_{MAP} = \arg\max_{h\in H} P(h|D) = \arg\max_{h\in H} \frac{P(D|h)P(h)}{p(D)} = \arg\max_{h\in H} P(D|h)P(h)}$$

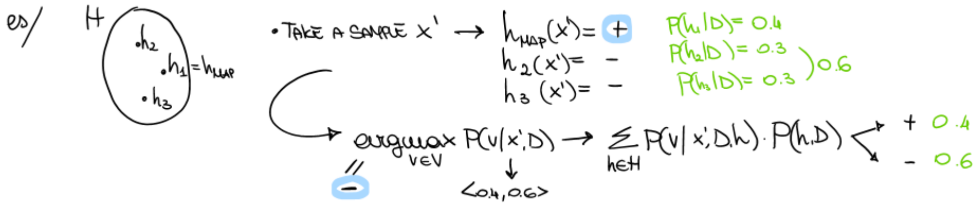
(removed the normalizing constant $P(D)$)

1. For each hypothesis $h \in H$, calculate the posterior probability $P(h|D)$
2. Output the hypothesis h_{MAP} with the highest posterior probability

Moreover if the prior distribution is **uniform**, i.e. $P(h_i) = P(h_j)$, $\forall i,j \in H$ we can use the **Maximum Likelihood** hypothesis h_{ML} and have:

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

We can estimate the maximum h_{MAP} by computing $P(h_i|D)$ for every $h_i \in H$ and then get the maximum, **but** <u>h_{MAP} return the most probable hypothesis, not the most probable classification</u>, so, given a new instance x' $h_{MAP}(x)$ might not the return the correct classification nor the most probable.



➔ So in general $h_{MAP} \neq argmax\, P(v|x', D)$

Bayes Optimal Classifier

The Bayes Optimal Classifier maximizes the probability that the new instance x' is classified correctly.

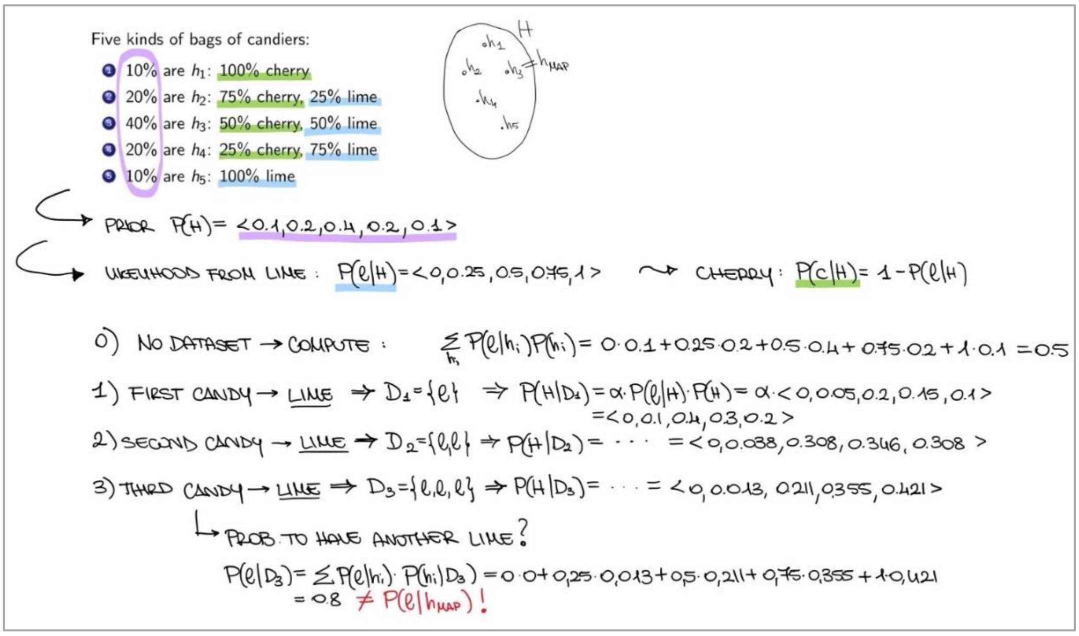
Given the target function $f: X \to V$ that maps an instance to a class v, a dataset D and a new instance x' we want to classify it correctly: $v^* = \hat{f}(x')$
In general: $v^* = \arg\max_{v \in V} P(v|x', D)$
Where $P(v_j|x',D)$ is the probability that x' belongs to the class v_j conditioned to the entire dataset D (every hypothesis).

If we consider $P(v_j|x',h_i)$ as the probability of a new instance x' to be classified as the class v_j by a hypothesis h_i. We have that:

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

And so the most probable class v_{OB} for a new instance x' would be:

$$v_{OB} = argmax_{v_i} \sum_{h_i} P(c_j|x', h_i)P(h_i|D)$$



Five kinds of bags of candiers:

1. 10% are h_1: 100% cherry
2. 20% are h_2: 75% cherry, 25% lime
3. 40% are h_3: 50% cherry, 50% lime
4. 20% are h_4: 25% cherry, 75% lime
5. 10% are h_5: 100% lime

H
h_1
h_2
$h_3 \neq h_{MAP}$
h_4
h_5

PRIOR $P(H) = <0.1, 0.2, 0.4, 0.2, 0.1>$

LIKELIHOOD FROM LIME : $P(\ell|H) = <0, 0.25, 0.5, 0.75, 1>$ → CHERRY : $P(c|H) = 1 - P(\ell|H)$

0) NO DATASET → COMPUTE : $\sum_{h_i} P(\ell|h_i)P(h_i) = 0 \cdot 0.1 + 0.25 \cdot 0.2 + 0.5 \cdot 0.4 + 0.75 \cdot 0.2 + 1 \cdot 0.1 = 0.5$

1) FIRST CANDY → LIME $\Rightarrow D_1 = \{\ell\} \Rightarrow P(H|D_1) = \alpha \cdot P(\ell|H) \cdot P(H) = \alpha \cdot <0, 0.05, 0.2, 0.15, 0.1>$
$= <0, 0.1, 0.4, 0.3, 0.2>$

2) SECOND CANDY → LIME $\rightarrow D_2 = \{\ell, \ell\} \Rightarrow P(H|D_2) = \cdots = <0, 0.038, 0.308, 0.346, 0.308>$

3) THIRD CANDY → LIME $\Rightarrow D_3 = \{\ell, \ell, \ell\} \Rightarrow P(H|D_3) = \cdots = <0, 0.013, 0.211, 0.355, 0.421>$

↳ PROB TO HAVE ANOTHER LIME?

$P(\ell|D_3) = \sum P(\ell|h_i) \cdot P(h_i|D_3) = 0 \cdot 0 + 0{,}25 \cdot 0{,}013 + 0{,}5 \cdot 0{,}211 + 0{,}75 \cdot 0{,}355 + 1 \cdot 0{,}421$
$= 0.8 \neq P(\ell|h_{MAP})$!

When we have to deal with a high hypothesis space, the Bayes Optimal Classifier is not practical anymore. A way to avoid computing every hypothesis is using conditional independence.

When X is conditionally independent of Y given Z:

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

Naive Bayes Classifier

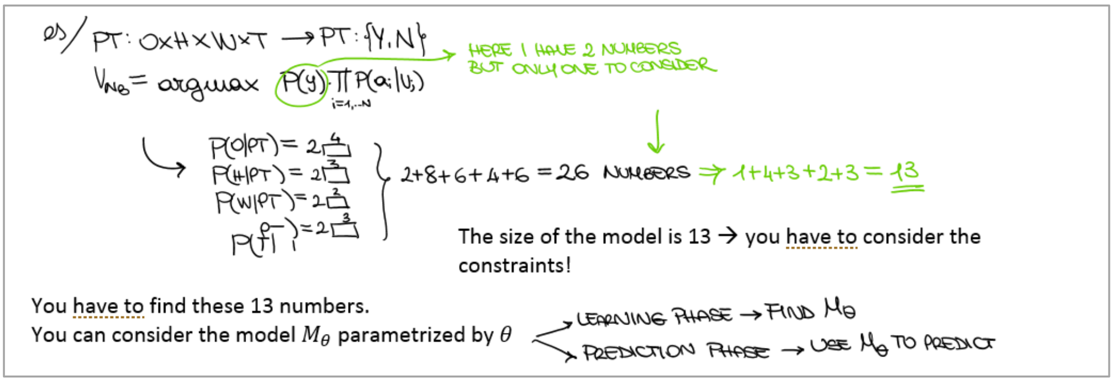
NBC uses conditional independence to approximate the solution.
Consider $f: X \rightarrow V$ where each instance x is described by attributes $< a_1, a_2, \dots, a_n >$.

$$V_{MAP} = arg \max_{v_j \in V} P(v_j | x, D) = arg \max_{v_j \in V} P(v_j | a_1, \dots, a_n, D) =$$

$$= argmax \frac{P(a_1, \dots, a_n | v_j, D)\, P(v_j, D)}{P(a_1, \dots a_n | D)} = argmax\, P(a_1, .., a_n | v_j, D) P(v_j | D)$$

assuming each a_i conditionally independent we get:

$$\boxed{V_{NB} = \arg\max_{v_j \in V} P(v_j | D) \prod_i P(a_i | v_j, D)} \neq V^*$$



es/ $PT: O \times H \times W \times T \rightarrow PT: \{Y, N\}$

$V_{NB} = argmax\, P(y) \cdot \prod_{i=1,..N} P(a_i | v_j)$

HERE I HAVE 2 NUMBERS BUT ONLY ONE TO CONSIDER

$P(O|PT) = 2\overset{4}{\square}$
$P(H|PT) = 2\overset{3}{\square}$
$P(W|PT) = 2\overset{2}{\square}$
$P(T| \;) = 2\overset{3}{\square}$

$2+8+6+4+6 = 26$ NUMBERS $\Rightarrow 1+4+3+2+3 = \underline{\underline{13}}$

The size of the model is 13 → you have to consider the constraints!

You have to find these 13 numbers.
You can consider the model M_θ parametrized by θ

- LEARNING PHASE → FIND M_θ
- PREDICTION PHASE → USE M_θ TO PREDICT

NB: if none of the training instances with target value v_j have attribute value a_i → : $\hat{P}(a_i | v_j, D) = 0$ → $P(v_j | D) \prod_i P(a_i | v_j, D) = 0$. In this case, to avoid the zero we can set a virtual prior to some arbitrary number that guarantee $\hat{P} > 0$

$$\hat{P}(a_i | v_j, D) = \frac{|\cdot| + mp}{|\cdot| + m}$$

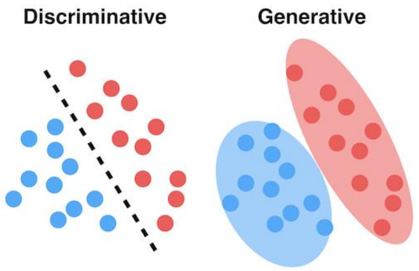
- p = prior estimate for P
- m = weight given to prior

6. Probabilistic Model for CLASSIFICATION

In statistical classification, there are two main models:

- In the case of **generative** models, to find the conditional probability $P(C_i | x, D)$, estimate the prior probability $P(C_i)$ and likelihood probability $P(x | C_i)$ with the help of the training data D and uses the Bayes Theorem to calculate the posterior probability $P(C_i\ | x)$ → e.g. ***naive Bayes classifier***

- In the case of **discriminative** models, to find the probability, directly assume some functional form for $P(C_i | \boldsymbol{x})$ and then estimate the parameters of $P(C_i | \boldsymbol{x})$ with the help of the training data. → e.g. ***logistic regression***



Probabilistic Generative Models

The posterior can be express as:

$$posterior = \frac{prior \ \times likelihood}{evidence} \rightarrow P(C_i|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|C_i)P(C_i)}{\sum_j P(\boldsymbol{x}|C_j)\,P(C_j)}$$

With $P(C_i) = \pi_i$ and $P(\boldsymbol{x}|C_i) = N(\boldsymbol{x}; \boldsymbol{\mu_i}, \sum)$ gaussian.

Assuming 2 classes C_1, C_2 and $D = \{(x_n, t_n)_{n=1}^N\}$ → $\begin{matrix} t_n \ = 1 \ \ if \ x_n \in C_1 \\ t_n \ = 0 \ if \ x_n \in C_2 \end{matrix}$

Let be $\begin{matrix} N_1 \ the \ n. of \ samples \ in \ D \in C_1 \\ N_2 \ the \ n. of \ samples \ in \ D \in C_2 \end{matrix}$

In this case $P(C_1|x) = \sigma(a) = \sigma(\boldsymbol{w}^T x + \boldsymbol{w_0})$ and $P(C_2|x) = 1 - P(C_1|x)$

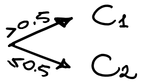
With $\begin{matrix} \boldsymbol{w} = \sum^{-1}(\mu_1 - \mu_2) \\ w_0 = -\frac{1}{2}\mu_1^T \sum^{-1} \mu_1 + \frac{1}{2}\mu_2^T \sum^{-1} \mu_2 + \ln\frac{P(C_1)}{P(C_2)} \end{matrix}$

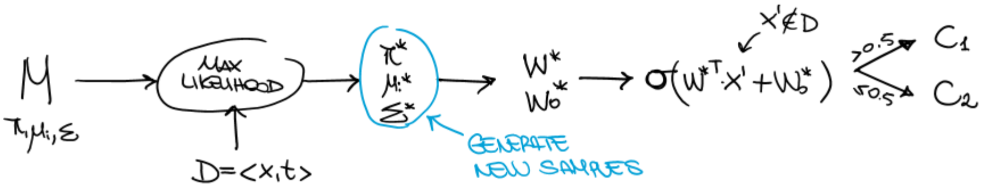
Then the optimal <u>maximum likelihood</u> is given by:

$$\pi^*, \boldsymbol{\mu_1^*}, \boldsymbol{\mu_2^*}, \Sigma^* = \arg \max_{\pi, \boldsymbol{\mu_1}, \boldsymbol{\mu_2}, \Sigma} P\big(\boldsymbol{t}\big|\pi, \boldsymbol{\mu_1}, \boldsymbol{\mu_2}, \Sigma, D\big)$$

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

Then the prediction of a new sample $x \notin D$ → $P(C_1|x') \ = \sigma(w^{*T}x' + w_0)$





Probabilistic Discriminative Models

without estimating the model parameters, estimate directly $P(C_k|\tilde{\boldsymbol{x}}, D) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$,

with $\tilde{x} = \binom{1}{x}$, $\tilde{\boldsymbol{w}} = \binom{w_0}{\boldsymbol{w}}$, $a_k = \boldsymbol{w^T x} + w_0 = \tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}}.$

The maximum likelihood: $\widetilde{w}^* = \arg \max_{\widetilde{w}} \ln P(\boldsymbol{t}|\widetilde{\boldsymbol{w}}, \boldsymbol{X})$

Logistic Regression

Given a target function $f: X \rightarrow C$ and a dataset D
Assume a parametric model for the posterior probability $P(C_k|\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{w}})$:

- $\sigma(\tilde{\boldsymbol{w}}^T\tilde{\boldsymbol{x}})$ if 2 classes
- $\frac{\exp(\tilde{\boldsymbol{w}}_k{}^T\tilde{\boldsymbol{x}})}{\sum_{j=1}^{k}\exp(\tilde{\boldsymbol{w}}_j{}^T\tilde{\boldsymbol{x}})}$ if k-classes

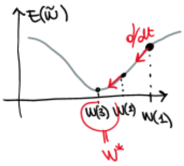
Define the Error function $E(\tilde{w})$ as the negative log-likelihood.

max log.likelihood ⟺ min[− log.likelihood] → E(w)
REPRESENTATION OF THE MODEL $\tilde{w}$

Solve the optimization problem: $\boxed{\tilde{\boldsymbol{w}}^* = \arg\min_{\tilde{w}} E(\tilde{\boldsymbol{w}})}$
Classify new sample $\tilde{x}'$ as C_{k^*} where $k^* = \arg\max_{k=1,\dots K} P(C_k|\tilde{x}', \tilde{w}^*)$

You can solve the minimization < *analytically* / *iterative* → ***Iterative re − weighted last squares***

Iterative re-weighted last squares: Apply Newton-Raphson iterative optimization for minimizing $E(\tilde{\boldsymbol{w}})$: $\nabla E(\tilde{\boldsymbol{w}}) = \sum_{n=1}^{N}(y_n - t_n)\,\tilde{\boldsymbol{x}}_n$



- PICK A RANDOM W
- ESTIMATE LOCAL DERIVATIVE
- COMPUTE THE NEXT POINT

. . . UP TO LOCAL MINIMUM.

→ Gradient descent step
$\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} - \boldsymbol{H}(\tilde{\boldsymbol{w}})^{-1}\nabla E(\tilde{\boldsymbol{w}})$
→ OUTPUT WILL BE W*

$\boldsymbol{H}(\tilde{w}) = \nabla\nabla \mathrm{E}(\tilde{\boldsymbol{w}})$ is the Hessian Matrix of $E(\tilde{\boldsymbol{w}})$.

Generalization:
Given a target function $f : X \rightarrow C$, and data set D

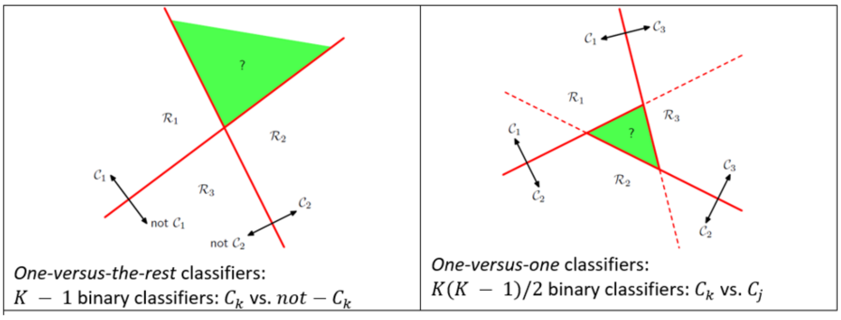
- assume a prediction parametric model $y(\boldsymbol{x}; \theta)$, $y(\boldsymbol{x}; \theta) \approx f(x)$
- Define an error function $E(\theta)$
- Solve the optimization problem $\theta^* = \arg\min_{\theta} E(\theta)$
- Classify new sample x' as $y(\boldsymbol{x}'; \theta^*)$.

NB: All methods described above can be applied in a transformed space of the input (***feature space***).
Given a function $\phi: \tilde{\boldsymbol{x}} \rightarrow \boldsymbol{\Phi}$ (Φ is the feature space) each sample $\tilde{\boldsymbol{x}}_{\boldsymbol{n}}$ can be mapped to a feature vector $\phi_n = \phi(\tilde{x}_n)$.

7. Linear Models for Classification

Assume that the dataset is linearly separable (if it exists some hyper-plane that splits the space in two regions such that different classes are separated). Such hyper-plane is generated by a function $f : R^n \rightarrow C = \{c_1, c_2, \ldots, c_m\}$

- If 2 classes: $y(x) = w^T x + w_0 = \tilde{w}^T \tilde{x}$
- If k-classes: $y(x) = \begin{pmatrix} y_1(x) \\ \ldots \\ y_k(x) \end{pmatrix} = \begin{pmatrix} w_1^T x + w_{1,0} \\ \ldots \\ w_k^T x + w_{k,0} \end{pmatrix} = \begin{pmatrix} \tilde{w}_1{}^T \\ \ldots \\ \tilde{w}_k{}^T \end{pmatrix} \tilde{x} = \widetilde{W}^T \tilde{x}$

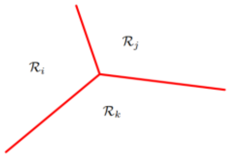


One-versus-the-rest classifiers:
$K - 1$ binary classifiers: C_k vs. $not - C_k$

One-versus-one classifiers:
$K(K - 1)/2$ binary classifiers: C_k vs. C_j

K-class discriminant comprising K linear functions (x not in dataset)

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

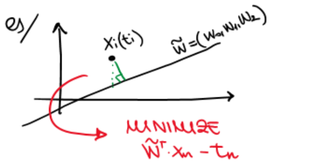


Classify $\boldsymbol{x}$ as C_k if $y_k(\boldsymbol{x}) > y_j(\boldsymbol{x})$ for all $j \neq k$ $(j, k = 1, \ldots, K)$
Where the decision boundary between C_k andC_j is: $\left(\tilde{\boldsymbol{w}}_k - \tilde{\boldsymbol{w}}_j\right)^T \tilde{\boldsymbol{x}} = 0$

Our goal is to determine $\widetilde{\boldsymbol{W}}$ such that $y(\boldsymbol{x}) = \widetilde{\boldsymbol{W}}^T \tilde{\boldsymbol{x}}$ is the K-class discriminant. To do this there are different approaches:

- Least squares
- Perceptron
- Fisher's linear discriminant
- Support Vector Machines

Least Squares

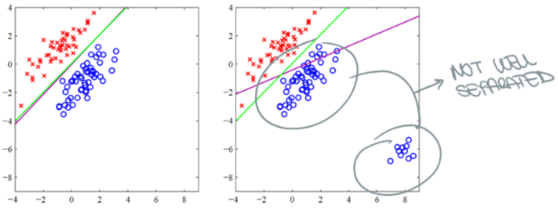


Given D, find the linear discriminant $y(x) = \widetilde{\boldsymbol{W}}^T \tilde{\boldsymbol{x}}$.
→Minimize the sum-of-square error function:
$E(\widetilde{W}) = \widetilde{W} = \left(\tilde{X}^T \tilde{X}\right)^{-1} \tilde{X}^T T = \tilde{X}^{\dagger} T \rightarrow y(x) = \widetilde{W}^T \tilde{x} = T^T \left(\tilde{X}^{\dagger}\right)^T \tilde{X}$

With $T = \begin{pmatrix} t_1 \\ \ldots \\ t_N \end{pmatrix}^T$ if $x \in C_k \rightarrow t_k = 1, t_j = 0, \forall j \neq k$.

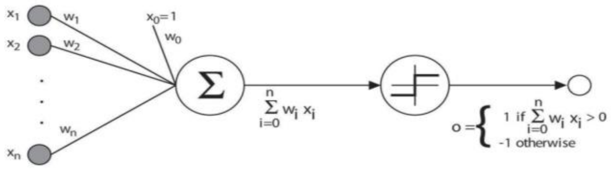
Classification of new instance x not in dataset:
Use learnt $\widetilde{\boldsymbol{W}}$ to compute $y(x)$ then assign class C_k to x, where $k = \arg \max_{i \in \{1, \ldots, k\}} \{y_i(x)\}$

PROBLEM: assume Gaussian conditional distributions
→ Not robust to outliers!



Perceptron

The Perceptron is a linear classification algorithm. It consists of a single node that takes a row of data as input and predicts a class label. This is achieved by calculating the *weighted sum* of the inputs and a bias (=1). The weighted sum of the input of the model is called the *activation*.

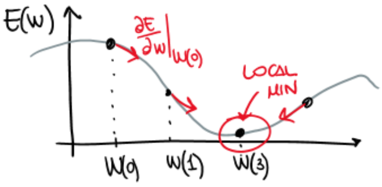

x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

perceptron model $o(x) = sign(w^T x)$ or $(w^T x)$ if unthreshold.
To learn w_i from training examples $D = \{(x_n, t_n)_{n=1}^{N}\}$ minimize the squared error (loss function):

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

Since we need to minimize this error we want to move to the direction of the gradient, thus computing the derivative of:

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



so we can update the weight w_i by $w_{i+1} = w_i + \Delta w$, where:

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

Perceptron algorithm:

Given perceptron model $o(x) = sign(w^T x)$ and data set D, determine weights w.

- The initial values for the model weights are set to small random values.
- $\widehat{w}_i \longleftarrow \widehat{w}_i + \Delta w_i$ Model weights are updated with a small proportion of the error each batch, and the proportion is controlled by a hyperparameter called the *learning rate*, typically set to a small value (resulting in a premature convergence):
  $w(t+1) = w(t) + learning\_rate * (expected\_i - predicted\_) * input\_i$
- Training stop when the error made by the model falls to a low level or no longer improves.

Batch mode: Consider all dataset $D \rightarrow \Delta w_i = \eta \sum_{(\mathbf{x},t)\in D} (t - o(\mathbf{x}))\, x_i$

Mini-Batch mode: Choose a small subset $S \subset D \rightarrow \Delta w_i = \eta \sum_{(\mathbf{x},t)\in S} (t - o(\mathbf{x}))\, x_i$

Incremental mode: Choose one sample $(\mathbf{x}, t) \in D \rightarrow \Delta w_i = \eta\,(t - o(\mathbf{x}))\, x_i$

- The perceptron is a linear classifier, it will classify all the inputs correctly if the training set D is linearly separable, and η sufficiently small.
- Incremental and mini-batch modes speed up convergence and are less sensitive to local minima.

NO

Fisher's linear discriminant

Given a two classes classification problem, Fisher's linear discriminant is given by the function $y = \boldsymbol{w}^T\boldsymbol{x}$ and the classification of new instances is given

$$x \in C_1 if\ y \geq -w_0$$
$$x \in C_2 \text{ otherwise}$$

Corresponding to the projection on a line determined by **w.**
Adjusting ***w*** to find a direction that maximizes class separation.

Consider a data set with N1 points in C1 and N2 points in C2: $\mathbf{m}_1 = \frac{1}{N_1}\sum_{n\in C_1}\mathbf{x}_n \qquad \mathbf{m}_2 = \frac{1}{N_2}\sum_{n\in C_2}\mathbf{x}_n$

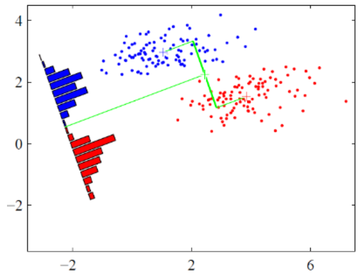
→choose w that maximises $J(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}}$

with $\mathbf{S}_B = (\mathbf{m}_2 - \mathbf{m}_1)(\mathbf{m}_2 - \mathbf{m}_1)^T$
Between class scatter

$\mathbf{S}_W = \sum_{n\in C_1}(\mathbf{x}_n - \mathbf{m}_1)(\mathbf{x}_n - \mathbf{m}_1)^T + \sum_{n\in C_2}(\mathbf{x}_n - \mathbf{m}_2)(\mathbf{x}_n - \mathbf{m}_2)^T$
Within class scatter

$\frac{d}{dw}J(w) = 0 \rightarrow : \boldsymbol{w}^* = \boldsymbol{S}_W^{-1}(\boldsymbol{m}_2 - \boldsymbol{m}_1)$ and $w_0 = \boldsymbol{w}^T\boldsymbol{m}$ → GLOBAL MEAN OF ALL D.

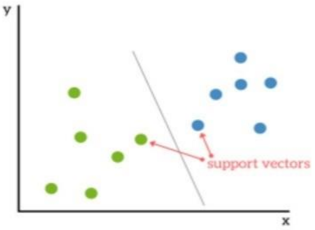
$\mathbf{w} \propto \mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1)$



For Multiple classes: $y = \boldsymbol{w}^T\boldsymbol{x}$ → Maximizing $J(\boldsymbol{W}) = Tr\{(\boldsymbol{W}\boldsymbol{S}_w\boldsymbol{W}^T)^{-1}(\boldsymbol{W}\boldsymbol{S}_B\boldsymbol{W}^T)\}$

Support Vector Machine [SVM]

SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes.



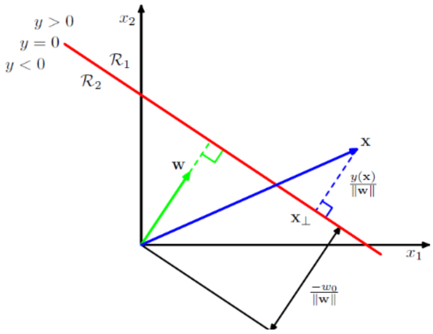
Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set.

Let's consider binary classification $f: X \rightarrow \{+1, -1\}$ with data set $D = \{(x_n, t_n)_{n=1}^{N}\}, t_n \in \{+1, -1\}$, and a linear model $y(x) \ = \ w^T x \ + \ w_0$

Assume D is linearly separable → $\exists\, \mathbf{w}, w_0 \ s.t. \ \begin{array}{l} y(\mathbf{x}_n) > 0, \ if \ t_n = +1 \\ y(\mathbf{x}_n) < 0, \ if \ t_n = -1 \end{array}$

$$\to \ t_n\, y(\mathbf{x}_n) > 0 \ \forall n = 1, \ldots N$$ ← WE RESTRICT THE SOLUTION TO BE A LINE THAT SEPARATE INTO TWO PARTS

Let x_k be the closest point of the dataset D to hyperplane $\bar{h}$: $\ \bar{w}^T x + \bar{w}_0 = 0$



The **margin** is estimated as the minimum distance among all the points in the dataset from the line

$$\boxed{\min_{n=1,\ldots,N} \frac{|y(\mathbf{x}_n)|}{||\mathbf{w}||} = \cdots = \frac{1}{||\mathbf{w}||} \min_{n=1,\ldots,N} [t_n(\bar{\mathbf{w}}^T \mathbf{x}_n + \bar{w}_0)]}$$

And to maximize the margin:

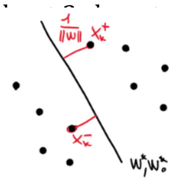
$$\boxed{\mathbf{w}^*, {w_0}^* = \operatorname*{argmax}_{\mathbf{w}, w_0} \frac{1}{||\mathbf{w}||} \min_{n=1,\ldots,N} [t_n(\mathbf{w}^T \mathbf{x}_n + w_0)]}$$

To solve:

Rescale all the points do not affect the solution in such a way that the closest point x_k we have: $t_k(\boldsymbol{w}^T \boldsymbol{x}_k + w_0) = 1$

When the maximum margin hyperplane w^*, w_0^* is found, there will be a points x_k^+ and x_k^- (one for each class)

The optimal solution is when both are at the same distance $\frac{1}{||w||}$



In the canonical representation of the problem the maximum margin hyperplane can be found by solving the optimization problem:

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

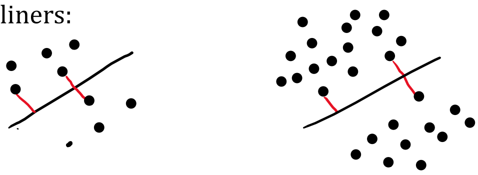
subject to $t_n(\mathbf{w}^T\mathbf{x}_n + w_0) \geq 1 \;\; \forall n = 1, \ldots, N$

Quadratic programming problem solved with Lagrangian method. →

$w^* = \sum_{n=1}^{N} a_n^* \, t_n \, x_n$ a_i = LAGRANGE MULTIPLIERS

- Very sparse (because of KKT condition)
- even if the dimension is larger, few values are > 1 so optimization will be better
- all the points/samples x_n for which $a_n^* = 0$ will not contribute to the solution!
- Robust to outliners:

es:



→ SAME SOLUTION

Support vectors: x_k such that $t_k \, y(x_k) = 1$ and $a_k^* > 0$ → $SV \equiv \{\mathbf{x}_k \in D \mid t_k \, y(\mathbf{x}_k) = 1\}$

Hyperplanes expressed with support vectors: $y(\boldsymbol{x}) = \sum_{x_j \in SV} a_j^* \, t_j \, \boldsymbol{x}^T x_j + w_0^* = 0$

In fact, other vectors $x_n \notin SV$ do not contribute $(a_n^* = 0)$

To compute w_0 : $$w_0^* = t_k - \sum_{\mathbf{x}_j \in SV} a_j^* t_j \mathbf{x}_k^T \mathbf{x}_j$$

Or a more stable solution is obtained by averaging over all the support vectors:

$$w_0^* = \frac{1}{|SV|} \sum_{\mathbf{x}_k \in SV} \left(t_k - \sum_{\mathbf{x}_j \in S} a_j^* t_j \mathbf{x}_k^T \mathbf{x}_j \right)$$

Given the maximum margin hyperplane determined by a_k^* , w_0^*

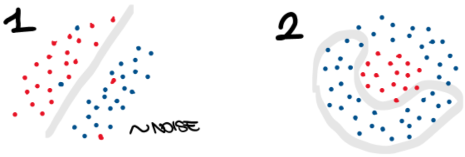
Classification of a new instance x' is given by the prediction model:

$$y(\mathbf{x}') = sign\left(\sum_{\mathbf{x}_k \in SV} a_k^* \, t_k \, \mathbf{x}'^T \mathbf{x}_k + w_0^* \right)$$

- Optimization problem for determining w, w_0 (dimension $d+1$, with $X = R^d$) transformed in an optimization problem for determining a (dimension $|D|$)
- Efficient when $d << |D|$ (most of ai will be zero).
- Very useful when d is large or infinite.

What if data are ALMOST (the majority) linearly separable?

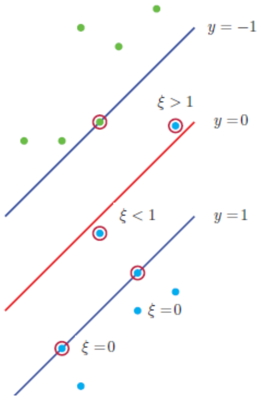
Two cases:



1) SVM with soft margin constraints

IDEA: to relax the constraints (add a cost!)

Let introduce a new variable: slack variables $\xi_n \geq 0, n = 1, \dots, N$



- $\xi_n = 0$ if point on or inside the correct margin boundary
- $0 < \xi_n \leq 1$ if point inside the margin but correct side
- $\xi_n > 1$ if point on wrong side of boundary

Soft margin constraint: $t_n\, y(x_n) \geq 1 - \xi_n$

Optimization problem:

$$\boxed{\mathbf{w}^*, w_0^* = \operatorname{argmin} \frac{1}{2}||\mathbf{w}||^2 + \overset{\text{C IS A CONSTANT}}{C} \sum_{n=1}^{N} \underbrace{\xi_n}_{\text{COST WHICH PENALIZES HIGH VALUES OF } \xi_n}}$$

- C=0 → UNCONSTRAINT
- C = HIGH → PENALIZE
- C= ∞ → UNCONSTRAINT

subject to

$$t_n\, y(\mathbf{x}_n) \geq 1 - \xi_n\,, \quad n = 1, \dots, N$$
$$\xi_n \geq 0\,, \quad n = 1, \dots, N$$

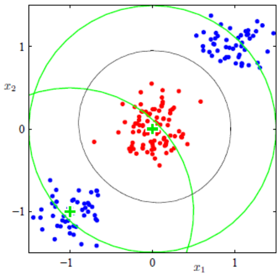
Solution similar to the case of linearly separable data → $\mathbf{w}^* = \sum_{n=1}^{N} a_n^*\, t_n\, \mathbf{x}_n$ $\quad w_0^* = ...$

with a_n^* computed as solution of a Lagrangian optimization problem.

2) Basis functions

IDEA: transform in a polar coordinate for each new value → replace $\boldsymbol{x}$ with $\underbrace{\phi(x)}_{\text{BASIS FUNCTION}}$ to all formulas.

- Decision boundaries will be linear in the feature space ϕ and non-linear in the original space **x**
- Classes that are linearly separable in the feature space ϕ may not be separable in the input space **x**.



↳ es: LINEAR
POLYNOMIAL
SIGMOID
...

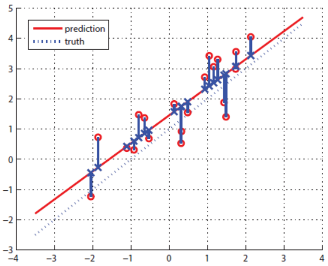
(*exists a family of basis functions that works well when you cannot find ϕ(x))

Linear models for non-linear functions:

To learn non-linear function $f : X \rightarrow \{C_1, \dots, C_K\}$ from data set D non-linearly separable → find a non-linear transformation ϕ and learn a linear model

- Two classes: $y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(x) + w_0$
- Multiple classes: $y_k(\boldsymbol{x}) = \boldsymbol{w}_k^T \phi(x) + w_{k0}$

8. Linear Models for Regression



Define a model $y(x; w)$ with parameter w to approximate the target function f, our model will be of the kind:

$$y(\boldsymbol{x}; \boldsymbol{w}) = w_0 + w_1 x_1 + \cdots + w_d x_d = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}$$

Where both ***w, x*** are vectors of dimension d.
There are some cases in which the dataset is non linear, so we can use a non linear function on x of the kind:

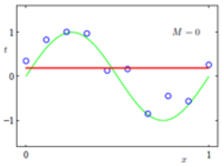
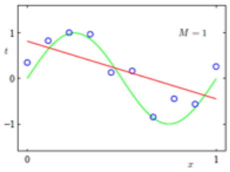
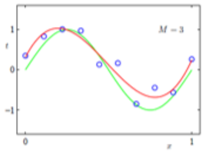
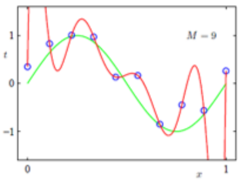
$$y(\boldsymbol{x}; \boldsymbol{w}) = \sum_{j=0}^{M} w_j \phi_j(x) = \boldsymbol{w}^{\boldsymbol{T}} \varphi(\boldsymbol{x})$$

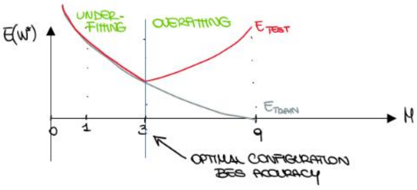
Which is still linear in w but not in x.

Es: *polynomial curve fitting* $\quad y = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M$

LINEAR MODEL

BUT NOT LINEAR FUNCTION !

▪ M=0  Polynomial of degree 0 is just a constant	▪ M=1  The mean-square error is better than in case M=0. - not a good model
▪ M=3  Data are not noising, can construct the same polynomial, but not possible in practise - good model, not perfect - mean error ≈ 0	▪ M=9  ← overfitting! The mean error =0



Learning algorithm for linear regression:

Maximum Likelihood

If our target value t is affected by noise ϵ: $t = y(\boldsymbol{x}; \boldsymbol{w}) + \epsilon$

We now have a probability that the target is correct given the regression. If we assume ϵ to be gaussian we have: $P(\epsilon|\beta) = \mathcal{N}(\epsilon|0, \beta^{-1})$, with precision (inverse variance) β.

Assume observations independent and identically distributed (i.i.d.)
We seek the **maximum of the likelihood function**: $P(t|\boldsymbol{x},\boldsymbol{w},\beta) = \mathcal{N}(t|y(\boldsymbol{x};\boldsymbol{w}),\beta^{-1})$

$$P(\{t_1,\ldots,t_N\}|\mathbf{x}_1,\ldots,\mathbf{x}_N,\mathbf{w},\beta) = \prod_{n=1}^{N}\mathcal{N}(t_n|\mathbf{w}^T\phi(\mathbf{x}_n),\beta^{-1}).$$

Or equivalently:

$$= \sum_{n=1}^{N}\ln\mathcal{N}(t_n|\mathbf{w}^T\phi(\mathbf{x}_n),\beta^{-1}) \quad = -\beta\,\overbrace{\frac{1}{2}\sum_{n=1}^{N}[t_n-\mathbf{w}^T\phi(\mathbf{x}_n)]^2}^{E_D(W)} - \frac{N}{2}\ln(2\pi\beta^{-1}).$$

Since the second term is constant we focuse on the first one. Then to solve the problem we do: $\boxed{w^* = \arg\min_{w} E_D(w)}$

But since ***Max Likelihood*** $\leftrightarrow$ ***Least Square error minimization***

$$\arg\max_{\mathbf{w}} P(\{t_1,\ldots,t_N\}|\mathbf{x}_1,\ldots,\mathbf{x}_N,\mathbf{w},\beta) \quad \Longleftrightarrow \quad \arg\min_{\mathbf{w}} E_D(\mathbf{w}) = \arg\min_{\mathbf{w}} \frac{1}{2}\sum_{n=1}^{N}[t_n-\mathbf{w}^T\phi(\mathbf{x}_n)]^2$$

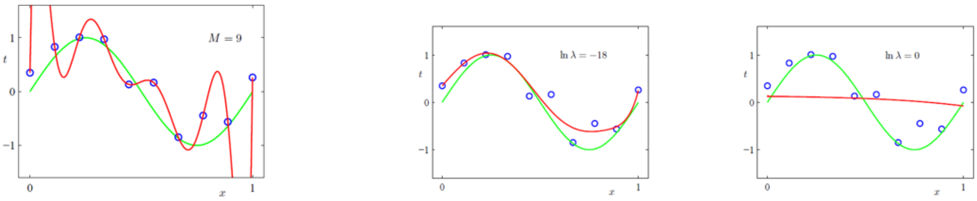
NB: if we plot we have some values of w very high ($||w_i|| \gg 0$) → this determine a not-smooth function! → To control **overfitting** we can set a **regularization** factor on the parameters of the kind:

$$\arg\min_{w} E_D(\boldsymbol{w}) + \lambda E_W(\boldsymbol{w})$$

→ NOT DEPEND ON D
→ REGULARIZATION TERM TO BALANCE (>0)

a common choice is $E_W(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$

$\arg\min_{\mathbf{w}} E_D(\mathbf{w})$ → $\arg\min_{\mathbf{w}} E_D(\mathbf{w}) + \lambda\frac{1}{2}\mathbf{w}^T\mathbf{w}$



Moreover, note that : $E_D(\boldsymbol{w}) = \frac{1}{2}(t-\boldsymbol{\phi w})^T(t-\boldsymbol{\phi w})$

$$\mathbf{t} = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \text{ and } \boldsymbol{\Phi} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_M(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_M(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_M(\mathbf{x}_N) \end{bmatrix}$$

Optimality condition → $\nabla E_D = 0 \leftrightarrow \boldsymbol{\phi}^T\boldsymbol{\phi w} = \boldsymbol{\phi}^T\,\boldsymbol{t}$ → $\boxed{\boldsymbol{w_{ML}} = (\boldsymbol{\phi}^T\boldsymbol{\phi})^{-1}\boldsymbol{\phi}^T\,\boldsymbol{t} = \boldsymbol{\phi}^{\dagger}\boldsymbol{t}}$

if big dimension, the pseudoinverse $\boldsymbol{\phi}^{\dagger}$ is complicated to compute → we compute the stochastic gradient descent algorithm (SGD)(**Sequential Learning**)

$$\widehat{\boldsymbol{w}} \leftarrow \widehat{\boldsymbol{w}} - \eta\,\nabla E_n \;\rightarrow\; \widehat{\boldsymbol{w}} \leftarrow \widehat{\boldsymbol{w}} - \eta\,[\mathrm{t_n} - \widehat{\boldsymbol{w}}^T\phi(\boldsymbol{x}_n)]\phi(\boldsymbol{x}_n)$$

If **Multiple outputs:** $y(\boldsymbol{x};\boldsymbol{W}) = \boldsymbol{W}^T\phi(\boldsymbol{x})$ and we have $\ln P(\mathbf{T}|\mathbf{X},\mathbf{W},\beta) = \sum_{n=1}^{N}\ln\mathcal{N}(\mathbf{t}_n|\mathbf{W}^T\phi(\mathbf{x}_n),\beta^{-1}\mathbf{I})$
similarly as before we obtain: $\boldsymbol{w_{ML}} = (\boldsymbol{\phi^T\phi})^{-1}\boldsymbol{\phi}^T\,\boldsymbol{T}$

9. Kernel Methods

Kernel methods overcome difficulties in defining non-linear models. Kernel methods use kernels (or basis functions) to map the input data into a different space. After this mapping, simple models can be trained on the new feature space, instead of the input space, which can result in an increase in the performance of the models.
This approach is called the "**kernel trick**", which avoids the explicit mapping that is needed to get linear learning algorithms to learn a nonlinear function or decision boundary

Consider a linear model $y(\boldsymbol{x};\boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x}$ with Dataset $D = \{(x_n, t_n)_{n=1}^N\}$
Minimize $J(\boldsymbol{w}) = (\boldsymbol{t}-\boldsymbol{Xw})^T(\boldsymbol{t}-\boldsymbol{Xw}) + \lambda\,||\boldsymbol{w}||^2$

→ Optimal solution:

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

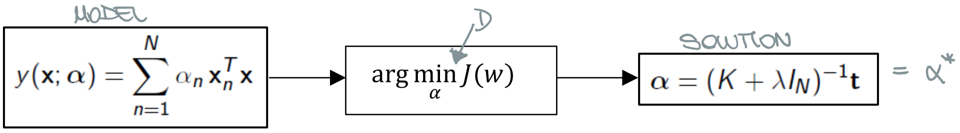
$$\mathbf{w}^* = \frac{1}{\lambda}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - t_n)\mathbf{x}_n$$

X is the design matrix (representation of the dataset).
We can express $\boldsymbol{w}^*$ by calling $\boldsymbol{\alpha} = (\boldsymbol{XX}^T+\lambda I_N)^{-1}\,\boldsymbol{t}$, $\quad \alpha_n = \frac{1}{\lambda}(\boldsymbol{w}^T x_n - t_n)$ →

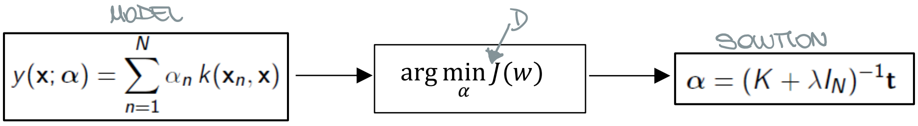
$$\boxed{\boldsymbol{w}^* = \boldsymbol{X}^T\boldsymbol{\alpha} = \sum_{n=1}^{N}\alpha_n x_n}$$

So our model will be: $y(\boldsymbol{x};\boldsymbol{w}^*) = \boldsymbol{w}^{*T}\boldsymbol{x} = \sum_{n=1}^{N}\alpha_n\boldsymbol{x}_n^T\boldsymbol{x}$, with $\alpha = (K+\lambda I_N)^{-1}$ and $K = \boldsymbol{XX^T}$ *Gram matrix.*

- linear model with linear kernel $k(\boldsymbol{x},\boldsymbol{x}') = \boldsymbol{x}^T\boldsymbol{x}'$


MODEL
$y(\mathbf{x};\alpha) = \sum_{n=1}^{N}\alpha_n\,\mathbf{x}_n^T\mathbf{x}$
D
$\arg\min_{\alpha} J(w)$
SOLUTION
$\alpha = (K+\lambda I_N)^{-1}\mathbf{t}$
$= \alpha^*$

- linear model with *any* k:


MODEL
$y(\mathbf{x};\alpha) = \sum_{n=1}^{N}\alpha_n\,k(\mathbf{x}_n,\mathbf{x})$
D
$\arg\min_{\alpha} J(w)$
SOLUTION
$\alpha = (K+\lambda I_N)^{-1}\mathbf{t}$

Kernel tricks: If input vector x appears in an algorithm only in the form of an inner product $\boldsymbol{x}^T\boldsymbol{x}'$ we can replace it with some kernel $k(\boldsymbol{x},\boldsymbol{x}')$.
Approach: use a similarity measure $k(\boldsymbol{x},\boldsymbol{x}') \geq 0$ between the instances $\boldsymbol{x},\boldsymbol{x}'$

- $k(\boldsymbol{x},\boldsymbol{x}')$ is called a ***kernel function***.
- Note: If we have $\phi(x)$ a possible choice is $k(\boldsymbol{x},\boldsymbol{x}') = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$

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

- We can apply kernelization also in *Regression* and *SVM*.
- Usually is good to normalize data.

Input normalization:
Input data in the dataset D must be normalized in order for the kernel to be a good similarity measure in practice.

Several types of normalizations:

- min-max $\bar{x} = \frac{x - min}{max - min}$
  min, max: minimum and maximum input values in D
- normalization (standardization) $\bar{x} = \frac{x - \mu}{\sigma}$
  μ mean and σ standard deviation of input values in D
- unit vector $\bar{x} = \frac{x}{||x||}$

Kernel families:

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

Kernelized SVM

it is one of the most effective ML method for **classification** and **regression**.
- Still requires model selection and hyper-parameters tuning.

Classification:
In SVM, solution has the form: $w^* = \sum_{n=1}^{N} \alpha_n x_n$

Linear model:

$$y(\boldsymbol{x}; \boldsymbol{\alpha}) = sign(w_0 + \textstyle\sum_{n=1}^{N} \alpha_n x_n^T x) \xrightarrow{\text{KERNEL TRICK}} y(\boldsymbol{x}; \boldsymbol{\alpha}) = sign(w_0 + \textstyle\sum_{n=1}^{N} \alpha_n k(x_n, x))$$

And can be solved as a lagrangian problem …

Regression:

Linear model for regression $y = \boldsymbol{w}^T \boldsymbol{x}$ and data set D

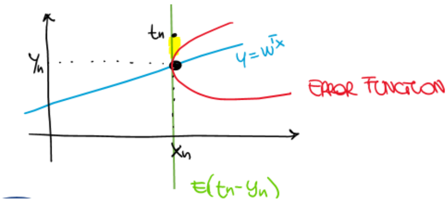
$y_n = w^T \cdot x_n$

- Minimize the regularized loss function: $J(w) = \sum_{n=1}^{N} E(y_n, t_n) + \lambda \, ||w||^2$

KERNEL TRICK

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

$$\boldsymbol{\alpha} = (K + \lambda I_N)^{-1} \mathbf{t}$$



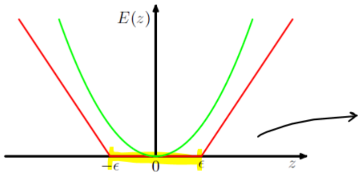
The IDEA:

- points close to the predict model are good enough →I don't consider them in an error of the model
- decrease the effects of the points faraway

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

With: C inverse of λ and E_ϵ and ϵ-insensitive error function: $E_\epsilon(y,t) = \begin{cases} 0 & \text{if } |y-t| < \epsilon \\ |y-t| - \epsilon & \textit{otherwise} \end{cases}$

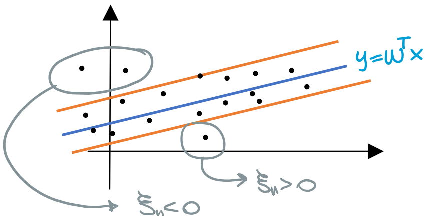
"ε-TUBE"



ERROR FUNCTION WILL NOT SEE WHAT IS IN $[-\epsilon,+\epsilon]$

Not differentiable → difficult to solve.

Introduce Slack Variables $\xi_n^+, \xi_n^- \geq 0$: $\begin{cases} t_n \leq y_n + \epsilon + \xi_n^+ \\ t_n \geq y_n - \epsilon - \xi_n^- \end{cases}$



Points inside the ϵ-tube $y_n - \epsilon \leq t_n \leq y_n + \epsilon \Rightarrow \xi_n = 0$

$\xi_n^+ > 0 \Rightarrow t_n > y_n + \epsilon$

$\xi_n^- > 0 \Rightarrow t_n < y_n - \epsilon$

- if all points inside the tube → I will have 0-error (optimal!)
- if I have some external points the solution will depend on them

⟹ loss function can be rewritten as: $\boxed{J(\boldsymbol{w}) = C\sum_{n=1}^{N}(\xi_n^+, + \xi_n^-) + \frac{1}{2}||\boldsymbol{w}||^2}$

subject to :

$$t_n \leq y(\mathbf{x}_n; \mathbf{w}) + \epsilon + \xi_n^+$$
$$t_n \geq y(\mathbf{x}_n; \mathbf{w}) - \epsilon - \xi_n^-$$
$$\xi_n^+ \geq 0$$
$$\xi_n^- \geq 0$$

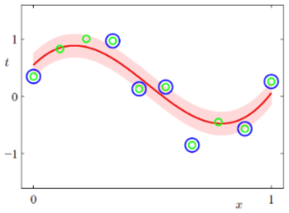
and can be solved as Lagrangian (it is a standard quadratic program).

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

- $\hat{a}_n > 0 \to \epsilon + \xi_n + y_n - t_n = 0$ → data point lies on or above upper boundary of the ϵ -tube
- $\hat{a}'_n > 0 \to \epsilon + \xi_n - y_n + t_n = 0$ → data point lies on or below lower boundary of the ϵ -tube

All other data points inside the ϵ -tube have $\hat{a}_n = 0$ and $\hat{a}'_n = 0$ and thus do not contribute to prediction.

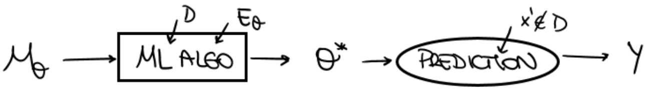
→ es:



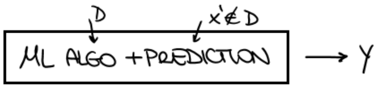
10. Instance based Learning

Recap:

- **Parametric** Algorithm: we have a fixed set of parameters such as θ that we try to find while training the data. After we have found the optimal values for these parameters, we can use the model with parameters to make predictions.



- **Non-Parametric** Algorithm: the number of parameters grows with the amount of data and the model is note explicit (e.g. Instance-based learning)



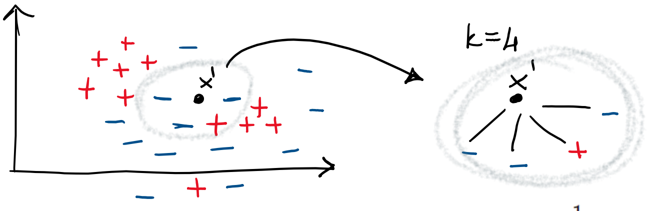
Instance-based learning involves memorizing training data in order to make predictions about future data points. This approach doesn't require any prior knowledge or assumptions about the data, which makes it easy to implement and understand. However, it can be computationally expensive since all of the training data needs to be stored in memory before making a prediction. Additionally, this approach doesn't generalize well to unseen data sets because its predictions are based on memorized examples rather than learned models.

K-nearest neighbors (KNN)

KNN is an algorithm that belongs to the instance-based learning class of algorithms.
It relies on a measure of similarity between each pair of data points (e.g. Euclidean distance).
Once the similarity between two points is calculated, KNN looks at how many neighbors are within a certain radius around that point and uses these neighbors as examples to make its prediction.
Classification rule:

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

es:



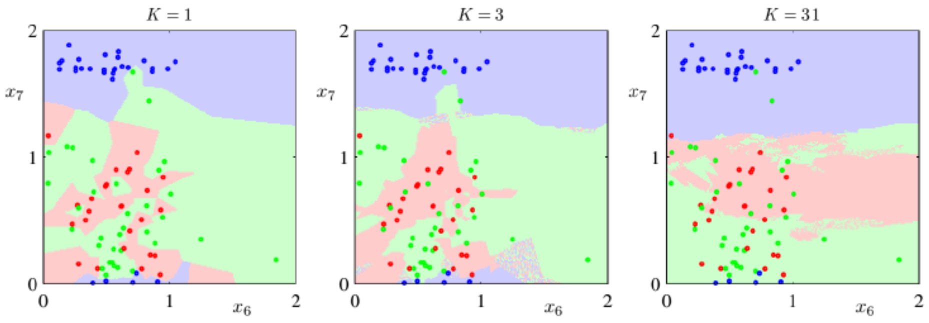
TAKE THE NEAREST POINTS

⇒ MAJORITY = ⊖

Likelihood of class c for new instance x: $p(c|\mathrm{x}, D, K) = \frac{1}{K} \sum_{\mathrm{x}_n \in N_K(\mathrm{x}_n, D)} \mathbb{I}(t_n = c)$

with $N_K(x_n, D)$ the K nearest points tox_n and $\mathbb{I}(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{if } e \text{ is false} \end{cases}$

- instead of creating a generalizable model from all of the data, KNN looks for similarities among individual data points and makes predictions accordingly.
- Require storage of all data
- Increasing K brings to smoother regions (reducing overfitting)



One of the many issues that affect the performance of the kNN algorithm is the choice of k. If k is too small, the algorithm would be more sensitive to outliers. If k is too large, then the neighborhood may include too many points from other classes.

K-NN can be **kernelized.**

Distance function in computing $N_K(x, D)$

$$\left|\left|x - x_n\right|\right|^2 = x^T x + x_n^T x_n - 2x^T x_n$$

Can be kernelized by using kernel $k(x, x_n)$.

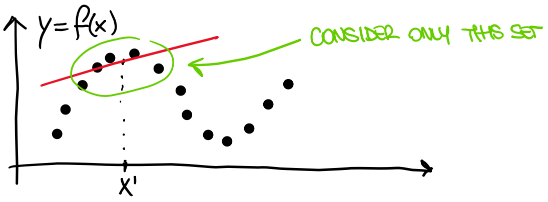
Locally weighted regression

Locally weighted regression methods are a generalization of k-Nearest Neighbour. Instead of fitting a single regression line, you fit many linear regression models. The final resulting smooth curve is the product of all those regression models.

→ fit a local regression model around the query sample x_q

1. Compute $N_K(x_q, D)$: K-nearest neighbors of x_q
2. Fit a regression model $y(x; w)$ on $N_K(x_q, D)$
3. Return $y(x_q; w)$

ES. WITH LINEAR KERNEL:



Issues:

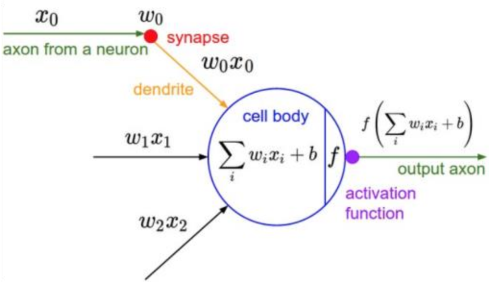
- DV are affected by scaling problem
- The model can be considered as a "compression"

11. Artificial Neural Network

Artificial neural networks (ANNs) are a subset of ML and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

FeedForward NN (FNN)

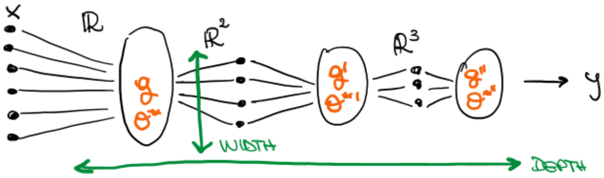
Most NN are feedforward, they flow in one direction only, from input to output. There are no loops or cycles in the network and the output of each layer is determined by the weights and biases of the connections between neurons, as well as the activation function of each neurons.



Hidden layer output can be seen as an array of unit (neuron) activations based on the connections with the previous units.
The final function is a composition of elementary functions f and parameters θ (one for each layer): $f(x;\theta) = f^{(3)}\left(f^{(2)}\left(x;\theta^{(1)}\right);\theta^{(2)}\right);\theta^{(3)}$

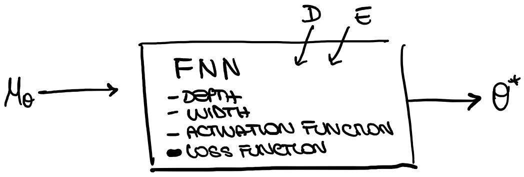
These many functions can tackle non-convex problems contrary to linear or kernel methods. Moreover, Linear models cannot model interactions between input variables.

In general, when you have multiple layers: you can see a layer which transform a space into another one → NN can be seen as a sequence of transformations.

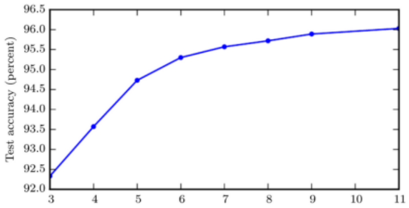


Architecture Design:

Choosing an appropriate architecture for a neural network is an important consideration, as it can impact the model's performance and ability to learn.



1. **Depth:** *given by the number of hidden layers.*
   Universal approximation theorem: *a FFN with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g., sigmoid) can approximate any Borel measurable function with any desired amount of error, provided that enough hidden units are used.*



The Depth is correlated to the performance: increasing too much layers <-> not increasing the performance

+ Overfitting problem if too powerful model

2. **Width:** *number of units (neurons) in each layer*.
   In general, it is exponential in the size of the input. In theory, a short and wide network can approximate any function. In practice a deep and narrow network is easier to train and provides better results in generalization.

3. **Activation function:** *which kind of units.*
   The activation function of a neuron determines the output of the neuron given a set of inputs. Different activation functions can be used in different layers of the network.

   They are used to introduce non-linearity into the network, allowing the model to learn and make more complex decisions.
   There are several types of activation functions that are commonly used in neural networks such as:
   - Rectified linear units (ReLU): $g(\alpha) = max(0,\alpha)$ which is easy to optimize but not differentiable at 0.
   - Sigmoid $g(\alpha) = \sigma(\alpha)$ and hyper tan $g(\alpha) = thanh(\alpha)$, both are: Easy to saturate since there is no logarithm at the output, slow, usefull for RNN and autoencoders

4. **Loss function:** *which kind of cost function.*
   It is the Cost function used for Training. This cost function is a guide to training process by providing a measure of how well the network is performing. The goal is to minimize the cost function by updating the weights during training (this process is usually done by using an optimization algorithm such as the Stochastic Gradient Descent).

NB: The loss function usually include the cost function + regularization term to prevent overfitting by penalizing large weights.

Recall the ML in which we wanted the class which maximized the conditional distribution $P(C_i|x,D)$, if we use the same principle here we get the **cross-entropy** loss function:

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

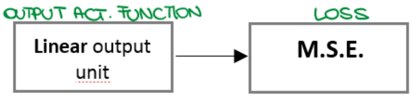
Choice of network output units and cost function are related.
($output\ activation\ function\ \leftrightarrow\ loss\ function$) They change together with the problem requirement:

1) Regression

We use the identity activation function $y\ = W^T h\ +\ b$
A Gaussian distribution noise model $p(t|x)\ =\ N(t|y, \beta^{-1})$
→ Cost function: maximum likelihood (cross-entropy) that is equivalent to minimizing mean squared error:

$$J(\theta) - \ln\left(P(t|x,\theta)\right.$$

Note: linear units do not saturate.



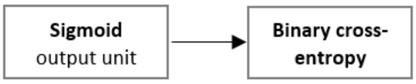
2) Binary Classification

We use the Sigmoid activation function $y\ = \sigma(w^T h\ +\ b)$
The likelihood corresponds to a Bernoulli distribution

$$J(\theta) = E_{x,t\sim D}[-\ln p(t|x)]$$

$$-\ln p(t|x) = \cdots = softplus((1-2t)\alpha$$

with $\alpha = w^T h\ +\ b$.
Note: Unit saturates only when it gives the correct answer.



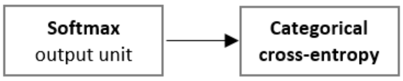
3) Multi-class Classification

We use the Softmax activation function $y_i = softmax\left(\alpha^{(i)}\right) = \frac{\exp(\alpha^{(i)})}{\sum_j \exp\,(\alpha_j)}$

The Likelihood corresponds to a Multinomial distribution

$$J_i(\theta) = E_{x,t\sim D}[-\ln\ softmax\left(\alpha^{(i)}\right)]$$

Note: Unit saturates on ly when there are minimal errors.

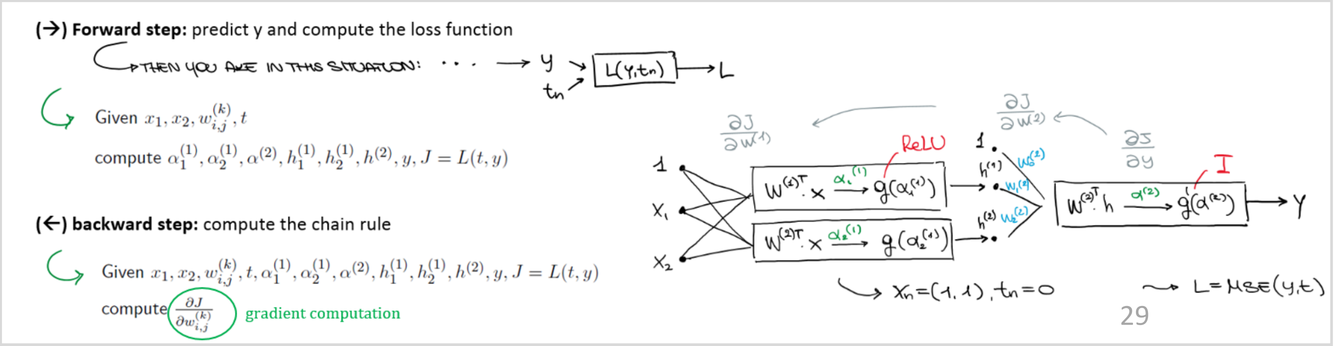


Gradient computation (BackProp)

Gradient computation "**BackProp**" (**Back-propagation**) is an algorithm used to train an ANN.
Algorithm steps:

a. Feed the input data through the network, and calculate the output of each neuron in each layer.
b. Calculate the error at the output layer using the cost function.
c. Propagate the error back through the network
d. Adjust the weights and biases of the connections between neurons
e. Repeat the process until the cost function is minimized and the network is able to make accurate predictions.

Since we propagate the error back trough the network, we compute the gradient of the cost function with respect to each parameter using the chain rule.



(→) **Forward step:** predict y and compute the loss function

THEN YOU ARE IN THIS SITUATION: · · · → y , t_n → $\boxed{L(y,t_n)}$ → L

Given $x_1, x_2, w_{i,j}^{(k)}, t$

compute $\alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t,y)$

(←) **backward step:** compute the chain rule

Given $x_1, x_2, w_{i,j}^{(k)}, t, \alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t,y)$

compute $\frac{\partial J}{\partial w_{i,j}^{(k)}}$ gradient computation

$\frac{\partial J}{\partial w^{(1)}}$
$\frac{\partial J}{\partial w^{(2)}}$
$\frac{\partial J}{\partial y}$
ReLU
I
1
x_1
x_2
$W^{(1)T} \cdot x \xrightarrow{\alpha_1^{(1)}} g(\alpha_1^{(1)})$
$W^{(2)T} \cdot x \xrightarrow{\alpha_2^{(1)}} g(\alpha_2^{(1)})$
1
$h^{(1)}$
$h^{(2)}$
$w_0^{(2)}$
$w_1^{(2)}$
$w_2^{(2)}$
$W^{(2)T} \cdot h \xrightarrow{\alpha^{(2)}} g'(\alpha^{(2)})$
Y
$x_n = (1,1), t_n = 0$
$L = MSE(y,t)$

Learning Algorithms

Since the backprop is just a method to compute the gradient (not learn) there are various other algorithms for this purpose.

- **Stochastic Gradient Descent:**

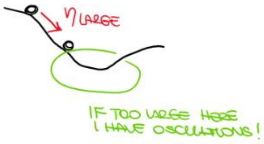
It involves calculating the gradient of the cost function with respect to the weights and biases of the connections between neurons and using that to update in such a way to minimizes the cost function.

SGD is an iterative process, and the *learning rate* determines the size of the steps taken to minimize the cost function. Using a learning rate η these method computes the gradient on a subset (minibatch) of samples from the dataset. This gradient is computed with the backprop method and the parameters are updated with the following formula:

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \eta g_i$$

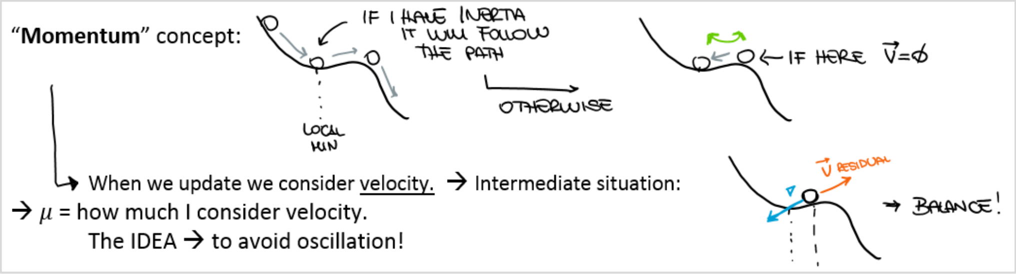
Where g is the value of the gradient with respect to θ_i.

Critical choice for η → η usually changes according to some rule through the iterations
(es. If we are faraway → η should be large, then should be constant



<u>optimization performance can be improved with momentum and adaptive learning rate:</u>

- **SGD with momentum:** To accelerate the training an additional parameter v can be used to increase or decrease the value of the update depending on the training iteration.
    - Momentum is applied before computing the gradient.
    - Sometimes it improves convergence rate.



"**Momentum**" concept:

When we update we consider <u>velocity.</u> → Intermediate situation:
→ μ = how much I consider velocity.
The IDEA → to avoid oscillation!

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

Which optimization algorithm should I choose? → Empirical approach.

generalization error can be reduced with regularization.

Regularization

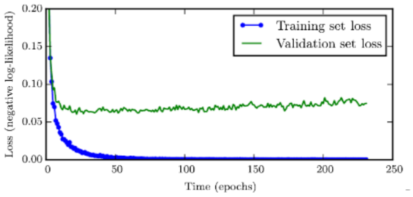
Technique used to reduce **overfitting.** In general it involves adding a penalty term to the cost function, which discourages the network from learning overly complex patterns in the training data. For FNN we have:

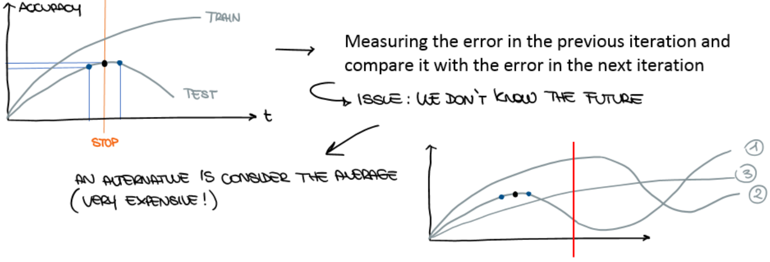
- **Parameter norm penalties:** add a regularization term to the cost function in order to decrease the magnitude of each parameter, so no parameter saturates.

$$\bar{J}(\theta) = J(\theta) + \lambda E_{reg}(\theta) \text{ with } E_{reg}(\theta) = \sum_j |\theta_j|^q$$

- **Dataset Augmentation:** Transform the dataset (image distortion, noise adding) in order to generate additional data in the Dataset. Done before training step.

- **Early stopping:** Stop iterations early to avoid overfitting to the training set of data, when train loss zeros out while train test loss increases. Use cross-validation to determine when to stop.

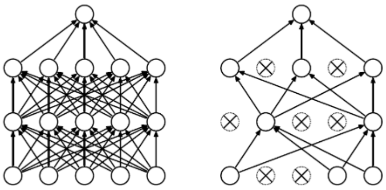



ACCURACY
TRAIN
TEST
t
STOP
ISSUE : WE DON'T KNOW THE FUTURE
AN ALTERNATIVE IS CONSIDER THE AVERAGE
(VERY EXPENSIVE !)
1
3
2

Measuring the error in the previous iteration and compare it with the error in the next iteration

- **Parameter sharing:** constrain subset of parameters to be equal. (limit the model = limit the overfitting)
  - Decrease memory consumption
  - In CNNs allow translation invariance.

- **Dropout**: Randomly remove network units with some probability α.

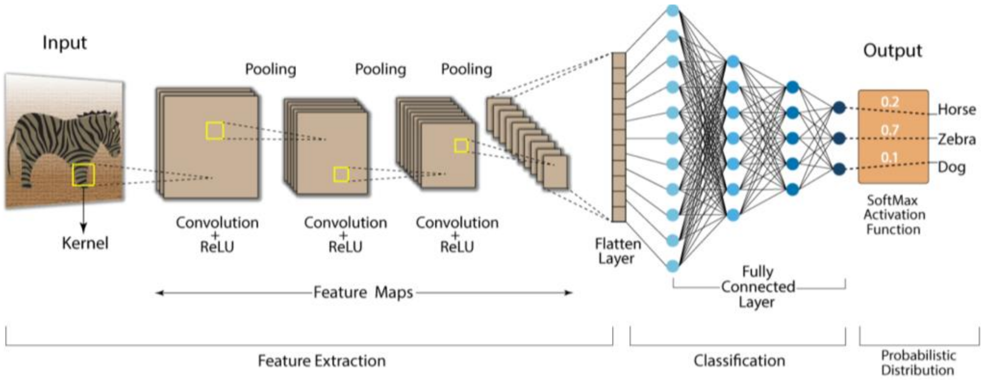


(a) Standard Neural Net

(b) After applying dropout.

12. Convolutional Neural Networks (CNNs)

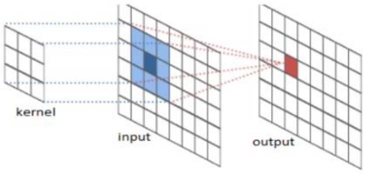
Convolutional neural networks (CNNs) are particularly effective at processing data that has a grid-like topology, such as an image. They take the image as an input, subjects it to combinations of weights and biases, extracts features and outputs the results (convolution). In simple word, it extract the feature of image and convert it into lower dimension without loosing its characteristics, by using a Kernel.



In CNNs there are usually more stages:

$$input \rightarrow \boxed{convolution + activation\ function \rightarrow pooling} \rightarrow \boxed{F.C \rightarrow softmax} \rightarrow output$$

1) **Convolution**: mathematical (linear) operation in the convo layer applied to an input image to produce a feature map by sliding the kernel matrix over the image, element-wise multiplying and summing the entries of the image and in such a way to extract some features. *



Different types of convolutions:

- 1D conv: kernel slide in 1D (e.g. sequence of words)
- 2D conv: kernel slide in 2D (e.g. image)
- 3D conv: kernel slide in 3D (e.g. video)

2) **Activation function (detector)**: Convo layer also contains use nonlinear activation function such as ReLU to make all negative value to zero.

3) **Pooling**: used to reduce the spatial volume of input image after convolution. It is an averaging of some sort (usually max or average) used to implement invariance to local translations.

If applied with a **stride** ≥ 0 then it reduces the dimension of the output.

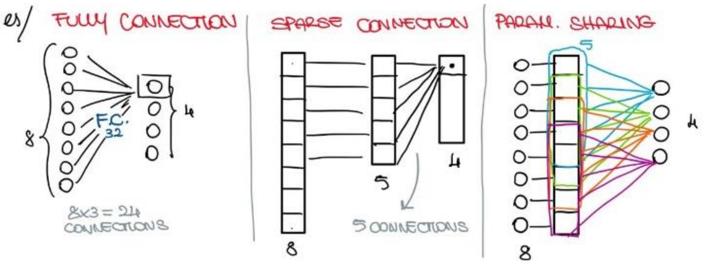
4) **Fully Connected Layer(FC):** involves weights, biases, and neurons. It connects neurons in one layer to neurons in another layer. It is used to classify images between different category by training.

5) **Softmax / Logistic Layer** resides at the end of FC layer. Logistic is used for binary classification and softmax is for multi-classification.

- **Stride** denotes how many steps we are moving in each steps in convolution (by default it is one). We can observe that the size of output is smaller that input.

- To maintain the dimension of output as in input , we use **Padding**: it is a process of adding zeros to the input matrix symmetrically.
  In Keras, this is specified via the "padding" argument on the Conv2D layer, which ha the default value:
  - **Valid** (no padding, $p = 0$): the filter is applied only to valid ways to the input.
  - **Same**: calculates and adds the padding required to the input image to ensure that the output has the same shape as the input ($p = w_k/2$)

- In order to reduce **overfitting**:
  - **Sparse connectivity:** outputs depend only on few inputs
  - **Parameter sharing**: learn only one set of parameters (for the kernel) shared to all the units k parameters instead of $m \times n$ (NB: $k \ll m$)

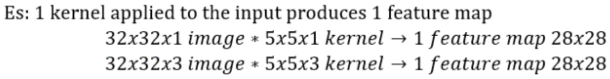


Parameter size:

Consider input size $w_{in} \times h_{in} \times d_{in}, d_{out}$ kernels of size $w_k \times h_k \times d_{in}$, stride s and padding p.

- dimension of the output **feature map** is:
  $w_{out} = \frac{w_{in} - w_k + 2p}{s} + 1$ and $h_{out} = \frac{h_{in} - h_k + 2p}{s} + 1$
- number of **trainable parameters** of the convolutional layer is:
  $$|\theta| = w_k \cdot h_k \cdot d_{in} \cdot d_{out} + d_{out}$$

(NB: $param. in\ the\ convolutional\ layer \ll param. in\ the\ F.C. layer$)



Es: 1 kernel applied to the input produces 1 feature map

$32x32x1\ image * 5x5x1\ kernel \rightarrow 1\ feature\ map\ 28x28$

$32x32x3\ image * 5x5x3\ kernel \rightarrow 1\ feature\ map\ 28x28$

Famous" CNNs

- **LeNet**: designed to recognize handwritten digits, and consists of a series of convolutional and pooling layers followed by F.C. (dense) layers. LeNet is a relatively small and simple CNN, but it was an important step in the development of modern CNNs.
- **AlexNet**, which won the ILSVRC in 2012 and significantly advanced the state of the art in image classification. Not commonly used anymore.
- **VGG**, which won the ILSVRC in 2014 and is known for its deep and narrow architecture. Commonly used also today.
- **Inception** by Google, which won the ILSVRC in 2014 and introduced the concept of "inception modules" which make use of multiple parallel convolutional and pooling layers. Commonly used also today.
- **ResNet** by Microsoft, which won ILSVRC 2015 and introduced the concept of residual connections, which made it possible to train very deep CNNs effectively.

Transfer Learning

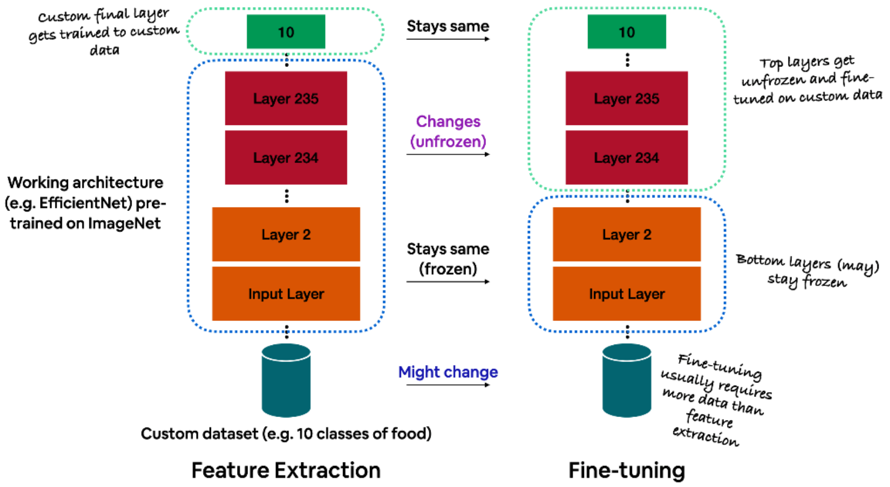
Transfer learning is a technique that involves using a pre-trained CNN as a starting point for training a new task, usually by adding additional layers or fine-tuning the existing layers on a new dataset. It is a useful technique when the amount of labeled data available for the new task is limited, as it allows the model to make use of the knowledge it has acquired from a related task.

GOAL *→ improve learning of f_T of the* ***t****arget learning task, using the knowledge in the* ***s****ource domain D_S and* ***s****ource learning task T_s (i.e., after training f_S)*

There are two main approaches to transfer learning:

1. **Fine-tuning**: consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model and these top layers.
   PRO: better performance as the model can learn task-specific features that are not present in the pre-trained model.
   CON: more computationally expensive than feature extraction.

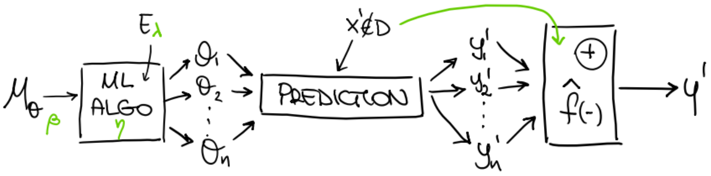
2. **Feature extraction:** This involves using the pre-trained model as a fixed feature extractor, where the output of the pre-trained model's layers is fed as input to a new model that is trained to perform the target task. The weights of the pretrained model are not updated during training.
   PRO: no need to train the CNN!
   CON: cannot modify features, source and target domains should be as compatible as possible.


Custom final layer gets trained to custom data
10
Layer 235
Layer 234
Layer 2
Input Layer
Working architecture (e.g. EfficientNet) pre-trained on ImageNet
Custom dataset (e.g. 10 classes of food)
Feature Extraction
Stays same
Changes (unfrozen)
Stays same (frozen)
Might change
10
Layer 235
Layer 234
Layer 2
Input Layer
Top layers get unfrozen and fine-tuned on custom data
Bottom layers (may) stay frozen
Fine-tuning usually requires more data than feature extraction
Fine-tuning

The main difference between the two is that in fine-tuning, more layers of the pre-trained model get unfrozen and tuned on custom data. This fine-tuning usually takes more data than feature extraction to be effective.

13. Multiple Learner

The main idea is to train multiple models/learners to solve a more complex model, by combining their results. By training multiple models and combining their predictions, it is often possible to achieve better results than with a single model. Another reason is to reduce the risk of overfitting.



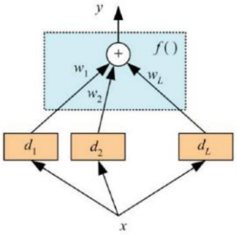
Models can be trained in parallel (**voting** or **bagging**) or in sequence (**boosting**).

Voting: simple method in which the models are trained in parallel on the same dataset *D* and then the outputs are summed (for regression) or chosen based on the most voted class (for classification).

1. use D to train a set of models $y_m(x)$, for $m = 1, \ldots, M$
2. make predictions with

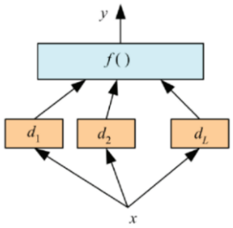
$$y_{voting}(\mathbf{x}) = \sum_{m=1}^{M} w_m y_m(\mathbf{x}) \qquad \text{(regression)}$$

$$y_{voting}(\mathbf{x}) = \underset{c}{\operatorname{argmax}} \sum_{m=1}^{M} w_m I(y_m(\mathbf{x}) = c) \qquad \text{weighted majority (classification)}$$

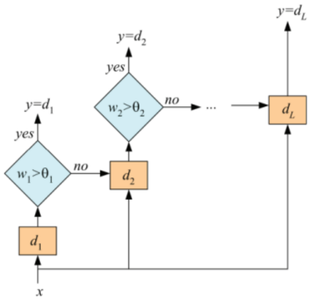


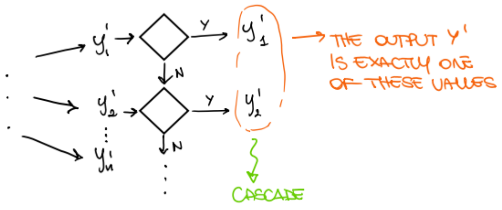
With $w_m \geq 0$, $\sum_m w_m = 1$ (prior probability of each model), $I(e) = 1$ if e is true, 0 otherwise.

Stacking: it is an ensemble method in which you train some base learners on the original data and then use them to make predictions on a hold-out set. These predictions are then used as input features for the second-level model, which is trained to predict the target variable using the base learner predictions as input.



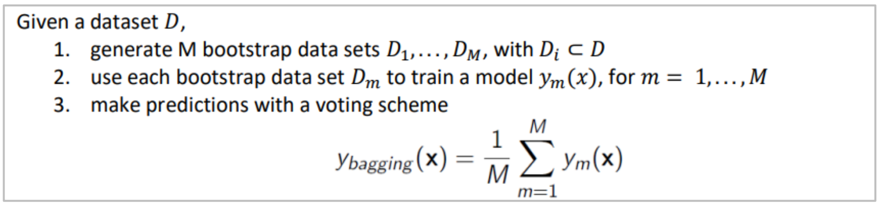
Cascading: ensemble learning based on the concatenation of several classifiers, using all information collected from the output from a given classifier as additional information for the next classifier in the cascade.





←Cascade learners based on confidence thresholds.

Bagging: is an ensemble method that involves training multiple models in parallel on random subsets of the training data. The final prediction is made by averaging the predictions of all the models.

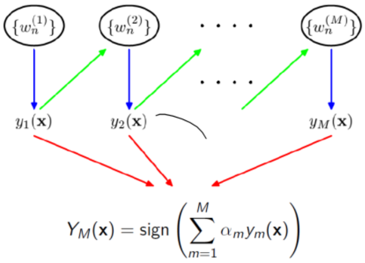


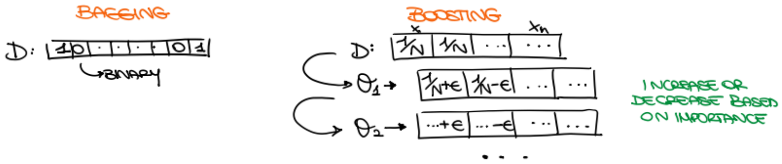
Given a dataset D,

1. generate M bootstrap data sets $D_1, \dots, D_M$, with $D_i \subset D$
2. use each bootstrap data set D_m to train a model $y_m(x)$, for $m = 1, \dots, M$
3. make predictions with a voting scheme

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

Boosting: is an ensemble method that involves training on weighted data multiple models sequentially (base classifier), with each model attempt to correct the mistakes of the previous model (points misclassified by previous classifiers are given greater weight). The final prediction is based on weighted majority of votes.

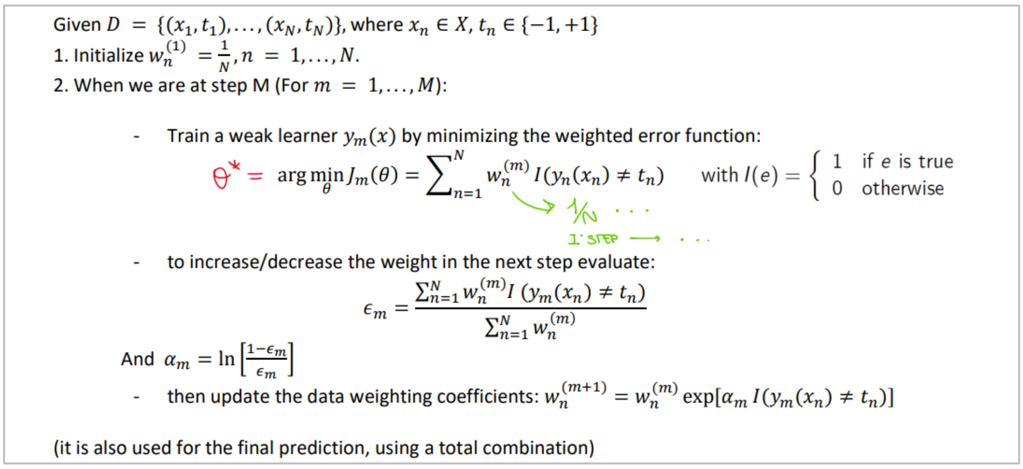




Adaboost: is a boosting algorithm which works by weighting the training data points that are misclassified by the weak learners in such a way that the next weak learner focuses more on the misclassified examples.

This process is repeated for a predetermined number of rounds or until a satisfactory level of accuracy is achieved.

The final prediction of the AdaBoost model is made by taking a weighted average of the predictions of all the weak learners, with the weights reflecting the learners' importance or accuracy.



Given $D = \{(x_1, t_1), \dots, (x_N, t_N)\}$, where $x_n \in X$, $t_n \in \{-1, +1\}$

1. Initialize $w_n^{(1)} = \frac{1}{N}, n = 1, \dots, N$.

2. When we are at step M (For $m = 1, \dots, M$):

- Train a weak learner $y_m(x)$ by minimizing the weighted error function:

$$\theta^* = \arg\min_{\theta} J_m(\theta) = \sum_{n=1}^{N} \underbrace{w_n^{(m)}}_{\to 1/N \cdots} I(y_n(x_n) \neq t_n) \qquad \text{with } I(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

1° STEP → . . .

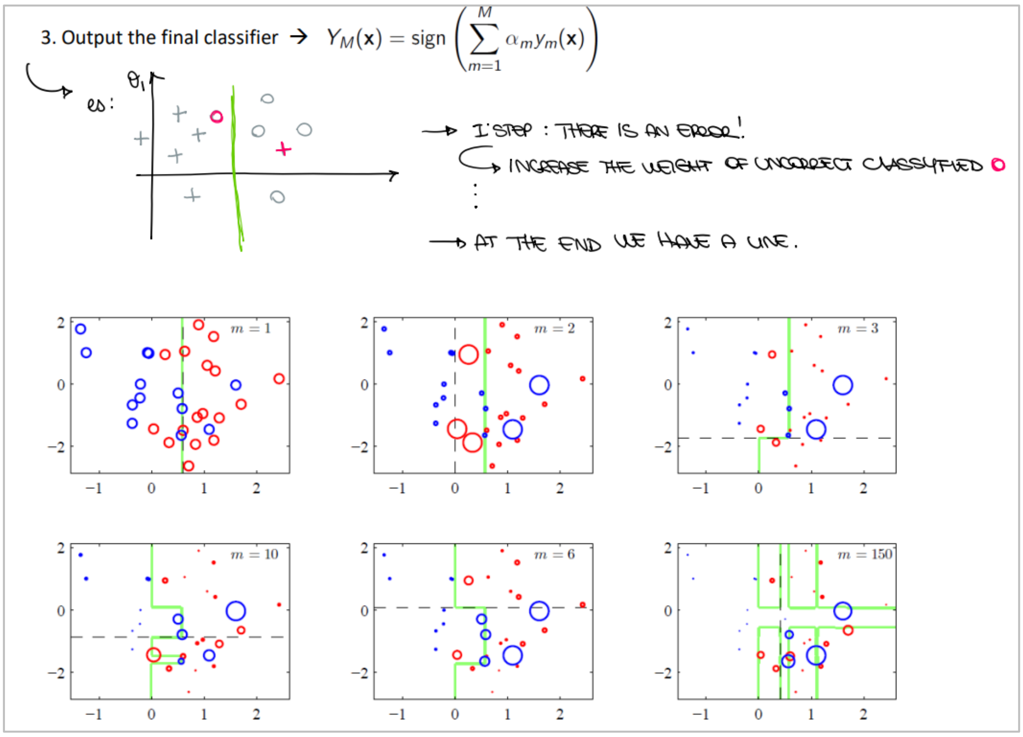
- to increase/decrease the weight in the next step evaluate:

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I\left(y_m(x_n) \neq t_n\right)}{\sum_{n=1}^{N} w_n^{(m)}}$$

And $\alpha_m = \ln\left[\frac{1-\epsilon_m}{\epsilon_m}\right]$

- then update the data weighting coefficients: $w_n^{(m+1)} = w_n^{(m)} \exp[\alpha_m I(y_m(x_n) \neq t_n)]$

(it is also used for the final prediction, using a total combination)



3. Output the final classifier → $Y_M(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right)$

↪ es:

→ I' STEP : THERE IS AN ERROR!

↪ INCREASE THE WEIGHT OF UNCORRECT CLASSYFIED ○

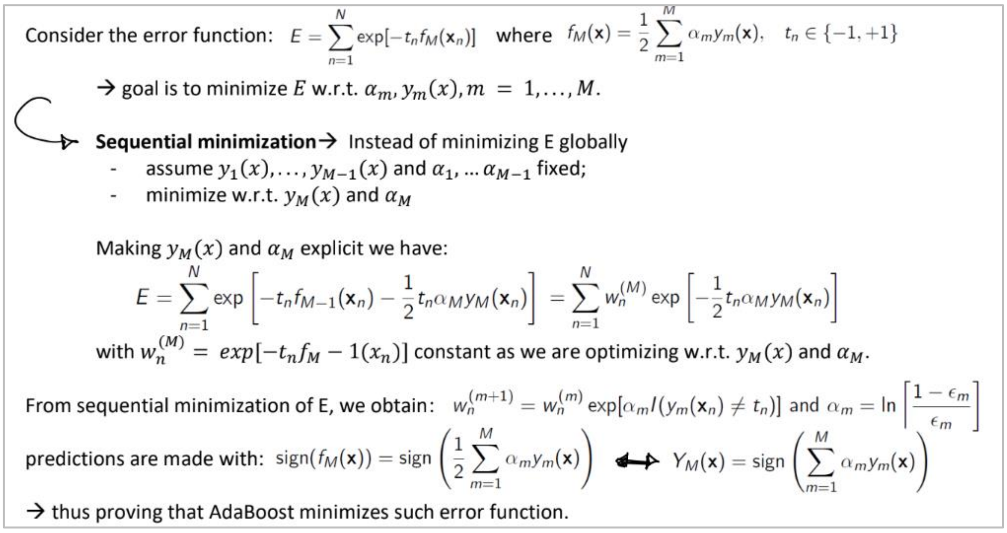
⋮

→ AT THE END WE HAVE A LINE.

It outperforms many other base learners in many problems.

AdaBoost can be explained as the **sequential minimization** of **an exponential error function**:

In AdaBoost, the weights are adjusted after each classifier is trained with the weights of misclassified examples being increased and the weights of correctly classified examples being decreased, in such a way that the error of the current classifier is minimized exponentially. with the weights of misclassified examples being increased and the weights of correctly classified examples being decreased.



Consider the error function: $E = \sum_{n=1}^{N} \exp[-t_n f_M(\mathbf{x}_n)]$ where $f_M(\mathbf{x}) = \frac{1}{2}\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})$, $t_n \in \{-1, +1\}$

→ goal is to minimize E w.r.t. $\alpha_m, y_m(x), m = 1, \ldots, M$.

↪ **Sequential minimization→** Instead of minimizing E globally

- assume $y_1(x), \ldots, y_{M-1}(x)$ and $\alpha_1, \ldots \alpha_{M-1}$ fixed;
- minimize w.r.t. $y_M(x)$ and α_M

Making $y_M(x)$ and α_M explicit we have:

$$E = \sum_{n=1}^{N} \exp\left[-t_n f_{M-1}(\mathbf{x}_n) - \frac{1}{2} t_n \alpha_M y_M(\mathbf{x}_n)\right] = \sum_{n=1}^{N} w_n^{(M)} \exp\left[-\frac{1}{2} t_n \alpha_M y_M(\mathbf{x}_n)\right]$$

with $w_n^{(M)} = exp[-t_n f_M - 1(x_n)]$ constant as we are optimizing w.r.t. $y_M(x)$ and α_M.

From sequential minimization of E, we obtain: $w_n^{(m+1)} = w_n^{(m)} \exp[\alpha_m I(y_m(\mathbf{x}_n) \neq t_n)]$ and $\alpha_m = \ln\left[\frac{1-\epsilon_m}{\epsilon_m}\right]$

predictions are made with: $\text{sign}(f_M(\mathbf{x})) = \text{sign}\left(\frac{1}{2}\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right) \leftrightarrow Y_M(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right)$

→ thus proving that AdaBoost minimizes such error function.

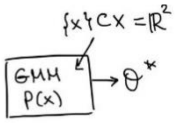
14. Unsupervised Learning

This kind of learning is used when a dataset does not have any label, so we need to cluster similar samples x based of some other metrics.

Gaussian Mixture Model (GMM)

One way to do it is to assume the data is generated from a mixture (sum) of a finite number k of Gaussian distribution with unknown parameters.
Given a dataset, a GMM estimates the parameters of each of the component Gaussians and the mixture weights, which represent the proportion of the data generated by each component. The probability density function of a GMM with K components is given by:

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



→Unsupervised Learning algorithms determine mixed probability distributions from data!

Each instance x_n is generated by:

- Choosing k according to prior probabilities $[\pi_1, \dots, \pi_K]$
- Generating an instance at random according to that Gaussian, thus using μ_k, Σ_k

- INVERSE WAY: from $(\pi_k,\ \mu_k, \Sigma_k)$, and by using $P(x)$ you find data.
- NORMAL WAY: making some assumptions (uniform $\pi_k = 1/k$, same covariance $\Sigma_k = \Sigma_k{}'$)→ we want to estimate the **k-means** of data.

K-means

The K-means is a clustering algorithm that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. It is an iterative algorithm that starts by randomly initializing k centroids, then assigns each observation to the cluster corresponding to the closest centroid. The centroids are then updated based on the mean of the points in each cluster, and the process is repeated until convergence.

Step 1. Initialize a decision on the value of k (= number of clusters you want to compute) randomly,

clustering

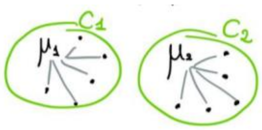
Step 2. Assign the training samples as follow:

- Take the first k training samples as single-element clusters
- Assign each of the remaining (N- k) training samples to the cluster with the nearest centroid. After each assignment, recompute the centroid of the new cluster.

Step 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 it to that cluster and update the centroid of the two clusters involved in the switch.

Step 4. Repeat step 3 until convergence is achieved, that is until a pass through the training sample causes no new assignments.

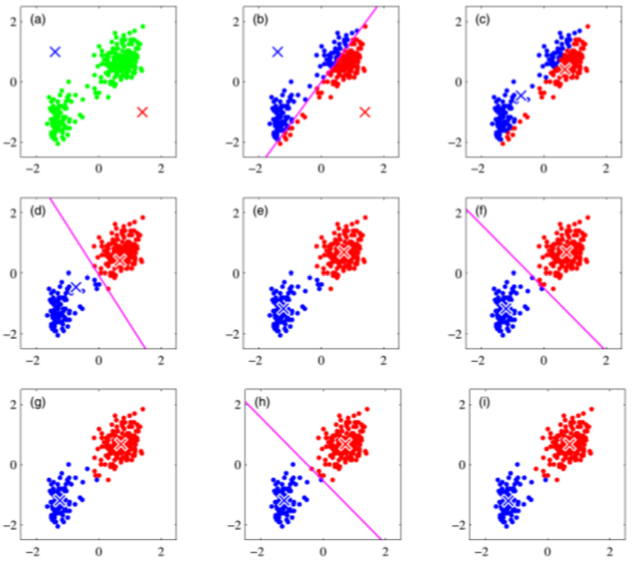
↳ es: 2-MEANS ⇒ μ_1, μ_2



CONS:

- sensitive to initial conditions, it can get stuck in local minima when a few data available.
- Not robust to outliers. Very far data from the centroid may pull the centroid away from the real one.

K-means example



(a) start from a distribution of input random and put two samples (1 blue and 1 red)

(b) portion the space

(c)now consider the red point and compute the mean. It will be where is the majority of red points

(d) compute new cluster (forget the past one)

Expectation Maximization (EM)

The EM algorithm is an iterative method for finding maximum likelihood or maximum a posteriori (MAP) estimates of parameters when the model depends on unobserved latent variables z_{nk}*.

(NB: it is an extended version of the K-means algorithm)

Step 1. Initialize the model parameters randomly μ_k^0 , π_k^0 , $\sum_k^0$

Step 2. Repeat until termination condition $t \ = \ 0, \ldots, T$:

- **E-STEP:** compute the expected value of the complete data log likelihood function given the current parameter estimates

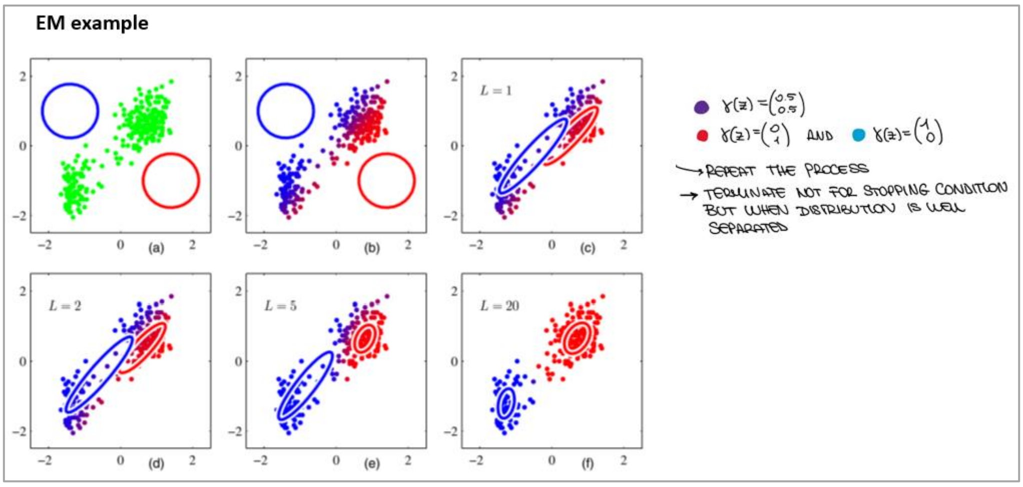
$$\gamma(z_{nk})^{(t+1)} = \frac{\pi_k^{(t)} \mathcal{N}(\mathrm{x}_n; \boldsymbol{\mu}_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} \mathcal{N}(\mathrm{x}_k; \boldsymbol{\mu}_j^{(t)}, \Sigma_j^{(t)})}$$

- **M-STEP:** maximize the expected value of the complete data log likelihood function with respect to the model parameters

$$\pi_k^{(t+1)} = \frac{N_k}{N}, \quad with \;\; N_k = \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)} \qquad \boldsymbol{\mu}_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)} \mathrm{x}_n$$

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

Step 3. Return the final estimates of the model parameters.



EM example

● $\gamma(z) = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$

● $\gamma(z) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ AND ● $\gamma(z) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

→ REPEAT THE PROCESS

→ TERMINATE NOT FOR STOPPING CONDITION BUT WHEN DISTRIBUTION IS WELL SEPARATED

General EM problem

Given:

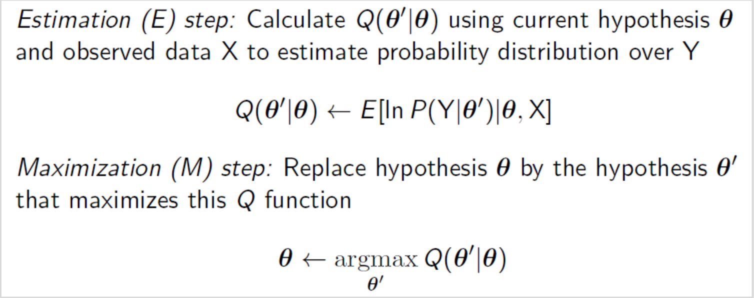
- Observed data $X = \{x_1, \ldots, x_N\}$
- Unobserved latent values $Z = \{z_1, \ldots, z_N\}$
- Parametrized probability distribution $P(Y|\theta)$, where

▪ $Y = \{y_1, \ldots, y_N\}$ is the full data $y_n = < x_n, z_n >$, zni

▪ θ are the parameters

Determine the values of the model parameters that best explain the observations:

- θ^* that (locally) maximizes $E[ln\ P(Y|\theta)]$



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

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

Maximization (M) step: Replace hypothesis θ by the hypothesis θ' that maximizes this Q function

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

*

Gaussian Mixture Model

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

Introduce new variables $z_k \in \{0, 1\}$, with $z = (z_1, \dots, z_K)^T$ using a 1-out-of-K encoding ($z_k = 1$ only for one value of k, 0 otherwise).

Let's define $P(z_k = 1) = \pi_k$, → thus: $P(z) = \prod_{k=1}^{K} \pi_k^{z_k}$

For a given value of z: $P(x|z_k = 1) = \mathcal{N}(x; \mu_k, \Sigma_k)$ → $P(x|z) = \prod_{k=1}^{K} \mathcal{N}(x; \mu_k, \Sigma_k)^{z_k}$

Joint distribution: $P(x, z) = P(x|z)P(z)$ (chain rule).

When z are variables with 1-out-of-K encoding and $P(z_k = 1) = \pi_k$

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

GMM distribution $P(x)$ can be seen as the marginalization of a distribution $P(x, z)$ over variables z.

Given observations $D = \{(x_n)_{n=1}^{N}\}$, each data point x_n is associated to the corresponding variable z_n which is unknown.

Note: $z_{nk} = 1$ denotes x_n sampled from Gaussian k. $\boldsymbol{z_n}$ are called **latent variables**.

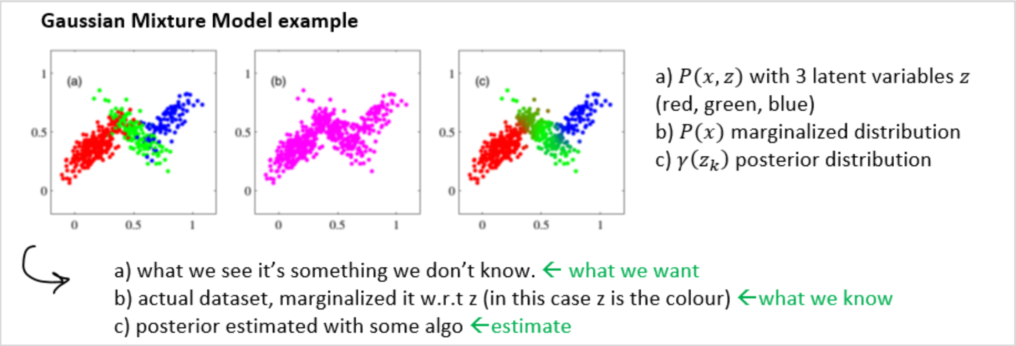
→ Analysis of latent variables allows for a better understanding of input data (e.g., dimensionality reduction).

Let's define the **posterior**

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

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

- π_k : prior probability of z_k
- $\gamma(z_k)$: posterior probability after observation of x.



Gaussian Mixture Model example

a) $P(x, z)$ with 3 latent variables z (red, green, blue)
b) $P(x)$ marginalized distribution
c) $\gamma(z_k)$ posterior distribution

a) what we see it's something we don't know. ← what we want
b) actual dataset, marginalized it w.r.t z (in this case z is the colour) ←what we know
c) posterior estimated with some algo ←estimate

15. Dimensionality Reduction

Dimensionality reduction is the process of reducing the number of features or variables in a dataset while preserving as much information as possible. Dimensionality reduction aims at identifying the real/intrinsic degrees of freedom of a data set. This can be done for a variety of reasons, such as to reduce noise in the data, to make the data more visually comprehensible, or to reduce the computational cost of certain algorithms.

Latent Variable: are variables that are not directly observed in the data but are inferred from the observed data. For ex. If we have an input as image we have many degrees of freedom and each composition of each parameter can generate a sample of our dataset, but some configurations have no meaning. The goal is to identify these variables and use them to represent the data in lower-dimensional space ($smaller\ dimensional\ space \leftrightarrow\ more\ representative\ of\ the\ problem$).

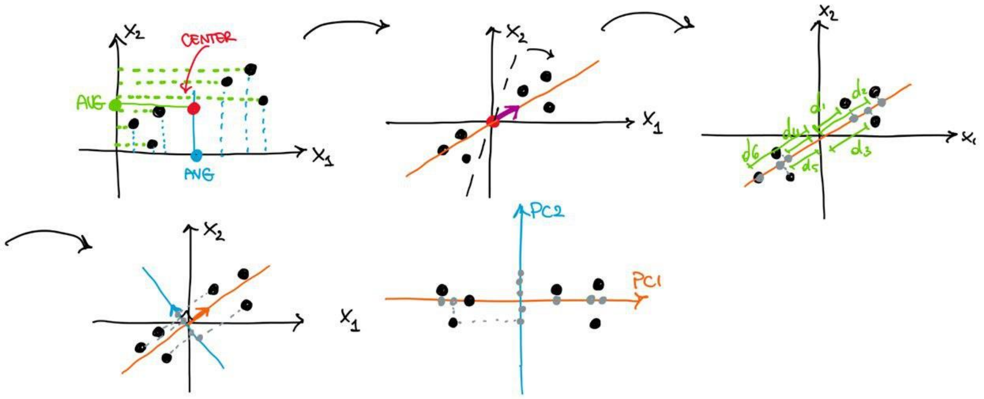
→ **Dimensionality reduction:** how to transform a problem in many dimensions in much less dimension, with the goal of keeping as much as possible the information.

Principal Component Analysis (PCA)

PCA is a linear technique for dimensionality reduction that is based on finding the directions of **maximum variance** in the data. PCA finds a set of orthogonal axes, called *principal components*, that capture the most information in the data. Data can then be projected onto these principal components to obtain a lower-dimensional representation of the data.

General idea:

- Start by calculate the average for x_1 and x_2 to get the center of the data.
- Shift the data in such a way the center of the graph is the center of the data and fit a line that fits the samples. Start with a random line until the best fit which is when, by projecting the samples on the line, that line minimizes the distance sample-projection or maximizes the distances projected point-origin = maximize the sum of squared distances (= eigenvalue for PC1). This line is the *principal component PC1*, which is a linear combination of x_1 and x_2
- The vector of that line is the singular vector or eigenvector for PC1
  When you do PCA with SVD, the vector is scaled so that this length =1
- In 2D the PC2 is simply the line through the origin that is perpendicular to PC1.
- Now rotate until horizontal for the final plot and use the projections to draw the samples



Projected points: $x_n^T u_1$
Note: $u_1^T u_1 = 1$

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

Data-centered matrix X (N × d): $\mathrm{X} = \begin{bmatrix} (\mathrm{x}_1 - \bar{\mathrm{x}})^T \\ \cdots \\ (\mathrm{x}_n - \bar{\mathrm{x}})^T \\ \cdots \\ (\mathrm{x}_N - \bar{\mathrm{x}})^T \end{bmatrix}$

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

Math steps:

Given data $\{x_n\} \in R^d$

1. Compute the *Covariance* of the dataset

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

2. Compute the *eigenvectors* $\{u_i\}$ and *eigenvalues* $\{\lambda_i\}$ of the covariance matrix: The eigenvectors of the covariance matrix are the *principal components*, and the eigenvalues are their corresponding variances. → we want to maximize the variance and it is when the eigenvector corresponds to the largest eigenvalue→ $\max_{u_1} u_1^T S\, u_1$

Left-multiplying by u_1^T and using $u_1^T u_1 = 1$, we have: $u_1^T\, S\, u_1 = \lambda_1$
Which is the variance after the projection.

We maximize this term → find the highest eigenvalue

3. Order the eigenvalues in decreasing order and select the k eigenvectors with the largest eigenvalues. By selecting the top k principal components, we are able to retain as much information as possible while still reducing the dimensionality of the data.

$$\{x_n\} \to S \longrightarrow \begin{array}{c} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \vdots \\ \lambda_D \\ \vdots \\ \lambda_M \end{array} \quad \begin{array}{c} \mu_1 \\ \vdots \\ \vdots \\ \mu_D \end{array}$$

4. Use this eigenvector matrix to transform the original dataset into a new k-dimensional subspace by computing the matrix multiplication.

PCA for high-dimensional data: If number of points is smaller than the dimensionality, i.e. $N < d$ → At least d-N+1 eigenvalues of S are zero.
Then consider $u_i = \frac{1}{\sqrt{N\lambda_i}} X^T v_i$ where v_i eigenvectors of XX^T

Probabilistic PCA

It is an extension of PCA that models the data as being generated by a probabilistic process. This allows for the data to be represented by a low-dimensional latent variable, which can be useful in cases where the data is noisy or incomplete. The goal is to find a set of latent variables that best explains the data distribution.

- Assume data as a linear combination of a low-dimensional latent variable Z, with some added Gaussian noise $x = Wz + \mu$
- Assume Gaussian distribution of z: $P(z) = N(z; 0, I)$
- Assume linear-gaussian relationship between latent variables and data

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

The goal is to find the parameters of the model (W, μ, Σ) that best explains the data distribution. This is done by maximizing the likelihood of the data, given the model parameters:

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

setting derivatives to 0, we have a closed form solution: $\mu_{ML} = \bar{\mathrm{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathrm{x}_n \quad \mathbf{W}_{ML} = \ldots \quad \sigma_{ML}^2 = \ldots$

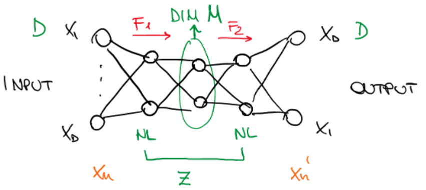
Maximum likelihood solution for the probabilistic PCA model can be obtained also with EM algorithm.

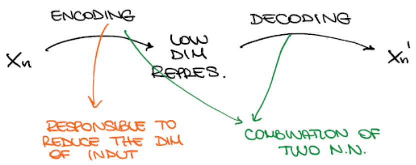
Linear representations are not sufficient for complex data, if you use PCA different points are projected equal.
→ *how to deal with Non-Linear transformation?* → **Non-linear latent variable models** (Autoencoders, GANs...)**.** These models use non-linear functions to map the observed data to the latent space.

Autoassociative Neural Networks (Autoencoders)

These are NN with reduced sized hidden layers (bottlenecks) in the middle which learn to reconstruct their input by minimizing a loss function.
It is a combination of NN: an **encoder**, which maps the input data to a lower-dimensional representation and the **decoder** which maps the lower-dimensional representation back to the original data.

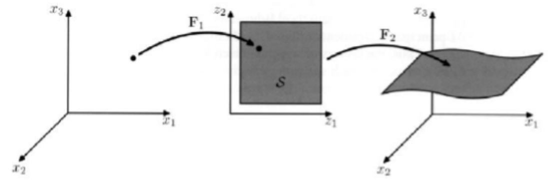
ENC
DEC
Z
GENERATES NEW SAMPLES



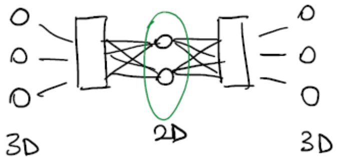


Function that transforms the input in intermediate value z (latence space) and then reconstruct in $x_n{}'$.
The **GOAL** is to minimize the difference between the input and the reconstruction ($x_n' = x_n$), this is typically done by training the model to minimize the reconstruction error (typically the MSE).

Autoencoder example:
Input: 3-D, Hidden layer: 2-D, Output: 3-D



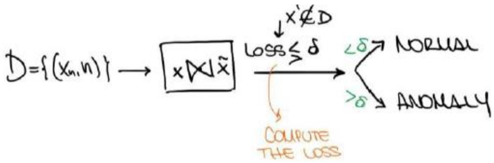
The 3D reconstruction from 2D hidden layer generates a 2D manifold (surface) in 3D.



8 values autoencoder
Data set: {0, . . . , 7} with 1-out-of-8 encoding: $\{< 1, 0, \dots, 0 >, < 0, 1, \dots, 0 >, \dots, < 0, 0, \dots, 1 >\}$
Autoencoder with 1 hidden layer with 3 nodes.

By minimizing the reconstruction error, the autoencoder learns a compressed representation of the data that captures the most important features and discards the noise.

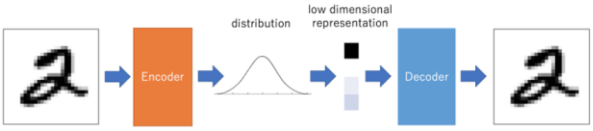
Autoencoders are very useful for **anomaly detection** (one-class classification):



Generative Models are models that learn to generate new data from some underlying probability distribution. Can be used for a variety of tasks, such as image synthesis, language generation and as a way to extract feature from a dataset.

Variational Autoencoder (VAE)

They are generative models that combine the encoder-decoder architecture of autoencoders with the probabilistic modelling framework: instead of encoding an input as a single point, we encode it as a *distribution* over the latent space.
It can be defined as being an autoencoder whose training is regularised to avoid overfitting and ensure that the latent space has good properties that enable generative process.



To produce a distribution consider parametric distributions (typically Gaussian)
The VAE generative model can be represented as:

$$p(x \mid z) = N(x \mid f(z), I) \; q(z \mid x) = N(z \mid g(x), \Sigma)$$

Where N is a Gaussian distribution, z is the latent variable and x is the data, f and g are the decoder and encoder networks.

The objective function of VAE is:

$$L = E[log\, p(x|z) - KL(q(z|x) \,||\, p(z))]$$

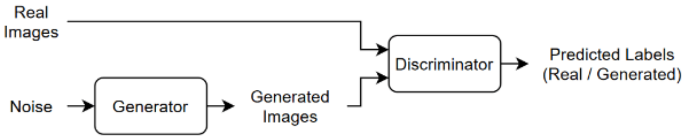
Where the first term is the likelihood of the data and the second term the ***Kullback-Leibler divergence***, is used as a regularization term, to ensure the encoder $q(z|x)$ approximates the prior distribution of the latent variable, $p(z)$.

- To prevent degeneration? → add loss term based on *Kullback-Leibler divergence* (Evidence Lower Bound)
- Sampling operation is not differentiable → *re-parametrization*

Generative Adversarial Networks (GANs)

GANs are an approach that can generate data with similar characteristics as the input real data. The idea is to use an inverted CNN and an adversarial training.
A GAN consists of two networks that train together:

- **Generator** (decoder) which given a vector of random values (latent inputs) as input, generates data similar to the real data;
- **Discriminator** (critic) which is trained to identify if the observations are from the generated or from the real data.



To train a GAN, train both networks simultaneously to maximize the performance of both (making the network competing with each other):

- Train the **generator** by using the entire model (generator + discriminator), with discriminator layers fixed with a batch of data $\{(r_k, Real)\}$, to generate data that "fools" the discriminator in believing that the sample is "real"
  (r_k are random values of the latent variable).
  → maximize the loss of the discriminator when given generated data;

- Train the **discriminator** with a batch of data $\{(x_n, Real)\}, \{(x'_m, Fake)\}$ to distinguish between *Real* and *Generated* data as good as possible.
  → minimize the loss of the discriminator when given batches of both real and generated data

Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.

16. MDP and RL

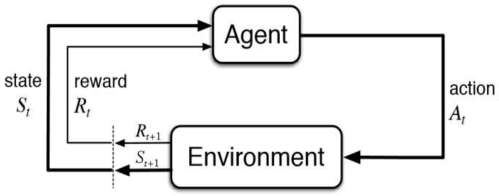
In a dynamic system, when the state is *fully observable*, the decision-making problem for an agent is to decide which **action** must be executed in a given state.

Reinforcement Learning: Learning a behaviour function $\pi: X \rightarrow A$, given
$D = \{< x_1, a_1, r_1, \ldots, x_n, a_n, r_n >_i\}$.
In a mathematical framework we consider the so called **Markov Decision Process**.

Markov Decision Process (MDP)

MDP is a discrete-time control process. It provides a mathematical framework for modelling decision-making in situations where outcomes are partly random and partly under the control of a decision maker. In MDP an agent (e.g. a human) observes the environment and takes actions.

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



At each time step, the process is in some state, and the decision maker may choose any available action. The process responds at the next time step by randomly moving into a new state and giving the decision maker a corresponding reward.

A MDP follows the ***Markov properties*** which state:

- Once the current state is known, the evolution of the dynamic system does not depend on the history of states, actions and observations.
- The current state contains all the information needed to predict the future.
- Future states are conditionally independent of past states and past observations given the current state.
- The knowledge about the current state makes past, present and future observations statistically independent.

Given an MDP, we want to find an optimal **policy *π***: $X \rightarrow A$, that takes as input some state x_t and chooses an action a_t to maximize the reward r_t.
Optimality = maximizing the (expected value of the) cumulative discounted reward:

$$V^{\pi}(x_1) = E[\bar{r}_1 + \gamma\bar{r}_2 + \gamma^2\bar{r}_3 + \ldots]$$

where $\bar{r}_t = r(x_t, a_t, x_{t+1})$, $a_t = \pi(x_t)$ and $\gamma \in [0,1]$ is the discount factor for future rewards.
Optimal policy: $\boxed{\pi^* = \arg\max_{\pi} V^{\pi}(x)}$ $\forall x \in X$.

- π^* is an optimal policy iff for any other policy π: $V^{\pi^*}(x) \geq V^{\pi}$, $\forall x$.
- For infinite horizon problems, a stationary MDP always has an optimal stationary policy.

One-state Markov Decision Processes (MDP)

$MDP =< \{x_0\}, A, \delta, r >$

- x_0 unique state
- A finite set of actions
- $\delta(x_0, a_i) = x_0, \ \forall a_i \in A$ transition function
- $r(x_0, a_i, x_0) = r(a_i)$ reward function

The optimal policy: $\pi^*(x_0) = a_i$.

1. If $r(ai)$ is **deterministic** and **known,** → we can compute without experiments the optimal policy: $\pi^*(x_0) = \arg\max_{ai\in A} r(ai)$
2. If $r(ai)$ is **deterministic** and **unknown,** → we do experiments, we execute action a_i and collect reward r_i and then the optimal policy: $\pi^*(x_0) = a_i$ with $i = argmax_{i=1\ldots|A|}\ r_{(i)}$ (NB: |A| iterations needed).
3. If $r(ai)$ is **non-deterministic** (gaussian distribution) and **known**, → we have information about the gaussian distribution, so we know the mean, so the optimal policy: $\pi^*(x_0) = argmax_{a_i\in A}\ E[r(a_i)]$. (no test needed).
4. If $r(ai)$ is **non-deterministic** and **unknown**, →we have no info for r →compute test (30 times) and compute the average.
   NB: if we collect info over time maybe we can know at a certain point if ai is better than aj and use less iterations.

When <u>δ and r are not known</u>, the agent cannot predict the effect of its actions. But it can execute them and then observe the outcome.

Approaches to Learning with MDP:

- **Policy iteration:** estimate directly π^*
- **Value iteration:** estimate the value function V^{π^*} from which it could determine the optimal policy π^*:

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

Only if δ and r are known.

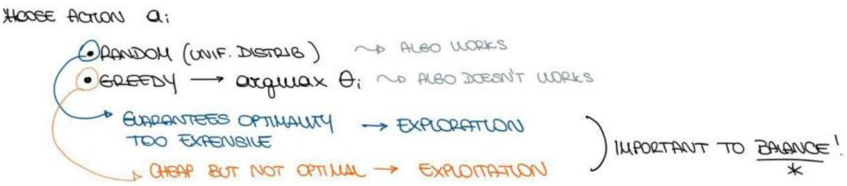
To determine the optimal policy without knowing δ and r, the agent learns **Q-table**, we get the policy just observing new state x' and immediate reward $\bar{r}$, after the execution of the chosen action.

Exploration-Exploitation trade-off

During trials, an agent has a set of actions to select from:

- some are previously selected→ **exploiting** what the agent already knows, selecting the action that maximizes $\hat{Q}(x,a)$
- others are never taken before→ **exploring** a random action (low value of $\hat{Q}(x,a)$)

When the agent *explores*, it can improve its current knowledge and gain better rewards in the long run; when it *exploits*, it gets more reward immediately → we wish to keep a balance between exploration and exploitation; not giving up on one or another.



HOOSE ACTION a_i

- RANDOM (UNIF. DISTRIB) → ALSO WORKS
- GREEDY → $\arg\max\ \theta_i$ → ALSO DOESN'T WORKS

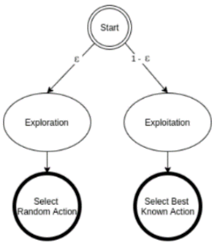
GUARANTEES OPTIMALITY → EXPLORATION
TOO EXPENSIVE

CHEAP BUT NOT OPTIMAL → EXPLOITATION

IMPORTANT TO BALANCE!

Action selection:

- **ε -greedy strategy:** It selects the best action with the highest estimated reward most of the time. It works by choose the best action with probability $1 - \varepsilon$ (exploitation) and a random action with probability ε (exploration). It selects the action with the highest estimated reward most of the time. ε can decrease over time to have a balance between exploration and exploitation (first exploration, then exploitation).



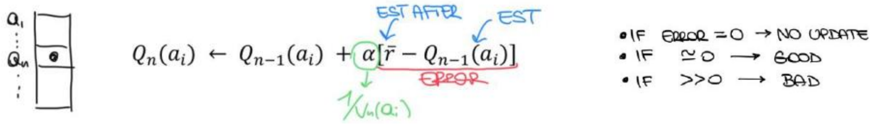
- **soft-max strategy:** it controls the relative levels of exploration and exploitation by mapping values into action probabilities: actions with higher $\hat{Q}$ are assigned higher probabilities, but every action is assigned a non-zero probability.

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

$k > 0$ determines how strongly the selection favours actions with $\hat{Q}$ high values.
k may increase over time (first exploration, then exploitation).

K-Armed bandit example:
The classic (stocastic) version of the k-armed bandit problem sees k slot machines, each one having some gaussian distribution of winning $N(\mu_i, \sigma_i) = r(a_i)$ and the goal is to earn the most money. In this context a RL agent can either:

- Perform x trials on each machine, estimate the mean winning rate and then choose the one with the highest.
- Adopt an ε -greedy strategy in which they play at random with prob ε and choose the optimal policy with $1 - \varepsilon$. In this case, the training rule is:



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

But if the probability μi changes (non-deterministic case) then the above solution would not work: the first would fail because μi can change after the x trials, the ε -greedy strategy will not reach an optimal value.

$\alpha = \frac{1}{1+v_{n-1}(a_i)}$ and $v_{n-1}(a_i)$ number of executions of action ai up to time $n - 1$.

Evaluating RL Agents: it is usually performed through the cumulative reward gained over time, which could be very noise → a better approach could be:

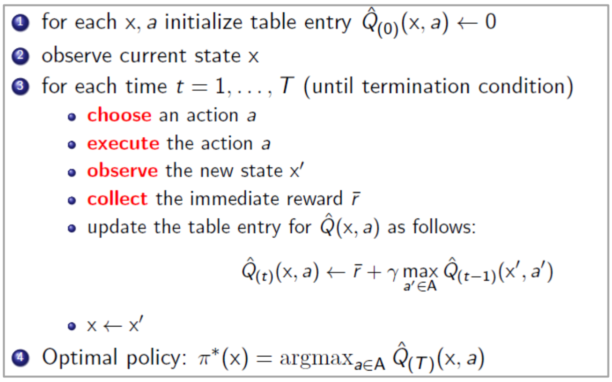
- Repeat: Execute k steps of learning and evaluate the current policy π_k
- Domain-specific performance metrics

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

$$Q(x,a) \equiv r(x,a) + \gamma V^{*}\big(\delta(x,a)\big) \quad \rightarrow \pi^{*} = \arg\max_{a \in A} Q(x,a)$$

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

→ $Q(x,a) \equiv r(x,a) + \gamma \max_{a \in A} Q(\delta(x,a), a')$ → Training rule: $\hat{Q}(x,a) \leftarrow \bar{r} + \gamma \max_{a'} \hat{Q}(x',a')$



```
❶ for each x, a initialize table entry $\hat{Q}_{(0)}(x,a) \leftarrow 0$
❷ observe current state x
❸ for each time $t = 1, \dots, T$ (until termination condition)
      • choose an action $a$
      • execute the action $a$
      • observe the new state $x'$
      • collect the immediate reward $\bar{r}$
      • update the table entry for $\hat{Q}(x,a)$ as follows:
            $\hat{Q}_{(t)}(x,a) \leftarrow \bar{r} + \gamma \max_{a' \in A} \hat{Q}_{(t-1)}(x',a')$
      • $x \leftarrow x'$
❹ Optimal policy: $\pi^{*}(x) = \text{argmax}_{a \in A} \hat{Q}_{(T)}(x,a)$
```

Q Function (non-deterministic case):

When the problem is **non deterministic** we know that the transition function $\delta(x,a)$ is some probability related to the current state and action:

$$P_a(x,x') = \mathrm{P}\,(x_{t+1} = x' | x_t = s, a_t = a).$$

To define our value function, we must take the expected values:

$V^{\pi}(x) = E[r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \dots]$ → the optimal policy: $\pi^{*} = \arg\max_{\pi} V^{\pi}(x)$

When we consider the reward function, to be non deterministic too, then we must include it in the value function:

$$Q(x,a) = E[r(x,a) + \gamma V^{*}\big(\delta(x,a)\big) = \dots = E[r(x,a)] + \gamma \sum_{x'} P(x'|x,a) \max_{a'} Q(x',a')$$

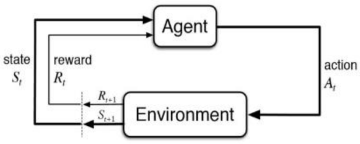
now the optimal policy shift to: $\pi^{*} = \arg\max_{a \in A} Q(x,a)$

Q learning generalizes to non-deterministic worlds with training rule:

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

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

with $\quad \alpha = \alpha_{n-1}(x,a) = \dfrac{1}{1 + visits_{n-1}(x,a)}$



Temporal Difference Learning

Temporal difference learning (TD) is a class of model-free RL methods which learn by bootstrapping* the current estimate of the value function. We can think of model-free algorithms as trial-and-error methods: the agent explores the environment and learns from outcomes of the actions directly, without constructing an internal model or a MDP, but by filling a **table** storing *state-action* pairs $Q(s,a)$.
There are two different TD algorithms:

- **On-policy** uses the same strategy for both the behaviour and target policy
- **Off-policy** algorithms use a different strategy for the behaviour and target policy

Q-Learning Algorithm

It updates the Q-value using the Q-value of the next state and the greedy action after that (off-policy). The goal is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward.
The formula that updates the Q-value is as follows:

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{current value}} \Bigg)}^{\text{temporal difference}}$$

This is called the **action-value function** or **Q-function**. The function approximates the value of selecting a certain action in a certain state.

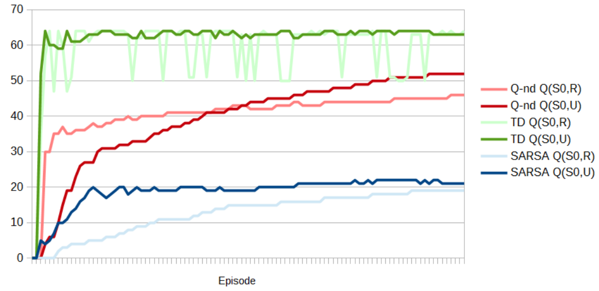
SARSA

SARSA (State–action–reward–state–action) is an **on-policy** as it uses an ε-greedy strategy for all the steps. It updates the Q-value using the Q-value of the next state and keeps following the policy for the next action.
In this case:

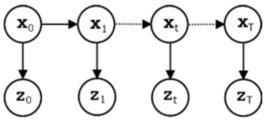
$$Q^{new}(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

<u>This means that SARSA uses the next action as a starting action in the next state (policy), where Q-Learning replaces it with the maximisation of the next action's reward.</u>

Convergence of non-deterministic algorithms:
Q-learning will converge faster to an optimal policy than SARSA. However, fast convergence does not imply better solution in the optimal policy.



*__bootstrapping__ usually refers to a self-starting process that is supposed to continue or grow without external input.



17. HMM and POMDP

Hidden Markov Model (HMM)

If the states x of a MDP are discrete and non-observable, but instead we have some observations z tied to them we are dealing with a **Hidden Markov Model** [**HMM**].

In this case the process $HMM = < X, Z, \pi_0 >$ is defined by:

- **observation model** $b_k(z_t) = P(z_t|x_t = k)$: a probability that a certain state is mapped with a specific observation
- **transition model**: $A_{ij} = P(x_t = j|x_{t-1} = i)$: probability that explain the transition to/from hidden states (emission probability).
- an **initial distribution** $\pi_0 = P(x_0)$ which define the probability that the model starts with the state x_0

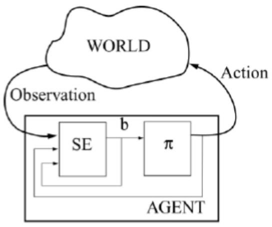
Training → Forward-backward algorithm: It is used to calculate the probability $P(x_t, z_{1:t})$ of a state at a certain time, when we know about the sequence of observations. Computing directly the probability would require marginalizing over all possible state sequences, the number of which grows exponentially with time ⍰the forward algorithm takes advantage of the conditional independence rules of the hidden Markov model (HMM) to perform the calculation recursively.

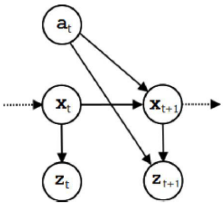
- Let $\alpha_t(x_t) = P(x_t, z_{1:t}) = \sum_{x_{t-1}} P(x_t, x_{t-1}, z_{1:t})$
- The HMM is based on augmenting the **Markov chain**, so we can apply the chain rule on the probabilities of the sequence (factorization):
  $P(x_t, z_{1:t}) = \sum_{x_{t-1}} P(z_t|x_t, x_{t-1}, z_{1:t-1}) P(x_t, x_{t-1}, z_{1:t-1}) P(x_{t-1}, z_{1:t-1})$
  $= P(x_0)P(z_0|x_0)P(x_1|x_0)P(z_1|x_1)P(x_2|x_1)P(z_2|x_2)\ldots$
- Thus, since $P(z_t \mid x_t)$ and $P(x_t|x_{t-1})$ are given by the model's distribution and transition probabilities, one can quickly calculate $\alpha_t(x_t)$ from $\alpha_{t-1}(x_{t-1})$ and avoid incurring exponential computation time.

Forward iterative steps → to compute $\alpha_t^k = P(x_t = k, z_{1:t})$	**Backward iterative steps** → to compute $\beta_t^k = P(z_{t+1:T}\|x_t = k)$
• For each state k do: • $\alpha_0^k = \pi_0 b_k(z_0)$ • For each time $t = 1, \ldots, T$ do: • For each state k do: • $\alpha_t^k = b_k(z_t) \sum_j \alpha_{t-1}^j A_{jk}$	• For each state k do: • $\beta_T^k = 1$ • For each time $t = T-1, \ldots, 1$ do: • For each state k do: • $\beta_t^k = \sum_j \beta_{t+1}^j A_{kj} b_j(z_{t+1})$

Learning: to determine the maximum likelihood estimate of the parameters (transition probabilities and emission probability):

- if states can be observed at training time → parameters can be estimated with statistical analysis,
- if states cannot be observed at training time →compute a <u>local</u> maximum likelihood with an Expectation-Maximization (EM) method (e.g., **Baum-Welch algorithm**).





Partially Observable MDP (POMDP)

POMDP is an MDP in which the agent cannot directly observe the underlying state. (combines decision making of MDP and non-observability of HMM).

$POMDP \; = < X, A, Z, \delta, r, o >$

- X is a set of states
- A is a set of actions
- Z is a set of observations
- $P(x_0)$ is a probability distribution of the initial state
- $\delta(x,a,x') = P(x'|x,a)$ is a probability distribution over transitions
- $r(x,a)$ is a reward function
- $o(x',a,z') = P(z'|x',a)$ is a probability distribution over observations

Solution concept for POMDP:

⇨ Option 1: map from history of observations to actions - too long!

⇨ Option 2: belief state - probability distribution over the current state

Belief MDP: A Markovian belief state allows a POMDP to be formulated as a MDP where every *belief* is a state. The ***belief states*** $b(x)$ is a probability distribution over the states (but belief states are infinite). The belief MDP is defined by:

- B is a set of belief states
- A is a set of actions
- τ (b,a,b') is a probability distribution over transitions
- $\rho(b,a,b')$ is a reward function

POMDP's **policy** is a mapping from the history of observations (or belief states) to the actions: $\pi: B \to A$

Given a current **belief state** b, the next belief state $b'(x)'$ which represents the probability of being in state s′ after b,a and o, is calculated as:

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

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

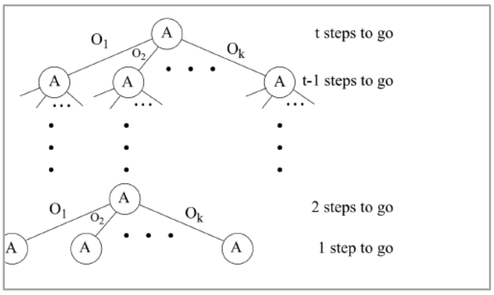
So by the current belief state, the observation probability, and the original transition probability.

- **Transition function:** $\tau(b,a,b') = P(b'|b,a) = \sum_{z\in Z} P(b'|b,a,z)P(z|b,a)$
- **Reward functions:** $\rho(b,a) = \sum_{x\in X} b(x)r(x,a)$
- **Value function:** similar in traditional MDPs, replacing states with belief states

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

→the optimal policy is obtained by optimizing the long-term reward: $\boxed{\pi^* = \arg\max_{\pi} V(b)}$

policy tree:


A
O1
O2
Ok
t steps to go
t-1 steps to go
2 steps to go
1 step to go