ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

MACHINE ET DEEEP LEARNING POUR LA CYBERSÉCURITÉ

Pre. OUHMIDA Asmae

Année universitaire 2025/2026

02

Réseaux de neurones artificiels (ANN)

ML vs DL

	Machine Learning	Deep Learning
Approche	repose sur des techniques et des algorithmes d'apprentissage automatique plus traditionnels, tels que les arbres de décision, les forêts aléatoires, la régression logistique,	repose sur des réseaux de neurones artificiels profonds composés de nombreuses couches cachées. Il est inspiré par la structure du cerveau humain.
Caractéristiques	les caractéristiques (features) des données sont généralement extraites manuellement par des experts en données. Cela signifie que la qualité des caractéristiques peut influencer les performances du modèle.	la capacité d'apprendre automatiquement des caractéristiques à partir des données, ce qui élimine souvent la nécessité d'une extraction manuelle de caractéristiques
Volume de données	efficace avec des ensembles de données de taille modérée. Il n'a pas nécessairement besoin de quantités massives de données.	un grand volume de données, et il peut être très efficace pour les problèmes nécessitant des données massives, notamment dans un contexte de Big Data.
Applications	la classification, la régression, la recommandation, la détection d'anomalies, l'analyse de texte, la vision par ordinateur, etc.	vision par ordinateur (reconnaissance d'images, détection d'objets), le traitement du langage naturel (traduction automatique, analyse de sentiment,,,,

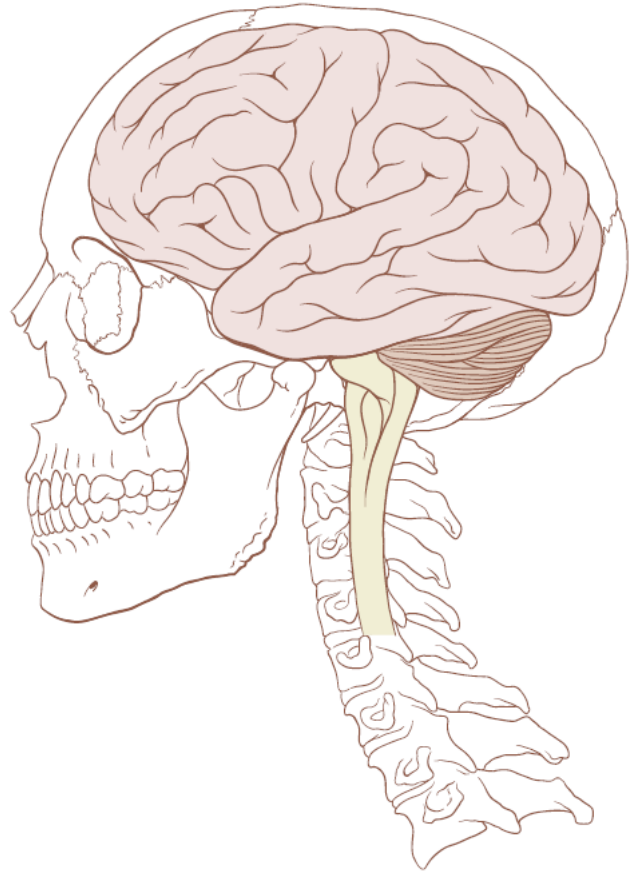
Quand utiliser Deep Learning

- Résoudre des problèmes complexes qui impliquent des données massives et des modèles hautement non linéaires.
- Il nécessite généralement un grand volume de données d'entraînement, une puissance de calcul significative et peut être plus difficile à interpréter que d'autres méthodes d'apprentissage automatique. Par conséquent, il est recommandé de l'utiliser lorsque les autres approches ne donnent pas de résultats satisfaisants ou lorsque la complexité de la tâche justifie son utilisation.

INTRODUCTION

Le cerveau humain

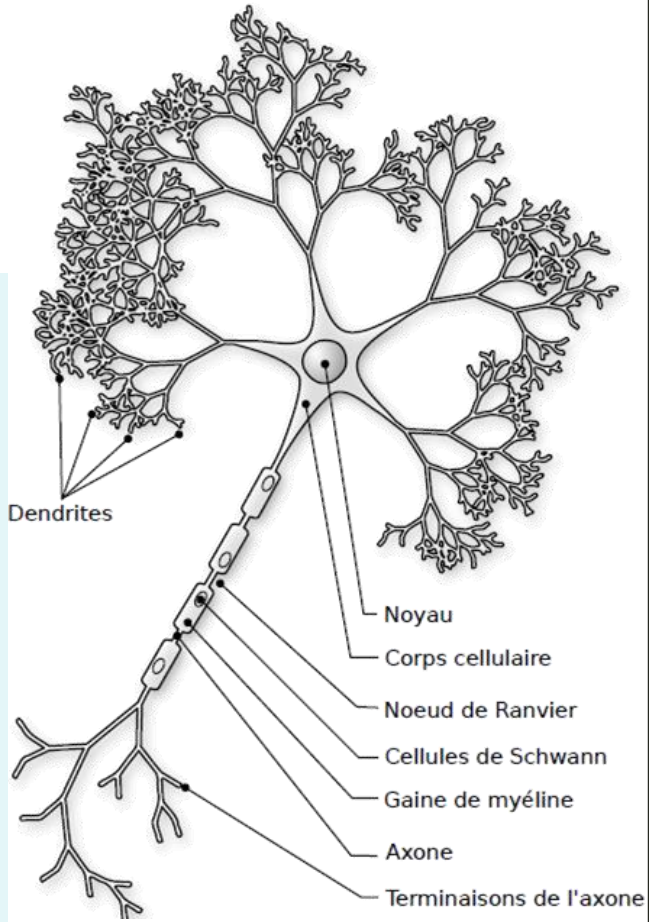
- Nombre de neurones dans le cerveau humain : 100 milliards
- Nombre moyen de connexions par neurone : 10 000
- 1mm3 de cortex contient un 1 milliard de connexions



INTRODUCTION

Le cerveau humain

- Un neurone est une cellule capable de transmettre des informations à d'autres neurones au travers de ses différentes connexions (synapses).
- Il existe plusieurs types de neurones (pyramide, panier, Purkinje, etc.) avec des fonctionnements différents (sensoriel, moteur, interneurones)
- Les neurones sont interconnectés et forment des réseaux



INTRODUCTION

Le cerveau humain

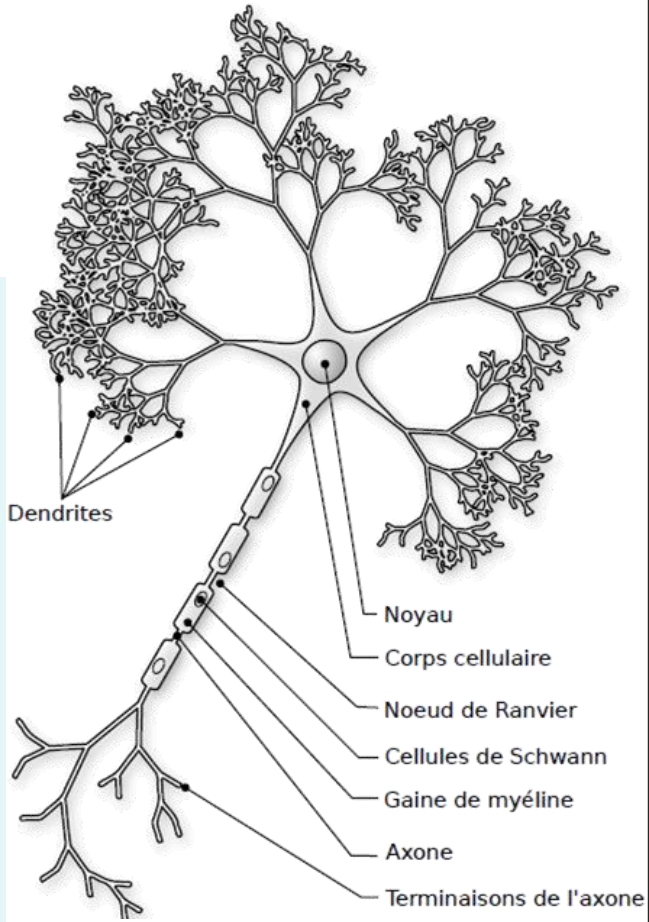
Une cellule neuronale du cerveau biologique est caractérisée par :

Un noyau (modélisée dans le neurone artificiel par la portion circulaire qui représente l'unité de calcul)

➔ Des dendrites qui sont des récepteurs ou canaux pour acheminer l'information (modélisées par des èches indiquant les entrées à l'unité de calcul),

➔ L'Axone (canal de sortie) qui permet de transférer l'information au monde extérieur,

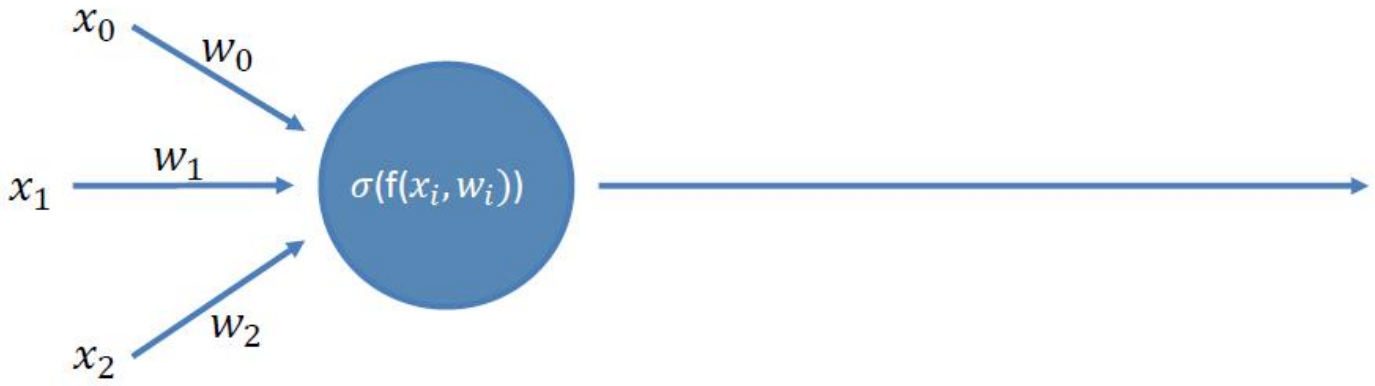
➔Enfin les synaptiques qui distribuent cette information aux autres neurones (modélisés par la flèche de sortie ou lien avec le monde extérieur).

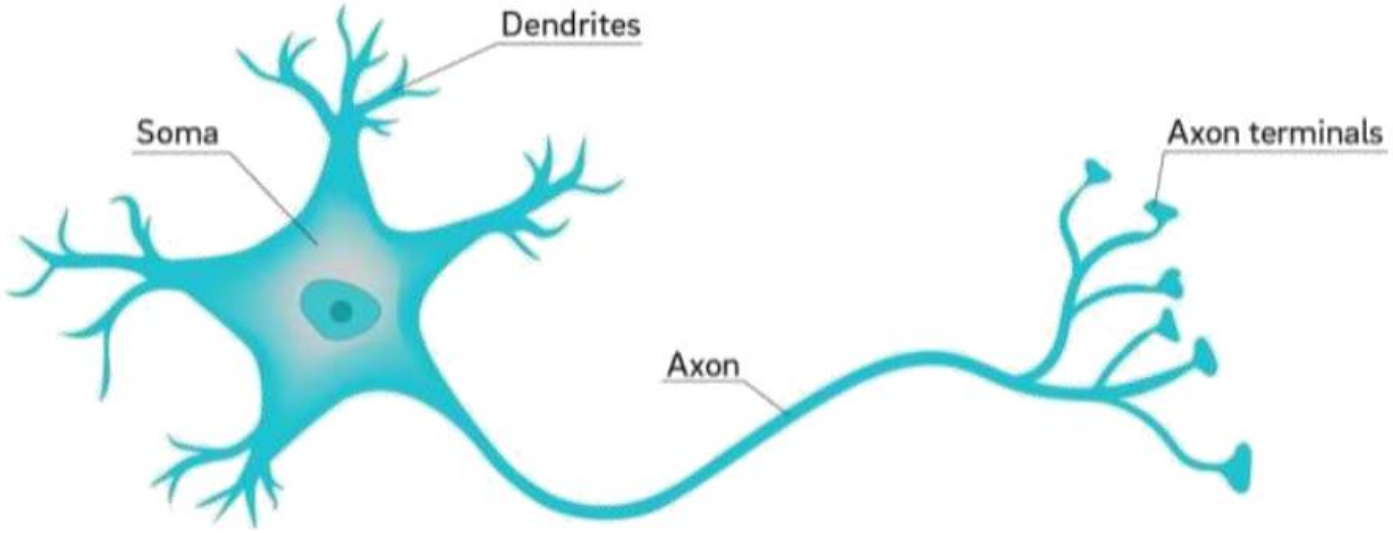


INTRODUCTION

Le cerveau humain

Le neurone artificiel a la même structure et le même comportement





INTRODUCTION

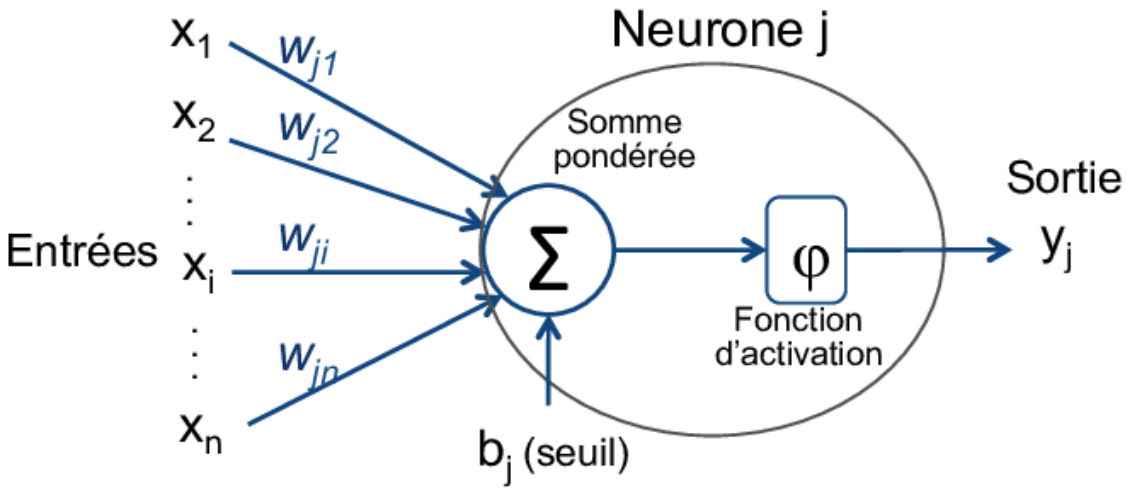
Un neurone artificiel

Chaque neurone j du réseau est un élément processeur, il est aussi définit comme "une fonction non linéaire, paramétrée, à valeurs bornées" .
Un neurone reçoit des valeurs en entrée x_i associées à des poids w_{ji} représentant l'importance de ces entrées.

Il renvoie une valeur unique comme sortie, qui peut être envoyée à plusieurs neurones.
Une fonction de combinaison calcule le potentiel du neurone qui est la somme pondérée des entrées et leurs poids à la quelle se rajoute le seuil b_j :

$$\mathbf{V_j = b_j + \sum x_i w_{ji}}$$

Une seconde fonction φ appelée fonction d'activation ou fonction de transfert est appliquée à ce potentiel pour générer la valeur en sortie y_j .

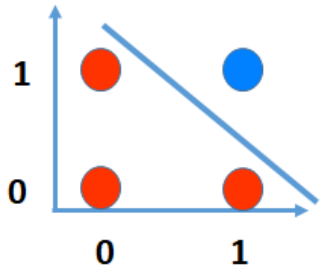


Problème non linéairement séparable

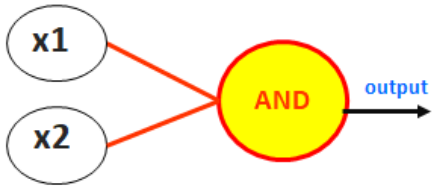
Séparation de And, Or et XOR

And

	F	T
T	F	T
F	F	F

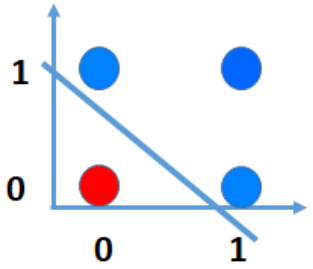


On peut avoir une séparation linéaire

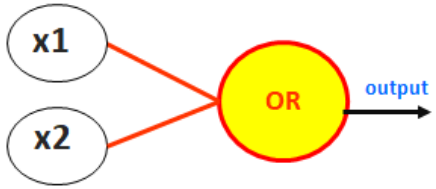


OR

	F	T
T	T	T
F	F	T

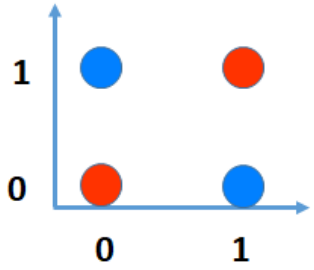


On peut avoir une séparation linéaire



XOR

	F	T
T	T	F
F	F	T



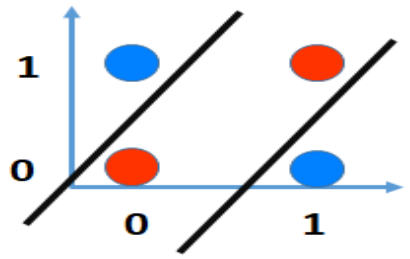
On ne peut pas avoir une séparation linéaire

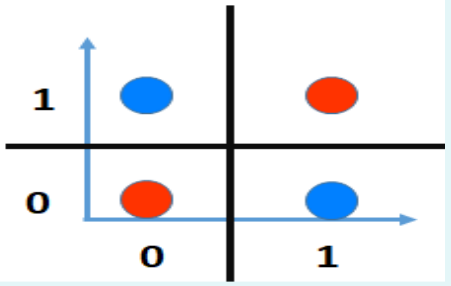
PRBLÈME NON LINÉAIREMENT SÉPARABLE

XOR

XOR

	F	T
T	T	F
F	F	T





➔ On peut utiliser plus qu'un séparateur linéaire pour faire la séparation

Théorèmes :Résultats sur les perceptrons

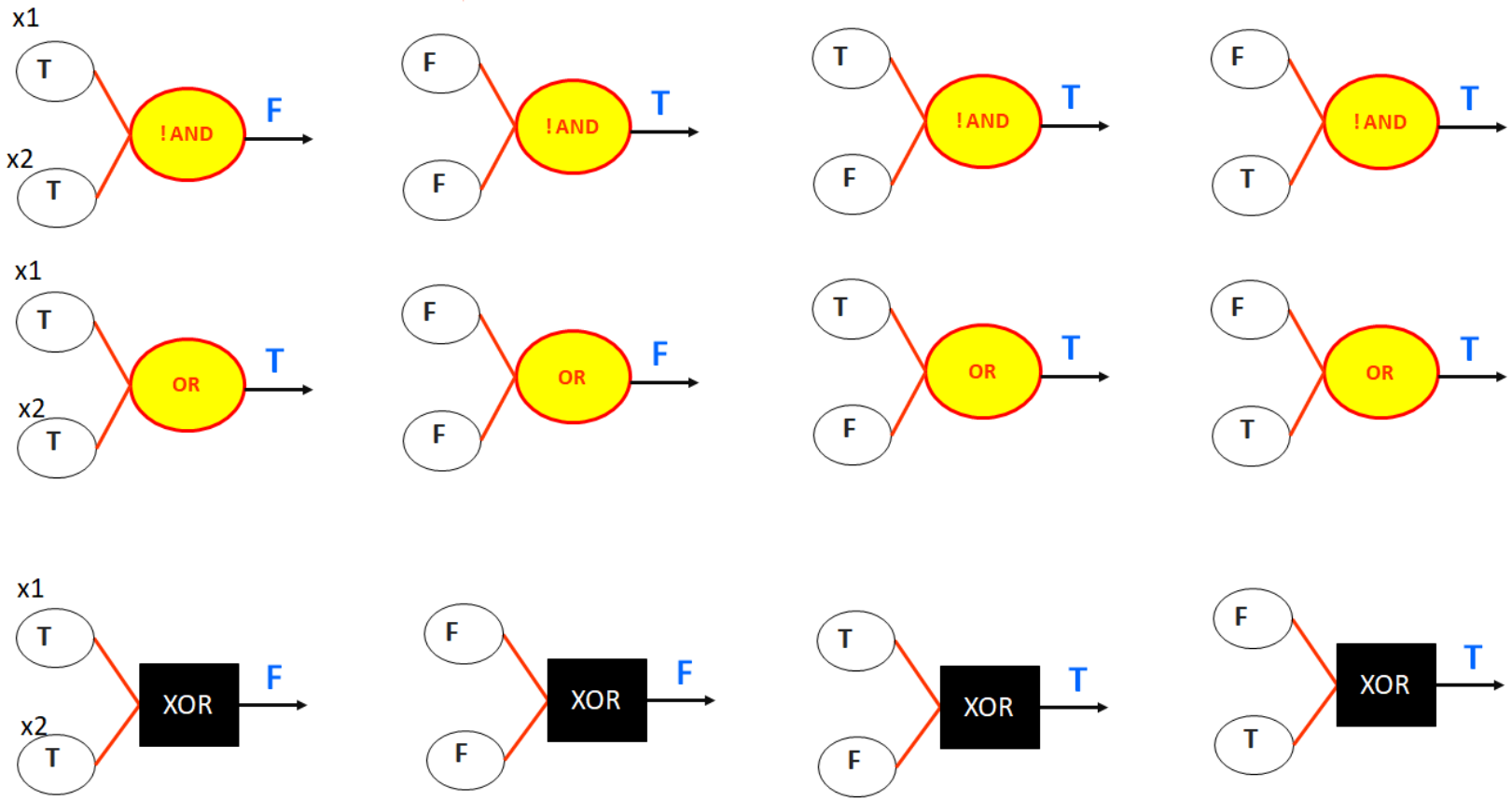
- Un perceptron linéaire à seuil à d entrées divise l'espace des entrées en deux sous-espaces délimités par un hyperplan.
- Tout ensemble linéairement séparable peut être discriminé par un perceptron.
- **Le XOR ne peut pas être calculé par un perceptron linéaire à seuil.**

Problème non linéairement séparable

Limites des perceptron simple

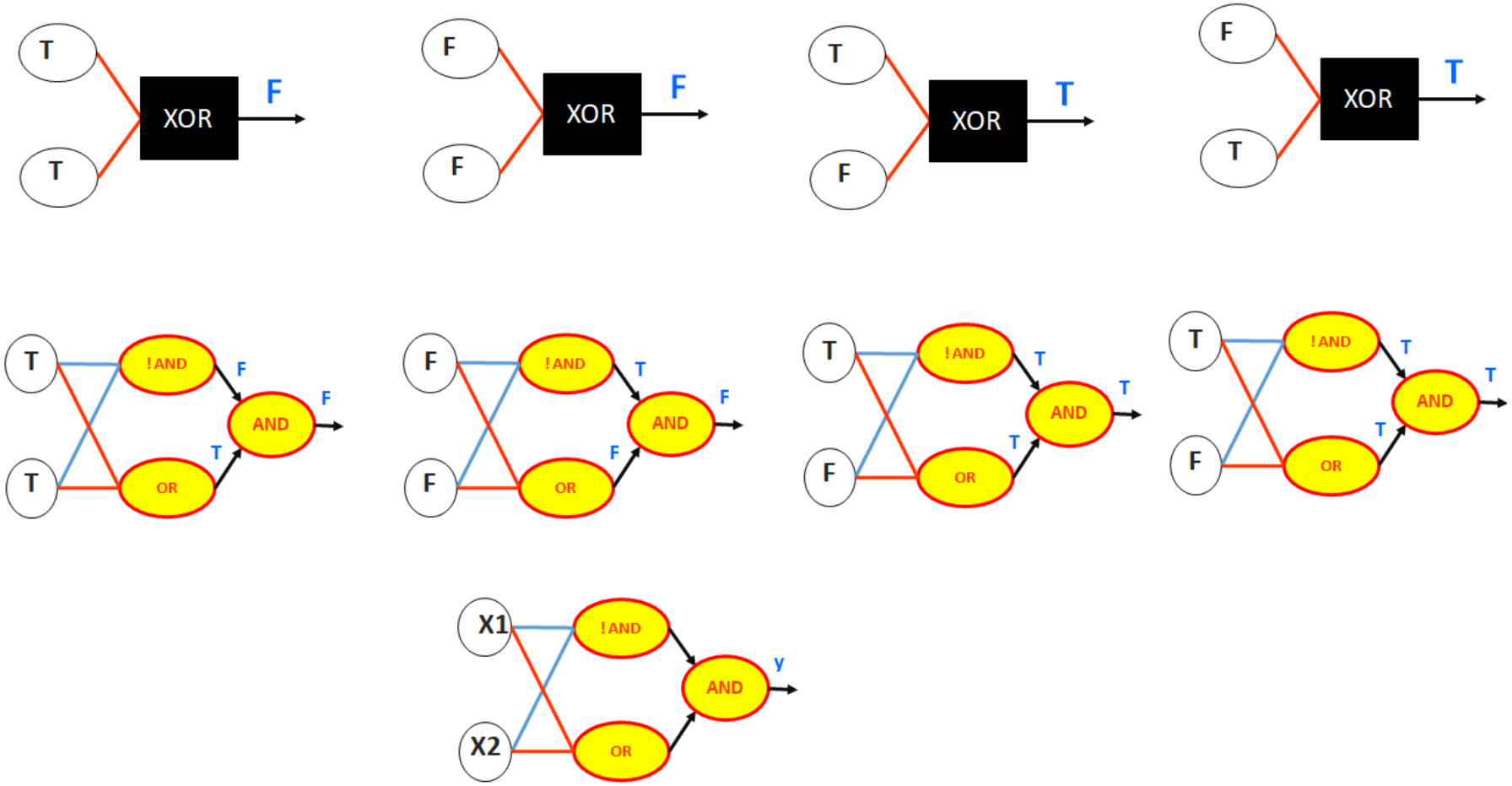
- Le perceptron simple ne peut résoudre que des problèmes linéairement séparables. Pour aller plus loin, il est nécessaire d'ajouter des couches.

XOR=(!AND) AND (OR)



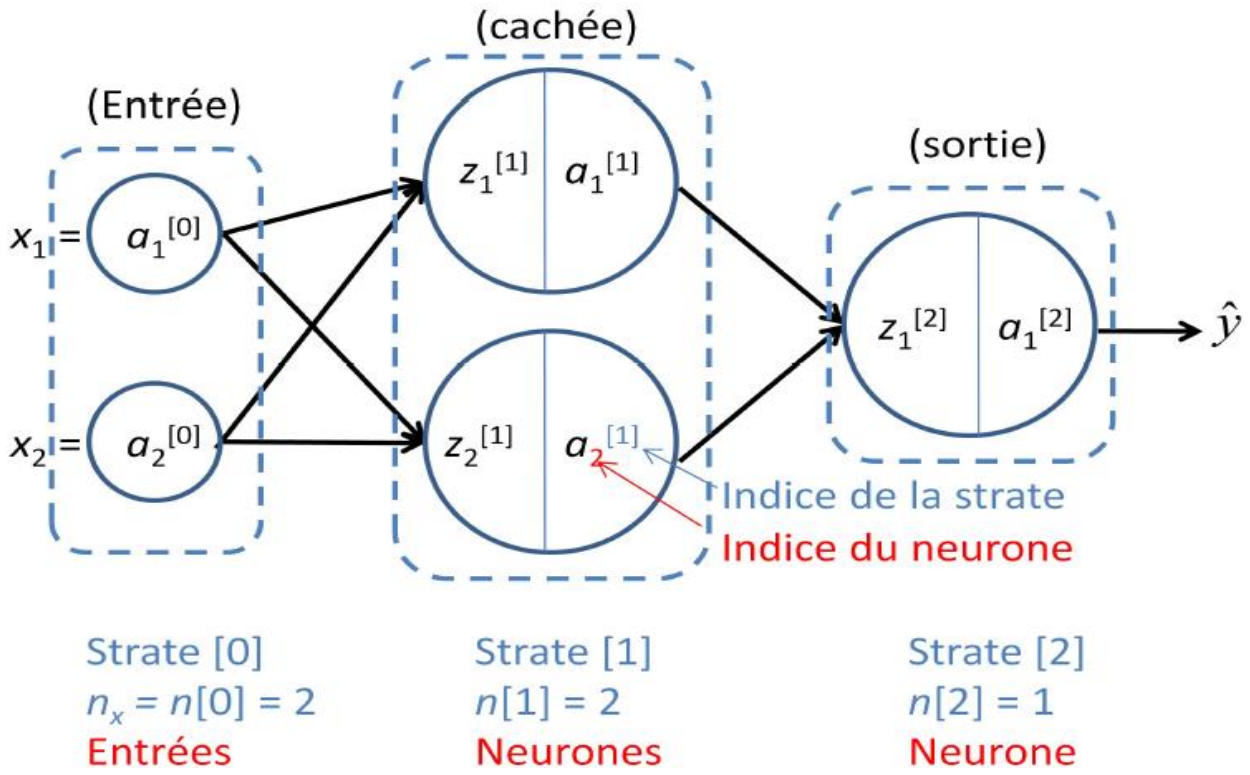
- On peut voir que XOR est la négation de AND et OR .

XOR=(!AND) AND (OR)



- On peut voir que XOR est la négation de AND et OR .

RÉSEAUX DE NEURONES ARTIFICIELS



Vocabulaire :

Strate[0]=strate d'entrée

Strate[1]=strate cachés

Strate[2]=strate de sortie

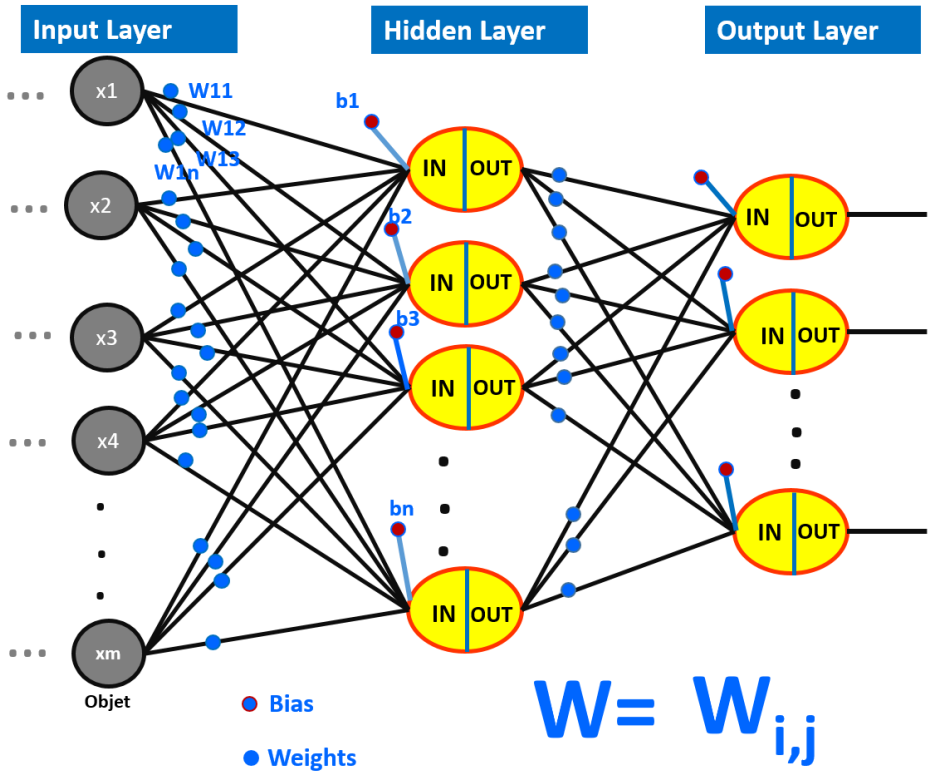
Notations

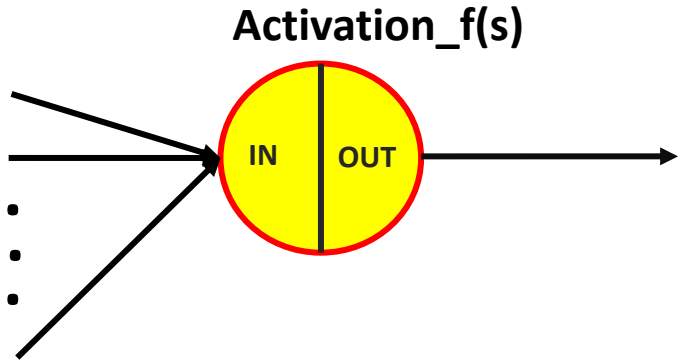
n[l]= nombre de neurones dans la strate [l]

$z_i^{[I]}$=portion linéaire du neurone i de la strate [l]

$a_i^{[I]}$ = activation du neurone i de la strate [l]

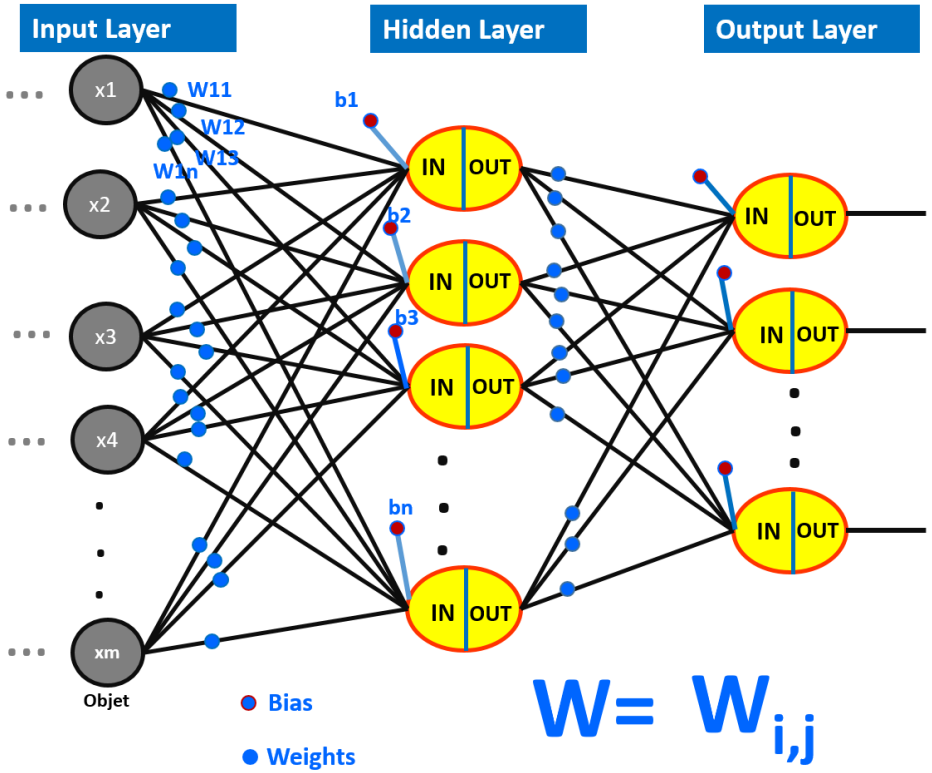
Les fonctions d'activation

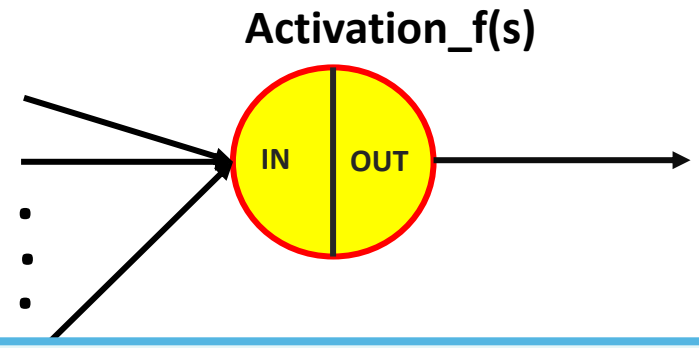




- Une fonction d'activation est une fonction mathématique appliquée à la sortie d'un neurone (ou unité) dans un réseau de neurones.
- Elle détermine si le neurone doit être activé (envoyer une sortie) ou désactivé (ne pas envoyer de sortie) en fonction de l'entrée qu'il reçoit.
- Elle joue un rôle crucial dans l'apprentissage profond et est un élément central de l'architecture d'un réseau de neurones.

Les fonctions d'activation



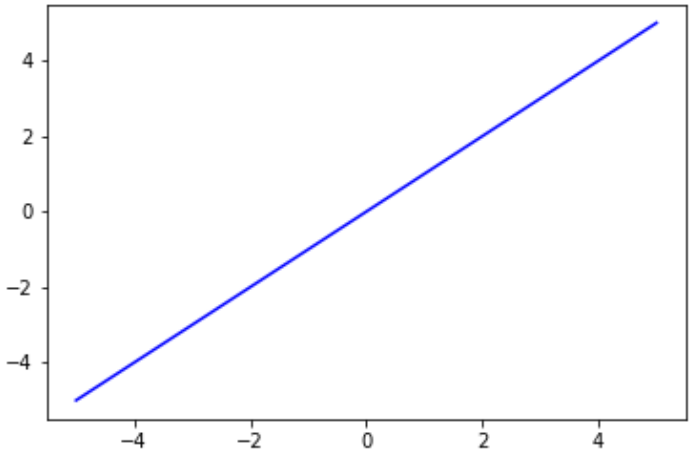


- Si on n'utilise pas la fonction d'activation, le signal d'output devient une simple fonction linéaire
- Sans fonction d'activation, notre réseau de neurones devient une simple ML (régression linéaire) limité
- La fonction d'activation rend le réseau de neurones capable de résoudre des problèmes non linéaires (images, son, vidéo…)
- Imaginons un réseau sans fonction d'activation : y = Wx

➔ Aucune non-linéarité, chaque couche est une simple transformation linéaire

- Les réseaux de neurones artificiels sont conçu autours des fonctions d'approximation et des poids
- L'algorithme de backpropagation est utilisé pour optimiser les poids

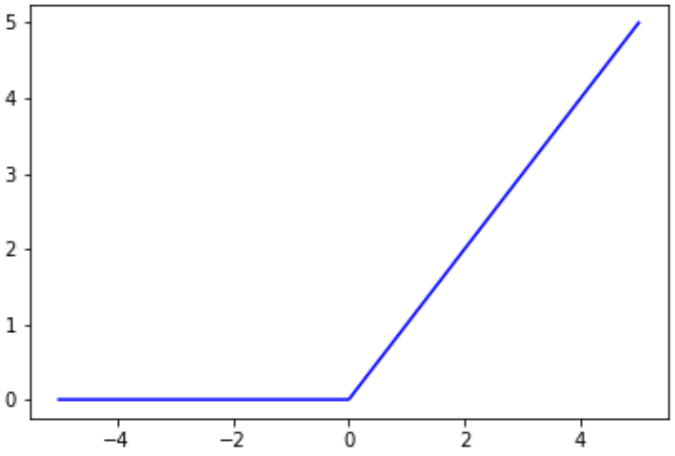
Les fonctions d'activation



$$u = x$$

Linear function

- Dédié à la régression
- Aucune transformation

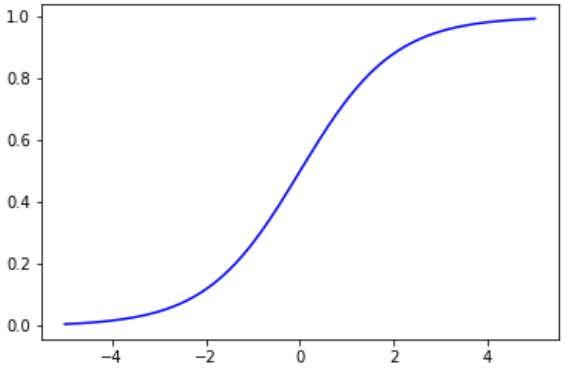


$$u = \max(0; x)$$

ReLU (Rectified Linear Units)

- Filtrer les valeurs négatives
- Utilisation en Deep Learning (facile à calculer (évaluation et dérivation)

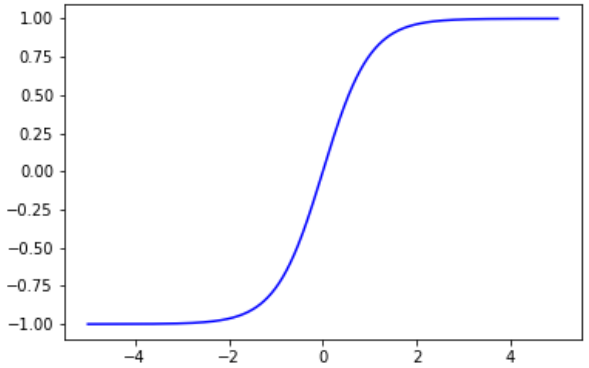
Les fonctions d'activation



$$u = \frac{1}{1 + e^{-x}}$$

Sigmoïd

- Déplacez l'activation dans la plage [0, 1]
- Utiliser pour la classification, non pertinent pour les couches cachées

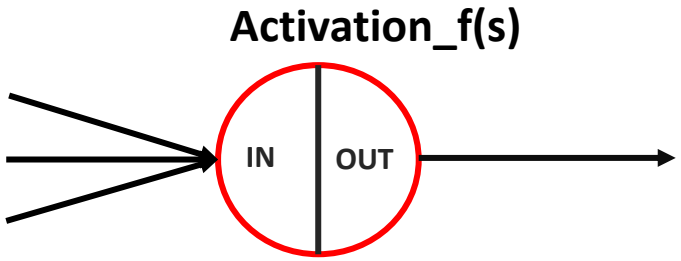


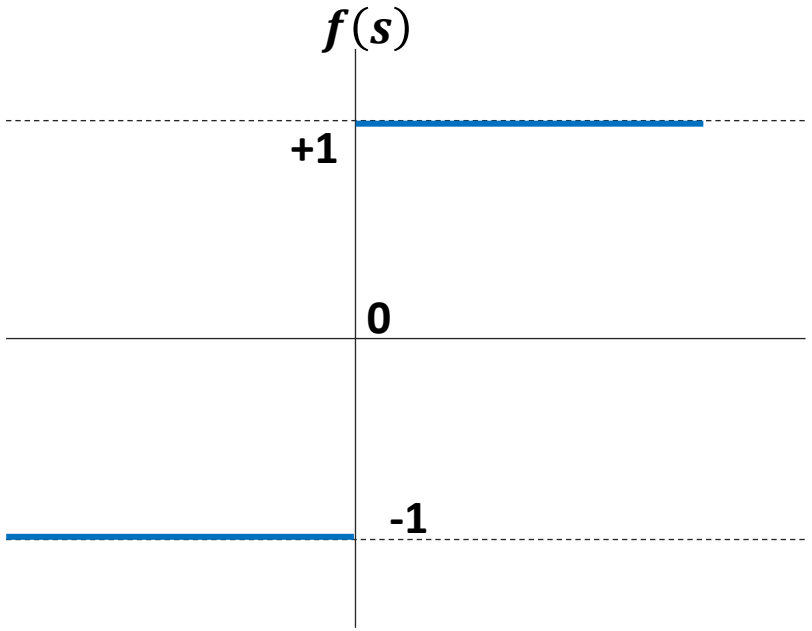
$$u = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Hyperbolic Tangent

- Déplacez l'activation dans la plage [-1, 1]
- Utiliser pour la classification, non pertinent pour les couches cachées

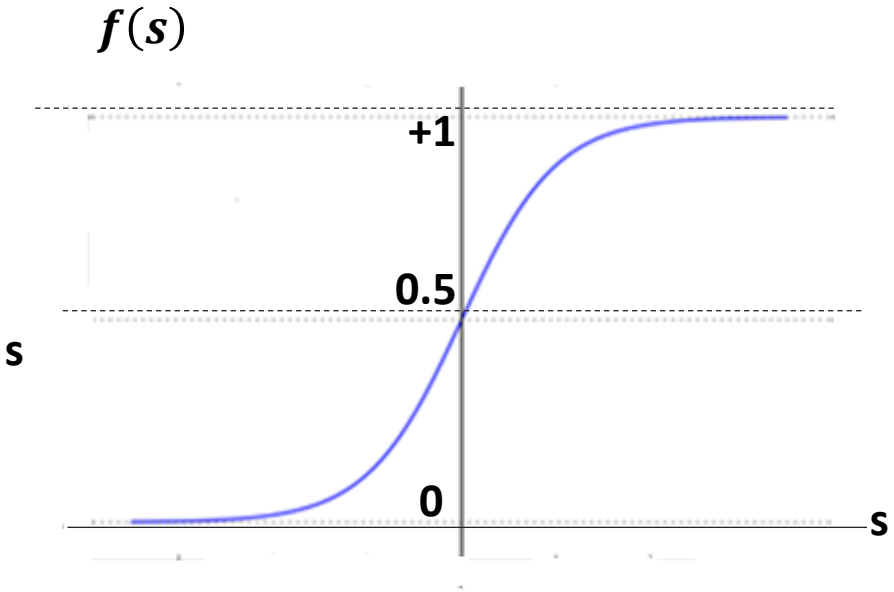
LES FONCTIONS D'ACTIVATION (SORTIE)





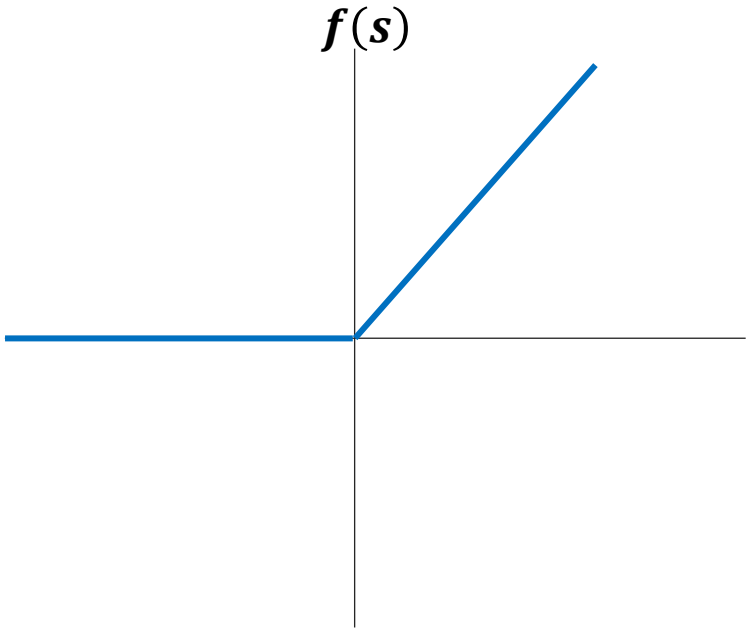
$$f(s) = \begin{cases} 1, \text{ si } s>=0 \\ -1, \text{ si non} \end{cases}$$

Sign function



$$f(s) = \frac{1}{1 + e^{-s}}$$

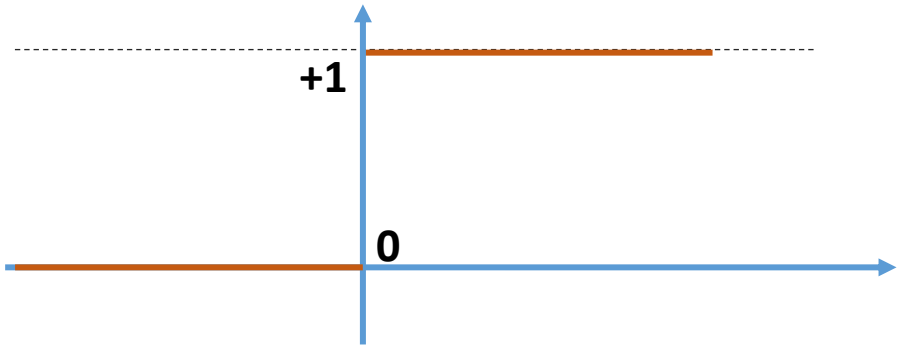
Sigmoid function
(Logistic function)



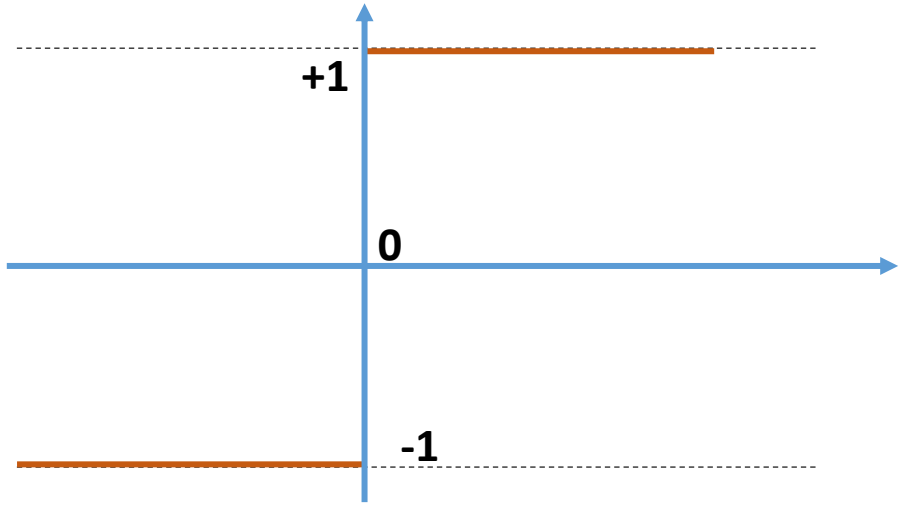
$$f(s) = \begin{cases} 0, \text{ si } s<=0 \\ s, \text{ si non} \end{cases}$$

Rectified Linear Unit
ReLU

Step function & Sign function



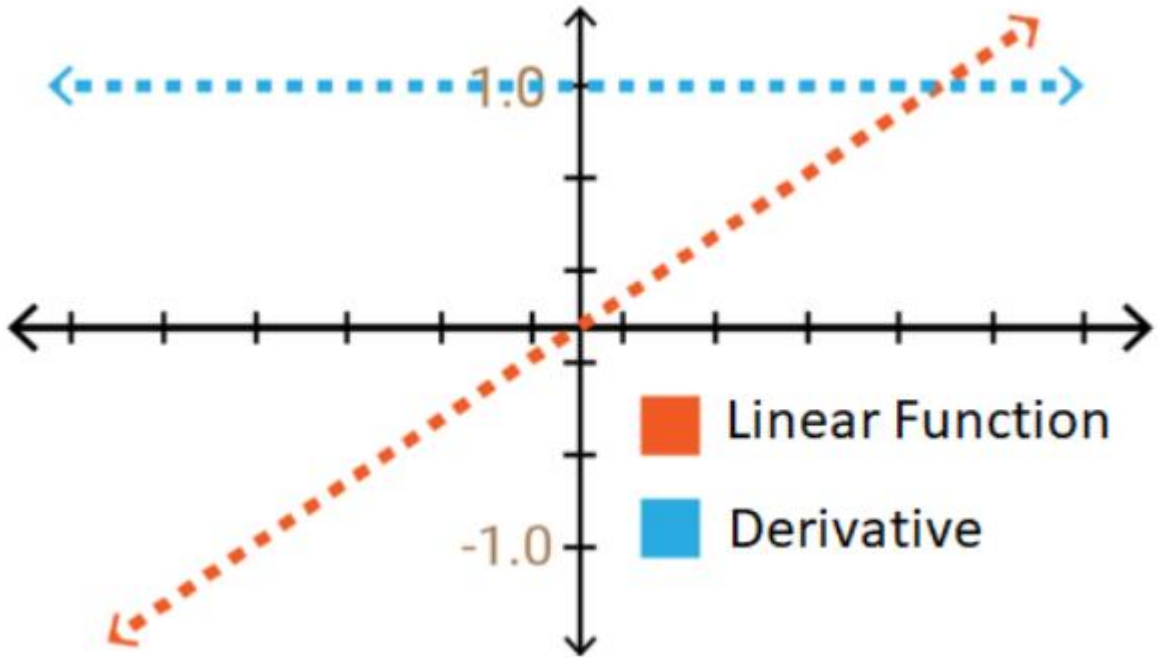
Step function



Sign function

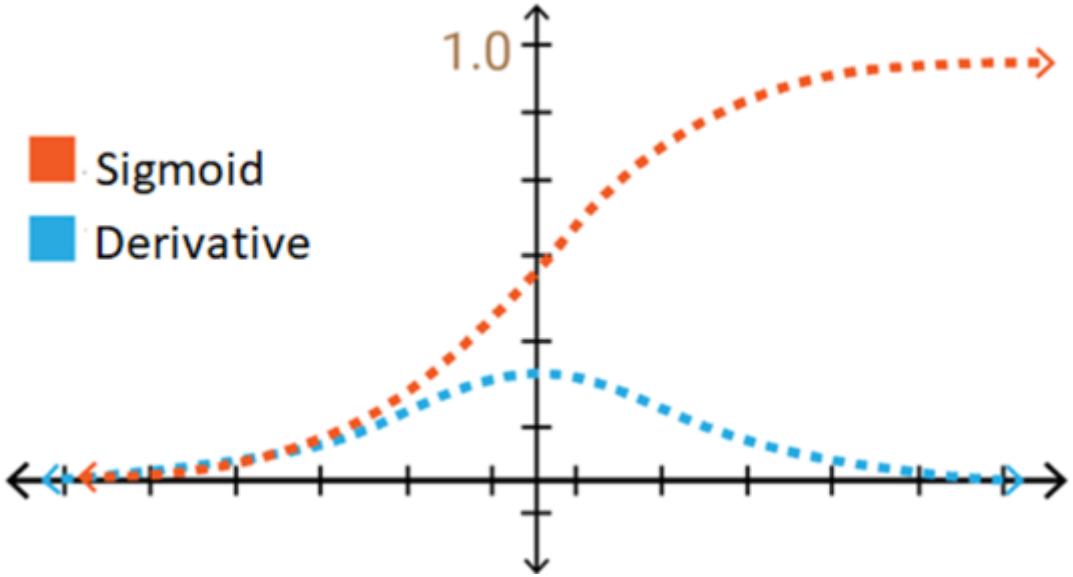
- Fournissent une sortie binaire: utilisées pour la classification binaire
- Dans le cas d'un réseau multilayer, elles sont préférées dans output-layer
- ce n'est pas recommandé de les utiliser dans hidden-layers

Linear function



- Au contraire de step function et sign function, linear function peut générer une série d'output
- Le grand problème de cette fonction est que sa dérivée est constante
- L'algorithme de backpropagation est basé sur les dérivées
- La dérivée d'une telle function f(x)=cx est f'(x)=c
- Donc le processus d'apprentissage ne peut pas dépendre des entrée: ce qui est faut
- On peut l'utiliser dans output-layer

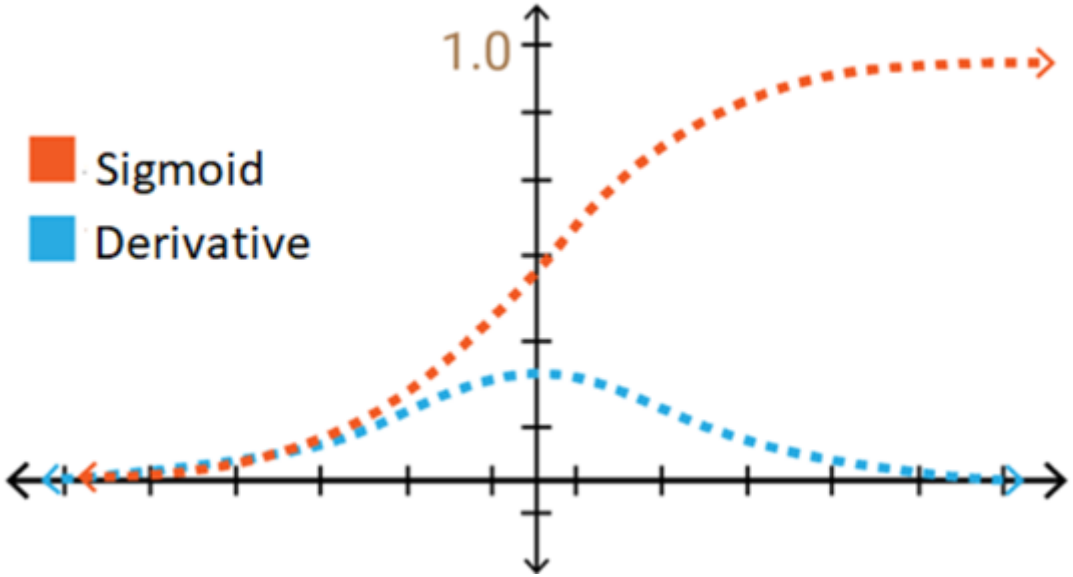
Sigmoid function



- Sa dérivée est différente de sign, step et linear functions: apprentissage possible
- On peut utiliser cette fonction pour faire de la classification:

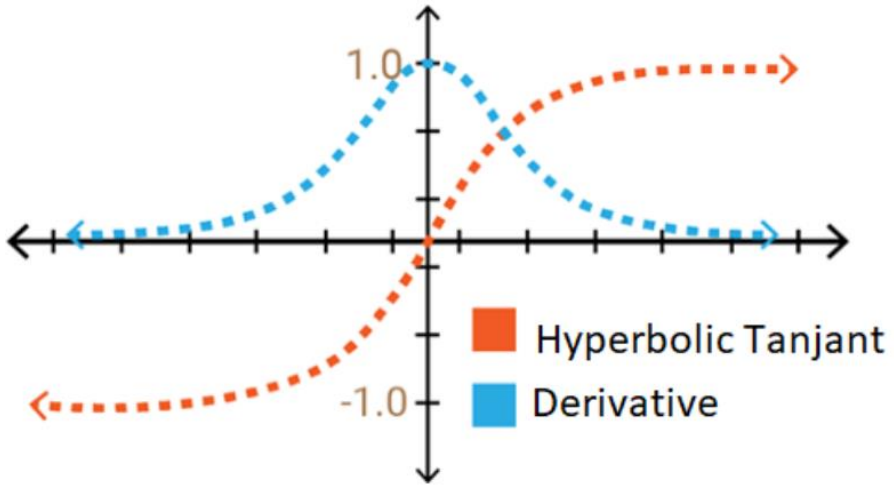
Le changement de y si x se trouve entre -2 et 2, on constate qu'il ya un changement rapide
Mais, le changement de y est très lent si on s'éloigne de cet intervalle
Parmi les fonctions d'activation les plus utilisées, mais il y a autres fonctions meilleures que sigmoid

SIGMOID FUNCTION-PROBLEM



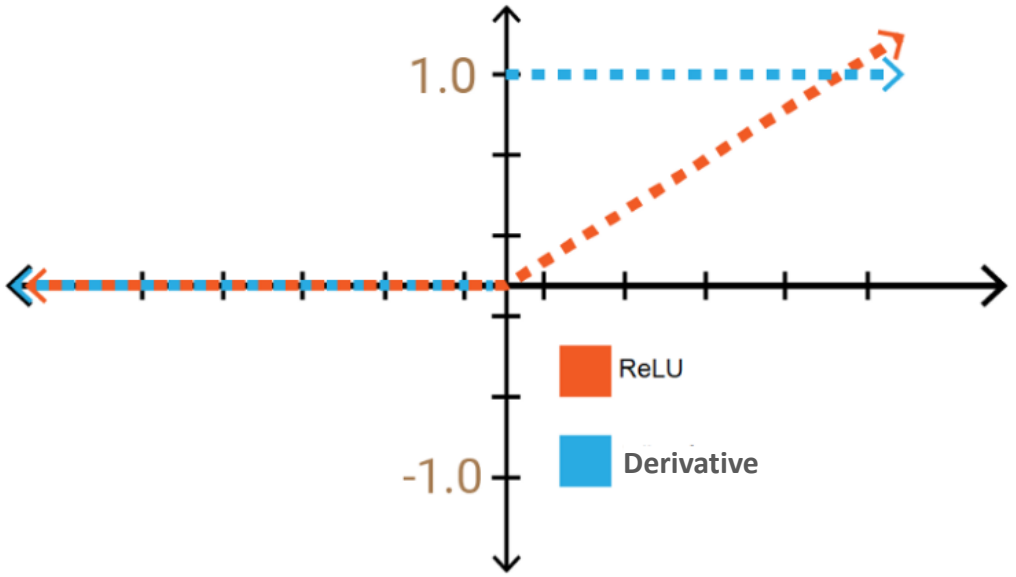
- Si on regarde attentivement le graphe de sigmoid, on constate qu'elle change très lentement quand x a des valeurs qui sont loin de 0
- Cela dit que: la dérivée de cette fonction dans ces valeurs sont très proches de 0: apprentissage très très lent

Hyperbolic Tangent function



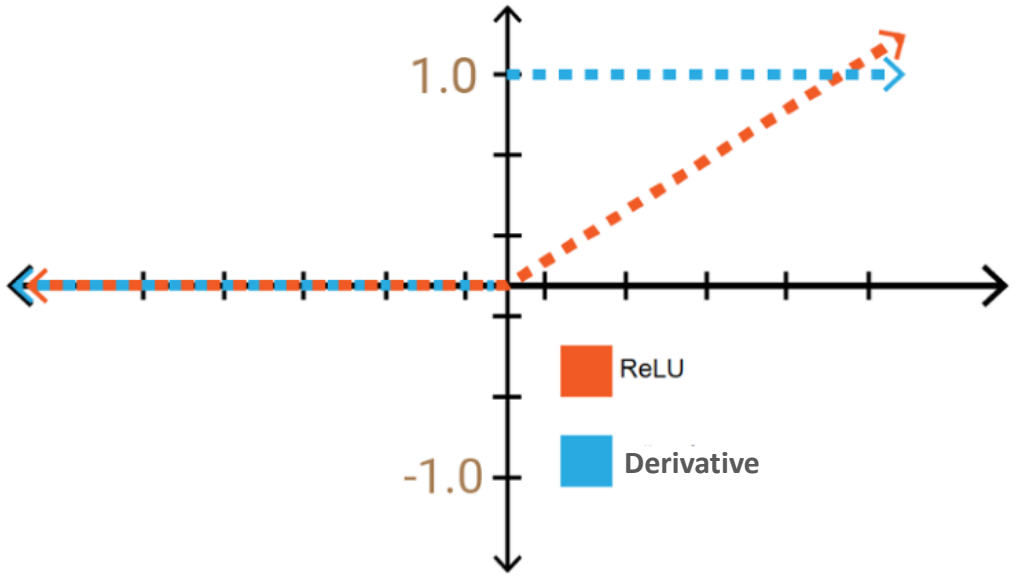
- Similaire à sigmoid, mais son y varie entre -1 et 1
- Plus de possibilité de sortie
- souffre du même problème que sigmoid

Rectified Linear Unit (ReLU) function



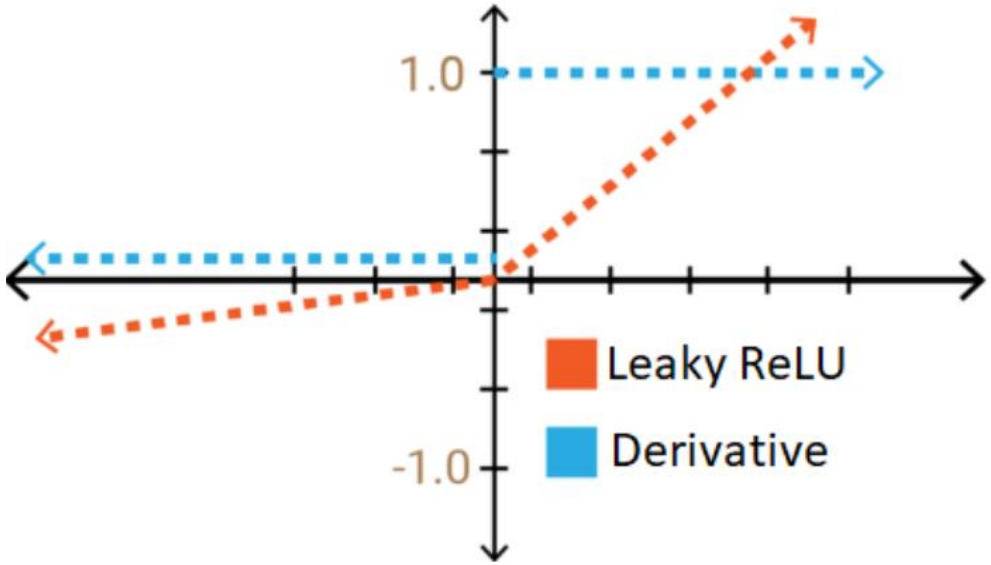
- Elle a presque la forme d'une fonction linéaire, mais elle n'est pas linéaire
- Utile dans des réseaux de neurones profonds
- Dans telle situation, sigmoid ou tanh procède à une activation intense des neurones
- ReLU peut désactiver des neurones: assurer un apprentissage rapide

RECTIFIED LINEAR UNIT (RELU) FUNCTION-PROBLEM



- ReLU assure une accélération de l'apprentissage à l'aide de la région de y=0
- Mais le problème qu'il n' y a pas d'apprentissage dans cette région

Leaky Rectified Linear Unit (Leaky ReLU) function



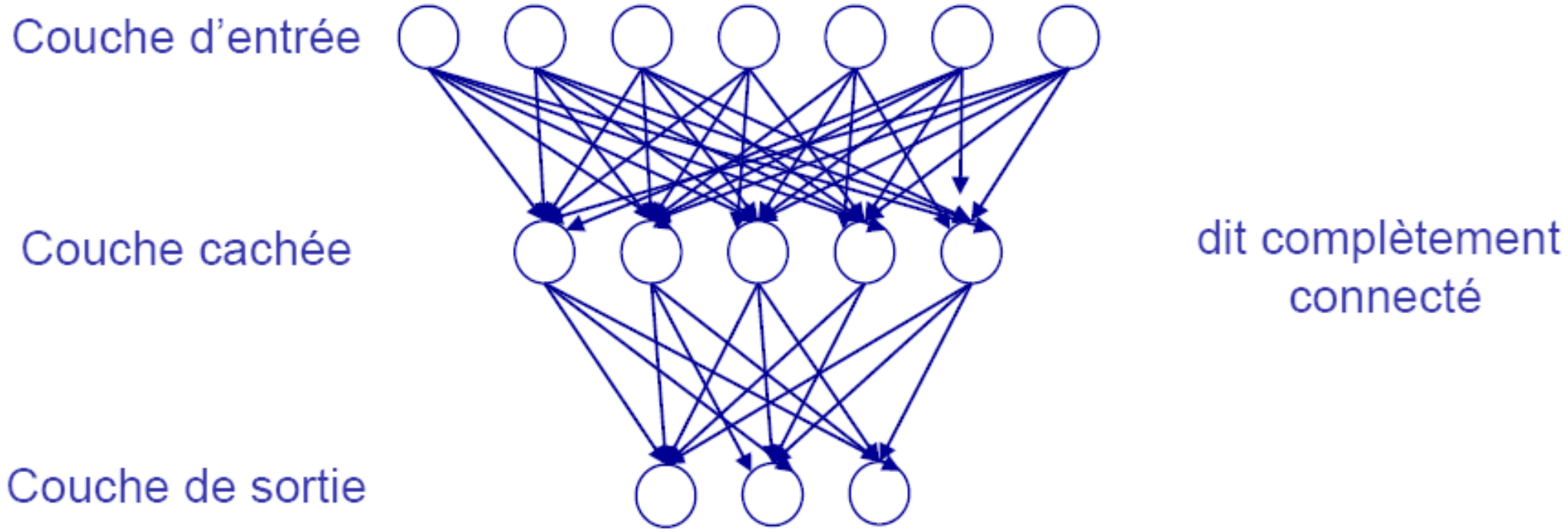
- Même rôle que ReLU en évitant le problème de la zone y=0

Les fonctions d'activation

- Les fonctions d'activation jouent un rôle important dans hidden –layer et output layer
- Le choix des fonctions d'activation dépend de la nature du problème et les contraintes à prendre en compte
- Il s'agit d'un problème d'optimisation (problème)
- Il existe plusieurs autres fonctions
  - Exemples:
  - Softmax: même structure que sigmoid et est elle préférée dans la couche de sortie en deep learning

Le Perceptron MultiCouches (PMC)

- Structure



Le Perceptron MultiCouches (PMC)

- **Comment choisir une telle structure ?**

– Par expérience

– Par les données :

- Couche d'entrée :

➔Prendre un nombre de neurones égal à la taille de la donnée

- Couche cachée :

➔Faire varier le nombre de neurones de manière à obtenir une meilleure précision

- Couche de sortie

➔Le nombre est égal au nombre de classes que l'on cherche à distinguer

Le Perceptron MultiCouches (PMC)

BackPropagation

- Mécanisme essentiel pour entraîner les réseaux de neurones.
- Permet d'ajuster les poids du réseau pour minimiser la fonction de perte.
- Trouver les poids optimaux qui minimisent une fonction de perte (ou coût) décrivant la différence entre les prédictions du réseau et les valeurs attendues.
- Les valeurs initiales de w et b sont choisies aléatoirement.

Le Perceptron MultiCouches (PMC)

Le problème de l'apprentissage dans les perceptrons multicouches est de **connaitre la contribution de chaque poids dans l'erreur globale du réseau**

L'algorithme de rétropropagation de l'erreur permet de faire cela.

1. Propagation Avant : Les données sont transmises à travers le réseau en utilisant la propagation avant, et les prédictions sont générées.
2. Calcul de la Perte : La fonction de perte est calculée en comparant les prédictions du réseau aux valeurs attendues.
3. Rétropropagation : Le gradient de la perte par rapport aux poids est calculé à l'aide des dérivées partielles. Cela permet de déterminer comment chaque poids affecte la perte.
4. Mise à Jour des Poids : Les poids sont ajustés en fonction du gradient de la perte. Les techniques d'optimisation, telles que la descente de gradient, sont couramment utilisées pour déterminer la taille des mises à jour des poids.

Conditions

Il faut une fonction d'activation dérivable car on a besoin de la dérivé pour rétro-propager l'erreur.

Le Perceptron MultiCouches (PMC)

- **Apprentissage**

– Supervisé

– Basé sur la correction des erreurs de classement des données

  • Erreurs produites au niveau de chaque nœud par la fonction d'activation f

– Correction

  • Ne pouvant constater l'erreur qu'à la sortie du réseau

  ➔ On remonte le réseau par la fin en essayant de trouver les responsables des erreurs pour les corriger à la hauteur de leurs erreurs

  ➔ Ensuite, cette correction est répercutée sur les précédents

  • Les erreurs dépendent des poids et des entrées

  ➔ Erreur = f(ω,E) où ω sont les poids et E les entrées

  • On corrige les poids ω en chaque nœud de manière à ce que à la prochaine rentrée des données, l'erreur générale soit moindre

Le Perceptron MultiCouches (PMC)

- **Apprentissage : en d'autres termes**

– Pour la couche de sortie, on peut appliquer l'apprentissage du Perceptron car on est près de la sortie et l'on sait exactement ce que l'on veut obtenir en sortie : on peut corriger en conséquence

– Pour les autres couches :

- Problème : car pas de contact direct avec la solution (sortie)
- Solution

➔On fait une estimation de l'erreur effectuée pour chaque neurone de la couche cachée

➔A partir de cette estimation, on corrige les poids

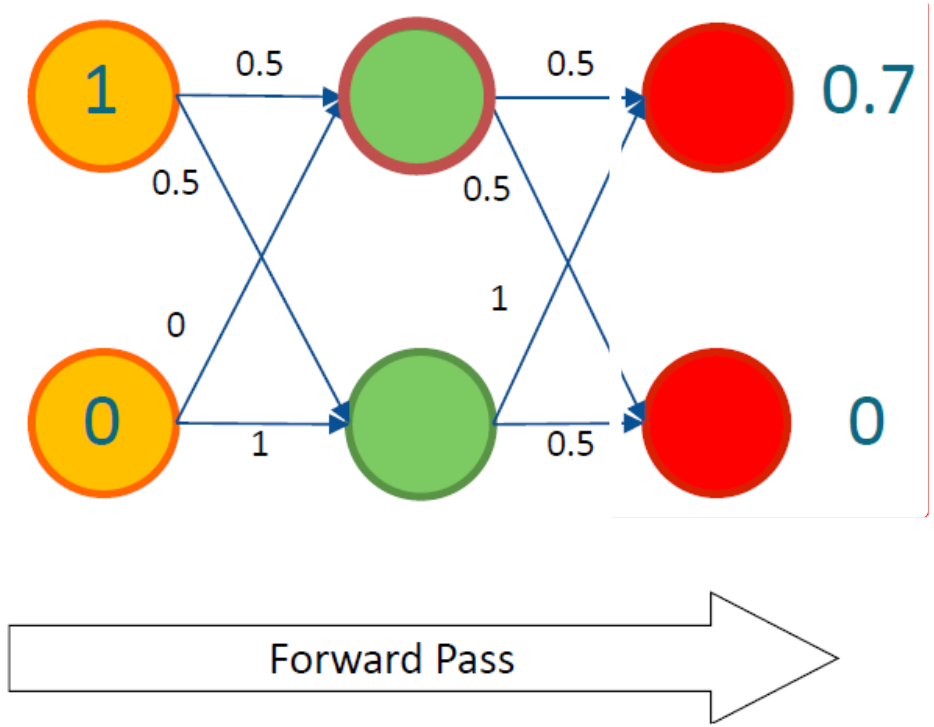
➔Cette solution s'appelle » **rétropropagation du gradient**

Le Perceptron MultiCouches (PMC)

Le processus suivi par un ANN pour évaluer ses résultats concernant un cas est appelé **le passage en avant (forward pass)**. Considérez ce réseau simple (sigmoïde et biais = 1).

A partir des entrées dont les sorties sont connues, pour chaque neurone :

- Calculer la somme pondérée de leurs entrées
- Ajoutez le biais
- Appliquer la fonction d'activation et générer la valeur de sortie
- Complétez la couche et passez à la suivante



Exemples de fonctions de perte

- Mean Square Method (MSM): $MSM = \frac{1}{n}.\sum_{i,j}(\widehat{y_{ij}(\theta)} - y_{ij})^2$
- Mean Absolute Error (MAE): $MAE = \frac{1}{n}.\sum_{i,j}|\widehat{y_{ij}(\theta)} - y_{ij}|$
- Cross Entropy Loss (for binary classifications):

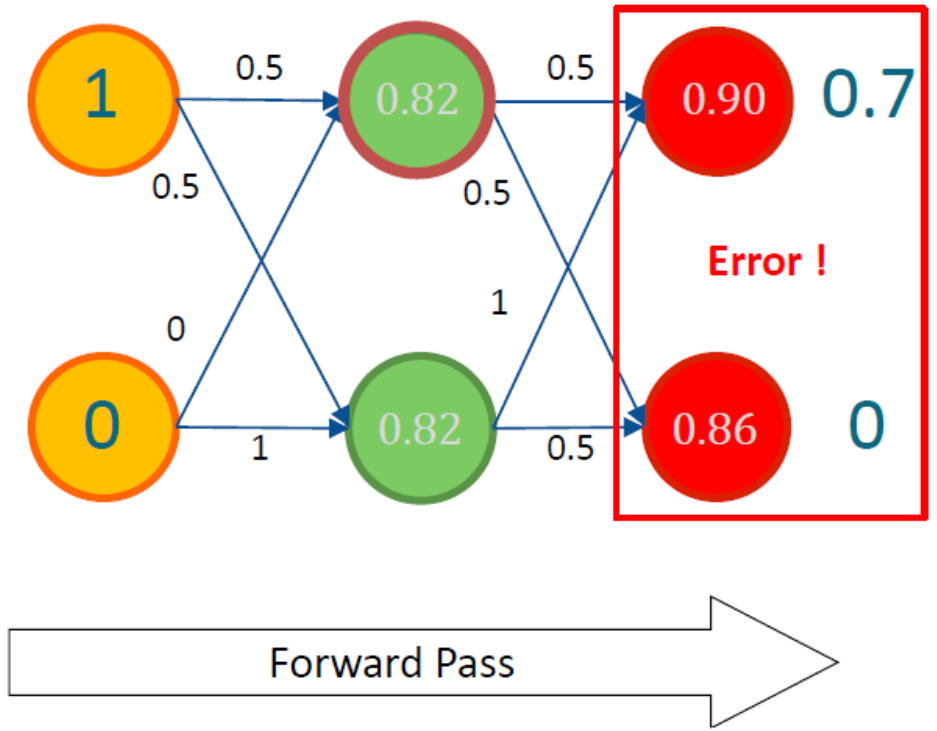
$$CEL = \frac{1}{n}.\sum_{i,j} -y_{ij}\log(\widehat{y_{ij}(\theta)}) + (1 - y_{ij})\log(1 - \widehat{y_{ij}(\theta)})$$

Le Perceptron MultiCouches (PMC)

Le processus suivi par un ANN pour évaluer ses résultats concernant un cas est appelé **le passage en avant (forward pass)**. Considérez ce réseau simple (sigmoïde et biais = 1).

A partir des entrées dont les sorties sont connues, pour chaque neurone :

- Calculer la somme pondérée de leurs entrées
- Ajoutez le biais
- Appliquer la fonction d'activation et générer la valeur de sortie
- Complétez la couche et passez à la suivante



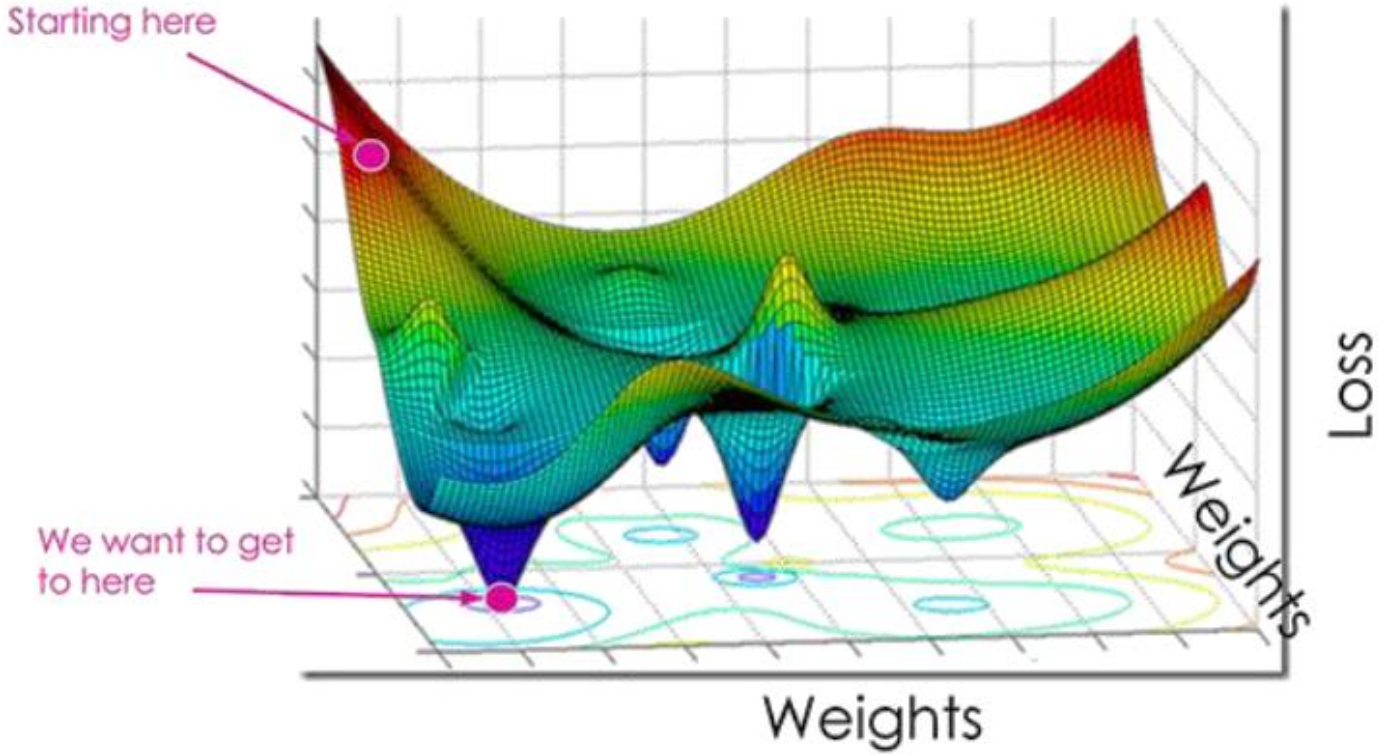
Exemples de fonctions de perte

- Mean Square Method (MSM): $MSM = \frac{1}{n}.\sum_{i,j}(\widehat{y_{ij}(\theta)} - y_{ij})^2$
- Mean Absolute Error (MAE): $MAE = \frac{1}{n}.\sum_{i,j}|\widehat{y_{ij}(\theta)} - y_{ij}|$
- Cross Entropy Loss (for binary classifications):

$$CEL = \frac{1}{n}.\sum_{i,j} -y_{ij}\log(\widehat{y_{ij}(\theta)}) + (1 - y_{ij})\log(1 - \widehat{y_{ij}(\theta)})$$

Le Perceptron MultiCouches (PMC)

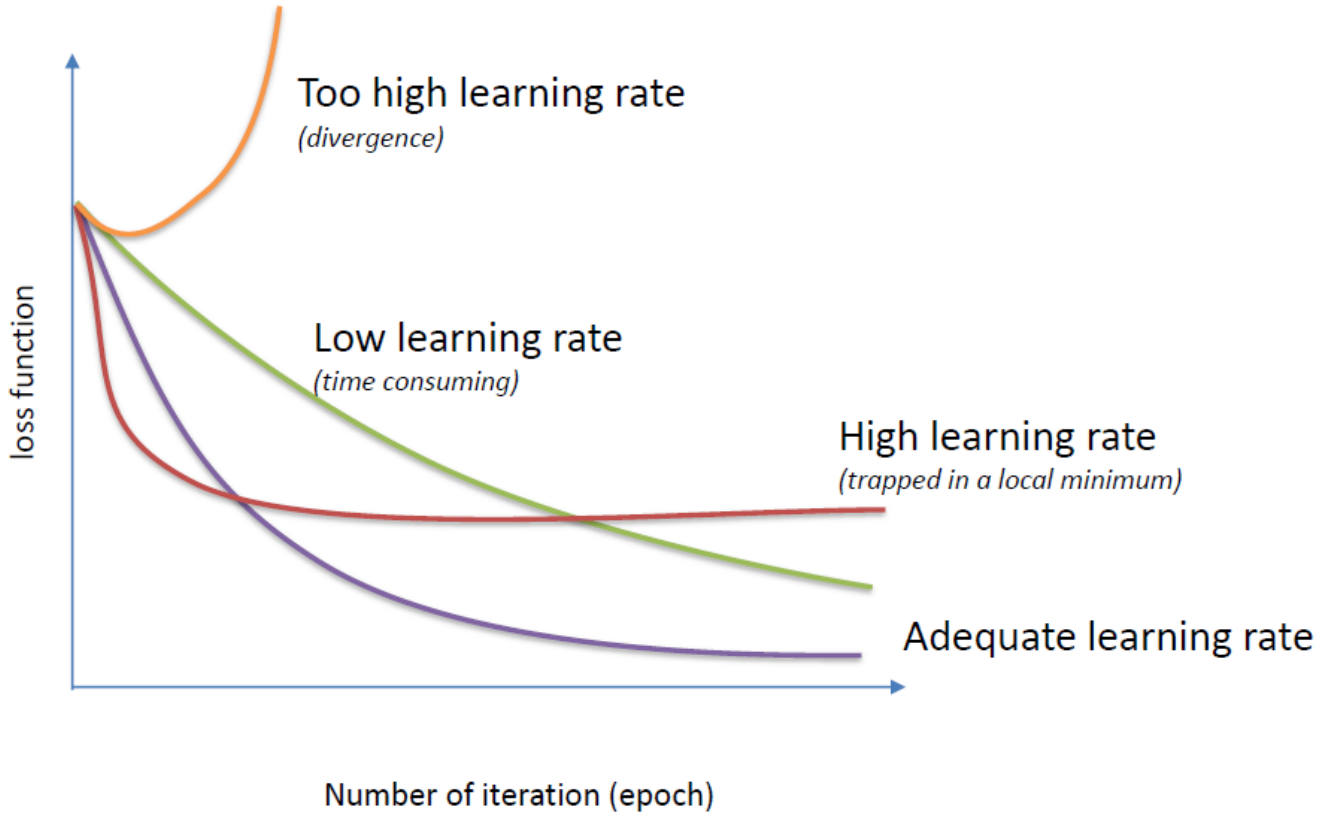
- Pour trouver la meilleure configuration du réseau, l'objectif est de trouver le minimum de la fonction de perte dans l'espace de domaine de tous les paramètres $\theta=(W,b)$).



- Comme aucune expression mathématique n'est disponible, les approches numériques sont obligatoires. (gradient descent algorithms)

Le Perceptron MultiCouches (PMC)

Tracer l'évolution de la perte pendant l'entraînement est une bonne façon d'ajuster son paramètre principal, le taux d'apprentissage λ.



Le Perceptron MultiCouches (PMC)

- **Algorithme d'apprentissage : 2 mouvements**

– A partir de l'erreur connue, calculée à la sortie :

1. Mouvement de droite à gauche

– On établit la "responsabilité" ou la contribution de chaque neurone des couches cachées dans la production de cette erreur

2. Mouvement de gauche à droite

– On corrige les poids à partir de l'estimation dans chaque neurone, sans utiliser de nouvelles données

– L'entrée de nouvelles données se fera après, une fois les poids corrigés

Le Perceptron MultiCouches (PMC)

- **Comment corriger ?**

– On se base sur la fonction générale de l'erreur f(ω,E)

- On suit la pente de la fonction

➔pour orienter, à chaque entrée, la correction des poids dans le bon sens, conduisant à une erreur encore plus petite de f, jusqu'à atteindre le minimum

- On utilise un "coefficient de correction" μ

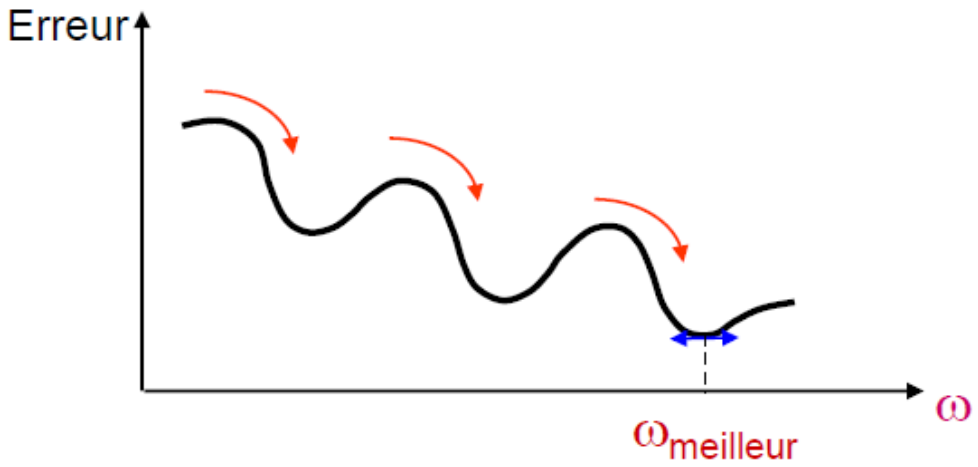
➔qui permet de ne pas corriger abusivement :

» si petit, risque de stagner,

» si grand : risque de ne pas converger, car modifications seront trop fortes

Le Perceptron MultiCouches (PMC)

- **But de la correction : recherche du minimum global du réseau**
  - –Les sauts et les bosses correspondent aux améliorations successives faites à chaque étape sur l'ensemble des données
  - –À chaque fois, on essaie de se sortir d'un minimum local pour atteindre le meilleur vecteur poids $\omega_{meilleur}$



Le Perceptron MultiCouches (PMC)

- **Réduction de l'erreur**

–On ne sait pas réduire l'erreur globalement pour le réseau

- Donc, on le fait neurone par neurone

–Principe :

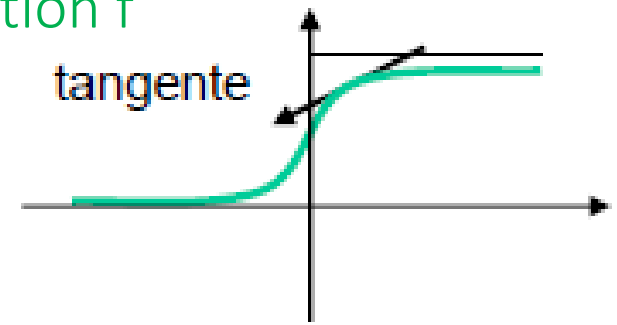
1. Examiner la réponse de la fonction d'activation du neurone
2. Estimer l'influence de cette erreur sur la correction des poids en entrée,

➔Loin de 0, i.e. dans une zone stable, alors correction faible

➔Près de 0, zone instable, faut en sortir au plus vite, en corrigeant drastiquement

»Cela correspond à suivre la pente (tg) de la fonction f

3. On corrige l'erreur en la pondérant par la pente



Le Perceptron MultiCouches (PMC)

- **Autre explication : réduction de l'erreur au niveau d'un neurone**

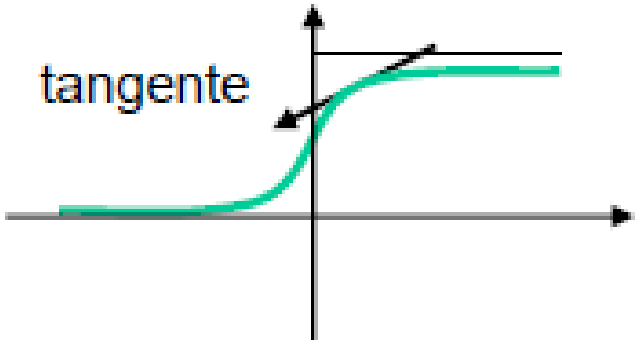
1.On pondère cette erreur d'après la réponse de sa fonction d'activation (sigmoïde) qui se traduit ainsi :

➔Si on est proche de 0 ou de 1, on a bien appris, la correction doit rester faible

➔Si la réponse n'est pas franche (proche de 0.5), le neurone hésite, on va donc le pousser vers une décision (0 ou 1)

»Pour ce faire, on pondère par la pente de la fonction f (tangente)

2.On corrige les poids en fonction de cette erreur pondérée, et des contributions des neurones en entrée



Le Perceptron MultiCouches (PMC)

- Correction des poids

$\Delta\omega_{jk} = \boldsymbol{\mu} \times \boldsymbol{s_j} \times \boldsymbol{err}$ où $\boldsymbol{err} = \boldsymbol{\delta_k} = \boldsymbol{f'(s_k)} \times \boldsymbol{E_k}$

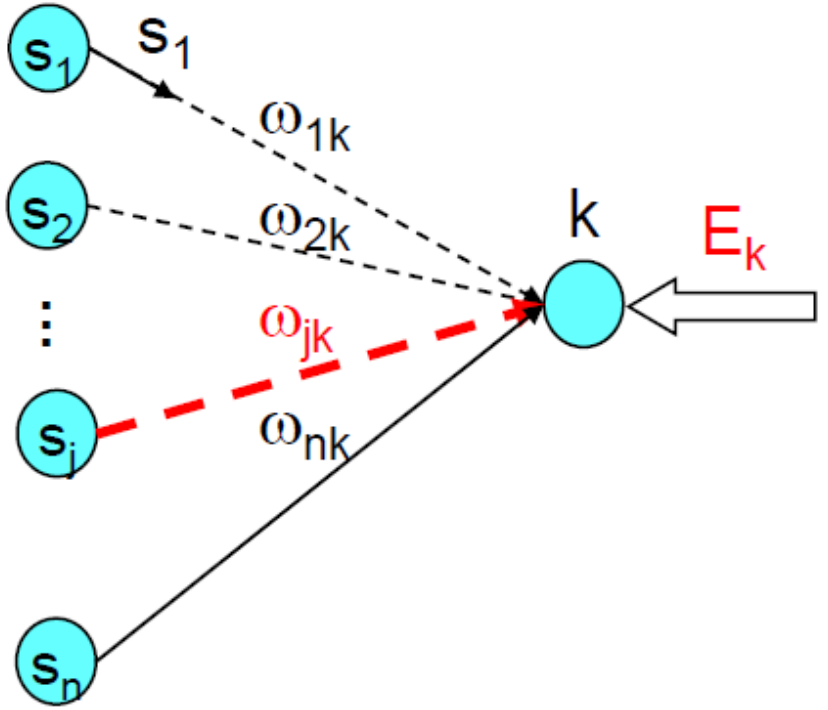
– $\boldsymbol{\mu}$: coefficient de correction

– $\boldsymbol{s_j}$: sortie du neurone j, en entrée de k, la correction est proportion à son importance

– $\boldsymbol{err}$: erreur pondérée

– $\boldsymbol{f'(s_k)}$: dérivée de la fonction du neurone

– $\boldsymbol{E_k}$: erreur commise par le neurone, la correction sera proportionnelle à son importance



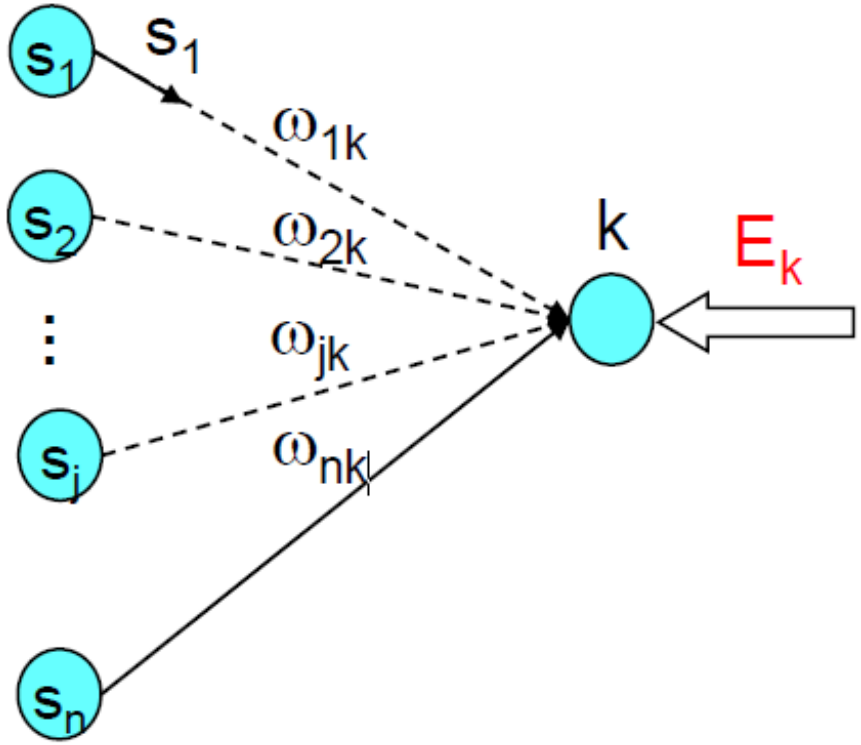
Le Perceptron MultiCouches (PMC)

- Calcul de l'erreur E_k :

–Premier cas : E_k : est le neurone de sortie

E_k = sortie désirée – s_k

- L'erreur est visible : égale à la différence entre la valeur désirée et la valeur obtenue (réelle)



Le Perceptron MultiCouches (PMC)

- **Calcul de l'erreur E_k :**

– Cas général : pour un neurone interne k

– L'erreur du neurone k s'est propagée à tous les neurones en aval (n)

– E_k = somme des erreurs provenant des neurones en aval, pondérées par :

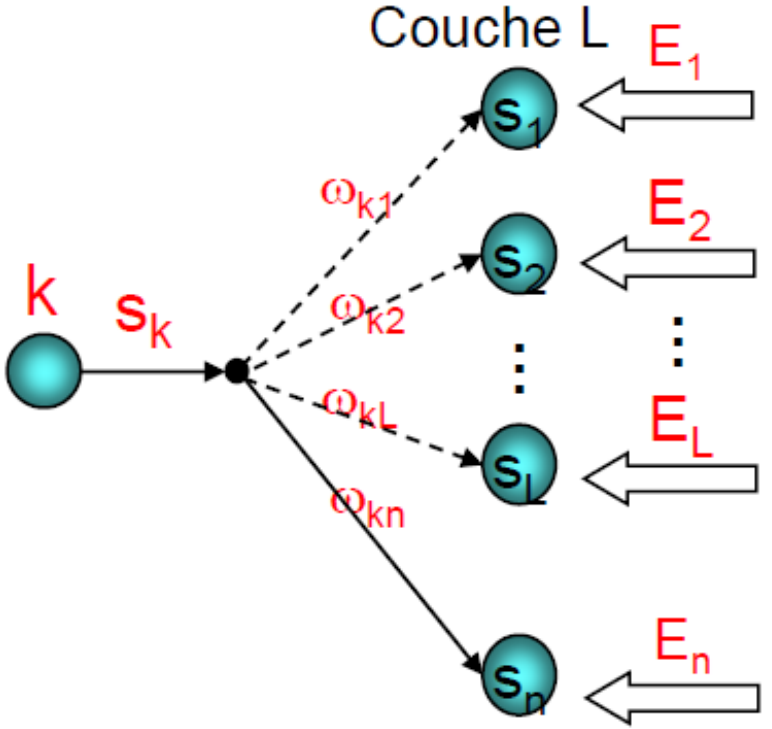
- les poids reliant le neurone k à chacun des neurones en aval

➔accentuant la correction, en réponse à l'importance de l'erreur qu'il aura produite en aval

- la pente de sa fonction d'activation

➔pour corriger cette erreur dans le bon sens

$$\delta_k = f'(s_k).\Sigma_L \omega_{kL}.\delta_L$$



Le Perceptron MultiCouches (PMC)

- L'algorithme

–Étape 1 : calcul de l'erreur pour la couche de sortie k :

$\boldsymbol{\delta_k}$ = f'($\boldsymbol{s_k}$). $\boldsymbol{E_k}$, $\boldsymbol{E_k}$ =(sortie désirée -sortie réelle) $\forall k \in$ couche de sortie

–Étape 2 : calcul de l'erreur pour tous les neurones des couches cachées j :

$\boldsymbol{\delta_j}$ = f'($\boldsymbol{s_j}$).Σ_L $\boldsymbol{\omega_{jL}}$. $\boldsymbol{\delta_L}$, $\forall L \in$ couche suivante

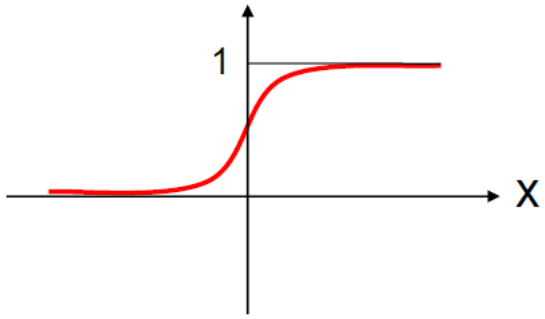
–Étape 3 : modification des poids

$\Delta\omega_{jL} = \mu \times \boldsymbol{s_j} \times \boldsymbol{\delta_L}$

$\omega_{jL}(\mathrm{t}+1) = \omega_{jL}(\mathrm{t}) + \Delta\omega_{jL}$

Le Perceptron MultiCouches (PMC)

- Pourquoi utilise-t-on la fonction sigmoïde ?



$$f(x)=\frac{1}{1+e^{-x}}$$

$$f'(x)=f(x).(1-f(x))$$

- Sa dérivée :

– L'intérêt de la sigmoïde est que la dérivée se calcule directement à partir de la fonction elle même

• Or, on a déjà calculé la valeur de la fonction lors de la propagation

Le Perceptron MultiCouches (PMC)

- **Comment choisir le pas de modification des poids (μ) ?**

–Dans l'algorithme standard

- La valeur du μ est entre 0 et 1, mais il n y a pas de règle pour le choix
- Le choix est empirique (expérimental)
- Pour résoudre les problèmes de "zigzag", on peut cependant proposer une heuristique :

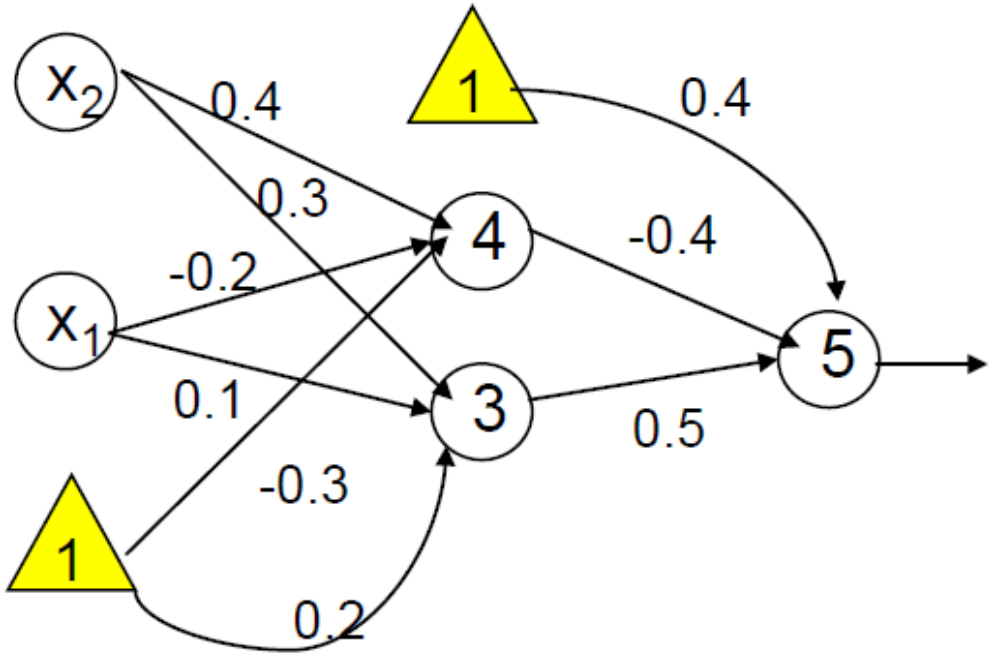
➔Soit on commence avec un μ grand et pendant l'apprentissage, on diminue cette valeur petit à petit

➔Soit on prend 2 valeurs (une petite et une grande) qu'on alterne durant l'apprentissage

Le Perceptron MultiCouches (PMC)

Exemple :

- Soit le réseau à 2 entrées x1 et x2 avec une couche cachée (composée de 2 neurones 3 et 4) et une seule sortie (5).
- Chaque couche contient un biais (neurone permettant d'intégrer le calcul du seuil S dans la somme pondérée, évitant de le faire manuellement).
- Ce neurone a toujours comme entrée 1



Le Perceptron MultiCouches (PMC)

- Problème à résoudre : XOR

–Cela veut dire qu'en prenant, par exemple, comme vecteur d'entrée x =(1,1), le résultat doit être égal à 0

x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Le Perceptron MultiCouches (PMC)

- Problème à résoudre : XOR

–Cela veut dire qu'en prenant, par exemple, comme vecteur d'entrée x =(1,1), le résultat doit être égal à 0 (voir détail du calcul ci-dessous)

x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3		
4		
5		

Résultat obtenu

Le Perceptron MultiCouches (PMC)

- Problème à résoudre : XOR

–Cela veut dire qu'en prenant, par exemple, comme vecteur d'entrée x =(1,1), le résultat doit être égal à 0 (voir détail du calcul ci-dessous)

x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3	$0.2+ 0.1\times1+0.3 \times1=0.6$	$1/(1+e^{-0.6}) \approx 0.65$
4	$-0.3+ -0.2\times1+0.4 \times1= -0.1$	$1/(1+e^{0.1}) \approx 0.48$
5	$0.4+ 0.5\times0.65-0.4 \times0.48= 0.53$	$1/(1+e^{-0.53}) \approx 0.63$

Résultat obtenu

Le Perceptron MultiCouches (PMC)

- **Correction (suite de l'apprentissage)**

– On va essayer par correction de rabaisser la valeur de sortie 0.63 pour l'entrée (1,1)

– On commence à appliquer l'algorithme :

- Pour chaque couche, on calcule le $\boldsymbol{\delta_j}$
- Étape 1 1er cas : on est sur le neurone 5 de la couche de sortie

- On calcule $\delta_5 = (0-0.63)\times 0.63\times(1-0.63) = -0.147$
- On calcule $\Delta\omega_{05}$, $\Delta\omega_{35}$, $\Delta\omega_{45}$ à partir de δ_5 en fixant la valeur de $\mu=1$
- $\Delta\omega_{05} = -0.147 \times 1 \approx -0.147$
- $\Delta\omega_{35} = -0.147 \times 0.65 \approx -0.1$
- $\Delta w_{45} = 0.48 \times -0.147 \approx -0.07$

Le Perceptron MultiCouches (PMC)

Correction (suite de l'apprentissage)

Étape 2 2 ème cas : on est sur la couche cachée, on commence par 4 puis 3

- $\delta_4 = y_4 \times (1- y_4) \times \delta_5 \times \omega_{45} = 0.48 \times (1-0.48) \times -0.147 \times - 0.4 \approx 0.015$
- $\Delta\omega_{14}$, $\Delta\omega_{24}$, $\Delta\omega_{04}$ à partir de δ_4 en fixant $\mu=1$
  - $\Delta\omega_{14}= 0.015 \times 1 \approx 0.015$
  - $\Delta\omega_{24}= 0.015 \times 1 \approx 0.015$
  - $\Delta\omega_{04}= 0.015 \times 1 \approx 0.015$
- $\delta_3 = y_3 \times (1- y_3) \times \delta_5 \times \omega_{35} = 0.65 \times (1-0.65) \times -0.147 \times 0.5 \approx -0.017$
- $\Delta\omega_{13}$, $\Delta\omega_{23}$, $\Delta\omega_{03}$ à partir de δ_3 en fixant la valeur de $\mu=1$
  - $\Delta\omega_{13}= -0.017 \times 1 \approx -0.017$
  - $\Delta\omega_{23}= -0.017 \times 1 \approx -0.017$
  - $\Delta\omega_{03}= -0.017 \times 1 \approx -0.017$

Le Perceptron MultiCouches (PMC)

- Correction (suite de l'apprentissage)
- Étape 3 Pour chaque connexion ω_{jL} on ajuste les poids selon la formule

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \Delta\omega_{ij}$$

$\omega_{05} = \omega_{05} + \Delta\omega_{05} = 0.4 - 0.147 \approx 0.25$

$\omega_{35} = \omega_{35} + \Delta\omega_{35} = 0.5 - 0.1 \approx 0.4$

$\omega_{45} = \omega_{45} + \Delta\omega_{45} = -0.4 - 0.07 \approx -0.47$

$\omega_{03} = \omega_{03} + \Delta\omega_{03} = 0.2 - 0.017 \approx 0.183$

$\omega_{13} = \omega_{13} + \Delta\omega_{13} = 0.1 - 0.017 \approx 0.083$

$\omega_{23} = \omega_{23} + \Delta\omega_{23} = 0.3 - 0.17 \approx 0.283$

$\omega_{04} = \omega_{04} + \Delta\omega_{04} = -0.3 + 0.015 \approx -0.285$

$\omega_{14} = \omega_{14} + \Delta\omega_{14} = -0.2 + 0.15 \approx -0.185$

$\omega_{24} = \omega_{24} + \Delta\omega_{24} = 0.4 + 0.15 \approx 0.415$

Le Perceptron MultiCouches (PMC)

- Correction (suite de l'apprentissage)

–Ensuite, avec les poids corrigés, on repasse le même vecteur (1,1) à travers le réseau, on peut remarquer une baisse de la valeur de sortie
–On recommence en utilisant le même vecteur ou un autre : (0,0) ou (1,0) ou (0,1)

x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone formel j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3		
4		
5		

Résultat obtenu

Le Perceptron MultiCouches (PMC)

- Correction (suite de l'apprentissage)

–Ensuite, avec les poids corrigés, on repasse le même vecteur (1,1) à travers le réseau, on peut remarquer une baisse de la valeur de sortie
–On recommence en utilisant le même vecteur ou un autre : (0,0) ou (1,0) ou (0,1)

x1	x2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Résultat attendu

Neurone formel j	Somme pondérée σ_j	Sortie $y_j=f(\sigma_j)$ où f=sigmoïde
3	0.183+ 0.083×1+0.283 ×1=0.55	$1/(1+e^{-0.55}) \approx 0.63$
4	-0.285+ -0.185×1+0.415 ×1= -0.055	$1/(1+e^{0.055}) \approx 0.51$
5	0.25+ 0.4×0.63-0.47 ×0.51= 0.26	$1/(1+e^{-0.26}) \approx 0.56$

Résultat obtenu

Le Perceptron MultiCouches (PMC)

Pour exemple, les algorithmes pré-entraînés de Google ou Facebook comptent en général entre 1000 et 2500 couches, telle est la complexité de ses algorithmes mais également de leur précision.

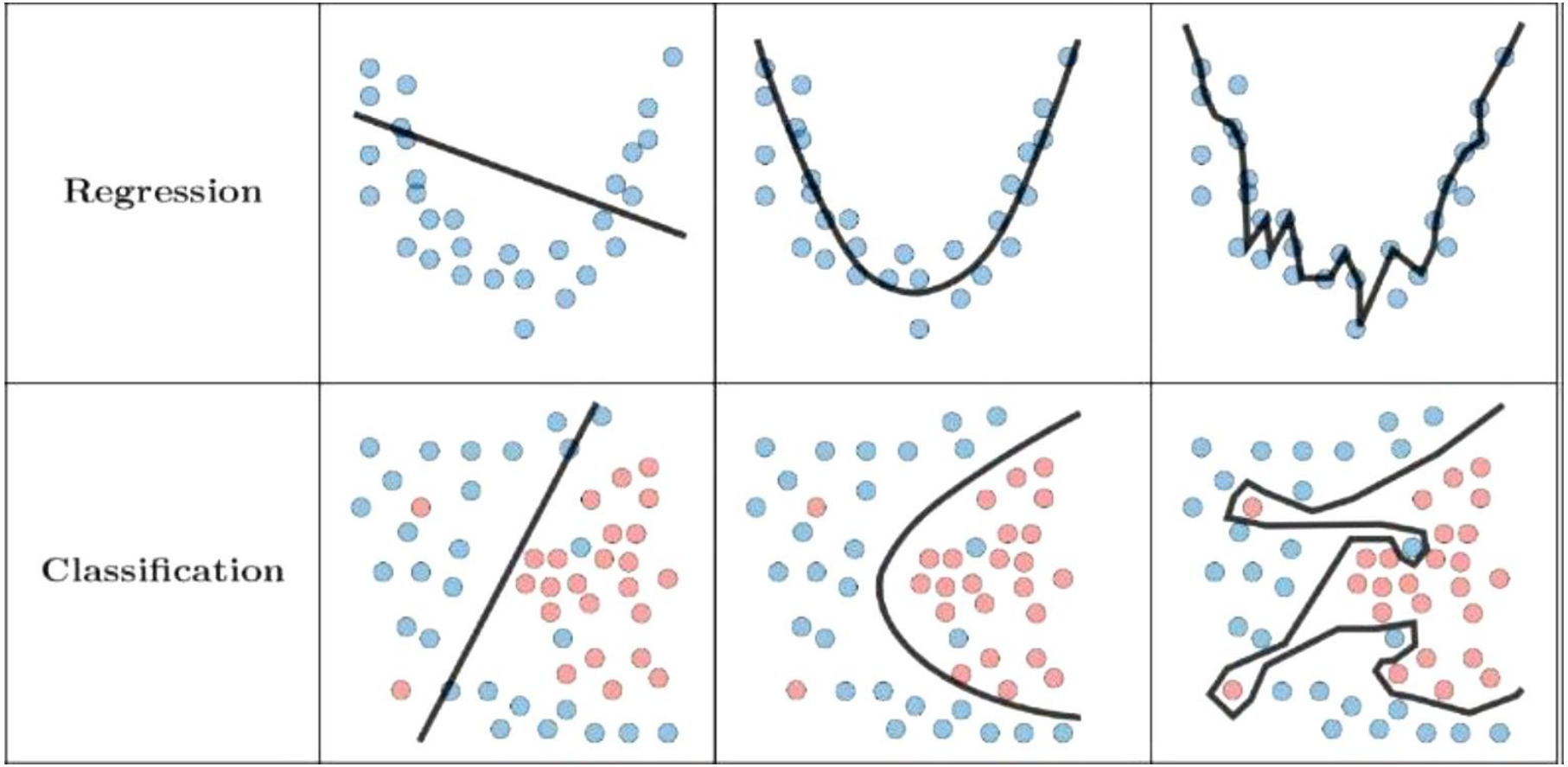
Pour les entraîner, vous allez avoir besoin de beaucoup de puissance calculatoire : sur votre ordinateur personnel par exemple, cela prendrait trop de temps d'entraîner votre algorithme. La contrainte de la quantité de données.

Le Perceptron MultiCouches (PMC)

- **Overfitting**

Le surapprentissage se produit lorsque la fonction de perte est bonne (très faible) sur l'ensemble d'apprentissage, mais n'est pas capable de généraliser ses prédictions à des exemples supplémentaires ou non vus.

Ce problème, dans le but de la régression, est très proche des problèmes d'approximation polynomiale. (selection of the right polynomial degree)

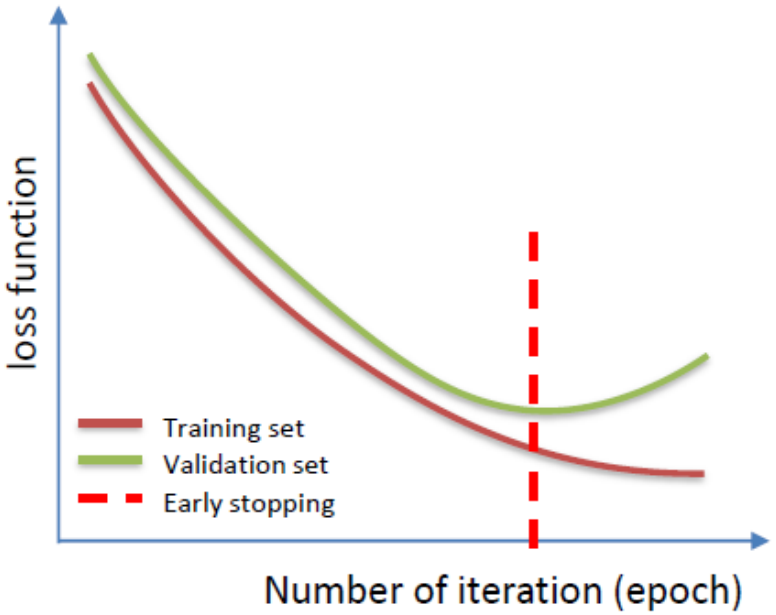


Le Perceptron MultiCouches (PMC)

- **Overfitting**

Pour éviter la situation de surapprentissage, deux solutions principales sont utilisées :

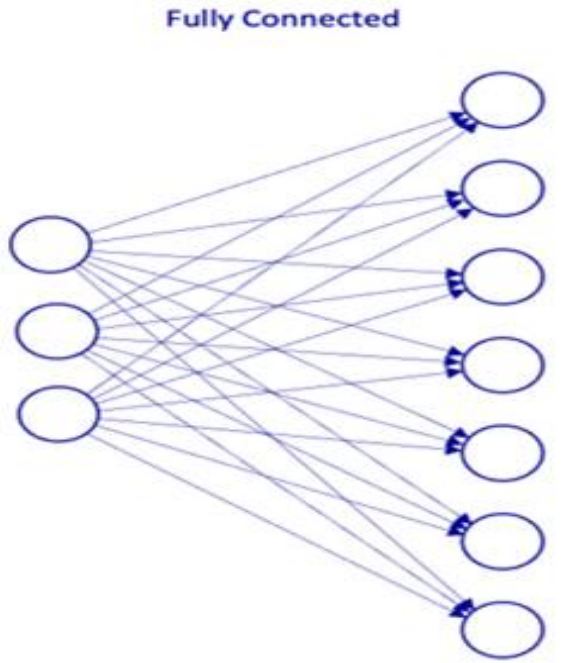
1) Arrêt précoce : Parmi les ensembles de données, certains sont conservés dans un ensemble de validation qui n'est pas utilisé par le réseau de neurones artificiels (ANN) pour sa phase d'apprentissage. La fonction de perte est calculée à la fois sur les ensembles d'entraînement et de validation. Si après plusieurs itérations la fonction de perte de l'ensemble de validation ne s'est pas améliorée, arrêtez l'entraînement.



Le Perceptron MultiCouches (PMC)

- **Overfitting**

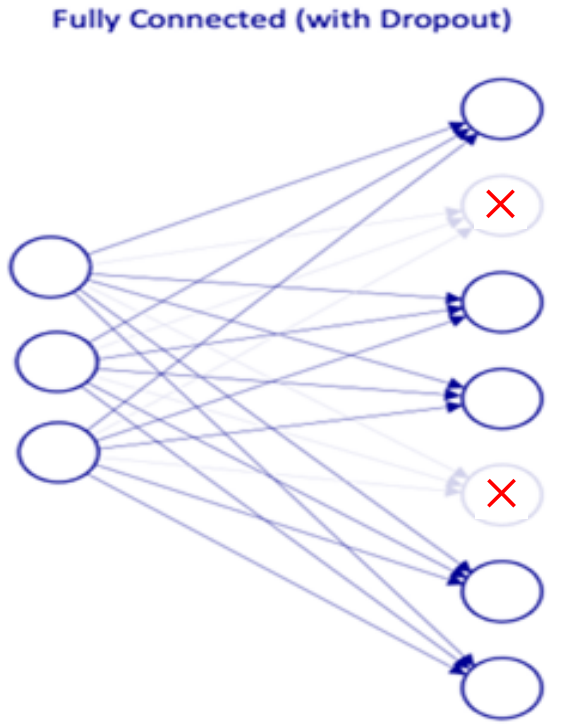
2) Dropout : Pour éviter que des neurones ne se spécialisent sur un exemple particulier, l'approche du dropout désactive aléatoirement certains neurones pendant l'entraînement.



Le dropout: "désactiver" aléatoirement un certain pourcentage de neurones dans une couche pendant chaque itération d'entraînement.

➔Lors d'une itération donnée, certains neurones ne contribuent pas à l'apprentissage et ne sont pas pris en compte dans la rétropropagation.

➔le modèle apprend à ne pas dépendre d'un ensemble spécifique de neurones pour faire des prédictions



Conclusion sur les réseaux de neurones

- Avantages
  - –Classifieurs universels offrant:
    - Rapidité d'exécution
    - Robustesse des solutions, résistance au bruit des données
    - Facilité de développement
- Inconvénients
  - –Le choix de l'architecture n’est pas évident
  - –Le temps d'apprentissage peut être long
  - –Présence de minima locaux de la fonction de coût

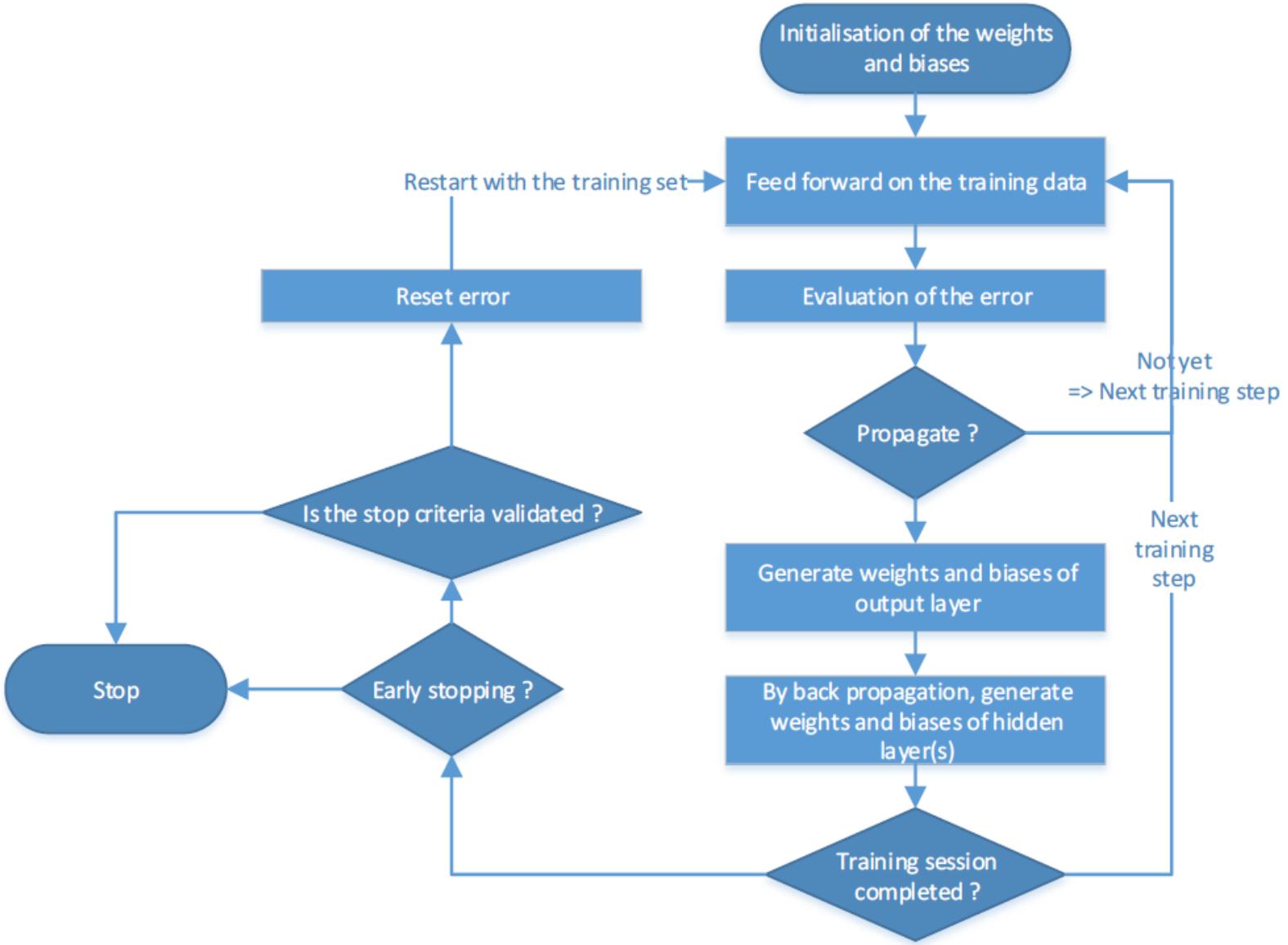
COMMENT DIVISER LA BASE DE DONNÉES ?

Pour entraîner un ANN, le tester et éviter qu'il ne soit surentraîné, trois ensembles de points sont nécessaires :

Où :

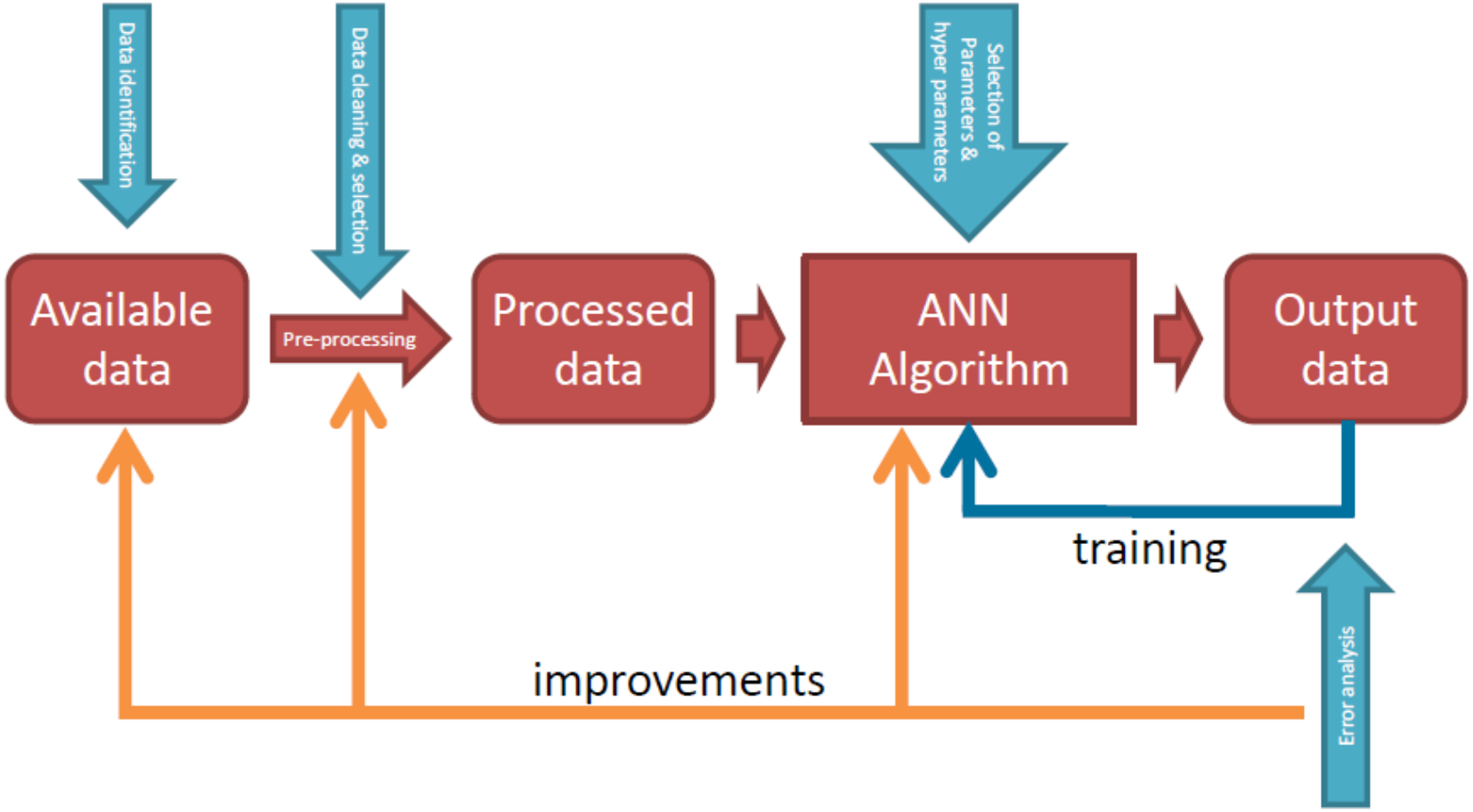
- L'ensemble d'apprentissage est utilisé pour entraîner le ANN (à des fins de rétropropagation)
- L'ensemble de validation, composé de cas non observés, est utilisé pour vérifier pendant l'apprentissage si le système est confronté à des problèmes de surajustement ou de surapprentissage.
- L'ensemble de test est utilisé lorsque l'apprentissage est terminé pour vérifier les performances du ANN entraîné. Essayez d'avoir un ensemble représentatif pour la validation finale (points bien distribués dans le domaine)
- Les distributions entre ces ensembles suivent généralement ces taux 80% / 15% / 5%

Algorithme complet du ANN



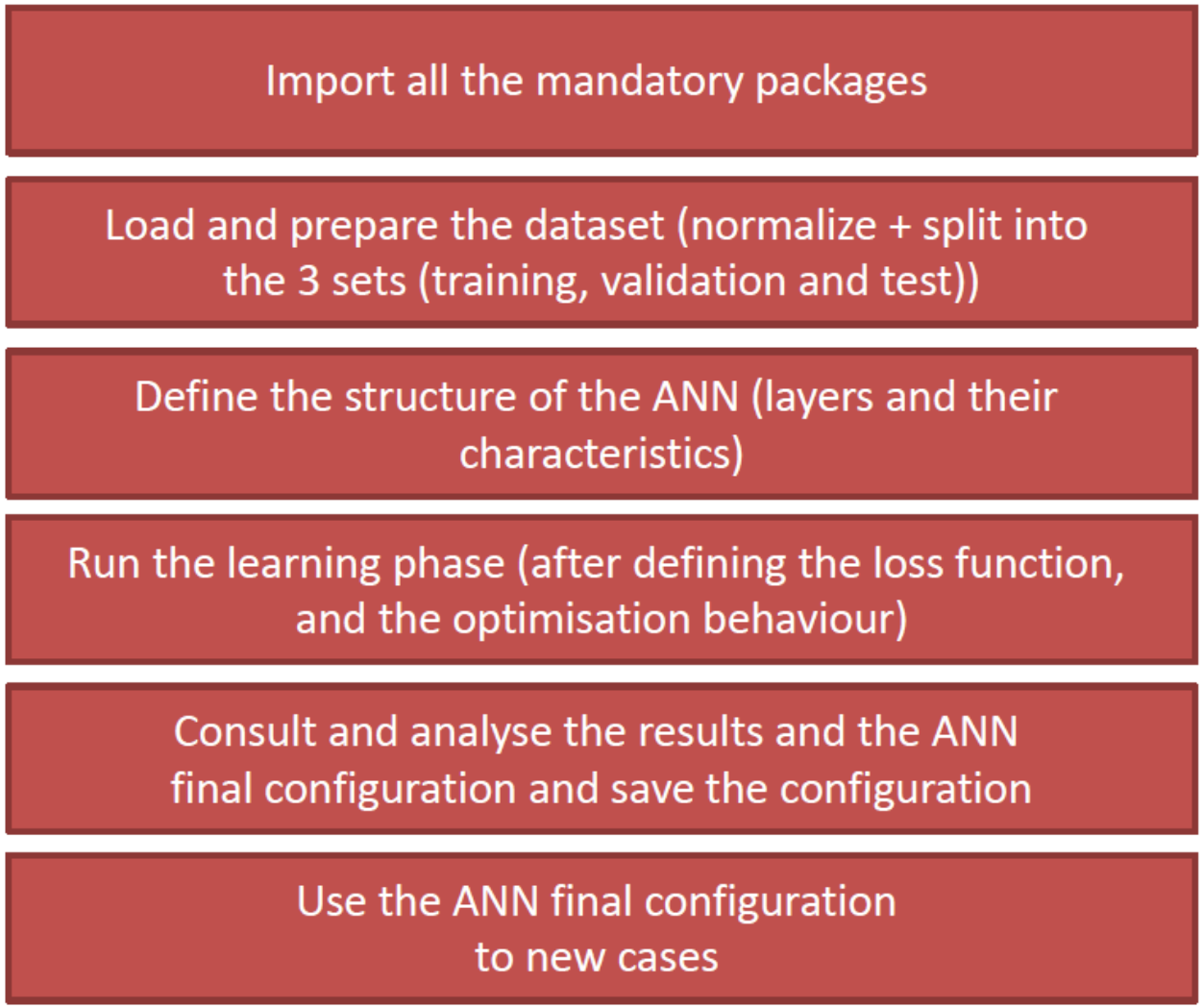
Algorithme complet du ANN

Vous devez prendre de nombreuses décisions pour construire et améliorer l'ANN en fonction de vos attentes et du type de données que vous souhaitez traiter.



Algorithme complet du ANN

La structure d'un ANN .



Atelier N°1

Classification of clothing images

But:

Être capable de concevoir et réaliser une AI permettant la classification des images de vêtements.

Travail à Faire:

- Démarrer le programme https://www.tensorflow.org/tutorials/keras/classification
- Expliquer ce programme
- Améliorer ce programme

Listing .1 Chargement du jeu de données Fashion MNIST depuis TensorFlow

```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Le chargement de l'ensemble de données renvoie quatre tableaux NumPy :

- Les tableaux train_images et train_labels sont l' ensemble d'apprentissage - les données que le modèle utilise pour apprendre.
- Le modèle est testé par rapport à l' ensemble de tests , aux tableaux test_images et test_labels .
- Les images sont des tableaux NumPy 28x28, avec des valeurs de pixels allant de 0 à 255.
  Les *étiquettes* sont un tableau d'entiers, allant de 0 à 9.

Chaque image est associée à une seule étiquette. Étant donné que les noms de classe ne sont pas inclus dans le jeu de données, stockez-les ici pour les utiliser ultérieurement lors du traçage des images :

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Regardons maintenant les données d'entraînement :

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Et voici les données de test :

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Le flux de travail sera le suivant : nous allons d'abord alimenter le réseau de neurones avec les données d'entraînement, ***train_images*** et ***train_labels***. Le réseau apprendra alors à associer images et étiquettes. Enfin, nous demanderons au réseau de produire des prédictions pour ***test_images***, et nous vérifierons si ces prédictions correspondent aux étiquettes de ***test_labels***..

Listing.2 Prétraiter les données

Les données doivent être prétraitées avant de former le réseau. Si vous inspectez la première image de l'ensemble d'apprentissage, vous verrez que les valeurs de pixel se situent dans la plage de 0 à 255 :

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

Mettez ces valeurs à l'échelle dans une plage de 0 à 1 avant de les alimenter au modèle de réseau neuronal. Pour ce faire, divisez les valeurs par 255. Il est important que l' ensemble d'apprentissage et l' ensemble de test soient prétraités de la même manière :

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Listing.3 L'architecture du réseau

Le bloc de construction de base d'un réseau de neurones est **la couche** . Les couches extraient des représentations des données qui y sont introduites. Espérons que ces représentations sont significatives pour le problème à résoudre.

L'essentiel de l'apprentissage en profondeur consiste à enchaîner des couches simples. La plupart des couches, telles que ***tf.keras.layers.Dense*** , ont des paramètres appris lors de la formation.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

La première couche de ce réseau, ***tf.keras.layers.Flatten*** , transforme le format des images d'un tableau bidimensionnel (de 28 par 28 pixels) en un tableau unidimensionnel (de 28 * 28 = 784 pixels). Considérez ce calque comme désempilant des rangées de pixels dans l'image et en les alignant. Cette couche n'a aucun paramètre à apprendre ; il ne fait que reformater les données.

Une fois les pixels aplatis, le réseau se compose d'une séquence de deux couches ***tf.keras.layers.Dense*** . Ce sont des couches neuronales densément connectées ou entièrement connectées. La première couche Dense compte 128 nœuds (ou neurones). La deuxième (et dernière) couche renvoie un tableau logits d'une longueur de 10. Chaque nœud contient un score indiquant que l'image actuelle appartient à l'une des 10 classes.

Listing .4 L'étape de compilation

Avant que le modèle ne soit prêt pour l'entraînement, il a besoin de quelques réglages supplémentaires. Ceux-ci sont ajoutés lors de l'étape de compilation du modèle :

- Fonction de perte : mesure la précision du modèle pendant l'entraînement. Vous souhaitez minimiser cette fonction pour "orienter" le modèle dans la bonne direction.
- Optimiseur : c'est ainsi que le modèle est mis à jour en fonction des données qu'il voit et de sa fonction de perte.
- Métriques : utilisées pour surveiller les étapes de formation et de test. L'exemple suivant utilise precision , la fraction des images qui sont correctement classées.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Listing .5 L'étape d’entraînement du modèle

L'entraînement du modèle de réseau neuronal nécessite les étapes suivantes :
Transférez les données de formation au modèle. Dans cet exemple, les données d'apprentissage se trouvent dans les tableaux train_images et train_labels .
Le modèle apprend à associer des images et des étiquettes.
Vous demandez au modèle de faire des prédictions sur un jeu de test — dans cet exemple, le tableau test_images .
Vérifiez que les prédictions correspondent aux étiquettes du tableau test_labels .

Nous sommes maintenant prêts à entraîner le réseau, ce qui se fait via un appel à la méthode d'ajustement du réseau fit. Nous adaptons le modèle à ses données d'entraînement :

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4986 - accuracy: 0.8253
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3751 - accuracy: 0.8651
```

Au fur et à mesure que le modèle s'entraîne, les métriques de perte et de précision sont affichées. Ce modèle atteint une précision d'environ 0,91 (ou 91%) sur les données d'apprentissage.

Listing .6 L'étape d’évaluation

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

Il s'avère que la précision sur l'ensemble de données de test est un peu inférieure à la précision sur l'ensemble de données d'apprentissage. Cet écart entre la précision de l'entraînement et la précision des tests représente un surapprentissage . Le surajustement se produit lorsqu'un modèle d'apprentissage automatique fonctionne moins bien sur de nouvelles entrées inédites que sur les données d'apprentissage. Un modèle surajusté "mémorise" le bruit et les détails dans l'ensemble de données d'apprentissage à un point tel qu'il a un impact négatif sur les performances du modèle sur les nouvelles données.