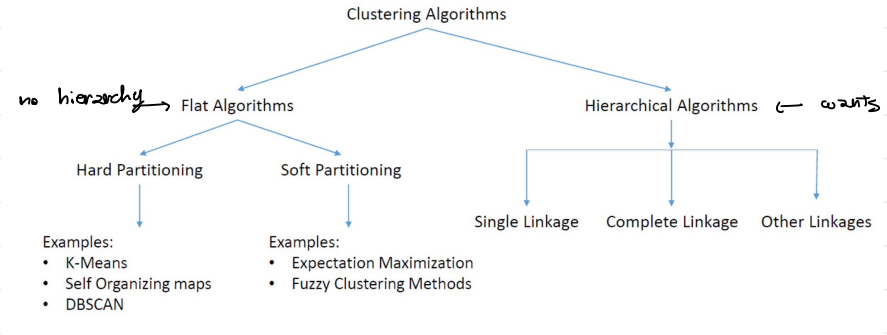
UNSUPERVISED LEARNING (Chapter 6)

Goal: Characterize the distribution $\mathbb{P}(x)$ of the inputs x without the target output y.

Clustering: we want to partition the data into clusters with low intra-cluster dissimilarity and at the same time maximizing the inter-cluster dissimilarity. Many approaches:

- Combinatorial methods (no probabilistic) ← we focus on them
- Mixture modeling: the data is i.i.d. draw from a mixture of distribution but this mixture is hidden
- Mode-seeker: look for the ditribution of x in a non-parametric manner

We can also divide "Clustering Algorithms" as it follows:



→ Object dissimilarity: it's a procedure to define a "distance" between pairs of data points.

So it is a function $d: \text{Dataset} \times \text{Dataset} \longrightarrow \mathbb{R}$ s.t.:

(1) $d(x,y) = d(y,x)$ (reflexivity) (2) Tipically symmetric: $d(x,y) = d(y,x)$ (3) No $\triangle$-inequa.

Examples:

- Weighted difference between attributes: $d(x,y) = \sum_{j=1}^{p} w_j \, d_j(x_j, y_j)$ where w_j's have to be found and, if they are quantitive variables, $d_j(x_j, y_j) = f_j(|x_j - y_j|)$ with f_j non-negative and non decreasing (different distances for different features).
- Categorical variables: often $d(x,y) = c \cdot \mathbb{1}_{\{x \neq y\}}$.

We can also compute the dissimilarity matrix $D \in \mathbb{R}^{N \times N}$ s.t. $D_{ij} = d(x_i, x_j)$.

(If the clustering requires a symmetrix matrix we can choose $\frac{D + D^T}{2}$).

→ How to choose the w_j's? If we have N points $x_1, \dots, x_N$, using weighted difference we have $d(x_i, x_j) = \sum_{K=1}^{p} w_K \, d_K(x_{iK}, x_{jK})$ where x_{iK} is the value

of the K-th feature of x_i. We can find the <u>total average dissimilarity</u> as:

$$\bar{d} = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d(x_i, x_j) = \sum_{K=1}^{P} \omega_K \underbrace{\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d_K(x_{iK}, x_{jK})}_{:=\bar{d}_K}$$

↑ it's the usual average (N^2 pairs)

$\bar{d}_K$ it's the total average dissimilarity for the K-th feature ($\frac{\text{normalization}}{\text{factor}}$)

If we want to treat all attributes equally $\omega_K \sim \frac{1}{\bar{d}_K}$ (approximately proportional) (it is not always appropriate).

Finally we want $\sum_{K=1}^{P} \omega_K = 1$.

If $d_K(x_{iK}, x_{jK}) = (x_{iK} - x_{jK})^2 \Rightarrow \bar{d}_K = 2\,\mathrm{Var}(x_K)$ where $\mathrm{Var}(x_K)$ is the variance of the K^{th}-feature, i.e. $\mathrm{Var}(x_K) = \frac{1}{N}\sum_{i=1}^{N}(x_{iK} - \bar{x}_K)^2$ with $\bar{x}_K = \frac{1}{N}\sum_{i=1}^{N} x_{iK}$; indeed:

$$\bar{d}_K = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}(x_{iK} - x_{jK})^2 = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}(x_{iK} - \bar{x}_K - (x_{jK} - \bar{x}_K))^2 =$$

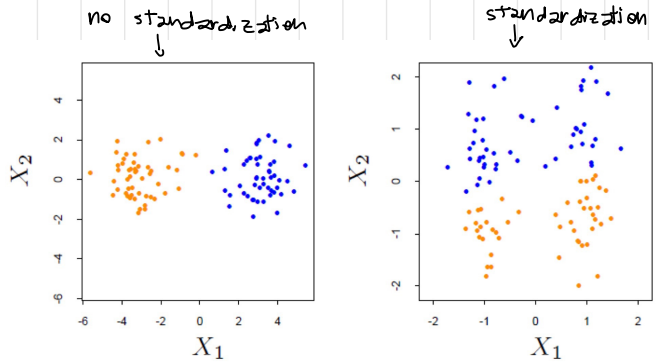
$$= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}\left((x_{iK} - \bar{x}_K)^2 + (x_{jK} - \bar{x}_K)^2 - 2(x_{iK} - \bar{x}_K)(x_{jK} - \bar{x}_K)\right) =$$

$$= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}\underbrace{(x_{iK} - \bar{x}_K)^2}_{\text{not depend on } j} + \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}\underbrace{(x_{jK} - \bar{x}_K)^2}_{\text{not depend on } i} - \frac{2}{N^2}\sum_{j=1}^{N}\sum_{j=1}^{N}(x_{iK} - \bar{x}_K)(x_{jK} - \bar{x}_K) =$$

$$= \frac{1}{N^2}\cdot N\sum_{i=1}^{N}(x_{iK} - \bar{x}_K) + \frac{1}{N^2}N\sum_{j=1}^{N}(x_{jK} - \bar{x}_K)^2 - \frac{2}{N^2}\left(\sum_{i=1}^{N}(x_{iK} - \bar{x}_K)\right)\left(\sum_{j=1}^{N}(x_{jK} - \bar{x}_K)\right)$$

$$= \mathrm{Var}(x_K) + \mathrm{Var}(x_K) - 2\underbrace{\left(\frac{\sum_{i=1}^{N} x_{iK}}{N} - \frac{N\bar{x}_K}{N}\right)}_{=0}\underbrace{\left(\frac{\sum_{j=1}^{N} x_{jK}}{N} - \frac{N\bar{x}_K}{N}\right)}_{=0} = 2\,\mathrm{Var}(x_K)$$

In general, if we want to normalize we are choosing $\omega_K = \frac{1}{\bar{d}_K}$ (in that case $\omega_K = \frac{1}{2\,\mathrm{Var}(x_K)}$). Here an example where doesn't work well the standardization:



Clustering in two groups.

Results completely different, the standardization has obscured the two well separated groups.

→ <u>Clustering algorithms</u>: A cluster algorithm with K clusters and $N >> K$ points should produce:

- a cluster maps $f: \{x_1, \dots, x_N\} \longrightarrow \{1, \dots, K\}$ $f(x_i) =$ "the cluster where is x_i"

and so the t-th cluster is $C_t = \{x_i : f(x_i) = t\}$.

- The cluster centers m_t $t = 1, \dots, K$: m_t approximates all points in the t^{th} cluster.

Note also that by the formula of total dissimilarity :

$$\text{total dissimilarity} = \sum_{i=1}^{N} \sum_{j=1}^{N} d(x_i, x_j) = \longrightarrow \text{2 types of partition of the dataset}$$

$$\text{partition } \sum_{i=1}^{N} = \sum_{t=1}^{K} \sum_{i \in C_K}$$

$$\text{partition } \sum_{j=1}^{N} = \sum_{j \in C_K} + \sum_{j \notin C_K}$$

$$= \sum_{t=1}^{K} \sum_{i \in C_K} \left(\sum_{j \in C_K} d(x_i, x_j) + \sum_{j \notin C_K} d(x_i, x_j) \right) =$$

$$= \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} d(x_i, x_j) + \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \notin C_K} d(x_i, x_j) =$$

$$\textcircled{*} \quad =: \text{Within-cluster dissimilarity} + \text{Between-cluster dissimilarity}$$

If there is simmetry the usual definition is with a factor $\frac{1}{2}$ (we don't want to compute twice the same distance) $\Rightarrow$ Within-cluster diss. $= \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} d(x_i, x_j)$

and Between-cluster dissimilarity $= \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \notin C_K} d(x_i, x_j)$

So we want to find clusters $C = \{ C_t : t = 1, \dots K \}$ so that the within-cluster dissimilarity is minimized or equivantely that between cluster dissimilarity is maximized (the LFH of $\textcircled{*}$ does not depend on the clusters so it is "constant").

Now, fixed K, there are $S(N,K) = \frac{1}{K!} \sum_{t=0}^{K-1} (-1)^t \binom{K}{t} (K-t)^N \approx K^N$

Proof : Suppose the K clusters are ordered. If we find the result for this case, then I divide by $\frac{1}{K!}$ (permutations). Now, I have N data points and I have to assign a cluster $\Rightarrow K \cdot K \cdot \dots \cdot K = K^N$ ways. But there are some cases where a cluster is empty violating the hyp. of clustering. So we compute the case when a cluste is empty : $\binom{K}{1}$ to choose the empty cluster and $(K-1) \cdot \dots \cdot (K-1) = (K-1)^N$ ways to assign the $K-1$ cluster remaining to the N points. But in this case we have counted twice the cases where 2 or more clusters are empty, so by inclusion-exclusion principle :

$$K^N - \binom{K}{1} (K-1)^N + \binom{K}{2} (K-2)^N - \dots + (-1)^{K-1} \binom{K}{K-1} 1^N =$$

$$= \sum_{t=0}^{K-1} (-1)^t \binom{K}{t} (K-t)^N$$

$S(N,K)$ grows rapidly $\Rightarrow$ no chance to find a global optimum. We try to find a local optimum.

→ **K-means clustering**: Characteristics:

① The dissimilarity measure is $d(x,y) = \|x-y\|_2^2$

② For each C_K the center m_K is: $m_K \overset{\text{def.}}{=} \underset{m}{\operatorname{argmin}} \sum_{x_i \in C_K} d(x_i, m) = \underset{m}{\operatorname{argmin}} \underbrace{\sum_{x_i \in C_K} \|x_i - m\|_2^2}_{=:f}$

To solve the minimum: $\left(\nabla_m f\right)_j = -2 \sum_{x_i \in C_K} (x_{ij} - m_j) = 0 \quad \forall j$

$\Rightarrow \sum_{x_i \in C_K} x_{ij} = N_K m_j \Rightarrow m_j = \frac{1}{N_K} \sum_{x_i \in C_K} x_{ij}$.

So $\boxed{m_K = \frac{1}{N_K} \sum_{i \in C_K} x_i}$.

③ Within-cluster dissimilarity is:

$$W(C) = \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} d(x_i, x_j) = \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} \|x_i - x_j\|_2^2 =$$

$$= \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} \|(x_i - m_K) - (x_j - m_K)\|_2^2 =$$

$$= \frac{1}{2} \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} \|x_i - m_K\|_2^2 + \|x_j - m_K\|^2 - 2 \langle x_i - m_K, x_j - m_K \rangle =$$

$$= \frac{1}{2} \sum_{t=1}^{K} N_K \sum_{i \in C_K} \|x_i - m_K\|_2^2 + \frac{1}{2} \sum_{t=1}^{K} N_K \sum_{j \in C_K} \|x_j - m_K\|^2 - \sum_{t=1}^{K} \sum_{i \in C_K} \sum_{j \in C_K} \langle x_i - m_K, x_j - m_K \rangle$$

$$= \cancel{2} \cdot \frac{1}{\cancel{2}} \sum_{t=1}^{K} N_K \sum_{i \in C_K} \|x_i - m_K\|_2^2 - \sum_{t=1}^{K} \Big\langle \underbrace{\sum_{i \in C_K} x_i - N_K m_K}_{=0}, \underbrace{\sum_{j \in C_K} x_j - N_K m_K}_{=0} \Big\rangle$$

$$= \sum_{t=1}^{K} N_K \sum_{i \in C_K} \|x_i - m_K\|_2^2$$

So the K-means clustering problem is:

$$\min_{C} \min_{\{m_t\}_{t=1}^K} \sum_{t=1}^{K} \sum_{i \in C_t} \|x_i - m_t\|_2^2$$

It is not possible to solve that so:

Step 1 We fix the number of clusters K

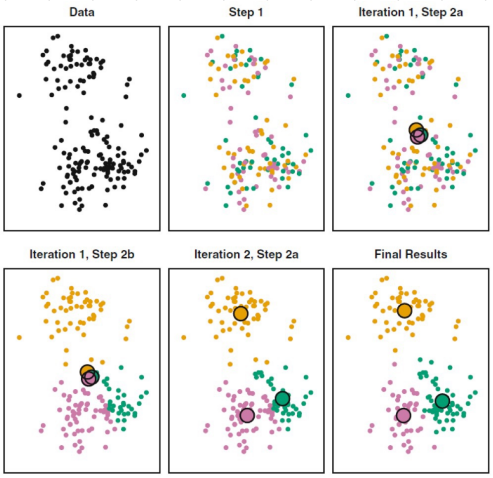
Step 2 Fix clusters $\{C_t\}_{t=1}^K$ randomly and compute $m_t = \frac{1}{N} \sum_{i \in C_t} x_i$. Now,

x_i is associated to the closest m_t, i.e. the new cluster

$$C_t^{\text{new}} = \{ x_i \text{ s.t. } t = \underset{s}{\operatorname{argmin}} \|x_i - m_s\|_2^2 \} \quad \text{(Voronoi Cells)} \qquad \forall t = 1, \dots, K.$$

Step 3: compute the new m_t^{new} centers and iterate. The algorithms converges to a local minimum.

Example of the algorithm:



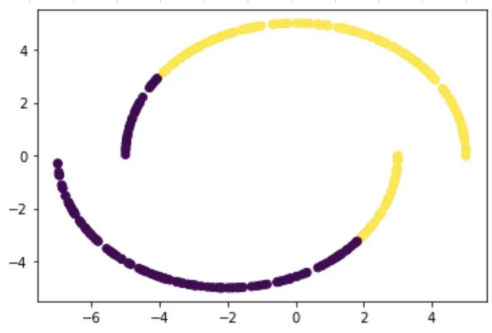
Starting with different values leads to different results

Vector quantization: consider a greyscale image of 1024×1024 pixels. It can be represented by a matrix $X \in \mathbb{R}^{1024 \times 1024}$ where each entry is a greyscale value from 0 to 255 (1 byte to storage $\Rightarrow$ the entire image is 1 megabyte).

Now, suppose to divide the matrix by 2×2 matrices: $\begin{pmatrix} X_{2i-1,2j-1} & X_{2i-1,2j} \\ X_{2i,2j-1} & X_{2i,2j} \end{pmatrix}$ obtaining a collection $\mathcal{X} = \{x_{ij}\}_{i,j=1}^{512}$ where $x_{ij} \in \mathbb{R}^4$ (the 2×2 matrices are seen as vectors).

$\mathcal{X}$ is our data points, we run the K-means clustering, obtaining the final centers $\{m_t\}_{t=1}^{K}$ for each class j then we substitute each 2×2 block in the image s.t. its associated vector in in cluster t with the 2×2 matrices obtained by m_t ($\text{mat}(m_t)$). This is a compression algorithm.

Problem of K-means clustering:
- clusters are non spherical
- clusters have different size
- outliers
- Clusters are not linearly separable ⟶
- Clusters have overlap



↓

Kernel K-means clustering: replace the eucledian distance by a Kernel, for example:

- Polynomial: $K^{(p)}(x_i, x_j) = (1 + x_i^T x_j)$, $p \in \mathbb{N}$
- Gaussian: $K^{(g)}(x_i, x_j) = \exp\left(-\|x_i - x_j\|_2^2 / 2\sigma^2\right)$, $\sigma \in \mathbb{R}$

For Mercer's theorem $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ for some $\phi: \mathbb{R}^p \to \mathbb{R}^d$ with tipically $d \gg p$.

(I use higher dimension to have linear separable data).

Note that to compute distances:

$$\|\phi(x_i) - \phi(x_j)\|_2^2 = \left(\phi(x_i) - \phi(x_j)\right)^T \left(\phi(x_i) - \phi(x_j)\right) =$$

$$= \phi(x_i)^T \phi(x_i) + \phi(x_j)^T \phi(x_j) - 2\,\phi(x_i)^T \phi(x_j) =$$

$$= K(x_i, x_i) + K(x_j, x_j) - 2K(x_i, x_j)$$

we don't need to find ϕ!

Also, the clusters centers are $m_k = \frac{1}{N_k} \sum_{i \in C_k} \phi(x_i)$ but to assign points to clusters we don't need to compute them, in fact:

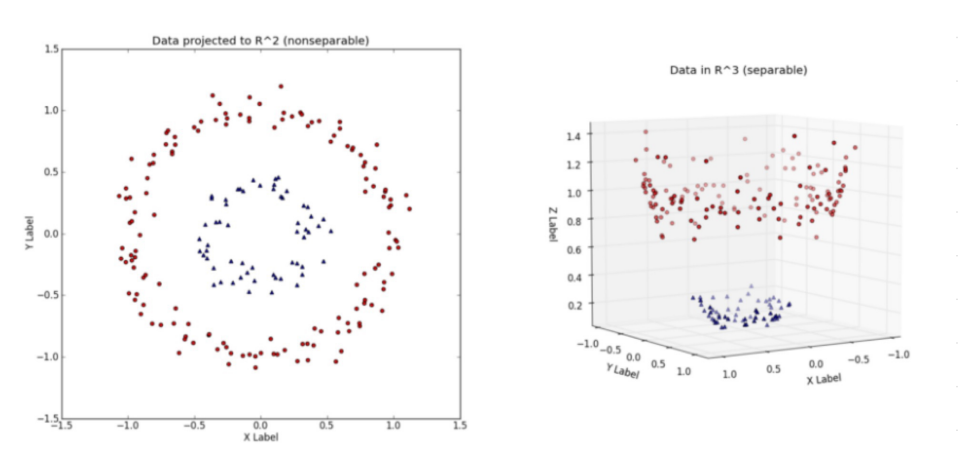
$$C_t = \{x_i : t = \underset{s}{\operatorname{argmin}} \|\phi(x_i) - m_s\|_2^2\} \quad \text{but}$$

$$\|\phi(x_i) - m_s\|_2^2 = \left(\phi(x_i) - m_s\right)^T \left(\phi(x_i) - m_s\right) =$$

$$= \phi(x_i)^T \phi(x_i) - 2\,\phi(x_i)^T m_s + m_s^T m_s =$$

$$= K(x_i, x_i) - \frac{2}{N_k} \sum_{j \in C_k} K(x_i, x_j) + \frac{1}{N_k^2} \sum_{j, l \in C_k} K(x_j, x_i), \text{ we}$$

don't need ϕ $\Rightarrow$ we don't have to work in higher dimension.



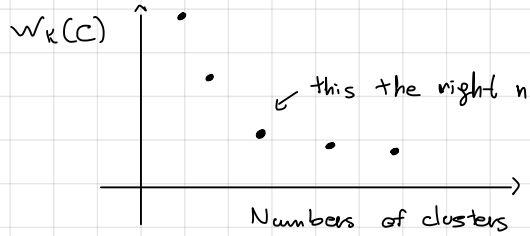
→ <u>K-medoids clustering</u>: K-means is sensitive to outliers because outliers produce large distances and so influence the centers too much. We can choose a different distance. Otherwise we choose as cluster center (medoid) $m_t := x_i$ s.t. $i = \underset{x_l \in C_t}{\operatorname{argmin}} \sum_{x_j \in C_t} d(x_j, x_l)$

so it's the datapoint (in K-mean is not a point of the dataset a priori) minimizing the sum of pairwise dissimilarities. We update the cluster as in K-mean:

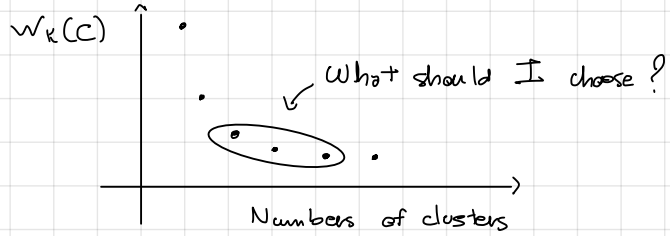
$$C_t = \{ x_i \text{ s.t. } t = \underset{s}{\operatorname{argmin}}\, d(x_i, m_s) \}$$

K-medoids is more robust than K-means but also more computationally complex

→ Choiche of the number of clusters : sometimes it's obvious using the context of the problem, sometimes not: let $W_K(C)$ be the within-cluster dissimilarity with K clusters. We should have $W_K(C) > W_{K+1}(C)\ \forall K$, so if K^* is the "true" number of clusters, we should have $W_{K^*+1}(C)$ close to $W_{K^*}(C) \Rightarrow$ we can plot $W_K(C)$ as a function of K and searching for the bend of an elbow : ← "piega del gomito"



Sometimes it's difficult to interpret the plot:



So, there is another method: fix K the number of clusters. Suppose $\tilde{W}_K(C)$ is the within-cluster distances of a data uniformly distributed over a suitable region of the feature space (usually a p-dimensional rectangle). Define:

$$\boxed{Gap(K) = \log\left(W_K(C)\right) - \log\left(\tilde{W}_K(C)\right)}$$

So take $K^* := \underset{K}{\operatorname{argmax}}\, Gap(K)$, because it means that, with respect to a "benchmark" data, we have had the largest decrease of the within-cluster distance.

→ Hierarchical clustering : Characteristics:

- There are levels of clusters ;
- Each clusters at the lowest level contains a single observation ;
- Clusters at a given level are formed by merging ← fondere clusters at the next lower level ;
- At the highest level there is just one cluster containing all the data.

- The algorithm can be agglomerative (bottom-up) or divise (top-down). We focus on agglomerative.

We define 3 possible distance between clusters G and H:

- Single linkage: $d_{SL}(G,H) := \min_{x_i \in G, x_j \in H} d(x_i, x_j)$ (In terms of dissimilarity $s_{SL}(G,H) = \min_{\substack{x_i \in G \\ x_j \in H}} s_{ij}$)
- Complete linkage: $d_{CL}(GH) = \max_{x_i \in G, x_j \in H} d(x_i, x_j)$ (In terms of diss.: $s_{CL}(G,H) = \min_{\substack{x_i \in G \\ x_j \in H}} s_{ij}$)
- Group average: $d_{GA}(G,H) = \frac{1}{N_H N_G} \sum_{x_i \in G, x_j \in H} d(x_i, x_j)$

Algorithm:

Step 1: Begin with N observations and a measure (such as the Eucledian distance) of all the $\binom{N}{2} = \frac{N(N-1)}{2}$ pairwise dissimilarities.

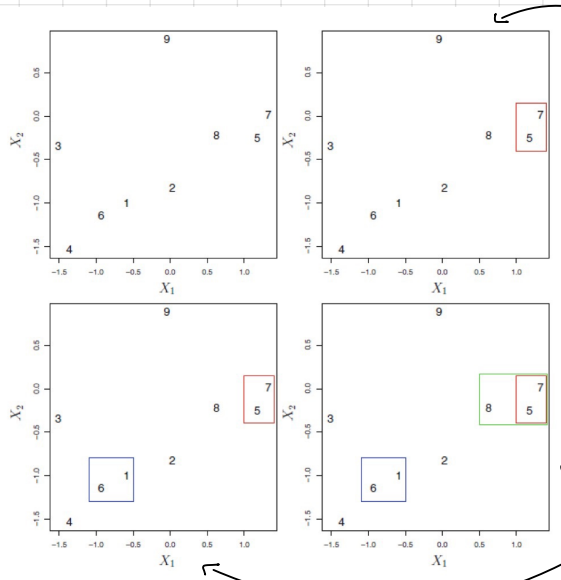
Step 2: For $i = N, N-1, \dots, 2$:

A. Look at all pairwise dissimilarities among the i clusters and identify the pair of clusters that are least dissimilar. Merge these two clusters together. The dissimilarity between these two clusters indicates the height in the dendogram (see later the example) at which the merge should be placed

(if there are two couples with same distance we choose randomly ← CHAT GPT answer)

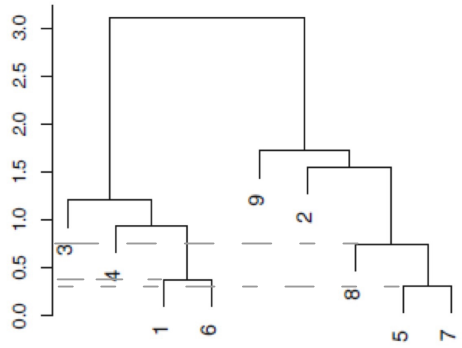
B. Compute the new pairwise dissimilarities among the $i-1$ remaining clusters

Example:

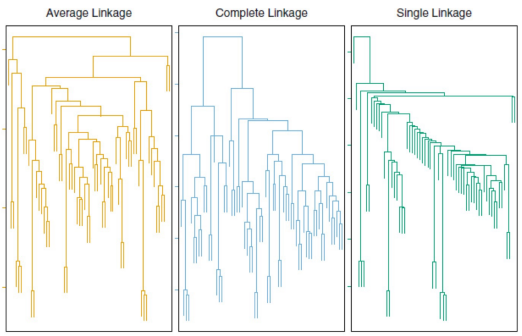


- Step 1/2.A: we compute all the distance between points (eucledian distance in this case) and we find that the points 7, 5 are the closest.
- Step 2.B: now we have the clusters: $\{1, 2, 3, 4, 6, 8, 9, \{5,7\}\}$, we compute the distance between these new clusters (now I have a cluster with 2 points and I will use for example Single linkage to compute for example $d(1, \{5,7\})$)
- We find that 1, 6 are the closest, same story
- We find that 8, $\{7,5\}$ are the closest

So the dendogram will be:



There are differences using different clustering distances.

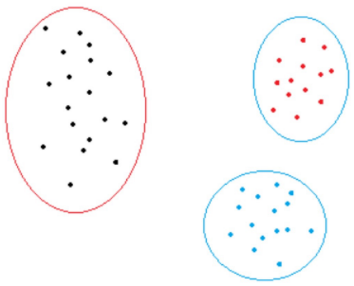


Modelling choices: what dissimilarity measure should be used, what linkage should be used, where should we cut the dendogram in order to obtain the cluster.

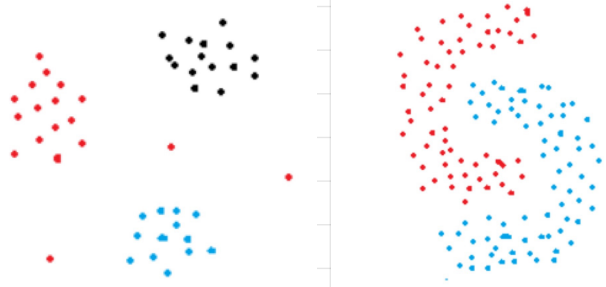
Pairing clustering methods: we can use K-means and hierarchical clustering together:

- K-means to cluster the data into a large number of groups
- Then use single linkage hierarchical clustering to agglomerate groups.

→ DBSCAN (Density-based spatial clustering of Applications with Noise): density based clustering, works better with arbitrary shaped clusters or outliers.



K-mean works better



DBSCAN works better

Main intuition: a point belongs to a cluster if it is close to many points from that cluster.

↳ two parameters → ε: the distance that specifies the neighborhoods. 2 points are in the same neighborhood if $d(x,y) \leq \varepsilon$

→ MinPts: minimum number of data points to define a cluster

↳ 3 different types of point:

Core point: $x \in$ Dataset

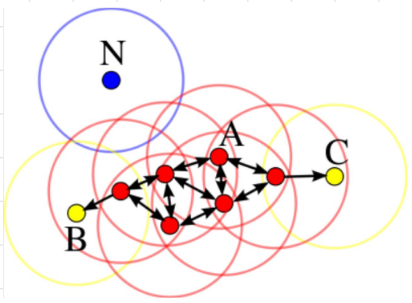
s.t. $|\overline{B(x,\varepsilon)} \cap \text{Dataset}| \geq \text{MinPts}$

Border point: $x \in$ Dataset

s.t. $|B(x,\varepsilon) \cap \{\text{Corepoints}\}| \geq 1$

but $|\overline{B(x,\varepsilon)} \cap \text{Dataset}| < \text{MinPts}$

Outlier: not a core point and and not a border point



MinPts = 4

Red points are core points, yellow points are border points and blue points are outliers.

Algorithm: Step 1: start with a random point;

Step 2: If there at least MinPts number of points in the neighborhood, the point is marked as core point and a cluster formation starts. If not, the point is marked as noise;

Step 3: Once a cluster formation starts (let's say cluster A), all the points within the neighborhood of initial point become a part of cluster A. If these new points are also core points, the points that are in the neighborhood of them are also added to cluster A

Step 4: Next step is to randomly choose another point among the points that have not been visited in the previous steps. The procedure is finished when all points are visited

Remarks:
- it finds region with high-density
- A cluster includes core points and all the border points of these core points
- The noise points are not in any clustering
- If clusters are very different in terms of in-cluster densities, DBSCAN doesn't work well. (for example in the previous picture).

Principal component decomposition (PCA): the idea is that we have a large set of correlated variables, principal components allows us to summarize this set with a smaller number of representative variables that collectively explain most of the variability of the original set.

Suppose we have a d-dimensional random vector X. A dataset is a set of n values assumed by this random vector, given by a matrix $X \in \mathbb{R}^{d \times n}$ where

the columns are the observations and the rows the values of a fixed features.

So $x = \begin{pmatrix} x_1 \\ | \\ x_d \end{pmatrix}$, x_i is the r.v. for feature i.

PCA constructs d random variables $p_i = \sum_{j=1}^{d} \omega_{ij} x_j$ $\forall i = 1, \dots, d$ linear combinations of x_j such that:

- $\mathbb{E}[p_i p_j] = 0 \quad \forall i \neq j$ (orthogonal)
- p_i's are ordered such that: p_1 explains the largest percentage of the total variance and p_i explains the largest percentage of the total variance that has not already been explained by $p_1, \dots, p_{i-1}$.

Now the first step is to standardize the features r.v. x_i: $\frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{Var}(x_i)}}$ (ensure they have comparable measurement units).

In practice we compute the mean and the variation across observations and we standardize.

The key object of interest will be the variance-covariance matrix of x:

$$\Sigma := \mathbb{E}\left[(x - \mathbb{E}[x])(x - \mathbb{E}[x])^T\right] \in \mathbb{R}^{d \times d}$$

↖ impropriate scriptove but there is only one sense

Σ is symmetric and positive semidefinite

The empirical counterpart can be estimated as: $Q = \frac{1}{n-1} X X^T$ where X is assumed standardized.

→ Engine decomposition: by spectral theorem we know $\exists\, \Delta$ diagonal and Γ orthogonal such that $\Sigma = \Gamma \Delta \Gamma^T$; in particular $\Delta = \text{diag}(\lambda_1, \dots, \lambda_d)$ $\lambda_1 \geq \dots \geq \lambda_d$ engine values of Σ and $\Gamma = (\gamma_1, \dots, \gamma_d)$ engine vectors of Σ s.t. $\gamma_1^T \gamma_1 = 1$. Also, Σ is positive semi-definite $\Rightarrow \lambda_i \geq 0 \quad \forall i = 1, \dots, d$. The principal components of x are denoted by $p \in \mathbb{R}^d$ given by:

$$\boxed{p = \Gamma^T x}$$

Since $\mathbb{E}[x] = 0$ (by assumption) $\Rightarrow \mathbb{E}[p] = 0$ and

$$\text{Cov}(p) = \mathbb{E}\left[(p - \mathbb{E}[p])(p - \mathbb{E}[p])^T\right] = \mathbb{E}\left[(\Gamma^T x - \mathbb{E}[\Gamma^T x])(\Gamma^T x - \mathbb{E}[\Gamma^T x])^T\right] =$$

$$= \Gamma^T \mathbb{E}\left[(x - \mathbb{E}(x])(x - \mathbb{E}[x])\right] \Gamma = \Gamma^T \Sigma \Gamma = \Gamma^T \Gamma \Delta \Gamma^T \Gamma = \Delta.$$

In fact, we have constructed d p_i r.v. for $i = 1, \dots, d$, linear combination of $x_1, \dots, x_d$,

orthogonal since $\mathbb{E}[p_i p_j] = \mathrm{Cov}(p_i, p_j) = \Delta_{ij} = 0 \ \forall i \neq j$. And $\mathrm{Var}(p_i) = \mathrm{Cov}(p_i, p_i) = \lambda_i$.

($\mathbb{E}[p_i] = \mathbb{E}[p_j] = 0$)

The matrix Γ^T is called matrix of factor loading.

So, in practice, we have X and Q, obtaining $P = \Gamma^T X$ with $P \in \mathbb{R}^{d \times n}$ and Γ, Δ obtained by Q.

→ How do we reduce the dimension? We can measure the ability of the first few principal components to explain the total variance:

$$\sum_{i=1}^{d} \mathrm{Var}(p_i) = \sum_{i=1}^{d} \lambda_i = \mathrm{tr}(\Sigma) = \sum_{i=1}^{d} \mathrm{Var}(x_i)$$

(the sum of eigenvalues is the trace)

($\mathrm{Cov}(p) = \Delta \Rightarrow \mathrm{Var}(p_i)$ is Δ_{ii}; $\mathrm{Cov}(x) = \Sigma \Rightarrow \mathrm{Var}(x_i)$ is Σ_{ii})

So it makes sense to define the ratio:

$$\frac{\sum_{i=1}^{K} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \qquad (\text{remember } \lambda_1 \geq \dots \geq \lambda_d)$$

that is the percentage of total variance explained by the first K principal components.

So we choose a threshold and keep the first K principal component (= first K lines of P) such that the ratio is $\geq$ threshold.

→ The maximum variance interpretation: • let $p_1 = \gamma_1^T x$ be the first principal component.

γ_1 solves this optimization problem: $\max_a \mathrm{Var}(a^T x)$ subject to $a^T a = 1$ (← normalization):

$$\mathrm{Var}(a^T x) = \mathrm{Cov}\left(\sum_{i=1}^{d} a_i x_i, \sum_{j=1}^{d} a_j x_j\right) = \sum_{i,j=1}^{d} a_i a_j \mathrm{Cov}(x_i, x_j) = a^T \Sigma a$$

So we can write the lagrangian

$$L(a, \lambda) = a^T \Sigma a - \lambda(a^T a - 1)$$

$$\nabla_a L = 2\Sigma a - 2\lambda a = 0 \Rightarrow \Sigma a = \lambda a$$ (a has to be an eigenvector of Σ)

So $\mathrm{Var}(a^T x) = a^T \Sigma a = \lambda a^T a = \lambda$, so to maximize it we have to choose $\lambda = \lambda_1$ (the greatest eigenvalue) $\Rightarrow a = \gamma_1$ (the corresponding eigenvector).

• More generally, let $p_i = \gamma_i^T x$ be the i^{th} principal component for $i = 1, \dots, d$, then γ_i solves: $\max_a \mathrm{Var}(a^T x)$ subject to $a^T a = 1$ (← normalization) and $a^T \gamma_j = 0 \ \forall j = 1, \dots, i-1$ (← orthogonality):

as above we want to maximize $a^T \Sigma a$:

$$L(a, \lambda, \beta) = a^T \Sigma a - \lambda (a^T a - 1) - \sum_{j=1}^{i-1} \beta_j (a^T \gamma_j)$$

$$\nabla_a L = 2\Sigma a - 2\lambda a - \sum_{j=1}^{i-1} \beta_j \gamma_j = 0 \Rightarrow 2(\Sigma a - \lambda a) = \sum_{j=1}^{i-1} \beta_j \gamma_j$$

$$\Rightarrow \Sigma a - \lambda a = \sum_{j=1}^{i-1} \frac{\beta_j}{2} \gamma_j \Rightarrow \Sigma a = \lambda a + \sum_{j=1}^{i-1} \frac{\beta_j}{2} \gamma_j \quad \text{but let's multiply by } \gamma_m^T$$

$$(m \in \{1, \dots, i-1\}) \Rightarrow \gamma_m^T (\Sigma a) = \lambda \underbrace{\gamma_m^T a}_{=0} + \sum_{j=1} \frac{\beta_j}{2} \underbrace{\gamma_m^T \gamma_j}_{=\delta_{mj}} \Rightarrow$$

$$\Rightarrow (\gamma_m^T \Sigma) a = \frac{\beta_m}{2} \underset{\Sigma \text{ symmetric}}{\Rightarrow} (\Sigma \gamma_m)^T a = \frac{\beta_m}{2} \Rightarrow \lambda_m \gamma_m^T a = \frac{\beta_m}{2} \Rightarrow \beta_m = 0.$$

So, $\beta_j = 0 \ \forall j \in \{1, \dots, i-1\} \Rightarrow \Sigma a = \lambda a \Rightarrow a$ is an eigenvector of Σ (can't be γ_j for $j \in \{0, \dots, i-1\}$ because $a^T \gamma_j = 0$) and as above $\mathrm{Var}(a^T x) = \lambda \Rightarrow$ to maximize $\lambda = \lambda_i$ and so $a = \gamma_i$.

→ <u>The projection interpretation</u>: Consider the following optimization problem:

$$\min_{\tilde{\Gamma}, \tilde{P}} \| X - \tilde{\Gamma} \tilde{P} \|_F^2 \quad \text{with } \tilde{\Gamma} \in \mathbb{R}^{d \times K} \text{ and } \tilde{P} \in \mathbb{R}^{K \times n} \quad (*)$$

where $\|A\|_F = \sqrt{\sum_{ij} a_{ij}^2}$ is the Frobenius norm.

If we show (← NON L'HO DIMOSTRATO (SERVE?)) that a solution of the problem is $\tilde{\Gamma}$:= first K columns of Γ and $\tilde{P}$ = first K rows of P, we would have that the PCA minimizes the error arising projecting the data (rows of X!) (← for a fixed feature) onto the K dimensional subspace spanned by the first K principal components (it's a row-spanned subspace: the first K rows of P!)

<u>Remark</u>: $\tilde{\Gamma}$ and $\tilde{P}$ solutions of $(*)$ are not unique; we could impose in addition $\tilde{\Gamma}^T \tilde{\Gamma} = Id_K$ but even in this case $\tilde{\Gamma}$ and $\tilde{P}$ are unique up to a rotation indeed:

if $U \in \mathbb{R}^{K \times K}$ is a rotation matrix, consider $\tilde{\Gamma} U$ and $U^T \tilde{P} \Rightarrow$

$$(\tilde{\Gamma} U)^T \tilde{\Gamma} U = U^T \tilde{\Gamma}^T \tilde{\Gamma} U = U^T Id\, U = U^T U = Id \quad \text{and}$$

$$\| X - (\tilde{\Gamma} U)(U^T \tilde{P}) \|_F = \| X - \tilde{\Gamma} Id \tilde{P} \|_F = \| X - \tilde{\Gamma} \tilde{P} \|_F.$$

→ <u>Singular value decomposition</u>: an alternative way to compute $\tilde{P}$, is using the SVD decomposition: $X = U D V^T$ $U \in \mathbb{R}^{d \times d}$, $V \in \mathbb{R}^{n \times n}$ orthogonals, $X \in \mathbb{R}^{d \times n}$ and $D \in \mathbb{R}^{n \times n}$

diagonal ($D_{11} \geq \dots \geq D_{nn}$). Note that in the empirical way Γ and Δ are obtained using $Q = \frac{1}{n-1} X X^T$, i.e. $Q = \Gamma \Delta \Gamma^T$. But

$$Q = \frac{1}{n-1} X X^T = \frac{1}{n-1} U D V^T V D^T U^T = \frac{1}{n-1} U D^2 U^T = U \left(\frac{1}{\sqrt{n-1}} D\right)^2 U^T$$

so the engine vectors of Q are given by U and so, since Γ was the matrix gave the engine values for Σ, so $Q = \Gamma \Rightarrow P = U^T X$. (The engine values are $\frac{1}{n-1} \lambda_i^2$ where $D = (\lambda_1, \dots, \lambda_n)$)

→ Application to LSA: we can apply PCA to the variance-covariance matrix of X (embedding of corpus of documents). Weaknesses: the engine-vectors represent topics and it is hard to interprete what the topics are; engine vectors can have negative components: how can a word contribute negatively to a topic?

Alternative methods: Probabilistic LSA or Latent Dirichlet Allocation.

→ Unbalanced Panels: a huge limitation of PCA is that it needs a balanced panels (each variable has observations for the same set of time periods) to estimate the variance-covariance matrix. How can we fix this problem? Remember that PCA finds $\tilde{\Gamma} \in \mathbb{R}^{d \times K}$ and $\tilde{P} \in \mathbb{R}^{K \times n}$ s.t. are solutions of $\min_{\tilde{\Gamma}, \tilde{P}} \| X - \Gamma P \|_F$ that is equivalent to find $M \in \mathbb{R}^{d \times n}$ s.t. is solution of $\min_M \| X - M \|_F^2$ $\text{rank}(M) \leq K$. So we extend the optimization problem as follows:

$$\min_{\tilde{X}, M} \| \tilde{X} - M \|_F^2 \quad \text{s.t. } \text{rank}(M) \leq K \text{ and } \tilde{X}_{ij} = X_{ij} \text{ if } X_{ij} \text{ is not missing}$$

(the unbalanced panels)

we can solve this problem using "iterative SVD":

1. Let's start with $\tilde{X} = X$ (original data) and $M = \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & 0 & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$.
2. At each step replace the missing values of $\tilde{X}$ with the corresponding values from M
3. Apply SVD decomposition to $\tilde{X} = U \Sigma V^T$ and truncate U, Σ, V to rank K. Update M as the rank-K approximation obtained from the truncated SVD.
4. Repeat 2-3 until convergence

↳ Non linear extension of PCA: Kernel trick : apply PCA to the matrix

$\tilde{X} = [\phi(x_1), \dots, \phi(x_n)]$, no need to compute $\tilde{X}$ only $\tilde{X}\tilde{X}^T$.

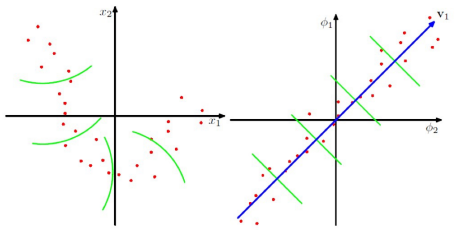
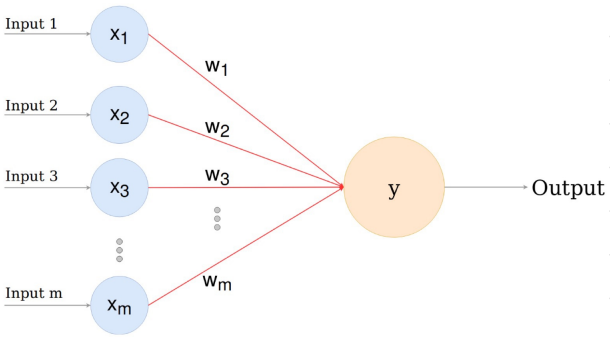


Figure 12.16 from Bishop: Schematic illustration of kernel PCA. A data set in the original data space (left-hand plot) is projected by a nonlinear transformation $\phi(\mathbf{x})$ into a feature space (right-hand plot). By performing PCA in the feature space, we obtain the principal components, of which the first is shown in blue and is denoted by the vector $\mathbf{v}_1$. The green lines in feature space indicate the linear projections onto the first principal component, which correspond to nonlinear projections in the original data space. Note that in general it is not possible to represent the nonlinear principal component by a vector in x space.

NEURAL NETWORKS (Chapter 7)

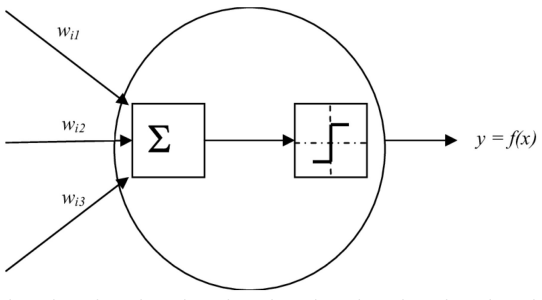
Feed-forward networks : also called multilayer perceptrons (MLP). Is supervised ML, use hierarchical layers of abstraction to represent high-dimensional non linear prediction.

↳ Perceptron : is a type of neural network unit. It is of the form



- Blue nodes: input nodes (m input features)
  Blue nodes form the input layer
- Red arrows = synapses. Each synaps has a weight attached to it
  They form the synapse layer
- Orange node = output node

The output node can be seen as a neuron that has:

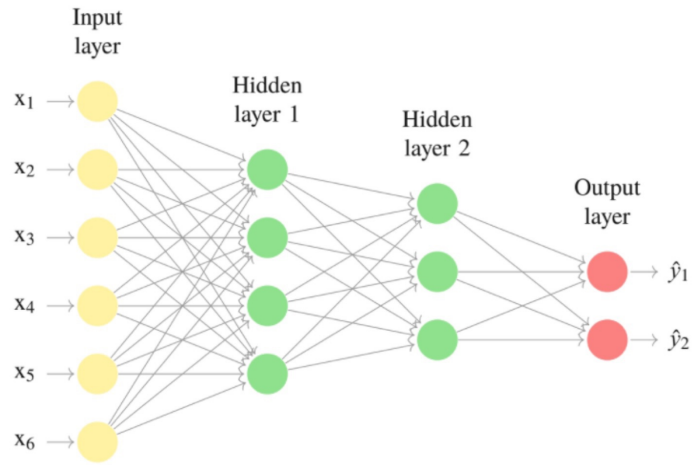


- a set of weighted input w_i (synapses)
- an adder (Σ) that sums the inputs signals
- an activation function ($\sqcap$) that decides wether the neuron "fires" or not
- Usually we have that Σ do a weighted sum

and we will have $f(x) = f\left(\sum_{i=1}^{m} w_i x_i\right)$ (activation function; there will be a bias term (see later)).

↳ Feedforward architectures : we can merge multiple neuron together obtaining feedforward

architectures, as:



- 6 features (Input layer)
- 2 hidden layers
- 2 Outputs (Output layer)

We say that is a "shallow network" when there is only one hidden layer.

When there are more hidden layers we talk about "Deep network".

General form: $\boxed{Y = F_{W,b}(X) + \varepsilon}$

where $F_{W,b}$ is a deep neural network with L layers (is not included the input layer) and ε is an i.i.d. error. In particular:

- $F_{W,b}(X) = f^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}}(X)$ where $f^{(i)}_{W^{(i)},b^{(i)}}$ denotes the transformation function for the i-th layer, including output layer L;
- $W = (W^{(1)}, \dots, W^{(L)})$ collection of "weight matrices". $W^{(l)} \in \mathbb{R}^{m \times n}$ ← depends on l weight matrix for layer l with $[W^{(l)}]_{ij}$ = weight from j-th node in layer $l-1$ to the i-th node in layer l (layer 0 = input layer)
- $b = (b^{(1)}, \dots, b^{(L)})$ a collection of bias vectors, $b^{(l)} \in \mathbb{R}^m$.

Let's define $f^{(l)}_{W^{(l)},b^{(l)}} : \mathbb{R}^n \to \mathbb{R}^m$ as $f(v) = W^{(l)}\sigma^{(l-1)}(v) + b^{(l)}$ where $v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$ ← semiaffine function in v

$\sigma^{(l-1)} = \begin{pmatrix} \sigma_1^{(l-1)}(v_1) \\ \vdots \\ \sigma_n^{(l-1)}(v_n) \end{pmatrix}$ with $\sigma_i^{(l-1)}$ the activation function of the i-th node of layer $l-1$.

* if there is an activation function in the output node we have to apply also it to $F_{W,b}$ (ex. sheet)

All $\sigma_i^{(l)}$ are of the form $\sigma : \mathbb{R} \to B \subseteq \mathbb{R}$ with B bounded, continuous, monotonically increasing function;

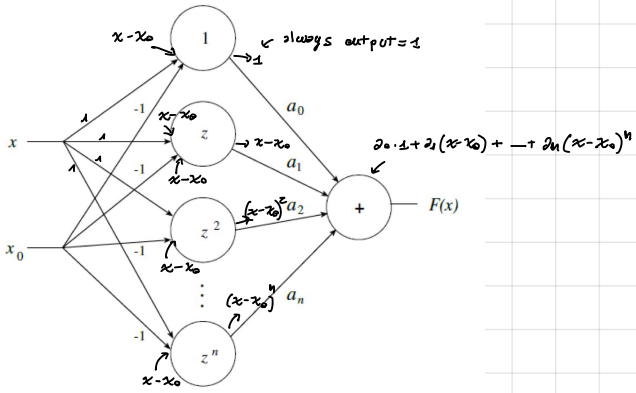
↓

Universal approximator property: multilayer feedforward networks can approximate any function to any desired degree of accuracy, provided sufficiently many hidden units are available.

↓

example for Taylor series, approximating F around x_0:

$F(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)^2 + \_ + a_n(x-x_0)^n + \varepsilon$, $a_0, \_, a_n$ depends on F.



→ Activation function : are attached to each neuron in the network, they determine if the neuron "fires" or not (without activation function we would be able to compute only linear functions). Also help to normalize output between 0 and 1 or -1 and 1.

→ Examples :

(1) Step activation function : also called Heaviside function. Usually used in Perceptron. If the input value is above of below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer. It does not allow multi-value outputs (for example, it cannot support classifing inputs)

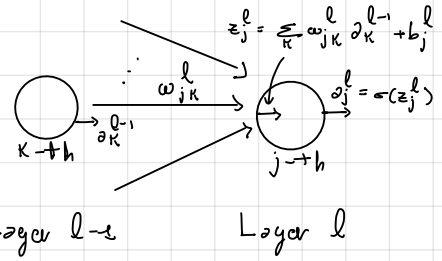
(2) Linear activation : If all the activation function are linear, $F_{W,b}$ is just a linear regression. Hidden layers are redundant: for any such network one can always find an equivalent network without hidden units. (composition of linear function is linear)

(3) Sigmaid function : $S(x) = \frac{e^x}{e^x + 1}$, smooth gradient, preventing "jumps" in output values. Output is between 0 and 1, so it is normalized.

(Using sigmaid function we are back to logistic regression (classification) but training is different (logistic regression uses MLE, we will see how estimation is done for neural networks).

→ Input / Output of a neuron : Recall $W = (W^{(1)}, \_, W^{(L)})$ with $W^{(l)} \in \mathbb{R}^{m \times n}$ and biases are in $b = (b^1, \_, b^n)$; w^l_{jk} element of $W^{(l)}$ representing the weight given to the output of neuron k (from layer $l-1$), when computing the input of neuron j at layer l. If o_k^{l-1} is the output of neuron k at layer $l-1$ and by z_j^l the input of neuron

j at layer l: $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$. The output of neuron j at byer l is then $a_j^l = \sigma(z_j^l)$.



→ Training: pair the input with expected output, until a sufficient close match has been found. Goal: learn the optimal weights and biases (this is conditional on some modelling choices as the number of layers, number of neuron in each layer and activation function, we will get back to these choices after, for now assume they are made):

- start with input-outputs $D = \{ Y^i, X^i \}_{i=1}^N$ and a loss function for predictor $\hat{Y}$, $L(Y, \hat{Y})$ (for example for regression with Gaussian model $L(Y, \hat{Y}) = \|Y - \hat{Y}\|^2$, for binary classification $L(Y, \hat{Y}) = -Y \log \hat{Y}$)
- Optimization problem: $\min_{W,b} J(W,b) + \lambda \phi(W,b)$ with $J(W,b) = \frac{1}{N} \sum_{i=1}^N L(Y^i, \hat{Y}(X^i))$

and λ is a global regularization parameter and $\phi(W,b)$ is a regularization penalty.

→ Gradient descent: generic optimization algorithm used to find optimal solutions. The idea is to tweak ("modificare leggermente") parameters iteratively in order to minimize a cost function:

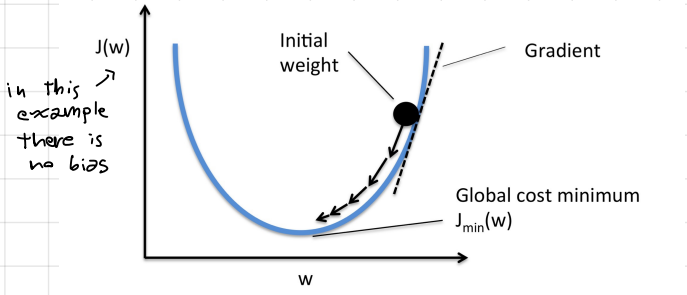
- Use the model with the current set of parameters to make prediction
- Compare prediction to the real expected outcomes
- Using error to update parameters (in our case W and b are updated incrementally).

Formally we have:

- Start with initial W_0, b_0;
- Compute the cost gradient g_k:

$$g_k := \frac{1}{N} \sum_{i=1}^N \nabla_{W,b} \left(Y^i, \hat{Y}^k(X^i) \right)$$

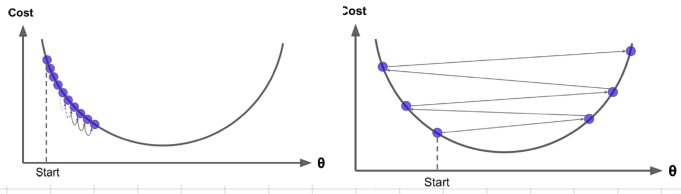
- Update $(W,b)^{k+1} = (W,b)^k - t_k g^k$ where t_k (hyperparameter) determine the size of the step called learning rate (can be taken proportional to g^k so the steps gradually get smaller near the minimum)



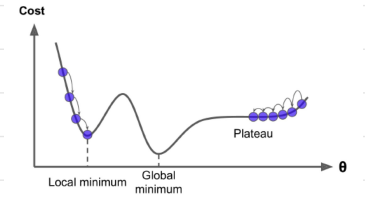
in this example there is no bias

Problems:

- small learning rate are computationally intensive; large learning rate the algorithm can diverge →



- Not all cost function are convex, so try different starting points →



- Scaling features (between 0 and 1 usually) so the algorithm is faster to converge
- Calculating $g^{(x)}$ over the whole training set is computationally heavy, compute for a training sample and use "Stochastic Gradient descent" or "Mini-Batch Gradient descent".

Terminology:
- Sample: one observation (input vector)
- Batch: number of samples used to make predictions (before updating the internal model parameters): in the Standard Descent (also called the Batch gradient descent) algorithm, batch = whole training set; stochastic gradient descent the Batch is 1; for the mini-Batch gradient descent $1 <$ Batch $<$ Size of training set
- Epoch: one complete pass through the entire training set

Example: assume you have a dataset with 200 samples and you choose a batch size of 5 and 1000 epochs.

How many batches are there? $\frac{200}{5} = 40$ batches

How many samples does each batch contain? 5 samples

Based on how many batches will the model weights will be updated? The model is updated after each batch (1 batch).

How many batches does one epoch contain? 1 epoch = use all the dataset $\Rightarrow$ 40 batches

How many updates of the model are done within one epoch? 40 updates

How many times will the model pass through the whole dataset? One for each epoch $\Rightarrow$ 1000 times

How many batches will the model pass through during the training phase? $\underbrace{40}_{\text{batche for 1 epoch}} \cdot \underbrace{1000}_{\text{n° of epochs}} = 40000$

→ Stochastic Gradient descent : for each sample i, randomly selected without replacement (the algorithm runs through all samples in a random order):

- calculate the gradient $g^k = \nabla_{W,b} L(Y^i, \hat{Y}^k(X^i))$
- update the solution $(W,b)^{k+1} = (W,b)^k - t_k g^k$

Repeat this as many times as the number of epochs.

→ Mini-Batch Gradient descent: for each batch B_k (batches randomly formed as a partition of the training set):

- calculate the gradient $g^k = \frac{1}{b_k} \sum_{i \in B_k} \nabla_{W,b} L(Y^i, \hat{Y}(X^i))$, b_k the batch size $|B_k|$
- update the solution $(W,b)^{k+1} = (W,b)^k - t_k g^k$

Repeat this as many times as the number of epochs.

→ Calculation of the gradient : $\nabla_{W,b} L$ contains derivatives w.r.t. all weights and biases from different neurons and layers. Denote by a_k^{l-1} the output of neuron k at layer $l-1$ and z_j^l the input of neuron j at layer l: $z_j^l = \sum_k w_{jk} a_k^{l-1} + b_j^l$

Using the chain rule: $\frac{\partial L}{\partial w_{jk}^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \underbrace{\frac{\partial L}{\partial z_j^l}}_{\text{error rate} =: \delta_j^l} \cdot a_k^{l-1} =: \delta_j^l \cdot a_k^{l-1}$ (we will see similar for bias)

→ Back propagation : algorithm that involves a forward and a backward pass:

- Initialization: initialize the weights and biases of the network with small random values.

  It is important to initialize them randomly!

  If they are all initialized to zero, all neurons in a given layer will be perfectly identical ($z_j^1 = \sum_{i=1}^n w_{ij}^1 x_i + b_j^1 = 0 + b_j^1 = b_j^1 \Rightarrow a_j^1 = \sigma(z_j^1) = \sigma(b_j^1)$ ... and so on).

  So the back propagation will affect them in exactly the same way $\Rightarrow$ they will remain identical.

- Forward pass: use mini-batch and for each of them: pass the input through the network layer by layer computing and saving $a_k^{(l)}, z_j^{(l)}$ (we will use the in the backward pass)
- Calculate the error $L(Y, \hat{Y})$

- Backward pass: compute how much each weight and bias contributed to the error by applying the chain rule, we have to compute δ_j^l:

Weights

Goal: $\frac{\partial L}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1}$ ↑ saved in step for layer $l-1$

Biases

Goal: $\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot 1 = \delta_j^l$

So, we need to compute δ_j^l:

$a_j^L = \sigma(z_j^L)$

$$\delta_j^L = \frac{\partial L}{\partial z_j^L} = \frac{\partial L}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \underbrace{\frac{\partial L}{\partial a_j^L}}_{\text{computed directly perturbating output } a_j^L} \underbrace{\sigma'(z_j^L)}_{\text{known}}$$

Now, we know:

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} = \frac{\partial L}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial L}{\partial a_j^l} \sigma'(z_j^l) =$$

$$= \left(\sum_k \frac{\partial L}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial a_j^l} \right) \sigma'(z_j^l) =$$

$$= \left(\sum_k \delta_k^{l+1} w_{jk}^{l+1} \right) \sigma'(z_j^l)$$

↑ we are using the results of layer $l+1$

$z_k^{l+1} = \sum_j w_{jk}^{l+1} a_j^l + b_k^{l+1}$

so I find a_j^l in every z_k^{l+1} in layer $l+1$ ⇒ I use

$\frac{\partial v}{\partial u} = \sum_w \frac{\partial v}{\partial w} \frac{\partial w}{\partial u}$ where $w = z_k^{l+1}$

Activation function in backpropagation:

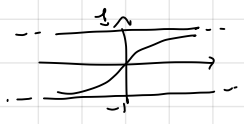
↳ the step function contains only flat segment and so there is no derivative to work with.

↳ The initial suggested activation function was the sigmoid function cause has non-zero well-defined derivative everywhere

↳ Backpropagation also works well with:

the hyperbolic tangent activation:

$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



In $\tanh$ $(-1, 1)$, differentiable, makes layers outputs to be around 0 at the beginning of the training; speed up convergence

rectified linear unit:

$\text{Relu}(x) = \max(x, 0)$; continuous but not differentiable in 0, despite this works well and fast convergence ⇒ has become the default

→ **Fine-tuning hyperparameters**: there are many hyperparameters to tweak: number of layers,

number of neurons in each layer, choice of activation functions, batch size for training, number of epochs, learning rate, regularization parameters, initialization of weights and biases.

Possible methods: grid search (very expensive), randomized search.

↳ Number of layers: rule of thumb: for many problems, start with a single hidden layer, it should give reasonable result. Then you can add in a way that you save time.

Intuition: if I have to draw a forest I can draw every tree (large amount of time) or I can draw a leave, copy the leave, draw a branch, paste many leaves attached to the branch, copy the branch with leaves, draw a trunk, paste branches with leaves to have a tree, copy the tree and paste many trees to have a forest.

↳ Number of neurons in each layer: before we used to do a pyramid-shape (e.g. 300 neurons in the first layer, 200 in the second, ...), but using the same number of neurons performs as well (even better sometimes) ⇒ rule of thumb, so start with a low number of neurons and then increase.

↳ Choice of activation function:

→ output layer
- → regression with positive output: ReLU
- → regression with bounded outputs: sigmoid or hyp. tangent
- → classification with binary output: sigmoid
- → classification with multiple outputs: softmax

↓

Hidden layer(s): ReLU most common

Softmax activations: can be used for the output layers, taking as inputs the outputs of several neurons; is an extension of sigmoid function: $SM(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_i}}$

→ Vanishing gradient problem: from the back propagation equations:

$$\frac{\partial L}{\partial w_{jk}^{l}} = a_k^{l-1}\delta_j^l = a_k^{l-1}\left(\sum_k w_{kj}^{l+1}\delta_k^{l+1}\right)\sigma'(z_j^l) = a_k^{l-1}\left(\sum_k\left(\sum_p \underbrace{w_{kj}^{l+1}w_{pk}^{l+2}}_{(*)}\right)\sigma'(z_k^{l+1})\right)\sigma'(z_j^l)\dots$$

so if the derivative of the activation function is small ⇒ in deep networks the resulting gradient will be 0 and this makes the network hard to train. So tanh and ReLU are more useful

than sigmoid for this reason. Another solution could be normalize the inputs so that they remain in the range where σ' is large.

→ Why ReLU is so popular? Fast to compute, has derivative 0 or 1, not exposed to vanishing gradient problem; sparcity: when iput $< 0 \Rightarrow \nabla = 0$, should be choice 1 as long as the function to estimate can be approximated by piecewise linear function. Problem: if too many activations get below zero then most of the neurons will output zero.

→ Exploding gradient problem: even if $\sigma'(z) < 1$, we $\sigma'(z) > 1$ and so (*) can explode $\Rightarrow$ lead to NaN values $\Rightarrow$ we have to carefully initialize weights.

Xavier (Glorot) initialization: $Var(z_j^l) = Var\left(\sum_K w_{jK}^l a_K^{l-1} + b_j^l\right)$

so if b_j^l is initialize to zero + $w_{jK}^{(l)}$ i.i.d. and indipendent from a_K^{l-1} i.i.d. as well: ← + centered

$$Var(z_j^l) = \sum_K Var\left(w_{jK}^l a_K^{l-1}\right) = \sum_K Var\left(w_{jK}^l\right) Var\left(a_K^{l-1}\right) =$$

$$= n^{l-1} Var(w^l) Var(a^{l-1})$$

Since we want to preserve the variance of activations functions through layers:

$$Var(z_j^l) = Var(a^l) \simeq Var(a^{l-1}) = \frac{Var(z_j^l)}{n^{l-1} Var(w^l)}$$

$\Rightarrow Var(w^l) = \frac{1}{n^{l-1}}$ $\Rightarrow$ if we initialize $W_{ij}^l \sim N(\mu = 0, \sigma^2 = 0{,}01)$ $\Rightarrow$ we scale by a factor $\frac{1}{\sqrt{n^{l-1}}}$ $\Rightarrow$ $\frac{w_{ij}^l}{\sqrt{n^{l-1}}}$.

Remark: works well with Tanh, not for ReLU

→ Other hyperparameters:

↳ Batch size: we don't know well (seems larger is better)

↳ Learning rate: algorithm diverge for high learnin rate (quite obvious)

↳ Standardization: heterogeneous scales leads to large weights $\Rightarrow$ slow learning. Also it is easier to have vanishing/exploding gradient problem

↳ Pros and cons of neural network: Pros: flexible and benefit from large data

Cons: hard to interprete, perform poorly with small data, can be computationally heavy, choice of hyperpar.

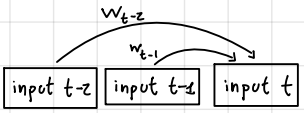
Recurrent neural networks (Chapter 8): a feed forward network is capable of approximating any function ⇒ why do we need advanced networks? The goal is to reduce the complexity of the network by exploiting the structure of the data: increase parsimony of parameters, decrease overfit and training time. RNNs useful in time series models, to learn sequences.

Indeed in this type of problem the data are dipendent from each other (the value at time t is dipendent on the values at previous time steps): $\hat{y}_t = f(X_t)$ where $X_t = (x_{t-T+1}, \dots, x_t)$

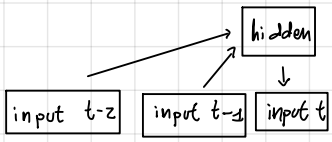
If the data are i.i.d. no sequence is needed ($T=1$) ⇒ use feed forward network.

↳ Memoryless models for sequences:

- autoregressive models: predict the next term in a sequence from a fixed number of previous terms



- feed-forward networks: generalize autoregressive models by using one or more layers of non-linear hidden units

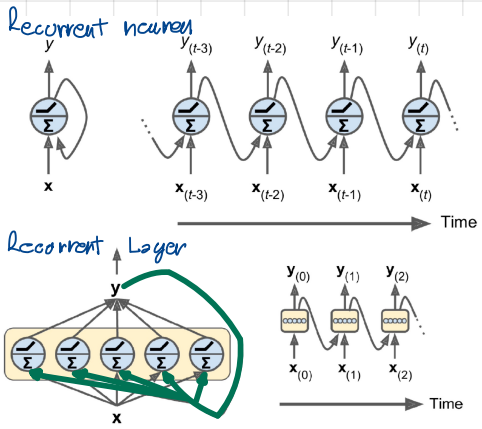


So what's new now?

Let's introduce the recurrent neuron: it is necessary to handle sequence prediction tasks by maintaining a memory of previous inputs. The idea is that at each time t (frame), the neuron receives the inputs $x(t)$ and its own output from the previous step $y(t-1)$ (the output in the first step is 0). Formally for a layer of recurrent neurons:

- each recurrent neuron has 2 sets of weights: one for the inputs $x(t)$, w_x, and one for the output of the previous step $y(t-1)$, w_y ⇒ two weighted matrices W_x, W_y. The output is given by:

$$y(t) = \sigma\left(W_x^T x(t) + W_y^T y(t-1) + b\right)$$



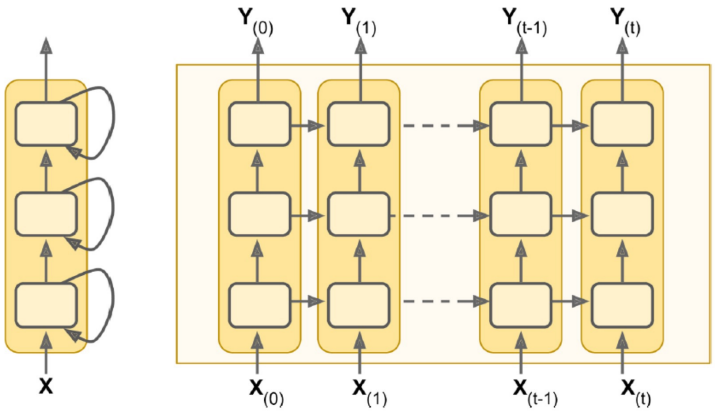
↳ Training recurrent neural networks: always two steps:

- unroll it through time (forward pass) (same as before but now it runs also through time)
- use back propagation (backward pass) (now the gradient is compute with respect to W_x, W_y, b)

→ Trend and seasonality : trend refers to the long-term direction in the data over a period of time. It represents the underlying movement. Seasonality refers to regular, repeating patterns in the data that occur at specific intervals due to seasonal factors.

With autoregressive models we need to remove trend and seasonality first, with RNN it is not necessary to do it, the model can learn them.

→ Deep recurrent neural network :



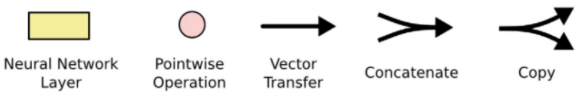
→ Problem of long-term dependencies : to train RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network that could leads to:

- vanishing/exploding gradient
- gradually forgets the first inputs in the sequence and so a long dependence (for example "The clouds are in the ..." "sky" has a short dependence but "I grew up in France ... I spak fluent..." "French" has a dependence on the first part, difficult to capture with RNN).

In RNN ReLU is not optimal : if the weights slightly increase ⇒ explosion of outputs.

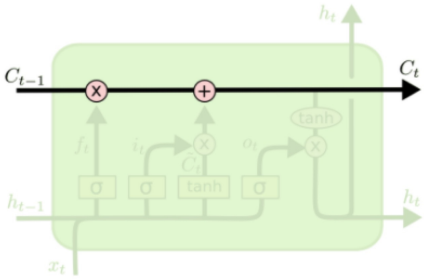
→ Long short-term memory networks : Main strenght: LSTM don't have the problem of long-term dependencies. It's a special case of RNN. We now understand how is done:

→ Notation :



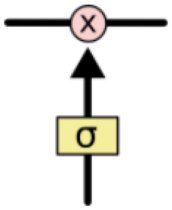
→ Components : • We have the **long-term memory cell** C(t) that carries information

across different steps :



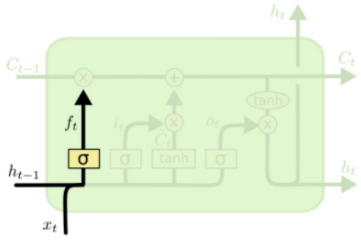
- The long-term $C(t-1)$ traverses the network from left to right, going through:
  - a forget gate (later) (dropping some memories ⊗)
  - an "addition" operation, adding some new memories ⊕
- The result C_t is sent straight out

- **Gates** : gates are a way to optionally let information through :



- They are composed by a sigmoid neural layer (outputs $\in [0, 1]$) and a multiplication operation
- In the extreme cases: if output is 0 they close the gate (let nothing through), if output is 1 they open it (let everything through)

- **Forget gate** : it's a gate so:



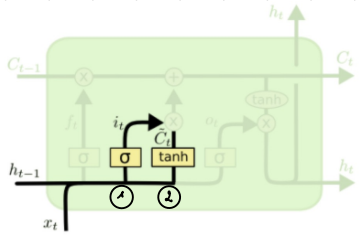
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

(σ: sigmoid; h_{t-1}: short-term (output))

- $f_t \in [0, 1]$
- Example: model try to predict next word in a sentence. C_{t-1} might include the gender of the present subject. If x_t is a new subject ⇒ forget the gender of the old subject.

- **Information to add** : is divided in two part:

1° part:

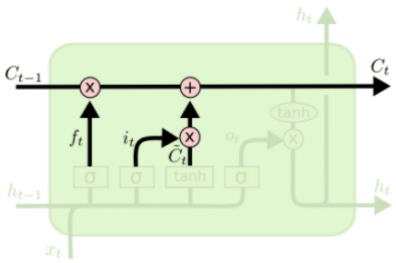


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

- ① is a sigmoid layer = input gate layer. Decides which values we'll update
- ② Tanh layer: create a vector of new candidate values $\tilde{C}_t$
- Next step: combine these two to create an update to the state

- Example: add the gender of the new subject to the cell state + replace the old one we're forgetting

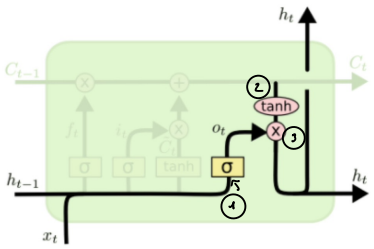
2° part : Update of the old cell C_{t-1} into the new cell C_t :



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Multiply the old state by f_t (forgetting parts)
- Add $i_t \otimes \tilde{C}_t$: new candidate values, scaled by how much we decided to update each state value

- **Output** : is a filtered version of the cell state C_t

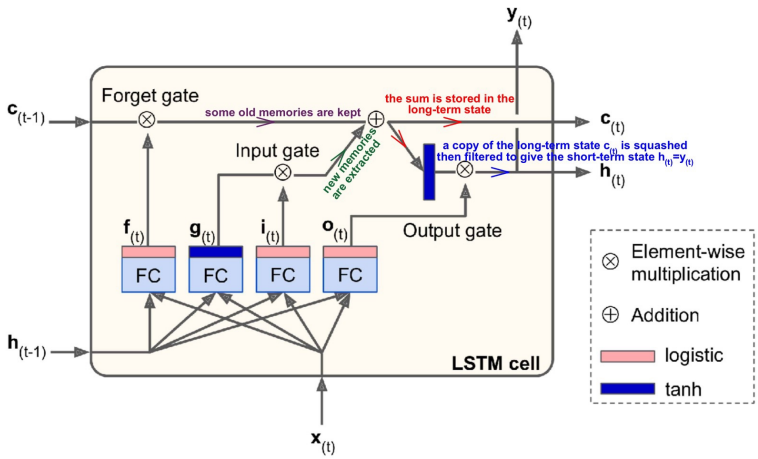


$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

- (1) sigmoid layer: decides what parts of the cell state we're going to output
- (2) Apply tanh to cell state (values $\in [-1, 1]$).
- (3) Multiply values by the output of the sigmoid gate $\Rightarrow$ we only output the parts we want

- Example: just saw a subject, might output whether it is singular or plural $\Rightarrow$ know what form the verb should be conjugated into.

Summary:



→ Application RNN to forecasting : we want returns 2 days ahead from daily historical data :

2 ways to do it: • r_{t+1} from $(r_t, \dots, r_{t-T})$, then r_{t+2} from $(\hat{r}_{t+1}, r_t, \dots, r_{t-T+1})$

• (r_{t+1}, r_{t+2}) from $(r_t, \dots, r_{t-T})$ (better)

Convolutional Neural Networks : consider an image with 224×224 pixels with 3 possible colours (RGB) for each pixel. If we use a feed forward network with 1 hidden layer and 1000 nodes, the # of connections between the pixel/colour (input) and 1st layer would be $\approx 150 \cdot 10^6$.

⟹ CNN dramatically reduce parametrization.

→ Architecture: they have three layers:

→ Convolution: we apply a filter called "Kernel" to the matrix data to extract some features.

The filter is a smaller matrix that moves across the matrix data.

For example:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image matrix

1	0	1
0	1	0
1	0	1

Kernel (representing a certain feature)

⇝ 1° Step.

1 (×1)	1 (×0)	1 (×1)	0	0
0 (×0)	1 (×1)	1 (×0)	1	0
0 (×1)	0 (×0)	1 (×1)	1	1
0	0	1	1	0
0	1	1	0	0

[ ] = Kernel matrix

and sum: $1+0+1+0+1+0+0+0+1 = 4$

so in the convolved matrix we'll have a 4 in position (1,1)

⇝ go on ⇝ convolved matrix:

4	3	4
2	4	3
2	3	4

Example:

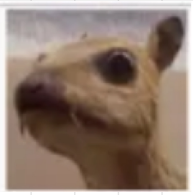


Image matrix

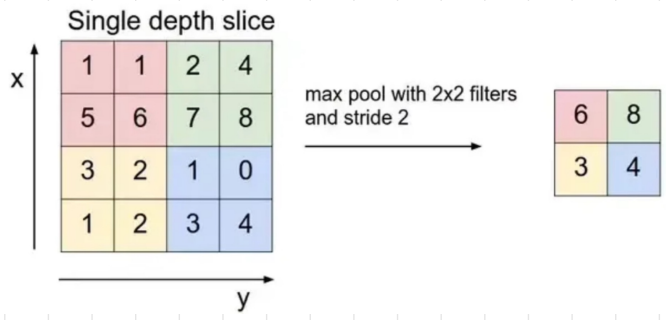
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Kernel for find edges

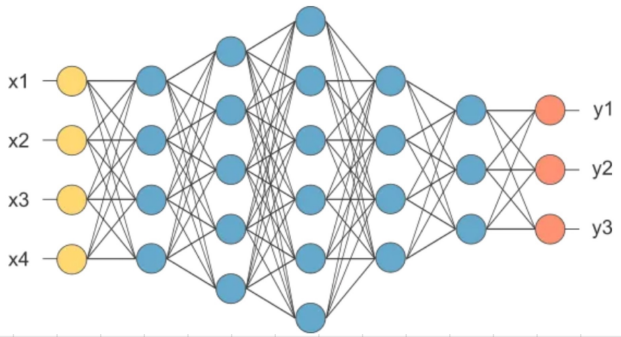
The stride is the number of pixels the kernel shifts over the input matrix (in the previous example was one).

Remark: in convolution there could be a ReLU layer to introduce non linearity.

→ Pooling: reduces dimensionality for large pictures. There is max pooling, average pooling and sum pooling (natural definition seeing the example:

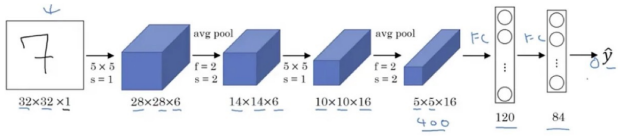


↳ Fully connected layer : transform output of pooling step into final output :



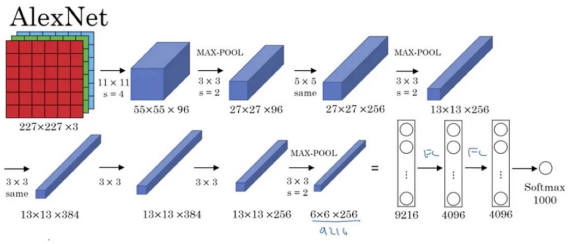
↳ Two examples:

LeNet-5 (1998)



- 5× 5 filters with stride of 1
- 6 filters, then 15
- Average pooling with 2× 2 filter and stride of 2
- Fully connected network 120 / 84 nodes
- sigmoid or tanh activation
- 60,000 parameters to learn

AlexNet (2012)



- 60 million parameters
- ReLU activation
- Dropout regularization

Remark : we can do "dropout regularization" that prevent overfitting randomly ignoring certain nodes during training. Hyperparameter : p = prob. that a node is dropped out. Applied to each batch.

NLP with Neural Networks (Chap. 10)

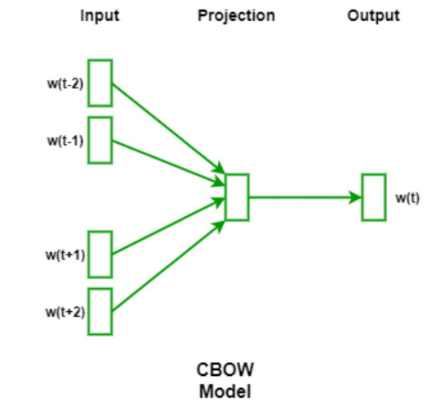
World embedding with neural networks

↳ Vectors representation for document vocabulary → 2 types:
- frequency based embeddings: the vectors contain statistical measurements like word count and TF-IDF
- prediction based embeddings: the vectors contains probabilities (ex. World2Vec, Glove)

↳ World 2 Vec : two layer neural network that analyzes the corpus and produces a set of vectors that represents the words. Generates a single word embedding representation for each word in the vocabulary (the word "bank" would have the same representation in "bank" deposite and in "riverbank")

↳ Two models for training

→ Continuous Bag of words (CBOW): predicts the probability of a word given the context. It's faster and has best performance with large data sets.

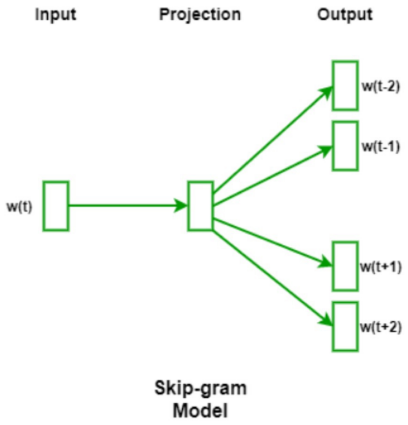


- We decide a context window of size $2n$ (we watch words the n words after and the n words before) (x_i = target, $\{x_{i-n}, x_{i-1}, x_{i+1}, \dots, x_{i+n}\}$ window)
- We try to predict the target word based on the context window: $P(x_i \mid x_{i-n}, \dots, x_{i-1}, x_{i+1}, x_{i+n}) = \exp\left(u_i^T \frac{1}{k} \sum v_k\right) / \sum_j \exp\left(u_j^T \frac{1}{k} \sum v_k\right)$

(u, v as in skip gram later)

Remark: The cross-entropy loss function can be used

→ Skip-gram: predicts the context given the target word. Works well with small data sets, and better representation for rare words.



- n-gram: contiguous sequence of n words
- skip-gram: drops items from the n-gram
- If a word appears in a sentence, how likely is it that the word x_j will appear in a window of a fixed size around x_i? Use neural network to predict $P(x_j \mid x_i)$: 1 hidden layer with linear activation, 1 output layer with softmax activation: $P(x_j \mid x_i) = \text{softmax}\left(U^T V x_i\right) = \text{softmax}\left(U^T v_i\right)$ (with a dictionary of N words and a hidden layer of d neurons we have $U, V \in \mathbb{R}^{d \times N}$ (d is the dimension of the embedding). The rows of U are the input vector representation and rows of V the output vector representation (can be used as embeddings)

Remark: as loss function the cross entropy loss function, $l(i,j) = u_j^T v_i - \log\left(\sum_j \exp\left(u_j^T v_i\right)\right)$

RNN for text analysis: text analysis can be done with feedforward networks but need to represent both short range and long range dependencies.

For the short range can use a Markov model $P(w_t \mid w_{t-1}, w_{t-2}, \dots) = P(w_t \mid w_{t-1})$

For the long range dependencies we need to extend to n-grams:

$$P(w_t \mid w_{t-1}, w_{t-2}, \_) = P(w_t \mid w_{t-1}, \_, w_{t-n})$$

Hidden variable: $P(w_t \mid w_{t-1}, \_, w_{t-2}) \approx P(w_t \mid h_{t-1})$ where h_t approximately encodes all history up to t and is a function of the previous state and current input

$$h_t = f(h_{t-1}, x_t)$$

With RNN, y_t is the predicted output (e.g. $P(w_t \mid w_{t-1}, w_{t-2}, \_)$ with:

$$h_t = \sigma(Ux_t + Wh_{t-1})$$

$$y_t = \text{softmax}(Vh_t)$$ (historical information is stored in the output of hidden neuron)

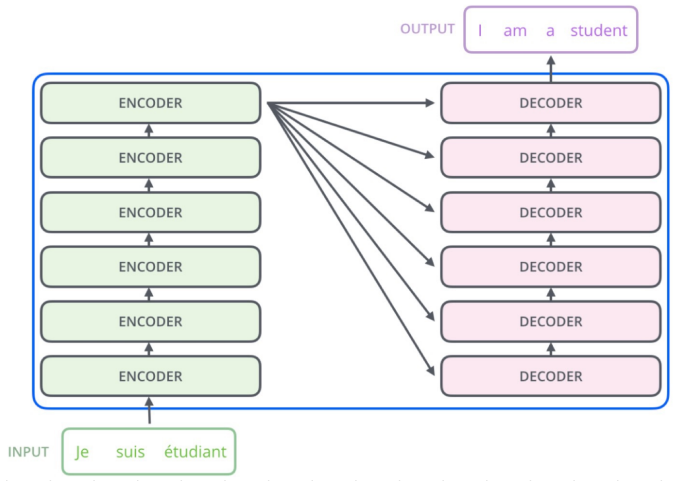
Problems: • often, words after contains useful information. We could use bidirectional RNN but not very good to capture long-range dependencies

TRANSFORMERS: type of neural network architecture used in NLP. Let's start with two essential components of transformers:

→ Encoder: processes the input sequence and transforms it into a continuous representation that captures the essential features and contextual information of the input.

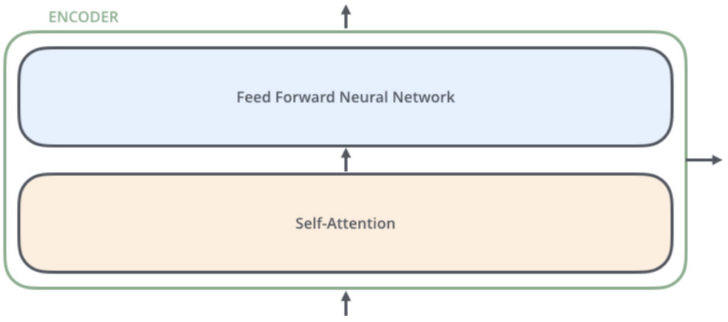
→ Decoder: generates the output sequence one token at time, using the encoder's output and the tokens it has generated so far.

→ Architecture of transformers: in the original paper there are 6 encoders and 6 decoders:



Each encoder has 2 components:

- Self-attention component: helps the encoder look at other words in the input sentence as it encods a specific word
- Feedforward architecture (to avoid vanishing/exploding gradient)



→ Self attention: let x_i be the inputs (word embeddings), each x_i plays 3 roles:

- Query (q): determines how much focus to place on other parts of the input sequence;
- Key (K): represents the data points used to compute the attention score for the queries
- Value (v): the actual data that are aggregated based on the attention scores.

We obtain q_i, K_i, v_i thanks to learnable weight matrices W_q, W_K, W_v:

$$q_i = W_q x_i \quad , \quad K_i = W_K x_i \quad , \quad v_i = W_v x_i$$

Computed q_i, K_i, v_i, we define:

$$w_{ij} := \frac{q_i^T K_j}{\sqrt{d}} \longrightarrow W_{ij} := \text{softmax}(w_{ij}) \longrightarrow y_i = \sum_j W_{ij} v_i$$

($\sqrt{d}$: makes the gradient more stable; then normalizes)

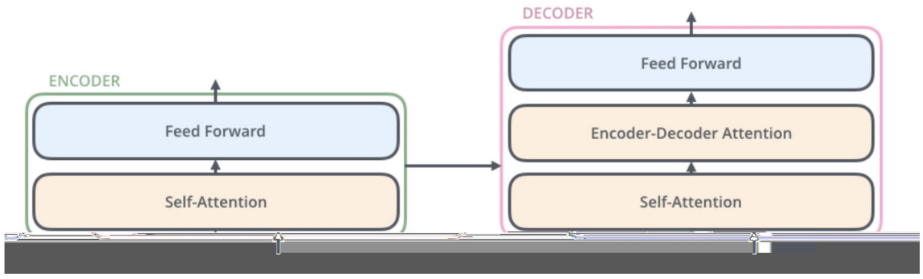
Remark: we can concatenate different self-attention mechanism to increase flexibility, we talk about Multi-head self-attention. Each head (self-attention) is indexed with r, the output are computed for each head and then aggregated:

$$y_i = W_y \, \text{concat}\left(y_i^1, \dots, y_i^r, \dots \right)$$

(W_y: linear trasformation (learnable matrix))

Remark: Self attention is permutation equivalent (problem because "Jack eats an hamburger" is equivalent to "an hamburger eats Jack") ⇒ we add a vector in the embedding that encodes the position of the token.

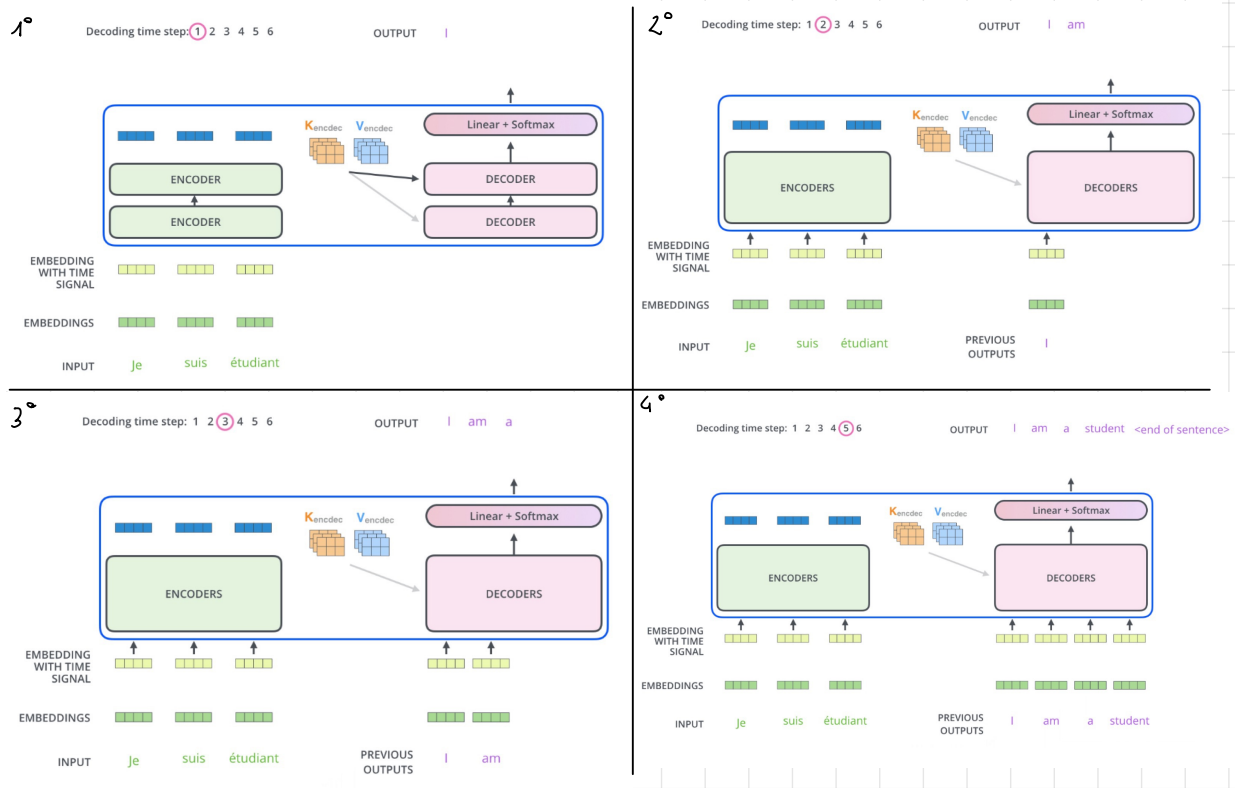
→ Decoder : the decoder has 3 components :



- The additional attention layer helps the decoder focus on relevant parts of the input sentence

- The output at each step is a vector of probabilities. The cell with the highest probability is chosen, and the word associated with it is produced as the output.

→ Step by step : how transformers change "Je suis étudiant" in "I am a student".



Note that decoder has as output a vector of floats. We turn that into a word using linear + softmax layers : linear projects the vector into a vector of the size of dictionary (logits vector) and assigns a score to each word ; softmax layer turn score in probabilities, the cell with the highest probability is chosen and the word associated is the output for this step.