Projet réseau de neuronnes : Vaincre Malo Guichard au snake

VALLS Alexis, CONSTANZA Corentin

14 mai 2024

1 Introduction

Notre projet vise à implémenter un réseau de neurones capable d'apprendre à jouer au jeu Snake. L'objectif ultime étant de créer un agent intelligent qui pourrait rivaliser avec succès contre un adversaire humain, en l'occurrence l'un des tous meilleurs joueurs que la Terre ai porté : MALO GUICHARD (voir figure 1).

Nous avons choisit la variante de base du jeu Snake ; le serpent meurt s'il se mord la queue ou s'il touche un bord, il grandit d'une case quand il mange une pomme, et il meurt lorsqu'il vit trop longtemps sans manger de pomme (cette dernière règle sert à éviter que l'agent "tourne en rond" indéfiniment). Notons également que nous avons choisit arbitrairement de prendre une grille de 20 par 20, de placer le serpent au milieu à gauche du plateau et de faire apparaître les pommes aléatoirement dès le début du jeu.

Ce choix du jeu Snake présente plusieurs intérêts particuliers pour l'entraînement d'un réseau de neurones. Tout d'abord, il présente une complexité suffisante pour tester les capacités d'un réseau de neurones. Il requiert une prise de décision en temps réel, une planification stratégique et une coordination précise des mouvements pour atteindre l'objectif de collecte de nourriture tout en évitant les obstacles. De plus, le jeu Snake a une interface relativement simple, ce qui facilite son inplémentation et la visualisation des résultats. Enfin, ce choix nous force à utiliser de l'apprentissage non supervisé car il n'existe pas de jeu de données associant chaque état du jeu à la bonne décision à prendre.

Dans la suite de ce rapport, nous allons aborder les méthodes que nous avons choisit d'utiliser et leurs aspects techniques. Puis dans une autre partie, nous traiterons de la mise en pratique et des résultats obtenus.

FIGURE 1 – Malo Guichard : joueur professionnel de Snake et homme à vaincre

2 Méthodes d'apprentissage

Nous avons choisi pour ce projet d'utiliser deux algorithmes évolutionnistes ainsi qu'un algorithme ayant une approche basé sur le renforcement. La principale différence entre ces deux méthodes réside dans leur approche fondamentale pour résoudre un problème.

Les algorithmes d'apprentissage par renforcement se basent sur une fonction coût (dont les variables sont les poids et biais d'un réseau de neurones) que l'on cherche à minimiser par descente de gradient. Là où les algorithmes évolutionnistes évaluent les résultats de plusieurs réseaux de neuronnes aux caractéristiques différentes et les mettent en compétition pour maximiser une fonction "fitness". Ces algorithmes évolutifs sont inspirés de la théorie de l'évolution.

Nous avons choisit de nous focaliser dans un premier temps sur les algorithmes évolutifs car, nous pensions que l'implémentation serait plus simple et demanderait moins de temps de calcul. Cependant, nous avons également essayé d'implémenter une méthode de renforcement par la suite.

2.1 Algorithme génétique

Le fonctionnement général d'un algorithme génétique peut être divisé en plusieurs étapes clés :

Initialisation de la population : Une population initiale d'individus est créée, généralement de manière aléatoire. Chaque individu est représenté par un ensemble de gènes qui encode les caractéristiques de la solution.

Évaluation de la population : Chaque individu de la population est évalué en fonction d'une fonction d'évaluation prédéfinie, qui mesure sa performance par rapport au problème à résoudre. Cette fonction peut être basée sur des critères de performance tels que la fitness, le score ou le coût.

Sélection des parents : Les individus de la population sont sélectionnés pour devenir les parents de la génération suivante. La sélection est généralement basée sur leur fitness relative, favorisant les individus les mieux adaptés pour la reproduction. Différentes

méthodes de sélection peuvent être utilisées, telles que la roulette wheel selection ou le tournoi de sélection.

Reproduction : Les individus sélectionnés sont utilisés pour créer une nouvelle génération d'individus. Cela implique des opérations génétiques telles que le croisement (crossover) et la mutation. Le croisement combine les gènes des parents pour créer de nouveaux individus, tandis que la mutation introduit une variation aléatoire dans les gènes.

Remplacement : La nouvelle génération remplace l'ancienne population, créant ainsi une nouvelle population pour la prochaine itération de l'algorithme.

Répétition des étapes 2 à 5 : Les étapes d'évaluation, de sélection, de reproduction et de remplacement sont répétées pour un certain nombre d'itérations ou jusqu'à ce qu'un critère d'arrêt prédéfini soit atteint. Ce critère d'arrêt peut être un nombre maximum d'itérations, une fitness seuil ou une convergence des solutions.

2.2 NEAT (NeuroEvolution of Augmenting Topologies)

L'algorithme NEAT est aussi un algorithme évolutionniste. Il comporte 5 étapes similaires à l'algorithme génétique (voir 2.1). La principale différence intervient lors de la 4ème étape : la reproduction. En effet, en plus de changer les poids et biais des individus d'une population à l'autre, l'algorithme NEAT peut changer la structure des réseaux de neurones : comme pour les probabilités de mutation des poids, il y a une probabilité d'apparition ou de suppression d'un neurone dans une couche intermédiaire.

Comparé à l'algorithme génétique vu précédement, NEAT présente plusieurs avantages. Le fait que NEAT puisse faire évoluer la topologie du réseau de neurones implique qu'il peut optimiser le nombre de neurones, les connexions entre les neurones et l'architecture globale du réseau. Cette approche permet à NEAT de trouver des architectures de réseau plus efficaces et adaptées à la tâche spécifique. Cela permet à NEAT d'explorer un espace de recherche plus large et de trouver des solutions potentiellement meilleures.

Enfin ces architectures sont généralement plus explicites dans leurs fonctionnement de part leurs nombres plus faible de neuronnes et de connexions, il est donc plus simple d'interpréter le fonctionnement du réseau de neurone (voir figures 2 et 3).

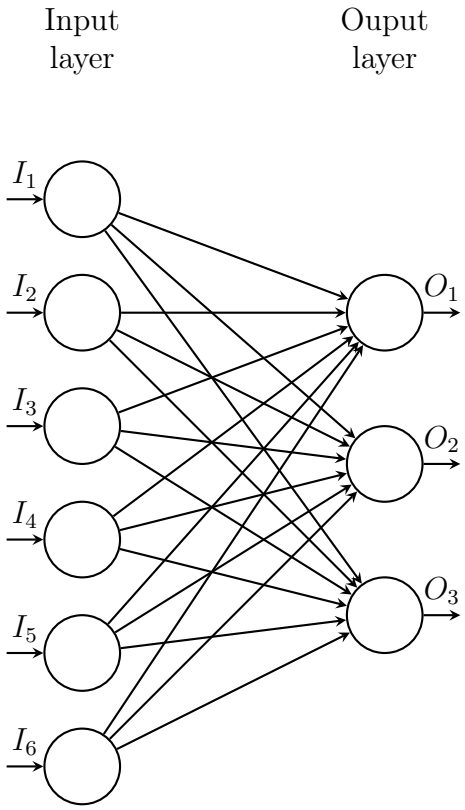


FIGURE 2 – Exemple d'une architechture simple donné au NEAT avant l'entrainement

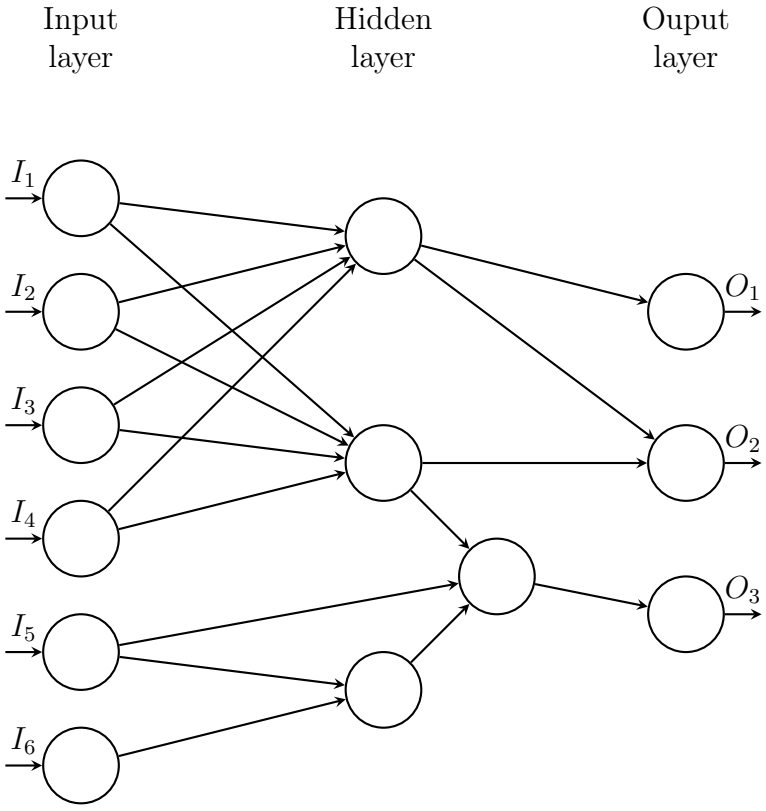


FIGURE 3 – Exemple d'une architechture "simple" renvoyer par le NEAT après entrainement

2.3 Apprentissage par renforcement (Deep Q Learning)

L'apprentissage par renforcement est une approche du Machine Learning où un agent apprend à prendre des décisions en interagissant avec un environnement. L'idée principal est de récompenser ou des sanctionner l'agent en fonction de ses actions. L'objectif de l'apprentissage par renforcement est donc de maximiser les récompenses cumulatives sur le long terme.

Le Deep Q Learning est une approche qui étend l'apprentissage par renforcement en utilisant des reseaux de neuronnes profonds. Ce réseau de neurones profond, souvent appelé réseau Q, est utilisé pour approximer la fonction Q, qui est une estimation de la valeur de chaque état-action dans un environnement donné. La fonction Q attribue une valeur à chaque paire état-action, représentant la somme des récompenses futures attendues lorsqu'une action donnée est prise à un état donné.

Le processus de d'entrainement du réseau Q se déroule en plusieurs étapes :

Initialisation : On initialise les poids de notre reseau. Il est à noter que l'on utilise 2 réseau ayant la même architecture : le réseau cible et le réseau principal. Tout les N itérations, les poids du reseau pricipal sont copier dans le réseaux cible.

Sélection de l'action : L'agent choisit une action à prendre en fonction d'une politique d'exploration-exploitation. Au début de l'apprentissage, l'agent explore davantage l'environnement pour collecter plus de données d'apprentissage, puis il commence à exploiter les connaissances acquises pour prendre des décisions.

Prise d'expérience : L'agent effectue l'action, collecte des expériences sous la forme de transitions (état, action, récompense, nouvel état). Ces transitions sont stockées dans une mémoire à long terme (replay memory)..

Evaluation de la recompense : On évalue la recompence de l'action. En utilisant la mémoire à long terme, on effectue ces évaluations aussi sur de petit batch d'action.

Mise à jour du réseau et entrainement : On mets à jour le réseau main selon l'équation de Bellman et on l'entraine à minimier la fonction coût.

Répétition des étapes 2 à 5

Interéssons nous à l'équation de Belleman :

$$NewQ(E, A) = Q(E, A) + \alpha[R(E, A) + \gamma maxQ'(E', A') - Q(E, A)]$$

détaillons chacun de ces termes :

Q : valeur actuelle de Q

E : un état du systeme donnée (Input du réseau de neurone)

A : l'action prise dans cet état

α : Taux d'apprentissage

R : la recompense (positive ou négative) donner à l'agent après avoir choisi l'action A dans l'état E.

γ : Discount Rate

maxQ' : la recompense futur attendu maximum étant données le nouvel état E' et toute les possibilitées A' possible dans ce nouvel état.

Notre fonction coût est $(NewQ - Q)^2$. Il s'agit de l'erreur quadratique moyenne (MSE).

3 Mise en pratique et résultats

3.1 Pygad : Python Genetic Algorithm Library

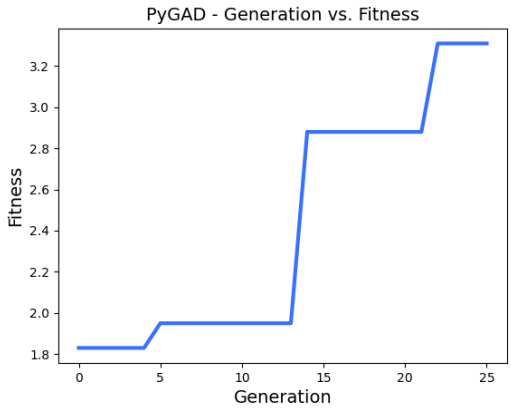
Pour l'algorithme génétique, nous avons utilisé le package PyGAD (Python Genetic Algorithm Library). L'utilisation du package PyGAD facilite la mise en œuvre des étapes énoncées au point 2.1. Il fournit des fonctions prêtes à l'emploi pour l'initialisation de la population, l'évaluation des individus, la sélection des parents, le croisement, la mutation et la génération de la nouvelle population. De plus, PyGAD offre des fonctionnalités avancées telles que la possibilité de personnaliser les paramètres de l'algorithme génétique, d'appliquer des contraintes spécifiques et de surveiller la convergence de l'algorithme.

Nous avons utilisé cet algorithme génétique pour faire évoluer plusieurs structures de réseau de neurones évaluées sur plusieurs fonctions fitness différentes.

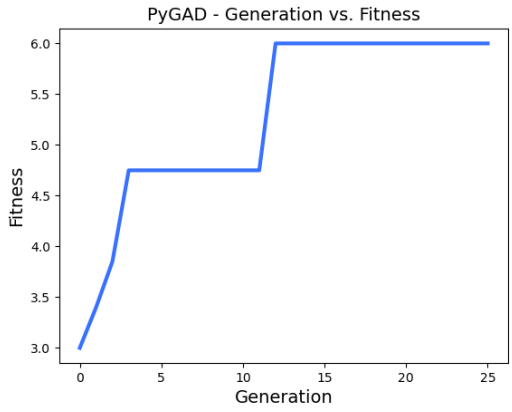
— La première structure comporte 28 inputs, deux couches de neurones intermédiaires de 14 et 7 neurones respectivement, et 4 outputs (un pour chaque direction). Les 28 inputs correspondent à la ditance qui sépare la tête des murs, de la queue et de la pomme dans 8 directions (3×8 inputs) et à la direction du serpent (4 inputs one-hot encoded).

— La deuxième structure comporte les mêmes inputs et outputs mais a seulement une couche intermédiaire qui comporte 32 inputs.

Pour ces deux structures, nous avons fait évoluer 25 générations de 200 individus (les temps de calcul étaient trop longs pour faire plus). Comme le montre la figure 4, les scores obtenus sont assez peu satisfaisants. Nous n'avons donc pas persévéré avec cet algorithme, et nous sommes passé à la méthode suivante.



(a) Fonction fitness de la première structure en fonction des générations



(b) Fonction fitness de la seconde structure en fonction des générations

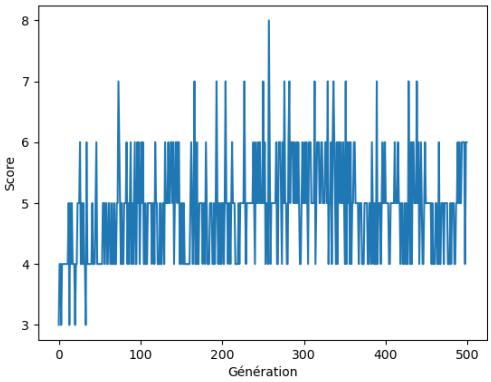
FIGURE 4 – Evolution de la fonction fitness pour les algorithmes génétiques

3.2 Neat-python

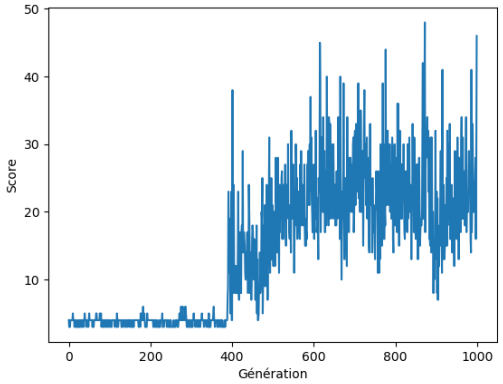
Pour cet algorithme, nous avons utilisé le package NEAT-python. Afin de simplifier l'apprentissage, nous avons modifiée la structure du réseau de neuronne. Pour cette partie, il prend seulement 9 inputs : la distance entre la tête et l'obstacle dans 7 directions, et le vecteur tête-pomme. En sortie, il n'a que 3 outputs : tourner à droite, tourner à gauche, et aller tout droit. Comme expliqué au point 2.2, la structure au début de l'algorithme ne possède pas de couche intermédiaire, celles-ci seront éventuellement créées durant l'apprentissage.

Nous avons lancé l'algorithme avec deux fonctions fitness différentes. L'une relativement élaborée, tenant compte entre autre du nombre de cases parcourues par le serpent entre chaque pomme et de sa tendance à se rapprocher de la pomme à chaque étape. L'autre étant la fitness la plus triviale, à savoir le nombre de pommes collectées par l'agent.

De façon assez surprenante, on observe que la fonction fitness "élaborée" ne permet pas d'obtenir des résultats intéressants, là où la fonction triviale permet une convergence rapide de la population vers des scores relativement bons (voir figure 5)



(a) Evolution du meilleur score de chaque génération pour la fonction fitness élaborée



(b) Evolution du meilleur score de chaque génération pour la fonction fitness simple

FIGURE 5 – Comparaison de la convergence de l'algorithme NEAT pour deux fitness différents

3.3 Deep Q Learning

Pour la mise en place de cet algorithme, nous avons utiliser uniquement le package pytoch et numpy. Cette implementation est basé sur le programme Snake de Patrick Loeber, disponible sur le git-hub suivant : https ://github.com/patrickloeber/python-fun/tree/master/snake-pygame.

Voici comment nous definisson notre espace d'état (input) : $E = (D, Dir, P)^T$ avec :

Le vecteur de "danger immédiat" : $D = [D_{devant}, D_{droite}, D_{gauche}]^T$

Le vecteur direction (one-hot encoded) : $D = [Dir_{gauche}, Dir_{droite}, Dir_{haut}, Dir_{bas},]^T$

Le vecteur pomme : $D = [P_{gauche}, P_{droite}, P_{haut}, P_{bas},]^T$ qui nous informe de la direction relative de la pomme par rapport à la tête du serpent (haut dessus à gauche par exemple).

Voici un exemple d'état :

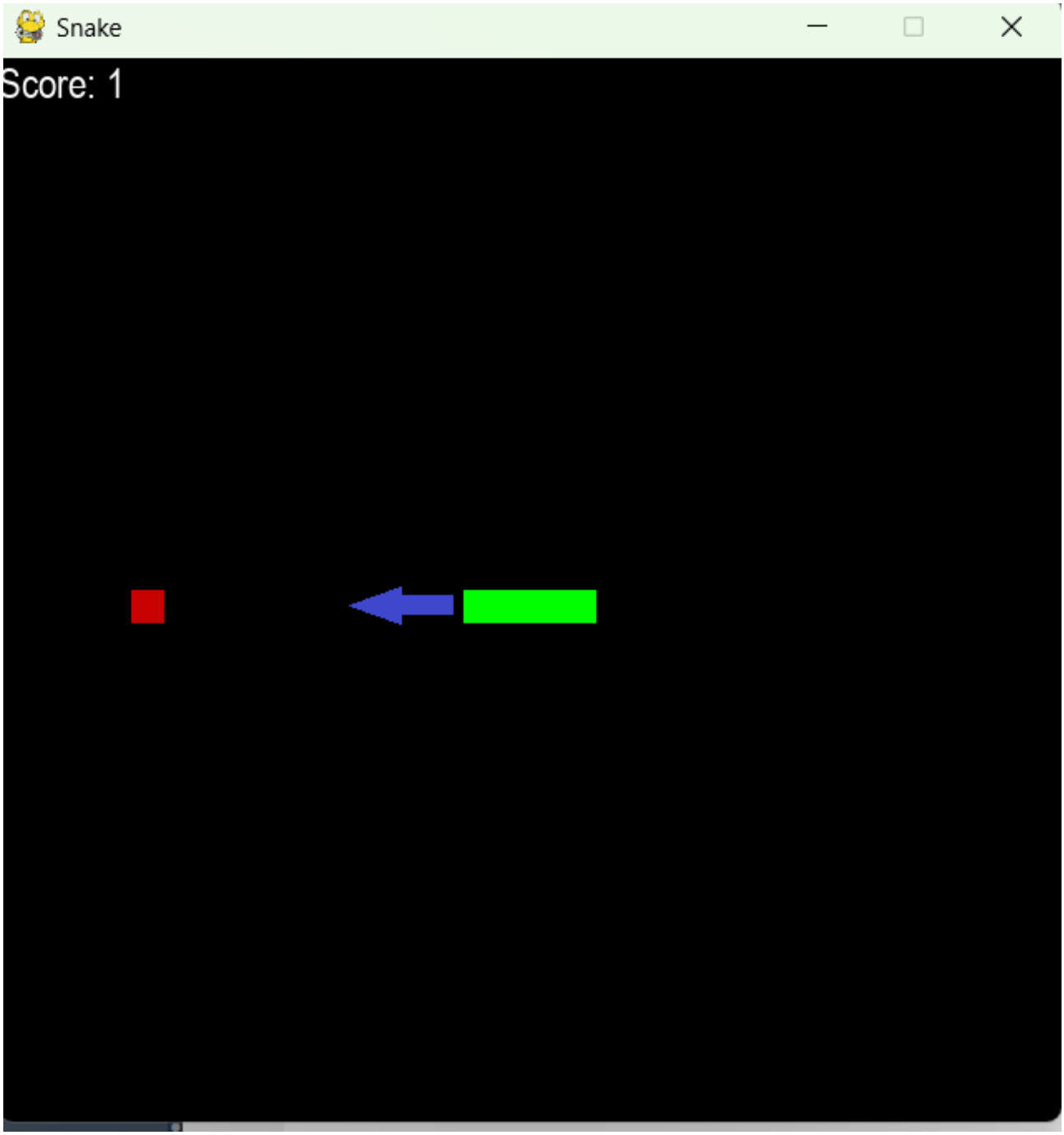


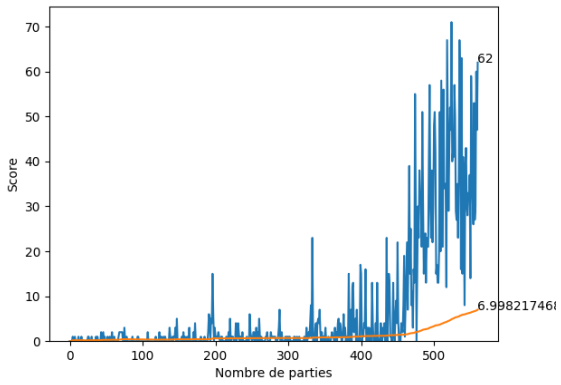
FIGURE 6 – Exemple d'un état quelconque

Dans cet exemple, l'état est $E = [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]^T$

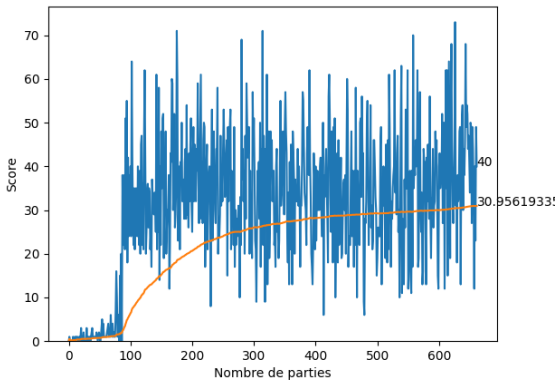
Concernant les outputs, nous en avions 3 : aller tout droit, tourner à droite et tourner à gauche.

Nous avons essayer différentes architecture afin d'en voir l'impact sur le score et le score moyen en fonction du nombre d'itération.

Nous allons commencer par des reseaux avec une seul couche caché, on va simplement augmenter le nombre de neuronnes dans celle-ci :



(a) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,64,3)

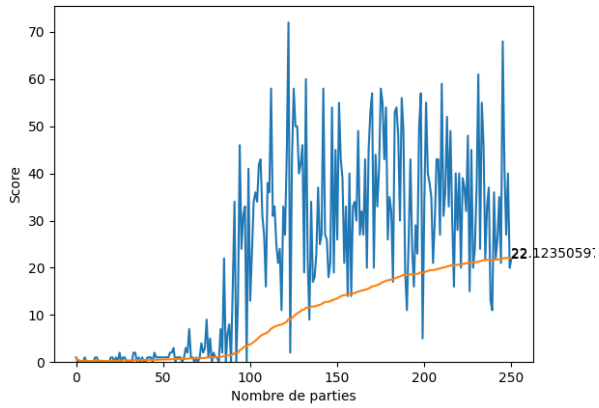


(b) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,256,3)

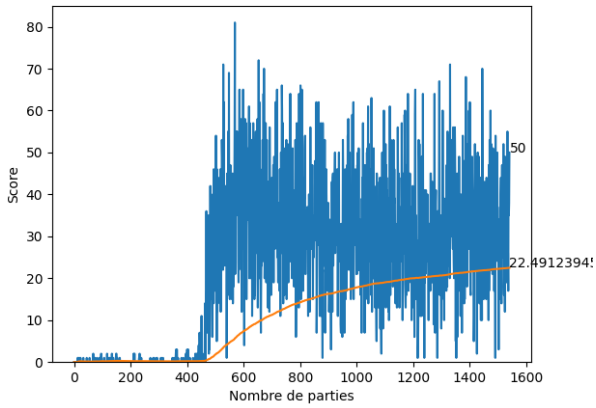
FIGURE 7 – Impact de l'ajout de neurone dans la couche caché

On peut voir que plus il y a de neurones, plus on converge vite vers des résultats étant autour de 35 en moyenne.

Nous allons maintenant augmenter le nombre de couche afin de tester l'impact de la complexité :



(a) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,64,64,3)

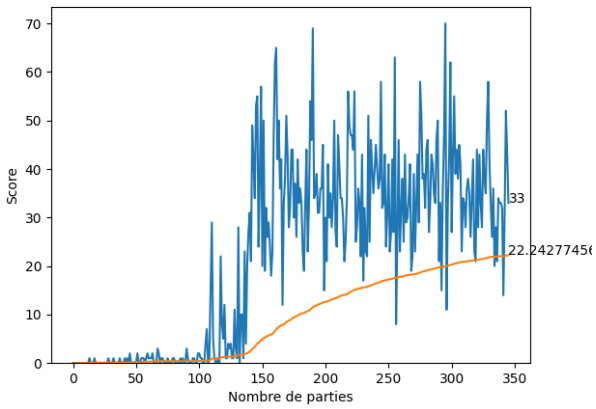


(b) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,256,256,128,3)

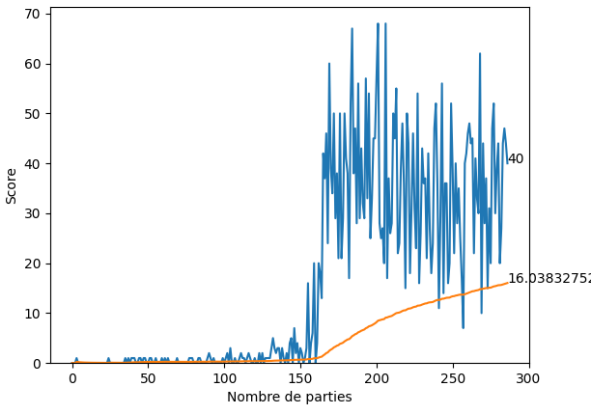
FIGURE 8 – Impact de l'ajout de couche

On voit qu'un reseau trop complexe n'est pas une bonne chose. En effet, notre espace d'état étant très simple, (seulement 11 valeur binaire), il est très facile à expliquer et un reseau trop complexe n'est pas forcement plus performant qu'un reseau plus simple et prends beaucoup plus de temps à entrainer.

Voici aussi un test en modifiant ϵ sur le modèle (11,128,3). On le mets plus grand afin de "laisser une plus grande place à l'exploration".



(a) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,128,3) avec $\epsilon_1 = 80$



(b) Evolution du score et du score moyen en fonction du nombre d'itération pour une architecture de type (11,128,3) avec $\epsilon_2 = 2 * \epsilon_1$

FIGURE 9 – Impact de la politique d'exploration/exploitation

On peut voir que ce n'est pas très concluent on semble seulement avoir ralentie l'apprentissage du modèle, on stagne toujours autours du même score (35).

En fin de compte il semble que la principale limite de notre modèle ne soit pas l'architecture mais la définitions de notre espace d'état.Nous avons reussi à avoir des comportement intéressant malgrés un espace d'état très simple. L'une des perspective serait donc de complexifier ce dernier. De plus nous pourrions pour aller plus loins nous pourrions faire differents essaie pour voir l'impact des differents Hypers Paramètres (taux d'apprentissage, discount rate, taille des Batch).

4 Conclusion

Ce projet aura été l'occasion de decouvrir plusieurs méthodes d'apprentissage et de s'initier à l'implémentation de réseau de neuronne avec Pytorch. Comme dit précédement Nous avons obtenu des resultats relativement concluant cependant Malo reste ... **inbattable**.