CentraleSupélec

Rapport de projet

Ready Player One : Deep Reinforcement Learning

Étude des réseaux neuronaux, puis application au développement d'une intelligence artificielle pour des jeux Atari

2018 – 2019

Nathan Cassereau
Paul Le Grand Des Cloizeaux
Thomas Estiez
Raphaël Bolut

Professeurs encadrants :
Joanna Tomasik
Arpad Rimmel

Établissement :
CentraleSupélec cursus Supélec (promotion 2020)

Table des matières

Table des figures

Introduction

Le projet long Ready Player One a pour but d'étudier le fonctionnement d'algorithmes d'apprentissage automatique. Cette étude, orientée recherche et développement, cherche à appliquer une branche du machine learning, le Q-Learning, à l'intelligence artificielle (IA) du jeu vidéo Pong. Cette IA se formera par elle-même sur ce jeu.

Le projet est mené par deux groupes de quatre étudiants, afin de pouvoir comparer les performances des deux produits finaux. Notre groupe, nommé « Éponge », est composé de Nathan Cassereau, Raphaël Bolut, Thomas Estiez, et Paul Le Grand Des Cloizeaux.

Les deux groupes sont encadrés par Joanna Tomasik et Arpad Rimmel, qui nous guident et nous donnent des pistes pour assurer l'avancée du projet, et à qui nous rendont compte chaque semaine du travail réalisé.

L'étude du projet se fait en plusieurs parties. Comme la tâche à réaliser est importante, et que le projet a pour but de nous apprendre les mécanismes du machine learning, nous étudierons plusieurs algorithmes différents au cours de l'année, dont nous expérimenterons le fonctionnement. Les différents codes utilisés lors de ce projet sont consultables sur GitHub.

Dans un premier temps, nous allons étudier le fonctionnement du perceptron, un réseau de neurones basique, que nous allons entraîner à la reconnaissance de chiffres manuscrits de la base de données MNIST de Yann LeCun. Cette première étude a pour but de nous faire comprendre le fonctionnement global du machine learning, et les différents mécanismes d'optimisations utilisés.

Puis nous étudierons les réseaux neuronaux à convolution, version améliorée du perceptron. Ces réseaux sont particulièrement adaptés à l'analyse de certaines données comme les images en couleur.

Nous allons ensuite rentrer dans le vif du sujet : Le Q-learning puis le Deep Q-Learning, appliqué au jeu vidéo Pong. Nous allons pour cela nous interfacer avec une bibliothèque Python grâce à laquelle notre algorithme pourra agir sur le jeu. Afin de nous faciliter la tâche, nous utiliserons l'outil TensorFlow (bibliothèque Python), qui permet de faire des calculs de machine learning de façon optimisée et de nous affranchir des difficultés d'implémentation posées par le réseau à convolution et sa rétropropagation.

1 Le Perceptron

1.1 Modèle du perceptron

Le perceptron est un des algorithmes de base du machine learning. Son invention remonte aux années 70, mais l'algorithme a été abandonné en raison de son exécution trop coûteuse pour les performances des ordinateurs de l'époque. Ce n'est que récemment qu'il a pu resurgir, grâce à l'amélioration des processeurs et des cartes graphiques, particulièrement adaptées aux calculs matriciels.

Le modèle du neurone est le suivant :

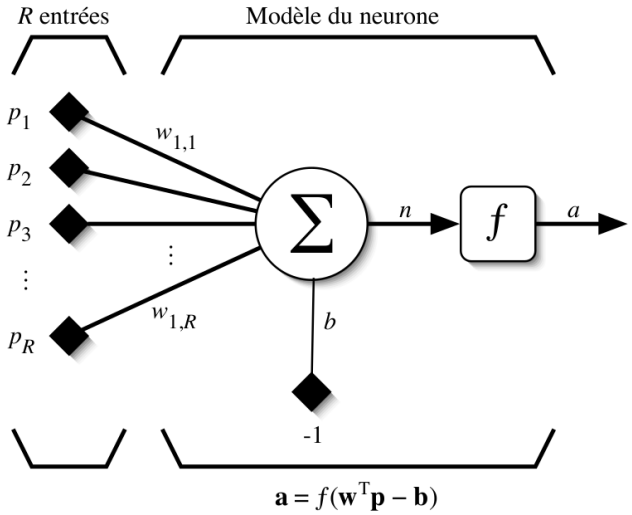


Figure 1: Modèle d'un neurone artificiel

Le neurone est composé de différents élements :

- p_1, p_2, ..., p_R constituent les R variables d'entrées du perceptron. Le nombre d'entrées est souvent imposé par le système lui-même.
- $w_{1,1}$, $w_{1,2}$, ..., $w_{1,R}$ sont les poids associés respectivement à chaque entrée. Ils mesurent l'importance accordée à chaque entrée. Un poids plus important signifie que l'entrée associée est plus pertinente pour ce neurone que les autres.
- Le biais b
- Le niveau d'activation n
- La fonction d'activation f
- La sortie a

Ainsi, on associe les entrées et les poids par un produit scalaire, pour en sortir une valeur qui caractérise l'entrée, le niveau d'activation. On ajoute un biais pour régler l'importance accordée au niveau d'activation. On peut utiliser une notation matricielle pour simplifier les calculs. On pose alors $\mathbf{w_1} = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \end{pmatrix}^T$ et $\mathbf{p} = \begin{pmatrix} p_1 & p_2 & \dots & p_R \end{pmatrix}^T$, les vecteurs colonnes représentant respectivement les entrées et les poids du neurone. On a alors :

$$n = \sum_{i=1}^{R} w_{1,i} p_i - b = \mathbf{w_1}^T \mathbf{p} - b \tag{1}$$

On cherche alors à discriminer les différentes possibilités pour le niveau d'activation. C'est le rôle de la fonction d'activation. Si l'on souhaite séparer le cas d'un n supérieur ou non à un seuil donné, alors on utilise la fonction seuil $f : x \mapsto \mathbb{1}_{n \geq 0}$. On remarquera qu'il n'est pas nécessaire de changer le seuil de la fonction car c'est le rôle incarné par le biais. Néanmoins, d'autres fonctions peuvent être utilisées à la place du seuil telles que la sigmoide ($\sigma : x \mapsto \frac{1}{1+e^{-x}}$) ou encore la tangente hyperbolique. On préfère généralement des fonctions différentiables pour permettre au réseau d'apprendre sur les données fournies.

On a alors :

$$a = f\left(\mathbf{w_1}^T\mathbf{p} - b\right) \tag{2}$$

Si on revient au cas de la fonction seuil, on remarquera qu'elle permet de séparer le plan en deux espaces : l'un où la sortie est nulle, l'autre où la sortie est égale à 1. Puisque le niveau d'activation résulte d'un produit matriciel, cela définit l'équation d'un hyperplan, la séparation est donc linéaire.

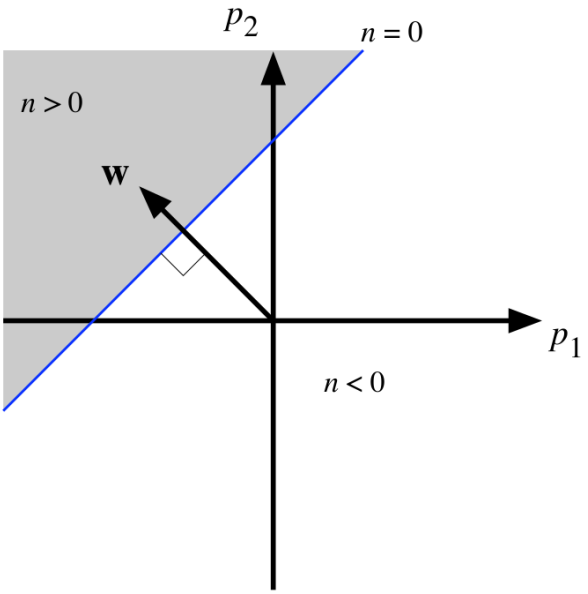


Figure 2: Domaine de séparation du neurone

Ce neurone n'est capable de traiter que les données qui peuvent être séparées linéairement (par un hyperplan). Pour des jeux de données plus complexes, on a parfois besoin de définir des ensembles plus élaborés. Pour cela on utilise plusieurs neurones sur une même couche. Tous les neurones reçoivent la même entrée, mais chacun possède ses propres poids et son propre biais. Ainsi, chaque neurone de la couche définit un hyperplan de séparation des données. On peut alors à nouveau représenter le modèle de manière matricielle. Un vecteur de sortie définit les différentes valeurs des neurones, une matrice de poids définit les poids pour chaque neurone (à chaque neurone est associée une ligne de la matrice). De la même manière, on retrouve un vecteur de biais (qui sont essentiellement des poids dont l'entrée est constante à -1), et un vecteur de niveaux d'activation. Finalement, on retrouve ce modèle :

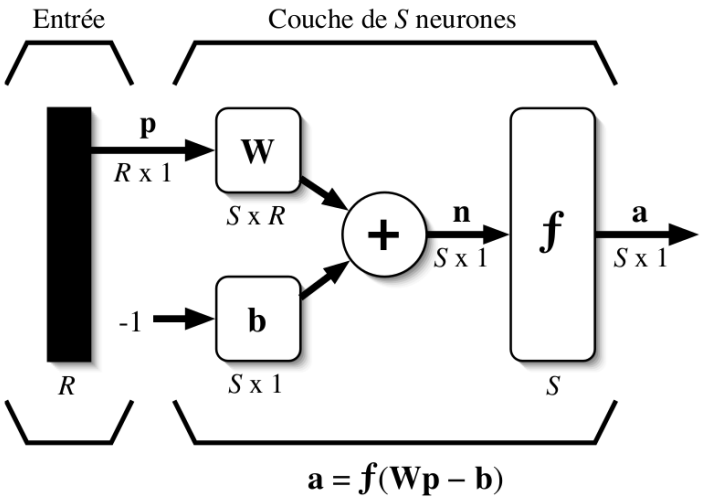


Figure 3: Représentation matricielle d'une couche de S neurones recevant R entrées

Pour pouvoir définir des ensembles de solutions plus complexes, on ajoute d'autres couches de neurones. Chaque couche prend en entrée le vecteur de sortie de la couche qui la précède. Cela permet donc de traiter

les différents hyperplans de la première couche, et de les lier (par exemple pour en faire l'intersection). Ainsi, avec deux couches, le réseau peut représenter n'importe quel ensemble convexe. Une troisième couche permet de représenter des ensembles non convexes.

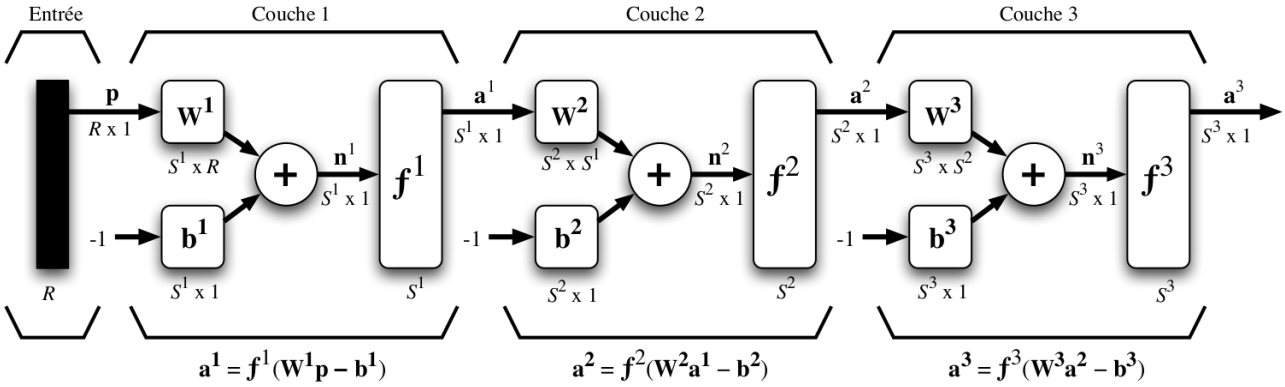


Figure 4: Modèle du perceptron multicouches

1.2 Apprentissage

L'intérêt du modèle serait limité s'il devait être calculé manuellement. L'objectif est d'avoir un algorithme qui trouve lui-même les paramètres du réseau (poids et biais) pour s'adapter à un jeu de données collectées au préalable. Il existe plusieurs méthodes pour réaliser l'apprentissage. Dans le cas du perceptron, on utilise souvent un apprentissage supervisé. Cela signifie que les données collectées contiennent la "bonne" réponse pour que le réseau puisse apprendre en conséquence. D'autres méthodes comme l'apprentissage non supervisé existent, ce dernier reposant uniquement sur les jeux d'entrées ; dans ce cas, le réseau doit les discriminer lui-même sans connaître la "bonne" réponse.

Pour réaliser cela, on présente à notre réseau une entrée ; dans la mesure où l'on dispose de la sortie attendue, il est possible de quantifier l'erreur faite par le réseau. C'est le rôle de la fonction d'erreur. Plus celle-ci est importante, moins le réseau est adapté pour cette donnée. Il existe différentes fonctions d'erreur. Une des plus utilisées est la somme des carrés des écarts entre la valeur attendue et la valeur calculée :

$$F(\mathbf{x}) = \mathbf{e}(\mathbf{x})^T \mathbf{e}(\mathbf{x}) \tag{3}$$

où $\mathbf{e}(\mathbf{x}) = \mathbf{d}(\mathbf{x}) - \mathbf{a}(\mathbf{x})$, $\mathbf{d}(\mathbf{x})$ la valeur attendue et $\mathbf{a}(\mathbf{x})$ la valeur calculée.

L'apprentissage consiste donc en la minimisation de cette fonction de coût F. À chaque calcul d'erreur, on modifie les différents poids du réseau. À une couche k donnée, le poids entre l'entrée j et le neurone i est modifié de la manière suivante :

$$\Delta w_{i,j}^k(t) = -\eta \frac{\partial F}{\partial w_{i,j}^k} \tag{4}$$

En se plaçant dans l'espace des poids (cela inclut les biais qui sont des poids particuliers), cela revient à chercher la direction dans laquelle l'erreur est diminuée de la manière la plus significative. Le facteur η est le taux d'apprentissage (Learning Rate en anglais). Il représente le pas de chaque itération vers le minimum de la fonction de coût. C'est un paramètre du réseau que nous devons choisir en amont de l'apprentissage.

Marc Paruzeau démontre en 2004 les formules de rétropropagation que nous avons utilisées. Pour cela il introduit un paramètre intermédiaire. Les sensibilités sont définies ainsi :

$$\mathbf{s}^k = \frac{\partial F}{\partial \mathbf{n}^k} \tag{5}$$

On note également l'utilisation du raccourci suivant :

$$\dot{\mathbf{F}}^k(\mathbf{n}^k) = \begin{bmatrix} \dot{f}^k(n_1^k) & 0 & \dots & 0 \\ 0 & \dot{f}^k(n_2^k) & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \dot{f}^k(n_{S^k}^k) \end{bmatrix} \text{ où } S^k \text{ est le nombre de neurones de la couche } k \tag{6}$$

Pour un réseau de M couches, la rétropropagation se déroule de la manière suivante :

- On propage notre entrée $\mathbf{p}$ dans le réseau

$$\mathbf{a}^k = \mathbf{f}^k\left(\mathbf{W}^k\mathbf{a}^{k-1} - \mathbf{b}^k\right), \text{ pour } k \in [\![1\,;M]\!] \text{ et } \mathbf{a}^0 = \mathbf{p} \tag{7}$$

- On calcule les sensibilités

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M\left(\mathbf{n}^M\right)\mathbf{e} \tag{8}$$

$$\mathbf{s}^k = \dot{\mathbf{F}}^k\left(\mathbf{n}^k\right)\left(\mathbf{W}^{k+1}\right)^T\mathbf{s}^{k+1}, \text{ pour } k \in [\![1\,;M-1]\!] \tag{9}$$

- On calcule les changements de poids

$$\Delta\mathbf{W}^k = -\eta\mathbf{s}^k\left(\mathbf{a}^{k-1}\right)^T, \text{ pour } k \in [\![1\,;M]\!] \tag{10}$$

$$\Delta\mathbf{b}^k = \eta\mathbf{s}^k, \text{ pour } k \in [\![1\,;M]\!] \tag{11}$$

1.3 La fonction d'erreur

Puisque l'on réalise un apprentissage supervisé, on suppose qu'à chaque jeu de données, on connaît la sortie attendue. Il est alors nécessaire de mesurer l'erreur entre la sortie attendue et la sortie calculée par le réseau neuronal.

Il existe plusieurs formulations de cette erreur, telles que l'erreur quadratique (norme euclidienne du vecteur d'erreur), l'erreur moyenne (norme 1 du vecteur d'erreur)... Pour obtenir l'erreur d'un groupe de données (batch), on somme les erreurs de chaque données. Par soucis de simplicité, nous avons décidé d'utiliser l'erreur quadratique pour notre perceptron. Néanmoins il existe une autre fonction d'erreur : l'entropie croisée. Nous allons voir dans ce rapport pourquoi cette fonction possède de meilleures propriétés que l'erreur quadratique.

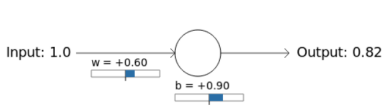
La formule de l'entropie croisée est la suivante :

$$C = -\frac{1}{n}\sum_x \left(y\ln(a) + (1-y)\ln(1-a)\right) \quad \begin{array}{l}, n \text{ la taille du batch} \\ x \text{ les exemples du batch} \\ a \text{ la sortie calculée} \\ y \text{ la sortie attendue}\end{array} \tag{12}$$

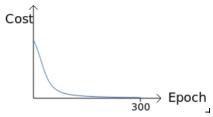
1.3.1 Comportement pour une erreur quadratique

Pour comprendre l'intérêt de cette formule, nous devons comprendre pourquoi la norme euclidienne échoue. Le concept d'entropie provenant directement de la théorie des probabilités, on doit donc choisir judicieusement la fonction d'activation de notre couche de sortie. On considère souvent que l'entropie correspond naturellement à une fonction d'activation de sortie de type sigmoïde.

Pour simplifier le raisonnement, nous utilisons un réseau neuronal trivial (un neurone à une entrée et une sortie) de fonction d'erreur quadratique. Néanmoins, les phénomènes observés sur ce neurone unique restent vrais pour des réseaux plus complexes. On dispose donc d'un neurone, avec une entrée (et un biais) et une sortie, auquel on souhaite apprendre le comportement suivant : lorsque l'entrée vaut 1, la sortie doit valoir 0. Les poids du neurone sont initialisés de manière aléatoire. D'après la figure 5a, on a après initialisation un neurone qui renvoit 0.82 lorsque l'entrée vaut 1. On entraîne ce neurone et obtient la courbe d'apprentissage 5b.



(a) Réseau utilisé et son initialisation



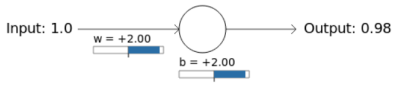
(b) Courbe d'apprentissage du neurone

Figure 5: Première initialisation du réseau d'exemple utilisant l'erreur quadratique

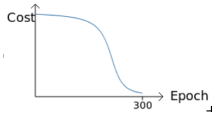
Sur la courbe 5b, on ne pose pas de valeur concernant le coût. En effet, les informations pertinentes de cette courbe ne sont pas les valeurs initiales et finales (augmenter le nombre d'itérations permettrait de réduire cela de manière arbitrairement faible). On s'intéresse plutôt à l'allure générale de la courbe. L'apprentissage sur cet exemple est très satisfaisant.

On considère maintenant un second exemple. Le même réseau est utilisé, mais avec une initialisation différente, de sorte que la sortie soit plus proche de 1. On est donc plus éloigné de l'objectif, puisque l'on cherche à retrouver

0. Les données et résultats de cet exemple sont illustrés par la figure 6b. On remarque que l'apprentissage est de qualité moindre. En effet, juste après l'initialisation, le neurone commettait une erreur plus importante, mais l'apprentissage est beaucoup plus lent. On doit donc réaliser un nombre d'itérations supérieur pour retrouver le cas 5 et pouvoir apprendre correctement.



(a) Réseau utilisé et son initialisation



(b) Courbe d'apprentissage du neurone

Figure 6: Seconde initialisation du réseau d'exemple utilisant l'erreur quadratique

1.3.2 Origine de cet échec

Puisque $C = \frac{(y-a)^2}{2}$, on peut vérifier que l'on a :

$$\frac{\partial C}{\partial \omega} = (a-y)\,\sigma'(z)\,x, \text{ où } z \text{ est l'antécédant de } a \text{ par la fonction d'activation} \tag{13}$$

$$\frac{\partial C}{\partial b} = (a-y)\,\sigma'(z) \tag{14}$$

En observant la fonction sigmoïde représentée sur la figure 7, on se rend compte que le problème provient de $\sigma'(z)$. En effet, la seconde initialisation ayant une erreur initiale très importante (a proche de 1), on se retrouve dans la partie droite de la courbe, où la pente est très faible. Cela provient du fait que $\sigma'(z) = a\,(1-a)$.

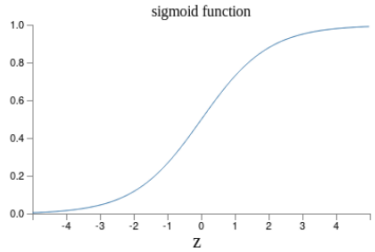


Figure 7: Graphe de la fonction sigmoïde

1.3.3 Une solution à ce problème d'apprentissage

L'apprentissage n'en est que ralenti, ce qui n'est évidemment pas souhaitable. Pour rendre le réseau moins sensible à une mauvaise initialisation, il faut donc changer ce comportement. A nouveau, cela revient à utiliser une analogie avec le comportement humain, puisque l'humain a tendance à apprendre plus vite lorsque il commet de fortes erreurs. On veut donc garder un apprentissage plus lent lorsque l'on se rapproche du minimum de la fonction d'erreur. Ainsi, on aimerait obtenir ces équations :

$$\frac{\partial C}{\partial \omega} = (a-y)\,x \tag{15}$$

$$\frac{\partial C}{\partial b} = (a-y) \tag{16}$$

La formule de la chaîne nous donne

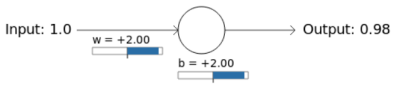
$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial b} = \frac{\partial C}{\partial a}a\,(1-a) \tag{17}$$

En utilisant l'expression voulue (équation 16), on obtient

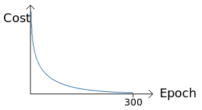
$$\frac{\partial C}{\partial a} = \frac{a-y}{a\,(1-a)} = \frac{-y}{a} + \frac{1-y}{1-a} \tag{18}$$

$$C = -y\ln(a) - (1-y)\ln(1-a) + Const \tag{19}$$

On comprend alors que la formule de l'entropie croisée n'est pas simplement une formule qui se trouve avoir des propriétés intéressantes, mais que l'on peut construire cette fonction de coût pour respecter les conditions des équations 15 et 16. L'expression 12 est alors une condition suffisante et quasiment nécessaire au respect desdites contraintes. Le "quasiment" provient de la constante d'intégration. Puisque ce n'est pas la fonction de coût en elle-même qui est intéressante mais plutôt ses variations et ses dérivées partielles, alors on peut se contenter d'une constante nulle, ce qui conserve la positivité de C. On s'attend alors à une amélioration considérable de l'apprentissage observé figure 6. La courbe d'apprentissage 8b montre effectivement un comportement beaucoup plus intéressant. La pente à l'origine est bien plus importante lorsque le réseau commet une forte erreur. On a ainsi un réseau moins sensible à l'initialisation et qui apprend d'autant plus qu'il commet une erreur importante. Ce comportement est parfois obtenu en faisant varier le taux d'apprentissage au cours du temps. Sur ces exemples, le taux d'apprentissage est constant. Ce comportement souhaité étant un artéfact de la fonction de coût, on s'attend à des performances supérieures de la part des réseaux utilisant l'entropie croisée.



(a) Réseau utilisé et initialisation identique à la figure 6



(b) Courbe d'apprentissage du neurone

Figure 8: Seconde initialisation du réseau d'exemple en utilisant l'entropie croisée

1.3.4 Une explication intuitive

La section 1.3.3 a permis de trouver par le calcul la formule 12. On cherche cette fois à obtenir une explication plus intuitive pour mieux comprendre pourquoi cette fonction de coût est pertinente.

La formule de l'entropie d'une variable aléatoire X est une somme sur les éventualités non improbables :

$$H(X) = -\sum_{i=1}^{n} p_i \log(p_i) \tag{20}$$

L'entropie est une mesure de l'incertitude d'une loi de probabilité. Par exemple, si on réalise une expérience de boules et que l'on considère la couleur de la boule tirée, alors l'entropie sera maximale lorsque les différentes éventualités sont équiprobables.

En traitement de l'information, cette grandeur permet de coder efficacement les symboles à transmettre. Si on a des symboles équiprobables, l'entropie est maximale. Dans ce cas, le nombre de bits nécessaires pour définir un symbole avec certitude est $\log_2(N)$, où N est le nombre de symboles différents. Cependant, si l'on considère un texte en français, les lettres RSTLNE sont beaucoup plus fréquentes que WXYZ. La diminution de l'entropie caractérise le fait qu'un système de codage ingénieux utilise en moyenne moins de bits pour transmettre l'information avec certitude.

L'entropie croisée $-p\log(q)$ est, en mathématiques, une mesure de la distance entre deux distributions p et q. On peut là encore y retrouver une interprétation dans le domaine du traitement de signal. En effet, si on considère un système de codage adapté à un texte dans lequel chaque lettre apparaît selon une distribution q, alors l'entropie croisée $-p\log(q)$ quantifie l'adaptation de ce même système pour chiffrer un texte dans lequel chaque lettre apparaît selon la distribution p.

Ainsi, si on dispose d'un jeu de données dont les sorties attendues suivent une distribution empirique p, et d'un réseau neuronal dont les sorties à ces données suivent une distribution q, alors adapter le réseau pour qu'il puisse reproduire fidèlement les observations p revient à diminuer l'entropie croisée de ces deux distributions.

1.3.5 Estimateur de l'entropie croisée

On considère un batch de taille n contenant N données distinctes $\{x_1, x_2, \dots, x_N\}$, chaque observation x_i (pour $i \in [\![1\,;N]\!]$) apparaît dans le batch np_i fois.

Notre réseau est alors ainsi constitué : la couche de sortie est composée d'un seul neurone (une somme de l'entropie sur les neurones peut être faite si on a une couche de plusieurs neurones) et d'une fonction d'activation sigmoïde donnant une sortie q_i lorsque soumise à l'exemple x_i. On a alors :

$$\mathbb{P}(X = y_i) = \begin{cases} q_i & \text{si } y_i = 1 \\ 1 - q_i & \text{si } y_i = 0 \end{cases}, \text{ où } X \text{ représente notre réseau et } y_i \text{ représente l'observation à la donnée } x_i \tag{21}$$

$\mathbb{P}$, tel que défini par l'équation 21, représente la probabilité que notre réseau trouve la bonne solution. Sa densité de probabilité est donnée sous une forme différente par l'équation 22

$$\mathbb{P}(X = y_i) = {q_i}^{y_i} (1 - q_i)^{1-y_i} \tag{22}$$

Les équations 23 et 24 ci-dessous présentent respectivement la vraisemblance et la log-vraisemblance de notre loi.

$$\mathbb{L} = \prod_{i=1}^{N} {q_i}^{np_i} (1 - q_i)^{n(1-p_i)} \tag{23}$$

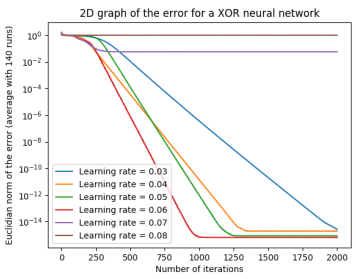
$$\frac{1}{n} \log(\mathbb{L}) = \sum_{i=1}^{N} (p_i \ln(q_i) + (1 - p_i) \ln(1 - q_i)) \tag{24}$$

D'après les équations 12 et 24, la log-vraisemblance de notre système est ainsi proportionnelle à l'opposé de l'entropie croisée. Ainsi, on comprend que minimiser l'entropie croisée revient à utiliser un estimateur de maximum de vraisemblance. Notre système est donc d'autant plus performant que son entropie croisée est faible.

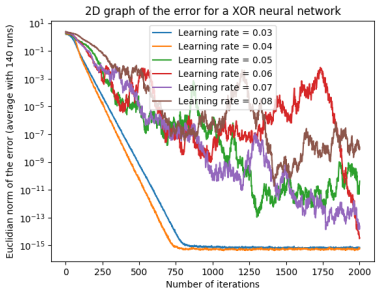
1.4 La fonction XOR

Pour tester notre perceptron, nous utilisons la fonction XOR prenant deux valeurs binaires et retournant 1 si et seulement si les deux valeurs binaires sont différentes. XOR est une fonction relativement simple qui a la particularité d'être non linéaire. Ainsi, on ne peut se contenter d'un réseau qui ne contient aucune couche cachée. Cela nous permet donc d'en tirer nos premières conclusions.

Pour cela, nous configurons notre réseau de la manière suivante. La couche d'entrée ainsi que la couche cachée contiennent 2 neurones tandis que la couche externe est constituée d'un neurone. La couche cachée et la couche de sortie sont chacun suivie d'une fonction d'activation. Nous avons utilisé une tangente hyperbolique pondérée, reprenant des valeurs utilisées par Yann LeCun et al : $x \mapsto 1.7159 \tanh \frac{2x}{3}$. Nous avons évalué l'apprentissage en utilisant différents taux d'apprentissage.



(a) XOR avec 2 neurones cachés



(b) XOR avec 30 neurones cachés

Figure 9: Étude du XOR avec différents taux d'apprentissage

La figure 9a représente l'évolution de la norme euclidienne de l'erreur du réseau en fonction du nombre d'itérations. On remarque qu'un taux d'apprentissage trop faible ralentit fortement la vitesse d'apprentissage. Au contraire, un taux d'apprentissage trop important peut faire échouer l'apprentissage. Cela s'explique par le fait que le réseau n'arrive pas à converger vers le minimum global car ses pas sont trop importants. Il oscille alors autour du minimum sans s'en rapprocher suffisamment. Si le taux d'apprentissage est trop important, on peut même s'écarter de la solution ! On comprend alors que le choix du taux d'apprentissage est un facteur déterminant de la convergence de notre réseau.

La figure 9b a, quant à elle, été tracée en utilisant 30 neurones dans la couche cachée. Cela nous permet d'étudier les capacités d'un réseau "trop intelligent" par rapport à la tâche qu'il doit apprendre. On remarque que le réseau a beaucoup plus de difficulté à trouver le minimum. Les courbes sont globalement très bruitées. Néanmoins, cette fois-ci, tous les réseaux ont convergé même à haut taux d'apprentissage. Mais ce sont les taux les plus bas qui ont été le moins affecté par cette modification. Augmenter la taille du réseau permettrait alors de contrer les problèmes de divergence. Le bruit introduit peut être atténué avec un taux d'apprentissage relativement faible (la convergence est alors même accélérée).

1.5 MNIST

MNIST est une base de données d'images de chiffres manuscrits. L'objectif est d'apprendre à notre perception à reconnaître le chiffre en analysant les 784 pixels de l'image. Nous avons utilisé la configuration suivante : la couche d'entrée (784 neurones) est reliée à une première couche intermédiaire de 16 neurones. Celle-ci mène à une autre couche de même configuration. Enfin la sortie se fait par une couche de 10 neurones. Nous avons conservé la fonction d'activation tanh pondérée utilisée sur le XOR puisque celle-ci avait donné des résultats très satisfaisants (meilleure convergence que la sigmoïde). Pour tester notre perceptron, nous avons tracé le taux d'erreur sur l'échantillon test en fonction du nombre d'itérations. Afin de "moyenner" la courbe et d'annuler les fluctuations statistiques de l'apprentissage (provenant de l'initialisation aléatoire du réseau et de l'ordre de présentation aléatoire des exemples), nous avons réalisé cet apprentissage 280 fois. Le taux d'apprentissage utilisé est 0.005.

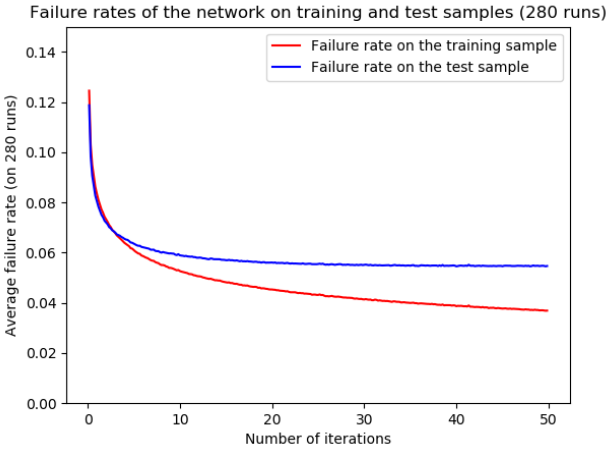


Figure 10: Taux d'erreur sur MNIST

D'après la figure 10, le taux d'erreur sur l'échantillon d'apprentissage est toujours inférieur à celui sur l'échantillon de test. Cela est normal car l'apprentissage du réseau consiste à coller au mieux aux données. Afin d'éviter le sur-apprentissage (on apprend par coeur les exemples mais n'est plus capable de généraliser les propriétés), on cherche à minimiser l'erreur sur l'échantillon de test. Sur MNIST, cela a relativement peu d'incidence car la base de données contient beaucoup d'exemples variés ce qui nous protège de la sur-interprétation. On converge assez rapidement (une dizaine d'itérations) vers 94% de réussite, ce qui est un résultat très satisfaisant de notre perceptron. Un réseau avec 32 neurones sur la première couche cachée change assez peu le résultat, si ce n'est que l'on converge vers une valeur légèrement meilleure : 96% de réussite sur l'échantillon de test.

Afin d'évaluer l'influence du taux d'apprentissage, nous réalisons l'entraînement sur 19 taux d'apprentissage différents (régulièrement placés de 0.001 à 0.01). Chaque apprentissage étant constitué de 50 itérations et étant moyenné sur 140 réalisations. Nous obtenons alors la figure 11 avec un intervalle de confiance à 95%.

On remarque qu'un fort taux d'apprentissage augmente également l'écart-type des données. L'apprentissage est alors plus sensible à l'initialisation. On préfère naturellement éviter un faible taux pour garder une vitesse de convergence convenable. En comparant toutes ces données, nous avons évalué que 0.003 était un des taux les plus intéressants de ces points de vue, la convergence étant rapide et avec un faible écart-type. Néanmoins, l'impact sur les performances est assez négligeable (moins de 1% de différences).

1.6 Un laïus sur les différents optimiseurs Paul ?

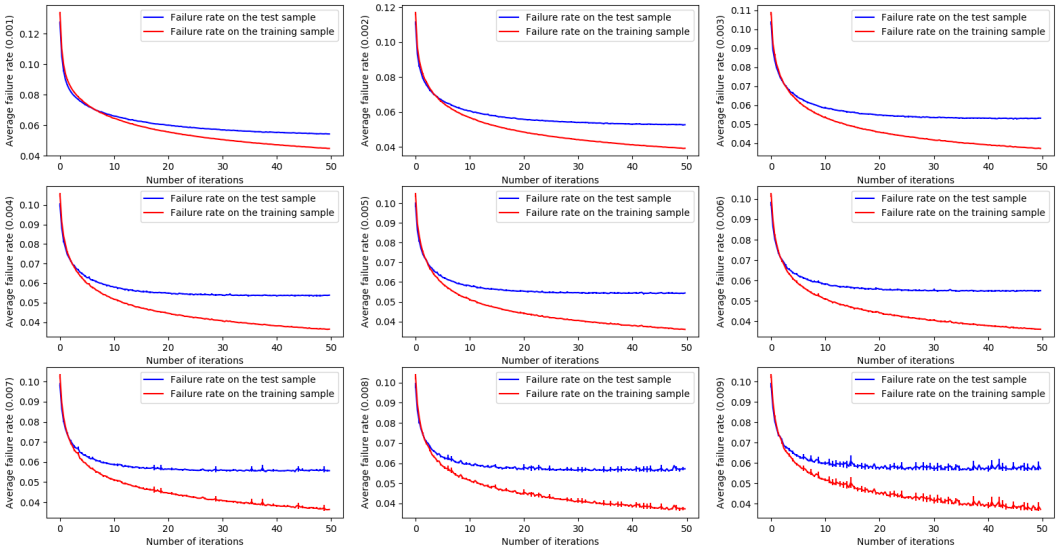


Figure 11: Apprentissage MNIST avec différents taux d'apprentissage

2 Le Q-Learning

2.1 Qu'est-ce que le Q-Learning ?

L'objectif du Q-Learning est de permettre à un agent (notre IA) d'évoluer efficacement dans un environnement (par exemple un jeu). Pour cela, on utilise un système de récompense. Dans un état donné, l'IA choisit l'action qui lui rapporte le plus de récompense. Néanmoins, on ne veut pas que notre IA choisisse une action parce qu'elle lui offre une récompense immédiatement. L'IA doit prendre en compte les récompenses qu'elle obtiendrait plus tard en effectuant certaines actions. Cette caractéristique du Q-Learning permet l'établissement de stratégies.

Pour pouvoir suivre ces stratégies, notre IA se donne des récompenses Q (imaginaires) qui reflètent à la fois la satisfaction obtenue immédiatement mais également la satisfaction que l'on peut espérer obtenir à l'avenir. Il faut alors que notre système de récompense satisfasse l'équation de Bellman :

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{25}$$

Dans l'équation 25, s représente l'état actuel, a l'action choisie, r la récompense obtenue à cet instant et s' l'état dans lequel on arrive après avoir effectué l'action a. Ainsi, la satisfaction obtenue en effectuant l'action a depuis l'état s correspond à la fois à la récompense obtenue immédiatement ainsi que la meilleure satisfaction que l'on puisse espérer à l'avenir (on recherche un maximum car l'action a' sera choisie par notre agent). γ est le facteur d'actualisation. Il mesure la préférence à obtenir une récompense à l'instant présent plutôt qu'à l'avenir. Pour ne pas que la satisfaction diverge, il faut que γ soit plus petit que 1. Plus il est proche de 1, plus le réseau prend en compte des instants éloignés et établis des stratégies sur le long terme. La valeur du facteur d'actualisation dépend alors de l'environnement dans lequel est plongé notre agent.

Le principe du Q-Learning repose sur l'apprentissage autonome par l'IA de ces différentes valeurs de Q (que nous autres humains ne connaissons pas). Pour cela, on distingue deux phases. La phase d'exploration correspond à agir de manière aléatoire sans prendre en compte la satisfaction. Cela permet de découvrir des voies jusqu'à lors inconnues et potentiellement trouver de meilleures stratégies que celles connues à cet instant. Puis, lors de l'exploitation, on utilise les valeurs de satisfaction trouvées pour choisir notre action. Dans les deux cas, on modifie notre réseau en conséquence (l'exploitation permet de préciser les valeurs de la fonction de satisfaction Q). Le taux d'exploration est au début de la partie de 1 : on a aucun a priori sur les actions à choisir donc on essaie aléatoirement pour collecter un maximum d'informations sur le jeu. Puis à mesure que l'apprentissage progresse, le taux d'exploration diminue. Il n'atteint généralement pas 0, on veut toujours explorer un peu de temps en temps dans l'espoir de trouver de meilleures stratégies que celles connues.

Il existe également une autre politique pour l'exploitation. Plutôt que de choisir systématiquement l'action procurant le plus de satisfaction, on peut établir des probabilités en fonction des satisfactions calculées et choisir notre action selon ces probabilités. Cette politique ne prend pas seulement en compte l'action qui a le plus de satisfaction mais également les satisfactions de toutes les actions. Néanmoins, dans nos tests sur différents jeux, nous avons conservé la politique (dite greedy) qui consiste à choisir l'action possédant la plus grande satisfaction.

2.2 Calculer la fonction Q

Lorsque l'on joue (exploration ou exploitation), on enregistre les transitions vécues sur lesquelles on réalise l'entraînement. L'apprentissage se passe de la manière suivante. D'après 25, $r + \gamma \max_{a'} Q(s',a')$ est ce que l'on cherche à obtenir et $Q(s,a)$ correspond à ce que l'on a effectivement actuellement. On peut alors retrouver une situation d'apprentissage supervisé (un peu particulière car les données d'apprentissage sont calculées par le réseau et évoluent à mesure que le réseau évolue).

Ainsi, on peut utiliser la modification suivante en introduisant le taux d'apprentissage λ :

$$\Delta Q(s,a) = \lambda \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{26}$$

Pour chaque transition (constituée d'un état initial, d'une action, d'une récompense et d'un état final), on calcule la valeur voulue pour la fonction Q (équation 25, ou simplement r si la partie se termine) puis on applique le changement décrit équation 26.

Pour un espace des états réduits et un nombre d'actions limité, les valeurs de la fonction Q peuvent être stockées dans un tableau et modifiées de la manière décrite précédemment.

Lors d'un apprentissage, la transition est choisie aléatoirement. En effet, si on apprend sur les transitions dans l'ordre d'apparition, on augmente le poids de la dernière transition obtenue. Or, puisqu'un faible changement de Q-table peut engendrer un important changement de stratégies, on veut éviter ce genre de dépendance, c'est pourquoi la transition sur laquelle on apprend est choisie aléatoirement dans l'ensemble des transitions obtenues.

2.3 Jeu des bâtonnets

Pour tester le Q-Learning, on utilise le jeu des bâtonnets. Il y a deux joueurs et douze bâtonnets sur la table. Alternativement, chaque joueur peut prendre 1, 2 ou 3 bâtonnets. Le joueur contraint de prendre le dernier bâtonnet a perdu. Ce jeu a l'avantage d'avoir un petit espace des états. En effet, l'état est entièrement décrit par le nombre de bâtonnets restants qui est un entier compris entre 1 et 12. Le nombre d'action est également très faible (3). Ainsi ce jeu se prête parfaitement à l'utilisation d'une Q-table pour représenter la fonction Q.

On remarque qu'il existe une stratégie imbattable pour gagner une partie. Si vous commencez la partie, il suffit de prendre 3 bâtonnets puis 4-x bâtonnets à chaque coup où x est le nombre de bâtonnets pris par l'adversaire. Notre objectif est donc que notre IA découvre cette stratégie et s'y tienne.

Pour tester notre implémentation, nous faisons jouer à notre IA 100 000 parties (pour annuler les fluctuations statistiques) contre un adversaire jouant aléatoirement (cela permet de comparer les tests lorsque les apprentissages sont faits dans des contextes différents). Le taux d'apprentissage est fixé à 0.1 et le taux d'actualisation à 0.9, ce qui est faible mais adapté vu que les parties sont très courtes (moins de dix coups). À chaque coup, la probabilité d'être en exploration évolue exponentiellement en fonction du nombre de parties jouées. À la n-ième partie, la probabilité d'exploration est 0.999^n. Puisqu'il existe une stratégie gagnante, on ne peut s'attendre à gagner 100% des parties car il existe une probabilité $(\frac{1}{54})$ que le joueur aléatoire l'applique. Ainsi, on ne peut espérer converger qu'à 98.15% de victoires. Lors des tests, on est bien sûr constamment en phase d'exploitation, sans entraînement sur les transitions.

Nous avons testé différents contextes d'apprentissage en utilisant différents adversaires. Tout d'abord nous avons appris sur une IA qui apprend en même temps. Puis nous avons essayé de reproduire les résultats en apprenant sur un joueur jouant aléatoirement. Enfin, notre IA devra apprendre contre un joueur connaissant et appliquant la stratégie gagnante. Dans tous les cas, les tests sont fait contre un adversaire jouant aléatoirement pour pouvoir comparer les résultats.

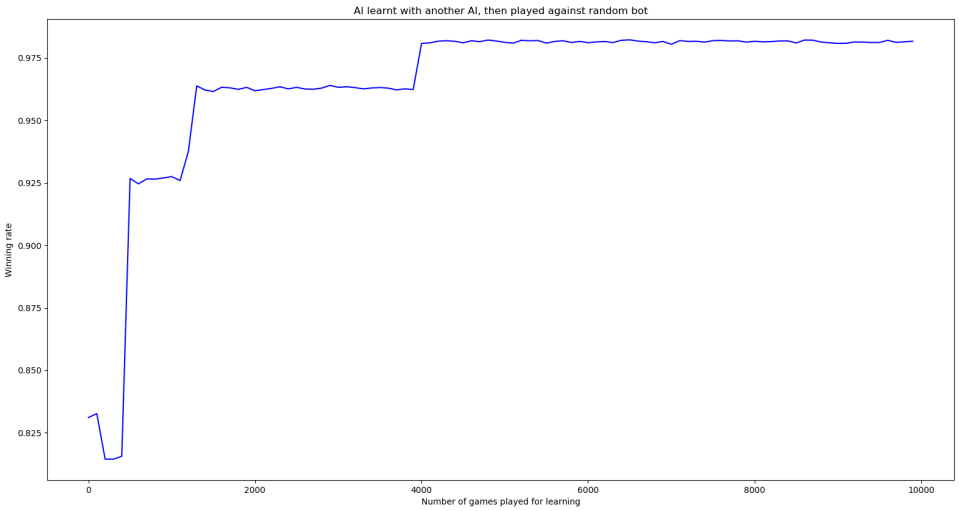


Figure 12: Apprentissage contre une autre IA apprenant en même temps

La figure 12 nous présente les résultats lorsque notre IA apprend contre une autre IA apprenant également à jouer à ce jeu. On remarque plusieurs choses. La convergence à 98.15% ainsi que la Q-table nous confirme que notre IA a réussi à apprendre quelle était la stratégie parfaite pour jouer à ce jeu. L'apprentissage est donc réussi. On remarque également que la courbe semble être en escalier. Cela provient du fait que pour choisir une action lors de l'exploitation, on choisit celle avec la meilleure satisfaction mais leurs valeurs respectives n'ont pas plus d'incidence. Ainsi, deux Q-table différentes mais ayant la même action préféré à chaque état auront des taux de victoires similaires. La modification de la Q-table n'entraîne pas nécessairement un changement de stratégie, c'est pourquoi on observe ces plateaux. Pour la même raison, on observe des sauts. Même si deux actions procurent presque la même satisfaction, on choisit toujours celle qui en procure le plus. Ainsi, un faible changement de la Q-table peut entraîner un changement radical dans la stratégie adoptée par notre IA. Ce changement se ressent alors fortement dans le taux de victoires lorsque l'IA est confrontée au joueur aléatoire. Vu la simplicité du jeu, un nombre raisonnable de parties (quelques milliers) est nécessaire pour comprendre la stratégie, ce qui s'effectue

très rapidement (en quelques minutes).

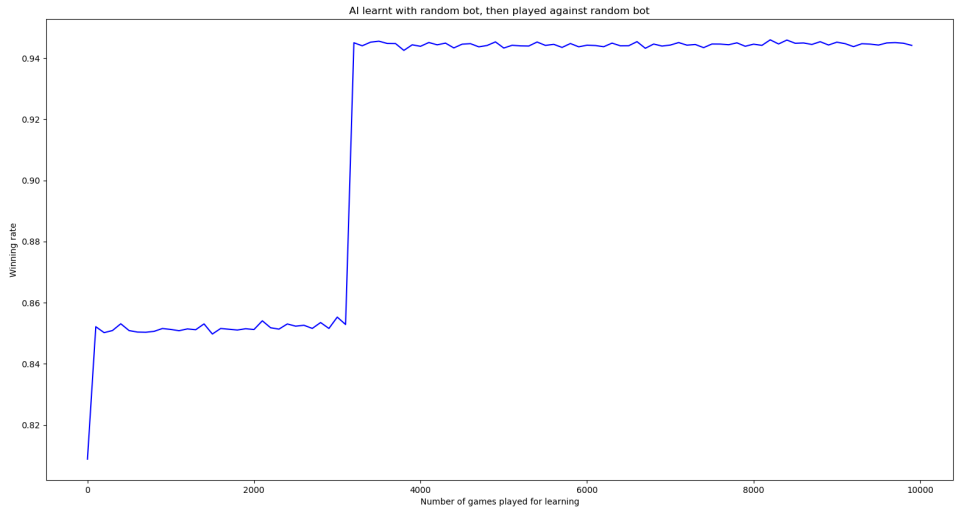


Figure 13: Apprentissage contre un joueur jouant aléatoirement

La figure 13 nous présente les mêmes résultats lorsque l'IA apprend sur un joueur sélectionnant son action aléatoirement. On remarque que la convergence est plus lente et imparfaite (à 94% de victoires environ). Cela provient du fait que le joueur adverse joue aléatoirement. Ainsi, même si notre IA réalise un mauvais coup (contraire à la stratégie), il existe une probabilité que le joueur adverse n'en profite pas pour appliquer la stratégie mais redonne l'avantage à notre IA. Cela signifie alors que notre IA surestime la satisfaction de certains coups. Elle n'a donc pas compris la stratégie car le joueur adverse, trop complaisant, ne la sanctionnait pas lorsqu'elle choisissait une mauvaise action. On en conclut qu'il faut proscrire l'apprentissage contre un joueur jouant aléatoirement.

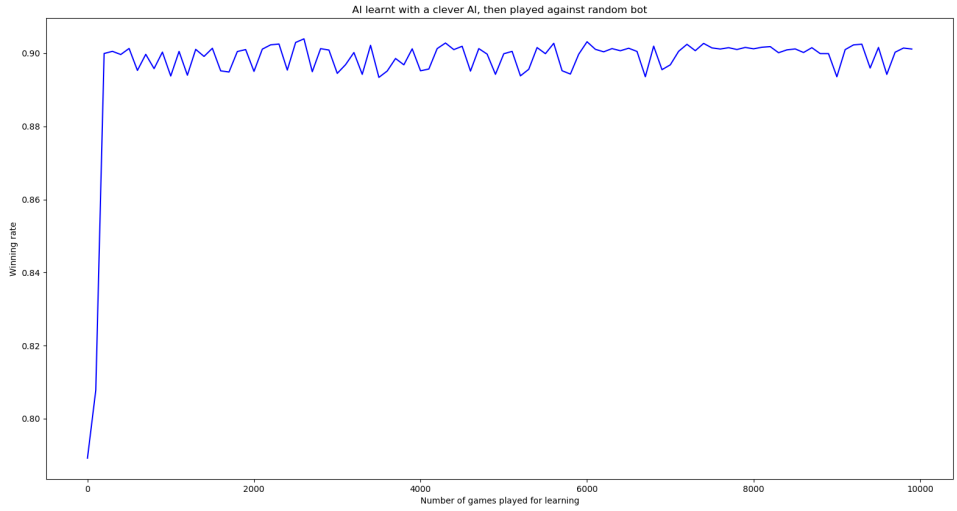


Figure 14: Apprentissage contre un joueur appliquant la stratégie gagnante

Enfin, la figure 14 reprend le même jeu mais notre IA apprend contre un joueur ayant compris la stratégie gagnante et l'appliquant systématiquement. On observe deux résultats importants : la convergence est plus rapide mais imparfaite (environ 90% de victoires). La rapidité de la convergence provient du fait que le joueur adverse sanctionne instantanément tout coup contraire à la stratégie (car il joue parfaitement). Ainsi, notre IA

comprend rapidement que certains coups ne doivent pas être joués. L'imperfection de la convergence provient également du fait que l'adversaire applique la stratégie. Il y a donc certains états dans lequel il ne veut surtout pas se retrouver car cela serait à l'avantage de notre IA. Cette tendance à éviter certains états empêche notre IA d'explorer l'ensemble de l'espace des états. Elle ne peut donc pas trouver quelle est la meilleure stratégie car une partie de l'espace des états demeure inconnue.

La rapidité de convergence est néanmoins intéressante. On peut alors imaginer commencer l'apprentissage contre un joueur très compétent pour vite cerner les quelques principes fondamentaux du jeu, puis finir l'apprentissage contre un joueur apprenant en même temps pour pouvoir explorer la totalité de l'espace des états et continuer d'apprendre relativement efficacement. Néanmoins, la simplicité du jeu en question joue probablement en faveur de cette convergence. On peut se demander si cette caractéristique demeurerait sur un jeu plus complexe où l'espace des états est beaucoup plus importants.

2.4 Le labyrinthe

On se place sur un plan quadrillé. Chaque case contient une récompense (positive, négative ou nulle). L'objectif est d'atteindre l'arrivée avec le plus de satisfaction. Atteindre l'arrivée est une condition de victoire (le jeu s'arrête). Certaines cases, en plus de procurer une récompense négative, peuvent également provoquer une défaite instantanée.

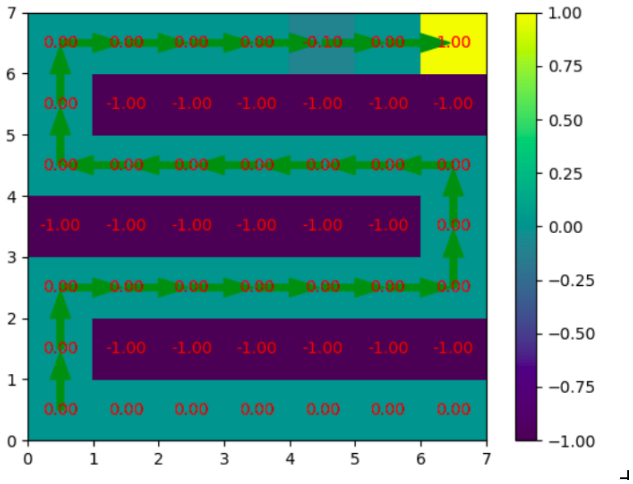


Figure 15: Chemin choisi par l'IA lorsque confrontée à un labyrinthe donné

La figure 15 représente le chemin choisi par l'IA lorsque celle-ci est confronté à un labyrinthe particulier, en serpentin. On remarque l'IA choisit un chemin qui l'amène à l'arrivée, tout en maximisant les récompenses obtenues sur le chemin. L'apprentissage sur d'autres labyrinthes générés aléatoirement font également apparaître un apprentissage réussi et donc une capacité à atteindre la sortie du labyrinthe.

3 Deep Q-Learning

3.1 Principe du Deep Q-Learning

Le Q-Learning requiert de stocker les valeurs de la Q-function de quelque manière que ce soit. Notamment, on a utilisé un tableau (la Q-table) dans le jeu des bâtonnets. Néanmoins, on peut être confronté à des espaces des états de très grande taille : par exemple si les états sont des images (même de taille raisonnable), alors on peut imaginer des milliards d'états possibles. Bien que généralement plus restreint, le nombre d'actions est également à l'origine de ce problème puisque la taille de la matrice est le produit entre le nombre d'états différents et le nombre d'actions possibles. Puisque notre IA jouant à Pong doit recevoir des images du jeu, il est impensable de stocker chaque valeur en mémoire. Le Deep Q-Learning résout ce problème en remplaçant la Q-table par un réseau neuronal. Son objectif est alors d'interpoler la fonction de satisfaction. Cette approximation nous permet d'économiser une importante quantité de mémoire.

On a alors un réseau neuronal qui reçoit en entrée un état et une action pour retourner la satisfaction qui y est associée. Pour pouvoir choisir une action selon notre algorithme favorisant la plus grande satisfaction, il est nécessaire de connaître la valeur du réseau lorsqu'excité par toutes les actions. Ainsi le nombre d'évaluations du réseau est égal au nombre d'actions possibles. Cela est d'autant plus inefficace qu'il sera nécessaire d'exécuter cette opération très régulièrement. Pour régler ce problème, notre réseau ne reçoit plus que l'état en question en entrée. En sortie, le réseau nous donne les satisfactions associées à chaque état, de sorte qu'une seule évaluation du réseau suffise à choisir l'action à effectuer.

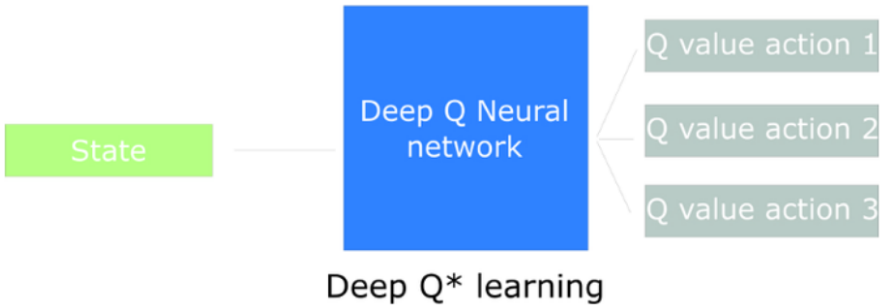


Figure 16: Schéma du Deep Q-Network

Lorsque nous réalisons l'apprentissage, nous sélectionnons une transition qui contient l'action choisie à l'instant de la transition. Puisque notre réseau renvoie toutes les satisfactions, il est important de veiller à ce que le réseau ne réalise la rétropropagation que sur l'action de la transition. Cela peut être effectué en remplaçant la valeur voulue pour les autres actions par la valeur calculée, de sorte que l'erreur soit nulle, sauf pour l'action de la transition. Pour cette dernière, on conserve l'équation 25 pour l'expression voulue de la fonction de satisfaction.

Pour faciliter la réalisation de notre algorithme et nous débarasser des grosses difficultés posées par le réseau neuronal telles que la rétropropagation, il a été convenu d'utiliser TensorFlow. Cette bibliothèque Python simplifie considérablement la conception d'un tel algorithme. Néanmoins, pour ne pas trop occulter ce qu'il se passe dans le réseau, nous voulons rester suffisamment bas niveau. Ainsi, nous nous interdisons toute surcouche de TensorFlow telle que Keras.

3.2 Jeu des bâtonnets

Pour valider notre implémentation du Deep Q-Learning, on le teste sur un jeu simple que l'on sait résoudre : le jeu des bâtonnets. Les règles du jeu sont les mêmes que précedemment. Pour la rétropropagation, nous utilisons l'optimiseur Adam particulièrement efficace, avec le taux d'apprentissage 0.001. Le facteur d'actualisation est conservé, de même que les taux d'exploration. On stocke un maximum de 10000 transitions. Après chaque partie jouée, on apprend sur *toutes* les transitions enregistrées jusqu'à lors (par minibatch de 32 transitions). Le réseau utilisé possède 4 neurones d'entrée (chaque état est codé binairement), 10 neurones de couche cachée puis 3 neurones de sortie pour les 3 différentes actions fournies par le jeu. Ce réseau sert à la fois à apprendre sur les transitions et à générer la cible, conformément à l'équation 25. Bien que cela ne soit pas l'implémentation la plus optimale, elle suffit pour un jeu aussi simple que celui des bâtonnets.

Notre réseau apprend sur 2000 parties (contre une IA qui est également en train d'apprendre). Pour nous persuader de l'efficacité de notre algorithme, nous reconstituons la Q-table à l'issue de l'apprentissage en interrogeant le réseau sur les différents états accessibles lors d'une partie. Le résultat est le suivant :

Bâtonnets restants	Prendre 1 bâtonnet	Prendre 2 bâtonnets	Prendre 3 bâtonnets
1	-0.9966581	-0.98666275	-0.99692893
2	1.0102367	-0.9285556	-0.9973395
3	0.1279521	1.0385635	-1.001752
4	-0.17204374	-0.1530217	1.0058613
5	-0.509134	-0.48264277	-0.49259257
6	0.9031452	-0.11602497	-0.24787879
7	0.10454643	0.92308414	-0.1744315
8	0.12821698	0.2924267	0.9102776
9	-0.06348622	-0.0588069	-0.05864894
10	0.82183206	0.18388963	0.5014814
11	0.3764193	0.82838273	0.14708841
12	0.26232004	0.44059992	0.8352009

Figure 17: Fonction de satisfaction Q approximée par le réseau neuronal

Le tableau 17 nous prouve que le réseau neuronal a compris la stratégie et l'applique. En effet, pour chaque état, la satisfaction maximale est donnée au coup prédit par la stratégie. Certaines lignes n'ont pas de maximum évident, car cela correspond à des états où la stratégie nous donne perdant, ainsi les différents coups se valent. On remarque que la satisfaction dépasse parfois en valeur absolue la récompense effectivement perçue par l'IA, cela peut paraître absurde puisque cette récompense est l'objectif final, mais peut s'expliquer par le fait que le réseau ne fait qu'une approximation, et ne calcule donc pas la Q-function exactement. Néanmoins l'erreur reste assez faible. Enfin, on remarque que la satisfaction accordée aux bons coups est meilleure lorsque l'on est proche de la fin de la partie. Cela provient du fait que la propagation de la récompense favorise les états proches de celui où la satisfaction est effectivement attribuée. Quoiqu'il en soit, notre IA respecte parfaitement la stratégie, qu'elle a pu apprendre en moins de 10 minutes malgré une implémentation imparfaite.

3.3 Changements effectués pour le passage à Pong

Maintenant que nous nous sommes assurés de la fiabilité de notre implémentation du Deep Q-Learning sur le jeu des bâtonnets, nous voulons l'appliquer sur Pong. Pong est bien plus compliqué que le jeu des bâtonnets, ainsi plusieurs modifications seront nécessaires.

3.3.1 Deux Deep Q-Networks

Lorsque nous réalisons la rétropropagation, nous devons connaître l'objectif de notre réseau. Celui-ci est donné par l'équation 25. Ainsi, il est nécessaire de faire une propagation vers l'avant pour évaluer l'objectif de notre réseau. Notre réseau se "pourchasse" donc lui-même. Cela peut mener à de gros problèmes de convergence, voire à une instabilité du réseau. Cela n'avait pas été observé précédemment à cause de la simplicité du jeu en question.

L'instabilité du réseau se caractérise par deux choses d'après nos observations. Tout d'abord, le Deep Q-Network ignore totalement l'entrée qui lui est donnée, il retourne toujours les mêmes satisfactions (précision de 10^{-6} pourtant !). De plus, les satisfactions accordées à chaque action diffèrent très peu (une différence de quelques unités de 10^{-6}). Ainsi, notre réseau ne sait pas quelle action choisir, l'une est choisie par défaut. Et pourtant, notre réseau décide de s'y tenir quoiqu'il arrive. Cela se conclut par, ou bien, une immobilité de notre IA, ou bien, une insistance sur l'un des mouvements (c'est-à-dire notre IA se coince sur l'un des rebords du plateau de jeu et y reste pendant toute la partie).

Pour éviter ce problème, nous considérons deux réseaux différents. Le réseau principal est celui qui apprend, sur lequel on réalise la rétropropagation et celui qui choisit l'action à effectuer. Le second réseau a pour rôle de donner l'objectif à atteindre par le premier réseau. Son seul but est de calculer la valeur de l'expression 25. Il est basé sur le premier réseau (copie conforme). Mais pour éviter les problèmes de stabilité du fait que le réseau principal joue au chat et à la souris avec lui même, le second réseau n'est synchronisé sur le premier que de temps en temps. Le nombre de coups séparant deux synchronisations est un hyperparamètre à déterminer. Nous avons repris celui utilisé par DeepMind, avec une synchronisation tous les 10000 coups.

Pong étant un jeu de durée bien plus long que le jeu des bâtonnets, il fallait, pour permettre l'établissement de stratégies, renforcer l'appréciation d'une récompense future. Ainsi le facteur d'actualisation utilisé était 0.99. Une autre constante a changé : le taux d'exploration. Auparavant de décroissance exponentielle, le taux d'exploration varie dorénavant linéairement de 1 à 0.1 sur l'ensemble des parties.

3.3.2 Optimiseur RMSProp

À chaque apprentissage, nous devons calculer la satisfaction visée grâce au second réseau neuronal, synchronisé régulièrement sur le premier. Cela implique que les données d'apprentissage changent constamment. Précédemment, nous utilisions l'optimiseur Adam. Nous avons découvert dans différents articles qu'Adam tolérait mal les changements de données d'apprentissage. Il était alors recommandé d'utiliser l'optimiseur RMSProp. Ainsi, nous avons pris cet optimiseur avec un taux d'apprentissage de 0.00025.

3.3.3 Changement de configuration de réseau

Le jeu étant beaucoup plus complexe, il est nécessaire de complexifier également notre réseau. Ainsi, bien que nous gardons une couche dense de 3 neurones en sortie, nous introduisons une couche dense de plusieurs centaines de neurones (256 ou 512 selon les tests) précédant la sortie.

En outre le réseau reçoit désormais une photographie de la table de jeu pour représenter l'état. Nous traitons donc cette donnée via un réseau à convolution précédant les couches denses. Celui-ci est formé de trois couches à convolution : une couche de 32 filtres de taille 8 par pas de 4, une couche de 64 filtres de taille 4 par pas de 2 puis une couche de 32 filtres de taille 3 par pas de 1. Nous avons testé une configuration alternative où chaque couche convolutionnelle est suivie d'une couche de max pooling (taille 2 par pas de 2).

Ce réseau ne reçoit pas les images directement. En effet, la couleur est retirée par un grayscale. Et chaque image (initialement de taille 210×160) est rognée, sous-échantillonnée et retaillée de sorte à obtenir une image de taille 80×80. Pour aider notre réseau, un état n'est pas formé de la dernière image traitée, mais des quatre dernières images reçues, traitées, puis superposées. Cela permet au réseau d'identifier le mouvement de la balle grâce à des images prises à des instants différents.

3.3.4 Apprentissage

Nous retenons un échantillon d'un million de transitions sur lesquelles nous effectuons notre apprentissage. Lorsque nous dépassons ce quotat, les transitions les plus anciennes sont oubliées. À chaque coup réalisé, nous apprenons sur un minibatch de 32 transitions choisies aléatoirement parmi toutes les transitions disponibles. L'apprentissage n'est réalisé qu'à partir de la 50000ème transition, nous jugeons ne pas avoir assez de données pour apprendre avant ce seuil.

3.3.5 Problèmes de mémoire

N'ayant pas l'habitude de réaliser des algorithmes aussi gourmands en temps de calcul et en RAM, nous nous sommes laissés piéger par la consommation de RAM de notre IA. En effet, le stockage d'un million de transitions, chacune formée de deux images (une de départ et une d'arrivée) de taille $80 \times 80 \times 4$, chaque pixel étant sous forme d'un flottant 32 bits, demande des ressources excessives : presque 200 GB de RAM. Pour lutter contre cette utilisation déraisonnable des ressources de la machine, nous avons stocké chaque image sous la forme d'un entier non signé sur un octet, ce qui permet exactement de sauvegarder la valeur d'un pixel (entre 0 et 255). Ainsi, notre programme consomme moins de 50 GB de RAM par processus lancé.

Une autre méthode consisterait à corriger le fait que la plupart des images sont stockées deux fois. En effet, l'état final d'une transition correspond à l'état initial de la transition suivante. Il y aurait moyen de diviser les ressources consommées par deux en corrigeant ce problème. Toute façon, cela risquait d'en entraîner d'autres. Il faut en effet s'assurer du bon stockage de la transition sans trop complexifier l'accès aux données pour ne pas sacrifier la vitesse de calcul. Finalement, les ressources du serveur le permettant, cette solution n'a pas été mise en place bien que nous aurions pu y avoir recours si nécessaire.

Enfin, il aurait suffit de diminuer le nombre de transitions enregistrées (passer d'un million à cent mille par exemple) pour limiter les problèmes de mémoire. L'impact de ce changement sur les performances de l'algorithme nous était inconnu. Ainsi, une telle modification est concevable pour aller plus loin, mais nous conservons des hyperparamètres similaires à ceux des scientifiques en attendant de faire fonctionner correctement notre réseau (soit un million de transitions).

3.4 Pong

Après avoir réalisé les multiples modifications citées ci-dessus, nous avons pu tester notre IA sur Pong. Pour simuler l'environnement du jeu, nous utilisons la bibliothèque Python Atari[gym].

3.4.1 Apprentissage sur 3000 parties

Tout d'abord nous avons entraîné notre IA sur 3000 parties. Bien que celle-ci n'arrive pas à vaincre son adversaire, les résultats au bout de 2000 parties sont encourageants. En effet, vers environ 2000 parties apprises, notre IA marque entre 7 et 12 points à chaque partie. Elle a même réussi à marquer 16 points lors d'une partie. Ces résultats encourageants tendent à valider notre algorithme bien que très imparfait. On remarque que ces résultats sont tirés de l'apprentissage, ansi notre IA obtient ces scores malgré le fait qu'elle soit handicapée par un fort taux d'exploration (40%).

3.4.2 Apprentissage sur 7000 parties

Nous entraînons maintenant notre IA sur 7000 parties. Pour la tester, nous n'utilisons plus les logs de l'apprentissage car ceux-ci sont très imparfaits (IA handicapée par un taux d'exploration qui évolue constamment, et le réseau change à chaque coup). Nous utilisons alors une session de test séparée. Pour cela, nous reprenons le réseau obtenu à l'issu de l'apprentissage et le testons sur 5000 parties sans exploration et sans entraînement. Cela nous permet d'évaluer plus justement les capacités de notre Deep Q-Network. Pour obtenir une mesure des résultats plus fine que le simple taux de victoire, nous nous intéressons à un score "relatif". Celui-ci est défini par le nombre de buts marqués auquel on retire le nombre de buts concédés. Ainsi, une défaite totale donne un score relatif de -21 et une victoire totale de $+21$. Entre les deux, chaque entier relatif peut être atteint sauf 0 (il faut obligatoirement un vainqueur). Les résultats sont présentés sous la forme d'un histogramme.

Nous étudions deux configurations de réseaux : sans maxpooling avec une couche dense de 512 neurones puis avec maxpooling et une couche dense de 256 neurones.

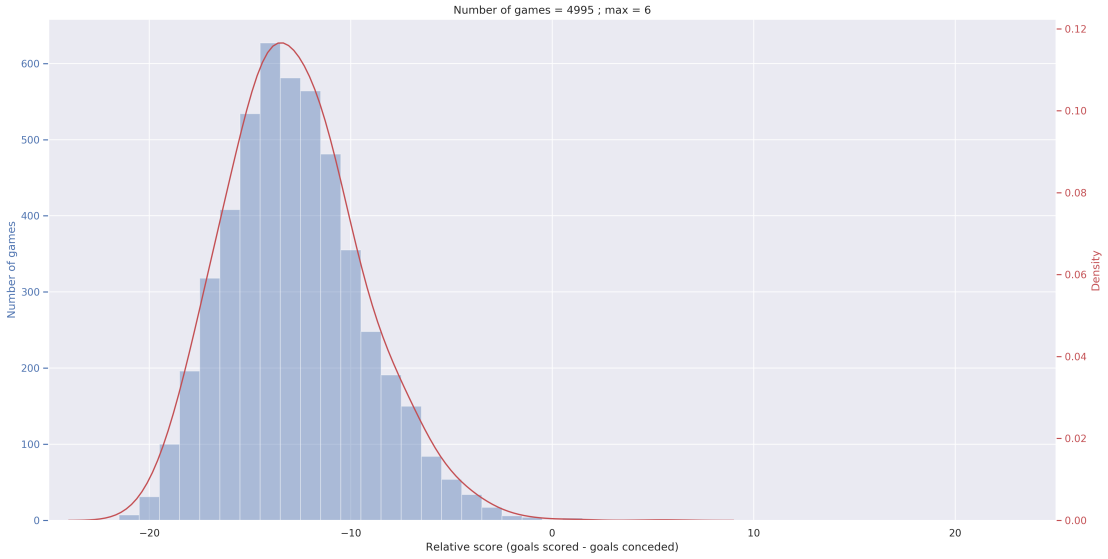


Figure 18: Entraînement du Deep Q-Network sur 7000 parties sans Max pooling

La figure 18 nous présente l'histogramme obtenu. On constate qu'en moyenne notre réseau perd en marquant environ 7 ou 8 buts. Au-delà de 10 buts marqués, la densité décroit fortement. On remarque cependant que cette décroissance n'empêche pas notre réseau d'avoir eu quelques très bons résultats. Ainsi le repérage du maximum (+6) nous permet de conclure que notre réseau a réussi à gagner une partie avec le score 21–15. Nous avons donc là notre premier cas de victoires de notre IA ! Bien que celle-ci subit de nombreuses défaites (parfois cuisantes), ce résultat très encourageant confirme que nous sommes sur la bonne voie et que notre implémentation est correcte et relativement performante.

Le nombre important de défaites peut être expliqué par la grande dimension de notre espace. En effet, notre réseau est extrêmenent complexe et peut ajuster énormément de poids pour interpoler correctement notre fonction de satisfaction Q. Pour pouvoir entraîner correctement un tel réseau, il faudrait avoir un apprentissage assez long ou un taux d'apprentissage plus élevé. Nous n'augmentons pas le taux d'apprentissage pour ne pas altérer la stabilité de notre système. Une piste que nous allons explorer par la suite est de rallonger l'apprentissage (à 15000 parties jouées).

Avant cela, nous voulons étudier dans le même contexte (donc 7000 parties) un réseau plus modeste. Il s'agit du réseau avec les couches de max pooling. La dimension de l'espace étant beaucoup plus faible, nous espérons une convergence plus rapide, et donc de meilleures résultats après 7000 parties.

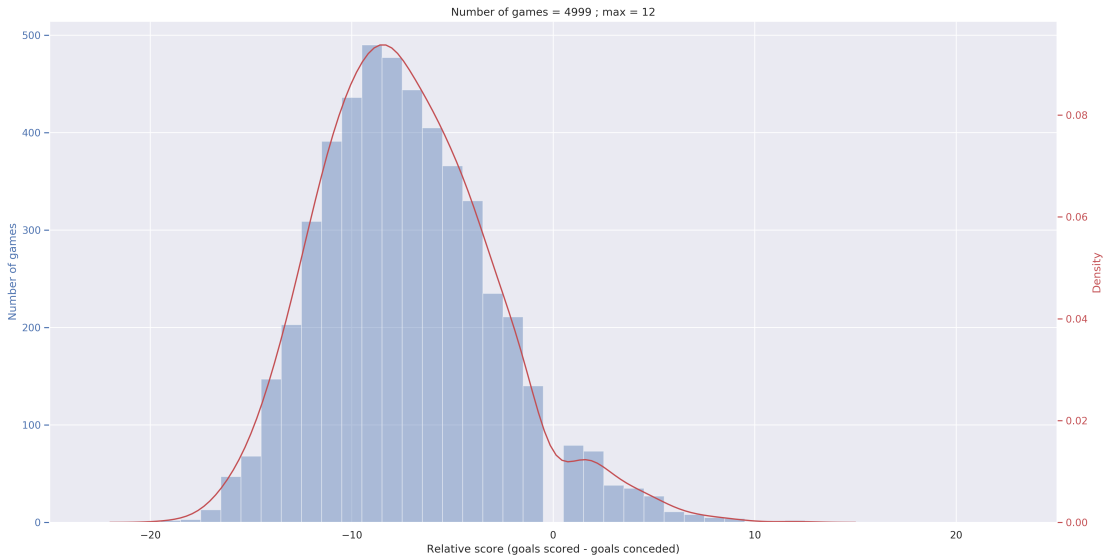


Figure 19: Entraînement du Deep Q-Network sur 7000 parties avec Max pooling

Nos attentes sont confirmées par la figure 19. D'une part, le maximum est bien plus important : +12. Cela signifie que notre IA a réussi à remporter la victoire 21–9 face à son adversaire ! D'autre part, c'est la totalité de la distribution des scores qui est modifiée. En effet, on situe plus la moyenne vers −8 (soit une défaite 13–21). En outre, les scores relatifs positifs (correspondant donc à une victoire) sont bien plus fréquents. Ils étaient totalement négligeables sur la figure 18. Ils sont désormais visibles bien que minoritaires. Cette IA se comporte globalement mieux dans l'environnement Pong. Elle a mieux appris (probablement car la dimension de l'espace est bien plus faible) et est par suite bien plus forte à ce jeu.

Notre objectif principal de ce projet est accompli ! Nous avons réussi à gagner à Pong grâce au Deep Q-Learning. Nos résultats demeurant toutefois imparfaits, nous allons étudier plus en profondeur l'apprentissage. Par exemple, nous pouvons prolonger ce dernier. Ou alors nous pouvons tester notre réseau plus régulièrement. Jusqu'à présent, les tests sont réalisés avec le réseau final et nous laissent peu d'informations pour interpréter la totalité de l'apprentissage. Une solution serait de récupérer le réseau à différentes étapes de l'entraînement et de le tester de la sorte pour obtenir des informations plus riches sur l'entraînement de notre IA.