

Intelligence Artificielle et Analyse de données



Applications Python-PyTorch

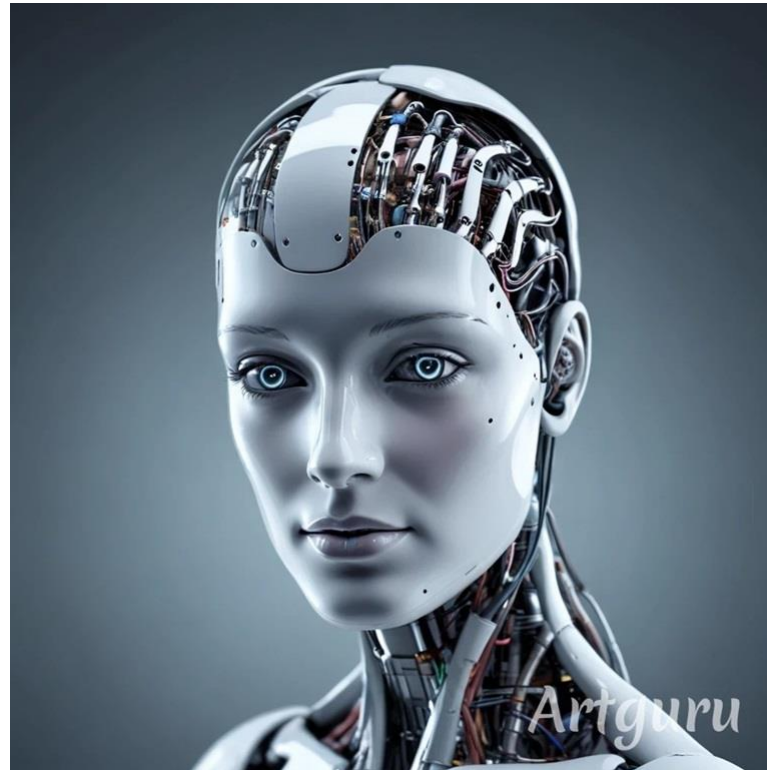


Johan Peralez

Plan du cours :

1. Définir l'Intelligence Artificielle (IA)
2. Prérequis (programmation-mathématiques)
3. Introduction aux réseaux de neurones (NNs, *Neural Networks*)
 - 4.1. Problèmes de régression
 - 4.2. Descente de gradient, dérivation automatique
 - 4.3. Un NN simple (MLP)
4. Les réseaux de neurones récurrents (RNNs)
 - 4.1. Prédiction de trajectoire
 - 4.2. Principe des RNNs
 - 4.3. Implémentation des RNNs
5. Les réseaux de neurones convolutifs (CNNs)
 - 5.1. Problèmes de classification
 - 5.2. Principe des CNNs
 - 5.3. Implémentation des CNNs
6. Applications
 - 6.1. Apprentissage par renforcement
 - 6.2. Classification d'images (données CIFAR)

1. Définir l'Intelligence Artificielle (IA)



- L'IA est une notion floue et qui évolue rapidement \Rightarrow sa définition dépend:
 - du domaine (traitement d'image, contrôle, etc.)
e.g. distinguer des visages vs faire marcher un robot
 - de l'époque (depuis le milieu du XXe siècle)
e.g. Deep Blue vs Alpha Go
- Exemples:
 - « L'**automatisation** d'activités que nous **associons à la pensée humaine**, comme la prise de décision, la résolution de problème ou l'apprentissage. » (BELLMAN 1978)
subjectif (nous ?)
 - « L'étude de comment **programmer les ordinateurs** pour qu'ils réalisent des tâches pour lesquelles les êtres humains sont **actuellement meilleurs**. » (RICH & KNIGHT 1991)
paradoxal (jeu d'échec vs football ?)
 - « Ensemble de **théories et de techniques** mises en œuvre en vue de réaliser des machines capables de **simuler l'intelligence humaine**. » (Larousse 2024)
vague (définir intelligence ?)

- Difficulté à définir l'IA = difficulté à définir l'Intelligence.

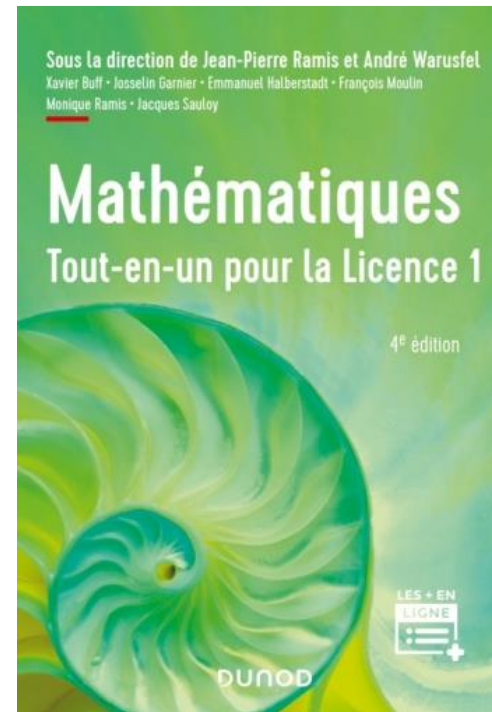
Selon [Larousse 2024], l'intelligence =

1. « Ensemble des fonctions mentales ayant pour objet la connaissance conceptuelle et rationnelle »
2. « Aptitude d'un être humain à s'adapter à une situation, à choisir des moyens d'action en fonction des circonstances »

- Approche “pragmatique”:

- IA = ensemble des méthodes qui sont habituellement classées dans l'IA par les gens de son domaine.
- En traitement de l'image : réseaux de neurones (NNs) convolutifs (CNNs), algorithmes de segmentation, etc.
- En contrôle : NNs récurrents (RNNs), algorithmes d'apprentissage par renforcement (RL), etc.

2. Prérequis (programmation-mathématiques)



- L'IA et l'analyse de données nécessitent des compétences :
 - en **programmation**
Python (langage le + utilisé en IA)
 - en **mathématiques**
Algèbre linéaire, calcul différentiel, probabilités, statistiques
- Ressources **Python** en ligne:
 - Cours et exos de base :
 - <https://www.learnpython.org/>
 - <https://www.france-ioi.org/algo/chapters.php>
 - Cours sur la Librairie NumPy :
 - <https://courspython.com/apprendre-numpy.html>
 - Programmer en ligne :
 - <https://www.programiz.com/python-programming/online-compiler/>
 - <https://colab.research.google.com/>

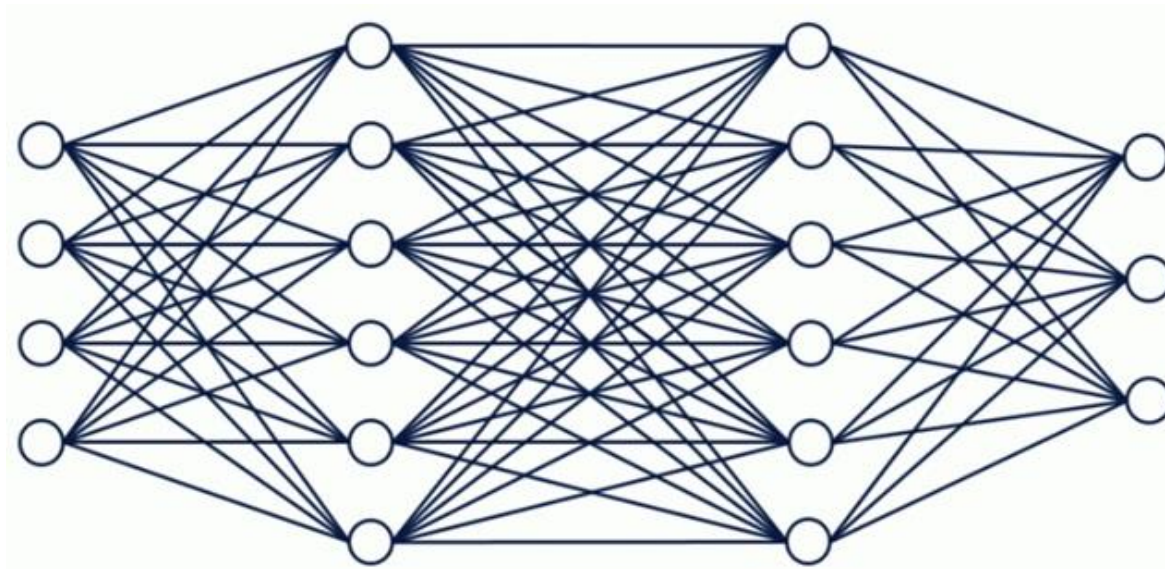


3. Introduction aux réseaux de neurones

3.1. Problèmes de régression

3.2. Descente de gradient, dérivation automatique

3.3. Un réseau de neurones simple (MLP)



3.1 Problèmes de régression

Formulation : **apprendre à prédire** une valeur de sortie y à partir d'une donnée d'entrée x

Diagram illustrating the regression equation $\hat{y} = h(x, \theta)$. The components are annotated as follows:

- \hat{y} : valeur \hat{y} prédite
- h : fonction h choisie « à la main »
- θ : paramètre(s) θ à apprendre

afin de minimiser une fonction **coût** L (*loss*) :

Diagram illustrating the cost function minimization $\min_{\theta} L(\{y\}, \{\hat{y}\})$. The components are annotated as follows:

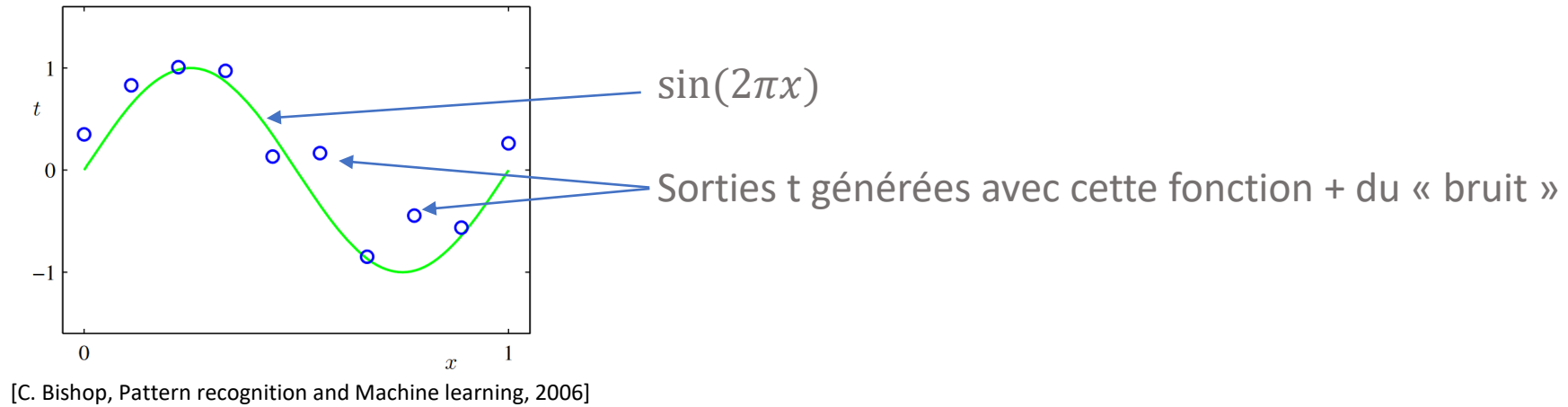
- $\{y\}$: données
- $\{\hat{y}\}$: prédictions

⇒ **problème d'optimisation**

Remarque : nous verrons plus tard que h peut-être un réseau de neurones ...

Exemple :

Données :



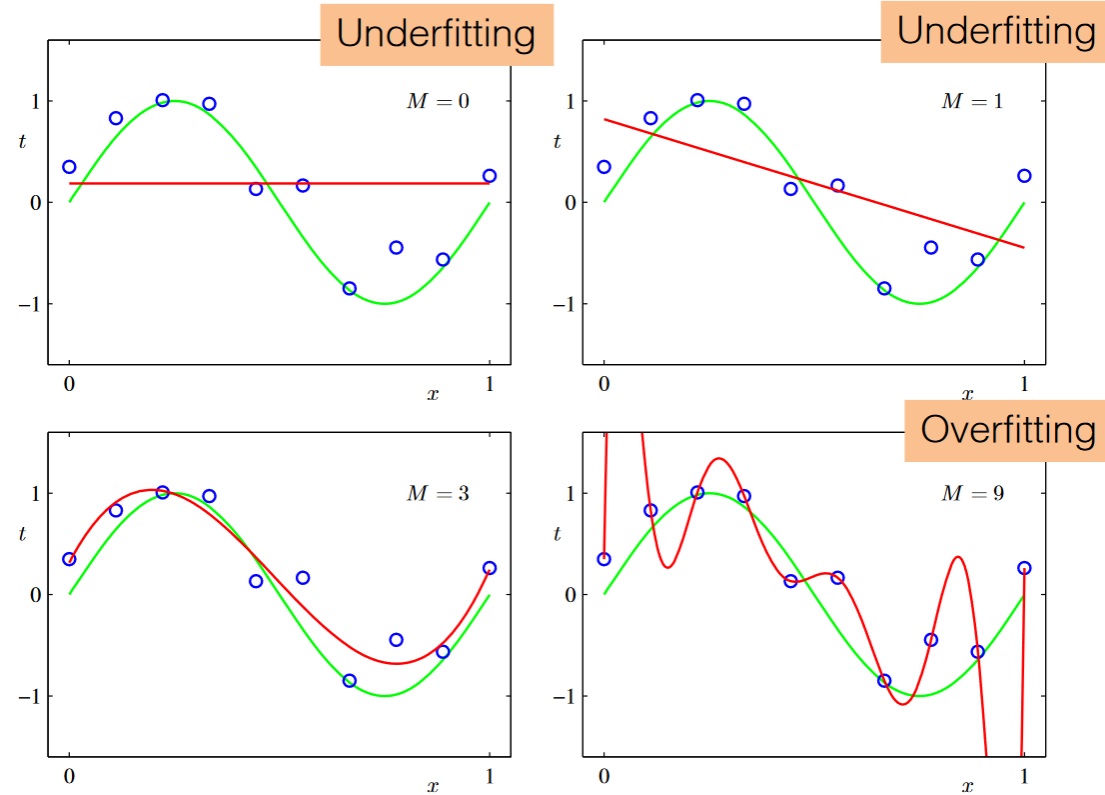
On choisit de prédire avec un polynôme d'ordre M :

$$\hat{t} = h(x, \theta) = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_M x^M$$

Et une fonction de coût « moindres carrés » :

$$L = \frac{1}{2} \sum (t - \hat{t})^2$$

Quel ordre M choisir pour notre polynôme ?



[C. Bishop, Pattern recognition and Machine learning, 2006]

- Pour M trop petit : problème de sous-apprentissage (***underfitting***).
- Pour M trop grand : problème de sur-apprentissage (***overfitting***).

Remarque (**polynômes de Lagrange**) : pour n données distinctes il existe un (unique) polynôme d'ordre $n-1$ qui passe exactement par chaque donnée.

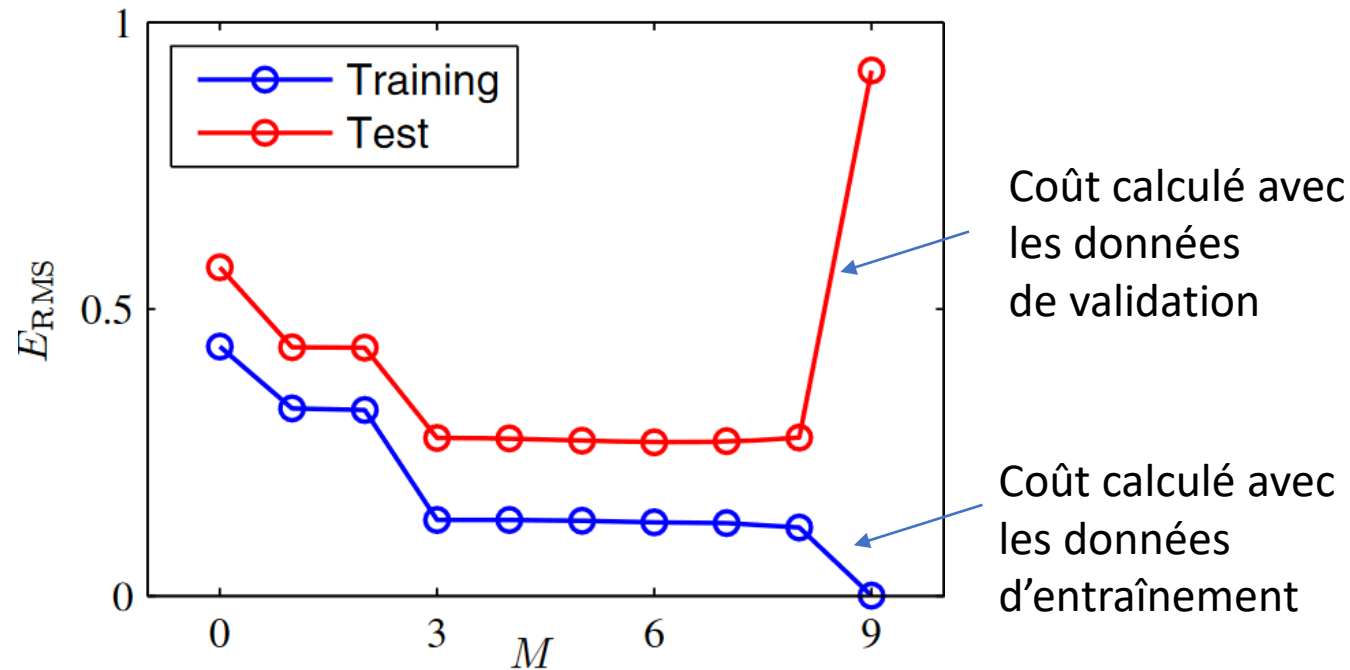
Séparation des données en, au moins, 2 ensembles :

- Données d'entraînement (*training set*).

Pour l'apprentissage de θ , i.e. la résolution du problème d'optimisation.

- Données de validation (*validation set*).

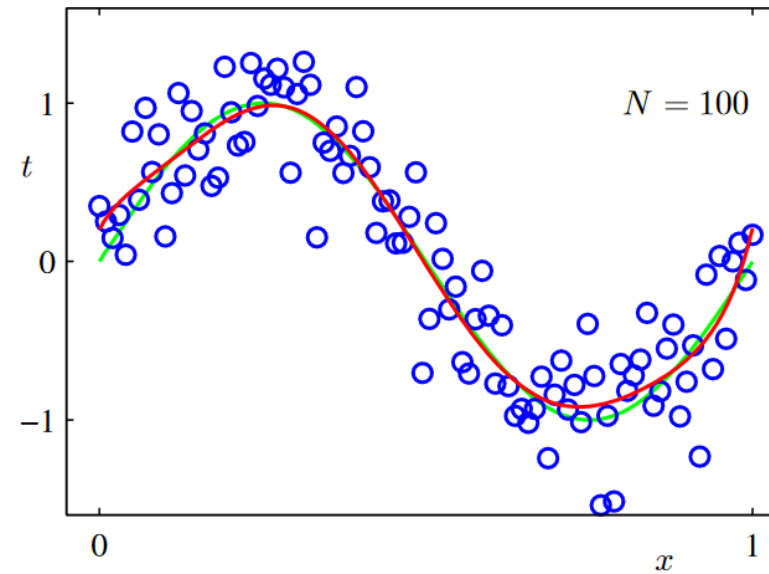
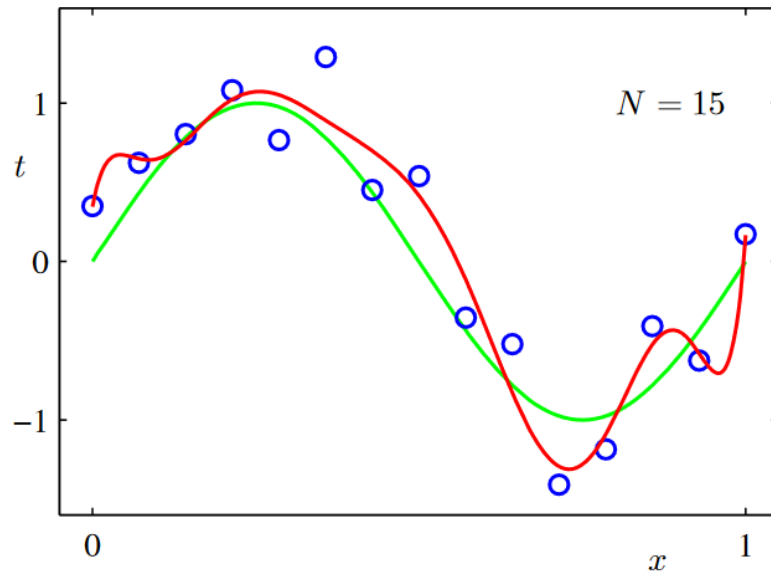
Permet de détecter le surapprentissage :



[C. Bishop, Pattern recognition and Machine learning, 2006]

Big Data ?

Augmenter (fortement) le nombre de données d'apprentissage permet de réduire le surapprentissage :



$M=9$

[C. Bishop, Pattern recognition and Machine learning, 2006]

3 grandes difficultés des problèmes d'apprentissage :

1. Expressivité

Mon modèle, i.e. ma fonction h , peut-elle apprendre des phénomènes complexes ?

2. Difficulté à entraîner

Le problème d'optimisation, i.e. minimiser la différence entre prédictions et données, est-il difficile ?

e.g. si on résout par descente de gradient, la fonction coût est-elle dérivable ? lisse ?

3. Généralisation

Comment mon modèle se comporte sur des données qui ne sont pas dans le *training set* ?

Capacité à interpoler / extrapoler ?

e.g. le surapprentissage implique une mauvaise généralisation.

Exercice 1 (Python, NumPy):

Générer et visualiser un jeu de données similaire à l'exemple ci-dessus.

1. Importer les librairies *numpy* et *matplotlib*

```
import numpy as np
from matplotlib import pyplot as plt
```

2. Paramètres

```
N = 30 # nombre de points
```

3. Générer (aléatoirement entre 0 et 1) les données d'entrées $\{x\}$, de taille N .

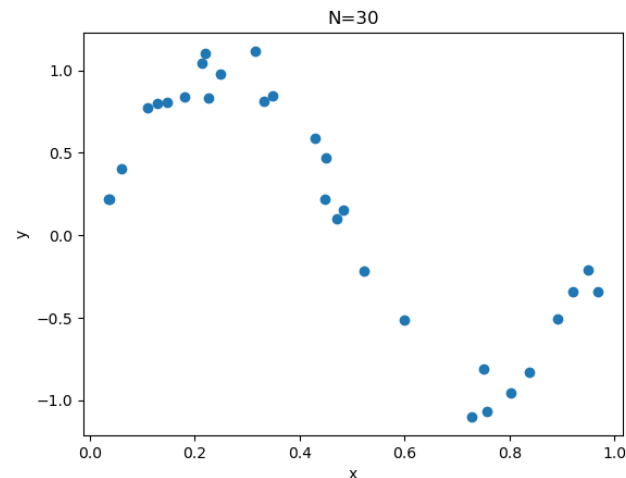
Utiliser la fonction `np.random.uniform(...)` pour utiliser une distribution uniforme.

4. Générer les données de sorties $\{y\}$, telles que $y = \sin(2\pi x) + w$.

Utiliser la fonction `np.random.normal(...)` pour générer w à partir d'une distribution normale centrée d'écart type (*standard deviation*) de 0.1

5. Visualiser le jeu de données

Utiliser `plt.plot`, `plt.legend`, `plt.xlabel`, etc.



3.3 Descente de gradient, dérivation automatique

Objectif : minimiser la fonction coût L .

Idée : mise à jour des paramètres, dans la direction qui fait diminuer L le plus fortement.

Mises à jour :

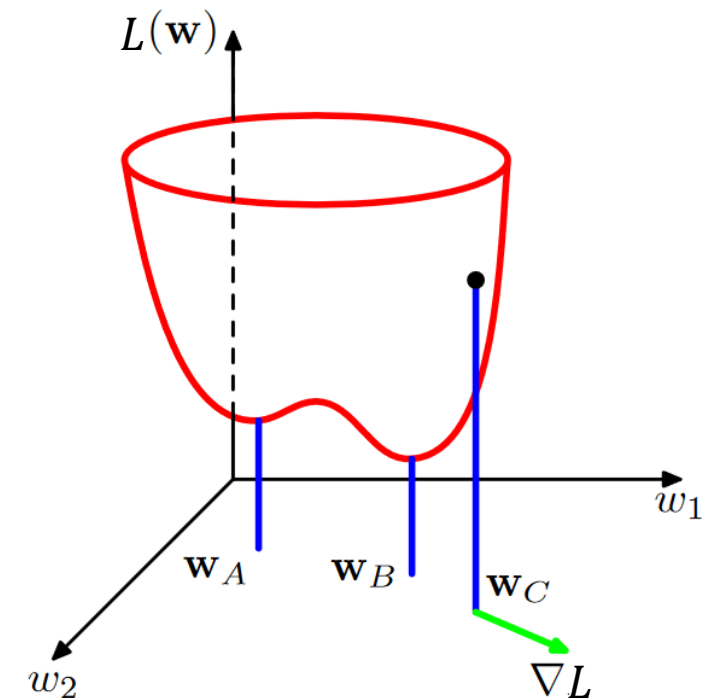
$$\theta^{k+1} = \theta^k - \eta \cdot \nabla_{\theta} L(h(x, \theta^k), y)$$

Paramètre à l'itération k Pas (*learning rate*) Gradient (se note aussi $\frac{\partial L}{\partial \theta}$)

Convergence : garantie pour un problème convexe, et

pour $0 < \eta < \frac{2}{k}$.

Illustration (non convexe): W_a est un minimum local.



Calculer le gradient $\nabla_{\theta} L = \left[\frac{\partial L}{\partial \theta_0}, \frac{\partial L}{\partial \theta_1}, \dots \right]^T$ revient à calculer les dérivées partielles $\frac{\partial L}{\partial \theta_i}$.

Ce calcul est à faire à chaque itération k (dépend de la valeur de θ) \Rightarrow calcul « à la main » prohibitif.

Calcul par différences finies :

$$\frac{\partial L}{\partial \theta_i} \approx \frac{L(h(x, \tilde{\theta}), y) - L(h(x, \theta), y)}{\epsilon}, \text{ avec } \tilde{\theta} = [\theta_0, \dots, \theta_{i-1}, \theta_i + \epsilon, \theta_{i+1}] \text{ et } \epsilon \text{ petit}$$

\Rightarrow **approximatif** et **coûteux** (nombreuses évaluations de L).

Calcul par dérivation automatique :

dérivation **exacte** (formelle) à l'aide d'un **logiciel**

\Rightarrow plus précis et plus rapide.

Exemple (bibliothèque Pytorch) :

```
import torch
x = torch.tensor([3.1], requires_grad = True) # initialise un scalaire (vecteur de taille 1)
y = x ** 2
grad = torch.autograd.grad(y, x) # calcule dy/dx = 2x
print(grad) # 6.2
```

```
x = torch.tensor([3.1, 2.0], requires_grad = True)
b = torch.tensor([5.5, 7.7])
y = b @ x # produit scalaire : y = b_1 * x_1 + b_2 * x_2
grad = torch.autograd.grad(y, x) # calcule dy/dx = [b_1, b_2]
print(grad) # [5.5, 7.7]
```

Exercice 1.suite (PyTorch):

Par descente de gradient, chercher le polynôme d'ordre 9 qui minimise la fonction coût L (moindres carrés). Compléter le programme:

```
# convertir données (numpy -> pytorch)
import torch
x, y = torch.tensor(x), torch.tensor(y)

# initialise theta
M = 3
theta = torch.zeros(M + 1, requires_grad = True)

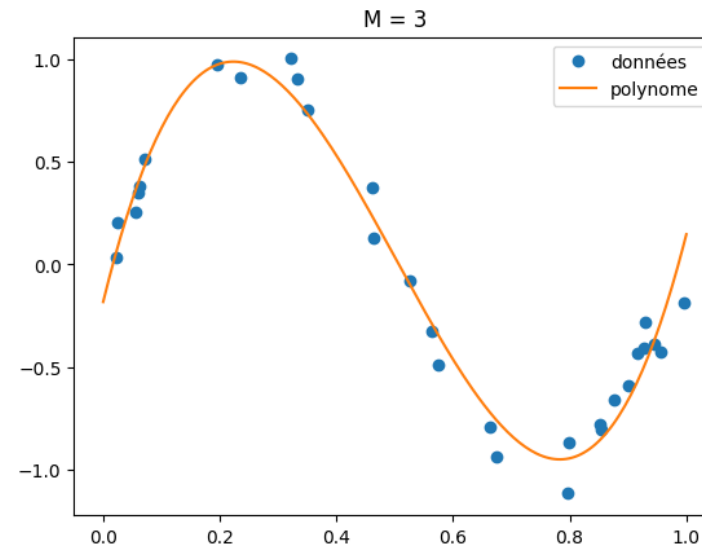
# fonction polynome
def get_predict(x, coefs):
    y_predict = torch.zeros_like(x)
    for i in range(len(theta)):
        y_predict = y_predict + coefs[i] * x**i
    return y_predict

# fonction coût
def get_loss(y_predict):
    return ((y_predict - y)**2).mean()

# descente de gradient
. . .
```

Tester pour:

- $M = 1, 2, 3, 8$.
- Différents *learning rate* (=0.01, 0.5, 2)



Pour aller plus loin : générer un 2^{ème} ensemble de données (*test set*) et tracer le coût en fonction de M .

3.2 Un réseau de neurones simple : le **MLP** (multi-layers perceptron)

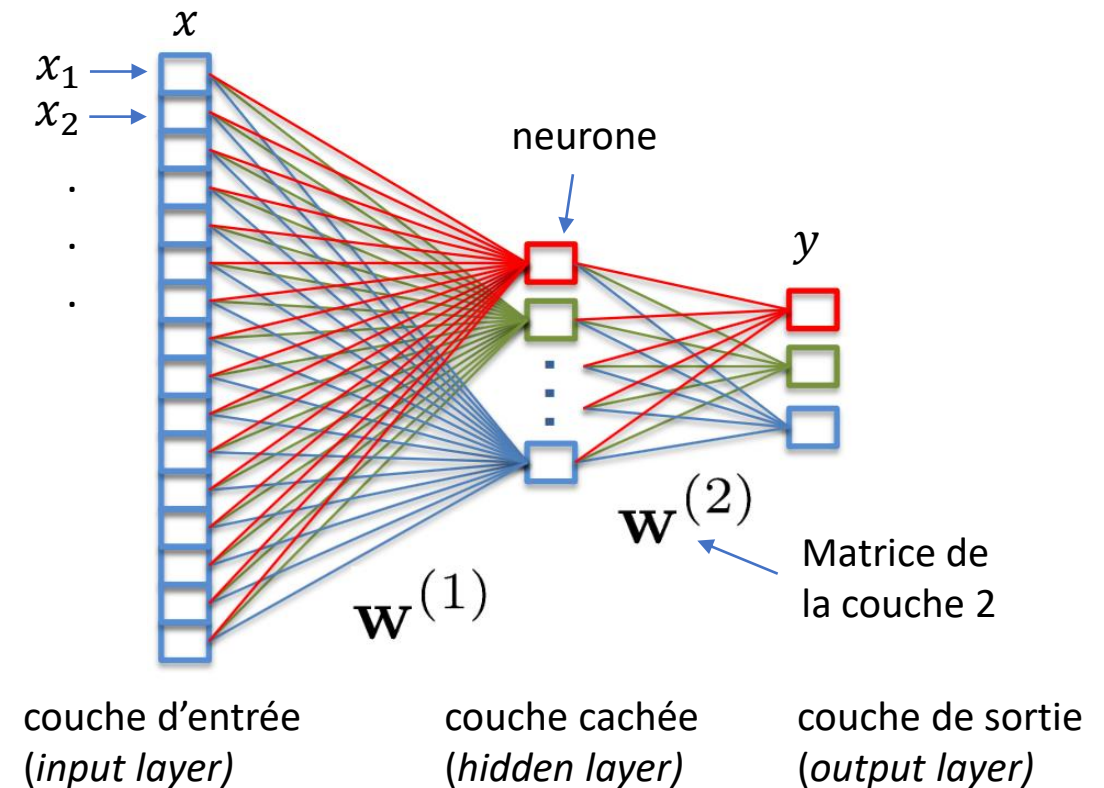
Principe du MLP (réseau dense) :

- Plusieurs couches (*layers*)
- Une couche transforme un vecteur d'entrée e en vecteur de sortie s .
- s est une combinaison linéaire de e , suivie (pour les couche cachées) par une non-linéarité σ .

⇒ le MLP est une fonction :

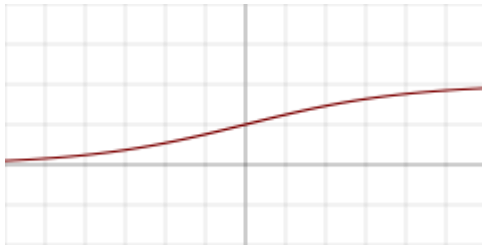
- Sortie d'une couche i : $s^{(i)} = \sigma(W^{(i)} \times e^{(i)} + b^{(i)})$
- Sortie du réseau : $y = W^{(2)} \times \sigma(W^{(1)} \times x + b^{(1)}) + b^{(2)}$
- Sortie d'un neurone : $s_j^{(i)} = w_j^{(i)} * e^{(i)} + b_j^{(i)}$

Entraîner un MLP = apprendre (identifier) ses paramètres,
i.e. les matrices $W^{(i)}$ (les poids) et les vecteurs $b^{(i)}$ (les biais).



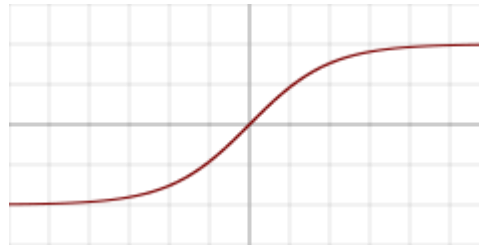
Non-linéarités σ (fonction d'activation):

- Fonction scalaire, i.e. $\sigma: \mathbb{R} \rightarrow \mathbb{R}$
- Sur un vecteur, s'applique élément par élément, i.e. $\sigma([x_1, x_2, \dots]) = [\sigma(x_1), \sigma(x_2), \dots]$
- Les activations les + courantes :



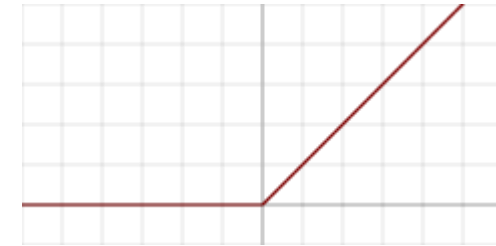
Sigmoïde

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



Tangente hyperbolique

$$\sigma(x) = \frac{2}{1 + e^{-2x}} - 1$$



ReLU

$$\sigma(x) = \max(0, x)$$

Théorème d'approximation universelle :

Un MLP peut approcher d'aussi près que l'on veut n'importe quelle fonction continue.

Pour cela une seule couche cachée suffit (à condition de prendre une taille suffisante, i.e. une matrice $W^{(1)}$ suffisamment grande) et une fonction d'activation non polynomiale (les fonctions ci-dessus respectent cette condition). Et ... il faut trouver les bons paramètres !

Implémentation du MLP (PyTorch):

Hérite de la classe *torch.nn.Module*.

Exemple pour une entrée de taille 3, une matrice $W^{(1)}$ (couche cachée) de taille 3×32 , et une sortie de taille 2.

```
class MLP(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1=torch.nn.Linear(3, 32) #  $W^1, b^1$ 
        self.fc2=torch.nn.Linear(32, 2) #  $W^2, b^2$ 

    def forward(self, x): # méthode appelée par MLP.forward(x) ou MLP(x)
        y = self.fc2(torch.functional.F.relu(self.fc1(x)))
        return y

model = MLP()
print(model)
print(list(model.parameters()))
```

Question: de quelle taille sont les vecteurs de biais ?

Entraînement du MLP :

PyTorch propose des outils pour faciliter la chaîne d'actions (*pipeline*) de l'entraînement :

- Classe `DataSet` pour la génération des données (*train set*, et *valid set*).
- L'implémentation de NNs.
- Des *optimizer* pour gérer la descente de gradient et la mise du *learning rate*.

Dans le code liés à ce cours, vous trouverez un exemple de pipeline. La descente de gradient utilisée est stochastique (*stochastic gradient descent*, **SGD**).

Avec SGD, à chaque itération, les données *train* sont découpées en sous-ensemble (***mini-batch***) et un calcul de gradient et une mise à jour des paramètres sont faits sur chaque *mini-batch*. Le découpage est fait aléatoirement à chaque itération (i.e. stochastique). Les avantages sont:

- Apprendre plus vite sur les grandes quantités de données (mises à jour plus fréquentes).
- Moins de risque d'être bloqué dans un minimum local.

Résultats (problème de régression simple):

Soit N le nombre de données d'entraînement, et M le nombre de colonnes de $W^{(1)}$.

On se propose d'étudier leur effet sur le sur-apprentissage (*overfitting*).

Pour $N=200$, $M=100$ (M =taille couche cachée)

i) courbe données + sortie MLP ii) training / data loss, fonction de N (overfit pour petit N)

Pour $N=10$, $M=100$ (overfit)

Pour $N=10$, $M=?$

=> Montrer aussi q'on peut réduire en réduisant le NN