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

Introduction **4**

1 Le Perceptron **5**
1.1 Modèle du perceptron . . . 5
1.2 Apprentissage . . . 7
1.2.1 Rétropropagation . . . 7
1.2.2 L'apprentissage par batch . . . 8
1.2.3 Taux d'apprentissage adaptatif . . . 8
1.3 La fonction d'erreur . . . 10
1.3.1 Comportement pour une erreur quadratique . . . 11
1.3.2 Origine de cet échec . . . 11
1.3.3 Une solution à ce problème d'apprentissage . . . 12
1.3.4 Une explication intuitive . . . 13
1.3.5 Estimateur de l'entropie croisée . . . 13
1.4 La fonction XOR . . . 14
1.5 MNIST . . . 14

2 Le CNN **16**
2.1 Le modèle du CNN . . . 16
2.2 Application pratique . . . 17
2.3 Résultats . . . 20
2.4 Utilisation de TensorFlow . . . 21

3 Le Q-Learning **23**
3.1 Qu'est-ce que le Q-Learning ? . . . 23
3.2 Calculer la fonction Q . . . 23
3.3 Jeu des bâtonnets . . . 24
3.4 Le labyrinthe . . . 26

4 Le Deep Q-Learning **27**
4.1 Principe du Deep Q-Learning . . . 27
4.2 Jeu des bâtonnets . . . 27
4.3 Changements effectués pour le passage à Pong . . . 28
4.3.1 Deux Deep Q-Networks . . . 28
4.3.2 Optimiseur RMSProp . . . 29
4.3.3 Changement de configuration de réseau . . . 29
4.3.4 Apprentissage . . . 29
4.3.5 Problèmes de mémoire . . . 29
4.4 Pong . . . 29
4.4.1 Apprentissage sur 3000 parties . . . 30
4.4.2 Apprentissage sur 7000 parties . . . 30
4.5 CartPole . . . 31
4.5.1 Principe du jeu . . . 31
4.5.2 Apprentissage de CartPole . . . 32
4.5.3 Apprentissage avec un seul réseau . . . 33
4.6 Flappy Bird . . . 34
4.6.1 Principe . . . 34
4.6.2 Apprentissage . . . 34

Conclusion **35**

Table des figures

1 Modèle d'un neurone artificiel . . . 5
2 Domaine de séparation du neurone . . . 6
3 Modèle d'une couche de neurones . . . 6
4 Modèle du perceptron multicouche . . . 6
5 Illustration des oscillations du gradient . . . 8
6 Erreur en test sur MNIST en fonction du temps d'entraînement pour différentes tailles de batch . 9
7 Erreur en test sur MNIST en fonction de l'époque pour différents taux d'apprentissage . . . 9
8 Erreur en test sur MNIST pour différents optimiseurs (SGD, RMSprop et ADAM) . . . 10
9 Première initialisation du réseau d'exemple utilisant l'erreur quadratique . . . 11
10 Seconde initialisation du réseau d'exemple utilisant l'erreur quadratique . . . 11
11 Graphe de la fonction sigmoïde . . . 12
12 Seconde initialisation du réseau d'exemple en utilisant l'entropie croisée . . . 12
13 Étude du XOR avec différents taux d'apprentissage . . . 14
14 Taux d'erreur sur MNIST . . . 15
15 Apprentissage MNIST avec différents taux d'apprentissage (régulièrement placés de 0,001 à 0,01) 15
16 Principe des filtres de convolution . . . 16
17 Image à détecter . . . 17
18 Choix des filtres pour la convolution . . . 17
19 Résultats de la convolution de l'image par les différents filtres . . . 18
20 Activation par la fonction reLU des différents canaux . . . 18
21 Pooling max des différents canaux avec S = 2 . . . 18
22 Image à éviter . . . 19
23 Choix des filtres pour la convolution . . . 19
24 Résultats de la convolution de l'image par les différents filtres . . . 19
25 Activation par la fonction reLU des différents canaux . . . 20
26 Pooling max des différents canaux avec S = 2 . . . 20
27 Comparaison des performances pour des perceptrons et pour un CNN . . . 21
28 Comparaison des performances de CNN à différentes tailles de filtres . . . 21
29 TensorBoard avec visualisation en direct des performances du réseau de neurones . . . 22
30 Taux de réussite en fonction du nombre d'apprentissages pour notre CNN sous TensorFlow . . . 22
31 Apprentissage contre une autre IA apprenant en même temps . . . 24
32 Apprentissage contre un joueur jouant aléatoirement . . . 25
33 Apprentissage contre un joueur appliquant la stratégie gagnante . . . 25
34 Chemin choisi par l'IA lorsque confrontée à un labyrinthe donné . . . 26
35 Schéma du Deep Q-Network . . . 27
36 Fonction de satisfaction Q approximée par le réseau neuronal . . . 28
37 Entraînement du Deep Q-Network sur 7000 parties sans Max pooling . . . 30
38 Entraînement du Deep Q-Network sur 7000 parties avec Max pooling . . . 31
39 Représentation du jeu CartPole . . . 32
40 Apprentissage de CartPole . . . 32
41 Comparaison des performances avec d'autres codes publics . . . 33
42 Apprentissage de CartPole avec un seul réseau . . . 33
43 Illustration de l'espace des états de Flappy Bird . . . 34

Introduction

Le projet long Ready Player One a pour but d'étudier le fonctionnement d'algorithmes d'apprentissage automatique. Ce projet d'un an, 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. Nous vous présenterons les résultats du perceptron que nous avons codé en Python.

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 :

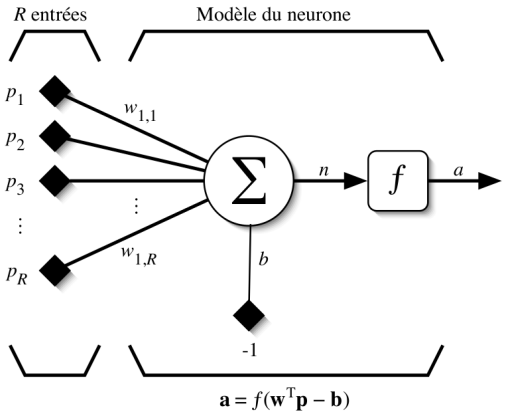


Figure 1: Modèle d'un neurone artificiel

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

On associe les entrées et les poids par un produit scalaire pour obtenir le niveau d'activation. Cette valeur caractérise l'entrée. On ajoute un biais pour régler l'importance accordée à ce niveau d'activation. On peut utiliser une notation matricielle pour simplifier les calculs. On pose $\mathbf{w_1} = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \end{pmatrix}^T$ 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 obtient :

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

On cherche alors à discriminer les différentes valeurs que peut prendre le niveau d'activation. C'est le rôle de la fonction d'activation. Par exemple, si l'on souhaite séparer les cas où n est supérieur ou inférieur à un seuil donné, 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 sigmoïde ($\sigma : x \mapsto \frac{1}{1+e^{-x}}$) ou encore la tangente hyperbolique. On préfère généralement utiliser des fonctions différentiables. On verra en effet dans la suite que ces fonctions permettent au réseau d'apprendre sur les données fournies.

Finalement on obtient :

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

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 vaut 1. Puisque le niveau d'activation résulte d'un produit matriciel, cela définit l'équation d'un hyperplan. La séparation est donc linéaire.

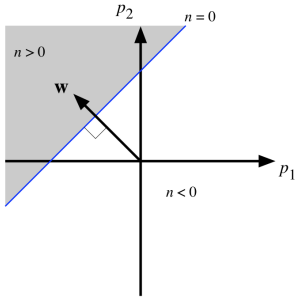


Figure 2: Domaine de séparation du neurone

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 peuvent être vus comme des poids dont l'entrée est constante à -1), et un vecteur de niveaux d'activation. Finalement, on retrouve le modèle suivant :

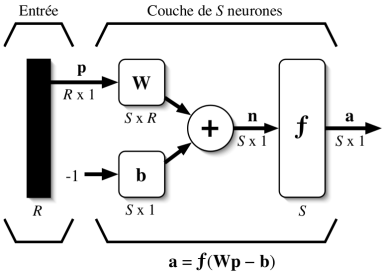


Figure 3: Représentation matricielle d'une couche de S neurones recevant R entrées

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.

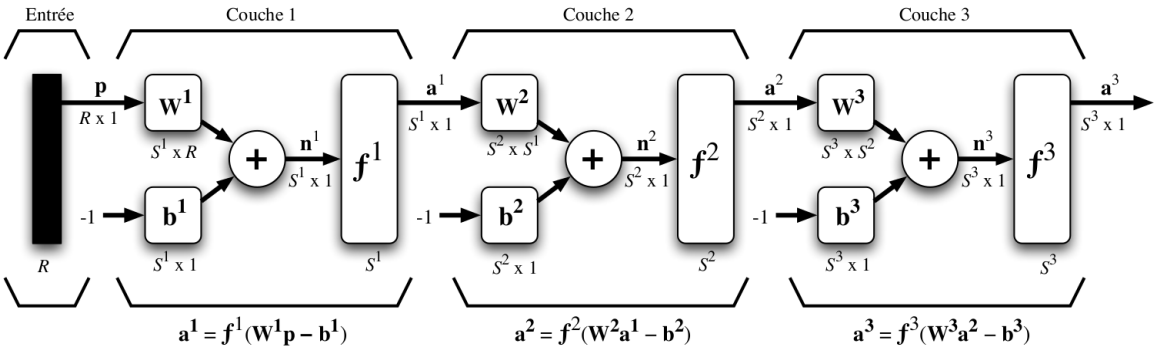


Figure 4: Modèle du perceptron multicouche

1.2 Apprentissage

1.2.1 Rétropropagation

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 un apprentissage supervisé, 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 l'erreur est importante, moins le réseau est adapté pour cette donnée. C'est-à-dire que les poids du réseau ne permettent pas de déduire la bonne réponse à partir de l'entrée.

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 :

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

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

L'apprentissage consiste donc à minimiser 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 :

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

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 fixer en amont de l'apprentissage.

Marc Paruzeau a démontré en 2004 les formules de rétropropagation que nous avons utilisées. Pour cela il introduit une variable intermédiaire, la sensibilité. La sensibilité est définie par :

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

On note également l'utilisation du raccourci suivant :

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

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

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

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

La rétropropagation permet de calculer le gradient de chacun des poids relatifs à l'erreur pour chacun des couples du set d'entraînement. Il faut ensuite soustraire ce gradient aux poids du réseau de neurones pour minimiser l'erreur. Mais la méthode triviale n'est pas nécessairement celle qui va faire converger le réseau de neurones le plus rapidement vers le meilleur minimum.

1.2.2 L'apprentissage par batch

Comme le but est de diminuer l'erreur totale sur l'ensemble d'entraînement, on peut penser que la solution optimale serait de calculer le gradient sur l'ensemble des éléments avant de le soustraire, mais cette méthode a plusieurs désavantages :

- Il faut calculer l'erreur sur tous les éléments de l'ensemble d'entraînement à chaque mise à jour à jour des poids, ce qui est coûteux
- On reste dans le premier minimum local dans lequel on tombe (pour peu qu'il soit plus grand que le pas avec lequel on avance i.e. le taux d'apprentissage)

En effet la descente de gradient ne garantie pas la convergence vers un minimum global, et peut très bien se contenter d'un mauvais minimum local. Il existe donc de nombreuses heuristiques visant à améliorer la vitesse de convergence, et la qualité du minimum trouvé.

Le but de l'apprentissage par batch est de segmenter les données d'apprentissage en groupe appelés batchs pour accélérer l'apprentissage, et introduire une composante aléatoire dans la manière dont le modèle converge vers un minimum.

Pour un apprentissage par batch de taille N, on découpe l'ensemble d'entraînement en groupes de N éléments choisis au hasard.

Pour chacun de ces batchs :

- on calcule *sans mettre à jour le réseau de neurone* le gradient pour chacun des N éléments du batch
- on moyenne le gradient sur les N éléments du batch
- on applique le gradient moyenné au réseau de neurone

On ne calcule donc pas le gradient sur l'entièreté de l'ensemble de test, mais sur seulement des parties tirées au hasard. Cela a deux conséquence :

- on réalise le calcul de l'erreur sur seulement N éléments pour chaque mise à jour du réseau
- le fait que le batch soit tiré au hasard permet de ne pas aller directement dans le sens du gradient global, mais d'osciller légérement autour de celui-ci. Ces oscillations permettent d'éviter de rester dans un mauvais minimum local trop peu profond.

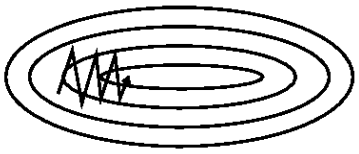


Figure 5: Illustration des oscillations du gradient

Évidemment, la taille N du batch est un paramètre important à considérer.

Si N est grand, l'apprentissage prend plus de temps car il faut calculer l'erreur sur un plus grand nombre d'éléments avant de mettre à jour le modèle.

Si N est petit, il est plus difficile d'utiliser les capacités de calcul du hardware (CPU ou GPU) pour faire la rétropropagation en parallèle sur les N éléments du batch. De plus le gradient a tendance à beaucoup osciller, rendant la convergence moins rapide.

Dans les faits, il est préférable d'utiliser des mini-batchs d'une taille de l'ordre de la dizaine d'éléments.

1.2.3 Taux d'apprentissage adaptatif

Pour l'instant nous nous contentons de soustraire le gradient aux poids, multiplié par un facteur η, le taux d'apprentissage.

Dans ce processus, il est important de choisir un taux d'apprentissage adapté au modèle, et à l'ensemble d'entraînement. Plus le taux d'apprentissage est petit, plus la convergence est longue, mais si ce taux est trop grand, on peut avoir du mal à converger en fin d'apprentissage vers un bon minimum, parce qu'on fait des pas plus grands que le minimum vers lequel on voudrait converger.

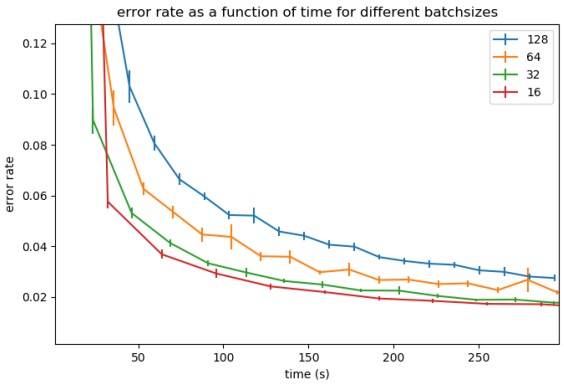


Figure 6: Erreur en test sur MNIST en fonction du temps d'entraînement pour différentes tailles de batch

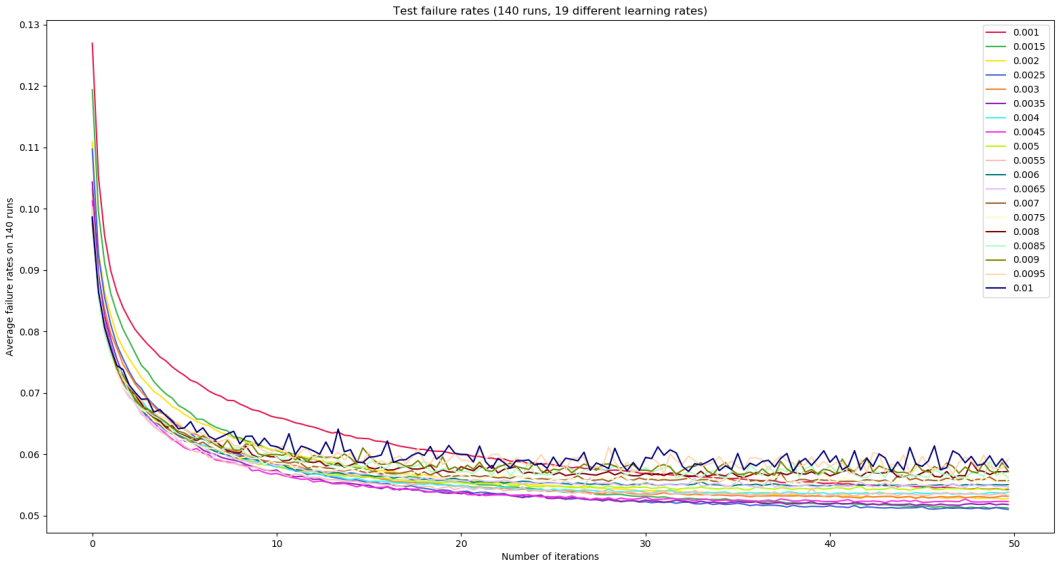


Figure 7: Erreur en test sur MNIST en fonction de l'époque pour différents taux d'apprentissage

On remarque sur la figure 7 qu'il existe pour ce modèle un taux optimal de 0.003, et que le taux d'apprentissage peut avoir une légère influence sur le taux d'erreur, de l'ordre du pourcent.

Il est donc intéressant premièrement de faire varier le taux d'apprentissage au long de l'apprentissage de manière à aller vite au début et plus lentement à la fin. En outre, on peut faire varier le taux d'apprentissage individuellement pour chacun des paramètres du modèle.

RMSprop RMSprop est l'une de ces méthodes. Le but de RMSprop est de se pré-occuper uniquement du sens du gradient, et non de sa valeur.

On note g_t le gradient du modèle au t-ième apprentissage, η le taux d'apprentissage, $\epsilon = 1e-8$, θ_t les poids du modèle et $\beta \in [0, 1]$. On pose :

$$E[g^2]_{t+1} = \beta E[g^2]_t + (1-\beta) g_t^2 \tag{12}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{13}$$

RMSprop est censé normaliser chaque paramètre du gradient de sorte qu'il ne soit ni trop important ni trop faible.

ADAM ADAM est une seconde méthode de gradient adaptatif. Avec les mêmes notations que précédemment, on a :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{14}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{15}$$

m_t et v_t estiment respectivement le moment d'ordre 1 et 2 du gradient. Ils sont cependant biaisés, on pose donc

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \tag{16}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{17}$$

et finalement on met à jour les poids

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{18}$$

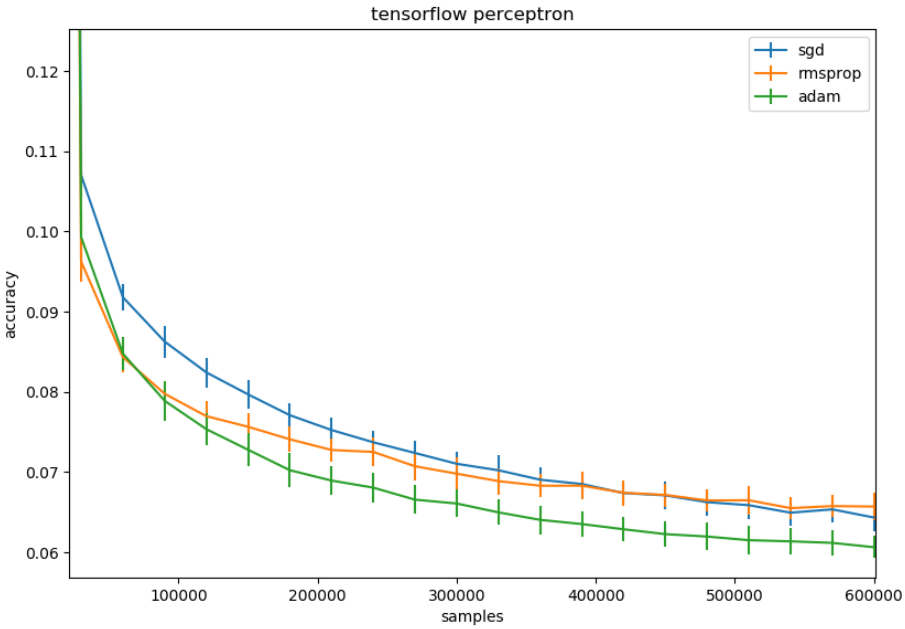


Figure 8: Erreur en test sur MNIST pour différents optimiseurs (SGD, RMSprop et ADAM)

On remarque que, dans le cas du modèle de la figure 8, ADAM est beaucoup plus performant que la descente de gradient simple et que RMSprop. Ça n'est cependant pas toujours le cas.

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) ou l'erreur moyenne (norme 1 du vecteur d'erreur). Pour obtenir l'erreur d'un groupe de données, appelé batch, on somme les erreurs de chaque donnée. Par souci 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 la suite pourquoi cette fonction possède de meilleures propriétés que l'erreur quadratique.

La formule de l'entropie croisée est la suivante :

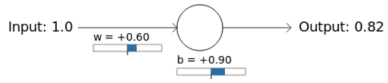
$$C = -\frac{1}{n}\sum_x \left(y \ln(a) + (1-y)\ln(1-a)\right) \text{ , avec } \begin{array}{l} n \text{ la taille du batch} \\ x \text{ les exemples du batch} \\ a \text{ la sortie calculée} \\ y \text{ la sortie attendue} \end{array} \tag{19}$$

1.3.1 Comportement pour une erreur quadratique

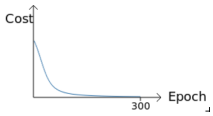
Pour comprendre l'intérêt de l'entropie croisée, nous devons comprendre pourquoi la norme euclidienne n'est pas satisfaisante.

Le concept d'entropie provient 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, on considèrera un réseau neuronal trivial (un neurone à une entrée et une sortie) avec une fonction d'erreur quadratique. Les phénomènes observés sur ce neurone simple pourront être généralisés aux réseaux plus complexes. On dispose donc d'un neurone, avec une entrée (avec un biais) et une sortie. On souhaite lui faire 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 9a, on a, après initialisation, un neurone qui renvoie 0,82 lorsque l'entrée vaut 1. On entraîne ce neurone et obtient la courbe d'apprentissage 9b.



(a) Réseau utilisé et son initialisation

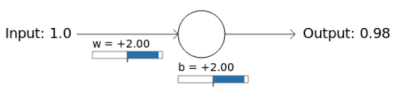


(b) Courbe d'apprentissage du neurone

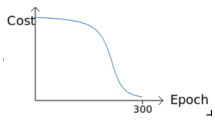
Figure 9: Première initialisation du réseau d'exemple utilisant l'erreur quadratique

Sur la courbe 9b, le coût est en unité arbitraire. Sa valeur en soit n'a pas beaucoup d'importance. En effet, les informations pertinentes de cette courbe ne sont pas les valeurs initiales et finales mais plutôt son allure générale. Augmenter le nombre d'itérations permettrait en effet de réduire ces valeurs de manière arbitrairement faible. 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 10b. On remarque que l'apprentissage est de moins bonne qualité. En effet, juste après l'initialisation, le neurone commet une erreur plus importante et l'apprentissage est beaucoup plus lent. On doit donc réaliser un nombre plus d'itérations pour retrouver le cas 9 et pouvoir apprendre correctement.



(a) Réseau utilisé et son initialisation



(b) Courbe d'apprentissage du neurone

Figure 10: 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 :

$$\frac{\partial C}{\partial \omega} = (a-y)\,\sigma'(z)\,x, \text{ où } \begin{array}{l} z \text{ est l'antécédent de } a \text{ par la} \\ \text{fonction d'activation} \end{array} \tag{20}$$

$$\frac{\partial C}{\partial b} = (a-y)\,\sigma'(z) \tag{21}$$

En observant la fonction sigmoïde représentée sur la figure 11, 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 à droite de la courbe, où la pente est très faible. Cela provient du fait que $\sigma'(z) = a(1-a)$. L'apprentissage est alors très ralenti, ce qui n'est évidemment pas souhaitable.

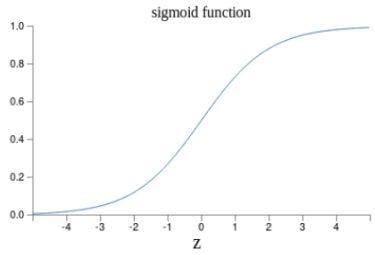


Figure 11: Graphe de la fonction sigmoïde

1.3.3 Une solution à ce problème d'apprentissage

Pour rendre le réseau moins sensible à une mauvaise initialisation, il faut corriger le comportement observé précédemment. Si l'on fait une analogie avec le comportement humain, l'humain a tendance à apprendre plus vite lorsqu'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 :

$$\frac{\partial C}{\partial \omega} = (a-y)\,x \tag{22}$$

$$\frac{\partial C}{\partial b} = (a-y) \tag{23}$$

La formule de la chaîne nous donne

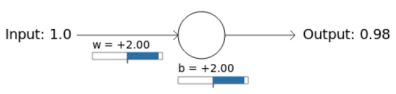
$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial b} = \frac{\partial C}{\partial a}a(1-a) \tag{24}$$

En utilisant l'expression voulue (équation 23), on obtient

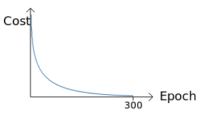
$$\frac{\partial C}{\partial a} = \frac{a-y}{a(1-a)} = \frac{-y}{a} + \frac{1-y}{1-a} \tag{25}$$

$$C = -y\ln(a) - (1-y)\ln(1-a) + Const \tag{26}$$

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 22 et 23. L'expression 19 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 10. La courbe d'apprentissage 12b 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 10



(b) Courbe d'apprentissage du neurone

Figure 12: 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 19. 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 :

$$H(X) = -\sum_{i=1}^{n} p_i \log(p_i) \tag{27}$$

L'entropie est une mesure de l'incertitude d'une loi de probabilité. Par exemple, si on réalise une expérience de tirage 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.

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 pour 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 :

$$\mathbb{P}(X = y_i) = \begin{cases} q_i & \text{si } y_i = 1 \\ 1 - q_i & \text{si } y_i = 0 \end{cases}, \text{ où } X \text{ représente notre réseau et } y_i \text{ représente l'observation à la donnée } x_i \tag{28}$$

$\mathbb{P}$, tel que défini par l'équation 28, représente la probabilité que notre réseau trouve la bonne solution. Sa densité de probabilité est donnée sous une forme différente par l'équation 29

$$\mathbb{P}(X = y_i) = {q_i}^{y_i} (1 - q_i)^{1-y_i} \tag{29}$$

Les équations 30 et 31 ci-dessous présentent respectivement la vraisemblance et la log-vraisemblance de notre loi.

$$\mathbb{L} = \prod_{i=1}^{N} {q_i}^{np_i} (1 - q_i)^{n(1-p_i)} \tag{30}$$

$$\frac{1}{n}\log(\mathbb{L}) = \sum_{i=1}^{N} (p_i \ln(q_i) + (1 - p_i)\ln(1 - q_i)) \tag{31}$$

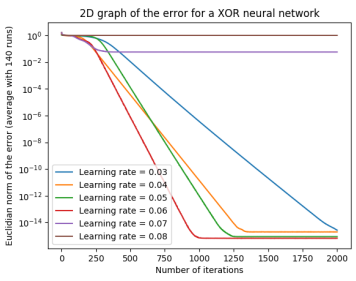
D'après les équations 19 et 31, la log-vraisemblance de notre système est ainsi proportionnelle à l'opposé de l'entropie croisée. Ainsi, on comprend que minimiser l'entropie croisée revient à utiliser un estimateur de maximum de vraisemblance. Notre système est donc d'autant plus performant que son entropie croisée est faible.

1.4 La fonction XOR

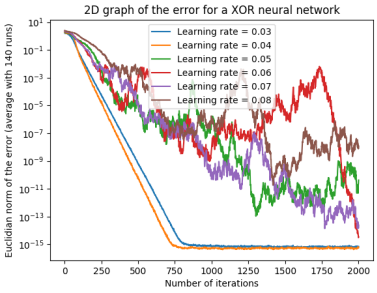
Nous avons développé notre propre perceptron en Python à partir des équations vues plus haut.

Pour le tester, nous avons utilisé la fonction XOR. Elle prend en entrée deux valeurs binaires et retourne 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. Il faut que notre réseau possède une couche intermédiaire.

Nous configurons notre réseau de la manière suivante. La couche d'entrée ainsi que la couche cachée contiennent deux neurones tandis que la couche externe est constituée d'un neurone. La couche cachée et la couche de sortie sont chacune suivies 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. Cela nous permet de tirer nos premières conclusions.



(a) XOR avec 2 neurones cachés



(b) XOR avec 30 neurones cachés

Figure 13: Étude du XOR avec différents taux d'apprentissage

La figure 13a 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 13b 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 difficultés à trouver le minimum. Les courbes sont globalement très bruitées. Néanmoins, cette fois-ci, tous les réseaux ont convergé même lorsque le taux d'apprentissage était élevé. C'est cependant les taux les plus bas qui ont été le moins affectés 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 perceptron à reconnaître le chiffre dans l'image en analysant ses 784 pixels. 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 (les 10 chiffres possibles). 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 les résultats 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 (280 runs). Le taux d'apprentissage utilisé est 0.005.

D'après la figure 14, 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. On cherche à minimiser l'erreur sur l'échantillon de test pour éviter le sur-apprentissage, qui consiste à apprendre par cœur les exemples au lieu d'apprendre les propriétés généralisables. Sur MNIST, cela a relativement

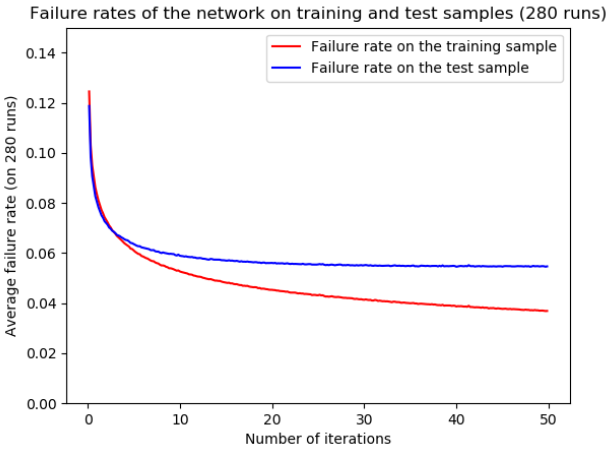


Figure 14: Taux d'erreur sur MNIST

peu d'importance 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 15 avec un intervalle de confiance à 95% (notamment visibles sur la dernière case).

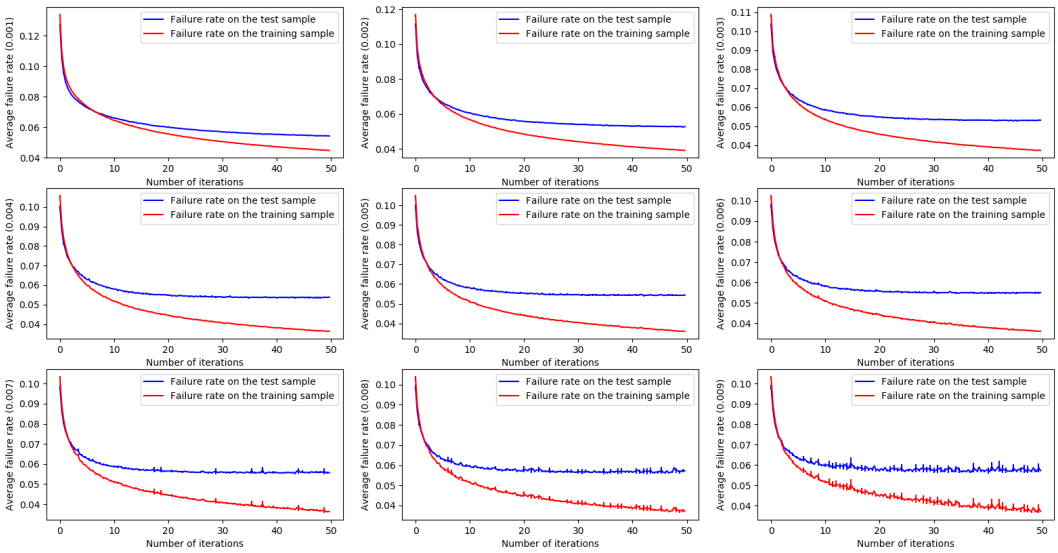


Figure 15: Apprentissage MNIST avec différents taux d'apprentissage (régulièrement placés de 0,001 à 0,01)

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, la convergence étant rapide et avec un faible écart-type. Néanmoins, l'impact sur les performances est assez négligeable (moins de 1% de différence).

2 Le CNN

2.1 Le modèle du CNN

Bien qu'étant un modèle efficace, le perceptron reste coûteux en calcul : il faut réaliser les opérations matricielles sur tous les neurones de chaque couche. Les réseaux neuronaux convolutifs (en anglais Convolutional Neural Network, CNN) permettent de réduire le nombre de calculs réalisés. Ils se basent sur la structure des données à classifier. Les images sont un bon exemple de structure permettant de diminuer le nombre de calculs à réaliser. En effet, une image peut être caractérisée par les motifs locaux qui la constituent. Ces motifs sont en général localisés. Il n'est donc pas nécessaire de chercher un lien entre deux points éloignés d'une image.

Les CNN sont entraînés à chercher des motifs sur une partie restreinte de l'image. Le principe est le même que celui des perceptrons, sauf qu'au lieu d'appliquer un réseau entièrement connectés sur toutes les couches, on va réaliser une convolution par différents filtres sur l'image. Chaque filtre aura pour but de détecter un motif dans l'image.

Une couche d'un CNN est typiquement constituée de 3 éléments :

1. Une convolution
2. Une fonction d'activation
3. Une fonction de pooling

La couche de convolution prend en entrée une image de dimension $(H,\ L,\ N)$ avec H la hauteur de l'image, L sa largeur et N le nombre de canaux (pour une image RGB, on aura $N = 3$). On applique en parallèle sur cette image M filtres qui donneront M images de dimension deux. Ces filtres sont de même dimension $(h,\ l,\ n)$ avec $h < H$, $l < N$ et $n = N$. On obtient donc M images de même dimension en sortie. Ces M images peuvent être vues comme une image à trois dimensions avec M canaux. On peut alors répéter le processus en considérant cette nouvelle image comme une entrée pour les M' filtres suivants.

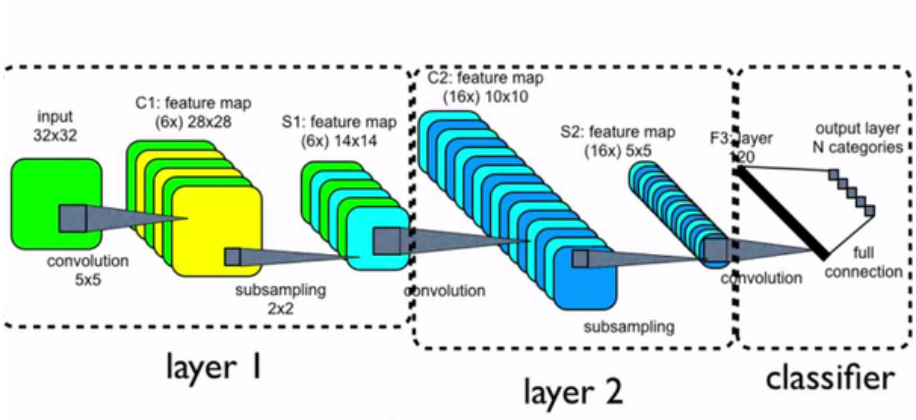


Figure 16: Principe des filtres de convolution

Pour une image I de dimension $(H,\ L,\ N)$ et un filtre F de dimension $(h,\ l,\ N)$, on obtient une image J de dimension $(H - h,\ L - l)$, la convolution est réalisée de la manière suivante :

$$J(x,y) = I * H\,(x,y) = \sum_{i=0}^{h-1}\sum_{j=0}^{l-1}\sum_{k=0}^{N-1} I\,(x+i, y+j, k) \times H\,(i,j,k) \tag{32}$$

On obtient alors une sortie de dimension $(H - h,\ L - l,\ M)$.

Intuitivement, cette partie sert à chercher la ressemblance entre le filtre et l'image. On devrait avoir $J(x,y) < 0$ si la partie de l'image I en (x,y) est très différente du filtre appliqué, et $J(x,y) > 0$ sinon, avec un score plus ou moins élevé selon la ressemblance.

On applique ensuite la couche d'activation. On utilise une fonction non linéaire comme dans le cas du perceptron. Généralement, la fonction utilisée pour les couches de convolution intermédiaires est la fonction reLU :

$$reLU(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases} \tag{33}$$

Cette fonction sert à ramener le résultat dans les réels positifs pour éviter une divergence au niveau des calculs, et pour augmenter l'écart relatif entre un bon score et un mauvais score : un score de 0 et en score de -1 obtenu lors de la convolution sont alors considéré comme tout aussi mauvais.

Enfin, on applique une couche de pooling, qui sert à réduire le nombre de données obtenues en sortie des 2 couches précédentes. Le pooling cherche à "résumer" les scores d'une partie de l'image, en donnant un score qui dépend des pixels de cette partie. Il existe plusieurs méthode de pooling, comme par exemple le pooling par moyenne ou le pooling par maximum. C'est ce dernier qui est le plus utilisé en pratique : pour une image J, on choisi un paramètre S qui correspond au pas ($S < L$, $S < H$) et on prend la sortie K telle que :

$$K(x, y) = max_{i,j \in [0,S-1]^2} J(x \times S + i, y \times S + j) \tag{34}$$

On enchaîne ainsi les couches de convolution jusqu'à une couche finale, qui sera reliée à un réseau complètement connecté (qui n'est rien d'autre qu'un simple perceptron). Les poids à optimiser sont alors les poids des matrices qui définissent les filtres et les poids de la couche entièrement connectée. Dans le principe, un CNN est comme un perceptron (à part pour la couche de pooling, on pourrait construire un CNN à partir d'un un perceptron), à l'exception du fait que certains poids seraient liés lors de l'apprentissage.

2.2 Application pratique

Afin de mieux comprendre le principe du CNN, nous avons réalisé un exemple simple mais clair : on cherche à reconnaître le motif d'une croix. Pour cela, on prend les filtres f_1 (diagonale gauche), f_2 (diagonale droite) et f_3 (croix centrale). Alors en appliquant la convolution, l'activation reLU et un pas à $S = 2$, on obtient le résultat suivant (figure 19) :

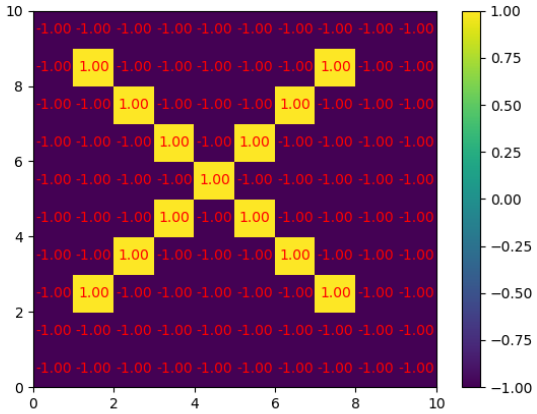
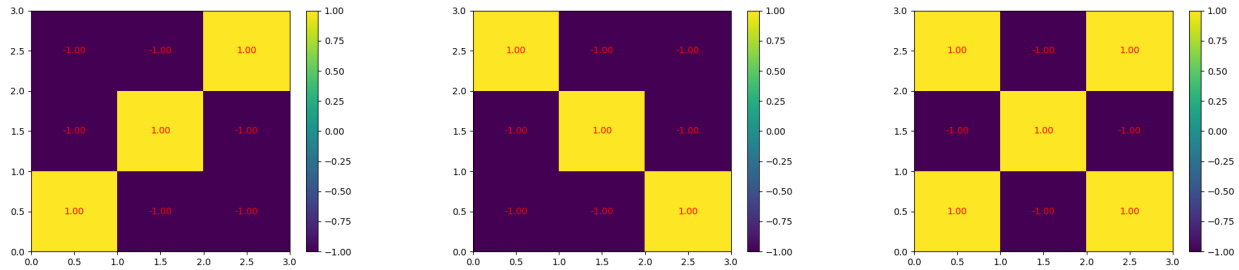
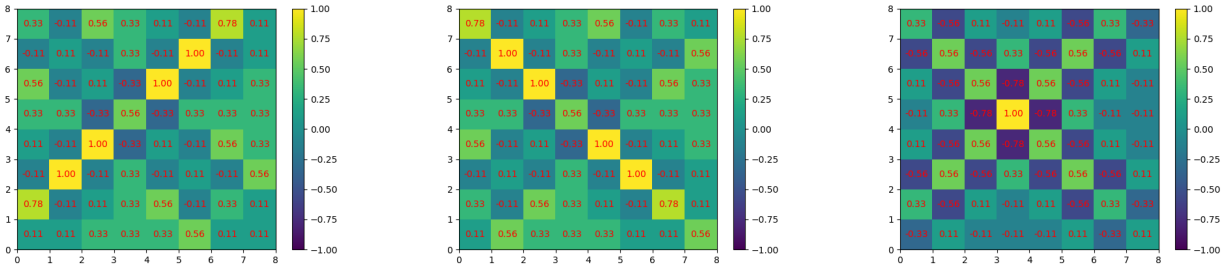


Figure 17: Image à détecter



(a) Filtre 1 (b) Filtre 2 (c) Filtre 3

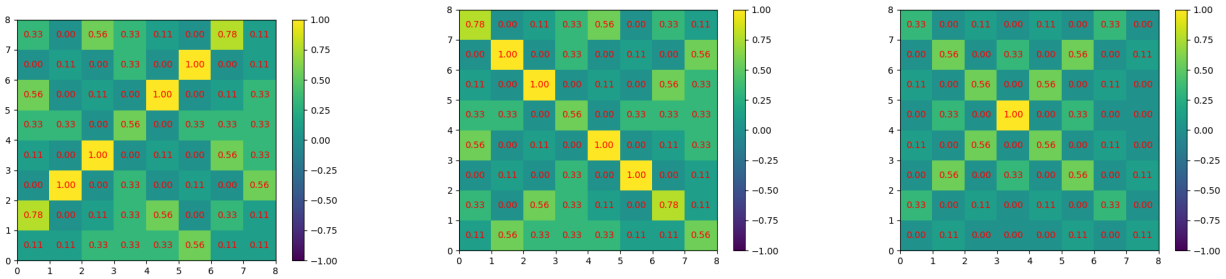
Figure 18: Choix des filtres pour la convolution



(a) Convolution par le filtre 1 (b) Convolution par le filtre 2 (c) Convolution par le filtre 3

Figure 19: Résultats de la convolution de l'image par les différents filtres

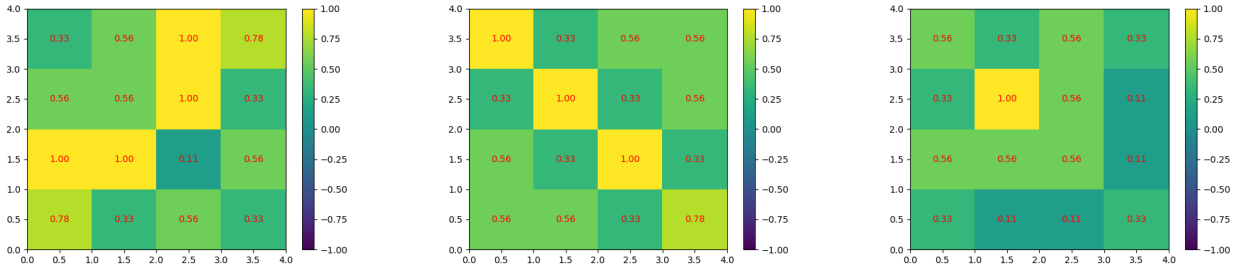
On remarque que les filtres en question réalisent bien l'action voulue : pour la première convolution, les positions des diagonales gauches sont mises en valeur, pour la seconde convolution, les diagonales droites sont mises en valeurs, et pour la troisième convolution, la croix centrale est mise en valeur. On obtient ainsi 3 canaux.



(a) Activation reLU pour le canal 1 (b) Activation reLU pour le canal 2 (c) Activation reLU pour le canal 3

Figure 20: Activation par la fonction reLU des différents canaux

Après l'activation, on remarque peu de changement quant aux positions des motifs recherchés : ces positions sont toujours bien présentes, tandis que dans le reste de l'image où on ne retrouve pas les motifs, les valeurs sont "mises à niveau" pour bien insister sur le fait qu'elles n'apporteront pas d'informations pertinentes lors du traitement par perceptron. On applique ensuite une étape de pooling.



(a) Pooling du canal 1 (b) Pooling du canal 2 (c) Pooling du canal 3

Figure 21: Pooling max des différents canaux avec S = 2

On peut ainsi réduire les dimensions de l'image, tout en conservant une position globale des caractéristiques intéressantes.

On aperçoit alors l'apparition de zones à scores élevés par rapport au reste. On passe ensuite ces résultats dans un "réseau complètement connecté" (fully connected network) afin de pouvoir exploiter le contraste que ce réseau à convolution a mis en valeur.

Si on applique le même traitement à une autre figure (ici un carré), on remarque que le résultat est beaucoup moins contrasté :

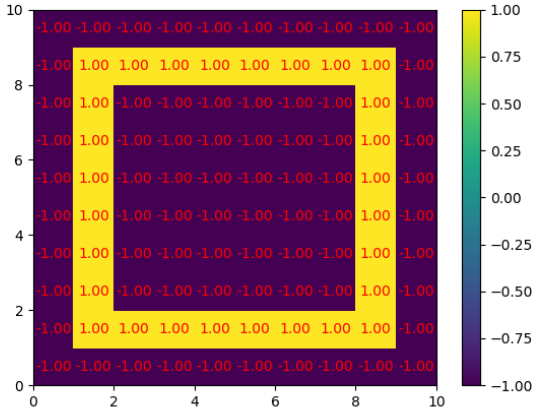
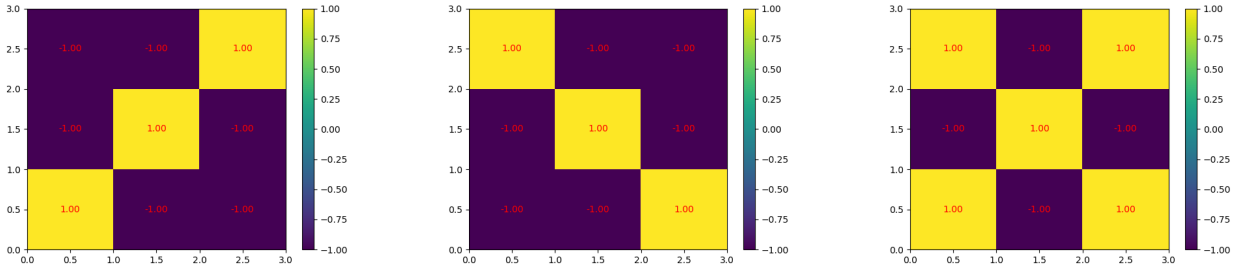
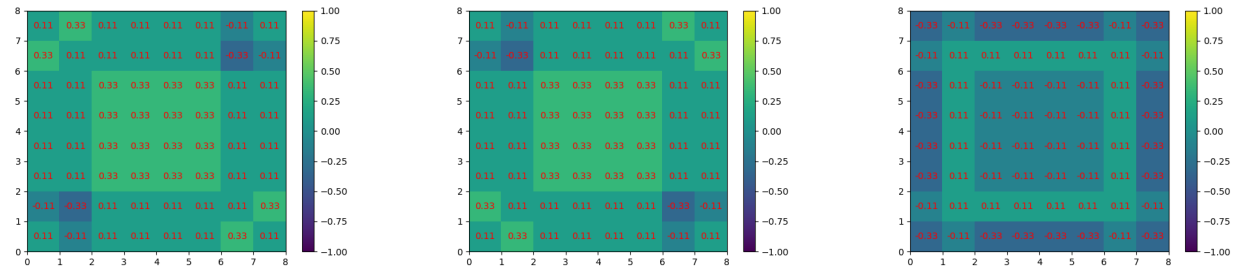


Figure 22: Image à éviter



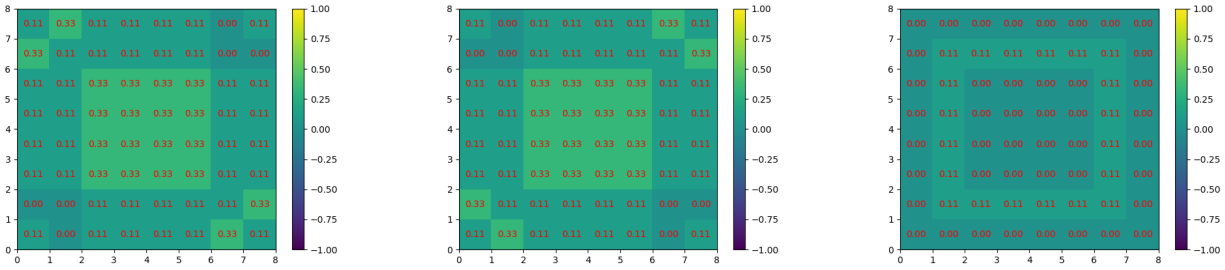
(a) Filtre 1 (b) Filtre 2 (c) Filtre 3

Figure 23: Choix des filtres pour la convolution



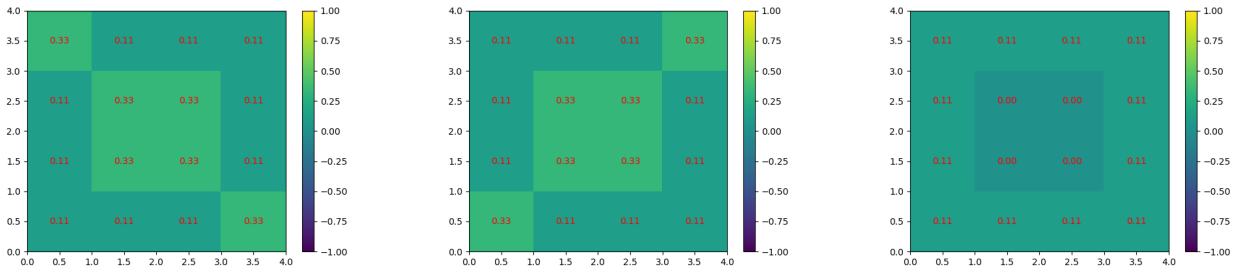
(a) Convolution du carré par le filtre 1 (b) Convolution du carré par le filtre 2 (c) Convolution du carré par le filtre 3

Figure 24: Résultats de la convolution de l'image par les différents filtres



(a) Activation reLU pour le canal 1 (b) Activation reLU pour le canal 2 (c) Activation reLU pour le canal 3

Figure 25: Activation par la fonction reLU des différents canaux



(a) Pooling du canal 1 (b) Pooling du canal 2 (c) Pooling du canal 3

Figure 26: Pooling max des différents canaux avec S = 2

On remarque que le résultat final est beaucoup plus flouté, et n'apporte pas d'informations pertinentes pour le réseau de neurones arrivant derrière.

Dans le cas présenté ci-dessus, nous avons choisi nous-même les paramètres des filtres appliqués pour la convolution, en se basant sur l'intuition. En pratique, les poids des matrices composants les filtres sont appris comme des paramètres standards d'un réseau de neurones.

2.3 Résultats

Nous avons comparé les performances du CNN avec un réseau de neurones complètement connecté sur la base de données MNIST.

Les entraînements ont été réalisé sur 60000 éléments, par minibatches de 32.

Les réseaux considérés sont les suivants :

- Un perceptron de petite dimension (une couche cachée de taille 512)
- Un perceptron de grande dimension (deux couches cachées, de taille 4096 et 64)
- Un CNN (32 filtres de tailles 3×3, 64 filtres de tailles 3×3, un pooling max de taille 2× 2, une couche cachée de taille 128)

Le perceptron de grande dimension a été choisi de façon à avoir un nombre équivalent de paramètres avec le CNN.

On voit (figure 27) qu'au bout d'un passage complet sur les données (par minibatches de 32), le CNN obtient de meilleurs résultats que les perceptrons. De plus, on remarque que le perceptron de grande dimension converge plus lentement que le réseau de petite dimension. Le CNN sert alors à obtenir une convergence plus rapide (comme un perceptron de petite taille), et obtient de meilleurs résultats à termes car il n'est pas limité par son nombre de paramètres (comme un perceptron de grande dimension).

La taille de filtres a également une influence sur l'entraînement du CNN. En effet, les filtres ont pour but de trouver des "motifs" dans l'image et de le mettre en valeur par rapport au reste de l'image. Plus le motif est petit, plus il sera facile d'entraîner le filtre, mais le motif alors trouvé ne sera pas forcement pertinent pour la

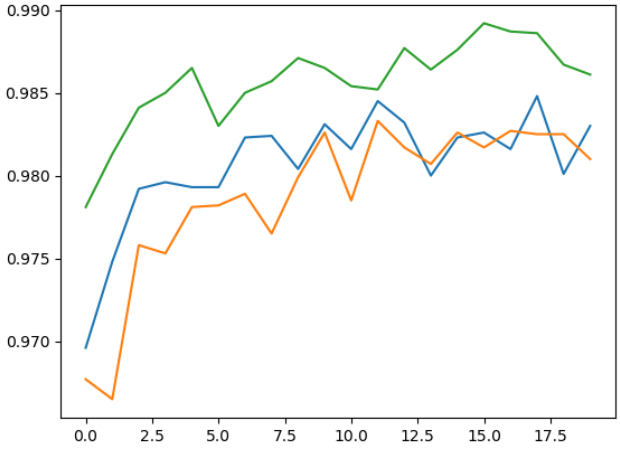


Figure 27: Comparaison des performances pour des perceptrons et pour un CNN en fonction du nombres d'époques. CNN en vert, perceptron de petite dimension en bleu, perceptron de grande dimension en orange

classification. Inversement, plus le filtre est de grande dimension, plus il pourra rechercher des motifs complexes, mais la convergence sera alors plus compliquée et pas aussi stable.

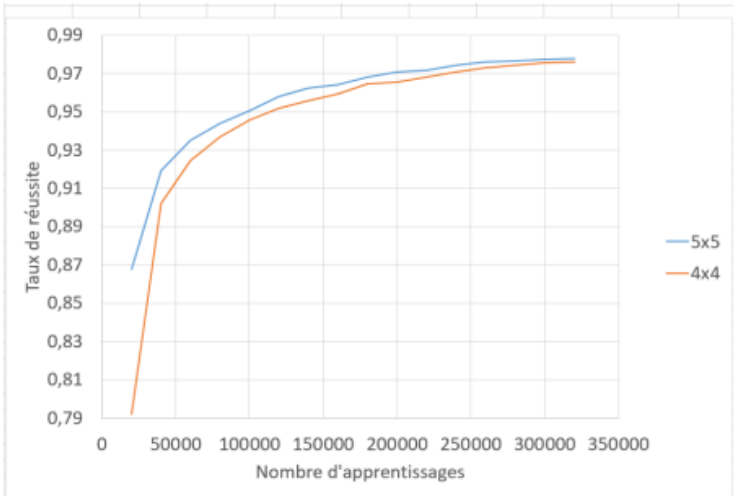


Figure 28: Comparaison des performances de CNN à différentes tailles de filtres

En observant les résultats, on remarque que les filtres de 5×5 ont un bon rapport performance/vitesse de convergence.

2.4 Utilisation de TensorFlow

TensorFlow a été utilisé pour développer le CNN. Un développement à la main du CNN aurait été possible et très instructif mais complexe et chronophage. L'utilisation de TensorFlow a permis d'avancer plus rapidement sur la programmation du CNN. Par ailleurs, le fonctionnement de chaque bloc utilisé par TensorFlow pour développer un CNN avait été vu en détail précédemment, ce qui a permis de comprendre ce qui se cachait derrière les fonctions de haut niveau proposées par TensorFlow.

Cependant, en plus de réduire grandement le temps nécessaire pour programmer un CNN efficace, TensorFlow offre une interface graphique (figure 29) très utile pour suivre en direct la convergence du CNN. On peut ainsi détecter rapidement le mauvais paramétrage du CNN et le modifier sans attendre d'avoir mis à jour le CNN sur les n batches prévus à l'avance.

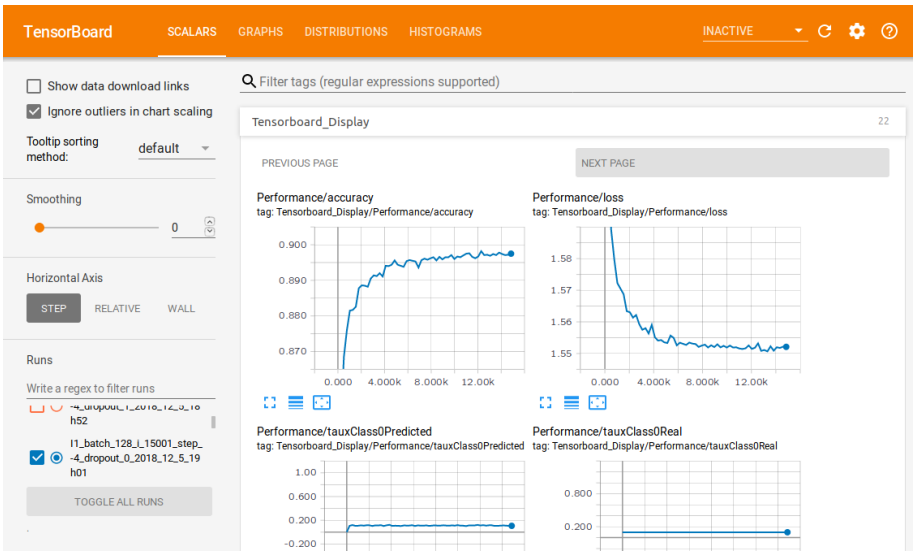


Figure 29: TensorBoard avec visualisation en direct des performances du réseau de neurones

L'exploitation des résultats a posteriori s'avère plus compliquée. L'interface de TensorFlow offrant des fonctionnalités plutôt limitées, l'utilisation d'un post-traitement est indispensable. Il est par exemple impossible de réaliser des moyennes sur plusieurs apprentissages. Après quelques recherches, nous avons réussi à extraire le taux de réussite en fonction du nombre d'apprentissages à partir des données brutes générées par TensorFlow. L'extraction du taux de réussite en fonction du temps de calcul s'est avérée plus compliquée. Nous remercions Julien Gérard qui nous a beaucoup aidé sur ce point. Il nous a en effet indiqué avoir trouvé dans le code source de TensorFlow la fonction `wall_time` qui permet d'extraire le temps écoulé depuis le lancement des calculs pour chaque relevé du taux de réussite.

Nous avons testé TensorFlow avec un CNN pour MNIST. Grâce au post-traitement, nous avons pu tracer une moyenne du taux de réussite en fonction du nombre d'apprentissages réalisées (figure 30).

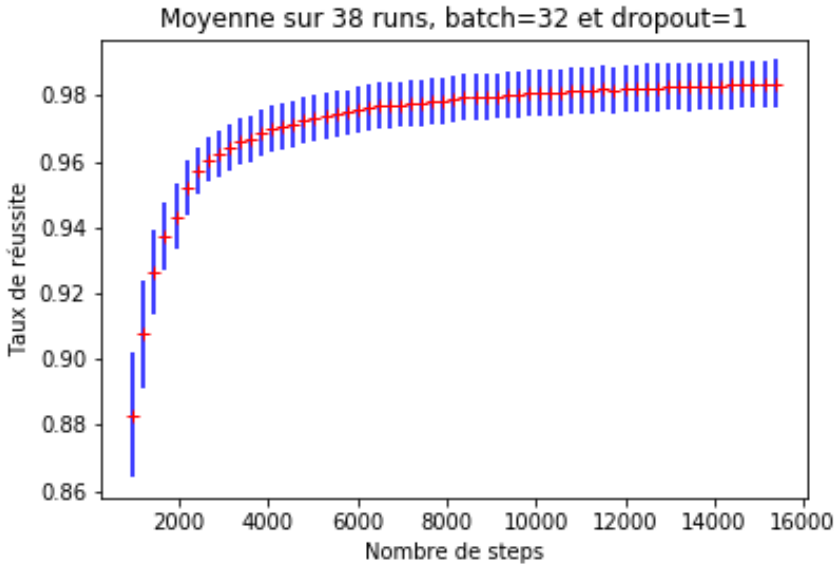


Figure 30: Taux de réussite en fonction du nombre d'apprentissages pour notre CNN sous TensorFlow

Le post-traitement pourraient être utilisé pour déterminer les effets de la taille des batchs ou encore du dropout. Se pose alors la question de la nature de l'abscisse. Est-il préférable d'utiliser le nombre d'apprentissages ou le temps ? Nous avons choisi de ne pas traiter en profondeur ces questions pour nous concentrer sur le cœur de notre sujet, le Q-Learning. Cette question reste cependant une piste qui pourrait être explorée dans un futur projet.

3 Le Q-Learning

3.1 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 :

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{35}$$

Dans l'équation 35, 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 établit 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.

3.2 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 35, $r + \gamma \max_{a'} Q(s',a')$ 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 :

$$\Delta Q(s,a) = \lambda \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{36}$$

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 35, ou simplement r si la partie se termine) puis on applique le changement décrit équation 36.

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.

3.3 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.

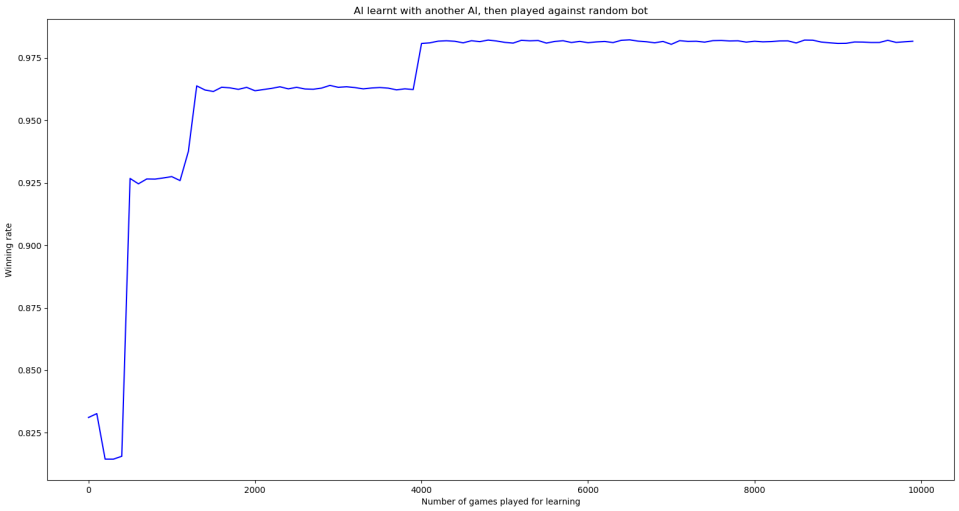


Figure 31: Apprentissage contre une autre IA apprenant en même temps

La figure 31 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).

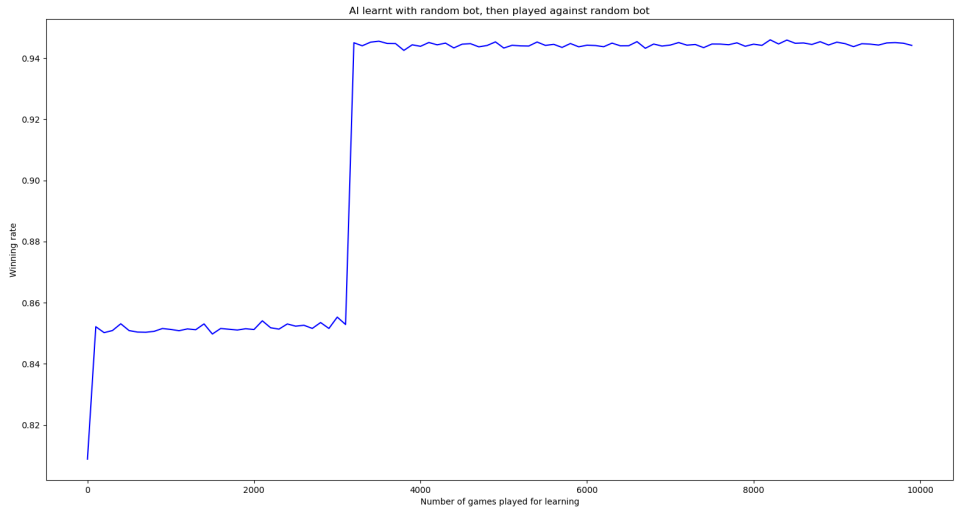


Figure 32: Apprentissage contre un joueur jouant aléatoirement

La figure 32 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.

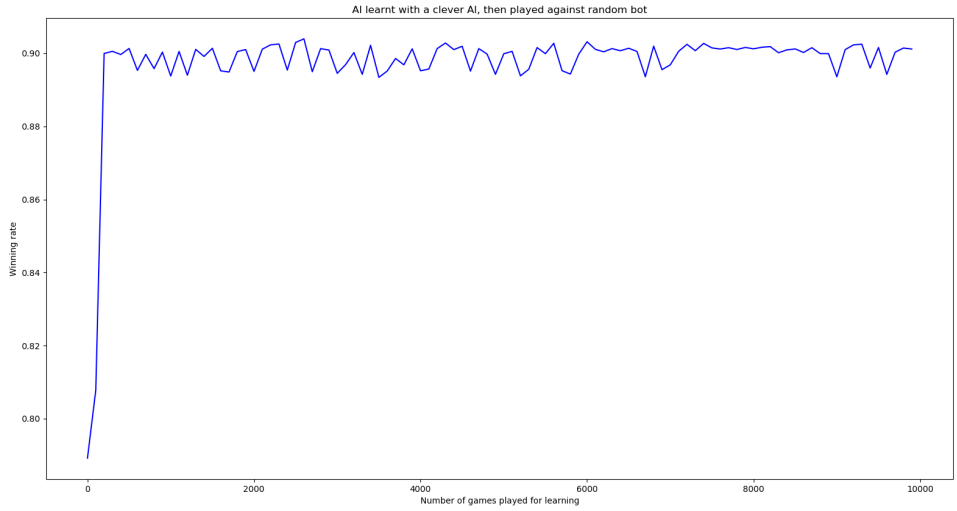


Figure 33: Apprentissage contre un joueur appliquant la stratégie gagnante

Enfin, la figure 33 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 important.

3.4 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.

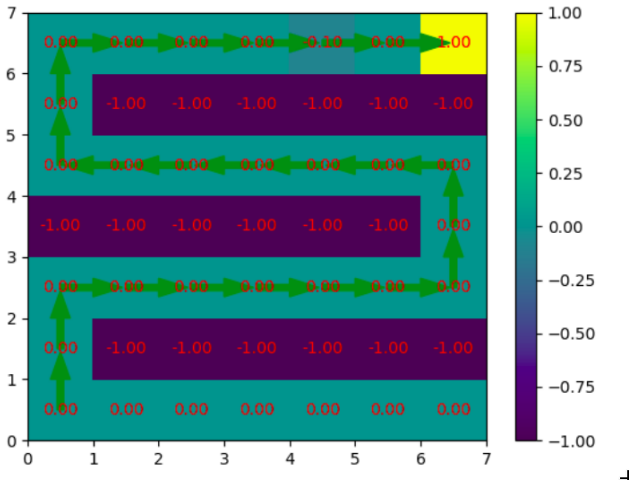


Figure 34: Chemin choisi par l'IA lorsque confrontée à un labyrinthe donné

La figure 34 représente le chemin choisi par l'IA lorsque celle-ci est confrontée à un labyrinthe particulier, en serpentin. On remarque que 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.

4 Le Deep Q-Learning

4.1 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 action, de sorte qu'une seule évaluation du réseau suffise à choisir l'action à effectuer.

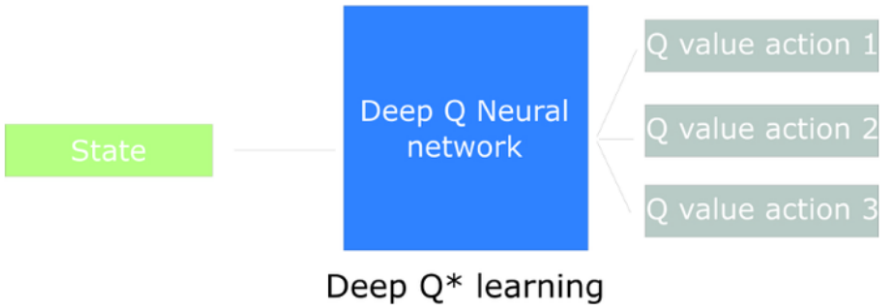


Figure 35: Schéma du Deep Q-Network

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 35 pour l'expression voulue de la fonction de satisfaction.

Pour faciliter la réalisation de notre algorithme et nous débarrasser 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.

4.2 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 précédemment. 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 35. 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 36: Fonction de satisfaction Q approximée par le réseau neuronal

Le tableau 36 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.

4.3 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.

4.3.1 Deux Deep Q-Networks

Lorsque nous réalisons la rétropropagation, nous devons connaître l'objectif de notre réseau. Celui-ci est donné par l'équation 35. 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 35. 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.

4.3.2 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.

4.3.3 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.

4.3.4 Apprentissage

Nous retenons un échantillon d'un million de transitions sur lesquelles nous effectuons notre apprentissage. Lorsque nous dépassons ce quota, 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.

4.3.5 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. Toutefois, 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).

4.4 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].

4.4.1 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, aux environs de 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%).

4.4.2 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.

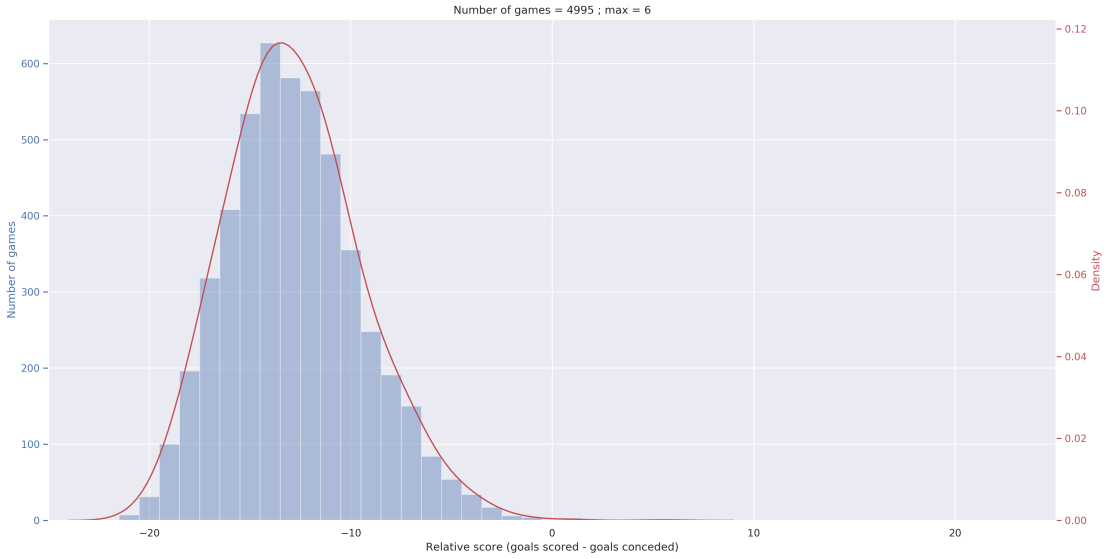


Figure 37: Entraînement du Deep Q-Network sur 7000 parties sans Max pooling

La figure 37 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 meilleurs résultats après 7000 parties.

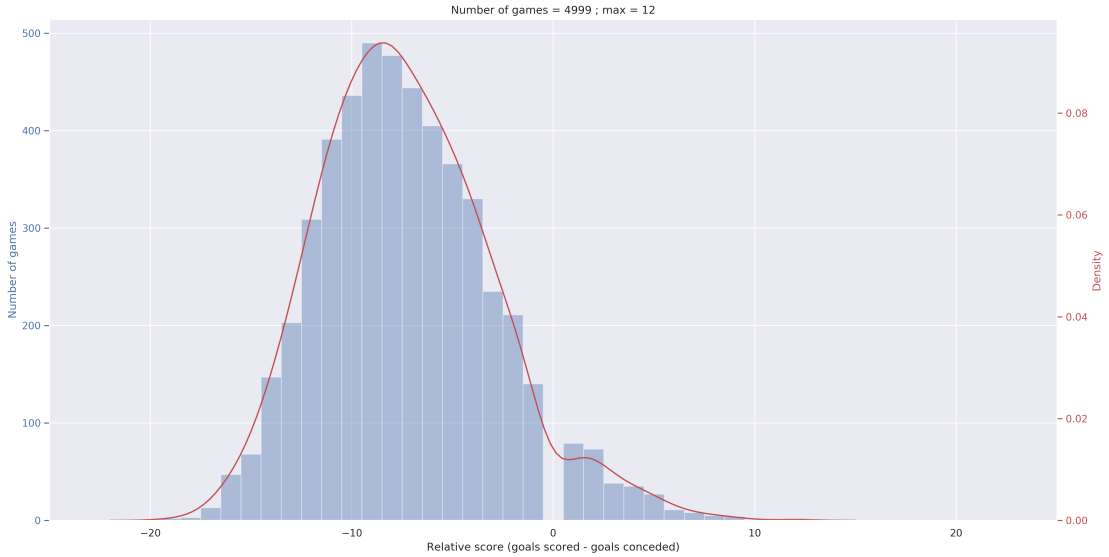


Figure 38: Entraînement du Deep Q-Network sur 7000 parties avec Max pooling

Nos attentes sont confirmées par la figure 38. 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 37. 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 voulons étudier plus en profondeur l'apprentissage. Cependant, il nous est difficile de tester différentes configurations pour mieux comprendre comment fonctionne le Deep Q-Learning car un apprentissage sur Pong peut être très long. Utiliser le Deep Q-Learning pour jouer à un jeu complexe comme Pong nécessite énormément de ressources. Nous avons pu le voir avec notre implémentation sous TensorFlow, mais également dans de nombreuses implémentations disponibles sur GitHub. Nous avons même testé l'implémentation fournie par DeepMind que nous avons dû faire tourner pendant plusieurs dizaines d'heures pour commencer à gagner. Nous avons donc décidé de nous concentrer sur d'autres jeux que Pong.

4.5 CartPole

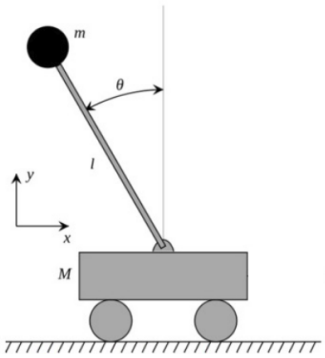
4.5.1 Principe du jeu

CartPole est un jeu disponible sur gym, représenté figure 39a. Le principe est le suivant : on dispose d'un pendule simple fixé à sa base à un chariot. L'objectif est de stabiliser le pendule en équilibre instable en déplaçant ledit chariot. Étant donné qu'apprendre un jeu à partir de photographies de l'écran est une contrainte très forte qui ralentit énormément l'apprentissage, nous préférons obtenir un espace des états plus petits pour pouvoir faire les tests voulus. CartPole nous donne un état à 4 dimensions (Figure 39b) composé de l'angle et la vitesse angulaire du pendule, et de la position et la vitesse du chariot. Nous disposons de deux actions : aller à gauche ou à droite. Ne rien faire ne fait pas partie des actions disponibles.

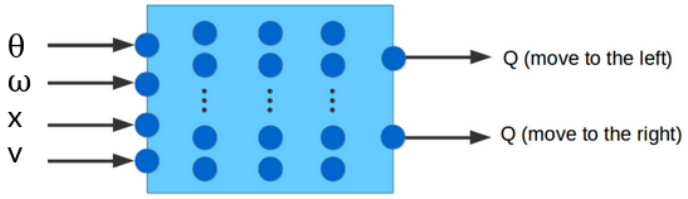
CartPole est un jeu de survie. Pour survivre à une étape, le pendule ne doit pas être trop incliné, et le chariot ne doit pas être trop éloigné de sa position initiale. Si notre IA survit 500 étapes, alors le jeu est réinitialisé. Cela permet de la remettre en difficulté dans le cas où l'IA a trouvé un point stable. On considère que l'IA a réussi à jouer à CartPole si elle est capable de survivre 195 étapes. Cette limite est arbitraire mais souvent utilisée

sur Internet pour CartPole. Reprendre cette même convention nous permet de nous situer par rapport à autres réalisations que l'on peut trouver sur GitHub.

Pour résoudre ce jeu, il convient de trouver une fonction de satisfaction adéquate. Si l'on meurt, on reçoit bien évidemment une récompense négative, que nous avons situé à -1. Nous voulons encourager notre IA à survivre, c'est pourquoi nous lui offrons une satisfaction pour le simple fait de rester en vie. Nous avons décidé de donner une récompense de 1 à chaque étape survécue.



(a) Schéma du jeu CartPole



(b) Espace des états sur CartPole

Figure 39: Représentation du jeu CartPole

4.5.2 Apprentissage de CartPole

Nous réalisons l'apprentissage sur CartPole. Le réseau utilisé est constitué de deux couches intermédiaires, toutes deux composées de 24 neurones. Les hyperparamètres utilisés sont détaillés sur la figure 40.

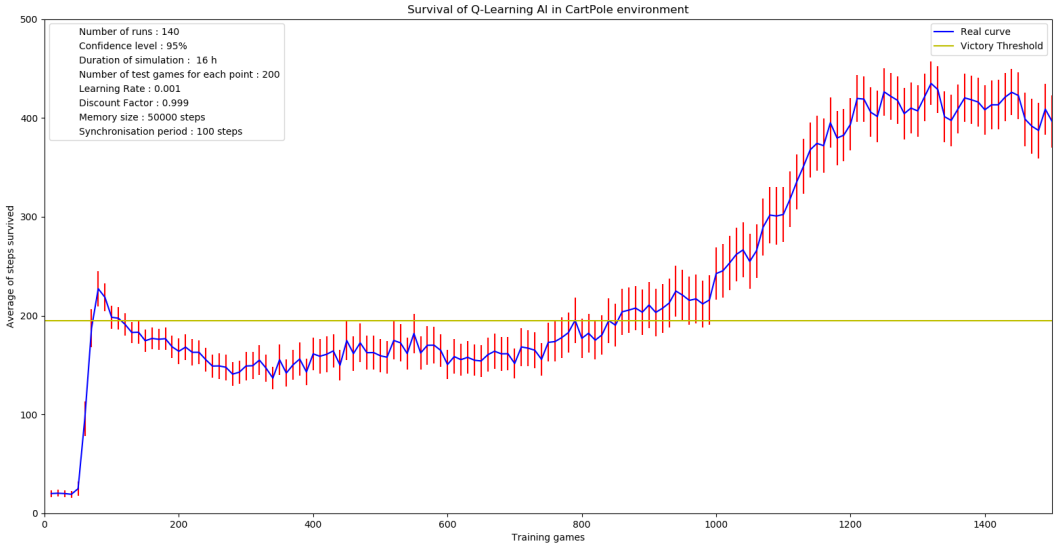


Figure 40: Apprentissage de CartPole

On remarque que la barre des 195 étapes survécues a été franchi, ce qui signifie que l'on peut considérer que notre IA a réussi à apprendre CartPole. Afin de pouvoir nous situer par rapport aux réalisations d'autres personnes sur GitHub, nous superposons notre courbe à celle obtenue avec un code disponible sur GitHub (figure 41).

Le code public dépasse brièvement (et dans le meilleur des cas) la barre des 195 étapes survécues mais ne maintient pas ce résultat. À l'instar de beaucoup d'articles scientifiques, seul le meilleur des cas permet de satisfaire les conditions de réussite, l'expérience étant arrêtée dès que le jeu est résolu. En ce qui nous concerne,

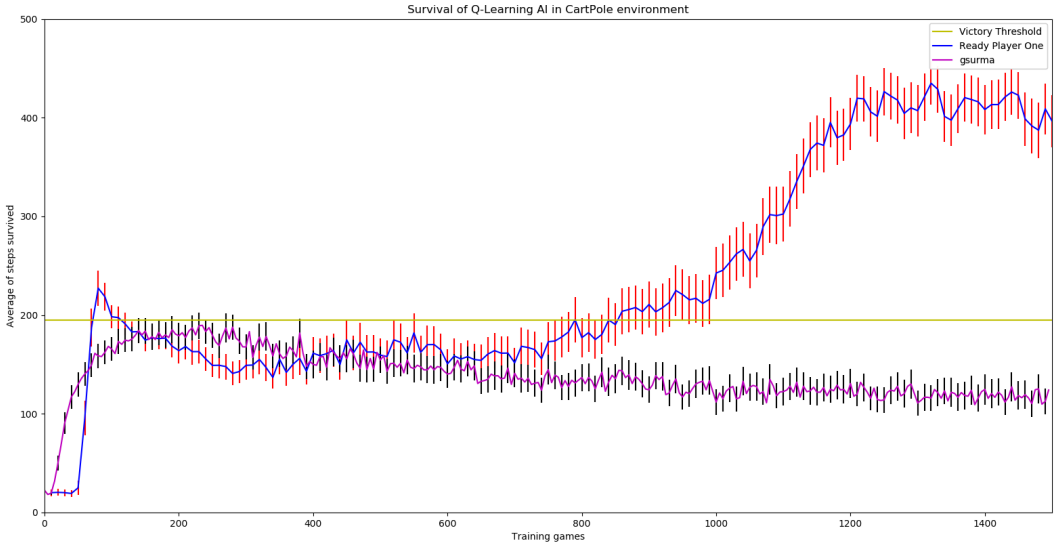


Figure 41: Comparaison des performances avec d'autres codes publics

nous continuons l'expérience, ce qui nous permet de conclure sur la stabilité de notre algorithme. En outre, nous moyennons nos résultats (en l'occurrence sur 140 réalisations) pour résoudre le jeu en moyenne et pas dans le meilleur des cas.

Sur GitHub, on a disposé deux vidéos après victoire de notre IA. La première montre un pendule qui survit mais demeure relativement chancelant. En poursuivant l'apprentissage, on obtient la seconde vidéo. On remarque alors que notre IA trouve assez rapidement un point d'équilibre du pendule.

4.5.3 Apprentissage avec un seul réseau

Dans la section 4.3.1, nous avions expliqué que deux réseaux aidaient à stabiliser l'apprentissage, et qu'un seul réseau pouvait parfois mener à des oscillations voire une instabilité de notre IA. Pour confirmer cela, nous avons voulu tester un apprentissage avec un seul réseau sur CartPole. Le résultat est représenté figure 42.

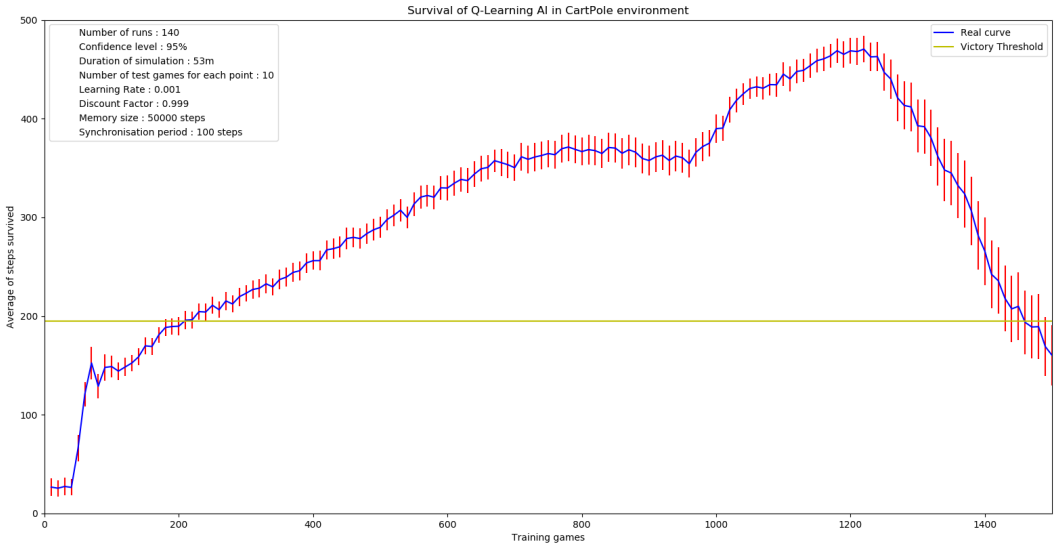


Figure 42: Apprentissage de CartPole avec un seul réseau

Nous remarquons une importante chute du temps de survie à la fin de l'apprentissage. Cela marque donc la forte instabilité de notre IA induite par le fait que le réseau est unique. On remarque toutefois que le jeu est résolu et que l'on atteint des scores très élevées. Cela provient du fait que CartPole est un jeu suffisamment simple pour qu'un seul réseau ne soit pas trop handicapant. Il est néanmoins suffisamment complexe pour faire apparaître l'intérêt d'un second réseau.

Puisqu'un apprentissage sur CartPole est relativement rapide, nous avons pu tester différentes configurations, et ainsi avoir une meilleure intuition des hyperparamètres à choisir. Nous pouvons donc mettre cela en application avec un jeu un peu plus complexe : Flappy Bird.

4.6 Flappy Bird

4.6.1 Principe

Pour pouvoir manipuler la difficulté du jeu comme on le souhaite et pour avoir accès à l'espace des états que l'on veut, on a recréé le jeu Flappy Bird en C. Notre IA dispose de deux actions possibles : bondir ou ne rien faire (et donc se laisser tomber).

Nous avons défini l'espace des états comme représenté figure 43. Notre oiseau voit les données suivantes : la distance horizontale qui le sépare du prochain tuyau, les distances verticales qui le séparent du haut et du bas du passage, ainsi que du haut et du bas de l'écran, et sa vitesse verticale. Toutes ces données ont été normalisées en prenant en compte les dimensions caractéristiques du jeu.

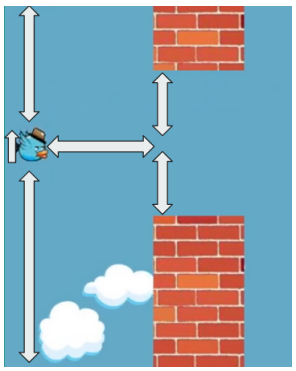


Figure 43: Illustration de l'espace des états de Flappy Bird

Nous devons également définir une fonction de récompense. Si l'oiseau meurt (sort de l'écran ou entre en collision avec un mur), la récompense est égale à -1. Si l'on passe une porte, notre score est incrémenté et la récompense est égale à 1. Toutefois, pour que l'oiseau soit capable d'apprendre à passer les portes, il doit tout d'abord réussir à en passer quelques unes par hasard afin qu'il comprenne que cette action rapporte des récompenses. Or, il est assez difficile d'avoir un score non-nul à Flappy Bird lorsqu'on joue aléatoirement. Pour encourager notre IA à vouloir rester en vie, on ajoute une composante survie à notre fonction de récompense. Ainsi, à chaque étape survécue, la récompense est de 0.1.

4.6.2 Apprentissage

On réalise l'apprentissage du Flappy Bird que nous avons réalisé. Le réseau utilisé est constitué d'une couche intermédiaire de 16 neurones. Les hyperparamètres utilisés sont les suivants : taux d'apprentissage à 0.001, facteur d'actualisation à 0.99, synchronisation des deux réseaux toutes les 500 étapes et décroissance exponentielle du taux d'exploration.

On remarque que l'apprentissage est un succès. On obtient une IA capable de survivre à l'infini. On remarque également une certaine instabilité du réseau qui disparaît et ré-apparaît régulièrement à mesure que l'on poursuit l'apprentissage.

Ce résultat étant arrivé en fin de projet, nous n'avons pu tracer de courbes permettant de mieux cerner comme s'est déroulé l'apprentissage. Toutefois, une vidéo de l'apprentissage est disponible sur GitHub.

Conclusion

Plusieurs objectifs ont été remplis lors de ce projet. Tout d'abord nous avons pu acquérir les bases sur les réseaux neuronaux. L'étude de la documentation scientifique nous a permis de comprendre précisément comment fonctionne un perceptron et d'en coder un nous même en Python. Nous avons été confrontés aux questions telles que le choix de la méthode de descente du gradient, de la fonction d'erreur ou encore de la taille des batchs. Nous avons ainsi pu apprendre à calibrer notre perceptron sur des problèmes simples comme la fonction XOR avant de s'attaquer à des problèmes plus complexes tel MNIST.

Ce problème lié à la reconnaissance d'images nous a naturellement conduit à étudier les réseaux de neurones à convolution. Encore une fois, l'étude de problèmes simples tels que la reconnaissance d'une croix nous ont permis d'appréhender facilement les différents paramètres des réseaux convolutifs. Le choix a alors été fait de ne pas coder nous même le réseau convolutif mais d'utiliser la bibliothèque Python TensorFlow. Le réseau développé en TensorFlow est en effet plus efficace qu'un réseau codé à la main. TensorFLow nous a également permis de gagner le temps que nous aurions passé sur l'implémentation du réseau. Notre projet était en effet d'une durée d'un an, et nous devions nous concentrer sur le cœur du sujet, le Q-Learning.

Nous avons pu nous familiariser sur le Q-Learning en résolvant des problèmes simples, tel que le jeu des bâtonnets ou le jeu du labyrinthe. Notre but final étant de jouer au jeu Pong, nous ne pouvions nous contenter du Q-Learning. Nous avons dû utiliser le Deep Q-Learning car le jeu est beaucoup trop complexe. Nous avons alors fait appel à nos connaissances sur les réseaux convolutifs pour remplacer la Q-table utilisée dans le Q-Learning. Nous avons alors été confronté à de nombreux problèmes pour résoudre le jeu Pong : utilisation de deux réseaux, choix de l'optimiseur, configuration du réseau, utilisation de la mémoire, etc. Le plus grand problème a cependant été les capacités de calcul pour que le réseau apprenne à résoudre le jeu. Nous avons réussi à gagner des parties mais il nous est impossible de réaliser des centaines de runs pour tester différents paramétrages du réseau. Nous avons donc décidé de nous replier sur un jeu plus simple donc moins gourmand en ressources. Nous avons choisi le jeu CartPole sur lequel nous avons étudié l'évolution de taux de succès en fonction du taux d'entraînement. Notre algorithme était stable et est arrivé à résoudre le jeu dans un temps raisonnable. Nous avons enfin utilisé notre algorithme pour résoudre avec succès le jeu Flappy Bird.

Ce projet nous a montré que le Deep Q-Learning permettait de résoudre des problèmes complexes comme le jeu Pong mais qu'il nécessitait d'énormes puissances de calcul. Par ailleurs, le processus d'apprentissage reste assez opaque et parfois instable.