Projet 2 : Apprentissage par renforcement

Le deuxième projet de ce cours était basé sur l'apprentissage par renforcement. L'apprentissage par renforcement s'inspire de la manière dont les humains apprennent par l'expérience. Contrairement à l'apprentissage supervisé, où un modèle est formé à partir d'un ensemble de données étiquetées, ou à l'apprentissage non supervisé, où il cherche des structures dans des données non étiquetées, l'apprentissage par renforcement repose sur l'interaction avec un environnement dynamique.

Dans ce cadre, un agent prend des décisions dans un environnement donné, dans le but d'atteindre un objectif. L'agent apprend en explorant l'environnement, en recevant des récompenses ou des punitions en fonction de ses actions, et en ajustant sa stratégie pour maximiser le cumul des récompenses au fil du temps.

Dans un premier temps, nous allons vous présenter le contexte du problème donné et ensuite nous aborderons les deux approches de l'apprentissage par renforcement que nous avons étudié à savoir le Q-Learning et le Deep Q-Learning.

Contexte

L'objectif du projet était de développer un jeu et d'apprendre à un agent quel était le chemin le plus court pour arriver à la récompense finale. Dans un premier temps, nous avons dû modéliser l'espace de jeu qui devait ressembler au schéma suivant :

L'espace de jeu est de dimension 4 par 4. L'agent démarre à la case *Start* et doit arriver jusqu'à l'arrivée où se trouve la récompense finale (case *Jail*). Certaines cases contiennent des dragons qui caractérisent des malus. Lorsque l'agent atterrit sur une de ces cases, il obtient une récompense négative et retourne à la case départ. Si l'agent sort de l'espace du jeu, il retourne sur la case précédente.

Pour créer l'espace de jeu, nous avons décider de créer un tableau Numpy de dimension 4 par 4 et contenant les valeurs des récompenses de chaque case. Pour cette première partie, nous avons choisi les récompenses suivantes : -1 pour les cases où se trouvent les dragons, 0 pour les cases neutres et 1 comme récompense finale à l'arrivée.

```
[[ 0.  0.  0.  0.]
 [-1.  0. -1.  0.]
 [ 0.  0.  0. -1.]
 [ 0. -1.  0.  1.]]
```

Ensuite, il a fallu créer une fonction permettant de simuler le déplacement de l'agent. Cette fonction est appelée *application_action* et prend en entrée 3 paramètres : l'action (*action*), la position actuelle de l'agent (*position*) et l'espace de jeu de la partie (*space*).

Lorsque l'agent joue un tour, il a 4 possibilités d'action. Il peut se déplacer d'une case sur le terrain de jeu soit vers le haut, soit vers la droite, soit vers le bas ou soit vers la gauche. Dans le code, ces actions sont caractérisées par un chiffre allant de 0 à 3 comme sur ce schéma.

La fonction *application_action* fonctionne de la manière suivante. On applique l'action précisée en paramètres à la position actuelle puis on récupère la nouvelle position de l'agent en s'assurant de le remettre sur l'espace de jeu si il sort des limites. A partir de la nouvelle position, on récupère la

récompense associée. Ensuite, si l'agent est tombé sur une case où se trouve un dragon, on le renvoit a la case départ et si il a atteint l'arrivée, on passe une variable *fin* à *true*. En sortie de cette fonction simulant un d éplacement de l'agent, on obtient donc sa nouvelle position (*nouvelle_position*), la récompense qu'il a obtenu grâce à ce déplacement (*reward*) et la variable de fin qui nous dit si la partie est terminée ou non (*fin*).

Développement du Q-Learning

Le Q-Learning est une méthode d'apprentissage par renforcement qui permet à un agent d'apprendre une politique optimale pour atteindre un objectif dans un environnement donné. Il s'agit d'un algorithme fondamental qui repose sur l'idée de calculer les valeurs Q, lesquelles quantifient la qualité d'une action spécifique dans un état donné.

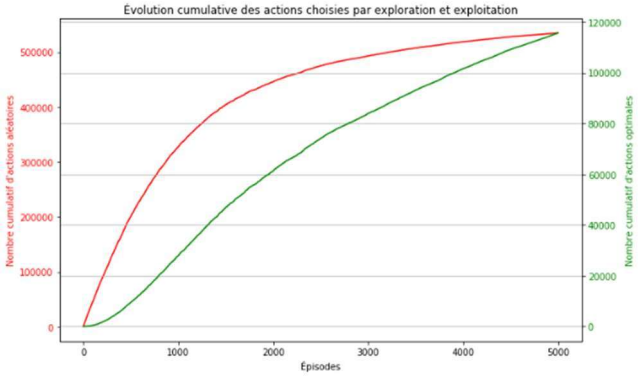
Dans le cadre du Q-Learning, la Q-table est une structure de données utilisée pour stocker les valeurs Q associées à chaque paire (s,a), où s représente un état de l'environnement et a une action que l'agent peut entreprendre dans cet état. Ces valeurs Q, notées $Q(s,a)$, représentent une estimation de la qualité d'effectuer une action a dans un état s, en supposant que l'agent suit une politique optimale par la suite. En d'autres termes, $Q(s,a)$ indique à quel point il est bénéfique pour l'agent de choisir une action donnée dans un état donné.

Dans notre projet, nous avons dû mettre en place tous ces principes du Q-Learning en l'appliquant à notre environnement pour permettre à l'agent de trouver la politique optimale au bout d'un certain nombre de parties.

Dans un premier temps, nous avons créé une fonction *choose_action* qui permet de savoir comment l'on choisit l'action à effectuer par l'agent. Dans le Q-Learning, on parle d'epsilon-greedy qui est une stratégie utilisée pour gérer le compromis entre exploration et exploitation. La variable epsilon est calculé avec le rapport entre le nombre de parties total et le nombre de parties total plus le nombre de parties déjà effectuées.

Au début de l'itération des parties, l'agent est dans un environnement inconnu et va donc choisir les actions a effectué au hasard pour explorer. Au bout d'un certain temps, il aura assez exploré l'environnement et va pouvoir commencer à se baser sur la Q-table pour choisir son action et prenant l'action qui possède la valeur la plus important dans l'état où il se situe.

Sur ce graphique, on observe que la valeur d'epsilon diminue de 1 jusqu'à 0,5 en fonction du nombre d'épisodes à savoir du nombre de parties.

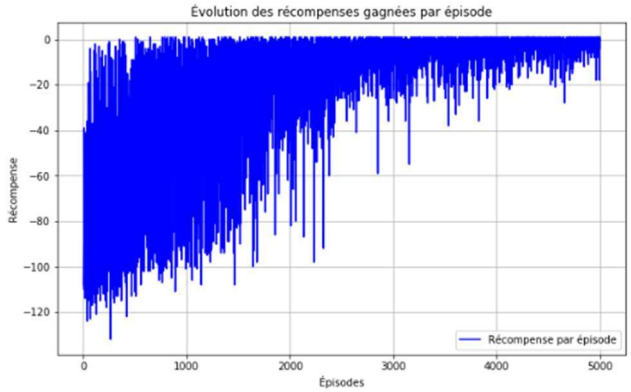


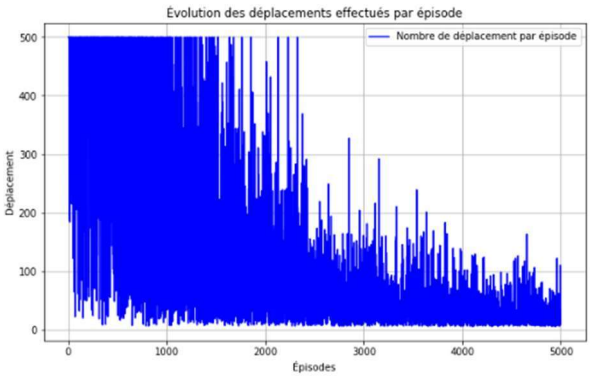
Sur ce graphique, on observe que pour 5000 parties, l'agent va beaucoup choisir ses actions au hasard au début lorsqu'il est en phase d'exploration tandis que vers la fin, il ne choisit plus que grâce à la Q-table car il est en phase d'exploitation.

Après avoir créé une fonction permettant de choisir l'action grâce à la méthode epsilon-greedy, il faut maintenant appliquer créer une fonction permettant le déplacement de l'agent dans l'espace de jeu. Pour cela, nous avons mis en place une fonction appelée *onestep* qui prend comme paramètres la Q-table, l'état actuelle de l'agent c'est-à-dire sa position et ensuite la valeur d'epsilon.

Dans un premier temps, on transforme la position de l'agent en index allant de 0 à 15 pour les 16 cases de l'espace de jeu. Ensuite, on utilise la fonction *choose_action* pour récupérer l'action que l'agent va devoir effectuer. Après cela, on applique cette action grâce à la fonction *application_action* présentée précédemment. Pour finir, il ne reste plus qu'à mettre la Q-table à jour grâce à l'équation de Bellman. Cette fonction *onestep* nous permet donc de modéliser un déplacement de l'agent mais désormais pour l'entrainer, il faut lui faire faire des parties entières.

Tout d'abord, on détermine un nombre de parties sur lequel nous souhaitons entraîner notre agent. Ensuite, nous effectuons une boucle sur le nombre de parties à faire puis nous initialisons de variables tel que l'epsilon, la position de départ de la partie ou encore le nombre de déplacement maximum que l'agent peut faire dans une partie afin que les parties ne durent pas indéfiniment si l'agent n'arrive pas jusqu'à l'arrivée. Pour finir, nous appelons la fonction *onestep* qui permet à l'agent d'effectuer une action et de mettre la Q-table à jour puis nous répétons cela tant que la partie n'est pas fini ou que le nombre de déplacement maximum n'est pas atteint.

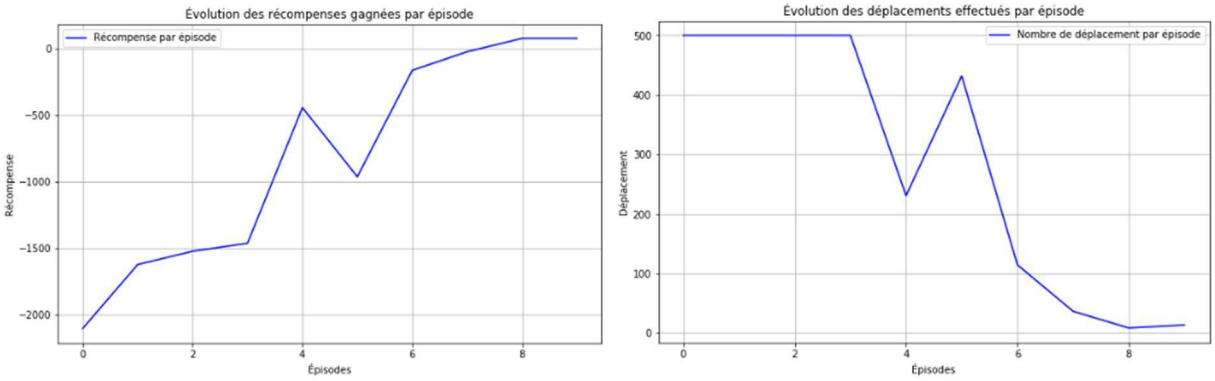




Sur ces graphiques, on observe l'évolution des récompenses obtenues et l'évolution du nombre de déplacements effectués au cours des différentes parties. On remarque clairement que sur les premières parties, l'agent obtient des récompenses très négatives et atteint très souvent le maximum de déplacement qui était fixé à 500 ce qui peut s'expliquer notamment par le fait qu'il est en phase d'exploration et qu'il ne connait pas l'environnement. Au fil des parties, on peut voir que les

récompenses obtenues augmentent et que le nombre de déplacements effectués par partie diminue car l'agent rentre en phase d'exploitation et qu'il se base sur la Q-table pour faire les meilleurs choix.

Dans un second temps, nous avons essayé de modifier le nombre de parties ainsi que les récompenses pour voir l'influence de ces paramètres sur l'apprentissage. Nous avons donc testé de mettre des récompenses à -20 sur les cases où se trouvent les dragons et à 100 pour la case d'arrivée. En ayant effectué plusieurs tests, nous avons remarqué qu'avec seulement 10 parties, l'entraînement pouvait suffire à trouver le chemin optimal.



Après l'entrainement, la Q-table est donc complétée avec des valeurs et peut nous permettre à chaque position quelle action est la meilleure. A partir de cela, on peut jouer une partie avec la Q-table et observe le chemin proposé.

```
Table Q après l'entraînement :
 [[ 71.40958623  78.83686071  47.45022526  71.4336833 ]
 [ 71.25102729  63.40201367  83.67379596  63.40289946]
 [ 59.12151173   0.          43.02435363  66.04485603]
 [  0.           0.           0.          44.03338181]
 [  0.           0.           0.           0.        ]
 [ 78.19540927  43.03311687  87.60661945  43.40020876]
 [  0.           0.           0.           0.        ]
 [  0.           0.         -19.99982123 -19.99999355]
 [ -2.0669483    0.           0.           0.        ]
 [ 68.3242996   91.8193277   51.00301638   0.        ]
 [ 42.13687867  41.24164283  95.93700099  71.53238347]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [ 91.2186349   99.99529541  91.89970655  33.45539317]
 [  0.           0.           0.           0.        ]]
Parcours optimal : [[0, 0], [0, 1], [1, 1], [2, 1], [2, 2], [3, 2], [3, 3]]
```

Grâce à la Q-table, on obtient le parcours optimal obtenu par l'agent après entraînement. On remarque que ce chemin correspond bien au chemin le plus rapide pour arriver à la récompense finale.

Deep Q-Learning

Le Deep Q-Learning est une avancée majeure en apprentissage par renforcement, qui combine la puissance des réseaux neuronaux profonds avec l'approche classique du Q-Learning. Ce cadre a été popularisé par DeepMind en 2015, lorsqu'ils ont démontré qu'un agent pouvait apprendre à jouer à des jeux Atari à un niveau surpassant parfois celui des humains, sans aucune connaissance préalable des règles du jeu. Le Deep Q-Learning est particulièrement adapté pour résoudre des problèmes où l'espace d'états est très vaste ou continu, ce qui rend impossible l'utilisation directe d'une Q-table pour représenter toutes les paires état-action possibles.

Bien que notre problème fonctionne avec l'algorithme de Q-Learning, il est intéressant d'observer comment on peut trouver le chemin optimal grâce au Deep Q-Learning. Cela pourrait également nous permettre par la suite d'augmenter l'espace de jeu avec plus de cases.

Comparé au Q-Learning présenté précédemment, certaines fonctions vont rester les mêmes mais certains changements vont avoir lui comme le remplacement de la Q-table par un réseau de neurones pour prédire la meilleure action dans un état donné.

Dans un premier temps, nous avons initialisé notre espace de jeu avec des récompenses différentes du premier exemple. Ici, nous avons fixé une récompense à -20 pour les cases des dragons, -1 pour les cases neutres et 100 pour la récompense finale c'est-à-dire l'arrivée. Nous avons également ajouté une fonction qui nous permet de transformer une position avec un X et un Y en un vecteur.

```
Position X/Y : [0,0]
Position en vecteur :  [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Ensuite, nous avons créé un modèle qui comprend deux couches Dense : une première composée de 8 neurones et une deuxième composée de 4 neurones soit le nombre d'actions possibles. Ce modèle va prendre un entrée notre vecteur représentant l'état de l'agent et en sortie va nous donner une valeur par neurone qui représente les différentes actions et où l'on va récupérer le maximum pour savoir quelle action choisir en phase d'exploitation.

Après avoir déterminé notre action, on crée le vecteur du prochain état de l'agent après son action. Ensuite, nous allons utiliser une copie de notre modèle pour prédire les valeurs des sorties du réseau selon le prochain état puis récupérer le maximum de ces valeurs. On calcule ensuite pour finir la valeur cible grâce à cette formule.

$$target = Reward + gamma * next\_Q\_max * (1-fin)$$

La dernière partie de cet algorithme de Deep Q-Learning consiste à programmer notre propre descente de gradient. Cela correspond au calcul de la perte (loss), du gradient, et à l'application des gradients pour mettre à jour les poids du modèle.

Après avoir fini d'entraîner notre modèle, on peut jouer une partie suivant le parcours optimal que le modèle va prédire et observer s'il est similaire à celui trouver avec l'algorithme de Q-Learning. Lorsque le modèle s'entraîne sur 300 parties, il arrive à trouver le chemin optimal qui permet d'arriver le plus rapidement à la récompense finale.

```
Parcours optimal avec DQN : [[0, 0], [0, 1], [1, 1], [2, 1], [2, 2], [3, 2], [3, 3]]
```

Grâce à cet algorithme de Deep Q-Learning, nous pourrions désormais établir un espace de jeu de la dimension que l'on souhaite car l'on pourrait traiter ce problème dans n'importe quelle situation ce qui n'est pas le cas avec l'algorithme de Q-Learning où la Q-table nous restreint sur les possibilités d'actions et d'états.