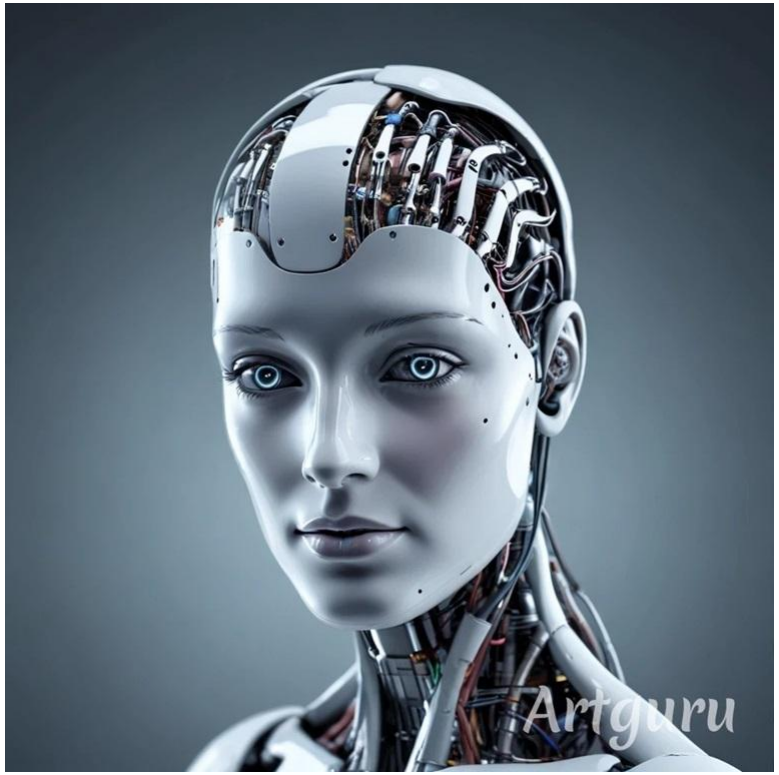
Intelligence Artificielle
et
Analyse de données

Applications : Python-PyTorch

Johan Peralez

1. Définir l'Intelligence Artificielle (IA)
2. Prérequis –ressources
3. Introduction aux réseaux de neurones (NNs, *Neural Networks*)
    - 3.1. Problèmes de régression
    - 3.2. Descente de gradient, dérivation automatique
    - 3.3. Un NN simple (MLP)
    - 3.4. Problèmes de classification
4. Les réseaux de neurones récurrents (RNNs)
    - 4.1. Données séquentielles
    - 4.2. Principaux RNNs
    - 4.3. Implémentation
5. Apprentissage pas renforcement (RL)
    - 5.1. Principe et définitions
    - 5.2. Equations de Bellman
    - 5.3. Algorithme tabulaire (Q-learning)
    - 5.4. Approximation des fonctions de valeurs (NNs)
6. Travaux Pratiques
    - 6.1. Classification d'images : les réseaux de neurones convolutifs (CNNs)
    - 6.2. RNNs
    - 6.3. RL
    - 6.4. RL

1. Définir l’Intelligence Artificielle (IA)



- L’IA est une notion floue et qui évolue rapidement ⇒ 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 ?)

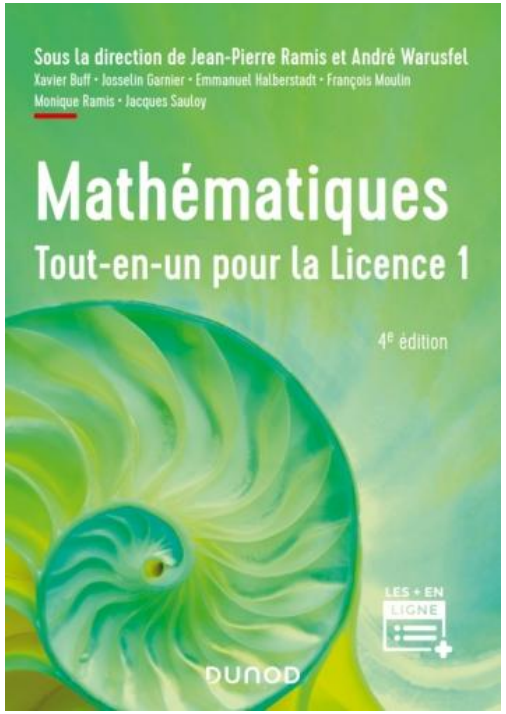
- Difficulté à définir l'IA = difficulté à définir l'Intelligence.

  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 »

- Définition ''pragmatique'':
  - IA = ensemble des méthodes qui sont habituellement classées dans l'IA par les spécialistes 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 -ressources (programmation-mathématiques)



- L'IA et l'analyse de données nécessitent des compétences :
  - en **programmation**

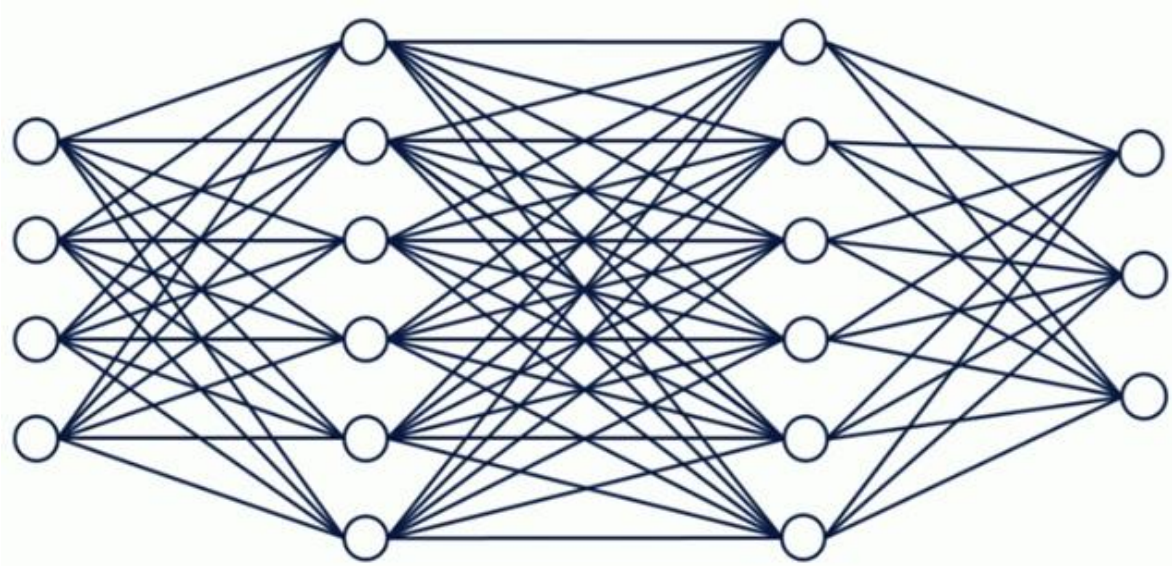
    Python (langage le + utilisé en IA)
  - en **mathématiques**

    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
    - https://courspython.com/apprendre-numpy.html
  - Installation (programmer en local)
    - https://anaconda.org
    - https://pytorch.org/get-started/locally/
  - Programmer en ligne :
    - https://www.programiz.com/python-programming/online-compiler/
    - https://colab.research.google.com/

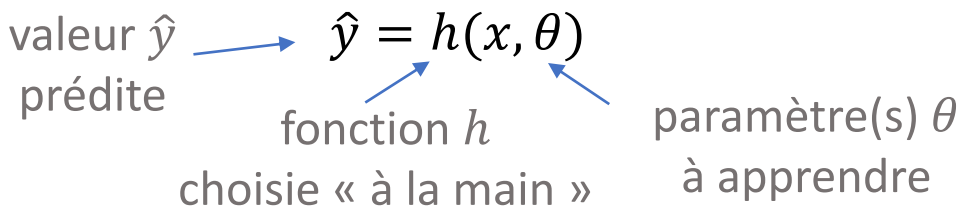
3. Introduction aux réseaux de neurones

3.1. Problèmes de régression
3.2. Descente de gradient, dérivation automatique
3.3. Un réseau de neurones simple (MLP)
3.4. Titanic

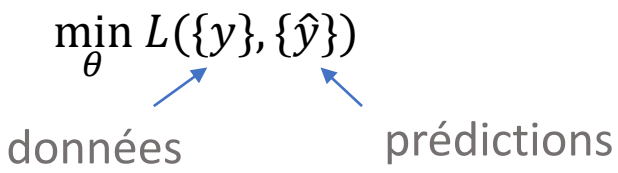


3.1 Problèmes de régression

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



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

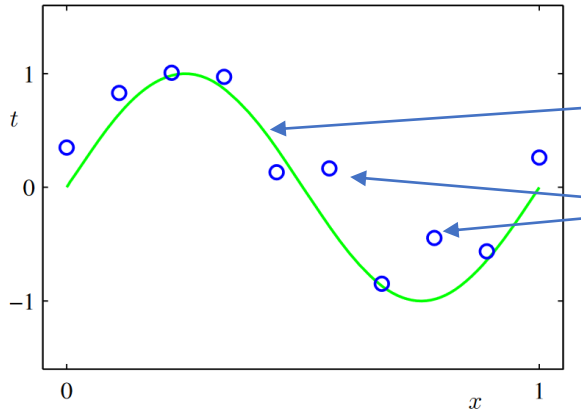


$\Rightarrow$ **problème d'optimisation**

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

Exemple :

Données :



$\sin(2\pi x)$

Sorties t générées avec cette fonction + du « bruit »

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

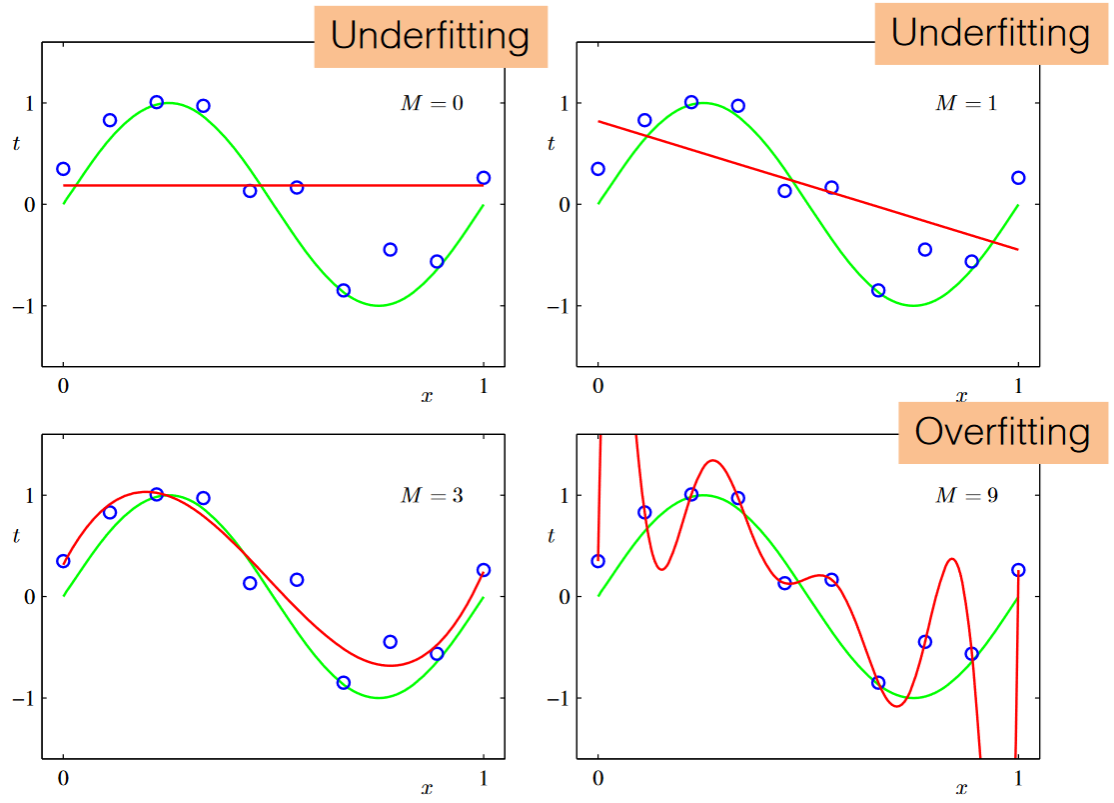
On choisit de prédire avec un polynôme d'ordre M :

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

Et une fonction de coût « moindre carrés » (moyenné):

$$L = \frac{1}{N}\sum_{i=1}^{N}(t_i - \hat{t}_i)^2, \text{ avec } N \text{ le nombre de données.}$$

Quel ordre M choisir pour notre polynôme ?



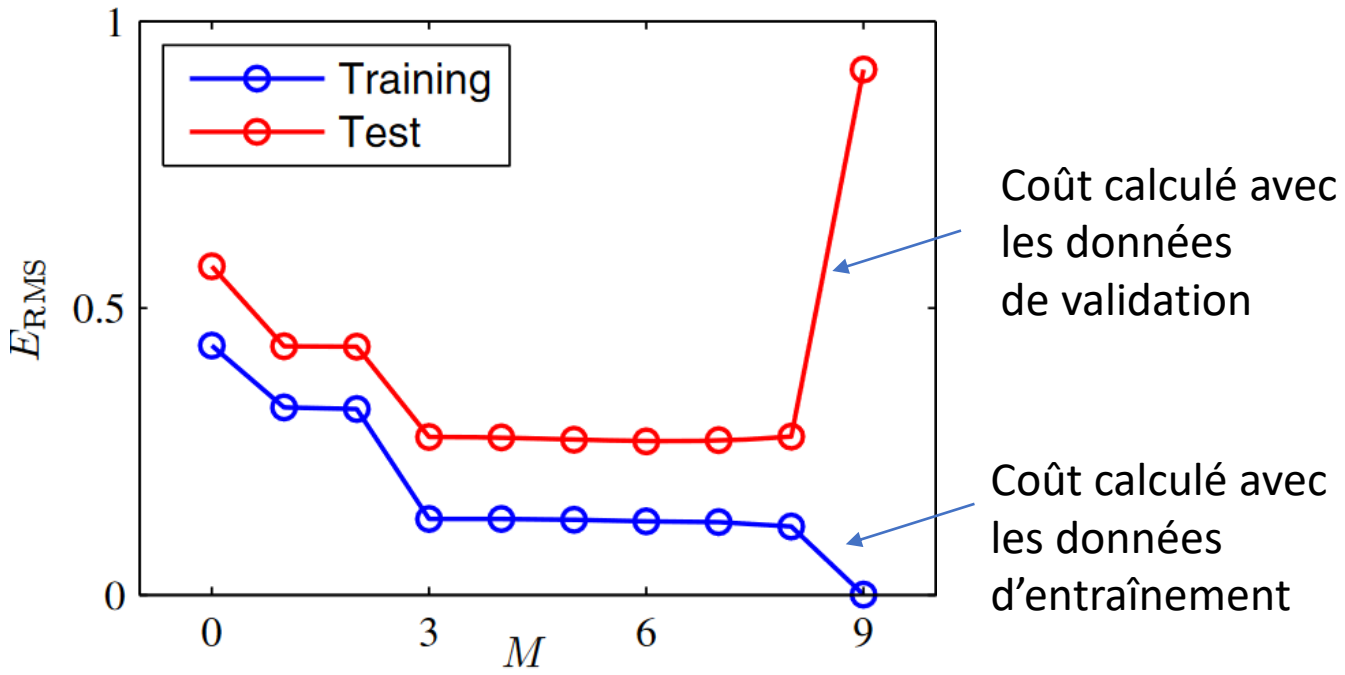
[C. Bishop, Pattern recognition and Machine learning, 2006]

- 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 :



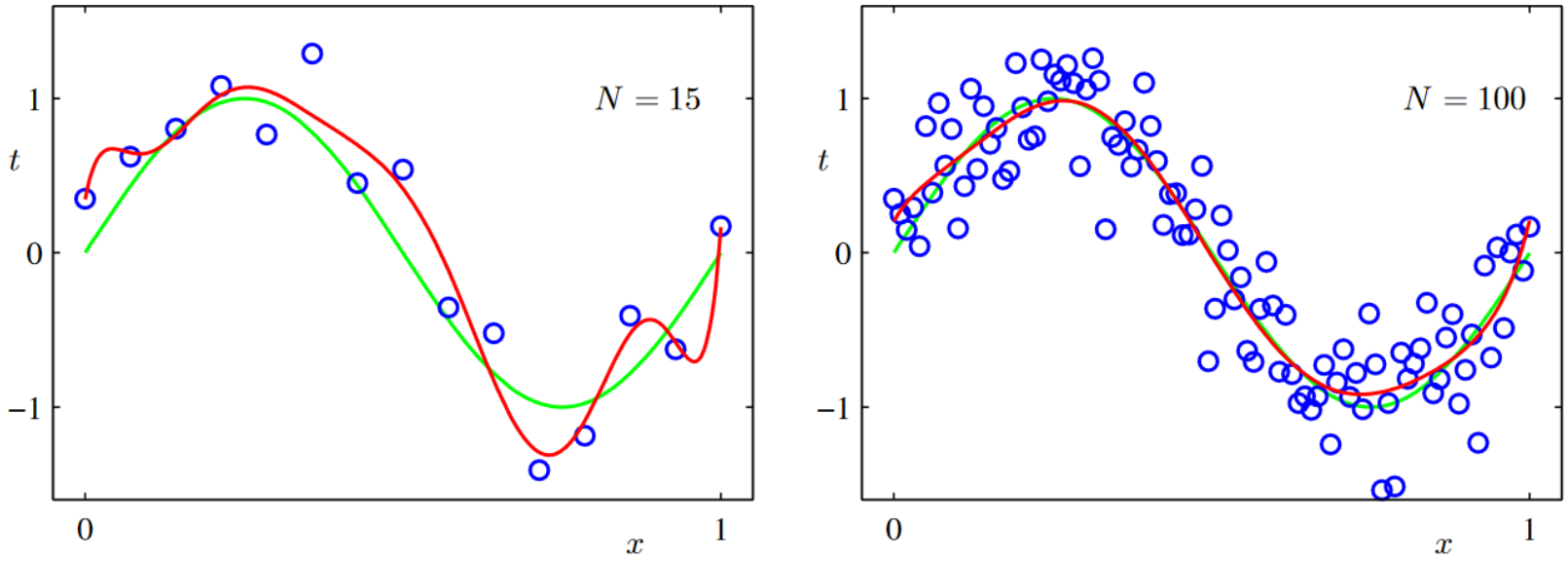
Coût calculé avec les données de validation

Coût calculé avec les données d'entraînement

[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]

Trois 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 3.a (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*.*

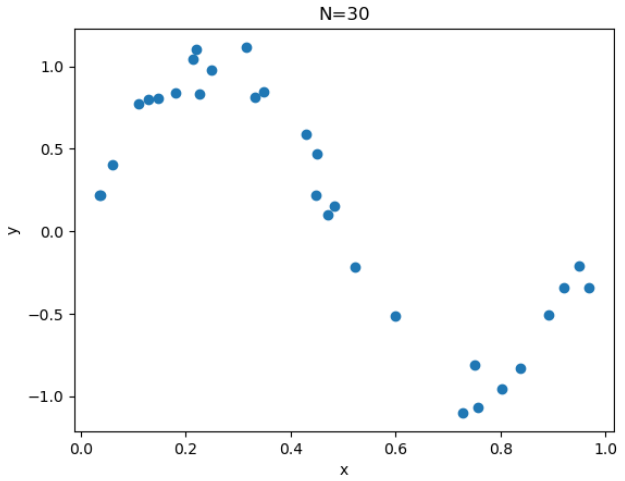
   Utiliser la fonction *np.random.uniform(…)* pour utiliser une distribution uniforme.

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.



3.2 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.

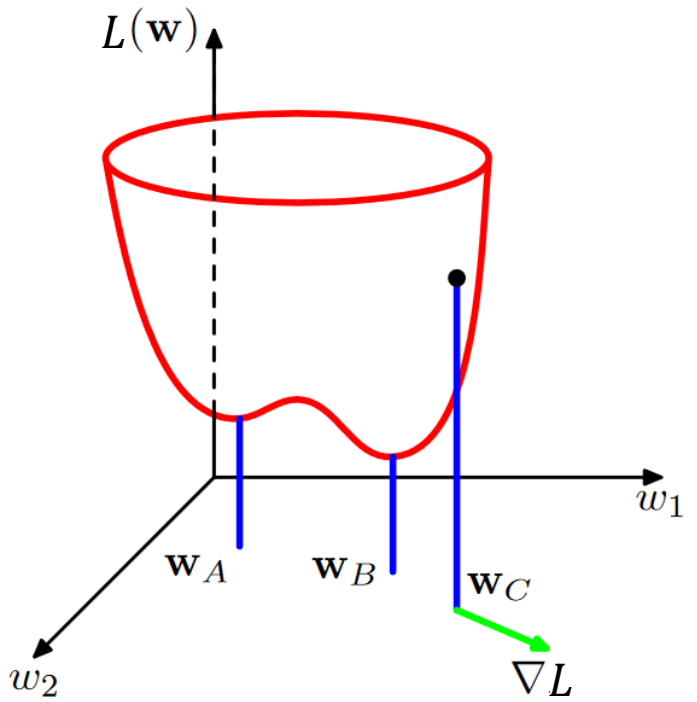
Mises à jour :

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

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) ⇒ calcul « à la main » prohibitif.

Calcul par différences finies :

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

⇒ **approximatif** et **coûteux** (nombreuses évaluations de L).

Calcul par dérivation automatique :

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

⇒ plus précis et plus rapide.

Exemple (librairie 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]
```

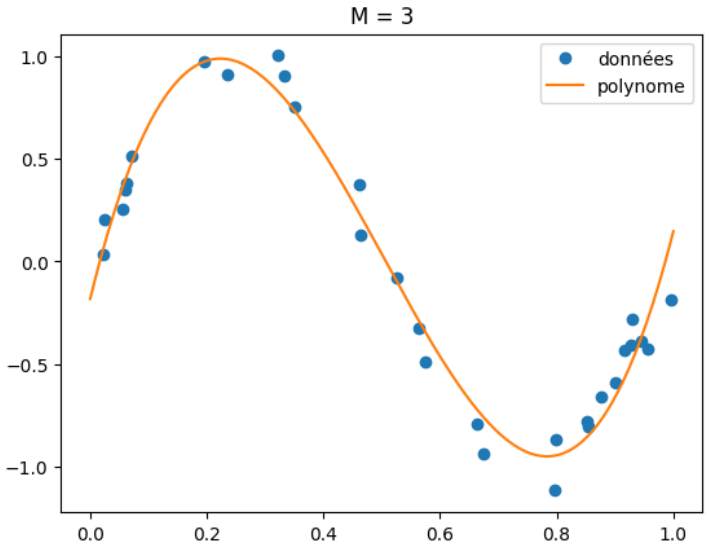
Exercice 3.a.suite (PyTorch):

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)

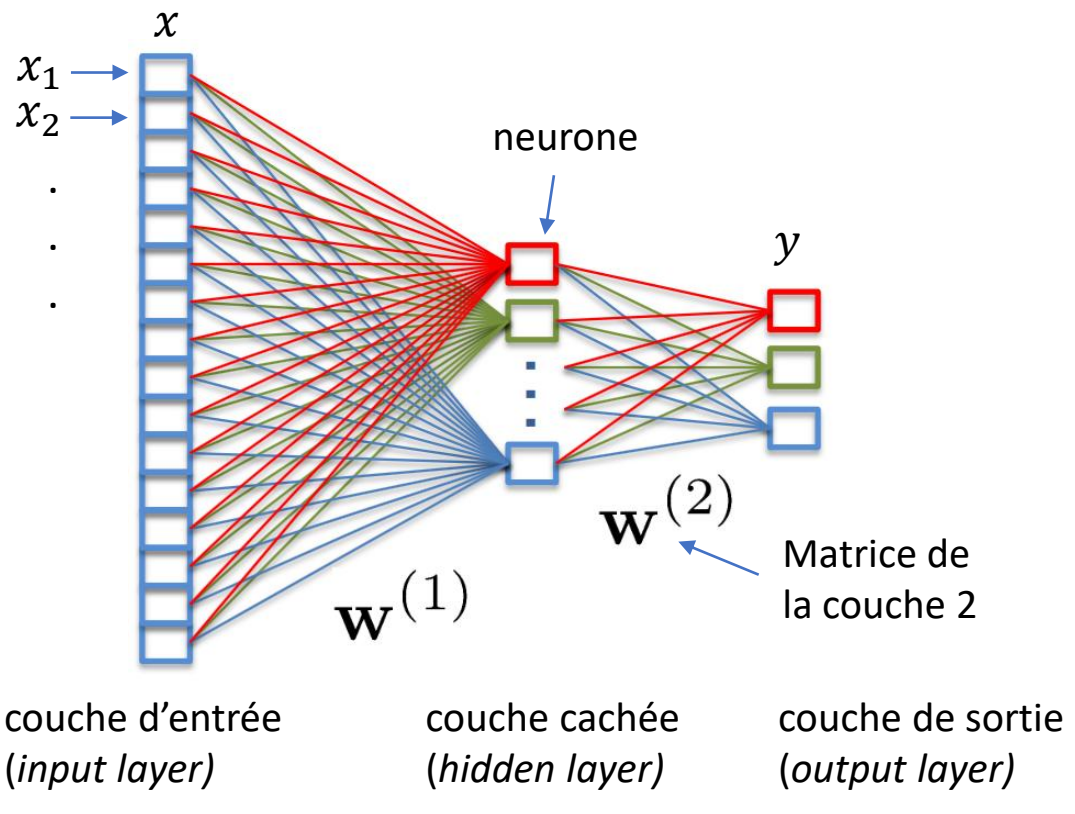


Pour aller plus loin : générer un 2ème ensemble de données (*test set*) et tracer le coût en fonction de M.

3.3 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 :

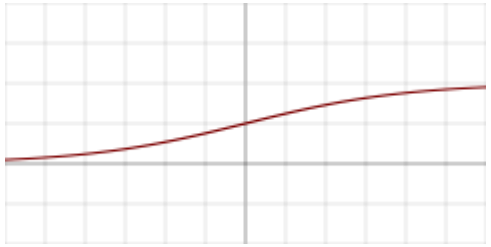
- Sortie d'une couche i: $s^{(i)} = \sigma(W^{(i)} \times e^{(i)} + b^{(i)})$
- Sortie du réseau : $y = W^{(2)} \times \sigma\left(W^{(1)} \times x + b^{(1)}\right) + b^{(2)}$
- Sortie d'un neurone : $s_j^{(i)} = w_j^{(i)} * e^{(i)} + b_j^{(i)}$

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

[illustration: https://chriswolfvision.github.io/www/teaching/

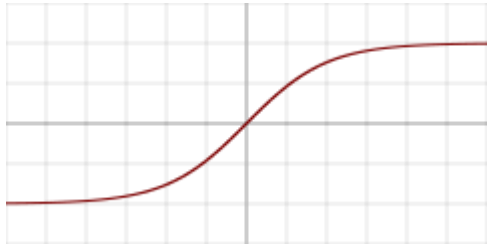
Non-linéarités σ (fonction d'activation):

- Fonction scalaire, i.e. $\sigma: \mathbb{R} \to \mathbb{R}$
- 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 :



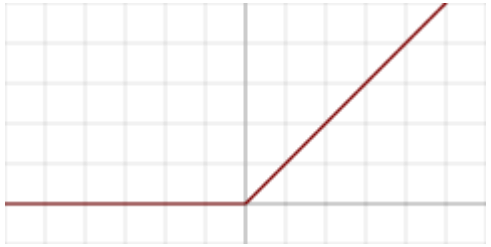
Sigmoïde

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



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**).

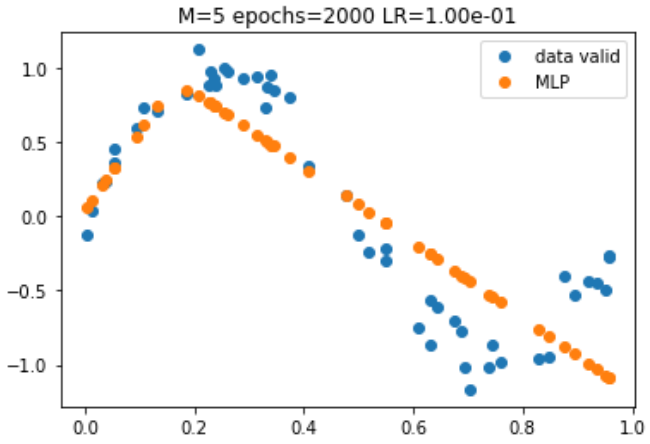
Avec SGD, à chaque itération, les données *train* sont découpées en sous-ensemble (***mini-batch***) et un calcul de gradient et une mise à jour des paramètres sont faits sur chaque *mini-batch*. Le découpage est fait aléatoirement à chaque itération (i.e. stochastique). Les avantages sont:

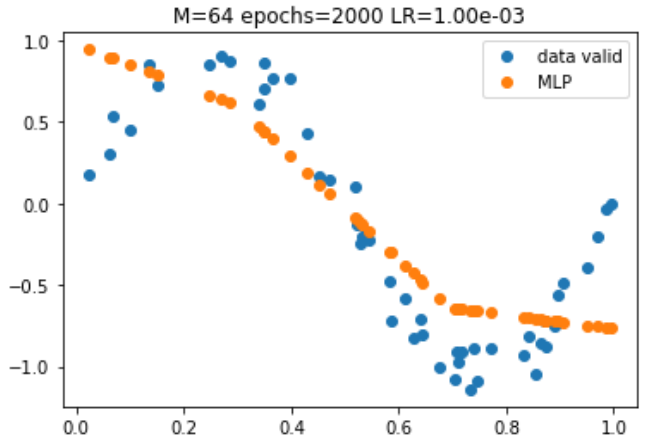
- Apprendre plus vite sur les grandes quantités de données (mises à jour plus fréquentes).
- Moins de risque d'être bloqué dans un minimum local.

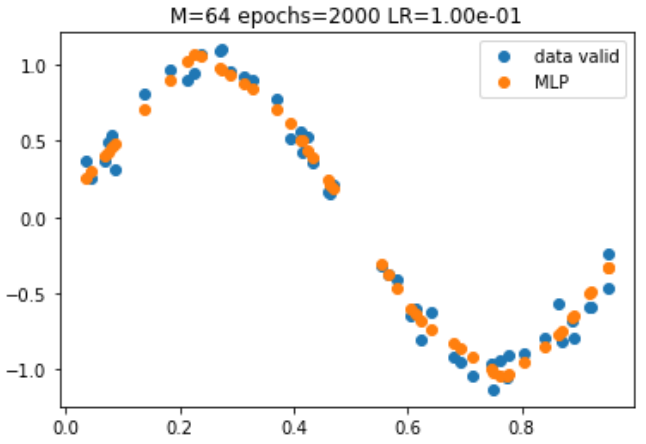
Résultats (problème de régression simple):

Soit N le nombre de données d'entraînement. Nous fixerons $N = 200$ (ce sont des données du problème).

Soit M le nombre de colonnes de $W^{(1)}$, *epochs* le nombre d'itérations, LR le *learning rate*.
$M, epochs, LR$ sont des réglages (**hyperparamètres**) de l'entraînement.





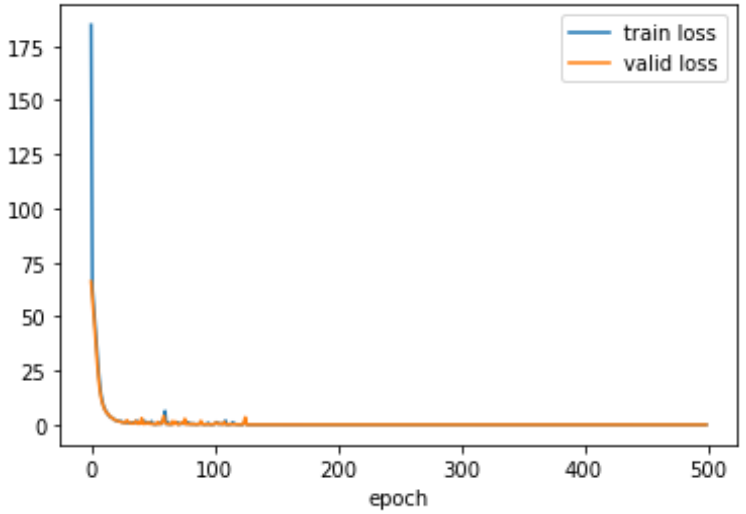


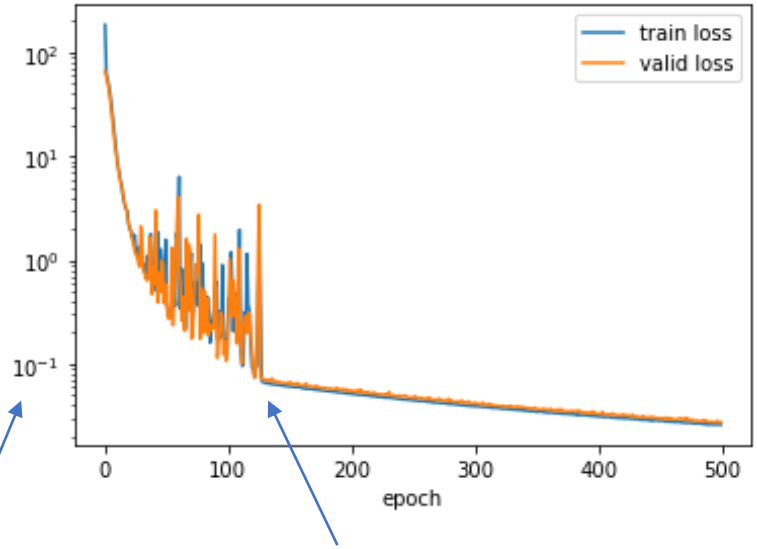
Exercice 3.b : Résoudre le problème de régression avec les données (vectorielle) générées ainsi:

- $x \in \mathbb{R}^2, x_1 \in [-1; 1], x_2 \in [1; 4]$ (aléatoire uniforme)
- $y = \left[\sqrt{x_1 + x_2}; x_2^3\right] + N(0, .1)$ (bruit gaussien)

Résultats pour

- Nombre de données : 2000 (entraînement) et 500 (validation)
- Nombre d'itérations (*epochs*): 500
- Learning rate : $1e^{-3}$ puis $2e^{-4}$





Echelle semilog

Réduction du *learning rate*

3.4 Problèmes de classification

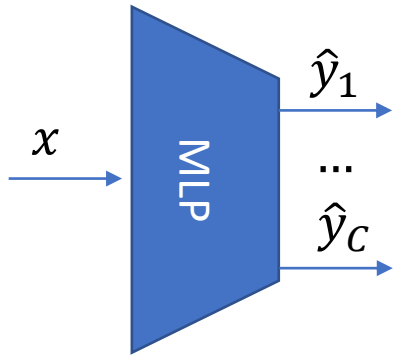
Les données de sortie (qu'on cherche à prédire) sont des valeurs discrètes,

i.e. un échantillon y appartient à un ensemble finis de classes $\{1, \dots, C\}$.

Ex: à partir d'une image d'un animal, prédire s'il s'agit d'un chat, d'un chien, etc.

<u>Sorties du modèle</u> :

La sortie du modèle est un vecteur $\hat{y} \in \mathbb{R}^C$.



A chaque $\hat{y}_i$ on associe la probabilité $\hat{P}(i|x) = \frac{\exp(\hat{y}_i)}{\sum_{c=1}^{C} \exp(\hat{y}_c)}$

La prédiction du modèle est la classe avec la plus grande probabilité.

Remarque : on vérifie qu'il s'agit d'une loi de proba : $\hat{P}(i|x) > 0$ et $\sum_{i=1}^{C} \hat{P}(i|x) = 1$.

<u>Fonction coût (***cross entropy***)</u> : Soit $y = c$ pour l'entrée x (i.e. la *vraie* classe est c), alors

$$L = -\log \hat{P}(c|x)$$

Analyse des données :

- Type d'entrées : numérique, catégorie (binaire, multiple).
- Statistique simple : moyennes, écart-types, min, max.
- Corrélations :

  Coefficient de Pearson (r ou R) : mesure de la force et de la direction de la relation entre deux variables.

$$R(x,y) = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2}\sqrt{\sum(y-\bar{y})^2}}$$

  Propriétés :

  1. $-1 \leq R \leq 1$ (avec, e.g., relation linéaire positive parfaite pour $r = 1$, aucune relation pour $R = 0$).
  2. $R(x,y) = R(y,x)$
  3. $R(x,x) = 1$

  Matrice de corrélation : pour des variables {x, y, z, …}
  matrice carrée représentant les coeffs de Pearson.
  Remarques :
  propriété 2 ⇒ la matrice est symétrique.
  propriété 3 ⇒ diagonale remplies de 1.

	x	y	z
x	R(x,x)	R(x,y)	R(x,z)
y	R(y,x)	…	
z	…		

Pré-traitement des données :

Certaines données brutes sont mal adaptées aux NN.

Pour faciliter leur entrainement, i.e. rendre la descente de gradient plus simple, il faut :

- Eviter des valeurs numériques extrêmes (problèmes de conditionnement, explosion ou anulation du gradient).

  Ex : représentation d'une pression en *hPa* plutôt qu'en Pascal (pression atmosphérique > $1e^5$ Pa)

- Favoriser une représentation informative des entrées.

  Ex : la représentation d'une image sous forme de 3 matrices RGB (*red, green, blue*) est plus informative que sous la forme d'un vecteur (notions de voisinage spatial, couleur).

Normalisation (valeurs numériques) : Afin de s'assurer que les données respectent des valeurs centrées autour de zéro, et un écart-type de 1, on peut normaliser les données ainsi:

$$x^{norm} := \frac{x - \bar{x}}{\sigma_x}$$ où $\bar{x}$ représente la moyenne et σ_x l'écart-type.

One-hot encoding (classes) : plutôt que de représenter la classe par un seul nombre entier, on représente par un vecteur de longueur le nombre de classe; le vecteur est alors formés de zéros et d'un seul 1.

Ex: soit une donnée appartenant à $\{1,2,3,4,5\}$. L'appartenance à la classe 1 est codée [1,0,0,0,0], à la 3 [0,0,1,0,0].

Interprétation des résultats / utilisation du modèle:

Une fois entraîné, le modèle nous intéresse pour :

- La **prédiction d'une classe** = index de la sortie avec la plus grande valeur, i.e.

$$classe\ prédite = arg \max_{i \in \{1,...C\}} y_i$$

- La **confiance** du modèle en sa prédiction = $P(y = i|x) = \frac{\exp(y_i)}{\sum_{c=1}^{C} \exp(y_c)}$

Les fonctions de coût utilisé en classification (e.g. *cross entropy*) sont des **métriques indirectes** de la qualité du modèle.

La métrique (quantité qui évalue notre modèle) qui nous intéresse vraiment est le **pourcentage d'erreur dans la prédiction des classes**. C'est une métrique dont l'évolution est à regarder pendant l'entraînement et pour interpréter la qualité du modèle. Mais cette métrique ne peut pas être utilisé pour l'entrainement par descente de gradient car pas dérivable.

Exemple: Titanic

Données : pour 891 passagers, prix de leur billet, genre (H/F), tarif de leur billet, classe de leur cabine, un identifiant et si ils ont survécu.

Problème : entraîner un modèle qui prédise si un passager a survécu.

Type de données :

Affichons les premières lignes du fichier de données (*titanic.csv).*

Chaque ligne correspond à un passager.

- Sortie :
  - *« Survived »* (survécu vs pas survécu) = classe binaire
- Entrées :
  - *« p_id »* (identifiant) = classe multiples,
  - *« cabin_class »* = classe multiples,
  - « female » (femme vs homme) = classe binaire,
  - « fare » (tarif du billet) = numérique.

```
# p_id, survived, cabin_class, female, fare
0,0,3,0,7.250
1,1,1,1,71.283
2,1,3,1,7.925
3,1,1,1,53.100
4,0,3,0,8.050
5,0,3,0,8.458
6,0,1,0,51.862
7,0,3,0,21.075
8,1,3,1,11.133
```

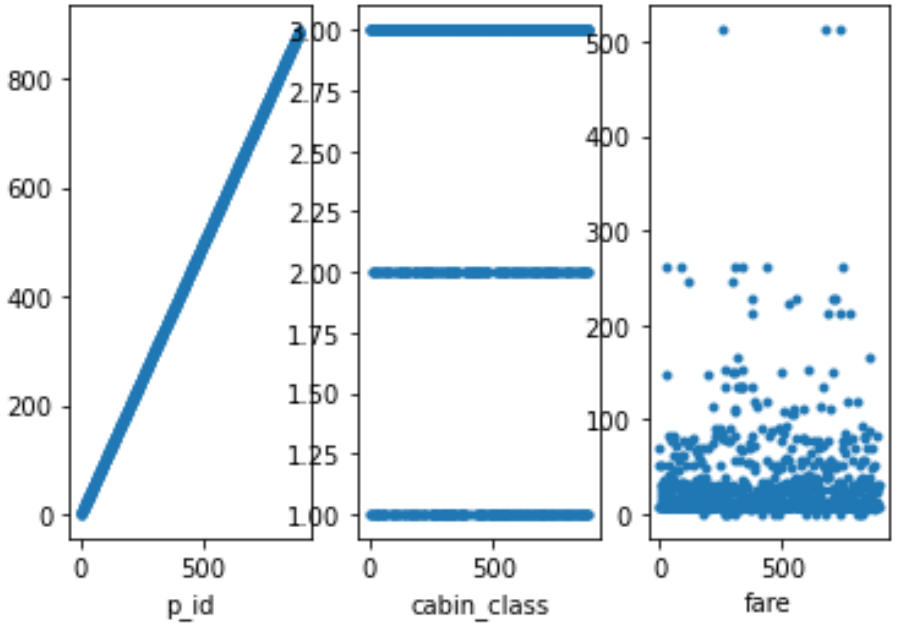
Traçons les valeurs des champs « p_id » et « cabin_class ».

p_id attribue une valeur unique à chaque passager.

Cabin_class est une classe multiple; ses éléments sont {1,2,3}:

```
print(np.unique(cabin_class))
```

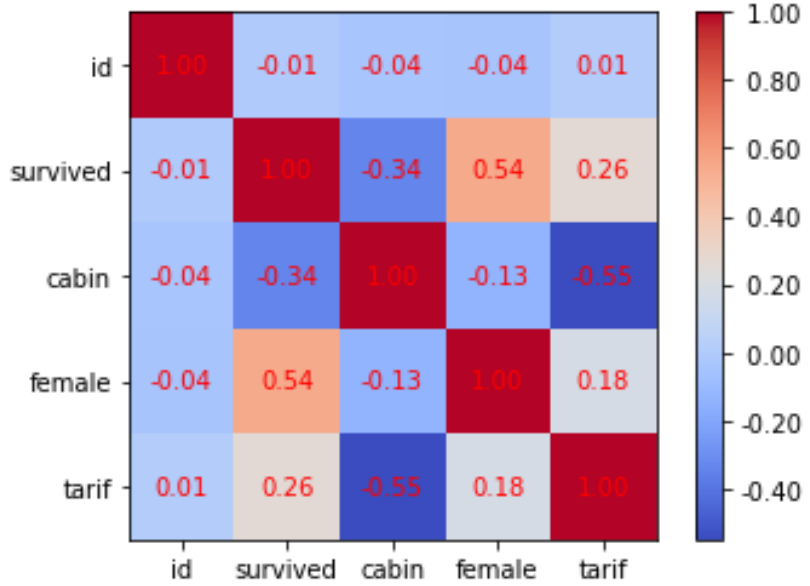
Combien de passagers ont survécu ? `print(sum(survived))`



Statistiques simples :

```
print("fare: min %.3f  max % .3f écart-type  %.3f" %
    (fare.min(), fare.max(), fare.std()))
```

>> fare: min 0.000 max 512.329 écart-type 49.666



Corrélation :

Forte corrélation négative entre « cabin_class » et « survived », i.e. les passagers de 1ère classe ont plus de chance de survie que ceux en 3ème classe.

Pré-traitement des données:

Normalisation de « fare » (valeur numérique).

Encodage *one-hot* de « p_id », « cabin_class » et « female » (classes).

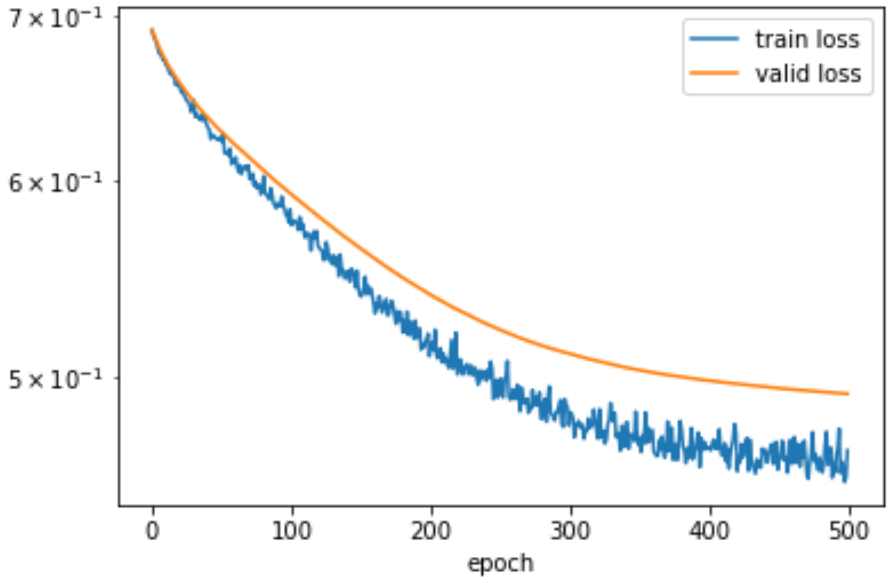
Il est préférable d'**éliminer l'entrée non pertinente** « p_id » : n'a pas de lien avec ce qu'on cherche à prédire (ce que confirme l'analyse de corrélation). Au contraire cette entrée peut :

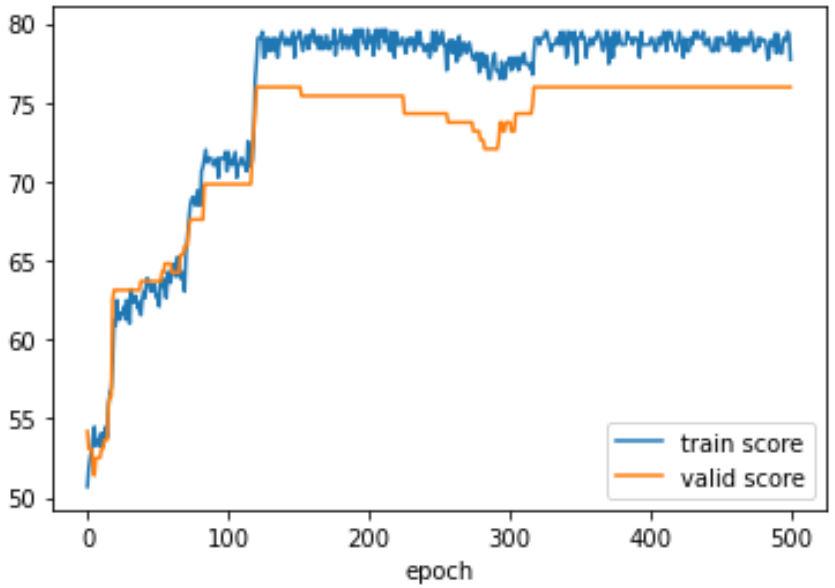
- Complexifier le modèle (plus d'entrées impliquent plus de paramètres).
- Favoriser le sur-apprentissage (en apprenant une relation causale qui n'existe pas).

Remarque : puisque les passagers sont identifiés par un nombre unique, « p_id » il est possible d'apprendre à prédire de façon parfaite, sur les données d'entraînement, si un passager à survécu seulement avec cette entrée. Mais sur de nouvelles données, la prédiction sera totalement aléatoire (cas extrême de surapprentissage).

Résultats :

Pour un NN avec 1 couche caché de taille 32, learning rate à $1e^{-3}$, des mini-batch de taille 32 :



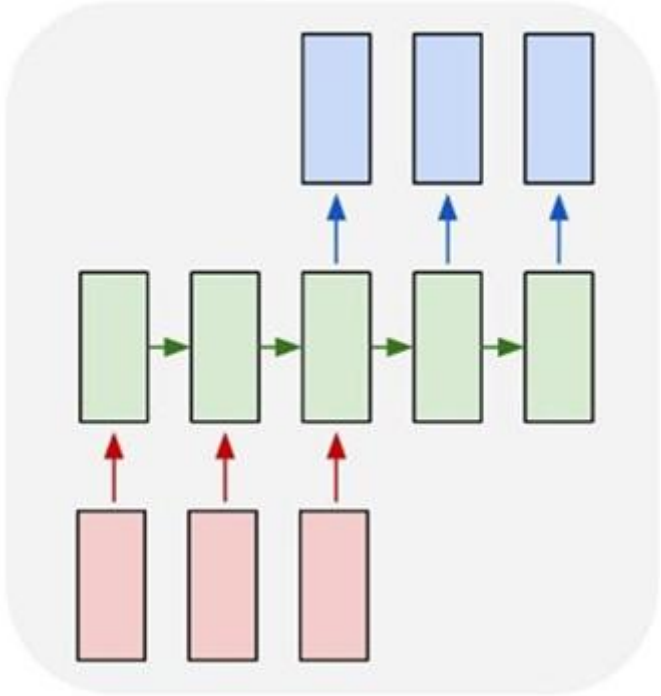


La courbe de droite correspond au % de réussite dans la prédiction de la classe de sortie.

Remarque : dans cette exemple (Titanic) le nombre de classe en sortie est de 2 (survécu / pas survécu). Il est donc logique qu'un modèle non entraîné ait un score proche de 50% (réponse « au hasard » en début d'entrainement).

4. Les réseaux de neurones récurrents (RNNs)

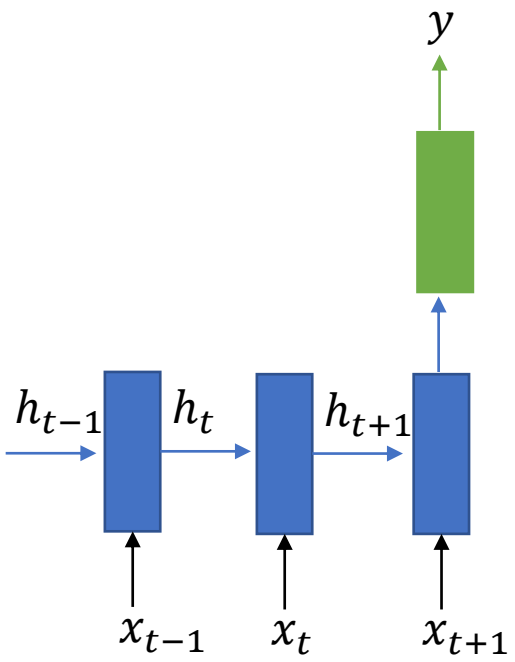
4.1. Données séquentielles
4.2. Principaux RNNs
4.3. Implémentation



4.1 Données séquentielles

Données simples

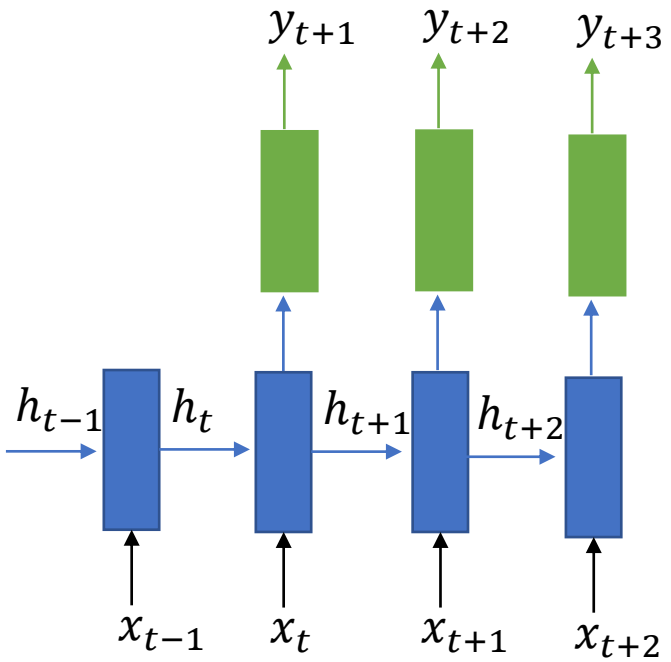
$y = f(x)$



Entrées séquentielles

$y = f_2(h_{t+1}, x_{t+1})$

$h_{t+1} = f_1(h_t, x_t)$



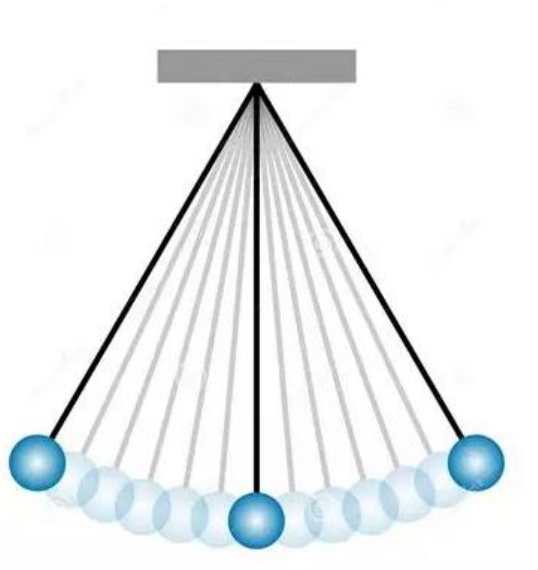
Entrées et sorties séquentielles

$y_{t+1} = f_2(h_t, x_t)$

$h_{t+1} = f_1(h_t, x_t)$

Récurrence

Exemple (pendule) :



Problème 1 :

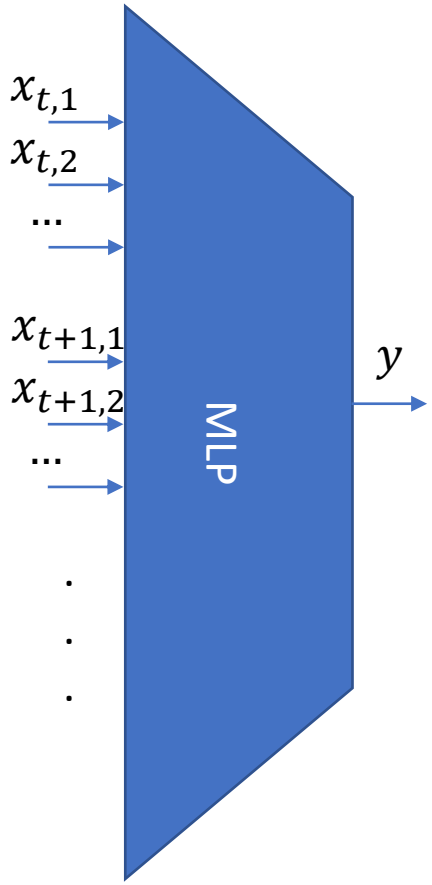
On enregistre la position angulaire θ au cours du temps (données d'entrées) pour différentes longueurs l et masse m du pendule (données de sortie) ⇒ apprendre un modèle pour estimer les valeurs de l et m.

Problème 2 :

On enregistre la position angulaire θ au cours du temps (données d'entrées x) et on cherche à prédire les futures valeurs de θ.

Approche simple :

- Représentation de la **séquence en simple vecteur** (*flat*).
- Utilisation d'un **MLP**.



Mal adapté :

- La structure du MLP est mal adaptée à la **structure des données** (e.g. la similitude entre un vecteur x_t et x_{t+1} n'est pas exploitée).
- Mal adapté à des séquences de longueurs variables.

⇒ expressivité limitée

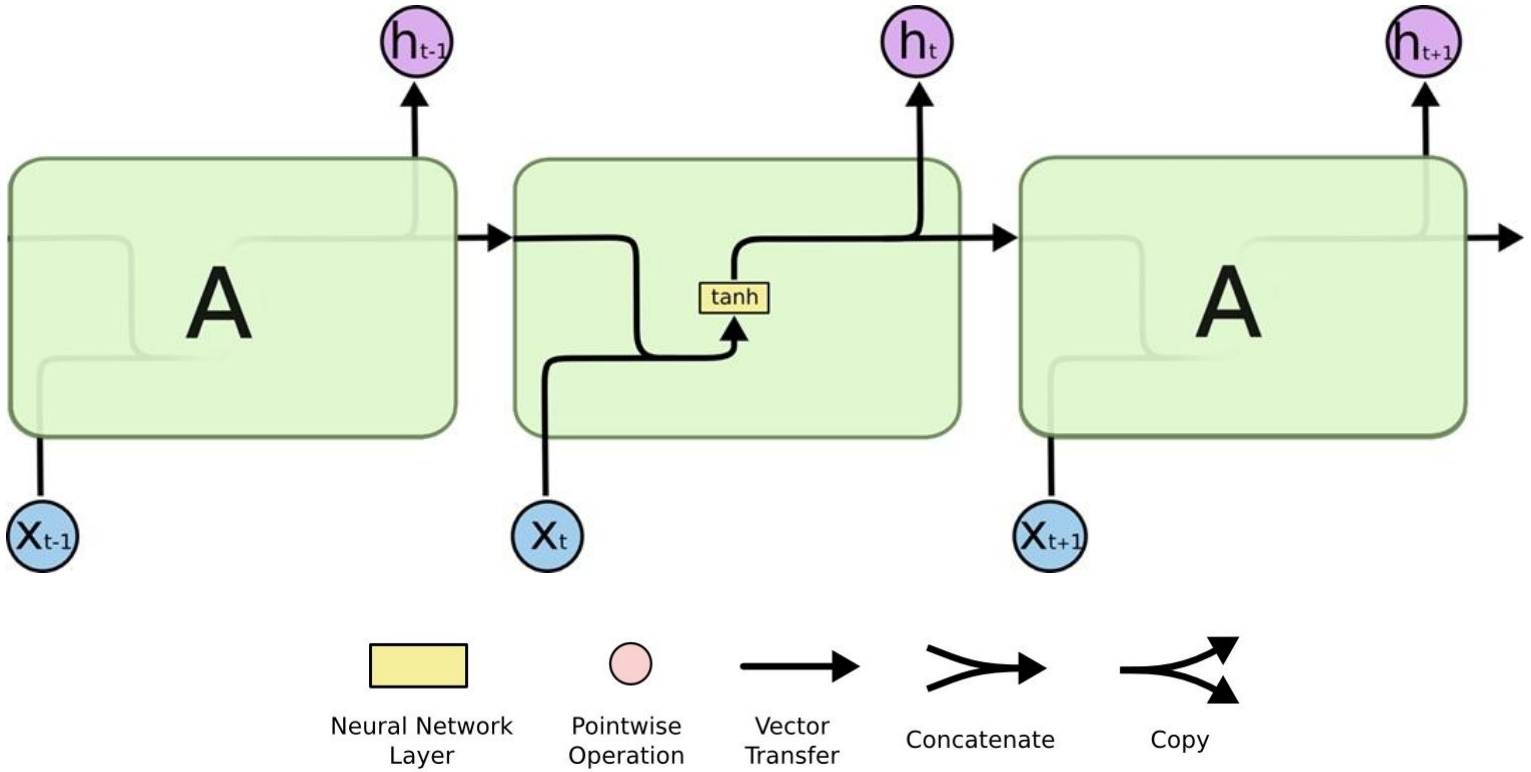
⇒ prédictions peu précises et/ou nécessite de nombreux paramètres (beaucoup de neurones)

⇒ risque de surapprentissage.

4.2 Principaux RNNs

Réseau Elman (= *vanilla*):

$$h_t = tanh(W_x x_t + W_h h_{t-1} + b_h)$$



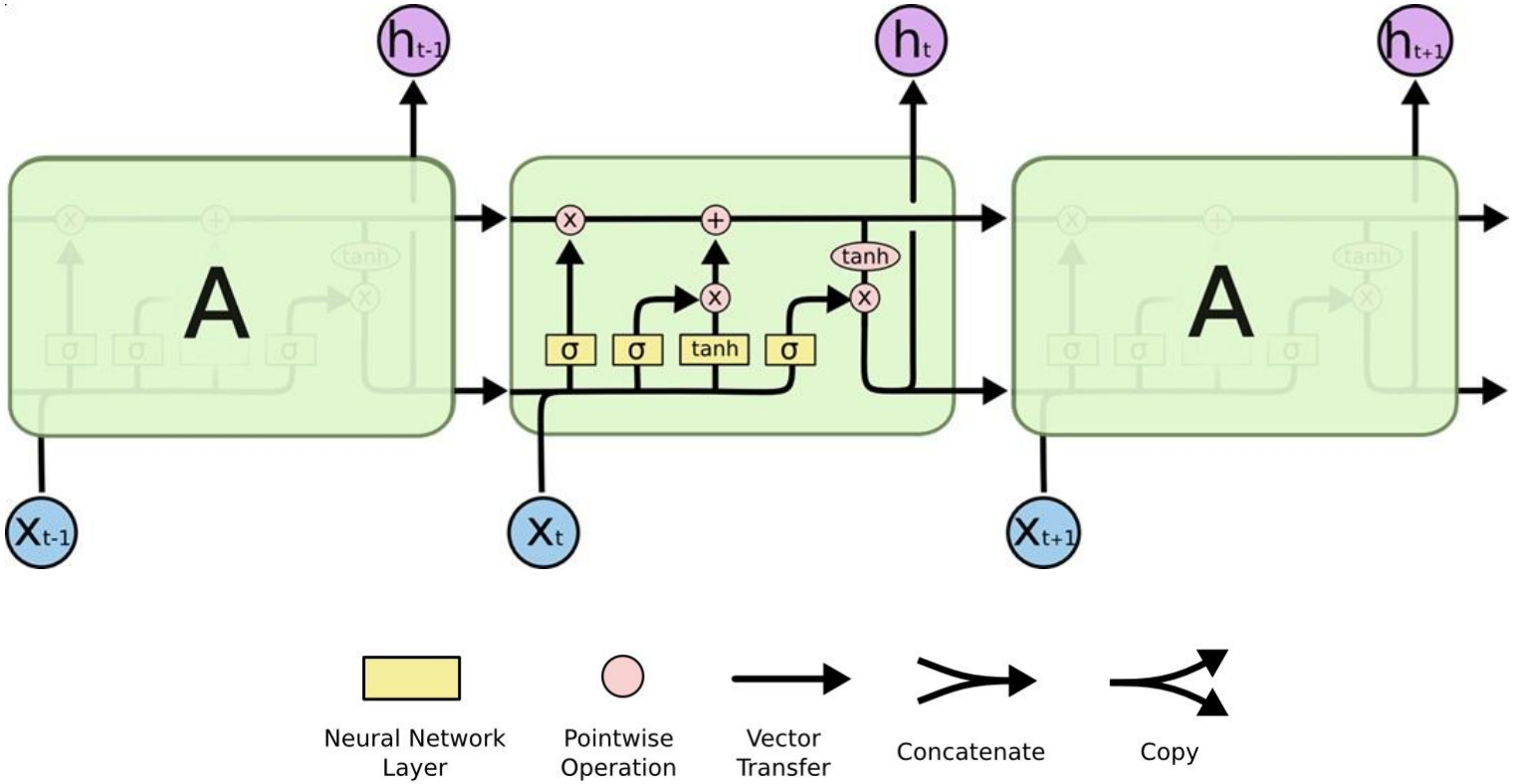
y_t est une fonction de h_t, par exemple $y_t = W h_t + b_y$ (relation linéaire).

Amélioration par rapport au MLP : ajout d'une mémoire ⇒ partage des paramètres entre les pas de temps t ⇒ plus facile à entraîner (nécessite moins de paramètres).

Limitations : temps-invariant, difficile à entrainer sur de longues séquences (perte de sensibilité = *vanishing gradient*) .

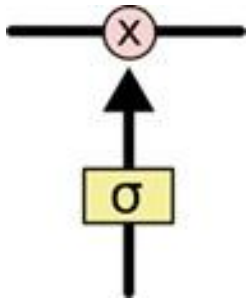
[illustration: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM :



Mécanisme de porte (***gate***) pour décider si on laisse passer une information :

σ représente la fonction sigmoide, avec en sortie un signal entre 0 (ne laisse rien passer) et 1 (laisse tout passer).

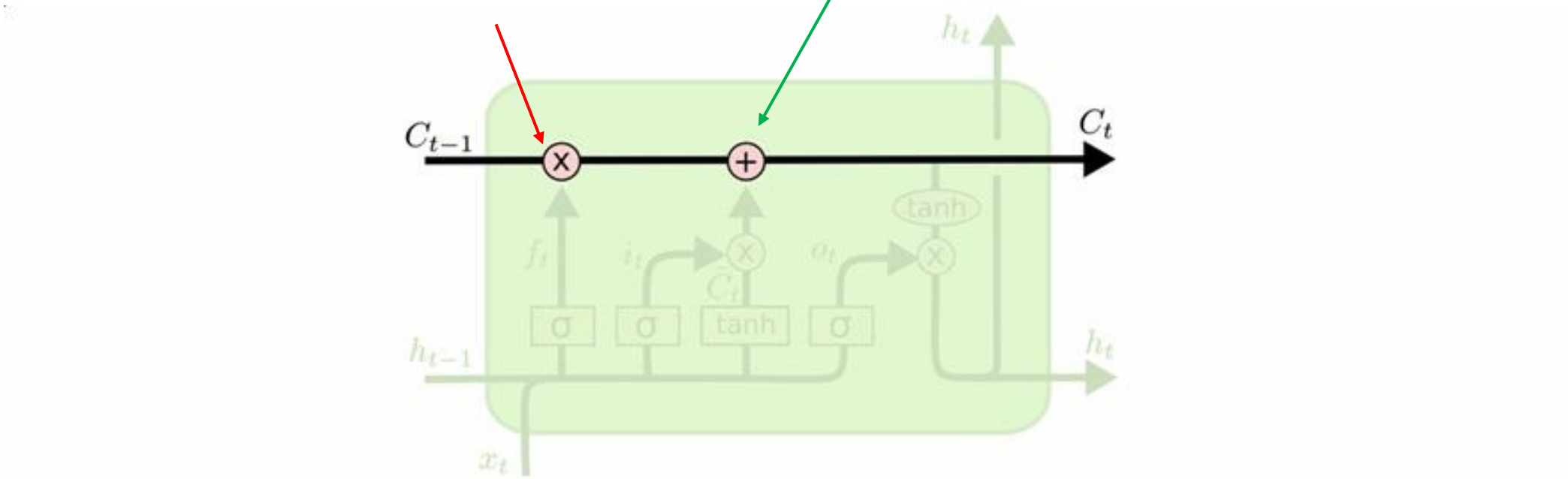


[illustration: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM :

Oublier (ou pas) une partie de l'ancienne information C_{t-1}

Ajouter de nouvelles informations à C_t



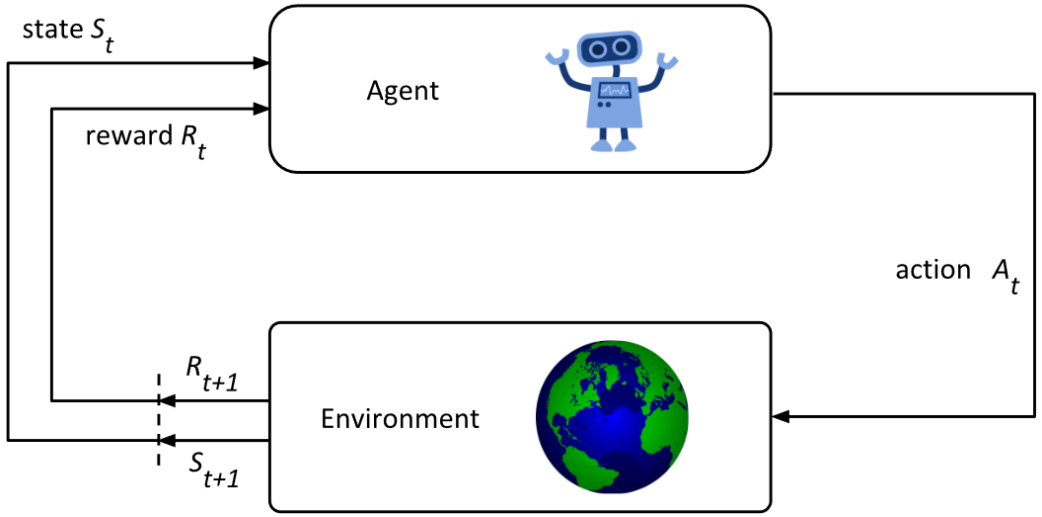
Expérimentalement, les LSTM ont montré leur efficacité pour modéliser des systèmes complexes.

Système d'équations difficiles à analyser ⇒ restent encore mal compris.

[illustration: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

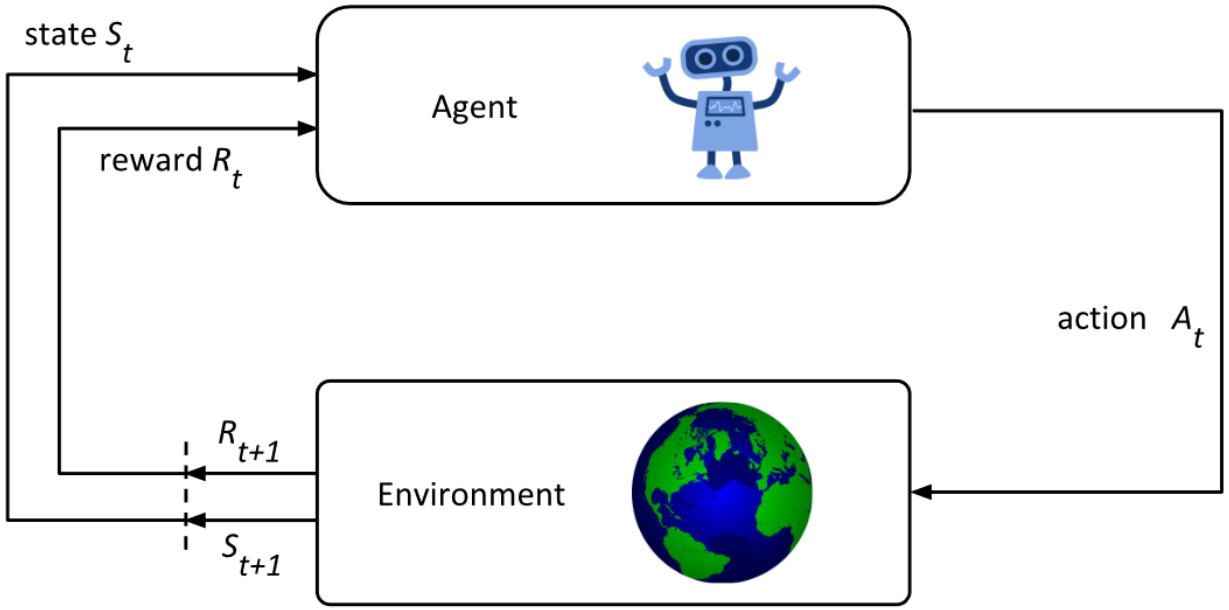
5. Apprentissage par renforcement (RL)

5.1. Principe et définitions
5.2. Equations de Bellman
5.3. Algorithme tabulaire (Q-learning)
5.4. Approximation des fonctions de valeurs (NNs)



5.1 Principe et Définitions

Reinforcement learning (RL) :



Un agent autonome **:** apprend à se comporter à partir d'expériences (**sans modèle** de l'environnement).

Interactions agent-environnement:

- **Etat** (*state*) $s \in \mathbb{R}^n$
- **Action** $\mathrm{a} \in \mathbb{R}^d$
- **Récompense** (*reward*) $\mathrm{r} \in \mathbb{R}$, fonction de l'état et de l'action.

Remarque : on supposera s_t mesuré (en RL on parle d'*observations complètes*).

[illustration: https://www.geeksforgeeks.org/what-is-reinforcement-learning/

Politique (π) : règle de décision $a = \pi(s)$.

Objectif : apprendre une politique optimale π^* (maximiser les récompenses).

Environnement déterministe:

- la transition de s_{t+1} en fonction de s_t et a_t est déterministe.
- Objectif : optimiser la somme pondérée des récompenses G

$$G = r_1 + \gamma\, r_2 + \gamma^2 r_3 + \ \ldots = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

avec $\gamma \in [0; 1[$ qui favorise les récompenses rapides (*discount factor*).

Environnement stochastique:

- la transition de s_{t+1} en fonction de s_t et a_t est une densité de probabilité.
- Objectif : optimiser l'espérance de G

Remarque : Dans la suite de ce cours, nous supposons que l'environnement est déterministe.

Fonction de valeur des états, V :

- V est la valeur de la somme (pondérée) des futures récompenses.
- Utilisée pour évaluer la valeur d'un état.
- Dépend de la politique utilisée

$$V_\pi(s_t) = r_{t+1} + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \ \ldots$$

- **Si on connaît la fonction optimale V_{π^*} et le modèle ⇒ l'action optimale est** :

$$a^*(s_t) = \arg\max_a V_{\pi^*}(s')$$

avec s' l'état qui succède à s quand l'action a est prise.

Fonction de valeur des couple (états-actions), Q :

- $Q_\pi(s, a)$ est la valeur des futures récompenses, sachant qu'on choisit l'action a, puis qu'on utilise la politique π.
- **Si on connaît la fonction optimale Q_{π^*} ⇒ l'action optimale est** :

$$a^*(s_t) = \arg\max_a Q_{\pi^*}(s, a)$$

⇒ 1. Apprendre $V_{\pi^*}(s)$ nécessite modèle pour prise de décision (***model-based***).

Remarque : le modèle peut-être appris à partir d'expériences.

⇒ 2. Apprendre Q_{π^*}(s,a) = Q-learning (***model-free***).

- <u>Relations entre V et Q</u>:

$$Q_\pi(s,a) = r + \gamma V_\pi(s') \quad ,\forall\pi$$

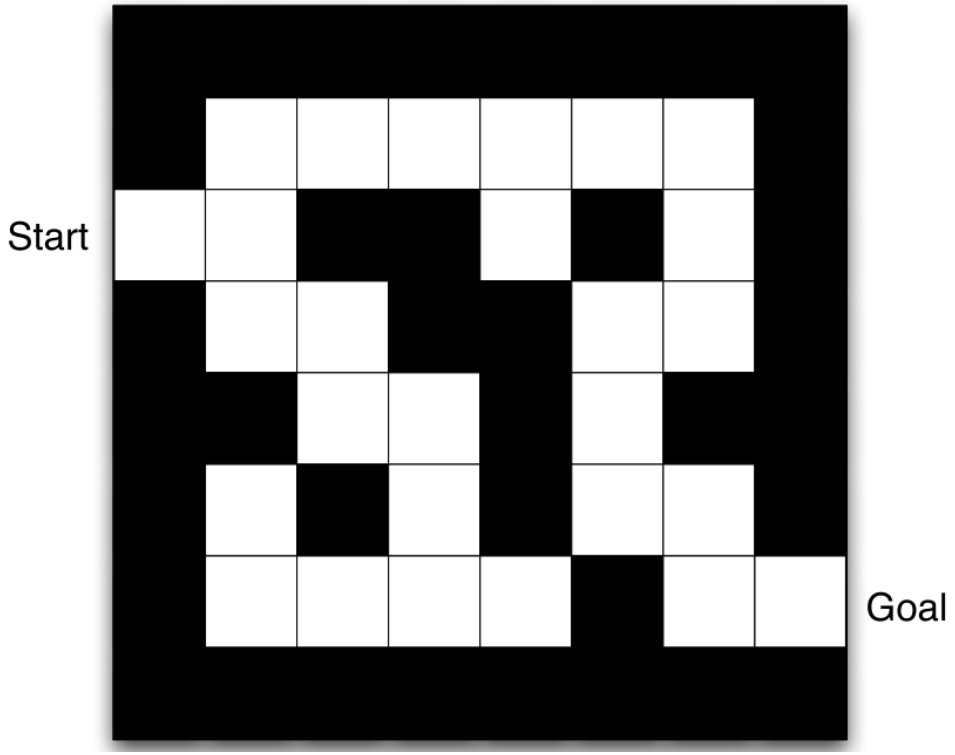
$$V_\pi(s) = Q(s,\pi(s)) \,,\forall\pi$$

- <u>Relation entre V^* et Q^*</u> :

$$V_{\pi^*}(s) = \max_a Q_{\pi^*}(s,a)$$

Exemple (labyrinthe) :

- Récompenses : -1 à chaque pas de temps
- Actions : Gauche, Droite, Bas, Haut
- Etats : position de l'agent



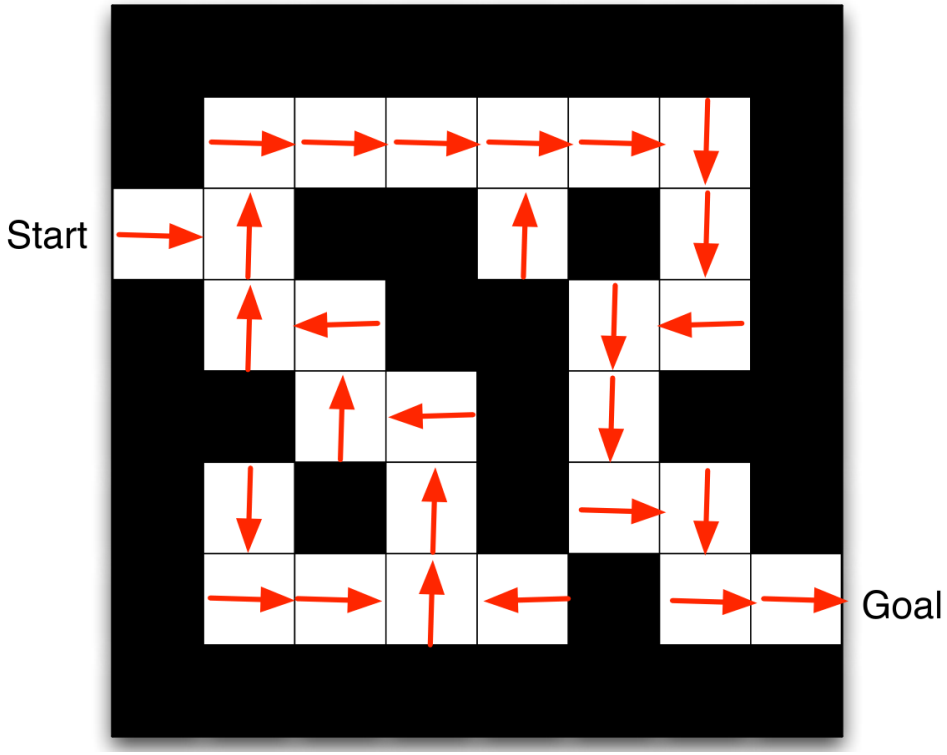
- Remarque : ici (**environnement déterministe**) l'agent est sûr de se déplacer dans la direction qui correspond à sa demande. En stochastique, par exemple, on considère une probabilité de « glisser » dans une mauvaise direction.

[illustration: https://www.davidsilver.uk/teaching/]

Exemple (labyrinthe) :

Politique optimale π^* ($\forall\ \gamma > 0$):

- sortir du labyrinthe le plus vite possible
- e.g. $\pi^*(s = Start) = Droite$

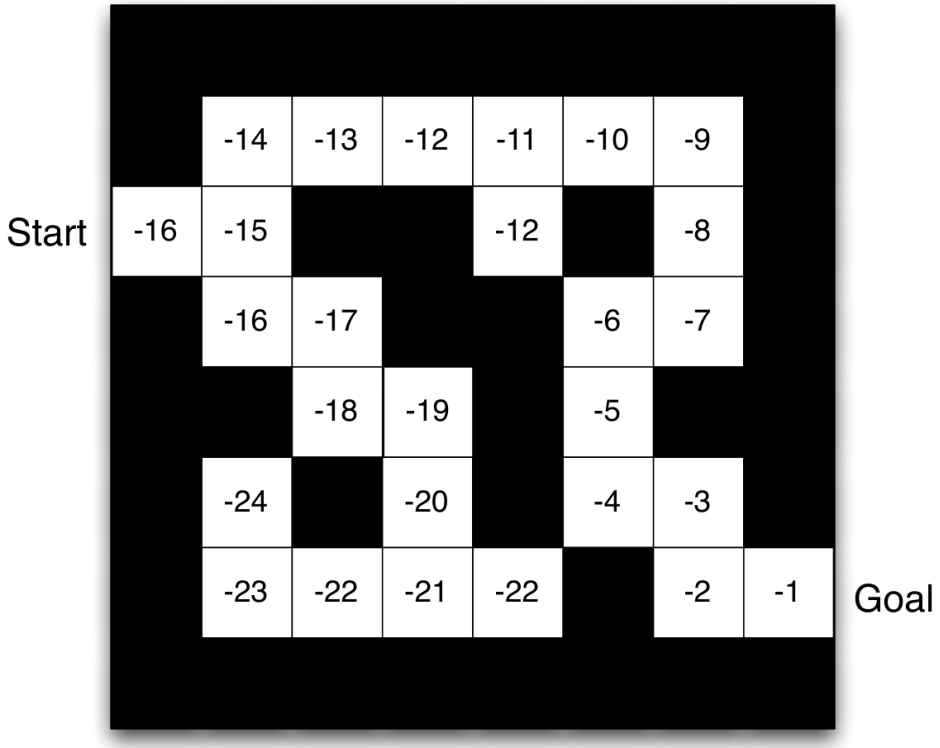


Remarque : pour $\gamma = 0$ n'importe quelle politique est optimale, car
$V^*(s) = r(s,a) + 0 \times V(s') = r(s,a) = -1, \forall a$

[illustration: https://www.davidsilver.uk/teaching/]

Exemple (labyrinthe) :

Valeur des états V_{π^*} :



Questions : On représente les lignes par des lettres et les colonnes par des chiffres, e.g. $V(c3) = -16$ et $V(c4) = -15$. **Que vaut $\boldsymbol{Q(c3, Droite)}$? $\boldsymbol{Q(c4, Bas)}$?**

[illustration: https://www.davidsilver.uk/teaching/]

5.2 Equations de Bellman

La fonction de valeur V peut être décomposée en 2 termes:

- Une **récompense immédiate** r_{t+1}
- Une **valeur pour l'état suivant** $\gamma\ V(s_{t+1})$

$$\begin{aligned} V_\pi(s_t) &= r_{t+1} + \gamma\ r_{t+2} + \gamma^2 r_{t+3} + \ \dots \\ &= r_{t+1} + \gamma\ (r_{t+2} + \gamma r_{t+3} + \ \dots) \\ &= r_{t+1} + \gamma\ V_\pi(s_{t+1}) \end{aligned}$$

avec r_{t+1}, r_{t+2}, … obtenus en suivant la politique π, i.e. $r_{k+1} = r_{k+1}\big(s_t, \pi(s_t)\big) \forall k \geq t$.

Pour Q:

$$Q_\pi(s_t, a_t) = r_{t+1} + \gamma\ Q_\pi(s_{t+1}, \pi(s_{t+1}))$$

avec r_{t+1} obtenu avec a_t puis r_{t+2}, r_{t+3}, … obtenus en suivant la politique π.

- Les décompositions

$$V_\pi(s) = r + \gamma\, V_\pi(s')$$

$$Q_\pi(s,a) = r + \gamma\, Q_\pi(s', \pi(s'))$$

où s' désigne le successeur de s (l'état suivant).

montrent l'intérêt du **compromis** entre recherche de récompense immédiate et à plus long terme.

Remarque : **stratégie gloutonne (*greedy*)** maximise la prochaine récompense ⇒ optimale pour $\gamma = 0$.

- Equations d'optimalité :

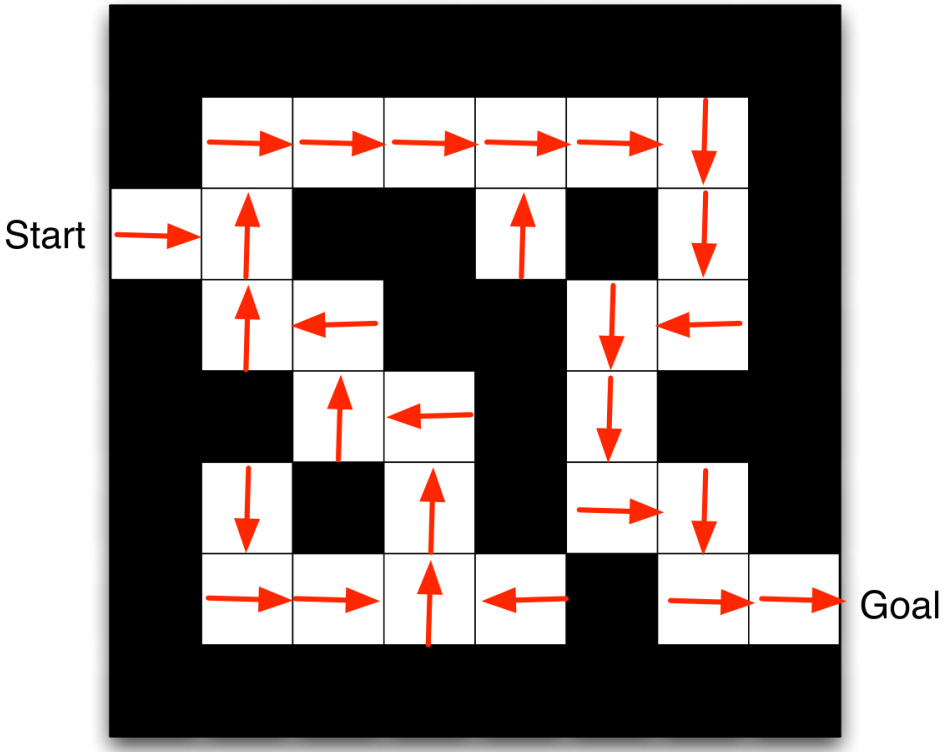
$$V_{\pi^*}(s) = \max_a(r(s,a) + \gamma V_{\pi^*}(s'(s,a))$$

$$Q_{\pi^*}(s,a) = \max_a(r(s,a) + \max_{a'} Q(s',a'))$$

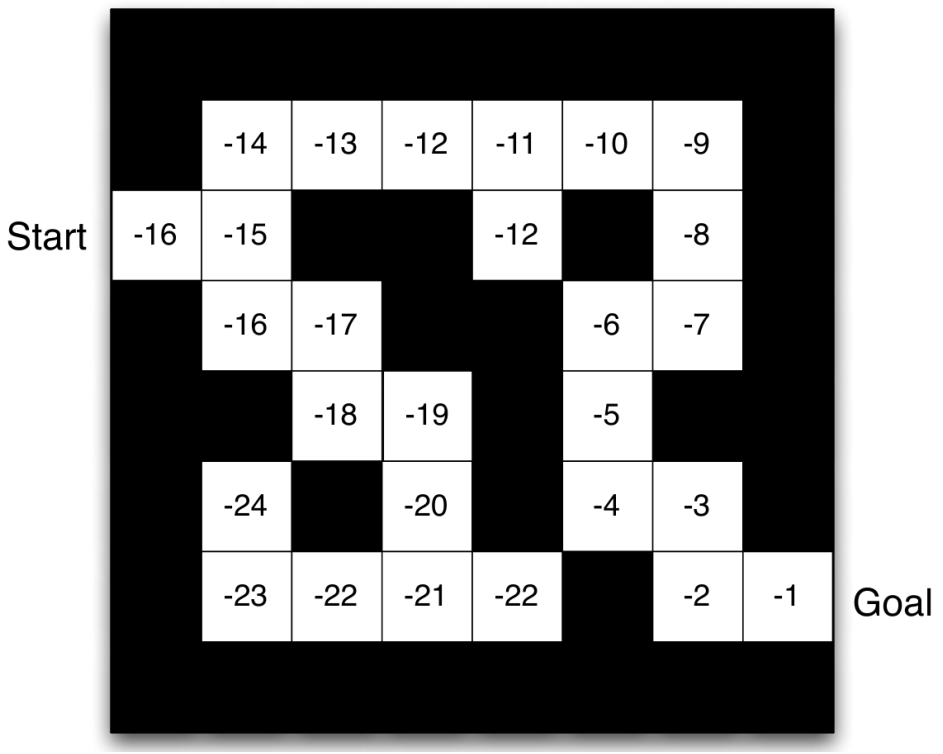
Exercice (labyrinthe) :

Vérifier pour quelques cases les équations d'optimalité.

π^*



V_{π^*}



[illustration: https://www.davidsilver.uk/teaching/]

5.3 Algorithme tabulaire

Objectif : résoudre (au moins) une équation d'optimalité de Bellman :

$$V_{\pi^*}(s) = \max_a(r(s,a) + \gamma V_{\pi^*}(s'(s,a))$$

$$Q_{\pi^*}(s,a) = \max_a(r(s,a) + \max_{a'} Q(s',a'))$$

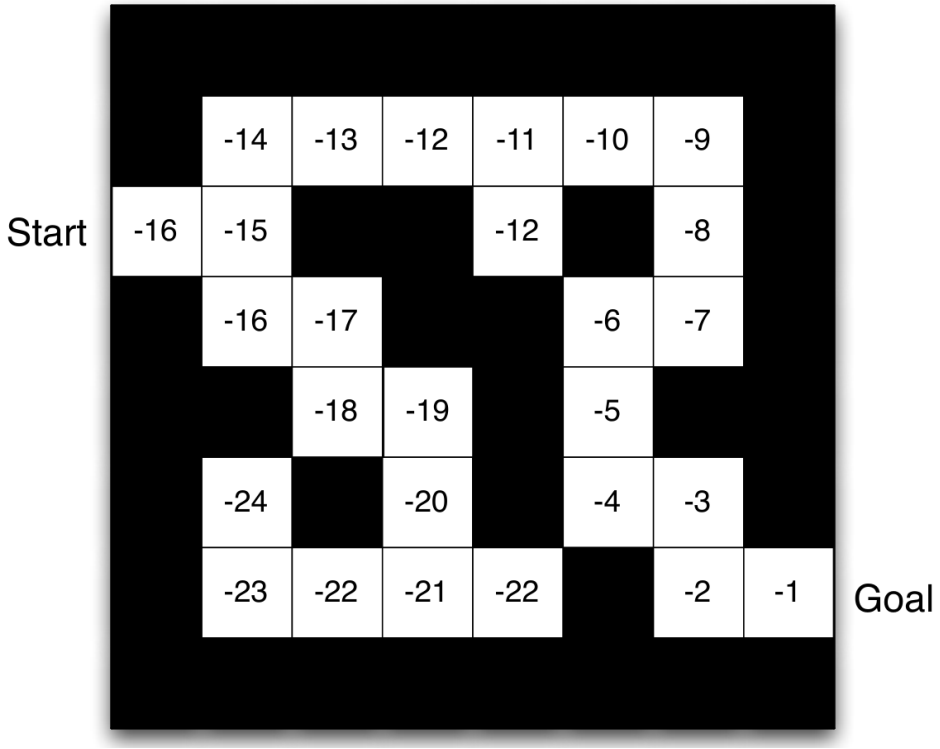
i.e. **déterminer V_{π^*} et/ou Q_{π^*}** de manière exacte ou approchée ($\hat{V}_{\pi^*}$, $\hat{Q}_{\pi^*}$).

Fonctions tabulaires : **représentation de V ou Q sous forme de tableau**.

⇒ 1. représentation exacte possible pour des modèles spatialement discrets, i.e. s et a appartiennent à des ensembles finis.

⇒ 2. limité à de petits modèles, i.e. s et a de petites dimensions.

Exemple (labyrinthe) :



Représentation de $V^*(s)$:

État s	C1	C2	B2	B4	…
$V^*(s)$	-16	-15	-14	-16	

Représentation de $Q^*(s,a)$:

	C1	C2	B2	B4	…
Droite					
Gauche					
Bas					
Haut					

Nombreuses méthodes de résolution itératives (dont les estimations de V et/ou Q convergent vers les valeurs exactes), e.g. Value itération, Policy itération, **Q-learning,** Sarsa.

Q-learning :

- apprentissage de Q, par mises à jour :

$$\hat{Q}_{\pi^*}(s,a) \coloneqq (1-\alpha)\hat{Q}_{\pi^*}(s,a) + \alpha(r + \max_{a'} \hat{Q}_{\pi^*}(s',a'))$$

avec s, r, s' collectés expérimentalement,

avec $\alpha \in]0;1]$ = ***learning rate***.

- Alternance d'expériences et de mises à jour de $\hat{Q}$.
- Les expériences utilisent l'estimation $\hat{Q}$ pour choisir de meilleurs actions = **exploitation**

  $\Rightarrow \hat{Q}$ précis pour les (s,a) d'intérêt.
- Pour garantir la convergence de $\hat{Q}$ vers Q, le choix d'action garde une part de hasard = **exploration**.

Remarque : en environnement stochastique, α « filtre » la variance expérimentale. En déterministe, on peut garder $\alpha = 1$ pour un apprentissage plus rapide.

Q-learning, pseudo-code (avec exploration ϵ-greedy):

entrée : $1 \geq \alpha > 0$, $\varepsilon > 0$, $k > 0$

sortie : tableau Q[., .]

initialiser Q[s, a] = 0 pour tout s et tout a

Répéter k fois // k = nombre d'épisodes

s := état initial

répéter // étapes d'un épisode

choisir une action a: // ϵ-greedy

si $\epsilon <$ random(0,1)

a := $\max_A Q(s, A)$ // exploitation

sinon

a :=random(actions) // exploration

exécuter l'action a, observer la récompense r et le nouvel état s' // nouvelle données

Q[s, a] := Q[s, a] + α (r + $\gamma \max_{a'}$ Q[s', a'] – Q[s, a]) // mise à jour de $\hat{Q}$

s := s'

jusqu'à ce que s soit l'état terminal

Exercice (Marche aléatoire) :

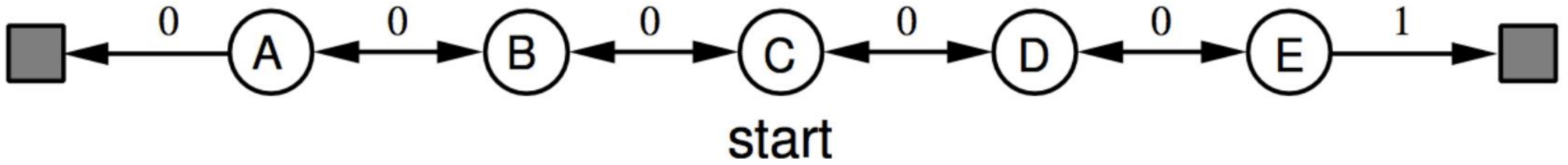
Objectif : atteindre la case de droite.

Actions : Droite, Gauche

Etats : A, B, C, D, E, But, Piège

Récompenses = 1 sauf si action Droite en E

Discount : $\gamma = 0.9$



Dresser un tableau de V^* (commencer par Goal, puis E, D, …).

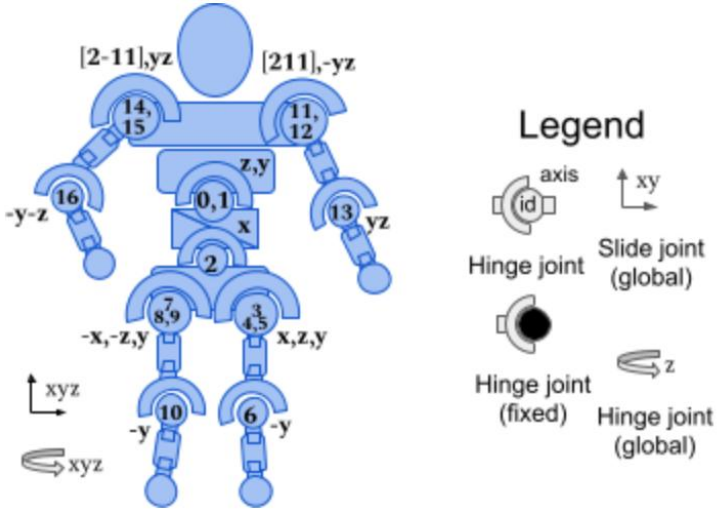
Dresser un tableau de Q^* pour E, D, C.

[illustration: https://www.davidsilver.uk/teaching/]

5.4 Approximation des fonctions de valeurs (NNs)

Les méthodes de RL peuvent être **appliquées sur de grands modèles** :

- Jeux (échecs, Go, Backgammon, …) ⇒ 10^{20} états au backgammon, beaucoup plus au échecs.
- Pilotage de drones ⇒ états/actions continus



Faiblesse des fonctions tabulaires :

- Limite du nombre d'états/actions : **capacité de stockage en mémoire**.
- Apprend chaque valeur $Q(s, a)$ individuellement (**pas de généralisation**) ⇒ lent.

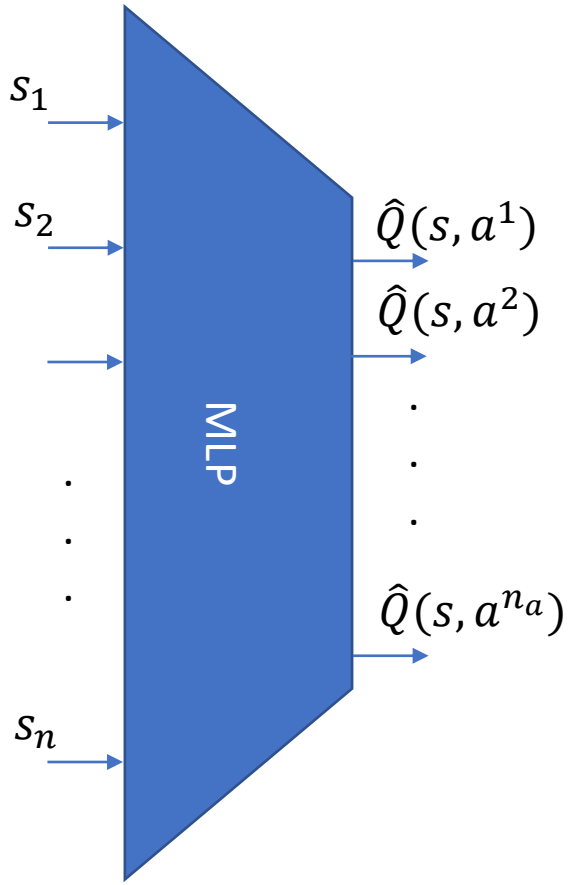
[illustration: https://gymnasium.farama.org]

Approximation des fonctions de valeurs par NNs:

- Remplace le tableau de valeurs $\hat{Q}[s, a]$ par un NN
- $s \in \mathbb{R}^n$ entièrement mesuré ⇒ utilisation d'un MLP possible (sinon besoin d'un RNN)
- a discret, prend n_a valeurs ⇒ entrées du NN = s, n_a sorties

Exemple (MLP):

- État continu : $s = [s_1, s_2, \dots, s_n] \in \mathbb{R}^n$
- Actions discrètes : $a \in \{a^1, a^2, \dots, a^{n_a}\}$



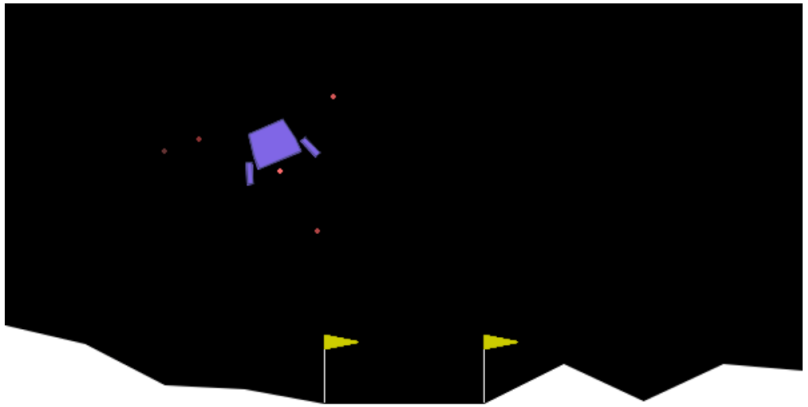
Adaptation de l'algorithme Q-learning :

- Utilisation d'un NN pour apprendre $Q(s, a)$
- Mise à jour de Q, i.e. du NN ne se fait plus après chaque action :
  - Utilisation de mini-batch, i.e. mise à jour sur plusieurs données (s, a, r, s')
  - Le NN doit être entrainée régulièrement avec les anciennes expériences (risque d'oubli)

  ⇒ utilisation d'un *dataset* qui mémorise les expériences (***Replay Buffer***)

Pour aller plus loin :

- De nombreuses évolution d'algorithmes sont disponibles, e.g. **DQN** propose l'utilisation de 2 NNs pour stabiliser l'apprentissage.
- Des librairies permettent de tester les algos sur des problèmes variés, e.g. https://gymnasium.farama.org/



Q-learning, avec NN pseudo-code (avec exploration ϵ-greedy):

entrée : $\eta > 0$, $\varepsilon > 0$, $k > 0$

sortie : un réseau de neurone Q

Initialiser un Replay Buffer : buffer = {}

Répéter k fois // k = nombre d'épisodes

s := état initial

répéter // étapes d'un épisode

choisir une action a: // ϵ-greedy

si $\epsilon <$ random(0,1) : a := $\max_A Q(s, A)$ // exploitation

sinon a :=random(actions) // exploration

exécuter l'action a, observer la récompense r et le nouvel état s' // nouvelle données

buffer := buffer U (s, a, r, s')

s := s'

jusqu'à ce que s soit l'état terminal

entraîner Q, avec Loss = r + $\gamma \max_{a'}$ Q[s', a'] – Q[s, a] // mise à jour de $\hat{Q}$