# Deep Learning Apprentissage

## Apprentissage neuronal profond 2IA, ENSIAS

Pr.Raddouane chiheb

Mme.hanaa EL Afia

# Partie 1: Multi-Layer Perceptron (MLP)

# Plan

- Introduction aux Réseaux de Neurones
- Le Perceptron Simple
- Le Perceptron Multicouche (MLP)
- Apprentissage et Optimisation
- Évaluation et Performances du MLP

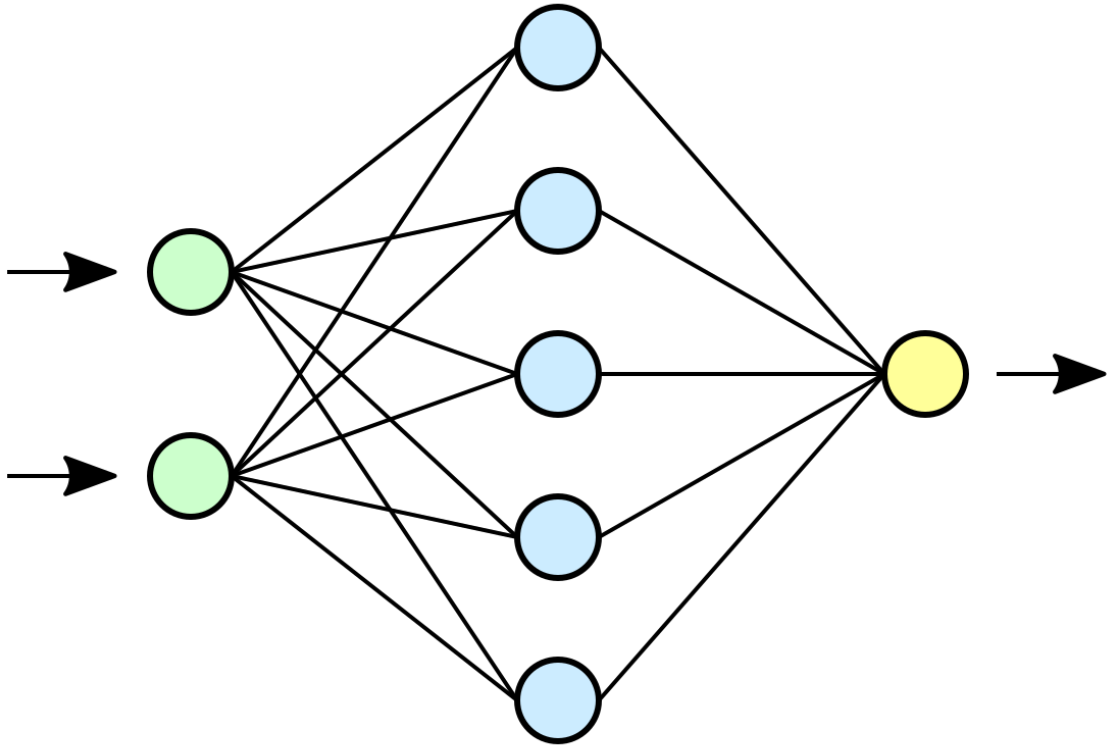
# Introduction aux Réseaux de Neurones

***Définition des Réseaux de Neurones:***

Un réseau de neurones artificiels est composé de neurones artificiels organisés en couches :

- **Couche d'entrée** : reçoit les données initiales.
- **Couches cachées** : traitent les informations en appliquant des transformations linéaires et non linéaires.
- **Couche de sortie** : fournit le résultat final du modèle.

Chaque neurone effectue une combinaison linéaire de ses entrées, pondérée par des poids, à laquelle s'ajoute un biais. Cette somme est ensuite passée à travers une fonction d'activation pour introduire de la non-linéarité, permettant au réseau de modéliser des relations complexes.



## *Différences entre les Réseaux Biologiques et Artificiels*

Bien que les RNA s'inspirent des réseaux neuronaux biologiques, plusieurs différences notables existent :

- **Complexité** : Les réseaux biologiques sont immensément plus complexes, avec des milliards de neurones interconnectés, tandis que les RNA sont constitués de quelques centaines à des millions de neurones artificiels.

- **Fonctionnement** : Les neurones biologiques communiquent via des signaux électrochimiques, alors que les neurones artificiels utilisent des opérations mathématiques pour traiter les informations.

- **Apprentissage** : Les réseaux biologiques apprennent de manière adaptative et continue, tandis que les RNA nécessitent des processus d'entraînement spécifiques avec des ensembles de données étiquetées.

## *Applications et Enjeux des Réseaux de Neurones:*

Les RNA ont révolutionné de nombreux domaines grâce à leur capacité à modéliser des relations complexes :

- **Reconnaissance d'images** : Identification d'objets, de visages et de scènes.
- **Traitement du langage naturel** : Traduction automatique, analyse de sentiments.
- **Santé** : Diagnostic assisté par ordinateur, analyse d'images médicales.
- **Finance** : Prévision des marchés, détection de fraudes.

Cependant, leur utilisation soulève des enjeux importants :

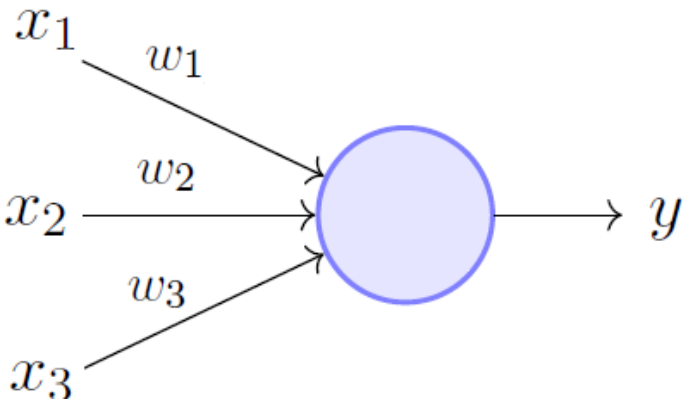
- **Interprétabilité** : Les modèles complexes sont souvent perçus comme des "boîtes noires", rendant difficile l'explication de leurs décisions.
- **Dépendance aux données** : Les RNA nécessitent de grandes quantités de données pour un entraînement efficace.
- **Consommation énergétique** : L'entraînement de modèles de grande taille peut être énergivore.

# Le Perceptron Simple

## *Structure d'un neurone artificiel*

Le perceptron simple est un modèle de neurone artificiel qui classe une entrée en deux catégories. Voici la structure de ce neurone.

- $\boldsymbol{Entrées}$ $(\boldsymbol{x})$ : Données introduites dans le neurone, représentant les caractéristiques de l'exemple à classer.
- $\boldsymbol{Poids}$ $(\boldsymbol{w})$ : chaque entrée est associée à un poids qui détermine l'importance de cette entrée dans le calcul de la sortie. Les poids sont ajustés lors de l'entraînement pour minimiser l'erreur.
- $\boldsymbol{Biais}$ $(\boldsymbol{b})$ : Paramètre permettant d'ajuster le modèle




- **Somme pondérée** :Le neurone calcule la somme des entrées pondérées par leurs poids

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- $\boldsymbol{Fonction\ d'activation}$ $(\boldsymbol{f})$ : Introduit une non-linéarité et détermine si le neurone doit être activé ou non

## *Algorithme d'Apprentissage du Perceptron*

### Principe de Fonctionnement

1. **Initialisation** des poids et du biais.
2. **Propagation avant** : calcul de la prédiction.
3. **Mise à jour des poids** si la prédiction est incorrecte.
4. **Répéter** jusqu'à convergence ou atteindre un nombre maximal d'itérations.

## *Algorithme d'Apprentissage du Perceptron*

**Entrées :**

- $X = [x_1, x_2, \ldots\ldots, x_n]$: données d'entrée.
- Y= $[y_1, y_2, \ldots\ldots, y_m]$: étiquettes $y \in \{-1,1\}$
- $\eta$: taux d'apprentissage $(0 < \eta \leq 1)$.
- $N$ : nombre d'itérations.

**Initialisation :**

- Poids W= $[w_1, w_2, \ldots\ldots, w_n]$: initialisés à $0$ ou des valeurs aléatoires.
- Biais $b = 0$.

## Étapes de l'Algorithme:

**Pour** $t = 1$ à $N$ (**nombre d'itérations**) :

**Pour chaque** paire $(x_i, y_i)$dans $(X, Y)$ :

1. **Calcul de la sortie :**

$$z = W.x_i + b$$

$$\hat{y} = \begin{cases} 1 & si\ z \geq 0 \\ -1 & si\ z < 0 \end{cases}$$

2. **Mise à jour des poids et biais**
   - Si $\widehat{y_i} \neq y_i$(mauvaise prédiction) :

$$W = W + \eta \cdot y_i \cdot x_i$$
$$b = b + \eta \cdot yi$$

   - Sinon : ne rien changer.

3. **Vérifier la convergence** : si toutes les prédictions sont correctes, arrêter.

**Sortie :**

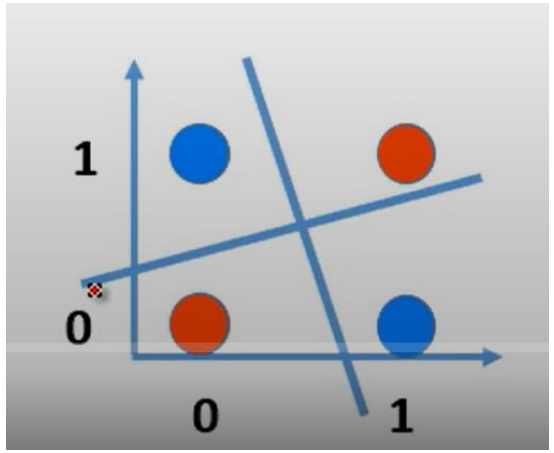
- Les **poids** W et le **biais** $b$ correctement ajustés.
- Un modèle capable de séparer les données **linéairement**

## *Limites du Perceptron Simple : Problème de Non-linéarité*

Le perceptron simple **ne peut pas résoudre des problèmes qui ne sont pas linéairement séparables**. Cela signifie qu'un perceptron simple ne pourra pas apprendre des modèles complexes où les classes ne peuvent pas être séparées par une seule droite ou hyperplan.

***Exemple classique : le problème XOR:***

| Entrée $x_1$ | Entrée $x_2$ | Sortie $y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




**On ne peut pas avoir une séparation linéaire**

# Le Perceptron Multicouche (MLP)

## *Architecture du MLP*

Le **Perceptron Multicouche (MLP)** est un réseau de neurones **supervisé** composé de plusieurs couches entièrement connectées. Il est structuré en trois types de couches:

Couche d'entrée (Input Layer):

- Elle reçoit directement les données brutes.
- Chaque neurone de cette couche représente une **caractéristique** ou **variable** d'entrée:
  - **Exemple (Tabulaire)** : 5 variables → **5 neurones**.
  - **Exemple (Image)** : Image 28×28 pixels → **784 neurones**.
- **Formule:**

$$A^0 = X = (x_1, x_2, \dots\dots, x_n)$$

$X$est le vecteur d'entrée.

## Couches cachées (Hidden Layers):

- Ce sont les couches intermédiaires qui effectuent des **transformations complexes** sur les données.
- Chaque neurone d'une couche cachée reçoit les sorties de tous les neurones de la couche précédente
- Le nombre de couches cachées et de neurones par couche détermine la **profondeur** et la **capacité d'apprentissage** du réseau.
- **Formule:**

$$Z^{(l)} = W^{(l)} A^{(l-1)} + b^{l}$$
$$A^{l} = f\left(z^{l}\right)$$

## Couche de sortie (Output Layer):

- Produit la **prédiction finale**.
- **Nombre de neurones** : Dépend du problème.
  - **Classification binaire** → 1 neurone (**Sigmoïde**).
  - **Classification multiclasse** → $K$ neurones (**Softmax**).
  - **Régression** → 1 neurone (sans activation ou **ReLU**).

$$Z^{(l)} = W^{(l)} A^{(l-1)} + b^{l}$$
$$\hat{Y} = f\left(z^{l}\right)$$

## *Propagation Avant (Forward Propagation) dans un MLP*

La **Propagation Avant** est le processus par lequel une entrée traverse le réseau pour produire une sortie. Chaque neurone applique une transformation **affine** suivie d'une **fonction d'activation non linéaire**.

### Étape 1 : Calcul de la Combinaison Linéaire:

Pour chaque couche $l \in \{1,2,\dots,L\}$ ,on calcule la somme pondérée des activations de la couche précédente :

$$Z^{(l)} = W^{(l)} A^{(l-1)} + b^{l}$$

### Étape 2 : Application de la Fonction d'Activation:

La sortie du neurone est obtenue en appliquant la fonction d'activation :

$$A^{l} = f\left(z^{l}\right)$$

## Étape 3 : Calcul de la Sortie

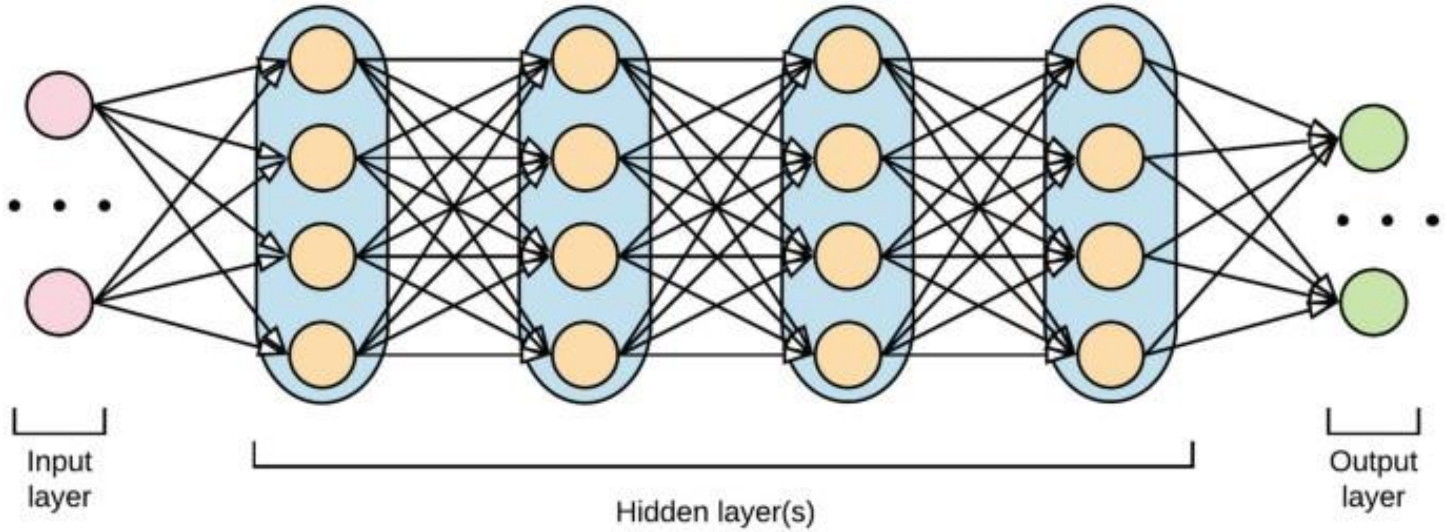
À la couche de sortie (l=L), on applique la dernière transformation :

$$\hat{Y} = f\left(W^{(l)} A^{(l-1)} + b^{l}\right)$$

La **Propagation Avant** suit ces étapes :

1. **Combinaison linéaire** : somme pondérée des entrées.
2. **Non-linéarité** : activation avec $f(z)$.
3. **Propagation** de couche en couche jusqu'à la sortie.

Ce processus transforme les **entrées** en **sorties prédictives** à travers des transformations successives.


Input
layer
Hidden layer(s)
Output
layer

## *Fonctions d'Activation*

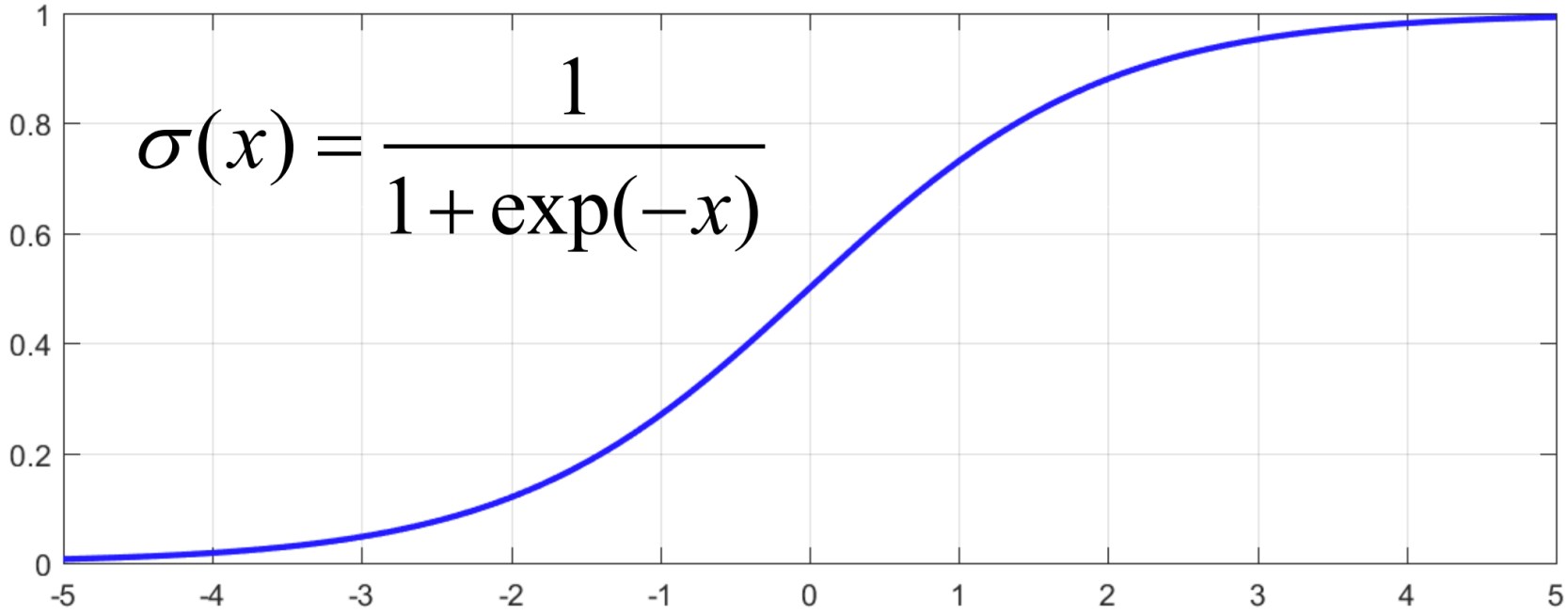
Les **fonctions d'activation** jouent un rôle **essentiel** dans les réseaux de neurones artificiels, en particulier dans les **Perceptrons Multicouches (MLP)**. Elles permettent au réseau de modéliser des relations **complexes** et **non linéaires**.

Principaux Rôles des Fonctions d'Activation:

- Introduire de la Non-linéarité
- Adapter la Sortie aux Problèmes Spécifiques
- Optimiser la Convergence de l'Apprentissage
- Considérations sur le Temps de Calcul

## Fonction d’Activation Sigmoïde:

La fonction sigmoïde est une fonction d'activation classique utilisée dans les réseaux de neurones, surtout pour la classification binaire. Elle permet de transformer toute valeur réelle en un nombre compris entre 0 et 1, ce qui la rend idéale pour prédire des probabilités.



## Exemple:

Imaginons un modèle de réseau de neurones qui doit prédire si un email est un **spam** (classe 1) ou **non-spam** (classe 0). Le modèle reçoit une entrée X (qui pourrait être, par exemple, un ensemble de caractéristiques extraites du contenu de l'email, comme la fréquence de certains mots, la longueur de l'email, etc.).

Considérons :

- Poids $W = [1.2, -0.8]$
- Biais $b = 0.5$
- Entrée $X = [2,3]$

Étape 1 : Calcul de la somme pondérée:

$$z = (1.2 \times 2) + (-0.8 \times 3) + 0.5 = 2.4 - 2.4 + 0.5 = 0.5$$

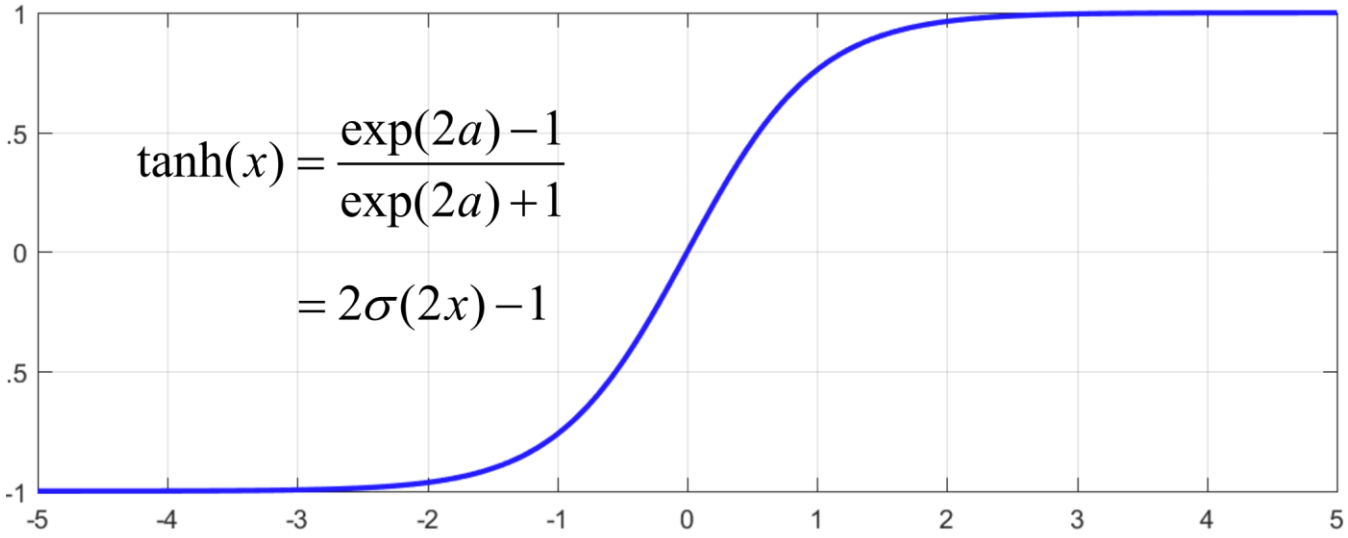
Étape 2 : Application de la fonction sigmoïde

$$\sigma(0.5) \approx 0.622$$

La sortie du neurone, après avoir appliqué la fonction sigmoïde, est environ **0.621**. Cela signifie que, selon le modèle, la probabilité que que l'email soit un spam est d'environ **62.1%**.

# Fonction d’Activation Tanh (Tangente Hyperbolique):

La fonction Tanh (ou tangente hyperbolique) est une amélioration de la fonction sigmoïde. Contrairement à la sigmoïde qui génère des sorties entre 0 et 1, la fonction Tanh produit des sorties entre -1 et 1, ce qui la rend plus adaptée pour des données centrées autour de zéro.



## Exemple:

Considérons :

- Poids $W = [0.5, -1]$
- Biais $b = 0.2$
- Entrée $X = [1,2]$

Étape 1 : Calcul de la somme pondérée:

$$z = (0.5 \times 2) + (-1 \times 3) + 0.2 = -1.3.$$

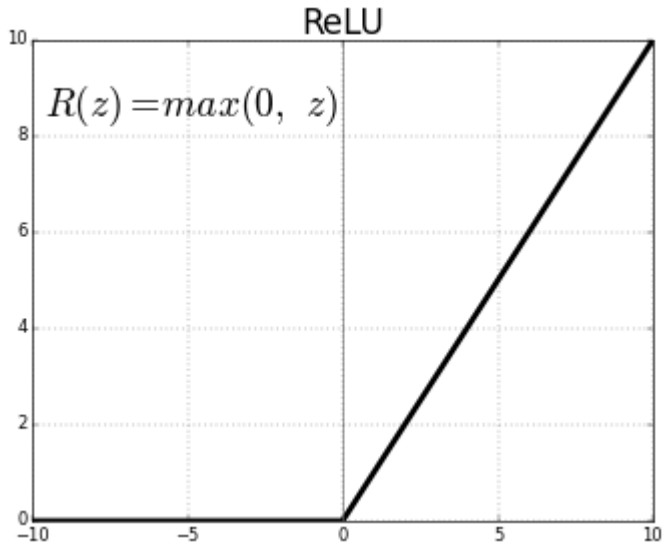
Étape 2 : Application de la fonction Tanh

$$\tanh(-1.3) \approx -0.861$$

La sortie du neurone est $\mathbf{-0.861}$, ce qui indique une forte appartenance à la classe négative.

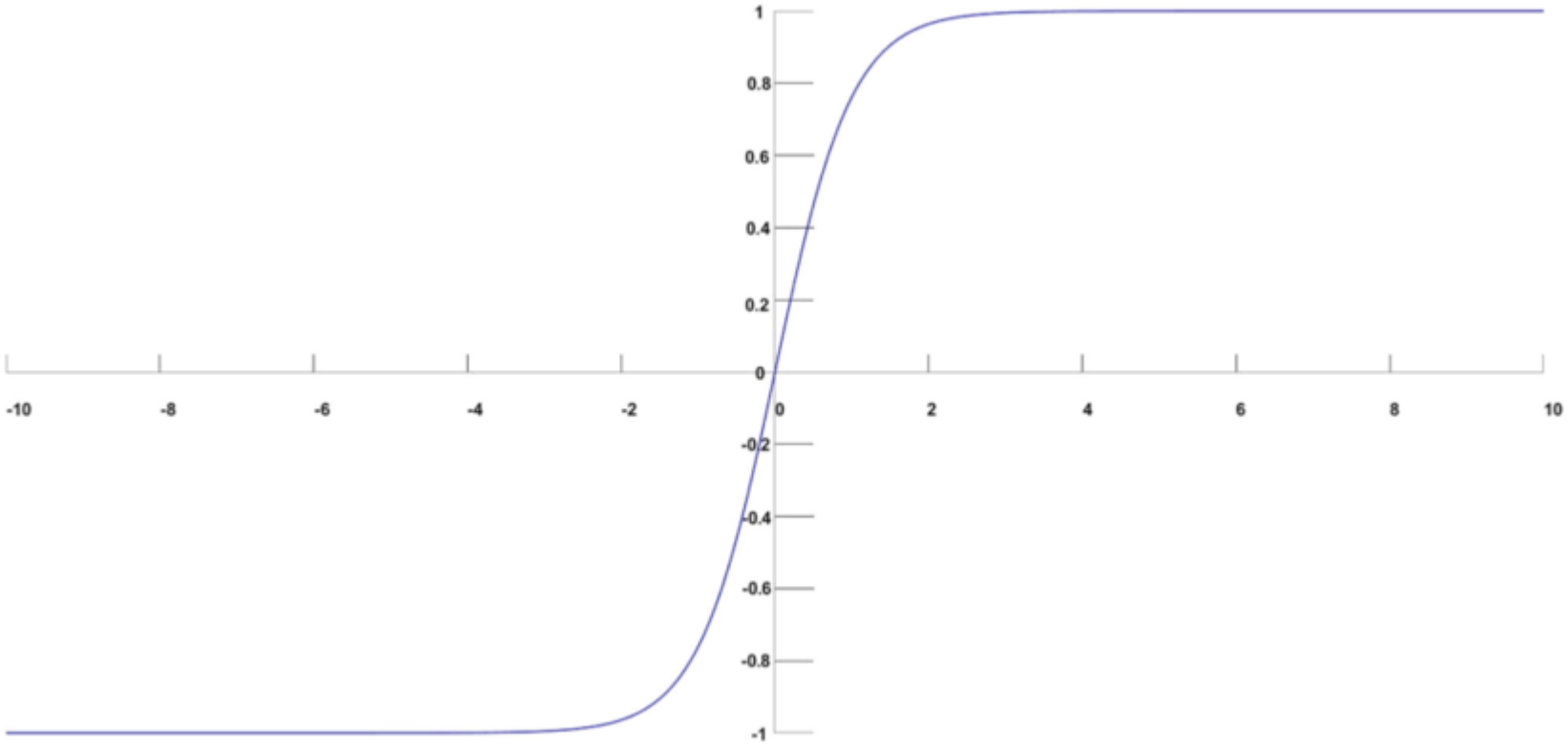
# Fonction d'Activation ReLU:

La ReLU est aujourd'hui l'une des fonctions d'activation les plus utilisées dans les réseaux de neurones, notamment pour les réseaux profonds. Elle est simple, rapide à calculer et efficace pour résoudre le problème du gradient qui disparaît.



# Fonction d'Activation Softmax

La **Softmax** est une fonction d'activation utilisée principalement dans la **couche de sortie** des réseaux de neurones pour les **problèmes de classification multiclasse**. Elle transforme un vecteur de valeurs réelles en un vecteur de **probabilités** dont la somme est égale à 1.



# Apprentissage et Optimisation

## *Fonction de Perte (Loss Function)*

**Définition** :

La **fonction de perte** mesure l'écart entre la **prédiction** du modèle $\hat{y}$ et la **valeur réelle** $y$ pour **chaque observation individuelle**. Son objectif est de quantifier l'erreur commise afin de guider l'apprentissage du modèle.

**Rôle et Importance de la Fonction de Perte:**

- Mesure de la Performance du Modèle
- Guide l'Optimisation du Modèle
- Influence sur la Vitesse et la Qualité de l'Apprentissage

## *Types de Fonctions de Perte selon les Problèmes*

### Régression (Problème de Prédiction Continue avec l'Erreur Quadratique Moyenne (MSE)):

L'**Erreur Quadratique Moyenne (MSE)** est une fonction de perte couramment utilisée pour les problèmes de **régression**. Elle mesure la moyenne des carrés des écarts entre les valeurs prédites et les valeurs réelles, pénalisant fortement les grandes erreurs, Sa formule est la suivante:

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

- $y$: Valeur réelle
- $\hat{y}$:Valeur prédite par le modèle
- n:Nombre d'exemples dans le jeu de données

Exemple : Prédiction du Prix des Maisons

Imaginons que nous essayons de prédire le **prix d'une maison** en fonction de ses caractéristiques, comme la surface en mètres carrés. Ce problème de régression vise à prédire une valeur continue (prix de la maison) à partir de certaines données d'entrée

| Surface ($m^2$) | Prix réel (en €) | Prix prédit (en €) |
|---|---|---|
| 80 | 250,000 | 240,000 |
| 100 | 300,000 | 310,000 |
| 120 | 350,000 | 340,000 |
| 150 | 450,000 | 440,000 |

$$MSE=100\ 000\ 000$$

Erreur Quadratique Moyenne (MSE) est de 100,000,000, ce qui signifie qu'en moyenne, chaque prédiction de prix est déviée de 10,000 € par rapport à la réalité (en termes absolus).

## Classification Binaire (Entropie Croisée Binaire (Binary Cross-Entropy))

L'entropie croisée binaire est utilisée lorsque le problème de classification comporte **deux classes** (0 ou 1).

**Formule:**

$$L(\hat{y}, y) = -(y.\log \hat{y} + (1-y)\log(1-\hat{y}))$$

- $y$ : **Étiquette réelle** (0 ou 1)
- $\hat{y}$ **:probabilité prédite** que l'échantillon appartienne à la classe 1

Plus la prédiction est proche de la vérité, plus la perte est faible.

Exemple : Détection de Spam

| Email | Vraie Classe | Probabilité prédite | Perte (BCE) |
|---|---|---|---|
| A | spam | 0.9 | 0.105 |
| B | spam | 0.1 | 2.302 |
| C | Non spam | 0.8 | 1.609 |
| D | Non spam | 0.05 | 0.051 |

**Analyse :**

- **Email A** : Bonne prédiction → **Perte faible**
- **Email B** : Mauvaise prédiction → **Perte élevée**
- **Email C** : Mauvaise prédiction → **Perte élevée**
- **Email D** : Bonne prédiction → **Perte faible**

## Classification Multiclasse Entropie Croisée Catégorique (Categorical Cross-Entropy)

Utilisée pour les problèmes où il y a **plus de deux classes**. Elle compare un **vecteur one-hot** des vraies classes avec les **probabilités prédites**.

**Formule:**

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j . \log \hat{y}_j$$

- $y$ : **Étiquette réelle**
- $\hat{y}$ **:probabilité prédite pour classe j**
- C**: nombre de classe**

**Explication:**

- **One-hot encoding** est utilisé pour $y$
- Seule la probabilité de la **classe correcte** impacte la perte.

Exemple : Reconnaissance de Chiffres (0 à 9):

Un modèle doit prédire le chiffre représenté par cette image:

| Classe | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $y$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\hat{y}$ | 0.01 | 0.05 | 0.6 | 0.1 | 0.05 | 0.04 | 0.03 | 0.06 | 0.03 | 0.03 |

**Calcul de la Perte :**

$$L(\hat{y}, y) = 0.511$$

Le modèle prédit la classe **2** avec 60% de probabilité → **Perte modérée**.

## *Algorithme de Rétropropagation (Backpropagation)*

L'algorithme de rétropropagation est un **élément fondamental** dans l'entraînement des réseaux de neurones multicouches (**MLP**). Il permet de **calculer efficacement les gradients** de la fonction de perte par rapport aux poids et biais du réseau, afin de les mettre à jour et améliorer les performances du modèle.

### Principe de la Rétropropagation:

L'objectif est de **minimiser la fonction de perte** en ajustant les poids W et les biais b du réseau

**Deux grandes étapes** :

1. **Propagation avant (Forward Pass)** :
   Calcul des sorties du réseau pour une entrée donnée.
2. **Propagation arrière (Backward Pass)** :

   Calcul des gradients de la fonction de perte par rapport aux poids et biais

# Étapes de la Rétropropagation:

**Étape 1 :Erreur de la couche de sortie**

On calcule l'erreur entre la sortie prédite $\hat{y}$ et la sortie réelle $y$

$$\delta^L = \frac{\partial L}{\partial z^l} = (A^L - y) \odot f'(z^l)$$

**Étape 2 Erreur des couches cachées:**

$$\delta^l = (W^{(l+1)T} \delta^{l+1}) \odot f'(z^l)$$

**Étape 3: Calcul des gradients**

$$\frac{\partial L}{\partial W^l} = \delta^l A^{(l-1)T}$$

$$\frac{\partial L}{\partial b^l} = \delta^l$$

**Étape 4 Mise à Jour des Poids et Biais**

$$W^l = W^l - \eta \frac{\partial L}{\partial W^l}$$

$$b^l = b^l - \eta \frac{\partial L}{\partial b^l}$$

## Points Clés:

- La propagation avant calcule la sortie du réseau.
- La propagation arrière calcule comment ajuster les poids pour réduire l'erreur.
- La descente de gradient met à jour les paramètres pour améliorer les performances.
- L'efficacité de la rétropropagation dépend du choix des fonctions d’activation et du taux d’apprentissage.

## Optimisation

L’optimisation consiste à ajuster les poids et les biais d’un réseau de neurones pour minimiser la fonction de perte . Cela permet au modèle d’améliorer ses prédictions

**Gradient Descent:**

**Principe** :

Met à jour les paramètres dans la direction opposée au **gradient** de la fonction de perte.

**Formule** :

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta)$$

- $\theta$: Poids et biais.
- $\eta$: Taux d’apprentissage (learning rate).
- $\nabla_\theta L(\theta)$: Gradient de la perte.

**Variantes** :

- **Batch Gradient Descent** : Calcul avec tout le dataset.
- **Stochastic Gradient Descent (SGD)** : Mise à jour à chaque exemple.
- **Mini-Batch Gradient Descent** : Mise à jour par petit lot d'exemples.

**Algorithmes Avancés**

- **Momentum** : Accélère la convergence en ajoutant un terme de vitesse.
- **RMSProp** : Ajuste le taux d'apprentissage pour chaque paramètre.
- **Adam** : Combine Momentum et RMSProp, adaptatif et rapide.

## Régularisation

## Objectif:

La régularisation est une technique utilisée pour **réduire le surapprentissage** (overfitting) et améliorer la capacité du modèle à se généraliser sur des données non vues. En ajoutant une pénalité aux poids du modèle, elle aide à éviter qu'il ne s'adapte trop précisément aux données d'entraînement, y compris leur bruit.

## Problème du Surapprentissage:

Le surapprentissage survient lorsque le modèle **s'adapte trop précisément** aux données d'entraînement, capturant à la fois les relations sous-jacentes et le **bruit** présent dans les données. Cela conduit à un modèle performant sur les données d'entraînement mais moins efficace sur des données nouvelles.

**Symptômes du surapprentissage :**

Très bonne performance sur les données d'entraînement, mais **mauvaise performance sur les données de test**.

## Méthodes de Régularisation:

Les techniques de régularisation agissent principalement en modifiant la fonction de perte $L(w)$ ,où $w$ représente les poids du modèle. Voici les principales méthodes de régularisation:

### Régularisation L1 (Lasso):

**Principe :**
Ajout d'une pénalité proportionnelle à la **somme des valeurs absolues** des poids $W$.

**Formule de la fonction de perte:**

$$L(w) = L_{initiale}(w) + \lambda \sum_{i=1}^{n} |w_i|$$

- $L_{initiale}(w)$:Fonction de perte sans régularisation
- $\lambda$ : Hyperparamètre qui contrôle l'importance de la régularisation.

**Effet :**

- Encourage les poids $w_i$ à devenir **exactement égaux à zéro**, ce qui peut éliminer des caractéristiques inutiles.
- Convient pour des modèles où certaines caractéristiques sont peu informatives (sparse models).

## Régularisation L2 (Ridge):

**Principe :**
Ajout d'une pénalité proportionnelle à la **somme des carrés** des poids W.

**Formule de la fonction de perte :**

$$L(w) = L_{initiale}(w) + \lambda \sum_{i=1}^{n} {w_i}^2$$

**Effet :**

- Réduit les valeurs des poids $w_i$, mais ne les force pas à être nuls.
- Privilégie un modèle plus stable en **limitant les poids extrêmes**, ce qui améliore la généralisation.

## Dropout:

**Principe :**
Pendant l'entraînement, **désactiver aléatoirement une fraction des neurones** à chaque itération pour réduire la dépendance du modèle à des combinaisons spécifiques de neurones.

**Formule conceptuelle :**

$$h_i^{dropout} = h_i . r_i \quad r_i \sim \text{Bernoulli(p)}$$

$h_i$ : Activation du neurone $i$.

$r_i$ : Masque binaire (0 ou 1) généré aléatoirement avec une probabilité $p$ de garder le neurone actif.

**Effet :**

- Force le modèle à apprendre des représentations redondantes et plus robustes.
- Réduit le surapprentissage dans les réseaux profonds.

## Early Stopping

**Principe :**
Arrêter l'entraînement lorsque la performance sur les données de validation cesse de s’améliorer.

**Procédure :**

- Suivre l’évolution de la fonction de perte sur les données de validation.
- Interrompre l’entraînement lorsque la perte de validation commence à augmenter, ce qui indique un surapprentissage.

**Effet :**

- Empêche le modèle de s'adapter trop aux données d'entraînement.

# Évaluation et Performances

## Mesures de Performance

Pour évaluer un MLP, on utilise des métriques adaptées aux tâches :

Régression :

- MAE (Erreur Absolue Moyenne) $$MAE = \frac{1}{m}\sum_{i=1}^{m}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

- $R^2$ (Coefficient de Détermination) $$R^2 = 1 - \frac{\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{m}\left(y^{(i)} - \bar{y}\right)^2}$$

Classification Binaire :

Précision, Rappel et F1-Score :

Exemples des formules pour la classe positive $(y = 1)$

$$\text{Précision} = \frac{\text{VP}}{\text{VP} + \text{FP}}, \quad \text{Rappel} = \frac{\text{VP}}{\text{VP} + \text{FN}}, \quad F1 = 2 \cdot \frac{\text{Précision} \cdot \text{Rappel}}{\text{Précision} + \text{Rappel}}$$

Classification Multiclasse :

Exactitude (Accuracy) : $$\text{Exactitude} = \frac{\text{Nombre de Prédictions Correctes}}{\text{Nombre Total d'Échantillons}}$$

## Solution de Surapprentissage

1. Régularisation
2. Dropout
3. Early Stopping
4. Augmentation des Données
5. Réduction de la Complexité du Modèle
6. Hyperparamètres

## Hyperparamètres

- Taille et Nombre des Couches Cachées
- Fonction d’Activation
- Taux d'Apprentissage (Learning Rate)
- Taille de Batch (Batch Size)
- Nombre d’Époques
- Méthode de Régularisation