Introduction au deep learning

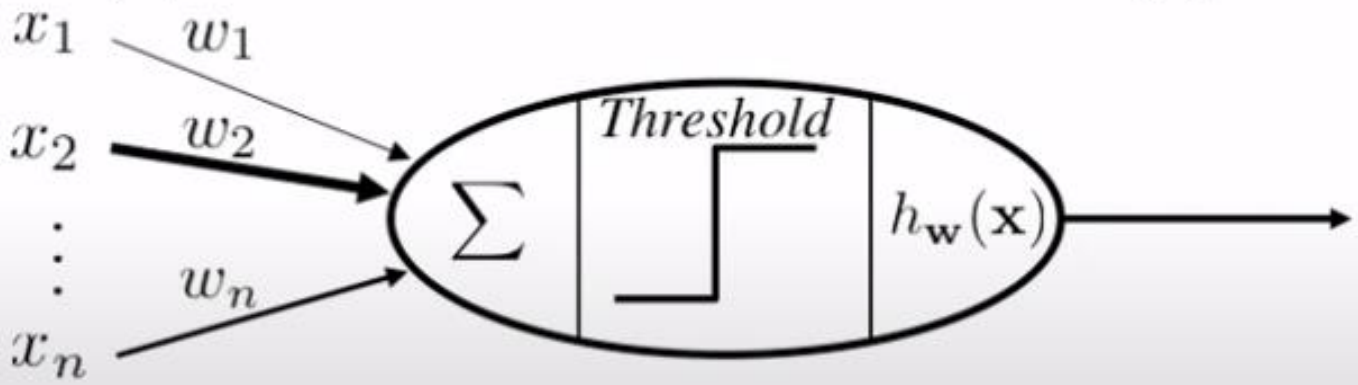
Sommaire

1. Perceptron

- **Idée**: modéliser la décision à l'aide d'une fonction linéaire, suivi d'un seuil:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w} \cdot \mathbf{x})$$

où $Threshold(z) = 1$ si $z \geq 0$, sinon $Threshold(z) = 0$



- Le vecteur de **poids** $\mathbf{w}$ correspond aux paramètres du modèle
- On ajoute également un **biais** *b* qui équivaut à rajouter une entrée $x_{n+1} = 1$

Algorithme du perceptron

- L'algorithme d'apprentissage doit adapter la valeur des paramètres de façon à ce que $h_{\mathbf{w}}(\mathbf{x})$ soit la bonne réponse sur les données d'entraînement

Algorithme du perceptron

1. pour chaque paire $(\mathbf{x}_t, y_t) \in D$
   a. calculer $h_{\mathbf{w}}(\mathbf{x}_t) = Threshold(\mathbf{w} \cdot \mathbf{x}_t)$
   b. si $y_t \neq h_{\mathbf{w}}(\mathbf{x}_t)$
      - $w_i \leftarrow w_i + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i} \quad \forall i$ (mise à jour des poids et biais)
2. retourner à 1 jusqu'à l'atteinte d'un critère d'arrêt (nb. maximal d'itérations atteint ou nb. d'erreurs est 0)

- La mise à jour des poids est appelée la **règle d'apprentissage du perceptron**. Le multiplicateur α est appelé le **taux d'apprentissage**

Algorithme du perceptron

- L'algorithme d'apprentissage doit adapter la valeur des paramètres de façon à ce que $h_{\mathbf{w}}(\mathbf{x})$ soit la bonne réponse sur les données d'entraînement

Algorithme du perceptron

1. pour chaque paire $(\mathbf{x}_t, y_t) \in D$
   a. calculer $h_{\mathbf{w}}(\mathbf{x}_t) = Threshold(\mathbf{w} \cdot \mathbf{x}_t)$
   b. si $y_t \neq h_{\mathbf{w}}(\mathbf{x}_t)$
      - $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{x}_t$
2. retourner à 1 jusqu'à l'atteinte d'un critère d'arrêt (nb. maximal d'itérations atteint ou nb. d'erreurs est 0)

Forme vectorielle

- La mise à jour des poids est appelée la **règle d'apprentissage du perceptron**. Le multiplicateur α est appelé le **taux d'apprentissage**

Apprentissage vu comme la minimisation d'une perte

- Le problème de l'apprentissage peut être formulé comme un problème d'optimisation
  - pour chaque exemple d'entraînement, on souhaite minimiser une certaine distance $Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))$ entre la cible y_t et la prédiction $h_{\mathbf{w}}(\mathbf{x}_t)$
  - on appelle cette distance une **perte**

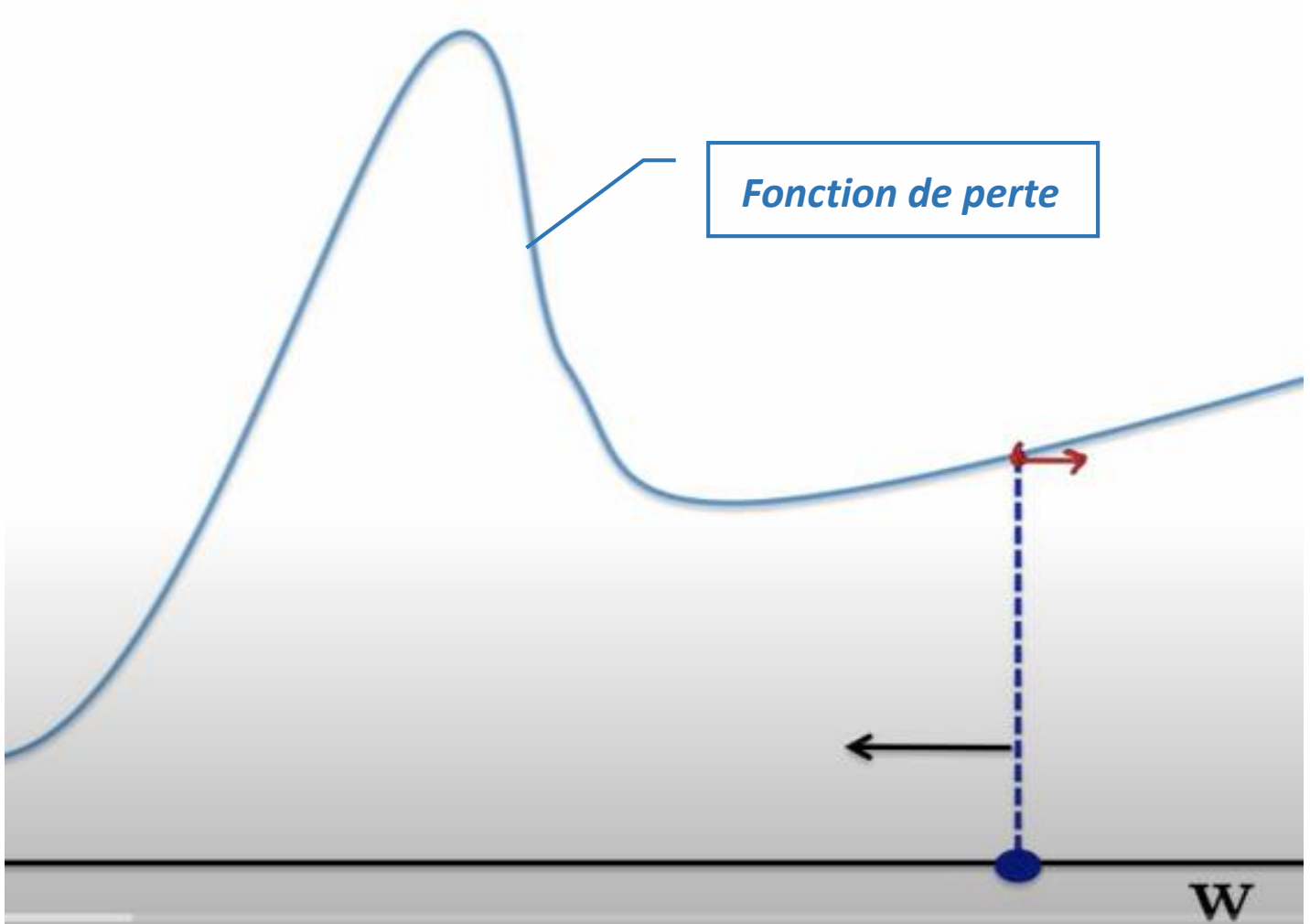
- Dans le cas du perceptron:

$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{w} \cdot \mathbf{x}_t$$

  - si la prédiction est bonne, le coût est 0
  - si la prédiction est mauvaise, le perte est la distance entre $\mathbf{w} \cdot \mathbf{x}_t$ et le seuil à franchir pour que la prédiction soit bonne

Algorithme de descente de gradient

- Etant donnée une fonction de perte, la direction opposée au gradient nous donne la direction à suivre :



Algorithme de descente de gradient

- En apprentissage automatique, on souhaite optimiser:

$$\frac{1}{|D|} \sum_{(\mathbf{x}_t, y_t) \in D} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t))$$

- Le gradient par rapport à la perte moyenne contient les dérivées partielles:

$$\frac{1}{|D|} \sum_{(\mathbf{x}_t, y_t) \in D} \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

- Devrait calculer la moyenne des dérivées sur tous les exemples d'entraînement avant de faire une mise à jour des paramètres!

Descente de gradient stochastique

- **Descente de gradient stochastique**: mettre à jour les paramètres à partir du gradient (c.-à-d. des dérivées partielles) d'un seul exemple, choisi aléatoirement:

- Initialiser $\mathbf{w}$ aléatoirement
- Pour T itérations
  - Pour chaque exemple d'entraînement $(\mathbf{x}_t, y_t)$
    - $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$

- Cette procédure est plus efficace lorsque l'ensemble d'entraînement est grand
  - on fait $|D|$ mises à jour des paramètres après chaque parcours de l'ensemble d'entraînement, plutôt qu'une seule mise à jour avec la descente de gradient normale

Retour sur le perceptron

- On utilise le gradient (dérivée partielle) pour déterminer une direction de mise à jour des paramètres:

$$\frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = \frac{\partial}{\partial w_i} -(y_t - h_{\mathbf{w}}(\mathbf{x}_t))\mathbf{w} \cdot \mathbf{x}_t \cong -(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i}$$

- Par définition, le gradient donne la direction (locale) d'augmentation la plus grande de la perte
  - pour mettre à jour les paramètres, on va dans la direction opposée à ce gradient:

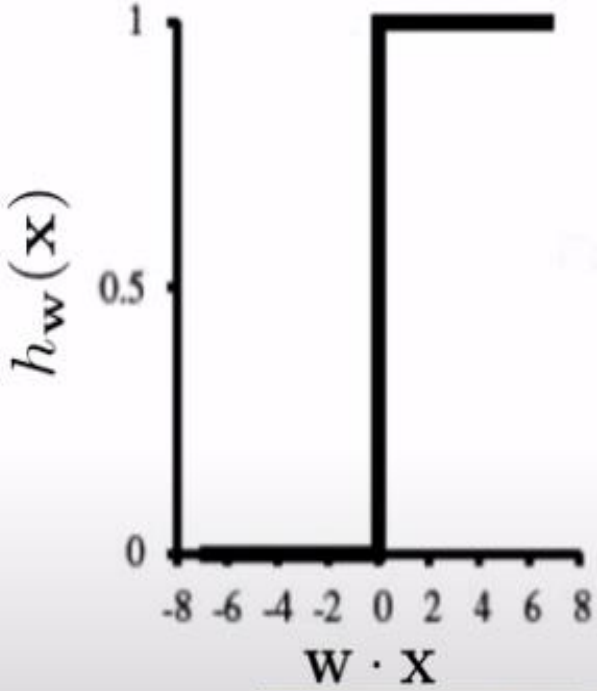
$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

  - on obtient la règle d'apprentissage du perceptron

$$w_i \leftarrow w_i + \alpha(y_t - h_{\mathbf{w}}(\mathbf{x}_t))x_{t,i} \quad \forall i$$

Retour sur le perceptron

- La procédure de descente de gradient stochastique est applicable à n'importe quelle perte dérivable partout
- Dans le cas du perceptron, on a un peu triché:
  - la dérivée de $h_{\mathbf{w}}(\mathbf{x})$ n'est pas définie lorsque $\mathbf{w} \cdot \mathbf{x} = 0$
- L'utilisation de la fonction *Threshold* (qui est constante par partie) fait que la courbe d'entraînement peut être instable

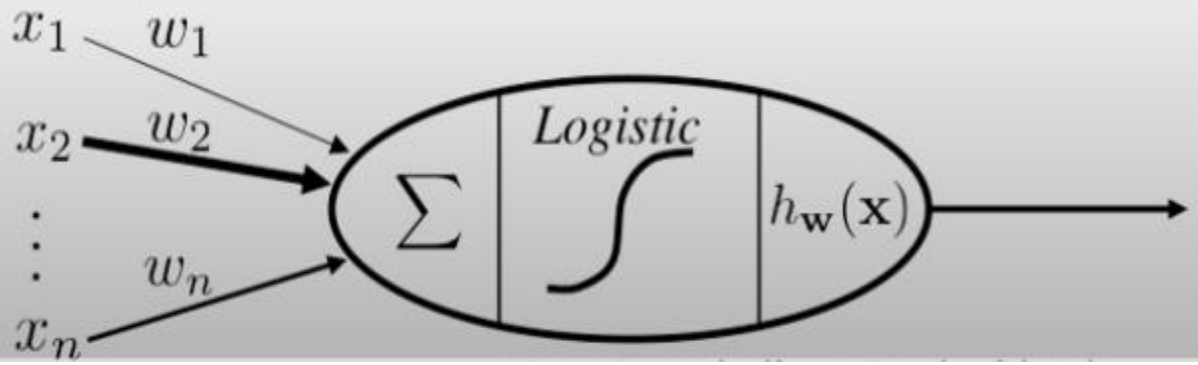


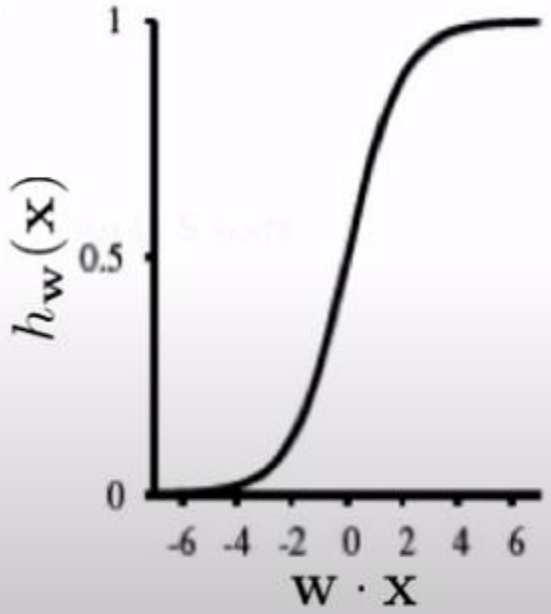
On remplace la threshold par une sigmoïde

- **Idée**: plutôt que de prédire une classe, prédire une probabilité d'appartenir à la classe 1

$$p(y = 1|\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- Choisir la classe la plus probable selon le modèle
  - si $h_{\mathbf{w}}(\mathbf{x}) >= 0.5$ choisir la classe 1
  - sinon, choisir la classe 0





Règle d'apprentissage

Cross-entropy

- Pour obtenir une règle d'apprentissage, on définit d'abord une perte

$$Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) = -y_t \log h_{\mathbf{w}}(\mathbf{x}_t) - (1 - y_t) \log(1 - h_{\mathbf{w}}(\mathbf{x}_t))$$

  - si $y_t = 1$, on souhaite maximiser la probabilité $p(y_t = 1|\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}_t)$
  - si $y_t = 0$, on souhaite maximiser la probabilité $p(y_t = 0|\mathbf{x}) = 1 - h_{\mathbf{w}}(\mathbf{x}_t)$

- On dérive la règle d'apprentissage comme une descente de gradient

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(y_t, h_{\mathbf{w}}(\mathbf{x}_t)) \quad \forall i$$

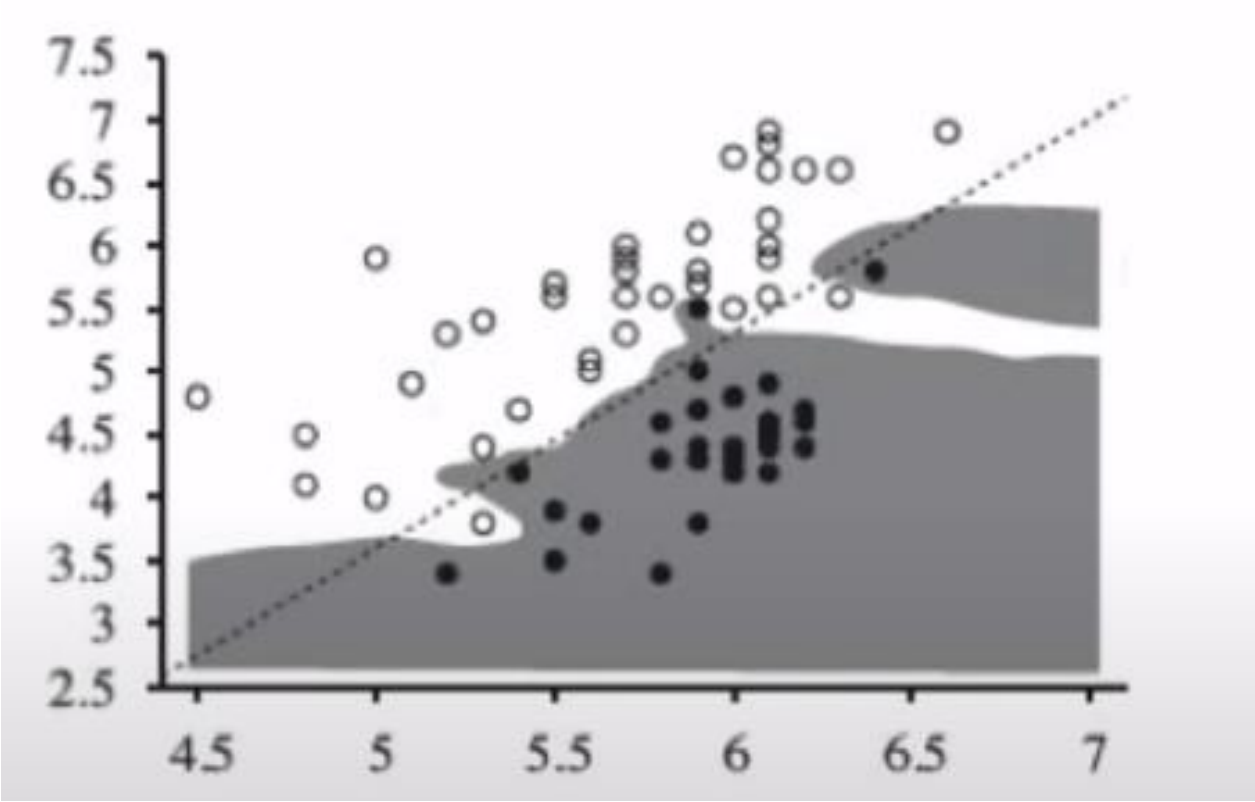
ce qui donne

$$w_i \leftarrow w_i + \alpha (y_t - h_{\mathbf{w}}(\mathbf{x}_t)) x_{t,i} \quad \forall i$$

- La règle est donc la même que pour le Perceptron, mais la définition de $h_{\mathbf{w}}(\mathbf{x}_t)$ est différente

2. Artificial Neural Network (ANN)

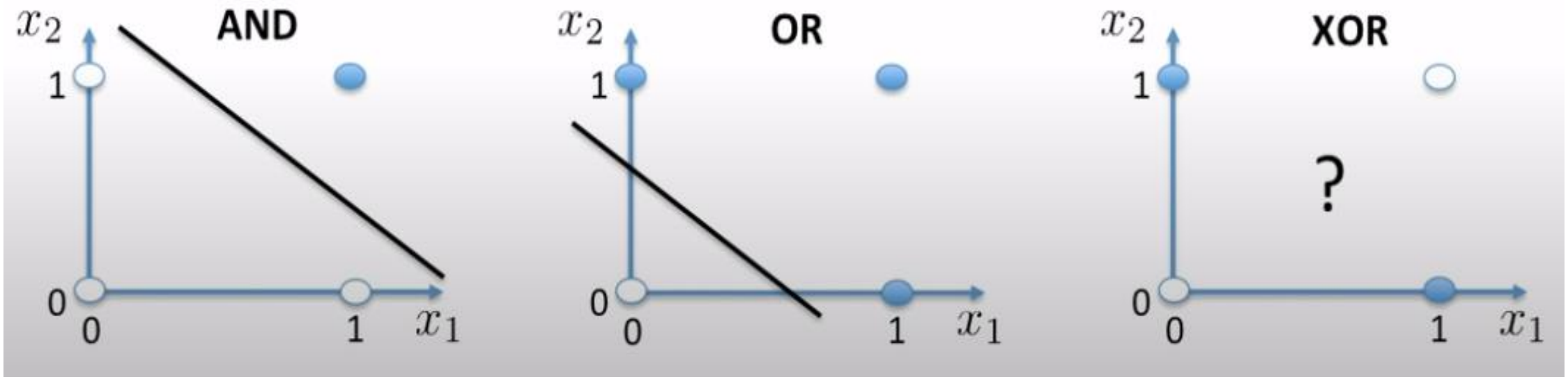
- Si les données d'entraînement sont séparables linéairement, le perceptron et la régression logistique vont trouver cette séparation



Limites des classifieurs linéaires

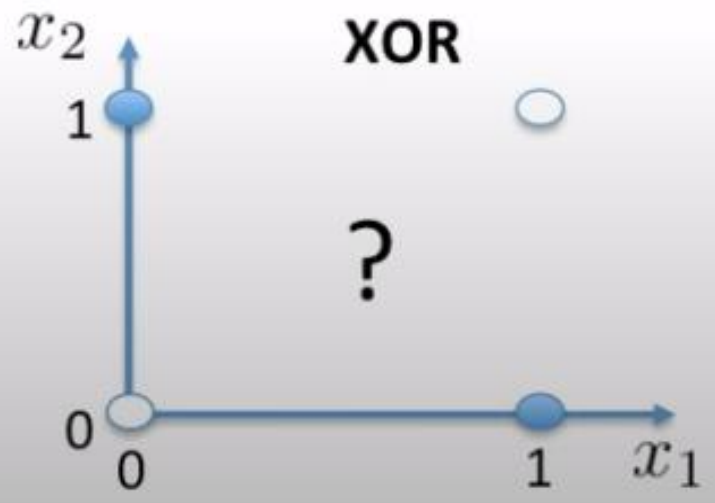
- Cependant, la majorité des problèmes de classification ne sont pas linéaires

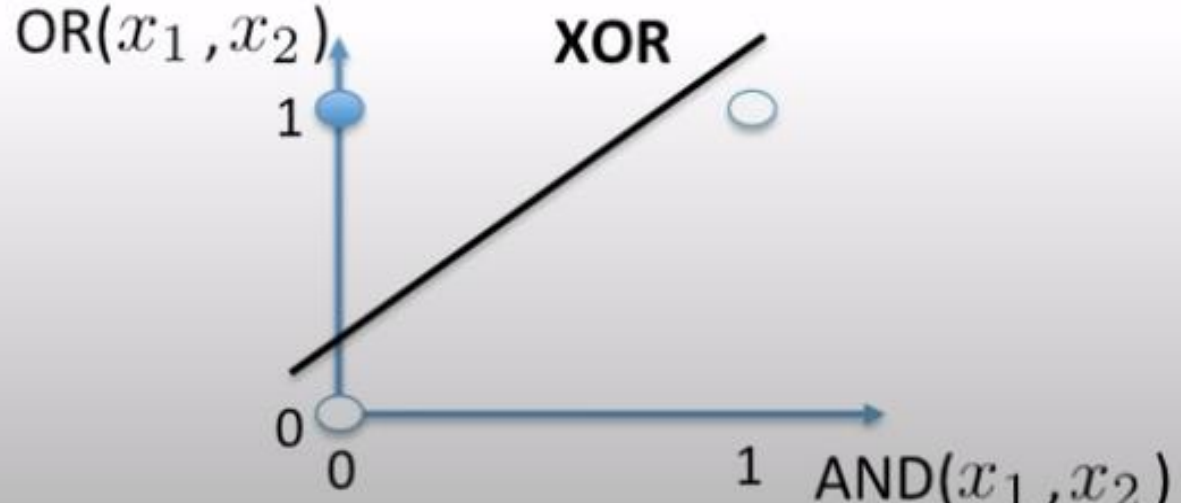
- En fait, un classifieur linéaire ne peut même pas apprendre XOR!



Limites des classifieurs linéaires

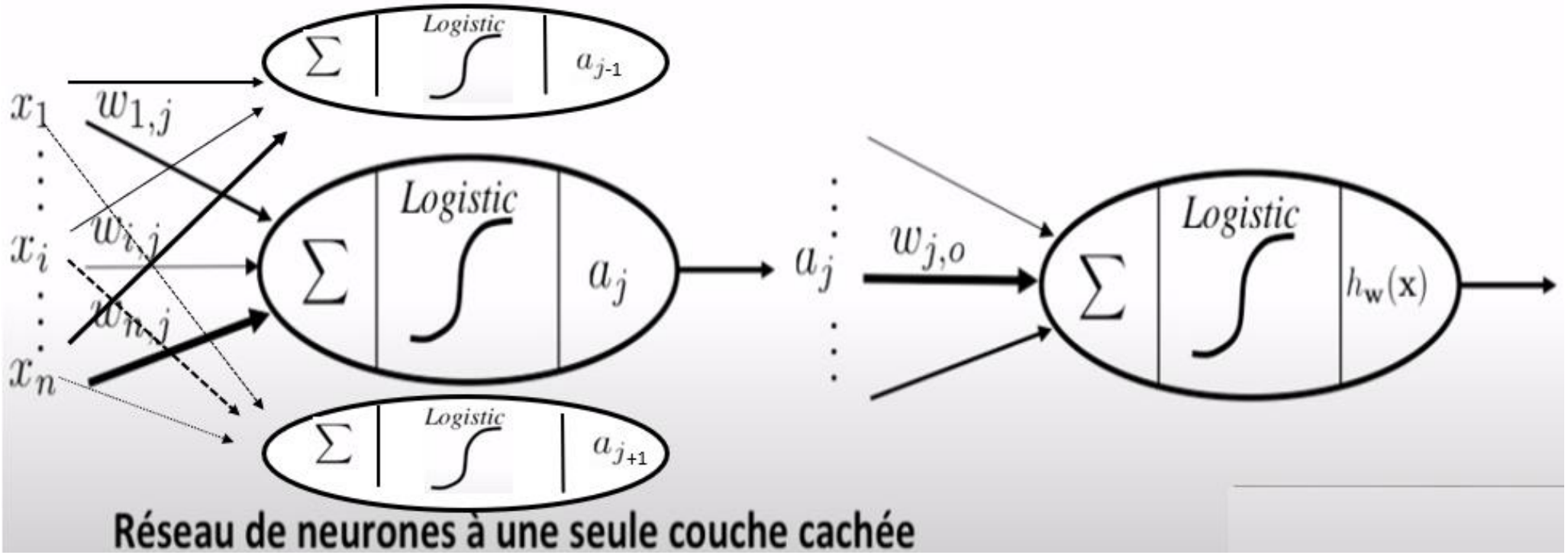
- Par contre, on pourrait transformer l'entrée de façon à rendre le problème linéairement séparable sous cette nouvelle représentation
- Dans le cas de XOR, on pourrait remplacer
  - x_1 par AND(x_1, x_2) et
  - x_2 par OR(x_1, x_2)





Artificial Neural Network (ANN)

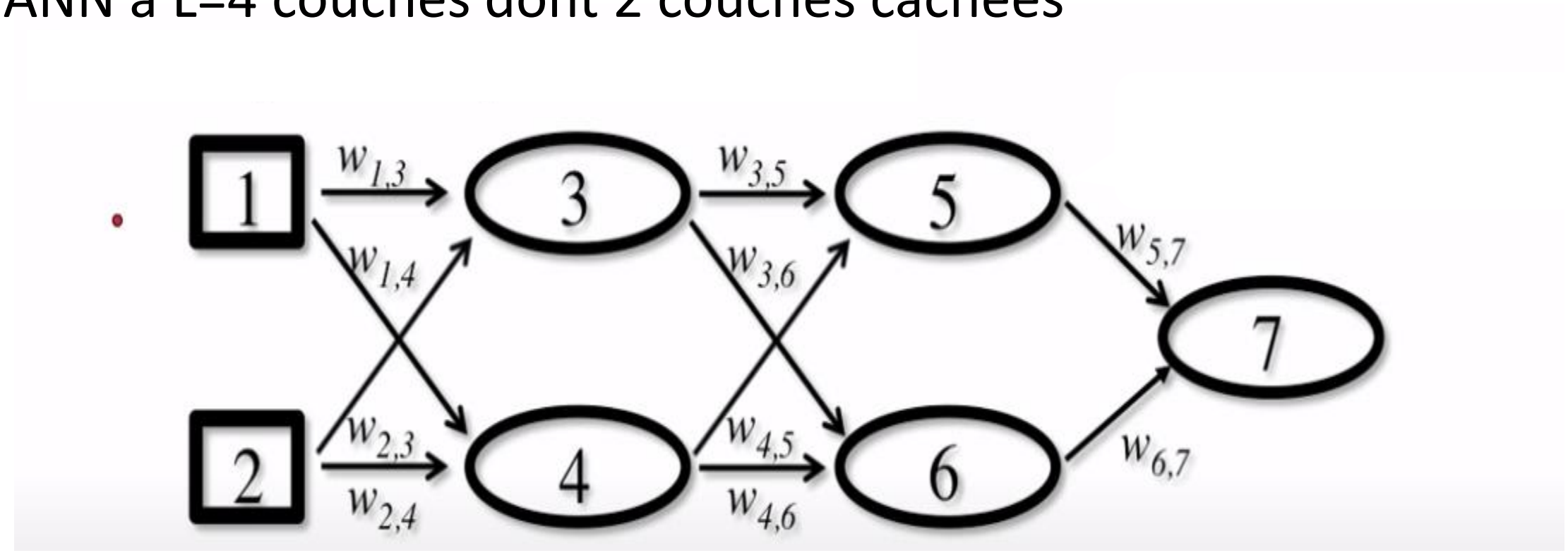
- **Idée**: apprendre les poids du classifieur linéaire **et** une transformation qui va rendre le problème linéairement séparable



Réseau de neurones à une seule couche cachée

ANN à L couches

- ANN à L=4 couches dont 2 couches cachées



3. Fonctions d’activation (activation functions)

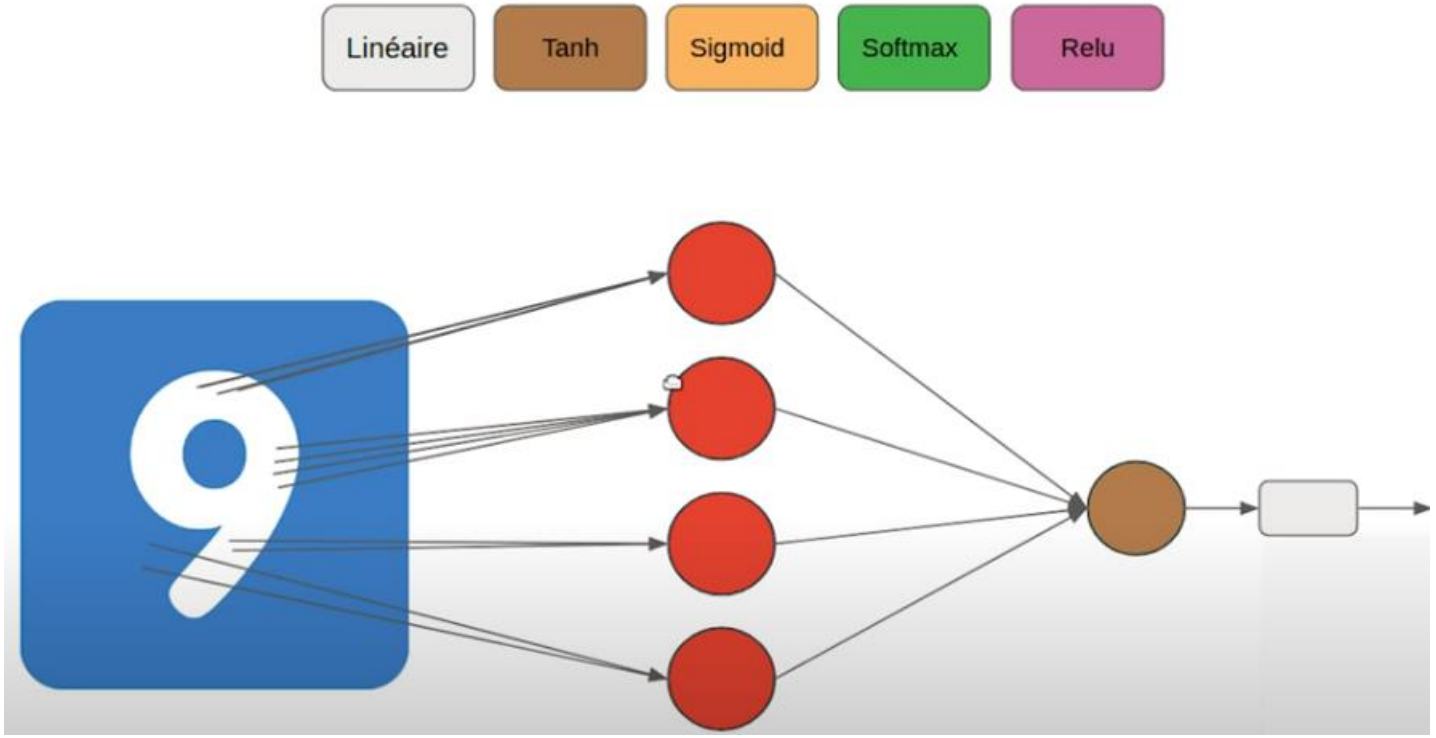
- Fonctions d’activation pour les neurones de sortie

- Fonctions d’activation pour les neurones des couches cachées

Fonctions d'activation pour les neurones de sortie : linéaire

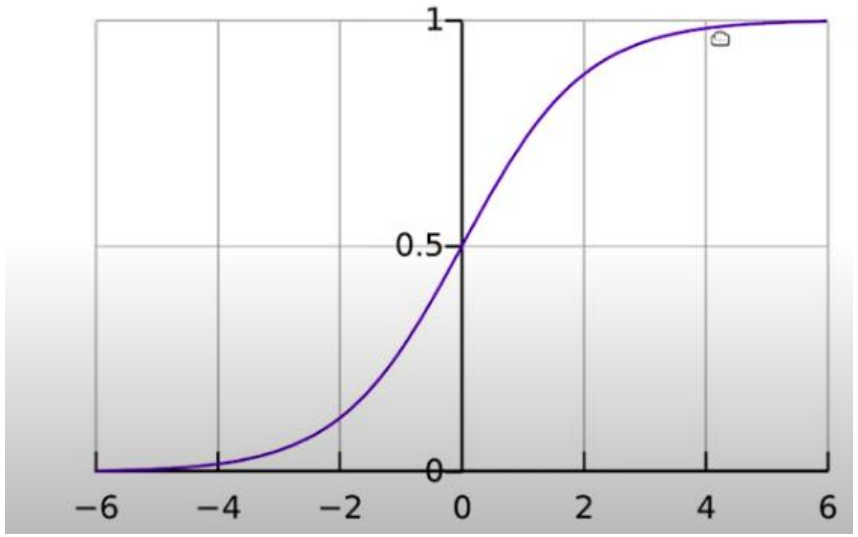
- Une activation linéaire signifie que la valeur d'activation correspond à la valeur de pré-activation : les valeurs obtenues varient dans IR.

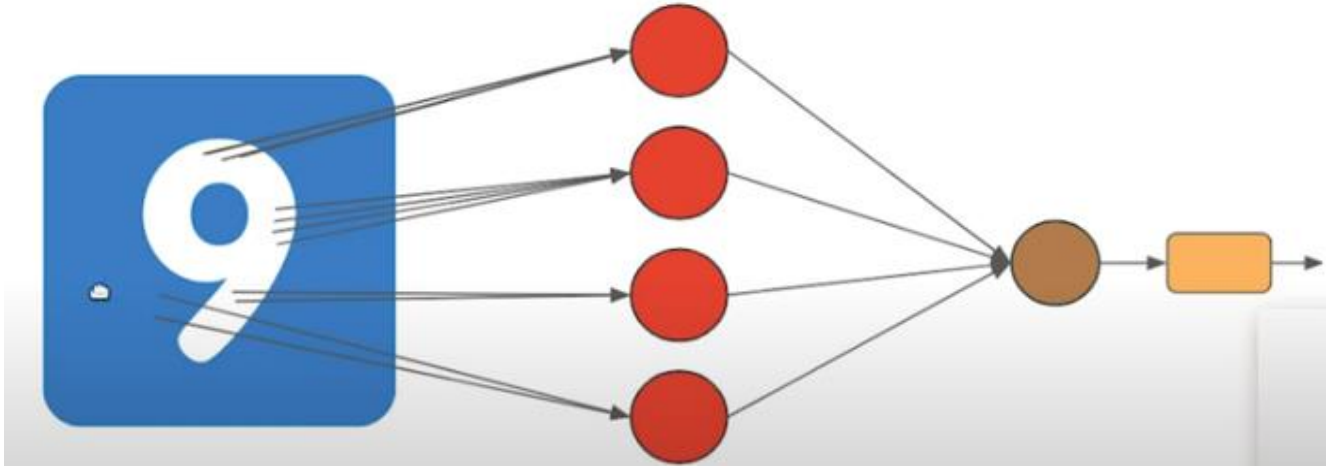
- Cette fonction servirait, par exemple, à estimer le prix d'une maison et plus généralement pour une régression.



Fonctions d’activation pour les neurones de sortie : Sigmoïde

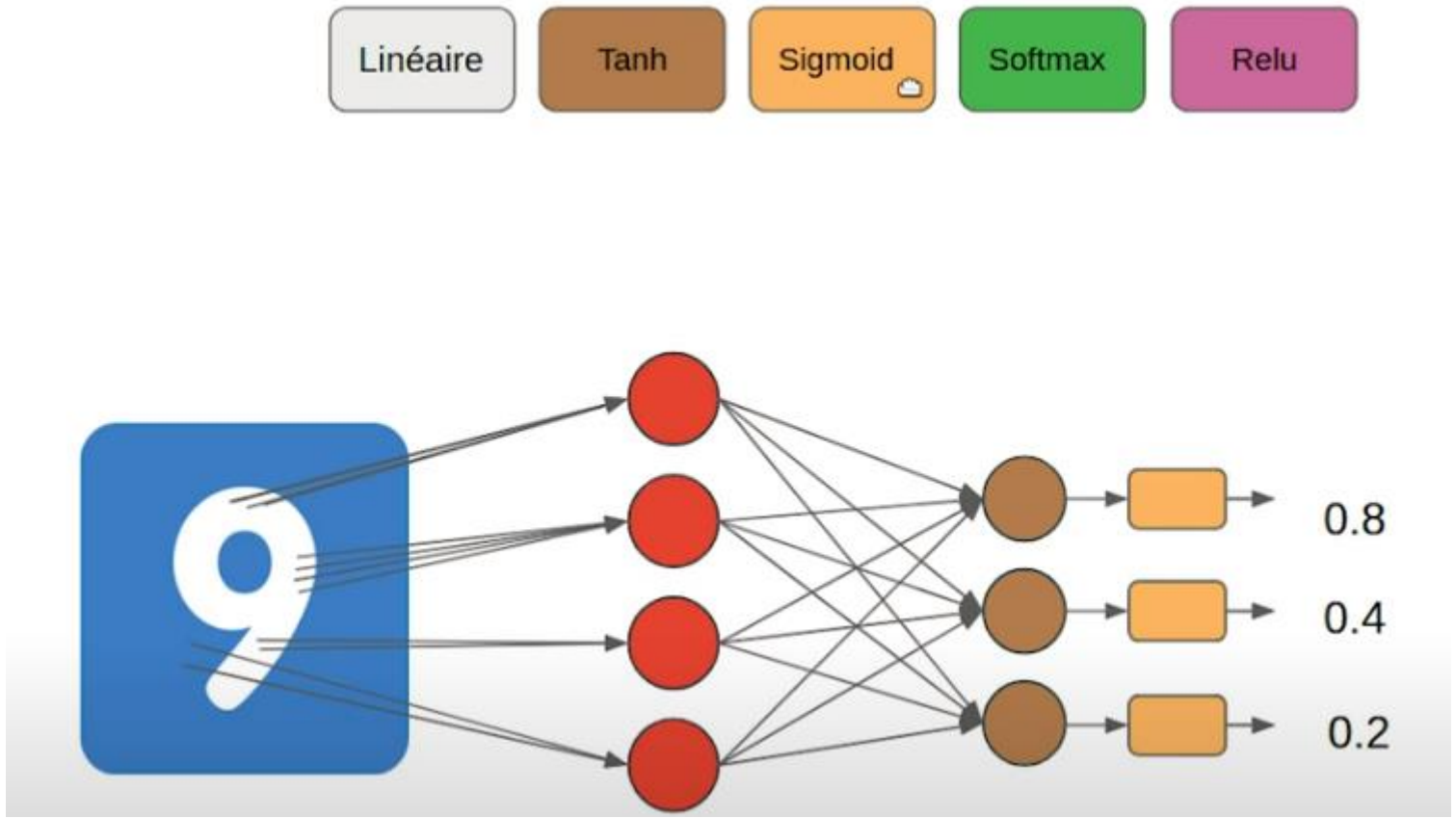
- La fonction Sigmoïde donne une valeur entre 0 et 1, une probabilité : elle est donc très utilisée pour les classification binaire.





Fonctions d’activation pour les neurones de sortie : Sigmoïde

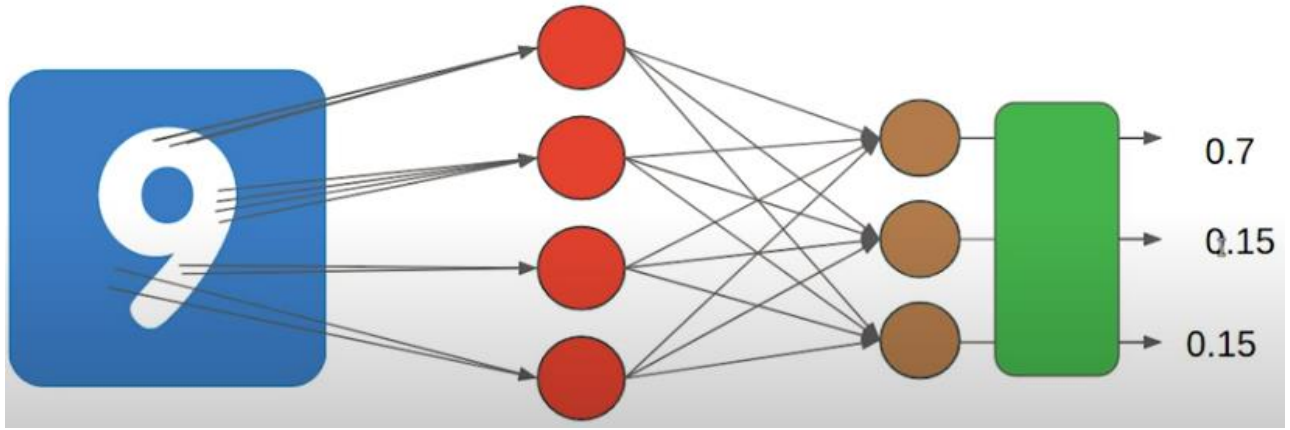
- En utilisant une Sigmoïde dans un réseau à plusieurs sorties donne des probabilités dont la somme n’est pas égale à 1.



Fonctions d’activation pour les neurones de sortie : Softmax

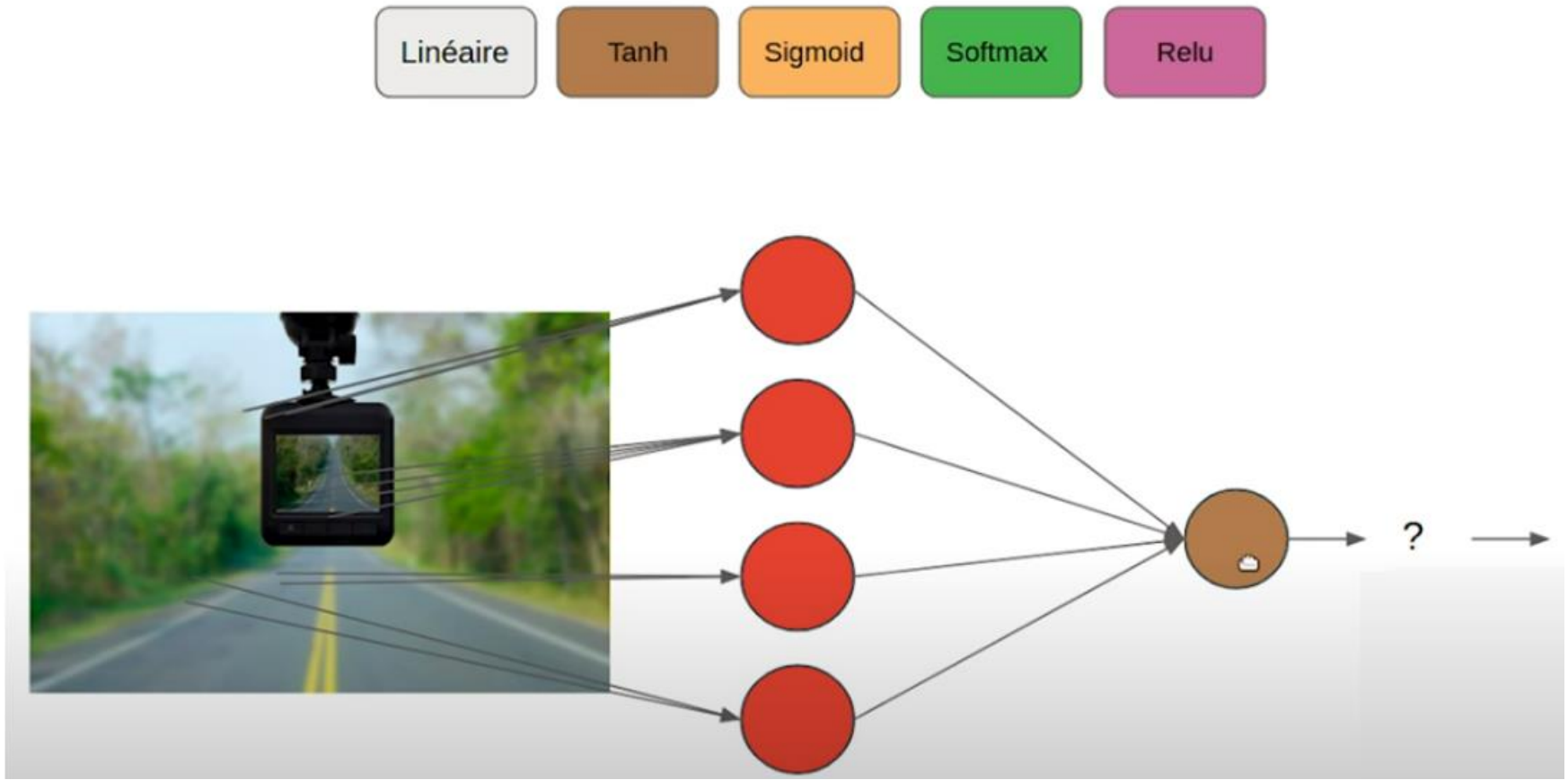
- La fonction Softmax permet de transformer un vecteur réel en vecteur de probabilité.
- On l’utilise souvent dans la couche finale d’un modèle de classification, notamment pour les problèmes multi-classes :

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$



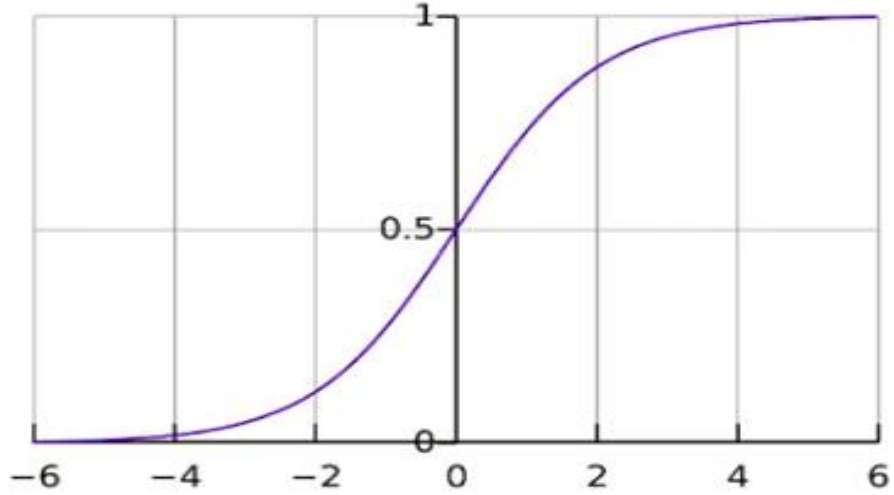
Fonctions d'activation pour les neurones de sortie : Tanh

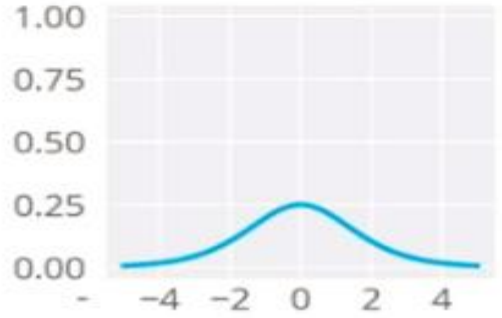
- On voudrait utiliser un RN pour estimer l'angle dans l'intervalle [-90° ; +90°] avec lequel une voiture autonome doit tourner.



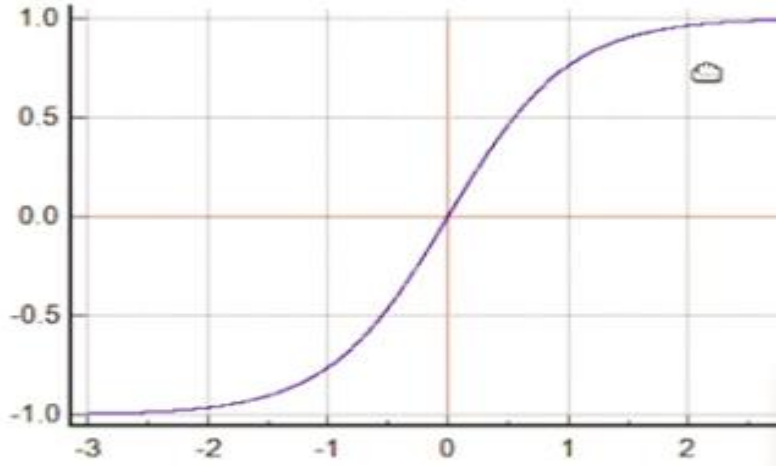
- Le gradient de la Sigmoïde se rapproche très rapidement de 0 avec un risque de ***vanishing gradient*** important contrairement au celui du Tanh qui se rapproche de 0 beaucoup plus lentement.
- Dans le cas où la sortie n'est pas une probabilité, il serait plus pertinent d'utiliser Tanh.

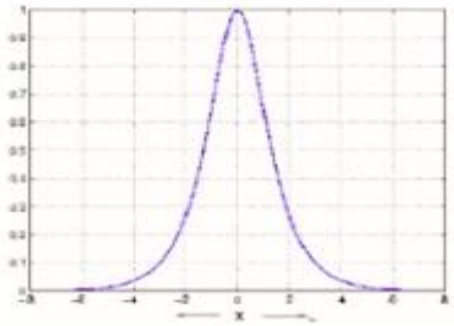
Sigmoid





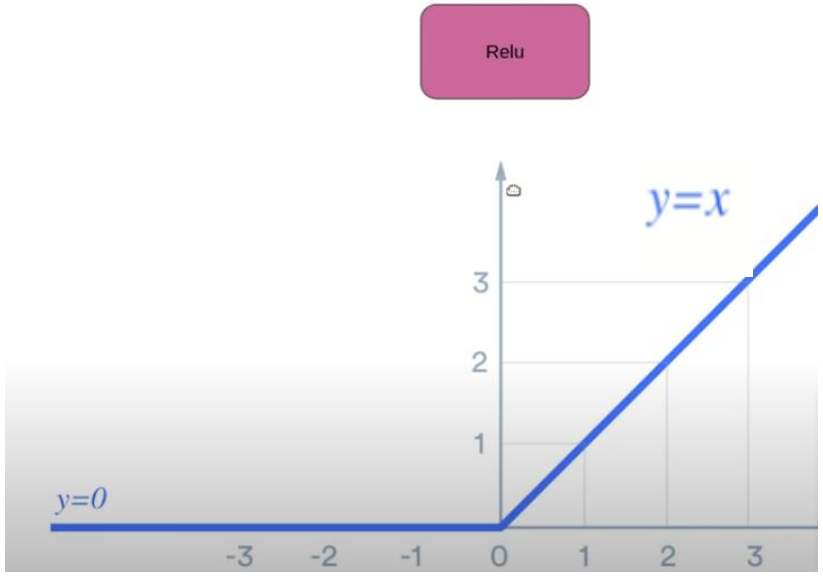
Tanh

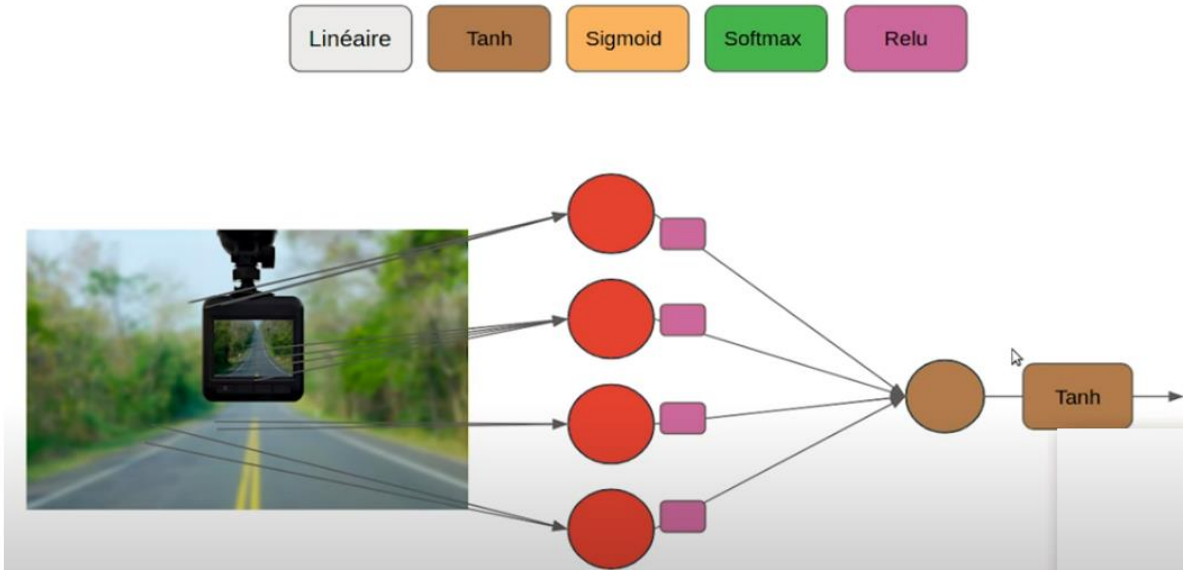




Fonction d'activation pour les neurones des couches cachées : ReLU

- La fonction Rectified Linear Unit (ReLU) est la fonction d'activation la plus simple et la plus utilisée. Elle donne x si x est supérieur à 0 et 0 sinon : le maximum entre x et 0.
- Cette fonction permet d'effectuer un filtre sur nos données : elle laisse passer les valeurs positives ($x > 0$) dans les couches suivantes du réseau de neurones.
- Elle est utilisée ans les couches intermédiaires.

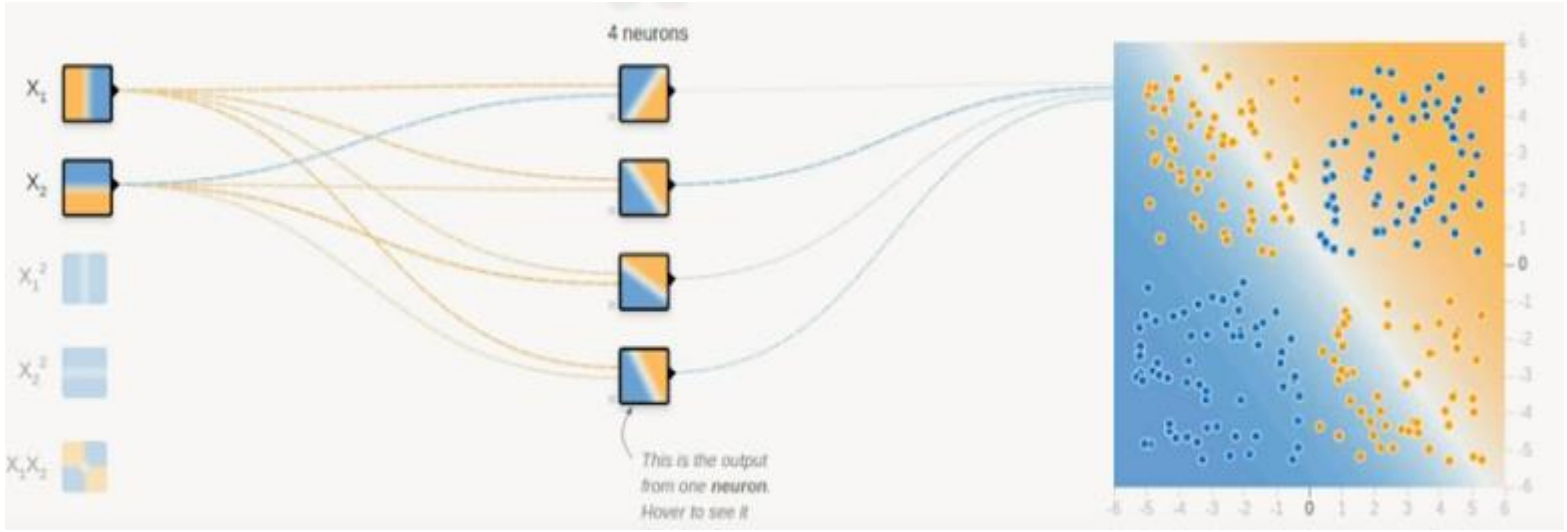




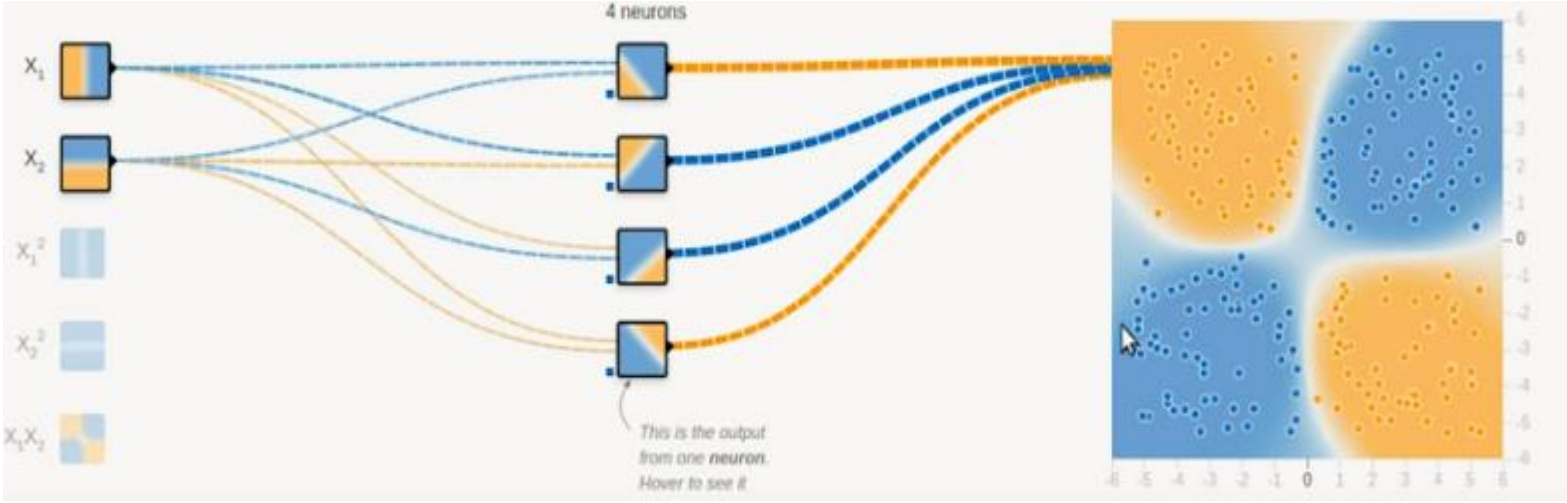
Fonction d’activation pour les neurones des couches cachées : ReLU

- Bonne propriété de convergence car le gradient est linéaire, et donc a moins tendance de se rapprocher de 0.
- Par contre pour les valeurs négatives le gradient sera nulle d’où, parfois, des neurones qui ne sont jamais activés.

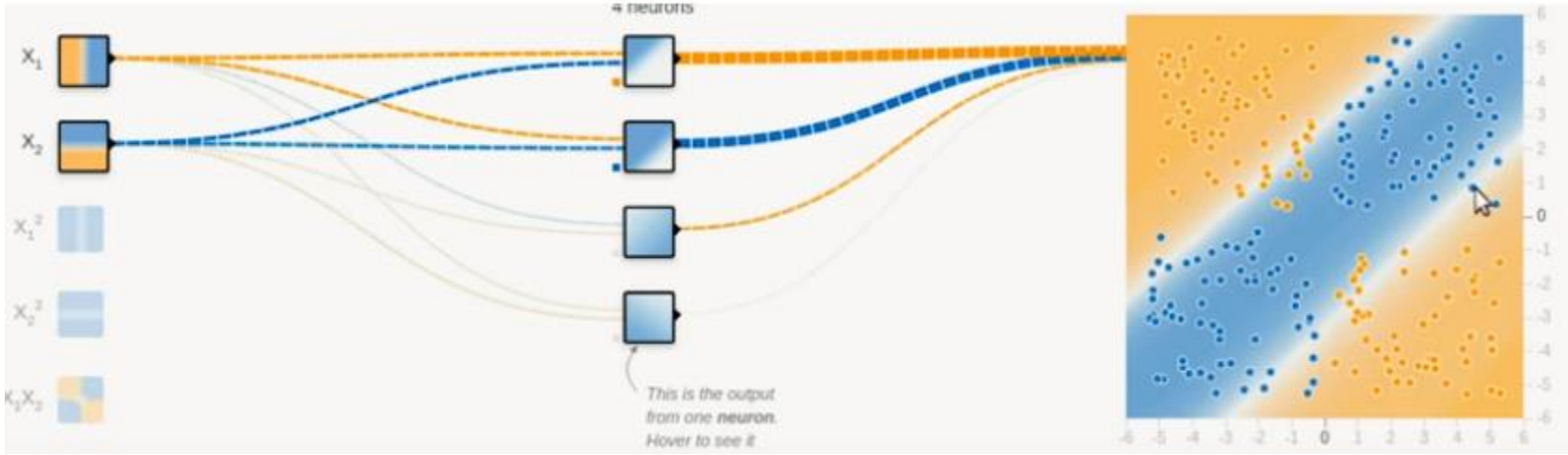
- FA : Linéaire



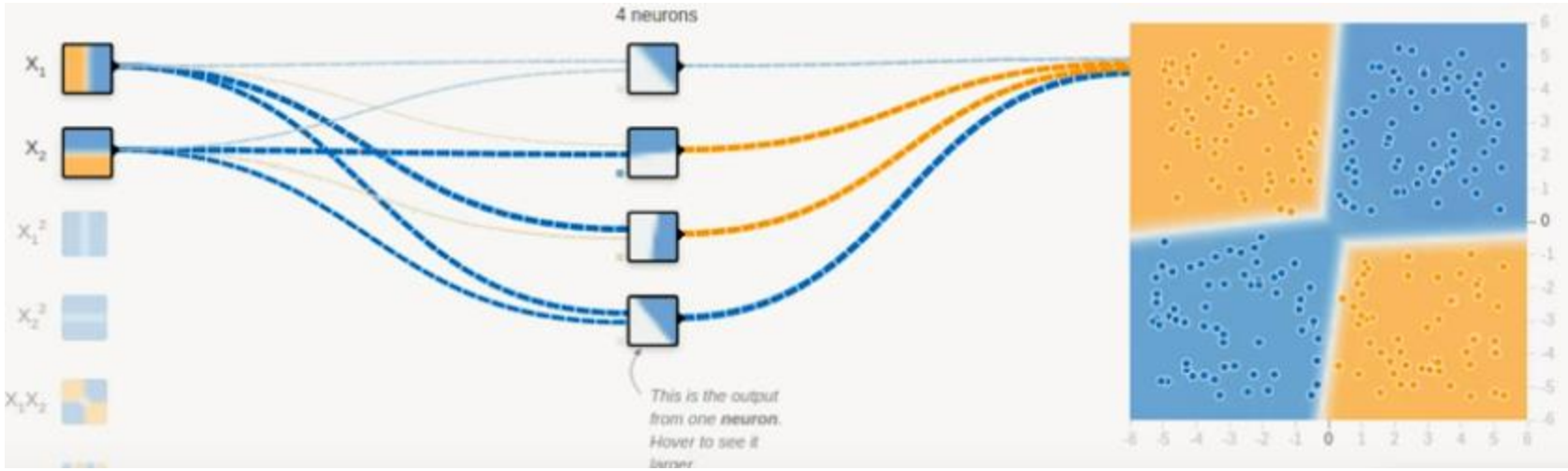
- FA : Tanh



- FA : Sigmoide



- FA : Relu



3. Fonctions de perte (loss functions)

Types de fonctions de perte : Deux principaux types de fonctions de perte existent, correspondant aux deux types majeurs de réseaux de neurones : les fonctions de perte pour la régression et celles pour la classification.

- **Fonctions de perte pour la régression :** utilisées dans les réseaux de neurones de régression. À partir d'une valeur en entrée, le modèle prédit une valeur de sortie correspondante (plutôt qu'une étiquette pré-sélectionnée). **Ex :** Erreur Quadratique Moyenne (Mean Squared Error, MSE), Erreur Absolue Moyenne (Mean Absolute Error, MAE).

- **Fonctions de perte pour la classification :** utilisées dans les réseaux de neurones de classification. Pour une entrée donnée, le réseau de neurones produit un vecteur de probabilités indiquant la probabilité que l'entrée appartienne à différentes catégories prédéfinies — la catégorie avec la probabilité la plus élevée est alors choisie. **Ex :** Entropie Croisée Binaire (Binary Cross-Entropy), Entropie Croisée Catégorielle (Categorical Cross-Entropy)

Mean Squared Error (MSE)

L'une des fonctions de perte les plus populaires, l'erreur quadratique moyenne (MSE) calcule la moyenne des carrés des différences entre les valeurs cibles et les valeurs prédites.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

- La MSE est une fonction convexe avec un minimum global bien défini.
- Cela permet d'utiliser plus facilement l'optimisation par descente de gradient pour ajuster les valeurs des poids.
- Inconvénient : très sensible aux valeurs aberrantes ; si une valeur prédite est beaucoup plus grande ou plus petite que sa valeur cible, cela augmentera considérablement la perte.

Mean Absolute Error (MAE)

La MAE calcule la moyenne des différences absolues entre les valeurs cibles et les valeurs prédites.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y^{(i)} - \hat{y}^{(i)}|$$

- Cette fonction de perte est utilisée comme alternative à la MSE dans certains cas.
- Comme mentionné précédemment, la MSE est très sensible aux valeurs aberrantes, ce qui peut affecter fortement la perte car la distance est élevée au carré.
- La MAE est utilisée lorsque les données d'entraînement contiennent un grand nombre de valeurs aberrantes afin de réduire cet effet.
- Inconvénient : lorsque la distance moyenne tend vers 0, l'optimisation par descente de gradient ne fonctionne plus, car la dérivée de la fonction en 0 est indéfinie.

Binary Cross-Entropy/Log Loss

Utilisée dans les modèles de classification binaire :

$$CE\ Loss = \frac{1}{n}\sum_{i=1}^{N} -\left(y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)\right)$$

- Les réseaux de neurones de classification produisent un vecteur de probabilités — la probabilité que l'entrée donnée corresponde à chacune des catégories prédéfinies ; puis, la catégorie ayant la plus haute probabilité est sélectionnée comme sortie finale.

- La Log Loss compare la valeur réelle (0 ou 1) avec la probabilité que l'entrée corresponde à cette catégorie (pi = probabilité que la catégorie soit 1 ; 1 — pi = probabilité que la catégorie soit 0).

Categorical Cross-Entropy Loss

Dans les cas où le nombre de classes est supérieur à deux, nous utilisons l'entropie croisée catégorique — cela suit un processus très similaire à l'entropie croisée binaire.

$$CE\ Loss = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \cdot log(p_{ij})$$

- L'entropie croisée binaire est un cas particulier de l'entropie croisée catégorique, où M = 2.
- L'entropie croisée catégorique et l'entropie croisée catégorique éparse (sparse categorical entropy) ont la même fonction de perte.
- La seule différence réside dans le format de *Yi* (c'est-à-dire les étiquettes réelles).

Categorical Cross-Entropy Loss

La seule différence réside dans le format de Yi (c'est-à-dire les étiquettes réelles) :
Exemple : pour une classification à 3 classes

- si *Yi* est encodé en one-hot : [1,0,0], [0,1,0], [0,0,1], utilisez l'entropie croisée catégorique
- mais si *Yi* est un entier : [1], [2], [3], utilisez l'entropie croisée catégorique éparse (Sparse Categorical Cross-Entropy).

- L'utilisation dépend entièrement de la façon dont vous chargez votre jeu de données.
- Un avantage de l'utilisation de l'entropie croisée catégorique éparse est qu'elle permet d'économiser du temps en mémoire ainsi qu'en calcul, car elle utilise simplement un entier pour une classe, plutôt qu'un vecteur complet.

5. Optimiseurs

- Exemple : cas de la regression linéaire
- **Modèle de régression linéaire :**

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- **Notations :**
  - $\hat{y}$ est la valeur prédite (se prononce généralement "y-chapeau"),
  - n est le nombre de variables,
  - x_i est la valeur de la i-ème variable.
- **Objectif de l'entraînement :**

  Trouver le vecteur θ qui minimise la RMSE.
- **Erreur quadratique moyenne (RMSE) :**

$$\mathrm{RMSE}(X, h) = \sqrt{\frac{1}{m} \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)} \right)^2}$$

Solution analytique : Equation normale

- **Objectif**

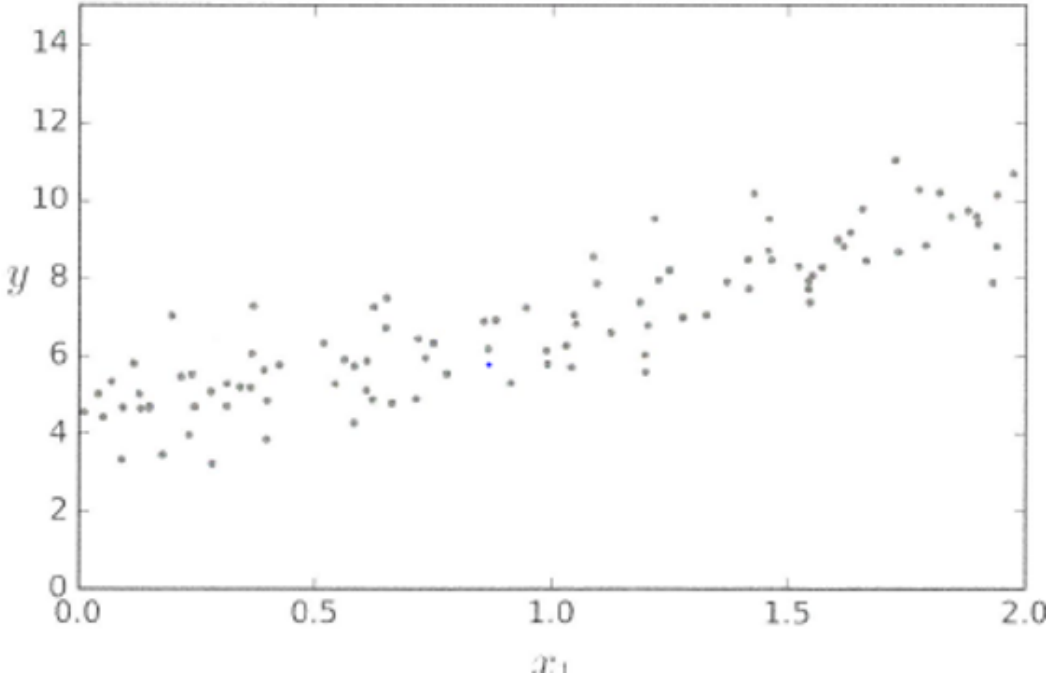
  Pour trouver la valeur de θ qui minimise la fonction de coût, on utilise une solution analytique appelée l'équation normale.

- **Équation Normale**

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

- **Exemple en Python**

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```



- **Résultat estimé de θ**

$$\theta_{\text{best}} = \begin{bmatrix} 4.21509616 \\ 2.77011339 \end{bmatrix}$$

- Solution par descente de gradient (GD)

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\theta^T \cdot x^{(i)} - y^{(i)}) \cdot x_j^{(i)}$$

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T \cdot (X \cdot \theta - y)$$

Une fois que vous avez le vecteur gradient (qui pointe vers le haut), il vous suffit d'aller dans la direction opposée pour descendre. Ce qui revient à soustraire $\nabla_\theta MSE(\theta)$ de θ. C'est ici qu'apparaît le taux d'apprentissage η : multipliez le vecteur gradient par η pour déterminer le pas de la progression vers le bas :

$$\theta \leftarrow \theta - \eta \nabla_\theta MSE(\theta)$$

• Solution par descente de gradient (GD)

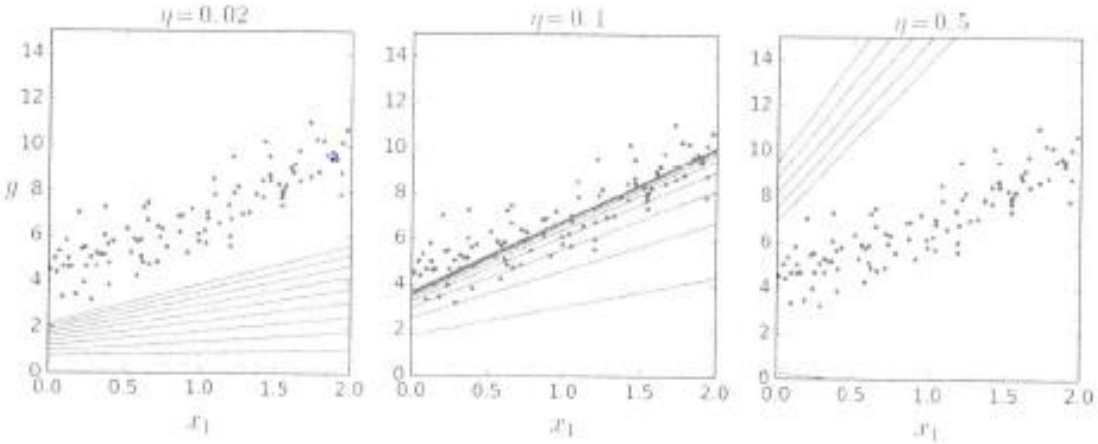
Voyons une implémentation rapide de cet algorithme:

```
eta = 0.1 # taux d'apprentissage
n_iterations = 1000
m = 100 # le nombre d'observations (len(X))

theta = np.random.randn(2,1) # initialisation aléatoire

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

>>> theta
array([[4.21509616],
       [2.77011339]])
```

Algorithme :

For every epoch:

$$\theta = \theta - \alpha \cdot \nabla J(\theta)$$



Solution par le Gradient Stochastique (SGD)

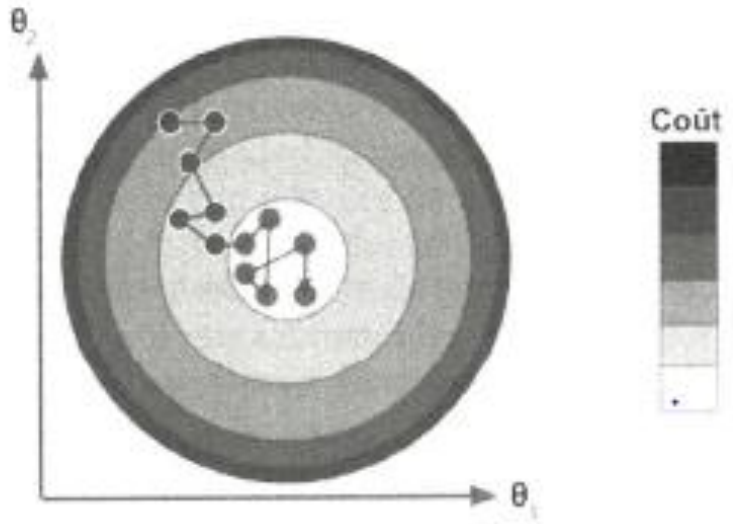
- Etant donné sa nature stochastique, cet algorithme est beaucoup moins régulier qu'une descente de gradient ordinaire.

- Au lieu de décroitre doucement jusqu'à atteindre le minimum, la fonction de coût va effectuer des sauts vers le bat et ne décroitra qu'en moyenne : les valeurs finales sont bonnes, mais pas optimales.

- Parmi les avantages de cette méthode, la possibilité d'effectuer des mises à jour du résultat à l'aide de nouvelles observations.

Algorithme :

For every epoch:
    For every sample:
$$\theta = \theta - \alpha . \nabla_\theta J(\theta)$$



- Solution par le Gradient Stochastique (SGD)

```
n_epochs = 50
t0, t1 = 5, 50  # hyperparamètres d'échéancier d'apprentissage

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2, 1)  # init. aléatoire

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients

theta
# Résultat attendu :
# array([[4.21076011],
#        [2.74856079]])
```

Solution par le Gradient Stochastique (SGD)

- Par convention, des cycles de m (i.e. taille des données) itérations sont effectuées.
- Chacun de ces cycles est dit *epoch*.
- Alors que le code de la DG oridinaire présenté plus haut effectue 1000 itérations sur l'ensemble du jeu d'apprentissage, ce code-ci ne parcourt le jeu d'apprentissage qu'environ 50 fois et aboutit à une solution satisfaisante.
- Une solution au problème de convergence consiste à réduire progressivement le taux d'apprentissage : les pas sont grand au début puis ils réduisent progressivement permettant à l'algorithme de s'arrêter au minimum global.

• Solution par le Mini-Batch Gradient Stochastique

Algorithme :

For every epoch:
    For every N samples:
        $\theta = \theta - \alpha \, . \nabla_\theta J(\theta)$

- A chaque étape, au lieu de calculer les dérivées partielles sur l'ensemble du jeu d'entraînement ou sur une seule observation, cet algorithme calcule le gradient sur de petits sous-ensemble de taille N d'observations tirées aléatoirement appelé batch.

- La progression de l'algorithme est ainsi moins désordonnée que dans le SDG et aboutira un peu plus près du minimum.

Modèle ANN : pour résumer !

- Paramètres
  1. Poids,
  2. Biais
- Hyperparamètres
  1. Nombre de couches, nœuds dans chaque couche, fonction d’activation
  2. Fonction de coût ou perte, optimiseurs, taux d’apprentissage
  3. Taille du lot et nombre d’epochs

6. Démarche sur Keras

- Installer et importer les librairies
- Importer les données : dans le cas des images (MNIST, par exemple), visualiser des images
- Préparer les données : adapter la target à la loss function (i.e. categorical ou sparse_categorical). Dans le cas des images, flatten, normalize
- Construire un Train/Validation/Test split
- Construire le modèle :
  - `sequential` : choix de l'architecture du modèle et des fonction d'activation dans les couches et cachées et celle de la couche de sortie
  - `model.summary` : on vérifiera le nombre de paramètres à estimer dans chaque couche

- Lancer le modèle sur Train et Validation :
  - `modèle.compile : loss, optimizer, metric`
  - `model.fit  : Train, batch_size, epochs,verbose, validation_split/validation_data`

- Utiliser history = model.fit pour représenter les courbes d'apprentissage : courbes accuracy et loss sur les données Train et Validation

- Evaluer le modèle sur Test : `model.evaluate`

- Effectuer des prédictions avec le modèle : `model.predict`

- Sauvegarder le modèle : `model.save` après avoir installer h5py

- Loader le modèle :
  - redémarrer le notebook,
  - faire les imports nécessaires et exécuter les parties du code relatives aux données
  - faire les prédictions avec le modèle sauvegardé après avoir exécuté `load_model`