Table des matières

Déclaration i
Remerciements ii
Liste des figures v
1. Introduction 1
2. Technologies utilisées 2
2.1 PyTorch 2
2.1.1 Explication du choix 2
2.2 OpenAI Gym 3
2.2.1 Explication du choix 3
3. Machine learning 4
3.1 Introduction 4
3.2 Réseaux de neurones artificiels 5
3.2.1 Introduction 5
3.2.2 Structure 6
3.2.3 Phase d'apprentissage 9
3.2.3.1 L'algorithme de descente du gradient 11
3.2.3.2 Calcul du gradient sur régression linéaire 12
3.2.3.2.1 Régression linéaire univariée 12
3.2.3.2.2 Explication mathématique 13
3.2.3.2.3 Implémentation Python 16
3.2.3.3 Calcul du gradient dans un réseau de neurones artificiels manuellement 17
3.2.3.4 Calcul du gradient dans un réseau de neurone avec PyTorch 20
3.3 Apprentissage par renforcement 23
3.3.1 Eléments du paradigme 24
3.3.2 Fonctionnement 25
3.3.3 Tabular Q-learning 27
3.3.3.1 Implémentation 28
3.3.4 DQN 30
3.3.4.1 Concepts 31
3.3.4.2 Fonctionnement 33
3.3.4.3 Implémentation 34
4. Conclusion 37
5. Bibliographie 38

Liste des figures

Figure 1 : Graphique comparatif PyTorch Tensorflow..2
Figure 2 : Structure d'un neurone..5
Figure 3 : Structure d'un neurone artificiel 5
Figure 4 : Introduction aux réseaux de neurones...6
Figure 5 : Structure d'un neurone..7
Figure 6 : Fonction d'activation ReLU..8
Figure 7 : Fonction d'activation sigmoïde 8
Figure 8 : Formule MSELoss..9
Figure 9 : Formule Cross Entropy Loss 9
Figure 10 : Formule Log Loss...10
Figure 11 : Introduction à l'algorithme de la descente du gradient 11
Figure 12 : Les différents taux d'apprentissage 12
Figure 13 : Exemple de régression linéaire 13
Figure 14 : Code algorithme du gradient sur régression 16
Figure 15 : Graphique d'évolution de la droite 17
Figure 16 : Code algorithme du gradient manuellement partie 1..........................17
Figure 17 : Code algorithme du gradient manuellement parties 2........................18
Figure 18 : Code algorithme du gradient avec PyTorch partie 1 21
Figure 19 : Code algorithme du gradient avec PyTorch partie 2 22
Figure 20 : Introduction à l'apprentissage par renforcement 23
Figure 21 : L'interaction entre l'agent et l'environnement dans le cadre MDP.............25
Figure 22 : Image de l'environnement MountainCar-v0 28
Figure 23 : Implémentation de l'algorithme tabular q learning partie 1 28
Figure 24 : Implémentation de l'algorithme tabular q learning parties 2 29
Figure 25 : Différence entre le q learning et le deep q learning............................31
Figure 26 : Fonctionnement des deux réseaux de neurones 32
Figure 27 : Fonctionnement d'un algorithme de DQN...33
Figure 27 : Image de l'environnement CartPole-v0..34
Figure 28 : Code du ReplayBuffer du DQN CartPole...35
Figure 29 : Code de la fonction act du DQN CartPole 35
Figure 30 : Code de la fonction train du DQN CartPole 36

1. Introduction

L'objectif à long-terme de l'intelligence artificielle est de résoudre des tâches avancées de la vie réelle. Les jeux-vidéos, depuis des dizaines d'années, servent de marche pour atteindre cet objectif, par le Backgammon (1992), aux Echecs (1997) puis Atari (2013) où l'Agent57, une intelligence artificielle développée par DeepMind, a appris à jouer à 57 jeux (Will Douglas Heaven, 2020). En 2016, AlphaGo, une IA également développée par DeepMind, remporte une partie contre le champion du monde de Go, un jeu de société dans lequel il existe un nombre considérable de combinaisons (10^{600} parties plausibles contre 10^{40} aux échecs (Wikipédia, 2020)), utilisant du *deep reinforcement learning* (apprentissage par renforcement profond) (OpenAI, 2019). Un ordinateur dont les capacités de réflexion dans un domaine complexe précis sont plus performantes que le meilleur des joueurs, comment ne pas être intéressé par le sujet ?

Afin de nous introduire dans la compréhension de ce travail de recherche, nous allons, dans un premier temps, faire un historique sur l'apprentissage automatique. Ensuite, nous verrons la structure et le fonctionnement d'un réseau de neurones. Puis, nous aborderons une partie technique sur l'entrainement d'un réseau de neurones.

Dans un second temps, nous parlerons de l'apprentissage par renforcement, comprendre ce qui l'entourent ainsi que son fonctionnement. Après cette partie théorique, on approchera le sujet dans une partie technique où l'on expliquera l'implémentation d'algorithme propre à l'apprentissage par renforcement.

A noter que ce travail de Bachelor est un projet de recherche et, par conséquent, aucun prototype ne sera délivré ou déployé à la fin celui-ci. Cependant, des parties techniques seront développées et expliquées pour la bonne compréhension du sujet.

2. Technologies utilisées

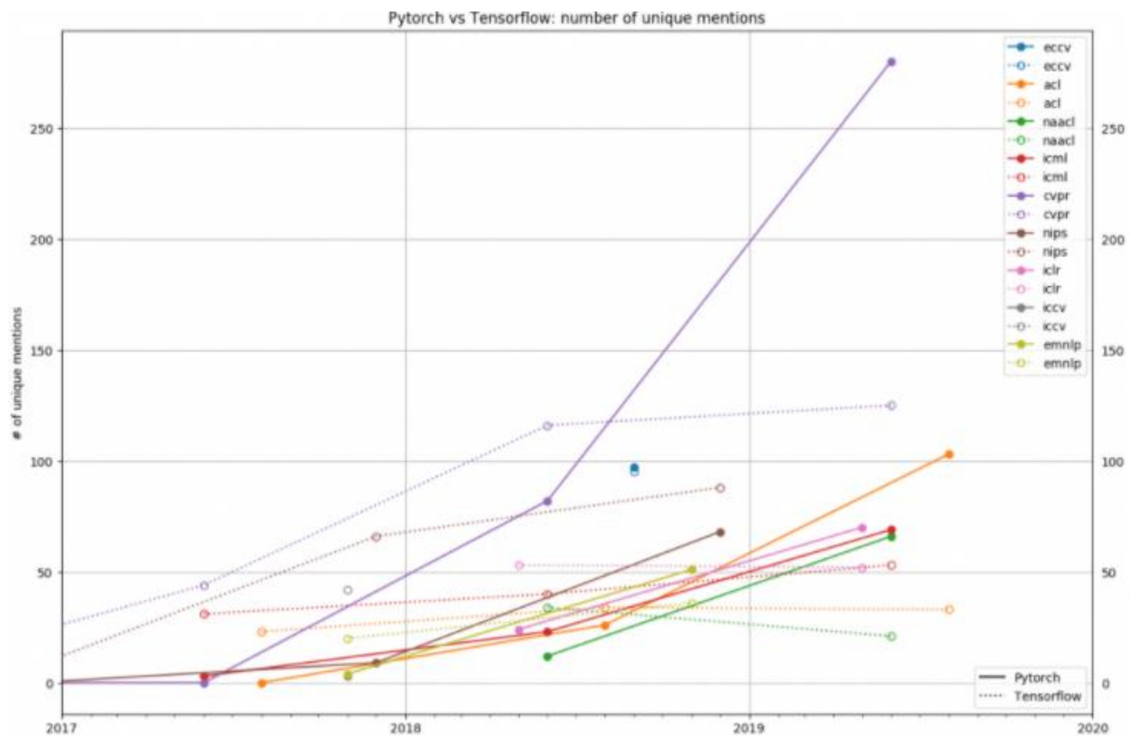
2.1 PyTorch

PyTorch est une libraire open source python d’apprentissage automatique profond développé par l’équipe de recherche sur l’intelligence artificiel de Facebook (Gillian Jakob Kieser, 2019). Cette libraire est un dérivé du logiciel Torch qui s’utilisait avec le langage Lua (Wikipédia, 2020). L’équipe PyTorch est composée d’environ 100 membres à l’interne comme à l’externe de Facebook en plus des 1000 contributeurs de la solution open source (Gillian Jakob Kieser, 2019). Aujourd’hui, des organisations tels que Standford ou Salesforce utilisent PyTorch pour alimenter leurs recherches sur l’apprentissage automatique (Gillian Jakob Kieser, 2019).

2.1.1 Explication du choix

The Gradient, un forum de publication sur l’intelligence artificielle, a publié un graphique montrant le nombre de thèse ayant mentionné PyTorch ou Tensorflow sur les trois dernières années.

Figure 1 : Graphique comparatif PyTorch Tensorflow



(Horace He, 2020)

Comme on peut le constater grâce à la Figure 1 : Graphique comparatif PyTorch Tensorflow, en 2018, PyTorch se trouvait clairement en minorité face à Tensorflow mais son utilisation à exploser en 2019.

Aujourd'hui, la plupart des chercheurs ont tendances à recommander cette libraire pour leurs recherches (Exxact Corporation, 2020). Le fait que PyTorch soit natif de python et qu'il s'intègre aisément avec les autres librairies du langage facilitent le choix des scientifiques. De plus, plusieurs d'entre eux affirment que l'interface de PyTorch est intuitive et plus facile à apprendre que Tensorflow (Exxact Corporation, 2020).

Les raisons pour lesquelles PyTorch a été choisi pour la réalisation de ce travail sont, tout d'abord liées à l'objectif de ce projet qui est de faire des recherches en rapport avec l'apprentissage automatique. En effet, PyTorch est la librairie la plus adaptée pour cela, tandis que TensorFlow serait plus adaptée pour une mise en production (Stephen Welch, 2020). Ajouté à cela, les tutoriels et documentations officiels PyTorch qui sont très complets et utiles pour la compréhension du sujet. Enfin, le directeur de mémoire de ce travail a fourni des exercices basés sur la librairie PyTorch confirmant le choix de cette librairie.

2.2 OpenAI Gym

OpenAI est une compagnie non lucrative de recherche sur l'intelligence artificiel créé par Elon Musk et Sam Altman (Ashish Rana, 2018). L'entreprise a développé Gym, une « boîte à outils » open source pour le développement et la comparaison d'algorithme d'apprentissage par renforcement. En effet, il s'agit d'une interface qui fournit toutes les informations nécessaires au bot afin qu'il puisse apprendre et évoluer. (Ashish Rana, 2018).

2.2.1 Explication du choix

En 2016, un sondage indiquait que plus de 70 % des chercheurs ont essayé et échoué de reproduire l'expérience d'autres scientifiques et, plus de la moitié ont échoués dans la reproduction de leurs propres expériences (Ashish Rana, 2018). OpenAI a été créé dans le but de résoudre le problème du manque de normalisation dans les thèses des chercheurs avec pour objectif de créer de meilleurs repères en donnant un nombre polyvalent d'environnement avec une grande facilité de mise en place. Le but principal de cet outil est d'augmenter la reproductibilité dans le domaine de l'apprentissage par renforcement et de fournir des outils avec lesquels tout le monde peut en apprendre les bases (Ashish Rana, 2018). De plus, le principal objectif derrière ce travail de recherche est l'apprentissage par renforcement. La création d'un environnement qui permette de reproduire ce que Gym offre ne serait pas adapté pour un travail de Bachelor.

3. Machine learning

3.1 Introduction

La différence entre l'analyse de données et l'apprentissage automatique est, dans un premier temps, dans l'approche. En effet, dans le domaine du *machine learning*, nous utilisons des algorithmes capables d'extraire automatiquement des patterns dans les données dans le but de résoudre des problèmes potentiellement complexes. A l'instar d'une analyse de données qui utilise des méthodes de statistiques classiques comme des estimateurs ou des analyses de corrélations (Yannis Chaouch, 2018) ce qui serait chronophage à exécuter manuellement.

Le *machine learning* est le fait de laisser un ordinateur apprendre et agir seul comme un humain pour que, avec le temps, il s'améliore. L'apprentissage est calculé avec les données et informations qui sont données à l'ordinateur sous formes d'observations ou d'interactions réelle (Daniel Faggella, 2020). Ce domaine de recherche tend à répondre à la question : Comment puis-je construire un système qui, automatiquement, s'améliore avec l'expérience et quels sont les lois appliquées aux processus d'apprentissage (Tom M. Mitchell, 2006) ?

La première grande distinction à faire en *machine learning* est la différence entre l'apprentissage supervisé et non supervisé. Dans le cas de l'apprentissage supervisé, le but est d'entrainer le système en lui fournissant une entrée et la réponse attendue aidant la machine dans ses prédictions. Par exemple, la classification d'image, où l'on entraine un algorithme avec des entrées dites annotées, c'est-à-dire que le système connait directement la catégorie de l'image. Le but étant de trouver les similitudes entre une image non annotées et les images utilisées pour l'entrainement afin d'en tirer une prédiction (Yannis Chaouch, 2018). A l'inverse, dans le cas du non supervisé, le système ne connait pas la réponse attendue et cherche des similarités et distinctions dans l'ensemble de données. Ce projet de recherche se concentrera sur une méthode non supervisée, l'apprentissage par renforcement. Ce modèle fonctionne avec des récompenses. Lorsque l'ordinateur améliore sa performance il se voit attribué d'une récompense favorisant l'action utilisée pour la situation dans laquelle il se trouvait (Yannis Chaouch, 2018). Prenons par exemple, les jeux vidéo, où l'on doit s'améliorer à mesure que l'on échoue. Le but sera d'entrainer le système en maximisant la performance grâce aux récompenses acquises au fur et à mesure.

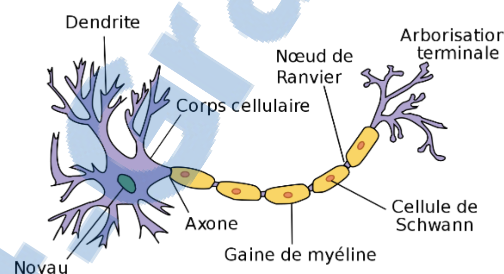
L'apprentissage par renforcement sera la seule méthode expliquée et analysée dans ce travail de bachelor. En effet, le but de celui-ci étant d'acquérir les bases à la compréhension du *machine learning* au travers de l'apprentissage par renforcement. Avant cela, il faut comprendre un concept important qui permettra de mieux appréhender l'apprentissage par renforcement, les réseaux de neurones artificiels.

3.2 Réseaux de neurones artificiels

3.2.1 Introduction

L'origine des réseaux de neurones artificiels remonte à 1957, avec l'invention du perceptron par Frank Rosenblatt : un réseau de neurone qui prend un modèle sur la structure d'une cellule nerveuse, en tenant compte des dendrites qui sont des prolongements du neurone à travers lesquels passent des informations sous forme d'électrochimique (Terence Tse, 2017), voir la Figure 2 : Structure d'un neurone.

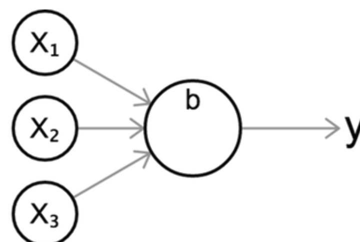
Figure 2 : Structure d'un neurone



(Wikipédia, 2020)

En effet, on peut observer par la structure du perceptron, qui ne contient qu'un seul neurone mathématique, qu'il a été réfléchi pour reproduire le comportement d'un neurone biologique. (Ada Lovelace, 1843), voir la Figure 3 : Structure d'un neurone artificiel.

Figure 3 : Structure d'un neurone artificiel



(Tibor Udvari, 2019)

Rapport-Gratuit.com

Dans les Figure 2 : Structure d'un neurone et Figure 3 : Structure d'un neurone artificiel certains remarqueront des similitudes où les dendrites captent l'informations (pour le neurone artificiel sont les x1, x2 et x3), puis le corps cellulaire traite l'information qu'il reçoit (le b pour le neurone artificiel) et l'information traitée est communiquée.

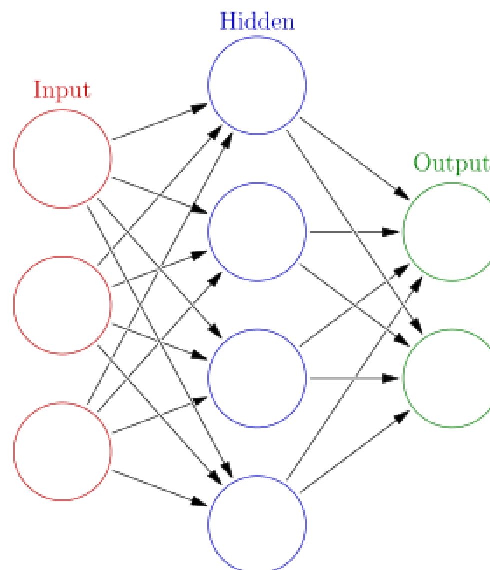
Aujourd'hui, la plupart des scientifiques déconseillent de prendre cette analogie trop au sérieux, car les réseaux neuronaux sont strictement conçus pour résoudre les problèmes d'apprentissage machine, plutôt que de représenter le cerveau précisément (Tibor Udvari, 2019).

Pour synthétiser, les réseaux de neurones artificiels, sont des structures qui permettent de traiter l'information. Et s'ils portent ce nom, c'est parce qu'il s'agit d'algorithme, d'instructions machine dont la structure rappel celle d'un cortex cérébral (Terence Tse, 2017).

3.2.2 Structure

Pour bien comprendre le fonctionnement d'un réseau de neurones, il faut comprendre sa structure :

Figure 4 : Introduction aux réseaux de neurones

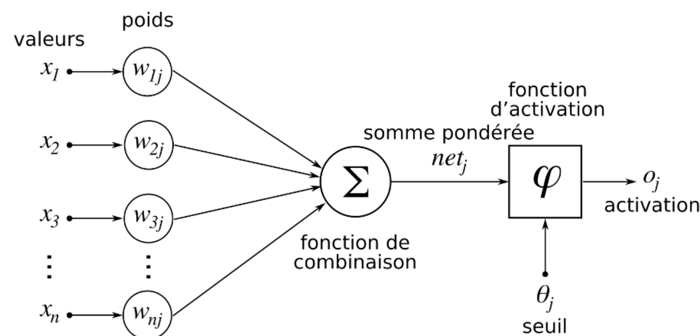


(Wikipédia, 2020)

La Figure 4 : Introduction aux réseaux de neurones ci-dessus représente la forme que peut prendre un réseau de neurones dont certains pourraient, fondamentalement, qualifier de simple. La couche rouge, appelée *input*, représente les entrées du réseau. Les entrées sont, par exemple, les valeurs des pixels d'une image. Dans la couche

bleue, appelée *hidden*, ce sont les neurones directement liés aux neurones de sorties. Cette couche permettra, en se basant sur les couches précédentes, d'avoir un résultat spécifique. Ce résultat peut correspondre à la classification d'une image, par exemple. Si le premier neurone a une valeur plus élevée que le deuxième, alors l'image en entrée est un chien ou un chat. Enfin, la couche verte, appelée *output*, ce sont les sorties. La sortie ayant la plus grande valeur sera la prédiction du modèle (Harrison, 2019). Pour comprendre l'entrainement effectué dans la couche *hidden*, il faut comprendre le fonctionnement d'un neurone.

Figure 5 : Structure d'un neurone



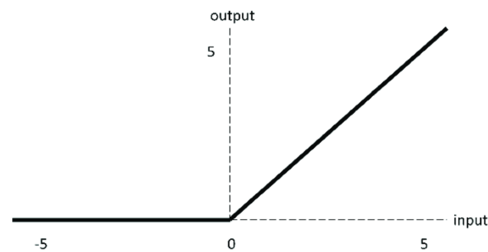
(Wikipédia, 2020)

En se référant à la Figure 5 : Structure d'un neurone, imaginons que nous recevons les valeurs des pixels d'une image en entrée et que le but est de prédire s'il s'agit d'un chien ou un chat. Nous envoyons les entrées (valeurs -> x_n sur la figure) dans le neurone qui va multiplier toutes les valeurs par des poids, puis les sommer ensemble (fonction de combinaison). Ensuite la valeur va passer dans une fonction d'activation pour produire de nouvelles valeurs dites normalisées qui seront comparées à un seuil, qui définira la connexion avec le prochain neurone. Si le résultat est supérieur au seuil, alors le traitement renforce la connexion entre le neurone courant et les suivants, sinon, la connexion ne se fait pas. Ce processus se répète autant de fois que le nombre de "couches" dans un réseau neuronal artificiel pour finalement avoir une sortie qui sera la prédiction. (Harrison, 2019). Ce traitement fonctionne par propagation des entrées jusqu'aux sorties.

Etant donné que les données utilisées peuvent varier, il est probable qu'il existe des problèmes de confiance avec les résultats obtenus. En effet, si la première entrée a des valeurs entre 0 et 1 et la deuxième entre 10 et 20, alors l'impact de la deuxième sera beaucoup plus important sans fondement. C'est pour cela que les valeurs d'entrées sont travaillées puis, la valeur est passée dans une fonction d'activation.

Il existe plusieurs fonctions d'activation mais dans le cadre de ce travail, nous allons parler de *rectified linear unit* (ReLU) et de sigmoïde.

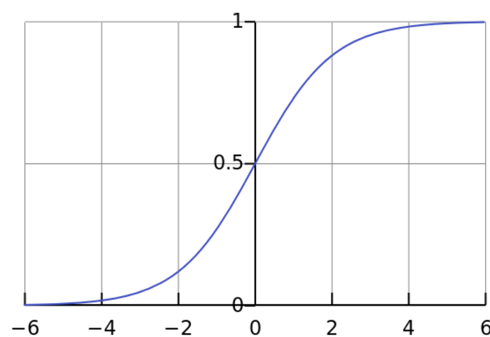
Figure 6 : Fonction d'activation ReLU



(Hossam H. Sultan, 2019)

La fonction d'activation ReLU, illustrée dans la Figure 6 : Fonction d'activation ReLU, est linéaire, pour toutes les valeurs positives et zéro pour toutes les valeurs négatives. Il s'agit d'une fonction qui n'a pas besoin de calculs et, par conséquent, sa complexité computationnelle est basse. Il est possible que la fonction soit peu utilisée ce qui augmentera la vitesse d'entrainement du modèle. Étant donné que la fonction ReLU est nulle pour toutes les entrées négatives, il est probable qu'une unité donnée ne s'active pas du tout (Sonish Sivarajkumar, 2019).

Figure 7 : Fonction d'activation sigmoïde



(Avinash Sharma, 2017)

La fonction d'activation sigmoïde, illustrée dans la Figure 7 : Fonction d'activation sigmoïde, n'est pas linéaire, pour toutes les valeurs d'entrées, les valeurs de sorties restent entre 0 et 1. Ce cas est utilisé s'il est nécessaire d'avoir des résultats continus entre 0 et 1 (Avinash Sharma, 2017).

3.2.3 Phase d’apprentissage

Lors du développement d’un modèle d’apprentissage automatique, une fonction de coût est définie dans le but de mesurer les capacités prédictives du modèle. Cette fonction pénalisera les erreurs du modèle en fonction des paramètres.

L’objectif est de minimiser la fonction de coût pour trouver les paramètres optimaux qui permettent au modèle de fournir les meilleurs résultats, selon une métrique définie (Jeremy Jordan, 2017).

Dans le cadre de ce travail, uniquement les fonctions de coût les plus couramment utilisées seront expliquées et présentées telles que : Mean Square Error Loss et Cross Entropy Loss/ Log Loss.

Mean Square Error Loss, appelé aussi MSELoss, est la plus utilisée pour des problèmes de régression où l’on veut, par exemple, prédire une quantité (Calle, 2020).

Figure 8 : Formule MSELoss

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

(Calle, 2020)

Dans la Figure 8 : Formule MSELoss, le N représente la taille de l’ensemble des données, le i représente la position dans l’ensemble des données, le y représente la valeur réelle et le $\hat{y}$ représente la prédiction. Mean Square Error Loss représente la différence de la moyenne au carré entre la valeur réelle et la prédiction (Calle, 2020).

La Cross Entropy Loss mesure la différence entre deux distributions de probabilités pour une entrée (Jason Brownlee, 2019). Elle calcule la performance de classification d’un modèle où la sortie est une valeur entre 0 et 1.

Figure 9 : Formule Cross Entropy Loss

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

(Paolo Perrotta, 2020)

Dans la Figure 9 : Formule Cross Entropy Loss, la représentation des valeurs est presque identique à la Figure 8 : Formule MSELoss, à la différence de la représentation de la taille de l’ensemble des données qui est m. La Cross Entropy Loss représente la

somme de la valeur négative de toutes les valeurs réelles multipliées par le logarithme de la prédiction (Paolo Perrotta, 2020).

La différence entre la Log Loss et la Cross Entropy Loss est que la première permet le calcul de l'erreur pour deux classes tandis que la Cross Entropy Loss est utilisée quand il y a deux classes ou plus (Paolo Perrotta, 2020).

Figure 10 : Formule Log Loss

$$L = -\frac{1}{m}\sum_{i=1}^{m}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

(Paolo Perrotta, 2020)

Dans la Figure 10 : Formule Log Loss, la représentation des valeurs est identique à la Figure 9 : Formule Cross Entropy Loss. La Log Loss est la somme de la valeur négative de toutes les valeurs réelles multipliées par le logarithme de la prédiction additionnée à la valeur négative de la valeur réelle multipliée par le logarithme de la valeur négative de la prédiction.

Le réseau de neurone à propagation est le premier et le plus simple type de réseau neuronal artificiel. Dans ce réseau, l'information se déplace vers l'avant, à partir des nœuds d'entrée, en passant par les couches cachées et vers les nœuds de sortie. Il n'y a pas de cycles ou de boucles dans le réseau (Wikipédia, 2020).

Cependant, pendant la phase d'apprentissage de ce type de réseau, il existe un problème : la propagation d'erreur. En effet, à chaque passage de couche, le réseau commet des erreurs qui vont se transmettre aux prochains neurones augmentant l'erreur de ceux-ci, jusqu'à arriver à la prédiction finale. Le risque est qu'au fil des passages dans les couches, l'erreur s'agrandisse faussant ainsi les résultats retournés.

Comme l'objectif est de minimiser l'erreur qu'un modèle produit et dans le but d'optimiser ses prédictions, une des solutions est l'utilisation de l'algorithme de la descente du gradient.

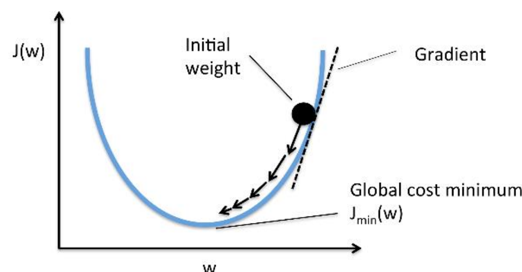
3.2.3.1 L'algorithme de descente du gradient

La descente du gradient est une technique d'optimisation utilisée dans les algorithmes de *machine learning* (Jeremy Jordan, 2017). Son objectif est de trouver les paramètres qui minimisent les erreurs dans les prédictions, en se déplaçant de manière itérative dans la direction opposée du gradient de la fonction de coût (Sara Iris Garcia, 2018).

En effet, le calcul du gradient est utilisé par la rétropropagation de l'erreur en parcourant les différentes couches d'un réseau de neurones de la couche de sortie vers la couche d'entrée, par exemple.

Le gradient est un vecteur dont chaque position est composée d'une dérivée partielle d'une variable et il représente la pente de la tangente d'une fonction f, pointant vers la direction qui augmentera la sortie de cette fonction f (Sara Iris Garcia, 2018). Pour mieux comprendre, imaginons que nous soyons aveugles, en haut d'une montagne et notre objectif est d'atteindre le point le plus bas de la montagne. La solution serait de faire de petits pas à la recherche de l'inclinaison de la pente. Puis, un pas après l'autre, descendre la montagne jusqu'à notre objectif. Il s'agit exactement de ce que l'algorithme tente de réaliser.

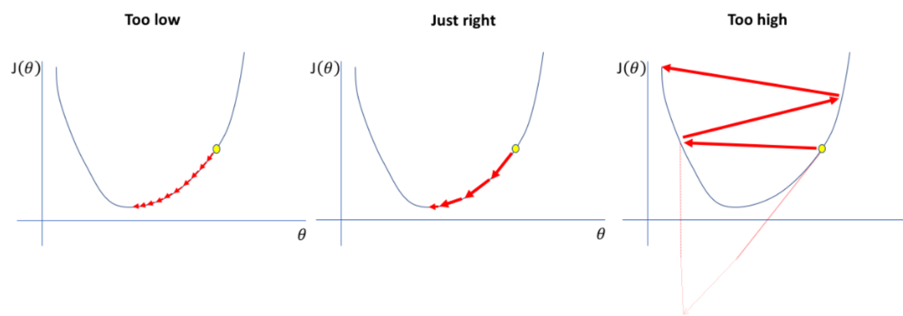
Figure 11 : Introduction à l'algorithme de la descente du gradient



En référence à la Figure 11 : Introduction à l'algorithme de la descente du gradient qui illustre la descente du gradient appliquée à une fonction convexe qui a la propriété de ne contenir qu'un seul minimum. Par analogie à l'exemple précédent, la montagne représente la fonction de coût, les petits pas représentent le taux d'apprentissage (*learning rate*) et la direction du déplacement est donnée par l'inverse du gradient (Garcia, 2018).

Le taux d'apprentissage est une variable qui permet de réguler la distance parcourue dans la direction du minimum sur la fonction de coût. Le gradient sera multiplié par le *learning rate* déterminant ainsi le prochain point (dans l'analogie de la montagne cela fait référence à un petit pas).

Figure 12 : Les différents taux d'apprentissage



(Jeremy Jordan, 2018)

Le *learning rate* peut être une vraie difficulté lors de l'entrainement d'un réseau neuronal. En effet, le taux est généralement fixé au début de l'entrainement et est arrangé au fur et à mesure des observations afin d'atteindre le rendement parfait. Un *learning rate* trop faible augmentera le temps de calcul et le nombre de mises à jour du gradient alors qu'un taux trop élevé cause des mises à jour trop drastiques ce qui créera des comportements différents d'un changement à l'autre (Jeremy Jordan, 2018). Force est de constater que dans la

Figure 12 : Les différents taux d'apprentissage. Nous nous apercevons en comparant les graphiques *too low* et *too high* au graphique *just right*, le nombre d'étape est bien plus important ou alors bien trop imprévisible pour évoluer.

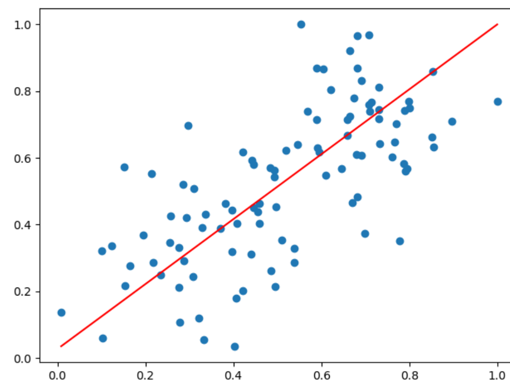
Avant de passer à l'implémentation technique de la descente du gradient, nous allons utiliser celle-ci sur la fonction de coût de l'algorithme de la régression linéaire au travers d'une seule mise à jour des poids.

3.2.3.2 Calcul du gradient sur régression linéaire

3.2.3.2.1 Régression linéaire univariée

La régression linéaire est un algorithme d'apprentissage supervisé utilisé dans les problèmes de régression, c'est-à-dire qu'il permet de prédire des valeurs continues à partir de variables prédictives. Son objectif est de trouver une droite qui se rapproche le plus possible d'un ensemble de points (Younes Benzaki, 2018). Les variables prédictives peuvent être représentées par des points et l'application optimisée de la régression linéaire peut-être représentée par une ligne de régression qui séparent ces points dans un espace. Schématiquement, voici un exemple de résultat recherché.

Figure 13 : Exemple de régression linéaire



(Alexandre da Mota, 2020)

En référence à la Figure 13 : Exemple de régression linéaire, les points bleus sont les données d'entrée et sont représentées par le couple (x^i, y^i), où chaque instance de x^i à une valeur à prédire y^i.

Les valeurs x^i sont les variables prédictives, et y^i les valeurs observées. Nous cherchons à trouver une droite $f(x) = m * x + b$ qui représente notre modèle de régression linéaire où m représente la pente, x représente l'entrée et b représente le *y-intercept*, ou l'ordonnée à l'origine.

La pente représente le taux de changement de y^i quand x^i change et le *y-intercept* représente la valeur de y^i quand la droite $f(x)$ traverse l'axe des y.

3.2.3.2.2 Explication mathématique

Comme présenté dans le chapitre sur la Régression linéaire univariée, l'objectif sera de définir la droite prédictive de notre modèle. Etant donné que la régression linéaire fait partie du paradigme supervisé du *machine learning*, la prédiction sera comparée à la valeur réelle. De ce fait, nous pouvons déduire l'erreur de notre modèle qui sera la différence entre la prédiction et la valeur réelle. Cela veut dire que notre prédiction de y^i par rapport à x^i sera la valeur que l'on trouve à l'intersection avec la droite soustraite à la valeur réelle de y^i. Grâce à cela, nous pouvons déterminer la fonction de coût de notre modèle qui sera $f(coût) = \sum_{i=1}^{n}(\widehat{y_i} - y_i)^2$ où n représente l'ensemble des points, $\widehat{y_i}$ représente la prédiction et y_i la valeur réelle.

Pour effectuer les mises à jour des valeurs du modèle lors de l'entrainement, l'objectif sera de mettre à jour les paramètres m et b de la fonction du modèle présenté au chapitre de la Régression linéaire univariée afin de minimiser l'erreur du modèle.

Une mise à jour de ces paramètres consisterait pour m à faire $m = m \pm \Delta m$, où Δm est le gradient de la fonction de coût par rapport à m. C'est-à-dire que nous voulons prendre m (la pente) et en rajouter son gradient. La mise à jour de b (le y-*intercept*) se fait comme pour m. Pour trouver l'erreur la plus basse, l'algorithme de la descente du gradient utilise un concept qui dit que si la dérivée d'une fonction est connue alors la direction pour minimiser cette fonction est connue aussi. Alors, afin de minimiser la fonction de coût, il faut trouver sa dérivée. Normalement, nous voulons minimiser l'erreur sur son ensemble mais, afin de mieux comprendre le fonctionnement du calcul du gradient, nous prendront une erreur à la fois. Pour cela, nous enlèverons la somme de la fonction de coût résultant à $f(coût) = (\widehat{y_i} - y_i)^2$.

Maintenant que notre fonction de coût locale est établie, nous devons trouver la contribution de m dans l'erreur de notre modèle et pour cela, il faut trouver la dérivée de la fonction de coût relative à la dérivée de m.

Pour calculer ces dérivées, il faut utiliser deux règles fondamentales de *calculus*. Originellement appelé *infinitesimal calculus,* il s'agit de l'étude mathématique du changement continu. Cette branche permet de comprendre les changements de valeurs liées à une fonction (Wikipédia, 2020). Parmi les multiples règles énoncées par cette étude, deux règles nous serons importantes, la règle de la puissance et la règle d'enchainement. La règle de la puissance dit que pour une fonction $f(x) = x^n$ alors sa dérivée est $n * x^{n-1}$ (Coding Train, 2017). Pour la règle de l'enchainement, supposons les fonctions $f(y) = x^2$ et que $f(x) = z^2$. Si nous souhaitons la dérivée de y relative à z, alors nous constatons que y dépend de x et que x dépend de z. Donc, la règle d'enchainement dit qu'il est possible de prendre la dérivée de y relative à x qui sera $2 * x$ (règle de la puissance) multiplié par la dérivée de x relative à z qui sera $2 * z$ donc, $2 * x * 2 * z$. Cela veut dire qu'il est possible d'enchainer les dérivés, en prenant la dérivée d'une fonction relative à une autre fonction, multiplié par la dérivée de cette autre fonction relative à une dernière fonction (Coding Train, 2017).

L'objectif est de trouver la dérivée de la fonction de coût relative à la dérivée de m, notée $\frac{\partial f(coût)}{\partial m}$. Grâce à la règle de puissance notre fonction de coût devient $2 * (\widehat{y_i} - y_i)$. Ensuite, nous remarquons que notre fonction de coût dépend de la fonction de notre modèle au travers de la prédiction. Dans ce cas, nous pouvons utiliser la règle d'enchainement et prendre la dérivée de la fonction de coût relative à m pour la multiplier par la dérivée de la prédiction relative à la dérivée de m, notée

$$\frac{\partial f(coût)}{\partial m} = 2 * (prediction - y) * \frac{\partial (prediction - y)}{\partial m}$$

Décomposons maintenant la dernière partie de notre enchainement de dérivé, et trouvons la dérivée de $(prediction - y)$. La prédiction est la fonction de notre modèle $f(x) = m * x + b$ où la valeur réelle est soustraite. Donc, $m * x + b - y$ et nous rentrons dans le monde des dérivées partielles. Nous parlons de dérivée partielle car elle implique de dériver des constantes. Dans la formule, nous constatons que b, y sont des constantes à l'instar de $m * x$. Maintenant, si nous prenons la dérivée de $m * x$, et appliquons la règle de la puissance, la formule devient $1 * x * m^0$, donc x. Ensuite étant donné que la dérivée « décrit » le changement d'une valeur, et comme une constante est par définition, inchangeable, sa dérivée est de 0. Voici la formule décrivant la dérivée de la fonction de coût relative à la dérivée de m

$$\frac{\partial f(coût)}{\partial m} = 2 * (prediction - y) * x$$

Enfin, pour la dérivée de la fonction de coût relative à la dérivée de b, tout est pareil que pour m. Cependant, dans la fonction $m * x + b - y$, $m * x$ devient une constante donc sa dérivée est de 0. y est toujours une constante donc 0. Enfin b, grâce à la règle de puissance devient $1 * b^0$, par conséquent égal à 0. Voici la dérivée de la fonction de coût relative à la dérivée de b

$$\frac{\partial f(coût)}{\partial b} = 2 * (prediction - y)$$

Rapport-Gratuit.com

3.2.3.2.3 *Implémentation Python*

Maintenant nous allons voir comment cette logique mathématique est implémentée en programmation. Le code implémenté sera une régression linéaire optimisée par l'algorithme de la descente du gradient. Le jeu de données utilisé est un ensemble de données spécifiquement adapté pour la régression linéaire et disponible sur kaggle.com/dataset.

Figure 14 : Code algorithme du gradient sur régression

```
10      m = 0
11      b = 0
12      while 1:
13          for i in range(200):
14              prediction = m * X + b
15              error = prediction - Y
16              m = m - learning_rate * (error * X)
17              b = b - learning_rate * error
```

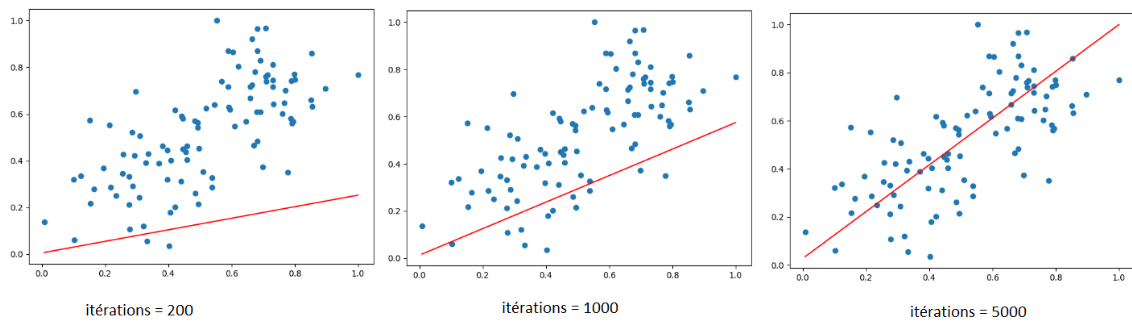
(Alexandre da Mota, 2020)

En référence à la Figure 14 : Code algorithme du gradient sur régression, tout d'abord, il s'agit d'initialiser les variables du modèle et créer une boucle qui, à chaque itération, mettra à jour les paramètres en minimisant l'erreur comme expliqué au chapitre Régression linéaire univariée. Ensuite, l'erreur, est définie par la différence entre la prédiction et la valeur réelle.

Pour finir, il faut mettre à jour les variables m, b et définir le taux d'apprentissage qui sera initialisé à 0.001. Pour trouver le bon *learning rate*, il n'existe pas de formule, généralement, il faut essayer plusieurs taux afin de voir lequel fonctionne le mieux (JORDAN, 2018). Dans la plupart des cas, un *learning rate* aux alentours des 0.001 est un bon taux. De ce fait, ceux-ci permettront au modèle de s'affiner à mesure des itérations pour finalement donner une droite qui coupe le jeu de donnée en deux parties.

A noter que l'explication dans le chapitre Explication mathématique, lors des mises à jour de m et b, les dérivées de la fonction de coût relative à la dérivée de m est $\frac{\partial loss}{\partial m} = 2 * (prediction - y) * x$. Etant donné que nous calculons pour un gradient manuellement, ici le 2 de la dérivée de la fonction de coût est enlevé car la mise jour des valeurs est régulée par le *learning rate* donc il n'est pas utile d'altérer plus la valeur du gradient. Les mêmes étapes sont faites pour la dérivée relative à b.

Figure 15 : Graphique d’évolution de la droite



(Alexandre da Mota, 2020)

Comme nous pouvons le constater dans la Figure 15 : Graphique d’évolution de la droite ci-dessus, il est possible de voir l’amélioration de la droite au fur et à mesure des itérations et, après 5000 itérations, la droite a atteint l’emplacement où l’erreur est à son minimum.

Après avoir décomposé l’algorithme du gradient, nous allons l’implémenter sur un réseau de neurones artificiels et minimiser l’erreur sur son ensemble.

3.2.3.3 Calcul du gradient dans un réseau de neurones artificiels manuellement

Pour cet exercice, nous allons utiliser la fonction d’activation ReLU, expliquée à la Figure 6 : Fonction d’activation ReLU et utiliser la fonction de coût Mean Square Error Loss, expliquée à la Figure 8 : Formule MSELoss

Figure 16 : Code algorithme du gradient manuellement partie 1

```
5   batch_size, input_dim, hidden_layer_size, output_dim = 64, 1000, 100, 10
6
7   x = np.random.randn(batch_size, input_dim)
8   y = np.random.randn(batch_size, output_dim)
9
10  w1 = np.random.randn(input_dim, hidden_layer_size)
11  w2 = np.random.randn(hidden_layer_size, output_dim)
```

```
43  for t in range(500):
44      h = x.dot(w1)
45      h_relu = np.maximum(h, 0)
46      y_pred = h_relu.dot(w2)
```

(Alexandre da Mota, 2020)

En référence à la Figure 16 : Code algorithme du gradient manuellement partie 1, pour l’initialisation du réseau de neurones artificiels, la librairie numpy a été utilisée pour créer des vecteurs initialisés avec des valeurs aléatoires utilisant la fonction de numpy. Nous avons donc créé x et y qui représentent respectivement nos entrées et nos sorties, ainsi que les vecteurs poids également initialisés de façon aléatoire. La taille du *batch* est de

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

64, la tailles des entrées est de 1000, la tailles des sorties est de 10 et il y a 100 neurones dans la couche cachée.

Un *batch* est un terme utilisé en *machine learning* faisant référence au nombre d'exemple utilisé depuis un ensemble de données pour une itération.

Ensuite, une boucle devra être créée pour entrainer le modèle en mettant à jours ses paramètres. Pour ce faire, nous effectuons une multiplication matricielle de nos entrées x avec un vecteur de poids $w1$ qui relie la couche d'entrée à la couche cachée. Ensuite, avant d'utiliser les poids qui relient la couche cachée à la couche de sortie, les valeurs sont passées dans une fonction d'activation. Comme dit au chapitre précédent, nous allons utiliser la fonction ReLU, qui s'occupe de mettre à 0 toutes les valeurs négatives.

A la ligne 44, le .dot() permet de faire la multiplication matricielle entre x et $w1$ $h = x * w1$. Ensuite, à la ligne 45, le .maximum($h, 0$) retourne la valeur maximum entre 0 et chaque élément du vecteur h, il s'agit de la fonction d'activation ReLU définie par $r(h) = \max(0, h)$. Enfin, nous faisons la multiplication matricielle entre le vecteur h_relu et $w2$ ce qui donne : $\mathrm{y_pred} = r(h) * w2$.

Figure 17 : Code algorithme du gradient manuellement parties 2

```
54        grad_y_pred = 2.0 * (y_pred - y)
55        grad_w2 = h_relu.T.dot(grad_y_pred)
56        grad_h_relu = grad_y_pred.dot(w2.T)
57        grad_h = grad_h_relu.copy()
58        grad_h[h < 0] = 0
59        grad_w1 = x.T.dot(grad_h)
```

```
64        w1 -= learning_rate * grad_w1
65        w2 -= learning_rate * grad_w2
```

(Alexandre da Mota, 2020)

Maintenant que le modèle a fait ses prédictions, la rétropropagation des gradients de la fonction de coût par rapport aux paramètres doit être appliquée, afin d'optimiser les prédictions du modèle. En référence à la Figure 17 : Code algorithme du gradient manuellement parties 2, à la ligne 54, nous devons calculer le gradient de la prédiction (grad_y_pred), c'est-à-dire, la dérivée de la fonction de coût qui est $f(x) = \sum_{i=1}^{n}(y_pred - y)^2$. Cette fonction permet de récupérer l'ensemble des erreurs. Actuellement, nous voulons effectuer la rétropropagation à la main donc, nous souhaitons prendre les erreurs une à une et les mettre à jour grâce à la fonction est $2 * (y_pred - y)$ (voir l'explication au chapitre Explication mathématique). Au niveau du code, le T est utilisé pour indiquer une transposition d'un vecteur ou d'une matrice (vecteur à deux dimensions au minimum), ce qui a pour effet d'inverser la disposition de

ses lignes avec celle de ses colonnes. Pour qu'une matrice x puisse être multipliée avec une matrice y, le nombre de colonnes de x doit correspondre au nombre de lignes de y.

Ayant maintenant le gradient de la prédiction, nous continuons la rétropropagation à la ligne 55. Il s'agit ici de calculer le gradient du poids des sorties (grad_w2) ce qui représente la dérivée de la fonction de coût relative à la dérivée des poids des sorties $w2$, notée $\frac{\partial f(coût)}{\partial w2}$. La fonction de coût étant liée à la prédiction et la prédiction est liée aux poids des sorties $w2$, la règle de l'enchainement des dérivées s'applique résultant à cette formule :

$$\frac{\partial f(coût)}{\partial w2} = \frac{\partial f(coût)}{\partial y_pred} * \frac{\partial y_pred}{\partial w2}$$

Sachant que la dérivée de la fonction de coût relative à la dérivée de y_pred est $2 * (y_pred - y)$ et que la dérivée de y_pred relative à la dérivée de $w2$ est le ReLU du vecteur des entrées, voici la formule du gradient de la fonction de coût relative à vecteur des poids de sortie, $\frac{\partial f(coût)}{\partial w2} = 2 * (y_pred - y) * r(h)$.

A la ligne 56, on continue la rétropropagation, ici nous cherchons à calculer le gradient du ReLU du vecteur des entrées (grad_h_relu). Pour cela, il faut calculer la dérivée de la fonction de coût relative à la dérivée du vecteur des entrées multipliées par les poids, représenté par la formule $\frac{\partial f(coût)}{\partial h}$. La fonction de coût est liée à la prédiction et la prédiction est liée au vecteur des entrées multipliés par les poids. Toujours en utilisant la règle de l'enchainement, cette formule représente le gradient recherché :

$$\frac{\partial f(coût)}{\partial h} = \frac{\partial f(coût)}{\partial y_pred} * \frac{\partial y_pred}{\partial h}$$

Connaissant la dérivée de la fonction de coût, il faut maintenant déterminer la dérivée de y_pred relative à la dérivée de h qui est le vecteur des poids de sorties, résultant $\frac{\partial f(coût)}{\partial h} = 2 * (y_pred - y) * w2$.

Ensuite, aux lignes 57 et 58, il faut créer une copie du gradient du vecteur des entrées multiplié par les poids (grad_h) pour faire le ReLU sur les nouvelles valeurs du vecteur.

Pour finir la rétropropagation, à la ligne 59, nous cherchons à calculer le gradient du vecteur des poids des entrées (grad_w1). Nous souhaitant la dérivée de la fonction de coût relative à la dérivée du vecteur des poids d'entrées représentée mathématiquement par cette formule $\frac{\partial f(coût)}{\partial w1}$. La fonction de coût est liée au vecteur des entrées multiplié

par les poids et, ce même vecteur, est lié au vecteur des poids d'entrées. Voici la formule représentant cet enchainement de dérivées

$$\frac{\partial f(coût)}{\partial w1} = \frac{\partial f(coût)}{\partial h} * \frac{\partial h}{\partial w1}$$

Donc, la dérivée de la fonction de coût relative à la dérivée de h représente le gradient de h. Quant à la dérivée de h relative à la dérivée de w1, où $h = x * w1$, nous pouvons utiliser la règle de la puissance donc $1 * x * w1^0 = x$ donc, $\frac{\partial f(coût)}{\partial h} = 2 * (y_pred - y) * w2 * x$.

Finalement, aux lignes 64 et 65, nous mettons à jour les nouveaux poids en les soustrayant des anciens multipliés par le *learning rate* qui est initialisé à 0.001.

Maintenant que nous avons vu comment faire manuellement l'algorithme de la descente du gradient, nous allons utiliser la librairie PyTorch et ses fonctionnalités.

3.2.3.4 Calcul du gradient dans un réseau de neurone avec PyTorch

Concernant l'initialisation du réseau de neurones artificiels, elle est réalisée de la même façon que pour le chapitre Calcul du gradient dans un réseau de neurones artificiels manuellement. Cependant, en vue d'utiliser librairie PyTorch, il nous est nécessaire d'appliquer un *tensor* afin d'utiliser les outils mis à disposition par le *framework*. Un *tensor* est, mathématiquement, un tableau de nombres ou fonctions, qui se transforment en accord avec certaines règles et de manière coordonnée (Davis, 2020). Il s'agit de la même chose qu'un tableau numpy, les *tensor* ne connaissent rien au *machine learning* ou gradient. Fondamentalement, il s'agit d'un tableau n-dimensionnel générique à utiliser pour le calcul numérique (PyTorch, 2020).

Figure 18 : Code algorithme du gradient avec PyTorch partie 1

```
batch_size, input_dim, hidden_layer_size, output_dim = 64, 1000, 100, 10

x = torch.randn(batch_size, input_dim)
y = torch.randn(batch_size, output_dim)

model = torch.nn.Sequential(
    nn.Linear(input_dim, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, output_dim),
)

loss_fn = nn.MSELoss(reduction='sum')
optimizer_fn = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

(Alexandre da Mota, 2020)

En référence avec la Figure 18 : Code algorithme du gradient avec PyTorch partie 1, nous utilisons le torch.randn(), pour initialiser nos entrées et sorties chargées de valeurs aléatoires. Quant au modèle, nous le créons grâce à la fonctionnalité Sequential depuis l'outil *neural network* (nn). Celui-ci nous permet de créer un réseau de neurones artificiels en indiquant la disposition des couches ainsi que les fonctions d'activation. Il y a une première couche linéaire des entrées, la fonction d'activation ReLU avant le passage à la seconde couche linéaire des sorties. Puis, nous initialisons notre fonction de coût qui est la MSELoss. Ici, le paramètre est reduction='sum' car nous désirons faire la somme des résultats. Puis, l'optimiseur Adam est initialisé et il lui sera fourni les paramètres du modèle (en rapport au réseau de neurones) et le *learning rate*. A noter que la MSELoss et l'optimiseur Adam sont directement implémentés dans PyTorch et utilisables depuis le *framework*.

Un optimiseur permet de relier la fonction de coût et les paramètres du modèle en le mettant à jour en accord avec la fonction de coût. C'est-à-dire, que les optimiseurs façonnent le modèle dans sa forme la plus précise possible en l'affutant avec les poids. Le fonction coût est le guide indiquant à l'optimiseur s'il est sur la bonne ou la mauvaise direction.

Figure 19 : Code algorithme du gradient avec PyTorch partie 2

```
31  for i in range(500):
32      y_pred = model(x)
33      loss = loss_fn(y_pred, y)
34
35      model.zero_grad()
36      loss.backward()
37
38      optimizer_fn.step()
```

(Alexandre da Mota, 2020)

Ensuite, en référence à la Figure 19 : Code algorithme du gradient avec PyTorch partie 2, il s’agit de créer la boucle qui nous permettra, itérativement, d’entrainer notre modèle. Pour ce faire, nous allons passer notre x dans le modèle pour qu’il nous donne une prédiction. Nous envoyons la prédiction et la valeur réelle à la fonction de coût qui nous donne l’erreur. Nous mettons les anciens gradients à 0 car nous ne voulons pas que les anciens gradients viennent perturber les nouveaux calculs. Nous faisons la rétropropagation utilisant la fonction de coût et la fonctionnalité backward() de la MSELoss disponible grâce à PyTorch. Enfin, nous mettons à jour nos poids avec notre optimiseur.

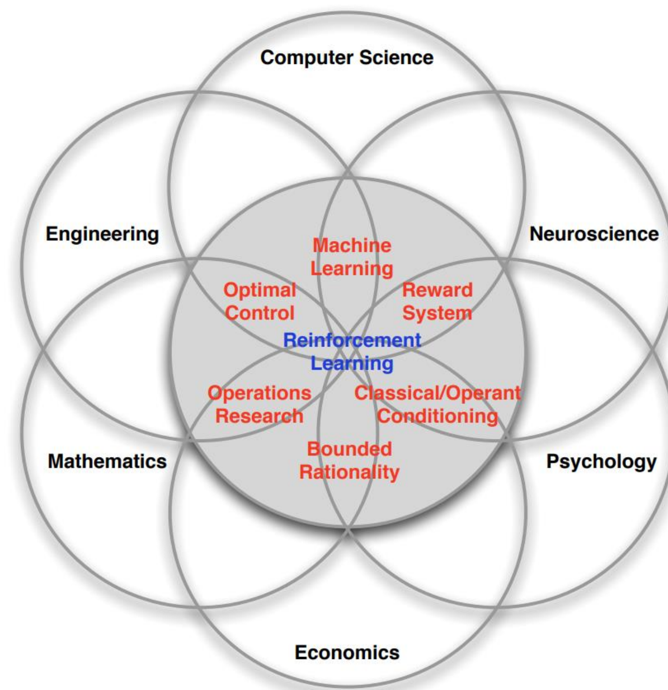
Avec ceci, le chapitre d’introduction aux réseaux de neurones artificiels et à l’algorithme de la descente du gradient sont clos. L’objectif maintenant sera d’approfondir la théorie concernant l’apprentissage par renforcement et ses algorithmes.

3.3 Apprentissage par renforcement

L'idée d'apprendre via des interactions avec notre environnement est probablement la première réponse à laquelle on pense quand on se pose des questions sur la nature de l'apprentissage (Richard S. Sutton, 2014). Quand un enfant (agent) joue, bouge les bras, ou observe, il n'a pas de professeur mais il peut capter son environnement. Expérimenter ces connections produit des informations à propos de la cause et de l'effet, à propos des conséquences des actions, et de ce qu'il faut faire afin d'atteindre un objectif (Richard S. Sutton, 2014).

La particularité de l'apprentissage par renforcement est qu'elle se trouve à l'intersection de plusieurs domaines scientifiques comme présenté dans la Figure 20 : Introduction à l'apprentissage par renforcement. Dans tous ces domaines d'expertise, il y a un problème que tous étudient, la science de la prise de décisions et cela représente l'essence même de l'apprentissage par renforcement (David Silver, 2015).

Figure 20 : Introduction à l'apprentissage par renforcement



(David Silver, 2015)

Les différences entre l'apprentissage par renforcement et les autres paradigmes du *machine learning* sont qu'il n'y a pas de superviseur. Aucun élément externe informe l'agent que cette action précise à ce moment donné est le meilleur choix. Ce processus fonctionne par le biais de récompenses. De plus, le résultat qu'engendre l'action ne survient pas directement, il faut attendre, généralement, plusieurs étapes avant d'avoir

un retour. Enfin, l'agent interagi avec son environnement ce qui provoque un dynamisme sur les informations qu'il reçoit et qui changent en fonction de l'action choisie (David Silver, 2015).

L'apprentissage par renforcement s'appuie sur cette logique pour apprendre à une machine. Un agent qui interagit avec son environnement et agi en fonction des informations qu'il reçoit. Le problème de l'apprentissage par renforcement est d'apprendre à agir. Comment associer situations et actions afin de maximiser la récompense (Richard S. Sutton, 2014).

Avant de penser à des solutions, comprenons les éléments qui structurent l'apprentissage par renforcement.

3.3.1 Eléments du paradigme

Au-delà de l'agent et de l'environnement, quatre sous-éléments principaux à un système d'apprentissage par renforcement sont identifiables : une politique, un signal de récompense, une *value function* et, optionnellement, un modèle de l'environnement (Richard S. Sutton, 2014).

Une politique défini le comportement de l'agent à un moment donné, c'est-à-dire, que cela représente le lien entre les états observés dans l'environnement par l'agent aux actions prises dans ces états (Richard S. Sutton, 2014).

Un signal de récompense défini l'objectif d'un problème d'apprentissage par renforcement. A chaque étape, l'environnement envoie une récompense à l'agent et son but est de maximiser le total des récompenses sur le long-terme. Un signal de récompense défini si les évènements sont bons ou mauvais pour l'agent. Dans un système biologique, il représenterait le plaisir et la douleur (Richard S. Sutton, 2014).

Si la récompense détermine ce qui est bon dans l'immédiat, la *value function* informe de ce qui est bon sur le long-terme. En effet, la *value function* d'un état est le total des récompenses que l'agent peut espérer obtenir dans le futur en partant de cet état. Pour continuer sur l'analogie humaine, si la récompense est comme le plaisir ou la douleur, une *value function* représente un jugement plus prévoyant. En effet, il s'agit de la récompense que l'agent pourrait espérer avoir plus tard si cette action est faite tout de suite (Richard S. Sutton, 2014). Il s'agit de la clé pour la résolution des problèmes d'apprentissage par renforcement.

Un modèle de l'environnement est optionnel selon le type et le nombre d'interaction que l'agent a avec l'environnement. Dans le monde du jeux-vidéos, l'apprentissage se ferait

sans modèle car l'agent interagit en continue avec l'environnement ce qui lui permet d'apprendre par expérience. A l'inverse, si l'agent interagit un nombre limité de fois avec l'environnement, l'objectif est de créer un modèle sur la base des premières interactions et utiliser le modèle construit pour simuler d'autres épisodes (Ziad Salloum, 2019).

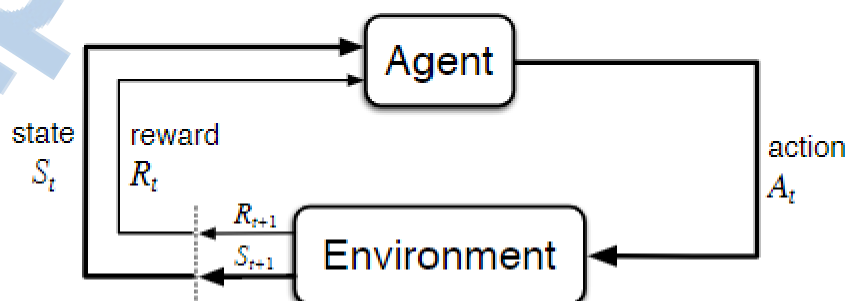
3.3.2 Fonctionnement

L'apprentissage par renforcement est une approche computationnelle sur la compréhension et l'automatisation d'un apprentissage par objectif et de prise de décision (Richard S. Sutton, 2014). Ce paradigme utilise le cadre de *Markof Decision Process* (MDP) pour définir les interactions entre l'agent et l'environnement en termes d'états, d'actions, et de récompenses. Le concept d'état est omniprésent dans l'apprentissage par renforcement. En effet, il est possible de voir cela comme une photo envoyée à l'agent de « comment est l'environnement » à cet instant précis (Richard S. Sutton, 2014).

Le cadre MDP est une formalisation classique d'une séquence de prise de décision où l'action influe sur la récompense immédiate mais également sur les états à venir ainsi que leurs récompenses. L'objectif du cadre MDP est d'estimer la valeur de chaque action dans un état $Q(s, a)$ par des problèmes mathématiques tels que la *value fonction* et *Bellman Equations* présentés plus tard dans le chapitre (Richard S. Sutton, 2014).

Comme présenté dans la Figure 21 : L'interaction entre l'agent et l'environnement dans le cadre MDP ci-dessous, l'agent représente l'apprenant /décideur et celui-ci interagis avec son environnement en continu. L'agent choisit une action et l'environnement réagit en fonction du choix, présentant ainsi à l'agent une nouvelle situation. L'environnement envoie alors une récompense, une valeur numérique que l'agent cherche à maximiser sur le long terme et son nouvel état (Richard S. Sutton, 2014).

Figure 21 : L'interaction entre l'agent et l'environnement dans le cadre MDP



(Richard S. Sutton, 2014)

L'agent et l'environnement interagissent ensemble dans une séquence finie de « pas » représentée dans le schéma par un t. En effet, à chaque « pas », l'agent reçoit une

Rapport-Gratuit.com

représentation de l'état de l'environnement, appelée S_t. Sur cette base, nous choisissons une action, représenté par A_t, à performer. Le « pas » d'après, l'agent reçoit une récompense, notée R_{t+1}, et un nouvel état, appelé $S_{t+1,}$ dû à la conséquence de l'action exécutée. La séquence créée par l'agent et l'environnement commence comme ceci : $S_{0,} A_{0,} R_{1,} S_{1,} A_{1,} R_{2,} \ldots$ Dans ce projet, nous travaillons uniquement avec des états, actions et récompenses qui ont un nombre fini d'éléments. De ce fait, la distribution des probabilités dépend uniquement de l'état et action du précédent pas, c'est-à-dire qu'il y a une probabilité que ces valeurs au pas t apparaissent en fonction de la valeur de l'état et de l'action précédente $p(s', r \mid s,\ a)$ (Richard S. Sutton, 2014).

Un des défis qui survient lors de l'apprentissage par renforcement est en rapport avec les façons d'apprentissage et le choix de l'action. En effet, la meilleure action à un moment donné n'est peut-être pas la meilleure action dans le long terme. Le but est de trouver le bon compromis entre exploitation et exploration (Richard S. Sutton, 2014). Pour ce faire, nous mettons en place un paramètre epsilon ε qui sera la probabilité que l'agent choisisse une action aléatoire dans le but d'explorer de nouvelle possibilité ou, à l'inverse, qu'il choisisse la meilleure action dans le but d'exploiter.

Comme dit plus haut, le but de l'agent lors d'un apprentissage par renforcement est représenté par une récompense. À chaque étape de l'apprentissage, l'objectif de l'agent est de maximiser les récompenses reçues sur le long terme. Nous pouvons imaginer vouloir la somme de toutes les récompenses. Le problème est qu'à ce moment-là, l'agent va maximiser la récompense immédiate et non les récompenses sur le long terme. Pour cela, nous avons besoin d'un concept appelé le *discount rate,* ou aussi gamma $\gamma,$ qui détermine la valeur actuelle d'une récompense future. C'est-à-dire que si la récompense de l'étape $t + 3$, par exemple, est reçue immédiatement, sa valeur sera réduite par le *discount rate.* En effet, selon cette approche, l'agent sélectionnera l'action pour que la somme des récompenses réduites dans le futur soit maximisée. Le *discount rate* est un paramètre qui sera supérieur ou égal à 0 et inférieur ou égal à 1. Si le *discount rate* est égal à 0, cela signifiera que l'agent sera « myope » et ne cherchera à maximiser que la récompense immédiate, à l'inverse, plus il se rapprochera de 1, plus il cherchera plus à maximiser les récompenses sur le long terme. Le gamma est ajouté ainsi sur le calcul des récompenses de l'agent γ^{t-1} donc $\text{total_reward} = R_t + \gamma R_{t+1} + {}^2\gamma R_{t+2} + {}^3\gamma R_{t+3} + \ldots$ (Richard S. Sutton, 2014).

La plupart des algorithmes d'apprentissage par renforcement impliquent l'estimation de *value functions*. Une fonction d'état qui estime les bénéfices que l'agent peut gagner s'il décide d'agir d'une façon dans un état précis. La notion de bénéfice est définie sur les

futures récompenses que l’agent peut espérer avoir dépendamment des actions qu’il entreprendra, mais, pour cela, il doit savoir comment agir dans une situation précise. La *value function* permet de définir les façons d’agir et créé une politique qui est un lien entre un état et les valeurs de chaque action disponible depuis cet état (Richard S. Sutton, 2014). A noter qu’il existe plusieurs *value functions* pour les différents problèmes d’apprentissage par renforcement. Dans ce travail, nous nous concentrerons sur l’*action-value function Bellman Equation*

$$Q(s,a) = r(s,a) + \gamma max Q(s',a')$$

La *q value* de l’état s et de l’action a est la somme de la récompense de l’action a sur l’état s additionnée au maximum de la *q value* de l’état s' et de l’action a'. On remarque ici que l’équation satisfait une relation récursive qui est une des propriétés fondamentales des *value functions* utilisée dans l’apprentissage par renforcement (Richard S. Sutton, 2014).

3.3.3 Tabular Q-learning

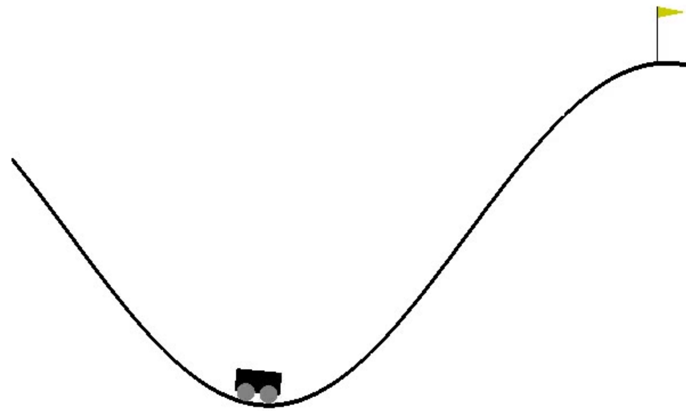
Pour mieux comprendre comment fonctionne un algorithme d’apprentissage par renforcement, nous allons implémenter une des techniques appelé *q learning*. Cette technique ne nécessite aucun modèle initial de l’environnement. La lettre q désigne la fonction qui mesure la qualité d’une action exécutée dans un état donné du système (Wikipédia, 2020) comme présenté lors du chapitre précédent.

Cette méthode permet d’apprendre une politique qui indiquera quelle action effectuer dans chaque état du système. Cela fonctionne par l’apprentissage d’une *action-value function notée* q. Grâce à elle, nous pourrons de déterminer le gain potentiel sur le long terme d’une récompense, apporté par l’exécution d’une action dans un état en suivant une politique optimale. Lorsque cette fonction est connue par l’agent, la politique optimale peut être construite en sélectionnant l’action à la valeur maximale pour chaque état. C’est-à-dire en sélectionnant l’action qui maximise la valeur quand l’agent se trouve dans un état (Tambet Matiisen, 2015).

Pour cet exercice, notre environnement est une voiture en bas d’une colline. En haut de celle-ci se trouve un drapeau. L’agent est la petite voiture et son objectif est de monter la colline pour atteindre le drapeau, voir Figure 22 : Image de l’environnement MountainCar-v0.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Figure 22 : Image de l'environnement MountainCar-v0



(OpenAI, 2020)

3.3.3.1 Implémentation

Pour l'implémentation, nous allons utiliser OpenAI Gym pour initialiser un environnement qui sera « MountainCar-v0 » et nous utiliserons la libraire numpy pour notre tableau qui permettra de créer notre q table qui sera notre politique (Sentdex, 2019).

Figure 23 : Implémentation de l'algorithme tabular q learning partie 1

```
1  import numpy as np
2  import gym
3
4  env = gym.make("MountainCar-v0")
5
6  discrete_obs_space_size = [20] * len(env.observation_space.high)
7  q_table = (np.random.uniform(low=-2, high=0, size=(discrete_obs_space_size + [env.action_space.n])))
```

```
9  discrete_obs_space_win_size = (env.observation_space.high - env.observation_space.low) / discrete_obs_space_size
10
11 def get_discrete_state(state):
12     discrete_state = (state - env.observation_space.low) / discrete_obs_space_win_size
13     return tuple(discrete_state.astype(np.int))
```

(Alexandre da Mota, 2020)

En référence à la Pour l'implémentation, nous allons utiliser OpenAI Gym pour initialiser un environnement qui sera « MountainCar-v0 » et nous utiliserons la libraire numpy pour notre tableau qui permettra de créer notre q table qui sera notre politique (Sentdex, 2019).

Figure 23 : Implémentation de l'algorithme tabular q learning partie 1, à la ligne 4, nous initialisons l'environnement grâce à OpenAI Gym, puis, à la ligne 6, nous déterminons la taille de la q table qui sera de taille 20 multipliée par la taille des observations de l'environnement. La taille est définie subjectivement en fonction des observations de l'environnement. Enfin, à la ligne 7, il s'agit de créer la q table que nous initialisons avec des valeurs aléatoires entre -2 et 0 car l'agent ne reçoit uniquement des récompenses

négatives jusqu'à ce que l'objectif soit atteint et la récompense sera de 0. Au niveau de la taille de la q table, nous avons besoin d'autant de cellules que d'actions pour chaque état. Maintenant, étant donné que les états fournis par l'environnement sont continus, nous avons besoin de les discrétiser. Pour cela, nous devons créer 20 états discrets en rapport aux états continus. A la ligne 9, nous déterminons la taille de la plage des états continus pour créer des états discrets. A la ligne 11, il s'agit de créer une fonction qui permettra de retourner un tuple, qui est une collection ordonnée et inchangeable, contenant notre état discret.

Figure 24 : Implémentation de l'algorithme tabular q learning parties 2

```
15  learning_rate = 0.1
16  discount = 0.95
17  epochs = 5000
18  show = 200
19
20  epsilon = 0.5
21  start_epsilon_decay = 1
22  end_epsilon_decay = epochs // 2
23  epsilon_decay_value = epsilon / (end_epsilon_decay - start_epsilon_decay)
24
25  for epoch in range(epochs):
26      if epoch % show == 0:
27          render = True
28      else:
29          render = False
30      discrete_state = get_discrete_state(env.reset())
31      done = False
32      while not done:
33          if random.random() > epsilon:
34              action = np.argmax(q_table[discrete_state])
35          else:
36              action = random.choice(np.arange(env.action_space.n))
37          new_state, reward, done, _ = env.step(action)
38          new_discrete_state = get_discrete_state(new_state)
39          if render:
40              env.render()
41          if not done:
42              max_futur_q = np.max(q_table[new_discrete_state])
43              current_q = q_table[discrete_state + (action, )]
44              new_q = (1 - learning_rate) * current_q + learning_rate * (reward + discount * max_futur_q)
45              q_table[discrete_state + (action, )] = new_q
46          elif new_state[0] >= env.goal_position:
47              q_table[discrete_state + (action, )] = 0
48          discrete_state = new_discrete_state
49
50      if end_epsilon_decay >= epoch >= start_epsilon_decay:
51          epsilon -= epsilon_decay_value
52
53  env.close()
```

(Alexandre da Mota, 2020)

En référence à la Figure 23 : Implémentation de l'algorithme tabular q learning parties 2, Regardons aux lignes 15 à 18, où nous initialisons nos hyperparamètres. Le *discount rate* est initialisé à 0.95 demandant ainsi à l'agent de maximiser les récompenses futures. Le *learning rate* est initialisé à 0.1 qui est un bon taux de rafraichissement pour

cet exercice (Sentdex, 2020). Aux lignes 20 à 23, nous définissons notre epsilon ε, qui fera en sorte que l'agent explore ou exploite l'environnement. Puis, aux lignes 25 à 29, nous définissons quand est-ce que notre environnement sera affiché. A la ligne 30, nous récupérons notre premier état en réinitialisant l'environnement et à la ligne 32, nous commençons notre entrainement. Aux lignes 33 à 37, grâce à l'epsilon, nous sélectionnons l'action de l'agent puis en fonction de cette action, nous effectuons une étape dans notre environnement et nous récupérons le nouvel état, la récompense ainsi que l'information si l'environnement est terminé. Aux lignes 41 à 47, nous mettons à jour notre *q table* grâce à la l'équation de Bellman. Si ce n'est pas terminé, nous prenons l'action du nouvel état qui a la plus grande *q value* dans notre *q table*, nous récupérons également notre *q value* actuel en respect avec l'action choisis auparavant et nous appliquons l'équation en prenant en compte le *learning rate* et le *discount rate* représentés par la formule mathématique suivante :

$$Q(s,a) = (1 - learning_rate) * Q(s,a) + learning_rate * (r(s,a) + \gamma maxQ(s',a'))$$

Dans le code référant ci-dessus, nous remarque que la *q value* actuel $Q(s,a)$ et que la *q value* maximum de la prochaine étape $maxQ(s',a')$ sont récupérées afin de remplir les conditions de l'équation. Nous mettons à jour la *q table* avec la nouvelle *q value* et, aux lignes 46 et 47, si l'agent a atteint son objectif, nous mettons à jour la *q table* avec la récompense maximum qui est de 0. Enfin, aux lignes 50 et 51, nous mettons à jour notre epsilon pour que l'agent explore moins et exploite plus au fur et à mesure des épisodes.

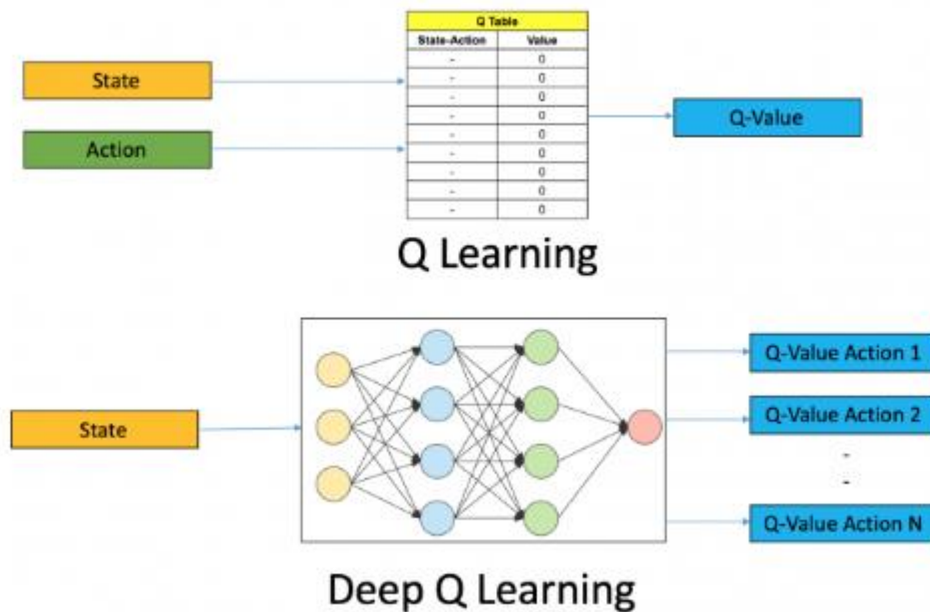
3.3.4 DQN

Le *q learning* expliqué et implémenté dans le chapitre précédent est un algorithme simple mais puissant. Il permet de créer un aide-mémoire pour notre agent qui lui permettra de déterminer exactement quelle action exécuter dans n'importe quelle situation. Mais que se passe-t-il si cet aide-mémoire est trop grand ? Imaginons un environnement avec 10'000 états et 1'000 actions possibles par état. Cela équivaudrait à une table avec 10 millions de cellules, ce qui est bien trop conséquent pour l'agent. Deux problèmes majeurs surviendraient à ce moment-là. Le premier problème est dû à la quantité de mémoire requise pour sauvegarder et mettre à jour la table.

Le second provient du temps requis pour explorer chaque état pour créer la q table qui serait irréalisable. Une solution a été pensée et a donné naissance au *deep q learning*, utiliser un réseau de neurones artificiels (Ankit Choudhary, 2019).

3.3.4.1 Concepts

Figure 25 : Différence entre le q learning et le deep q learning

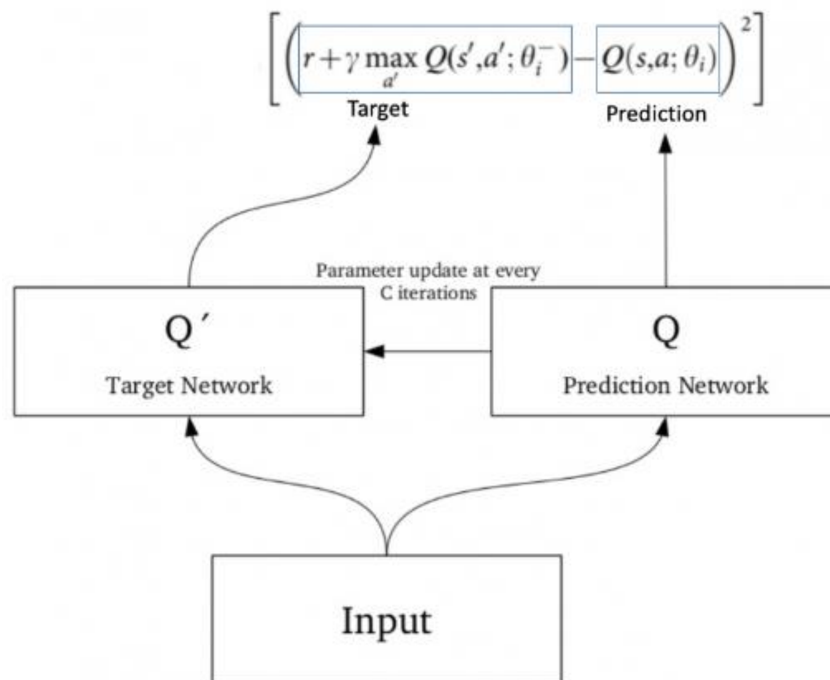


(Ankit Choudhary, 2019)

Dans la Figure 25 : Différence entre le q learning et le deep q learning, nous remarquons la différence entre le *q learning* et le *deep q learning*. Dans la première, la table reçoit un état et une action pour en sortir une *q value* alors que dans le deuxième, l'état est envoyé en tant qu'entrée au réseau de neurones et toutes les *q values* de toutes les actions possibles sont générés en sortie L'idée ici est de reproduire ce que nous pouvons voir en apprentissage profond (*deep learning*) (Ankit Choudhary, 2019).

Pour déterminer la *q value* de la prédiction et de la cible on utilise l'équation de Bellman présenté au chapitre ci-dessus. Le problème c'est que le modèle cible change à chaque itération. Lors de l'utilisation d'un modèle de *deep learning,* le modèle cible est stable mais cela n'est pas le cas pour l'apprentissage par renforcement. En effet, en apprentissage par renforcement, nous dépendons des politiques ou des *value function* pour déterminer le choix de l'action (Ankit Choudhary, 2019). Donc, le modèle change en continu plus l'apprentissage se développe. La solution qui a été trouvée est d'utiliser un autre réseau de neurones pour le modèle cible qui reste « stable » un nombre défini d'itération (Ankit Choudhary, 2019).

Figure 26 : Fonctionnement des deux réseaux de neurones



(Ankit Choudhary, 2019)

En référence à la Dans la Figure 25 : Différence entre le q learning et le deep q learning, nous remarquons la différence entre le *q learning* et le *deep q learning*. Dans la première, la table reçoit un état et une action pour en sortir une *q value* alors que dans le deuxième, l'état est envoyé en tant qu'entrée au réseau de neurones et toutes les *q values* de toutes les actions possibles sont générés en sortie L'idée ici est de reproduire ce que nous pouvons voir en apprentissage profond (*deep learning*) (Ankit Choudhary, 2019).

Pour déterminer la *q value* de la prédiction et de la cible on utilise l'équation de Bellman présenté au chapitre ci-dessus. Le problème c'est que le modèle cible change à chaque itération. Lors de l'utilisation d'un modèle de *deep learning,* le modèle cible est stable mais cela n'est pas le cas pour l'apprentissage par renforcement. En effet, en apprentissage par renforcement, nous dépendons des politiques ou des *value function* pour déterminer le choix de l'action (Ankit Choudhary, 2019). Donc, le modèle change en continu plus l'apprentissage se développe. La solution qui a été trouvée est d'utiliser un autre réseau de neurones pour le modèle cible qui reste « stable » un nombre défini d'itération (Ankit Choudhary, 2019).

Figure 26 : Fonctionnement des deux réseaux de neurones, le modèle cible (*target network*) a la même architecture que le modèle prédiction (*prediction network*) mais avec des hyperparamètres qui sont fixes pendant un certain temps et, après C itération, le

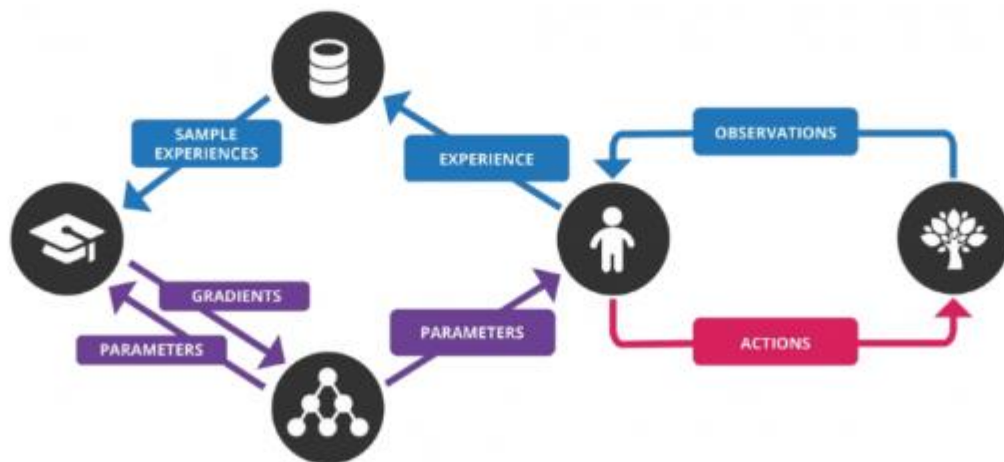
modèle de prédiction est copié dans le modèle cible. Grâce à cela, l'entrainement est plus stable car le modèle cible est fixe (Ankit Choudhary, 2019).

Le dernier concept à comprendre pour le DQN est l'expérience replay ou *replay buffer*. Au lieu d'utiliser la paire état/ action comment en *q learning*, le système enregistre toutes les données découvertes (état, action, récompense, prochaine état) par l'agent dans un tableau. Imaginons un environnement où chaque image représente un nouvel état, pendant l'entrainement, un lot aléatoire de 64 images sur les 100'000 enregistrées pour entrainer notre réseau est pris ce qui créera notre *batch*. Cela nous permet d'obtenir un sous-ensemble dans lequel la corrélation entre les échantillons est faible et fournit également une meilleure efficacité d'échantillonnage (Ankit Choudhary, 2019).

Maintenant que les concepts du DQN ont été expliqué, essayons de comprendre comment ils fonctionnent ensemble.

3.3.4.2 Fonctionnement

Figure 27 : Fonctionnement d'un algorithme de DQN



(Ankit Choudhary, 2019)

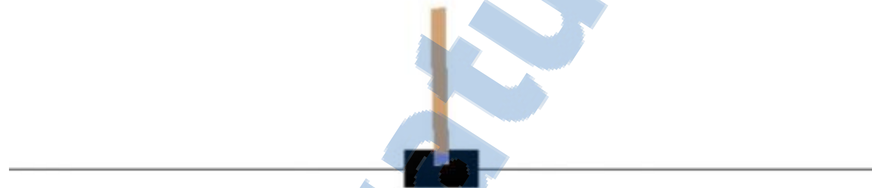
Tout d'abord, il faut alimenter notre algorithme qui retournera les *q values* de toutes les actions possibles dans l'état. Ensuite il faut sélectionner une action en utilisant la politique epsilon. Si la probabilité epsilon est acquise, une action aléatoire est sélectionnée parmi les actions possible sinon, l'action ayant la plus grande *q value* est choisi. Puis, l'action choisi est exécutée dans l'état afin de passer à un nouvel état et de recevoir une récompense. Toutes ses informations sont stockées dans le *replay buffer*, il s'agit d'une transition. Ensuite, il faut échantillonner quelques lots aléatoires de transitions à partir du *replay buffer* et calculer l'erreur grâce à la fonction de coût. Puis, la descente du gradient par rapport à nos paramètres réseau de prédiction est calculée afin de minimiser l'erreur. Après un nombre itérations C, le réseau de prédiction est copié

dans le réseau cible. Enfin, ce processus est répété pour un nombre M d'épisodes (Ankit Choudhary, 2019).

Maintenant que nous comprenons le fonctionnement du *deep q learning*, essayons de l'implémenter sur un environnement de OpenAI Gym, CartPole.

CartPole est un environnement disponible sur Gym. Il s'agit d'une voiture sur un terrain plat. Une barre est attachée sur le toit du véhicule. Il y a trois actions possibles, aller à droite, aller à gauche et freiner. L'agent est la petite voiture et son objectif et de tenir le bâton en équilibre sur le toit du véhicule, voir Figure 28 : Image de l'environnement CartPole-v0.

Figure 28 : Image de l'environnement CartPole-v0



(OpenAI, 2020)

3.3.4.3 Implémentation

Avant de commencer l'explication de l'implémentation, le code qui va être présenté est un exercice qui a été fournis par le directeur de mémoire. Le but de celui-ci étant de coder l'apprentissage d'un algorithme d'apprentissage par renforcement. De ce fait, le réseau de neurones, sa structure, la collecte des différentes interactions entre l'environnement et l'agent et la structure de l'agent ainsi celle du *replay buffer* sont déjà fournis. Le travail demandé est de coder la sélection de l'action, l'entrainement de l'agent, l'enregistrement des interactions dans le *replay buffer* et l'échantillonnage de ceux-ci.

Rapport-Gratuit.com

Figure 29 : Code du ReplayBuffer du DQN CartPole

```
class ReplayBuffer:

    def __init__(self, capacity=int(1e4)):
        """Replay buffer, defined as a FIFO data strcture that
        contains (at most) the `size` most recent experiences.
        """
        self.capacity = capacity
        self._keys = ['ob', 'ac', 'rew', 'done', 'next_ob']
        self._data = {k: deque(maxlen=self.capacity) for k in self._keys}

    def store(self, transition):
        assert set(self._data.keys()) == set(transition.keys()), "non-matching keys"
        for key in self._keys:
            self._data[key].append(transition[key])

    def sample(self, batch_size):
        indexes = np.random.choice(np.arange(0, self.size), batch_size)
        return {'ob': np.array(list(self._data['ob']), dtype=np.float32)[indexes],
                'ac': np.array(list(self._data['ac']), dtype=np.longlong)[indexes],
                'rew': np.array(list(self._data['rew']), dtype=np.float32)[indexes],
                'next_ob': np.array(list(self._data['next_ob']), dtype=np.float32)[indexes],
                'done': np.array(list(self._data['done']), dtype=np.float32)[indexes]}

    @property
    def size(self):
        return len(self._data['ob'])  # arbitrarily picking `ob`
```

(Alexandre da Mota, 2020)

En référence à la Avant de commencer l’explication de l’implémentation, le code qui va être présenté est un exercice qui a été fournis par le directeur de mémoire. Le but de celui-ci étant de coder l’apprentissage d’un algorithme d’apprentissage par renforcement. De ce fait, le réseau de neurones, sa structure, la collecte des différentes interactions entre l’environnement et l’agent et la structure de l’agent ainsi celle du *replay buffer* sont déjà fournis. Le travail demandé est de coder la sélection de l’action, l’entrainement de l’agent, l’enregistrement des interactions dans le *replay buffer* et l’échantillonnage de ceux-ci.

Figure 29 : Code du ReplayBuffer du DQN CartPole, aux lignes 19 et 20, il s’agit d’enregistrer les interactions dans le replay buffer. Pour cela, nous parcourrons nos *keys* qui représente les noms des catégories d’une transition. Nous retrouvons l’état, l’action, la récompense, si l’état est fini et le prochain état. Grâce à cela, nous pourrons récupérer dans notre variable transition reçue en paramètre les différents attributs et les enregistrer dans notre variable de classe _data. Ensuite, il faut créer la fonction l’échantillonnage. Pour ce faire, nous récupérons des indexes aléatoires dans la plage de nos transitions et de la taille du *batch* qui est de 32. En utilisant cette fonction, nous allons recevoir 32 transitions enregistrées dans le *replay buffer*.

Figure 30 : Code de la fonction act du DQN CartPole

```
42    def act(self, state, timesteps_so_far):
43        # Convert the state from a numpy to tensor
44        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
45        # Compute epsilon threshold choosing if the agent is going to explore or to exploit
46        eps_threshold = self.hps.eps_end + (self.hps.eps_beg - self.hps.eps_end) * math.exp(-1 * timesteps_so_far / self.hps.eps_decay)
47        if random.random() > eps_threshold:
48            return self.online_q_net(state).max(1)[1].view(1, 1)
49        else:
50            return random.choice(np.arange(self.env.action_space.n))
```

(Alexandre da Mota, 2020)

En référence à la En référence à la Avant de commencer l'explication de l'implémentation, le code qui va être présenté est un exercice qui a été fournis par le directeur de mémoire. Le but de celui-ci étant de coder l'apprentissage d'un algorithme d'apprentissage par renforcement. De ce fait, le réseau de neurones, sa structure, la collecte des différentes interactions entre l'environnement et l'agent et la structure de l'agent ainsi celle du *replay buffer* sont déjà fournis. Le travail demandé est de coder la sélection de l'action, l'entrainement de l'agent, l'enregistrement des interactions dans le *replay buffer* et l'échantillonnage de ceux-ci.

Figure 29 : Code du ReplayBuffer du DQN CartPole, aux lignes 19 et 20, il s'agit d'enregistrer les interactions dans le replay buffer. Pour cela, nous parcourrons nos *keys* qui représente les noms des catégories d'une transition. Nous retrouvons l'état, l'action, la récompense, si l'état est fini et le prochain état. Grâce à cela, nous pourrons récupérer dans notre variable transition reçue en paramètre les différents attributs et les enregistrer dans notre variable de classe _data. Ensuite, il faut créer la fonction l'échantillonnage. Pour ce faire, nous récupérons des indexes aléatoires dans la plage de nos transitions et de la taille du *batch* qui est de 32. En utilisant cette fonction, nous allons recevoir 32 transitions enregistrées dans le *replay buffer*.

Figure 30 : Code de la fonction act du DQN CartPole,ici, il s'agit de coder la sélection de l'action de l'agent dans la politique epsilon. A la ligne 44, nous convertissons l'état reçu en paramètre de numpy en tensor. Puis, à la ligne 46, nous calculons l'epsilon en fonction de l'avancée de l'agent dans les épisodes et de sa décroissance. Nous comparons l'epsilon avec une valeur aléatoire et, si la valeur aléatoire est plus grande que l'epsilon, alors nous récupérons l'action dont la *q value* est la plus grande dans l'état. Sinon, nous récupérons une action aléatoire parmi les actions possible.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Figure 31 : Code de la fonction train du DQN CartPole

```
91      def train(self):
92          # Get the random sample of transitions that the agent has observed
93          transitions = self.memory.sample(args.batch_size)
94          # Translate it to perform the optimization of the model
95          states = torch.from_numpy(transitions['ob']).float().to(device)
96          actions = torch.from_numpy(transitions['ac'].T).long().to(device)
97          rewards = torch.from_numpy(transitions['rew'].T).float().to(device)
98          next_states = torch.from_numpy(transitions['next_ob']).float().to(device)
99          done = torch.from_numpy(transitions['done'].T.astype(np.uint8)).float().to(device)
100         # Get max predicted Q values (for next states) from target model
101         next_qvalues_target = self.target_q_net(next_states).detach().max(1)[0].unsqueeze(1)
102         # Compute Q targets for current states
103         qvalues_target = rewards + ((args.gamma * next_qvalues_target) * (1 - done))
104         qvalues_target = qvalues_target[:, 0].unsqueeze(1)
105         # Get expected Q values from local model
106         expected_qvalues_online = self.online_q_net(states).gather(1, actions.unsqueeze(1))
107         # Compute the loss
108         loss = F.mse_loss(expected_qvalues_online, qvalues_target)
109         # Minimize the loss
110         self.optimizer.zero_grad()
111         loss.backward()
112         self.optimizer.step()
```

(Alexandre da Mota, 2020)

Enfin, en référence à la Figure 31 : Code de la fonction train du DQN CartPole, il s'agit de coder la fonction train de l'agent. A la ligne 93, les transitions de la taille du *batch* sont récupérées grâce à la fonction sample() codée plus haut. Aux lignes 95 à 99, les différents attributs sont récupérés afin de pouvoir commencer l'entrainement. A la ligne 101, les valeurs maximums du prochain état sont récupérées en provenance du modèle cible. Ensuite, aux lignes 103 et 104, les *q values* de l'état actuel en provenance du modèle cible sont calculées. Puis, à la ligne 106, les q values attendues en provenance du modèle local sont récupérés. Enfin, aux lignes 108 à 112, l'erreur est calculée grâce à la fonction de coût MSELoss et est minimisée utilisant la rétro propagation de l'erreur grâce à la fonction .backward(). Enfin, on met à jour les q values grâce à l'optimiseur.

4. Conclusion

Avant mon inscription pour le Bachelor of Science en Informatique de Gestion, je n'avais jamais essayé d'apprendre un sujet d'informatique, mes connaissances se limitaient à la bureautique. Trouvant le sujet passionnant, j'ai décidé de continuer ma formation dans le domaine dans l'espoir de trouver ma voie. Cinq ans de labeur se sont écoulés mais aucun intérêt particulier que l'on pourrait définir comme passions ne m'est venu. Arrivé à terme de mon cursus, la question s'est posée : est-ce que je vais continuer et me spécialiser ?

Des amis ont continué leurs parcours et, un jour, ils m'ont parlé plus spécifiquement de l'apprentissage automatique. J'ai toujours eu une vague idée de ce que cela représentait mais jamais je n'ai été plus loin dans l'apprentissage jusqu'au jour où l'on m'a parlé du Master en science de l'informations. Un intérêt particulier s'est éveillé en moi mais, âgé de 26 ans, est-ce que j'avais vraiment envie de continuer ma formation ?

Finalement, après l'investissement fournis à la réalisation de ce travail de bachelor, je suis certain d'avoir trouvé un sujet qui m'intéresse assez pour envisager un épanouissement professionnel.

5. Bibliographie

ANKIT, Choudhary, 2019. Deep Q-Learning | An Introduction To Deep Reinforcement Learning. In : *Analytics Vidhya* [en ligne]. 18 avril 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

ASHISH, Rana, 2018. Understanding OpenAI Gym. In : *Medium* [en ligne]. 23 mars 2018. [Consulté le 17 août 2020]. Disponible à l'adresse : https://medium.com/@ashish_fagna/understanding-openai-gym-25c79c06eccb.

ASHISH, Rana, 2019. Introduction : Reinforcement Learning with OpenAI Gym. In : *Medium* [en ligne]. 1 août 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2.

BENZAKI, Younes, 2017. Régression linéaire en Python par la pratique. In : *Mr. Mint : Apprendre le Machine Learning de A à Z* [en ligne]. 17 avril 2017. [Consulté le 17 août 2020]. Disponible à l'adresse : https://mrmint.fr/regression-lineaire-python-pratique.

BHEEMAIAH, Kariappa, ESPOSITO, Mark et TSE, Terence, 2017. Deep learning, des réseaux de neurones pour traiter l'information. In : *The Conversation* [en ligne]. 2 mai 2017. [Consulté le 17 août 2020]. Disponible à l'adresse : http://theconversation.com/deep-learning-des-reseaux-de-neurones-pour-traiter-linformation-76055.

BROWNLEE, Jason, 2019. A Gentle Introduction to Cross-Entropy for Machine Learning. In : *Machine Learning Mastery* [en ligne]. 20 octobre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://machinelearningmastery.com/cross-entropy-for-machine-learning/.

CALLE, 2020. *Mean squared error (MSE) loss function on the Peltarion Platform* [en ligne]. 3 avril 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.youtube.com/watch?v=aVf9ZnX_Vj4.

CHAOUCH, Yannis, 2020. Identifiez les différents types d'apprentissage automatiques. In : *OpenClassrooms* [en ligne]. 18 mars 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://openclassrooms.com/fr/courses/4011851-initiez-vous-au-machine-learning/4020611-identifiez-les-differents-types-dapprentissage-automatiques.

DAVIS, Warren, 2020. What is a tensor? In : [en ligne]. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.physlink.com/education/askexperts/ae168.cfm.

EXXACT, Corporation, 2020. PyTorch vs TensorFlow in 2020: What You Should Know. In : *Exxact* [en ligne]. 23 janvier 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://blog.exxactcorp.com/pytorch-vs-tensorflow-in-2020-what-you-should-know-about-these-frameworks/.

FAGGELLA, Daniel, 2020. What is Machine Learning? - An Informed Definition. In : *Emerj* [en ligne]. 26 février 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://emerj.com/ai-glossary-terms/what-is-machine-learning/.

GARCIA, Sara Iris, 2018. An introduction to Gradient Descent Algorithm. In : *Medium* [en ligne]. 3 juin 2018. [Consulté le 17 août 2020]. Disponible à l'adresse : https://medium.com/@montjoile/an-introduction-to-gradient-descent-algorithm-34cf3cee752b.

GILIAN JAKOB, Kieser, 2019. Leading open source ML advancements: an introduction to PyTorch. In : *CircleCI* [en ligne]. 12 mars 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://circleci.com/blog/leading-open-source-ml-advancements-an-introduction-to-pytorch/.

HE, Horace, 2019. The State of Machine Learning Frameworks in 2019. In : *The Gradient* [en ligne]. 10 octobre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

HOSSAM H. SULTAN, 2019. FIGURE 7. ReLU activation function. In : *ResearchGate* [en ligne]. Mai 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.researchgate.net/figure/ReLU-activation-function_fig7_333411007.

JANNER, Michael, 2019. Model-Based Reinforcement Learning:Theory and Practice – The Berkeley Artificial Intelligence Research Blog. In : [en ligne]. 1 décembre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://bair.berkeley.edu/blog/2019/12/12/mbpo/.

JORDAN, Jeremy, 2017. Gradient descent. In : *Jeremy Jordan* [en ligne]. 14 juillet 2017. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.jeremyjordan.me/gradient-descent/.

JORDAN, Jeremy, 2018a. Normalizing your data (specifically, input and batch normalization). In : [en ligne]. 26 janvier 2018. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.jeremyjordan.me/batch-normalization/.

JORDAN, Jeremy, 2018b. Setting the learning rate of your neural network. In : *Jeremy Jordan* [en ligne]. 2 mars 2018. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.jeremyjordan.me/nn-learning-rate/.

MITCHELL, Tom M, 2006 apr. J.-C. *The Discipline of Machine Learning*. S.l.

MITCHELL, Tom M, 2006. *Mitchell - The Discipline of Machine Learning.pdf* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020 a]. Disponible à l'adresse : http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf.

NG, Andrew, 2020. Machine Learning by Stanford University. In : *Coursera* [en ligne]. 17 août 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.coursera.org/learn/machine-learning.

OPENAI, BERNER, Christopher, BROCKMAN, Greg, CHAN, Brooke, CHEUNG, Vicki, DĘBIAK, Przemysław, DENNISON, Christy, FARHI, David, FISCHER, Quirin, HASHME, Shariq, HESSE, Chris, JÓZEFOWICZ, Rafal, GRAY, Scott, OLSSON, Catherine, PACHOCKI, Jakub, PETROV, Michael, PINTO, Henrique Pondé de Oliveira, RAIMAN, Jonathan, SALIMANS, Tim, SCHLATTER, Jeremy, SCHNEIDER, Jonas, SIDOR, Szymon, SUTSKEVER, Ilya, TANG, Jie, WOLSKI, Filip et ZHANG, Susan, 2019. Dota 2 with Large Scale Deep Reinforcement Learning. In : *arXiv:1912.06680 [cs, stat]* [en ligne]. 13 décembre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : http://arxiv.org/abs/1912.06680.

PERROTTA, Paolo, 2020. Grokking the Cross Entropy Loss. In : *Medium* [en ligne]. 30 avril 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://levelup.gitconnected.com/grokking-the-cross-entropy-loss-cda6eb9ec307.

RASHKA, Sebastien, 2020. Gradient Descent and Stochastic Gradient Descent - mlxtend. In : [en ligne]. 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/.

ROBERT, Kleim, 2019. How to Train a Basic Perceptron Neural Network - Technical Articles. In : [en ligne]. 24 novembre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.allaboutcircuits.com/technical-articles/how-to-train-a-basic-perceptron-neural-network/.

SALLOUM, Ziad, 2020. Model Based Reinforcement Learning. In : *Medium* [en ligne]. 1 mai 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://towardsdatascience.com/model-based-reinforcement-learning-cb9e41ff1f0d.

SENTDEX, 2019a. Python Programming Tutorials. In : [en ligne]. 5 juin 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://pythonprogramming.net/q-learning-analysis-reinforcement-learning-python-tutorial/?completed=/q-learning-algorithm-reinforcement-learning-python-tutorial/.

SENTDEX, 2019b. Python Programming Tutorials. In : [en ligne]. 23 septembre 2019. [Consulté le 17 août 2020]. Disponible à l'adresse : https://pythonprogramming.net/introduction-deep-learning-neural-network-pytorch/.

SILVER, David, 2015. *Silver - Lecture 1 Introduction to Reinforcement Learning.pdf* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020 b]. Disponible à l'adresse : https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf.

SILVER, David, 2015. *Silver - Lecture 2 Markov Decision Processes.pdf* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020 c]. Disponible à l'adresse : https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf.

SILVER, David, 2015a. Lecture 1 : Introduction to Reinforcement Learning. In : *Introduction to Reinforcement Learning*. S.l. 13 mai 2015.

SILVER, David, 2015b. Lecture 2 : Markov Decision Processes. In : *Markov Decision Processes*. S.l. 13 mai 2015.

SILVER, David, 2015c. *RL Course by David Silver - Lecture 1 : Introduction to Reinforcement Learning* [en ligne]. 13 mai 2015. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.youtube.com/watch?v=2pWv7GOvuf0.

SILVER, David, 2015d. *RL Course by David Silver - Lecture 2: Markov Decision Process* [en ligne]. 13 mai 2015. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.youtube.com/watch?v=lfHX2hHRMVQ&t=173s.

SIVARAJKUMAR, Sonish, 2019. ReLU — Most popular Activation Function for Deep Neural Networks. In : *Medium* [en ligne]. 15 mai 2019. [Consulté le 17 août 2020]. Disponible à l’adresse : https://medium.com/@sonish.sivarajkumar/relu-most-popular-activation-function-for-deep-neural-networks-10160af37dda.

STEPHEN, Welch, 2020. Pytorch vs. TensorFlow: What You Need to Know. In : *Udacity* [en ligne]. 11 mai 2020. [Consulté le 17 août 2020]. Disponible à l’adresse : https://blog.udacity.com/2020/05/pytorch-vs-tensorflow-what-you-need-to-know.html.

SUTTON, Richard S, BARTO Andrew G., 2014, 2015. *Sutton et Barto - 2014 - Reinforcement Learning An Introduction.pdf* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020 d]. Disponible à l’adresse : https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

SUTTON, Richard S et BARTO, Andrew G, 2014. *Reinforcement Learning: An Introduction*. S.l.

TAMBET, Matiisen, 2015. Demystifying Deep Reinforcement Learning | Computational Neuroscience Lab. In : [en ligne]. 19 décembre 2015. [Consulté le 17 août 2020]. Disponible à l’adresse : https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/.

THE CODING, Train, 2017. *3.4 Régression linéaire avec l’algorithme du gradient - Intelligence et apprentissage* [en ligne]. 31 mai 2017. [Consulté le 17 août 2020]. Disponible à l’adresse : https://www.youtube.com/watch?v=L-Lsfu4ab74&t=988s.

UDVARI, Tibor, [sans date]. Les réseaux de neurones. In : [en ligne]. [Consulté le 17 août 2020]. Disponible à l’adresse : https://ml4a.github.io/ml4a/fr/neural_networks/.

V, Avinash Sharma, 2017. Understanding Activation Functions in Neural Networks. In : *Medium* [en ligne]. 30 mars 2017. [Consulté le 17 août 2020]. Disponible à l’adresse : https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

WIKIPÉDIA, 2020a. *Algorithme du gradient stochastique* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l’adresse : https://fr.wikipedia.org/w/index.php?title=Algorithme_du_gradient_stochastique&oldid=171686970.

WIKIPÉDIA, 2020b. *Apprentissage automatique* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Apprentissage_automatique&oldid=173595588.

WIKIPÉDIA, 2020c. *Artificial neural network* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=972940778.

WIKIPÉDIA, 2020d. *Calculus* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://simple.wikipedia.org/w/index.php?title=Calculus&oldid=6994069.

WIKIPÉDIA, 2020e. *Calculus* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://en.wikipedia.org/w/index.php?title=Calculus&oldid=970671935.

WIKIPÉDIA, 2020f. *Go en informatique* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Go_en_informatique&oldid=166229357.

WIKIPÉDIA, 2020g. *Neuromorphologie* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Neuromorphologie&oldid=169228987.

WIKIPÉDIA, 2020h. *Q-learning* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Q-learning&oldid=173520672.

WIKIPÉDIA, 2020i. *Réseau de neurones artificiels* [en ligne]. S.l. : s.n. [Consulté le 17 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=R%C3%A9seau_de_neurones_artificiels&oldid=172267537.

WILL DOUGLAS, Heaven, 2020. DeepMind's AI can now play all 57 Atari games—but it's still not versatile enough. In : *MIT Technology Review* [en ligne]. 1 avril 2020. [Consulté le 17 août 2020]. Disponible à l'adresse : https://www.technologyreview.com/2020/04/01/974997/deepminds-ai-57-atari-games-but-its-still-not-versatile-enough/.

Rapport-Gratuit.com