Apprentissage par renforcement pour jeux vidéo

Jérôme Desroziers
Télécom Nancy - Promotion 2019
Email : jerome.desroziers.54@gmail.com

Jérémy Thivet
Télécom Nancy - Promotion 2019
Email : thivetj@gmail.com

Vincent Thomas
et Olivier Buffet
Chercheurs au LORIA
Emails : vincent.thomas@loria.fr
olivier.buffet@loria.fr

***Résumé*—Ce document retrace un travail effectué sur l'apprentissage automatique appliqué aux jeux vidéo, en particulier l'apprentissage par renforcement via l'utilisation d'un réseau de neurones connexioniste.**
Initialement le but de cette étude était de réussir à réaliser l'apprentissage d'un jeu par l'ordinateur via des techniques dites de Deep Learning, mais cela n'a pas été possible par manque de temps, d'où l'utilisation de réseaux de neurones simples.
L'article présente la théorie sur différentes notions : le Q-learning ainsi que les réseaux de neurones puis propose d'étudier les résultats obtenus sur différentes expérimentations, l'objectif final de l'étude étant de réussir à faire jouer un ordinateur à un jeu vidéo simple prévu pour l'être humain que nous avons programmé.

***Mots-clés*—**
— apprentissage automatique, jeux vidéo
— apprentissage par renforcement, Q-learning
— apprentissage profond, réseau de neurones

I. Introduction

L'intelligence artificielle (IA) est « l'ensemble de théories et de techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence » [7]. L'apprentissage automatique en est un champ d'application dans lequel on souhaite qu'un ordinateur soit en mesure d'effectuer certaines tâches complexes (classification, reconnaissance de formes, ...), moyennant un entraînement, lui permettant d'évoluer et d'apprendre par lui-même.
Un banc d'essai couremment utilisé est le jeu vidéo, où l'ordinateur peut être vu au sens large comme un agent, placé au milieu d'un environnement, parfois hostile envers lui, et dont le but est d'atteindre un objectif afin de gagner la partie. Ces jeux sont une source inépuisable de terrains d'entraînements pour ce genre d'algorithmes où les dernières recherches ont permis des avancées notoires en terme d'apprentissage par essais et erreurs. Le but étant, à terme, de soumettre un jeu vidéo à une intelligence artificielle, cette dernière étant capable d'y jouer après s'être entraînée dessus. (S'orienter toute seule dans un labyrinthe comportant des pièges par exemple.)
La résolution de ce problème se fera en plusieurs temps : tout d'abord, il est nécessaire d'assimiler les concepts d'apprentissage automatique, puis nous aborderons des techniques plus poussées reposant sur des modèles de réseaux de neurones (et relevant toujours de l'apprentissage automatique). Enfin, nous appliquerons nos recherches sur un jeu vidéo afin d'en tirer des conclusions.

14 mai 2018

II. Revue de littérature

A. Apprentissage automatique

L'apprentissage automatique au sens large définit l'ensemble des méthodes et algorithmes déployés sur des systèmes informatiques permettant à ces derniers d'adapter leur comportement et leurs réponses en se basant sur l'analyse de données empiriques. Un exemple des plus classiques est celui de la reconnaissance de formes : on soumet des données à un algorithme (bien souvent sous la forme d'images acquises par un capteur), et celui-ci doit être en mesure d'en extraire des formes identifiées (reconnaître les carrés, les disques, ...). Cette capacité à reconnaître les formes ne repose pas sur un seul traitement numérique de l'image (morphologie mathématique, quantification, binarisation, seuillage, ...) mais aussi sur un entraînement effectué au préalable. En effet, bien souvent, le nombre possible d'entrées soumises au système est très important, si bien que le nombre de comportements théoriques possibles en résultant devient vite incommensurable, il s'agit de l'explosion combinatoire. Tout l'enjeu de ces algorithmes d'apprentissage automatique est de simplifier cette complexité en soumettant des données empiriques à la machine afin que celle-ci ajuste au mieux son modèle comportemental.
Les variantes dans les algorithmes et dans les méthodes d'apprentissage du système ont donné naissance à plusieurs classes de problèmes de l'apprentissage automatique. On compte parmi eux l'apprentissage supervisé (les données soumises sont connues et classifiées, l'algorithme peut alors s'en servir établir son modèle), l'apprentissage non supervisé (les données soumises ne sont pas étiquetées, l'algorithme doit élaborer seul la classification, sans aide) ou encore l'apprentissage par renforcement.
Ce dernier est celui qui s'applique le mieux au cadre du jeu vidéo : l'algorithme fournit une action qui, passée à l'environnement, produit une valeur qui permettra à l'algorithme de s'orienter et de s'améliorer. Un exemple classique d'algorithme est celui du Q-learning que nous traiterons dans la section suivante.
Toutes ces méthodes possèdent déjà des applications concrètes dans le monde actuel : les algorithmes de reconnaissance vocale embarqués sur la plupart de nos smartphones, la re-

connaissance de caractères, les algorithmes de prédiction de comportement sur les moteurs de recherches, etc...

B. Apprentissage par renforcement : méthode du Q-learning

L'apprentissage par renforcement repose sur une mécanique d'essais/erreurs inspirée d'observations effectuées sur la psychologie animale [1] (comment un animal apprend-il de son environnement par essais/erreurs). D'ordinaire, on présente ce problème en utilisant l'image d'un agent plongé au sein d'un environnement. Ce même agent doit prendre des décisions en fonction de l'état (la situation courante) afin de maximiser la future récompense qu'il pourra obtenir de ce même environnement.
Mathématiquement parlant, un problème d'apprentissage par renforcement est défini par [2] :

- Un ensemble d'états S possibles de l'agent dans un environnement donné. (Position dans un environnement, état interne d'un agent face à un problème donné, ...)
- Un ensemble d'actions A que l'agent peut effectuer dans l'environnement. (Avancer dans l'environnement, modifier son état interne, ...)
- Un ensemble de scalaires de récompenses R que l'agent peut obtenir de l'environnement.

L'idée est la suivante : à un temps t, l'agent choisit une action $a \in A$ en fonction de l'état $s_t \in S$ dans lequel il se trouve. Il fournit ensuite cette action à l'environnement qui lui répond avec un nouvel état $s_{t+1} \in S$ ainsi qu'une récompense $r_{t+1} \in R$ (bien généralement cette récompense vaut 0 dans les états neutres et 1 dans les états importants de l'environnement). Toutes ces interactions avec l'environnement doivent permettre à l'agent d'établir une politique $f : S \to A$ qui doit retourner une action en fonction d'un état donné. L'action retournée doit bien entendu maximiser la récompense obtenue par l'agent plus ou moins loin dans le futur.
Le Q-learning est un algorithme d'apprentissage par renforcement. L'objectif de l'algorithme est de fournir l'action qui maximisera la somme des récompenses futures dans chaque état ; on peut alors construire une politique optimale qui vise à sélectionner l'action idéale dans chaque état. Autrement dit, une action qui peut apparaître comme intéressante sur un court terme n'est peut être pas finalement l'action qu'il faut faire à ce moment là pour avoir une récompense maximum à long terme. L'algorithme de Q-learning permet justement de prendre en compte cette nuance.
La fonction que l'on cherche à maximiser ici est $Q : S \times A \to \mathbb{R}$: en fonction d'un état et d'une action, on retourne la récompense qui y serait théoriquement associée à long terme. Ainsi pour un environnement donné, l'algorithme peut être vu comme suit :

1) Pour l'état $s_t \in S$ dans lequel on se trouve à l'instant t, on calcule $r_t = \max_{a \in A} Q(s_t, a)$. On a alors $r_t = Q(s_t, a_t)$
   Important : Dans certains cas, l'action à effectuer est déterminée aléatoirement par l'algorithme sans tenir compte de la fonction Q, on appelle ce comportement la composante exploratoire d'un algorithme. En effet, si l'on se basait uniquement sur cette fonction, l'agent pourrait s'enfermer dans une seule et même politique qu'il aurait trouvée au début, sans pour autant explorer les autres choix possibles qui pourraient s'avérer intéressants. On parle du compromis entre exploration et exploitation.
2) On soumet a_t, l'action optimale, à l'environnement et on reçoit une récompense r_{t+1} ainsi qu'un nouvel état s_{t+1}.
3) On met à jour la valeur que renvoie $Q(s_t, a_t)$ en fonction des nouveaux paramètres que l'on a.
   Au temps t, la fonction renvoie $r_t = Q_t(s_t, a_t)$. Au temps $t + 1$, on veut qu'elle renvoie une valeur $Q_{t+1}(s_t, a_t)$. La formule de mise à jour de la fonction est disponible en annexe (A-A). Cette formule comporte notamment des facteurs α et γ qui sont respectivement la *vitesse d'apprentissage* (détermine à quel point la nouvelle information est importante comparée à l'ancienne : à 0, on n'apprend rien de la nouvelle information, à un facteur 1, on ne garde aucune information du passé et on ne considère que la nouvelle information. En pratique il faut trouver un point d'équilibre pour cette valeur afin qu'elle permette à l'algorithme de converger) et au *facteur d'actualisation* (ce facteur détermine à quel point les récompenses futures sont importantes : à 0, l'agent ne considère que la récompense courante, alors qu'un facteur à 1 prendrait en compte des récompenses trop éloignées. De la même façon, cette valeur peut mener à une divergence si mal choisie).

Dans le cas d'un tâche épisodique, l'algorithme de Q-learning se termine lorsque l'état s_{t+1} est un état final.
Dans un environnement où le nombre d'états possibles est connu et résolument petit, il est très simple de représenter la fonction par un tableau multidimensionnel dont les dimensions sont les états et les actions. En revanche, dans des applications où l'espace états-actions n'est pas discret, une telle représentation laisse très vite apparaître des limites, c'est pourquoi on la substitue par un modèle connexionniste de type réseau de neurones.

C. Réseaux de neurones

1) Généralités: De façon générale, un réseau de neurones peut être assimilé à une fonction de $R^N \to R^M$ et, de fait, est une structure particulièrement adaptée à des tâches relevant de la classification.
Il est constitué d'une à plusieurs couches communiquant entre elles 2 à 2, elles-mêmes composées de neurones, que nous détaillons ci-après :

2) Neurone: Le neurone (figure 1) est la brique élémentaire d'un réseau de neurones.

Il est constitué des éléments suivants : N entrées x_i (réelles) pondérées par N poids w_i (réels), 1 sortie, et une **fonction**

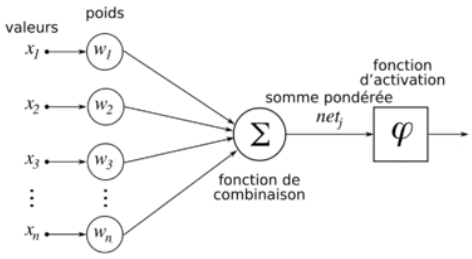

FIGURE 1. Structure d'un Neurone

d'activation.
Cette dernière peut prendre différentes formes et dépend de la nature de la sortie attendue. Elle doit notamment répondre à certaines contraintes, comme la dérivation (qui est une condition nécessaire à la rétropropagation du gradient comme vu en II-D).
Parmi les plus utilisées, on peut citer :

— la fonction **Sigmoïde** (figure 2), d'équation $y = \frac{1}{1+e^{-x}}$, avec $y \in [0, 1]$, utile lorsqu'on veut que le résultat donné par le neurone soit binaire.

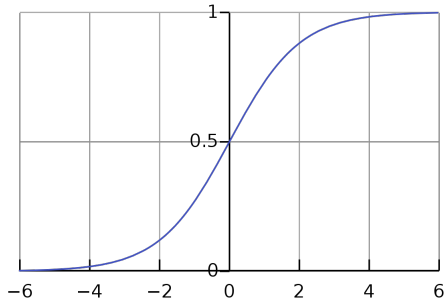

FIGURE 2. Fonction Sigmoïde

— la fonction identité, utile si on veut avoir une sortie sur R.

L'entrée x donnée à la fonction d'activation est en général égale à $(\sum_k w_i x_i)$ la somme pondérée des entrées du neurone.

Voyons à présent comment est constituée une couche quelconque d'un réseau de neurones :

3) Couche d'un réseau de neurone: Un réseau de neurones est divisé en couches , elle-même constituée de neurones.
On distingue généralement 3 types de couches :

— La couche d'entrée, constituée de N neurones.
— La couche de sortie, constituée de M neurones.
— Les couches dites cachées (qui peuvent être au nombre de 0), chacune constituée d'un nombre quelconque de neurones.

Dans les réseaux de neurones les plus simples (que nous utiliserons ici), ces couches suivent une organisation hiérarchique, c'est à dire (figure 3) qu'elles ne peuvent communiquer qu'avec les couches les précédant ou les suivant directement, les entrées d'une couche étant fournies par les sorties de la couche précédente.
Les neurones d'une même couche ne communiquent pas entre eux, mais ont chacun pour entrée l'ensemble des sorties des neurones de la couche précédente, qu'ils pondèrent avec leurs propres poids
Pour un neurone d'indice j appartenant à une couche i (noté $N_j^{(i)}$), on peut noter :
$x_j^{(i)} = S^{(i)}(\sum_k w_{jk}^{(i)} x_k^{(i-1)})$, avec $x_j^{(i)}$ sa valeur de sortie, **S** sa fonction d'activation, $x_k^{(i-1)}$ la sortie du k-ème neurone de la couche $i-1$, et $w_{jk}^{(i)}$ le poids associé au couple $\{N_j^{(i)}, N_k^{(i-1)}\}$
L'obtention d'une sortie par un réseau de neurones à partir d'un vecteur d'initialisation suit donc l'algorithme suivant :

1) Les N valeurs de sortie de la couche d'entrée ($i = 0$) sont calquées sur les N valeurs du vecteur d'initialisation.
2) Pour la couche i, $i \in [1, N]$:
   — Pour le neurone $x_k^{(i)}$, $k \in [1, nb\_neurones\_couche\_i]$, calculer la valeur de sortie.

A sa création, un réseau de neurones a ses poids initialisés de façon aléatoire, et pour un vecteur d'initialisation donné les valeurs de sorties données par le réseau ne seront en général pas égales aux valeurs attendues.
L'idée générale est donc d'entraîner le réseau et, pour chaque résultat incorrect obtenu, de corriger ses poids en utilisant la rétropropagation du gradient (section II-D) afin que ce dernier obtienne (empiriquement) le comportement souhaité.

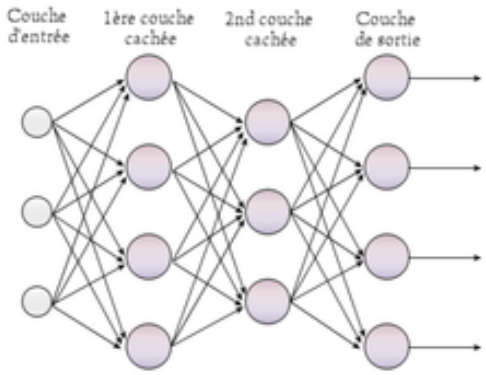

FIGURE 3. Représentation simplifiée d'un réseau de neurones

Note : les fonctions que l'on veut représenter grâce à un réseau de neurones ont souvent leur valeur en 0 non nulle ; or l'architecture décrite précédemment ne permet pas de modéliser de tels comportements (un vecteur d'entrées nul donnera un agrégat nul en entrée à la fonction d'activation).
Pour pallier ce problème, on ajoute à chaque couche i un neurone dont la valeur de sortie est 1, appelé $Biais^{(i)}$. Cette valeur est fixe et seuls les poids associés aux couples $\{Biais^{(i)}, N_j^{(i+1)}\}$ changeront lors d'une mise à jour du réseau.

Abordons maintenant le processus par lequel on effectue la mise à jour des poids du réseau.

D. Rétropropagation du gradient

1) Notion de descente de gradient: La descente de gradient [4] est un outil permettant, pour une fonction f donnée, d'approcher son minimum de façon itérative en suivant la direction de son gradient.
Pour un x_0 choisi au hasard, et pour un pas d'itération α fixé, on a : $x_{k+1} = x_k - \alpha f^{'}(x_k)$, avec x_n approchant le minimum de la fonction, pour n très grand. Cet outil possède néanmoins 2 défauts :

- Si la fonction oscille et possède plusieurs minima locaux (comme par exemple le sinus cardinal), un mauvais x_0 fera tendre x_n vers le minimum local le plus proche, et pas vers le minimum global.
- Le choix du α dépend de la fonction étudiée : une valeur trop petite aura pour conséquence une convergence extrêmement lente, tandis qu'une valeur trop grande peut conduire à une divergence de la suite (le pas entre un x_i et un x_{i+1} donnés étant si important qu'on passe d'un creux de la courbe à un autre)

2) Application aux réseaux de neurones: Faire en sorte qu'un réseau de neurones, pour un nombre n donné de vecteurs d'initialisation, donne n résultats corrects par rapports aux résultats attendus revient à représenter ce réseau par une fonction d'erreur quadratique que l'on veut faire tendre vers 0, de formule : $E = \frac{1}{2}\sum_k (t_k - y_k)^2$, où t_k et y_k sont respectivement la valeur attendue et la valeur obtenue du k-ème neurone de la couche de sortie.
L'idée est donc d'utiliser la descente de gradient pour minimiser cette fonction d'erreur et à terme obtenir un ensemble de poids adéquats.

En notant **sum** la somme pondérée des entrées du neurone k de la couche courante, et S' la dérivée de la fonction d'activation, l'algorithme de rétropropagation est le suivant :

- Pour chaque neurone de la couche de sortie :
  $e_k^{sortie} = S'(sum)(t_k - y_k)$
- Pour chaque couche i en remontant à partir de l'avant-dernière couche :
  - Pour chaque neurone k :
    $e_k^{(i-1)} = S'(sum)\sum_j w_{jk} e_j^{(i)}$, où w_{jk} est le poids associé au couple $\{N_k^{(i-1)}, N_j^{(i)}\}$
- Une fois chaque neurone du réseau associé à sa valeur d'erreur, on effectue la mise à jour des poids par descente de gradient selon la formule :
  $w_{jk}^{(i)} = w_{jk}^{(i)} + \alpha e_j^{(i)} x_k^{(i-1)}$, avec $x_k^{(i-1)}$ la valeur de sortie du neurone k.

E. Utilisation dans le cadre du Q-learning

Nous avons vu dans la section II-B que les limites du Q-learning résident dans la difficulté de représenter la fonction à apprendre (généralement un tableau dans les cas simples) dès lors que le nombre de couples {état,action} devient trop élevé.
Dans le cadre de ce projet par exemple, où l'on souhaite reproduire l'expérience décrite dans l'article "Playing Atari With Deep Learning" [5], représenter l'ensemble des états du jeu testé (qui correspond ici à chaque image différente qu'il est possible d'obtenir en jouant au dit jeu) par un tableau n'est pas du tout efficient.

Une solution consiste à utiliser l'algorithme du Q-learning classique tout en remplaçant son tableau d'états par un réseau de neurones (ici connexioniste), où la mise à jour du tableau après avoir effectué une action est remplacée par une mise à jour des poids du réseau de neurones via la rétropropagation du gradient.
Le réseau de neurones prend alors en entrée un vecteur représentant le couple {état,action} courant, la partie état du vecteur pouvant par exemple être la concaténation des lignes d'une image en nuances de gris ; et envoie en sortie le score $Q(etat, action)$ qu'il attribue à ce couple.
Les décisions réalisées par l'algorithme afin de savoir quelle action est la meilleure dans un état donné se basent donc sur les estimations de récompenses renvoyées par le réseau de neurones, estimations qui sont ensuite utilisées en combinaison aux récompenses réelles perçues après réalisation de l'action afin de réaliser une rétropropagation de l'erreur et corriger les poids du réseau.
L'exemple le plus connu de ce genre d'algorithme est celui du jeu de Go "AlphaGo Zero" développé par DeepMind [8] en 2017 et qui a réussi à battre le champion du monde Lee Sedol lors d'un tournoi organisé par Google à partir d'un entraînement en apprentissage par renforcement en jouant contre lui même ou bien des humains.

III. Méthodologie

A. Q-learning

L'apprentissage par renforcement basé sur la méthode du Q-learning a fait l'objet d'un premier travail que nous détaillons dans cette section.

1) Analyse et solution: Afin de pouvoir appliquer la théorie à un exemple, il était nécessaire d'élaborer un premier environnement dans lequel nous pourrions faire évoluer un agent. Nous avons opté pour un jeu de type "labyrinthe" dont les caractéristiques sont les suivantes :

- L'agent part d'un point de départ A et doit rejoindre un point d'arrivée B.
- Le labyrinthe comporte des cases piégées qui font perdre la partie et remettent l'agent au point A.
- Les récompenses sont attribuées comme suit :
  - Case B : Récompense de 100.
  - Cases pièges : Récompense de -100.
  - Cases intermédiaires neutres et case A : récompense de 0.
- On se situe dans un environnement en 2D dont les actions A sont {déplacement nord, déplacement est, déplacement sud, déplacement ouest}.
- L'ensemble d'états S possibles est l'ensemble des positions $\{X,Y\}$ pouvant être prises par l'agent dans le tableau représenté par le labyrinthe.

— Une composante de **risque** : L'environnement est "glissant", c'est à dire que parfois, une action va être décidée mais l'agent va glisser et aller dans une direction aléatoire.

L'algorithme du Q-learning effectuant ses choix sur une politique de maximisation de récompense à long terme, celui-ci ne fera pas forcément les choix les plus simples dans sa résolution. Par exemple, la figure 4 montre le prototype de labyrinthe expérimental sur lequel nous avons effectué nos manipulations. D'ordinaire, on pourrait penser que l'agent ait, au bout d'un certain nombre d'entraînements, opté pour le chemin en rouge. Cependant, celui-ci comporte des risques : en effet, en longeant les pièges de cette façon, l'agent peut glisser et perdre. Ainsi, en faisant suffisamment tourner l'algorithme, on se rendra compte que l'agent optera plutôt vers la solution tracée en bleu ici, celle qui maximise la récompense à long terme.

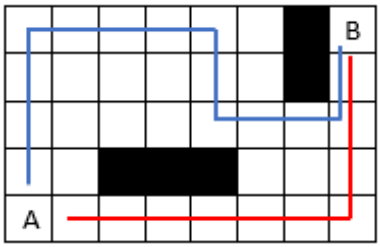

FIGURE 4. Modèle du labyrinthe d'expérimentation (En noir : les cases pièges)

2) Conception: La conception de cette première expérience a été réalisée sous Java, avec un labyrinthe entièrement paramétrable, ce qui nous a permis de tester différents agencements.

Nous avons pour notre exemple choisi 5 lignes (de 0 à 4) et 8 colonnes (de 0 à 7). Au yeux de l'algorithme (II-B), les états finaux sont soit : la case B (partie gagnée), soit les cases noires pièges (partie perdue).

La composante du dilemme exploration/exploitation est matérialisée par un choix aléatoire de la part du contrôleur sans se baser sur la fonction Q-Valeur avec une probabilité de 1/5. (Une fois sur cinq, on ne tient pas compte de la fonction Q et on donne une action aléatoire afin d'explorer l'environnement sans préjugés).

Nous avons opté dans ce cas pour un $\alpha = 0,1$ et un $\gamma = 0,8$, il s'agit de valeurs usuelles satisfaisantes dans la plupart des cas d'après nos recherches [3].

L'affichage nous permet d'afficher le modèle et la fonction Q-Valeur pour chaque état et chaque action afin de mieux comprendre le déroulement de l'algorithme.

La figure 5 est un aperçu de ce que peut représenter la vue globale. Nous avons défini la fonction $Q : S \times A \rightarrow \mathbb{R}$ comme un tableau multidimensionnel `qvaleur[no_ligne][no_colonne][action]` : les deux premières dimensions représentent l'état de l'agent, la troisième dimension représente l'action. Le tableau a ses cases initialisées avec la valeur 0.

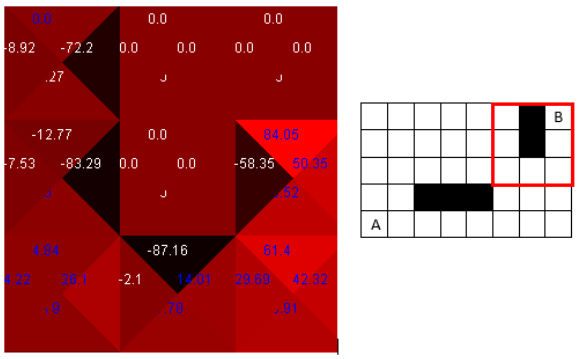

FIGURE 5. Aperçu des 9 cases du coin supérieur droit du labyrinthe après 5 minutes d'entrainement.

Par exemple, si l'agent se trouve sur la case juste en dessous de B, on a quatre actions possibles dans le tableau `qvaleur` : `qvaleur[1][7][nord]`, `qvaleur[1][7][est]`, `qvaleur[1][7][sud]`, `qvaleur[1][7][ouest]`.

On identifie également sur la figure un code couleur qui varie du noir (minimum de la fonction Q à un temps t donné) au rouge (maximum de la fonction Q à un temps t donné). Nous divisons une case du labyrinthe en quatre triangles : un pour chaque direction. Ainsi comme on peut le voir, les triangles qui mènent vers les pièges sont en noir ici (la récompense serait donc très faible si on choisit cette action), en revanche, l'action qui mène à la case B en case {1,7} est colorée en rouge : c'est elle qui maximise la récompense future. Nous avons également indiqué, sur chaque triangle, la valeur de Q à cet endroit, comme on peut le voir sur la figure 6 suivante, où, par exemple, `qvaleur[1][7][nord] = 84,05`. Nous avons également rajouté un code couleur pour les valeurs : en bleu les valeurs positives retournée par Q, en blanc les valeurs négatives.

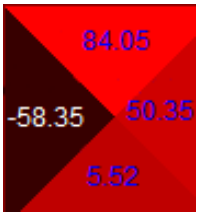

FIGURE 6. Zoom sur la case {1,7} du labyinthe.

Vous pouvez trouver en annexe A-B une capture de la vue du labyrinthe du Q-learning. On peut clairement y identifier les différents pièges (bordés de triangles noirs) et le chemin optimum bleu dont nous parlions en section III-A1 (apparaissant ici en bleu à travers les valeurs positives).

B. Réseau de neurones

1) Analyse et solution: Afin de tester le bon fonctionnement d'un réseau de neurones, il convient de l'entraîner sur des tâches basiques et d'étudier son comportement.

Les tests que nous avons effectués portaient sur les fonctions

usuelles sinus, exponentielle et logarithme, et peuvent être scindés en 2 groupes : les tests de classification et les tests d'apprentissage de fonction.
Les tests de classification consistent, pour un couple $\{x,y\}$ donné en entrée, à déterminer en sortie si ce dernier provient d'une des 3 fonctions précédemment citées.
Les tests d'apprentissage de fonction consistent quant à eux à entraîner un réseau de neurones à, pour une unique fonction donnée, fournir en sortie le y correspondant au x donné en entrée.

2) Conception: En utilisant les différentes informations obtenues à partir de l'état de l'art, nous avons pu modéliser en Java un ensemble de classes permettant de créer un réseau de neurones possédant un nombre de couches quelconque, chacune constituée d'un nombre variable de neurones se voyant attribuer une fonction d'activation (commune à tous les neurones d'une même couche).
Dans le cadre des tests réalisés sur les réseaux reproduisant des fonctions, des méthodes d'affichage permettant de donner un rendu des courbes ont été réalisées afin de mieux visualiser l'impact de facteurs tels que le nombre d'entraînements ou l'ordre de grandeur du coefficient α sur l'évolution du réseau.

C. Apprentissage par renforcement et réseaux de neurones

1) Analyse et solution: Nous avons choisi d'explorer deux pistes pour cette partie :

- L'adaptation du labyrinthe réalisé dans le cadre du Q-learning au réseau de neurones : étant donné que nous connaissons les résultats associés à ce jeu via le Q-learning, nous avons voulu voir si une adaptation au réseau donnait lieu à des observations similaires.
- Expérimentation d'un réseau de neurones sur un second jeu où l'accès à la récompense est beaucoup plus rapide. En effet le labyrinthe présente un inconvénient qui n'est pas des moindres : la récompense en haut à droite peut parfois être très longue à atteindre (étant donné que les premières itérations du réseau se basent uniquement sur les poids aléatoires), on a donc de grandes chances de se perdre au début et l'amélioration du réseau jusqu'à trouver la première fois la récompense n'est pas des plus efficiente. Nous avons donc implémenté un second jeu où l'accès à une récompense est beaucoup plus rapide et donc où le réseau s'améliorera de façon efficace sur des faits qui surviennent et se propagent plus rapidement dans le temps.

2) Conception: L'adaptation du labyrinthe a été simple à réaliser, en effet elle consiste basiquement à modifier l'algorithme de Q-learning et y insérer un réseau de neurones où les entrées peuvent prendre plusieurs formes : des scalaires ou des binaires, représentant la position du personnage dans le labyrinthe ainsi qu'un vecteur à 4 valeurs binaires, une pour chaque direction. (Une des quatre valeurs est mise à 1 pour indiquer que l'on choisit cette action là).

Pour le second jeu, nous avons choisi d'implémenter un jeu très simpliste représenté par une grille où nous avons deux éléments :

- Un agent (le joueur) qui peut se déplacer (gauche, droite ou bien rester sur place) sur la ligne du bas.
- Des objets qui tombent du haut de la grille : les bonus et les malus.

Le but de l'agent est de collecter les bonus (+1 point) et d'éviter les malus (-2 points). Lorsque le score est négatif, le joueur perd, lorsque le score est de 10, le joueur gagne la partie. Il y a **2** malus et **1** bonus qui tombent en boucle (7).

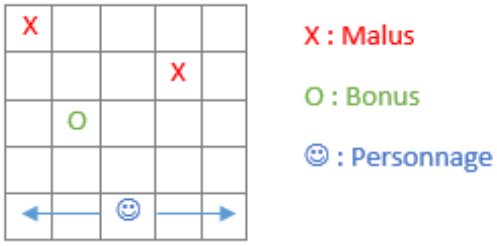

FIGURE 7. Modèle du second jeu "Falling Objects"

L'implémentation (réalisée également en Java) utilise la même logique que le jeu précédent. Ici, le tableau est de 5 par 5 cases et l'agent a le choix entre trois actions : ne rien faire, se déplacer vers la gauche ou se déplacer vers la droite.

IV. PRÉSENTATION DES RÉSULTATS

Cette partie présente les différents résultats observés lors de nos différentes expérimentations.

A. Q-learning

Le Q-learning tel que nous l'avons implémenté possède deux caractéristiques qu'il est important de distinguer afin d'observer les résultats correctement :

- La composante exploration/exploitation : Cette composante permet à l'algorithme du Q-learning de ne pas s'enfermer dans une politique et d'explorer au maximum tout l'environnement afin de dénicher une éventuelle politique omise qui s'avérerait être plus fructueuse que celle actuellement suivie. Cette composante n'a pas d'influence négative sur l'algorithme puisqu'il s'agit juste d'un choix aléatoire donné par l'algorithme, on mettra donc à jour la case de façon naturelle, comme si l'action avait été celle choisie comme étant l'action optimale.
- La composante de glisse intégrée au jeu : Cette composante est totalement indépendante du Q-learning et introduit une difficulté supplémentaire à l'algorithme qui doit composer avec ces événements aléatoires.
  En effet, étant à côté d'un piège dans un état s_{a_cote}, l'algorithme va déduire de sa fonction Q une direction a_{oppose} opposée à ce piège afin de ne pas tomber dedans. Du fait de la composante de glisse il se peut néanmoins que l'agent tombe dans le piège ; on obtient alors une récompense négative, mais l'algorithme de

mise à jour du Q-learning va mettre à jour la valeur $Q(s_{a\_cote}, a_{oppose})$ en pensant que l'action qu'il a fait l'a mené vers un piège alors que l'agent a simplement glissé. Au fil du temps, l'algorithme va donc comprendre que longer les pièges n'est pas forcément la bonne idée et va donc chercher à les contourner largement.

Nous avons donc repris les deux comportements anticipés sur la figure 4 et vérifié la convergence de l'algorithme dans les deux cas. Il s'avère qu'à chaque fois, la direction empruntée par l'agent, à terme, était bien celle prévue :

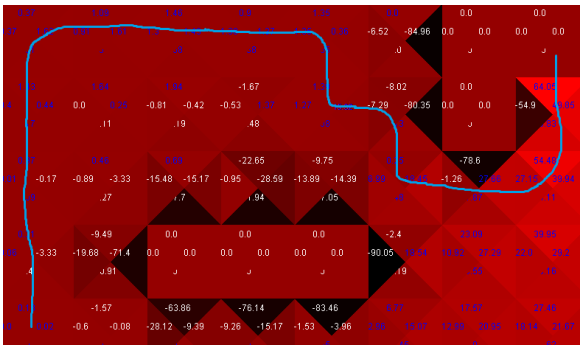

FIGURE 8. Labyrinthe avec Q-learning, lorsque la glissage est activée.

La figure 8 montre le résultat obtenu après avoir fait tourner l'algorithme pendant 10 minutes. Le chemin tracé en bleu est celui de la politique optimale finale : en choisissant à chaque fois l'action avec la Q-Valeur maximale. On observe ici, comme prévu, que l'agent effectue un détour par le haut du labyrinthe afin de minimiser le temps qu'il passe près des pièges. Néanmoins, pour atteindre l'arrivée, il doit se résoudre à passer dans une brèche qui est entourée de deux pièges, chose qu'il fait avant d'atteindre son objectif.

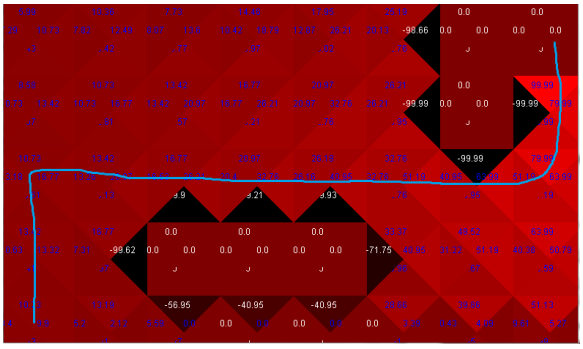

FIGURE 9. Labyrinthe avec Q-learning, lorsque la glissade est désactivée.

Si l'on désactive la glissade, on obtient la figure 9, on voit que la politique finale n'hésite pas à longer les pièges, puisque c'est sans risque pour elle dans cette configuration. On voit qu'ici, l'agent a très rapidement trouvé le chemin qui mène à B en passant par le haut. Si on laisse suffisamment tourner l'algorithme, la composante exploratoire permet d'obtenir le même résultat mais avec un résultat où l'agent passe vers le bas. Le chemin passant vers le bas serait rapidement trouvé si nous avions intégré une notion de temporalité dans le jeu (atteindre la récompense le plus rapidement possible, en un minimum d'états).

D'autre part, nous avons pu également observer qu'un α avec une valeur fixe comme $0,1$ n'est pas forcément la meilleure des solutions, en effet, après avoir trouvé une politique optimale comme en figure 8, il arrive que l'algorithme régresse (notamment à cause de la glissade justement) et revienne à un état intermédiaire où il cherchait encore la politique optimale, puis on revient à l'état final, etc... Il est alors pertinent d'opter soit pour un α plus petit (afin de limiter l'impact d'une récompense sur la Q-Valeur, mais cela ralentit l'algorithme, il faut plus d'entraînements) soit pour un α dégressif qui diminue au fur et à mesure de l'avancement de l'agent. (On peut par exemple baisser α en fonction du nombre de fois que l'agent arrive effectivement à la fin sur la case B).

B. Réseau de neurones

La structure qui a été utilisée pour les réseaux de neurones lors des tests a été choisie de façon à tester :

— la rétropropagation du gradient (i.e. le réseau doit posséder au moins une couche cachée)
— le comportement du réseau selon la nature des données attendues sur la couche de sortie (i.e. selon la fonction d'activation de la dernière couche)

Il a ainsi été décidé d'utiliser un réseau possédant deux couches cachées de 40 neurones chacune, et d'utiliser la fonction sigmoïde comme fonction d'activation sur toutes les couches, sauf sur la dernière (qui dans les cas où l'on attend une sortie réelle peut être la fonction identité).
Le mode opératoire des tests a été le même dans tous les cas : le réseau est mis en place et ses poids sont initialisés aléatoirement (à valeurs dans l'intervalle $[-1, 1]$), puis le réseau est entraîné à une unique tâche spécifique, en prenant des valeurs d'entrée aléatoires (dans les limites d'intervalles fixés). Par exemple l'entraînement du réseau devant reconnaître la fonction sinus a consisté en 500 000 répétitions de :

1) Sélectionner aléatoirement un entier $x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, puis calculer $y = sin(x)$ de manière à former un vecteur d'entrée $\{x,y\}$.
2) Faire évaluer ce vecteur d'entrée par le réseau de neurones.
3) Effectuer une rétropropagation du gradient pour mettre à jour les poids du réseau.

Hormis le fait que les différents tests ont permis de valider le bon fonctionnement du réseau de neurone, ils ont permis de mettre à jour certains points intéressants notamment dans le cas des tests portant sur l'apprentissage de fonctions :

1) Variation sur l'intervalle de sortie: Les tests portant sur la fonction exponentielle ont mis en exergue le lien entre le coefficient α (utilisé lors de la rétropropagation), et l'intervalle dans lequel se trouvent les valeurs de sortie.
Les images obtenues (figures 10,11,12) montrent la courbe

obtenue par les réseaux après l'avoir entraîné 50 000 fois sur un intervalle dont on a légèrement fait varier la borne supérieure tout en conservant le même α.

On peut ainsi observer qu'avec un α constant et un intervalle

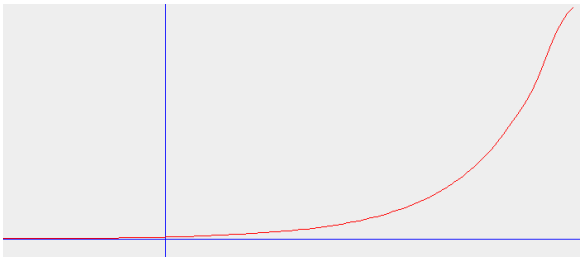

FIGURE 10. $x \in [-2;5]\,, \alpha = 0,001$

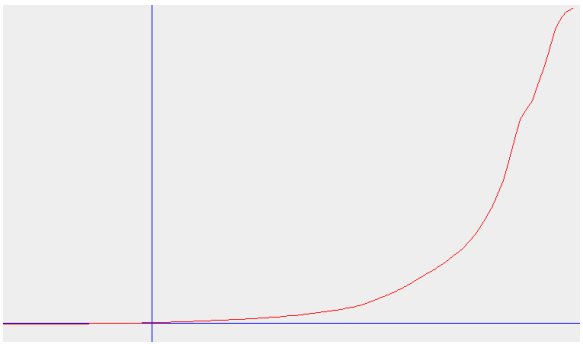

FIGURE 11. $x \in [-2;5,6]\,, \alpha = 0,001$

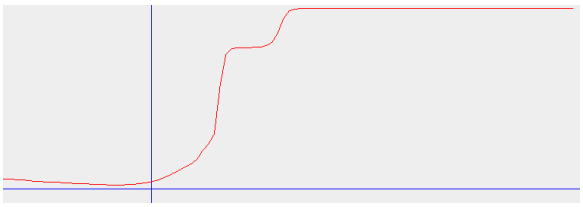

FIGURE 12. $x \in [-2;5,65]\,, \alpha = 0,001$

de sortie croissant, le réseau finit par saturer.
α doit donc être fixé en fonction de l'ordre de grandeur des valeurs que l'on attend en sortie.
D'autres tests ont aussi montré qu'à domaines d'entrée/sortie égaux, pour α fixé le réseau tendait à saturer si l'on augmentait le nombre de couches cachées (α doit donc être d'autant plus petit que le réseau est profond).

Sur la figure 13, pour un α 10 fois moindre que celui précédemment utilisé, on observe que le réseau n'a pas eu de difficultés à apprendre la fonction exponentielle sur un intervalle supérieur à celui précédemment utilisé.
En outre ces tests ont permis de montrer que les valeurs des poids du réseau convergeaient vers une valeur précise lorsque le nombre d'entraînements devient grand (dans notre cas 500 000), l'allure de la courbe n'évoluant plus dès lors que la barre des 100 000 est passée (dans le cas d'une saturation la fonction

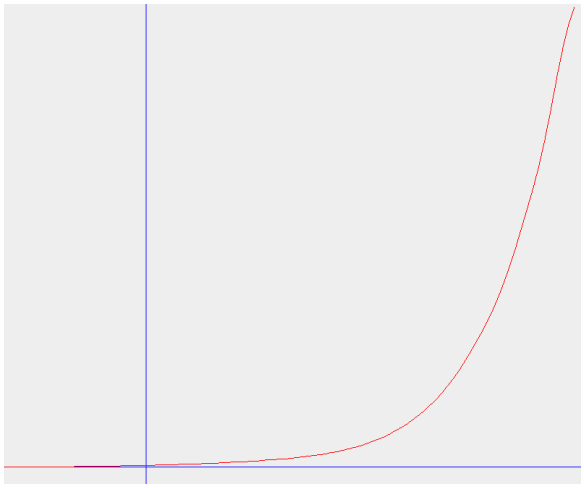

FIGURE 13. $x \in [-2;6,0]\,, \alpha = 0,0001$

représentée par le réseau tend alors vers une droite d'équation $y = cste$).

C. Apprentissage par renforcement et réseaux de neurones

1) Remarques generales: Nous avons dans un premier temps voulu tester le bon fonctionnement de l'algorithme du Q-learning en lui appliquant le même modèle de labyrinthe utilisant un tableau pour approcher la fonction de récompense (le but étant d'arriver au même résultat).
Cela nous a permis de mettre à jour de façon empirique plusieurs observations générales quant au comportement des réseaux de neurones connexionistes.
Il faut notamment mettre en avant le temps extrêmement long qui a dû être alloué à la réalisation des tests, du fait de plusieurs facteurs :

1) Plus le problème traité est complexe, plus le réseau associé l'est aussi, que ce soit du point de vue du nombre de neurones par couche que le nombre de couches lui-même (et donc : un nombre de poids plus important). De ce simple point de vue, les calculs liés à la rétropropagation de l'erreur s'en retrouvent décuplés. Ce problème n'a pas vraiment eu d'impact lors de nos tests (le réseau utilisé possédant environ 3200 poids), mais il faut le prendre en compte notamment lorsqu'on veut faire du traitement d'image (par exemple pour une image en nuances de gris de 128*128 pixels, la présence d'une unique seconde couche de 40 neurones fera monter le nombre de poids à 655360)
2) Plus le réseau est profond, plus le facteur α de la descente de gradient devra être faible afin d'éviter une saturation de la fonction d'activation (cf IV-B). Cette contrainte induit un temps de convergence plus long du réseau vers un état d'équilibre.
3) On n'échappe pas non plus au problème posé par la présence de minimum locaux (cf II-D) dans la fonction qu'est chargé de modéliser son réseau, il n'est donc pas exclu que cela fausse son entraînement.

Cela est à mettre en parallèle avec la modélisation par un tableau simple : si le réseau de neurones permet d'approcher des modèles plus complexes, on a pu constater qu'il n'est pas intéressant de l'appliquer aux problèmes qu'une modélisation plus simple peut déjà résoudre.
En effet la modélisation par tableau permet de résoudre le problème du labyrinthe en 1 minute, à l'opposé de celle par réseau de neurones qui nécessite une journée d'entraînement, et ce dans les cas qui sont concluants.

De plus, l'utilisation d'une graine aléatoire a un impact sur l'évolution du réseau :
Elle peut orienter de façon très importante son comportement dès sa création ; par exemple lors de nos tests (sur le modèle du labyrinthe), une des graines générées avait dès le début permis d'obtenir un taux de succès de 70%.
Mais si cela a un effet positif dans certains cas, cela dépend purement du hasard et ne peut pas être reproduit d'une session d'entraînement à une autre.
C'est d'autant plus problématique dans le cas de figure inverse, lorsque la graine initiale conduit le réseau à avoir un comportement qui ne correspond pas du tout à l'attendu, et qui coûtera un temps de calcul conséquent à la seule fin de corriger les poids du réseau (pour les ramener à un état "neutre").
Conjointement à la façon dont la fonction de récompense est mise en place, cela peut poser un problème de **surentraînement**.
En effet, toujours dans l'exemple du labyrinthe, là où le Q-learning nous fournissait un tableau renvoyant 0 lorsque le score de la case était nul, le biais induit par la graine initiale peut empêcher le réseau d'explorer certaines parties de la carte (ce même avec une composante exploratoire forte), ce qui va conduire le réseau non pas à aller vers la fin du labyrinthe (qu'il ne trouvera quasiment jamais), mais à se coincer dans les coins de la carte. Comme les seuls stimulus qu'il recevra seront négatifs (en tombant dans les cases piégées), il va se spécialiser dans cette tâche et il sera d'autant plus difficile pour lui de corriger ce comportement à mesure que le nombre d'entraînements augmentera.
Lors des tests nous avons notamment rencontré des cas de figure où le réseau ne se concentrait plus que sur le seul fait l'aller dans le coin supérieur gauche du labyrinthe. Cela est d'autant plus problématique que, la condition pour passer d'un entraînement à un autre étant soit la mort en tombant dans un trou soit le succès en arrivant sur la case de fin, il fallait parfois plusieurs secondes pour passer d'un entraînement à un autre (la fin de l'entraînement étant réalisée grâce à la composante exploratoire qui permettait au réseau de s'échapper du coin).
Comme l'entraînement du réseau prenait en moyenne 5 à 10 millions d'itérations pour dégager un comportement similaire à celui observé avec le Q-learning, il n'était pas rare de lancer l'entraînement en début de matinée pour s'apercevoir en fin de journée que ce dernier s'était enfermé dans ce type de comportement, devenant inutilisable.

2) Analyse des résultats pour le labyrinthe: Plusieurs formats d'entrée ont été testés afin d'observer leur impact sur l'apprentissage :
Note : en raison du temps de convergence important des réseaux, la glissade dans le labyrinthe a été désactivée.

1) **Vecteur binaire** :
   Le premier format qui a été testé consiste à passer en entrée un vecteur de longueur égale au nombre d'états possible (i.e. au nombre de cases dans le tableau), dont tout les éléments étaient égaux à 0, sauf pour celui représentant la position du réseau dans le labyrinthe, égal à 1.
   Ici les résultats sont concluant et on a pu observer une convergence des valeurs retournées par le réseau vers celles retournées par le tableau du Q-learning.
   Ces résultats ont été obtenus pour un réseau de dimensions 44-40-40-1, la couche d'entrée étant le fruit le la concaténation entre le vecteur ligne de 40 éléments représentant l'état courant du modèle et un vecteur ligne de 4 éléments (qui de manière similaire a tout ses éléments à 0 sauf celui représentant l'action à effectuer, qui est à 1).
   Le nombre d'entraînements nécessaire pour atteindre la convergence se situe vers 10 millions.
   Ci-après une capture 15 de l'état du réseau au terme de son entraînement : Comme on peut le voir, son allure

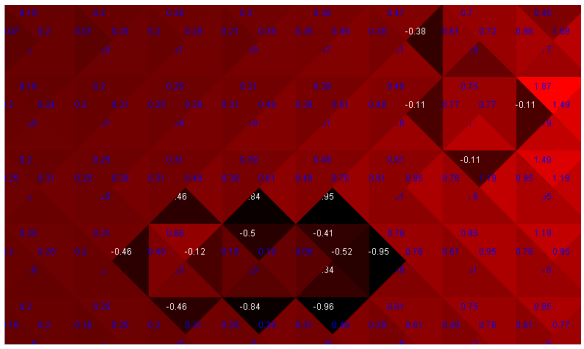

FIGURE 14. Réseau avec entrées binaires

   générale rejoint celle obtenue avec le Q-learning.

2) **Vecteur de coordonnées** :
   Le second format que nous avons testé consiste à modéliser la position dans le labyrinthe par un couple $\{x,y\}$.
   L'utilisation d'un réseau de structure identique à celui utilisé précédemment n'ayant pas été concluante, nous avons alors tenté d'y apporter plusieurs modifications (ajout d'une couche cachée supplémentaire, variations du α), mais sans succès (avec un taux de réussite approchant de 0 après 10 millions d'entraînements).
   Afin de vérifier qu'un réseau prenant un tel vecteur d'entrée existait bien, nous avons réalisé un réseau intermédiaire dont le but était d'associer à une entrée $\{x,y\}$ une sortie équivalente au format de l'entrée binaire

utilisée par le précédent réseau de neurones.
Cette expérience a été concluante et nous avons obtenu un taux de reconnaissance de 100% avec un réseau de la forme 2-40-40, après 5 millions d'entraînements et un α de 0,01.
Un autre test a été effectué avec un réseau de la forme 2-40 (sans couche cachée), mais n'a pas été concluant. Cela permet donc de prouver l'existence d'un réseau fonctionnel de la forme 2-...-1, dont la forme la plus simple (quoique strictement moins efficace que le réseau à entrées binaires) consiste en la concaténation du présent réseau intermédiaire avec le réseau à entrées binaires (les sorties du premier constituant les entrées du second).
Il est néanmoins difficile de mettre en place un tel réseau du fait du nombre important de couches cachées qu'elle possède, dû aux contraintes qu'elle induit en terme d'α (rendant la convergence très longue).
Une solution a finalement été trouvée, cette dernière consistant d'une part, à conserver un nombre de couches égal à la version binaire tout en portant le nombre de neurones par couches cachées à 160, et d'autre part à maintenir une composante exploratoire de 100% sur les 100000 premiers entraînements, afin de "gommer" tout biais lié à l'initialisation aléatoire des poids. Cette augmentation du nombre de poids a évidemment eu un coût, portant à 30 heures le temps nécessaire pour effectuer un million d'entraînements.
On peut observer que le réseau obtenu 15 est sensiblement similaire à son homologue binaire :

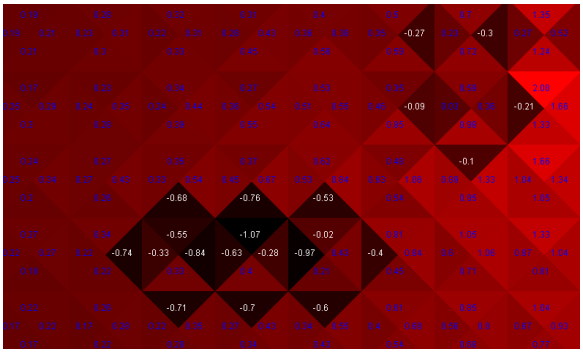

FIGURE 15. Réseau avec entrées cartésiennes

3) Deuxième jeu : "Falling Objects": Comme expliqué dans la partie Méthodologie, ce deuxième jeu nous permet d'entraîner un réseau de neurones sur un modèle où l'atteinte d'une récompense probante est beaucoup plus rapide que sur notre précédent labyrinthe. Ci-dessous les résultats que nous avons obtenu en fonction de plusieurs configurations possibles :

Cas 1 : Vecteur d'entrée image du jeu
Le premier réseau (2 couches intermédiaires de 40 neurones) a été entraîné avec une image du jeu en entrée : le réseau avait accès au "champ de vision" de l'agent (les trois dernières lignes tableau de jeu, voir figure 16). Le vecteur d'entrée du réseau comporte 3 parties : l'image du champ de vision, la position de l'agent (représenté par 5 valeurs binaires correspondant aux 5 cases de la ligne où se déplace l'agent : on met 1 pour désigner la position de l'agent) ainsi que 3 valeurs binaires pour l'action à effectuer (gauche, droite ou ne rien faire), donc $3 * 5 + 5 + 3 = 23$ valeurs d'entrée.

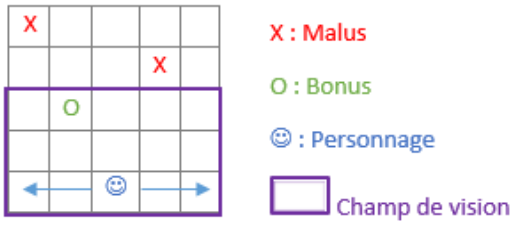

FIGURE 16. Champ de vision de l'agent

Dans le champ de vision, un malus est représenté par un **2**, un bonus un **1** et une case vide un **0**.
Avec une *exploration de 1/5* (une décision d'action aléatoire 1 fois sur 5), le réseau de neurones parvient à gagner avec une probabilité 1/2 au bout de 50 000 entraînements. Cette valeur de 1/2 stagne après ce pallier atteint, en effet la composante aléatoire peut conduire l'agent à se diriger sur un malus lorsque son score est encore faible et ainsi perdre la partie.
Avec une *exploration de 1/10*, le réseau de neurones parvient à gagner au jeu avec une probabilité 1/2 au bout de 20 000 entraînements. Au bout de 50 000 entraînements, on atteint une probabilité 8/10 de gagner le jeu, qui stagne, encore une fois à cause de la composante exploratoire.

Cas 2 : Vecteur d'entrées position des objets

Dans une deuxième version, nous avons modifié le vecteur d'entrées pour que celui-ci possède uniquement les scalaires associés aux différentes coordonnées $\{x,y\}$ des objets qui tombent et du personnage, on a ainsi un vecteur de la forme : `[X-Malus1, Y-Malus1, X-Malus2, Y-Malus2, X-Bonus, Y-Bonus, X-Agent, Y-Agent, ActionStatique, ActionGauche, ActionDroite]`
(les actions elles, sont toujours binaires)
Les résultats obtenus sont les suivants :
Toujours avec une *exploration de 1/10* et *2 couches intermédiaires de 40 neurones*, nous obtenons le résultat suivant au bout de 50 000 entraînements : le jeu est terminé par le réseau avec un taux de succès de 1/10. Après 100 000 entraînements, on retombe à une probabilité de 0.
Nous avons donc tenté de modifier le réseau en augmentant le nombre de neurones des couches intermédiaires sans changer la profondeur du neurones. (Un changement du nombre de couches aurait impacté le coefficient α utilisé lors de la rétropropagation). Nous avons maintenant un réseau à deux couches intermédiaires de *80 neurones* chacune. Autour de

50 000 itérations, nous avons une probabilité de 2/10 de finir le jeu. A 65 000 itérations, le réseau en est à une probabilité de 1/2. Enfin, au bout de 120 000 itérations, nous avons un taux de réussite de 8/10. Notons que pour ce deuxième réseau de 80 neurones par couche intermédiaire, les itérations sont beaucoup plus longues (3 voire 4 fois plus de temps est nécessaire), ceci était dû au fait qu'il y a plus de poids à mettre à jour.

Cette expérience nous a permis de constater plusieurs choses :

- La facilité d'accès à une première récompense semble conditionner l'apprentissage du réseau, contrairement au labyrinthe, nous avons ici des réseaux qui s'entraînent beaucoup plus rapidement et fournissent des résultats concluants.
- La composante exploratoire permet effectivement d'améliorer le réseau de neurones mais peut également le desservir : lorsque l'on tend à établir une stratégie idéale, on a toujours cette part d'aléatoire qui peut nous faire échouer. Il conviendrait donc d'opter pour une exploration décroissante : au début lorsque l'on connaît peu de chose, on s'accorde à beaucoup explorer, puis on réduit cette part au fur et à mesure que l'on connaît mieux l'environnement.
- L'information que l'on fournira au réseau de neurones pour son entraînement est très importante. En effet on voit dans notre cas 2 que la présence des scalaires n'empêche certes pas le réseau de s'améliorer mais ralentit grandement son apprentissage, là où l'image du jeu est très efficace dans le cas 1. Le ralentissement est en partie dû au fait que nous devons augmenter le nombre de poids (soit en augmentant le nombre de couches, soit le nombre de neurones par couche intermédiaire comme on l'a fait ici), il y a donc plus d'éléments à mettre à jour et cela prend plus de temps.
- Le deuxième cas possède un vecteur d'entrée plus petit (23 en cas 1 contre 11 en cas 2), on a donc moins de poids qui impactent l'issue du réseau. Ceci explique sans-doute l'atteinte d'un minimum local lors de notre premier essai, suivi de la régression immédiate du réseau. On a pu observer que l'ajout de neurones dans les couches intermédiaires a permis de corriger le problème.

V. Conclusion

En conclusion, nous avons donc pu faire différentes observations en trois temps.

D'abord sur le Q-learning, nous avons expérimenté l'importance de l'environnement sur la fonction Q (activation de glissade, les pièges, ...), les paramètres α et γ semblent être également des points très important et en particulier α dans notre cas qui peut faire régresser la fonction à cause de la glissade. Seulement le Q-learning est réservé à des espaces discrets et nous avons pour cela été contraints d'axer nos recherches sur les réseaux de neurones permettant de prendre en compte des environnements non discrets.

Ensuite, nous avons exploré différents aspects des réseaux de neurones allant de la théorie à la pratique. À travers de petits exercices de reconnaissances de fonction et de formes, nous avons pu apprécier l'importance ainsi que la mécanique du coefficient α dans la rétropropagation.

Finalement, nous avons substitué le tableau d'états par un réseau de neurones connexioniste dans l'exercice du labyrinthe. Ce premier essai a permis de mettre en exergue différents faits sur de tels les réseaux : leur complexité et le temps de calcul nécessaire, l'importance du choix d'α dans la convergence, problème des minimums locaux dans la descente de gradient. Notre premier jeu a également permis de soulever le problème d'accès à la récompense : lorsque ce dernier est conditionné par de l'aléatoire trop fort, il est parfois nécessaire d'entraîner le réseau pendant un temps considérable.

C'est pourquoi nous avons expérimenté un second jeu où l'accès à cette récompense est beaucoup plus rapide et certain, les résultats ont permis de révéler l'importance de l'accès à la récompense, du choix des entrées du réseau, de l'impact de la composante exploratoire et de la structure interne du réseau elle-même qui influe grandement sur les résultats obtenus.

Pour terminer, un type de réseau qui n'a pas été testé dans le cadre de ce projet mais qui est très adapté au traitement d'image sont les réseaux dits de **convolution**
Ces derniers relèvent de l'apprentissage profond, et permettent d'éviter l'explosion combinatoire que l'on observe au niveau du nombre de poids dans les réseaux entièrement connectés.
Le principe est le suivant : le vecteur d'entrée du réseau représente les pixels de l'image a traiter, et chaque neurone des couches suivantes du réseau de convolution est lié à une zone spécifique de l'image, i.e. il ne prend pas la totalité des sorties de la couche précédente en entrée.
De façon simplifiée, chaque neurone d'une même couche prend en entrée le même nombre n de variables que ses pairs, et l'ensemble n des poids associés à ces entrées est le même pour tous les neurones d'une même couche.
Cet ensemble de poids est assimilable à un filtre de convolution, similaire à ceux utilisés dans le cadre du traitement d'images (filtre de Sobel, etc...). De cette manière un réseau de convolution prenant une image de dimensions 128*128 en entrée et disposant d'une seconde couche de 40 neurones ne possédera en tout et pour tout que 100 poids si le filtre que l'on décide d'utiliser (donc la taille de chaque zone associée à un neurone de la seconde couche) est de dimension 10*10.
Plus la couche d'un réseau de convolution est profonde, plus elle traite d'éléments globaux associés à l'image en entrée, l'opérateur de convolution permettant de dégager couche après couche les éléments pertinents dégagés à partir du résultat de la couche précédente.
Ce réseau de convolution est en général suivi par un réseau

entièrement connecté classique qui s'occupe de traiter les données issues de la dernière couche du réseau de convolution (et opérant par exemple une classification sur l'image). Il est à noter qu'ici l'algorithme de rétropropagation du gradient est plus complexe que dans le cas connecté [11].

Remerciements

Nous souhaiterions dans un premier temps adresser notre profonde reconnaissance à MM. Olivier Buffet et Vincent Thomas pour leurs précieuses indications, leur accueil au laboratoire ainsi que leur encadrement tout au long de ce projet de recherche. Le sujet était très intéressant et nous a été très bénéfique à tous les deux. Nous remercions également Télécom Nancy de nous avoir permis de collaborer avec deux chercheurs du LORIA sur ce projet.

Références

[1] F. Woergoetter and B. Porr, *Reinforcement Learning*, Article sur Scholarpedia.org, 2008

[2] C. J. Watkins, *Learning from delayed rewards*, Kings College, Cambridge, mai 1989

[3] R Sutton et A. Barto, *Reinforcement Learning : An Introduction* , MIT Press, 1998

[4] B. Bouzy, *Descente de Gradient*, UFR Math-Info de Paris, 2005

[5] V. Mnih, K Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller *Playing Atari With Deep Reinforcement Learning* Dans NIPS Deep Learning Workshop, 2013

[6] Developpez.com - Article sur l'intelligence artificielle et les réseaux de neurones

[7] Wikipedia.org - Q-learning, Apprentissage par renforcement, La dérivée partielle, la rétropropagation du gradient.

[8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel et D. Hassabis, *Mastering the game of Go without human knowledge*, article de Nature.com, Vol. 550, Octobre 2017

[9] M. Nielsen, *Using neural nets to recognize handwritten digits*, Chap. 1 du e-book Neural Networks and Deep Learning, Determination Press, 2015

[10] A. Irpan, *Deep Reinforcement Learning Doesn't Work Yet*, Note de blog, alexirpan.com, 2018

[11] G. Gwardys, *Convolutional Neural Networks backpropagation : from intuition to derivation*, Note de blog, grzegorzgwardys.wordpress.com, 2016

ANNEXE A
Q-LEARNING

A. Formule de mise à jour de la fonction de valeur

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{actuel}} + \underbrace{\alpha}_{\text{vitesse d'apprentissage}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{recompense}} + \underbrace{\gamma}_{\text{facteur d'actualisation}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{valeur optimale estimee}}}^{\text{valeur apprise}} - \underbrace{Q_t(s_t, a_t)}_{\text{actuelle}} \right)$$

B. Aperçu complet du labyrinthe du Q-learning

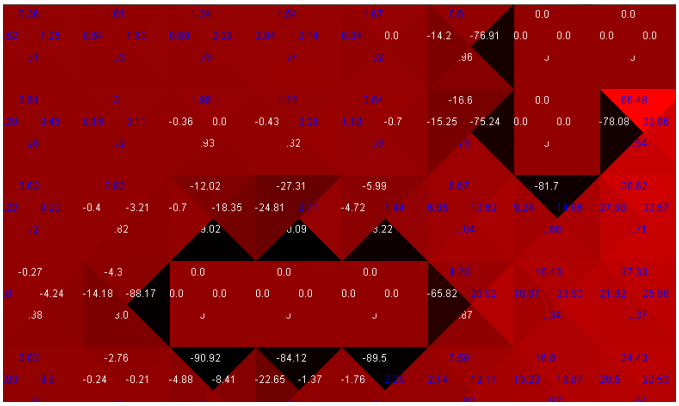

FIGURE 17. Labyrinthe du Q-learning complet