Chapitre 11

Apprentissage

11.1. Introduction

2019 : AlphaStar, l'IA de Google Deepmind, a battu des joueurs pro à Starcraft

En janvier 2019, DeepMind, l'intelligence artificielle née dans les labos de Google, a réussi un véritable exploit en dominant des humains à Starcraft II, l'un des jeux les plus populaires de stratégie en temps réel. Jusqu'à présent, l'intelligence artificielle était déjà parvenue à battre des joueurs humains sur des jeux Atari, Mario, Quake III ou encore Dota 2. Cependant, pour maîtriser StarCraft II, il était pour l'instant nécessaire de faire appel à des cartes simplifiées, de modifier les règles du jeu ou de créer des éléments du système à la main. AlphaStar est le premier à réussir l'exploit de battre un humain sans circonstances spéciales. Dans les faits, l'IA est ainsi parvenue à battre Grzegorz « MaNa » Komincz, de l'équipe *Team Liquid*, l'un des meilleurs joueurs professionnels au monde. Le programme a remporté la partie 5 à 0 dans des conditions de match professionnelles et sans restrictions. En fait, la seule restriction était pour la machine, trop rapide dans ses déplacements pour l'Homme. Il a donc fallu, volontairement, ralentir son exécution pour permettre au participant de jouer dans des conditions « humaines ».

Les chercheurs ont dû faire face à cinq obstacles de taille. Comme dans le jeu « pierre-papier-ciseaux », il n'existe pas de stratégie unique gagnante et l'IA doit constamment s'adapter. L'information est incomplète, les joueurs découvrant la carte au fur et à mesure. La relation de cause à effet n'est pas immédiate, et il faut être capable de planifier à long terme. De plus, le jeu se déroule en temps réel, plutôt qu'au tour par tour. Enfin, le nombre d'actions possibles est gigantesque, avec des centaines d'unités et de bâtiments à gérer dans l'immédiat. Les chercheurs avancent le chiffre de 10^{26} actions possibles à tout moment.

Ils ont réussi cet exploit grâce à un système reposant sur le principe du Deep Learning. Il utilise un réseau neuronal profond ainsi que l'apprentissage supervisé qui s'apparente à l'apprentissage par concept chez l'humain, et l'apprentissage par renforcement qui vise à obtenir une récompense maximale.

11.2. Apprentissage par renforcement

Cette technique commence à faire ses preuves pour de nombreuses applications. Et c'est surtout dans les jeux vidéos que l'apprentissage par renforcement impressionne.

L'apprentissage par renforcement est un mode d'apprentissage statistique, inspiré de l'apprentissage humain et animal. En tant qu'humain, nous expérimentons certaines choses. Lorsqu'elles ont un résultat positif et induisent des récompenses, on conclut que ces expériences sont positives et qu'elles doivent être réitérées. Inversement, si le résultat de l'expérience n'est pas concluant, on le mémorise pour ne plus faire la même erreur.

Ainsi, dans le cadre d'un apprentissage par renforcement, l'IA veut maximiser ses *récompenses*. Exactement de la même façon qu'un étudiant veut maximiser ses notes, par exemple.

Exercice 11.1 - Nim

Le principe du jeu de Nim est de faire s'affronter deux joueurs qui prendront chacun leur tour une, deux ou trois allumettes. Dans cette première variante, **celui qui prend la (ou les) dernière(s) allumette(s) a gagné**.

On verra qu'avec un algorithme simple et sans connaissance particulière sur la stratégie gagnante, on obtient une solution surprenante, où la « machine » devient infaillible à ce jeu.

Matériel

- 8 allumettes
- autant de gobelets que d'allumettes
- des petits jetons sur lesquels on peut écrire. Il en faut 3 par gobelet.

Pour préparer l'activité, on va numéroter les verres avec les nombres de 1 à 8, et mettre trois jetons par gobelet, numérotés 1, 2 et 3, sauf dans le gobelet 1 où on ne met qu'un jeton « 1 », et dans le gobelet 2 où on ne met que les jetons « 1 » et « 2 ».

a. Faire plusieurs parties en suivant la procédure suivante :

https://www.youtube.com/watch?v=IYEZbxB0h4s

- Mettre en place le matériel (jetons dans les gobelets, allumettes sur la table)
- Commencer une partie en laissant l'humain jouer en premier. La machine, quand c'est son tour, va :
  - compter le nombre d'allumettes et prendre le gobelet numéroté correspondant ;
  - piocher au hasard un jeton dans le gobelet, le poser sur la table juste devant ce gobelet, et prendre le nombre d'allumettes indiqué par le jeton (1, 2 ou 3) ;
  - en fin de partie (quand il n'y a plus d'allumettes),
    - soit la machine a gagné, et dans ce cas elle remet chaque jeton dans le gobelet où elle l'a pioché,
    - soit elle a perdu et dans ce cas elle remet tous les jetons piochés sauf le dernier qu'elle met de côté (c'était clairement un mauvais choix puisqu'il a permis à l'humain de gagner directement).
- Il peut arriver qu'un gobelet soit vide (parce que, par exemple, avec 4 allumettes il n'y a pas de bon choix). Dans ce cas :
  - on retourne le gobelet
  - si l'adversaire laisse la machine avec un nombre d'allumettes pour lequel le gobelet est retourné, elle abandonne la partie (car elle sait qu'elle va perdre) et remet tous les jetons piochés dans leur gobelet sauf le dernier pioché qu'elle met de côté.

Au bout de suffisamment de parties, quels jetons resteront dans les gobelets ?

b. Programmez cet algorithme. L'humain joue en premier.
c. Comme **b**, mais c'est la machine qui commence.
d. Comme **b**, mais celui qui prend la dernière allumette perd (c'est la variante misère).

Exercice 11.2 - Chomp : le jeu

Chomp est un jeu mathématique à deux joueurs qui se joue avec une tablette de chocolat, c'est-à-dire un rectangle composé de carrés. À son tour, un joueur choisit un carré et le mange, ainsi que tous les carrés situés à sa droite ou plus bas. Le carré en haut à gauche est empoisonné et celui qui le mange perd la partie.

Ce jeu a été inventé en 1952 par Frederik **Schuh**, en termes de choix de diviseurs à partir d'un entier donné, puis ré-inventé indépendamment en 1974 par David **Gale** sous sa formulation actuelle.

Situation initiale

1. Joueur 1

2. Joueur 2

3. Joueur 1

4. Joueur 2

5. Le joueur 1 a dû manger le carré empoisonné. Il a perdu.

a. Écrivez un programme (non graphique) où l'ordinateur joue des coups valides au hasard. Enfin presque : on ne prendra le carré empoisonné que si on est obligé. On appellera cet algorithme « Aléa ».
Lancez ensuite le programme un grand nombre de fois quand Aléa joue contre Aléa. Comptez le nombre de parties gagnées par chaque joueur. Que dire du résultat ?

Indications

La tablette (board) sera représentée en mémoire par un tableau bidimensionnel.

Un tableau bidimensionnel = une liste de listes

```
board = [['P', '#', '#', '#', '#', '#'],
         ['#', '#', '#', '#', '.', '.'],
         ['#', '.', '.', '.', '.', '.'],
         ['#', '.', '.', '.', '.', '.']]
```

`'P'` indique le carré empoisonné, `'#'` un carré comestible et `'.'` un carré déjà mangé.

Voici la version graphique, où on a noté les coordonnées des cases. Le premier indice est la ligne, le second la colonne. Ainsi, board[0][0] = `'P'`, board[1][2] = `'#'`, board[2][1] = `'.'`.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)

b. Faites une version graphique de ce programme.

Exercice 11.3 - Chomp : Positions gagnantes

Écrivez un programme Python qui trouve toutes les tablettes de chocolat de 6 colonnes et 4 lignes gagnantes. Voir *Positions P, positions S*, chapitre 10, page 10-9

Indications

Observons que chaque tablette a une propriété intéressante : les carrés de chocolat sont toujours « tassés » sur la gauche. De plus, il n'y a jamais de trous. Une ligne commence tout à gauche (s'il y a des carrés sur la ligne) et une fois que la ligne s'arrête, on est sûr qu'il n'y a plus de carrés.

Prenons une grille de 4 lignes et 6 colonnes. La tablette

pourra être codée sans équivoque 6411.

Chaque chiffre correspond à une ligne. Il indique le nombre de carrés de chocolat : 6 carrés sur la première ligne, 4 sur la deuxième, 1 et sur la troisième et 1 sur la quatrième.

On peut remarquer que la suite des chiffres composant le code est forcément décroissante (par exemple, la configuration 4522 est impossible).

Cette notation a plusieurs avantages :

- Elle est simple.
- Elle permet de générer rapidement toutes les configurations possibles : ce sont tous les nombres de 1000 à 6666 dont les chiffres sont décroissants.
- On peut aussi l'utiliser pour générer tous les coups légaux : ce sont les coordonnées des carrés bruns.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)		
(2,0)					
(3,0)					

Inconvénient : on doit se limiter à 9 colonnes. Si on veut plus de colonnes, on pourra utiliser une liste [6, 5, 1, 1] plutôt qu'un nombre. Pour cet exercice, neuf colonnes seront amplement suffisantes.

Par contre, le nombre de lignes est illimité.

Exercice 11.4 - Chomp : apprentissage

a. En vous inspirant de l'exercice 11.1, imaginez un processus d'apprentissage par renforcement qui permettra à votre IA de progresser à ce jeu. Au début, elle jouera des coups au hasard, puis, petit à petit, arrivera à sélectionner les meilleurs coups.

b. Programmez votre méthode.

c. Testez votre programme : IA vs Aléa (voir ex. 11.2). Comptez le nombre de parties gagnées par chaque joueur et observez la progression de l'IA.

d. Testez votre programme : Aléa vs IA.

e. Testez votre programme : IA vs IA.

Exercice 11.5 - Chomp : développer une stratégie

Après un grand nombre de parties du programme de l'ex 11.4e, écrivez dans un fichier les meilleurs coups pour toutes les tablettes.

Exercice 11.6 – Chomp : humain vs IA

Écrivez un programme où un humain joue contre l'IA. L'humain commencera (parce que c'est sa seule chance de gagner).

L'ordinateur regardera dans le fichier de l'ex 11.5 quel coup jouer.

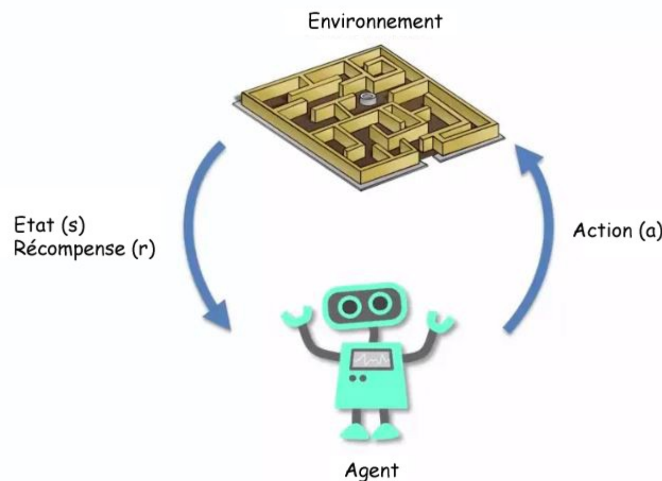
11.3. Q-learning

Un *agent* est par exemple un robot ou un programme informatique.

En intelligence artificielle, plus précisément en apprentissage automatique, le *Q-learning* est un algorithme d'apprentissage par renforcement. La lettre Q désigne la fonction qui mesure la qualité d'une action exécutée dans un état donné du système.

Considérons un système quelconque : par exemple, un jeu vidéo, un robot qui doit s'évader d'un labyrinthe, ou encore un robot qui doit apprendre à tenir un œuf. Un agent doit alors apprendre à réaliser une tâche : gagner une partie de jeu vidéo avec le plus de points, s'évader d'un labyrinthe sans se faire attraper, tenir l'œuf sans le casser.

Le Q-learning permet d'apprendre une stratégie, qui indique quelle action effectuer dans chaque état du système. Par exemple, le robot peut apprendre d'aller à droite quand il se trouve sur la case (2, 3), mais d'aller en haut s'il se trouve sur la case (4, 6), etc. À chaque étape, l'agent reçoit une récompense immédiate, qui est un nombre réel. Typiquement, l'agent peut recevoir 100 (points) s'il est sorti du labyrinthe, ou s'il a gagné le jeu. L'agent perd 1000 points s'il est piégé, ou si l'œuf est cassé. C'est le concepteur de l'environnement qui choisit ces récompenses immédiates lorsqu'il modélise le système.



Plus précisément, le Q-learning apprend une fonction de valeur notée Q. Cette fonction Q estime le gain potentiel, c'est-à-dire la somme des récompenses $Q(s,$a) sur le long terme, apportée par le choix d'une certaine action a dans un certain état s en suivant une politique optimale. Cette fonction Q s'appelle fonction de valeur actions-états, puisqu'elle prend comme argument un état s et une action a. Par exemple, Q(être en (2, 3), aller à droite) est une estimation de la somme des récompenses si l'agent est dans la case (2, 3) et décide d'aller à droite. Lorsque cette fonction de valeur Q d'actions-états est connue ou apprise par l'agent, la stratégie optimale peut être construite en sélectionnant l'action à valeur maximale pour chaque état, c'est-à-dire en sélectionnant l'action a qui maximise la valeur $Q(s,a)$ quand l'agent se trouve dans l'état s.

Autrement dit, si l'agent est sur la case (2, 3), il regarde les valeurs Q(être en (2, 3), aller à droite), Q(être en (2, 3), aller à gauche), Q(être en (2, 3), aller en haut), Q(être en (2, 3), aller en bas) et Q(être en (2, 3), rester sur place). Il choisit alors l'action qui maximise la valeur. Par exemple si :

- Q(être en (2, 3), aller à droite) = 78
- Q(être en (2, 3), aller à gauche) = 22
- Q(être en (2, 3), aller en haut) = -5
- Q(être en (2, 3), aller en bas) = 12
- Q(être en (2, 3), rester sur place) = -18

alors l'agent décide d'aller à droite (car 78 est le gain maximal).

La situation consiste en un agent, un ensemble d'états S et d'actions A. En réalisant une action a, l'agent passe d'un état s à un nouvel état s' et reçoit une récompense r (c'est une valeur numérique). Le but de l'agent est de maximiser sa récompense totale. Cela est réalisé par apprentissage de l'action optimale pour chaque état. L'action optimale pour chaque état correspond à celle avec la plus grande récompense sur le long terme. Cette récompense est la somme pondérée de l'espérance mathématique des récompenses de chaque étape future à partir de l'état actuel. La pondération de chaque étape peut être $\gamma^{\Delta t}$ où Δt est le délai entre l'étape actuelle et l'étape future, et γ un nombre compris entre 0 et 1 appelé *facteur d'actualisation*.

L'algorithme calcule une fonction de valeur action-état Q. Avant que l'apprentissage ne débute, la fonction Q est initialisée arbitrairement. Ensuite, à chaque choix d'action, l'agent observe la récompense et le nouvel état (qui dépend de l'état précédent et de l'action actuelle). Le cœur de l'algorithme est une mise à jour de la fonction de valeur. La définition de la fonction de valeur est mise à jour à chaque étape de la façon suivante :

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a')\right)$$

où s' est le nouvel état, s est l'état précédent, a est l'action choisie, r est la récompense reçue par l'agent, α est un nombre entre 0 et 1, appelé *facteur d'apprentissage*, et γ est le *facteur d'actualisation*.

Un épisode de l'algorithme finit lorsque s_{t+1} est un état final.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Pseudo-code du Q-learning

```
entrée: taux d'apprentissage α > 0 et taux γ > 0
sortie: tableau Q[., .]

initialiser Q[s, a] pour tout état s non final, toute action a de façon arbitraire,
        et Q(état terminal, a) = 0 pour toute action a

répéter
   // début d'un épisode
   s := état initial

   répéter
      // étape d'un épisode
      choisir une action a depuis s en utilisant la politique spécifiée par Q
      exécuter l'action a
      observer la récompense r et le nouvel état s'

      Q[s, a] := Q[s, a] + α[r + γ max_a' Q(s', a') - Q(s, a)]

      s := s'
   jusqu'à ce que s soit l'état terminal
```

Influence des variables sur l'algorithme

Facteur d'apprentissage

Le facteur d'apprentissage α détermine à quel point la nouvelle information calculée surpassera l'ancienne. Si α = 0, l'agent n'apprend rien. *A contrario*, si α = 1, l'agent ignore toujours tout ce qu'il a appris jusqu'à présent et ne considérera que la dernière information.

Dans un environnement déterministe, la vitesse d'apprentissage $\alpha_t(s,a)=1$ est optimale. Lorsque le problème est aléatoire, l'algorithme converge sous certaines conditions dépendant de la vitesse d'apprentissage.

En pratique, souvent cette vitesse correspond à $\alpha_t(s,a)=0.1$ pour toute la durée du processus.

Facteur d'actualisation

Le facteur d'actualisation γ détermine l'importance des récompenses futures. Un facteur γ = 0 rendrait l'agent myope en ne considérant que les récompenses courantes, alors qu'un facteur γ proche de 1 ferait aussi intervenir les récompenses plus lointaines.

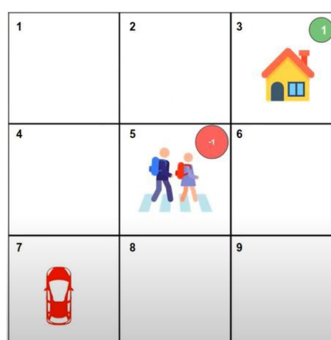
En pratique, on prend souvent γ = 0.9.

11.3.1. Un exemple complet

Voir [2]

Nous allons analyser le programme Python écrit par Thibault Neveu et qu'il explique dans sa vidéo disponible sur Youtube à l'adresse https://youtu.be/a0bVIyIJ074. On n'a modifié (en gras) que les commentaires.

Il s'agit pour la voiture (l'agent) d'apprendre le chemin qui lui permettra de retourner à la maison, sans renverser les piétons évidemment.



La case « maison » contient une récompense de 1, tandis que la case « piéton » contient une récompense de -1.

Les actions possibles sont « aller en haut », « aller en bas », « aller à gauche », « aller à droite ».

```
import numpy as np
from random import randint
import random

class EnvGrid(object):
    """
        docstring forEnvGrid.
    """
    def __init__(self):
        super(EnvGrid, self).__init__()

        # Grille avec les récompenses
        self.grid = [
            [0, 0, 1],
            [0, -1, 0],
            [0, 0, 0]
        ]
        # Position de départ de la voiture
        self.y = 2
        self.x = 0

        # Actions possibles (déplacement de la voiture)
        self.actions = [
            [-1, 0], # en haut
            [1, 0],  # en bas
            [0, -1], # à gauche
            [0, 1]   # à droite
        ]

    def reset(self):
        """
            Reset world
        """
        self.y = 2
        self.x = 0
        return (self.y*3+self.x+1)

    def step(self, action):
        # Déplace la voiture sur la grille selon l'action donnée en paramètre
        """
            Action: 0, 1, 2, 3
        """
        self.y = max(0, min(self.y + self.actions[action][0],2))
        self.x = max(0, min(self.x + self.actions[action][1],2))

        return (self.y*3+self.x+1), self.grid[self.y][self.x]

    def show(self):
        # Affiche la grille à l'écran
        """
            Show the grid
        """
        print("---------------------")
        y = 0
        for line in self.grid:
            x = 0
            for pt in line:
                print("%s\t" % (pt if y != self.y or x != self.x else "X"), end="")
                x += 1
            y += 1
            print("")

    def is_finished(self):
        return self.grid[self.y][self.x] == 1

def take_action(st, Q, eps):
    # Choisit une action : soit au hasard, soit en prenant celle qui donne la plus
    # grande récompense (greedy action). Dépend de eps.
```

```
    if random.uniform(0, 1) < eps:
        action = randint(0, 3)
    else: # Or greedy action
        action = np.argmax(Q[st])
    return action

if __name__ == '__main__':
    env = EnvGrid()
    st = env.reset()

    # Initialise la Q-table
    # Pour chaque case (la première ligne ne sert à rien), on note pour chacune
    # des actions possibles (haut, bas, gauche, droite) la récompense

    Q = [
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ]

    for _ in range(100):
        # Réinitialise l'épisode (il y en aura 100)
        st = env.reset()  # position de la voiture
        while not env.is_finished():
            #env.show()
            #at = int(input("$>"))
            at = take_action(st, Q, 0.4)  # action choisie

            stp1, r = env.step(at)  # nouvelle position de la voiture et récompense
            #print("s", stp1)
            #print("r", r)

            # Met à jour la Q-table
            atp1 = take_action(stp1, Q, 0.0)
            Q[st][at] = Q[st][at] + 0.1*(r + 0.9*Q[stp1][atp1] - Q[st][at])

            st = stp1

    for s in range(1, 10):
        print(s, Q[s])
```

robot.py

Exercice 11.7

Modifiez le programme graphique donné sur le site web compagnon pour que la voiture apprenne à trouver son chemin sur une grille plus grande.

Exercice 11.8 - Nim

Écrivez un programme en Python qui apprend à jouer au jeu de Nim (voir exercice 11.1b).

On supposera qu'il y a 15 allumettes au départ et que l'humain commence. En fait, l'humain est une fonction qui enlève aléatoirement de 1 à 3 allumettes. L'IA va apprendre à bien jouer grâce au Q-learning.

Les actions possibles sont « je prends 1, 2 ou 3 allumettes ».

Chacun des 16 états correspond au nombre d'allumettes restantes (de 0 à 15).

Attention ! L'état suivant pour l'IA ne sera pas le nombre d'allumettes qui restent après son coup, mais l'état obtenu après que l'humain ait joué son tour.

Affichez la table Q à la fin de l'apprentissage. Correspond-elle à ce que vous attendiez ?

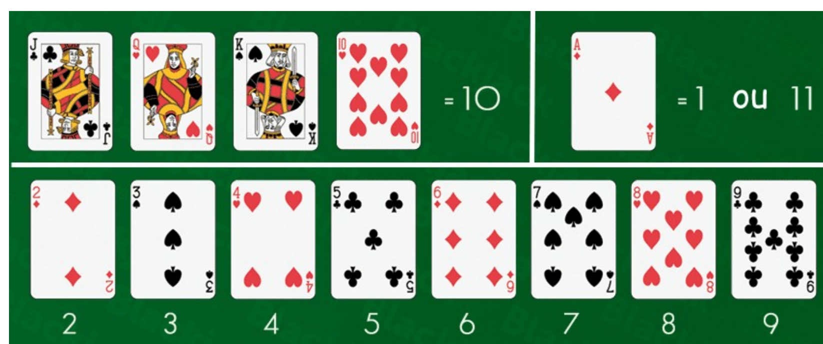
Exercice 11.9 – Blackjack (règles simplifiées)

Les règles viennent de Wikipédia. Elles ont été un peu simplifiées.

La partie oppose individuellement chaque joueur à la banque. Le but est de battre le croupier sans dépasser le score de 21. Dès qu'un joueur fait plus que 21, on dit qu'il « brûle » et il perd sa mise initiale.

Les valeurs des cartes sont les suivantes :

- de 2 à 9 : valeur nominale de la carte ;
- de 10 au roi (surnommées « bûche ») : 10 points ;
- as : 1 ou 11 points (au choix du joueur).



Un blackjack est la situation où le joueur reçoit, dès le début du jeu, un as et une bûche. Si le joueur atteint 21 points avec plus de deux cartes, on compte 21 points et non blackjack.

Au début de la partie, le croupier distribue une carte face visible à chaque joueur et une carte face visible pour lui. Il tire ensuite pour chaque joueur une seconde carte face visible et une seconde carte face cachée pour lui.

Les joueurs débutent donc la partie avec deux cartes visibles.

Si le joueur a un blackjack, il n'a aucun choix à faire. Le joueur attend donc l'annonce des résultats.

Si le joueur n'a pas de blackjack, plusieurs choix sont possibles :

- demander une ou plusieurs cartes supplémentaires, afin de se rapprocher de 21, sans le dépasser (en anglais, on dit « *Hit* ») ;
- s'arrêter ou « rester » (en anglais, on dit « *Stay* »), et donc conserver ses cartes.

Une fois tous les joueurs servis, le croupier joue pour son compte selon une règle universelle : « la banque tire à 16, reste à 17 ». Ainsi, le croupier tire des cartes jusqu'à atteindre un nombre supérieur ou égal à 17.

Une fois le croupier servi, il annonce les résultats pour chaque joueur.

Les joueurs ayant dépassé 21 points perdent leur mise dans tous les cas.

Les joueurs ayant un blackjack gagnent, sauf si le croupier a également un blackjack, auquel cas

il y a égalité.

Enfin, pour les joueurs ayant 21 points (sans blackjack) ou moins, le résultat dépend du nombre de points du croupier :

- Si le croupier a dépassé les 21 points, tous les joueurs encore dans le jeu (et donc n'ayant pas dépassé les 21 points) gagnent.
- Si le croupier a 21 points (sans blackjack) ou moins :
  - Les joueurs ayant moins de points que le croupier perdent ;
  - Il y a égalité si les joueurs ont autant de points que le croupier.
  - Les joueurs ayant plus de points que le croupier gagnent.

a. Écrivez un programme en Python qui apprend à jouer au blackjack en utilisant le Q-learning. Il y aura un seul joueur qui affrontera le croupier. Avec nos règles simplifiées, seules deux actions sont possibles : HIT ou STAY. Le but de l'apprentissage est de savoir quelle action choisir en fonction du nombre de points du joueur et de la valeur de la première carte du croupier.

b. On trouve sur le web un tableau présentant la stratégie optimale pour le joueur :

	Points du **croupier** après sa première carte									
Points du **joueur**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**As**
4-8	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT
9	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT
10	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT
11	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT	HIT
12	HIT	HIT	STAY	STAY	STAY	HIT	HIT	HIT	HIT	HIT
13	STAY	STAY	STAY	STAY	STAY	HIT	HIT	HIT	HIT	HIT
14	STAY	STAY	STAY	STAY	STAY	HIT	HIT	HIT	HIT	HIT
15	STAY	STAY	STAY	STAY	STAY	HIT	HIT	HIT	HIT	HIT
16	STAY	STAY	STAY	STAY	STAY	HIT	HIT	HIT	HIT	HIT
17+	STAY	STAY	STAY	STAY	STAY	STAY	STAY	STAY	STAY	STAY

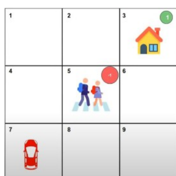
Est-ce que votre tableau ressemble à celui ci-dessus ?

11.3.2. Q-learning : résumé

L'algorithme Q-learning est une technique d'apprentissage automatique utilisée dans le domaine de l'apprentissage par renforcement pour prendre des décisions séquentielles. Voici une explication simplifiée de son fonctionnement général :

1. **État** (State) : L'environnement dans lequel un **agent** prend des actions est divisé en états. Chaque état représente une situation ou une configuration spécifique de l'environnement.

Exemple : une voiture (l'agent) se déplace dans un petit labyrinthe. Les cases sont les états. La case centrale est à éviter et le but est la maison.



2. **Action** (Action) : L'agent peut effectuer différentes actions à partir de chaque état. Ces actions sont les choix que l'agent peut faire pour interagir avec l'environnement.

Dans l'exemple, la voiture peut se déplacer verticalement ou horizontalement, sans sortir de la grille.

3. **Récompense** (Reward) : À chaque étape, l'agent reçoit une récompense de l'environnement en fonction de l'action qu'il a choisie et de l'état dans lequel il se trouve. L'objectif de l'agent est de maximiser les récompenses cumulées au fil du temps.

Dans l'exemple, il y a 1 point de récompense si on atteint la maison et -1 point si on renverse les piétons.

4. **Table Q** (Q-Table) : L'algorithme Q-learning maintient une table appelée « Q-table » qui stocke les valeurs Q. Chaque case de cette table représente une paire état-action et contient une valeur Q qui estime la récompense cumulée attendue en effectuant cette action à partir de cet état.

Q-table		**Actions**			
Etats		Nord	Sud	Ouest	Est
	1	0	0	0	0
	2	0	0	0	0
	3	0	0	0	0
	4	0	0	0	0
	5	0	0	0	0
	6	0	0	0	0
	7	0	0	0	0
	8	0	0	0	0
	9	0	0	0	0

5. **Exploration vs Exploitation** : L'agent doit trouver un équilibre entre l'exploration (essayer de nouvelles actions pour découvrir de meilleures stratégies) et l'exploitation (choisir les actions qui, selon la Q-table, semblent donner les meilleures récompenses).

6. **Mise à jour de la Q-table** : L'algorithme met à jour la Q-table en utilisant une formule de mise à jour basée sur la récompense reçue. La nouvelle valeur Q pour une paire état-action est calculée en prenant en compte la récompense instantanée, la valeur Q actuelle pour l'état suivant, et un taux d'apprentissage.

La formule pour la nouvelle valeur de Q est :

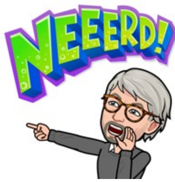
$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a')\right)$$

où s' est le nouvel état, s est l'état précédent, a est l'action choisie, r est la récompense reçue par l'agent, α est un nombre entre 0 et 1 (souvent 0.1), appelé *facteur d'apprentissage*, et γ est le *facteur d'actualisation* (souvent 0.9).

7. **Apprentissage itératif** : L'agent répète le processus de sélection d'une action, d'interaction avec l'environnement, de réception de récompenses et de mise à jour de la Q-table à de multiples reprises. Avec le temps, la Q-table se raffine pour mieux représenter les meilleures stratégies à suivre dans chaque état.

Q-table		**Actions**			
Etats		Nord	Sud	Ouest	Est
	1	0.33	0.39	0.41	0.90
	2	0.47	-0.41	0.42	0.99
	3	0	0	0	0
	4	0.81	0.29	0.38	-0.47
	5	0.69	0.70	0.14	0.06
	6	0.41	0.01	-0.35	0.03
	7	0.72	0.26	0.23	0.09
	8	-0.25	0.09	0.29	0
	9	0.05	0.01	0	0

8. **Convergence** : Avec suffisamment d'itérations, l'algorithme Q-learning converge vers une stratégie qui maximise les récompenses cumulées au fil du temps.



Source : [3]

11.4. Les réseaux de neurones

Un peu d'histoire

Au début des années 1950, Alan **Turing** propose la notion d'intelligence artificielle. Arthur **Samuel** crée le premier jeu informatique de dames capable d'auto-apprentissage et Marvin **Minsky** & Dean **Edmonds** construisent la première machine à réseau neuronal (Stochastic Neural Analog Reinforcement Calculator). En 1957 Frank **Rosenblatt** propose, à partir des travaux de **McCulloch** & **Pitts** (1943), un modèle mathématique du fonctionnement neuronal : le *perceptron* qui est à la base des réseaux de neurones artificiels et de l'intelligence artificielle dite connexionniste. En 1967 Marvin **Minsky** dénonce les limites du perceptron dans sa configuration mono-couche, et promeut l'intelligence artificielle symbolique à l'origine des systèmes experts des années 70-80.

L'intervention de Minsky a eu pour effet de couper les crédits de recherche alloués à la recherche en IA connexionniste, pour les reporter vers l'IA symbolique qui s'est développée autour de super-calculateurs. Pourtant, en 1974, Paul **Werbos** décrit une méthode d'apprentissage d'un réseau multicouche, par rétro-propagation de l'erreur. Cette thèse d'étudiant passe totalement inaperçue dans le contexte de l'époque où l'approche connexionniste a été enterrée.

En 1980 **Searle** montre avec son expérience de la « chinese room » les limites de systèmes experts, ou moteurs d'inférences basés sur les règles qui ne peuvent que simuler une intelligence (IA faible). Dans la même année **Fukushima** introduit la notion de réseau de neurones convolutif. Dans celui-ci les couches de neurones artificiels alternent entres des couches de convolution qui extraient des caractéristiques de l'objet étudié (features) et des couches qui normalisent et simplifient l'image obtenue pour garantir une certaine cohésion.

Finalement, en 1989, Yann **LeCun** montre que les réseaux de neurones convolutifs à rétropropagation de gradient permettent de remplir des tâches réelles comme la reconnaissance automatique des codes postaux écrits à la main. Le deep learning est né.

En 2005 avec **Hadoop**, le traitement distribué en réseau remplace les supercalculateurs. En 2006 Geoffrey **Hinton** contribue à généraliser l'utilisation des réseaux de neurones multi-couches avec rétropropagation du gradient.

En 2016, Alpha Go, mis au point par Google DeepMind, bat le champion Lee **Sedol** au jeu de go. Cette prouesse est remarquable dans la mesure où les méthodes combinatoires classiques sont ici tout à fait inenvisageables. Puis, en 2017, la nouvelle version Alpha Go zéro humiliait Alpha Go par 100 parties à 0. Qu'est-ce qui a changé dans la nouvelle version ? L'apprentissage par renforcement a été utilisé. L'ordinateur n'a pas seulement appris de parties jouées par des humains, mais il a appris en jouant des millions de parties contre lui-même.

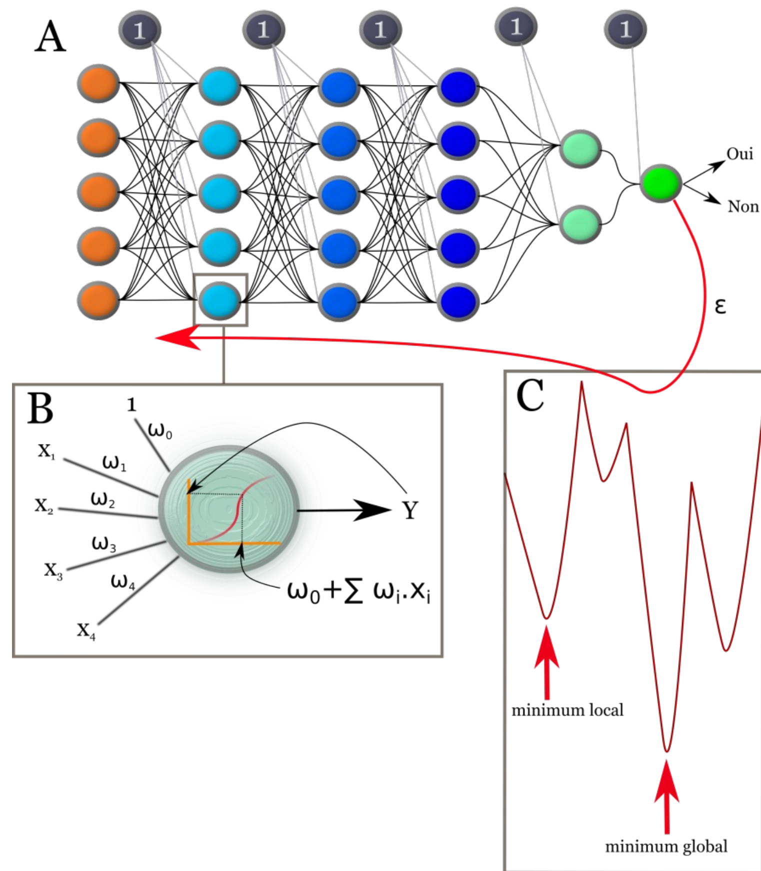
Principe des réseaux de neurones

(A) Le réseau de neurones est constitué d'unités d'intégration de l'information (neurone ou perceptron) reliées entre elles et organisées ici en couches. La première couche reçoit les caractères de l'objet à étudier (par exemple les pixels d'une image) et la dernière fournit la réponse. Les neurones gris sont des neurones de biais, ils délivrent toujours la valeur 1.

(B) Le neurone reçoit des informations X_1 à X_4 auxquelles sont appliqués des poids ω puis il les ajoute et passe le résultat à une fonction seuil qui détermine la réponse Y. Cette réponse constitue une entrée pour la couche de neurones plus profonde. Les bases mathématiques d'un réseau de neurones entraîné sont donc extrêmement simples et le calcul est facilement parallélisable.

Pour l'apprentissage, à chaque nouvelle expérience soumise le réseau fournit une réponse juste ou pas. Si la réponse est connue *a priori*, il est alors facile de mesurer l'écart entre la réponse fournie par le réseau et la réponse attendue. C'est ce qu'on appelle l'erreur ε et c'est cette erreur qui est rétropropagée pour optimiser les poids au niveau des entrées de chaque neurone.

(C) C'est là que la complexité mathématique apparaît car l'opération d'optimisation se heurte au problème des minima locaux. En effet, la solution de l'optimisation est rarement convexe comme le montre la figure C où un minimum global est entouré de minima locaux qui peuvent piéger l'algorithme de minimisation. Il faut avoir conscience que le problème représenté en C est un problème à une dimension et fini, alors que les problèmes soumis aux réseaux de neurones peuvent avoir plusieurs millions de dimensions avec une combinatoire qui flirte avec l'infini !



L'entraînement d'un réseau de neurones

Source : [5]

L'entraînement d'un réseau de neurones est similaire au processus d'essais et d'erreurs. Imaginons que vous deviez deviner un prix. Vous proposez un nombre, puis l'on vous dit « C'est moins ». Alors vous proposez un prix plus bas. Si l'on vous dit « C'est plus », vous proposez un prix plus élevé. De proche en proche, vous allez vous rapprocher du vrai prix.

Avec les réseaux de neurones, le processus est très similaire : on commence avec des poids **(weights)** aléatoires et des vecteurs de biais **(bias)**, on fait une prédiction, on la compare à la sortie souhaitée et on ajuste les vecteurs pour prédire plus précisément la prochaine fois. Le processus se poursuit jusqu'à ce que la différence entre la prédiction et les cibles correctes soit minimale.

Savoir quand arrêter l'entraînement et quel objectif de précision fixer est un aspect important de l'entraînement des réseaux de neurones, principalement en raison des problèmes de *surajustement* et de *sous-ajustement*.

Vecteurs, poids et produit scalaire

Travailler avec des réseaux de neurones consiste à faire des opérations avec des vecteurs. Les vecteurs sont utiles dans l'apprentissage en profondeur principalement en raison d'une opération particulière : le produit scalaire.

Calcul du produit scalaire de deux vecteurs : $\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$

Les principaux vecteurs à l'intérieur d'un réseau de neurones sont les poids et les vecteurs de biais. En gros, ce qu'on veut que le réseau de neurones fasse, c'est vérifier si une entrée est similaire à d'autres entrées qu'il a déjà vues. Si la nouvelle entrée est similaire aux entrées vues

précédemment, les sorties seront également similaires. C'est ainsi que vous obtenez le résultat d'une prédiction.

Nous utiliserons le module **NumPy** pour représenter les vecteurs d'entrée du réseau sous forme de listes. Mais avant, on va jouer avec les vecteurs en Python pour mieux comprendre comment ça marche.

Tout d'abord, nous allons définir trois vecteurs, un pour l'entrée et les deux autres pour les poids. Ensuite, nous calculons le produit scalaire. Puisque tous les vecteurs sont bidimensionnels, voici les étapes à suivre :

1. Multiplier le premier indice de `input_vector` par le premier indice de `weights_1`.
2. Multiplier le deuxième indice de `input_vector` par le deuxième indice de `weights_1`.
3. Additionner les résultats des deux multiplications.

Voici le code pour calculer le produit scalaire de `input_vector` et `weights_1` :

```
In [1]: input_vector = [1.72, 1.23]
In [2]: weights_1 = [1.26, 0]
In [3]: weights_2 = [2.17, 0.32]

In [4]: # Computing the dot product of input_vector and weights_1
In [5]: first_indexes_mult = input_vector[0] * weights_1[0]
In [6]: second_indexes_mult = input_vector[1] * weights_1[1]
In [7]: dot_product_1 = first_indexes_mult + second_indexes_mult

In [8]: print(f"The dot product is: {dot_product_1}")
Out[8]: The dot product is: 2.1672
```

Le résultat du produit scalaire est $1.72 \cdot 1.26 + 1.23 \cdot 0 = 2.1672$. Maintenant que nous savons comment calculer le produit scalaire, voici comment calculer `dot_product_1` en utilisant `np.dot()` :

```
In [9]: import numpy as np
In [10]: dot_product_1 = np.dot(input_vector, weights_1)
In [11]: print(f"The dot product is: {dot_product_1}")
Out[11]: The dot product is: 2.1672
```

`np.dot()` fait la même chose qu'avant, mais il suffit maintenant de spécifier les deux listes comme arguments. Calculons maintenant le produit scalaire de `input_vector` et `weights_2` :

```
In [10]: dot_product_2 = np.dot(input_vector, weights_2)
In [11]: print(f"The dot product is: {dot_product_2}")
Out[11]: The dot product is: 4.1259
```

Cette fois, le résultat est 4.1259.

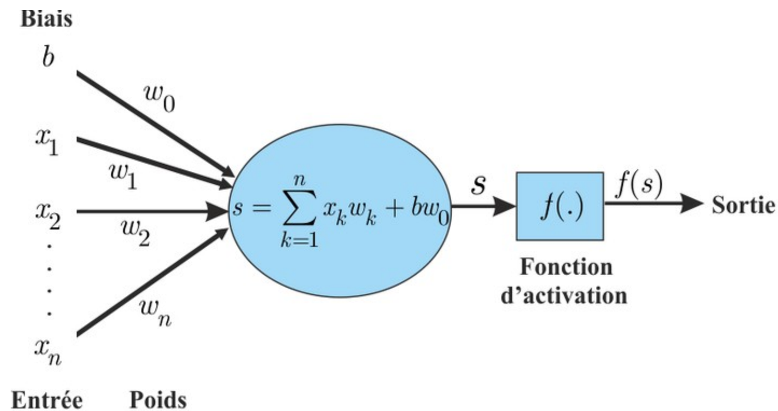
Notre premier réseau de neurones

La première étape de la construction d'un réseau de neurones consiste à générer une sortie à partir des données d'entrée. Nous ferons cela en créant une somme pondérée des variables. La première chose que nous devrons faire est de représenter les entrées avec Python et NumPy.

Le perceptron

Nous allons nous intéresser au réseau de neurones le plus simple : le perceptron. Le perceptron est un algorithme d'apprentissage supervisé de classifieurs binaires (c'est-à-dire séparant deux classes). Il a été inventé en 1957 par le psychologue américain Frank **Rosenblatt** au laboratoire d'aéronautique de l'université Cornell. Il s'agit d'un neurone formel muni d'une règle d'apprentissage qui permet de déterminer automatiquement les poids synaptiques de manière à séparer un problème d'apprentissage supervisé. Si le problème est linéairement séparable, un théorème assure que la règle du perceptron permet de trouver une séparatrice entre les deux classes.

Frank **Rosenblatt**
(1928-1971)



L'**apprentissage supervisé** est une tâche d'apprentissage automatique consistant à apprendre une fonction de prédiction à partir d'exemples annotés, au contraire de l'apprentissage non supervisé.

Le résultat de sortie peut être 0 ou 1. Il s'agit d'un problème de classification, un sous-ensemble de problèmes d'apprentissage supervisé dans lequel on dispose d'un ensemble de données avec les entrées et les cibles connues. Voici les entrées et les sorties du jeu de données :

Vecteur d'entrée (input)	Cible (Target)
[1.66, 1.56]	1
[2, 1.5]	0

La cible est la variable que nous souhaitons prédire. Dans cet exemple, nous avons affaire à un ensemble de données composé de nombres. Ce n'est pas courant dans un scénario de production réel. Habituellement, lorsqu'un modèle d'apprentissage en profondeur est nécessaire, les données sont présentées dans des fichiers, tels que des images ou du texte.

Première prédiction

Puisqu'il s'agit de notre tout premier réseau de neurones, nous créerons un réseau avec seulement une couche et un neurone… Jusqu'à présent, nous avons vu que les deux seules opérations utilisées à l'intérieur du réseau de neurones étaient le produit scalaire et la somme. Les deux sont des opérations linéaires.

Si nous ajoutons plus de couches mais que nous continuons à n'utiliser que des opérations linéaires, l'ajout de couches supplémentaires n'aura aucun effet car chaque couche aura toujours une certaine corrélation avec l'entrée de la couche précédente. Cela implique que, pour un réseau à plusieurs couches, il y aurait toujours un réseau avec moins de couches qui prédit les mêmes résultats.

Notre réseau utilisera la fonction d'activation sigmoïde. Les deux seules sorties possibles dans l'ensemble de données sont 0 et 1, et la fonction sigmoïde limite la sortie à une plage comprise entre 0 et 1.

Voici la formule pour exprimer la fonction sigmoïde : $\sigma(x)=\dfrac{1}{1+e^{-x}}$

e est la constante mathématique appelée nombre d'Euler (2.718281828459...), et nous pouvons utiliser `np.exp(x)` pour calculer e^x.

Si la sortie est supérieure à 0.5, alors nous dirons que la prédiction est 1. Si c'est en dessous de 0.5, alors nous dirons que la prédiction est 0.

```
In [12]: # Wrapping the vectors in NumPy arrays
In [13]: input_vector = np.array([1.66, 1.56])
In [14]: weights_1 = np.array([1.45, -0.66])
In [15]: bias = np.array([0.0])

In [16]: def sigmoid(x):
    ...:     return 1 / (1 + np.exp(-x))

In [17]: def make_prediction(input_vector, weights, bias):
    ...:      layer_1 = sigmoid(np.dot(input_vector, weights) + bias)
```

```
   ...:     return layer_1

In [18]: prediction = make_prediction(input_vector, weights_1, bias)

In [19]: print(f"The prediction result is: {prediction}")
Out[19]: The prediction result is: [0.7985731]
```

Le résultat brut de la prédiction est 0.79, qui est supérieur à 0.5, donc la sortie est 1. Le réseau a fait une prédiction correcte. Essayons maintenant avec un autre vecteur d'entrée, `np.array([2, 1.5])`. Le résultat correct pour cette entrée est 0. Nous n'avons qu'à changer la variable `input_vector` puisque tous les autres paramètres restent les mêmes :

```
In [20]: # Changing the value of input_vector
In [21]: input_vector = np.array([2, 1.5])

In [22]: prediction = make_prediction(input_vector, weights_1, bias)

In [23]: print(f"The prediction result is: {prediction}")
Out[23]: The prediction result is: [0.87101915]
```

Cette fois, le réseau a fait une mauvaise prédiction. Il faudrait que le résultat soit inférieur à 0.5 car la cible de cette entrée est 0, mais le résultat brut est 0.87. Il a fait une mauvaise supposition, mais quelle était la gravité de l'erreur ? La prochaine étape consiste à trouver un moyen d'évaluer cela.

Entraîner le réseau de neurones

Lors du processus de formation du réseau de neurones, on évalue d'abord l'erreur, puis on ajuste les poids en conséquence. Pour ajuster les poids, nous utiliserons les algorithmes de descente de gradient et de rétropropagation.

Calcul de l'erreur de prédiction

Pour comprendre l'ampleur de l'erreur, on doit choisir une manière de la mesurer. La fonction utilisée pour mesurer l'erreur est appelée fonction de coût **(cost function)** ou fonction de perte (**loss function**). Nous utiliserons ici l'erreur quadratique moyenne (MSE pour **Mean Squared Error**) comme fonction de coût. Elle se calcule en deux étapes simples :

1. Calculer la différence entre la prédiction et la cible.
2. Élever résultat au carré.

Le réseau peut faire une erreur en sortant une valeur supérieure ou inférieure à la valeur correcte. Étant donné que le MSE est la différence *au carré* entre la prédiction et le résultat correct, nous obtiendrons toujours une valeur positive.

Voici l'expression complète pour calculer l'erreur pour la dernière prédiction précédente :

```
In [24]: target = 0

In [25]: mse = np.square(prediction - target)

In [26]: print(f"Prediction: {prediction}; Error: {mse}")
Out[26]: Prediction: [0.87101915]; Error: [0.7586743596667225]
```

Dans l'exemple ci-dessus, l'erreur est 0.7586.

Régression linéaire

La régression est utilisée pour estimer la relation entre une variable et deux ou plusieurs autres variables indépendantes.

En modélisant la relation entre les variables comme linéaire, on peut exprimer la variable dépendante comme une somme pondérée des variables indépendantes. Ainsi, chaque variable indépendante sera multipliée par un vecteur appelé `weight`. Outre les poids et les variables indépendantes, on ajoute également un autre vecteur : le biais. Il définit le résultat lorsque toutes les autres variables indépendantes sont égales à zéro.

Comme exemple concret de construction d'un modèle de régression linéaire, imaginez que vous souhaitiez former un modèle pour prédire le prix des maisons en fonction de l'emplacement et de l'âge de la maison. Vous décidez de modéliser cette relation à l'aide d'une régression linéaire. Le bloc de code suivant montre comment vous pouvez écrire un modèle de régression linéaire pour le problème énoncé en pseudo-code :

```
price = (weights_area * area) + (weights_age * age) + bias
```

Dans l'exemple ci-dessus, il y a deux poids : `weights_area` et `weights_age`. Le processus d'apprentissage consiste à ajuster les poids et le biais afin que le modèle puisse prédire la valeur de prix correcte. Pour ce faire, vous devrez calculer l'erreur de prédiction et mettre à jour les pondérations en conséquence.

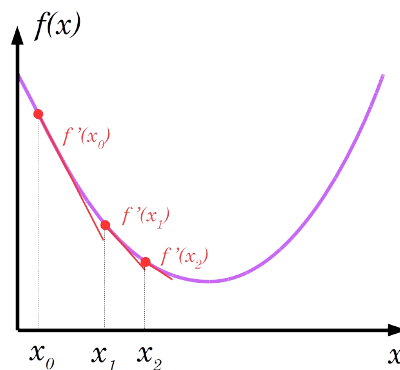
Ce sont les bases du fonctionnement du mécanisme de réseau neuronal. Il est maintenant temps de voir comment implémenter ces concepts en Python.

Comment réduire l'erreur ?

L'objectif est de modifier les pondérations et les variables de biais afin de réduire l'erreur. Pour comprendre comment cela fonctionne, nous allons un peu simplifier dans un premier temps : nous allons modifier uniquement la variable de pondération et laisser le biais fixe pour le moment ; nous pouvons également nous débarrasser de la fonction sigmoïde et n'utiliser que le résultat de `layer_1`. Il ne reste plus qu'à trouver comment modifier les pondérations pour que l'erreur diminue.

Vous calculez la MSE en faisant `error = np.square(prediction - target)`. Si nous traitons `(prediction - target)` comme une seule variable x, alors nous avons `error = np.square(x)`, qui est une fonction quadratique (une parabole). L'erreur est donnée par l'axe des ordonnées.

La descente de gradient est un algorithme d'optimisation permettant de trouver le minimum d'une fonction. Considérons la fonction dérivable $f(x)$ que l'on souhaite minimiser. L'algorithme de descente de gradient démarre à une coordonnée initiale arbitraire et converge vers le minimum de façon itérative, comme illustré ci-après. Nommons x_0 l'abscisse de départ de l'algorithme. Pour déterminer l'abscisse suivante x_1, la descente de gradient calcule la dérivée $f'(x)$, comme illustré sur cette figure :

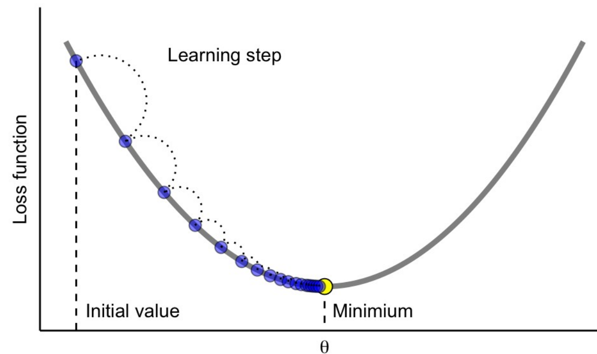


La dérivée est la pente de la tangente en ce point et nous indique dans quel sens aller. L'abscisse suivante est calculée grâce à la formule :

$$x_{n+1} = x_n + \alpha f'(x_n)$$

La dérivée de $f(x) = x^2$ est $f'(x) = 2x$, et la dérivée de x est 1.

L'expression de l'erreur est `error = np.square(prediction - target)`. Lorsque nous traitons `(prediction - target)` comme une seule variable x, la dérivée de l'erreur est $2x$. En prenant la dérivée de cette fonction, nous saurons dans quelle direction nous devons changer x pour amener le résultat de `error` vers zéro, réduisant ainsi l'erreur.



En ce qui concerne notre réseau de neurones, la dérivée nous indiquera la direction à prendre pour mettre à jour la variable de poids. Si la dérivée est positive, alors nous avons prédit trop haut et nous devons diminuer les poids. Si c'est un nombre négatif, alors nous avons prédit trop bas et nous devons augmenter les poids.

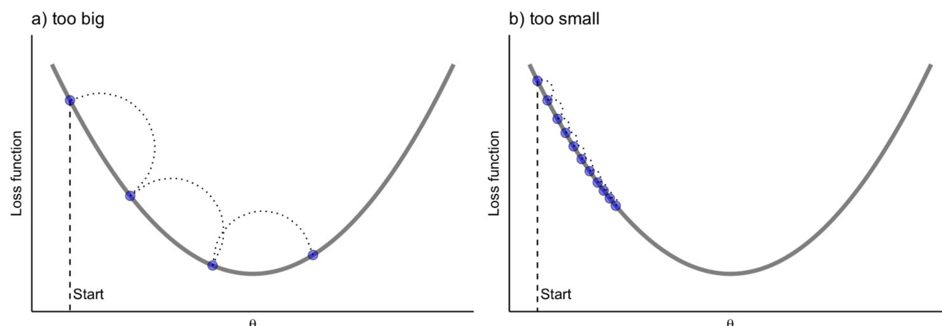
Il est maintenant temps d'écrire le code pour comprendre comment mettre à jour `weights_1` la prédiction erronée précédente. Si l'erreur quadratique moyenne est 0.75, devrions-nous augmenter ou diminuer les poids ? Puisque la dérivée est $2x$, il suffit de multiplier la différence entre la prédiction et la cible par 2 :

```
In [27]: derivative = 2 * (prediction - target)

In [28]: print(f"The derivative is {derivative}")
Out[28]: The derivative is: [1.7420383]
```

Le résultat est 1.74, un nombre positif. Nous devons donc diminuer les poids. Nous faisons cela en soustrayant le résultat dérivé du vecteur de poids. Nous pouvons maintenant mettre à jour `weights_1` en conséquence et refaire une prédiction pour voir comment cela affecte le résultat :

```
In [29]: # Updating the weights
In [30]: weights_1 = weights_1 - derivative

In [31]: prediction = make_prediction(input_vector, weights_1, bias)

In [32]: error = (prediction - target) ** 2

In [33]: print(f"Prediction: {prediction}; Error: {error}")
Out[33]: Prediction: [0.01496248]; Error: [0.00022388]
```

L'erreur est tombée à presque 0. Dans cet exemple, le résultat dérivé était petit, mais il y a des cas où le résultat dérivé est trop élevé. Prenons l'image de la fonction quadratique comme exemple. Des incréments élevés ne sont pas idéaux car vous pouvez continuer à aller d'un point à un autre sans jamais vous approcher de zéro. Pour remédier à cela, on mettra à jour les poids avec une fraction du résultat dérivé en ajoutant un paramètre α, également appelé taux d'apprentissage. Si on diminue le taux d'apprentissage, les pas seront plus petits. Si on l'augmente, les pas seront plus grands :



Comment savoir quelle est la meilleure valeur de taux d'apprentissage ? En essayant…

Remarque : les valeurs de taux d'apprentissage par défaut traditionnelles sont 0.1, 0.01et 0.001.

Si on prend les nouveaux poids et qu'on refait une prédiction avec le premier vecteur d'entrée, on verra que maintenant il fait une mauvaise prédiction pour celui-ci. Si le réseau de neurones fait une prédiction correcte pour chaque instance de l'ensemble d'apprentissage, on a probablement un modèle surajusté, où le modèle se souvient simplement comment classer les exemples au lieu d'apprendre à remarquer les caractéristiques dans les données.

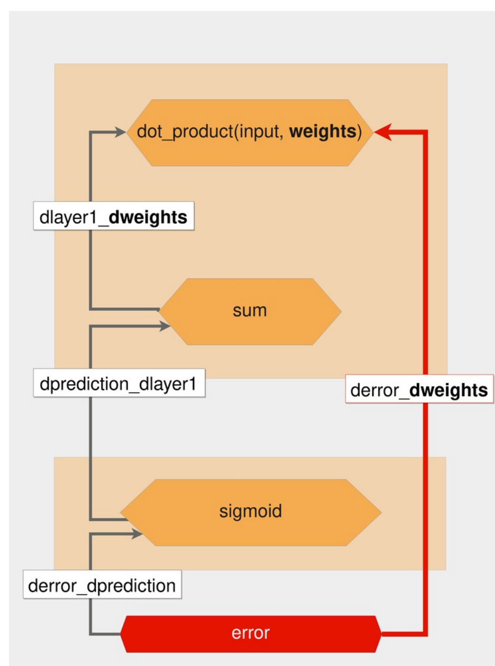
Maintenant que nous savons comment calculer l'erreur et comment ajuster les poids en conséquence, il est temps de reprendre la construction de votre réseau de neurones.

En fait, c'est un peu plus compliqué...

Dans notre réseau de neurones, nous devons mettre à jour à la fois les poids et les vecteurs de biais. La fonction que nous utilisons pour mesurer l'erreur dépend de deux variables indépendantes, les poids et le biais. Étant donné que les poids et le biais sont des variables indépendantes, nous pouvons les modifier et les ajuster pour obtenir le résultat souhaité.

Comme nous appliquons la fonction sigmoïde à la sortie du neurone, nous avons affaire à une composition de fonctions. Cela signifie que la fonction d'erreur est toujours `np.square(x)`, mais que x est maintenant le résultat d'une autre fonction. Attention donc à la dérivée de l'intérieur !

Voici une représentation visuelle de la façon dont nous appliquons la règle pour la dérivée d'une composition, afin de trouver la dérivée de l'erreur par rapport aux poids :



La flèche rouge en gras montre la dérivée que nous voulons : `derror_dweights`.

Dans l'image ci-dessus, chaque fonction est représentée par les hexagones orange et les dérivées partielles sont représentées par les flèches grises à gauche. En appliquant la règle de dérivation d'une fonction composée (**chain rule** en anglais), la valeur de `derror_dweights` sera la suivante :

```
derror_dweights = (
    derror_dprediction * dprediction_dlayer1 * dlayer1_dweights
)
```

Pour calculer la dérivée, on multiplie toutes les dérivées partielles qui suivent le chemin allant de l'hexagone d'erreur (le rouge) à l'hexagone tout en haut.

Ce chemin inverse s'appelle une passe arrière (**backward pass**). À chaque passe arrière, on

calcule les dérivées partielles de chaque fonction, on substitue les variables par leurs valeurs et enfin on multiplie le tout.

Cet algorithme de mise à jour des paramètres du réseau neuronal est appelé rétropropagation **(backpropagation)**.

Réglage des paramètres avec rétropropagation

Dans cette section, on va analyser le processus de rétropropagation étape par étape, en commençant par la façon dont on met à jour le biais. On veut prendre la dérivée de la fonction d'erreur par rapport au biais, `derror_dbias`. Ensuite, on continuera à revenir en arrière, en prenant les dérivées partielles jusqu'à ce qu'on trouve la variable `bias`. C'est la **chain rule**...

Puisque l'on part de la fin et que l'on revient en arrière, on doit d'abord prendre la dérivée partielle de l'erreur par rapport à la prédiction. C'est `derror_dprediction` dans l'image ci-après.

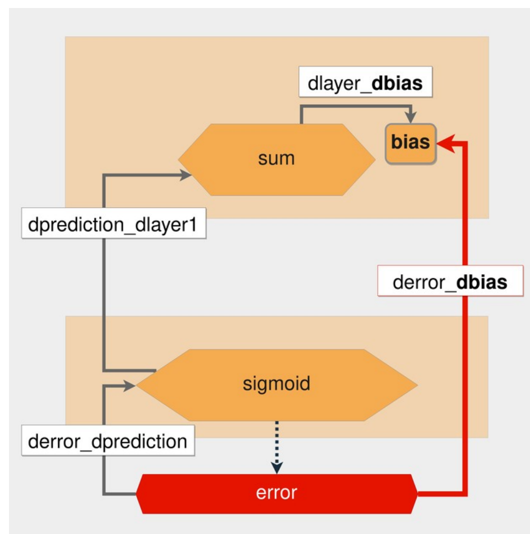
La fonction qui produit l'erreur est une fonction quadratique, et la dérivée de cette fonction est $2x$, comme nous l'avons vu précédemment. Nous avons appliqué la première dérivée partielle (`derror_dprediction`) mais n'avons toujours pas atteint le biais. Nous devons donc encore prendre la dérivée de la prédiction par rapport à la couche précédente, `dprediction_dlayer1`.

La prédiction est le résultat de la fonction sigmoïde.

La dérivée de la fonction sigmoïde σ est : $\sigma'(x)=\frac{e^{-x}}{(1+e^{-x})^2}$

On peut vérifier que l'on peut aussi écrire cette dérivée ainsi : $\sigma'(x) = \sigma(x)\cdot(1-\sigma(x))$.

Cette formule est très pratique car on peut utiliser le résultat sigmoïde qui a déjà été calculé pour calculer sa dérivée. On prend ensuite cette dérivée partielle et on continue à reculer.



Nous allons maintenant prendre la dérivée de `layer_1` par rapport au biais. La variable `bias` est un nombre, donc la dérivée vaut 1. Maintenant que nous avons terminé cette passe arrière, nous pouvons assembler le tout et calculer `derror_dbias` :

```
In [36]: def sigmoid_deriv(x):
    ...:     return sigmoid(x) * (1-sigmoid(x))

In [37]: derror_dprediction = 2 * (prediction - target)
In [38]: layer_1 = np.dot(input_vector, weights_1) + bias
In [39]: dprediction_dlayer1 = sigmoid_deriv(layer_1)
In [40]: dlayer1_dbias = 1

In [41]: derror_dbias = (
    ...:     derror_dprediction * dprediction_dlayer1 * dlayer1_dbias
    ...: )
```

Pour mettre à jour les poids, on suit le même processus, en remontant et en prenant les dérivées partielles jusqu'à ce que l'on arrive à la variable de poids. Puisque nous avons déjà calculé certaines des dérivées partielles, nous n'aurons qu'à calculer `dlayer1_dweights`. La dérivée du produit scalaire est la dérivée du premier vecteur multipliée par le deuxième vecteur, plus la dérivée du deuxième vecteur multipliée par le premier vecteur.

Création de la classe de réseau neuronal

Nous savons maintenant comment écrire les expressions pour mettre à jour à la fois les poids et le biais.

Il est temps de créer une classe Python pour le réseau de neurones. La classe `NeuralNetwork` génère des valeurs de départ aléatoires pour les pondérations et les variables de biais.

Lors de l'instanciation d'un objet `NeuralNetwork`, on doit passer le paramètre `learning_rate`. On utilisera `predict()` pour faire une prédiction, et les méthodes `_compute_derivatives()` et `_update_parameters()` pour faire les calculs que nous avons vus dans cette section.

Voici la classe `NeuralNetwork` finale :

```
class NeuralNetwork:
    def __init__(self, learning_rate):
        self.weights = np.array([np.random.randn(), np.random.randn()])
        self.bias = np.random.randn()
        self.learning_rate = learning_rate

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def _sigmoid_deriv(self, x):
        return self._sigmoid(x) * (1 - self._sigmoid(x))

    def predict(self, input_vector):
        layer_1 = np.dot(input_vector, self.weights) + self.bias
        prediction = self._sigmoid(layer_1)
        return prediction

    def _compute_gradients(self, input_vector, target):
        layer_1 = np.dot(input_vector, self.weights) + self.bias
        prediction = self._sigmoid(layer_1)

        derror_dprediction = 2 * (prediction - target)
        dprediction_dlayer1 = self._sigmoid_deriv(layer_1)
        dlayer1_dbias = 1
        dlayer1_dweights = (0 * self.weights) + (1 * input_vector)

        derror_dbias = (
            derror_dprediction * dprediction_dlayer1 * dlayer1_dbias
        )
        derror_dweights = (
            derror_dprediction * dprediction_dlayer1 * dlayer1_dweights
        )

        return derror_dbias, derror_dweights

    def _update_parameters(self, derror_dbias, derror_dweights):
        self.bias = self.bias - (derror_dbias * self.learning_rate)
        self.weights = self.weights - (
            derror_dweights * self.learning_rate
        )
```

Voilà ! C'est le code de notre premier réseau de neurones, qui rassemble toutes les pièces que nous avons vues jusqu'à présent.

Si on veut faire une prédiction, on crée d'abord une instance de `NeuralNetwork()`, puis on appelle `.predict()` :

```
In [42]: learning_rate = 0.1
```

```
In [43]: neural_network = NeuralNetwork(learning_rate)

In [44]: neural_network.predict(input_vector)
Out[44]: array([0.79412963])
```

Entraîner le réseau avec plus de données

Nous avons déjà ajusté les pondérations et le biais pour **une** instance de données, mais l'objectif est de généraliser le réseau sur **un ensemble entier** de données. La descente de gradient stochastique est une technique dans laquelle, à chaque itération, le modèle fait une prédiction basée sur une donnée d'apprentissage sélectionnée au hasard, calcule l'erreur et met à jour les paramètres.

Il est maintenant temps de créer la méthode `train()` de votre classe `NeuralNetwork`. On enregistrera l'erreur sur tous les points de données toutes les 100 itérations, car on va tracer un graphique montrant comment cette métrique change à mesure que le nombre d'itérations augmente. Voici la dernière méthode `train()` de notre réseau de neurones :

```
 1 class NeuralNetwork:
 2     # ...
 3
 4     def train(self, input_vectors, targets, iterations):
 5         cumulative_errors = []
 6         for current_iteration in range(iterations):
 7             # Pick a data instance at random
 8             random_data_index = np.random.randint(len(input_vectors))
 9
10             input_vector = input_vectors[random_data_index]
11             target = targets[random_data_index]
12
13             # Compute the gradients and update the weights
14             derror_dbias, derror_dweights = self._compute_gradients(
15                 input_vector, target
16             )
17
18             self._update_parameters(derror_dbias, derror_dweights)
19
20             # Measure the cumulative error for all the instances
21             if current_iteration % 100 == 0:
22                 cumulative_error = 0
23                 # Loop through all the instances to measure the error
24                 for data_instance_index in range(len(input_vectors)):
25                     data_point = input_vectors[data_instance_index]
26                     target = targets[data_instance_index]
27
28                     prediction = self.predict(data_point)
29                     error = np.square(prediction - target)
30
31                     cumulative_error = cumulative_error + error
32                 cumulative_errors.append(cumulative_error)
33
34         return cumulative_errors
```

Il se passe beaucoup de choses dans le code ci-dessus. Voici donc une explication ligne par ligne :

- **La ligne 8** sélectionne une instance aléatoire dans l'ensemble de données.
- **Les lignes 14 à 16** calculent les dérivées partielles et renvoient les dérivées pour le biais et les poids. Ils utilisent `_compute_gradients()`, que vous avez défini précédemment.
- **La ligne 18** met à jour le biais et les pondérations à l'aide de `_update_parameters()`, que nous avons définis dans le bloc de code précédent.
- **La ligne 21** vérifie si l'index d'itération courant est un multiple de 100. Vous faites cela pour observer comment l'erreur change toutes les 100 itérations.
- **La ligne 24** démarre la boucle qui passe par toutes les instances de données.
- **La ligne 28** calcule le résultat `prediction`.
- **La ligne 29** calcule `error` pour chaque instance.

- **La ligne 31** est l'endroit où on accumule la somme des erreurs à l'aide de la variable `cumulative_error`. On fait cela parce que l'on veut tracer un point avec l'erreur pour *toutes* les instances de données. Ensuite, à la ligne 32, on ajoute `error` à `cumulative_errors`, le tableau qui stocke les erreurs. On utilisera ce tableau pour tracer le graphique.

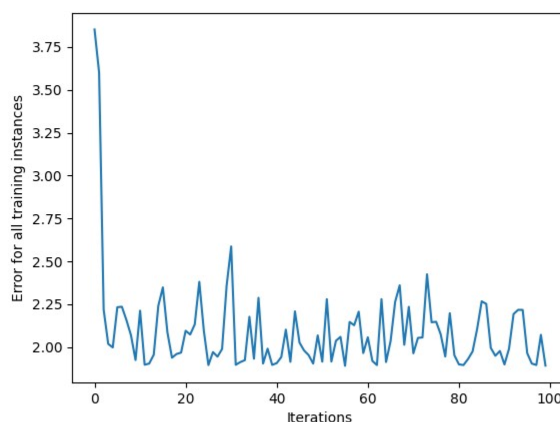
En bref, on choisit une instance aléatoire dans l'ensemble de données, calcule les gradients et met à jour les poids et le biais. On calcule également l'erreur cumulée toutes les 100 itérations et enregistre ces résultats dans un tableau. Nous allons tracer un graphique pour visualiser l'évolution de l'erreur au cours du processus de formation.

Pour simplifier les choses, on utilisera un ensemble de données avec seulement huit instances, le tableau `input_vectors`. on peut maintenant appeler `train()` et utiliser le module **Matplotlib** pour tracer l'erreur cumulée pour chaque itération :

rna2.py

```
In [45]: # Paste the NeuralNetwork class code here
    ...: # (and don't forget to add the train method to the class)

In [46]: import matplotlib.pyplot as plt

In [47]: input_vectors = np.array(
    ...:     [
    ...:         [3, 1.5],
    ...:         [2, 1],
    ...:         [4, 1.5],
    ...:         [3, 4],
    ...:         [3.5, 0.5],
    ...:         [2, 0.5],
    ...:         [5.5, 1],
    ...:         [1, 1],
    ...:     ]
    ...: )

In [48]: targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])

In [49]: learning_rate = 0.1

In [50]: neural_network = NeuralNetwork(learning_rate)

In [51]: training_error = neural_network.train(input_vectors, targets, 10000)

In [52]: plt.plot(training_error)
In [53]: plt.xlabel("Iterations")
In [54]: plt.ylabel("Error for all training instances")
In [54]: plt.savefig("cumulative_error.png")
```

Voici le graphique montrant l'erreur pour une instance d'un réseau de neurones :



L'erreur globale diminue, c'est ce qu'on veut. Après la plus forte diminution, l'erreur continue de

monter et descendre rapidement d'une itération à l'autre. C'est parce que l'ensemble de données est aléatoire et très petit, il est donc difficile pour le réseau de neurones d'extraire des caractéristiques.

Ajouter plus de couches au réseau de neurones

L'ensemble de données de ce didacticiel a été réduit à des fins d'apprentissage. Habituellement, les modèles d'apprentissage en profondeur nécessitent une grande quantité de données car les ensembles de données sont plus complexes et ont beaucoup de nuances.

Étant donné que ces ensembles de données contiennent des informations plus complexes, l'utilisation d'une ou deux couches seulement n'est pas suffisante. C'est pourquoi les modèles d'apprentissage en profondeur sont appelés *profonds* (deep). Ils ont généralement un grand nombre de couches.

En ajoutant plus de couches et en utilisant des fonctions d'activation, vous augmentez la puissance expressive du réseau et pouvez faire des prédictions de très haut niveau. Un exemple de ces types de prédictions est la reconnaissance faciale, par exemple lorsque vous prenez une photo de votre visage avec votre téléphone et que le téléphone se déverrouille s'il vous reconnaît.

Le processus d'entraînement d'un réseau de neurones consiste principalement à appliquer des opérations à des vecteurs. Ici, nous l'avons fait à partir de zéro en utilisant uniquement **NumPy**. Ceci n'est pas recommandé dans un environnement de production car l'ensemble du processus peut être sujet aux erreurs. C'est l'une des raisons pour lesquelles les modules **Keras**, **PyTorch** et **TensorFlow** sont si populaires pour faire du deep learning.

La meilleure distribution Python pour faire du machine learning est **Anaconda**. Vous trouverez comment l'installer sur la vidéo :

https://www.youtube.com/watch?v=jaw5FhWx2Bk

Exercice 11.10 - Réseau de neurones avec Thymio

Source de l'exercice

Elements of Robotics
Ben-Ari, Mondada
Springer Open, 2018
chapitre 13

Après avoir branché votre Thymio à votre ordinateur et l'avoir allumé, lancez « Thymio suite » et choisissez « Aseba studio ».

Dans Aseba allez ensuite dans le menu « Fichier>Ouvrir… » et ouvrez le fichier « RF-4_ann-structure.aesl » se trouvant sur le site web compagnon.

Chargez puis exécutez le programme.

Le programme a été conçu pour que Thymio réagisse en utilisant le réseau de neurones défini sur l'image ci-contre une fois que le bouton central a été appuyé. Les poids des neurones ont été définis pour que le robot avance et évite les obstacles.

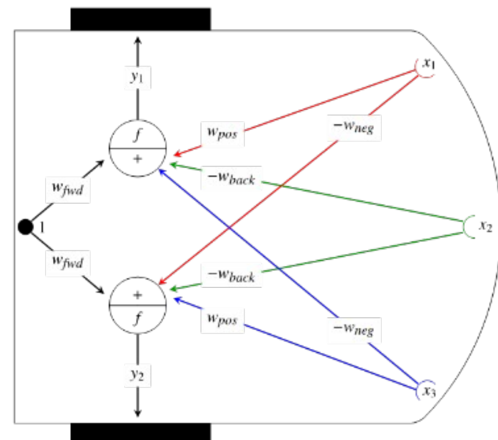


Fig. 13.4 Neural network for obstacle avoidance

Prenez le code en main, familiarisez-vous avec le réseau de neurones défini dans le programme, ainsi qu'avec ses poids, puis modifiez le programme pour que Thymio soit « agressif » : s'il ne perçoit pas d'obstacle il ne bouge pas, sinon il fonce dedans.

Sources

[1] Wikipédia, « Q-learning », <https://fr.wikipedia.org/wiki/Q-learning>

[2] Thibault Neveu, « Apprentissage par renforcement #5 : Introduction au Q-Learning », <https://youtu.be/a0bVIyIJ074>

[3] Data mutation, <http://datamutation.net/que-peut-on-apprendre-des-reseaux-de-neurones/>

[4] Jean-Claude Heudin , *Comprendre le deep learning – Une introduction aux réseaux de neurones*, Science.eBook, 2016

[5] Déborah Mesquita, « Python AI: How to Build a Neural Network & Make Predictions », <https://realpython.com/python-ai-neural-network/>