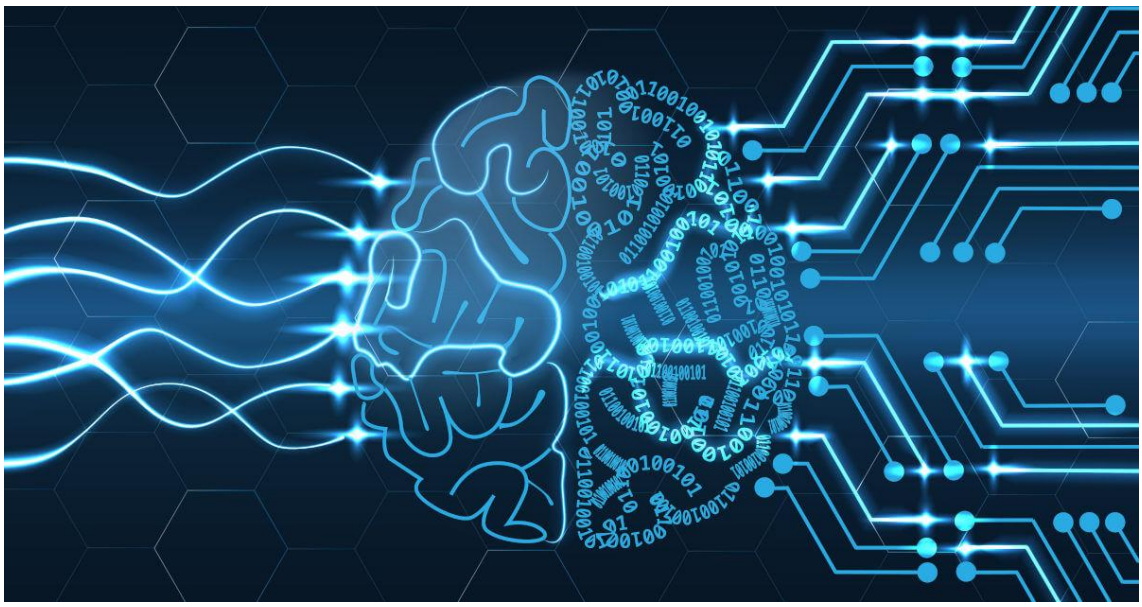
h e g

Haute école de gestion
Genève

Étude et la mise en place d'algorithmes pour l'apprentissage par renforcement.



Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Raphaël Lopes

Conseiller au travail de Bachelor :

Alexandros Kalousis

Carouge le 15.10.2022

Haute École de Gestion de Genève (HEG-GE)

Filière informatique de gestion

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre Bachelor d'informatique de gestion.

L'étudiant a envoyé ce document par email à l'adresse remise par son conseiller au travail de Bachelor pour analyse par le logiciel de détection de plagiat URKUND, selon la procédure détaillée à l'URL suivante : https://www.urkund.com .

L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève, 15.10.2022

Raphaël Lopes

Remerciements

La réalisation de travail de Bachelor a été possible grâce au concours de plusieurs personnes à qui je voudrais témoigner toute ma gratitude.

Je voudrais tout d'abord adresser toute ma reconnaissance à mon conseiller au travail de Bachelor Alexandros Kalousis pour ses judicieux conseils, qui ont contribué à alimenter ma réflexion et qui fut le premier à me faire découvrir le sujet qui a guidé ce travail.

Je voudrais exprimer ma reconnaissance envers les amis et membres de ma famille qui m'ont apporté leur soutien moral et intellectuel tout au long de ma démarche.

Enfin, je tiens à témoigner toute ma gratitude à maman pour son aide, sa confiance et son soutien inestimable.

Résumé

Aujourd'hui, l'informatique a une place très importante dans la société. L'informatique est omniprésente dans plein de domaines, que ce soit l'industrie, la santé, la science, et bien d'autres. De ce fait, un nombre astronomique de données sont générées constamment. En conséquence, le domaine de l'étude et l'analyse de celle-ci sont essentiels et il se nomme la science des données. Il y a aussi une demande émergente d'automatisation de tâche pour gagner en performance. Il est donc important de connaitre les bases dans ce domaine. C'est pourquoi ce travail consiste à découvrir l'apprentissage par renforcement en commençant par une introduction au Machine Learning.

L'objectif de ce travail est dans un premier temps d'introduire le domaine de Machine Learning pour ensuite, comprendre ce qu'est l'apprentissage par renforcement. C'est un sujet très vaste et complexe, c'est pourquoi le but était pour moi de le comprendre et d'expliquer la base de l'apprentissage par renforcement.

La deuxième partie consiste à analyser des algorithmes d'apprentissage par renforcement. Afin, d'effectuer des recherches sur ces algorithmes et de pouvoir comprendre leurs fonctionnements, leurs différences et analyser ces algorithmes.

La troisième étape quant à elle consiste à apprendre l'implémentation de ces algorithmes d'apprentissage par renforcement grâce à python et Pytorch. En commençant par comprendre la Library Pytorch et de pourvoir par la suite, implémentée des agents utilisant ces algorithmes dans le même environnement.

Enfin, la dernière étape est d'analyser le comportement des agents implémenté dans l'environnement. Observer les résultats individuellement et ensuite comparer les résultats entre les deux agents pour les interpréter.

Table des matières

Liste des tableaux

Liste des figures

1. Machine Learning

Le terme Machine Learning est un concept souvent confondu avec le terme intelligence artificiel or, celui-ci est un sous-domaine à l'intelligence artificiel. L'intelligence artificielle se définit comme la possibilité à un ordinateur d'imiter un humain à accomplir tâche.

La Machine Learning quant à lui consiste à répliquer des comportements intelligents d'un humain à apprendre. Il a été défini par le Arthur Samiel pionnier du Machine Learning *« Field of study that gives computers the ability to learn without being explicitly programmed. » (Arthur Samiel 1950)* qui se traduit par un domaine d'étude qui donne à un ordinateur la capacité d'apprends sans être explicitement programmé pour.

Nous pouvons donc voir la différence majeure entre les deux qui est que l'intelligence artificielle est plus globale. Il prend en compte toute machine qui peut résoudre une tâche qu'elle soit programmée « explicitement » à le faire ou non, alors que Machine Learning ne prend en compte que les machines qui ont été programmées à apprendre, mais pas « explicitement » à résoudre une tâche. Ce travail se focalisera sur une catégorie sur ce dernier.

Nous pouvons observer trois grandes catégories pour le Machine Learning :

- Apprentissage supervisé
- Apprentissage non supervisé
- Apprentissage par renforcement

1.1 Apprentissage supervisé

L'apprentissage supervisé consiste à prendre un modèle qui a un entrainement avec une « supervision ». Pour l'entrainer, nous allons lui donner un « jeu de données d'entrainement » qui contient les résultats connus à l'avance, il va par la suite en faire une prédiction. Dans un deuxième temps il va comparer sa prédiction par rapport à celle que nous lui avons donnée et si sa prédiction est juste ou fausse, il va modifier sa manière de prédire pour avoir des résultats de plus en plus corrects. Lorsque le modèle arrive à déterminer un pourcentage de prédiction satisfaisant. Nous allons ensuite fournir des données sans le résultat connu à l'avance et laisser le modèle nous retourner la prédiction. L'apprentissage supervisé se nomme ainsi, car, lors de la phase d'apprentissage, nous avons un superviseur qui va fournir des données connues qui permettront de déterminer ce qui fait que nous « supervisons » la phase d'apprentissage.

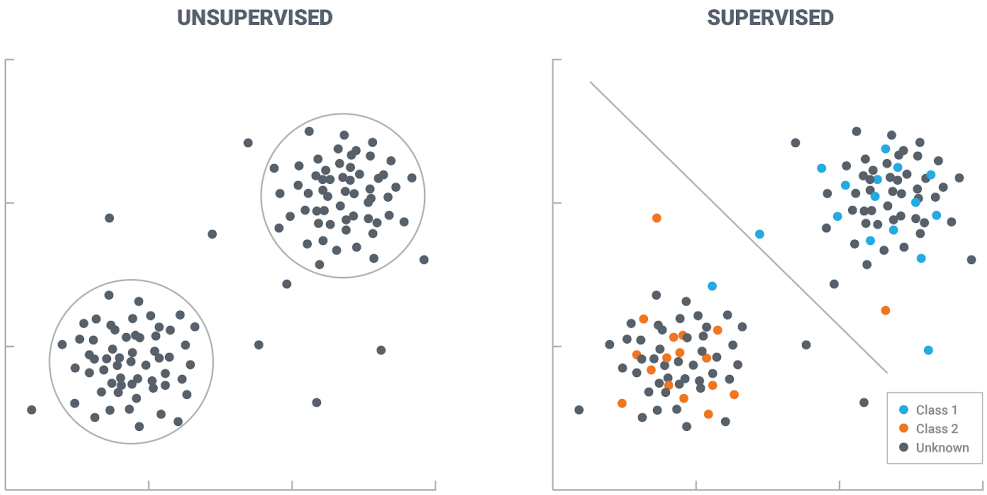
L'apprentissage supervisé est utilisé pour plein de tâches différentes. Pour donner un exemple, nous allons utiliser deux cas simples où l'on peut l'utiliser. Un de ces cas est lorsque nous voulons faire de la classification de donner, c'est-à-dire définir des classes et définir de quelle donnée cette classe fait partie. Exemple, est-ce que cette image est une photo de chien ou une photo de chat. L'autre utilisation est dans la régression, c'est lorsque nous souhaitons prédire la une tendance par rapport un jeu de données.

1.2 Apprentissage non supervisé

La deuxième catégorie est l'apprentissage non supervisé. Dans l'apprentissage non supervisé, nous allons fournir un jeu de données complètes sans résultats connus. La machine va par la suite tenter de regrouper les données par rapport à des patterns. Par mimétisme, il va essayer de légèrement modifier les données pour trouver des patterns pour regrouper des données. Lorsque le modèle considère des regroupements à chaque nouvelle donnée entrée il va tenter de trouver dans quel groupe elle appartint. À l'inverse de l'apprentissage supervisé, a aucun moment un superviseur n'est présent pour fournir un pattern ou un sens pour classifier les données. C'est le modèle qui le détermine par lui-même.

L’apprentissage non supervisé est utilisé plein de groupes de cas pratique. Comme pour l’apprentissage supervisé nous allons donner exemple, le premier est le clustering, il s’agit de regrouper des données similaires et définir un groupe par rapport à ces données. C’est le cas similaire à la classification de l’apprentissage supervisé or, en non supervisé, il n'y a pas un superviseur qui définit le type de classification. Le deuxième cas est la réduction de dimension qui consiste à exprimer les données sur des dimensions réduites. Cette méthode est souvent utilisée pour montrer visuellement la relation entre des variables et une dimension.

Figure 1 : un exemple de différence entre le clusturing non supervisé (à gauche) et la classification supervisée (à droite)



(awtomated.com, 2019)

2. Apprentissage par renforcement

La dernière catégorie est l'apprentissage par renforcement. L'apprentissage par renforcement a pour but d'apprendre par soi-même à accomplir une tâche avec la meilleure solution possible. Dans l'apprentissage par renforcement, nous avons un agent. Un agent est celui qui va faire des choix, un agent peut être soit un humain soit une machine et qui va apprendre à accomplir une tâche. L'agent se situera donc dans un environnement. Il va commencer dans l'état initial S de notre environnement. Tous les états sont reliés à un moment dans le temps et qu'on notera ainsi S_t. L'agent va observer l'environnement et prendre une action dans l'environnement A. Comme pour l'état, une action doit être liée au temps on le notera comme ceci A_t. Lorsque celui-ci aura pris l'action l'environnement. Ce dernier retournera une récompense liée au temps R_t qui détermine « l'efficacité » de l'action choisie et l'état que l'agent se trouve S_{t+1} en prenant l'action. L'agent doit apprendre grâce aux interactions qu'il fait avec l'environnement et au fil du temps exécuter des actions qui vont favoriser la récompense maximum dans un état pour mieux exécuter une tâche.

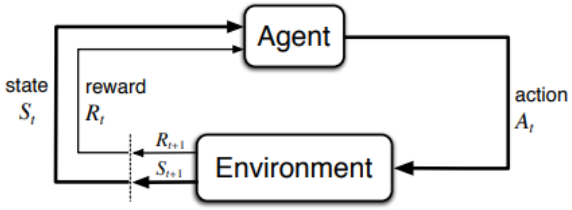
2.1 Processus de décision Markovien

La première base importante à comprendre dans l'apprentissage par renforcement est le processus de décision Markovien. Il s'agit d'une représentation mathématique d'un processus de décision stochastique, dans lequel une action va influer sur la récompense d'une décision et le prochain état de l'environnement. Il nous permet aussi de décrire et d'analysé n'importe quelle tâche de manière simple grâce à quadruplet *(Maxim Lapan, 2020)*.

$$(S, A, P, R)$$

- S Ensemble d'états : ceci représente est tous les états possibles dans laquelle agent peut se retrouver.
- A Ensemble d'actions : Nous avons ici toutes les actions que l'agent peut prendre dans l'environnement.
- P Probabilité de transition : l'ensemble de transitions avec une probabilité pour n'importe quel état s a s' en utilisant l'action a.
- R Récompense : l'ensemble des récompenses de pair de (s, a). C'est-à-dire l'ensable des récompenses de chacune des pairs action état.

Figure 2 : Cycle d'apprentissage par renforcement.



(towardsdatascience.com, 2019)

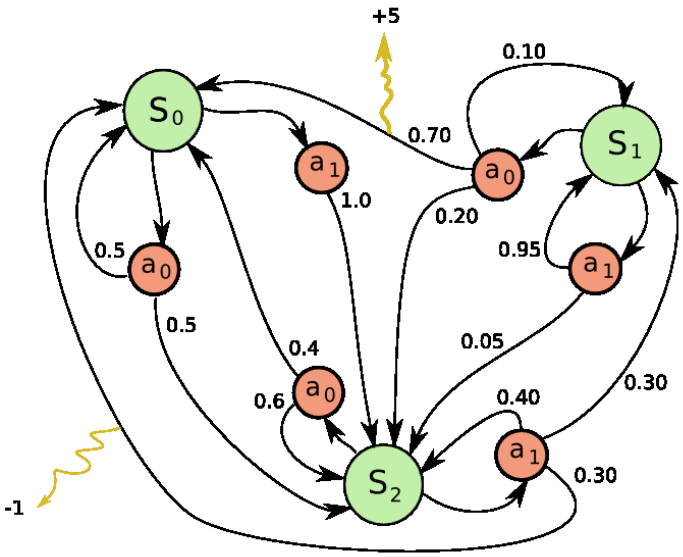
Dans ce processus un agent commencera par observer l'état de l'environnement S_t, il prendra ensuite une décision d'une action qu'il appliquera dans l'environnement A_t. Enfin l'environnement retournera la récompense de l'action R_{t+1} et le nouvel état de l'environnement S_{t+1}. La boucle se répète indéfiniment jusqu'à ce que, soit l'agent à finit la tâche ou que celle-ci s'arrête par un échec. Nous pouvons aussi noter une propriété intéressante du processus de décision Markovien, c'est que celui-ci utilise l'état actuel pour définir le prochain et donc n'a pas de mémoire. Ceci permet que la suite d'état que l'agent a choisi pour arriver à un état n'ait pas pris en compte et n'affectera donc pas notre prochaine décision. Ce qui est la propriété qui définit les processus Markoviens. *(Maxim Lapan, 2020)*

2.1.1 Type de décision Markovien

Le processus de décision Markovien peut se diviser en deux groupes les épisodiques ou les séquentiels *(Maxim Lapan, 2020)*.

- Un environnement épisodique contient une finalité naturelle, cette finalité peut se définir par n'importe quoi que ce soit du temps, un état final, une limite épisode, etc.
- Un environnement séquentiel quant à lui n'a pas de fin de l'épisode. Il va continuer indéfiniment.

Figure 3 : exemple de processus de décision Markovien



(wikipedia.org, 2017)

2.2 La trajectoire

La trajectoire est la suite d'éléments générée au fils du temps que l'agent agit sur l'environnement. Par exemple : nous allons commencer dans l'état 0 S_0. Ensuite l'agent va choisir une action on notera donc A_0. Après cette action l'environnement va nous retourner la récompense de l'action et le prochain état R_1 et S_1. On répète cette boucle tout au long du travail. On écrit la trajectoire avec la lettre grecque tau comme ceci :

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, \ldots$$

2.3 Le gain

Ce que l'agent cherche est de maximiser les récompenses. Il s'agira à chaque fois de choisir les actions qui retournent les récompenses les plus fortes. Or, il se peut qu'une action qui est retournée ait une bonne récompense, mais que dans l'état où l'agent se trouve nous ayons plus de récompenses intéressantes après cette action. C'est pourquoi il faut prendre en compte le gain. Dans l'apprentissage par renforcement on observe le gain appeler return en anglais est la somme de toutes les récompenses d'un épisode :

$$G = R_0 + R_1 + R_2 + \ldots + R_T$$

Dans la formule le symbole T représente le dernier état.

Maintenant notre agent aura pour but de favoriser le gain plutôt que la récompense. Néanmoins le problème avec cette formule, l'agent n'a pas de motivation à chercher le moyen le plus rapide à accomplir la tâche, car la somme de toutes les possibilités à parcourir peut-être égale en prenant le chemin le plus court ou le plus long, c'est pourquoi nous devons trouver un moyen de motiver l'agent à prendre en compte la meilleure récompense tout en favorisant tout de même le gain. C'est là que l'on va introduire un facteur de dévaluation qui va nous permettre de favoriser la récompense tout en prenant en compte le gain.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Dans cette formule y est le facteur dévaluation qui est un chiffre compris entre 0 et 1, plus le chiffre se rapproche de 1 plus nous allons donner de l'importance aux futurs. Grâce à cette formule plus la récompense va être loin moins elle aura de l'importance ce qui permet d'avoir ce compromis entre la récompense (court terme) et le gain (long terme).

2.4 Politique

La politique est une fonction qui décide quelle est l'action à faire dans un état qui maximise les gains. La politique s'exprime avec le symbole pi et se note ainsi :

$$\pi : S \rightarrow A$$

Une politique explique donc quelle action va prendre l'agent. Une politique peut avoir des probabilités pour des actions. C'est-à-dire une politique peut très bien dire qu'il y a 50% de chance de prendre une action et 50% de chance de prendre l'autre la seule importance est que la somme des probabilités de toutes les actions possibles doit être égale à 1 (100%).

$$\sum_{a} \pi(a|s) = 1$$

La notation $\pi(a|s)$ exprime la probabilité que la politique choisisse l'action a étant donné l'état s.

Pour n'importe quelle tâche il y a donc une politique optimale qui s'exprime ainsi :

$$\pi_*$$

Nous savons que l'agent a pour but de maximiser les gains et aussi de trouver la politique optimale d'une tâche *(Maxim Lapan, 2020)*.

2.5 Fonction de la valeur de l'état et Q-function

On peut maintenant définir la fonction de la valeur état, cette fonction exprime le gain que l'on peut avoir dans un état en suivant une politique π jusqu'à la fin de l'épisode. On la notera comme ceci *(Maxim Lapan, 2020)* :

$$V_\pi(s) = E[G_t|S_0 = s]$$

$$= E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} |S_0 = s\right]$$

La valeur d'un état est donc reliée à une politique et celle-ci va prendre des actions, deux politiques différentes peuvent ne pas retourner les mêmes gains attendus vu qu'elles ne prennent pas forcément les mêmes actions.

Ci-dessous nous pouvons voir la notation expliquant la fonction de la valeur d'état qui suit la ou les politiques optimaux $*$ et retournera le meilleur gain dans l'état S

$$V_*(s)$$

La deuxième fonction que l'on peut définir est la Q-function. Celle-ci définit la « Qualité » d'une action prise dans un état en retournant le gain de celle-ci. La fonction de valeur état-action peut se référer aussi en tant que Q-function et le résultat de cette Q-function sera donc la Q-value *(Maxim Lapan, 2020)*.

$$Q(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$$

La formule de la Q-function en suivant une politique :

$$Q_\pi(s,a) = E[G_t | S_0 = s, A_0 = a\,]$$

$$= E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\,|S_0 = s, A_0 = a\right]$$

Comme pour la fonction valeur état nous avons ici la notation de la ou les politiques optimaux qui dans un état s en prenant action a et en suivant la ou les politiques optimaux retournera la meilleure Q-Value.

$$Q_*(s,a)$$

2.6 Équation d'optimalité de Bellman

Comme expliqué dans la politique, il faut maintenant un agent performant qui doit maximiser les gains et aussi trouver la politique optimale d'une tâche. Le problème étant que si on souhaite maximiser les gains il nous faut donc la politique optimale et si on souhaite maximiser la politique, il faut les gains maximaux. La solution est dans le fait qu'une des propriétés fondamentales de la Q-function optimal est qu'elle doit satisfaire l'équation d'optimalité de Bellman *(Maxim Lapan, 2020)*.

$$Q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} Q_*(s',a')\right]$$

Cette équation explique, que pour chaque couple état-action, en commençant par l'état s en prenant l'action a et en suivant la politique optimale on retournera la récompense attendue G_{t+1} additionné par le gain maximum attendu en prenant l'action a' pour la prochaine paire état-action. Grâce à cette équation, nous pouvons faire ce qu'on appelle de la Policy Iteration. La Policy Iteration consiste à itérativement choisir la meilleure action en fonction de la politique, mettre à jour la fonction de valeur, pour ensuite, mettre à jour la politique. Jusqu'à arriver à la politique optimale et le gain maximal. Ce qui permet de faire une tâche de manière optimale en *(Maxim Lapan, 2020)*.

2.7 Table Q Learning

Pour augmenter le gain, nous allons prendre une table à deux dimensions pour stocker les Q.-value. Les deux dimensions sont bien sûr les actions et les états de notre tâche. Le problème étant qu'initialement nous avons partout la valeur 0 dans les tables. C'est pourquoi il faut donc explorer la tâche pour trouver des informations et mettre à jour notre table. Certes nous devons explorer, mais il faut aussi « exploiter » les chemins explorés lorsque nous aurons rempli la table pour trouver la solution optimale. C'est pourquoi nous allons utiliser une politique epsilon gourmand. Epsilon est une valeur qui diminue à chaque épisode. À chaque étape de l'épisode, nous allons générer un nombre aléatoire entre 0 et 1 si celui-ci est en dessous de la valeur de l'epsilon alors l'agent va « explorer » en prenant une décision aléatoire et stocker la valeur dans la table grâce à une formule qui va être présentée plus tard. L'épisode se finit, on réduit légèrement la valeur epsilon et un nouveau recommence. Si la valeur générée est au-dessus alors il va « exploiter » la meilleure solution, c'est-à-dire choisir l'action avec le meilleur Q-value dans l'état qu'il se trouve grâce à une formule. Cette formule, qui sera expliquée après nous permettra de mettre à jour dans notre table les valeurs afin d'accomplir la tâche.

Pour comprendre la formule permettant de choisir l'action avec le meilleur Q-value dans un état, nous devons d'abord comprendre le taux d'apprentissage noté α. Le taux d'apprentissage est comme pour epsilon est une valeur qui se situe entre 0 et 1 qui permet de déterminer à quel point nous donnons de l'importance à l'ancienne Q-value par rapport à la nouvelle calculer. Plus le chiffre est grand, moins l'ancienne Q-value à de l'importance *(Maxim Lapan, 2020)*.

Nous pouvons maintenant mettre à jour la Q-value de la table. Pour se faire nous allons prendre une décision, observer la récompense de celle-ci et ensuite appliquer l'équation de Bellman pour mettre à jour le couple état action que nous venons de prendre afin de permettre de converger vers la Q-value optimum. Pour le faire, nous allons donc itérativement comparer la perte de la Q-value et la Q-value optimum et ensuite le mettre à jours notre table.

$$Q_*(s,a) - Q(s,a) = perte$$
$$E\left[R_{t+1} + \gamma \overset{max}{a'} Q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] = perte$$

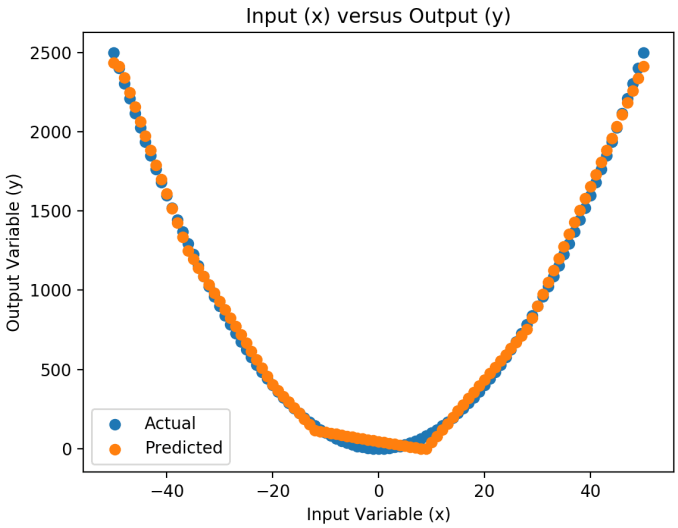
Pour mettre à jours cette Q-value nous allons utiliser une formule qui fait partie de la classe des Temporal Differences Learning. Cette classe regroupe toutes les méthodes dont le but est de trouver la Q-value où la valeur d'état optimal en se basant sur les estimations actuelles. Cette formule permet d'estimer la Q-value en « off-Policy » ce qui veut dire que nous n'allons pas utiliser la même politique pour l'exploration et l'exploitation. L'agent va explorer avec une politique et va exploiter pour apprendre la Q-function avec la politique optimal (*Maxim Lapan, 2020)* :

$$Q(s,a) = (1-\alpha)Q(s,a)) + \alpha\left(R_{t+1} + \gamma \overset{max}{a'}(s',a')\right)$$

3. Apprentissage par renforcement profond

L'apprentissage par renforcement profond se différencie de l'apprentissage par renforcement, car celui-ci ne va pas utiliser une table comme vue au-dessus, mais un réseau de neurones. Le problème des Tables Q Learning est qu'il faut stocker des tables, donc beaucoup d'information à stocker. Il est même impossible d'utiliser cette méthode dans le cas où les action et état sont continus or, l'apprentissage par renforcement profond utilise un Function Approximation qui permet d'estimer la fonction de la valeur des états tout en demandant beaucoup moins de mémoire. De plus, il permet de généraliser la valeur des états d'une nouvelle manière. Dans ce chapitre nous allons voir ce qu'est un réseau de neurones et les Function Approximation et à quoi ils servent dans l'apprentissage par renforcement profond.

Figure 4 : exemple de function approximation



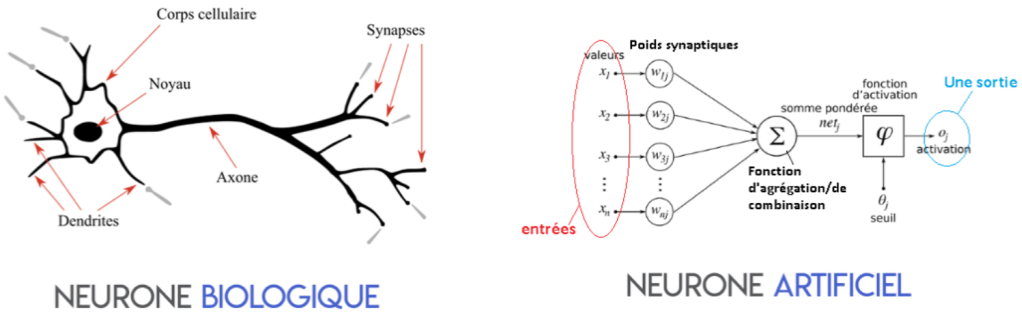
(machinelearningmastery.com, 2020)

Réseaux de neurones artificiels.

3.1.1 Introduction

Les réseaux de neurones artificiels (ANNs) sont une modélisation artificielle des neurones biologiques. On peut remarquer des similitudes dans le fonctionnement entre un réseau de neurones artificiel et biologique, mais il ne s'agit en rien d'une représentation exacte de ce dernier. Un neurone artificiel va donc recevoir des informations depuis ses points d'entrée par d'autres neurones puis les traiter pour ensuite les redistribuer aux autres neurones connectés aux sorties. On peut voir dans la figure ci-dessous : les similitudes entre un neurone biologique et artificiel. Dans un neurone biologique nous avons des dendrites qui sont les « entrés », nous avons ensuite le noyau qui est le cœur du neurone c'est lui qui va prendre des décisions et envoyer le signale vers l'axone qui est la « sortie » qui va se brancher aux autres neurones (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

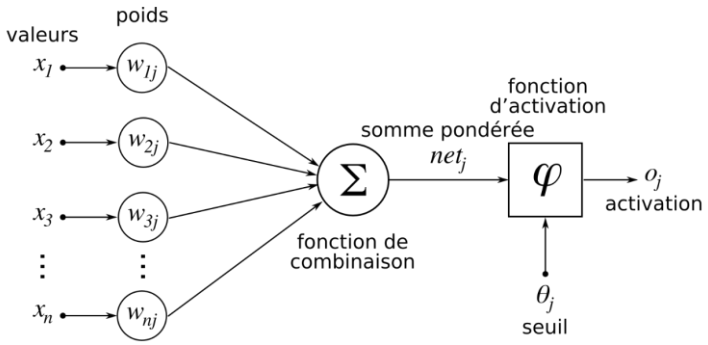
Figure 5 : comparaison entre un neurone biologique et artificiel



(Bastien Maureice, 2018)

3.1.2 Fonctionnement d'un neurone artificiel

Figure 6 : élément d'un neurone artificiel



(wikipedia.org, 2005)

Un neurone se définit par ce que l'on peut voir dans la figure ci-dessus. Nous allons observer attentivement chaque élément et leurs fonctionnements. Commençons par les valeurs X_i qui sont les valeurs que les autres neurones vont fournir à notre noyau. Avant de pouvoir intervenir dans notre noyau, nous avons un poids noté W_{ij} qui représente le poids du neurone i allant au neurone j. Le poids permettra de modifier la valeur retournée d'un neurone pour avoir de meilleur résultat. Ce processus sera expliqué plus tard. Le neurone recevra donc la valeur d'un neurone multiplier par son poids.

$$W_{ij} * X_i$$

Comme un neurone peut avoir une ou plusieurs entrées, la valeur sera donc la somme de toutes les valeurs multipliées par leurs poids.

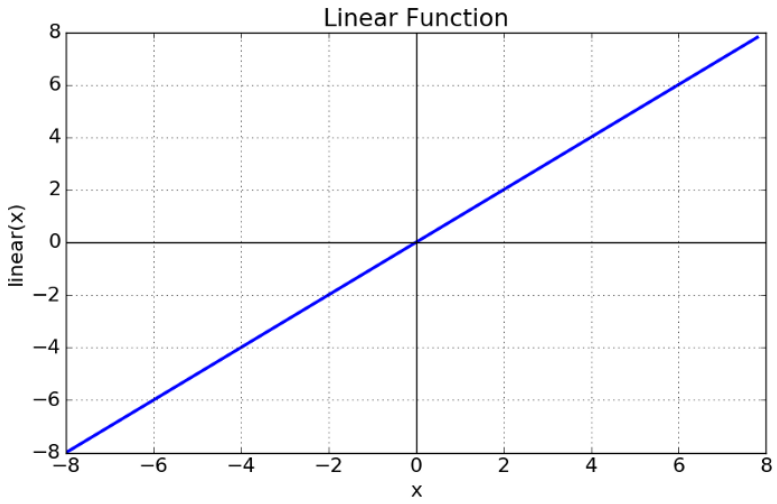
$$\sum_{i=1}^{n} W_{nj} * X_n$$

Notre neurone contient maintenant une valeur d'entrée. Nous pouvons donc maintenant utiliser cette valeur dans notre système (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

3.1.3 Fonctions d'activations

Un neurone biologique utilise des systèmes plus complexes qu'un neurone artificiel, mais le fonctionnement est similaire. Lorsque qu'un neurone reçoit des valeurs il doit faire un choix si oui ou non il envoie un signal et de quelle valeur. C'est là que la fonction d'activation existe. Cette fonction va prendre cette décision. Les fonctions d'activation peuvent se diviser en deux types linéaires et non linéaires. Pour commencer, nous allons parler d'une fonction linéaire. Il s'agit des plus simples et permettrons ensuite d'expliquer les plus complexes (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

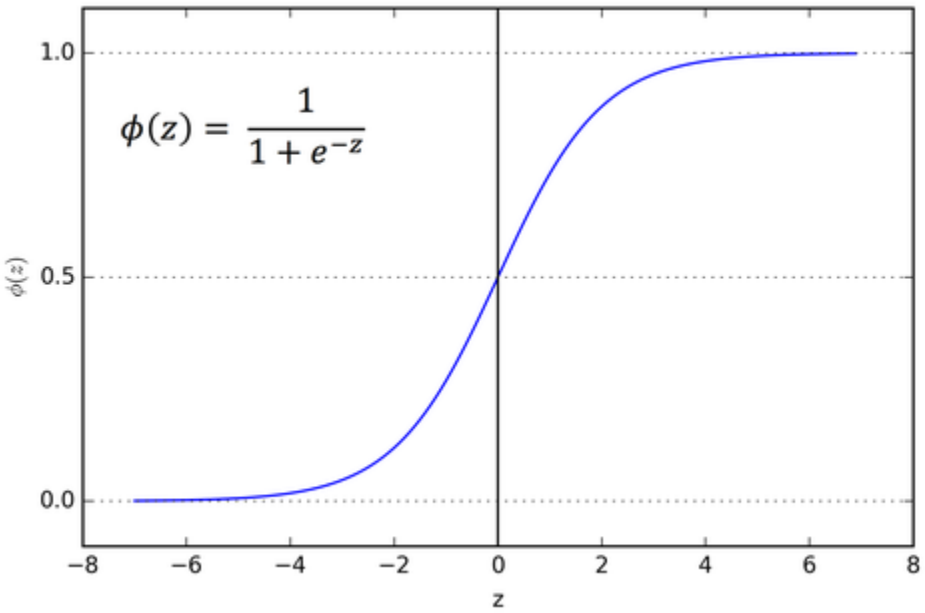
Figure 7: la fonction linéaire



(towardsdatascience.com, 2017)

Une fonction linéaire va donc consister d'une ligne et aura un intervalle $-\infty$ à. ∞. Dans l'axe x nous trouveront la somme des valeurs d'entrées pondérées, sur l'axe y la valeur envoyée en sortie du neurone. Le deuxième type est les fonctions non linéaires. Un exemple de fonction non linéaire est la fonction sigmoïde (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

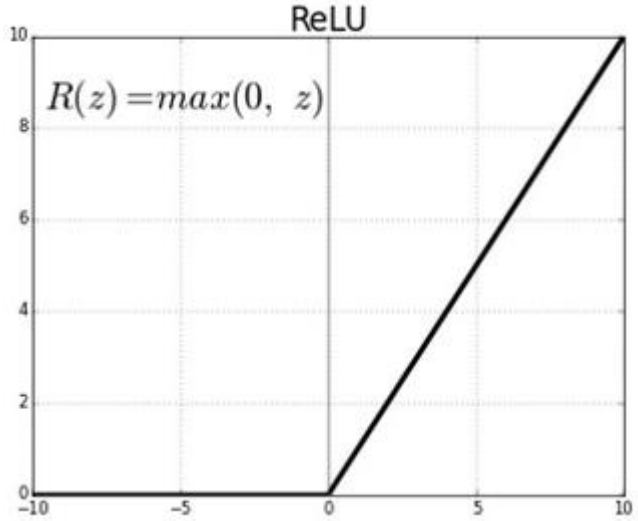
Figure 8: la fonction sigmoïde



(towardsdatascience.com, 2017)

La fonction sigmoïde à une forme de S et à un intervalle 0 à 1. Cette fonction est souvent utilisée pour la prédiction, vu que celle-ci est limitée de 0 à 1. Elle se note avec la formule incluse dans le graphique ci-dessus. Nous avons pour finir la fonction d'activation. La plus importante dans notre cas, car il s'agit de la fonction utilisée pour tout ce qui touche l'apprentissage. Elle se nomme la fonction ReLU (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

Figure 9: la fonction ReLU



(towardsdatascience.com, 2017)

Comme on peut le voir, elle retourne la valeur 0 tant que l'axe X ne contient pas de valeur positive. Elle fonctionne comme une fonction linéaire après 0 c'est-à-dire qu'elle ne change pas de valeur et peut arriver jusqu'à l'infini. Cette fonction a donc un intervalle de 0 à ∞ (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

3.1.4 Fonction de coût

Nous savons maintenant comment un neurone fonctionne. Le neurone est fonctionnel, mais il doit maintenant s'améliorer. Pour ce faire nous allons utiliser les fonctions de cout. Grâce à ses fonctions, nous pouvons déterminer la performance de notre réseau de neurones par rapport à une valeur attendue et modifier le poids de chacun des neurones dans notre réseau, pour que celui-ci pour nous retourne une valeur plus proche de la valeur attendue. Pour ce faire nous avons besoin de trois vecteurs. Le premier, est n qui représente la taille donnée. Le deuxième, y_i est la collection de valeur contenant la « vérité » pour la valeur. Enfin, $\hat{y}_i$ qui est la prédiction retournée par un neurone. Nous allons appliquer une formule pour déterminer « l'efficacité ».

La formule la plus utilisée est la Mean Square Error (MSE). Qui se traduit par erreur quadratique moyenne. Elle fait partie de la catégorie Régression Losses (pertes de régression) des fonctions de cout. Comme son nom l'indique, la Mean Square Error consiste à prendre la moyenne de la différence entre la valeur attendue et prédite au carré.

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

L'autre type est la Classification Losses (la perte de classification) à l'inverse de la régression celle-ci est utilisée lorsque nous voulons prédire des valeurs discrètes. Il y a par exemple la Cross Entropy Loss. C'est la plus commune pour la classification. Voici la formule

$$CrossEntropyLoss = -(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i))$$

La Cross Entropy Loss consiste à multiplier le logarithme de la probabilité prédite par les valeurs attendues. Un point intéressant de cette méthode et qu'elle pénalise beaucoup les prédictions confiantes, mais fausse. Nous avons ici deux exemples dans chaque catégorie il y a une longue liste de formule pour la fonction de coût, aucune n'est parfaite pour un cas elles ont chacune leurs forces et leurs faiblesses. Dans ce travail nous allons utiliser principalement la fonction Mean Square Error (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

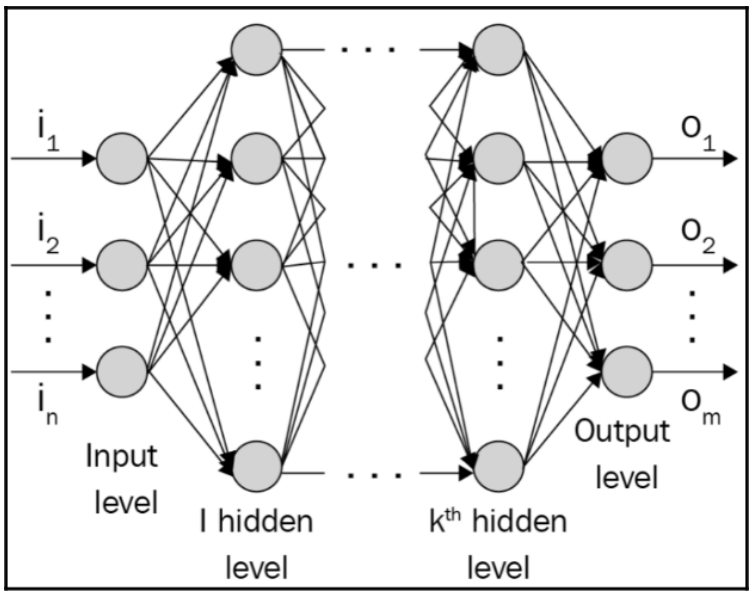
3.1.5 Réseaux de neurones

Un neurone tout seul ne peut rien faire, c'est quand ils sont connectés en réseaux qui fonctionnent. On peut visualiser un réseau de neurones comme une fonction mathématique avec des valeurs d'entrée et qui nous retourne une valeur de sortie. Nous avons trois composants clés dans un réseau de neurones.

- Couche d'entrée (input) : La couche d'entrées contient toutes les entrées possibles dans notre environnement.
- Couche cachée (hidden) : cette couche connecte la couche d'entrée à la couche de sorties. Il s'agit du cœur du réseau de neurones car, c'est grâce à cette couche que nous allons transformer les valeurs de la couche d'entré en sortie que nous attendons grâce au différent mécanisme expliqué au-dessus (fonction d'activation, poids, fonction de coût). Il peut y avoir une ou plusieurs couches connectées,
- Couche de sortie (output) : les sorties sont des noyaux qui retournent les valeurs qui sont attendues comme résultat. Le nombre de noyaux à la sortie sera toujours égal au nombre d'actions possibles dans la tâche à accomplir dans un cas discret. Dans un cas continu, nous n'avons qu'une sortie.

Il y a deux types de réseaux de neurones. Les Fully Connected, ce qui veut dire que chaque neurone à une connexion à tous les neurones dans la couche qui précède et qui suit. Les convolutif sont plus similaires au fonctionnement d'un cerveau, dans un réseau convolutif, il y a des connexions avec certains neurones pour créer des régions.

Figure 10 : exemple d'un réseau de neurones



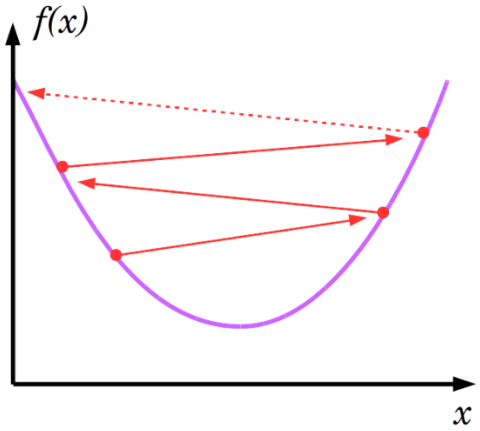
(Modern Computer Vision with PyTorch, P14)

Un point important à noter est la rétropropagation du gradient. Il s'agit de la méthode permettant aux réseaux de neurones d'apprendre. La rétropropagation du gradient agit à l'envers du fonctionnement normal du réseau de neurones, c'est-à-dire de droite à gauche. Il permet de mettre à jour les poids de chaque connexion entre neurones qui à l'initialisation du réseau sont aléatoires, permettant d'assurer une meilleure prédiction. Il s'agit de la base des méthodes du type Algorithme du gradient (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

3.1.6 Apprentissage des poids grâce à l'algorithme du gradient

Le problème avec la fonction de cout dans un réseau de neurones est la malédiction de la dimension. Voici un exemple pour comprendre le concept. Imaginons qu'ont créé un réseau Fully Connected avec 4 neurones d'entrée, 5 neurones en couche cachés, et un seul neurone en sortie. On se retrouve avec 25 poids à modifier. Si ses poids ont 1000 combinaisons possibles, cela va nous donner 10^{75} combinaisons. Il est donc impossible pour nous de tester toutes ces combinaisons rapidement. C'est pourquoi, pour gagner en temps de calcul, nous allons utiliser l'algorithme du gradient. Le concept est de prendre un point de mesure de calculer la dérivée de la fonction de cout pour déterminer la direction vers la valeur la plus basse. Pour ensuite prendre un autre point de mesure et recommencer les étapes. Ceci a pour but de trouver la valeur minimum, grâce à cela nous avons un compromis entre la meilleure valeur et le temps de calcule. Ci-dessous, on peut voir schéma de décent de gradient.

Figure 11 : Exempèle de décente de gradient



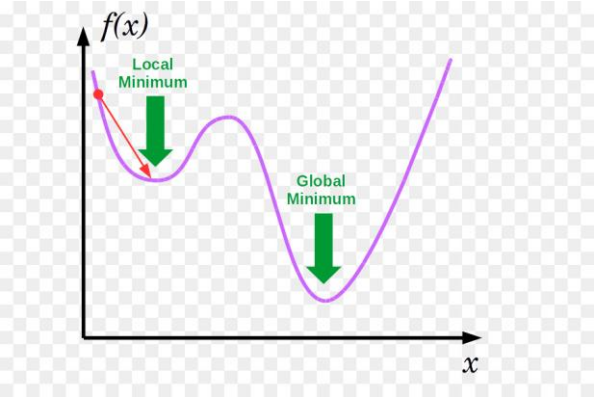
(lucidar.me, 2012)

Nous devons aussi déterminer le taux d'apprentissage que nous voulons choisir qui va être équivalent au « pas » que nous allons prendre pour la prochaine mesure. Si notre « pas » est trop petit alors nous allons prendre beaucoup de « pas » ce qui fait que nous allons prendre beaucoup plus de temps et perdre en performance. À l'inverse, si l'on prend un trop grand « pas », on risque de ne jamais pouvoir atteindre la valeur minimum, car nous allons faire des allers-retours en passant par-dessus cette valeur (*V Kishore Ayyadevara et Yeshwanth Reddy,2020)*.

Algorithme du gradient stochastique

Le problème de la descente de gradient est qu'il fonctionne correctement dans le cas d'une fonction convexe, mais dans une non convexe on a le risque de tomber dans la Local Minimum. Pour comprendre la Local Minimum, il faut comprendre que dans une fonction non convexe nous allons trouver plusieurs valeurs minimums. Un de ces minimums est la Global Minimum. C'est la valeur la plus basse de la fonction de cout et celle que nous voulons atteindre. Or, vu qu'il y a plusieurs miniums, il se peut que l'on ne se trouve pas dans la Global Minimum, dans ce cas, nous nous situons dans une Local Minimum et l'on ne récupéra pas la meilleure valeur (*V Kishore Ayyadevara et Yeshwanth Reddy,2020).* Voici un exemple.

Figure 12 : fonction non convexe

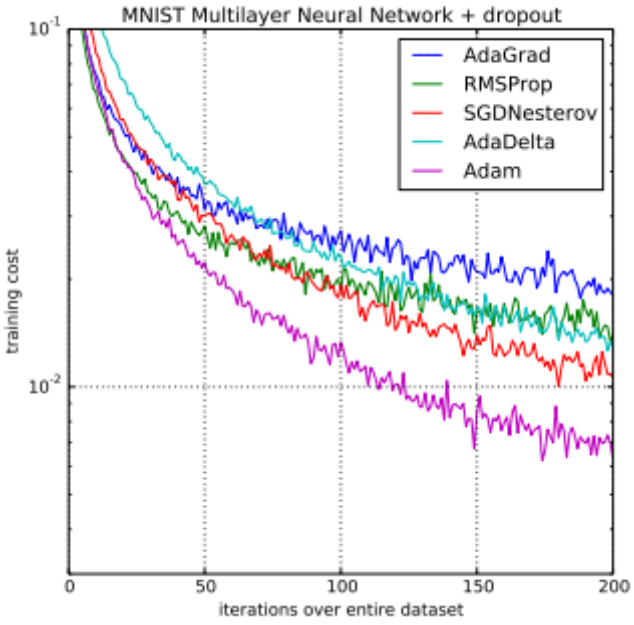


(freepng.fr, 2022)

3.1.7 Adam

Dans le domaine du Deep Learning, on utilise souvent le terme d'optimiseur pour parler des algorithmes qui utilisent un algorithme de descentes de gradient stochastique. Dans notre cas nous allons utiliser Adam. Adam est l'un des optimiseurs les plus performants, car, il donne les meilleurs résultats tout en étant très rapide *(Jason Brownlee, 2017)*.

Figure 13 : comparaison de Adam contre d'autres optimiseurs



(machinelearningmastery.com, 2017)

Adam est une combinaison de deux autres algorithmes de descente de gradient stochastique permettant d'avoir les avantages de ceux-ci. Il s'agit de Adaptive Gradient Algorithm (AdaGrad) qui permet d'enlever les besoins de modifier manuellement le taux d'apprentissage en maintenant et adaptant un seul taux d'apprentissage par dimension. Ensuite, Root Mean Square Propagation (RMSProp) qui est un algorithme très robuste qui a comme caractère principal le fait de normaliser les gradients pour permettre de modifier le « pas » dépendant de la fonction *(Jason Brownlee, 2017)*.

4. Algorithme d'apprentissage par renforcement profond

4.1 Deep-Q Learning

Le Depp Q Learning (DQN) est une suite au table Q Learning vu précédemment. Il s'agit d'un algorithme d'apprentissage « off-policy » (qui utilise deux politiques, une pour apprendre et une pour explorer) profond qui reprend la base de Q Learning. Comme expliquer dans le chapitre Apprentissage par renforcement profond (chapitre 3) , il utilise un réseau de neurones plutôt qu'un une table de Q-value. Dans ce réseau de neurones convolutif, nous aurons en entrée les états possibles (les points d'observation de l'environnement) et en couche de sortie, le nombre d'actions possibles avec pour chaque neurone de sortie la Q-value de chacune de ses actions. Comme pour la table Q Learning le but est de trouver la meilleur Q-function *(Maxim Lapan, 2020)*.

4.1.1 Expérience Replay

DQN introduit un concept qui est l'expérience replay. Il s'agit d'une approche d'apprentissage en apprentissage par renforcement qui permet d'utiliser des expériences déjà passées pour l'apprentissage. L'expérience passée est une liste de tuple constitué de quatre éléments.

$$e_t = (s_t, a_t, e_{t+1}, s_{t+1})$$

Dans ce tuple nous trouvons, s_t qui est l'état où l'agent se trouve initialement, a_t qui est l'action que l'agent a prise, r_{t+1} la récompense que cette action retourne et enfin, s_{t+1} qui est l'état où l'agent se trouve après l'action. L'apprentissage se fait en commençant par le début et en avançant dans l'épisode nous allons donc apprendre en séquentiel, le problème est que les états séquentiels sont corrélés ce qui fait que nous ne trouvons pas de solution optimale. Par exemple, si on apprend à une voiture à conduire et que le début consiste de ligne droite. Le réseau va apprendre à conduire en ligne droite or, dès que l'on va passer sur un virage notre apprentissage qui est uniquement basé sur des lignes droites ne sera donc pas du tout performant. Pour combler à ce problème nous allons créer une liste finie aléatoire d'expérience et entrainer notre réseau de neurones, ce qui nous permet d'avoir une indépendance aux actions en séquentiel. Vu que DQN est une off-Policy il n'y a aucun problème à apprendre de manière aléatoire *(Maxim Lapan, 2020)*.

4.1.2 Calculer la Q-value optimal

Cette partie est similaire au Table Q Learning. Pour rappel, nous devons calculer la Q-value optimal pour ensuite maximiser les gains. Pour se faire nous devons calculer la perte avec entre la Q-value optimal et la Q-value actuelle.

$$Q_*(s,a) - Q(s,a) = perte$$

$$E\left[R_{t+1} + \gamma \overset{max}{a'} Q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] = perte$$

Pour calculer la perte, il nous faut d'abord calculer $\overset{max}{a'} Q_*(s',a')$ La Q-value maximum de l'état suivant. Pour se faire, il faut faire passer l'étape d'après s' en entrée dans le réseau de neurones. Ensuite, il faut sélectionner l'action qui retourne la Q-value maximum. Grâce à cela, nous pouvons calculer la perte et modifier les poids de notre réseau. Pour éviter des risques d'instabilité causer par le fait que nous utilisons le même réseau de neurones pour calculer La Q-value maximum avec seulement une itération d'écart, nous allons créer un Target Network. Le but du Target Network est de créer une copie de notre réseau actuelle afin de calculer La Q-value maximum de l'état suivant. Une fois le calcul fait nous allons par la suite copier à nouveau le Main network dans le Target Network *(Maxim Lapan, 2020)*.

4.2 Advantage actor critic

Advantage actor critic (A2C) est un algorithme « on-policy » qui est l'inverse d'une « off-policy ». On-policy veut dire qu'il faut avoir une seule politique pour apprendre et explorer. Comme son nom l'indique cet algorithme contient deux réseaux de neurones un acteur et une critique. Pour se faire, ils vont utiliser la fonction d'avantage (Advantage Function) qui permet de calculer la TDError. Cette fonction nous permet de dire si la récompense est meilleure ou pire que celle qui est attendue. Si une action est meilleure alors nous allons encourager l'agent à prendre cette action en augmentant la probabilité de celle-ci, à l'inverse nous allons réduire la probabilité de celle-ci pour ne pas reprendre cette action. Voici la formule de la fonction d'avantage (*Chris Yoon, 2019)* :

$$A(s,a) = TDTarget - V(S)$$

Dans cette fonction Il y a la fonction $V(S)$ qui permet de calculer la valeur d'un état vu précédemment et la TD Target. La TD Target représente la récompense additionnée par la valeur du prochain état multiplié par le taux d'évaluation (Discount factor). À noter qu'à la dernière étape nous enlèverons la valeur du prochain état, car celle-ci n'existe pas :

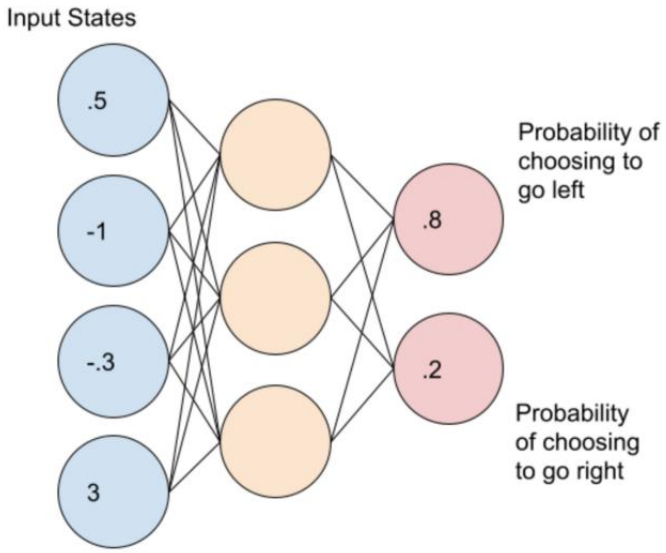
$$TDTarget = R + \gamma * V(S')$$

$$TDTarget = R$$

4.2.1 L’Acteur

Pour mieux comprendre le but de l’acteur et du critique, nous allons donner un exemple. Imaginons que nous voulons apprendre à conduire, nous aurons donc deux personnes dans la voiture l’élève conducteur qui apprend à conduire et le moniteur qui va aider l’élève dans son apprentissage. L'Acteur représente l’élève conducteur et son but et de choisir une décision lorsque celui-ci conduit. Nous pouvons voir si dessous le schéma d’un réseau de neurones de l'acteur. Il aura en entrée les points de contrôle de l’environnement (input) et en sortie une action avec une probabilité de la prendre (la somme des actions doit être égale à 1) *(Chris Yoon, 2019)*.

Figure 14 : schéma de l'acteur

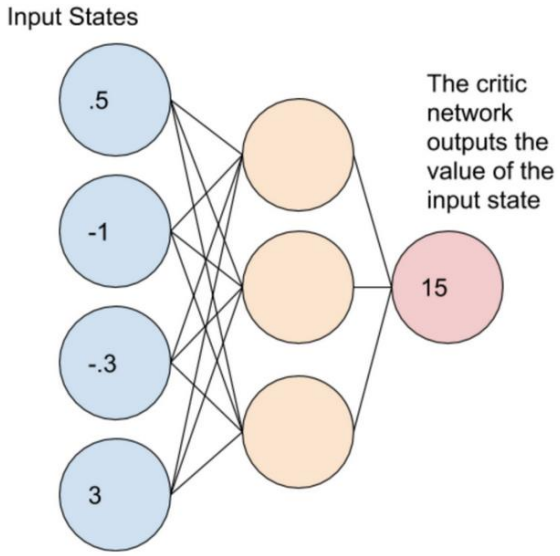


(towardsdatascience.com,2021)

4.2.2 Le critique

Revenons à notre exemple, le critique sera donc le moniteur qui va lui aussi recevoir les mêmes informations en même temps que l'élève conduit. Il ne va pas déterminer qu'elle est la meilleure action à prendre dans la situation. Il va plutôt chercher à savoir quel est le meilleur résultat autrement dit, le meilleur endroit où la voiture doit être pour continuer. Ce qu'il fera ensuite est de « critiquer » l'action que l'élève a prise afin que celui-ci modifie sa manière de penser pour un résultat optimal (*Chris Yoon, 2019)*.

Figure 15 : schéma du critique



(towardsdatascience.com,2021)

Comme expliqué dans l'exemple, notre réseau aura les mêmes entrées que l'acteur, mais il n'aura qu'une sortie étant la Q-value de l'état.

4.2.3 Mise à jour des neurones.

Nous avons deux réseaux et lorsque l'acteur prend une action le réseau critique l'évalue. C'est lors de la décision que la Advantage Function est utilisée. Elle va permettre de donner une valeur à l'action prise par notre acteur en comparant valeur de l'état actuel avec la valeur « optimum » que le critique a calculée. Dans le cas où, le résultat (TDError) est positif alors nous allons modifier les poids de notre neurone pour que l'action choisie à une meilleure probabilité et à l'inverse si la valeur est négative.

Il faut aussi mettre à jour le réseau de neurones du critique. On notera que plus la valeur de la TDError est proche de 0 plus notre critique est performant ce que nous allons faire et de mettre les poids de notre réseau pour que celle-ci se rapproche de 0 (*Chris Yoon, 2019)*.

5. Implémentation

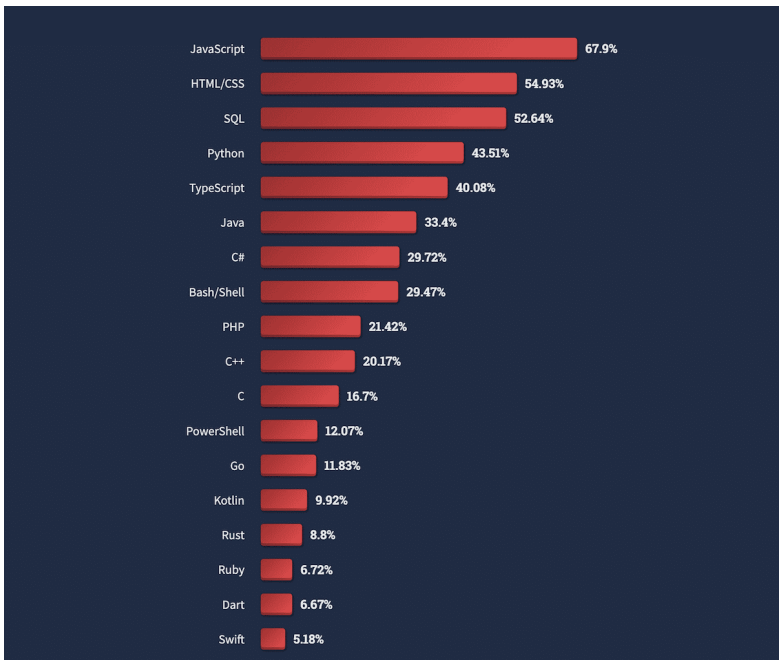
Dans ce chapitre nous allons voir les technologies utilisées et comment j'ai implémenté DQN et A2C en code pour que dans le chapitre suivant nous puissions analyser les résultats.

5.1 Technologie utilisée

5.1.1 Python

Le langage que j'ai utilisé pour l'implémentation est Python. Il s'agit d'un langage de programmation open source qui fait partie des tops 5 des langages les plus utilisés et le premier pour le domaine de la science des données et du Machine Learning. Ce langage est beaucoup utilisé pour plusieurs raisons : La première, il contient un grand nombre de bibliothèques dont certaines très utiles pour le Machine Learning comme Numpy par exemple, elle est plus performante que Python pour ce qui est des tableaux et des mathématiques. Deuxièmement, il est indépendant du système d'opération et peut donc tourner sur n'importe quelle machine. Enfin, c'est un des langages les plus utilisés et permet d'avoir beaucoup de ressource et de réponse aux questions en ligne.

Figure 16 : analyse des languages de programation par stackoverflow



(Stackoverflow.com 2022)

5.1.1 Pytorch

Pytorch est un Framework basé sur la bibliothèque Torch qui est open source. Il permet de développer et faciliter l'implémentation de programme telle que des agents d'apprentissage par renforcement. Pytorch fait partie d'une des principales bibliothèques Python pour l'apprentissage par renforcement. Étant un Framework, on peut l'utiliser grâce à des objets Python plutôt que des appels à une bibliothèque et permet d'avoir un code plus lisible et plus compréhensible. De plus la documentation de Pytorch est complète est très facilement accessible.

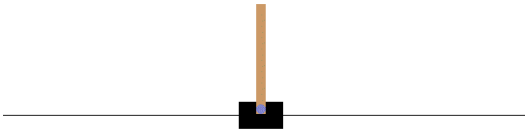
5.1.2 Gym

Gym est une bibliothèque d'Open AI en Python qui permet de développer et comparer facilement des algorithmes dans un environnement. Grâce à Gym, nous pouvons facilement créer un environnement avec un système API. Gym est devenu un standard pour ce qui des générations d'environnement dans le domaine de l'apprentissage par renforcement. Il y a une vingtaine d'environnements créer pour différent cas, en plus de cela une cinquantaine d'environnements venant de l'Atari 2600 permettant d'entrainer des agents sur des jeux de la console. Enfin, des logiciels tiers avec d'autres environnements. De plus, l'interface de Gym permet de facilement implémenter des environnements dans le code.

5.1.2.1 Cart Pole

L'environnement que j'ai choisi dans ce cas est Cart Pole. Je l'ai choisi, car il s'agit de l'environnement souvent utilisé pour commencer. Cet environnement contient un petit jeu d'action discret et un petit jeu d'observation. Il fait partie des plus faciles à mettre en place et souvent le premier pour apprendre l'apprentissage par renoncement. Le but de cet environnement est de faire tenir là une barre (Pole) en équilibre en faisant bouger sa base (Cart) à droite ou à gauche. La simulation peut se terminer sous trois conditions. La première dans le cas où la barre à un angle de ± 12° par rapport au centre. Si la position du Carte sort de du champ de jeu (± 2.4) ou la longueur de l'épisode est de 500 étapes. Pour ce qui est de la récompense, chaque étape l'environnement donne à l'agent +1 vu que le but est de tenir le plus longtemps sans que celui-ci se termine.

Figure 17 : Exemple de la tâche à accomplir dans l’environnement Cart pole



(gymlibrary.dev, 2022)

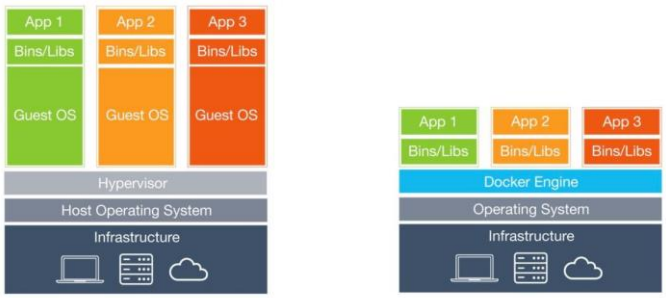
5.1.3 Weight and Biases

Weight and Biases est un outil pour le Machine Learning permettant de faciliter les versions et la visualisation des résultats des différents modèles. Weight and Biases fonctionne grâce à un serveur qui peut être physique ou virtuel. Nous allons envoyer des données au serveur et celui-ci va traiter les résultats. Il permet aussi de les afficher dans l’application web en forme de graphique. Weight and Biases est un outil très complet qui permet de stocker plusieurs projets, afficher les résultats d’une expérience, de pouvoir combiner plusieurs expériences pour avoir une moyenne des résultats.

5.1.4 Docker

Docker est système permettant la virtualisation d'application sans avoir besoin de faire fonctionner une machine virtuelle complète. Docker permet de faire des conteneurs avec à l'intérieur des applications pouvant tourner indépendamment de l'OS de la machine hôte et sans avoir à faire tourner un autre OS par-dessus. Dans le cadre de ce travail de Docker a été utilisé pour mettre en place un serveur local de Weight and Biases, car celui-ci (en version locale) est fourni pour une installation avec docker. Docker permettra à Weight and Biases de faire tourner l'application en « simulant » un serveur.

Figure 18 : différence entre une machine virtuelle et docker



(Infoworld.com, 2021)

5.2 Implémentation de DQN et A2C

Si vous cherchez à voir mon code de l'implémentation de DQN et A2C. Il suffit de se rendre à l'annexe 4 qui contient un lien GitHub avec les deux fichiers python et en plus une explication de mon implémentation du code DQN et en Annexe 5 le même lien GitHub et l'explication de mon code A2C.

6. Analyse des résultats

Tout d'abord, je voudrais dire que les résultats obtenus par mon implémentation ne sont en aucun cas des résultats qui peuvent être pris comme une généralité. Il s'agit simplement des résultats que j'ai pu obtenir avec mon implémentation et l'environnement peut aussi drastiquement changer les résultats obtenus.

Voici une petite description des annexes :

- En annexe 1 : Vous avez la liste des hyperparamètre utilisés par DQN et A2C.
- En annexe 2 : il s'agit de la moyenne des récompenses des agents par épisode avec en plus clair les valeurs réelles de chaque agent avec ça couleur.
- En annexe 3 : La moyenne des récompenses des agents groupés par épisode et leurs variances.

6.1 Apprentissage

Nous pouvons remarquer un point important avec l'agent DQN. C'est que celui-ci n'est pas optimal. On peut clairement voir que les moyennes des récompenses des agents groupés (annexe 2 et 3) dans un premier temps, à une grande variance, c'est-à-dire que deux agents ont des comportements très différents et que la récompense n'augmente pas constamment alors qu'un agent optimal devraient avoir une valeur constante. À l'inverse on peut voir que A2C a une variance beaucoup moins forte et une courbe beaucoup plus stable qui augmente progressivement ce qui montre qu'il apprend mieux que DQN.

6.2 Paramètre

A2C est un algorithme avec très peu de paramètres (annexe 1) qui permet de diminuer la complexité or, c'est à double tranchant, car avec plus de paramètres on rajoute de la complexité, mais cela permet aussi d'avoir des agents beaucoup plus flexibles et optimaux.

6.3 Rapidité

Un point intéressant avec DQN est que celui-ci a initialement une courbe d'apprentissage plus rapide que A2C, il arrive en moyenne à un épisode retournant une récompense totale de 200 proches du 500ème épisode alors que A2C quant à lui arrive à cette valeur plus vers le 1000ème épisode. Cependant, le temps d'exécution de DQN est plus lent que A2C et c'est à partir de ce point que DQN devient très variant.

7. Conclusion

Pour ce qui est des résultats je conclurais en disant que les deux algorithmes sont similaires et que chacun a des forces et des faiblesses, mais qu'aucun des deux n'est parfait pour n'importe quelle tâche. Je remarque aussi que mon implémentation de l'agent A2C est clairement supérieure à DQN, mais, comme expliquer précédemment il s'agit de mon implémentation et que celle-ci n'est pas une vérité.

On peut souvent entendre dire que le Machine Learning est le monde de demain. Il est vrai que le Machine Learning va être au centre de notre futur, mais je trouve que le Machine Learning est déjà présent aujourd'hui. On retrouve de plus en plus d'intelligences artificielles dans tous les domaines accomplissant des tâches plus complexes, la puissance de calcul des ordinateurs ne cesse d'augmenter ce qui facilite leurs implémentations. C'est pourquoi je trouve ce sujet très intéressant.

J'ai auparavant déjà travaillé dans le cadre d'un projet avec un réseau de neurones dans le but de lui apprendre à jouer au jeu Pong avec la Library Tensorflow. Je n'ai vu que la base des réseaux de neurones lors de ce projet. Lorsque j'ai commencé ce travail, il a été intéressant d'approfondir le sujet et de pouvoir comparer le fonctionnement de Pytorch à Tensorflow.

Je suis très content du résultat final et de ce que j'ai pu apprendre tout au long de ce travail. Il n'a pas été toujours facile de comprendre le fonctionnement de l'apprentissage par renforcement et j'ai eu l'occasion de rencontrer de vraies difficultés au long de mes recherches que j'ai fini par surmonter et m'a permis de rendre la réussite plus satisfaisante.

Grâce à ce travail, j'ai pu mettre un pied dans le domaine de la science des données et je trouve le domaine passionnant. Ne sachant pas actuellement dans quel domaine je souhaiterais travailler par la suite, ce travail m'a permis d'approfondir un domaine qui m'intéresse et qui a beaucoup d'avenir.

Annexe 1 : hyperparamètre

Tableau 1 : tableau des hyperparamètre de DQN

Nombre d'épisodes	5002
Learning rate	0.0001
Taille de la liste de l'Expérience Replay	10000
Taille du nombre d'échantillons retourner	64
Gamma	0.95
epsilon	1.0
Taux de décroissance d'epsilon	0.999
Valeur minimum d'epsilon	0.001
Dimension de la couche cachée	256
Fréquence de mise à jour du Target Network	5

(Raphaël Lopes, 2022)

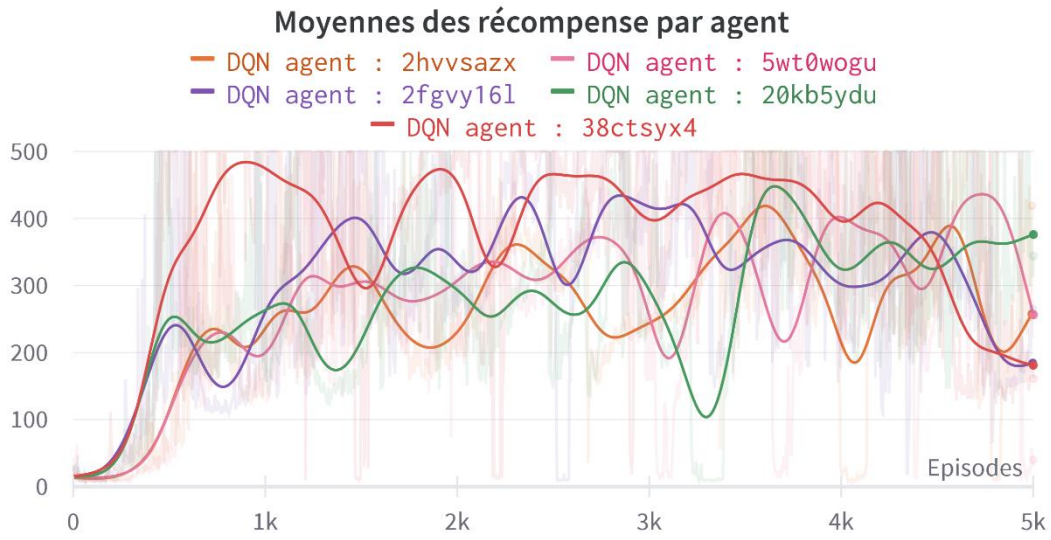
Tableau 2 : tableau des hyperparamètre de A2C

Nombre d'épisodes	5002
Learning rate	0.001
Dimension de la couche cachée de l'acteur	64
Dimension de la couche cachée du critique	64

(Raphaël Lopes, 2022)

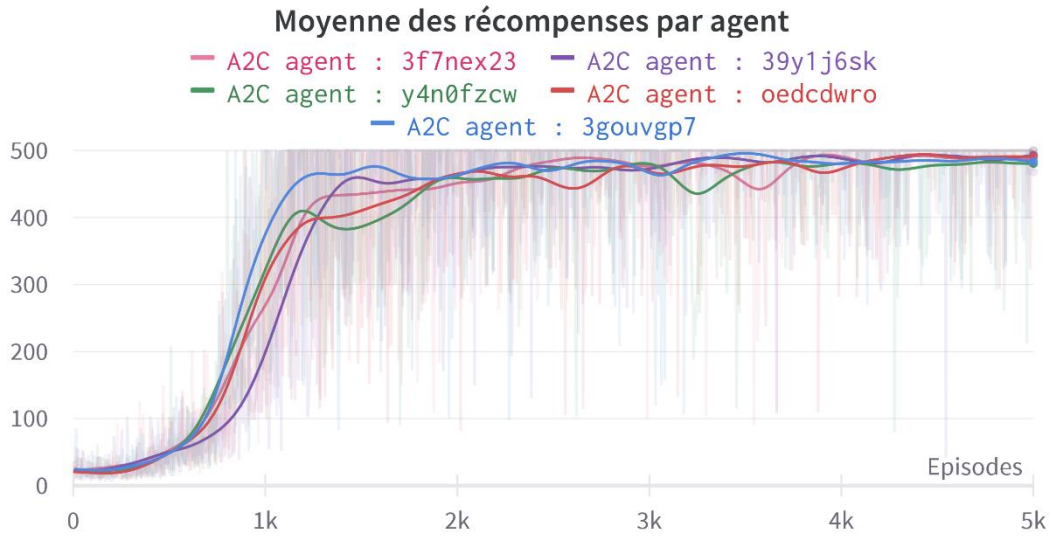
Annexe 2 : Moyenne des récompenses des agents par épisode DQN et A2C

Figure 19 : la valeur de la récompense de chaque agent DQN



(Raphaël Lopes, 2022)

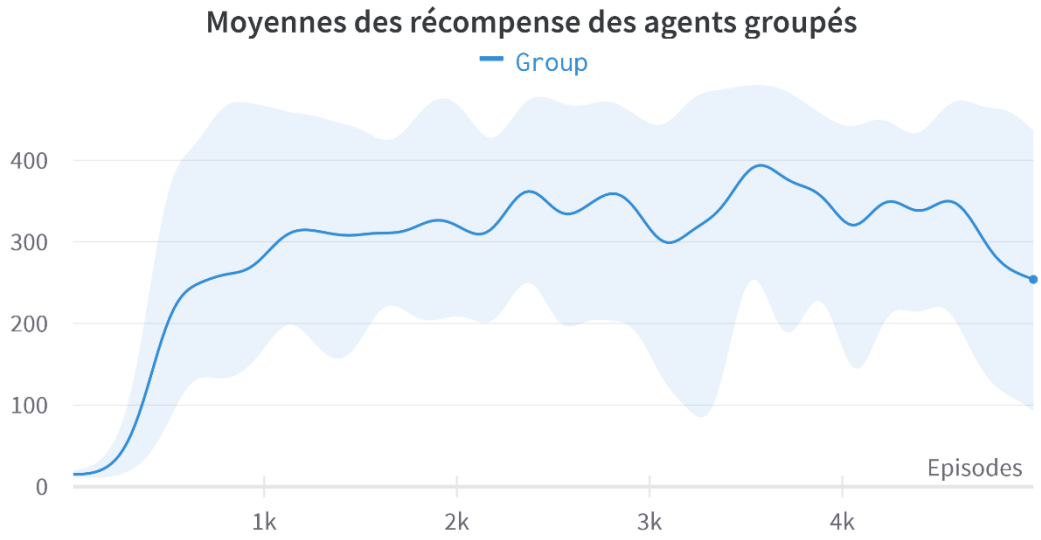
Figure 20 : la valeur de la récompense de chaque agent A2C



(Raphaël Lopes, 2022)

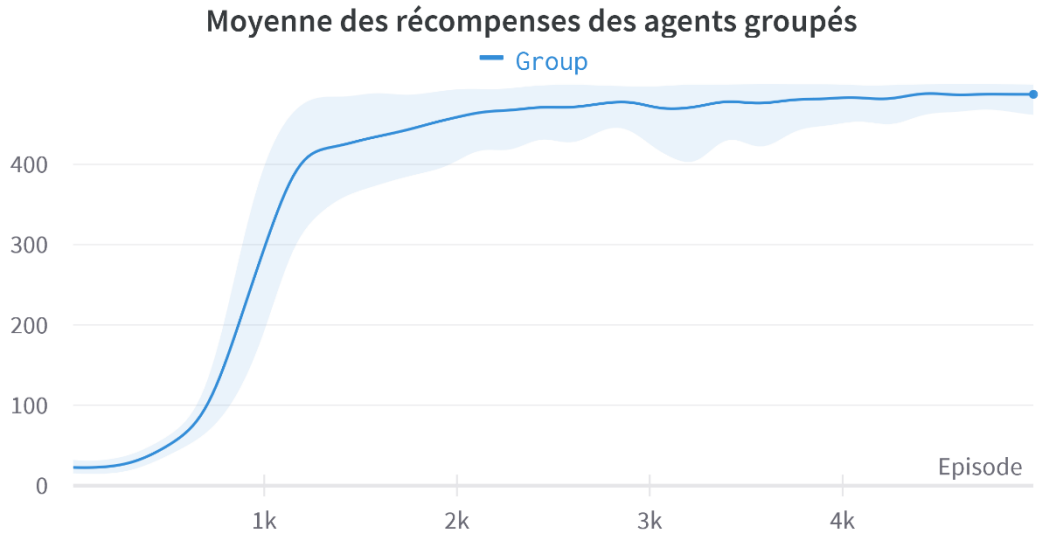
Annexe 3 : Moyenne des récompenses des agents groupés par épisode DQN et A2C

Figure 21 : la valeur de la récompense de chaque agent groupé DQN



(Raphaël Lopes, 2022)

Figure 22 : la valeur de la récompense de chaque agent groupé A2C



(Raphaël Lopes, 2022)

Annexe 4 : Code DQN

Le code est disponible sur ce repository Github[1]. Ci-dessous vous pouvez voir ci-dessous des explications et commentaires de chacune des parties du code.

Figure 23 : Importation des bibliothèque, initialisation des constantes et GYM

```
1   import torch
2   import torch.nn as nn
3   import torch.nn.functional as F
4   import torch.optim as optim
5   import math
6   import gym
7   import random
8   import numpy as np
9   import matplotlib.pyplot as plt
10  from tqdm import tqdm
11  import wandb
12  import time
13  import copy
14
15  env = gym.make('CartPole-v1')
16  env = gym.wrappers.RecordVideo(env, 'video_DQN_CartePole', episode_trigger = lambda x: x %  500 == 0)
17  observation_space = env.observation_space.shape[0]
18  action_space = env.action_space.n
19
20  EPISODES = 5002
21  LEARNING_RATE = 0.0001
22  MEM_SIZE = 10000
23  BATCH_SIZE = 64
24  GAMMA = 0.95
25  EXPLORATION_MAX = 1.0
26  EXPLORATION_DECAY = 0.999
27  EXPLORATION_MIN = 0.001
28  FC1_DIMS = 256
29  DEVICE = torch.device("cuda")
30  ARRAY_OF_SEED = [1,2,3,4,5]
```

(Raphaël Lopes, 2022)

La ligne 15 à 18 viennent de la Library GYM permettant d'expliquer à GYM l'environnement que nous souhaitons utiliser et récupérer les points de mesure et action possible dans l'environnement. Nous avons aussi un lambda qui permet de dire à Gym quand il doit enregistrer une vidéo de l'agent lorsque celui-ci est vrai. Ensuite, nous pouvons observer l'initiation des hyperparamètres. Un des paramètres est le « DEVICE ». Celui-ci permet de dire à Torch si nous souhaitons utiliser notre processeur pour les calculs ou d'utiliser notre processeur graphique avec la technologie CUDA de Nvidia qui permet de faire des traitements avec le processeur graphique qui est

[1] https://github.com/zabraf/TB_DQN_A2C_CartePole

beaucoup plus rapide. À la ligne 30, nous pouvons voir ARRAY_OF_SEED. Il s'agit d'une suite de seed que nous allons utiliser. Une seed est utilisée dans un pseudogénérateur aléatoire permettant de générer la même suite aléatoire ce qui nous permet d'avoir des résultats reproductibles.

Figure 24: Classe du réseau de neurones

```
class Network(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.input_shape = env.observation_space.shape
        self.action_space = action_space

        self.fc1 = nn.Linear(*self.input_shape, FC1_DIMS)
        self.fc2 = nn.Linear(FC1_DIMS, self.action_space)

        self.optimizer = optim.Adam(self.parameters(), lr=LEARNING_RATE)
        self.loss = nn.MSELoss()
        self.to(DEVICE)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

(Raphaël Lopes, 2022)

Cette classe permet à la création du réseau de neurones avec ces couches. Nous avons aussi initialisé les réseaux avec l'optimiseur, nous allons comme pour utiliser comme expliquer auparavant l'optimiseur Adam. La méthode forward est une méthode qui vient du module torch.nn qui permet de définir la fonction d'activation utilisée

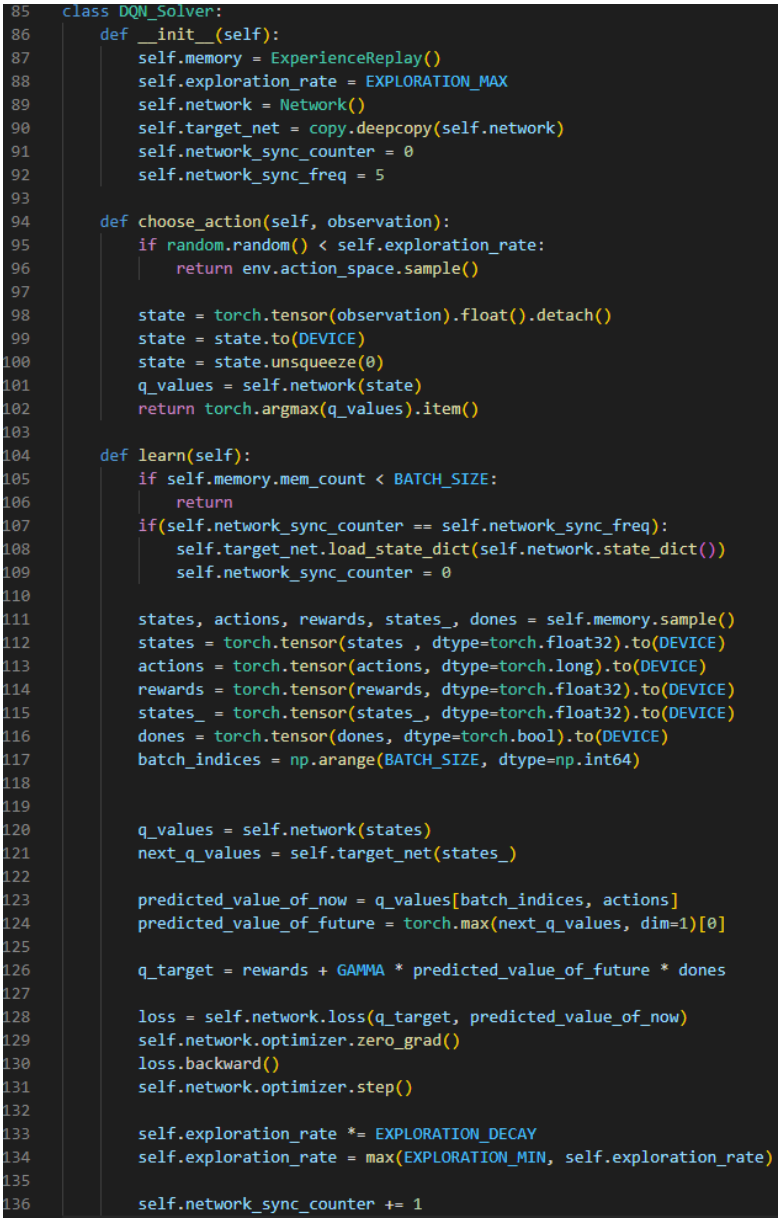
Figure 25 : Replay buffer

```python
class ExperienceReplay:
    def __init__(self):
        self.mem_count = 0

        self.states = np.zeros((MEM_SIZE, *env.observation_space.shape),dtype=np.float32)
        self.actions = np.zeros(MEM_SIZE, dtype=np.int64)
        self.rewards = np.zeros(MEM_SIZE, dtype=np.float32)
        self.states_ = np.zeros((MEM_SIZE, *env.observation_space.shape),dtype=np.float32)
        self.dones = np.zeros(MEM_SIZE, dtype=np.bool)

    def add(self, state, action, reward, state_, done):
        mem_index = self.mem_count % MEM_SIZE

        self.states[mem_index]  = state
        self.actions[mem_index] = action
        self.rewards[mem_index] = reward
        self.states_[mem_index] = state_
        self.dones[mem_index] =  1 - done

        self.mem_count += 1

    def sample(self):
        MEM_MAX = min(self.mem_count, MEM_SIZE)
        batch_indices = np.random.choice(MEM_MAX, BATCH_SIZE)

        states  = self.states[batch_indices]
        actions = self.actions[batch_indices]
        rewards = self.rewards[batch_indices]
        states_ = self.states_[batch_indices]
        dones   = self.dones[batch_indices]

        return states, actions, rewards, states_, dones
```

(Raphaël Lopes, 2022)

Cette classe est Experience Replay, nous avons un constructeur permettant de créer une liste de tuple vide d'une certaine taille. À l'intérieur nous pouvons voir la méthode add qui permet ajouter un tuple d'une expérience. À la ligne 63 nous pouvons voir le calcule permettant de remplacer le tuple le moins récent et le remplacer par le plus récent dans le cas où nous souhaitons ajouter lorsque la mémoire est pleine. Ensuite, nous avons l'argument sample qui permet de récupérer un nombre (BATCH_SIZE) de tuples aléatoire dans notre Experience Replay.

Figure 26 : Classe DQN



```python
class DQN_Solver:
    def __init__(self):
        self.memory = ExperienceReplay()
        self.exploration_rate = EXPLORATION_MAX
        self.network = Network()
        self.target_net = copy.deepcopy(self.network)
        self.network_sync_counter = 0
        self.network_sync_freq = 5

    def choose_action(self, observation):
        if random.random() < self.exploration_rate:
            return env.action_space.sample()

        state = torch.tensor(observation).float().detach()
        state = state.to(DEVICE)
        state = state.unsqueeze(0)
        q_values = self.network(state)
        return torch.argmax(q_values).item()

    def learn(self):
        if self.memory.mem_count < BATCH_SIZE:
            return
        if(self.network_sync_counter == self.network_sync_freq):
            self.target_net.load_state_dict(self.network.state_dict())
            self.network_sync_counter = 0

        states, actions, rewards, states_, dones = self.memory.sample()
        states = torch.tensor(states , dtype=torch.float32).to(DEVICE)
        actions = torch.tensor(actions, dtype=torch.long).to(DEVICE)
        rewards = torch.tensor(rewards, dtype=torch.float32).to(DEVICE)
        states_ = torch.tensor(states_, dtype=torch.float32).to(DEVICE)
        dones = torch.tensor(dones, dtype=torch.bool).to(DEVICE)
        batch_indices = np.arange(BATCH_SIZE, dtype=np.int64)


        q_values = self.network(states)
        next_q_values = self.target_net(states_)

        predicted_value_of_now = q_values[batch_indices, actions]
        predicted_value_of_future = torch.max(next_q_values, dim=1)[0]

        q_target = rewards + GAMMA * predicted_value_of_future * dones

        loss = self.network.loss(q_target, predicted_value_of_now)
        self.network.optimizer.zero_grad()
        loss.backward()
        self.network.optimizer.step()

        self.exploration_rate *= EXPLORATION_DECAY
        self.exploration_rate = max(EXPLORATION_MIN, self.exploration_rate)

        self.network_sync_counter += 1
```

(Raphaël Lopes, 2022)

Ce code est le plus important, il s'agit de l'algorithme DQN, qui va nous permettre de faire « apprendre » à notre agent d'effectuer la tâche Cart Pole. Dans initiation nous pouvons voir la création du Main network et du Target network. La méthode choose_action va permettre de prendre une action qui peut être d'exploration ou d'exploitation en fonction de la valeur générer contre la valeur l'epsilon. Nous avons ensuite la phase d'apprentissage (méthode learn) ou on va récupérer un tuple d'expérience. On va ensuite calculer la perte de celle-ci en appliquant la formule de coût, nous utiliserons la rétropropagation permettant de mettre à jour les poids du réseau de neurones pour favoriser une politique optimale.

Figure 27 : Boucle d'entrainement

```
138 for seed in ARRAY_OF_SEED:
139     torch.manual_seed(seed)
140     env.seed(seed)
141     np.random.seed(seed)
142     run = wandb.init(project="CartPole-v1-DQN",reinit=True)
143     wandb.run.name = "DQN agent : " + wandb.run.id
144 
145     wandb.config = {
146         "episodes": EPISODES,
147         "learning_rate": LEARNING_RATE,
148         "mem_size": MEM_SIZE,
149         "batch_size": BATCH_SIZE,
150         "gamma": GAMMA,
151         "exploration_max": EXPLORATION_MAX,
152         "exploration_decay": EXPLORATION_DECAY,
153         "exploration_min": EXPLORATION_MIN,
154         "fc1_dims": FC1_DIMS,
155         "device": "cuda",
156         "network sync frequency": 5,
157         "seed": [str(x) for x in ARRAY_OF_SEED]
158     }
159     agent = DQN_Solver()
160 
161 
162     total_start_time = time.time()
163     for i in tqdm(range(1, EPISODES + 1)):
164         state = env.reset()
165         state = np.reshape(state, [1, observation_space])
166         score = 0
167         done = False
168         while not done:
169             action = agent.choose_action(state)
170             state_, reward, done, info = env.step(action)
171             state_ = np.reshape(state_, [1, observation_space])
172             agent.memory.add(state, action, reward, state_, done)
173             agent.learn()
174             state = state_
175             score += reward
176 
177         wandb.log({"Reward": score})
178     total_stop_time = time.time()
179     print("the episode took : ", (total_stop_time - total_start_time))
180     run.finish()
```

(Raphaël Lopes, 2022)

La ligne 138 à 143 permet de faire 5 agents à la suite avec chacun une seed différent et permet aussi à Weight and Biases de générer des agents qui vont être stockés dans l'application. Nous avons ensuite l'initialisation / instatiation des variables et de la sauvegarde de la configuration dans Weight and Biases. Enfin, le code contient la boucle d'entrainement qui va utiliser les deux dernières méthodes vues précédemment learn et choose_action pour trouver la politique optimale.

Annexe 5 : Code A2C

Le code de A2C disponible sur ce repository Github[2]. Ci-dessous vous pouvez voir des explications et commentaires de chacune des parties du code.

Figure 28 : Importation des bibliothèque, initialisation des constantes et GYM

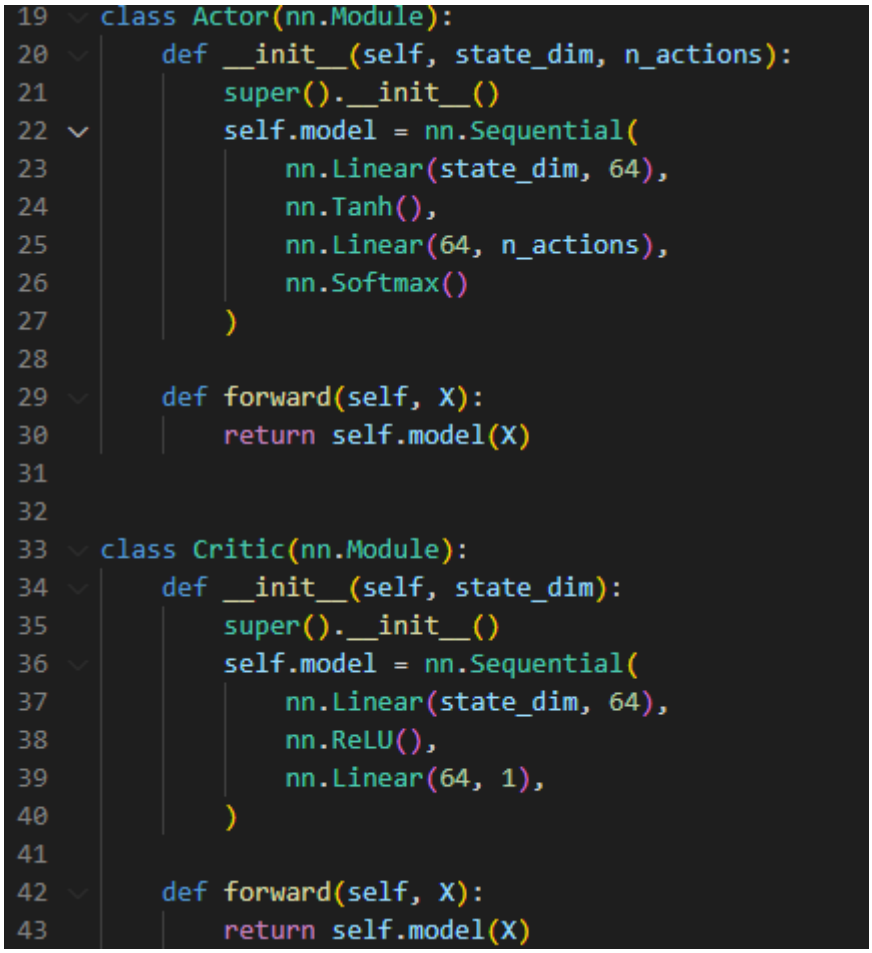
```python
import numpy as np
import torch
import gym
from torch import nn
import time
import wandb
from tqdm import tqdm


env = gym.make("CartPole-v1")
env = gym.wrappers.RecordVideo(env, 'video_A2C_CartePole', episode_trigger = lambda x: x % 500 == 0)
state_dim = env.observation_space.shape[0]
n_actions = env.action_space.n
GAMMA = 0.99
EPISODES = 5002
ARRAY_OF_SEED = [1,2,3,4,5]
```

(Raphaël Lopes, 2022)

Similaire à DQN, il y a dans la figure les bibliothèques importer et les constantes et l'initialisation de GYM.

[2] https://github.com/zabraf/TB_DQN_A2C_CartePole

Figure 29 : Code de la classe acteur et critique

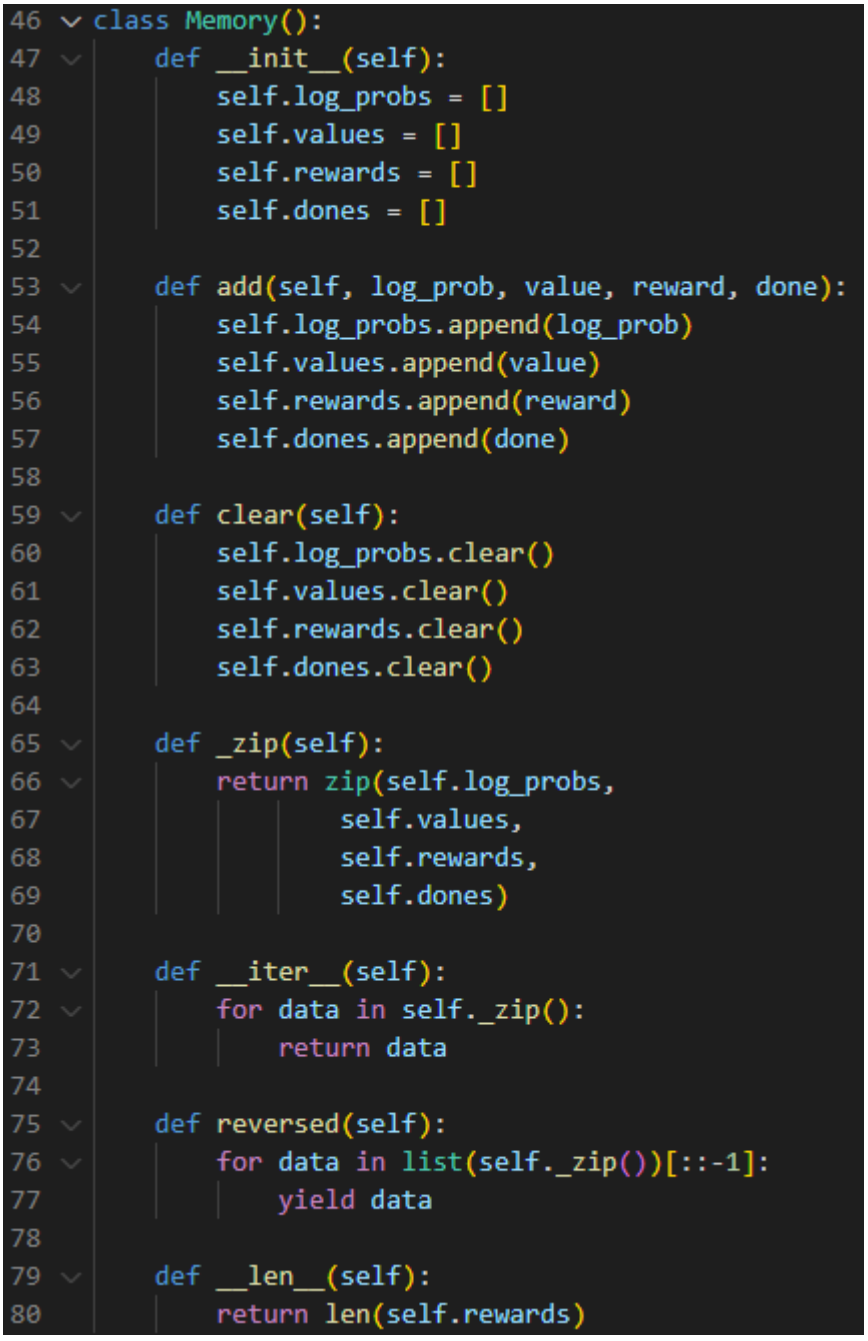


```
19  class Actor(nn.Module):
20      def __init__(self, state_dim, n_actions):
21          super().__init__()
22          self.model = nn.Sequential(
23              nn.Linear(state_dim, 64),
24              nn.Tanh(),
25              nn.Linear(64, n_actions),
26              nn.Softmax()
27          )
28
29      def forward(self, X):
30          return self.model(X)
31
32
33  class Critic(nn.Module):
34      def __init__(self, state_dim):
35          super().__init__()
36          self.model = nn.Sequential(
37              nn.Linear(state_dim, 64),
38              nn.ReLU(),
39              nn.Linear(64, 1),
40          )
41
42      def forward(self, X):
43          return self.model(X)
```

(Raphaël Lopes, 2022)

Nous pouvons lire à la ligne 25 à 36, la création du réseau de neurones de l'acteur contenant les entrées, les 64 neurones cachés et la couche de sortie avec un nombre de neurones égale au nombre d'actions. Nous pouvons voir aussi l'utilisation de Softmax qui permet de transformer les valeurs sortantes qui sont des nombres réels en valeur entre 0 et 1 dont la somme est égale à 1. Nous utilisons Softmax pour créer des pourcentages de probabilité de prendre une action. La création du réseau de neurones du critique est similaire à l'exception que celui-ci n'a qu'une sortie pour le calcul de la Q-value.

Figure 30 : Mémoire des résultats.



```python
class Memory():
    def __init__(self):
        self.log_probs = []
        self.values = []
        self.rewards = []
        self.dones = []

    def add(self, log_prob, value, reward, done):
        self.log_probs.append(log_prob)
        self.values.append(value)
        self.rewards.append(reward)
        self.dones.append(done)

    def clear(self):
        self.log_probs.clear()
        self.values.clear()
        self.rewards.clear()
        self.dones.clear()

    def _zip(self):
        return zip(self.log_probs,
                   self.values,
                   self.rewards,
                   self.dones)

    def __iter__(self):
        for data in self._zip():
            return data

    def reversed(self):
        for data in list(self._zip())[::-1]:
            yield data

    def __len__(self):
        return len(self.rewards)
```

(Raphaël Lopes, 2022)

Sur cette image nous pouvons lire la classe Memory. Il s'agit d'une mémoire des résultats faits par l'acteur dans un état qui permet d'avoir une meilleure performance en évitant de recalculer une opération depuis un état déjà calculé.

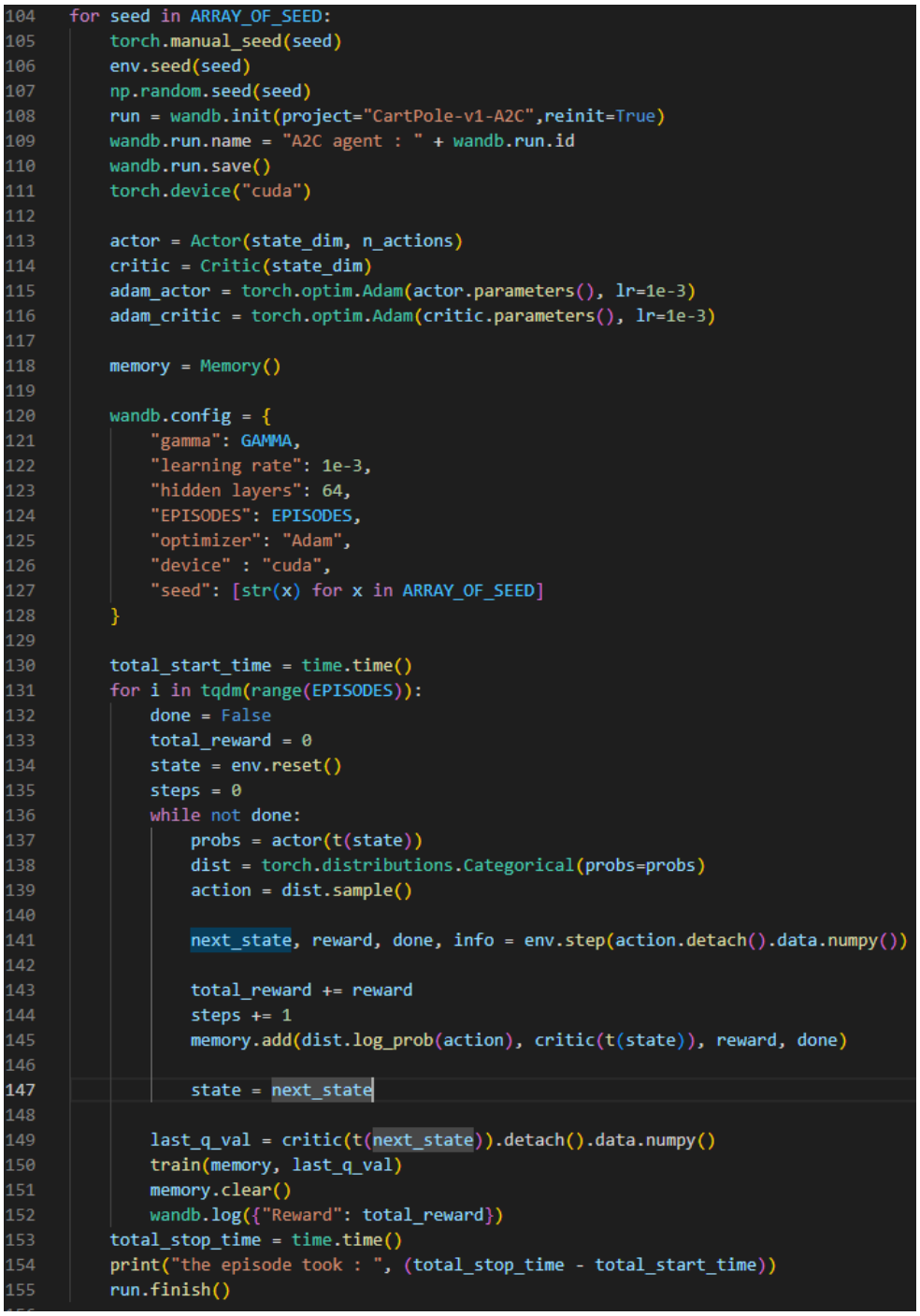
Figure 31 : méthode d'entrainement

```python
def train(memory, q_val):
    values = torch.stack(memory.values)
    q_vals = np.zeros((len(memory), 1))

    for i, (_, _, reward, done) in enumerate(memory.reversed()):
        q_val = reward + GAMMA *q_val*(1.0-done)
        q_vals[len(memory)-1 - i] = q_val

    advantage = torch.Tensor(q_vals) - values

    critic_loss = advantage.pow(2).mean()
    adam_critic.zero_grad()
    critic_loss.backward()
    adam_critic.step()

    actor_loss = (-torch.stack(memory.log_probs)*advantage.detach()).mean()
    adam_actor.zero_grad()
    actor_loss.backward()
    adam_actor.step()
```

(Raphaël Lopes, 2022)

La méthode d'entrainement (train) est ce que nous allons appeler lorsque nous voulons calculer la TDError en utilisant Advantage function permettant de mettre à jour les poids dans nos réseaux de neurones grâce à la rétropropagation et le calcul de perte.

Figure 32 : boucle d'entrainement



```python
for seed in ARRAY_OF_SEED:
    torch.manual_seed(seed)
    env.seed(seed)
    np.random.seed(seed)
    run = wandb.init(project="CartPole-v1-A2C",reinit=True)
    wandb.run.name = "A2C agent : " + wandb.run.id
    wandb.run.save()
    torch.device("cuda")

    actor = Actor(state_dim, n_actions)
    critic = Critic(state_dim)
    adam_actor = torch.optim.Adam(actor.parameters(), lr=1e-3)
    adam_critic = torch.optim.Adam(critic.parameters(), lr=1e-3)

    memory = Memory()

    wandb.config = {
        "gamma": GAMMA,
        "learning rate": 1e-3,
        "hidden layers": 64,
        "EPISODES": EPISODES,
        "optimizer": "Adam",
        "device" : "cuda",
        "seed": [str(x) for x in ARRAY_OF_SEED]
    }

    total_start_time = time.time()
    for i in tqdm(range(EPISODES)):
        done = False
        total_reward = 0
        state = env.reset()
        steps = 0
        while not done:
            probs = actor(t(state))
            dist = torch.distributions.Categorical(probs=probs)
            action = dist.sample()

            next_state, reward, done, info = env.step(action.detach().data.numpy())

            total_reward += reward
            steps += 1
            memory.add(dist.log_prob(action), critic(t(state)), reward, done)

            state = next_state

        last_q_val = critic(t(next_state)).detach().data.numpy()
        train(memory, last_q_val)
        memory.clear()
        wandb.log({"Reward": total_reward})
    total_stop_time = time.time()
    print("the episode took : ", (total_stop_time - total_start_time))
    run.finish()
```

(Raphaël Lopes, 2022)

Pour finir, nous avons la boucle d'entrainement qui va, comme pour DQN, il utilise une liste de seeds pour générer 5 agents reproductible. Ensuite, pour chacun de ses agents s'entrainer en répétant l'épisode jusqu'à sa fin en appelant la méthode train vu précédemment pour optimiser le moyen à résoudre la tâche. Comme on peut le constater, la mémoire se vide à chaque fin d'épisode.

Bibliographie

Apprentissage automatique, 2022. *Wikipédia*. [en ligne].
[Consulté le 20 septembre 2022]. Disponible à l'adresse:
https://fr.wikipedia.org/w/index.php?title=Apprentissage_automatique&oldid=197054257Page Version ID: 197054257

Apprentissage par renforcement, [sans date]. *Data Analytics Post*. [en ligne].
[Consulté le 5 octobre 2022]. Disponible à l'adresse:
https://dataanalyticspost.com/Lexique/apprentissage-par-renforcement/

Artificial intelligence, 2022. *Wikipedia*. [en ligne]. [Consulté le 30 septembre 2022].
Disponible à l'adresse:
https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=1112753222Page Version ID: 1112753222

AYYADEVARA, V. Kishore et REDDY, Yeshwanth, 2020. *Modern Computer Vision with PyTorch: Explore deep learning concepts and implement over 50 real-world image applications*. Birmingham Mumbai: Packt Publishing. ISBN 978-1-83921-347-2.

BABU, Alan Davis, 2019. Artificial Intelligence vs Machine Learning vs Deep Learning (AI vs ML vs DL). *Medium*. [en ligne]. 6 novembre 2019.
[Consulté le 19 septembre 2022]. Disponible à l'adresse:
https://medium.com/@alanb_73111/artificial-intelligence-vs-machine-learning-vs-deep-learning-ai-vs-ml-vs-dl-e6afb7177436

BROWNLEE, Jason, 2017. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. *Machine Learning Mastery*. [en ligne]. 2 juillet 2017.
[Consulté le 14 octobre 2022]. Disponible à l'adresse:
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

BROWNLEE, Jason, 2020. Neural Networks are Function Approximation Algorithms. *Machine Learning Mastery*. [en ligne]. 17 mars 2020. [Consulté le 14 octobre 2022].
Disponible à l'adresse: https://machinelearningmastery.com/neural-networks-are-function-approximators/

CAREY, Scott, 2021. What is Docker? The spark for the container revolution. *InfoWorld*. [en ligne]. 2 août 2021. [Consulté le 9 octobre 2022]. Disponible à l'adresse:
https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html

Cart Pole - Gym Documentation, [sans date]. [en ligne]. [Consulté le 16 septembre 2022]. Disponible à l'adresse: https://www.gymlibrary.dev/environments/classic_control/cart_pole/

Descente de gradient pour les réseaux de neurones, [sans date]. [en ligne]. [Consulté le 13 octobre 2022]. Disponible à l'adresse: https://lucidar.me/fr/neural-networks/single-layer-gradient-descent/

Docker (logiciel), 2022. *Wikipédia*. [en ligne]. [Consulté le 9 octobre 2022]. Disponible à l'adresse: https://fr.wikipedia.org/w/index.php?title=Docker_(logiciel)&oldid=194060763Page Version ID: 194060763

DQN explained line-by-line., 2021. [en ligne]. [Consulté le 13 septembre 2022]. Disponible à l'adresse: https://www.youtube.com/watch?v=fnVIgAGhA08

ESCAPE VELOCITY LABS, [sans date]. Advanced Reinforcement Learning in Python: cutting-edge DQNs. *Udemy*. [en ligne]. [Consulté le 11 octobre 2022]. Disponible à l'adresse: https://www.udemy.com/course/advanced-deep-qnetworks/

Frozen Lake - Gym Documentation, [sans date]. [en ligne]. [Consulté le 16 septembre 2022]. Disponible à l'adresse: https://www.gymlibrary.dev/environments/toy_text/frozen_lake/

Getting Started With OpenAI Gym, 2021. *Paperspace Blog*. [en ligne]. [Consulté le 16 septembre 2022]. Disponible à l'adresse: https://blog.paperspace.com/getting-started-with-openai-gym/

GORDIENKO, Andrew, 2021. Reinforcement Learning: DQN w Pytorch. *Medium*. [en ligne]. 5 mars 2021. [Consulté le 15 octobre 2022]. Disponible à l'adresse: https://andrew-gordienko.medium.com/reinforcement-learning-dqn-w-pytorch-7c6faad3d1e

Gym Documentation, [sans date]. [en ligne]. [Consulté le 16 septembre 2022]. Disponible à l'adresse: https://www.gymlibrary.dev/

HERMESDT, 2022. *hermesdt/reinforcement-learning*. [en ligne]. 24 juillet 2022. [Consulté le 15 octobre 2022]. Disponible à l'adresse: https://github.com/hermesdt/reinforcement-learning/blob/eb69484bb6d5a415633eb6e1fc07e12aa193cbb0/a2c/cartpole_a2c_episodic.ipynb

INFO@LAWTOMATED.COM, 2019. Supervised Learning vs Unsupervised Learning. Which is better? *lawtomated*. [en ligne]. 8 avril 2019. [Consulté le 11 octobre 2022]. Disponible à l'adresse: https://lawtomated.com/supervised-vs-unsupervised-learning-which-is-better/

KRISHNA, Velivela Vamsi, [sans date]. COMPARISON OF REINFORCEMENT LEARNING ALGORITHMS. . pp. 30.

Machine learning, 2022. *Wikipedia*. [en ligne]. [Consulté le 30 septembre 2022]. Disponible à l'adresse: https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=1112853960Page Version ID: 1112853960

Markov decision process, 2022. *Wikipedia*. [en ligne]. [Consulté le 6 octobre 2022]. Disponible à l'adresse: https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=1113076944Page Version ID: 1113076944

Markov Decision Processes, [sans date]. [en ligne]. [Consulté le 6 octobre 2022]. Disponible à l'adresse: https://cfml.se/blog/markov_decision_processes/

MNIH, Volodymyr, BADIA, Adrià Puigdomènech, MIRZA, Mehdi, GRAVES, Alex, LILLICRAP, Timothy P., HARLEY, Tim, SILVER, David et KAVUKCUOGLU, Koray, 2016. *Asynchronous Methods for Deep Reinforcement Learning*. [en ligne]. 16 juin 2016. arXiv. arXiv:1602.01783. [Consulté le 15 octobre 2022]. arXiv:1602.01783 [cs]

MNIH, Volodymyr, KAVUKCUOGLU, Koray, SILVER, David, GRAVES, Alex, ANTONOGLOU, Ioannis, WIERSTRA, Daan et RIEDMILLER, Martin, 2013. *Playing Atari with Deep Reinforcement Learning*. [en ligne]. 19 décembre 2013. arXiv. arXiv:1312.5602. [Consulté le 16 septembre 2022]. Disponible à l'adresse: http://arxiv.org/abs/1312.5602arXiv:1312.5602 [cs]

Bastien Maurice, 2018. Fonctionnement du neurone artificiel. *Deeply Learning*. [en ligne]. 15 septembre 2018. [Consulté le 12 octobre 2022]. Disponible à l'adresse: https://deeplylearning.fr/cours-theoriques-deep-learning/fonctionnement-du-neurone-artificiel/

openai/gym, 2022. [en ligne]. OpenAI. [Consulté le 9 octobre 2022]. Disponible à l'adresse: https://github.com/openai/gym

PARMAR, Ravindra, 2018. Common Loss functions in machine learning. *Medium*. [en ligne]. 2 septembre 2018. [Consulté le 13 octobre 2022]. Disponible à l'adresse: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23

Part 1: Key Concepts in RL — Spinning Up documentation, [sans date]. [en ligne]. [Consulté le 16 septembre 2022]. Disponible à l'adresse: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

PyTorch, [sans date]. [en ligne]. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://www.pytorch.org

PyTorch, 2022. *Wikipédia*. [en ligne]. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://fr.wikipedia.org/w/index.php?title=PyTorch&oldid=197159598Page Version ID: 197159598

Pytorch Vs Tensorflow Vs Keras: Here are the Difference You Should Know, 2020. *Simplilearn.com*. [en ligne]. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article

Q-learning, 2022a. *Wikipédia*. [en ligne]. [Consulté le 10 octobre 2022]. Disponible à l'adresse: https://fr.wikipedia.org/w/index.php?title=Q-learning&oldid=193543416Page Version ID: 193543416

Q-learning, 2022b. *Wikipedia*. [en ligne]. [Consulté le 10 octobre 2022]. Disponible à l'adresse: https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=1114881842#Discount_factorPage Version ID: 1114881842

Quelles sont les différences entre le Deep learning et le Machine learning ?, [sans date]. *IONOS Digital Guide*. [en ligne]. [Consulté le 1 octobre 2022]. Disponible à l'adresse: https://www.ionos.fr/digitalguide/web-marketing/search-engine-marketing/deep-learning-vs-machine-learning/

Qu'est-ce que le machine learning ?, [sans date]. [en ligne]. [Consulté le 2 octobre 2022]. Disponible à l'adresse: https://www.oracle.com/ch-fr/artificial-intelligence/machine-learning/what-is-machine-learning/

Reinforcement learning, 2022. *Wikipedia*. [en ligne]. [Consulté le 10 octobre 2022]. Disponible à l'adresse: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1114883736 Page Version ID: 1114883736

Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.12.0+cu102 documentation, [sans date]. [en ligne]. [Consulté le 10 août 2022]. Disponible à l'adresse: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Reinforcement Learning Series Intro - Syllabus Overview, [sans date]. [en ligne]. [Consulté le 14 octobre 2022]. Disponible à l'adresse: https://www.youtube.com/watch?v=nyjbcRQ-uQ8&list=PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv&index=1

Reprenons les bases : Neurone artificiel, Neurone biologique - Intelligence mécanique, [sans date]. [en ligne]. [Consulté le 12 octobre 2022]. Disponible à l'adresse: https://scilogs.fr/intelligence-mecanique/reprenons-bases-neurone-artificiel-neurone-biologique/

SAGAR, Ram, 2020. On-Policy VS Off-Policy Reinforcement Learning. *Analytics India Magazine*. [en ligne]. 11 avril 2020. [Consulté le 14 octobre 2022]. Disponible à l'adresse: https://analyticsindiamag.com/reinforcement-learning-policy/

SHARMA, SAGAR, 2021. Activation Functions in Neural Networks. *Medium*. [en ligne]. 4 juillet 2021. [Consulté le 12 octobre 2022]. Disponible à l'adresse: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

SILICON, 2017. What is the most popular language for machine learning? *Silicon Republic*. [en ligne]. 13 février 2017. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://www.siliconrepublic.com/advice/machine-learning-programming-language

Supervised vs. Unsupervised Learning: What's the Difference?, 2021. [en ligne]. [Consulté le 2 octobre 2022]. Disponible à l'adresse: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

TEWARI, Ujwal, 2020. Which Reinforcement learning-RL algorithm to use where, when and in what scenario? *Medium*. [en ligne]. 25 avril 2020. [Consulté le 5 octobre 2022]. Disponible à l'adresse: https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1

Top programming languages for data scientists in 2022, [sans date]. [en ligne]. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022

TOVAR, Alvaro Durán, 2020a. Advantage Actor Critic (A2C) implementation. *Deep Learning made easy*. [en ligne]. 3 janvier 2020. [Consulté le 15 octobre 2022]. Disponible à l'adresse: https://medium.com/deeplearningmadeeasy/advantage-actor-critic-a2c-implementation-944e98616b

Types of Machine Learning Algorithms with Use Cases Examples, 2019. *upGrad blog*. [en ligne]. [Consulté le 2 octobre 2022]. Disponible à l'adresse: https://www.upgrad.com/blog/types-of-machine-learning-algorithms/

Unsupervised learning, 2022. *Wikipedia*. [en ligne]. [Consulté le 2 octobre 2022]. Disponible à l'adresse: https://en.wikipedia.org/w/index.php?title=Unsupervised_learning&oldid=1107188105Page Version ID: 1107188105

WANG, Mike, 2021. Advantage Actor Critic Tutorial: minA2C. *Medium*. [en ligne]. 26 janvier 2021. [Consulté le 15 octobre 2022]. Disponible à l'adresse: https://towardsdatascience.com/advantage-actor-critic-tutorial-mina2c-7a3249962fc8

Why Use Python for AI and Machine Learning?, [sans date]. [en ligne]. [Consulté le 7 octobre 2022]. Disponible à l'adresse: https://steelkiwi.com/blog/python-for-ai-and-machine-learning/

YOON, Chris, 2019. Understanding Actor Critic Methods. *Medium*. [en ligne]. 17 juillet 2019. [Consulté le 20 septembre 2022]. Disponible à l'adresse: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

LAPAN, Maxim, 2018. Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more. Birmingham Mumbai: Packt Publishing. ISBN 978-1-78883-424-7.