DEEP LEARNING

Master 2 Actuariat, Data Science pour l'Actuariat

MountainCar & T-Rex, Run !

Le Mans Université, le 15 Février 2020

EPLENIER Mathieu

ROBERT Thibault

SADE Vincent

LE LOUARN Romain

Table des matières

Introduction

Ordinateurs, puissance de calculs, volumes grandissant de données et réseaux de neurones. L'informatique s'est largement développée ces dernières années et avec elle ont émergés de nouvelles possibilités, de nouveaux enjeux et des techniques innovantes qui ont et qui vont continuer de révolutionner nos outils numériques.

Nous sommes aujourd'hui à l'ère de l'intelligence artificielle, des pétaoctets de données et de l'apprentissage. Le Deep Learning, que l'on traduit littéralement par "apprentissage profond" est un type d'intelligence artificielle, dérivé du Machine Learning ou "apprentissage automatique".

Avec le Machine Learning, l'être humain a développé des programmes permettant à la machine d'apprendre par elle-même, de manière quasi autonome. Cette notion d'humanisation de la réflexion et d'apprentissage se retrouve dans le Deep Learning, au travers des réseaux de neurones sur lesquels ce-dernier repose. A la manière du cerveau humain, le réseau prend un ou des inputs et, à la suite d'un enchaînement d'opérations produit un résultat, ou output, en sortie. Le tout est articulé en un système de couches successives de neurones, où la couche "i" reçoit et traite les outputs de la couche "i-1".

Le Machine Learning se décompose en trois domaines. D'un côté nous trouvons l'apprentissage supervisé, où l'objectif du réseau est d'apprendre à prédire à partir d'une base de données "annotées", c'est à dire que les images constituantes de la base ont été étiquetées afin d'indiquer ce qu'elles représentent. Très logiquement, en face de l'apprentissage supervisé nous pouvons mentionner l'apprentissage non supervisé, qui comme son nom l'indique, ne fournit pas une étiquette à chaque élément de notre base. L'utilisateur se contente d'envoyer toutes les informations au réseau de neurones et l'apprentissage doit se faire par lui-même, l'enjeu étant alors de réussir à dégager une "structure de prédiction". Enfin, un dernier aspect du Machine Learning qui va particulièrement nous intéresser ici est le Reinforcement Learning. Celui-ci peut être défini par le développement des capacités, pour un agent précis évoluant dans un environnement donné, à devenir autonome au travers d'un système d'actions-récompenses. Autrement dit, l'agent va, au fur et à mesure de ses actions, apprendre dans le but de faire les bons choix et prendre les bonnes décisions d'actions à mener pour maximiser son score, ou tout du moins l'optimiser.

Dans le cadre de ce travail, notre réflexion s'est portée sur le Reinforcement Learning. Notre ambition était de travailler sur des environnements numériques et de faire évoluer des agents au sein de ces environnements grâce à un réseau de neurones. Les jeux sur lesquels nous nous sommes appuyés, à savoir "MountainCar" et "T-Rex, Run !", ne nous ont pas placés devant les mêmes contraintes et enjeux. Avec "MountainCar" notre apprentissage consistait en l'analyse des coordonnées de la voiture dans le plan et à l'affectation en conséquence d'une récompense qui serait maximale lorsque la montagne serait gravie ; pour "T-Rex Run", les enjeux étaient tout autres, puisqu'il nous fallait capter, découper et analyser l'image pour que l'agent puisse apprendre et finisse par prendre les décisions adéquates devant les obstacles qui se présentaient, toujours en vue de maximiser sa récompense, ou plutôt son score.

Cadre théorique

Le Reinforcement Learning étant une extension particulière du Deep Learning, cette dernière nécessite des outils mathématiques adaptés afin de permettre à l'ordinateur d'obtenir de meilleurs résultats au fil des parties. Il existe une base commune à plusieurs méthodes de Reinforcement Learning. Cette dernière est l'équation de Bellman, portant le nom du mathématicien Richard Bellman, à l'origine de la programmation dynamique, une méthode algorithmique pour résoudre des problèmes d'optimisation introduite dans les années 1950. L'équation de Bellman permet d'écrire la "valeur" d'une décision à un problème, à un temps donné, d'une part en fonction des gains obtenus pour avoir atteint l'état actuel, et d'autre part en fonction de la valeur de la décision future à prendre en fonction de toutes les actions possibles dans l'état suivant.

Cette méthode algorithmique se base sur un principe de "Reward and Return", ce qui veut dire que l'agent apprend à maximiser la somme future de ses gains. Une écriture usuelle de ce fondement est :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad avec \quad 0 < \gamma < 1$$

Cette équation décrit l'obtention des gains futurs. Cette écriture existe également sans γ, cependant cette série ne converge pas ; tandis qu'avec γ la série est définie à l'infini. γ est le facteur d'atténuation ou *discount rate*. Ce dernier permet de donner moins d'importance aux récompenses des actions futures. Cela est nécessaire car ces actions futures sont incertaines. Plus γ est petit, moins nous accordons d'importance aux gains futurs.

A présent, intéressons-nous à l'équation de Bellman, qui est essentielle dans notre projet. Voici une écriture possible de cette équation :

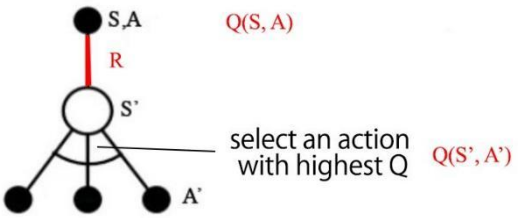
$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

où :
s est l'état actuel
s' est l'état futur
a est l'action
$Q(s,a)$ est la "target", c'est à dire la valeur de la décision au problème au temps s.
$r(s,a)$ est la reward acquise pour être arrivé à cet état avec cette action.
$\gamma max_a Q(s',a)$ est le maximum, atténué par γ, de la valeur de la décision future à prendre en fonction de toutes les actions possibles dans l'état suivant.

Cette équation sert de fondement aux Deep Q Network (DQN) et aux Double Deep Q Network (Double DQN). Ces deux méthodes d'apprentissage font partie du Q-Learning.

Le Deep Q Network (DQN) est un algorithme d'apprentissage par renforcement. Cette technique ne nécessite aucun modèle initial de l'environnement. Le but est "d'apprendre" la fonction d'action valeur (s, a) et d'estimer quelle est la meilleure action à réaliser lorsque nous sommes à un état particulier. Par exemple dans notre cas : 1- Pour le jeu MountainCar, il faut que la voiture estime s'il est préférable d'avancer, reculer, ou ne rien faire ; 2- Pour le jeu du "T-Rex Run !", le dinosaure doit estimer si dans un état donné il faut sauter pour esquiver un cactus, ou s'il faut simplement continuer à courir.

Dans un modèle de Q-Learning, on construit une matrice $Q(s, a)$ pour stocker en mémoire les Q-valeurs de toutes les combinaisons possibles de s et a, donc d'états et d'actions. Dans un cas pratique, nous simulons une action à partir de l'état actuel. Nous obtenons une reward et un nouvel état s'. A partir de la matrice $Q(s, a)$ nous déterminons la prochaine action a' qui possède la valeur maximum $Q(s', a')$.



Si l'on prend une action a et que l'on regarde sa reward r associée, nous obtenons une vision avec un pas d'avance. $r + Q(s', a')$ devient la target que $Q(s, a)$ doit atteindre. Nous retombons ainsi sur l'équation de Bellman.

Dans un simple DQN, nous pouvons faire face à des situations de surapprentissage et d'un mauvais fitting. En effet, nous ne pouvons avoir la certitude que la meilleure action pour le prochain état est l'action avec la plus grande Q-valeur. Nous savons que la précision des Q-valeurs dépend des actions que nous avons déjà tenté et des états voisins que nous avons explorés. Ce qui fait que lors des premières actions de la phase d'entraînement, il est difficile de savoir quelle est la meilleure action à effectuer. La maximum Q-valeur peut conduire à de mauvaises décisions. Une solution envisageable est de calculer la Q-target et d'utiliser en parallèle deux réseaux de neurones pour découpler la sélection de l'action à partir de la Q-valeur. C'est le Double Deep Q Network (Double DQN) :

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

Le Double DQN nous permet de réduire la surestimation des Q-valeurs. Cela nous permet par conséquent un entraînement plus rapide et un apprentissage plus stable.

Se lancer avec MountainCar

1) MountainCar, qu'est-ce que c'est ?

MountainCar est un "petit jeu" se trouvant sur "gym.openai.com" et développé afin de permettre à des utilisateurs, comme nous, de tester leurs réseaux de neurones. Le jeu est tout à fait simpliste dans sa conception. L'agent, ici la voiture, est positionné sur une piste située entre deux montagnes. L'objectif est alors de gravir la montagne de droite et d'atteindre le drapeau au sommet de cette dernière, pour mettre fin à la partie et la remporter. Cependant, la difficulté réside dans le fait que la voiture ne peut y parvenir en une seule accélération. La stratégie à adopter est alors d'utiliser les pentes des deux montagnes pour accumuler de la vitesse, et par ce jeu d'allers-retours successifs réussir à atteindre le drapeau.

Interagir avec le jeu se fait alors assez simplement au travers de trois actions : aller vers l'avant, aller vers l'arrière et ne rien faire. Le tout étant d'avoir le bon timing pour réussir à "optimiser" le mouvement de balancier créé par les allers-retours de notre agent et bien sûr atteindre le drapeau en un nombre réduit de coups.

2) L'intéraction avec l'environnement numérique

Intéragir avec l'environnement du jeu ne nous a pas posé de difficultés particulières. En effet, l'environnement "gym" étant déjà implémenté dans un module Python, nous n'avons pas rencontrés de contraintes "techniques" pour entrer en intéraction avec l'agent.

Grâce à un ensemble de fonctions fournies dans ce module, nous avons eu la possibilité, à chaque instant du jeu, de demander à la voiture de réaliser les trois actions mentionnées ci-dessus ; nous avons également pu récupérer les données nécessaires à l'entraînement de notre modèle, notamment les coordonnées de la voiture dans le plan ; enfin, il nous a été possible de capter la fin de la partie, c'est à dire que nous observons un "True" si le drapeau est atteint et un "False" dans le cas contraire.

A savoir qu'une fonction de récompense, propre au jeu, est d'ores et déjà présente dans "gym". Cependant nous avons dû la "retravailler" et l'adapter à nos besoins.

3) Développement et explications

Dans cette partie, chaque sous-partie correspond aux sections du code Python. Nous nous efforcerons d'expliquer ce code, les choix effectués et les sorties obtenues.

a) Environnement

Avec cette portion du code, nous chargeons le jeu contenu dans le module "gym". Nous pouvons alors récupérer le nombre d'actions qui seront allouées à la voiture, c'est à dire les trois possibilités de mouvement définies ci-dessus ; cela nous indique donc le nombre de sorties du modèle.

En outre, nous captons également la dimension des observations. Ces dernières sont de dimension 2, puisque l'agent évolue dans un plan. Le réseau de neurones aura alors deux paramètres à analyser ; les coordonnées (x,y) de la voiture dans le plan seront par conséquent les inputs du modèle.

b) Test de l'environnement

Le but est ici de lancer un certain nombre de parties avec des actions aléatoires. Cela permet à l'utilisateur de prendre connaissance des différentes actions possibles et de la récompense allouée à chaque action.

Cette étape n'a pas de but pratique pour notre modèle ; on peut la considérer comme facultative. Elle permet simplement de mieux comprendre comment se comporte l'environnement.

c) Paramètres

Nous définissons dans cette section les paramètres modifiables utilisés. Le réseau de neurones comprend un paramètre :

- "**learning_rate**" assimilable à la "vitesse d'apprentissage" : il correspond au pas de la descente de gradient de la fonction Loss. Nous avons fait le choix d'un apprentissage long, mais plus précis ; c'est pourquoi nous avons fixé notre *learning_rate* à 0.0025.

Le reste des paramètres est en lien direct avec le modèle utilisé :

- "**memory**" : représente la mémoire utilisée pour stocker toutes les expériences (historique des observations, actions et récompenses associées) accumulées au cours de l'entraînement. Nous avons arbitrairement fixé sa capacité maximale à 1000000 d'enregistrements.
- "**batch_size**" : correspond au nombre de fois où le réseau de neurones va s'entrainer sur les expériences passées entre chaque partie. Nous l'avons fixé à 48.

- “**discount_rate**” : représente le facteur utilisé dans l’équation de Bellman pour donner moins d’importance aux récompenses des actions futures. Au vu de ce qui se fait sur la plupart des algorithmes de Reinforcement Learning, nous l’avons établi à 0.95.
- “**epsilon**” : correspond au pourcentage d’actions aléatoires. Nous l’initialisons à 1, soit 100%.
- “**epsilon_min**” : représente le pourcentage final d’actions aléatoire. Nous l’avons fixé à 0.01, soit 1%.
- “**epsilon_decay**” : correspond à un coefficient multiplicatif de descente appliqué au pourcentage d’action aléatoires. Nous l’avons établi à 0.999, c’est à dire qu’à chaque partie la part d’actions aléatoires est multipliée par 0.999, ce qui la réduit au fur et à mesure de l’avancé de l'entraînement.

d) Réseau de neurones

Le réseau de neurones que nous avons décidé d’implémenter comporte trois couches. Développons les :

- *Input layer* : c’est une couche de 32 neurones qui reçoit les coordonnées (x,y) de la voiture dans le plan.
- *Hidden layer* : également composée de 32 neurones, elle possède une activation de type “relu”.
- *Output layer* : composée d’autant de neurones qu’il y a de nombre d’actions et dotée d’une activation de type “linear”.

Nous compilons ensuite le réseau en prenant une fonction loss de type “mse” (mean square error) et un optimiseur de type “Adam”.

Pourquoi le choix de cette structure ? A la vue des différents algorithmes dont nous nous sommes inspirés, cette structure relativement simple est apparue comme la plus fréquemment utilisée.

Pour la *Hidden layer*, l’activation de type “relu” se justifie par le fait qu’elle est relativement peu coûteuse en terme de calculs, ce qui améliore les performances de l’algorithme.

Concernant la *Output layer* nous avons utilisé une activation de type “linear” en raison de l’objectif poursuivi par le modèle. En effet, le Reinforcement Learning est globalement un problème de régression et non de classification. Ici, nous cherchons à prédire des valeurs réelles représentant les récompenses associées à chaque action. Ainsi, l’utilisation d’une activation de type “softmax”, utilisée pour la classification, n’est ici pas adéquate.

De plus, la fonction loss de type “mse” prend tout son sens ici car optimale dans un contexte de régression.

Enfin, les tendances actuelles en Reinforcement Learning montrent qu’un *optimizer* de type “Adam” est optimal. Il permet de gagner en vitesse de calculs, en comparaison de l’*optimizer* de type “RMSprop”, lui aussi populaire.

e) Pré-entraînement de la mémoire

Cette étape de notre code nous permet d'emmagasiner dans la *memory* un petit nombre d'expériences basées sur des actions d'une part aléatoires, et d'autre part prédites par un réseau non encore entraîné.

Pour pouvoir commencer l'entraînement, le nombre d'expériences minimum requis dans la *memory* est de 48, soit le *batch_size* défini plus haut. Ces 48 expériences sont nécessaires dans la première étape de l'entraînement du modèle, ce qui justifie cette étape.

f) Entrainement du modèle

Tout d'abord, il convient de débuter notre propos en définissant les concepts d'exploration et d'exploitation. L'exploration représente la part d'actions aléatoires, soit le *epsilon* à la partie p. L'exploitation est quant à elle la part d'actions prédites par le modèle à la partie p. On en déduit très logiquement que "exploitation = 1 - exploration" ; ainsi, à mesure que l'entrainement du modèle avance et que l'exploration diminue, le niveau d'exploitation.

D'un point du vue pratique, cela signifie que nous diminuons la part d'actions aléatoires possibles au profit des actions apprises au cours de l'apprentissage du modèle. Le but poursuivi est d'éviter que l'agent ne se focalise sur une éventuelle solution sub-optimale.

Nous allons maintenant expliquer les différentes étapes constituant l'entraînement du modèle. Ces dernières sont valables pour chaque partie jouée.

Nous réinitialisons en premier lieu l'environnement afin de replacer la voiture dans son état initial, c'est à dire entre les deux montagnes.

Dès à présent, chacune des étapes détaillées ci-après est valable pour chaque action lors d'une partie.

(1) Choix entre une action aléatoire et une action prédite par le modèle à l'aide de l'*epsilon*.

(2) L'agent effectue l'action choisie, puis nous récupérons les nouvelles informations de l'environnement :
   - Le nouvel état dans lequel se trouve l'agent, c'est à dire les nouvelles coordonnées (x,y) de la voiture dans le plan.
   - La récompense associée à l'action menée par l'agent.
   - Le fait que la partie soit gagnée, sous forme de booléen "True/False", c'est à dire que le drapeau est atteint par l'agent.

(3) Utilisation de la récompense collectée à l'étape (2), et transformation à l'aide d'une nouvelle fonction de récompense, implémentée par nos soins.
   Nous avons décidé de modifier la fonction de récompense déjà présente dans le jeu avec la nôtre, afin de gagner en performance sur le modèle. En effet, notre approche permet de mieux ajuster la récompense en fonction des actions choisies.

(4) Collecte de l'expérience passée, c'est à dire mémorisation de :
    - L'état à l'instant précédent.
    - L'action choisie à l'instant précédent.
    - La récompense obtenue à la suite de l'action précédente.
    - Le booléen traduisant le gain ou non de la partie permettant de savoir si l'action précédente a mené ou non au gain de la partie.

(5) Si l'agent atteint le drapeau, ou si le nombre d'actions qui lui sont allouées est écoulé, la partie prend fin.

Désormais, nous allons expliquer ce qui se passe entre chaque partie, soit l'entraînement du modèle sur les expériences emmagasinées. Nous sélectionnons aléatoirement dans la *memory* un nombre d'expériences correspondant au *batch_size*.

Pour chacune de ces expériences, nous allons faire apprendre le réseau de neurones de la manière suivante :

(1) Deux cas peuvent être rencontrés :
    - L'expérience a mené à une victoire ; nous appliquons alors l'équation de Bellman.
    - Dans le cas contraire, nous appliquons l'équation de Bellman diminuée des récompenses liées aux actions futures.

(2) Nous "fittons" le réseau de neurones entre les inputs, à savoir l'état précédent ; et les outputs, représentés par les récompenses maximales associées aux actions.

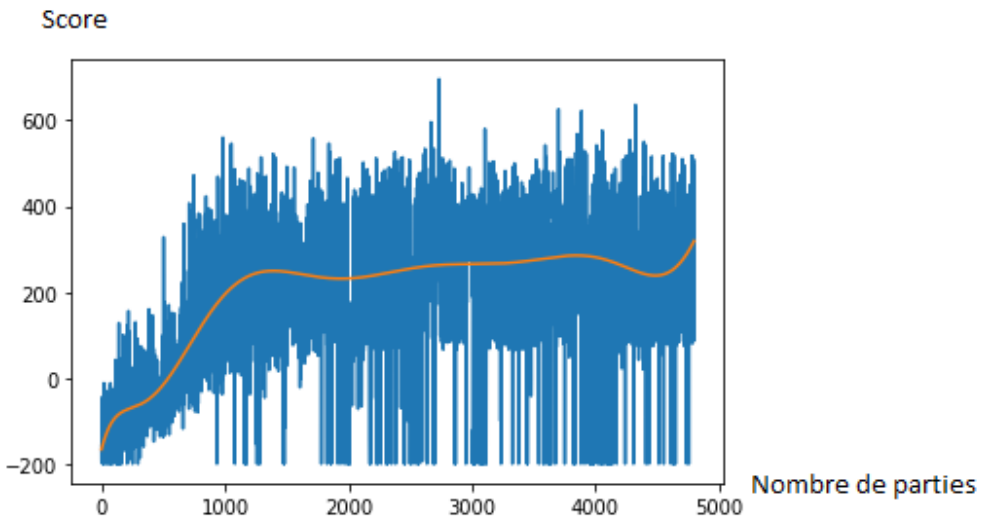
Toujours entre chaque partie, nous diminuons la part d'exploration grâce à *l'epsilon_decay*.

Enfin, toutes les 50 parties, nous réalisons un "checkpoint". Cela consiste à venir sauvegarder les poids du réseau de neurones. L'intérêt pratique est multiple :

- Pouvoir arrêter l'entraînement et le reprendre plus tard.
- Avoir un système de sauvegarde en cas de bug de l'algorithme.

De plus, nous traçons également l'évolution des scores en fonction du nombre de parties, toujours toutes les 50 parties pour rendre compte de la tendance de l'apprentissage.

Ci-dessous un exemple de courbe des scores obtenus :



Après 5000 parties jouées, nous constatons que la tendance, en orange sur le graphe, évolue globalement positivement ; néanmoins des phases de stagnation, voire de légères décroissances sont observées.

Même avec une tendance à la hausse des scores, nous remarquons qu'il y a tout de même une volatilité relativement significative. Cela peut provenir de la fonction de récompense codée par nos soins, qui peut à priori se révéler non optimale. Nous pourrions également augmenter la durée de l'apprentissage, c'est à dire le nombre de parties jouées, tout en augmentant l'*epsilon_decay* ; cela aura pour conséquence d'augmenter le temps de passage de l'exploration pure à l'exploitation pure et de permettre ainsi de ne pas retenir les mauvaises actions, conduisant à un mauvais score.

Remarque :

Vous trouverez ci-dessous, deux liens pour accéder à deux vidéos intitulées *MC_NoTrain* et *MC_Train*. La première représente le modèle non entraîné, tandis que la seconde représente le modèle entraîné après 5000 parties.

MC_NoTrain : :
https://drive.google.com/file/d/11wbA0pXlkG0XM4TDMG9eRMjeuPFHGdXO/view?usp=sharing
MC_Train :
https://drive.google.com/file/d/1mCDFXKI1xLCfZpWl6e6OcYvyqJsIM-9R/view?usp=sharing

T-Rex, Run !

1) Pourquoi nous lancer sur ce jeu ?

Notre objectif initial était de pouvoir faire du Reinforcement Learning en travaillant sur un jeu à la portée de tous les utilisateurs sur leur ordinateur. L'idée du "T-Rex, Run !" nous est apparue comme idéale puisqu'il reste relativement accessible, de par son nombre d'actions possibles ou en ce qui concerne la simplicité des graphismes, tout en étant cependant plus complexe que le MountainCar. Le but étant de pouvoir adapter nos algorithmes sur de futurs jeux de plus en plus avancés.

2) Quelques mots sur T-Rex, Run !

"T-Rex, Run !" est un jeu disponible sur le navigateur "Google Chrome" qui s'ouvre automatiquement lorsqu'un utilisateur ne dispose d'aucune connexion internet. L'agent, cette fois un dinosaure, s'élance dans une course sans fin. Ce type de jeu est qualifié de "Endless Runner". Son objectif est alors d'éviter tous les obstacles qui se dresseront sur son chemin afin de courir le plus longtemps possible et maximiser son score. La partie prend fin au moment où l'agent heurte un cactus ou encore un ptérodactyle. La difficulté réside ici dans le fait que le dinosaure ne peut que sauter ou se baisser pour les esquiver, le jeu avançant automatiquement devant lui de plus en plus vite.

3) L'intéraction avec l'environnement

Dans ce deuxième jeu, intéragir avec l'environnement du jeu a nécessité plus de recherche. En effet, ce-dernier n'étant pas directement implémenté dans un module Python comme précédemment, nous avons dû capter les images directement sur le navigateur et récupérer les informations du jeu contenues sur la page internet pour pouvoir intéragir avec elles.

Grâce à un ensemble de fonctions provenant de différents packages Python, nous avons pu, à chaque instant du jeu, demander à notre dinosaure de réaliser deux des trois actions possibles : le saut et ne rien faire. Nous avons également réussi à récupérer les données nécessaires à l'entraînement de notre modèle, notamment le score de manière dynamique, ainsi que communiquer avec lui en le relançant automatiquement suivant les échecs de l'agent. Enfin, nous avons capté le fait que la partie se termine suite à un contact avec un obstacle. Il faut également préciser que nous avons choisi de ne pas mettre

d'accélération de défilement du paysage sur notre jeu afin de nous faciliter l'apprentissage, le jeu étant toujours généré aléatoirement mais de manière linéaire.

De plus, avec ce deuxième jeu, nous avons dû créer nous-même la fonction de récompense, que nous détaillerons un peu plus loin.

4) Développement et explications

Cette partie s'articulera de la même manière que précédemment. Cependant nous détaillerons ici les choix qui nous ont amenés à essayer trois différents modèles ; l'explication de ces modèles sera réalisé dans l'ordre de leur test.

Nous avons tout d'abord tenté d'adapter notre code DQN provenant du MountainCar afin de tester son efficacité sur une autre catégorie de jeu. Puis, nous avons changer notre modèle, passant du DQN a un "Double DQN", censé être plus performant.

Enfin nous avons comparé nos résultats à l'aide d'un code déjà présent pour ce jeu sur internet que nous avons retravaillé pour nos besoin.

a) Environnement

Avec cette portion du code, nous chargeons le jeu et communiquons avec lui à l'aide du module "selenium". Ce dernier nous permet de fixer les paramètres du jeu désirés grâce au "chromedriver", nous assurant la communication entre notre code Python et notre navigateur Chrome. Il nous est alors possible de récupérer le score ainsi que l'état dans lequel se trouve le jeu ("Game over", "running", etc...). Celui-ci nous a enfin permis de pouvoir relancer la course à chaque fin de partie de notre agent et de lui communiquer ses deux actions possibles.

Suite à cette communication, il nous a fallu capter les images du jeu de manière dynamique afin que notre dinosaure puisse savoir où il se situe à tout moment. Pour ce faire, nous avons testé deux packages différents : Pillow et MSS. Leur fonctionnement est très similaire, cependant, MSS permet une capture d'image plus rapide que Pillow. Ces images ont ensuite dû être retravaillées afin de disposer d'une vitesse de calcule plus optimale. Pour se faire, nous avons réduit la taille de l'image à analyser, que ce soit en modifiant la taille de la fenêtre du navigateur, le nombre de pixels ainsi que la couleur des images, c'est à dire le passage des images en noir et blanc.

b) Test de l'environnement

De même que pour le MountainCar, le but est encore une fois de pouvoir lancer un certain nombre de parties avec des actions aléatoires. Cela permet à l'utilisateur de prendre connaissance des différentes actions possibles et de la récompense allouée à chaque action.

Cette étape n'a ici pas non plus de but pratique pour notre modèle ; on peut la considérer comme facultative. Elle permet simplement de mieux comprendre comment se comporte l'environnement.

c) Paramètres

Nous définissons dans cette section les paramètres modifiables utilisés sans entrer dans les détails de ceux présentés sur le MountainCar.

Le réseau de neurones comprend un paramètre :

- "**learning_rate**" que nous avons fixé à 0.001.

Le reste des paramètres est en lien direct avec le modèle utilisé :

- "**memory**" : nous avons arbitrairement fixé sa capacité maximale à 30000 d'enregistrements.
- "**batch_size**" : nous l'avons fixé à 32 pour le simple DQN et 48 pour le double DQN.
- "**discount_rate**" : nous l'avons établi à 0.95.
- "**epsilon**" : nous commençons à 0.1, soit 10%.
- "**epsilon_min**" : nous l'avons fixé à 0.0001, soit 0.01%.
- "**epsilon_decay**" : nous l'avons établi à 0.99995.
- "**reward**" : les rewards servent à donner une "récompense" aux différentes actions effectuées par notre agent, ce qui permet de donner un "poids" aux différents choix que l'agent peut faire. Notre choix ici a été de fixer à -100 en cas de crash de l'agent, -5 en cas de saut et 1 s'il ne fait rien. Le but ici est de faire comprendre à l'agent qu'il ne faut pas se heurter à un obstacle et essayer de faire une course linéaire, c'est-à-dire ne pas sauter inutilement.

d) Réseau de neurones

Le premier réseau de neurones que nous avons décidé d'implémenter comporte cinq couches. Développons les :

- *Couches de convolutions* : nous disposons de trois couches de convolution ; une première de 32 filtres, suivie de deux de 64 filtres qui reçoivent les images.
- *Flatten* : sert à faire le liens entre la dernière couche de convolution et la première couche de neurone.
- *Couches de neurones* : une couche *Hidden layer* composée de 512 neurones et une couche *Output layer* composée d'autant de neurones qu'il y a de nombre d'actions.

Nous compilons ensuite le réseau en prenant une fonction loss de type "mse" (mean square error) et un optimiseur de type "Adam".

Pourquoi le choix de cette structure ? Cette architecture de réseau de neurones de convolution a été développée par Google DeepMind et est une des plus utilisées dans le domaine.

Concernant les choix des types d'activations et d'optimizer, les explications sont les mêmes que pour MountainCar.

La structure du Double DQN nécessite de créer deux réseaux de neurones identiques, un "*model*" et un "*target_model*". Nous avons pris la structure de Google DeepMind pour nos deux CNN. Leur fonctionnement et leur utilité sera expliqué ci-dessous dans la section intitulée "Entraînement des modèles".

e) Pré-entraînement

Dans l'algorithme du simple DQN, nous avons choisi de commencer l'entraînement une fois une grande quantité d'action stockés dans la memory (750 parties) afin de pouvoir réduire l'aléatoire et le risque de sur-apprentissage sur un mauvais fitting. Concernant l'autre modèle, cette étape n'est pas nécessaire car le "Double DQN" se veut moins sensible au sur-apprentissage, par conséquent nous pouvons directement commencer l'entraînement du modèle.

f) Entraînement des modèles

De même que pour la partie sur le MountainCar, nous avons gardé une partie d'exploration et d'exploitation.

Nous allons maintenant expliquer les différentes étapes constituant l'entraînement de nos deux modèles respectifs. Ces dernières sont valables pour chaque partie jouée.

Nous réinitialisons en premier lieu l'environnement afin de relancer la course du dinosaure.

Dès à présent, chaque étape détaillée est valable pour chaque action lors d'une partie.

(1) Choix entre une action aléatoire et un action prédite par le modèle à l'aide de l'*epsilon*.

(2) L'agent effectue l'action choisie, puis nous récupérons les nouvelles informations de l'environnement :
- Le nouvel état dans lequel se trouve l'agent, c'est à dire sa position ainsi que la position de son environnement.
- La récompense associée à l'action menée par l'agent.
- Le score final ainsi que le fait que la partie soit terminée suite à un choc contre un obstacle.

(3) Utilisation de la récompense collectée à l'étape (2), et transformation à l'aide d'une nouvelle fonction de récompense, implémentée par nos soins.

(4) Collecte de l'expérience passée, c'est à dire mémorisation de :
- L'état à l'instant précédent.
- L'action choisie à l'instant précédent.
- La récompense obtenue à la suite de l'action précédente.
- Le booléen traduisant la fin de la partie permettant de savoir si l'action précédente a mené au Game Over.

(5) Si l'agent se heurte à un obstacle, la partie prend fin.

Désormais, nous allons expliquer ce qui se passe entre chaque partie, soit l'entraînement du modèle sur les expériences emmagasinées. Nous sélectionnons aléatoirement dans la *memory* un nombre d'expériences correspondant au *batch_size*.

Dans un des deux modèle, le système est un DQN simple, le principe est donc le même que MountainCar. Le deuxième modèle étant en double DQN, le fonctionnement est différent. Voici respectivement les différentes méthodes:

(a) Deux cas peuvent être rencontrés :
- L'expérience n'a pas mené à une défaite ; nous appliquons alors l'équation de Bellman.
- Dans le cas contraire, nous appliquons l'équation de Bellman diminuée des récompenses liées aux actions futures.

(b) Deux cas peuvent être rencontrés :
- L'expérience n'a pas mené à une défaite ; nous appliquons alors l'équation modifiée de Bellman (voir l'équation du double DQN ci-dessus dans la section cadre théorique).
- Dans le cas contraire, nous appliquons l'équation de Bellman diminuée des récompenses liées aux actions futures.

Le Double DQN estime l'action à prendre au prochain état grâce au *CNN model* et finit par estimer la Q_valeur associée au fait de choisir cette action future par rapport à cet état futur grâce au *CNN taget_model.*

(a) Nous "fittons" le réseau de neurones entre les inputs, à savoir l'état précédent ; et les outputs, représentés par les récompenses maximales associées aux actions.

(b) Nous "fittons" le réseau de neurones entre les inputs, à savoir l'état précédent ; et les outputs, représentés par les récompenses associées aux actions actuelles et futures.

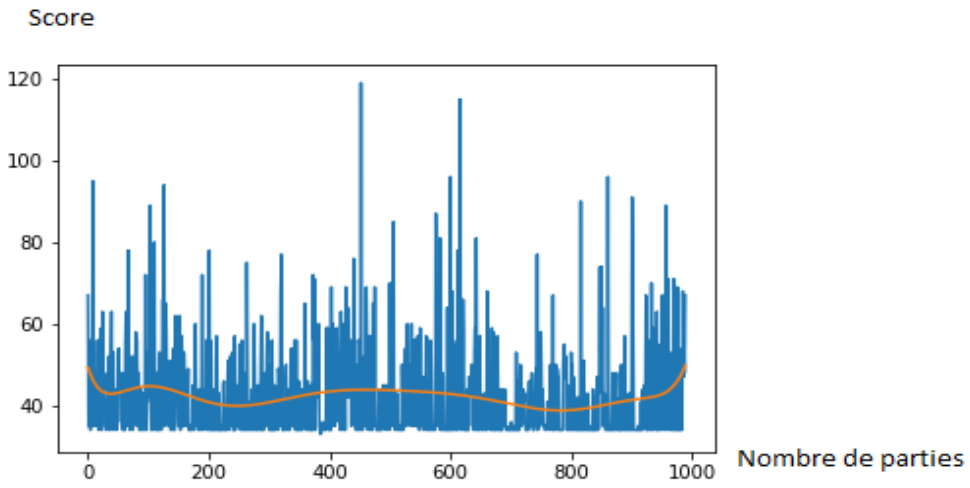
Toujours entre chaque partie, nous diminuons la part d'exploration grâce à *l'epsilon_decay*.

Enfin, nous réalisons des "checkpoints" après un certain nombre de parties, afin de sauvegarder les poids du modèle ainsi que les scores et les expériences précédentes. Cela

nous permet, de même que pour le MountainCar, de pouvoir relancer les entraînements même après une interruption ou de se prémunir d'un problème de l'algorithme. Finalement, nous traçons encore l'évolution des scores en fonction du nombre de parties, toujours pour rendre compte de la tendance de l'apprentissage.

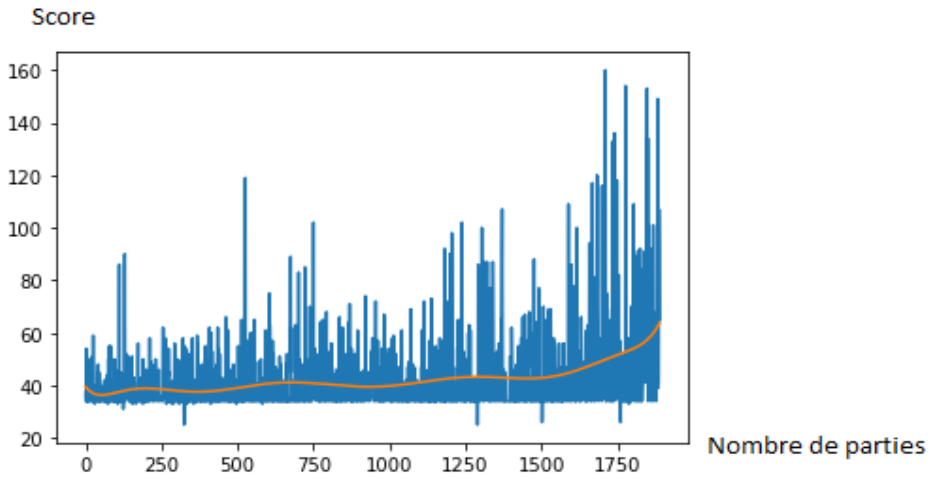
Voici les courbes des résultats obtenus :

(1) Après 1000 entraînements :



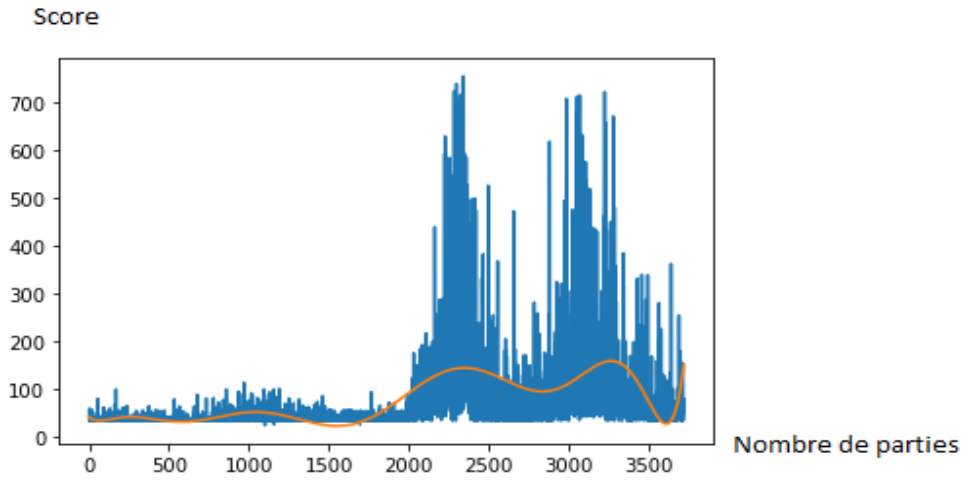
Nous remarquons que nous commençons à avoir une tendance croissante de l'apprentissage bien que nous ayons beaucoup de volatilité.

(2) Après 2000 entraînements :



Même conclusion que sur le graphique précédent, nous voyons plus clairement l'évolution du score moyen par partie de la part de l'agent grâce à la régression présente sur le graphe (ici en orange).

(3) Après environ 5000 parties :



Ici, nous remarquons que nous avons eu deux “pics” de score. Malgré les scores importants, nous constatons une grande volatilité des résultats, provenant très certainement d’un manque de temps d’apprentissage au modèle.

Remarque :

Vous trouverez ci-dessous, deux liens pour accéder à deux vidéos intitulées *T_Rex_NoTrain* et *T_Rex_Train*. La première représente le modèle non entraîné, tandis que la seconde représente le modèle entraîné après 4000 parties.

T_Rex_NoTrain : https://drive.google.com/open?id=1MymKppdnftitQ5uDEcb4VYq5pgjrQyBy
T_Rex_Train : https://drive.google.com/open?id=10484ejpbROIK7jyTtYFEWfVp55OKrXMc

Conclusion

Ce projet nous a permis de nous plonger entièrement dans le Deep Reinforcement Learning, discipline émergente de la fin de la décennie. Ce domaine dispose d'un panel important de difficultés et de sujets divers, pouvant être traités par des débutants comme des experts.

Nous avons ainsi pu découvrir ce sujet grâce au MountainCar et l'approfondir avec le "T-Rex Run!". Pour ce dernier, nous avons pu appréhender certaines difficultés comme le manque de puissance de calcul de nos machines pour certains algorithmes, ou encore le temps d'apprentissage pouvant aller jusqu'à plusieurs jours pour réussir à observer des résultats. Effectivement, l'algorithme dont nous nous sommes inspiré entraînait le réseau après chaque action, ce qui nécessite des capacités de calcul assez importantes, et de ce fait qui ne pouvait être utilisé sur toutes les machines. Par conséquent, pour des raisons de puissance, nous avons donc dû choisir l'apprentissage à la fin de chaque partie.

Enfin, il faut avoir à l'esprit que le Reinforcement Learning a de beaux jours devant lui. Il sera en effet de plus en plus accessible et performant dans les prochaines années, notamment grâce à la puissance quantique en développement et prometteuse de performances de calcul inégalées jusqu'à aujourd'hui.

Webographie

(1) Informations générales :
https://www.lemonde.fr/pixels/article/2015/07/24/comment-le-deep-learning-revolutionne-l-intelligence-artificielle_4695929_4408996.html

(2) Informations sur les DQN :
https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4

(3) Informations sur l'équation de Bellman :
https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/

(4) Informations sur les DQN :
https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/?fbclid=IwAR2QIHuxhpFB47_Ws_g3qG7kSAdBKlvhG3uxslHpcDW0baQvfU9sYs1TvCE

(5) Informations sur les DQN :
https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2

(6) Inspiration de l'algorithme du "T-Rex, Run !" :
https://medium.com/acing-ai/how-i-build-an-ai-to-play-dino-run-e37f37bdf153