Université IBN TOFAIL – Kénitra
ECOLE NATIONALE DES SCIENCES APPLIQUEES

Cours Intelligence Artificielle

CH7: Deep Learning:

Les Réseaux de Neurones Artificiels (ANN)

Par:

Hassan EL FADIL

elfadil.h@gmail.com

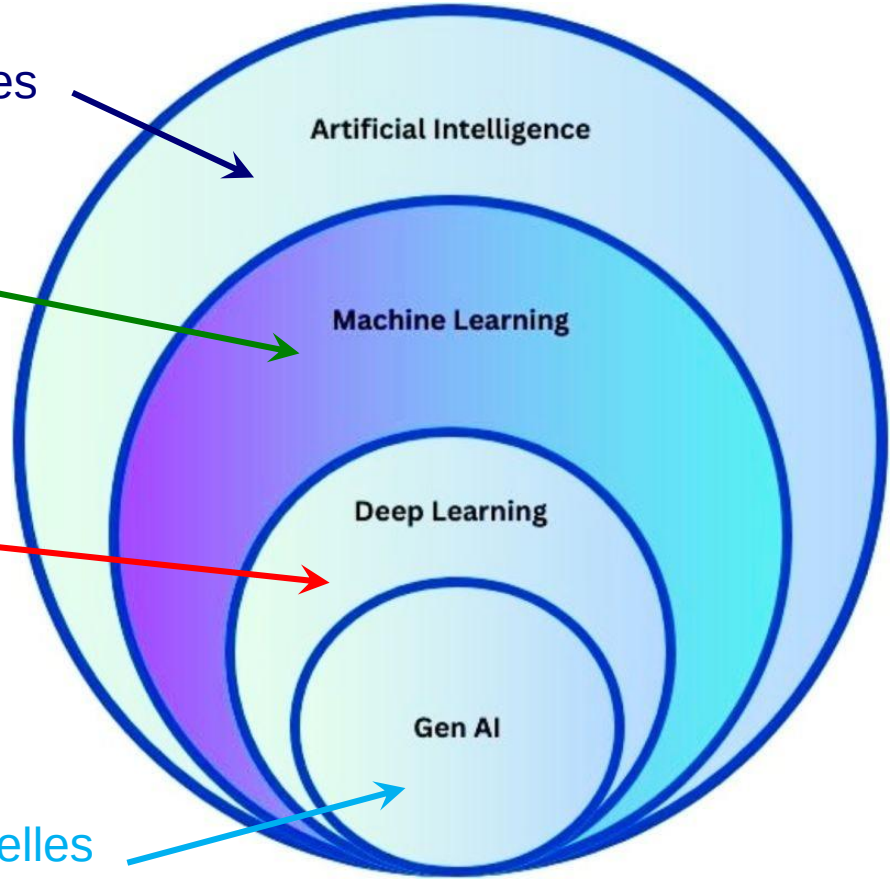
1. Introduction au Deep Learning Apprentissage profond)

1.1. Pourquoi le Deep Learning?

Intelligence Artificielle (IA) : Champ global visant à créer des machines capables de simuler l'intelligence humaine.

Machine Learning (ML) : Sous-domaine de l'IA, se concentre sur l'apprentissage automatique des modèles à partir de données sans programmation explicite.

Deep Learning (DL) : Sous-catégorie du ML, utilise des réseaux de neurones profonds pour traiter des données complexes et résoudre des problèmes avancés comme la vision ou le langage naturel.

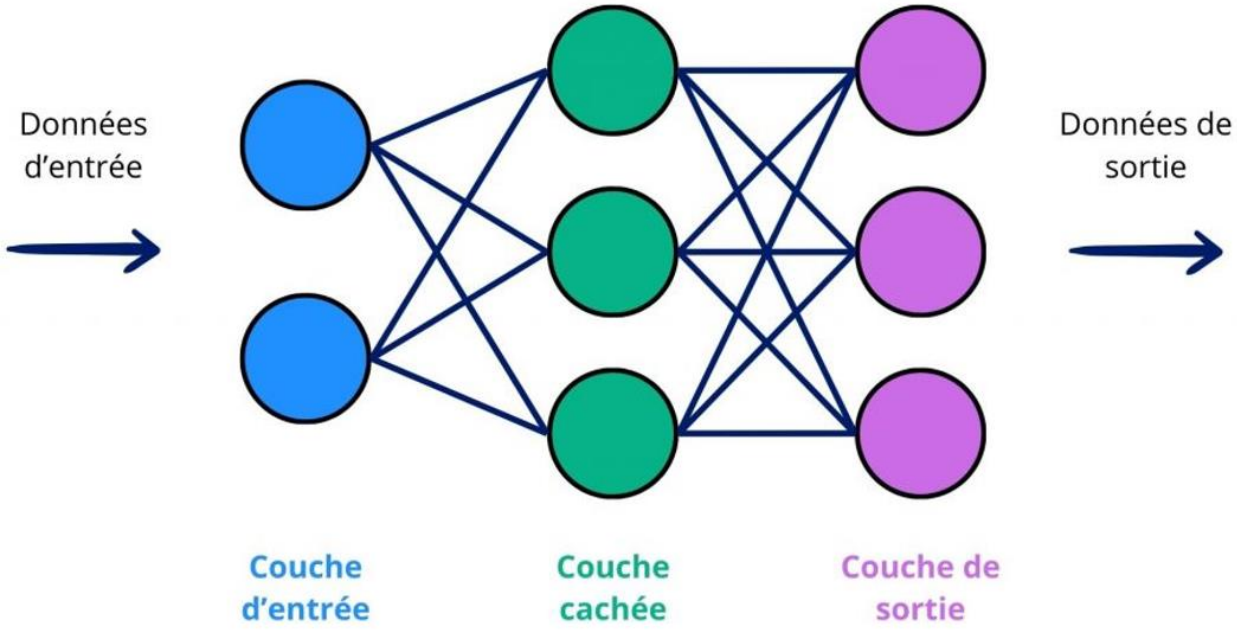


L'**IA générative** utilise des modèles pour créer de nouvelles données, telles que des images, du texte ou de la musique, qui imitent des exemples réels.

… Pourquoi le Deep Learning?

Le **Deep Learning** est utilisé pour extraire automatiquement des caractéristiques complexes et résoudre des problèmes non linéaires à grande échelle, grâce à des réseaux de neurones profonds capables de traiter de vastes quantités de données.

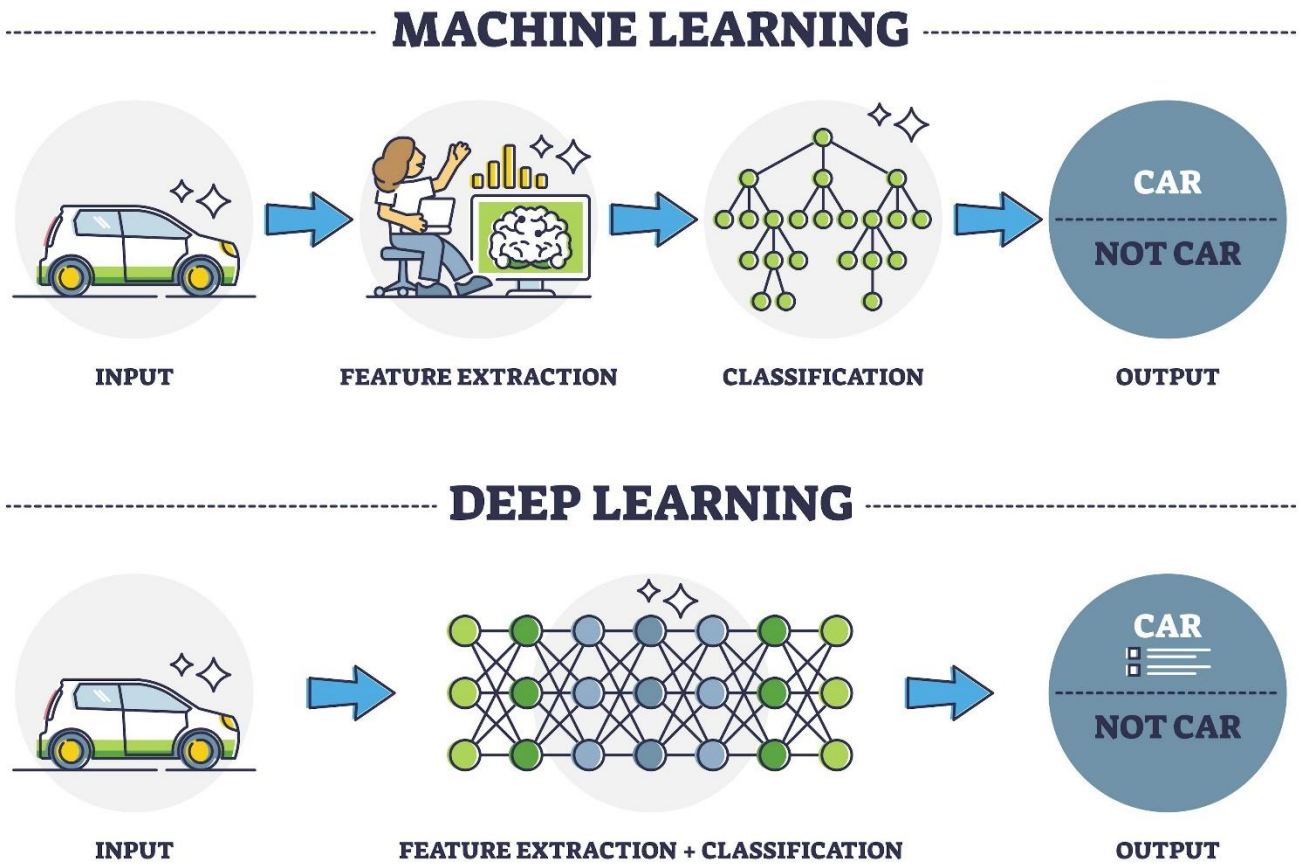
Réseaux de neurones artificiels



1.2. Machine Learning vs. Deep Learning:

- En **Machine Learning (ML)**, la **feature extraction** (extraction de caractéristiques) et la **classification** sont généralement deux étapes distinctes.
- Les ingénieurs doivent d'abord sélectionner ou concevoir manuellement des caractéristiques pertinentes des données, puis utiliser ces caractéristiques pour entraîner un algorithme de classification ou de régression

- En **Deep Learning (DL)**, ces étapes sont **automatiquement intégrées** dans les réseaux de neurones profonds.



1.3. Applications du Deep Learning

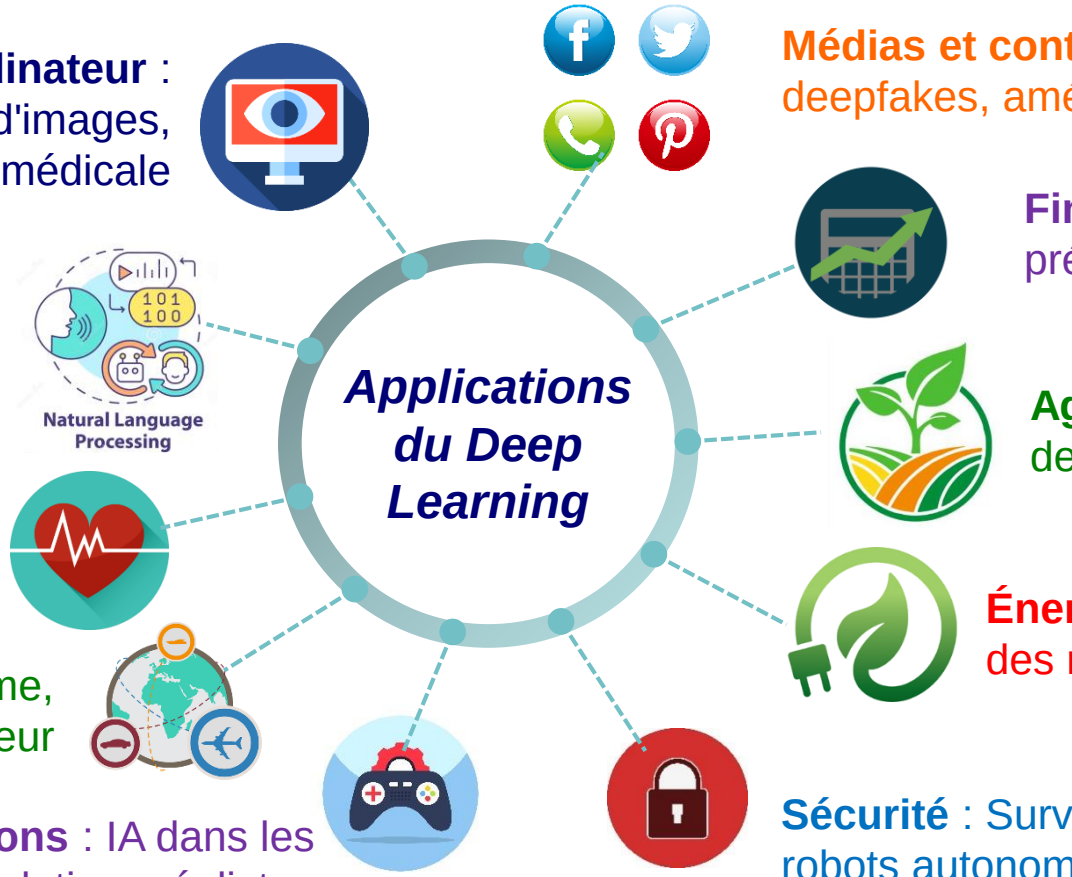
Vision par ordinateur : Reconnaissance d'images, détection d'objets, vision médicale

Traitement du langage naturel (NLP) : Traduction automatique, reconnaissance vocale, génération de texte

Santé : Diagnostic assisté, génomique, robotique chirurgicale.

Transport : Conduite autonome, assistance au conducteur

Jeux et simulations : IA dans les jeux, simulations réalistes



Médias et contenu : Synthèse vocale, deepfakes, amélioration d'images

Finance : Détection de fraudes, prédiction boursière

Agriculture : Détection des maladies des cultures, robots agricoles

Énergie : Prévision énergétique, gestion des renouvelables

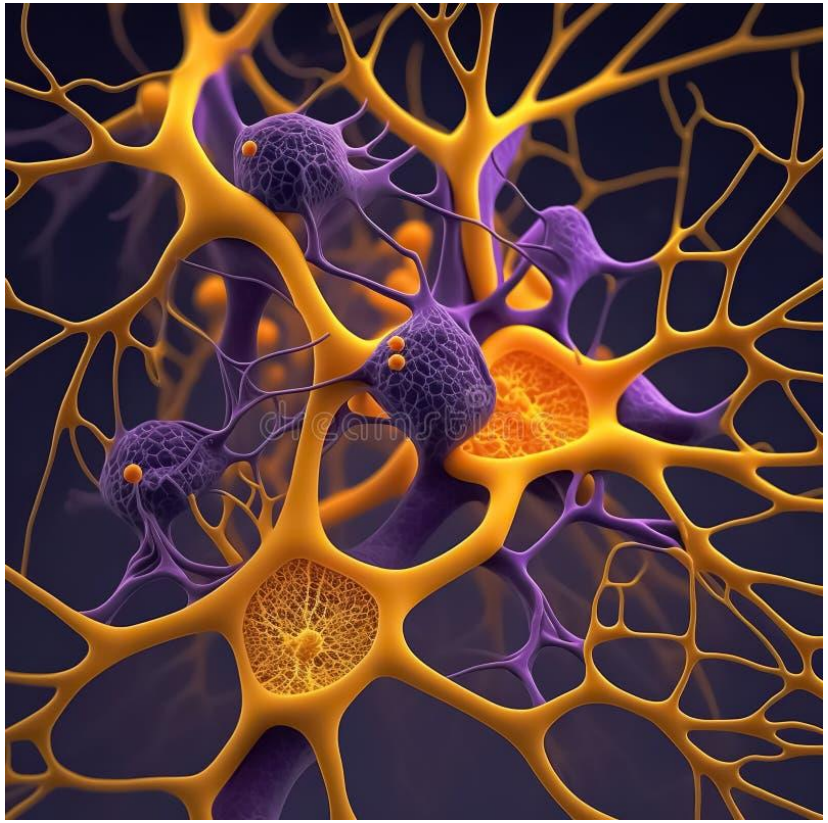
Sécurité : Surveillance, cybersécurité, robots autonomes

2. Structure des réseaux de neurones artificiels (ANNs)

2.1. Neurone Biologique:

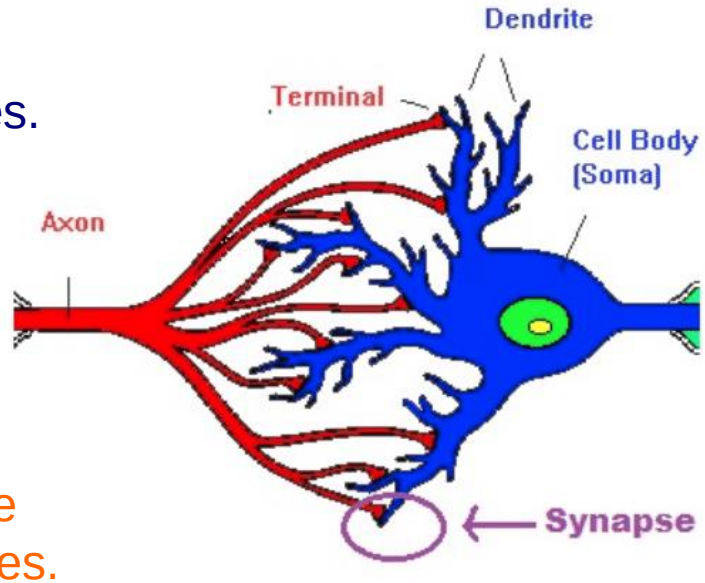
- Un neurone biologique est une cellule nerveuse capable de transmettre des informations à d'autres neurones au travers de ses différentes connexions (synapses)

- Un neurone biologique reçoit des signaux chimiques ou électriques via ses dendrites, qui sont intégrés dans le soma. Si cette intégration dépasse un seuil, le neurone génère un potentiel d'action qui est transmis le long de l'axone jusqu'aux neurones connectés.



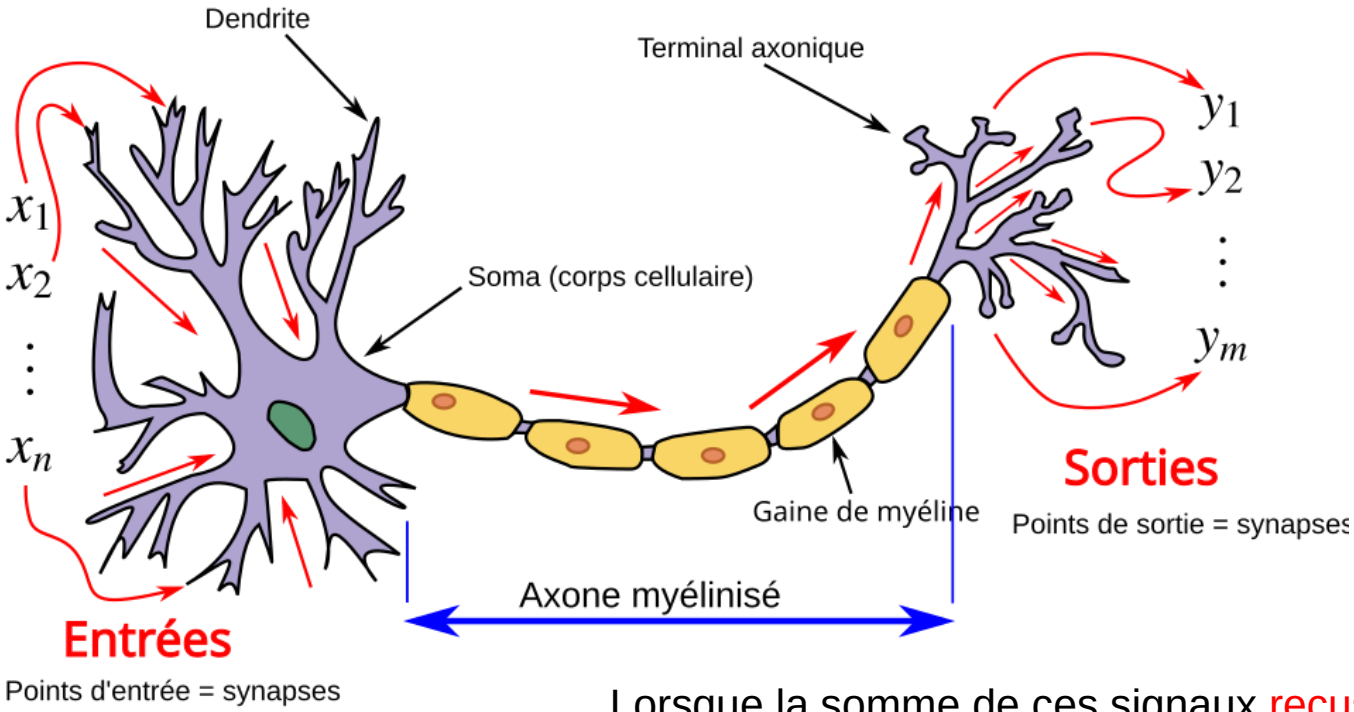
2.1. Neurone Biologique:

- **Dendrite:** Prolongements ramifiés du corps cellulaire du neurone, qui reçoivent les **signaux électriques ou chimiques** provenant d'autres neurones.
  - Captent les messages envoyés par d'autres neurones via des synapses.
  - Acheminent ces signaux électriques vers le corps cellulaire pour intégration.
- **Synapse:** Le **point de connexion** entre l'extrémité d'un axone (neurone émetteur) et une dendrite ou le corps cellulaire (neurone récepteur).
  - Permet la **transmission des signaux** entre les neurones.
- **Soma (corps cellulaire)** : C'est la partie centrale du neurone qui contient le noyau et qui est responsable de l'intégration des signaux reçus des dendrites.
- **Axone** : C'est la structure allongée qui transmet les signaux électriques du soma vers les terminaisons synaptiques pour communiquer avec d'autres neurones.



2.1. Neurone Biologique:

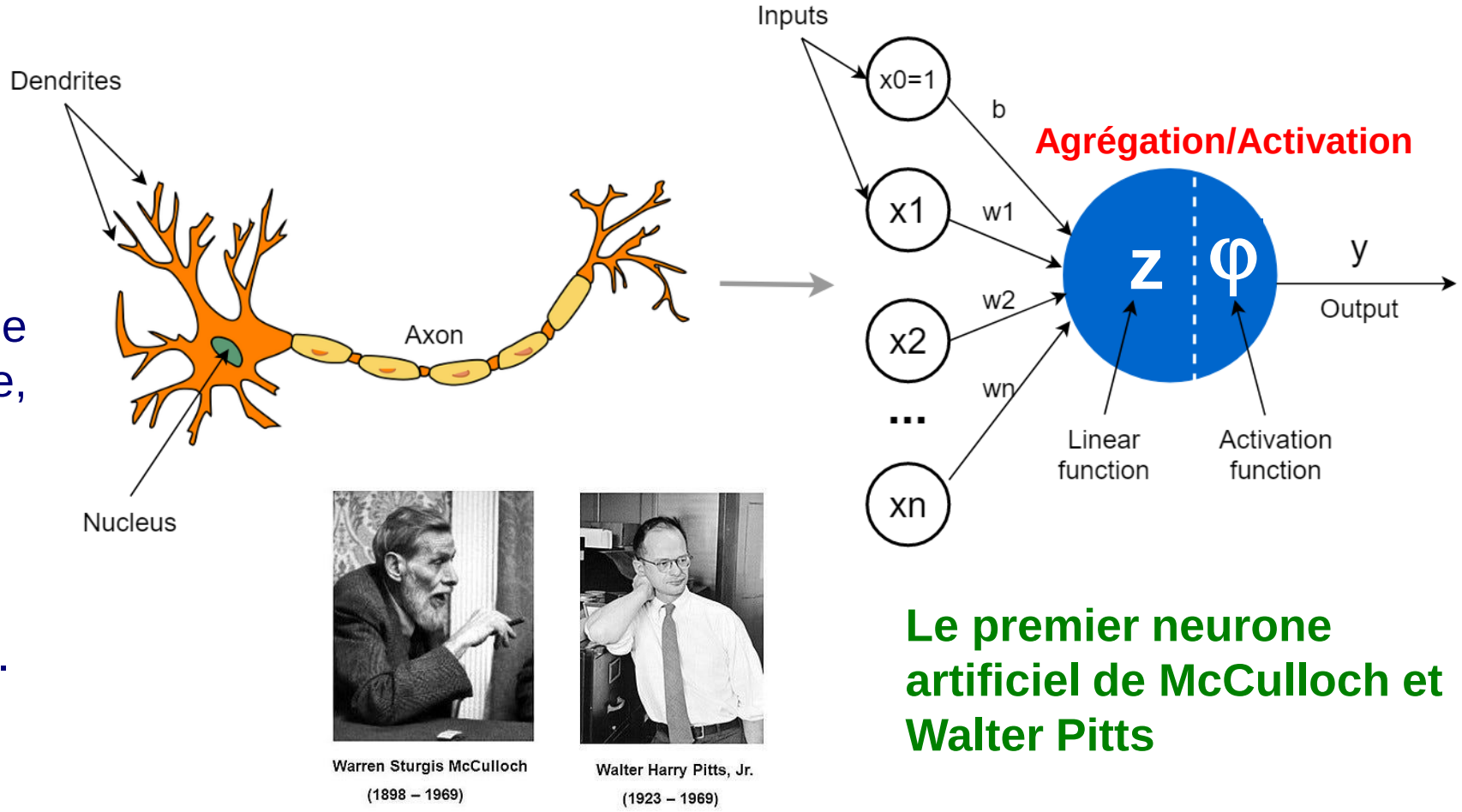
Les signaux reçus des neurones précédents peuvent être de type Excitateurs ou Inhibiteurs



Lorsque la somme de ces signaux reçus dépasse un certain seuil, le neurone s'active et produit alors un signal électrique. Ce signal circule le long de l'axone en direction des terminaisons pour être envoyé à son tour vers d'autres neurones

2.2. Le premier Neurone Artificiel : Une inspiration du neurone biologique:

- **1943 : Modèle de McCulloch et Pitts**: Warren McCulloch et Walter Pitts publient un article décrivant un modèle mathématique de neurone, capable de réaliser des calculs logiques simples.
- Ce travail est considéré comme le point de départ des réseaux de neurones.

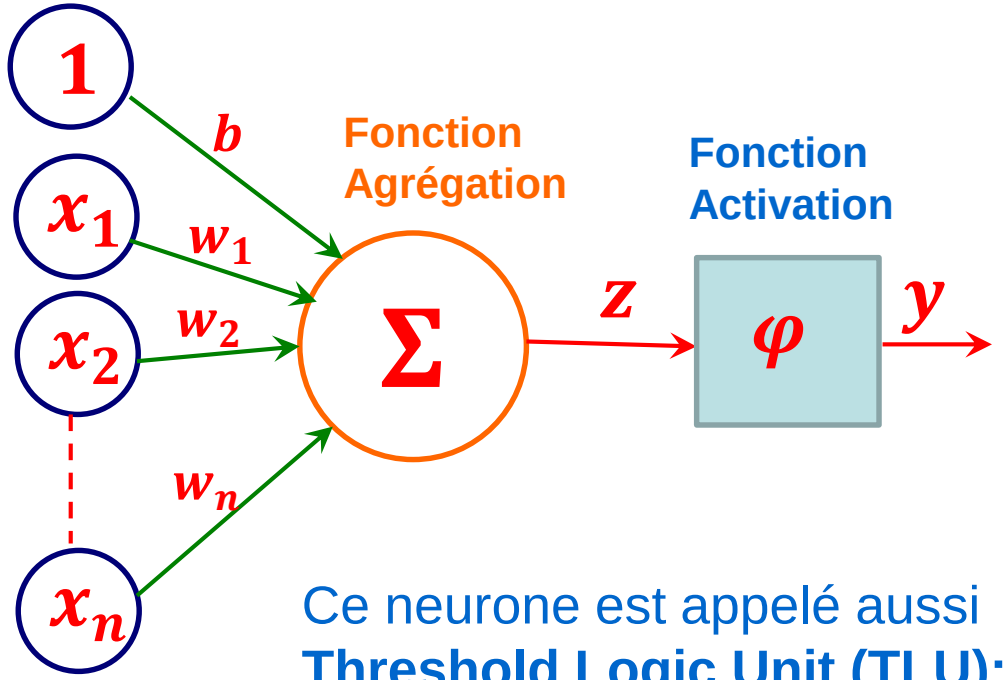


Le premier neurone artificiel de McCulloch et Walter Pitts

- ***Etape d'Agrégation:***

$$z = b + \sum_{i=1}^{n} w_i x_i = b + w_1 x_1 + \cdots + w_n x_n$$

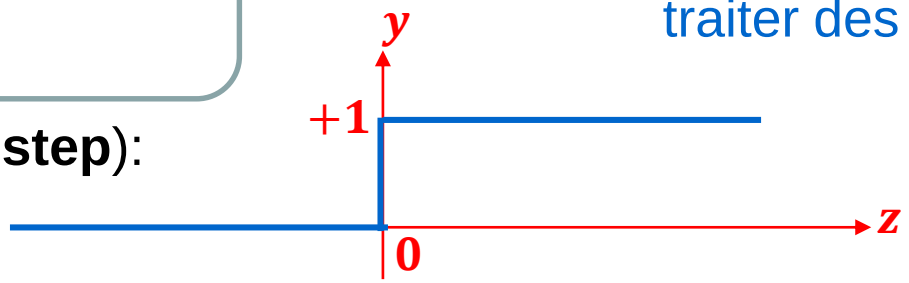
w_i : Les poids (weights)
b : Le biais (bias)



- ***Etape d'Activation:***

$$y = \begin{cases} +1 \;\; si \;\; z \geq 0 \; (seuil) \\ 0 \qquad\quad sinon \end{cases}$$

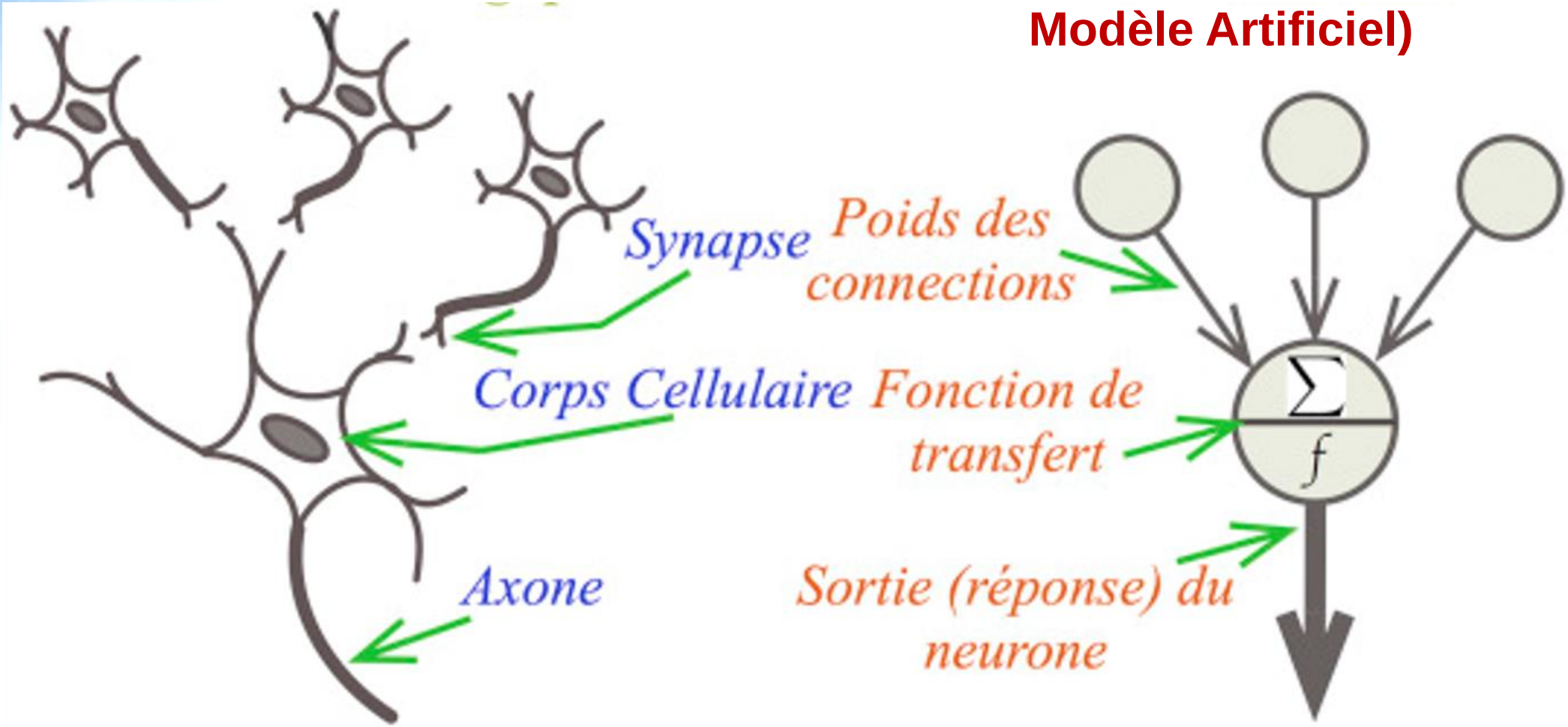
fonction seuil (ou **fonction step**):
(**threshold function**)



Ce neurone est appelé aussi **Threshold Logic Unit (TLU):** Modèle conçu initialement pour traiter des données logiques.

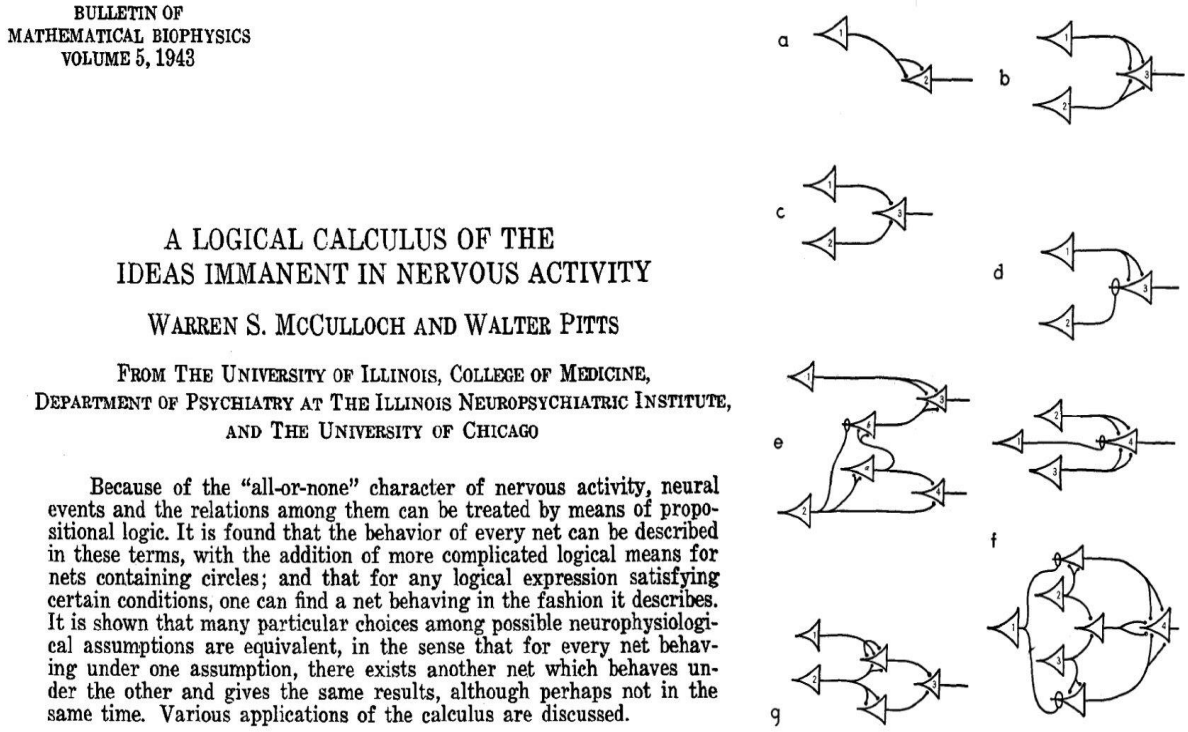
Le neurone Biologique

Le neurone Formel (Le Modèle Artificiel)



Limitations du Neurone Formel de McCulloch et Pitts

- La combinaison de plusieur neurones formels de McCulloch et Pitts permettaient de modéliser et de simuler des fonctions logiques complexes.
- Ils constituent une base fondamentale pour la conception des réseaux de neurones artificiels utilisés en intelligence artificielle.
- Néanmoins, ils sont limités par leur incapacité à apprendre, car leurs poids w_i, bias b, et seuils sont fixes, ce qui restreint leur application à des tâches simples préalablement définies.



BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

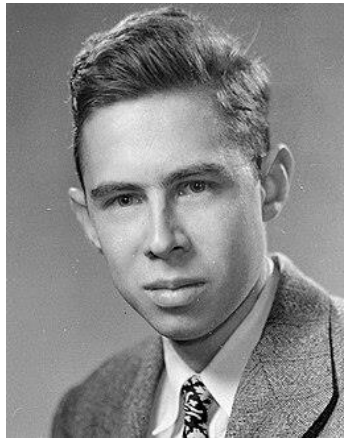
A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

2.3. Le Perceptron (Rosenblatt):

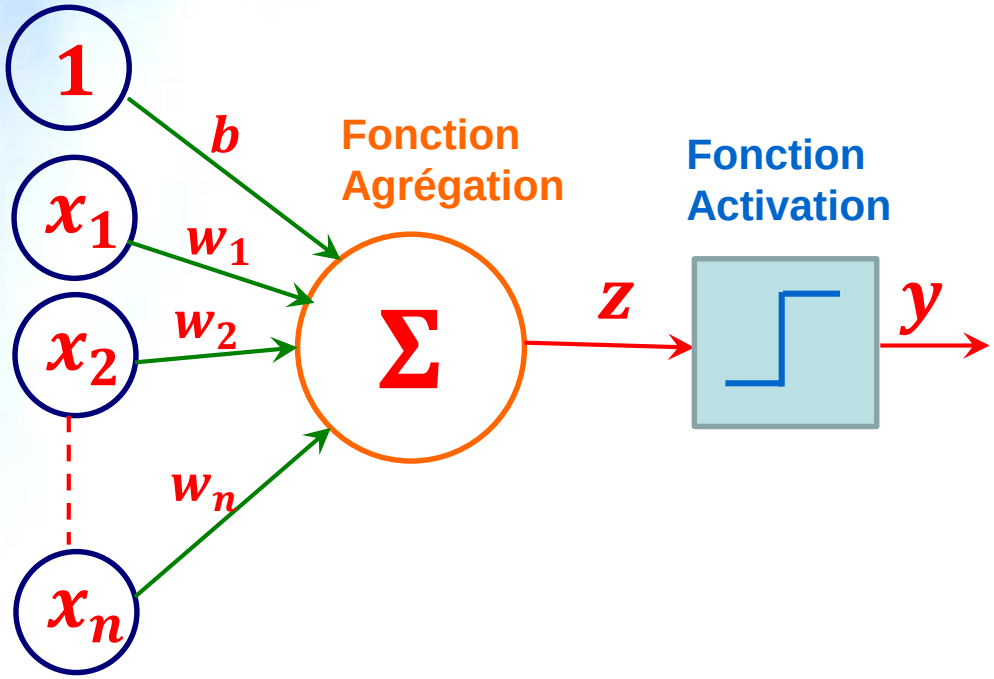


Frank Rosenblatt

- Le perceptron a été créé par **Frank Rosenblatt** en **1958** (les bases théoriques en **1957**).
- Le perceptron de Rosenblatt s'inspire du neurone formel de McCulloch et Pitts, auquel il intègre un algorithme d'apprentissage (1er algorithme d'apprentissage du Deep Learning).
- Son algorithme s'inspire de la théorie de **Hebb**:
  - *La théorie de Hebb, formulée par Donald Hebb en 1949, stipule que les connexions entre les neurones se renforcent lorsqu'ils sont activés simultanément,*
  - *Un principe souvent résumé par l'expression "les neurones qui s'activent ensemble se connectent ensemble".*
  - *Cette théorie est à la base de l'apprentissage synaptique, suggérant que les expériences répétées renforcent les voies neuronales.*
  - *En neuroscience c'est qu'on appelle la plasticité synaptique: c'est ce qui permet à notre cerveau de construire sa mémoire, d'apprendre de nouvelles choses ou encore de faire de nouvelles associations.*

Donald Hebb

2.3. Le Perceptron (Rosenblatt):



Algorithme:

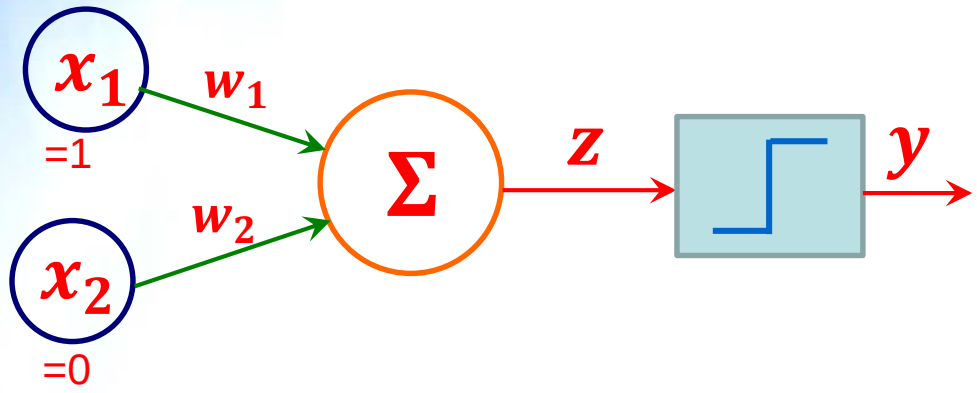
Entrainer le neurone artificiel sur des données de référence (*X*, *y*) pour que celui-ci renforce ses paramètres *W* à chaque fois qu'une entrée *X* est activée en même temps que la sortie y présente dans ces données

$$W = W + \alpha(y_{true} - y)X$$

y_{true}: sortie de référence (sortie désirée)
y : sortie produite par le neurone
X: entrée de neurone
$\alpha > 0$: vitesse d'apprentissage (pas d'apprentissage)

2.3. Le Perceptron (Rosenblatt):

Exemple: neurone à 2 entrées x_1 et x_2



$$z = w_1 x_1 + w_2 x_2$$

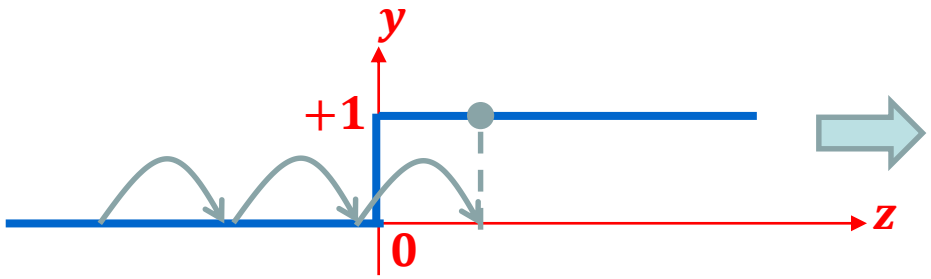
Va augmenter

Ce qui rapproche d'avantage le neurone au seuil d'activation

Supposons que le neurone produit une sortie différente de ce qu'on veut (pour $x_1 = 1$ et $x_2 = 0$):
Exemple: $y = 0$ (**z négatif**) alors que $y_{true} = 1$.
L'algorithme s'écrit:

$$W = W + \alpha X$$

Pour les entrées X qui valent 1 leur poids sera renforcé: $w_1 = w_1 + \alpha$
Pour les entrées égales à 0 leur poids reste inchangé: $w_2 = w_2$



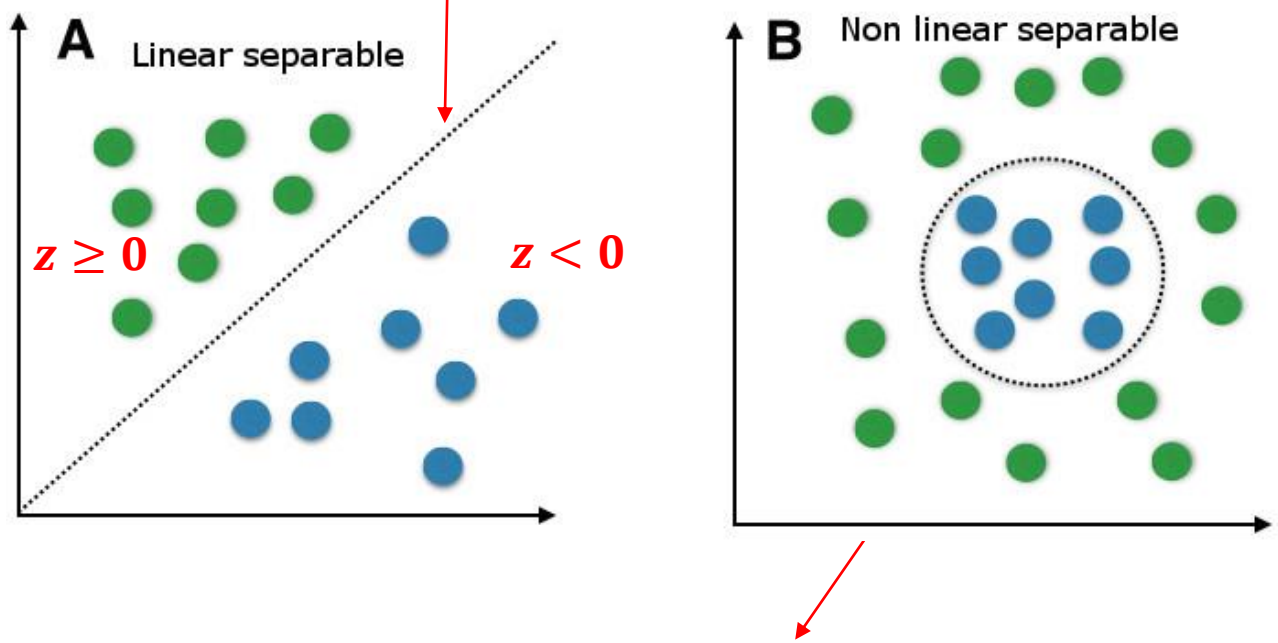
Quand z dépasse le seuil (0) y passe alors à **1**

Limitations du Perceptron de Rosenblatt

Incapacité à résoudre des problèmes non linéaires : Le perceptron ne peut résoudre que des problèmes de classification linéaire, c'est-à-dire des problèmes où les classes peuvent être séparées par une droite (ou un hyperplan dans des dimensions supérieures). Il échoue sur des problèmes plus complexes, où les classes ne peuvent pas être séparées par une ligne droite.

Fonction d'agrégation linéaire

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

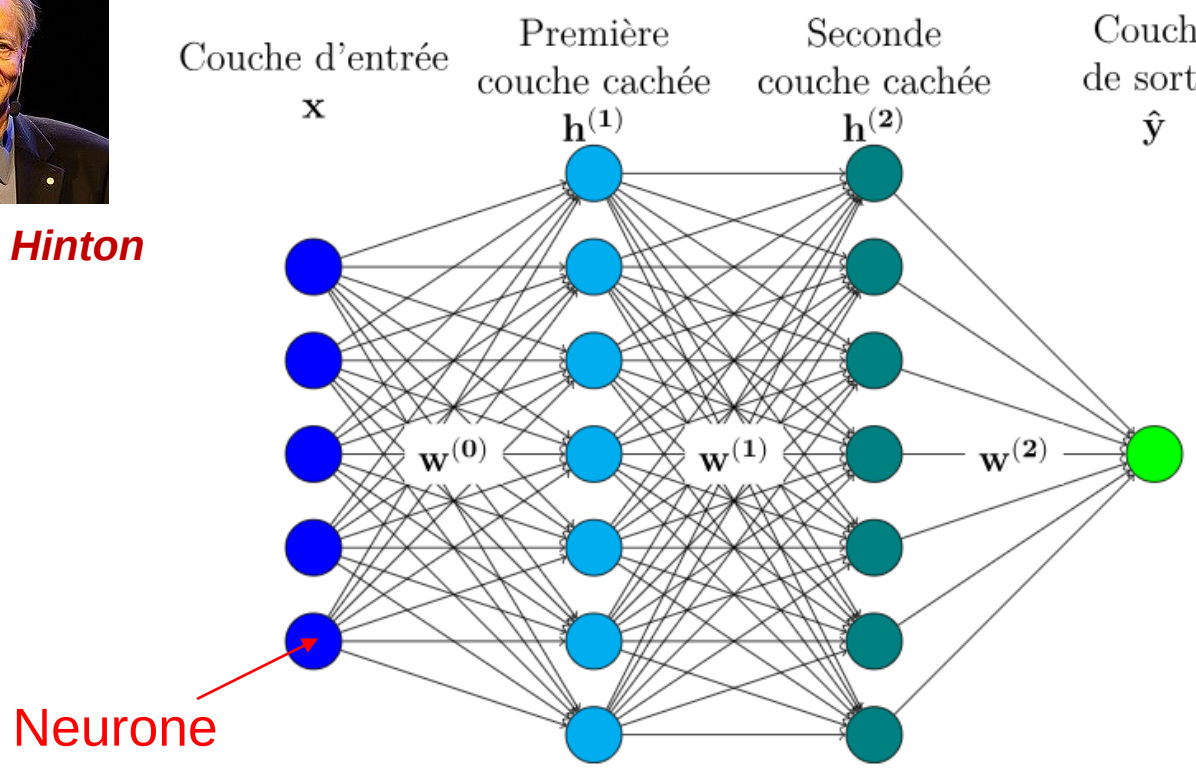


Comment faire alors pour les séparer?

2.4. Le Perceptron Multicouches (MLP) (Geoffrey Hinton):

- **Geoffrey Hinton** a développé le **perceptron multicouches** en **1986**, qui est considéré comme le premier véritable réseau de neurones artificiels.
- Cette avancée a marqué un tournant majeur dans le développement de l'apprentissage profond et a permis l'émergence de nombreuses applications en intelligence artificielle.
- Il est ainsi considéré comme l'un des pionniers du Deep Learning.

Geoffrey Hinton

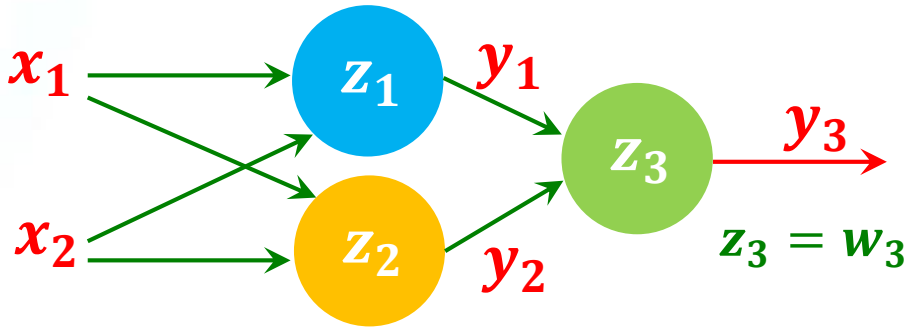


2.4. Le Perceptron Multicouches (MLP) (Geoffrey Hinton):

Exemple: Perceptron deux couches : Une couche d'entrée (y1et y2) et une couche de sortie (y3).

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$y_1 = \begin{cases} 1 & si \quad z_1 \geq 0 \\ 0 & si \quad z_1 < 0 \end{cases}$$

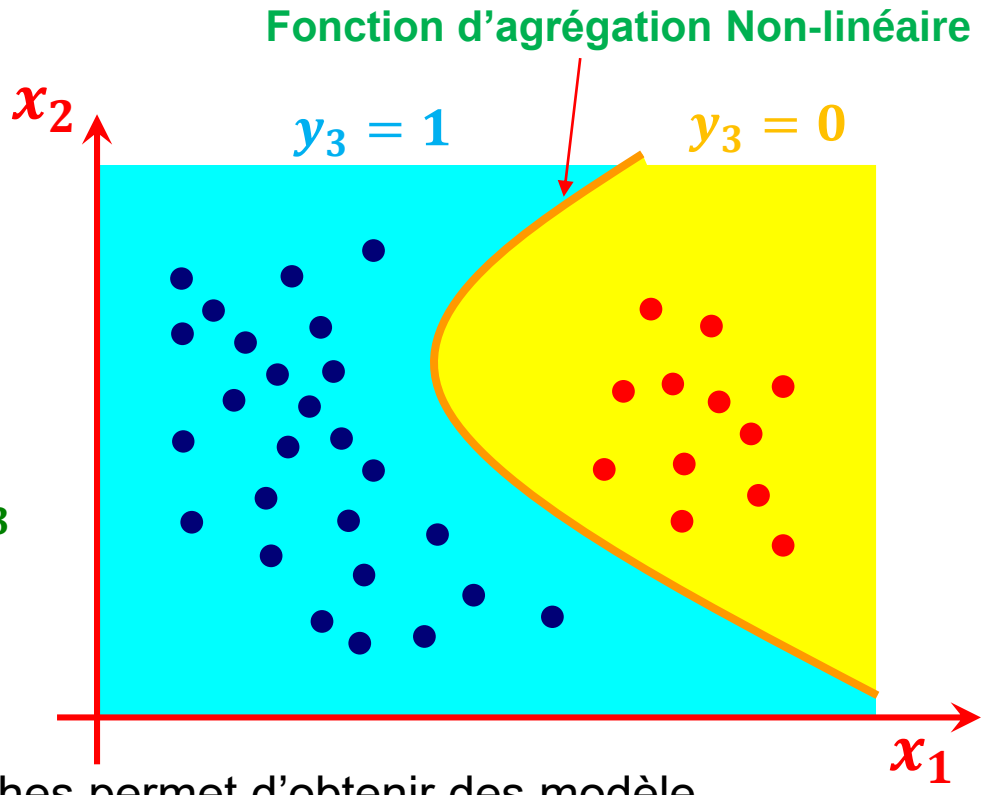


$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$y_2 = \begin{cases} 1 & si \quad z_2 \geq 0 \\ 0 & si \quad z_2 < 0 \end{cases}$$

$$z_3 = w_{31}y_1 + w_{32}y_2 + b_3$$

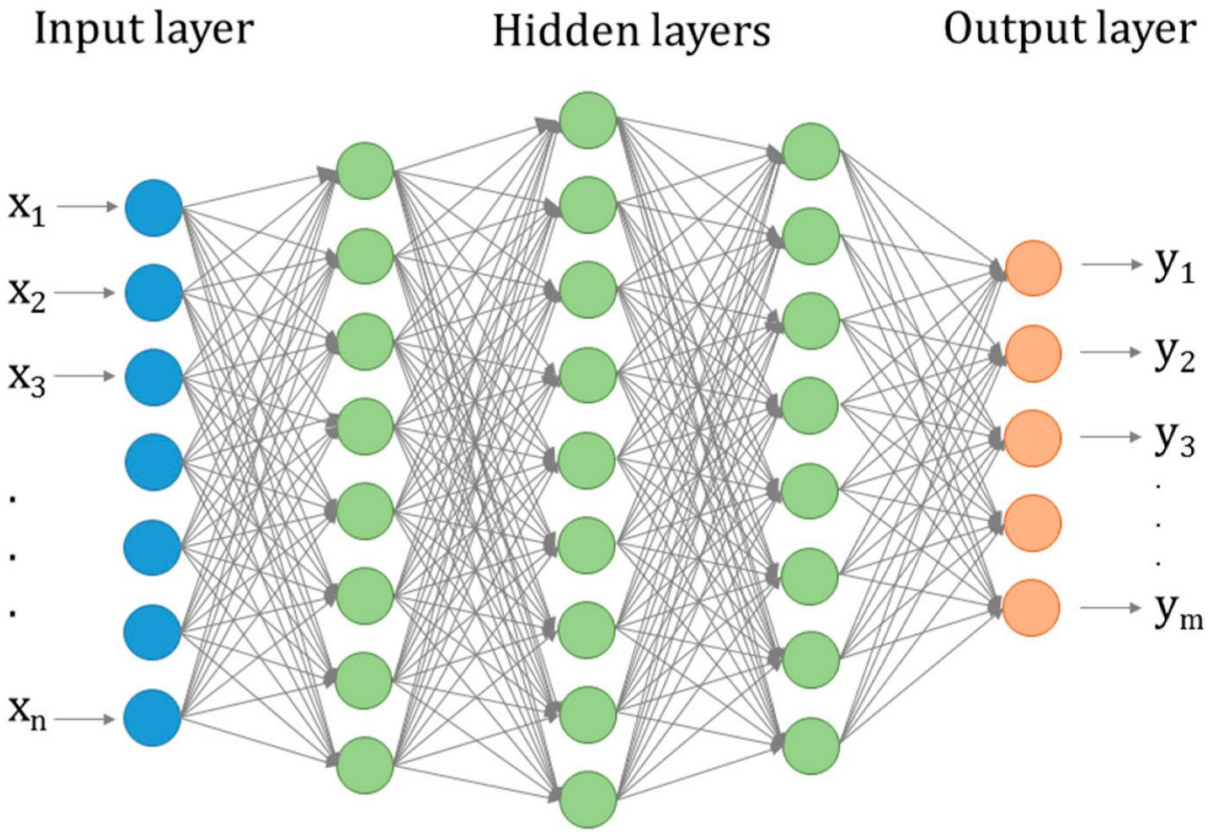
$$y_3 = \begin{cases} 1 & si \quad z_3 \geq 0 \\ 0 & si \quad z_3 < 0 \end{cases}$$



Ajouter plusieurs couches permet d'obtenir des modèle non linéaires plus complexes!!

… Structure des ANNs

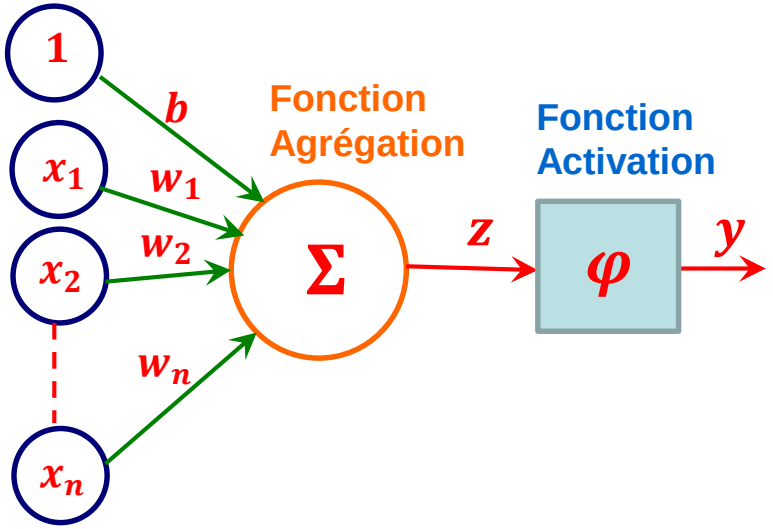
- Le perceptron multicouche (**MLP**: Multilayer Perceptron), est une architecture fondamentale des réseaux de neurones artificiels.
- C'est l'un des types de réseaux de neurones les plus simples et constitue souvent une introduction à l'apprentissage profond.



3. Fonctions d'activation

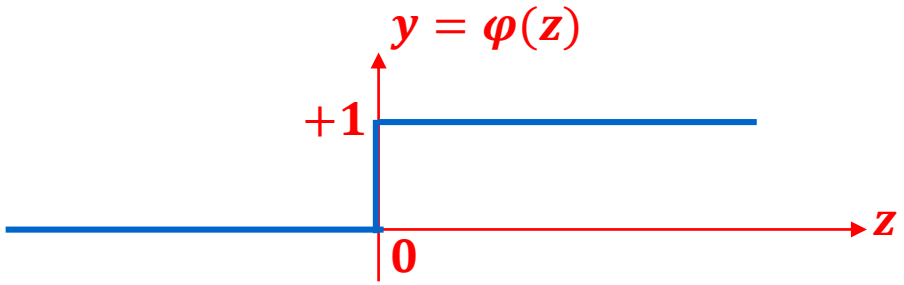
3.1. Fonction Seuil (Threshold):

- Le perceptron de base (original) utilise une **fonction seuil (step function)** comme fonction d'activation



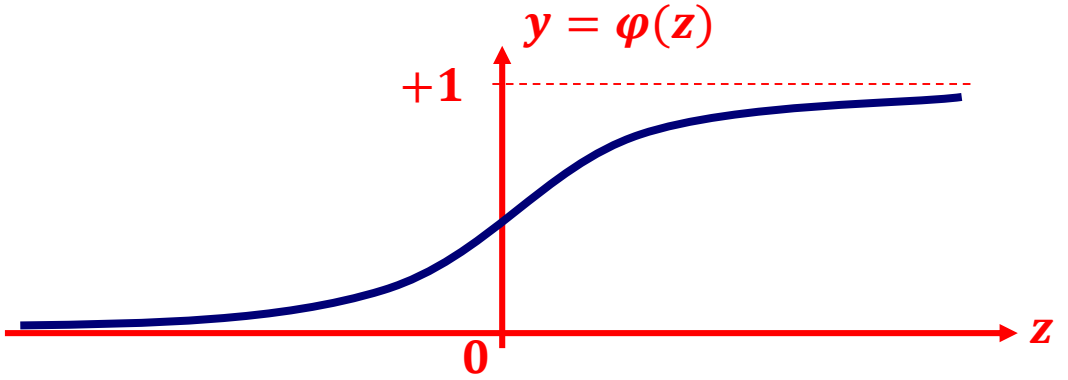
$$y = \varphi(z) = \begin{cases} 1 & si \quad z \geq 0 \\ 0 & si \quad z < 0 \end{cases}$$

- La fonction seuil est non différentiable (dérivable), ce qui rend difficile l'entraînement avec des algorithmes comme la backpropagation



3.2. Fonction Sigmoïde:

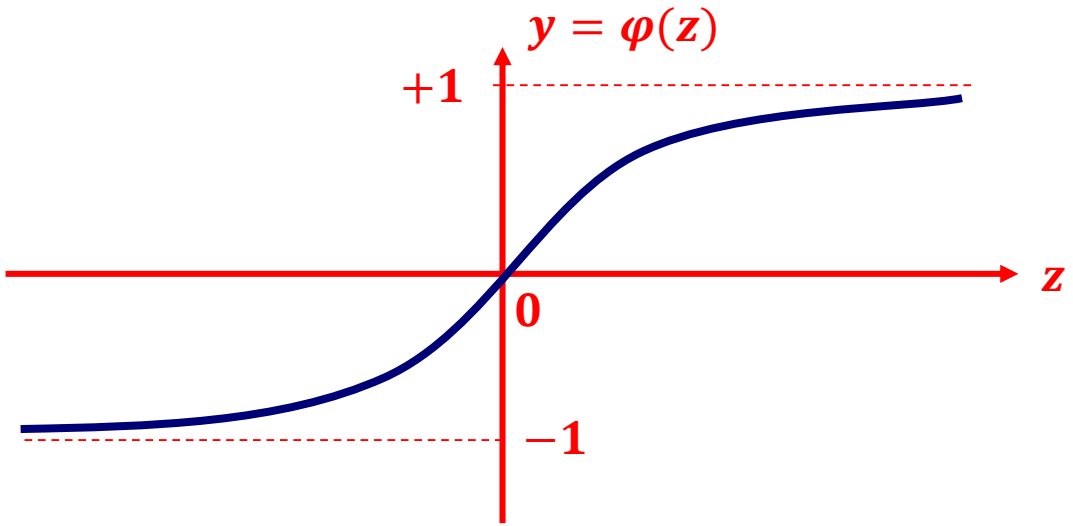
$$y = \varphi(z) = \frac{1}{1 + e^{-z}}$$



- Adaptée pour les probabilités et les problèmes de classification binaire.
- Sensible aux problèmes de gradients qui disparaissent.

3.3. Fonction Tangente Hyperbolique (Tanh)

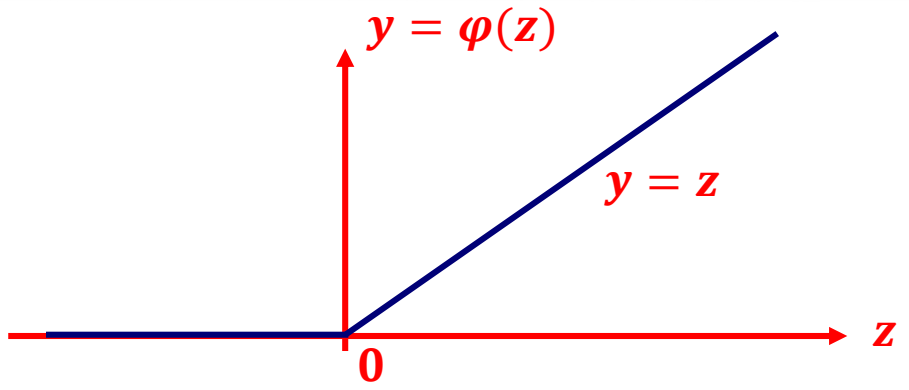
$$y = \varphi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



- Centrée sur zéro, ce qui accélère la convergence.
- Problème similaire de gradients faibles pour des entrées extrêmes.

3.4. Fonction ReLU (Rectified Linear Unit):

$$y = \varphi(z) = \max(0, z)$$

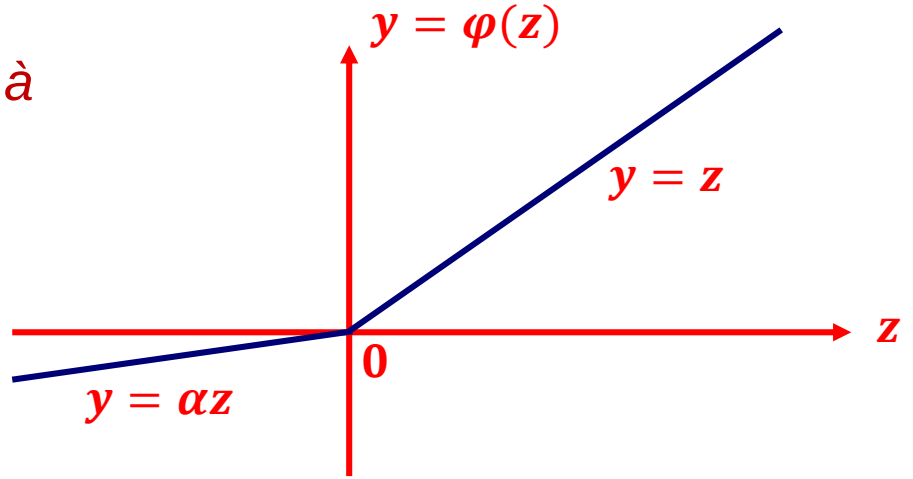


- Sortie : Zéro pour les entrées négatives, linéaire pour les entrées positives.
- Avantage: Simple, efficace et réduit le problème des gradients qui disparaissent.
- Inconvénient: Peut souffrir du problème des neurones "morts" (sorties toujours nulles). **Solution: utiliser la fonction Leaky ReLU**

Un neurone "mort" est un neurone qui cesse de participer à l'apprentissage car sa sortie reste constamment nulle

3.5. Fonction Leaky ReLU:

$$y = \varphi(z) = \begin{cases} z & si \quad z > 0 \\ \alpha z & si \quad z \leq 0 \end{cases} \qquad \alpha > 0$$

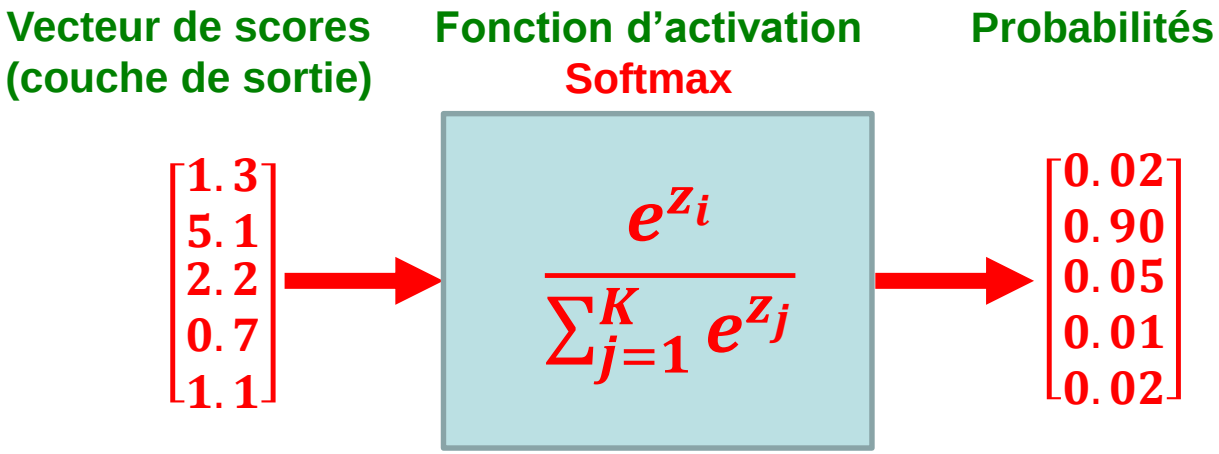


3.6. Fonction Softmax:

- La fonction **Softmax** est une fonction d'activation utilisée dans les réseaux de neurones pour la **classification multiclasse**.
- Elle transforme un vecteur de scores en **probabilités** normalisées (somme = 1).
- **Comparaison avec Sigmoïde** :
  - **Sigmoïde** → Classification **binaire** (sortie entre 0 et 1 pour une seule classe).
  - **Softmax** → Classification **multiclasse** (probabilités pour plusieurs classes).

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

K : Nombre total de classes dans le problème de classification



4. La fonction Coût

4.1. Définition:

- La **fonction coût** (aussi appelée fonction de perte ou **loss function** en anglais) est un élément central dans le processus d'entraînement des réseaux de neurones profonds.
- Elle mesure **l'écart entre les prédictions du modèle** et les **valeurs réelles** attendues (les étiquettes ou outputs).

4.2. Rôle de la fonction coût :

1. **Évaluer la performance du modèle** :
   - Plus la valeur de la fonction coût est faible, meilleure est la précision du modèle.
   - Une valeur élevée indique un écart important entre les prédictions et les résultats réels.
2. **Guider l'apprentissage** :
   - L'algorithme d'optimisation (souvent basé sur la méthode de descente de gradient) utilise la dérivée partielle de la fonction coût pour ajuster les poids et biais du réseau.

4.3. Principaux types de fonctions coût:

$$z = w \cdot x + b = w_1 x_1 + w_2 x_2 \cdots + w_n x_n + b$$

$\hat{y}_i = \varphi(z)$: **Prédiction** du modèle pour l'exemple i

$\varphi(\cdot)$. **Fonction d'activation**

1. **Pour des tâches de classification** : La **log loss** (entropie croisée)
   - Classification multi-classe (K quelconque):

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_{i,j}\log(\hat{y}_{i,j})$$

- $y_{i,j}$: Indicateur binaire (1 si l'exemple *i* appartient à la classe *j*, sinon 0).
- *m* : Nombre total d'exemples d'entrainement.
- *K*: Nombre de Classes

   - Pour une classification binaire (**K=2**) :

$$J = -\frac{1}{m}\sum_{i=1}^{m}[y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)]$$

y_i : étiquette réelle (0 ou 1).

4.3. Principaux types de fonctions coût:

1. **Pour des tâches de régression** :

- Erreur quadratique moyenne (**Mean Squared Error, MSE**)

$$J = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

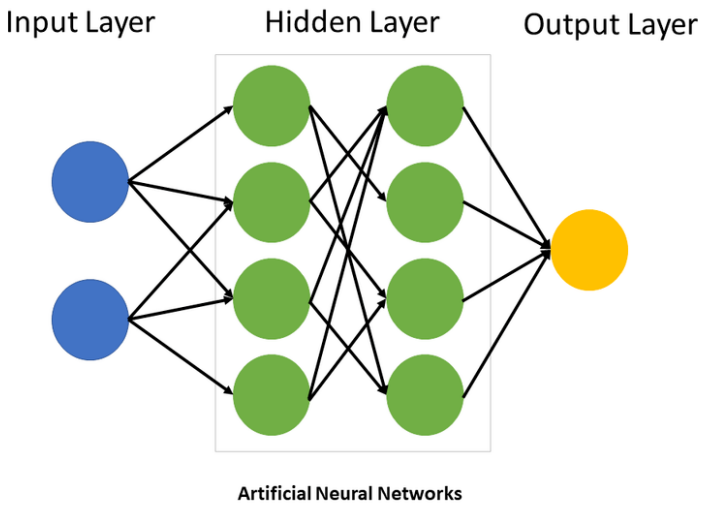
- Erreur absolue moyenne (**Mean Absolute Error, MAE**)

$$J = \frac{1}{m}\sum_{i=1}^{m}|\hat{y}_i - y_i|$$

5. Apprentissage des ANN

5.1. Qu'est-ce que l'apprentissage ?

- Un réseau de neurones est composé de **neurones artificiels** organisés en couches :
  - **Couches d'entrée** : Reçoivent les données initiales.
  - **Couches cachées** : Effectuent les calculs complexes.
  - **Couches de sortie** : Produisent les résultats.

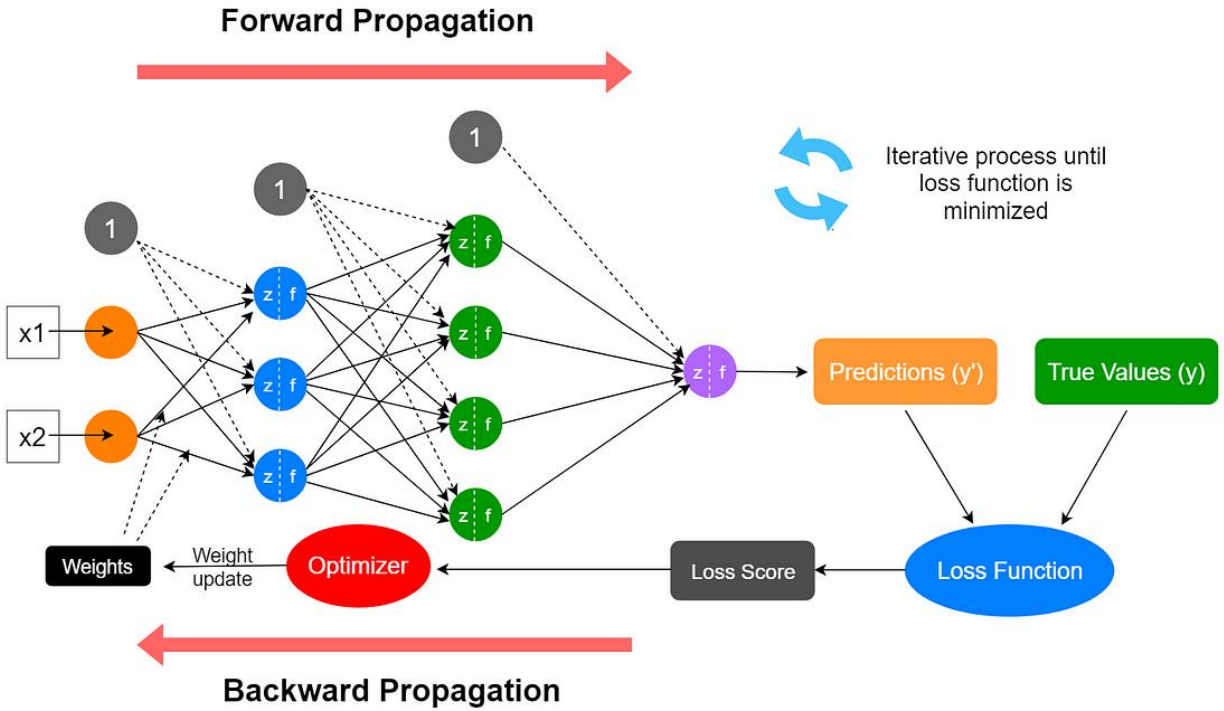


Artificial Neural Networks

- **Apprentissage**: Processus d'adaptation des **poids** w_{ij} des connexions entre les neurones et du biais b pour minimiser l'erreur entre les prédictions et les résultats attendus.
- Deux étapes principales :
  - **Propagation avant (forward propagation)** : Calcul des prédictions.
  - **Propagation arrière (backpropagation)** : Ajustement des poids grâce à l'algorithme de descente de gradient.

5.2. Étapes principales de la backpropagation:

1. **Propagation avant (Forward Propagation)** :
   - Les données d'entrée traversent le réseau.
   - À chaque couche, les neurones calculent une valeur en combinant les entrées, les poids, et la fonction d'activation.
   - Une prédiction finale est produite par le réseau.



5.2. Étapes principales de la backpropagation:

2. **Calcul de l'erreur** :
   - L'erreur est calculée en comparant la sortie du réseau avec la sortie attendue, à l'aide d'une fonction de perte (Cost Function) (par ex., erreur quadratique moyenne, entropie croisée).

3. **Propagation arrière de l'erreur (Backward Propagation)** :
   - L'erreur est propagée depuis la couche de sortie vers les couches précédentes, en calculant les gradients de la fonction de perte par rapport aux poids.
   - On mesure comment la fonction coût varie par rapport à chaque couche.

4. **Mise à jour des poids et du biais** :
   - Les poids et le biais sont ajustés en utilisant un algorithme d'optimisation comme la **descente de gradient**.

On répète ces 4 étapes jusqu'à ce que la fonction coût soit minimisée

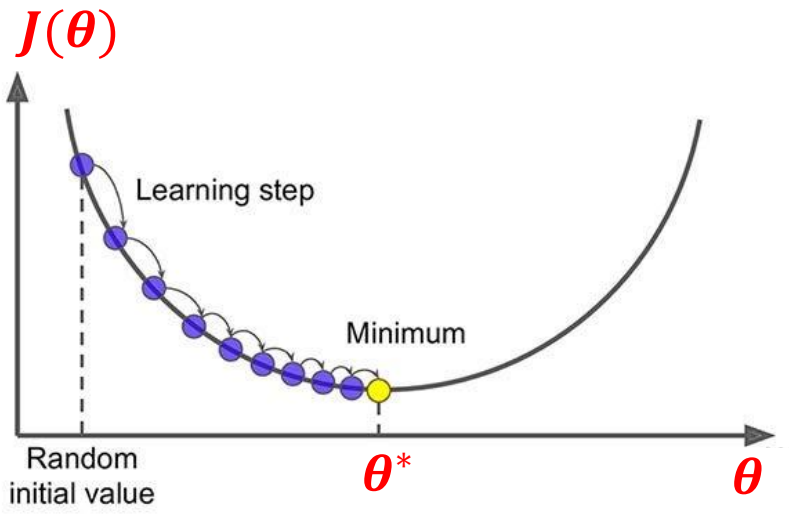
6. Algorithme Descente de Gradient

6.1. Définition :

- L'**algorithme de descente de gradient** est une méthode d'optimisation fondamentale utilisée pour minimiser une fonction coût (ou fonction de perte) dans les réseaux de neurones et d'autres modèles d'apprentissage automatique.

6.2. Principe de base :

- **Objectif** : Trouver les **paramètres optimaux** (poids w_{ij} et biais b_i du réseau) qui minimisent la fonction coût $J(\theta)$, où θ représente les paramètres du modèle ($\theta = (w, b)$).
- **Idée principale** :
  - Partir d'une configuration initiale des paramètres.
  - Calculer la **dérivée partielle** (ou gradient) de la fonction coût par rapport à chaque paramètre.
  - Mettre à jour les paramètres dans la direction opposée au gradient, car le gradient indique la direction d'augmentation de la fonction coût.



6.3. Étapes de l'algorithme :

1. **Initialisation :**
   - Initialiser les paramètres (w, b) avec des valeurs aléatoires ou définies.
   - Choisir un taux d'apprentissage η.

2. **Répéter jusqu'à convergence :**
   - **Calculer la fonction coût :** Évaluer $J(w, b)$ en utilisant les données d'entraînement.
   - **Calculer le gradient :** Trouver $\nabla J(w, b)$, c'est-à-dire la dérivée partielle de J par rapport à chaque paramètre ($\frac{\partial J}{\partial w}$, $\frac{\partial J}{\partial b}$).
   - **Mettre à jour les paramètres :** Appliquer la règle de mise à jour :

$$w = w - \eta \frac{\partial J}{\partial w} \qquad b = b - \eta \frac{\partial J}{\partial b}$$

3. **Arrêt :**
   - Lorsque $J(w, b)$) converge (les changements deviennent négligeables).
   - Ou après un nombre fixe d'itérations.

w : est le poids.
η : est le taux d'apprentissage.
$\frac{\partial J}{\partial w}$: est le gradient de la fonction coût J par rapport à w
$\frac{\partial J}{\partial b}$: est le gradient de la fonction coût J par rapport à b

6.4. Calcul pratique des Dérivées partielles de la fonction coût:

Exemple:

- Fonction de coût: **Log Loss (Binary Cross-Entropy):** $J = -\frac{1}{m}\sum_{i=1}^{m}[y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)]$
- Fonction d'activation **: Sigmoïde :** $\varphi(z) = \frac{1}{1+e^{-z}}$ $z = w_1x_1 + w_2x_2 \cdots + w_nx_n + b$

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)x_{i,j}$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)$$

Explication des termes :

- $\hat{y}_i = \varphi(z_i)$: la sortie de la fonction sigmoïde pour l'exemple i,
- y_i : la vraie étiquette pour l'exemple i,
- $x_{i,j}$: la valeur de la caractéristique j pour l'exemple i.

7. Vecteurisation des Calculs (Calculs Matriciels)

7.1. Matrice des caractéristiques (ou Feature Matrix en anglais):

- La **vectorisation** des calculs dans les réseaux de neurones permet de manipuler efficacement de grandes quantités de données et d'accélérer l'entraînement.
- En remplaçant les boucles itératives par des multiplications matricielles, on peut profiter de l'efficacité des bibliothèques optimisées pour les calculs numériques, comme **NumPy**, **TensorFlow** ou **PyTorch**, qui sont conçues pour gérer ces opérations de manière efficace sur des **GPU** ou CPU hautes performances
- si vous avez un jeu de données avec $\boldsymbol{m}$ exemples d'entraînement et $\boldsymbol{n}$ caractéristiques, alors vous pouvez organiser vos données sous la forme d'une matrice $\boldsymbol{X}$ de taille $\boldsymbol{m \times n}$, où chaque ligne représente un exemple d'entraînement et chaque colonne une caractéristique.

$$\boldsymbol{X} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$$

$$\boldsymbol{X} \in \mathbb{R}^{m \times n}$$

7.2. Combinaison linéaire des entrées pondérées:

$$Z = XW + b$$

$$Z = \begin{pmatrix} z^{(1)} \\ z^{(2)} \\ \vdots \\ z^{(m)} \end{pmatrix} \in \mathbb{R}^{m \times 1} \qquad W = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \in \mathbb{R}^{n \times 1}$$

7.3. Fonction de coût:

$$J(W, b) = -\frac{1}{m} Y^T log(\hat{Y}) + (1 - Y)^T log(1 - \hat{Y})$$

$$Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix} \in \mathbb{R}^{m \times 1} \qquad \hat{Y} = \varphi(Z) = \in \mathbb{R}^{m \times 1}$$

7.4. Dérivées partielles de la fonction de coût:

$$\frac{\partial J(W,b)}{\partial W} = \frac{1}{m} X^T(\hat{Y} - Y)$$

$$\frac{\partial J(W,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i) = \frac{1}{m} 1^T(\hat{Y} - Y)$$

$$1 = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathbb{R}^{m\times 1}$$

7.5. Mise à jour des paramètres:

$$W = W - \eta \frac{\partial J(W,b)}{\partial W}$$

$$b = b - \eta \frac{\partial J(W,b)}{\partial b}$$

$$\frac{\partial J(W,b)}{\partial W} = \begin{pmatrix} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_n} \end{pmatrix} \in \mathbb{R}^{n\times 1}$$

Jacobien

8. Les Variantes des ANN

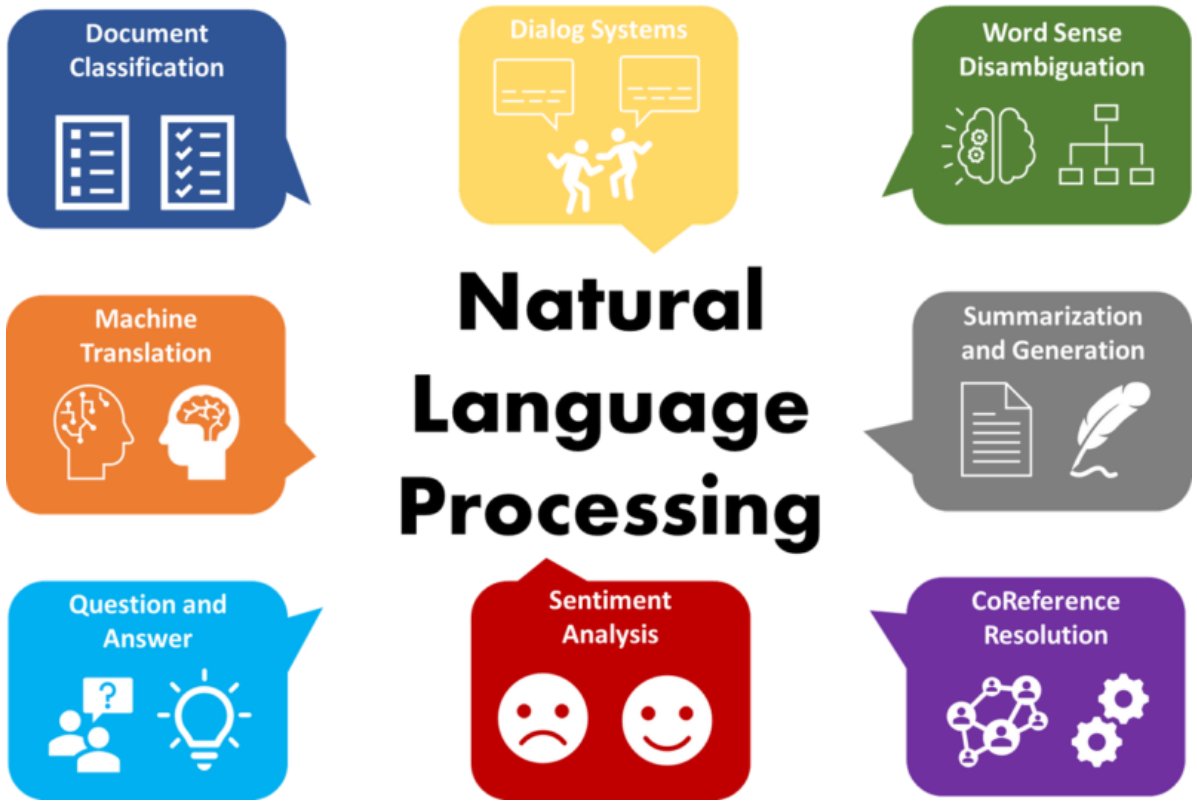
8.1. Les modèles courants:

Les ANN (Réseau Neuronal Artificiel) est un terme générique qui désigne tout type de réseau neuronal inspiré du cerveau humain. Il englobe diverses architectures, y compris :

- ***MLP*** *(Perceptron multicouche): c'est l'architecture de base des ANN*
- ***CNN*** *(Convolutional Neural Network): Reconnaissance d'images, segmentation d'images, détection d'objets.*
- ***RNN*** *(Recurrent Neural Network): Traduction automatique, analyse de sentiment, génération de texte. Présente le problème de* ***Vanishing Gradient*** *: Lorsque l'on entraîne un RNN sur des séquences longues, les gradients qui sont rétropropagés à travers le temps (pendant l'entraînement) peuvent diminuer exponentiellement. Cela empêche le modèle de mémoriser des informations pertinentes sur de longues séquences.*
- ***LSTM*** *(Long Short-Term Memory): Une version améliorée du RNN qui utilise une* ***cellule de mémoire*** *et des* ***portes*** *pour contrôler le flux d'information à travers le réseau.*

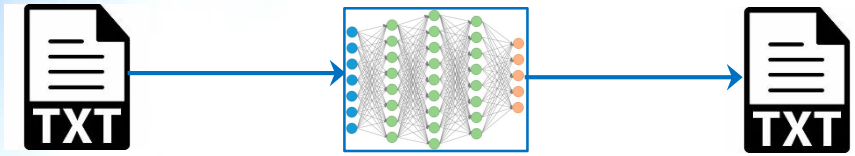
… Les Variantes des ANN

- ***GAN*** *(Generative Adversarial Network): Utilisés pour générer des contenus réalistes comme des images, vidéos, ou sons.*
- ***Transformers****: Pour NLP:* ***Natural Language Processing***
  - ***BERT*** *(Bidirectional Encoder Representations from Transformers) pour les tâches de compréhension.*
  - ***GPT*** *(Generative Pre-trained Transformer) pour la génération de texte.*
  - ***T5*** *(Text-to-Text Transfer Transformer) pour la traduction et d'autres tâches NLP.*

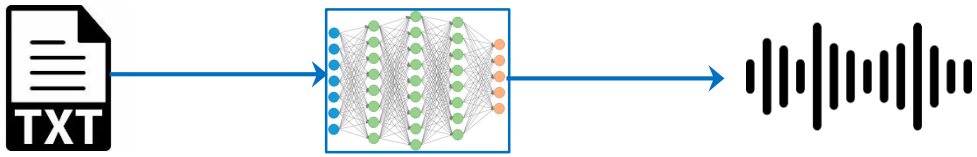


Exemples des modèles IA générative (Generative AI) :

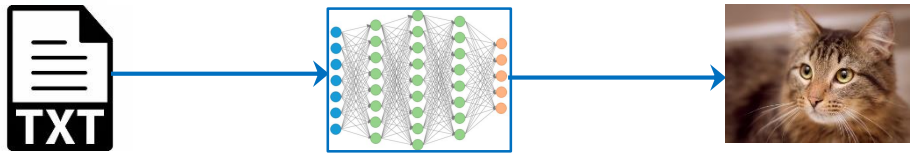
Text-to-Text



Text-to-Audio



Text-to-Image



Audio-to-Text

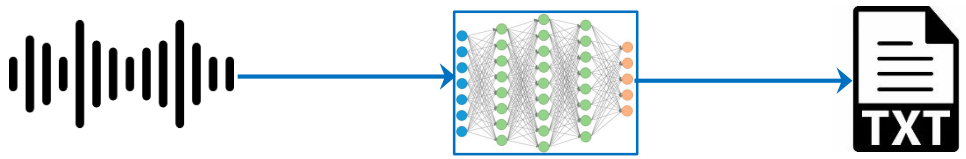
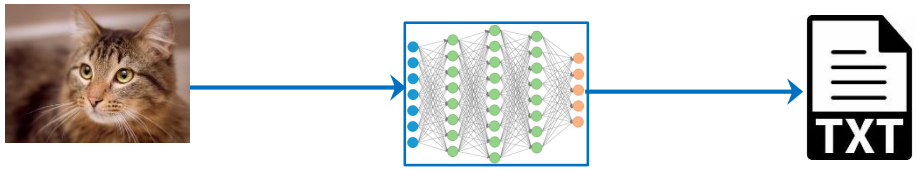
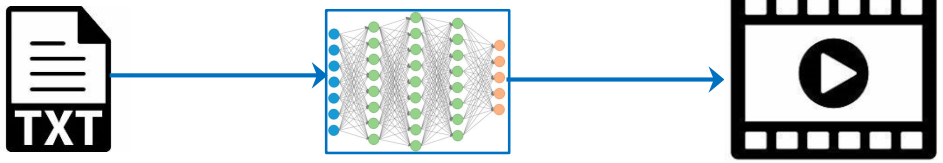


Image-to-Text

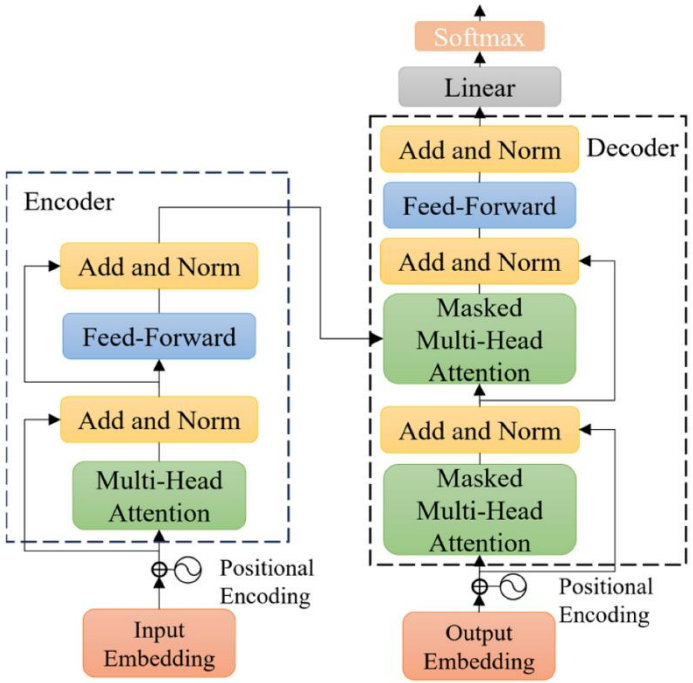


Text-to-Video

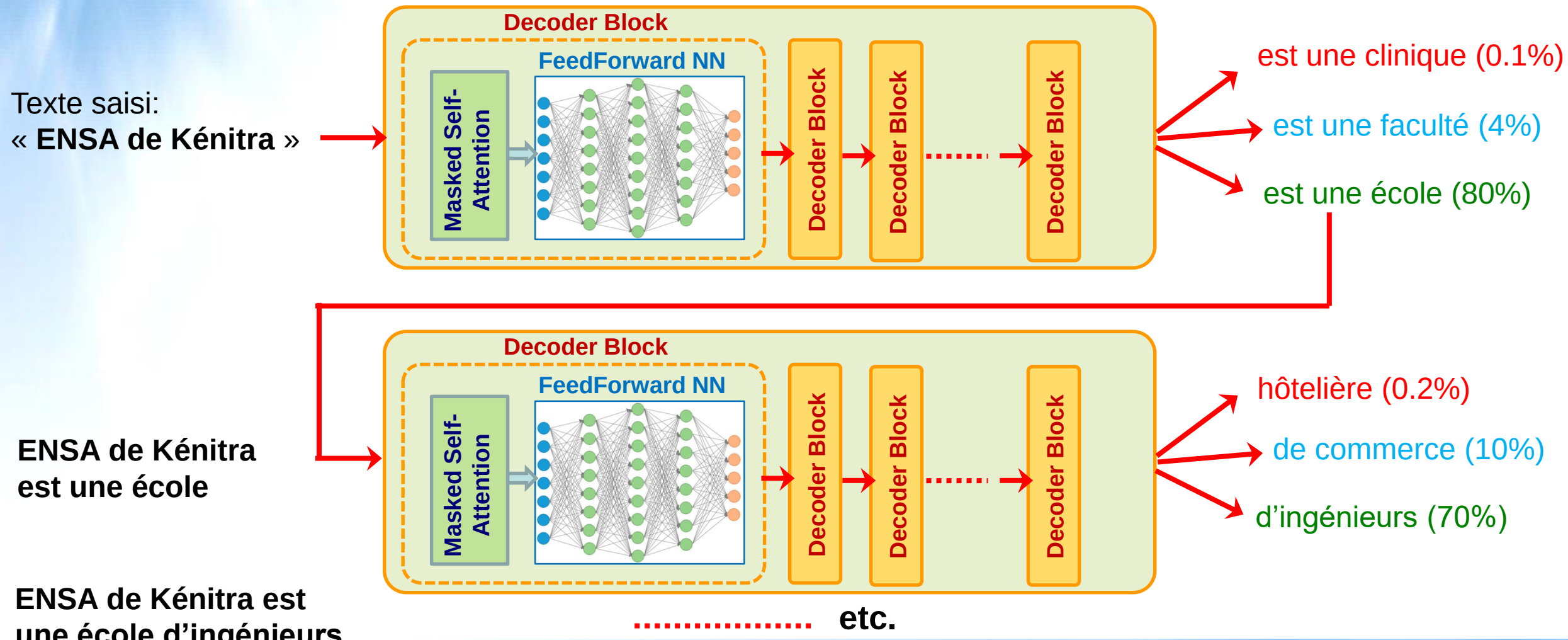


8.2. Les Transformers: les nouveaux modèles de NLP:

- Les **Transformers** ont été introduits pour la première fois en **juin 2017**, dans l'article scientifique intitulé : **"Attention Is All You Need"**, publié par Vaswani.
- C'est la base de nombreux modèles de traitement du langage naturel (NLP) actuels.
- Actuellement, ChatGPT utilise le modèle **GPT-4 d'OpenAI** basé sur l'architecture Transformer.
- Le Transformer repose sur des mécanismes **d'Attention**
- Avec cette architecture de Transformer, ChatGPT par exemple génère des réponses aux requêtes en prédisant le mot suivant en fonction de ceux qui précèdent: Principe de prédiction!
- **Mécanisme d'attention** : Grâce au mécanisme d'attention, le modèle évalue l'importance de chaque mot dans la requête, permettant de se concentrer sur les éléments clés pour comprendre le sens global.



Principe de prédiction des Transformers (Utilisation de Décodeur)



… ANN dans les modèles NLP

ENSA de Kénitra est une école d'ingénieurs qui se distingue par son excellence académique et son engagement dans la formation des leaders de demain. Avec un large éventail de filières innovantes telles que les systèmes embarqués, les énergies renouvelables, l'intelligence artificielle, et les réseaux industriels, l'ENSA de Kénitra forme des ingénieurs polyvalents et compétents, capables de relever les défis technologiques actuels et futurs. Les étudiants bénéficient d'une pédagogie basée sur des projets, un enseignement de qualité, et un partenariat solide avec le monde de l'industrie.

… ANN dans les modèles NLP

- Après les **Transformers**, est-ce que c'est l'ère des **Titans de Google** ?
- **En 2024**: Google AI présente **Titans** : Une architecture d'apprentissage révolutionnaire qui mémorise intelligemment en temps réel !
- Les Titans représentent une nouvelle architecture neuronale qui combine une **mémoire à long terme (s**équences très longues **>2Millions tokens**) et l'Attention des Transformers.

Titans: Learning to Memorize at Test Time

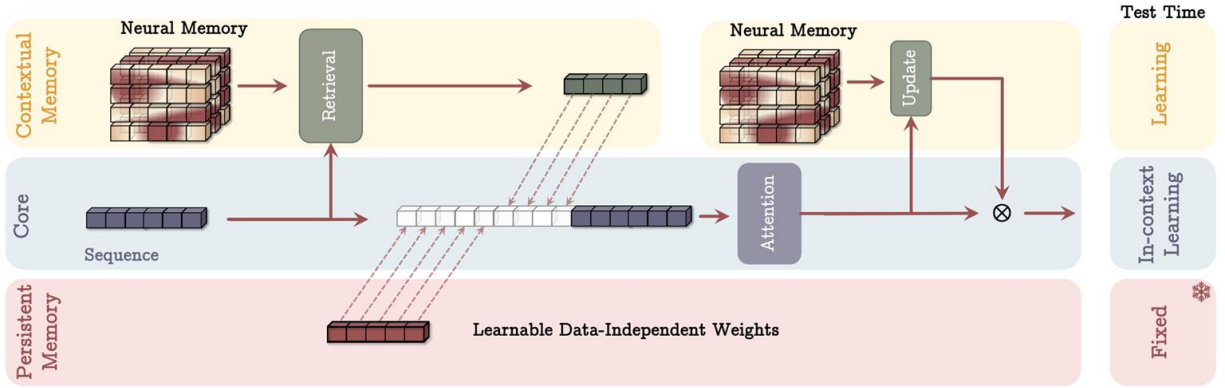
Ali Behrouz[†], Peilin Zhong[†], and Vahab Mirrokni[†]

[†]Google Research

{alibehrouz, peilinz, mirrokni}@google.com

Abstract

Over more than a decade there has been an extensive research effort of how effectively utilize recurrent models and attentions. While recurrent models aim to compress the data into a fixed-size memory (called hidden state), attention allows attending to the entire context window, capturing the direct dependencies of all tokens. This more accurate modeling of dependencies, however, comes with a quadratic cost, limiting the model to a fixed-length context. We present a new neural long-term memory module that learns to memorize historical context and helps an attention to attend to the current context while utilizing long past information. We show that this neural memory has the advantage of a fast

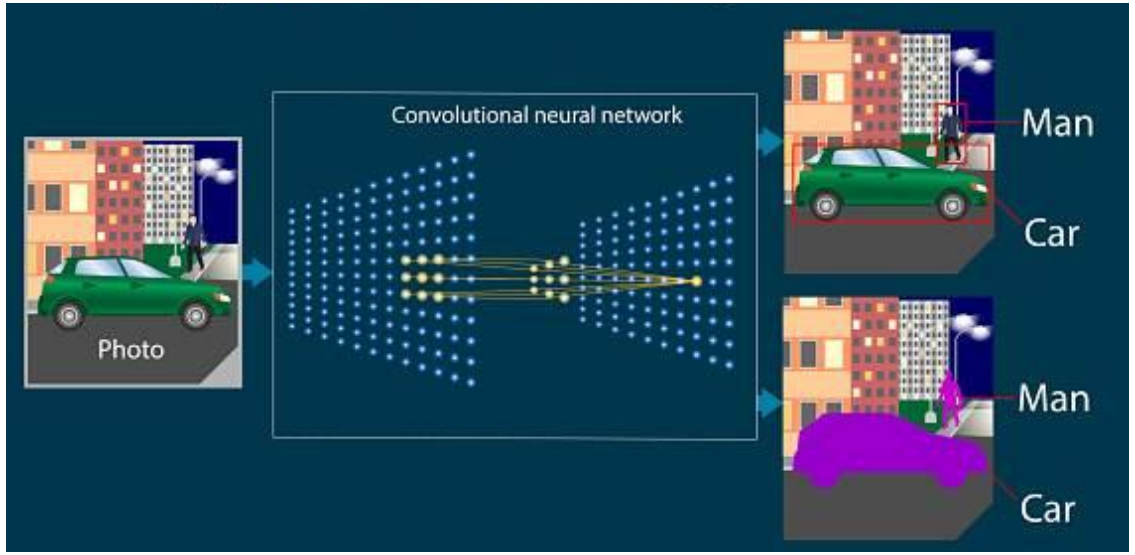


9. Les CNN

9.1. Introduction aux CNN:

Les CNN sont conçus pour extraire automatiquement des caractéristiques locales dans les données, comme des motifs ou des textures, sans nécessiter de prétraitement manuel. Ils sont largement utilisés pour :

- **Vision par ordinateur** : Classification d'images, détection d'objets, segmentation d'images.
- **Traitement du signal** : Analyse audio et reconnaissance vocale.
- **Données structurées** : Séries temporelles et graphes.



9.3. Différences entre MLP et CNN :

Aspect	MLP	CNN
Structure	Réseau de neurones entièrement connecté : chaque neurone d'une couche est relié à tous ceux de la suivante.	Réseau composé de couches de convolution qui extraient des caractéristiques locales.
Entrée	Données aplaties (vecteurs 1D) : nécessite que les images soient transformées en un vecteur.	Données structurées (matrices 2D ou 3D) : conserve la structure spatiale des images.
Paramètres	Nombre de paramètres élevé pour des entrées de grande dimension.	Moins de paramètres grâce au partage des poids dans les couches de convolution.

9.3. Différences entre MLP et CNN :

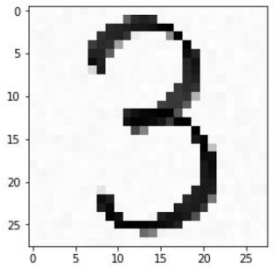
Scénario	MLP	CNN
Données tabulaires (ex. fichiers CSV)	✓	✗
Images, vidéos, ou données structurées	✗	✓
Séries temporelles	✓ (avec mise en forme adéquate)	✓ (si les relations locales sont importantes)

9.4. Pourquoi le MLP n'est pas adapté pour les images?

- Les images contiennent souvent un grand nombre de pixels. Par exemple, une image de taille 28×28 (comme celles de la base MNIST) contient 784 pixels, ce qui signifie 784 entrées pour le MLP.
- Une image HD (1920×1080) contient plus de 2 millions de pixels.
- Le MLP traite chaque pixel individuellement comme une entrée.
- Si chaque pixel est connecté à tous les neurones de la couche suivante (connections fully connected), le réseau aura un nombre énorme de paramètres, ce qui entraîne des besoins en calcul et en mémoire très élevés.

9.4. Pourquoi le MLP n'est pas adapté pour les images?

Exemple: Pour une image avec 1000 neurones seulement, nous aurons:



0.0008 M Pixels
28x28, 8bits

→ **784.000 Paramètres**

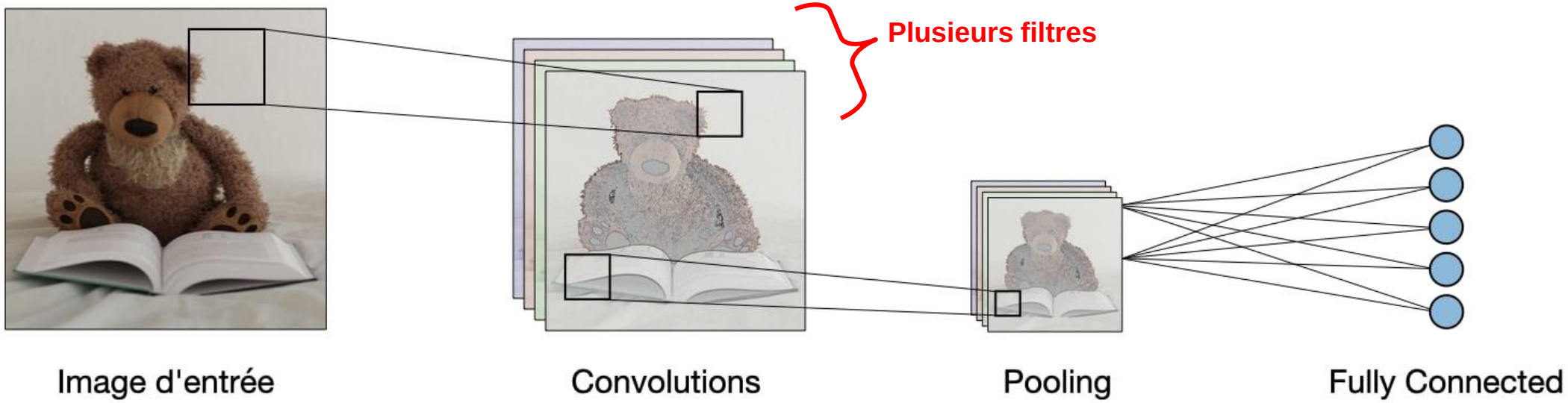
24 M Pixels
(r,v,b) 3x8bits

→ **72.10^9 Paramètres !!!!**

→ **C'est pourquoi on préfère utiliser les CNN au lieu des MLP pour les images et les vidéo**

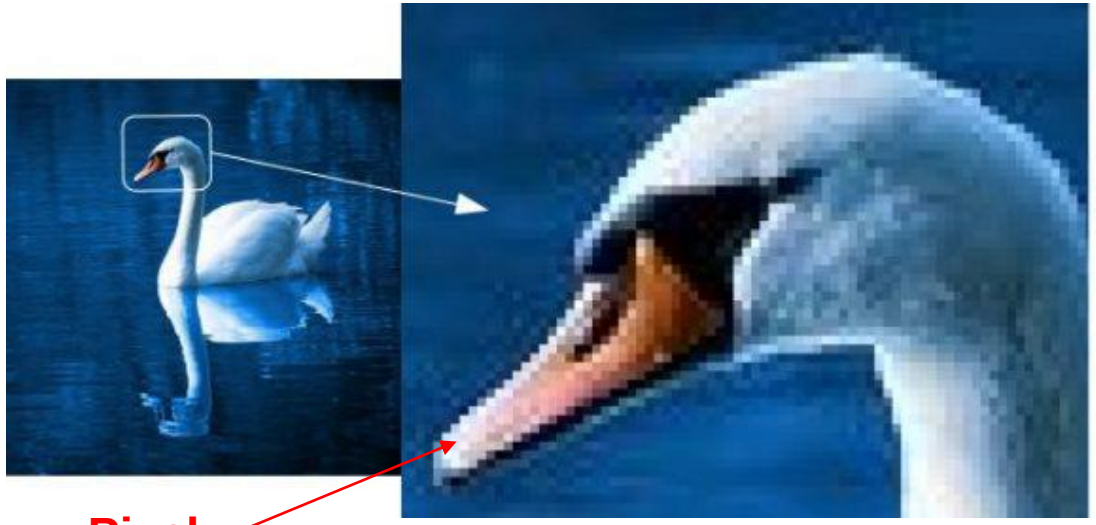
9.5. Principe de Convolution dans les CNN :

- Le **principe de convolution** dans les réseaux de neurones convolutionnels (**CNN**) repose sur l'utilisation de filtres (**ou noyaux**) pour extraire des caractéristiques pertinentes des images.
- Cette opération réduit le nombre de paramètres et capte efficacement les relations spatiales dans les données.
- Chaque filtre extrait des caractéristiques différentes (bords, textures, motifs...)

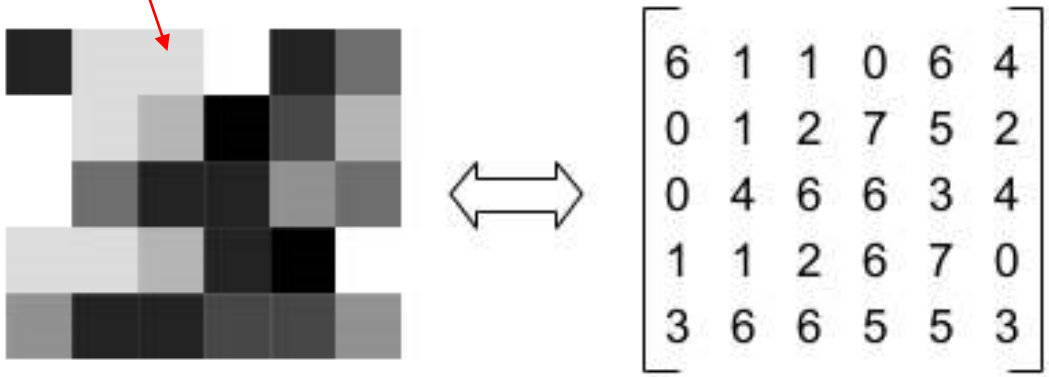


9.6. L'image Numérique:

- Une image numérique est une représentation visuelle constituée de pixels, chacun ayant une valeur définissant sa couleur ou son intensité, créée et stockée sous forme de données numériques.



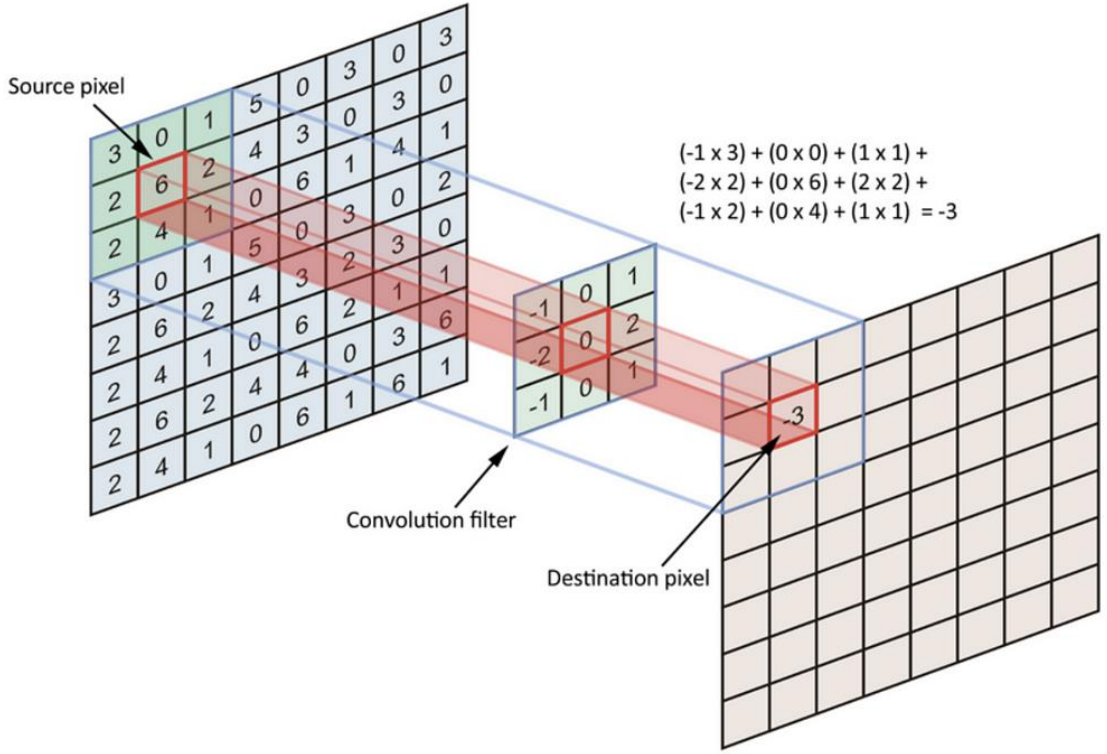
- Une image en niveau de gris est une image numérique où chaque pixel est représenté par une valeur d'intensité lumineuse sur **8 bits**, variant de **0** (**noir**) à **255** (**blanc**).



9.7. Étapes clés du principe de convolution :

1. **Filtre ou noyau (Kernel)**
   - Un **filtre** est une petite matrice (par exemple, **3×3** ou **5×5**) avec des poids appris.
   - Il glisse (s'applique) sur toute l'image (ou une matrice intermédiaire) pour produire une **feature map**.
   - Chaque valeur du filtre est multipliée par les valeurs correspondantes des pixels de l'image, et la somme des produits est calculée.

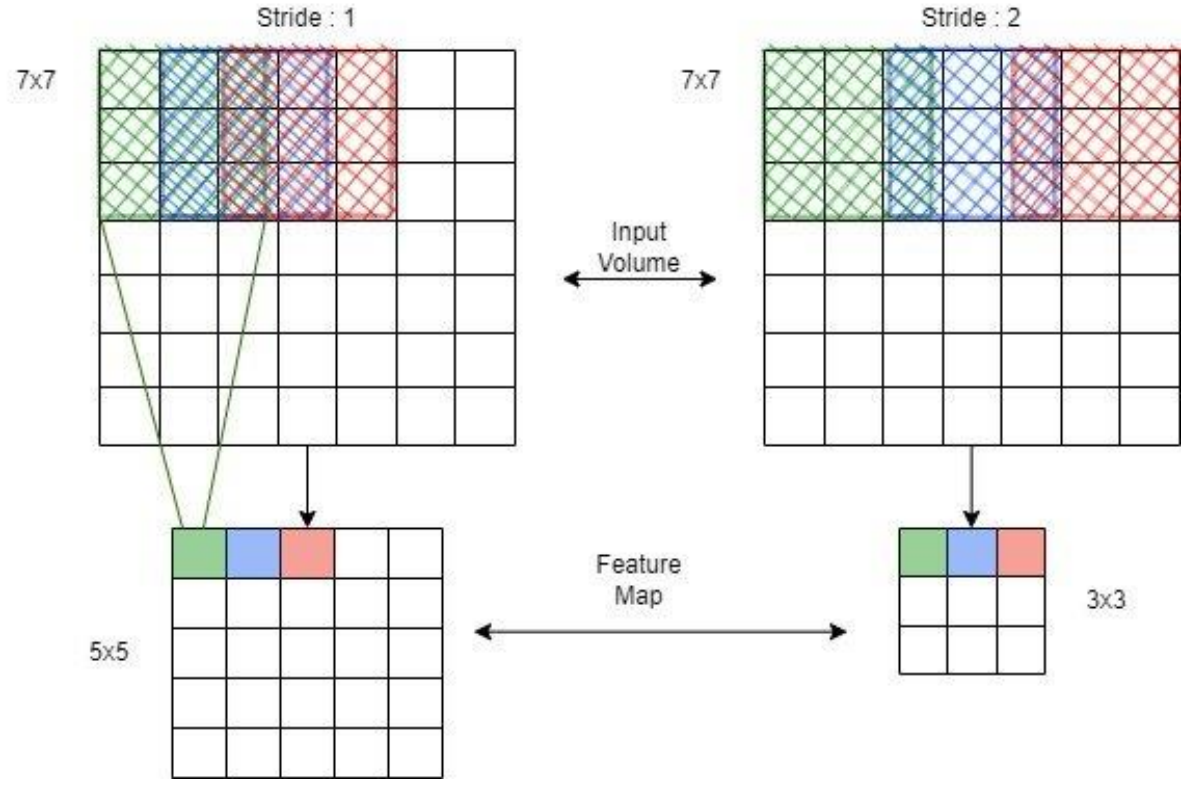
Convolution 2D



9.7. Étapes clés du principe de convolution :

2. Déplacement du filtre (Strides)

- Le filtre est ensuite décalé (glissé) selon une distance appelée **stride**.
- Par exemple, avec un **stride de 1**, le filtre se déplace d'un pixel à la fois. Avec un **stride de 2**, il se déplace de 2 pixels.



9.7. Étapes clés du principe de convolution :

3. Le Remplissage (Padding)

- Pour contrôler la taille des feature maps, des pixels supplémentaires (souvent des zéros) peuvent être ajoutés autour de l'image d'entrée.
- **Valid Padding :** Aucune bordure n'est ajoutée → la taille des feature maps diminue.
- **Same Padding :** Des zéros sont ajoutés pour conserver la même taille que l'image d'entrée.

9.7. Étapes clés du principe de convolution :

Exemple sans padding (P=0) et stride S=1:

Image d'entrée | **Elément Image** | **Kernel (2x2)** | **Convolution**

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \quad \begin{bmatrix}1 & 2\\ 4 & 5\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \begin{bmatrix}-4 & \\ & \end{bmatrix}$$

Multiplication de Hadamard puis sommation ⇒ $\mathbf{1\times 1+2\times 0+4\times 0+5\times(-1)=-4}$

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \quad \begin{bmatrix}2 & 3\\ 5 & 6\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \begin{bmatrix} & -4\\ & \end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \quad \begin{bmatrix}4 & 5\\ 7 & 8\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \begin{bmatrix} & \\ -4 & \end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \quad \begin{bmatrix}5 & 6\\ 8 & 9\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \begin{bmatrix} & \\ & -4\end{bmatrix}$$

⇒ **Finalement la Feature Map obtenue est:** $\begin{bmatrix}-4 & -4\\ -4 & -4\end{bmatrix}$

9.8. Taille de la Feature Map :

La taille de la **feature map** générée après une opération de convolution dépend de plusieurs facteurs :

- **Dimensions de l'image d'entrée ($H_{in} \times W_{in}$)**
- **Taille du filtre (noyau)** ($F_h \times F_w$).
- **Stride** (S) : le décalage entre deux applications successives du filtre.
- **Padding** (P) : ajout de bordures autour de l'image d'entrée pour contrôler la taille de la sortie.
- La formule pour calculer les dimensions de la feature map est la suivante :

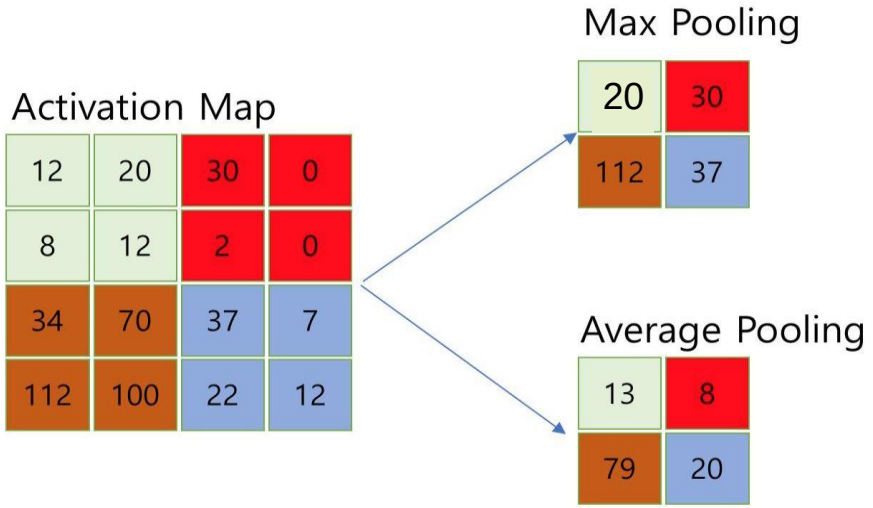
$$H_{out} = \left(\frac{H_{in} - F_h + 2P}{S}\right) + 1$$

$$W_{out} = \left(\frac{W_{in} - F_w + 2P}{S}\right) + 1$$

- H_{in}, W_{in} : Hauteur et largeur de l'image d'entrée.
- F_h, F_w : Hauteur et largeur du filtre (noyau de convolution).
- S : Stride (décalage en pixels entre deux positions du filtre).
- P : Padding (ajout de zéros autour de l'image d'entrée).
- H_{out}, W_{out} : Hauteur et largeur de la feature map.

9.9. Le Sous-Echantillonnage (Pooling):

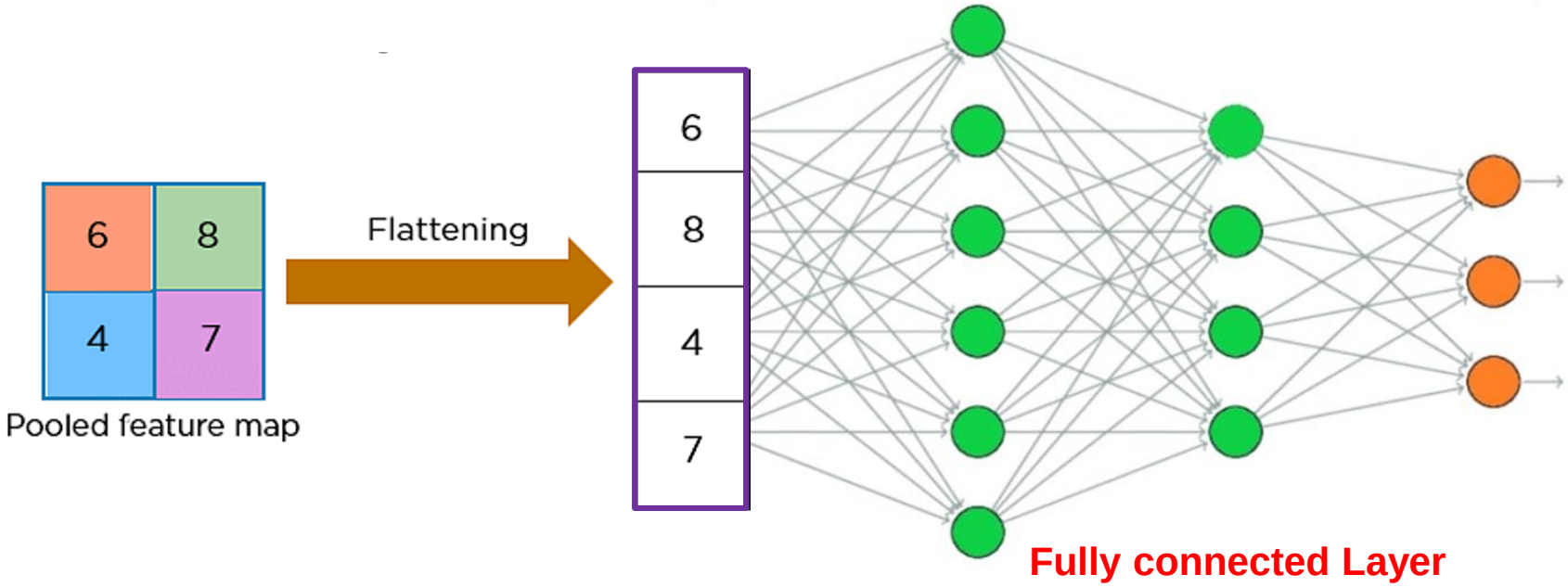
- Le pooling est une opération clé dans les réseaux de neurones convolutifs (CNN) utilisée pour réduire la taille spatiale (hauteur et largeur) des feature maps tout en conservant les informations importantes.
- Cela diminue le nombre de paramètres, améliore l'efficacité du réseau, et aide à gérer les variations dans les données d'entrée.



- **Max Pooling** : Sélectionne la valeur maximale dans une région définie (fenêtre) de la feature map
- **Average Pooling** : Calcule la moyenne des valeurs dans une région définie (fenêtre)

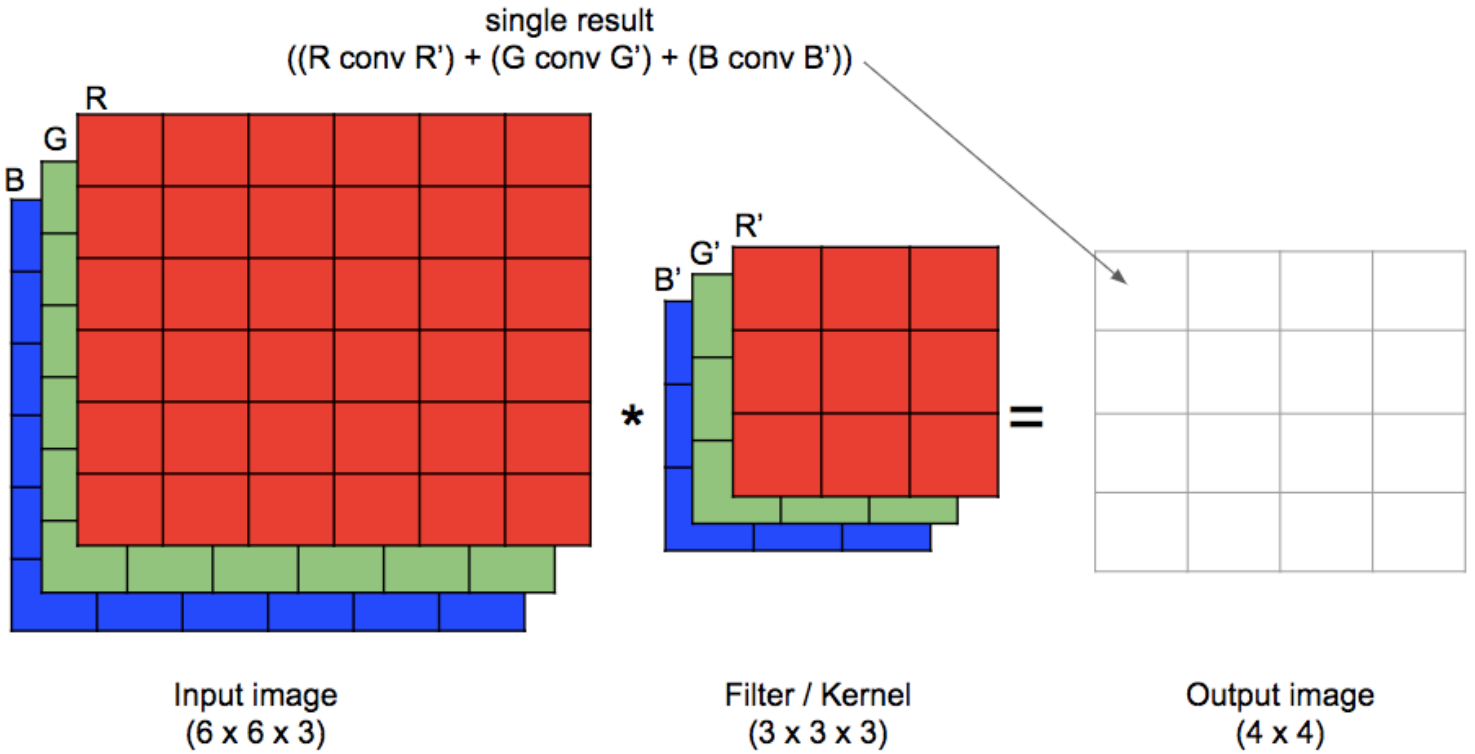
9.10. L'aplatissement (Flattening):

- Le **flattening** consiste à transformer les matrices multidimensionnelles en un vecteur unidimensionnel.
- Cela permet de connecter ces données à une couche entièrement connectée (Dense Layer), qui nécessite une entrée de forme unidimensionnelle

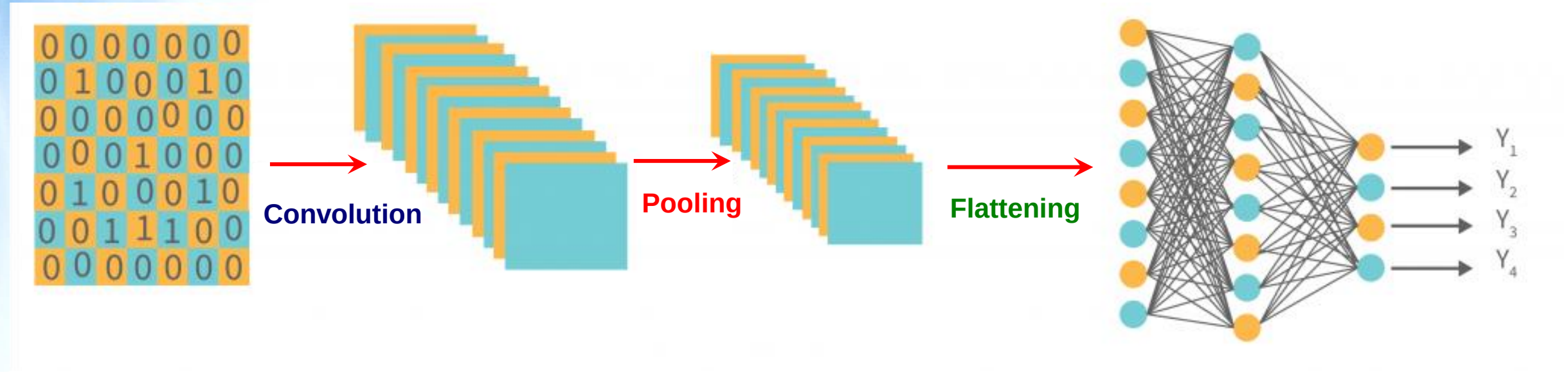


9.11. Convolution 3D:

La convolution 3D dans un CNN est utilisée pour traiter des entrées ayant plusieurs canaux, comme des images en couleur (où chaque image a trois canaux : **rouge**, **vert**, **bleu**).



9.12. Architecture typique des CNN:



10. Bibliothèques Python pour les ANN

Il existe plusieurs bibliothèques Python populaires pour la création et l'entraînement de réseaux de neurones artificiels (ANN). Voici les principales:

Bibliothèque	Développeur	Points forts	Applications principales
TensorFlow	Google	Calcul distribué, support GPU, API haut niveau (Keras)	Vision par ordinateur, NLP, modèles complexes
Keras	François Chollet (initialement) / Google (maintenant)	Facilité d'utilisation, intégré à TensorFlow	Modèles rapides, prototypage d'architectures
PyTorch	Facebook	Calcul dynamique des graphes, flexibilité	Recherche, vision par ordinateur, NLP
scikit-learn	Équipe collaborative (initialement par David Cournapeau)	Modèles classiques, facile à utiliser	Classification, régression, clustering

11. Les GPU pour les ANN

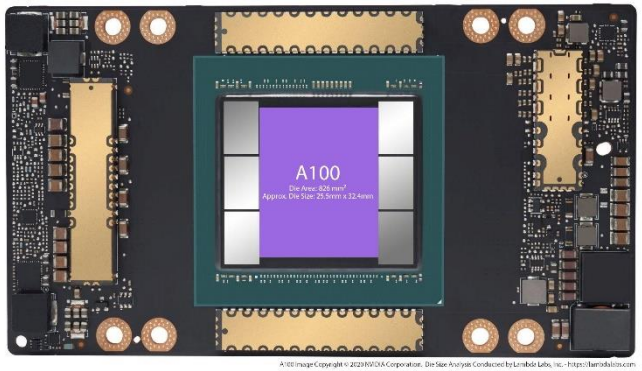
Les **GPU** (Graphics Processing Units) jouent un rôle crucial dans l'entraînement des **réseaux de neurones artificiels** (ANN), en particulier lorsqu'il s'agit de **deep learning** et de tâches d'apprentissage automatique nécessitant des calculs intensifs.

Caractéristique	CPU (Central Processing Unit)	GPU (Graphics Processing Unit)
Architecture	Cœurs limités, optimisés pour des tâches séquentielles	Cœurs nombreux (des milliers), optimisés pour le calcul parallèle
Applications	Tâches générales, calculs séquentiels, gestion de l'OS	Calculs massivement parallèles, deep learning, rendu graphique
Vitesse de traitement	Moins rapide pour les calculs parallèles	Très rapide pour les calculs parallèles (comme la multiplication de matrices)
Mémoire	Mémoire cache (L1, L2, L3), plus faible bande passante	Mémoire haute bande passante (GDDR6/HBM3)
Cœurs	Cœurs puissants mais limités (généralement entre 4 et 32)	Des milliers de cœurs plus simples pour les calculs parallèles
Consommation d'énergie	Consommation modérée, plus faible que le GPU pour des tâches simples	Consommation plus élevée en raison du nombre de cœurs et de la charge de travail parallèle

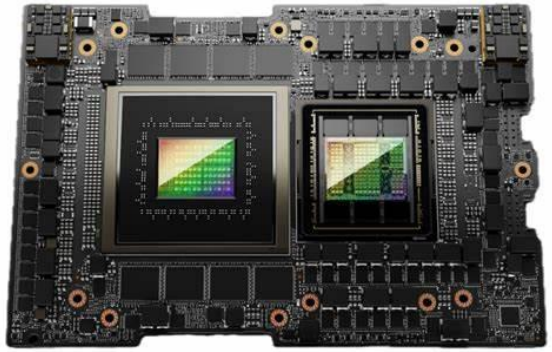
Exemples de GPU pour les ANN:

NVIDIA Tesla V100

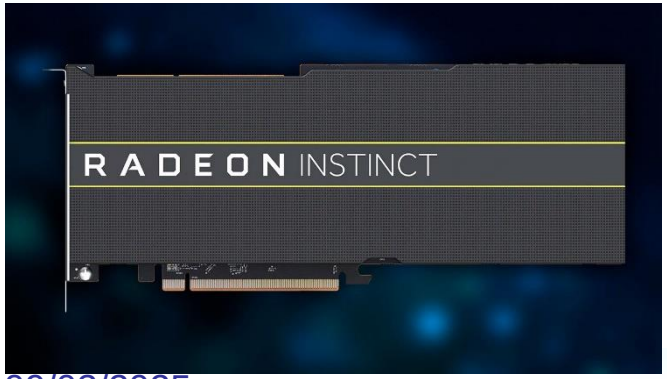
NVIDIA A100 (2021)



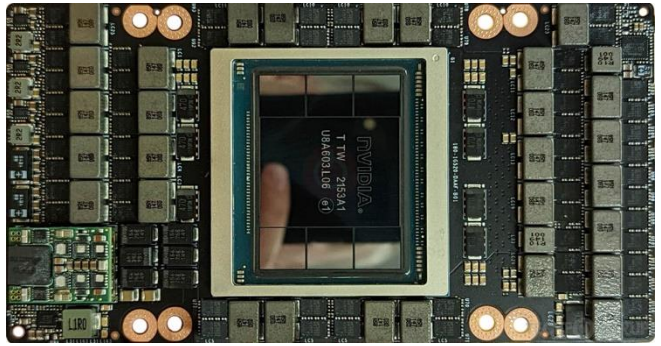
NVIDIA H200 (2024)

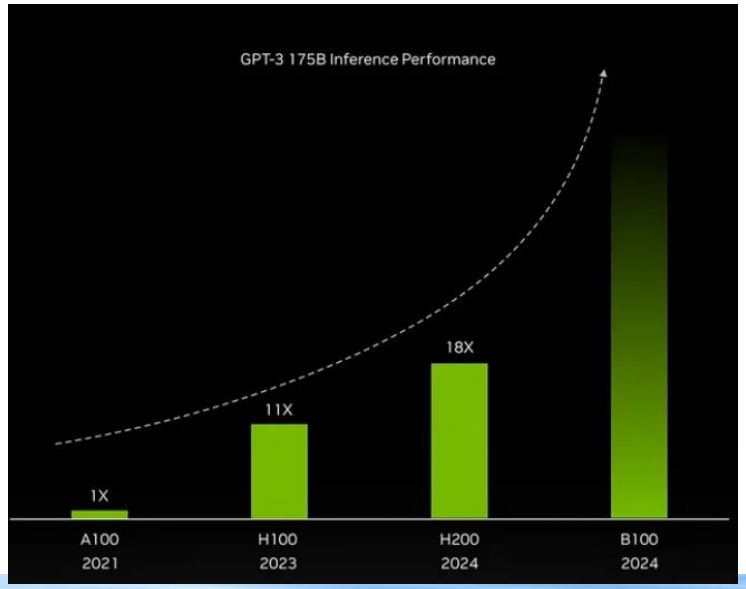


AMD Radeon Instinct MI50 / MI100



NVIDIA H100 (2023)





… Les GPU pour les ANN

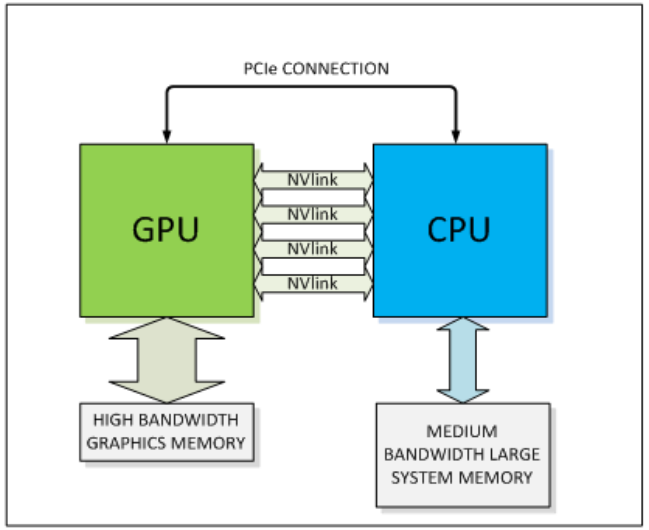
- **xAI : le superordinateur Colossus composé de 100 000 GPU H100** de Nvidia a été monté en Septembre 2024.
- Le projet a coûté environ 3 milliards de dollars et doublera encore de taille avec l'ajout de **50 000 GPU H200**

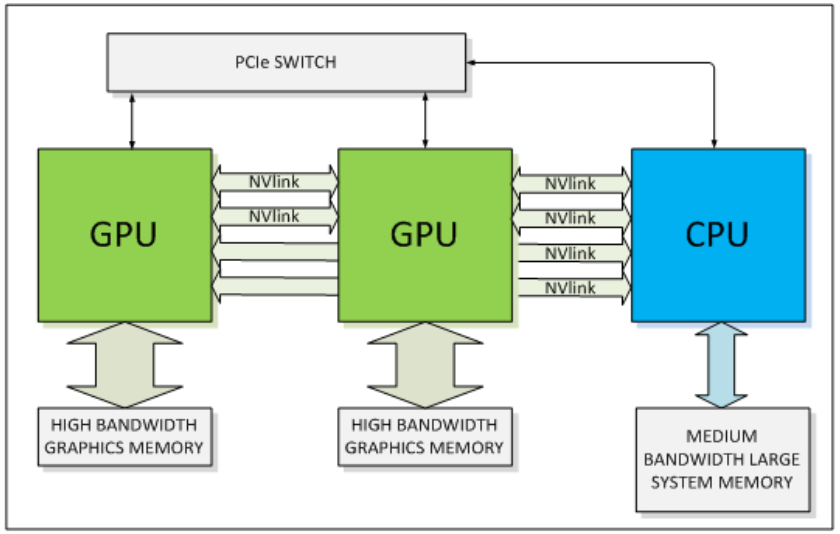
Pourquoi utiliser des GPU pour l'IA ?

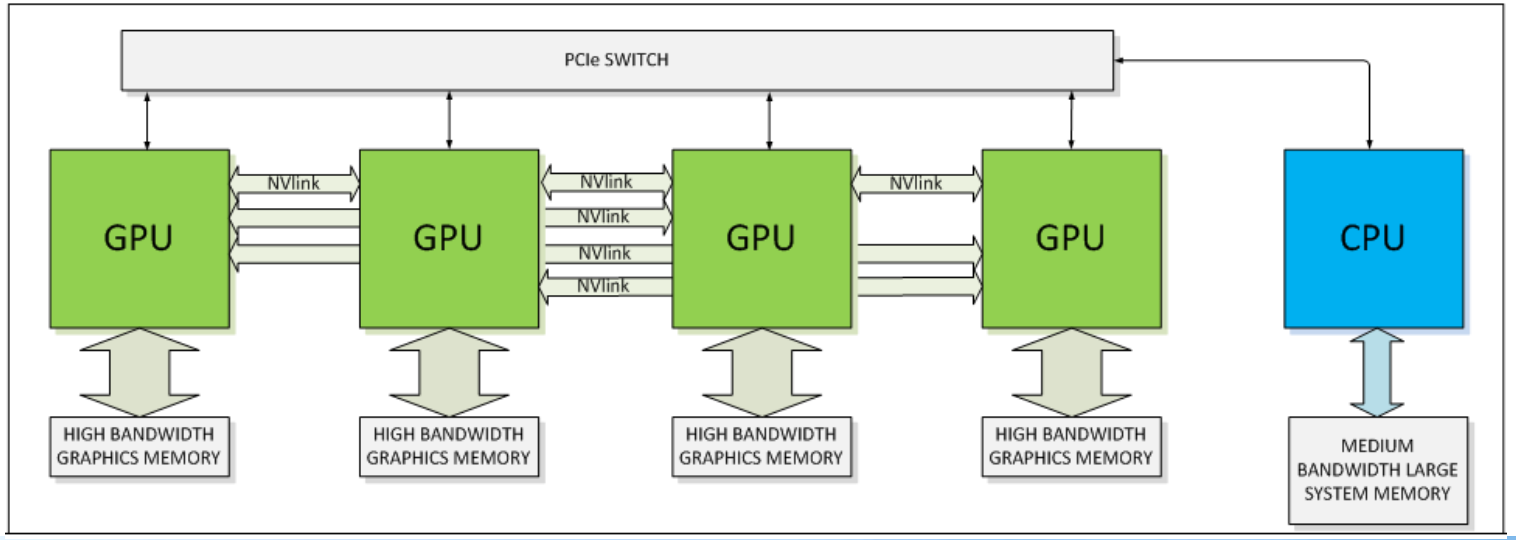
- **Optimisé pour le calcul parallèle** → Exécution de millions d'opérations simultanées.
- **Mémoire rapide** → **HBM3, GDDR6** pour traiter d'énormes volumes de données.
- **Accélération matérielle** → Cores **CUDA**, **Tensor Cores** (NVIDIA), **Matrix Cores** (AMD).

… Les GPU pour les ANN

Schémas de systèmes **NVLink**: technologie d'interconnexion haute vitesse développée par **NVIDIA** pour relier plusieurs **GPU**, **CPU** et d'autres composants dans un système haute performance







Étapes du fonctionnement de l'IA sur GPU:

1. **Prétraitement des données (CPU)**
   - Chargement des données (images, texte, son, etc.).
   - Normalisation et mise en forme des données en **tenseurs** (*Tensor*).
2. **Transfert des données vers le GPU**
   - Le **CPU envoie les données au GPU** via un **bus PCIe** ou une mémoire dédiée (*HBM* ou *GDDR*).
   - L'IA utilise des frameworks comme **CUDA** (NVIDIA) ou **ROCm** (AMD) pour optimiser le calcul.
3. **Exécution des calculs massivement parallèles (GPU)**
   - Les **GPU traitent les opérations matricielles** (produit matriciel, convolution, activation…).
   - Utilisation d'algorithmes d'apprentissage profond (**réseaux neuronaux**).
4. **Mise à jour des poids du réseau neuronal**
   - Une fois le calcul terminé, les résultats sont envoyés au CPU.
   - Le CPU **met à jour les paramètres du modèle** et relance un nouveau cycle d'apprentissage.
5. **Inférence en temps réel**
   - Après entraînement, l'IA peut faire des prédictions directement sur le GPU.
   - Exemple : Un modèle de reconnaissance d'image peut traiter **des millions d'images en temps réel**.

12. Application: Étapes pour entraîner un CNN sur le dataset CIFAR-10

12.1. Objectif:

- Entraîner un réseau de neurones convolutifs (CNN) sur le dataset **CIFAR-10** (Canadian Institute For Advanced Research) en utilisant Google Colab.
- CIFAR-10 est un jeu de données composé de 60 000 images en couleur réparties en 10 classes différentes, avec 6 000 images par classe.
- Les types d'images dans CIFAR-10 sont les suivants : **Avion** (airplane); **Automobile** (car); **Oiseau** (bird); **Chat** (cat); **Cerf** (deer); **Chien** (dog); **Grenouille** (frog); **Cheval** (horse); **Navire** (ship); **Camion** (truck);

12.2. Nouveau Notebook:

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
- Renommer le **CNN-CIFAR-10.ipynb**

12.3. Code à Utiliser dans Google Colab:

1. Importer les bibliothèques nécessaires:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical
```

Pour créer et définir le modèle CNN

Dans le format one-hot, chaque classe est représentée par un vecteur binaire où un seul élément est égal à 1, et tous les autres éléments sont égaux à 0

Pour convertir les étiquettes en format one-hot

2. Charger le jeu de données CIFAR-10:

```
# Charger le jeu de données CIFAR-10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normaliser les images : les valeurs des pixels sont dans l'intervalle [0, 255],
# il est préférable de les normaliser entre [0, 1] pour améliorer les performances du modèle
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convertir les labels en format one-hot (vectorisation des étiquettes)
# Cela transforme les étiquettes en vecteurs de taille 10, avec un '1' à l'indice de la classe.
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

3. Afficher un échantillon d'images:

```
# Afficher un échantillon d'images du jeu de données CIFAR-10
fig, axes = plt.subplots(2, 5, figsize=(10, 5))  # 2 lignes et 5 colonnes
axes = axes.ravel()

# Afficher les 10 premières images
for i in np.arange(0, 10):
    axes[i].imshow(x_train[i])  # Afficher l'image à l'index i
    axes[i].set_title(f"Label: {np.argmax(y_train[i])}")  # Afficher l'étiquette de l'image
    axes[i].axis('off')  # Désactiver les axes pour une meilleure présentation

plt.subplots_adjust(wspace=0.5)
plt.show()
```

4. Construire le modèle CNN:

```
# Définir le modèle CNN
model = models.Sequential([
    # Première couche convolutionnelle avec 32 filtres de taille 3x3, activation ReLU et entrée de forme (32, 32, 3)
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),  # Couches de pooling pour réduire la taille spatiale
    # Deuxième couche convolutionnelle avec 64 filtres de taille 3x3
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    # Troisième couche convolutionnelle avec 64 filtres de taille 3x3
    layers.Conv2D(64, (3, 3), activation='relu'),
    # Aplatir les données pour les couches denses
    layers.Flatten(),
    layers.Dense(64, activation='relu'),  # Couche dense avec 64 neurones et activation ReLU
    layers.Dense(10, activation='softmax')  # Couche de sortie avec 10 neurones pour les 10 classes (activation softmax)
])
# Afficher un résumé du modèle
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36,928
flatten_1 (Flatten)	(None, 1024)	0
dense_2 (Dense)	(None, 64)	65,600
dense_3 (Dense)	(None, 10)	650

5. Compiler le modèle:

```
# Compiler le modèle
model.compile(optimizer='adam',  # Utilisation de l'optimiseur Adam
              loss='categorical_crossentropy',  # Fonction de perte pour la classification multi-classe
              metrics=['accuracy'])  # Mesurer la précision pendant l'entraînement
```

6. Entraîner le modèle:

```
# Entraîner le modèle
history = model.fit(x_train, y_train, epochs=10,  # 10 époques d'entraînement
                    validation_data=(x_test, y_test))  # Validation sur l'ensemble de test
```

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 47ms/step - accuracy: 0.3358 - loss: 1.7813 - val_accuracy: 0.5335 - val_loss: 1.2943
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.5591 - loss: 1.2330 - val_accuracy: 0.5792 - val_loss: 1.1930
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 46ms/step - accuracy: 0.6206 - loss: 1.0760 - val_accuracy: 0.6392 - val_loss: 1.0147
Epoch 4/10
```

Constater l'amélioration de la précision à chaque Epoch

*Une **époque d'entraînement** (ou **epoch** en anglais) correspond à un passage complet de toutes les données d'entraînement à travers le modèle.*
En d'autres termes, pendant une époque, chaque image du jeu de données d'entraînement est utilisée une fois pour ajuster les poids du modèle.

Remarque: Les lots (batches)

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 47ms/step - accuracy: 0.3358 - loss: 1.7813 - val_accuracy: 0.5335 - val_loss: 1.2943
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.5591 - loss: 1.2330 - val_accuracy: 0.5792 - val_loss: 1.1930
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 46ms/step - accuracy: 0.6206 - loss: 1.0760 - val_accuracy: 0.6392 - val_loss: 1.0147
Epoch 4/10
```

- Un lot (**batch**)est un sous-ensemble d'images parmi l'ensemble d'entraînement complet.
- Au lieu de traiter toutes les images d'un coup, on les divise en petites parties appelées **batches**, ce qui permet d'accélérer l'entraînement et de réduire la charge mémoire.
- Les **60 000 images de CIFAR-10** sont réparties comme suit : **50 000 images pour l'entraînement** et **10 000 images pour le test**.
- Avec un **batch size de 32**, cela signifie qu'il y a environ **1563 batches par époque** (50 000 divisé par 32), et les **poids du modèle** sont mis à jour à chaque batch après le calcul de la perte et des gradients.

7. Évaluer la performance du modèle:

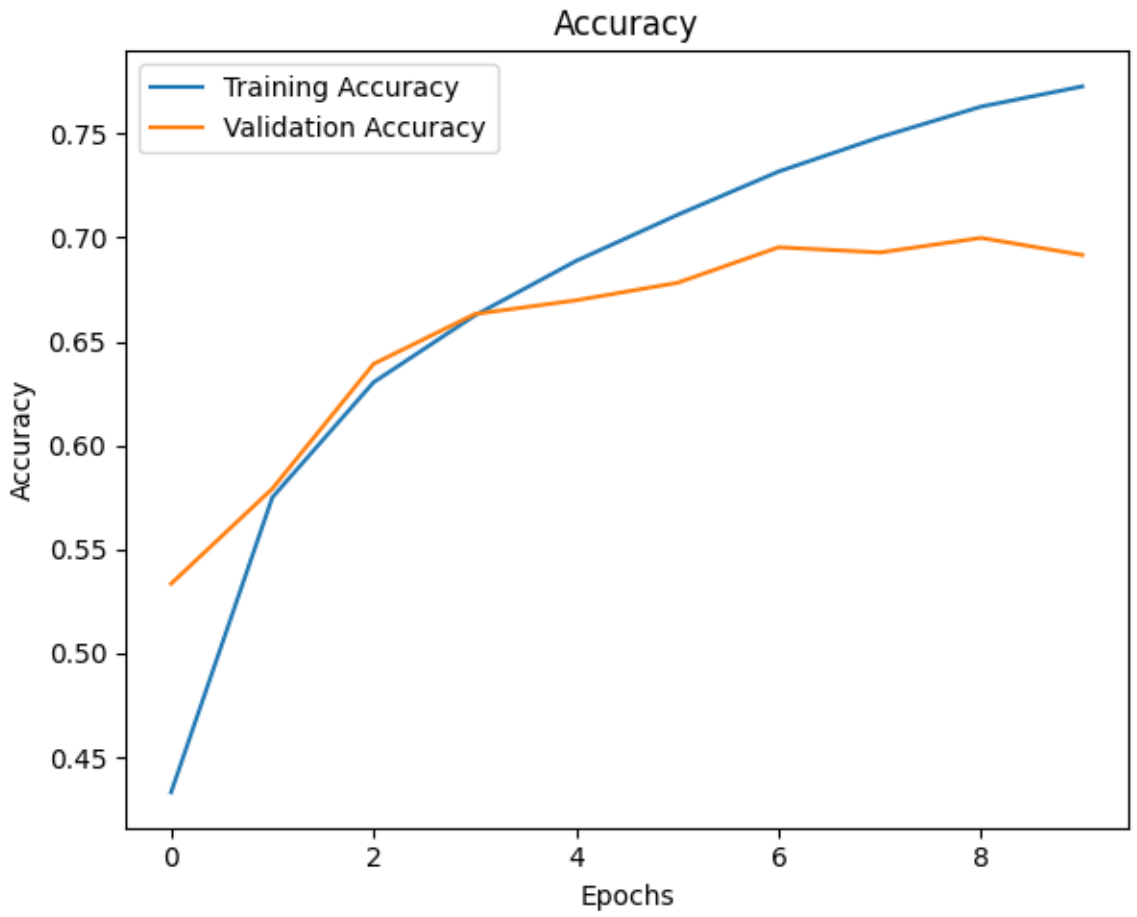
```
# Évaluer le modèle sur les données de test
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")  # Afficher la précision sur les données de test
```

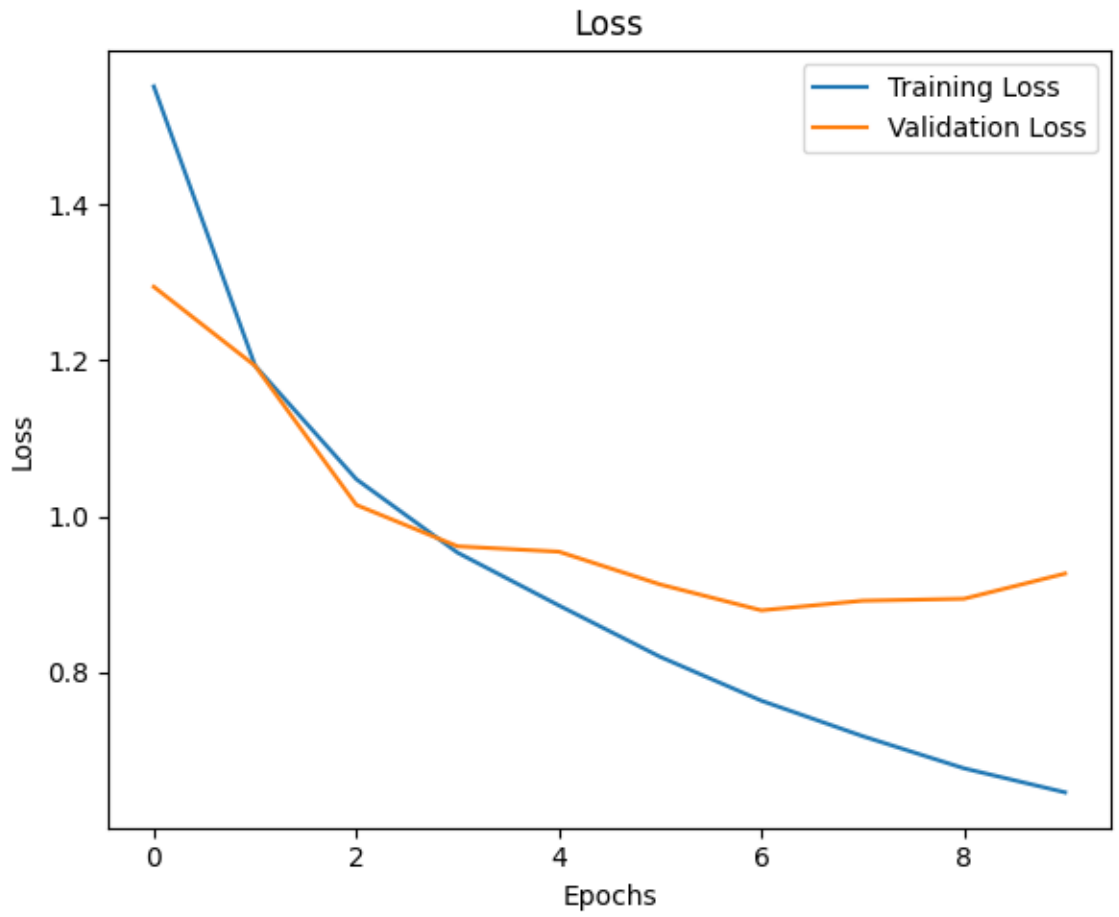
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - accuracy: 0.6900 - loss: 0.9230
Test accuracy: 0.6916
```

8. Tracer l'historique de l'entraînement:

```
# Tracer les courbes de précision et de perte
plt.figure(figsize=(12, 5))
# Précision
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')  # Précision d'entraînement
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')  # Précision de validation
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
# Perte
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')  # Perte d'entraînement
plt.plot(history.history['val_loss'], label='Validation Loss')  # Perte de validation
plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

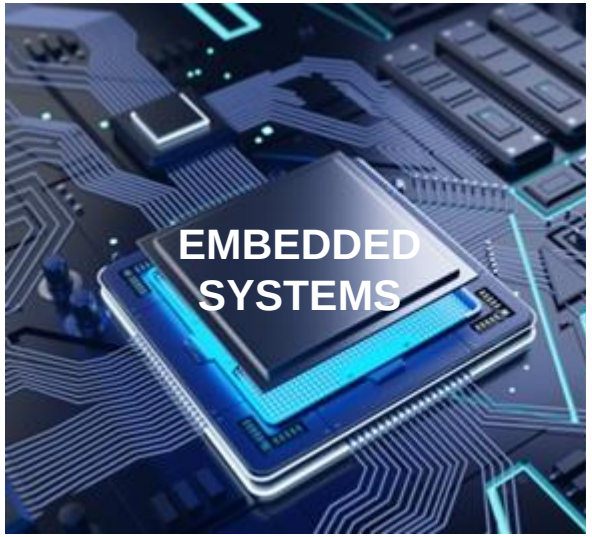

Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0
2
4
6
8


Loss
Training Loss
Validation Loss
Loss
Epochs
1.4
1.2
1.0
0.8
0
2
4
6
8

12.4. Exercice:

Test du modèle CNN entraîné sur CIFAR-10 avec une image personnalisée:

1. Importer une image extérieure (par exemple, une image d'un avion ou d'un chien).
2. Redimensionner cette image à 32x32 pixels et normaliser ses valeurs.
3. Utiliser le modèle pour prédire à quelle classe appartient cette image.
4. Afficher le nom de la classe prédite.



FIN !
Questions ?