

Étude sur l'apprentissage par renforcement dans les jeux

Projet de Bachelor 2018 - hepia

Étudiant : Federico Pfeiffer
Professeur : Guido Bologna

18 juin 2018

Résumé

// TODO

Table des matières

1	Introduction	3
1.1	Contexte	3
1.2	Objectifs	3
1.3	Réalisation	3
2	Théorie	5
2.1	Définitions	5
2.2	Notions de base	7
2.2.1	Fonctionnement général de l'apprentissage par renforcement	7
2.2.2	Valeur d'un état	7
2.2.3	$V(s)$: Équation de Bellman	9
2.2.4	Apprentissage par renforcement : pseudo code	10
2.3	Algorithme Q-Learning	11
2.3.1	Explore vs Exploit	11
2.3.2	Valeur d'un état avec $Q(s,a)$	12
2.3.3	Algorithme Q-Learning	12
2.3.4	Limitations de l'algorithme Q-Learning	14
2.4	Réseaux neuronaux	14
2.4.1	Enjeux	14
2.4.2	Présentation	15
2.4.3	Prédiction	16
2.4.4	Apprentissage	21
2.4.5	Difficultés	22
2.5	Algorithme Deep Q-Network (DQN)	23
2.5.1	Mise à jour de la fonction $Q(s,a)$: ajustements à effectuer	23
2.5.2	Pseudo-code de l'algorithme Deep Q-Network	25
3	Implémentation	28
3.1	Fonctionnement	28
3.1.1	Vue d'ensemble : choix d'une action	28
3.1.2	Apprentissage	29

3.1.3	Regarder l'agent jouer	30
3.1.4	Tester soi-même l'environnement	30
3.1.5	Configuration	30
3.2	Architecture	33
3.2.1	Diagramme de classes	33
3.2.2	Méthodes clés	34
4	Résultats	36
4.1	Fonction récompense	36
4.1.1	Problème rencontré	36
4.1.2	Explication	37
4.1.3	Fonctions récompense utilisées	39
4.2	Batch size	43
4.2.1	Problème rencontré	43
4.2.2	Explication	43
4.3	Input X des réseaux neuronaux et discount factor γ	44
4.4	Learning rate	45
4.5	Hyper-paramètres finaux	46
5	Conclusion	47
5.1	Difficultés	47
5.2	Perspectives	47
5.2.1	Nouvel environnement	47
5.2.2	Nouvel algorithme	48
6	Annexes	49
6.1	Installation	49
6.2	Jouer soi-même au jeu	49
6.3	Voir l'agent évoluer	49
6.4	Entraîner l'agent depuis zéro	50

Chapitre 1

Introduction

1.1 Contexte

L'apprentissage par renforcement est une branche de l'intelligence artificielle connaissant un essor depuis quelques années, et qui a permis des avancées technologiques telles que la voiture autonome, le logiciel "AlphaGo" de Google réussissant à battre le champion du monde de Go, ou encore la création de robots capables d'apprendre à se mouvoir sans intervention humaine. Le champ d'application est vaste, et est actuellement sujet à de nombreuses recherches pour découvrir de nouveaux champs d'applications.

J'ai profité du travail de Bachelor afin d'étudier par la pratique l'apprentissage par renforcement. Pour ce faire, un environnement de jeu a été créé pour faire évoluer un agent dans celui-ci. L'étude des réseaux de neurones artificiels a également été nécessaire étant donné qu'une grande partie des algorithmes d'apprentissage par renforcement utilisent de tels réseaux de neurones.

1.2 Objectifs

Les objectifs de ce présent travail de bachelor sont les suivants :

- Comprendre les modèles de réseaux de neurones artificiels standards et profonds
- Comprendre les concepts de l'apprentissage par renforcement
- Comprendre et utiliser au moins un "framework" pour l'utilisation de l'apprentissage par renforcement
- Construire un jeu avec un agent qui apprend par renforcement
- Analyse des résultats
- Rédaction du rapport

1.3 Réalisation

Le jeu dans lequel l'agent évolue est le suivant : un agent (en bleu) doit éviter des ennemis représentés en rouge, tout en récupérant de la nourriture (en vert). Au fur et à mesure que l'agent avance, l'endurance de l'agent faiblit (le diamètre de l'agent diminue). Si l'endurance de l'agent est trop faible, le jeu se termine. Si l'agent se fait attraper par un ennemi, le jeu est également terminé. Si l'agent atteint la nourriture, son endurance est remise au maximum, et une nouvelle nourriture est placée aléatoirement dans le jeu. Le but de l'agent est jouer le plus longtemps sans que ce dernier se termine.

Les actions possibles pour l'agent sont d'avancer dans une direction des points cardinaux (N, NE, E, SE, S, SO, O, NE), ou ne rien faire. Le nombre d'actions possibles équivaut donc à neuf.



FIGURE 1.1 – Présentation de l'environnement (Jeu)

Afin de faciliter l'apprentissage, ce dernier a été divisé en 3 parties :

1. Apprendre à éviter les ennemis (sans nourriture)
2. Apprendre à attraper la nourriture (sans ennemis)
3. Une fois les deux premières compétences apprises, l'agent doit ensuite apprendre à jongler entre les deux compétences afin de jouer convenablement au jeu.

Cela implique les contraintes suivantes :

- L'environnement (jeu) doit être configurable pour qu'il puisse être utilisé indépendamment avec uniquement des ennemis, que de la nourriture, ou les deux en même temps.
- L'agent doit être implémenté de manière générique, de sorte à ce qu'il puisse apprendre les connaissances indépendamment. (éviter les ennemis, attraper la nourriture, ou les deux en même temps).
- Une fois une compétence apprise, l'on doit pouvoir la sauvegarder afin d'être réutilisable pour la suite de l'apprentissage : une fois que l'agent a appris à éviter les ennemis ou attraper de la nourriture, ces deux compétences doivent être chargées pour la dernière étape de l'apprentissage, sans que l'agent n'ait à réapprendre ces compétences à chaque fois que le programme démarre.

Le "framework" utilisé dans ce présent travail de bachelor est la librairie *Tensorflow* développée par Google, spécialisée dans la création de réseaux de neurones artificiels.

Chapitre 2

Théorie

2.1 Définitions

Apprentissage par renforcement

L'apprentissage par renforcement consiste à considérer un agent autonome évoluant dans un environnement donné. La seule chose que connaît l'agent sont les différentes actions qu'il peut effectuer au sein de son environnement. Le but de l'agent est d'obtenir un score maximal par le biais de récompenses vis à vis des actions qu'il a entrepris.

Au début, l'agent choisit les actions de manière aléatoire, et on lui indique si l'action qu'il a entrepris lui confère une récompense, un malus, ou pas de retour particulier. Si l'état terminal est atteint, l'environnement est remis à zéro et l'agent continue ses expériences depuis cet état initial. L'agent peut ainsi "apprendre par la pratique", les actions à entreprendre en fonction de l'état où il se trouve, dans le but de maximiser le score.

Agent

Un agent est un programme informatique autonome, ou une partie d'un programme qui est autonome, c'est à dire qu'il ne dépend pas d'interventions extérieures pour évoluer. (Il ne doit pas rester bloqué en attente d'une intervention humaine).

Les entrées (input) de l'agent permettent de percevoir l'environnement dans lequel l'agent évolue. Les sorties (output) de l'agent sont les actions choisies par l'agent pour influencer sur l'environnement dans lequel il évolue. Il n'y a pas d'autres entrées/sorties. Toute la logique applicative liée à l'agent est donc implémentée au sein de l'agent. Il est important de signaler que l'agent n'a pas forcément conscience de l'état complet de l'environnement. La représentation de l'environnement que l'agent se fait dépend donc des capteurs qu'il détient.

Environnement

L'environnement est le monde dans lequel l'agent évolue et dans lequel l'agent, de par ses actions, modifie l'état de l'environnement. L'environnement est séparé de l'agent, et le seul lien entre ces derniers sont les *inputs* de l'agent, c'est à dire les capteurs par lesquels l'agent peut percevoir l'environnement, et les *outputs* de l'agent, c'est à dire les différentes actions possibles pour l'agent pour influencer sur l'environnement.

État et état terminal

Un état est une "photo instantanée" du monde perçu par l'agent. Un état comprend l'état de l'environnement ainsi que l'état de l'agent. Un état terminal représente un état dans lequel il n'est plus possible d'effectuer une action. Dans cet état terminal, l'environnement est alors "remis à zéro" afin que l'agent puisse recommencer à évoluer.

Action et récompense

Chaque action mène à un changement d'état. Une action est la seule possibilité pour un agent d'influencer l'environnement dans lequel il évolue. C'est aussi la seule manière pour un agent de savoir si l'action qu'il a entreprise est bonne, mauvaise ou "sans effet". Pour chaque action, l'agent reçoit une récompense (bonne, mauvaise ou neutre). Si la récompense est "bonne", l'agent reçoit un score positif ; si l'action est "mauvaise", l'agent reçoit une récompense négative. L'agent peut alors ajuster son expérience dans le but de favoriser les actions menant à maximiser un score positif, ou minimiser un score négatif.

Si la récompense est neutre, l'agent enregistre néanmoins l'action qu'il a entreprise au cas où l'action neutre mène à futur état permettant un score positif ou négatif. Si l'agent est un joueur d'échec et que l'action qu'il entreprend ne permet pas de gagner, il est quand même important d'enregistrer l'état/action vécu si jamais cela l'emmène par la suite dans un état/action où il gagne ou perd. De même, si une voiture doit tourner à gauche et qu'elle continue tout droit pendant une fraction de seconde, elle doit pouvoir savoir si le fait de tourner à gauche la fraction de seconde d'après l'amènera à une sortie de route, ou si elle peut quand même tourner à gauche sans sortir de route.

L'important pour l'agent est de pouvoir enregistrer les actions prises d'une manière où d'une autre, en vue de savoir si cette action mène plutôt à un échec, une réussite, ou à un état neutre.



FIGURE 2.1 – Action et récompenses

Épisode

Un épisode commence depuis l'état initial (environnement mis à zéro) et termine à un état terminal. Lorsque l'agent atteint un état terminal, l'environnement est réinitialisé et l'agent recommence un nouvel épisode. L'agent enregistre chaque récompense durant toute la durée de l'épisode afin de connaître le score total que l'agent a eu pendant l'épisode.

2.2 Notions de base

2.2.1 Fonctionnement général de l'apprentissage par renforcement

L'apprentissage par renforcement consiste à donner une récompense à un agent lorsque ce dernier parvient à effectuer une tâche demandée, ou lorsqu'il s'en approche. De la même manière, l'agent reçoit un malus lorsque ce dernier s'éloigne de la tâche demandée. Le but de l'agent est d'obtenir un maximum de récompense. De cette manière, le comportement de l'agent est influencé par les récompenses et malus qu'il reçoit. Au fur et à mesure que l'agent effectue des actions et reçoit des récompenses/malus, ce dernier tend à renforcer un comportement menant à un maximum de récompenses, et éviter les actions menant à des malus. À noter qu'à partir de maintenant nous ne parlerons que de récompenses : un malus sera simplement traduit par une récompense négative.

Plus concrètement, l'agent commence par effectuer des actions aléatoires et reçoit une récompense / malus pour chaque action effectuée. Pour chaque action commise, l'agent enregistre l'état S dans lequel il se trouvait, l'action effectuée, la récompense/malus reçue et l'état S' dans lequel il se retrouve suite à l'action. Si l'agent se retrouve dans un état terminal, l'environnement est réinitialisé mais l'agent garde en mémoire les actions / états / récompenses vécus. À une fréquence définie par le développeur, l'agent passe par une phase d'ajustement (*training*) où il parcourt tous les actions / états / récompenses vécus de sorte à ajuster son comportement (actions choisies) lors des états futurs qu'il rencontrera, dans le but de maximiser les récompenses.

2.2.2 Valeur d'un état

Jusqu'à la fin du chapitre, nous définirons le score d'un épisode par la somme des récompenses obtenues au cours des actions effectuées. Le but de l'agent est d'obtenir le meilleur score possible, et de retenir les états par lesquels il est passé afin d'obtenir ce meilleur score. (Et également de retenir les états par lesquels il est passé qu'il l'ont mené à un mauvais score).

Afin d'apprendre à choisir les bonnes actions à effectuer en fonction de l'état où l'agent se trouve, ce dernier a pour but d'apprendre la valeur de chaque état qu'il rencontre : un état qui le rapproche d'un meilleur score a une meilleure valeur qu'un état qui l'éloigne d'un meilleur score.

L'enjeu est donc de trouver une manière d'attribuer une valeur à chaque état en fonction des états qui vont suivre, afin de choisir l'action appropriée menant à un meilleur état. Prenons l'exemple (arbitraire) comprenant 2 épisodes, avec un état initial s_0 (*s pour state*) :

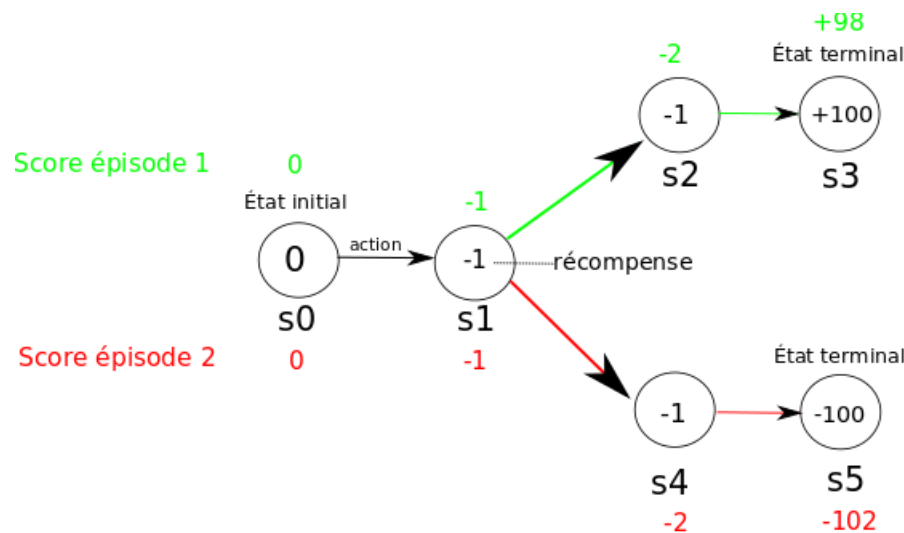


FIGURE 2.2 – Valeur d'un état : introduction 1

Dans le premier épisode, l'agent commence à l'état s_0 (état initial), puis choisit les actions passant par s_1 , s_2 et s_3 . s_3 étant un état terminal, l'épisode 1 s'arrête et l'agent obtient comme score +98 (qui est la somme des récompenses obtenues pendant son épisode). Dans le deuxième épisode, l'agent commence à l'état s_0 (état initial), puis choisit les actions passant par s_1 , s_4 et s_5 . s_5 étant un état terminal, l'épisode 2 s'arrête et l'agent obtient un score de -102 pour le deuxième épisode.

Comment pourrions-nous attribuer une valeur à chaque état ? Intuitivement, nous concluons qu'il est nécessaire d'attribuer une valeur à chaque état une fois qu'un épisode est terminé. En effet, une fois l'épisode terminé, nous pouvons évaluer chacune des actions effectuées par l'agent, étant donné que nous savons si les actions effectuées ont mené à l'état terminal voulu. Une première idée serait donc d'attribuer à chaque état la récompense finale auquel l'état peut mener. Si un état mène à deux récompenses différentes, nous pouvons par exemple attribuer la valeur de l'état en faisant la somme des deux récompenses auxquels l'état peut mener.

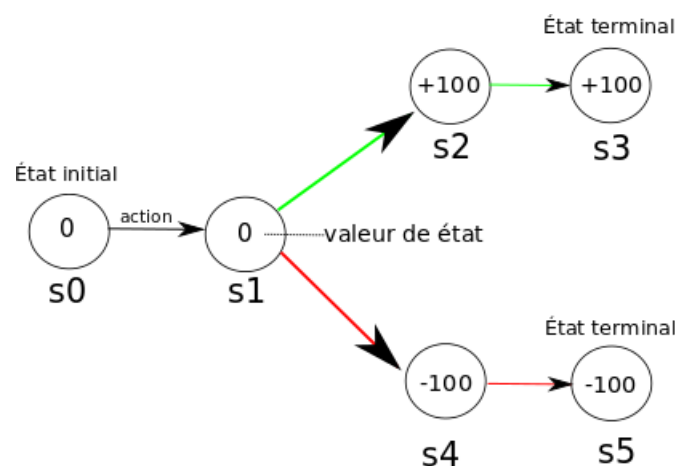


FIGURE 2.3 – Valeur d'un état : introduction 2

Après avoir effectué sa phase d'ajustement (*training*) dans le but de connaître la valeur de chaque état (phase d'apprentissage), l'agent peut ensuite savoir que depuis l'état s_1 , il est préférable de choisir l'action menant à l'état s_2 , étant donné qu'un meilleur score l'attend.

Discount factor (γ)

Le problème de la méthode précédente pour attribuer une valeur à un état est que l'on ne tient pas compte de la distance à laquelle se trouve la récompense finale.

En prenant l'exemple suivant, il serait préférable pour l'agent de passer directement à s3 depuis s1. (L'agent a meilleur temps d'effectuer l'action qui le mène le plus rapidement à l'état menant à une forte récompense). Or, avec la théorie vue jusqu'ici l'agent n'a pas de moyen de savoir qu'il lui est préférable de passer directement par s3 :

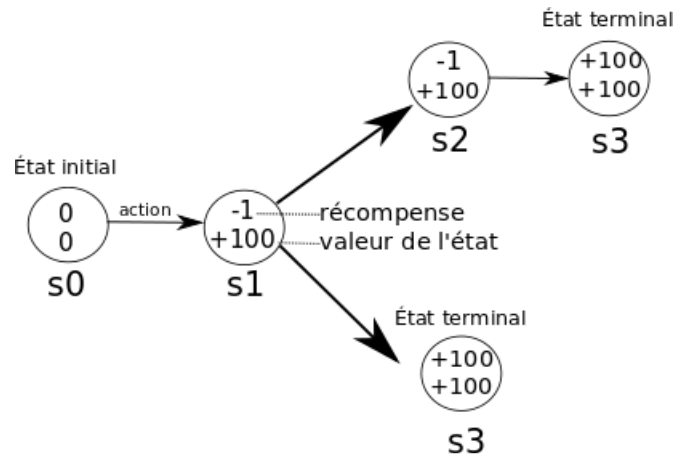


FIGURE 2.4 – Valeur d'un état : introduction sans discount factor

Une solution pour palier à ce problème est de calculer la valeur d'un état en ajoutant un coefficient nommé *discount factor* γ de cette manière : $V(s_t) = \gamma V(s_{t+1})$ avec $\gamma \in [0, 1]$. En prenant $\gamma = 0.9$, on parvient à attribuer une valeur à chaque état qui inclut cette notion de distance envers la récompense la plus forte. Depuis s1, l'agent va donc tendre vers s3 directement car la valeur de s3 est plus grande que s2.

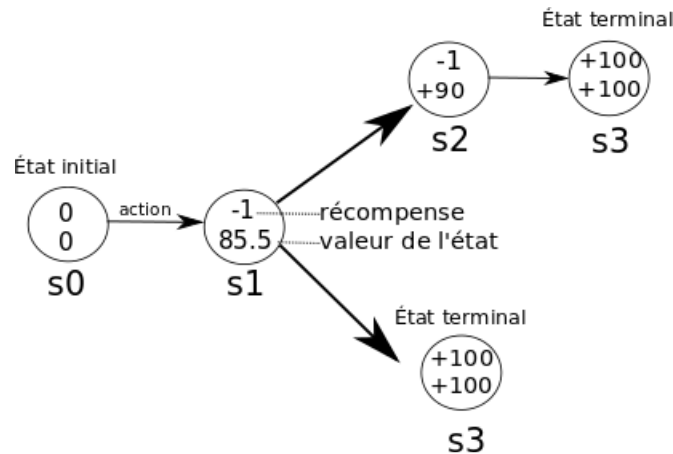


FIGURE 2.5 – Valeur d'un état : introduction avec discount factor

2.2.3 $V(s)$: Équation de Bellman

La fonction $V(s) = \gamma V(s_{t+1})$ est incomplète. La version formelle vient de l'équation de Bellman. Elle prend en compte plusieurs notions supplémentaires :

Elle prend en compte la récompense reçue à chaque état, ce qui donne

$$V(s_t) = r + \gamma V(s_{t+1})$$

Aussi, il est possible qu'un environnement soit fait de telle sorte qu'une transition d'un état à

un autre ne soit pas déterministe. Par exemple, si un agent choisit une action menant à un état S' , on peut imaginer que l'environnement aie un comportement aléatoire menant l'agent à un état qui n'était pas prévu par l'agent. (dans l'exemple d'une voiture autonome, le fait de voir un enfant jouer sur un trottoir n'implique pas forcément que celui-ci reste sur le trottoir). Pour un état s_t , il faudrait y inclure la probabilité de se retrouver dans un état s_{t+1} , et recevoir une récompense r après avoir fait une action a . En d'autres termes, pour un état s_t , il faut évaluer tous les états s_{t+1} et toutes les récompenses r , et en calculer la probabilité.

La fonction $V(s)$ devient

$$V(s) = \sum_{s_{t+1}} \sum_r p(s_{t+1}, r | s_t, a) \{r + \gamma V(s_{t+1})\}$$

Enfin, il se peut que pour un certain état, l'agent aie une certaine probabilité de choisir une certaine action. (Il peut par exemple connaître la meilleur action à entreprendre, mais il a une politique d'avoir 10% de probabilité d'effectuer une action aléatoire dans le but d'explorer certaines actions). La fonction est alors agrémentée du terme suivant : $\sum_a \pi(a|s)$. Le π représente la politique de l'agent.

Ainsi, l'équation de Bellman indique que la valeur d'un état peut se calculer de la manière suivante :

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \{r + \gamma V_\pi(s')\}$$

Le terme $\pi(a|s)$ représente la probabilité de choisir une action a étant donné l'état s , suivant une politique π . (Dans le cas où la politique de l'agent est de choisir une action aléatoire tous les 10% par exemple). Si l'agent a une politique visant à ne choisir que la meilleure action et que le choix de cette action n'est pas probabiliste, ce terme vaut simplement 1. De même, si l'agent n'a qu'une action possible pour un certain état s , le terme $\sum_a \pi(a|s)$ équivaut à $\pi(a|s)$

Le terme $p(s', r | s, a)$ représente, depuis un état s , la probabilité de se retrouver dans l'état suivant s' et de gagner la récompense r , ayant choisi l'action a . Pour connaître la valeur de l'état s , il faut donc pouvoir envisager tous les états s' possibles, et toutes les récompenses r en choisissant une action a . C'est pour cela que l'on "itère" en faisant la somme de tous les $p(s', r | s, a)$ en parcourant tous les états s' possibles et toutes les récompenses r possibles depuis un état s . D'où les deux sommes $\sum_{s'} \sum_r$ devant $p(s', r | s, a)$: $\sum_{s'} \sum_r p(s', r | s, a)$

L'équation de Bellman peut donc être lue de la manière suivante : depuis un état s , on itère toutes les actions possibles depuis cet état. Dans cette itération, on itère toutes les récompenses possibles et tous les états s' qui découlent en choisissant l'action a . A chaque itération, on multiplie la probabilité $\pi(a|s)$ à la probabilité $p(s', r | s, a)$ et multiplie à la $\{r + \gamma V_\pi(s')\}$

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \{r + \gamma V_\pi(s')\}$$

Dans ce projet de bachelor, nous considérerons l'environnement déterministe, ce qui réduit l'équation de Bellman à ceci :

$$V_\pi(s) = \sum_a \pi(a|s) \{r + \gamma V_\pi(s')\}$$

2.2.4 Apprentissage par renforcement : pseudo code

Maintenant que nous connaissons comment attribuer une valeur à chaque état, nous pouvons imaginer un premier pseudo-code permettant à un agent d'apprendre à évoluer dans un environnement. Le problème de cette équation est qu'elle part du principe que nous connaissons

tous les états et actions possibles. Il faut donc procéder de manière itérative jusqu'à ce que la valeur d'un état converge vers une valeur fixe. Nous nous arrêtons lorsque toutes les valeurs des états semblent avoir été calculées. C'est à dire, lorsque la valeur d'un état ne diverge pas plus qu'une valeur $\Delta_{\text{max_divergence}}$:

```
# phase d'apprentissage
world = new World()
agent = new Agent()
gamma = 0.9

max_divergence = 0.1
divergence = inf
while divergence > max_divergence :
    for state in world.get_all_states() :
        for action in world.get_possible_actions(state) :
            reward, next_state = world.do_action(state, action)
            old_v = agent.V[state]
            agent.V[state] = reward + gamma * agent.V[next_state]
            divergence = max(divergence, abs(old_v-agent.V[state]))

# phase une fois l'apprentissage effectué
state = world.reset()
while not world.is_in_terminal_state() :
    action = agent.choose_action_to_best_state(state)
    state = world.do_action(action)
```

Supposer que l'on doit connaître tous les états et actions possibles pose évidemment problème dans le cas de l'apprentissage par renforcement, étant donné que l'agent doit apprendre en explorant lui-même à travers les actions qu'il entreprend. Pour des raisons de temps de calcul, nous ne souhaitons pas explorer toutes les actions possibles, mais souhaitons explorer que celles qui nous semblent nous mener à un meilleur comportement de l'agent. Heureusement, des algorithmes ont été mis en place afin de pouvoir estimer correctement la fonction de Bellman ($V(s)$) sans avoir à tout explorer, permettant ainsi à l'agent de pouvoir estimer raisonnablement la valeur de l'état dans lequel il se trouve, et choisir l'action adéquate pour maximiser son score.

2.3 Algorithme Q-Learning

2.3.1 Explore vs Exploit

Dans la précédente section, nous avons vu un algorithme permettant d'évaluer $V(s)$ en supposant que l'on connaisse tous les états, actions et récompenses possibles suivant l'environnement donné. Dans la majorité des cas, il n'est évidemment pas possible de connaître toutes ces valeurs à l'avance. Il convient donc d'utiliser l'agent de sorte à ce qu'il explore de lui même toutes ces différentes valeurs dans le but d'estimer correctement la fonction $V(s)$.

Au fur et à mesure que l'agent explore les actions, la fonction $V(s)$ va petit à petit atteindre une estimation correcte, et l'agent va petit à petit connaître les meilleurs états à rencontrer, ou les meilleures actions à effectuer pour chaque état.

Comment savoir si l'agent dispose de connaissances suffisantes pour arrêter d'explorer/tester des actions ? A quel moment l'agent doit-il utiliser la connaissance qu'il dispose de $V(s)$? Une solution est d'effectuer une action aléatoire avec une probabilité ϵ . La valeur de ϵ peut varier en fonction du temps selon ce qu'a choisit le programmeur.

2.3.2 Valeur d'un état avec $Q(s,a)$

Utiliser $V(s)$ pour attribuer une valeur à chaque état peut s'avérer limitant : on n'enregistre finalement que la valeur d'un état en fonction de l'état en question. Au lieu d'attribuer une valeur à un état, nous pourrions attribuer une valeur à un couple état/action effectué.

La fonction attribuant une valeur à un couple état/action s'appelle la fonction *Q-value*, ou $Q(s,a)$, et peut être représentée de la sorte : (ressemblant fortement à $V(s)$) :

$$V_{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \{r + \gamma V_{\pi}(s')\} \quad (2.1)$$

$$Q_{\pi}(s, a) = \sum_{s'} \sum_r p(s', r | s, a) \{r + \gamma Q_{\pi}(s', a')\} \quad (2.2)$$

Comme indiqué plus haut, l'environnement utilisé est déterministe, ce qui permet de réduire les équations à ceci :

$$V_{\pi}(s) = \sum_a \pi(a|s) \{r + \gamma V_{\pi}(s')\} \quad (2.3)$$

$$Q_{\pi}(s, a) = \{r + \gamma Q_{\pi}(s', a')\} \quad (2.4)$$

2.3.3 Algorithme Q-Learning

Si nous connaissions tous les états, actions et récompenses possibles, il serait possible de définir $Q(s,a)$ dans un environnement déterministe, de la manière suivante :

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

Or, ne connaissant pas tous les états, actions et récompenses possibles, (et ne voulant pas tous les explorer), l'idée du Q-learning est d'ajuster petit à petit $Q(s,a)$, par une fraction α (appelée *learning-rate*).

$$Q(s, a) = old_Q(s, a) + \alpha(Q(s', a') - old_Q(s, a))$$

Le terme α est appelé "learning_rate" est choisi arbitrairement par le développeur et est compris entre $[0, 1]$. Il correspond à l' α utilisé lors d'un "gradient descent".

La méthode Q-Learning met donc $Q(s,a)$ à jour selon la formule suivante :

$$Q(s, a) = Q(s, a) + \alpha(\{r + \gamma Q(s', a')\} - Q(s, a))$$

Le pseudo-code de la méthode Q-learning peut être défini de la manière suivante :

```
# phase d'apprentissage
world = new World()
agent = new Agent()
max_episodes = 100000
gamma = 0.9
epsilon = 0.1
alpha = 0.1

for i in range(max_episodes) :
    state = world.reset()
    while not world.game_over() :
        action = agent.choose_action(state, epsilon)
        reward, next_state = world.do_action(state, action)
        a2 = agent.choose_action(next_state, epsilon)

        if world.game_over() :
            agent.Q[state][action] += alpha*(reward)
        else :
            agent.Q[state][action] += alpha*(reward+gamma*agent.Q[next_state][a2])

    state = next_state

# phase une fois l'apprentissage effectué
state = world.reset()
while not world.game_over() :
    action = agent.choose_action(state, epsilon)
    state, _ = world.do_action(action)
```

2.3.4 Limitations de l'algorithme Q-Learning

L'inconvénient majeur de l'algorithme Q-Learning vient du fait qu'il faut pouvoir stocker la fonction $Q(s,a)$ dans un tableau représentant chaque couple (état, action) : si le fait de garder en mémoire $V(s)$ utilise un tableau à 1 dimensions de taille équivalant à autant d'états possibles, la fonction $Q(s,a)$ nécessite beaucoup plus de mémoire car elle doit enregistrer les valeurs dans un tableau à 2 dimensions, de taille $|S| * |A|$ cases.

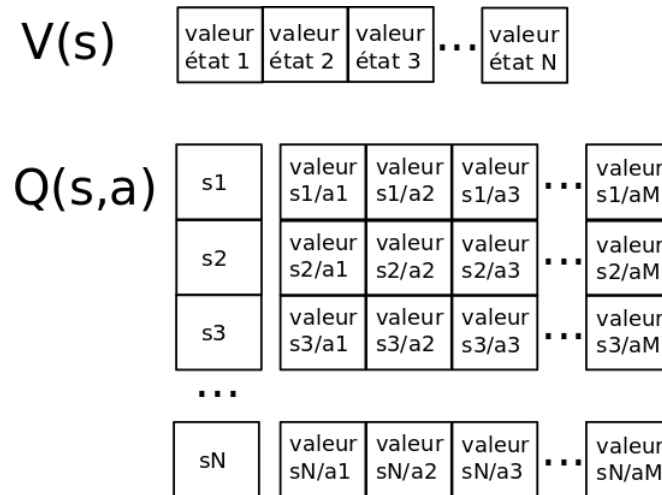


FIGURE 2.6 – Enregistrement $V(s)$ et $Q(s,a)$

Dans des environnements avec de nombreux (voir infinités) d'états et d'autant d'actions possibles, le stockage de la valeur de $Q(s,a)$ devient très difficile vu la taille qu'elle représente. De plus, si l'environnement présente autant d'états et d'action possibles, l'agent va devoir effectuer beaucoup d'exploration afin d'évaluer la valeur de chaque couple (état, action), ce qui rendrait l'apprentissage extrêmement lent, voir inefficace. Une solution pour palier à cet inconvénient est de remplacer le tableau $Q[s][a]$ par un réseau neuronal.

2.4 Réseaux neuronaux

2.4.1 Enjeux

Le problème majeur des méthodes d'apprentissage par renforcement vues jusqu'ici est que lorsque l'environnement devient complexe et qu'il comporte trop d'états/actions possibles, il devient difficile (voir impossible) d'arriver à une estimation correcte que $V(s)$ ou $Q(s,a)$. Les réseaux neuronaux se sont révélés être de très bons estimateurs dans de nombreux champs d'applications.

Les réseaux de neurones sont utilisés de la manière suivante :

1. On crée un réseau de neurones qui a pour but d'estimer (prédire) quelque chose de précis. Au début, le réseau de neurone renvoie des réponses aléatoires.
2. On entraîne ce réseau de neurone en lui donnant de nombreux exemples avec la solution à estimer pour chaque exemple. (exemple d'input, solution)
3. Au bout d'un certain nombre d'exemples donnés, le réseau de neurone s'est ajusté de sorte à pouvoir estimer (prédire) de nouveaux inputs en les estimant correctement.

De nombreuses applications ont été trouvées pour les réseaux de neurones : on peut par exemple fournir des milliers d'images comportant des tumeurs et des milliers d'images ne comportant pas de tumeurs pour qu'à la fin de son apprentissage, le réseau puisse indiquer si l'image (qu'il n'a

jamais vue) comporte ou ne comporte pas de tumeur. On peut également fournir un millier d'images de chiffres écrits à la main pour qu'à la fin de son apprentissage, le réseau puisse reconnaître le chiffre écrit dans une image quelconque que le réseau n'a pas encore vu. Le champ d'application est vaste : de la reconnaissance vocale à la proposition de contenu personnalisé pour Youtube ou Facebook, de nombreux domaines utilisent les réseaux neuronaux, suivant ce même principe de base. (Apprentissage, puis utilisation pour prédire quelque chose).

Dans le cas de l'apprentissage par renforcement, les réseaux de neurones permettent d'approximer correctement les fonctions $V(s)$ ou $Q(s,a)$ dans de nombreux environnements. L'utilisation d'un réseau de neurone permet les méthodes telles que le Deep Q-Learning que nous verrons dans la prochaine section.

2.4.2 Présentation

Un réseau de neurones est constitué des parties suivantes :

- Une couche "d'input", représentant la couche par laquelle les données rentrent dans le réseau.
- Un certain nombre de couches, appelées "hidden layers". C'est dans ces couches que le réseau neuronal fait ses calculs pour effectuer une prédiction.
- Une couche "d'output". C'est à travers cette couche que le résultat de la prédiction est transmise.

L'idée est que chaque neurone est relié à tous les neurones de la couche suivante, afin d'effectuer un calcul particulier à chaque couche. A la fin, la couche "output" indique le résultat de la prédiction du réseau neuronal.

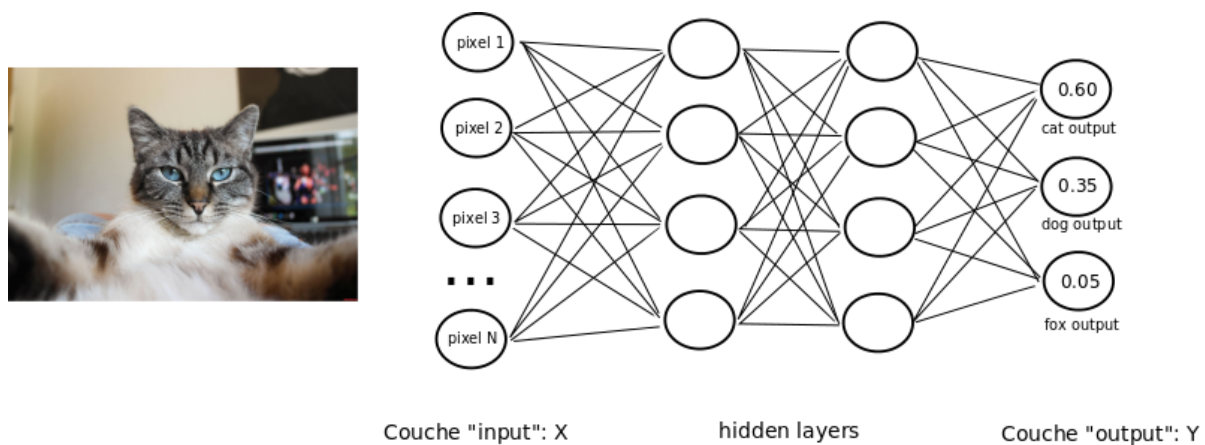


FIGURE 2.7 – Réseau neuronal : présentation 1

Dans la figure ci-dessus, le but du réseau neuronal est de prédire si l'image donnée dans la couche "input" est un chat, un chien, ou un renard. Le résultat de la prédiction indique que le réseau neuronal estime que l'image est à 60% un chat, à 35% un chien, et à 5% un renard. Pour interpréter le résultat, on prends le neurone de sortie dont l'estimation est la plus haute : cela correspond à catégorie "chat", ce qui indique que le réseau neuronal a estimé que l'image donnée en entrée est un chat.

Dans le deuxième exemple, le but du réseau neuronal est de prédire la valeur de $Q(s,a)$ dans un environnement où l'agent a quatre actions possibles pour un état donné. Ici, l'état est représenté par la position de l'agent (couche "input"). En sortie, le réseau indique l'estimation de la valeur $Q(s,a)$ de chaque action possible dans l'état donné en entrée. Si l'agent ne décide pas d'effectuer une action aléatoire, l'agent effectuera l'action dont la valeur est la plus haute : dans ce cas il s'agit de l'action 'Right' (se diriger vers la droite).

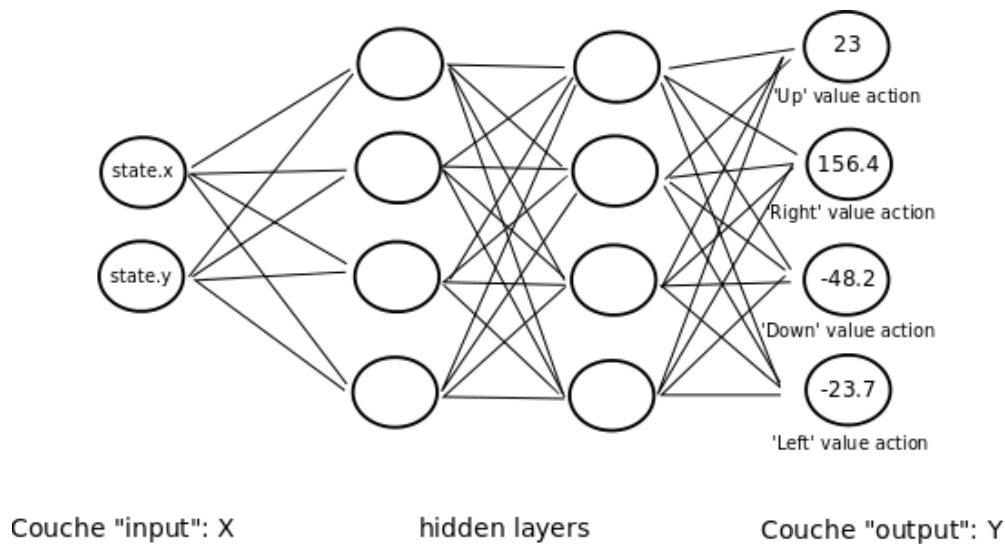


FIGURE 2.8 – Réseau neuronal : présentation 2

2.4.3 Prédiction

Notion de base

Pour décrire le fonctionnement d'un réseau neuronal, nous commencerons par la manière dont il effectue les prédictions.

Prenons l'exemple le plus simple : une valeur d'entrée (un neurone d'entrée) dont la valeur est x , une simple couche cachée d'un seul neurone contenant un poids w (weight), et une seule valeur de sortie (un seul neurone d'output) qui ne fait rien d'autre que ressortir la valeur de la couche cachée. La prédiction du neurone est calculée comme suit :

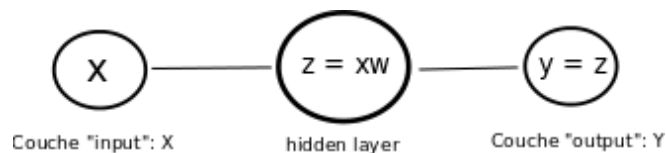


FIGURE 2.9 – Réseau neuronal : théorie 1

Le but de chaque neurone caché est de prendre la valeur d'entrée, de modifier cette valeur en fonction de poids (w) (weight) que le neurone contient, et de transmettre son résultat au neurone suivant. Dans cet exemple, la prédiction vaudra : $y = z = xw$

Notation matricielle

Prenons maintenant le même exemple, mais avec deux valeurs en entrées. x_1 et x_2 . La théorie veut que le neurone caché doit avoir 2 poids afin d'effectuer son calcul.

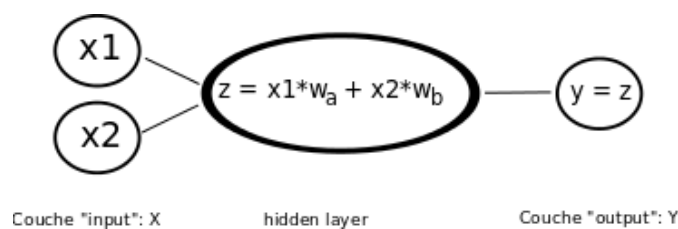


FIGURE 2.10 – Réseau neuronal : théorie 2

Dans cet exemple, la prédiction vaut donc $y = z = x_1w_a + x_2w_b$

On peut reconnaître le caractère matriciel de la prédiction et représenter la prédiction selon la notation suivante. X représente les valeurs d'entrées, W représente les poids du neurone caché, Z représente le résultat du neurone caché, et Y représente la valeur de sortie :

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2.5)$$

$$W = (w_a, w_b) \quad (2.6)$$

$$Z = WX = (w_a, w_b) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (x_1w_a + x_2w_b) \quad (2.7)$$

$$Y = Z = (x_1w_a + x_2w_b) \quad (2.8)$$

À noter que dans la théorie qui suit, nous utiliserons la notation 'X' (majuscule) pour définir une entrée (couche input), et la notation 'Y' (majuscule) pour définir les valeurs de sortie (output).

Grâce à la notation matricielle, on peut maintenant imaginer un réseau avec 2 entrées et une couche de 2 neurones cachés. Comme la sortie de la couche cachée contient 2 valeurs et qu'on décide que le neurone de sortie ne fait toujours rien d'autre que de copier la sortie de la couche cachée, le neurone de sortie renverra 2 valeurs en sortie.

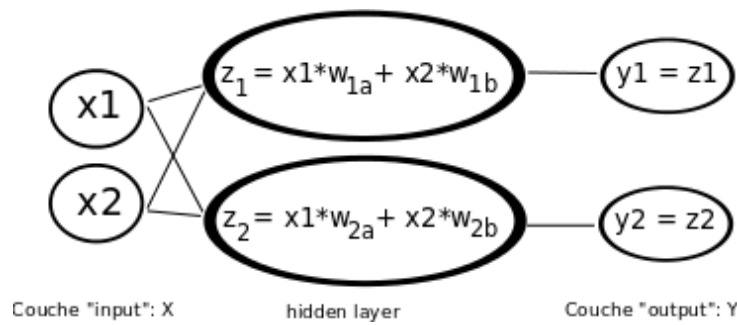


FIGURE 2.11 – Réseau neuronal : théorie 3

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2.9)$$

$$W = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix} \quad (2.10)$$

$$Z = WX = \begin{pmatrix} w_{1a}, w_{1b} \\ w_{2a}, w_{2b} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix} \quad (2.11)$$

$$Y = Z = \begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix} \quad (2.12)$$

Bien entendu, si l'on souhaite n'avoir qu'une valeur en sortie, il convient de modifier le neurone de sortie de sorte à ce qu'il fasse lui aussi un calcul dont le résultat n'a qu'une matrice de 1x1. La mécanique de base reste néanmoins la même : utiliser une version matricielle pour décrire les couches et interactions entre nos neurones. De la même manière, il est possible d'ajouter plusieurs couches cachées en représentant les matrices de la bonne manière afin d'effectuer un calcul de prédiction plus complexe.

Fonction d'activation

Il reste encore une notion à ajouter aux réseaux neuronaux vus jusqu'ici. Les réseaux neuronaux ont été directement inspirés des vrais réseaux de neurones et de la manière dont ils fonctionnent.

En termes biologiques, un neurone fait suivre une information (valeur) uniquement si le résultat de son "calcul" est supérieur à un certain seuil. En d'autres termes, un neurone ne renvoie une information que si il a suffisamment été excité.

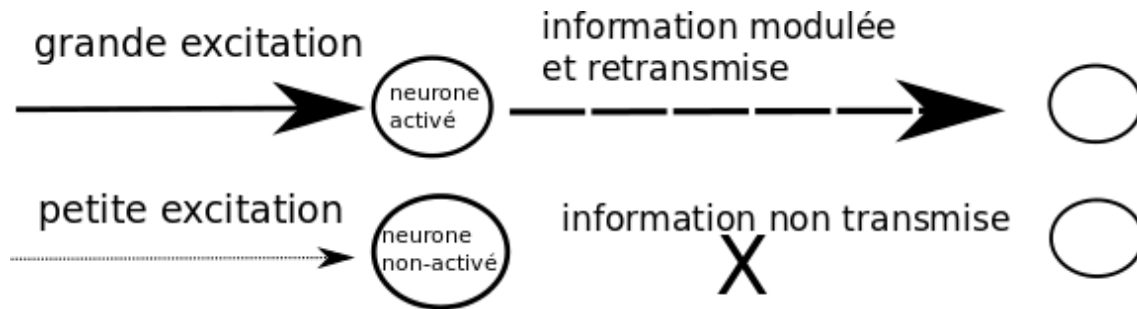


FIGURE 2.12 – Activation d'un neurone : représentation biologique

Ce seuil d'activation permet aux réseaux de neurones biologiques de transmettre une information d'une manière régulée en fonction des seuils (poids) de chaque neurone. Cela permet aux réseaux de neurones biologiques d'accomplir des calculs complexes tels que la nature nous l'a montré. La notion de seuil d'excitation implique que l'intensité de chaque information transmise entre neurones doit rester comprise au sein d'un seuil biologiquement acceptable. En d'autre termes, l'intensité de l'information transmise ne peut augmenter de neurones en neurones jusqu'à atteindre une valeur trop grande, mais doit garder une intensité telle qu'un neurone puisse stopper l'information ou au contraire la retransmettre. Mathématiquement parlant, l'idée serait de garder des valeurs de sorties comprises entre $[0,1]$.

D'un point de vue mathématique, les opérations effectuées par les réseaux neuronaux que nous avons vues sont uniquement linéaires (multiplications de matrices). Afin d'effectuer des prédictions complexes, il convient d'ajouter une notion "non-linéaire" aux opérations effectuées. Aussi, afin de pouvoir réguler facilement les poids de chaque neurone lors de la phase d'apprentissage, il convient de faire en sorte que chaque neurone renvoie une valeur de sortie comprise entre $[0,1]$.

Ainsi, chaque neurone passe son résultat dans une fonction non linéaire $f(x) -> [0,1]$.

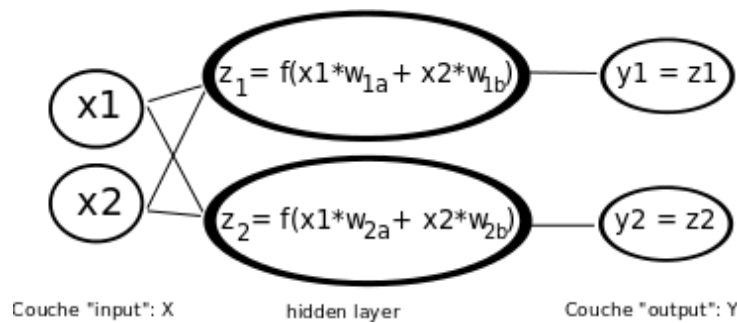


FIGURE 2.13 – Activation d'un neurone : représentation biologique

L'exemple ci-dessus correspond donc à

$$Z = f(WX) = f\left(\begin{pmatrix} w_{1a} & w_{1b} \\ w_{2a} & w_{2b} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = f\left(\begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}\right) \quad (2.13)$$

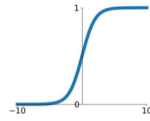
$$Y = Z = f\left(\begin{pmatrix} w_{1a}x_1 + w_{1b}x_2 \\ w_{2a}x_1 + w_{2b}x_2 \end{pmatrix}\right) \quad (2.14)$$

Voici les principales fonctions d'activations utilisées. Certaines se sont vue efficaces, même si elles ne renvoient pas une valeur comprise entre $[0, 1]$

Activation Functions

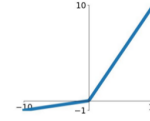
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



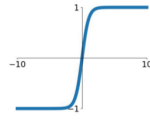
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

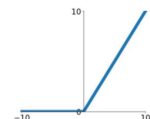


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

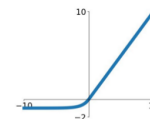


FIGURE 2.14 – Fonctions d'activation

Pour terminer, le fait d'utiliser des fonctions d'activations peut comporter un problème : si un neurone émet zéro à sa sortie, il y a de fortes chances pour que le neurone suivant émette aussi 0 à sa sortie. On peut utiliser un "biais" permettant d'inhiber / stimuler l'activation d'un neurone :

$$Z = f(WX + b)$$

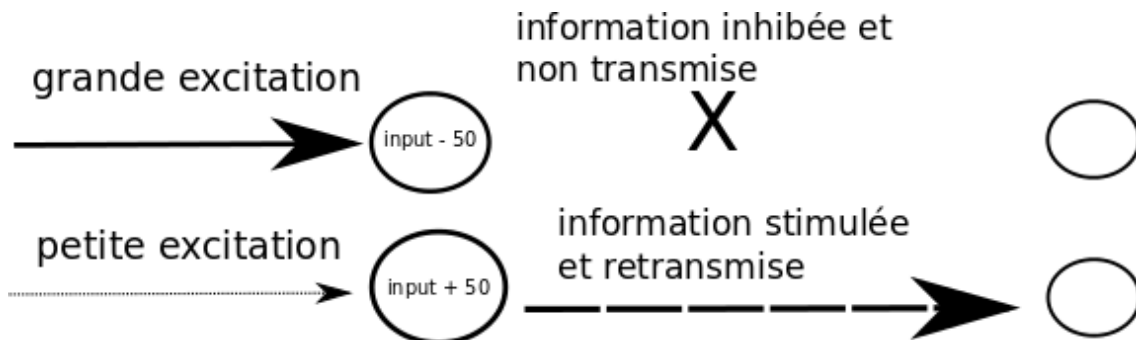
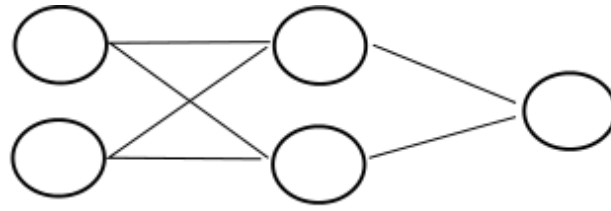


FIGURE 2.15 – Biais

Voici deux derniers exemples de réseaux neuronaux définis mathématiquement selon la théorie vue jusqu'ici :



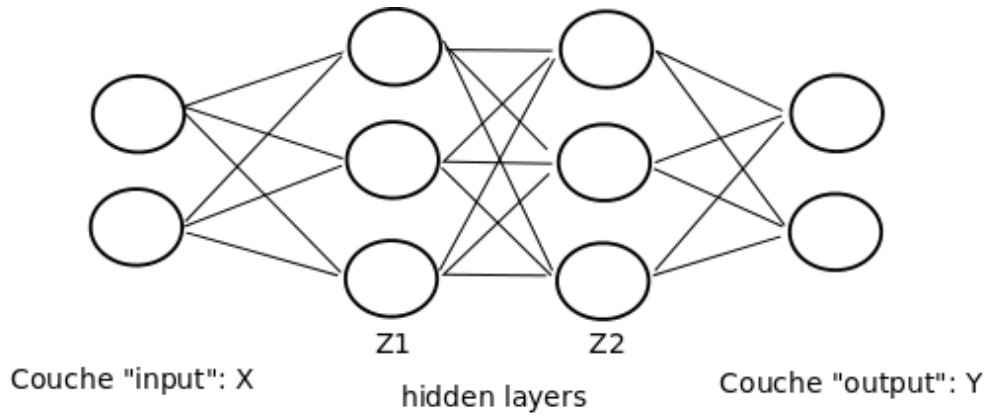
Couche "input": X hidden layer Z Couche "output": Y

FIGURE 2.16 – Réseau neuronal à 1 couche cachée de 2 neurones

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2.15)$$

$$Z = f(W_Z X + b_Z) = f\left(\begin{pmatrix} w_{z1a} & w_{z1b} \\ w_{z2a} & w_{z2b} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_{z1} \\ b_{z2} \end{pmatrix}\right) = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \quad (2.16)$$

$$Y = f_2(W_Y Z + b_Y) = f_2\left(\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} (w_{ya}, w_{yb}) + b_y\right) = (Y) \quad (2.17)$$



Couche "input": X Z1 Z2 hidden layers Couche "output": Y

FIGURE 2.17 – Réseau neuronal à 2 couches cachées de 3 neurones

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2.18)$$

$$Z1 = f(W_{z1} X + b_{z1}) = \begin{pmatrix} z1_1 \\ z1_2 \\ z1_3 \end{pmatrix} \quad (2.19)$$

$$Z2 = f_2(W_{z2} Z1 + b_{z2}) = \begin{pmatrix} z2_1 \\ z2_2 \\ z2_3 \end{pmatrix} \quad (2.20)$$

$$Y = f_3(W_Y Z2 + b_y) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \quad (2.21)$$

2.4.4 Apprentissage

Back-propagation

Nous avons vu comment un neurone peut prédire un résultat Y avec des valeurs X comme entrée. Au début, tous les poids W et biais b de chaque neurone sont initialisés aléatoirement. Le neurone va donc prédire un résultat aléatoire.

L'idée de l'apprentissage est la suivante : on donne une valeur d'entrée X , dont on connaît le résultat correct de la prédiction (target T). Le neurone va faire sa prédiction et sortir une valeur Y . Il faut alors ajuster chaque poids de sorte à réduire l'erreur $|Y - T|$. L'erreur peut être représentée de plusieurs manières différentes, comme par exemple $(Y - T)^2$. On représente l'erreur par une fonction $J(Y, T)$ à choix. La fonction s'appelle le "cost function". Le "phénomène" d'ajuster les poids en fonction de Y et T s'appelle *back-propagation*, vu qu'un calcul va s'effectuer de la sortie vers l'intérieur du réseau de neurones.

Nous pourrions imaginer d'ajuster les poids de la manière suivante : $W = W + J(Y, T)$. Si l'erreur est grande, le poids sera "grandement" ajusté. Si l'erreur est petite, le poids sera ajusté qu'un tout petit peu. Cependant, la fonction coût $J(Y, T)$ étant positive, il faut trouver une méthode plus subtile. D'autant plus que s'il y a plusieurs couches cachées (Z_1, Z_2), il faudrait pouvoir ajuster les poids de Z_1 uniquement en fonction de l'erreur causée par les poids de Z_1 , et les poids de Z_2 uniquement en fonction de l'erreur causée par les poids de Z_2 .

La méthode utilisée pour ajuster les poids est d'utiliser la dérivée de $J(Y, T)$. En dérivant vis à vis de chaque couche, on peut trouver l'erreur liée à chaque couche. Par conséquent, pour ajuster les poids de Z_1 , on effectue l'ajustement suivant : $W_{z1} = W_{z1} + \frac{dJ(Y, T)}{dW_{z1}}$. De même, pour ajuster les poids de Z_2 , on effectue l'ajustement suivant : $W_{z2} = W_{z2} + \frac{dJ(Y, T)}{dW_{z2}}$.

Gradient Descent

Une dernière subtilité doit être ajoutée à la manière dont les poids sont ajustés. Afin d'éviter d'ajuster les poids de manière trop brutale, on ajuste (à nouveau) par une fraction α (learning rate). Ainsi, la méthode utilisée devient la suivante, et s'appelle la méthode du *gradient descent* :

$$W = W + \alpha \frac{dJ(Y, T)}{dW}$$

Pour donner un exemple de calcul, prenons un cas simple, un réseau avec une couche d'entrée et une couche de sortie. (Pas de couche cachées).

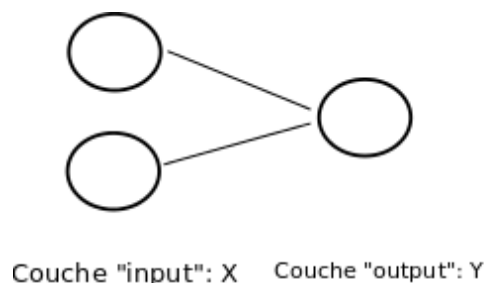


FIGURE 2.18 – Réseau neuronal à 1 couche d'entrée et 1 couche de sortie

$$Y = f(W_Y X + b_Y) \quad (2.22)$$

$$\frac{dJ}{dW_Y} = \frac{dJ}{dY} \frac{dY}{dW_Y} = \frac{dJ}{dY} \frac{dY}{df} \frac{df}{dW_Y} \quad (2.23)$$

Grâce au fait que nous travaillons avec des dérivées, l'on peut ainsi développer suffisamment pour pouvoir effectuer le calcul de $\frac{dJ}{dW_Y}$

Prenons maintenant un exemple avec une couche d'entrée, une couche cachée et une couche de sortie :

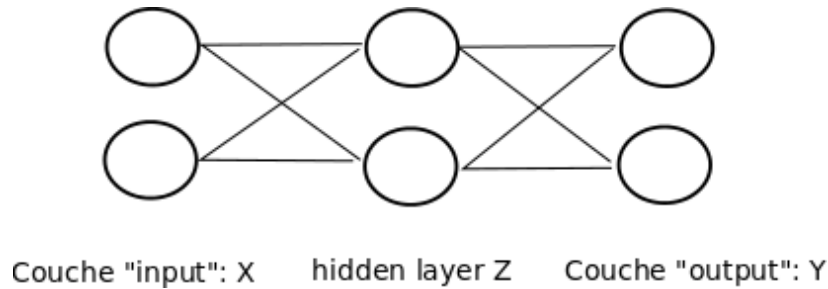


FIGURE 2.19 – Réseau neuronal à 1 couche d'entrée, 1 couche cachée et 1 couche de sortie

$$Z = f(W_Z X + b_Z) \quad (2.24)$$

$$Y = f_2(W_Y Z + b_Y) \quad (2.25)$$

À présent, il y a deux matrices de poids à ajuster : W_Y et W_Z . Il faut calculer leur dérivées séparément :

$$\frac{dJ}{dW_Y} = \frac{dJ}{dY} \frac{dY}{dW_Y} = \frac{dJ}{dY} \frac{dY}{df_2} \frac{df_2}{dW_Y} \quad (2.26)$$

$$\frac{dJ}{dW_Z} = \frac{dJ}{dY} \frac{dY}{dW_Z} = \frac{dJ}{dY} \frac{dY}{df_2} \frac{df_2}{dW_Z} = \frac{dJ}{dY} \frac{dY}{df_2} \frac{df_2}{dZ} \frac{dZ}{dW_Z} = \frac{dJ}{dY} \frac{dY}{df_2} \frac{df_2}{dZ} \frac{f_1}{dW_Z} \quad (2.27)$$

Nous constatons (heureusement) qu'un certain schéma se répète dans le calcul des $\frac{dJ}{W_i}$, permettant ainsi le calcul des dérivées automatiquement par programmation. Plusieurs bibliothèques fournissent cette possibilité. Pour ce travail de bachelor, la bibliothèque TensorFlow développée par Google a été utilisée.

Il est important de signaler qu'un réseau de neurone ne peut s'ajuster qu'avec un seul X et T. Il faut bien entendu répéter le calcul de back-propagation un grand nombre de fois afin d'arriver à des valeurs des poids permettant un pourcentage de prédiction correct. Une fois le réseau de neurones bien entraîné (après avoir répété la back-propagation de nombreuses fois), on peut ensuite fournir une valeur X sans connaître son résultat correct, mais supposant raisonnablement que la prédiction sera correcte.

2.4.5 Difficultés

Le champ d'application des réseaux neuronaux est vaste

Nous tenons à préciser ici que la théorie que nous avons vue dans ce chapitre ne constitue que les notions essentielles permettant appréhender correctement la suite de ce rapport. De nombreux modèles de réseaux neuronaux diffèrent selon le champ d'application. Pour la reconnaissance d'images par exemple, il faut rajouter des couches de "convolution" afin de traiter les images correctement avant de faire suivre l'information dans un réseau de neurones tel que nous l'avons vu. Ces modèles de réseaux de neurones s'appellent les "convolutional neural networks".

De même, pour la reconnaissance de voix et prédire la phrase prononcée par un utilisateur, il faut rajouter la notion de réseaux neuronaux récurrents, dont des sorties de certaines couches de neurones sont les entrées de couches positionnées en amont (reccurent neurals networks).

Données fournies à l'apprentissage

Nous pouvons constater que l'efficacité d'apprentissage d'un réseau de neurones dépend fortement des données qui lui sont transmises en phase d'apprentissage. Pour donner un exemple concret, voici l'histoire d'un groupe de programmeurs ayant développé un réseau de neurones capables de décerner la différence entre un chien et un loup. Après de nombreuses sessions d'apprentissages, le réseau ne pouvait discerner un chien que lorsqu'il ne se trouvait pas entouré de neige. De même, seuls les loups étaient reconnus lorsqu'ils étaient entourés de neige. Après des semaines d'études, les développeurs se sont rendus compte que la majorité des images de loups données lors de l'apprentissage était une image d'un loup dans la neige. Les développeurs se sont ainsi rendus compte que le réseau de neurone avait appris à reconnaître la neige pour discerner un chien d'un loup.

Il convient donc de faire attention à fournir des données suffisamment diversifiées lors de l'apprentissage afin que celui-ci soit le plus performant possible.

2.5 Algorithme Deep Q-Network (DQN)

Dans la section "Algorithme Q-Learning", nous avons vu une méthode pour enseigner à un agent comment apprendre à évoluer dans un environnement donné en explorant les actions possibles. La méthode s'avère efficace lorsque l'environnement ne possède qu'une petite quantité d'états / actions possibles. Dans ces petits environnements, l'agent peut garder une copie de chaque états et actions visitées afin d'estimer correctement la fonction $Q(s,a)$ de sorte à apprendre à savoir quelle action effectuer pour un état donné, dans le but d'effectuer un score maximal lors d'un épisode.

Lorsque l'environnement devient plus complexe, l'espace état / actions devient beaucoup plus grand et la tâche devient presque impossible étant donné la taille du tableau $Q(s,a)$ à garder en mémoire. Les réseaux neuronaux constituent une solution adéquate pour effectuer une estimation complexe d'un problème donné. L'enjeu de l'algorithme *Deep Q-learning* est d'utiliser les réseaux neuronaux pour effectuer une estimation suffisante de la fonction $Q(s,a)$ sans devoir connaître / explorer l'espace complet état / actions d'un environnement donné.

Un réseau neuronal peut être appelé "Deep neural network" lorsque un réseau neuronal comprend de nombreuses couches cachées ; la notion de "Deep" dans "Deep Q-learning" correspond donc à appliquer la méthode Q-learning avec un réseau de neurones à plusieurs couches cachées (hidden layers).

Dans le cadre de ce travail, la valeur des actions a en fonction de l'état s $Q(s,a)$ est définie par la prédiction d'un réseau neuronal. Nous fournissons un vecteur d'entrée $X = s$ représentant l'état de l'environnement. Par exemple :

```
X = s = [agent.x, agent.y, enemy.x, enemy.y]
```

Le résultat du réseau neuronal est une estimation de $Q(s,a)$ pour chaque action possible lors de l'état s . La taille du vecteur de sortie correspond au nombre total d'actions possibles lors de l'état s . L'action choisie sera l'index correspondant à la plus haute valeur rencontrée.

```
Y = predicted_values = [value_a1, value_a2, value_a3, va4]  
action = argmax[predicted_values]
```

Une action est donc un nombre entier compris entre $[0, nb_actions_possibles]$

2.5.1 Mise à jour de la fonction $Q(s,a)$: ajustements à effectuer

Afin de faire fonctionner un agent avec l'algorithme Q-Learning en utilisant un réseau neuronal pour estimer la valeur $Q(s,a)$, il y a deux ajustements principaux à effectuer pour mettre à jour

la fonction $Q(s,a)$.

Experience Replay

Comme évoqué lors du chapitre sur les réseaux neuronaux (difficultés : données fournies à l'apprentissage), afin qu'un réseau neuronal puisse apprendre correctement à estimer son sujet, les valeurs d'entrées X lors de la phase d'apprentissage doivent être suffisamment homogènes afin que l'apprentissage soit le plus complet possible. Or, l'idée du Q -Learning est d'ajuster $Q(s,a)$ à chaque état parcouru de l'agent. Imaginons que l'agent commence toujours un épisode dans un même état initial :

Si l'agent met à jour son réseau neuronal ($Q(s,a)$) à chaque état qu'il rencontre, l'apprentissage de $Q(s,a)$ sera biaisé dans le sens où il va renforcer l'ajustement de $Q(s,a)$ sur les premiers états (récurrents à chaque nouvel épisode), et laissera les états suivants (sporadiques) de côté pour son apprentissage. Il en résultera par exemple que l'agent connaisse les 20 premières actions à faire, mais sera incapable d'estimer correctement la suite des actions à faire.

Pour palier à ceci, l'idée est d'enregistrer tous les états / actions / récompenses dans une liste d'échantillons dont la longueur est définie par le programmeur. D'ici à ce que la liste d'échantillons ne soit pas pleine, l'agent ne met pas à jour la fonction $Q(s,a)$. Si la liste est pleine, les échantillons (états / actions / récompenses) les plus vieux sont remplacés par les nouveaux échantillons rencontrés.

Une fois que l'on dispose d'une liste d'échantillons suffisamment grande, l'agent met à jour la fonction $Q(s,a)$ en choisissant **aléatoirement** une **petite** partie de ces échantillons (batch). L'on peut par exemple mettre à jour la fonction $Q(s,a)$ à chaque nouvel épisode, pour autant que la liste soit suffisamment grande afin de permettre de choisir **aléatoirement** un groupe d'échantillons de la liste d'échantillons. L'on peut ainsi supposer que l'agent puisse mettre à jour la fonction $Q(s,a)$ parmi un choix d'échantillons aléatoires et suffisamment homogène.

Apprentissage du réseau neuronal : estimation Y et target T

L'autre inconvénient majeur d'utiliser un réseau neuronal est que pour l'entraîner, il nous faut pour chaque entrées X , connaître les solutions T . En effet, le réseau neuronal ajuste ses poids lors de l'apprentissage à l'aide de la fonction $J(Y, T)$. Or, nous ne connaissons pas T étant donné que l'idée de l'algorithme *Q-Learning* est d'explorer sans connaître son environnement. Comment choisir T étant donné que la meilleure estimation de $Q(s,a)$ est définie par la prédiction de $Q(s,a)$ par QNetwork ?

En d'autres termes, si nous utilisons pour chaque X , une solution $T = Y$, le taux d'erreur sera de 0 et l'agent n'apprendra rien étant donné que la solution T correspond à la prédiction Y .

Une méthode qui a fait ses preuves est d'avoir deux réseaux neuronaux. Le réseau pour estimer $Q(s,a)$ (QNetwork), et un deuxième réseau pour estimer les solutions T (TNetwork). TNetwork est une copie conforme à QNetwork, à la seule différence est que TNetwork est copié de QNetwork toutes les N actions de l'agent (ou tous les N épisodes). À l'action "0" de l'agent, QNetwork et TNetwork sont exactement les mêmes et l'apprentissage de $Q(s,a)$ ne se fera pas. En revanche, pour les actions suivantes, QNetwork sera mis à jour et TNetwork reste le même, continuant à prédire ses valeurs T .

Ainsi, $Q(s,a)$ pourra être mise à jour étant donné que les valeurs $Q(s,a) = Y$ de QNetwork(s) et les valeurs de TNetwork(s) = T ne fourniront pas les mêmes valeurs.

2.5.2 Pseudo-code de l'algorithme Deep Q-Network

Partie principale

Voici un pseudo code effectuant l'algorithme Deep Q-Network :

```
#####
#MAIN
#####
# phase d'apprentissage
world = new World()
max_episodes = 100000
gamma = 0.9 # discount factor
epsilon = 0.1 # probabilité d'effectuer une action aléatoire
alpha = 0.1 # learning rate
QNetwork = NeuralNetwork()
TNetwork = NeuralNetwork(QNetwork)
experience = []
min_experience_size = 10000
cp_target_period = 10000
batch_size = 32 # taille de l'échantillon utilisé pour le back-propagation

for i in range(max_episodes) :
    state = world.reset()
    while not world.game_over() :
        action = choose_action(state, epsilon, QNetwork)
        reward, next_state = world.do_action(state, action)

        experience.append((state, next_state, action, reward, game_over))
        if experience.length >= min_experience_size :
            train(QNetwork, TNetwork, experience, batch_size, alpha)

        if i % cp_target_period == 0 :
            TNetwork.copy_from(QNetwork)

    state = next_state

# phase une fois l'apprentissage effectué
state = world.reset()
while not world.game_over() :
    action = choose_action(state, epsilon, QNetwork)
    state, _ = world.do_action(action)
```

Fonction choose_action()

Voici le pseudo code de la fonction choose_action() :

```
def choose_action(state, epsilon, QNetwork) :  
    if random() < epsilon :  
        action = random_action()  
    else :  
        prediction = QNetwork.predict(state)  
        action = argmax(prediction)  
    return action
```

Un état est représenté par un vecteur contenant les informations suffisantes pour illustrer l'état du jeu.

```
state = [agent.x, agent.y, ennemy.x, ennemy.y]
```

La prédiction du réseau QNetwork prend en entrée le vecteur state, et retourne en sortie un vecteur contenant les récompenses (rewards) estimées pour chacune des actions possibles. La taille du vecteur de sortie correspond donc au nombre d'actions possibles pour l'agent.

```
prediction = [ra_1, ra_2, ra_3, ra_4, ra_5] = QNetwork.predict(state)
```

La fonction choose_action() retourne l'index de la plus grande récompense estimée. Dans ce travail de bachelor, cet index équivaut à l'action choisie. Par exemple :

```
prediction = [1.5, 12, -2, 1.4, 0.56] = QNetwork.predict(state)  
action = 1 = argmax(prediction)
```

Fonction train()

```
def train(QNetwork, TNetwork, experience, batch_size, alpha) :  
    inputs = []  
    choosen_actions = []  
    targets = []  
  
    batch = experience.pop_batch(batch_size)  
    for sample in batch :  
        s1, s2, w, game_over, action = sample.extract()  
  
        inputs.append(s1)  
  
        predicted_values = QNetwork.predict(s1)  
        predicted_values = 0 for all values except for predicted_values[action]  
        choosen_actions.append(predicted_values)  
  
        target_Q_values_s2 = TNetwork.predict(s2)  
        action_value = target_Q_values_s2[action]  
        target = zeros(nb_possible_actions)  
        target[action] = r if game_over else r + gamma * action_value  
        targets.append(target)  
  
    QNetwork.back_propagation(inputs, choosen_actions, targets, alpha)
```

Pour entraîner le réseau QNetwork, celui-ci a besoin d'un input (vecteur représentant l'état de l'environnement), d'un vecteur représentant les valeurs qu'il a estimées (action), et les valeurs correctes qu'il aurait retourner.

```
state = [agent.x, agent.y, ennemy.x, ennemy.y]
predicted_values = [1.5, 12, -2, 1.4, 0.56]
correct_values = target = [2.1, 6, -2.1, 3.5, 2.21]
```

Les valeurs correctes ne sont connues uniquement pour les actions qui ont été effectuées ; pour chaque action effectuée, nous connaissons sa récompense et l'état résultant de cette action. La valeur correcte $Q(s_{visit}, a_{choisie})$ est calculée selon l'algorithme *Q-Learning*, en utilisant le réseau TNetwork :

```
target_Q_values_s2 = TNetwork.predict(s2)
action_value = target_Q_values_s2[action]
correct_value = target = r if game_over else r + gamma * action_value
```

Or, un réseau neuronal a besoin pour s'entraîner (back-propagation) de vecteurs de la même taille que les vecteurs qu'il a en sortie. Pour entraîner le réseau uniquement sur l'action effectuée, nous créons le vecteur T (target) composé uniquement de zéros, sauf là où le réseau doit être ajusté. De même, le vecteur *predicted_values* est mis à zéro partout sauf là où l'action a été effectuée. Dans cet exemple, l'action effectuée est l'action = 3.

```
# avant modification
state = [agent.x, agent.y, ennemy.x, ennemy.y] = X
predicted_values = [1.5, 12, -2, 1.4, 0.56] = Y
correct_values = target = [2.1, 6, -2.1, 3.5, 2.21] = T

# après modification
state = [agent.x, agent.y, ennemy.x, ennemy.y] = X
predicted_values = [0, 0, 0, 1.4, 0] = Y
correct_values = target = [0, 0, 0, 3.5, 0] = T
```

De cette manière, lors du "back-propagation", l'ajustement des poids du QNetwork ne sera mis à jour que là où la dérivée $\frac{dJ(Y,T)}{dW} \neq 0$. Y = predicted_values, et T = target.

$$W = W + \alpha \frac{dJ(Y,T)}{dW}$$

Enfin, les bibliothèques permettent d'entraîner un réseau neuronal en lui fournissant une matrice d'input (X), *predicted_values* (Y) et *target* (T) (à la place d'un vecteur à la fois). Le principe reste le même, mais cela permet d'effectuer les calculs beaucoup plus rapidement qu'en fournissant à chaque fois qu'un seul vecteur. C'est la raison pour laquelle le pseudo code entraîne le QNetwork avec des matrices (X = inputs), (Y = predicted_values) et (T = targets).

Chapitre 3

Implémentation

3.1 Fonctionnement

3.1.1 Vue d'ensemble : choix d'une action

Les actions possibles dans l'environnement implémenté sont d'avancer dans une direction des points cardinaux (N, NE, E, SE, S, SO, O, NE), ou ne rien faire. Le nombre d'actions possibles équivaut donc à neuf.

Comme indiqué dans l'introduction, l'apprentissage est divisé en trois parties. Dans un premier temps, l'agent doit pouvoir évoluer dans un monde qu'avec des ennemis, en utilisant un réseau neuronal dédié estimant la meilleure action à effectuer. Dans un deuxième temps, l'agent doit pouvoir évoluer uniquement avec de la nourriture utilisant un autre réseau neuronal dédié à attraper la nourriture. Enfin, l'agent doit pouvoir évoluer dans un monde "complet", avec ennemis et nourriture. Dans ce dernier cas, l'agent dispose de trois réseaux neuronaux : un pour choisir la meilleure action de sorte à éviter les ennemis, un deuxième estimant la meilleure action pour attraper la nourriture, et un troisième réseau neuronal estimant quel est le meilleur réseau neuronal à écouter : effectuer l'action évitant les ennemis, ou effectuer l'action pour attraper la nourriture.

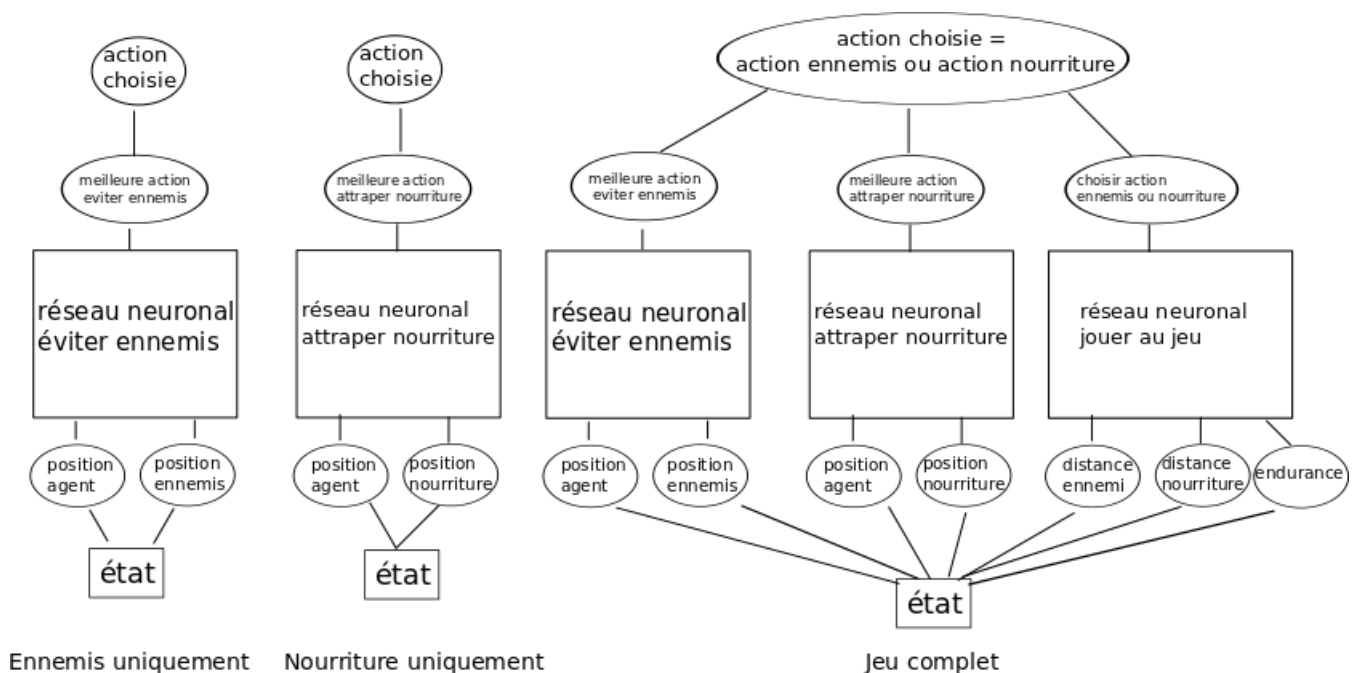


FIGURE 3.1 – Vue d'ensemble : choix d'une action

Il aurait été possible de n'utiliser qu'un grand réseau neuronal pour apprendre directement à jouer dans un monde complet, sans fractionner l'apprentissage. Il y a plusieurs raisons pour laquelle nous avons choisi d'utiliser plusieurs réseaux neuronaux / phases d'apprentissage : la première vient de la difficulté à choisir une fonction de récompense adéquate à ce que nous voulons enseigner à l'agent. L'apprentissage est plus efficace lorsque la fonction de récompense renforce le comportement de l'agent dans un contexte bien spécifique (éviter ennemis ou attraper la nourriture).

La deuxième raison vient du fait que plus un réseau neuronal est grand, moins l'apprentissage est rapide. Plus il y a d'inputs dans le réseau neuronal, plus ce dernier est grand. Il est donc préférable d'entraîner l'agent sur des problèmes spécifiques avant de l'entraîner à jouer de manière complète. Comme indiqué dans la figure ci-dessus, avec cette méthode, le réseau neuronal capable de jouer pas besoin de beaucoup d'inputs. Ce dernier ne connaît même pas la position de la nourriture, de l'agent ni des ennemis.

Enfin, une fois qu'un agent a appris une tâche spécifique, il n'y a pas besoin de le ré-entraîner (pour autant que le monde et les inputs concernés ne changent pas). Imaginons maintenant que nous ayons utilisé un seul grand réseau neuronal pour jouer, et que l'on veuille ajouter une nouvelle tâche à apprendre à l'agent. Il faudrait ré-entraîner le tout depuis zéro. En fractionnant l'apprentissage en tâches spécifiques, il y aurait seulement besoin d'entraîner le nouveau réseau concernant la nouvelle tâche, et le réseau capable de jouer.

3.1.2 Apprentissage

L'apprentissage de l'agent se fait à travers trois scripts qui durent entre 24 et 72 heures. Chaque script a pour but d'enseigner à l'agent une "compétence" : éviter les ennemis, attraper la nourriture ou apprendre à jongler entre les deux pour jouer au jeu complet. Chaque compétence est contenue dans un réseau neuronal QNetwork ; l'apprentissage correspond à modifier les poids de ces réseaux (back-propagation) de sorte à pouvoir estimer correctement la fonction $Q(s, a)$.

- **trainer_avoid_enemies.py** :
 1. charger un monde qu'avec des ennemis
 2. créer un nouveau réseau neuronal
 3. entraîner l'agent à éviter les ennemis
 4. enregistrer l'état du réseau neuronal lorsque l'agent a appris à éviter les ennemis puis quitter le script
- **trainer_fetch_object.py** :
 1. charger un monde qu'avec de la nourriture
 2. créer un nouveau réseau neuronal
 3. entraîner l'agent à attraper la nourriture
 4. enregistrer l'état du réseau neuronal lorsque l'agent a appris à attraper la nourriture puis quitter le script
- **trainer_play_game.py** :
 1. charger le monde complet (ennemis + nourriture)
 2. charger les deux réseaux neuronaux estimant correctement les actions pour éviter les ennemis et attraper la nourriture
 3. créer un nouveau réseau neuronal (qui apprendra à jongler entre les deux réseaux neuronaux chargés)
 4. entraîner l'agent à jouer au jeu complet
 5. enregistrer l'état du réseau neuronal lorsque l'agent a appris à jouer au jeu complet puis quitter le script

3.1.3 Regarder l'agent jouer

Trois scripts supplémentaires ont été créés permettant de visualiser en temps réel l'agent évoluer dans le monde voulu. Ces scripts chargent le monde et chargent l'agent en important les réseaux neuronaux adéquats à partir des sauvegardes contenues dans le dossier *saves*.

- `see_avoid_enemies_in_action.py`
- `see_fetch_object_in_action.py`
- `see_play_game_in_action.py`

3.1.4 Tester soi-même l'environnement

Afin de tester le monde dans lequel va évoluer l'agent, il est possible d'exécuter le fichier **World.py** afin de pouvoir jouer soi-même.

```
$ python world.py
```

3.1.5 Configuration

Le choix de tous les hyper-paramètres tels que la fonction de récompense, le discount factor γ , la taille de l'expérience, le learning rate α , etc. sont tous définis dans les scripts directement (et null part ailleurs). Cela facilite grandement lorsque qu'il s'agit de voir quels hyper-paramètres ont été utilisés lorsque un agent a spécialement réussi un apprentissage. De plus, à chaque fois que l'un script de type *trainer* est exécuté, une copie de celui-ci est enregistrée dans le dossier choisi pour les sauvegardes.

Dans cette sous-section, nous passons en revue le contenu des scripts définis plus haut.

Chargement de l'environnement

Le chargement de l'environnement se fait de la manière suivante en choisissant la configuration désirée. La fonction de récompense est définie à ce moment là : c'est elle qui retourne la récompense. Cette fonction est appelée par le monde (classe *World*) lorsque l'agent effectue une action dans celui-ci. A noter que pour les scripts de type `see_*_in_action.py`, la fonction récompense n'est pas utilisée étant donné que l'agent n'est pas chargé pour apprendre. L'exemple suivant est importé du script **trainer_fetch_object.py**.

```
# create the world
def reward_function(world):
    if world.game_over:
        return - 5
    max_distance = 73
    return (1 - Direction.distance(world.agent, world.food) / max_distance)
world_config = {
    'food' : True,
    'enemies' : False,
    'print_reward' : False,
    'reward_function' : reward_function
}
world = World(world_config)
world.reset()
```


Choix du dossier de sauvegarde et création de graphiques Tensorboard

Les réseaux neuronaux sont une boîte noire difficile à déboguer. Afin de visualiser l'apprentissage par des graphiques, nous utilisons l'outil *Tensorboard* disponible dans la librairie *Tensorflow*. La création des graphiques et le choix du dossier dans lequel vont être enregistrés les réseaux neuronaux (compétences) se définissent de la manière suivante, à travers la classe statique **Debug** :

```
Debug.USE_TENSORBOARD = True
Debug.SAVE_MAIN_FILE = True
Debug.SAVE_FOLDER = '../tmp_saves/avoid_enemies'
Debug.OUTPUT_TO_TENSORBOARD_EVERY_N_EPISODES = 5000
```

Création d'un réseau neuronal

Comme vu dans la figure 3.1, un réseau neuronal doit pouvoir être configuré de sorte à ce que les entrées et sorties puissent être définies manuellement au cas par cas. Au lieu de donner l'état du monde tel quel en entrée, nous passons par une fonction *input_adapter()* qui retourne le vecteur X désiré. De même, l'on peut définir une fonction *output_adapter()* afin de convertir la sortie d'un réseau neuronal en une action particulière. La fonction *output_adapter()* est facultative.

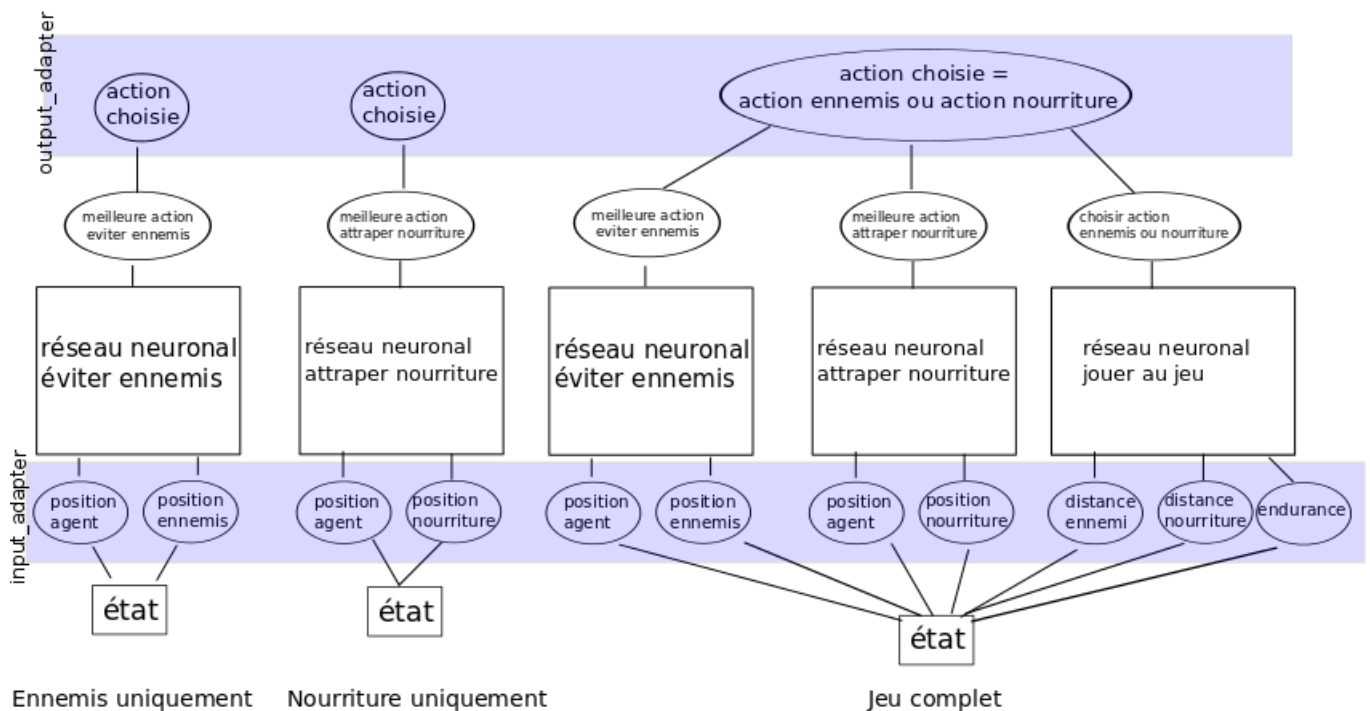


FIGURE 3.2 – Input et Output adapters

La création d'un réseau neuronal se crée de la manière suivante : nous créons d'abord un modèle contenant l'architecture du réseau neuronal, le nom de celui-ci ainsi que le *learning_rate* α .

```
fetch_object_model = Model (session , name, nb_input, learning_rate,
    [[size_layer1 , activation_function_layer1],
    [size_layer2 , activation_function_layer2],
    [ size_output_layer , activation_function_layer3]]
)
```

Puis on crée le réseau neuronal à partir de l'architecture définie plus haut. Voici un exemple tiré du script **trainer_fetch_object.py**. La variable *bus* contiendra les états *s1* et *s2* nécessaires pour définir la fonction *input_adapter*

```
# create the neural network that will learn to fetch the food
fetch_object_model = Model(session, 'fetch_object', 4, 1e-1,
    [[40, 'relu'],
     [40, 'relu'],
     [Action.NB_POSSIBLE_MOVE_ACTION, 'linear']]
)
# fetch_object_model = ImportModel(session, Debug.SAVE_FOLDER, 'fetch_object')
def fetch_object_input_adapter(bus, next_state=False) :
    index = 'next_state' if next_state else 'state'
    agent_position = bus[index].get_agent_position_layer()
    food_position = bus[index].get_food_position_layer()
    return [np.array([agent_position, food_position]).flatten()]

fetch_object_network = Network(
    fetch_object_model,
    fetch_object_input_adapter,
    True, # continue_exploration
    True, # is_training (has to be trained during process)
    None, # add experience hook
    None # output_adapter
)
```

Création de l'agent

Maintenant que le monde et un réseau neuronal est créé, l'on peut créer l'agent. Nous définissons le reste des hyper-paramètres tels que la fonction epsilon ϵ , la taille d'un batch (taille de l'échantillon utilisé pour l'apprentissage), ou du discount factor γ . Comme il peut y avoir plusieurs réseaux neuronaux, il faut définir quel est le réseau neuronal de sortie. Dans cet exemple, la valeur epsilon ϵ reste stable durant tout l'entraînement.

```
# create agent and his hyperparameters (config)
epsilon = Epsilon(0.1)
def update_epsilon(epsilon) :
    epsilon.value = epsilon.value
epsilon.set_epsilon_function(update_epsilon)

agent_config = {}
agent_config['epsilon'] = epsilon
agent_config['networks'] = [fetch_object_network]
agent_config['output_network'] = fetch_object_network
agent_config['copy_target_period'] = 10000
agent_config['min_experience_size'] = 50000
agent_config['max_experience_size'] = 400000
agent_config['batch_size'] = 32
agent_config['gamma'] = 0.5

agent = Agent(agent_config)
```

3.2 Architecture

3.2.1 Diagramme de classes

Nous avons vu jusqu'ici le contenu des scripts permettant l'apprentissage de l'agent et des scripts permettant la visualisation de l'agent en train d'évoluer dans le monde. Voici un diagramme des classes simplifié des classes utilisées par ces scripts.

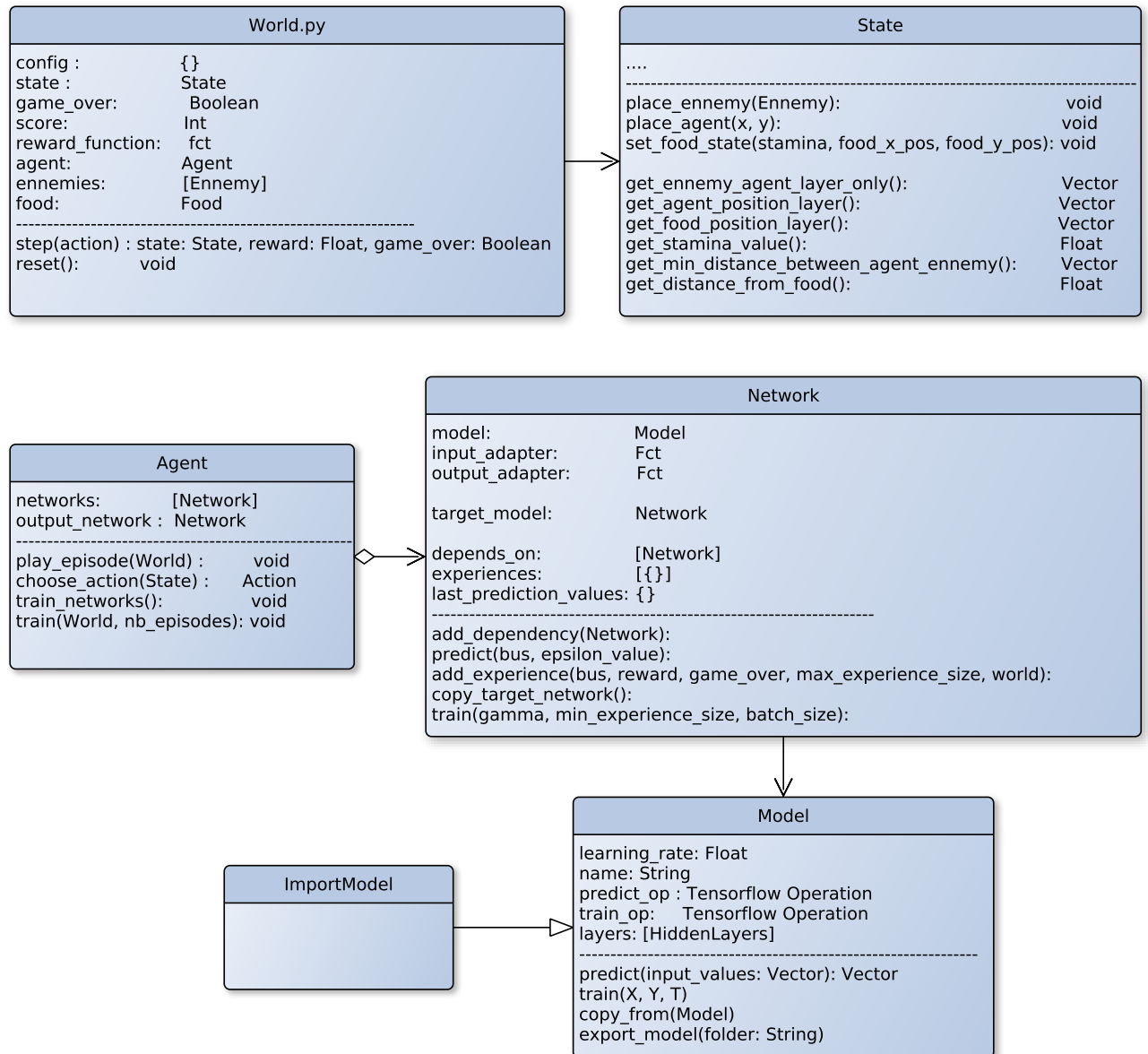


FIGURE 3.3 – Diagramme des classes

3.2.2 Méthodes clés

Apprentissage

Les scripts de type *trainer_*.py* ont pour but de créer un agent et de lui enseigner une compétence de type "éviter les ennemis", "attraper la nourriture" ou "jouer au jeu complet". Une fois l'environnement chargé et les réseaux neuronaux créés, l'agent apprend par lui-même à évoluer dans l'environnement défini à travers la méthode *Agent.train()*, et est résumée par le code suivant :

```
def train(self, world, nb_episodes) :  
    for i in range(nb_episodes) :  
        self.play_episode(world)
```

La méthode *Agent.play_episode()* est quant à elle résumée par ceci :

```
def play_episode(self, world, max_episode_steps=None) :  
    current_state = world.reset()  
  
    episode_done = False  
    while not episode_done :  
        action = self.choose_action(current_state)  
        next_state, reward, game_over, world_debug = world.step(action)  
  
        if self.nb_steps_played % self.copy_target_period == 0 :  
            self.copy_target_networks()  
  
        self.add_experience(next_state, reward, game_over, world)  
        self.train_networks()  
  
        if game_over :  
            episode_done = True  
  
        current_state = next_state  
        self.epsilon.update_epsilon()  
        self.nb_steps_played += 1
```

Choix d'une action

Comme un agent peut contenir plusieurs réseaux neuronaux, il convient de choisir correctement comment choisir son action. En effet, un réseau neuronal tel que le réseau *play_game*, dépend des réseaux neuronaux *avoid_enemies* et *fetch_object* (nourriture) avant d'indiquer l'action à effectuer. La méthode *Agent.choose_action()* est implémentée de la sorte :

```
def choose_action(self, state) :
    self.bus['state'] = state

    nb_networks_that_predicted = 0
    while nb_networks_that_predicted != len(self.networks) :
        for network in self.networks :
            if len(network.depends_on) is 0 \
            or all(dependency.prediction_done for dependency in network.depends_on) :
                network.predict(self.bus, self.epsilon.value)
                nb_networks_that_predicted += 1

    return self.output_network.get_last_action()
```

Chapitre 4

Résultats

Concevoir et implémenter l'architecture est une chose. Faire en sorte que l'agent puisse apprendre correctement en est une autre. Il a beaucoup d'hyper-paramètres à régler pour obtenir un résultat convenable. Au début du travail de bachelor, il fallait exécuter le script *trainer_*.py* pendant une durée allant jusqu'à 14 jours avant de pouvoir affirmer si l'apprentissage se faisait ou non. De plus, étant donné qu'il y a une part de hasard dans l'apprentissage (valeurs initiales des poids W des réseaux neuronaux, position des ennemis et fonction *epsilon*, l'apprentissage de l'agent peut varier d'une session à l'autre, même en gardant les mêmes hyper-paramètres.

Après de nombreux essais et de tests, nous présentons dans ce chapitre les paramètres principaux ayant majoritairement contribué à ce que l'agent puisse apprendre à évoluer correctement dans son environnement.

4.1 Fonction récompense

4.1.1 Problème rencontré

Un premier paramètre déterminant est le choix de la fonction récompense. Au début, afin d'entraîner l'agent à éviter les ennemis, la fonction récompense était définie comme suit :

```
def reward_function(world) :  
    if world.game_over :  
        return - 1  
    else :  
        return 1
```

L'idée était simplement de récompenser d'un point à chaque fois que l'agent effectue une action sans tomber sur un ennemi. S'il tombe sur un ennemi, l'agent reçoit une récompense de -1 et l'épisode était terminé.

En constatant que l'apprentissage ne se faisait pas, (que le score de l'agent dans le jeu était constamment médiocre), nous avons utilisé *Tensorboard* afin de tracer l'histogramme des actions effectuées par l'agent, et l'histogramme de l'évolution des poids des réseaux neuronaux. (cf figure suivante).

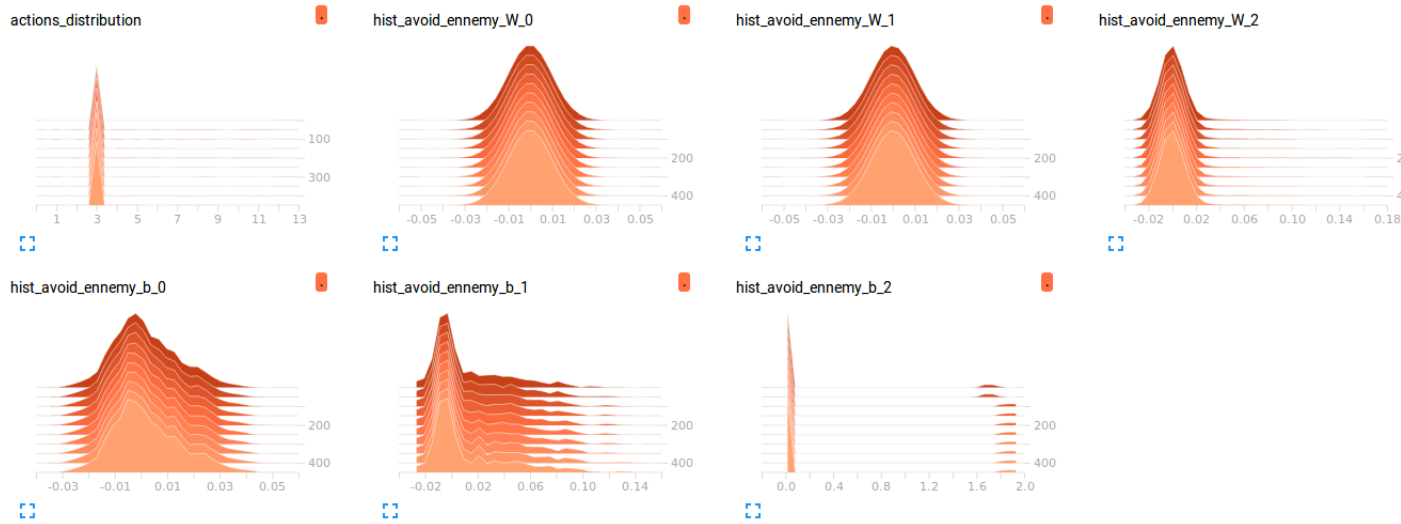


FIGURE 4.1 – Histogrammes des actions et poids avec une mauvaise fonction récompense

Nous avons pu en déduire que l'agent n'effectuait qu'une seule action (graphique en haut à gauche), et que l'apprentissage ne se faisait pas car l'histogramme des poids (autres graphiques) n'évoluaient pas.

4.1.2 Explication

Début de l'agent

A sa création, le réseau neuronal possède des poids d'une valeur répartie aléatoirement entre $[-0.5, 0.5]$. Prenons par exemple un réseau estimant la valeur $Q(s, a)$ avec une couche de sortie d'un vecteur de taille 3 ; c'est à dire que le réseau estime $Q(s, a)$ dans un environnement supposé où il y a 3 actions possibles. L'évaluation de la fonction $Q(s, a)$ commencera par une certaine valeur :

$$Q(s_t, a) = [0.1, 0.56, 0.22]$$

À noter que l'utilisation du réseau neuronal a l'avantage de pouvoir estimer directement les 3 actions possibles, et n'a donc besoin que de l'état t pour effectuer son estimation. Ainsi, au lieu de faire 3 fois le calcul $Q(s_t, a_0) = 0.1$, $Q(s_t, a_1) = 0.56$, $Q(s_t, a_2) = 0.22$, le réseau neuronal estime simplement $Q(s) = [0.1, 0.56, 0.22]$. Nous garderons cependant la notation suivante :

$$Q(s_t, a) = [0.1, 0.56, 0.22]$$

Dans ce même exemple, nous prendrons la fonction récompense suivante :

```
def reward_function(world) :
    if world.game_over :
        return - 0.5
    else :
        return 1
```

En continuant l'exemple donné, l'agent choisira donc l'action numéro 1, étant donné qu'il s'agit de l'action dont la valeur est la plus haute (0.56) (nous comptons à partir de 0). En prenant la fonction récompense vue plus haut, l'agent reçoit la récompense de 1 pour son action numéro 1. L'agent stocke donc dans son expérience les valeurs suivantes :

```
experience = [{'state' : s_t, 'action' : 1, 'reward' : 1}]
```

Dans les états qui suivent, étant donné que le réseau neuronal est à peine créé, il y a de fortes chances qu'il donne des valeurs équivalentes à l'état s_{t+1} :

$$Q(s_{t+1}, a) = [0.101, 0.562, 0.219]$$

Par conséquent, l'agent va à nouveau choisir l'action numéro 1 à l'état s_{t+1} . Les valeurs de l'expérience de l'agent devient :

```
experience = [{'state' : s_t,      'action' : 1, 'reward' : 1},
              {'state' : s_t+1, 'action' : 1, 'reward' : 1}]
```

De même, à l'état s_{t+2} , l'action choisit à nouveau l'action numéro 1, mais cette fois-ci l'agent se fait attraper par un ennemi. La récompense qu'il reçoit est de -0.5 , et l'expérience devient :

```
experience = [{'state' : s_t,      'action' : 1, 'reward' : 1},
              {'state' : s_t+1, 'action' : 1, 'reward' : 1},
              {'state' : s_t+2, 'action' : 1, 'reward' : -0.5}]
```

Mise à jour de $Q(s,a)$

Imaginons maintenant que l'expérience est suffisamment grande afin de commencer l'entraînement du réseau neuronal. Pour rappel, la mise à jour de la fonction $Q(s, a)$ se fait de la manière suivante :

$$Q(s_t, a_i) = r + \gamma Q(s_{t+1}, a_j)$$

Pour simplifier les calculs, commençons par mettre à jour la fonction pour $s = s_{t+2}$. Prenons un γ fixe de $\gamma = 1$. Comme il n'y a pas eu d'état s_{t+3} , (l'épisode s'est arrêté à l'état s_{t+2} , où l'agent a été attrapé par un ennemi), l'équation est ajustée de la sorte :

$$Q(s_{t+2}, a = 1) = -0.5 + \gamma Q(s_{t+3}, a) \quad (4.1)$$

$$Q(s_{t+2}, a = 1) = -0.5 \quad (4.2)$$

$$Q(s_{t+1}, a = 1) = 1 + \gamma Q(s_{t+2}, a = 1) \quad (4.3)$$

$$Q(s_{t+1}, a = 1) = 1 - 0.5 = 0.5 \quad (4.4)$$

$$Q(s_t, a = 1) = 1 + \gamma Q(s_{t+1}, a = 1) \quad (4.5)$$

$$Q(s_t, a = 1) = 1 + 0.5 = 1.5 \quad (4.6)$$

Nous constatons que la valeur de l'action 1 s'est vue renforcée pour s_t . Si l'agent se retrouve dans le même état s_t dans un autre épisode, il va à nouveau choisir l'action 1 étant donné que les valeurs des actions pour $Q(s_t, a)$ vaudra $Q(s_t, a) = [0.1, 1.5, 0.22]$. Si l'agent se retrouve dans l'état s_{t+1} , il va également choisir l'action 1 car celle-ci, bien que diminuée à 0.5, elle reste la valeur la plus élevée : $Q(s_{t+1}, a) = [0.101, 0.5, 0.219]$.

Aussi, la mise à jour de $Q(s, a)$ n'est faite uniquement que pour les actions que l'agent a effectué. Par conséquent, les valeurs des autres actions ne seront pas mises à jour tant que l'agent n'effectue pas d'autres actions que l'action 1. Dans notre exemple, il faut attendre que l'agent se retrouve à l'état s_{t+2} avant qu'il choisisse une autre action : $Q(s_{t+2}, a) = [0.12, -1, 0.211]$.

Comme l'épisode dure en général beaucoup plus longtemps qu'en trois actions effectuées, c'est à dire que la valeur de l'action numéro 1 n'est diminuée qu'après beaucoup plus d'actions numéro 1 choisies, l'action 1 a tendance à gagner en valeur et cette dernière n'est pas prête de diminuer.

4.1.3 Fonctions récompense utilisées

L'enjeu de la fonction récompense est double. Le premier enjeu, comme vu dans la sous-section précédente, il faut choisir une fonction de sorte à ne pas trop renforcer les actions effectuées. Dans l'idéal, la somme des récompenses lors d'un épisode devrait être comprise entre $[0, 1]$. Le deuxième enjeu est de choisir une fonction qui stimule l'apprentissage dans une tâche spécifique.

Les fonctions qui ont fait leur preuves dans le cadre de ce travail sont les suivantes :

Éviter les ennemis

```
def reward_function(world) :  
    if world.game_over :  
        return - 1  
    else :  
        safe_distance = 20  
        min_distance = float('inf')  
        for enemy in world.enemies :  
            distance = Direction.distance(enemy, world.agent)  
            if distance < min_distance :  
                min_distance = distance  
  
        if min_distance >= safe_distance :  
            return math.log(safe_distance+0.01) -1  
        elif min_distance <= 1 :  
            return -1  
        else :  
            return math.log(min_distance+0.01) -1
```

L'idée dans la fonction ci-dessus est de donner une récompenses élevée si l'agent se trouve éloigné de l'ennemi le plus proche. S'il se rapproche d'un ennemi, la récompense diminue. Dans les graphiques ci-dessous, le graphique de gauche représente la récompense en fonction de sa distance avec l'ennemi le plus proche. Dans le graphique de droite, l'on trouve les valeurs de toutes les récompenses en fonction de la positions des ennemis (en blanc).

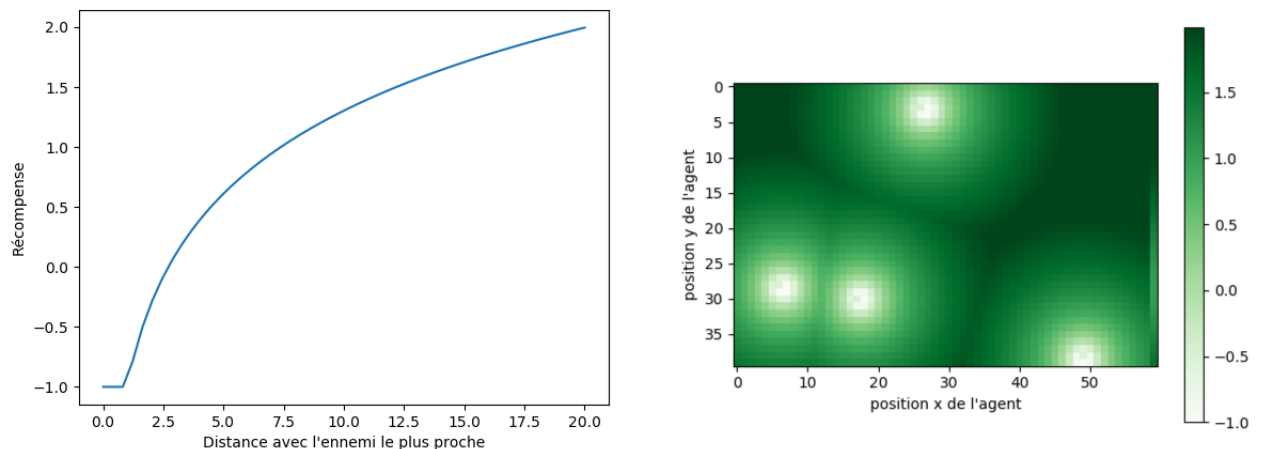


FIGURE 4.2 – Eviter ennemis : récompenses

Ci-dessous, les histogrammes de l'agent en train d'apprendre. Nous constatons que la distribution des actions s'est répartie entre les 9 actions possibles, et que les poids du réseau neuronal évolue. Nous pouvons donc en conclure que l'apprentissage se fait.

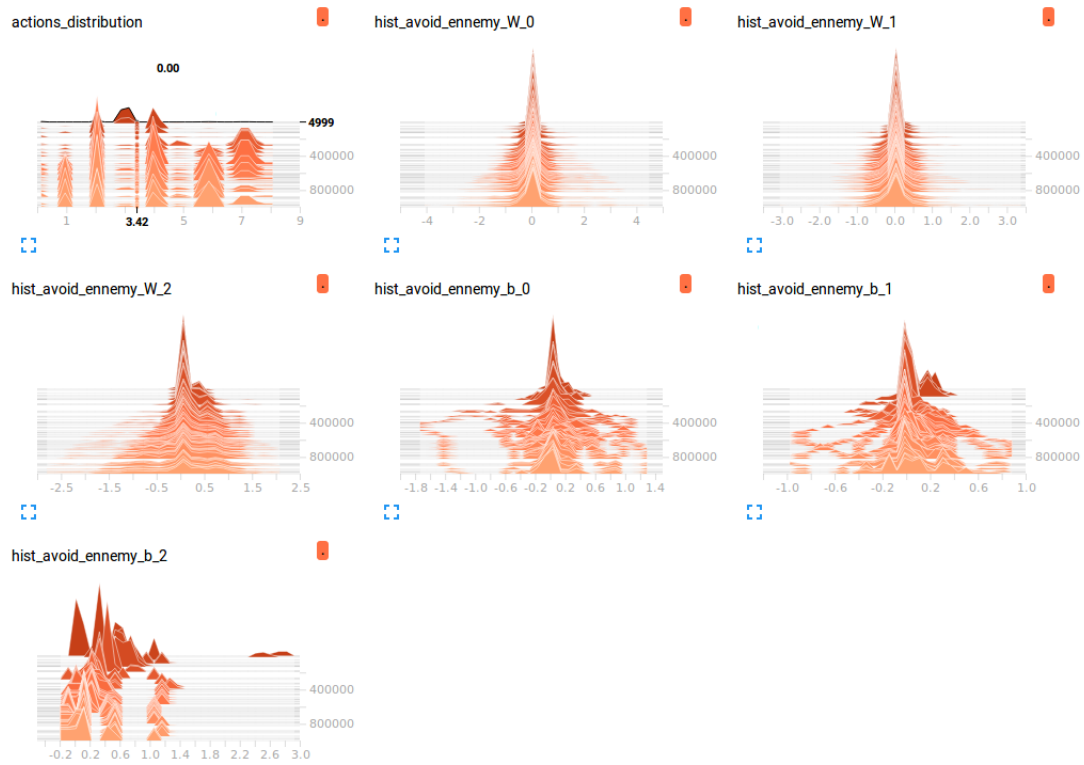


FIGURE 4.3 – Eviter ennemis : histogrammes

Attraper nourriture

La fonction récompense pour apprendre à attraper la nourriture dépend de la distance de l'agent vis-à-vis de la nourriture. Plus il est proche de celle-ci, plus la récompense sera élevée. Si l'endurance de l'agent tombe à 0, le jeu est terminé. L'agent reçoit à ce moment là un malus de -5 .

```
def reward_function(world) :  
    if world.game_over :  
        return - 5  
    max_distance = 73  
    return (1 - Direction.distance(world.agent, world.food) / max_distance)
```

Ci dessous à gauche la récompense en fonction de la distance de l'agent vis-à-vis de la nourriture. A droite, la récompense en fonction de la position de l'agent ; la nourriture est dans la zone la plus verte.

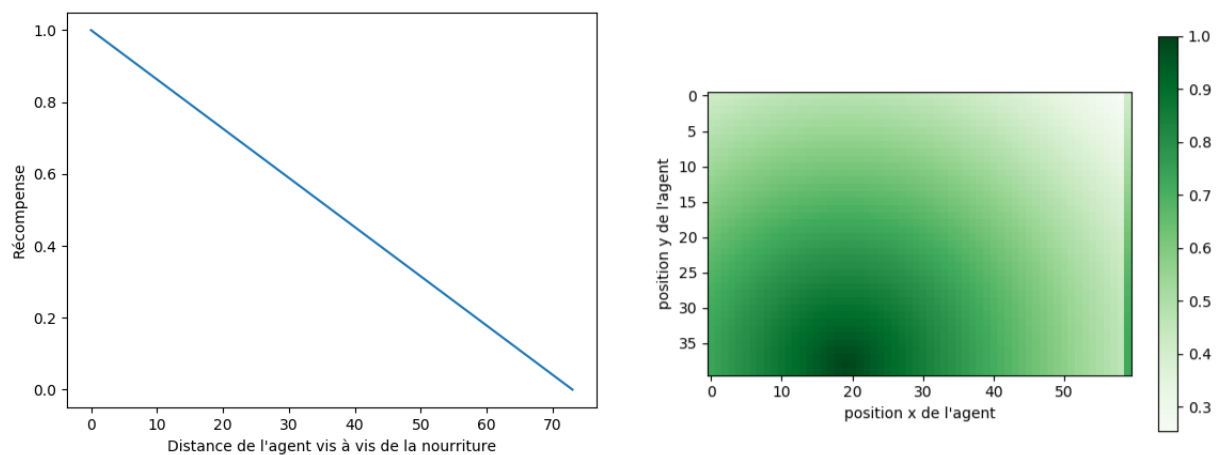


FIGURE 4.4 – Attraper nourriture : récompense

Ci dessous les histogrammes de l'agent en train d'apprendre à attraper la nourriture.

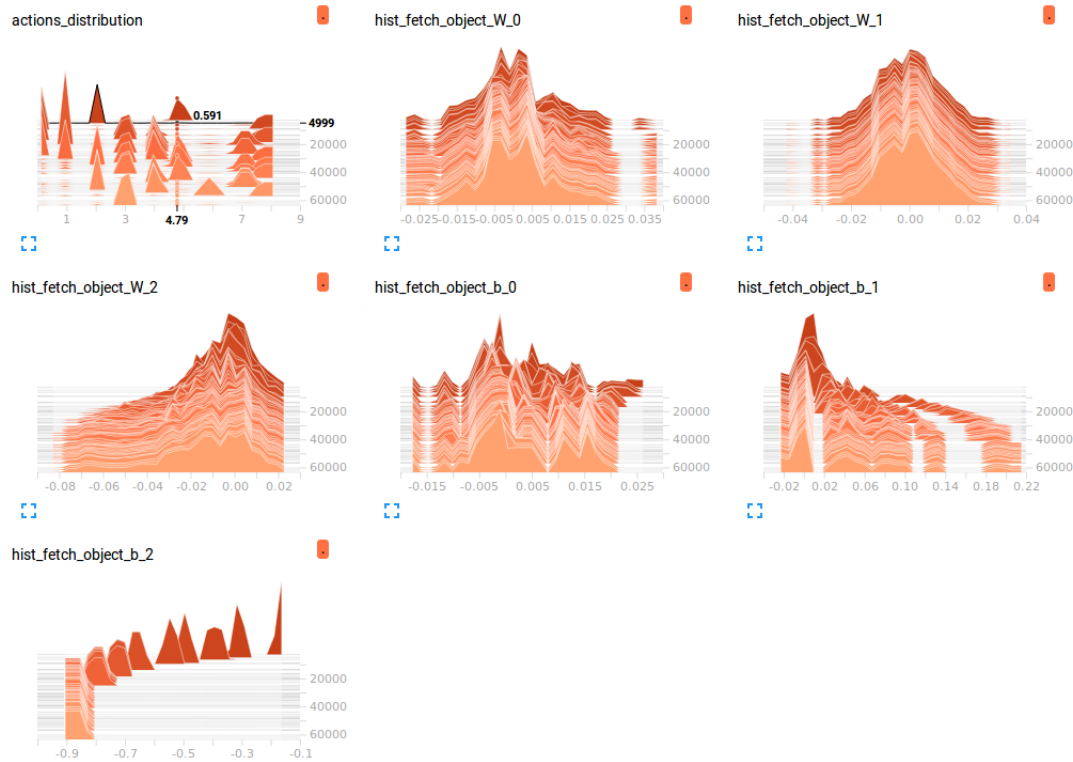


FIGURE 4.5 – Eviter ennemis : histogrammes

Jouer au jeu complet

Enfin, lorsque l'agent doit apprendre à choisir quel réseau neuronal écouter entre le réseau "éviter les ennemis" et "attraper la nourriture", la même fonction que "attraper la nourriture" a été utilisée. Si l'endurance de l'agent tombe à zéro, le jeu est terminé. Si l'agent se fait attraper par un ennemi, le jeu est également terminé. Dans les deux cas, la fonction récompense attribue un malus de -5 si le jeu se termine. L'accent a été mis sur l'importance d'attraper de la nourriture.

Conclusion : choix de la fonction récompense

Au vu des graphiques présentés plus haut, nous avons pu constater lorsque l'apprentissage ne se faisait pas (histogramme des actions présentant qu'une seule action effectuée, et histogrammes des poids des réseaux neuronaux stables). En changeant la fonction récompense de manière adéquate, nous avons pu constater qu'un apprentissage s'effectue (histogrammes des actions répartis sur les diverses actions possibles, et le poids des réseaux neuronaux évolue).

4.2 Batch size

4.2.1 Problème rencontré

Bien qu'une fonction récompense adéquate permet à l'agent d'apprendre, cela ne veut pas forcément dire que l'agent apprend dans des délais raisonnables. En effet, l'apprentissage pour éviter les ennemis prenait environ 14 jours avant d'arriver à un résultat médiocre. Afin d'accélérer le processus d'apprentissage, nous avons essayé d'augmenter la taille de l'échantillon (*batch size*) avec lequel l'agent apprenait. Pour rappel, lorsque l'expérience de l'agent est suffisamment grande, la mise à jour des réseaux neuronaux est effectuée en prenant un échantillon aléatoire de la taille du paramètre *batch size* sur l'expérience.

Au moment où l'apprentissage pour éviter les ennemis durait 14 jours, la taille de l'échantillon était de 32, pioché sur une expérience de taille 50'000 ; la proportion $\frac{32}{50'000}$ est choisie de sorte à s'assurer que l'apprentissage se fasse bien sur des échantillons suffisamment aléatoires. Nous avons donc tenté d'augmenter la taille de l'échantillon à 32'000 pour sur une expérience de taille 50'000'000 (en gardant la même proportion aléatoire), en espérant que l'agent puisse apprendre plus rapidement en fonction de la taille de l'échantillon.

Malheureusement, cela a empiré la durée de l'apprentissage. Ci dessous le graphique du score en fonction du nombre d'épisodes lors de l'apprentissage à éviter les ennemis. En bleu, le score pour un échantillon de taille 32, en orange un échantillon de taille 32'000.

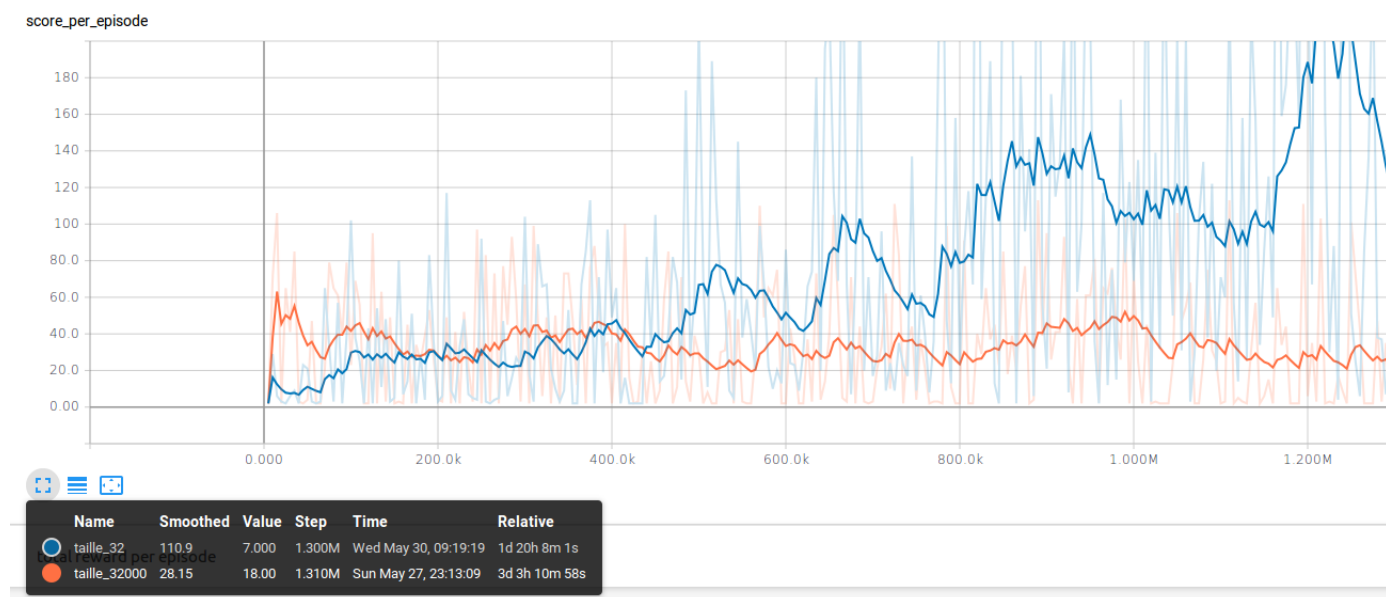


FIGURE 4.6 – Score par épisode avec deux tailles d'échantillons

4.2.2 Explication

Comme bien souvent, la conclusion nous a paru évidente qu'une fois avoir effectué les tests. Pour un échantillon de taille 32 et d'expérience 50'000, l'agent effectue 50'000 actions avant d'ajuster son réseau neuronal. Cela veut dire qu'il doit attendre 50'000 actions avant d'améliorer son comportement. Avec un échantillon de taille 32'000 et d'expérience de taille 50'000'000, l'agent doit attendre d'avoir effectué 50'000'000 actions avant d'améliorer son comportement ; l'apprentissage se fait donc beaucoup plus lentement.

A l'heure où ces lignes sont écrites, nous nous rendons compte que l'importance de cette section n'est peut être pas la taille de l'échantillon, mais de la taille de l'expérience avant d'effectuer la mise à jour du réseau neuronal. Il semblait important au moment des tests de garder une pro-

portion $\frac{\text{taille échantillon}}{\text{taille expérience}}$ suffisamment basse pour garantir l'homogénéité des échantillons utilisés lors de la mise à jour du réseau neuronal. Il doit cependant y avoir un juste milieu concernant ces paramètres.

4.3 Input X des réseaux neuronaux et discount factor γ

Le premier paramètre déterminant la vitesse d'apprentissage et de son efficacité, fut de choisir adéquatement les données fournies au réseaux neuronaux pour représenter l'état de l'environnement. Lorsque l'apprentissage pour éviter les ennemis durait 14 jours, les données fournies au réseau neuronal étaient la position de l'agent et la position des ennemis ; la vitesse et l'efficacité de l'apprentissage ont drastiquement augmenté lorsque nous avons fourni les trois dernières positions de l'agent et les trois dernières positions des ennemis pour représenter l'état de l'environnement.

Cet ajout permet probablement au réseau neuronal d'enregistrer non seulement la position des acteurs de l'environnement, mais également leurs directions, ce qui permet d'anticiper plus facilement quelle est la meilleure action à effectuer.

Cet ajout n'a cependant pas été le seul facteur déterminant. Le choix du *discount factor* γ est également décisif. Pour rappel, l'hyper-paramètre *discount factor* γ représente en quelque sorte la valeur d'une récompense future, c'est à dire d'une récompense probable dans des états futurs (cf Chapitre 2, section "Notion de base"). Avec $\gamma = 1$, la valeur d'un état menant à une récompense future est la même qu'un état attribuant cette même récompense. Une valeur $\gamma = 0$, la récompense future ne nous intéresse pas, et seule la récompense immédiate a de l'importance. Le *discount factor* γ répond en quelque sorte à la question : est-ce qu'il vaut mieux accepter une récompense de 1 immédiatement, ou attendre une récompense probable de 100 future ?

Ci dessous le graphique représentant le score par épisode en fonction du nombre d'épisodes vécus, pour l'enseignement à "éviter les ennemis". En choisissant un $\gamma = 0.8$ et en fournissant les trois dernières positions de l'agent et ennemis au réseau neuronal, la durée de l'apprentissage a pu être réduit à quatre / cinq jours au lieu de 14 jours.

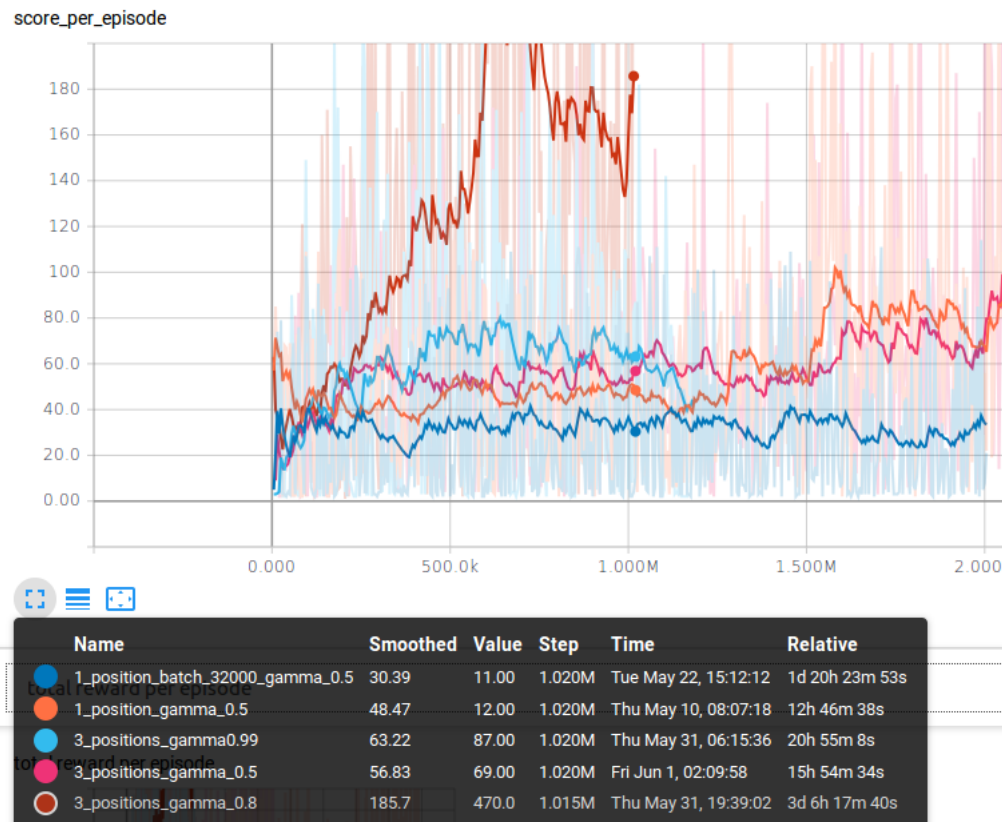


FIGURE 4.7 – Score par épisode avec en fonction du nombre d'épisodes vécus

4.4 Learning rate

Le dernier paramètre décisif dans la vitesse d'apprentissage fut de choisir un "learning rate" α plus élevé. Pour rappel, le poids des réseaux neuronaux sont ajustés de la manière suivante (cf chapitre 2, section 2.3.2 : Valeur d'un état avec $Q(s,a)$).

$$W = W + \alpha \frac{dJ(Y, T)}{dW}$$

L'idée du paramètre *learning rate* α est de mettre à jour le poids W petit à petit, afin d'éviter une mise à jour trop brusque, faussant l'apprentissage. Jusqu'ici, la valeur du learning rate était de $\alpha = 0.01$. Choisir α trop petit rend l'apprentissage plus long, mais choisir α trop élevé peut totalement casser l'apprentissage. Une valeur dix fois plus élevée $\alpha = 0.1$ pour l'apprentissage à éviter les ennemis et attraper la nourriture a permis de réduire la durée de l'apprentissage à 72 heures maximum au lieu de quatre / cinq jours maximum. Pour l'apprentissage à jouer au jeu complet, la valeur $\alpha = 0.1$ a cependant cassé l'apprentissage, et la valeur de $\alpha = 0.01$ a été gardée.

4.5 Hyper-paramètres finaux

Pour terminer ce chapitre, voici la liste des hyper-paramètres utilisés pour chacun des scripts **trainer_*.py**. Bien que les plus importants aient été décrits dans les sections précédentes, il paraît important de signaler que les apprentissages tel qu'ils ont été décrit jusqu'ici n'auraient pas été les mêmes sans la combinaison de tous les hyper-paramètres à prendre en compte dans l'apprentissage par renforcement. Le seul paramètre qui n'est pas décrit dans cette section est le choix de la fonction de récompense, vu qu'elle a été décrite dans une section précédente.

Hyper-paramètres communs aux 3 scripts :

$$\epsilon \text{ (probabilité d'action aléatoire)} = 0.1 \text{ (constante)} \quad (4.7)$$

$$\text{fréquence de la mise à jour du TNetwork} = 10'000 \quad (4.8)$$

$$\text{taille experience} = 50'000 \quad (4.9)$$

$$\text{batch size} = 32 \quad (4.10)$$

Script éviter les ennemis :

$$\alpha \text{ (learning rate)} = 0.1 \quad (4.11)$$

$$\gamma \text{ (discount factor)} = 0.8 \quad (4.12)$$

$$\text{input adapter} = 3 \text{ dernières position de l'agent et ennemis} \quad (4.13)$$

Réseau neuronal:

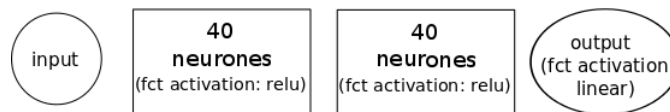


FIGURE 4.8 – Architecture réseau neuronal utilisé

Script attraper la nourriture :

$$\alpha \text{ (learning rate)} = 0.1 \quad (4.14)$$

$$\gamma \text{ (discount factor)} = 0.5 \quad (4.15)$$

$$\text{input adapter} = \text{position agent et position nourriture} \quad (4.16)$$

Script jouer au jeu complet :

$$\alpha \text{ (learning rate)} = 0.01 \quad (4.17)$$

$$\gamma \text{ (discount factor)} = 0.5 \quad (4.18)$$

$$\text{input adapter} = \text{endurance, distance ennemis et nourriture} \quad (4.19)$$

Chapitre 5

Conclusion

5.1 Difficultés

Pour terminer, il paraît important de relever les difficultés rencontrées lors de l'étude de l'apprentissage par renforcement. Le champ de celui-ci est vaste et est encore sujet à de nombreuses recherches. Il évolue d'années en années, et de nouveaux algorithmes voient le jour régulièrement. Il n'est évident de se plonger dans cette branche du machine learning tant les ressources sont diverses et pointues, et la théorie est bien souvent loin d'algorithmes concrets à reproduire chez soi.

Par ailleurs, le nombre important d'hyper-paramètres à ajuster rend l'implémentation difficile lorsque l'on cherche à résoudre un problème donné, bien que l'architecture créée pour cela aie été développée sans erreurs.

Enfin, le temps d'apprentissage est également une des principales contraintes. Si la compilation de certains programmes peut durer des heures, il n'est pas rare de devoir attendre des semaines afin de voir si l'agent apprend comme prévu.

5.2 Perspectives

Cela reste néanmoins un sujet passionnant, d'actualité, qui mérite d'être approfondi. Avec l'avancée d'années en années de la recherche sur ce sujet, qui sait de quoi sera faite la technologie de demain. Il me paraît important de suivre l'évolution de celui-ci pour pouvoir y garder un avis critique.

5.2.1 Nouvel environnement

Si le temps me permettait d'améliorer ce travail de bachelor, je commencerais par changer d'environnement : l'environnement utilisé s'est avéré inapproprié. En effet, bien qu'il aie été possible d'enseigner à l'agent des tâches spécifiques telles qu'éviter des ennemis ou attraper de la nourriture, enseigner à l'agent à évoluer dans l'environnement complet en utilisant uniquement deux compétences n'est pas adaptée. Si l'ennemi le plus proche de l'agent et la nourriture se trouvent en bas à droite de l'agent, l'agent n'a pas la possibilité de choisir une action convenable afin d'éviter l'ennemi et de se diriger vers la nourriture.

L'exemple ci-dessus illustre ces propos. Dans cet état, le réseau neuronal "éviter ennemis" recommande de se diriger en haut à gauche, alors que le réseau "attraper nourriture" recommande de se diriger en bas à droite. Le réseau "jouer au jeu complet" devrait choisir entre une de ces deux actions recommandées, et n'a pas possibilité de choisir une action intermédiaire.

Un environnement plus adapté à l'architecture de ce travail de bachelor serait par exemple un

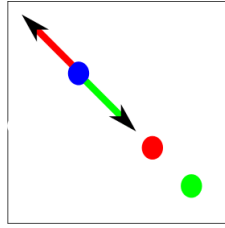


FIGURE 5.1 – Situation non adaptée à l'architecture choisie

corps dans lequel l'agent doit apprendre à se mouvoir.

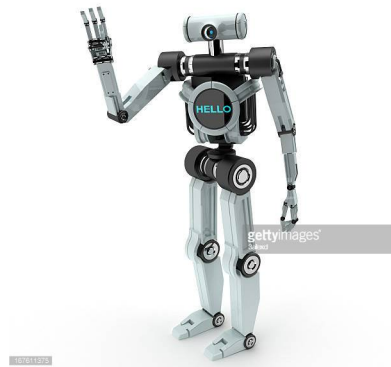


FIGURE 5.2 – Nouvel environnement

Nous pourrions par exemple envisager un réseau neuronal pour apprendre à marcher, un autre pour apprendre à se relever lorsque le corps tombe, un troisième pour apprendre à attraper un objet à portée de main, et le dernier réseau pour apprendre à jongler entre les réseaux neuronaux. Les réseaux neuronaux ne seraient pas en contradictions contrairement à l'environnement de ce projet de bachelor.

5.2.2 Nouvel algorithme

Un autre élément que je changerais est l'algorithme d'apprentissage par renforcement utilisé. L'algorithme "Deep Q-Network" utilisé dans ce travail de bachelor date de 2015 est à présent presque devenu un "cas d'école". Le désavantage de celui-ci est qu'il est "lent" d'apprentissage : il a pour but de diminuer l'erreur de Bellman (optimiser l'équation de Bellman) en n'ajustant que les valeurs pour les actions que l'agent a effectuées. D'autres algorithmes plus performants ont depuis vu le jour, comme les algorithmes "actor / critic" : au lieu d'utiliser un QNetwork et un TNetwork, l'on peut par exemple utiliser un réseau neuronal ayant pour but d'estimer la meilleure action dans un état donné (actor), et un autre réseau ayant pour but d'estimer "l'erreur" du réseau "actor". Le réseau "actor" ne se mettrait pas à jour en fonction des récompenses reçues, mais uniquement en fonction des prévisions effectuées par le réseau "critic". L'avantage est que l'acteur mettrait à jour l'entièreté des valeurs de $Q(s,a)$, et non uniquement des actions effectuées par l'agent. Le réseau "critic" se mettrait à jour en fonction des récompenses reçues en fonction de chaque état rencontré. Cette séparation (semble t'il) rend l'apprentissage plus générique et efficace.

Chapitre 6

Annexes

6.1 Installation

Les pré-requis sont **python3**, **pip**, **python3-tk** et **git**

```
# télécharger le projet
git clone https://github.com/Eldorico/deep-q-network-bachelor.git
cd deep-q-network-bachelor

# créer un environnement python virtuel dans le dossier
pip install virtualenv
virtualenv virtual_env_folder

# activer l'environnement virtuel
source virtual_env_folder/bin/activate

# installer les dépendances
pip install tensorflow
pip install gym
pip install pygame
```

6.2 Jouer soi-même au jeu

```
# activer l'environnement virtuel
source virtual_env_folder/bin/activate

# configurer le fichier src/world.py en fonction du besoin
# (dans la partie __main__)
# puis lancer l'environnement
cd src
python world.py

# s'ouvriront 2 fenetres : une petite noire, et l'autre avec le jeu :
# avoir le focus sur la fenetre noire,
# et jouer avec les fleches directionnelles
```

6.3 Voir l'agent évoluer

```
# activer l'environnement virtuel
source virtual_env_folder/bin/activate

# se déplacer ds dossier src
cd src

# executer l'un des scripts suivants
# python see_avoid_enemies_in_action.py
# python see_fetch_object_in_action.py
# python see_play_game_in_action.py
```

6.4 Entraîner l'agent depuis zéro

```
# activer l'environnement virtuel
source virtual_env_folder/bin/activate

# se déplacer ds dossier src
cd src

# lancer le script pour apprendre à éviter les ennemis
python trainer_avoid_enemies.py
# attendre 72 heures. Si le script n'est pas terminé, arrêter avec ctrl+c

# lancer le script pour apprendre à attraper la nourriture
python trainer_fetch_object.py
# attendre 72 heures. Si le script n'est pas terminé, arrêter avec ctrl+c

# copier les réseaux enregistrés dans le dossier un dossier play_game
mkdir ../tmp_saves/play_game
cp ../tmp_saves/avoid_enemies/* ../tmp_saves/play_game
cp ../tmp_saves/food/* ../tmp_saves/play_game

# lancer le script pour apprendre à jouer le jeu complet
python trainer_play_game.py
# attendre 72 heures. Si le script n'est pas terminé, arrêter avec ctrl+c

# pour voir ensuite l'agent évoluer, ajuster les variables
# Debug.SAVE_FOLDER en conséquence dans les scripts
# src/see*
```