

Rapport de Stage

Comparaison de méthodes d'apprentissage automatique

par Mathieu PONT

dans le cadre de la

Licence Informatique 3^{ème} année à l'Université Paul Sabatier de Toulouse

effectué à

L'Institut de Recherche en Informatique de Toulouse (IRIT)

du

9 avril au 6 juillet 2018

Tuteurs de l'organisme d'accueil : M. Frederic MIGEON et M. Jerome MENGIN

Tuteur universitaire : M. Franck SILVESTRE

Table des matières

Introduction.....	4
Présentation du sujet.....	5
Travaux effectués.....	6
2.1 Représentation d'un état et des récompenses pour le jeu de Mario.....	6
2.2 Etat de l'art.....	7
2.2.1 Deep Q Learning.....	7
2.2.1.1 Processus de décision markovien.....	7
2.2.1.2 Fonction Q.....	8
2.2.1.3 Principe général des réseaux de neurones.....	9
2.2.1.4 Approximer la fonction Q avec un réseau de neurones.....	11
2.2.1.5 Améliorations du Deep Q Learning.....	13
2.2.2 NEAT.....	15
2.2.2.1 Principe des algorithmes génétiques.....	15
2.2.2.2 Evoluer l'architecture d'un réseau de neurones avec les algorithmes génétiques ..	16
2.2.3 NEAT + Q.....	18
2.2.4 AMOEBA.....	19
2.3 Mise en œuvre des systèmes.....	21
Résultats.....	22
Discussion.....	28
Bibliographie.....	34
Annexes.....	36

Introduction

Afin de clôturer ma formation dans le cadre de la Licence Informatique à l'Université Paul Sabatier de Toulouse, j'ai effectué un stage de 3 mois du 9 avril au 6 juillet 2018 à l'Institut de Recherche en Informatique de Toulouse (IRIT).

Durant ce stage je devais comparer différentes méthodes d'apprentissage automatique en évaluant les performances d'un système multi-agents (déjà construit) et celles de réseaux de neurones (que je devais moi même mettre en œuvre).

Ce document retrace de façon synthétique toutes les activités que j'ai réalisées durant mon stage. Un dépôt github associé à ces travaux a été créé¹.

Je présenterai d'abord le sujet qui m'a été confié, je parlerai ensuite des travaux que j'ai effectués en présentant l'état de l'art, la mise en œuvre des systèmes et les résultats obtenus. Je terminerai enfin par un bilan sur les résultats obtenus.

J'aimerais remercier Frédéric MIGEON, Jérôme MENGIN, Bruno DATO et Franck SILVESTRE pour m'avoir accompagné durant ce stage, ainsi que toute l'équipe dans laquelle j'ai été durant ces 3 mois pour leur accueil chaleureux.

Je voudrais particulièrement remercier Frédéric MIGEON pour m'avoir permis de faire ce stage durant lequel j'ai appris énormément et où j'ai pu mettre en œuvre des notions qui m'intéressent grandement.

¹ <https://github.com/MatPont/MarioBros-MachineLearning>

Présentation du sujet

L'apprentissage automatique est un champ d'étude de l'intelligence artificielle permettant à un système « d'apprendre » dans le sens où il peut améliorer de lui-même ses performances sur une tâche spécifique.

En 1959, Arthur Samuel définit ce terme par le « champ d'étude qui donne aux ordinateurs la capacité d'apprendre sans être explicitement programmés »¹. En effet, dans la programmation « classique », nous devons définir de manière claire ce que le système doit faire pour chaque situation qu'il pourrait rencontrer. Avec l'apprentissage automatique, ce n'est plus le cas, c'est le système qui va trouver ce qu'il doit faire de lui même (d'où le terme « apprendre »).

Le système multi-agents auto-adaptatif AMOEBA a été développé dans le cadre de la thèse de Julien Nigon [Nigon 2017] pour réaliser de l'apprentissage automatique.

L'objectif du stage est de réaliser une étude comparative d'AMOEBA avec d'autres systèmes d'apprentissage dans le cadre de *benchmarks* tels que la plateforme MarioAI [Karakovskiy et al. 2012]. Je devais donc trouver des méthodes d'apprentissage performantes dans ce contexte, les mettre en œuvre et ensuite les comparer avec AMOEBA sur le jeu de Mario.

Il existe différentes catégories d'apprentissage, celle qui nous intéressera ici sera l'apprentissage par renforcement qui consiste à dire au système ce qu'il fait de bien ou de mal à l'aide de récompenses.

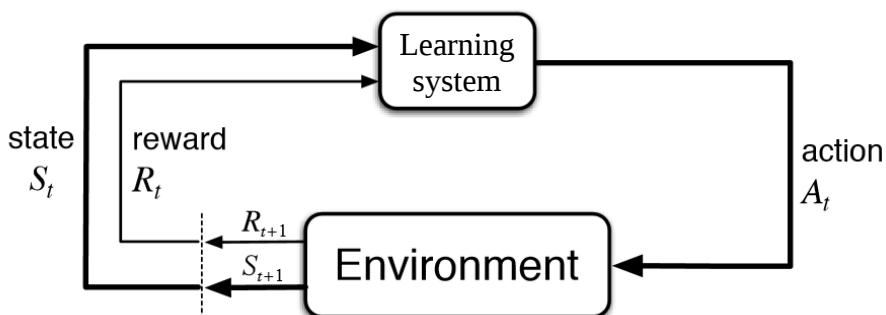


Figure 1.1 – Représentation d'un système apprenant et de son environnement dans le cadre d'un apprentissage par renforcement

Un système apprenant, se trouvant dans un environnement, effectue une action et observe l'effet de cette dernière sur l'environnement. Le système arrive dans un nouvel état (*state*) et peut recevoir une récompense (*reward*) dûe à l'action réalisée. L'objectif du système est de maximiser les récompenses reçues en choisissant les bonnes actions. En introduisant la notion de récompense, on définit ici la notion d'apprentissage par un problème d'optimisation de la récompense et de recherche des actions qui permettent cette optimisation.

¹ <https://toiledefond.net/le-machine-learning-quand-les-ordinateurs-predisent-lavenir/>

Travaux effectués

Nous devons d'abord nous demander, dans le cas du jeu de Mario, comment nous pouvons représenter un état qui correspond à ce que « voit » le système, ce sont les informations qu'il reçoit de l'environnement et à partir desquelles il fera des choix. Puis, comment représenter les récompenses qui permettront au système de savoir les actions qu'il doit privilégier et celles qu'il doit éviter dans un état donné.

2.1 Représentation d'un état et des récompenses pour le jeu de Mario

Nous avons plusieurs manières de représenter un état pour le jeu de Mario. L'une d'elle est de travailler directement sur l'image brute en pixels. Une autre consiste à utiliser une grille (fournie par le *benchmark* Mario, généralement de taille 19x19) qui décrit l'environnement autour du personnage comme montré en *figure 2.1*. Chaque case contient une valeur pouvant représenter un obstacle, un ennemi, une pièce etc.

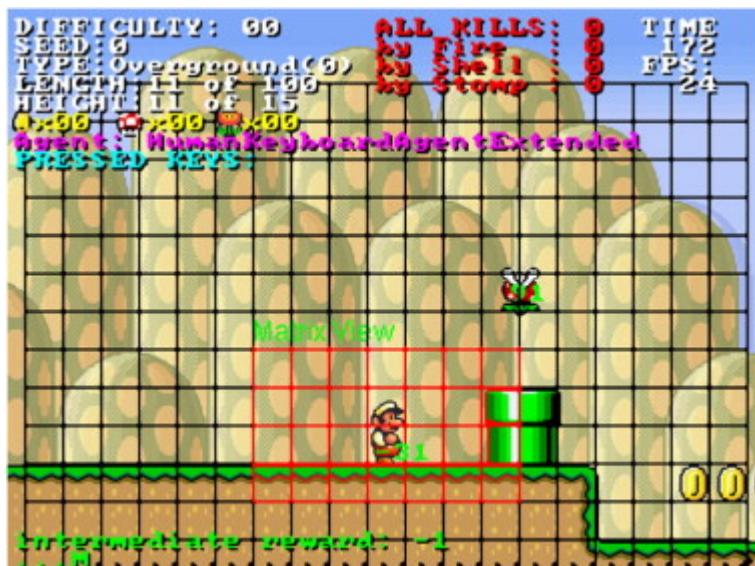


Figure 2.1 – Grille d'observation de Mario

Lors des précédentes expérimentations avec AMOEBA une représentation plus simplifiée a été choisie consistant en 4 paramètres : la vitesse de Mario sur l'axe X et Y et la distance au premier obstacle à droite et en bas du personnage.

On peut utiliser comme récompense les points rapportés par les actions de Mario, cette valeur augmentera par exemple lorsqu'il trouve des pièces, des objets, finit le niveau et elle diminuera lorsqu'il rentre en collision avec des ennemis, perd le niveau etc.

Lors des précédentes expérimentations avec AMOEBA, une autre fonction de récompense a été choisie : Mario est récompensé de manière positive lorsqu'il avance vers la droite (la fin du niveau

se situant à droite) et vers le haut (pour l'encourager à franchir des obstacles) et de manière négative lorsque c'est vers la gauche et le bas.

Dans un premier temps, nous avons choisi de comparer comme représentation d'état celle utilisant la grille et celle avec les 4 paramètres. En terme de fonction de récompense nous avons choisi de comparer celle calculée sur le score obtenu par Mario et celle utilisée par AMOEBA. Ce qui nous fait en tout 4 combinaisons de représentation d'état/récompense. Dans un second temps, nous pourrions aussi ajouter la combinaison des deux représentations d'états et des deux fonctions récompenses ce qui nous ferait en tout 9 combinaisons état/récompense.

2.2 Etat de l'art

J'ai commencé par faire un état de l'art sur les méthodes d'apprentissage par renforcement pertinentes dans notre contexte et trois principales en sont ressorties.

La première est le Deep Q Learning¹ [Mnih et al. 2013]. C'est un réseau de neurones ayant pour objectif d'approximer une fonction nommée Q [Watkins 1989]. Cette dernière permet d'évaluer la qualité d'une action pour un état donné. Ainsi, le système peut choisir l'action qui a la plus grande qualité dans chaque état (c'est à dire l'action qui lui semble la meilleure dans telle situation).

La deuxième est NEAT² (NeuroEvolution of Augmenting Topologies) [Stanley et al. 2002]. C'est un réseau de neurones utilisant les algorithmes génétiques pour faire croître son architecture en fonction du problème traité.

La dernière méthode est un mélange des deux précédentes. Elle exploite le fonctionnement de NEAT pour trouver l'architecture adéquate et utilise le Q Learning avec la rétropropagation pour apprendre [Whiteson et al. 2006].

2.2.1 Deep Q Learning

2.2.1.1 Processus de décision markovien

Pour modéliser de manière formelle les états, récompenses et actions comme présentés dans la figure 1.1 on peut utiliser les processus de décision markovien (*Markov Decision Process* ou MDP). On peut les définir par un quadruplet $\langle S, A, T, R \rangle$ définissant :

- S : l'ensemble des états.
- A : l'ensemble des actions.
- T : une fonction de transition $T : S \times A \times S \rightarrow [0 ; 1]$ ou $T(s, a, s')$ qui définit la probabilité d'arriver dans l'état s' en effectuant l'action a dans l'état s . Dans un environnement déterministe on aura $T : S \times A \rightarrow S$.

1 <https://ai.intel.com/demystifying-deep-reinforcement-learning/>

2 <http://gekkoquant.com/2016/03/13/evolving-neural-networks-through-augmenting-topologies-part-1-of-4/> (et les parties 2, 3 et 4)

- R : une fonction de récompense $R : S \times A \times S \times \mathbb{R} \rightarrow [0 ; 1]$ ou $R(s, a, s', r)$ qui définit la probabilité d'obtenir la récompense r après être passé de l'état s à s' en ayant effectué l'action a . Avec des récompenses déterministe on aura $R : S \times A \times S \rightarrow \mathbb{R}$.

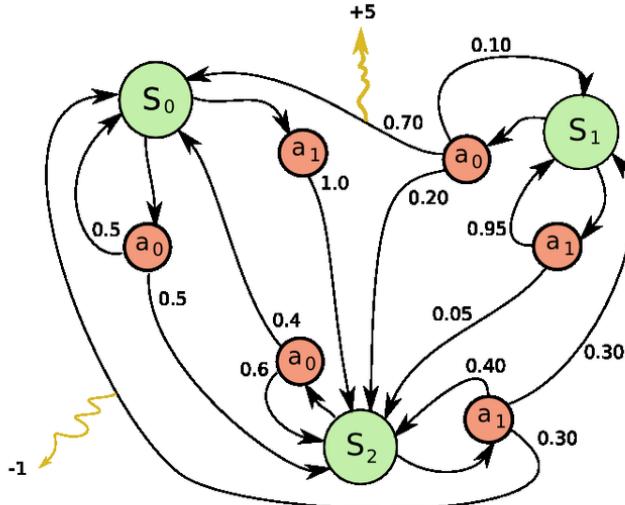


Figure 2.2 – Exemple de processus de décision markovien

Un graphe orienté peut représenter les MDPs comme en figure 2.2. Certains sommets sont des états avec des arcs vers d'autres sommets représentant les actions possibles à partir de cet état. À partir de ces sommets représentant une action on trouve des arcs vers d'autres sommet-états avec la probabilité d'arriver dans celui-ci après avoir effectué cette action et la potentielle récompense obtenue.

2.2.1.2 Fonction Q

La fonction Q évalue la qualité d'une action dans un état donné, cette qualité représente « la récompense cumulée possible à la fin du jeu après avoir fait telle action dans tel état ».

En effet, pour avoir un agent performant sur le long-terme nous ne devons pas prendre en compte uniquement la récompense immédiate mais aussi les futures récompenses.

Dans un MDP un « épisode » (ou une partie par exemple) forme une séquence finie d'état, action, récompense :

$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots, s_{n-1}, a_{n-1}, r_n, s_n$ avec s_n : état terminal (fin de la partie).

La récompense totale d'un épisode peut se noter :

$$R = r_1 + r_2 + \dots + r_n$$

La future récompense totale à partir d'un instant t se note :

$$\begin{aligned} R_t &= r_t + r_{t+1} + \dots + r_n \\ &= r_t + R_{t+1} \end{aligned}$$

On ajoute un facteur appelé coefficient d'actualisation (*discount factor*) qui permet de pondérer la récompense future pour lui donner plus ou moins d'importance, ce qui nous donne :

$$R_t = r_t + \gamma R_{t+1}$$

Une bonne stratégie pour l'agent serait de toujours prendre l'action qui maximise cette récompense future.

On définit donc dans un premier temps la fonction Q comme étant la récompense future maximale possible en réalisant telle action dans tel état :

$$Q(s_t, a_t) = \max R_{t+1} \quad (1)$$

Pour choisir une action dans un état donné il suffit de prendre l'action avec la valeur Q la plus haute, on peut ainsi définir la politique du choix de l'action pour un état donné :

$$\pi(s) = \operatorname{argmax}_a Q(s, a)$$

Il est à noter qu'il existe des méthodes essayant directement d'approximer cette fonction, ce sont les *Policy-based methods*. Contrairement à ici où on passe par une fonction intermédiaire (la fonction Q) pour définir cette politique.

On cherche donc à définir la fonction Q de telle sorte que l'on peut puisse la calculer pour un état et une action donnés, contrairement à (1). Pour cela on utilise l'équation de Bellman :

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

Ainsi on définit la future récompense maximale possible comme étant la récompense immédiate plus la future récompense maximale possible à l'état suivant.

2.2.1.3 Principe général des réseaux de neurones

Un neurone artificiel est inspiré biologiquement des neurones de notre cerveau qui, pour résumer, prennent en entrées des impulsions électriques et si ces dernières dépassent un certain seuil alors le neurone envoie une impulsion électrique en sortie.

Un neurone artificiel est un « automate doté d'une fonction de transfert qui transforme ses entrées en sortie selon des règles précises »¹. En entrée on a plusieurs variables x_0, x_1, \dots, x_n qui sont chacune pondérée par un poids $w_{k0}, w_{k1}, \dots, w_{kn}$. Le neurone possède une fonction d'agrégation qui effectue un travail sur l'entrée pondérée et dont le résultat est donné en entrée à une fonction d'activation qui va jouer le rôle de seuil.

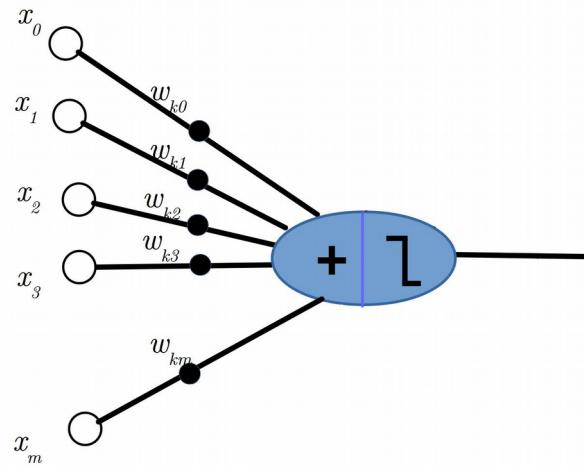


Figure 2.3 – Neurone artificiel

Généralement la fonction d’agrégation correspond à la somme des entrées pondérées. Il n’y a pas de consensus pour la fonction d’activation même si les plus utilisées aujourd’hui sont *sigmoid*, *ReLU* (et ses variantes) ou encore *TanH*.

Lorsque l’on connecte plusieurs neurones artificiels entre eux on peut former un réseau. Aujourd’hui ils sont généralement structurés en couche.

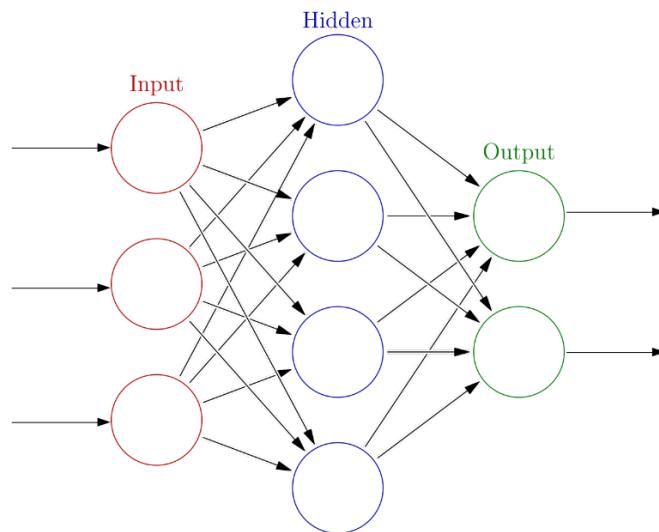


Figure 2.4 – Exemple de réseau de neurones

Il y a la couche d’entrée qui va nous permettre de rentrer les données que l’ont veut faire analyser au système et la couche de sortie qui nous dira la prédition qui a été faite par le système à partir de ces données.

Un réseau de neurones peut être vu comme un approximateur de fonction et les couches cachées (celles entre la couche d'entrée et celle de sortie) permettent de complexifier cette fonction (en rajoutant des couches, des neurones etc.).

L'apprentissage consiste ici à trouver de bonnes valeurs pour les connexions entre les neurones qui permettront au système de faire le moins d'erreur possible dans ses prédictions. L'erreur peut généralement être vue comme la différence entre la prédition faite par le système et celle qu'il aurait du vraiment faire.

Le réseau peut utiliser la rétropagation (*backpropagation*) pour apprendre. Cette méthode permet de modifier des valeurs de connexions entre les neurones afin de minimiser son erreur en utilisant la descente de gradient. Pour être utilisée nous devons connaître la véritable prédition que le système aurait dû faire pour ce jeu de données.

2.2.1.4 Approximer la fonction Q avec un réseau de neurones

La version originale du Deep Q Learning proposée par Deep Mind utilise un réseau de neurones à convolution¹, leur objectif était de travailler directement sur l'image brute en pixels d'un jeu. Avec cette façon le réseau de neurones est plus polyvalent dans le sens où avec les mêmes hyper-paramètres leur architecture peut être utilisé sur différents jeux. De plus elle a surpassée un joueur humain professionnel sur 7 jeux différents dans leur premier papier [Mnih et al. 2013] et sur une trentaine dans le second [Mnih et al. 2015].

Si nous ne voulons pas travailler sur une image nous pouvons très bien utiliser un réseau de neurones classique pour approximer la fonction Q.

L'idée ici est de modifier comment la fonction Q sera calculée, au lieu de donner l'état et l'action en entrée et d'avoir la qualité de cette action en sortie on va uniquement donner l'état en

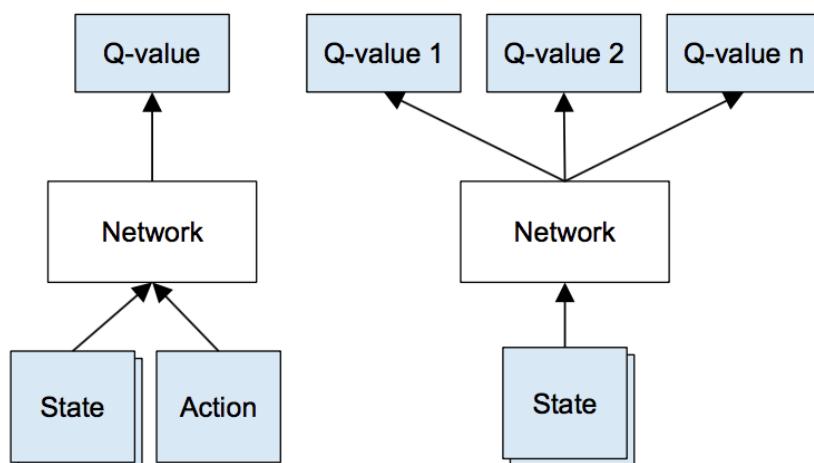


Figure 2.5 – Gauche : architecture naïve d'un Deep Q Network ; Droite : architecture optimisée d'un Deep Q Network utilisée dans les papiers de Deep Mind

¹ <http://cs231n.github.io/convolutional-networks/>

entrée et avoir en sortie la qualité de chaque action. Cela permet de faire une seule traversée du réseau pour avoir la qualité de chaque action.

Pour entraîner le réseau de neurones nous allons utiliser la rétropropagation en essayant d'optimiser l'erreur quadratique de la différence entre la prédiction cible et la prédiction faite.

$$L = \frac{1}{2} \left[\underbrace{r + \gamma \max_{a'} Q(s', a')}_{target} - \underbrace{Q(s, a)}_{prediction} \right]^2$$

Pour apprendre à partir d'une transition $< s, a, r, s' >$ (passer de l'état s à s' en faisant l'action a et en obtenant la récompense r) :

- On fait une première traversée du réseau avec l'état s pour évaluer la prédiction faite $Q(s, a)$, c'est à dire la qualité de l'action réalisée.
- On fait une autre traversée du réseau avec l'état s' pour calculer $\max_{a'} Q(s', a')$.
- On peut calculer la prédiction cible $target = r + \gamma \max_{a'} Q(s', a')$.
- On a maintenant tous les termes nécessaires pour faire la rétropropagation. L'erreur de toutes les actions autre que a est mise à 0.

Exploration/Exploitation

Si nous utilisons le Q Learning tel qu'il vient d'être expliqué le système va toujours exploiter ses connaissances pour trouver l'action à faire. L'un des problèmes est qu'au début de l'entraînement le système n'a aucune connaissance et qu'il a besoin d'explorer, c'est à dire tenter des actions pour en voir le résultat au lieu de suivre sa politique.

Il existe plusieurs méthodes pour gérer l'exploration, la plus connue est « *ϵ -greedy exploration* » qui fait réaliser au système une action aléatoire avec une probabilité ϵ . La technique proposée par DeepMind est de diminuer la valeur ϵ au fur et à mesure. Ainsi, le système commencera par beaucoup explorer l'environnement, pour apprendre, et au fur et à mesure il utilisera de plus en plus ses connaissances pour choisir l'action.

Il existe d'autres méthodes pour l'exploration¹ mais je n'ai pas eu le temps de correctement les étudier.

Mémoire d'expériences

Durant le jeu chaque transition d'un état à un autre est enregistré dans une mémoire sous la forme d'un tuple $< s, a, r, s', t >$ où s est l'état courant, s' l'état dans lequel on arrive après avoir fait l'action a , r la récompense obtenue et t un booleen disant si s' est un état terminal ou non.

L'entraînement se fait par *batch* et non directement sur l'état que l'on vient de traiter. Après avoir fait une action, on enregistre le tuple représentant cette expérience dans la mémoire puis on y en prend, selon une politique, un certain nombre et on entraîne le réseau avec celles-ci.

Si t indique que s' est terminal alors nous ne faisons pas la deuxième traversée du réseau pour calculer $\max_a Q(s', a')$, on prend uniquement $target = r$.

La politique de choix d'expériences la plus simple est de prendre de manière aléatoire dans la mémoire. Nous verrons plus tard une manière plus performante qui choisie les expériences selon leur importance.

2.2.1.5 Améliorations du Deep Q Learning

Dans cette partie je résume les différentes améliorations qui ont été apportées au Deep Q Learning. Néanmoins je ne rentrerai pas dans tous les détails et survolerai certains points. Pour plus d'informations sur l'une de ces améliorations je vous conseille de lire le papier de recherche associé.

Target Network – [Mnih et al. 2015]

Cette méthode fait utiliser un deuxième réseau afin de calculer les valeurs de Q cibles (*target*) utilisées pour calculer l'erreur. N'utiliser qu'un seul réseau peut causer des problèmes d'approximation et de stabilité, en effet à chaque étape d'entraînement les valeurs des paramètres du réseau change, si nous utilisons à chaque fois des jeux de valeurs changeant constamment l'estimation peut devenir incontrôlable (dans le sens où les valeurs pourraient changer trop drastiquement menant à de mauvaises estimations).

Pour éviter ce problème on utilise un réseau cible (*target network*) dont les paramètres sont fixes et mis à jour périodiquement (en copiant les paramètres du réseau principal). Le fait que les paramètres du réseau cible soit fixes pendant n étapes (hyper-paramètre à ajuster) fait gagner en stabilité.

Double Q Learning – [Hasselt et al. 2015]

L'intuition principale derrière le Double Q Learning est que le Deep Q Learning original surestime souvent les valeurs cibles de Q à cause de l'opération *max*. L'idée ici est donc d'utiliser le réseau principal pour choisir l'action et le réseau cible pour calculer la valeur de Q à partir de cette action.

On modifie donc le calcul de la cible par :

$$target = r + \gamma Q(s', \text{argmax}_a Q(s', a, \theta), \theta')$$

Ici, le troisième paramètre θ de la fonction Q fait référence au réseau utilisé pour calculer la fonction Q. Avec θ pour le réseau principal et θ' pour le réseau cible.

Recurrent Network – [Hausknecht et al. 2015]

Ici on remplace une couche « normale » par une couche récurrente (généralement celle juste après les couches de convolution). Cette dernière permet de donner au système la notion de dépendance temporelle entre les données qu'il traite. Ainsi, le traitement d'un état est dépendant des états traités précédemment. Cela permet au réseau d'apprendre (ou de tirer des informations) sur une séquence d'état au lieu d'uniquement sur un seul à la fois.

On doit légèrement modifier comment fonctionne la mémoire d'expériences. Au lieu de prendre expérience par expérience on va prendre des séquences d'expériences dont la taille et le nombre sont des hyper-paramètres à ajuster.

Lors de l'apprentissage, la politique du choix des expériences nous en donne un certain nombre à traiter. Pour chaque expérience nous devons former une séquence et il existe plusieurs manières de la former. Pour chaque expérience nous pouvons former la séquence en mettant l'expérience en début, en fin ou en milieu de séquence.

De plus dans leur papier il est conseillé de masquer (mettre à 0) la première moitié des erreurs de chaque séquence d'expériences lors de l'apprentissage.

On utilise généralement une couche LSTM¹ comme couche récurrente.

Prioritized Experience Replay – [Schaul et al. 2016]

L'idée ici est de donner plus ou moins d'importance à certaines expériences. Ainsi au lieu de prendre de manière aléatoire dans la mémoire d'expériences on prendra celles qui ont le plus d'importance en respectant une loi de distribution de probabilité afin de ne pas prendre toujours les mêmes.

Cela se fait généralement par l'implantation d'un *SumTree* qui permet de stocker d'une manière relativement optimisée les expériences en fonction de leur importance et de réduire la complexité lors de la recherche des expériences importantes.

L'importance d'une expérience est définie par l'erreur calculée à partir de cette expérience. Plus l'erreur est grande plus l'expérience est importante. En effet, si l'erreur liée à une expérience est grande c'est qu'elle a beaucoup à apprendre au réseau.

Dueling Network – [Wang et al. 2016]

L'architecture dueling propose de décomposer la fonction Q en deux termes. Le premier est la fonction de valeur V(s) qui indique à quel point il est bon d'être dans un état donné. Le second est la fonction d'avantage A(s, a) qui nous dit à quel point prendre une certaine action dans cet état serait mieux comparé aux autres. On peut maintenant voir Q comme la somme de V et de A :

$$Q(s, a) = V(s) + A(s, a)$$

En réalité cette équation est simplifiée et pose un problème lié au fait que pour un Q donné nous ne pouvons retrouver V et A, ils ont donc légèrement modifié l'équation pour résoudre ce problème, l'explication détaillée et la nouvelle équation est disponible dans le papier de recherche partie « 3. *The Dueling Network Architecture* ».

L'objectif de cette architecture est de calculer séparément la fonction de valeur et d'avantage et de les combiner ensuite pour faire la fonction Q sur la dernière couche du réseau comme en *figure 2.6*.

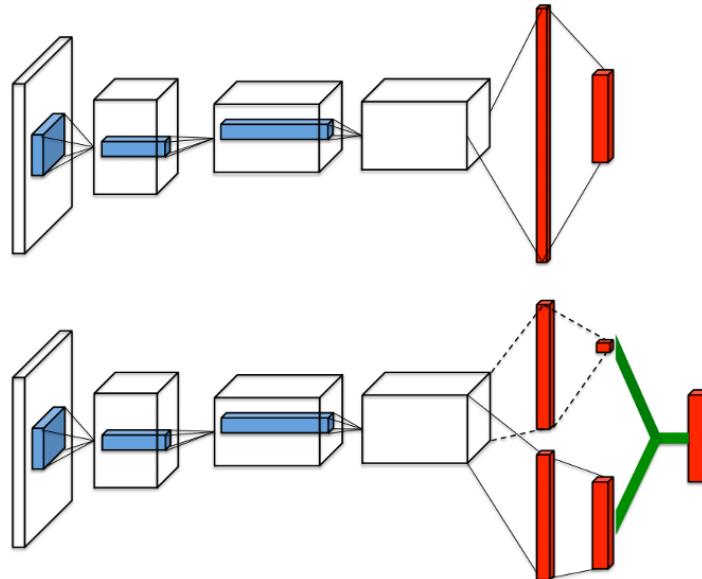


Figure 2.6 – Haut : architecture régulière du DQN ; Bas : Dueling DQN

Asynchronous Advantage Actor-Critic (A3C) – [Mnih et al. 2016]

Cette méthode est l'une des dernières avancées de Deep Mind sur le Deep Q Learning et selon les résultats elle semble extrêmement performante. Je précise que c'est sûrement la méthode que j'ai le moins étudiée car je n'ai pas pu la mettre en œuvre mais elle est ressortie dans mon état de l'art et cela me semble pertinent d'en parler.

L'idée principale ici est d'avoir plusieurs agents au lieu d'un seul. Il y a un réseau de neurones global et chaque agent travaille sur celui-ci de son côté avec son propre environnement. Comme l'expérience de chaque agent avec son environnement est différente des autres et qu'ils évoluent de manière parallèle on peut facilement voir l'amélioration qu'apporte cette technique.

2.2.2 NEAT

2.2.2.1 Principe des algorithmes génétiques

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes dont le principe s'inspire de la théorie de l'évolution pour résoudre des problèmes. Ils utilisent la notion de

sélection naturelle et l'applique à une population de solutions potentielles au problème donné afin de lui trouver une bonne solution.

Le principe est le suivant, nous avons une population constituée de différents individus pouvant résoudre le problème, un individu est représenté par un génome qui définit ses caractéristiques. Chaque individu est testé sur le problème afin d'évaluer sa *fitness* qui représente à quel point il est performant. Il y a ensuite trois phases : sélection, croisement et mutations qui vont permettre de faire évoluer les individus en créant de nouvelles générations.

Pour la phase de sélection on garde les individus avec une chance d'autant plus grande que leur *fitness* l'est. Ainsi, on a plus de chance de garder les individus performants mais nous n'excluons pas les individus qui le sont moins car leur descendance pourrait offrir des possibilités intéressantes.

Dans la phase de croisement on sélectionne de manière aléatoire deux individus pour en créer un nouveau, l'idée est de combiner les génomes pour en créer un autre.

Pour finir, dans la phase de mutation on modifie de manière aléatoire le génome de certains individus (choisis de manière aléatoire).

On répète ces étapes jusqu'à arriver à une certaine valeur de *fitness* ou au bout d'un certain nombre de générations.

2.2.2.2 Evoluer l'architecture d'un réseau de neurones avec les algorithmes génétiques

Avec la méthode NEAT le génome d'un individu correspond à l'architecture d'un réseau de neurones. Pour utiliser les algorithmes génétiques nous devons définir comment représenter le génome d'un individu de façon à pouvoir faire un croisement entre deux génomes et enfin quelles sont les mutations pouvant être intéressantes. La *fitness* sera ici évaluer à l'aide de la fonction de récompense choisie.

Représentation du génome

Un réseau de neurones est constitué de neurones et de connexions entre eux. La façon la plus simple de représenter le génome est d'avoir une liste de neurones (chacun associé à son type, comme entrée, sortie ou caché), et la liste des connexions entre deux neurones (chacune associée à son poids et aux deux neurones qu'elle lie).

Chaque gène du génome correspond donc à un neurone ou une connexion entre deux neurones.

Genome (Genotype)					
Node Genes	Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden
Connect. Genes	In 1 Out 4 Weight 0.7 Enabled Innov 1	In 2 Out 4 Weight -0.5 DISABLED Innov 2	In 3 Out 4 Weight 0.5 Enabled Innov 3	In 2 Out 5 Weight 0.2 Enabled Innov 4	In 5 Out 4 Weight 0.4 Enabled Innov 5
	In 1 Out 5 Weight 0.6 Enabled Innov 6	In 4 Out 5 Weight 0.6 Enabled Innov 11			

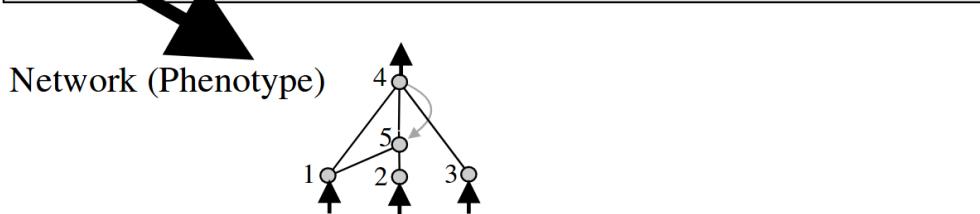


Figure 2.7 – Exemple de génome et le réseau de neurones correspondant

Croisement

Pour arriver à mélanger deux génomes pour en créer un nouveau il faut trouver un moyen d'identifier quel gènes sont identiques (ou se ressemblent) parmi les deux parents pour pouvoir en choisir un des deux. Il est proposé dans le papier original de NEAT d'associer à chaque nouveau gène une valeur nommée « nombre global d'innovation » (*global innovation number*) qui est incrémenté à chaque fois qu'un nouveau gène apparaît. Ainsi, si pour chaque parent on a un gène avec le même nombre global d'innovation on choisira alors le gène du parent avec la plus grande *fitness*.

Pour des individus différents si deux gènes ont le même nombre global d'innovation c'est qu'ils représentent une mutation structurelle identique ou presque (même connexion entre deux mêmes neurones mais pas forcément le même poids par exemple).

Parent1	1 1->4	2 2->4 DISAB	3 3->4	4 2->5	5 5->4	disjoint		8 1->5		
	1 1->4	2 2->4 DISAB	3 3->4	4 2->5	5 5->4 DISAB					
Parent2	1 1->4	2 2->4 DISAB	3 3->4	4 2->5	5 5->4 DISAB	6 5->6	7 6->4	disjoint disjoint excess excess		
	1 1->4	2 2->4 DISAB	3 3->4	4 2->5	5 5->4 DISAB	6 5->6	7 6->4			
Offspring	1 1->4	2 2->4 DISAB	3 3->4	4 2->5	5 5->4 DISAB	6 5->6	7 6->4	8 1->5	9 3->5	10 1->6

Figure 2.8 – Exemple de croisement

Mutations

De manière aléatoire on réalise des mutations dans les individus de la population actuelle. Dans le papier original on trouve deux mutations, l'ajout d'une connexion entre deux neurones et l'ajout d'un neurone sur une connexion.

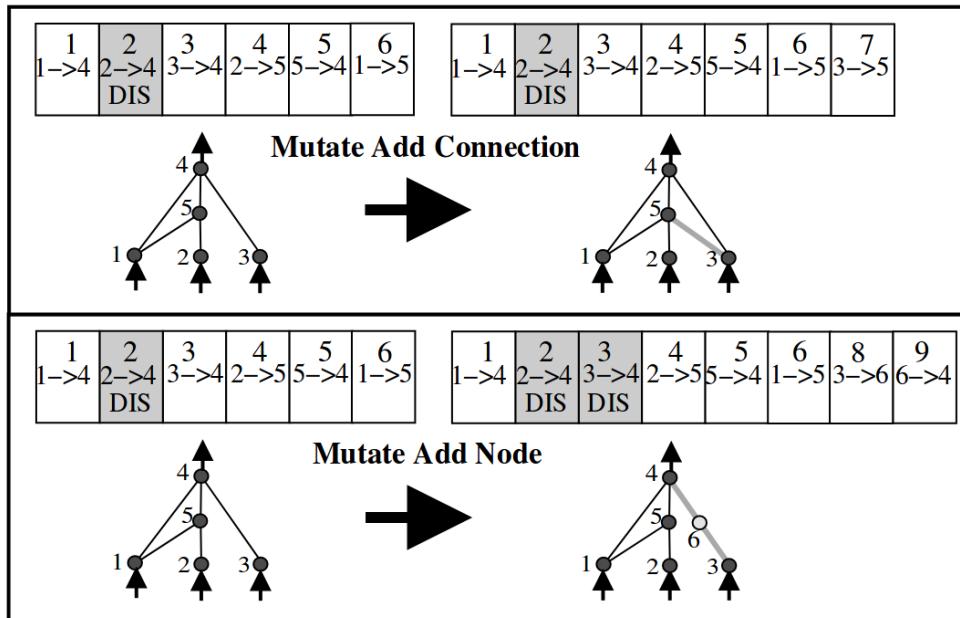


Figure 2.9 – Exemples de mutation

On peut aussi avoir une mutation modifiant la valeur du poids d'une connexion ou encore l'activation ou la désactivation d'un neurone ou d'une connexion.

Spéciation

La spéciation consiste à prendre les génomes d'une génération et de les diviser en des groupes distincts que l'on nommera espèces. Cela permet à un individu d'être en compétition avec d'autres qui lui ressemblent au lieu de l'être avec la population en entière. En effet certains individus peuvent avoir besoins de plusieurs mutations avant d'être performant et d'offrir des possibilités intéressantes par la suite. Sans la spéciation ces individus ne seraient pas gardés et on perdrat donc l'innovation qu'ils pourraient apporter.

2.2.3 NEAT + Q

Pour cette méthode il suffit de se dire que le réseau de neurones généré par NEAT essaye d'approximer la fonction Q et que les valeurs en sorties ne sont plus des valeurs arbitraires mais la qualité de chaque action. Ainsi on peut utiliser la rétropropagation pour changer la valeur des connexions entre les neurones comme dans le Deep Q Learning.

Contrairement à NEAT où le réseau d'un individu reste fixe durant l'évaluation de sa *fitness*, ici, la valeur des connexions entre les neurones variera grâce au Q Learning.

Lors du croisement nous avons deux manières différentes de choisir le génome des parents. Il y a l'approche Lamarckienne qui dit de prendre le génome comme il est à la fin de l'entraînement de l'individu (c'est à dire avec les modifications apportées par le Q Learning). Ensuite il y a l'approche Darwinienne qui dit de prendre le génome tel qu'il était avant l'entraînement (sans les modifications du Q Learning).

L'avantage de l'approche Lamarckienne est plutôt évident, cela permet à chaque génération de ne pas devoir répéter l'apprentissage des précédentes générations. Cependant, l'approche Darwinienne peut être avantageuse car elle permet à chaque génération de se baser sur un génome dont la base a menée à un succès plutôt que de s'appuyer sur un génome altérée dont l'évolution future pourrait être limitée.

On pourrait se dire que le Q Learning ne sert pas vraiment dans l'approche Darwinienne car les changements apportés ne sont pas pris en compte directement par la descendance d'un individu. En fait ils sont pris en compte indirectement grâce à l'effet Baldwin [Baldwin 1896]. Pour plus d'informations je conseille au lecteur de se renseigner auprès du papier original de NEAT + Q [Whiteson et al. 2006].

2.2.4 AMOEBA

Le système multi-agents AMOEBA est constitué de différents types d'agents :

- Les agents percepts qui observent l'environnement et s'occupent chacun d'une dimension de l'espace d'entrée. Ils récupèrent les données et les transfèrent aux agents contextes.
- Les agents contextes vont générer les prédictions à partir des données reçues par les agents percepts. Ils peuvent s'adapter pour faire de meilleures prédictions. C'est eux qui vont approximer la fonction recherchée. Chaque agent contexte s'occupe d'une portion de l'espace d'entrée et ils vont tous approximer la fonction par morceau. La portion dont s'occupe un agent contexte est délimité par les plages de validité qui sont des bornes minimum et maximum sur chaque dimension de l'espace d'entrée. La prédition est réalisée à l'aide du modèle local qui permet de « transformer » la situation traitée en prédition.
- L'agent head va évaluer les prédictions faites par les agents contextes. Connaissant la prédition qui aurait dû être faite il va pouvoir notifier les agents contextes de la véracité de leurs prédictions.

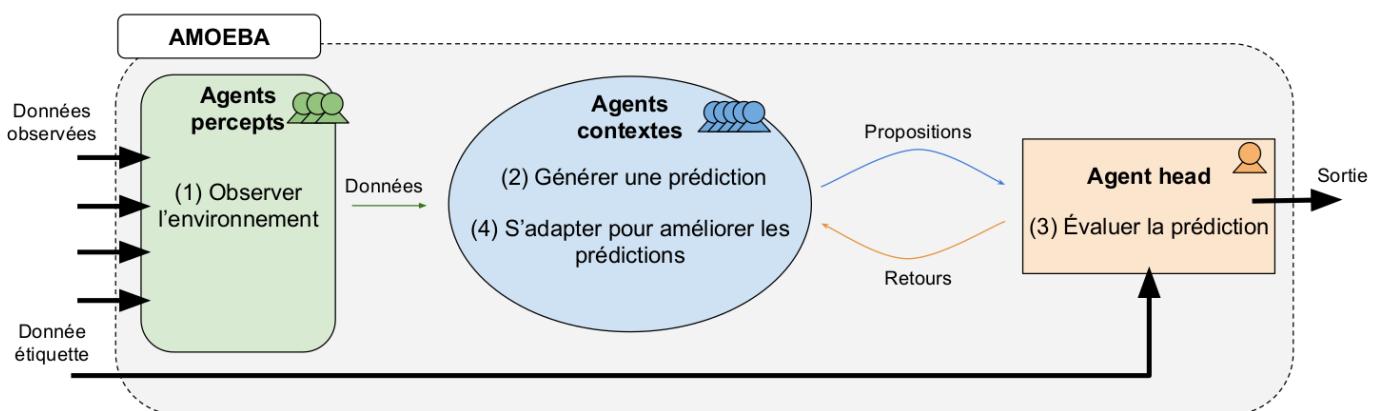


Figure 2.10 – Fonctionnement général de AMOEBA

L'apprentissage dans AMOEBA passe par la résolution de situations de non-coopération (SNC) :

- SNC 1 : incompétence de l'agent head (pas de proposition)

L'agent head doit recevoir une prédiction des agents contexte à chaque étape d'exécution d'AMOEBA. Si ce n'est pas le cas il ne peut pas lui-même fournir une prédiction en sortie du système. Pour ce faire il va interroger l'agent contexte le plus « proche » de la situation courante afin d'avoir une prédiction.

Pour éviter que ce problème se présente à nouveau il faut qu'un agent contexte soit capable de fournir une proposition dans une situation similaire. Soit on modifie les plages de validité d'un agent contexte soit on en créer un nouveau.

- SNC 2 : conflit d'un agent contexte (prévisions fausses)

L'agent head doit fournir une prédiction en sortie suffisamment proche (défini par un seuil d'erreur) de la donnée étiquetée. Au dessus de ce seuil la prédiction est considérée comme fausse, la faute étant à l'agent contexte ayant fourni la prédiction.

Pour que ce problème ne se reproduise plus on modifie les plages de validité de l'agent contexte fautif afin qu'il ne fasse plus de proposition pour une situation similaire.

- SNC 3 : conflit d'un agent contexte (prévisions inexactes)

Cette SNC est presque identique à celle précédente, sauf qu'au lieu du seuil d'erreur on utilise ici le seuil d'inexactitude. Au dessus de celle-ci la prédiction est considérée comme inexacte.

Comme la prédiction est considérée comme inexacte ce n'est pas les plages de validité qu'il faut modifier mais le modèle local, afin de rendre la prévision moins inexacte.

- SNC 4 : incompétence de l'agent head (plusieurs propositions)

L'agent head ne doit recevoir qu'une seule proposition des agents contextes, si il en reçoit plusieurs il choisit celle venant de l'agent contexte ayant la valeur de confiance la plus élevée. Cette valeur représente la certitude de l'agent contexte quant à la véracité de sa proposition.

- SNC 5 : concurrence entre agents contextes

Si plusieurs agents contexte font une proposition exacte c'est qu'ils réalisent la même activité là où un seul suffirait. Tous les agents contextes autre que celui avec la valeur de confiance la plus élevée vont donc modifier leurs plages de validité pour ne plus faire de prédictions dans une situation similaire.

- SNC 6 : inutilité d'un agent contexte

Suite aux ajustements des plages de validité d'un agent contexte elles peuvent définir un intervalle nul ou négatif. Cet agent contexte n'est donc plus utile au système, il s'autodétruit.

2.3 Mise en œuvre des systèmes

J'ai ensuite pris un Deep Q Network déjà implanté pour le jeu de Mario que j'ai largement modifié pour l'adapter à nos besoins. Premièrement, il fallait lier le système au benchmark de Mario qu'utilisait AMOEBA. Pour ce faire, j'ai réalisé une interface Java-Python (le benchmark étant en Java et le réseau de neurones en Python).

Ensuite, il nous fallait utiliser une architecture différente pour les deux représentations d'état possibles, le détail des architectures choisies est disponible en *annexe 2.1*. Je me suis inspiré de différentes architectures pour réaliser les miennes que j'ai améliorées en essayant différents paramètres (nombre de couches, leur « type », nombre de neurones par couche etc.) et en les testant sur le jeu.

Pour le réseau de neurones à convolution j'ai commencé avec 3 couches de convolution comme le présentait Deep Mind [Mnih et al. 2015]. Etant donné que nous ne travaillons pas directement sur l'image en pixels mais sur une grille dont les dimensions sont bien plus petites (presque 20 fois) je me suis rendu compte qu'avec moins de couches de convolution et bien moins de filtres nous pouvons avoir de meilleurs résultats. Au final je n'utilise que deux couches de convolution avec un nombre de filtre extrêmement petit.

Pour le réseau de neurones classique je n'utilise qu'une seule couche cachée, après plusieurs expérimentations je me suis rendu compte qu'en rajouter faisait diminuer les performances. De même, y avoir plus de 256 neurones n'augmentait pas les performances (nous utilisons souvent des puissances de 2 dans le nombre de neurones car cela accélère les calculs sur les GPUs).

Quant à AMOEBA, j'ai rajouté la représentation d'état de la grille et la fonction de récompense utilisant le score du jeu. La version de AMOEBA utilisée pour Mario était une ancienne version, j'ai modifié l'algorithme du choix des actions pour utiliser la nouvelle version d'AMOEBA (en utilisant le *request*). J'ai été obligé de le faire car l'ancienne version posait des problèmes avec la représentation d'état utilisant la grille. En effet, avec l'ancienne version la plupart du temps l'algorithme ne trouvait pas de meilleur contexte (*bestcontext = null*) et le système prenait donc une action aléatoire. Il s'en suivait que la majorité des actions étaient en fait des actions aléatoires.

Lors de la comparaison des deux systèmes, il fallait bien faire attention qu'ils soient homogénéisés, c'est à dire qu'ils utilisent la même représentation d'état, la même fonction de récompense, les mêmes valeurs pour hyper-paramètres communs (comme le ratio exploration/exploitation, c'est à dire le pourcentage de chance de réaliser une action aléatoire) etc.

Résultats

Pour réaliser la plupart de mes expérimentations j'ai utilisé le cluster de calcul OSIRIM¹. J'ai du apprendre à l'utiliser afin d'exécuter à la fois les expériences de AMOEBA (utilisant des CPUs) et les expériences des réseaux de neurones (utilisant des GPUs).

Lors des expérimentations j'entraînais le système sur 300 parties en répétant les 20 premiers niveaux (sauf le 2 et le 8 car ils présentent des cul-de-sacs complexes à franchir). Le système passe au niveau suivant qu'il ait gagné ou non. Je réalisais 4 expérimentations identiques pour faire une moyenne.

Pour la suite nous ferons référence pour les représentations d'état :

- à **S0** pour celle avec la grille.
- à **S1** pour celle avec les 4 paramètres.

Nous ferons référence pour les fonctions de récompense :

- à **R0** pour celle liée au score.
- à **R1** pour celle récompensant le fait d'aller à droite et en haut.

Statistiques

		Deep Q Learning	AMOEBA
S0 – R0	Victoires (/300)	19,25 (25 ; 45 ; 3 ; 4)	* (0/121 ; 0/64)
	Récompense par partie (en moyenne)	229,25	-6,43
S0 – R1	Victoires (/300)	73,75 (55 ; 89 ; 65 ; 86)	* (5/18 ; 5/10 ; 4/18 ; 3/21)
	Récompense par partie (en moyenne)	18028,66	5571,49
S1 – R0	Victoires (/300)	83,5 (78 ; 73 ; 75 ; 108)	0 (0 ; 0 ; 0 ; 0)
	Récompense par partie (en moyenne)	891,89	-1,815
S1 – R1	Victoires (/300)	109 (117 ; 85 ; 114 ; 120)	68,25 (42 ; 82 ; 83 ; 66)
	Récompense par partie (en moyenne)	19104,65	3173,57

* voir plus bas, au point « AMOEBA » de la partie « Convergence des systèmes »

¹ Les expériences présentées dans ce rapport ont été (pour la plupart) réalisées en utilisant la plateforme OSIRIM qui est administrée par l'IRIT et soutenue par le CNRS, la région Midi-Pyrénées, le gouvernement français, et le FEDER (voir <http://osirim.irit.fr/site/fr>).

Convergence des systèmes

Nous réalisons des mesures propre à chaque système pour évaluer leur convergence. Pour le Deep Q Learning on utilisera la valeur de Q moyen sur chaque partie [Mnih et al 2013], cette valeur correspond donc à la valeur de Q en moyenne après chaque action pour la partie en cours. Pour AMOEBA nous utiliserons le nombre d'agents contexte et la criticité moyenne pour chaque partie (pareillement, la criticité moyenne correspond à la criticité que l'on a en moyenne après chaque action pour la partie en cours).

- **Deep Q Learning**

- **S0 – R0**

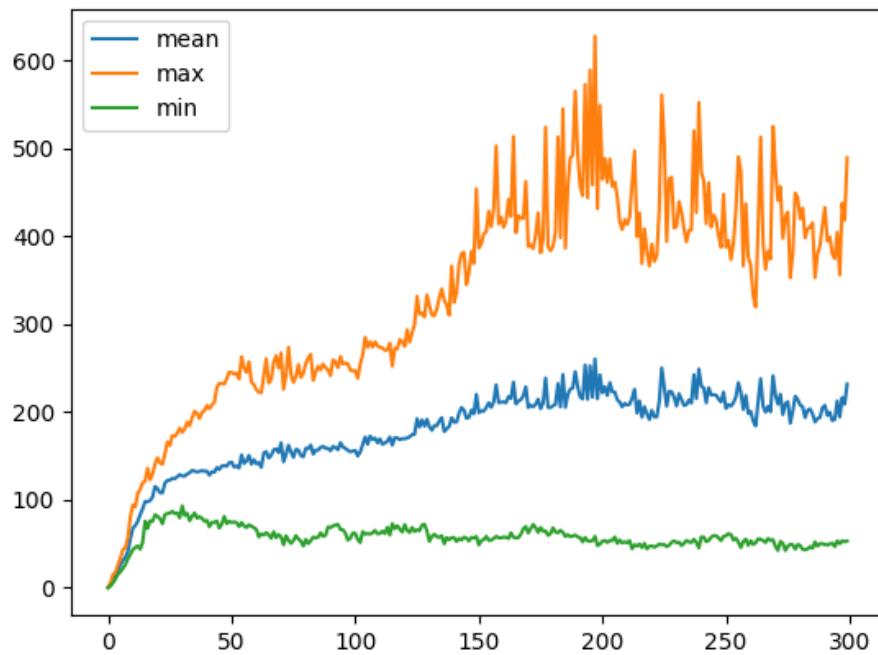


Figure 3.1 – Q moyen par partie sur 300 parties pour S0 - R0

- **S0 – R1**

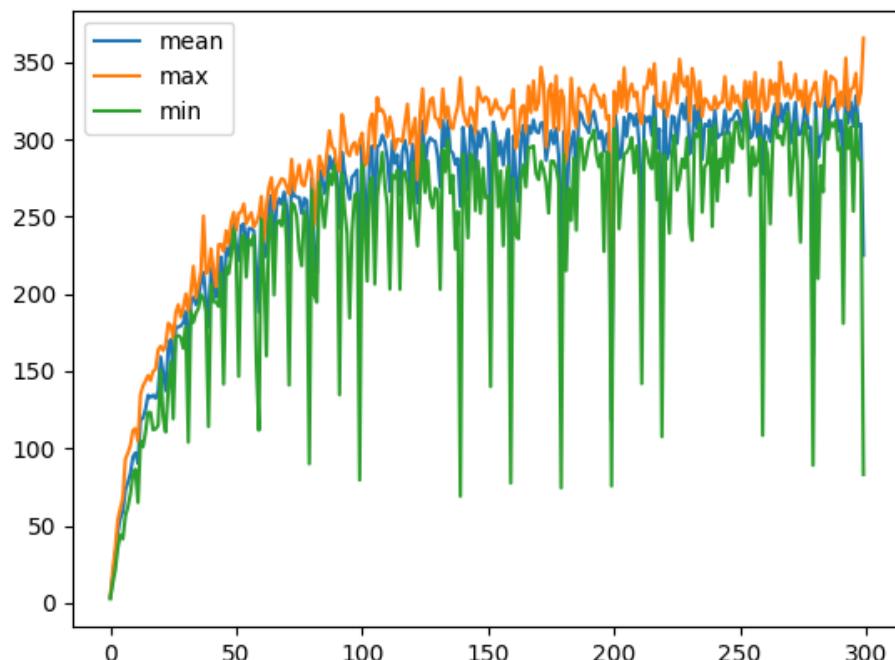


Figure 3.2 – Q moyen par partie sur 300 parties pour S0 - R1

- **S1 – R0**

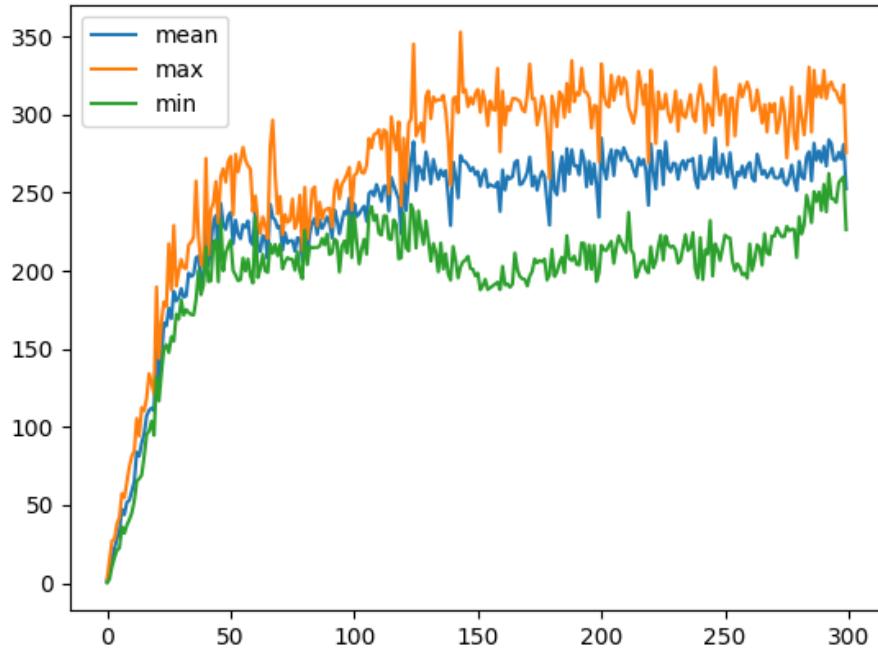


Figure 3.3 – Q moyen par partie sur 300 parties pour S1 - R0

- **S1 – R1**

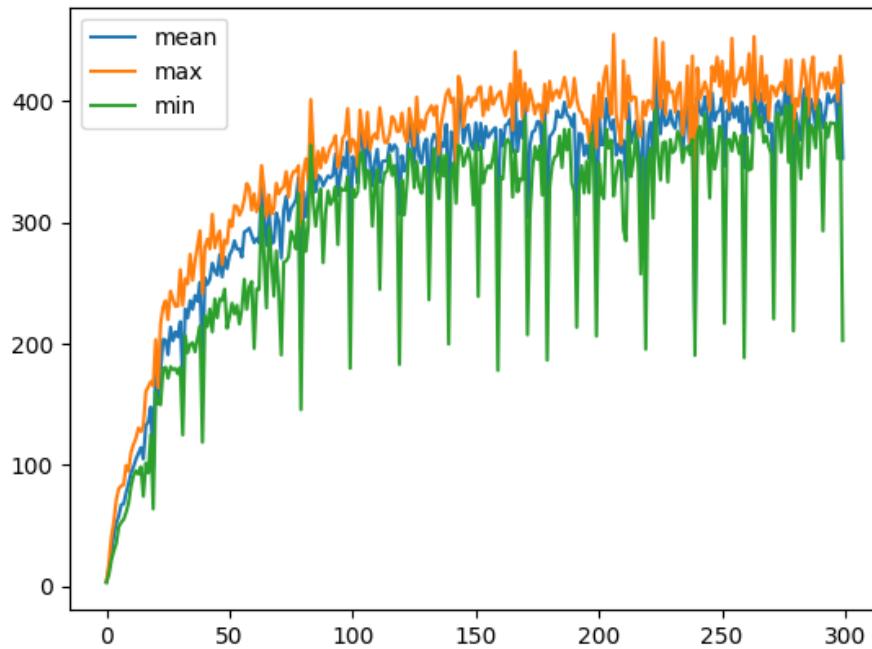


Figure 3.4 – Q moyen par partie sur 300 parties pour S1 - R1

- **AMOEBA**

- **S0 – R0**

Pour cette combinaison état/récompense je n'ai pas pu avoir de résultats concrets, en effet les expérimentations ne sont pas encore terminées (39 jours de calcul pour 121 parties, et 23 jours pour 65 parties) et pour le moment aucune partie n'a été gagnée.

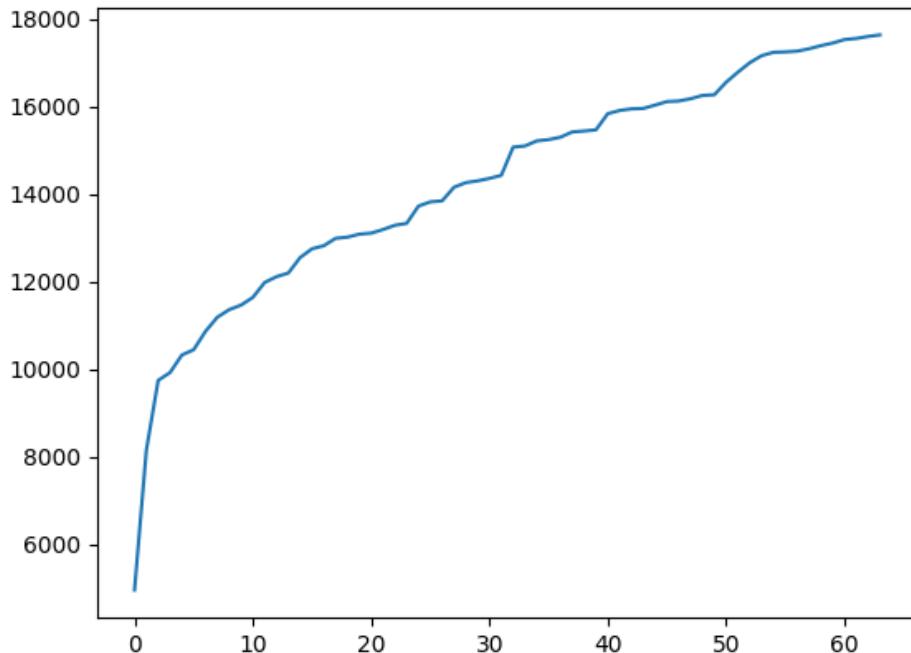


Figure 3.5 – Nombre d'agents contexte par partie pour la deuxième expérimentation de S0 - R0

- **S0 – R1**

Je n'ai pas pu avoir de résultats concrets non plus pour cette combinaison état/récompense. En effet à chaque fois l'expérimentation s'arrêtait prématurément du à un « *Out Of Memory* », les erreurs complètes sont disponibles en *annexe 3.1*.

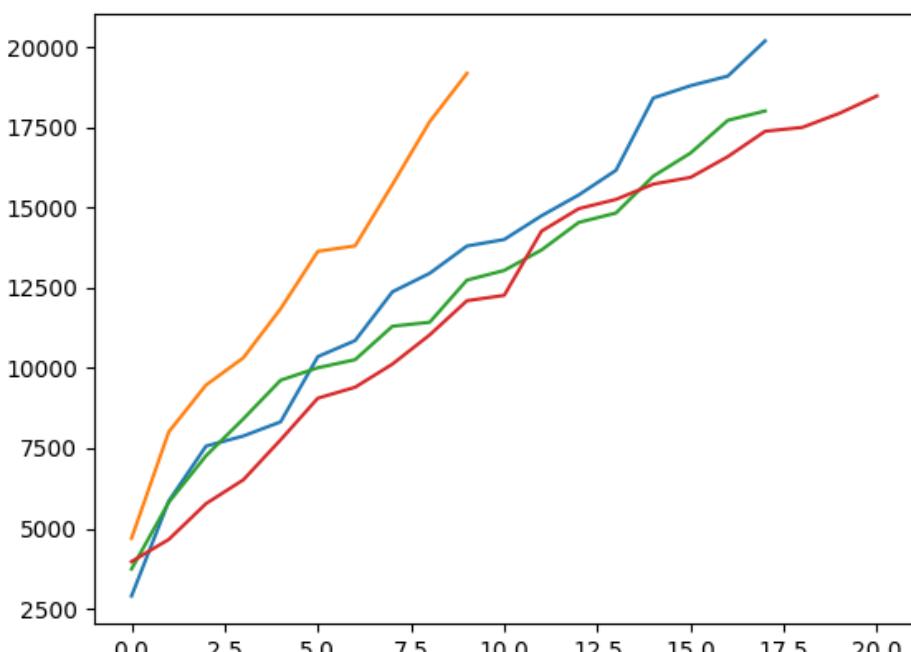


Figure 3.6 – Nombre d'agents contexte par partie pour chacune des 4 expérimentations de S0 - R1

◦ **S1 – R0**

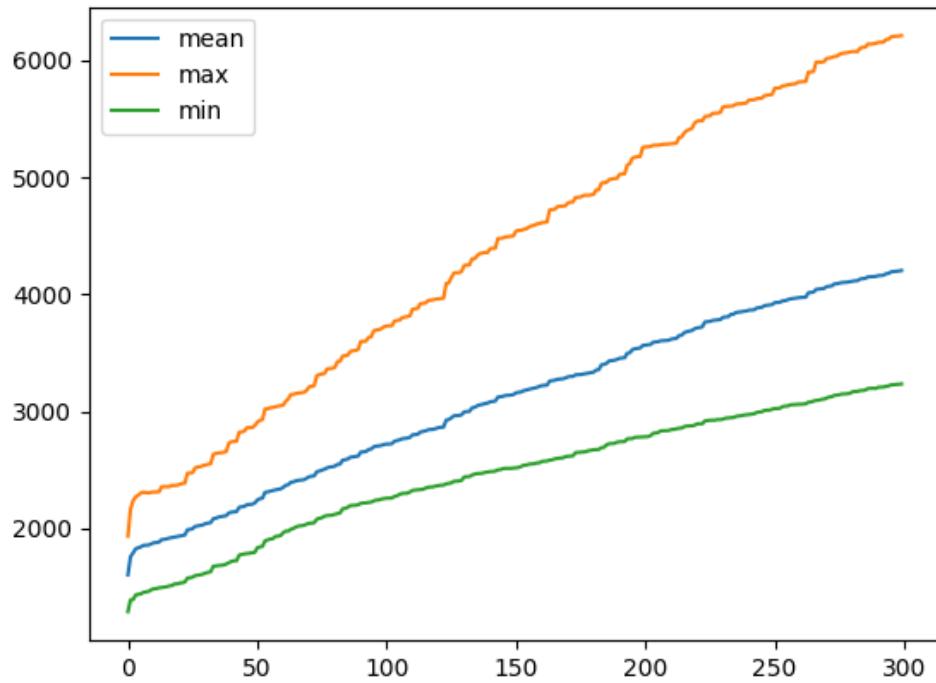


Figure 3.7 – Nombre d'agents contexte par partie pour S1 - R0

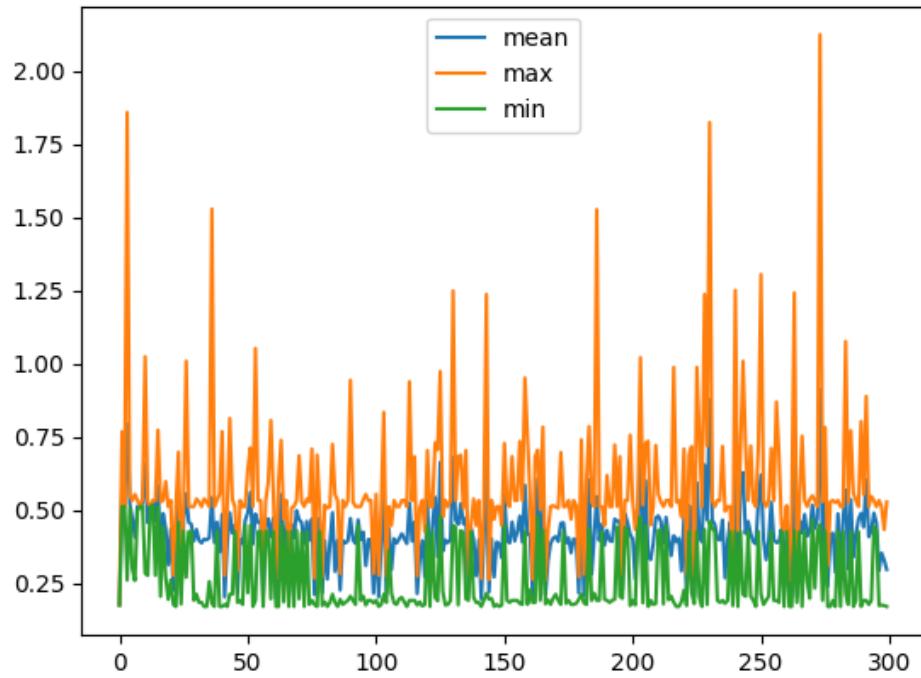


Figure 3.8 – Criticité moyenne par partie pour S1 - R0

Moyenne : 0,42

- **S1 – R1**

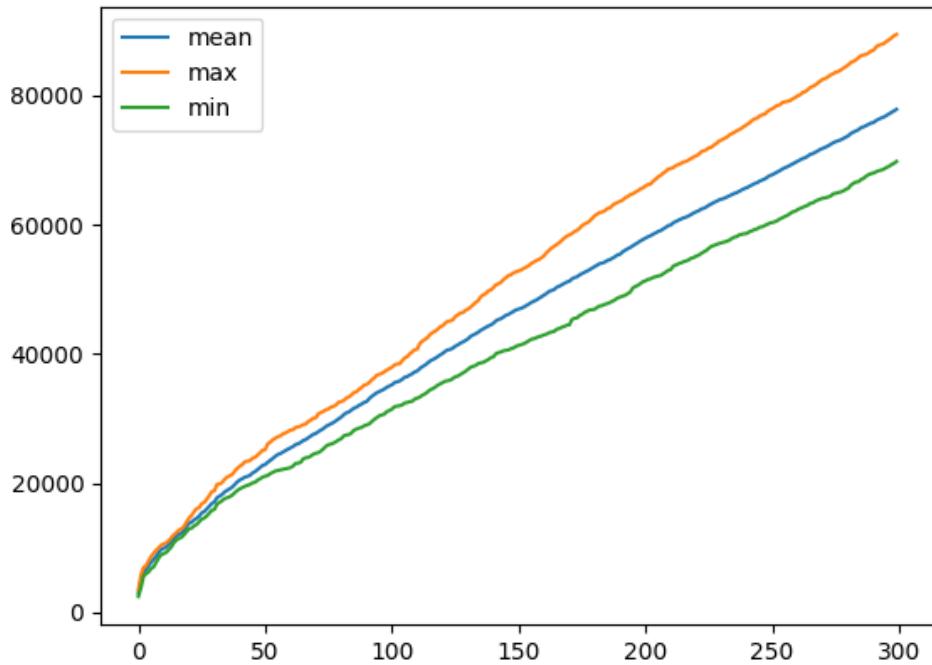


Figure 3.9 – Nombre d'agents contexte par partie pour S1 - R1

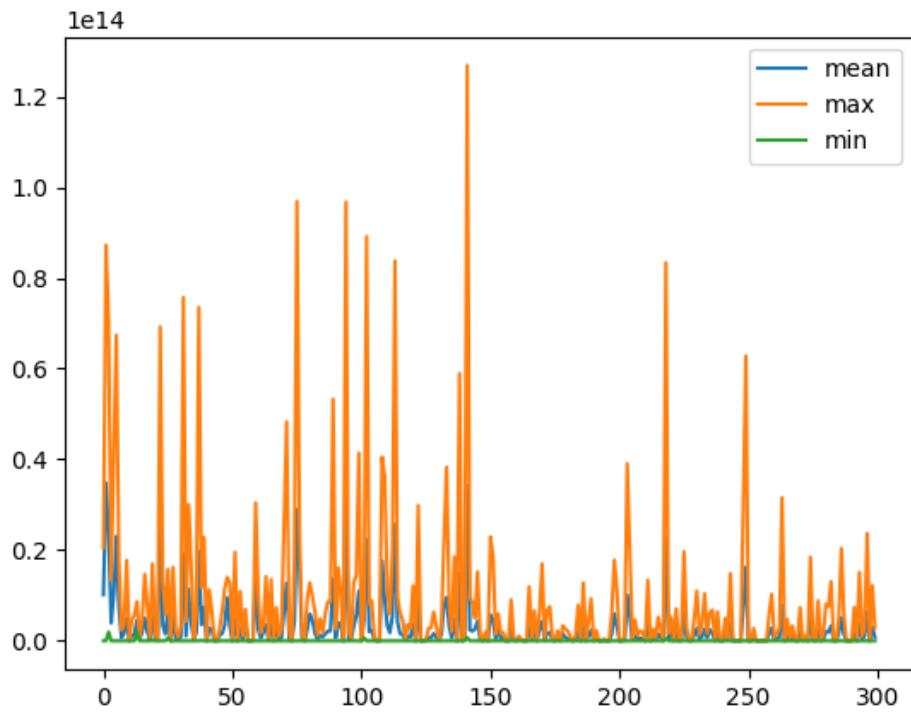


Figure 3.10 – Criticité moyenne par partie pour S1 - R1

Il faut multiplier les valeurs de l'axe des ordonnées par 1e14

Moyenne : 3,07e12

Discussion

Problème des agents contexte sur AMOEBA

On voit sur les résultats d'AMOEBA que le nombre d'agents contexte ne fait qu'augmenter alors qu'il devrait se stabiliser. La criticité moyenne devrait baisser au fur et à mesure de l'entraînement et ce n'est pas ce qu'elle semble faire.

J'ai donc cherché ce qui pouvait causer un tel problème. J'ai d'abord executé le code de base de Julien Nigon pour voir si le problème venait de mes modifications. J'utilise ici **S1** et **R1**.

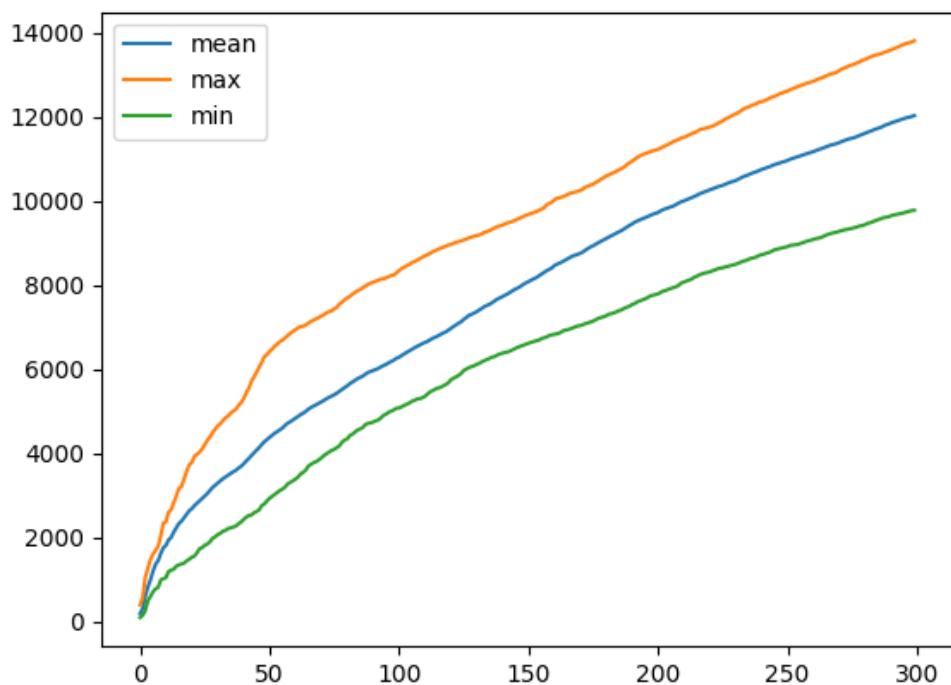


Figure 4.1 – Nombre d'agents contexte en fonction du nombre de parties jouées, moyenne, maximum et minimum sur 4 expérimentations, sur le code de base de Julien Nigon.

Victoires (/300) : 234,5 (261 ; 224 ; 224 ; 229).

Récompense par partie (en moyenne) : 10459,71

Temps de calcul : 1h 02m ; 1h 23m ; 1h 52m ; 1h 56m.

On voit que le nombre de parties gagnées est bien plus grand que dans les résultats présentés dans le tableau plus haut et dépasse en moyenne le nombre de victoires du Deep Q Learning pour cette combinaison état/récompense, pourtant la valeur moyenne de récompense par partie lui est toujours inférieure. Une explication possible, qui avait été soulevée par Julien Nigon, est que plus on va vite (donc dans ce cas, plus la valeur de récompense est élevée) plus on a de chance de rentrer en collision avec des ennemis ce qui nous donne donc plus de chance de perdre la partie. Le nombre d'agents contexte est bien plus petit que sur la *figure 3.9* mais ne semble toujours pas converger comme on le voudrait.

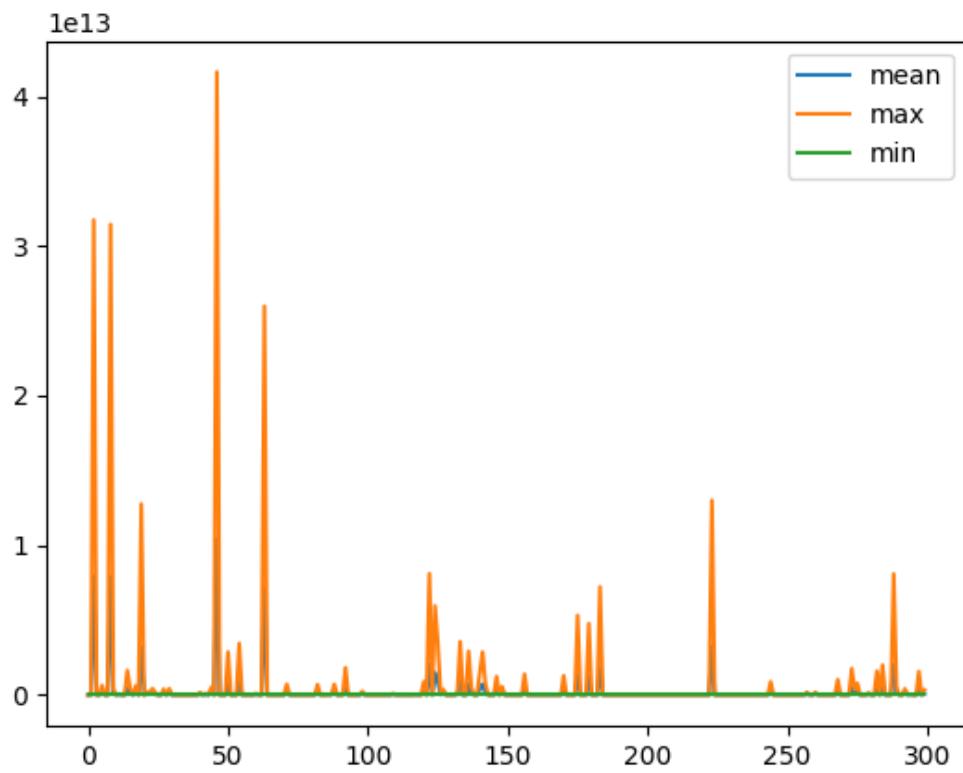


Figure 4.2 – Criticité moyenne par partie liée à l’expérience de la figure 4.1 (code de base)

Il faut multiplier les valeurs de l’axe des ordonnées par 1e13

Moyenne : 2,09e11

Le graphique des situations non coopératives des deux premières parties de cette expérience est disponible en *annexe 4.1*.

J’ai ensuite listé les éléments que j’avais modifié sur le code de base pour voir ce qui pouvait influer sur ces résultats :

- L’entraînement ne se fait plus que sur le premier niveau mais sur les 20 premiers (sauf 2 et 8). Cet enchainement de 20 niveaux est répété 15 fois (en tout 300 parties). On passe au niveau suivant que l’on ait gagné ou non.
- Le choix des actions se fait uniquement sur les 14 actions possibles (et non sur les 32 comprenant des combinaisons de touche menant à une contradiction comme aller à droite et à gauche en même temps par exemple).
- Le choix des actions se fait avec le request de AMOEBA.
- La probabilité d’exploration (de faire une action aléatoire) commence à 1 et diminue jusqu’à un minimum de 0,05 en étant multiplié à chaque action par 0,9999 (avant elle était fixée à 0,01).
- Lors du calcul de la distance au premier obstacle vers la droite et vers le bas les boules de feu envoyées par Mario ne sont plus prises en compte.
- Le temps limite est de 400 mario-secondes (et non plus 200).

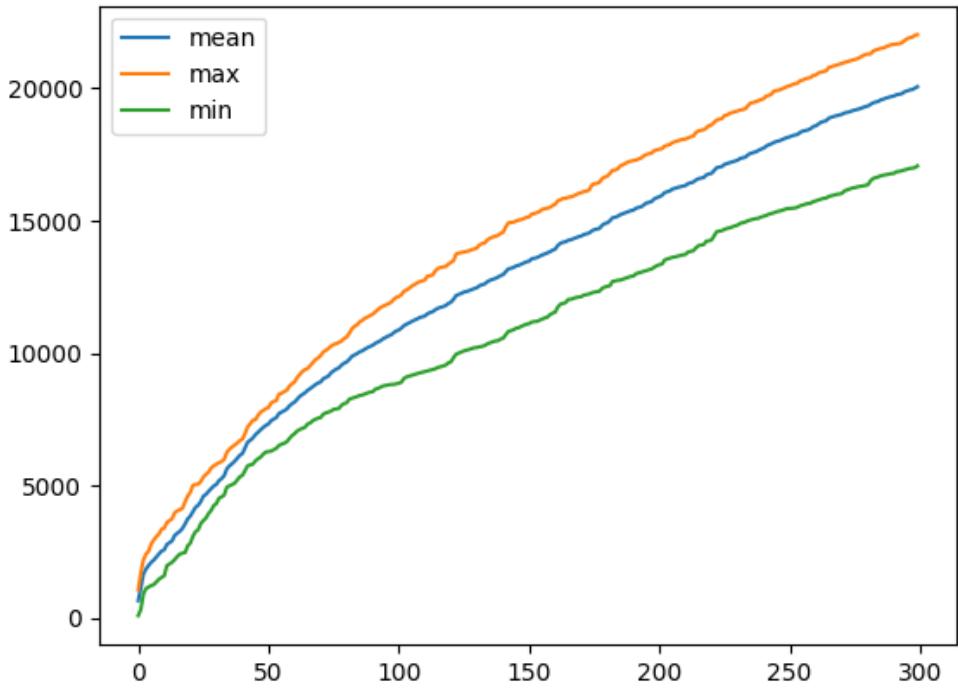


Figure 4.3 – Nombre d'agents contexte en fonction du nombre de parties jouées, moyenne, maximum et minimum sur 4 expérimentations, sur le code de base de Julien Nigon (en répétant les 20 premiers niveaux).

Victoires (/300) : 159,25 (185 ; 150 ; 139 ; 163).

Récompense par partie (en moyenne) : 11037,31

Temps de calcul: 4h 46m ; 5h 44m ; 5h 30m ; 3h 25m.

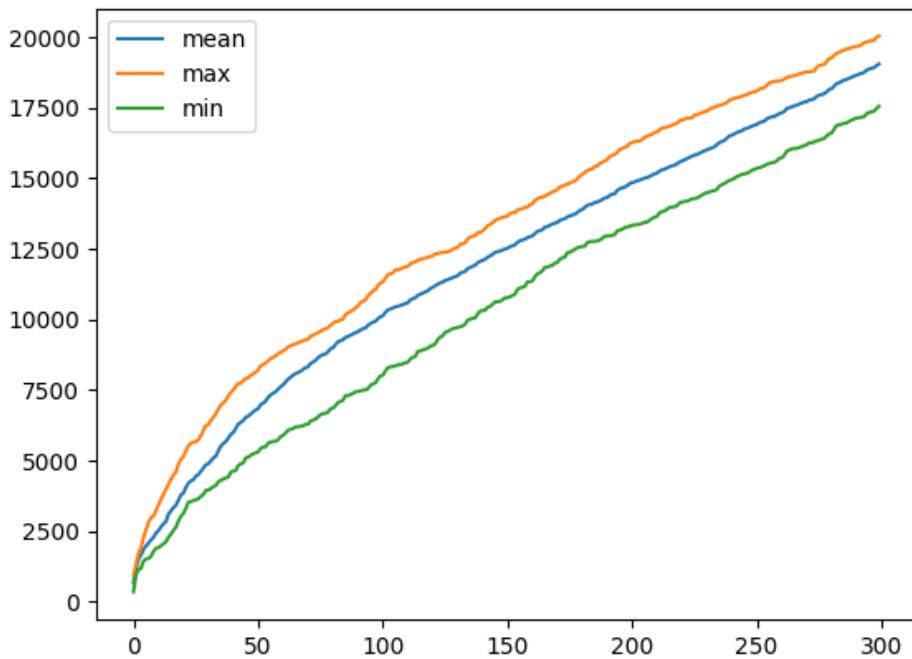


Figure 4.4 – Nombre d'agents contexte en fonction du nombre de parties jouées, moyenne, maximum et minimum sur 4 expérimentations, sur le code de base de Julien Nigon (en répétant les 20 premiers niveaux et en mettant 14 actions possibles au lieu de 32).

Victoires (/300) : 92,25 (59 ; 99 ; 108 ; 103).

Récompense par partie (en moyenne) : 9516,83

Temps de calcul : 3h 13m ; 3h 36m ; 4h 19m ; 4h 30m.

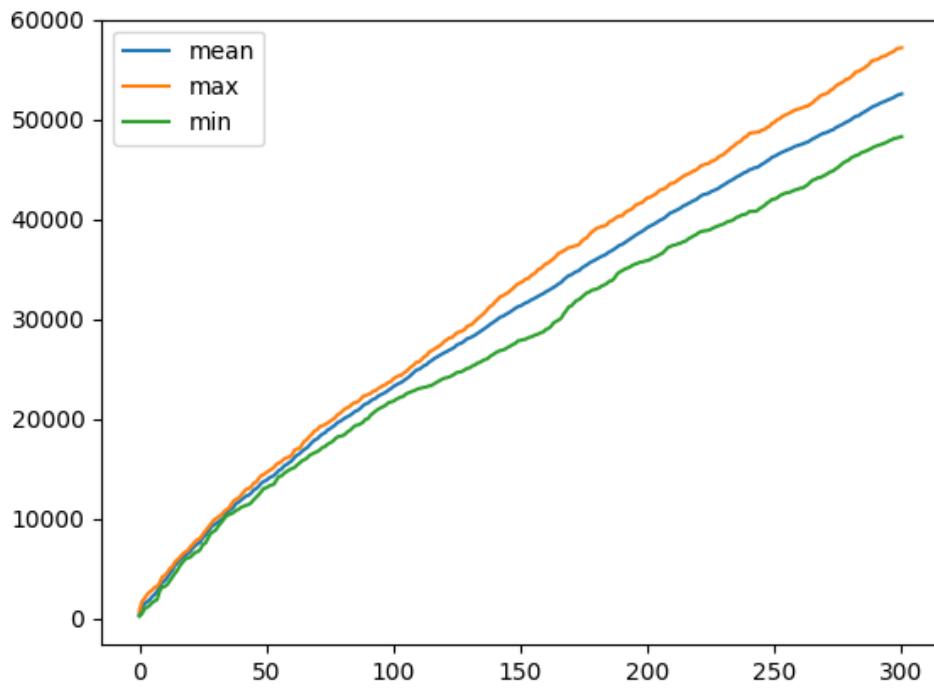


Figure 4.5 – Nombre d'agents contexte en fonction du nombre de parties jouées, moyenne, maximum et minimum sur 4 expérimentations, sur le code de base de Julien Nigon (en répétant les 20 premiers niveaux, en mettant 14 actions possibles au lieu de 32, en utilisant le request).

Victoires (/300) : 51,75 (62 ; 63 ; 31 ; 51)

Récompense par partie (en moyenne) : 2604,81

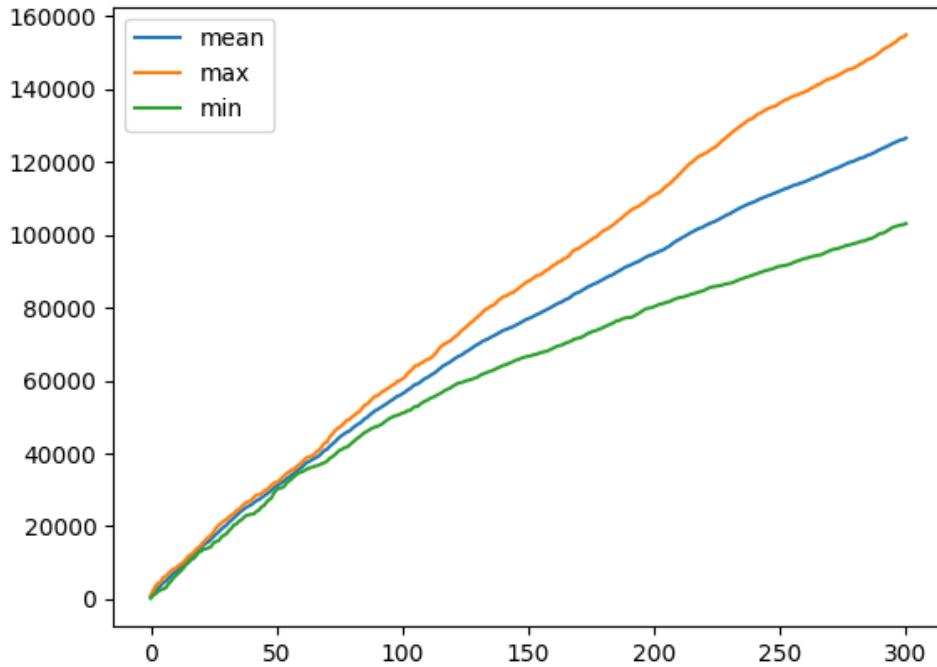


Figure 4.6 – Nombre d'agents contexte en fonction du nombre de parties jouées, moyenne, maximum et minimum sur 4 expérimentations, sur le code de base de Julien Nigon (en répétant les 20 premiers niveaux, en utilisant le request).

Victoires (/300) : 153,5 (164 ; 155 ; 164 ; 131)

Récompense par partie (en moyenne) : 5038,44

Conclusion et perspectives

Nous voyons sur les résultats du Deep Q Learning (*figure 3.1, 3.2, 3.3, 3.4*) que le système à bien l'air de converger pour chaque configuration. Pourtant, les résultats avec la grille (S0) sont moins bien qu'avec les 4 paramètres (S1), je m'attendais à l'inverse car S0 donne plus d'informations au système sur l'environnement, une raison possible est que j'ai peut-être mal ajusté le réseau de neurones à convolution, bien que les paramètres liés à l'architecture semblent particulier (très petit nombre de filtre par exemple) ce sont ceux avec lesquels j'ai eu les meilleurs résultats.

Il est intéressant de noter que pour S0 - R0 (*figure 3.1*) la courbe de Q moyen la plus basse (courbe verte) est associée à l'expérimentation ayant eu le plus de victoire (45) et la courbe la plus haute (courbe orange) est liée à l'expérimentation ayant eu presque le moins de victoires (4). Une potentielle explication peut être que R0 n'avantage pas uniquement le fait de gagner le niveau, il y a différentes récompenses liées à d'autres actions, comme récupérer des pièces, tuer des ennemis etc. qui peuvent donc éloigner le système de l'objectif principal qui est de terminer le niveau.

Pour ce qui est de AMOEBA, après un entretien avec Julien Nigon, nous nous sommes aperçus que la version de Mario utilisée n'était pas forcément la dernière. Si c'est le cas, il faudrait refaire les expérimentations pour nous assurer des résultats présentés dans ce rapport. De plus je n'ai pas du tout touché aux paramètres de AMOEBA, à savoir les seuils d'erreur et d'inexactitude, ce qui pourrait influer grandement sur les résultats.

C'est d'ailleurs un point positif pour AMOEBA par rapport au Deep Q Learning, en effet ce dernier possède énormément d'hyper-paramètres à ajuster (une liste non exhaustive est disponible en *annexe 2.1* pour ceux liés à l'architecture et *2.3* pour les autres) alors que AMOEBA en possède très peu. Le temps d'ajustement de ces paramètres est donc bien plus petit pour AMOEBA.

Il est intéressant de voir que lorsque nous utilisons 14 actions au lieu de 32 les résultats semblent moins bien (*figure 4.4* comparée à la *figure 4.3*) alors que nous aurions pu penser l'inverse. Il faudrait voir si cela fait la même chose avec le Deep Q Learning, pour voir si le problème est lié à AMOEBA ou à l'environnement de Mario.

Pour les graphiques liés au problème des agents contexte de AMOEBA, on voit sur les figures que c'est principalement l'utilisation du request qui semble poser problème. L'une des différences avec l'algorithme précédent est que ce dernier ne faisait qu'une « traversée » pour choisir l'action tandis qu'avec le request on en fait une pour chaque action.

Comme on pouvait s'y attendre les systèmes s'en sortent bien mieux avec R1 qu'avec R0 (ce qui pourrait expliquer pourquoi les courbes avec R0 sont moins « lisse » pour le Deep Q Learning que celles avec R1).

Il y a plusieurs raisons, tout d'abord, avec R1 le système est récompensé lorsqu'il avance vers la fin du niveau, ce qui lui donne une information relativement importante sur le jeu. De plus, avec R1 nous n'avons pas le problème des récompenses éparses car à chaque action le système en reçoit une, contrairement à R0 où le nombre d'actions avant une récompense peut être assez grand. Les récompenses éparses font que le système peut mettre du temps avant de comprendre quelles sont les actions pouvant être positives et celles pouvant être négatives dans une situation donnée.

Il pourrait être intéressant d'ajouter à R1 une récompense négative lorsque le système perd le niveau et une positive lorsqu'il le gagne. En effet, cela pourrait lui permettre d'être plus performant, en essayant d'éviter les situations menant à une défaite et en favorisant celles menant à une victoire.

De plus il faudrait essayer d'utiliser une fonction de récompense mélangeant R0 et R1 et de comparer les résultats.

Pour ce qui est des autres méthodes, à savoir NEAT et NEAT + Q, j'ai pu à peine commencer NEAT, mais j'espère pouvoir les mettre en œuvre plus tard (si possible dans les mois à venir) et les travaux seront mis sur le dépôt github associé (dont le lien est disponible sur la page d'introduction).

Les fichiers contenant tous les résultats qui ont permis de tracer les graphiques de ce rapport sont disponibles sur le dépôt github. Vous pouvez utiliser le script python *resultManager.py* pour tracer les courbes et analyser les données.

Bibliographie

Mario

[Karakovskiy et al. 2012] Sergey Karakovskiy and Julian Togelius. The Mario AI Benchmark and Competitions. 2012.

[Curran et al. 2015] William Curran, Tim Brys, Matthew E. Taylor and William D. Smart. Using PCA to Efficiently Represent State Spaces. 2015.

AMOEBA

[Nigon 2017] Julien Nigon. Apprentissage artificiel adapté aux systèmes complexes par auto-organisation coopérative de systèmes multi-agents. 2017.

Deep Reinforcement Learning (Deep Q Learning)

[Watkins 1989] C. J. Watkins. Learning from delayed rewards. 1989.

[Liao et al. 2012] Y. Liao, K. Yi and Z. Yang. Final Report Reinforcement Learning to Play Mario. 2012.

[Mnih et al. 2013] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Ridmiller. Playing Atari with Deep Reinforcement Learning. 2013.

[Mnih et al. 2015] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis. Human-level control through deep reinforcement learning. 2015.

[Hasselt et al. 2015] H. Hasselt, A. Guze and D. Silver. Deep Reinforcement Learning with Double Q-learning. 2015.

[Hausknecht et al. 2015] Matthew Hausknecht and Peter Stone. Deep recurrent Q-learning for partially observable MDPs. 2015.

[Klein 2016] Sean Klein. Final Report Deep Q-Learning to Play Mario. 2016.

[Schaul et al. 2016] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. Prioritized Experience Replay. 2016.

[Lillicrap et al. 2016] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra. Continuous Control With Deep Reinforcement Learning. 2016.

[Wang et al. 2016] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. 2016.

[Mnih et al. 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. 2016.

[Wolfshaar 2017] Jos van de Wolfshaar. Deep Reinforcement Learning of Video Games. 2017.

[Andersen 2018] P. Andersen. Deep Reinforcement Learning using Capsules in Advanced Game Environments. 2018.

Evolutionary Algorithm (NEAT)

[Montana et al. 1989] David J. Montana and Lawrence Davis. Training Feedforward Neural Networks Using Genetic Algorithms. 1989.

[Stanley et al. 2002] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks through Augmenting Topologies. 2002.

[Stanley et al. 2003] Kenneth O. Stanley, Bobby D. Bryant, Risto Miikkulainen. Evolving Adaptive Neural Networks with and without Adaptive Synapses. 2003.

[Hausknecht et al. 2013] M. Hausknecht, J. Lehman, R. Miikkulainen and P. Stone. A Neuroevolution Approach to General Atari Game Playing. 2013.

[Miikkulainen et al. 2017] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy and Babak Hodjat. Evolving Deep Neural Networks. 2017.

NEAT and Backpropagation (with Q Learning)

[Baldwin 1896] J. M. Baldwin. A new factor in evolution. 1896.

[Whiteson et al. 2006] Shimon Whiteson and Peter Stone. Evolutionary Function Approximation for Reinforcement Learning. 2006.

[Whiteson 2010] Shimon Whiteson. Adaptive Representations for Reinforcement Learning. 2010.

[Land et al. 2017] Julien van der Land, Thomas Nijman, Stephan Boomker and Jasper de Boer. Playing a volleyball game with reinforcement learning. 2017.

Annexes

2 Travaux effectués

- Annexe 2.1 – Architectures actuelles des réseaux de neurones (avec le Q Learning)

Architecture pour la représentation d'état utilisant la grille :

Couche de convolution 1	1 filtre de taille 3*3 (stride 2)
Couche de convolution 2	2 filtres de taille 2*2 (stride 1)
Couche LSTM	2048 neurones
Couche de sortie	14 neurones (pour les 14 actions possibles)

Architecture pour la représentation d'état utilisant les 4 paramètres :

Couche d'entrée	4 neurones
Couche LSTM	256 neurones
Couche de sortie	14 neurones

- Annexe 2.2 – Anciennes architectures pour le Q Learning

Architecture pour la représentation d'état utilisant la grille :

Couche de convolution 1	16 filtres de taille 7*7 (stride 4)
Couche de convolution 2	32 filtres de taille 3*3 (stride 2)
Couche de convolution 3	64 filtres de taille 2*2 (stride 1)
Couche LSTM	512 neurones
Couche de sortie	14 neurones (pour les 14 actions possibles)

Architecture pour la représentation d'état utilisant les 4 paramètres :

Couche d'entrée	4 neurones
Couche LSTM	8 neurones
Couche de sortie	14 neurones

- Annexe 2.3 – Hyper-paramètres des réseaux de neurones (Q Learning)

Hyper-paramètre	Valeur	Commentaire
Batch size	32	
Replay memory size	100000	
Target network update frequency	4000	
Compute action frequency	1	DeepMind utilise 4 comme valeur (principalement pour accélérer les calculs) mais j'ai eu de très mauvaises performances avec cette valeur et 24 FPS.
Discount factor	0,9	
Learning rate init	0,00025	
Learning rate decay	0,99999	(exponential decay)
Learning rate decay step	5	
Learning rate min	0,0001	
Initial exploration	1	
Final exploration	0,05	
Decay exploration	0,9999	
RNN trace length	8	
Mask half loss	True	
LSTM stateful	True	
Optimizer	RMSProp	Meilleures performances que Adam ici.
RMSProp momentum	0,95	
Loss function	MSE	Cross entropy loss semble moins performante.

3 Résultats

- Annexe 3.1 – Trois erreurs « Out Of Memory » différentes pour AMOEBA S0 R1

```
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.cov(MillerUpdatingRegression.java:504)
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.regress(MillerUpdatingRegression.java:936)
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.regress(MillerUpdatingRegression.java:905)
at agents.localModel.LocalModelMillerRegression.getFormula(LocalModelMillerRegression.java:132)
at agents.context.Context.toString(Context.java:483)
at agents.head.Head.playWithoutOracle(Head.java:298)
at agents.head.Head.play(Head.java:206)
at kernel.Scheduler.learn(Scheduler.java:186)
at kernel.AMOEBA.request(AMOEBA.java:159)
at experiments.mario.MarioManager2Regression2.computeGoodAction(MarioManager2Regression2.java:525)
at experiments.mario.MarioManager2Regression2.playOneStep(MarioManager2Regression2.java:432)
at kernel.AMOEBA.run(AMOEBA.java:130)
```

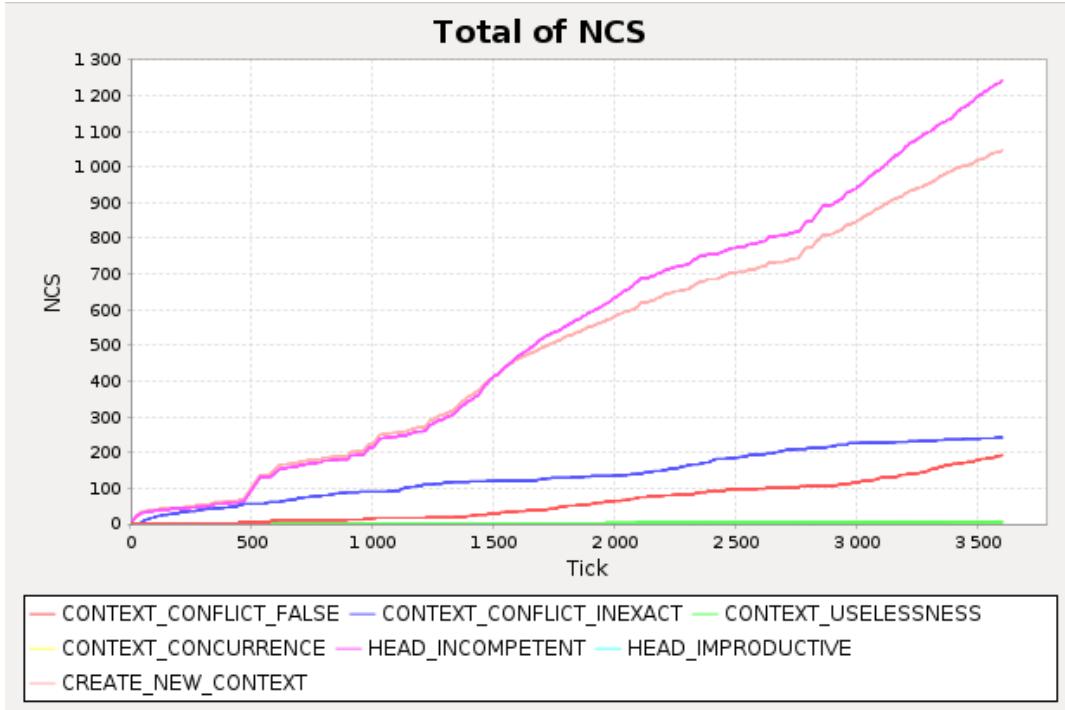
```
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.cov(MillerUpdatingRegression.java:504)
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.regress(MillerUpdatingRegression.java:936)
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.regress(MillerUpdatingRegression.java:905)
at agents.localModel.LocalModelMillerRegression.updateModel(LocalModelMillerRegression.java:184)
at agents.context.Context.buildContext(Context.java:114)
at agents.context.Context.<init>(Context.java:64)
at agents.head.Head.createNewContext(Head.java:324)
at agents.head.Head.play(Head.java:192)
at kernel.Scheduler.learn(Scheduler.java:186)
at kernel.AMOEBA.run(AMOEBA.java:127)
```

```

Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.<init>(MillerUpdatingRegression.java:113)
at org.apache.commons.math3.stat.regression.MillerUpdatingRegression.<init>(MillerUpdatingRegression.java:140)
at agents.localModel.Regression.<init>(Regression.java:16)
at agents.localModel.LocalModelMillerRegression.updateModel(LocalModelMillerRegression.java:167)
at agents.context.Context.buildContext(Context.java:114)
at agents.context.Context.<init>(Context.java:64)
at agents.head.Head.createNewContext(Head.java:324)
at agents.head.Head.play(Head.java:192)
at kernel.Scheduler.learn(Scheduler.java:186)
at kernel.AMOEBA.run(AMOEBA.java:127)

```

4 Discussion



Annexe 4.1 – Graphique des situations non coopératives sur les 2 premières parties pour l’expérience liée à la figure 4.1 (code de base). Je n’ai pas pu prendre le graphique pour les 300 parties car l’affichage de celui-ci fait énormément ralentir les calculs (à tel point qu’à partir du milieu de la deuxième partie le temps entre chaque action se compte en dizaines de secondes, sur mon ordinateur).