The background is a dark blue space scene. It features several bright blue stars with four-pointed flares. A large, dark blue planet with a thin orange-yellow ring of light is positioned on the right side. A smaller, orange-brown planet is visible above it. A bright orange and red comet with a long, thin tail is streaking across the upper right corner. The text is overlaid on the left side of the image.

# IA Géosciences - Deep learning: Entraîner un réseau de neurones multi couches

Romain Wenger (Laboratoire Image Ville  
Environnement)

# Table des matières

**01**

## **Introduction**

Pourquoi plusieurs couches ?

**02**

## **Rétropropagation du gradient**

Application du gradient aux couches cachées

**03**

## **Classification d'images**

Application sur des images

# Table des matières

**01**

## **Introduction**

Pourquoi plusieurs couches ?

**02**

## **Rétropropagation du gradient**

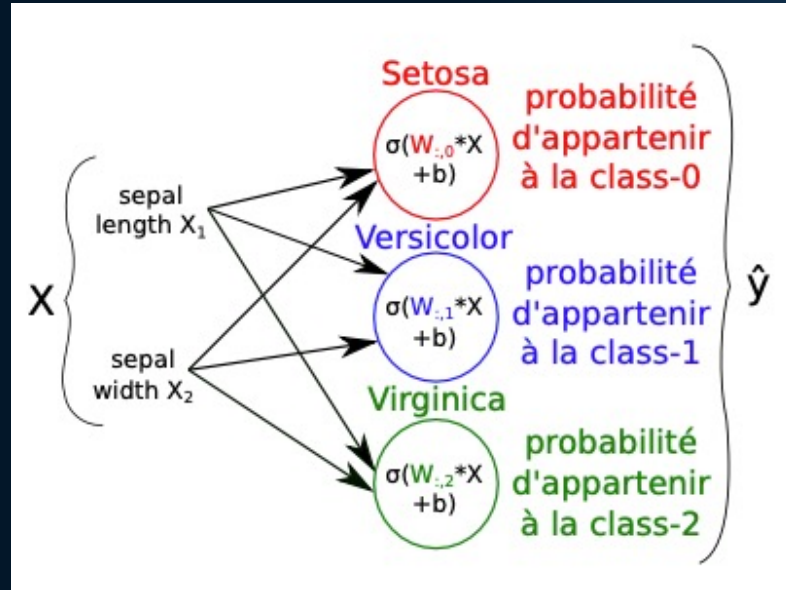
Application du gradient aux couches cachées

**03**

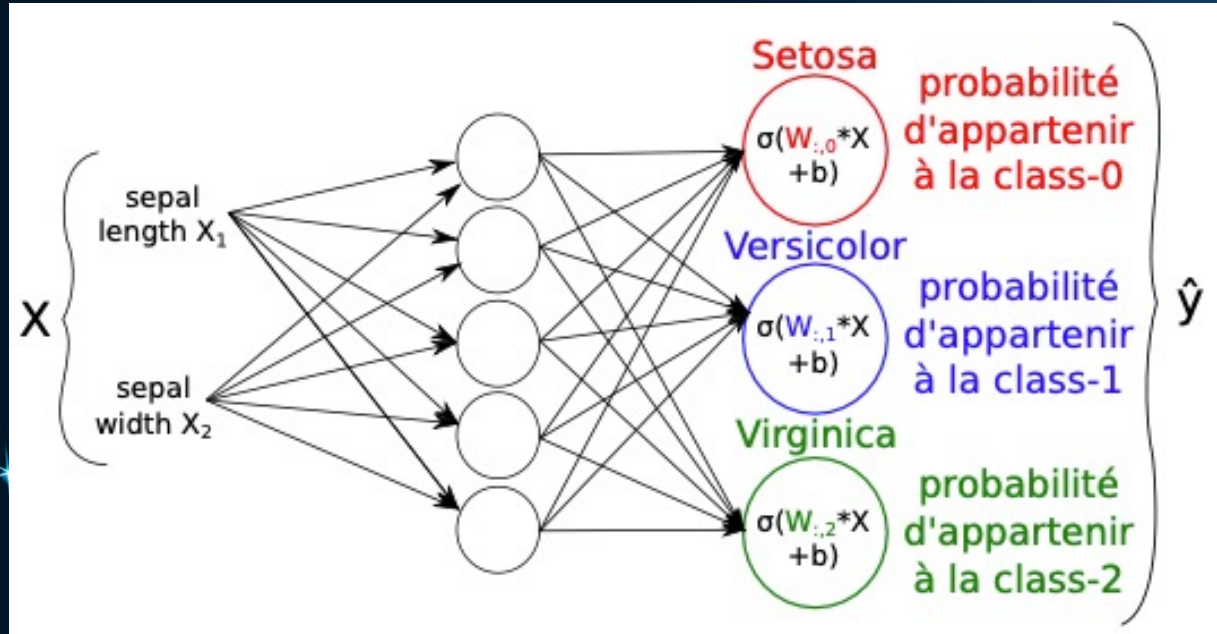
## **Classification d'images**

Application sur des images

# Un réseau à une seule couche (régression logistique)



# Un réseau à plusieurs couches (une seule couche cachée)



# Pourquoi a-t-on besoin d'ajouter des couches cachées ?

**Nécessaire pour apprendre des fonctions non-linéaires**

$$\hat{y} = h(X) \mid h \text{ étant une fonction non-linéaire}$$

**La régression logistique donne une décision linéaire car il n'y a pas d'interaction non-linéaire entre les termes  $X_1$  et  $X_2$  par exemple**

$$\hat{y} = \sigma(W_1 * X_1 + W_2 * X_2 + b)$$

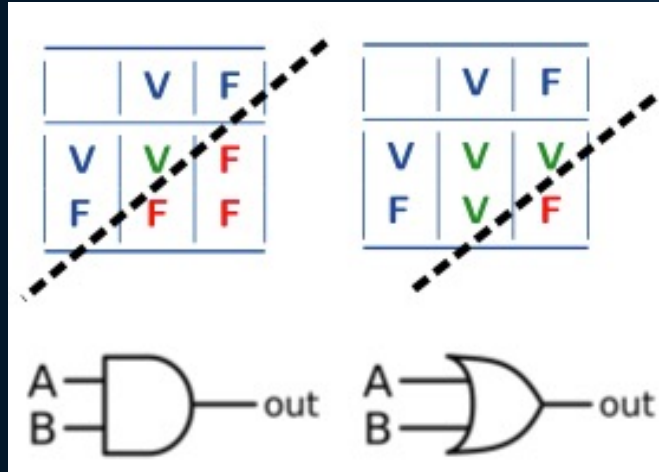
**Pour la rendre non-linéaire il faut ajouter des termes d'interaction**

$$\hat{y} = \sigma(W_1 * X_1 + W_2 * X_2 + W_3 * X_1 * X_2 + W_4 * X_1^2 + W_5 * X_2^2 + \dots + b)$$

Ceci est difficile à faire pour plusieurs raisons :

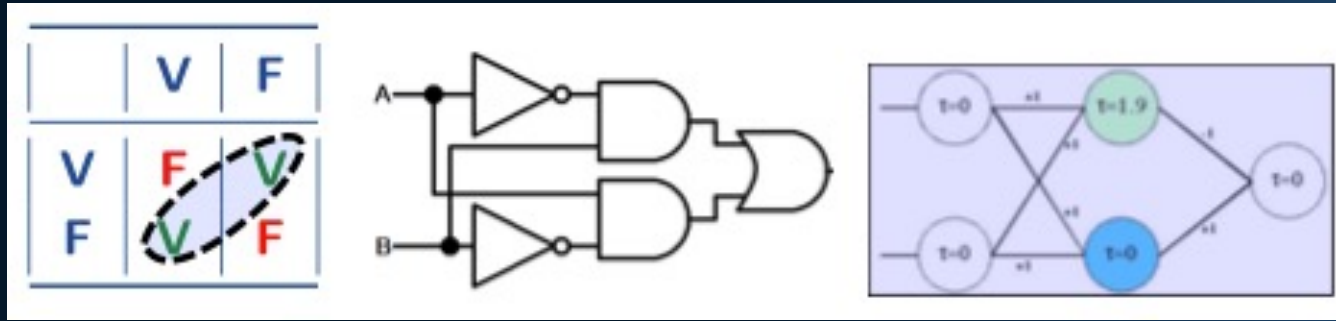
- Grand nombre d'attributs  $X$  : une image a  $128 * 128$  pixels
- On ne connaît pas quelle sorte d'interaction :  $X_1^2 * \sqrt{X_2}$  ou  $X_1^2 * X_2^3$

# Exemple de données séparables linéairement



Une décision linéaire comme celle-ci peut-être apprise par un simple Perceptron (ou un neurone)

# Exemple de données non-séparables linéairement



Par contre une décision non-linéaire comme celle-ci ne peut pas être apprise par un neurone ou un Perceptron à une seule couche.

⇒ il faut pouvoir entraîner un réseau à plusieurs couches

⇒ d'où l'algorithme de Rétropropagation du gradient



# Deep learning

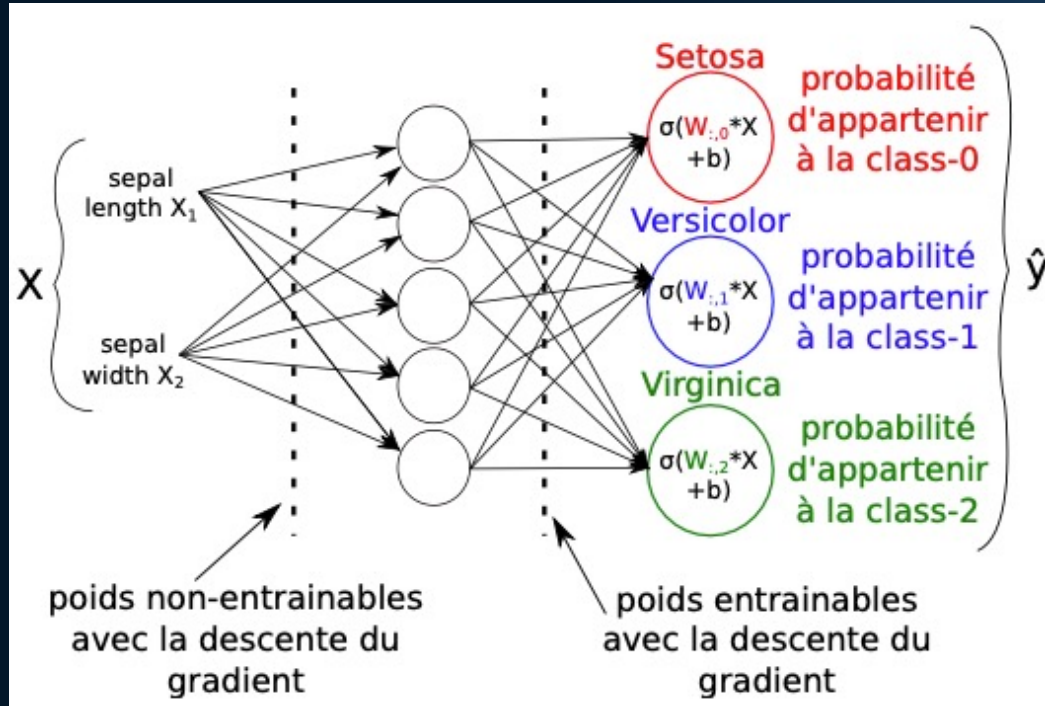
Deep learning : a fancy word for deep neural networks

Deep learning  $\equiv$  Réseau de neurones profonds

Réseau de neurones profonds  $\equiv$  Réseau à plusieurs couches

Le succès du deep learning est dû à l'entraînement des réseaux de neurones à plusieurs couches pour apprendre des décisions non-linéaires

# Pourquoi est-ce difficile d'entraîner des réseaux profonds ?



# Pourquoi est-ce difficile d'entraîner des réseaux profonds ?

On connaît uniquement ce que doit-être la sortie d'un neurone qui appartient à la dernière couche (non-cachée)

⇒ On peut calculer l'erreur pour ces neurones

⇒ On peut appliquer la descente du gradient

On ne connaît par ce que doit-être la sortie d'un neurone appartenant à une couche cachée

⇒ On ne peut pas calculer l'erreur pour ces neurones

⇒ On ne peut appliquer directement la descente du gradient

Il faut trouver un moyen pour estimer l'erreur pour un tel neurone appartenant à une couche cachée : rétropropagation

# Table des matières

01

## Introduction

Pourquoi plusieurs couches ?

02

## Rétropropagation du gradient

Application du gradient aux couches cachées

03

## Classification d'images

Application sur des images

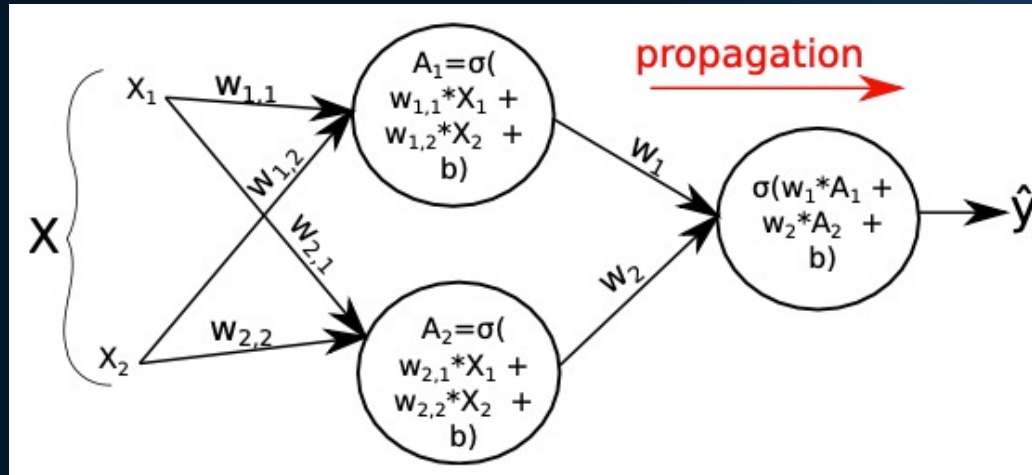
# La fonction de coût ne dépend pas du modèle

Pour un problème de classification binaire : Binary Cross-Entropy

$$L(w) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

Notre but est de minimiser  $L(w)$  en variant l'ensemble des  $w$

# Propagation (Forward pass)



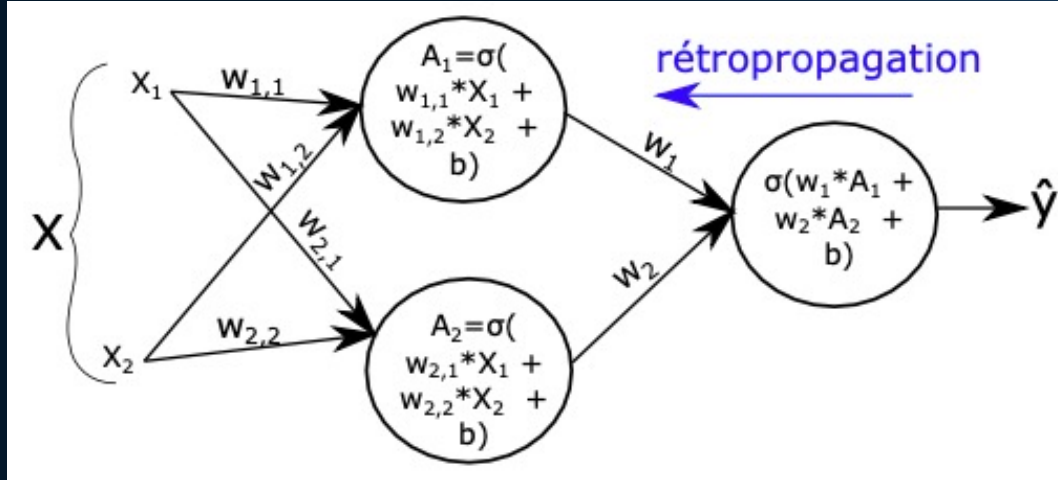
La sortie du réseau est calculée telle que :

$$\hat{y} = \sigma(w_1 * A_1 + w_2 * A_2)$$

$A_i$  les activations des neurones  $i$  appartenant à la couche cachée :

$$A_i = \sigma(w_{i,1} * X_1 + w_{i,2} * X_2)$$

# Rétropropagation (Backward pass)



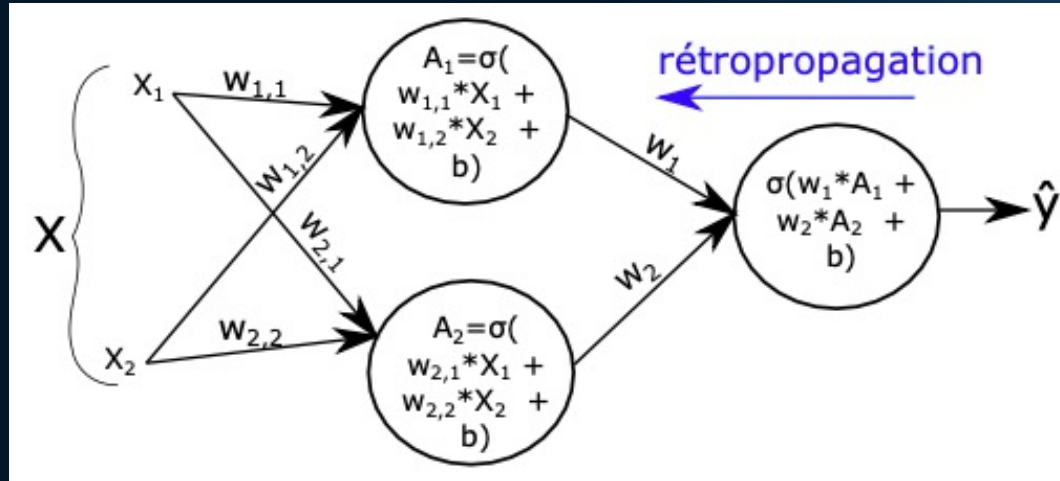
Pour  $w_1$  et  $w_2$  on peut appliquer facilement la descente du gradient

$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

On calcule la dérivée par rapport à  $w_i$

$$\frac{\partial L}{\partial w_i} = (\hat{y} - y) * A_i$$

# Rétropropagation (Backward pass)



Pour  $w_{i,1}$  (ou  $w_{i,2}$ ) on ne peut pas calculer directement la dérivée

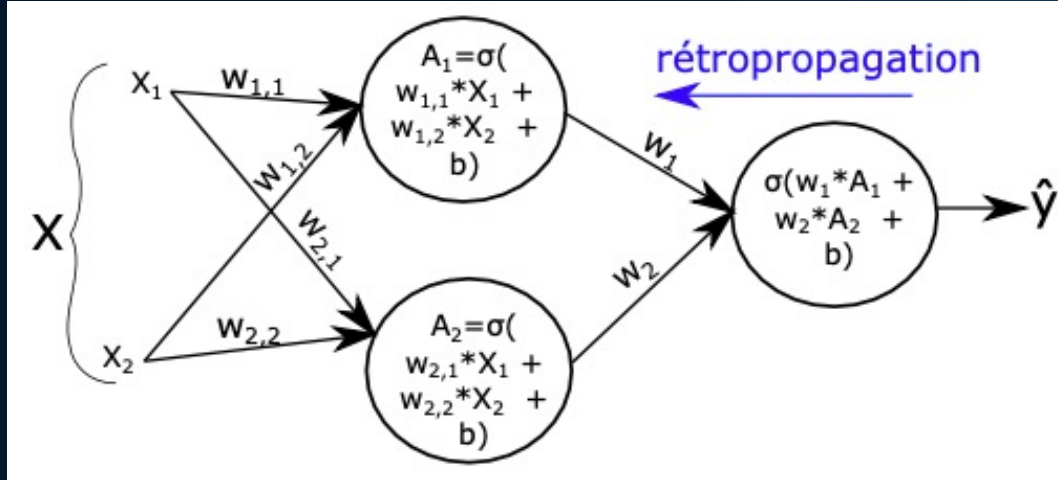
$$\frac{\partial L}{\partial w_{i,1}}$$

Or d'après le théorème de dérivation des fonctions composées

$$\frac{\partial L}{\partial w_{i,1}} = \frac{\partial L}{\partial A_i} * \frac{\partial A_i}{\partial w_{i,1}}$$



# Rétropropagation (Backward pass)



$$\frac{\partial L}{\partial w_{i,1}} = \frac{\partial L}{\partial A_i} * \frac{\partial A_i}{\partial w_{i,1}}$$

Cette équation permet de calculer les dérivées pour n'importe quel nombre de couches  $\Rightarrow$  permet d'entraîner un réseau multi-couches

# Dérivée pour un poids $w$ d'un neurone caché

Il faut calculer la dérivée par rapport à  $w_{i,1}$  par exemple :

$$\frac{\partial L}{\partial w_{i,1}} = \frac{\partial L}{\partial A_i} * \frac{\partial A_i}{\partial w_{i,1}}$$

On commence par le premier terme

$$\frac{\partial L}{\partial A_i} = (\hat{y} - y) * w_i$$

Pour le second terme

$$\frac{\partial A_i}{\partial w_{i,1}} = X_1 * A_1(1 - A_1)$$

$$\frac{\partial L}{\partial w_{i,1}} = w_1 * X_1 * A_1(1 - A_1)(\hat{y} - y)$$

# Dérivée pour un poids $w$ d'un neurone caché

La descente du gradient peut-être maintenant appliquée

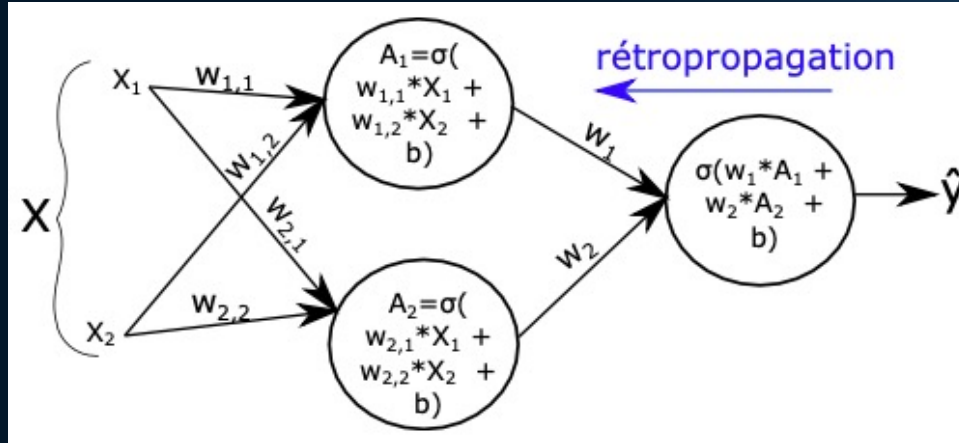
$$w_{i,1} = w_{i,1} - \alpha * w_i * X_1 * A_i(1 - A_i)(\hat{y} - y)$$

On peut voir comment le terme de l'erreur  $(\hat{y} - y)$  (calculée pour la couche de sortie) a été **propagé** pour effectuer la descente du gradient sur un neurone appartenant à une couche cachée

**Propagée**  $\Rightarrow$  **Rétropropagation** du gradient (ou de l'erreur)

**En répétant ce processus pour chaque couche cachée on entraîne un réseau profond** : l'erreur se propage d'une couche à une autre

# Calculer le nombre de paramètre à apprendre



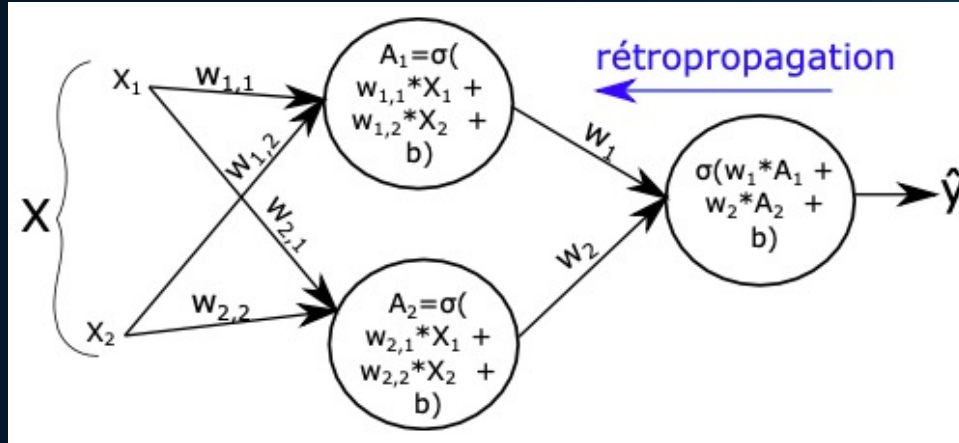
**Pour la première couche en ignorant les  $b$  :**

On a deux neurones et deux entrées  $\Rightarrow 2 * 2 = 4$  paramètres

**Pour la deuxième couche en ignorant les  $b$  :**

On a un neurone et deux entrées ( $A_1$  et  $A_2$ )  $\Rightarrow 2 * 1 = 2$  paramètres

# Calculer le nombre de paramètre à apprendre



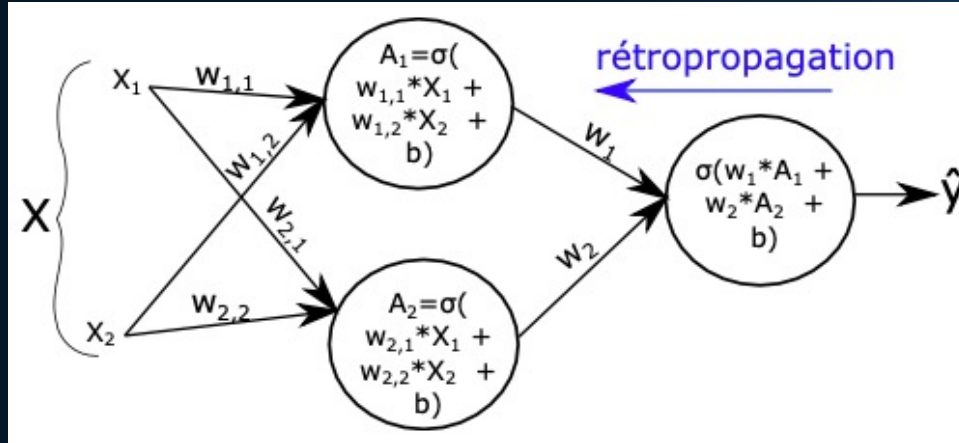
**Pour la deuxième couche en ignorant les  $b$  :**

On a un neurone et deux entrées ( $A_1$  et  $A_2$ )  $\Rightarrow 2 * 1 = 2$  paramètres

**En ajoutant le  $b$  :**

Chaque neurone a un  $b \Rightarrow 2 + 1 = 3$  paramètres

# Calculer le nombre de paramètre à apprendre

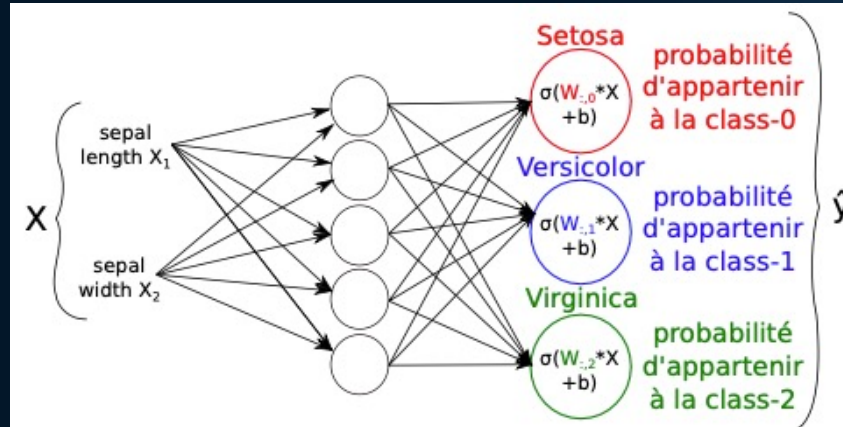


**En ajoutant le  $b$  :**

Chaque neurone a un  $b \Rightarrow 2 + 1 = 3$  paramètres

**Au total :** 6 paramètres (couche 1) + 3 paramètres (couche 2) = 9

# Calculer le nombre de paramètre à apprendre



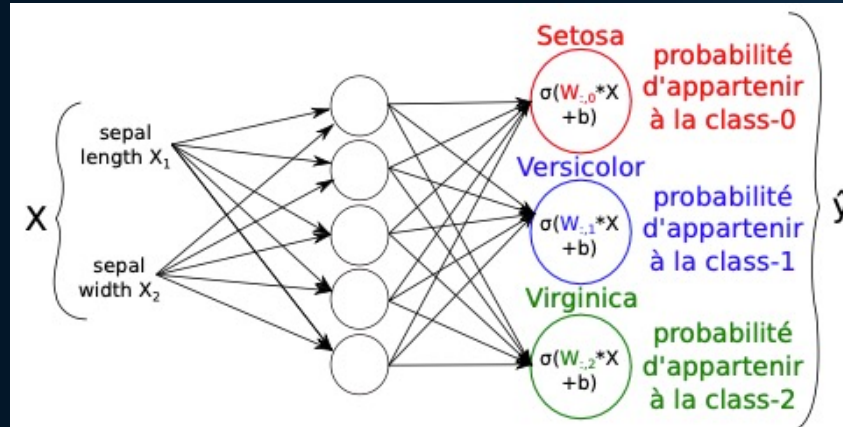
**Pour la première couche en ignorant les  $b$  :**

On a 5 neurones et 2 entrées  $\Rightarrow 2 * 5 = 10$  paramètres

**En ajoutant le  $b$  :**

Chaque neurone a un  $b \Rightarrow 10 + 1 + 1 + 1 + 1 + 1 = 15$  paramètres

# Calculer le nombre de paramètre à apprendre



**Pour la deuxième couche en ignorant les  $b$  :**

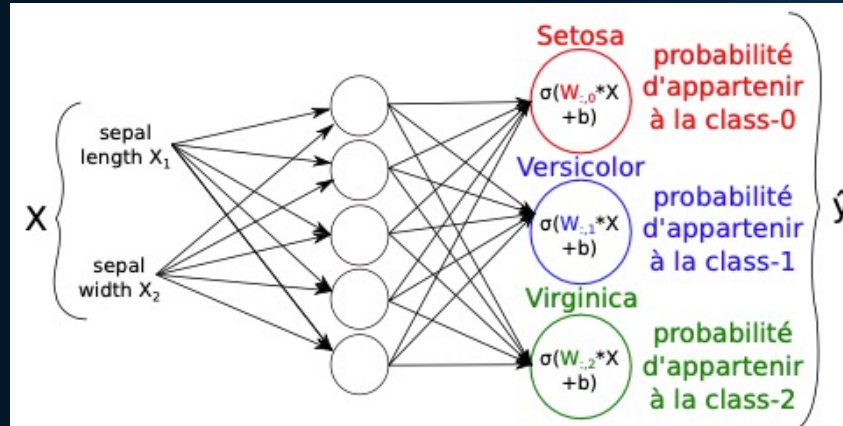
On a 3 neurones et 5 entrées  $\Rightarrow 5 * 3 = 15$  paramètres

**En ajoutant le  $b$  :**

Chaque neurone a un  $b \Rightarrow 15 + 1 + 1 + 1 = 18$  paramètres



# Calculer le nombre de paramètre à apprendre

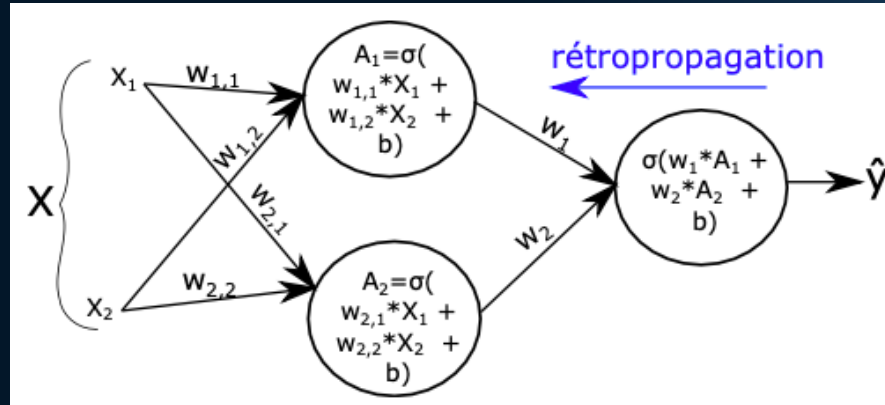


**En ajoutant le  $b$  :**

Chaque neurone a un  $b \Rightarrow 15 + 1 + 1 + 1 = 18$  paramètres

**Au total :** 15 paramètres (couche 1) + 18 paramètres (couche 2) = 33

# Fonction d'activation



Un neurone caché reçoit de la couche précédente :  $z = W * X + b$

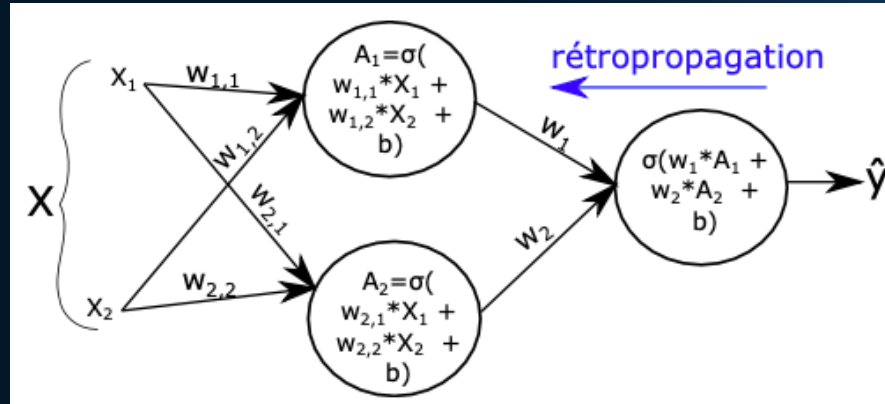
Ce neurone effectue une interaction non-linéaire entre ses entrées

Pour cela il faut que la sortie (ou activation) du neurone :  $A = f(z)$

Avec  $f(.)$  étant une fonction non-linéaire par exemple :  $\sigma(.)$ ,  $\tanh(.)$

Si  $f(.)$  linéaire, le réseau ne va apprendre que des décisions linéaires

# Représentation matricielle



Pour la première couche :

$$W^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

Pour la deuxième couche :

$$W^{(2)} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$$

# Hyperparamètres et paramètres

## **Paramètres** d'un réseau de neurones

- Tout ce que le réseau apprend lui-même avec la descente du gradient
- Par exemple : les poids  $w$  et les biais  $b$

## **Hyperparamètres** d'un réseau de neurone

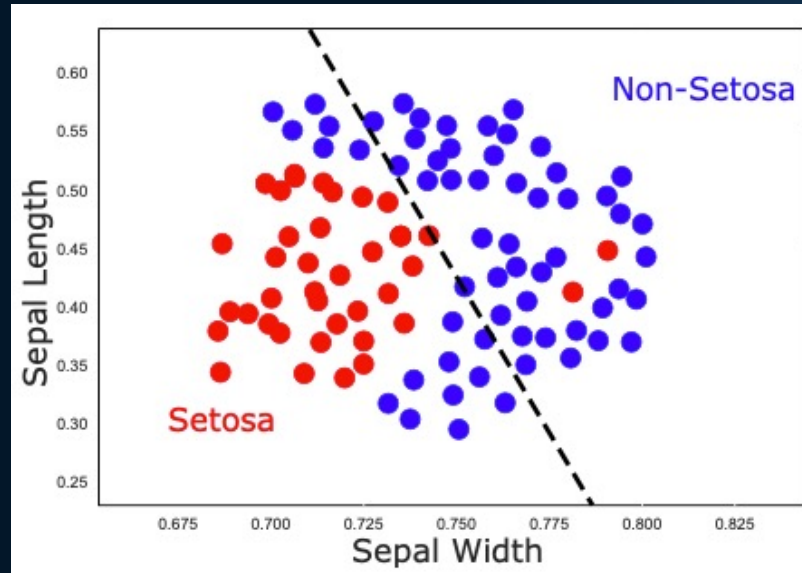
- Toute sorte de choix effectué avant l'entraînement
- Par exemple : nombre de neurones, le taux d'apprentissage  $\alpha$ , etc.

# Train/Validation/Test

Supposons qu'on a un jeu de données contenant 100 images

- Ensemble d'entraînement : **train set**
  - Il est utilisé pour apprendre les *paramètres* du réseau
- Ensemble de validation : **validation set**
  - Il est utilisé pour guider le choix des *hyperparamètres* du réseau
- Ensemble de test : **test set**
  - Il est utilisé uniquement pour évaluer un modèle
  - L'optimisation des paramètres ne doit pas se baser sur la performance du réseau sur l'ensemble de test
  - Le choix des hyperparamètres ne doit pas se baser sur la performance du réseau sur l'ensemble de test

# Sous-apprentissage (underfitting)



# Sous-apprentissage (underfitting)

**L'erreur est**

- Elevée sur le train/validation

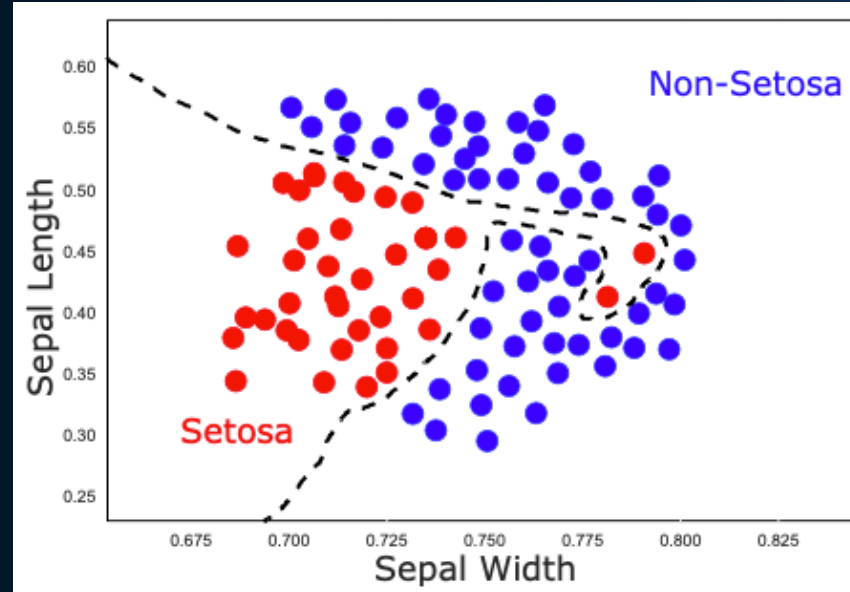
**On dit que le modèle (réseau) a**

- Un « biais » élevé (on est loin du but/la bonne décision)

**⇒ Le modèle n'arrive pas à apprendre une décision complexe**

- Il faut augmenter sa capacité de modélisation (par exemple augmenter le nombre de neurones, ajouter des couches cachées, etc.)

# Sur-apprentissage (overfitting)





# Sur-apprentissage (overfitting)

**L'erreur est**

- Très faible sur le train
- Très élevé sur la validation

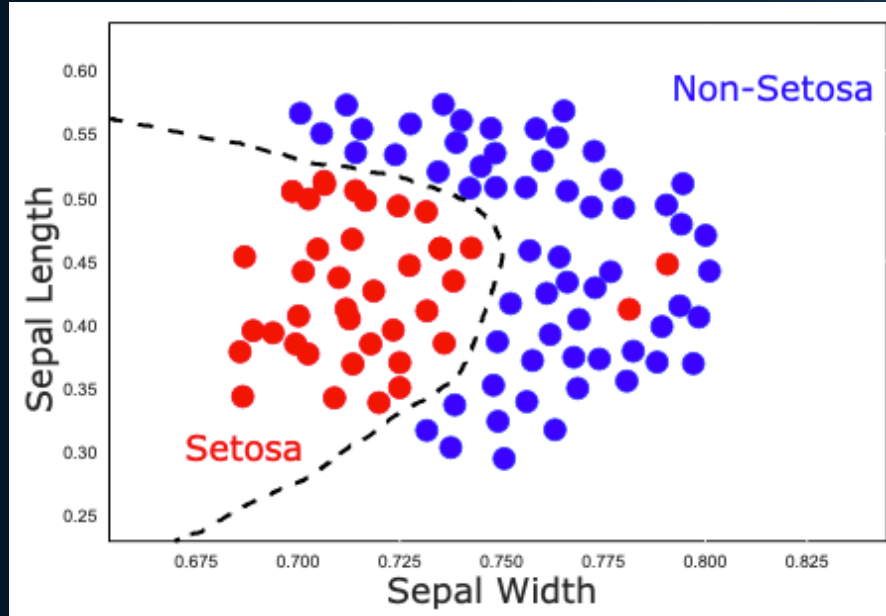
**On dit que le modèle (réseau) a**

- Un « biais » faible (on est proche du but – la bonne décision)
- Une « variance » élevée (faible performance sur la validation par rapport au train)

**⇒ Le modèle apprend une décision qui est trop complexe**

- Il faut diminuer sa capacité de modélisation (par exemple diminuer le nombre de neurones, enlever des couches cachées, etc.)

# Bonne généralisation : ni overfitting ni underfitting



# Bonne généralisation : ni overfitting ni underfitting

L'erreur est

- Très **faible** sur le train
- Très **faible** sur la validation

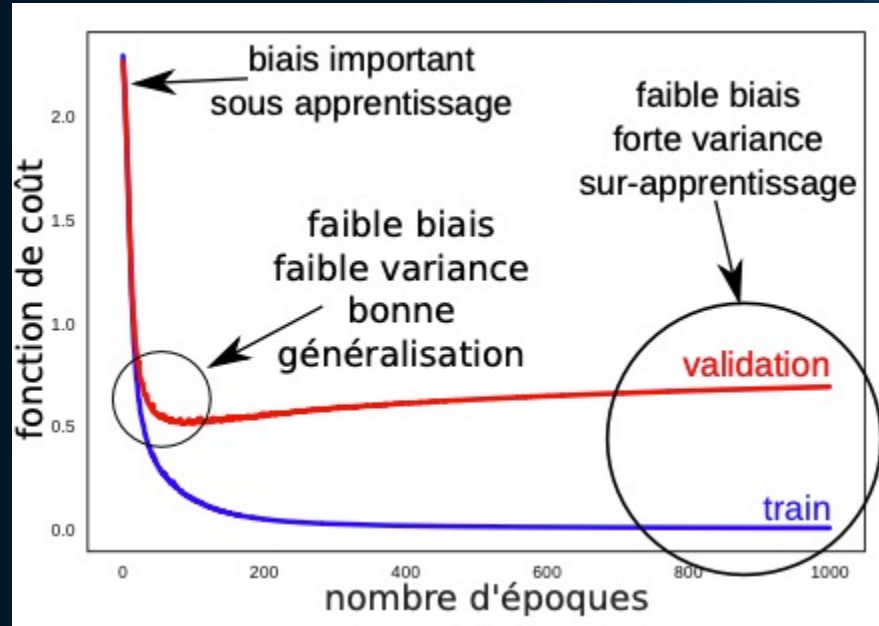
On dit que le modèle (réseau) a

- Le « biais » est faible (on est proche du but – la bonne décision)
- La « variance » est faible (bonne performance sur la validation par rapport au train)

⇒ **Le modèle apprend une décision qui est ni trop complexe ni trop simple**

- Le modèle arrive à ignorer les anomalies lors de l'apprentissage

# Bias-variance tradeoff



Choisir le modèle qui minimise le coût pour le validation set

# Table des matières

01

## Introduction

Pourquoi plusieurs couches ?

02

## Rétropropagation du gradient

Application du gradient aux couches cachées

03

## Classification d'images

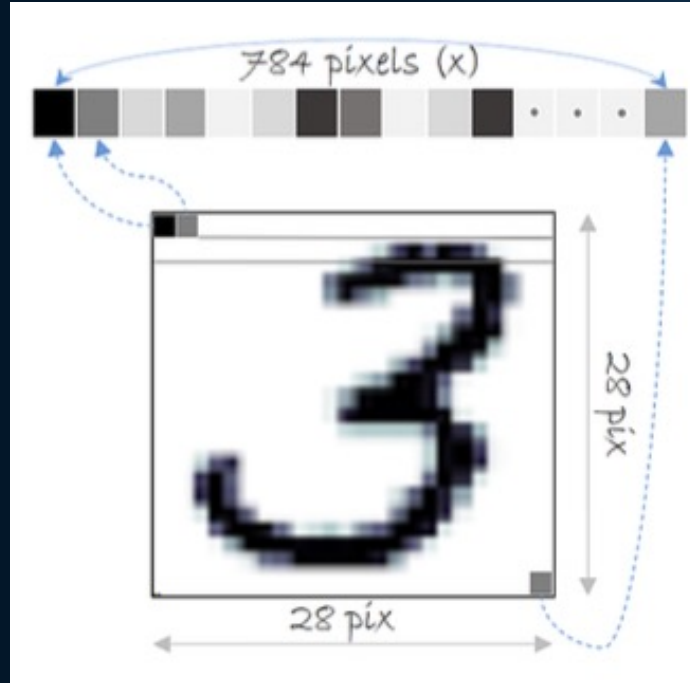
Application sur des images

# Jeu de données : MNIST

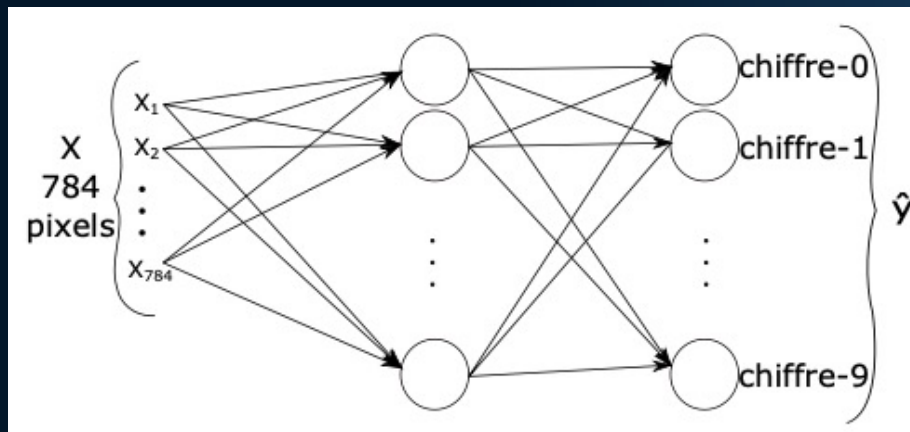
**MNIST** : Modified National institute of Standards and Technology



# Transformation de l'image en vecteur (aplatir)



# Un réseau de neurones pour MNIST

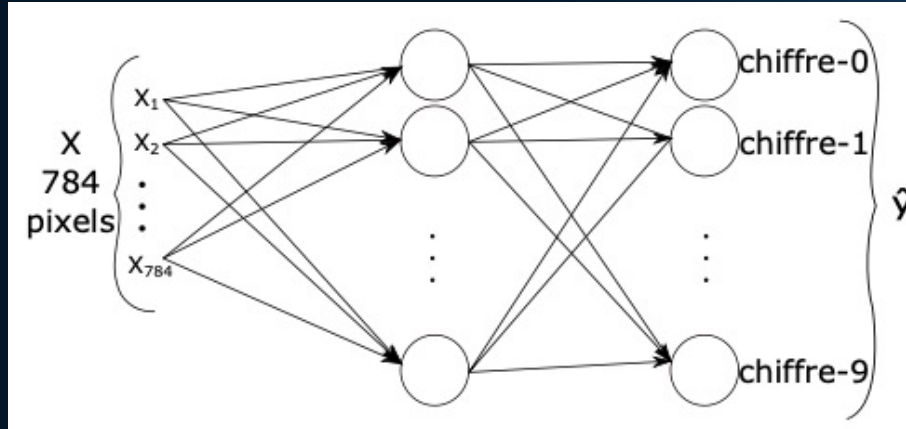


Perceptron multi-couche/multi-layer perceptron (MLP) pour classifier en 10 chiffres des images de  $28 * 28 = 784$  pixels (niveau de gris)

**C'est un réseau à 2 couches : la sortie qui contient les 10 classes et la couche cachée, qui contient un certain nombre de neurones. Généralement la couche d'entrée (les pixels) n'est pas comptée.**

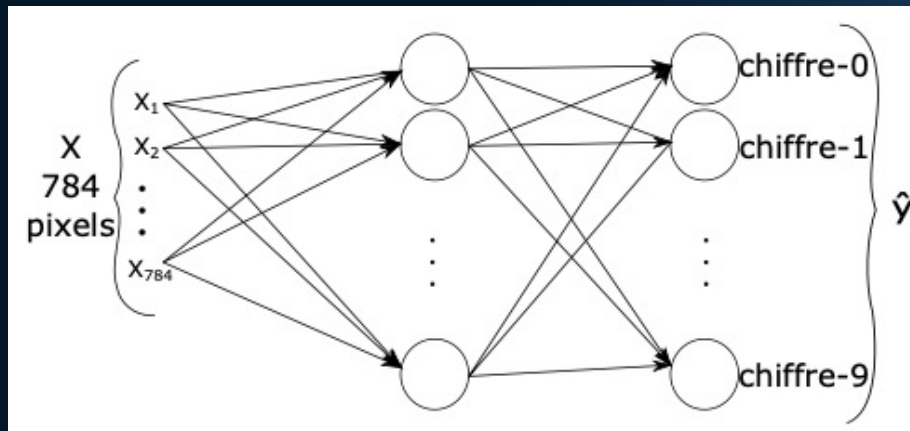


# Un réseau de neurones pour MNIST



- On a 784 attributs en entrée
- On a 10 sorties
- Pour  $n$  neurones cachés on a :
  - $784 * n + n$  paramètres pour la première couche (cachée)
  - $n * 10 + 10$  paramètres pour la deuxième couche (sortie)

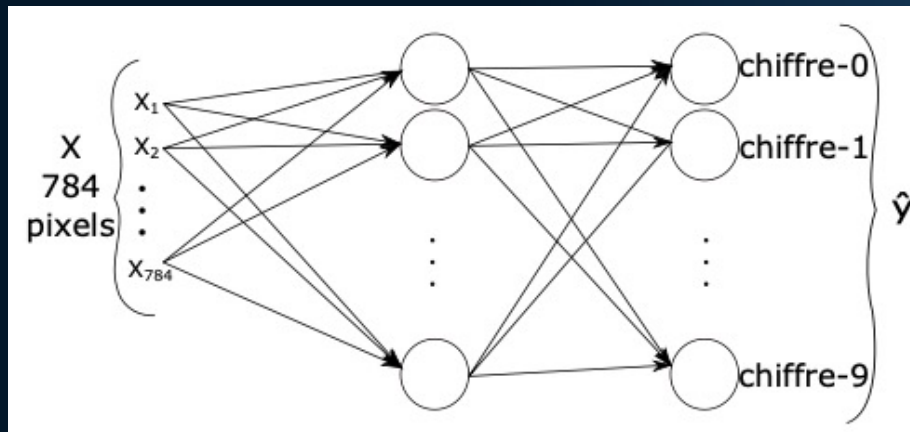
# Un réseau de neurones pour MNIST



Pour la première couche avec  $n$  neurones cachés

$$W^{(1)} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,784} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,784} \end{bmatrix}$$

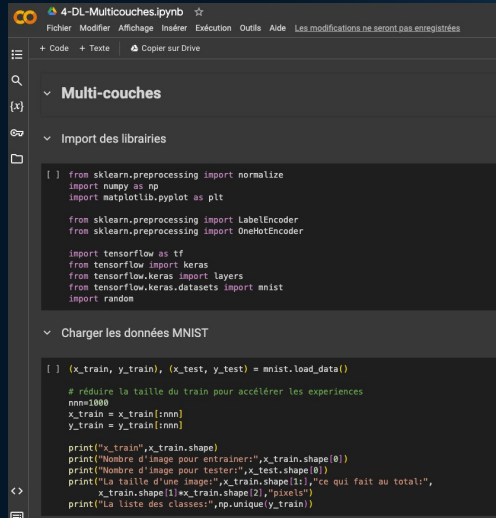
# Un réseau de neurones pour MNIST



Pour la deuxième couche avec  $n$  neurones cachés

$$W^{(2)} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{10,1} & \cdots & w_{10,n} \end{bmatrix}$$

# De la théorie vers la pratique



```
4-DL-Multicouches.ipynb
Fichier Modifier Affichage Insérer Exécution Outils Aide Les modifications ne seront pas enregistrées

+ Code + Texte Copier sur Drive

Multi-couches

Import des librairies

[ ] from sklearn.preprocessing import normalize
import numpy as np
import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist
import random

Charger les données MNIST

[ ] (x_train, y_train), (x_test, y_test) = mnist.load_data()

# réduire la taille du train pour accélérer les expériences
nnon=1000
x_train = x_train[nnon]
y_train = y_train[nnon]

print("x_train", x_train.shape)
print("Nombre d'image pour entrainer:", x_train.shape[0])
print("Nombre d'image pour tester:", x_test.shape[0])
print("La taille d'une image", x_train.shape[1], "ce qui fait au total:",
      x_train.shape[1]*x_train.shape[2], "pixels")
print("La liste des classes:", np.unique(y_train))
```

Télécharger le fichier ([https://github.com/r-wenger/cours\\_ml-m2-OTG/blob/main/IA\\_geosciences\\_M2/CM/4\\_DL\\_Multicouches.ipynb](https://github.com/r-wenger/cours_ml-m2-OTG/blob/main/IA_geosciences_M2/CM/4_DL_Multicouches.ipynb))  
et l'importer sur Google Colab