READY-PLAYER-ONE

Rapport du groupe Harpon

Martin Lehoux, Pierre Biret
Sacha Seksik, Loïc Audoin

18 juin 2019

ABSTRACT

Le projet "Ready player one", constitué de deux équipes (Harpon et Eponge) a pour but l'apprentissage par un réseau de neurones d'une stratégie gagnante au jeu "Pong" sur Atari.

L'interêt du projet étant la compréhension des mécanismes des réseaux de neurones par l'ensemble des étudiants, ce projet sera donc constitué de plusieurs parties qui mèneront à terme à la réalisation du projet.

1 RÉALISATION DU PERCEPTRON

1.1 Premières approches

la première étape de notre projet conserne la création d'un perceptron (ou MLP). pour cela, nous avons d'abord eu une approche théorique, afin de comprendre ce qu'était un réseau de neurones au niveau mathématique et en quoi il pouvait présenter une solution pour résoudre numériquement un problème de classification dans un espace à n dimensions. Pour celà, nous nous sommes évidemment appuyés sur les travaux fondateur de Yann LeCun, et ceux-ci nous serviront régulièrement par la suite.

L'important a été de comprendre en quoi la minimisation d'une fonction d'erreur était faisable en temps fini. Pour celà on s'interesse à la fonction associant un vecteur d'entrée et les paramètres (Poids, Biais) liés à chaque neurone à un vecteur de sortie du réseaux de neurones. Cette opération complexe est réalisées par applications successives des fonctions d'activations liées à chaque couche de neurones aux résultats (pondérés par des poids et biais) de la couche précédentes. Le calcul de cette fonction est noté dans ce futur rapport comme frontpropagation ou propagation directe.

La fonction d'erreur mentionnée plus haut représentant la distance entre les sorties de la frontpropagations pour un vecteur d'entrées et les sorties voulues pour cette entrée, le principe du réseau de neurone va être de minimiser cette erreur en utilisant sa dérivée par rapports au différents paramètres du réseau. On remarque rapidement que le calcul d'une dérivée par rapport à un poids ou biais associé à un neurone d'une couche donnée va faire intervenir toutes les dérivées par rapports aux couches suivantes. le calcul de la déroulée va donc se faire dans le sens indirect, d'où l'usage du terme backpropagation.

De plus, il est utile de comprendre que les fonction d'activation choisies auront la propriété utile de permettre en temps réduit le calcul de leur dérivée en fonction de leur résultat : on a pour tout x, $f'(x) = G(f(x))$, ce qui va permettre, en retenant les résultats de chaque neurone pour la propagation directe, d'effectuer rapidement la propagation indirecte.

Enfin, le processus de minimisation se fait par des méthodes basés sur la méthode du gradient vectoriel :

$$Sys(n+1) = Sys(n) - Erreur[Sys(n)] \times Jacobienne(Erreur[Sys(n)])^{-1}$$

L'algorithme donc dépendre, en plus du choix de la fonction de coût et de la fonction d'activation des neurones, du choix de la méthode de minimisation et de ses paramètres associés (learning rate).

Avant de commencer la programmation d'un perceptron, il a fallu commencer par comprendre le fonctionnement opération par opération d'un neurone. Une feuille de calcul a donc été mise en place pour calculer à la main une itération de l'apprentissage d'un réseau de neurones de la fonction XOR à deux entrées avec rétropropagation, après visualisation par tous les étudiants du groupe d'une vidéo très bien réalisée sur le fonctionnement des réseaux neuronaux.

La mise en place de cette feuille de calcul a mis en évidence la complexité des opérations effectuées par les réseaux de neurones, ainsi que l'impossibilité d'effectuer tous les calculs à la main. Toutefois ce premier exemple de réseaux a permis à l'ensemble de l'équipe de comprendre comment se comportait un réseau de neurones à plusieurs couches, et leur a permis de se lancer dans le vif du sujet.

valeur des entrees	0		valeurs des poids	-0,1	0,3	0		0,5
valeur des entrees	0		valeurs des poids	-0,8	0,4	0		0,5
			valeurs des poids	0,7	-0,2	0		0,5621765
correspondance	x0		correspondance	a000	a010	b00		sigmoide 0
correspondance	x1		correspondance	a100	a110	b10		sigmoide 1
			correspondance	a01	a11	b1		Yexp
poids:	initialisation	derive	nouveau					
a000	-0,1	0	-0,1		cout0	0,5621765		
a010	0,3	0	0,3		cout1	0,432782		
b00	0	0,0484298	0,048429779		cout2	0,4298723		
a100	-0,8	0	-0,8		cout3	0,5755719		
a110	0,4	0	0,4		cout tot	0,254854		
b10	0	-0,0138371	-0,01383708					
a01	0,7	0,1383708	0,838370797					
a11	-0,2	0,1383708	-0,061629203					
b1	0	0,2767416	0,276741595					

Figure 1 – *modélisation d'un apprentissage sur une entrée pour un réseau 2-2-1 sur Excel*

L'étape suivante fut donc la programmation en Python d'un réseau de neurones, celle-ci reposant sur la dualité entre parcours du réseau direct pour les calculs et inverse pour l'apprentissage (frontpropagation et backpropagation). La première difficulté rencontrée lors de la programmation a été la lenteur de nos algorithmes. En effet, des calculs qui auraient pu être effectués matriciellement (multiplication d'une couche par une matrice de poids et ajout de la matrice de biais pour obtenir l'entrée de la couche de neurones suivantes) étaient réalisés ligne par ligne à travers des listes. Le module numpy a donc été utilisé pour optimiser les calculs matriciels, améliorant la vitesse de calcul, comme on le voit sur la première courbe. L'utilisation de numpy, non essentielle pour nos premiers perceptrons approximants la fonctions XOR, fut nécessaire à l'apprentissage de réseaux plus grands, en particulier dans notre application suivante à la base de données MNIST pour lesquels les calculs passèrent de l'ordre de la minute à l'heure.

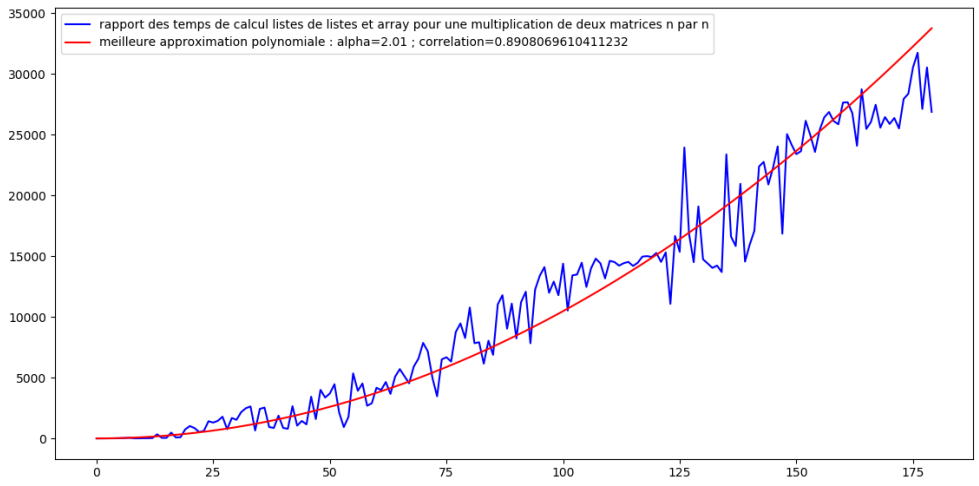


Figure 2 – *rapports entre le temps de calcul d'une multiplication matricielle avec numpy et en itérant selon la taille* n *de la matrice. la vitesse est multipliée par* n^2

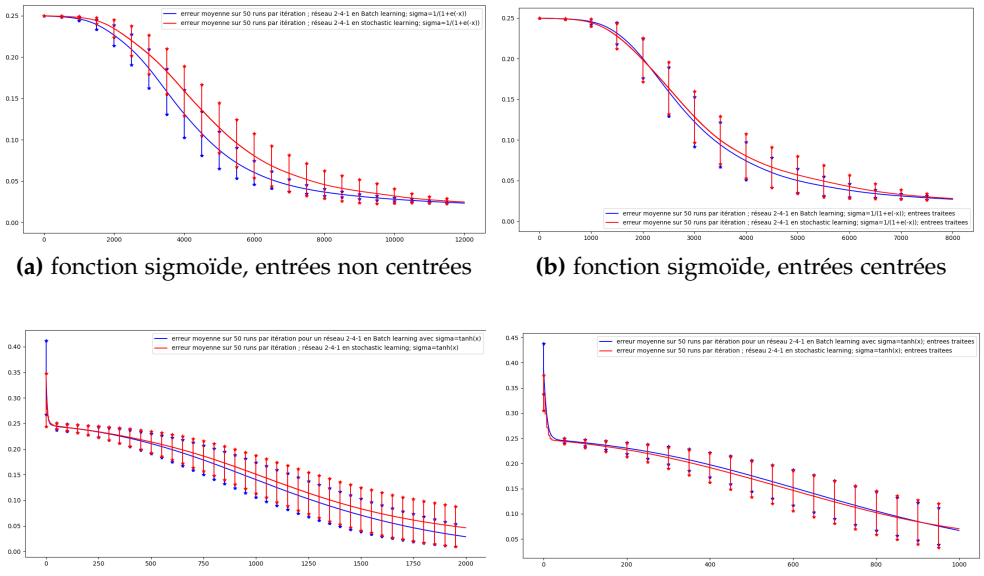
Chaque équipe a donc tenté lors de cette étape de coder sa version d'un réseau d'apprentissage du XOR, en utilisant pour commencer la fonction d'activation sigmoïde et un apprentissage stochasticque. Malgré la simplicité apparente de cette tâche, les résultats ont divergé, les réseaux n'arrivant pas à classifier correctement tous les cas du XOR s'il avait trop peu de neurones intermédiaires.

Le premier constat fut donc le suivant : L'assurance de la convergence de l'erreur vers 0 ne se fait que lorsque la couche intermédiaire du réseau a au moins 4 neurones.

Ces résultats nous permettent de vérifier pour ce problème le théorème selon lequel tout problème de classification peut être traité efficacement par un perceptron n'ayant qu'une couche intermédiaire si cette couche est assez grande. la nécessité d'avoir une couche intermédiaire vient pour le cas du XOR, de la non linéarité du problème.

Avant de s'atteler à une application plus complexe, nous avons pris du temps pour tester l'influence de différents paramètres sur la vitesse de convegernce de notre perceptron vers la fonction XOR. Nous avons isolé plusieurs hyperparamètres, à savoir la facon d'initialiser les poids et biais, la vitesse d'apprentissage (ou plus généralement la fonction de minimisation qui pourra ne pas être un gradient simple), la façon de gérer notre set d'entraînement (batch training, stochastich training), la fonction d'activation choisie, ou encore la forme du réseau. Une singularité rencontrée a été la valeur du learning rate, qui devait être 100 fois plus élevé que ceux de la littérature, ou même que celui de l'autre groupe, pour avoir la même vitesse de convergence. Nous nous sommes penchés plusieurs semaines sur ce détail (l'algorithme fonctionnait très bien à côté), qui devait surement être du à une erreur de programmation, mais nous n'en avons pas trouvé la cause, et sommes passés à la suite du projet.

Ce réseau développé main, bien qu'initialement seulement utile pour la compréhension des problèmes, nous servira à la fin du projet afin de réaliser nos résultats les plus concluants.



(a) fonction sigmoïde, entrées non centrées

(b) fonction sigmoïde, entrées centrées

(c) fonction tanh, entrées non centrées

(d) fonction tanh, entrées centrées

Figure 3 – *comparaison du nombre d'itérations pour approximer la fonction XOR avec un réseau 2-4-1 en apprentissage Batch et stochastique selon différents paramètres*

Nos conclusions nous ont menés, dans la suite de l'étude, à utiliser la fonction tangente hyperbolique plutot qu'une sigmoïde, à centraliser et réduire les entrées, et à choisir des poids initialisés autour de zéro et des biais initialisés nuls. Nous utilisons même une tangente hyperbolique pondérée $f(x) = 1.7159 \times \tanh(\frac{2x}{3})$ qui permet d'améliorer les résultats, principalement en accélérant les calculs de propagation par rapport à $f(x) = \frac{1}{1+e^{-x}}$. Cette tangente hyperbolique pondérée a été proposée par Lecun.

1.2 Application au problèmme de classification de chiffres manuscrits

Après avoir vérifié que la théorie fonctionnait sur une fonction aussi simple que la fonction XOR, nous devions tester les performances de l'algorithme sur un sujet un peu plus complexe.

Le problème classique que l'on a choisi de résoudre avec un réseau de neurones est celui de la classification de chiffres manuscrits. Pour cela, on utilise la base de données MNIST, un dataset disponible en ligne comprenant des dizaines de milliers d'images de 28×28 pixels en nuances de gris, chacune représentant un chiffre entre 0 et 9, celui-ci étant fourni avec l'image. le perceptron que l'on va utiliser pour classifier l'image aura donc un vecteur d'entrée de taille 784, et un vecteur de sortie de taille 10. Dans la suite du rapport, les résultats seront donnés après un léger post-traitement : on choisira pour classifier un nombre, la coordonnée du vecteur de sortie étant maximale, et nos pourcentages de réussites seront fondés sur ce principe.

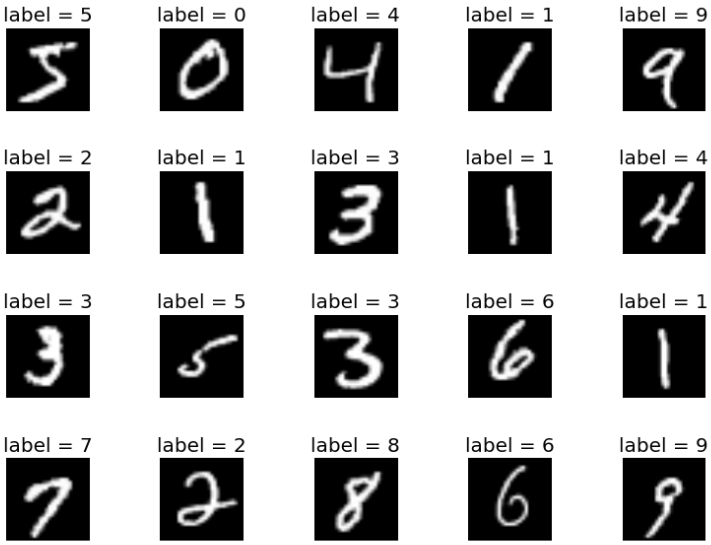


Figure 4 – *exemple d'images tirées de la base de données MNIST*

La suite consistait alors en la réalisation d'un réseau de neurones pouvant travailler sur la base de de données MNIST des chiffres manuscrits, et sachant à terme reconnaître les chiffres. Le premier souci en terme de travail d'équipe a été rencontré à ce moment-là : deux membres du groupe avaient chacun écrit une version de l'algorithme d'apprentissage, et après discussion tendues, les deux membres ont travaillé ensemble à la réalisation du perceptron. La nécessité d'utiliser les outils de programmation collectives tels GitHub d'une bonne manière s'est fait ressentir.

Nos perceptrons ont rapidement réussi à résoudre ce problème de classification, mais en regardant plus précisement, les paramètres liés à nos premiers résultats différaient énormément des résultats théoriques, surtout en terme de vitesse de calcul et en taux d'apprentissage : alors qu'il devrait être de l'ordre de 10^{-4} pour obtenir une convergence précise et suffisante, nous pouvions monter jusqu'à 10^{-1} voire 1, et toujours converger assez lentement (voir figure 5). Ce problème restera dans notre code sans en trouver de réelle raison pendant environ 1 mois.

Quant à la vitesse de calcul, elle était grandement affectée par le premier choix que nous avions fait de conserver les formules itératives des calculs de valeurs de neurones. L'équipe Eponge, en parallèle, avait un temps de calcul de l'ordre de la minute là où nous avions un temps de calcul de l'ordre de l'heure. Le passage de la forme itérative à la forme matricielle, pour le calcul des valeurs des réseaux de

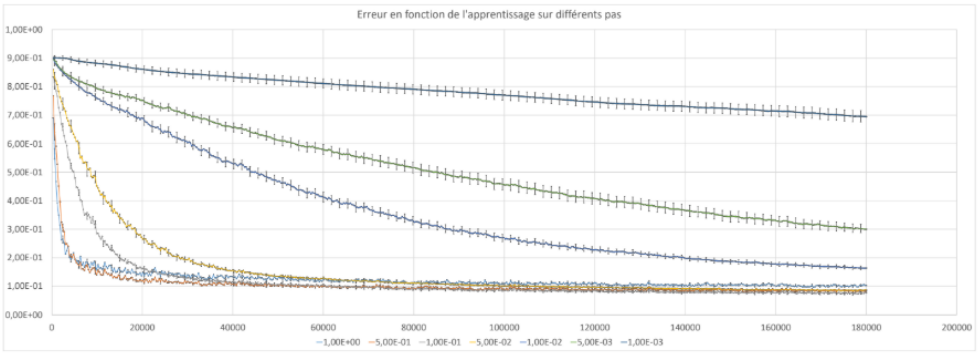


Figure 5 – *influence du learning rate sur la vitesse de convergence (en nombre d'itérations de l'entraînement)*

neurones optimisa grandement le temps de calcul, et est une preuve de l'incroyable efficacité des modules de calcul matriciel de Python.

Les valeurs des erreurs finales restant malgré tout satisfaisantes et cohérentes, nous avons donc décidé de travailler sur d'autres facteurs, tels que l'initialisation des paramètres du réseau, et la taille des batch (réduits à 1 élément dans le cas du stochastic training).

Les travaux de LeCun affirmaient que l'initialisation des valeurs des poids et des biais des neurones étaient à rendre aléatoires selon une loi normale centrée. Nous avons donc voulu tester différentes manières d'initialiser ces paramètres, ce qui nous a permis de rapidement exclure l'idée d'initialiser tous les poids à 0, comme on le voit sur la figure 6.

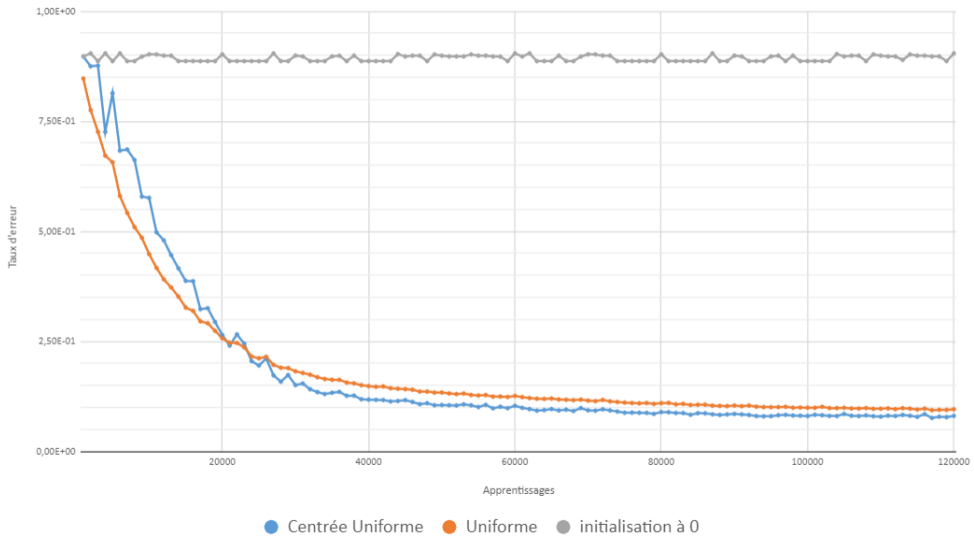


Figure 6 – *influence de l'initialisation des poids sur la vitesse de convergence (en nombre d'itérations de l'entraînement)*

Nous avons pour la suite du projet travaillé en "batch-training". Cette méthode d'apprentissage signifie que nous minimisons la fonction de coût selon la moyenne de son expression sur plusieurs entrées du set d'apprentissage, regroupées dans un batch.

La taille du batch, toutefois, était une question différente : des batchs plus petits, soit un modèle tendant vers un descente stochastique de gradient, permet d'augmenter la vitesse de l'apprentissage, mais le choix d'apprendre sur des batchs plus grands permet une plus grande robustesse à d'éventuels mauvaises données d'en-

traînement et donc de minimiser la fonction à chaque étape dans une direction plus proche de celle du minimum.

La conclusion qui nous est venue est la suivante : le meilleur compromis entre stabilité et précision de l'erreur finale est obtenu pour des batch de taille entre 16 et 32. Ces résultats sont appuyés par plusieurs travaux tiers cités par leCun.

Nous avons ensuite rencontré un second problème face au choix d'une nouvelle vitesse d'apprentissage (rappelons que ce paramètre externe au réseau représente la taille du pas vers la minimisation de la fonction de coût que nous effectuons à chaque itération de l'entraînement)

Nous avions, depuis le début, utilisé la fonction d'activation sigmoïde basique $f(x) = \frac{1}{1+e^{-x}}$ dans chaque neurone de notre réseau, et donc utilisé des vitesses appropriées pour cette fonction. Néanmoins, au vu de ses meilleurs résultats sur XOR, nous avons voulu changer vers un réseau de neurones utilisant la fonction d'activation tangente hyperbolique pondérée, $f(x) = 1.7159 \times \tanh(\frac{2x}{3})$.

En gardant un learning rate du même ordre, cette transition nous semblait au premier abord inefficace, voire semblait empirer les résultats. En effet, l'erreur de la reconnaissance des chiffres de la base MNIST ne convergeait même pas lors de nos tests avec la nouvelle fonction d'activation.

En réalité, l'extrême efficacité de cette fonction, suggérée encore une fois par LeCun, associée à nos taux d'apprentissage bien trop élevés, empêchaient la convergence du réseau de neurones : ainsi, en utilisant une descente de gradient simple, il existe un learning rate théorique maximal au delà duquel il est impossible de continuer de converger une fois proche d'un minimum de la fonction. Quelques jours plus tard, le groupe se rendit compte que la même fonction d'activation, avec des taux extrêmement bas, convergeaient à une vitesse fulgurante, voire trop rapide.

Une des solutions évoquées à cette problématique est de ne pas utiliser une descente de gradient simple avec un learning rate constant mais plutôt diverses méthodes plus complexes (RMSprop, Adam, une descente avec inertie...) qui modifient la vitesse d'apprentissage en fonction de la vitesse d'apprentissage aux étapes précédentes afin d'accélérer autour des zones plates de la fonction et de réduire la vitesse autour des minima.

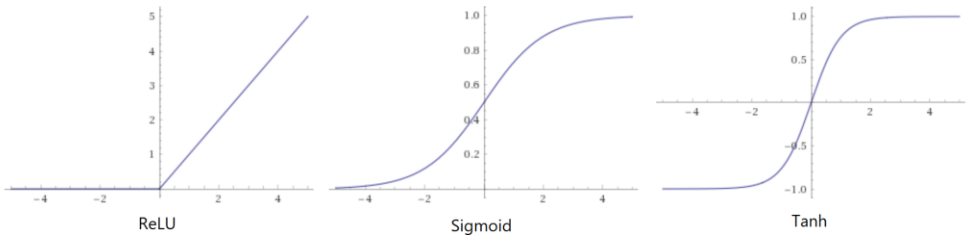


Figure 7 – *différentes fonctions d'activation pouvant être utilisées dans des réseaux de neurones*

Enfin, ces fonctions d'activation comportant une zone à pente forte (linéaire) autour de l'origine et une zone à pente faible (saturée) plus l'entrée augmente en valeur absolue, nous avons rapidement compris, pour augmenter la vitesse de l'apprentissage, l'intérêt de faire en sorte que l'entrée des neurones soit proche de 0.

Pour cela, nous avons associé à des poids initialement centrés et faibles des entrées globales centrées et réduites entre -1 et 1, ce qui a ouvert la voie à la possibilité de réaliser un prétraitement sur les entrées du réseau de neurones plus tard. Un prétraitement plus complexe sera utilisé dans la suite.

Cette nouvelle efficacité nous a permis de déterminer une toute dernière caractéristique de la base MNIST : sa taille et sa diversité sont si grandes qu'il nous a été impossible d'atteindre l'état de surapprentissage en s'entrainant sur la totalité du set d'entraînement.

Dans le même temps, nous avons rencontré une amélioration concrète de notre vitesse de calcul grâce à la machine virtuelle fournie par nos professeurs encadrants avec l'aide de l'école. L'un des membres de l'équipe disposait également d'une carte graphique NVIDIA 960M, mais au vu de la complexité d'utilisation du CUDA NVIDIA servant aux calculs mathématiques sur carte graphique, la décision a été prise de continuer sur la machine virtuelle.

Toutefois, étant légèrement en retard sur le planning de l'année, et sachant que nos algorithmes réalisaient à peu près ce qui était attendu d'un réseau de neurones, nous sommes passés à l'étape suivante du projet : l'apprentissage de la théorie des réseaux à convolution.

2 RÉSEAUX DE NEURONES À CONVOLUTIONS

Notre étude des réseaux neuronaux à convolution n'a été dans un premier temps que théorique : l'intérêt n'était pas d'aprendre à en coder un "avec les mains" mais surtout de se renseigner sur le fonctionnement et le rôle des différents paramètres pour pouvoir l'utiliser à bon escient pour résoudre des problèmes pour lesquels c'est un un outil adapté, en particulier l'analyse d'images. Ainsi, lors de notre utilisation future de réseaux à convolutions (ou CNN) fournis par des modules comme tenserflow, nous ne nous retrouvions pas à utiliser l'outil comme une boite noire.

Les réseaux de neurones à convolutions, comme nous allons le voir, sont souvent utilisés dans l'analyse d'image, car ils compensent le principal problème des perceptrons classiques dans le cas de vecteurs d'entrées de haute dimension : la lenteur de l'apprentissage. En effet, le nombre de poids et donc le nombre de dérivées à calculer va vite trop augmenter : par exemple, si l'on veut classer une image 300×300 selon un critère binaire et si n est la taille de la couche intermédiaire, on aurait plus de $90000n$ paramètres selon lesquels il faudrait calculer la dérivée partielle à chaque itération de l'entraînement, ce qui rendrait l'apprentissage impossible en temps fini.

Ce problème est résolu par les réseaux à convolutions. La particularité de ce réseau vient du fait qu'entre la couche d'entrée et la couche classique précédent la couche de sortie (appelée "fully-connected") vont venir plusieurs type de filtres :

— des couches de convolutions, qui vont traiter les données reçues avec des filtres fixes. le nombre de filtres sera la troisième dimension de la sortie.

— des couches de pooling, permettant de compresser l'information en sortie des couches de convolutions

— des couches de corrections, permetant de recentrer les données de sorties par des fonctions d'activations (on utilise souvent ReLu)

Les caractéristiques recherchées des CNN sont dues au fonctionnement des couches de convolutions : celles-ci vont déplacer une matrice filtre sur l'image d'entrée de manière à obtenir en sortie une image, un point de la matrice de sortie est le résultat de la convolution du filtre (souvent choisi 3×3 ou 5×5) par un carré de la matrice d'entrée de même taille. Ce type de filtrage lui confère les propriétés suivantes :

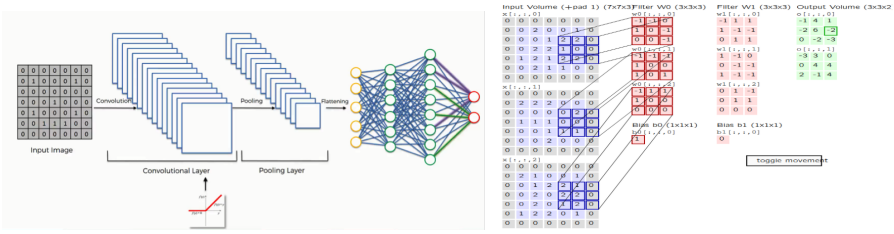
— Premièrement, chaque sortie du filtre ne va plus dépendre de toutes les entrées mais d'un groupes d'entrées proches dans la matrice d'entrée. On va donc introduire la notion de localisation, ce qui va imposer de transmettre le vecteur d'entrée sous une forme adaptée.

— Deuxièmement, on peut remarquer que l'association de la convolution avec le ReLU va faire en sorte que l'activation d'un neurone soit uniquement liée à la similarité du groupe de points de l'entrée auquel il correspond avec le filtre, et non de la position de l'entrée : le résultat est invariant par translation.

— Enfin, ces propriétés permettent aux CNN d'obtenir une meilleure résistance aux erreurs dans l'estimation des paramètres des filtres puisque, pour un training set fixé, la quantité de données par paramètres est plus grande que pour un MLP. Le partage de poids permet aussi de réduire considérablement le nombre de paramètres à minimiser. On diminue donc les besoins en mémoire et en quantité d'opérations.

La figure 8 montre les points sur lesquels un réseau à convolution diffère d'un réseau neuronal classique.

Le CNN va donc s'entraîner à adapter au mieux ses filtres pour pouvoir reconnaître au mieux les motifs (pattern detection) permettant la classification des entrées. Comme dans le cas du MLP, nous allons étudier l'influence de plusieurs paramètres sur la vitesse d'apprentissage.



(a) alternance des couches dans un CNN **(b)** une couche de convolutions

Figure 8 – *Visualisation du fonctionnement d'un réseau de neurones à convolutions*

2.1 Tensorflow

A cette période de l'année, nous avons eu l'accès à une machine virtuelle pour faire tourner nos programmes. Cela nous a permis d'effectuer des apprentissages plus longs que sur nos machines personnelles, ainsi que d'en faire tourner un plus grand nombre en parallèle. C'était donc le bon moment pour se pencher sur une bibliothèque python permettant de faire du machine learning de manière optimisée.

Nous avons donc, pour commencer, appliqué un tutoriel basique d'apprentissage sur MNIST, en CNN, fourni par tensorflow. Nous apprendrons plus tard que ce tutoriel utilisait des fonctions obsolètes, nous empêchant de travailler avec tensorboard, l'interface graphique de tensorflow. Toutefois, les résultats en terme d'efficacité de tensorflow étaient bien plus optimisés que nos réseaux neuronaux, et la grande diversité de paramètres nous a introduit à un concept qui ne nous avait pas intéressés jusque-là : le dropout.

Le dropout consiste à désactiver un certain pourcentage de neurones de façon aléatoire (p d'activation du neurone fixé en paramètre). Ce système d'apprentissage " partiel " permet en réalité aux neurones d'être moins dépendants des neurones précédents, et donc plus dépendants de l'entrée en elle-même, mais surtout permet d'éviter le surapprentissage. Toutefois, comme vu auparavant, la base MNIST de par sa taille et sa diversité empêche en elle-même le surapprentissage, ce concept sera donc à garder pour d'autres éventuels systèmes neuronaux. Un example de dropout est présenté dans la figure 9.

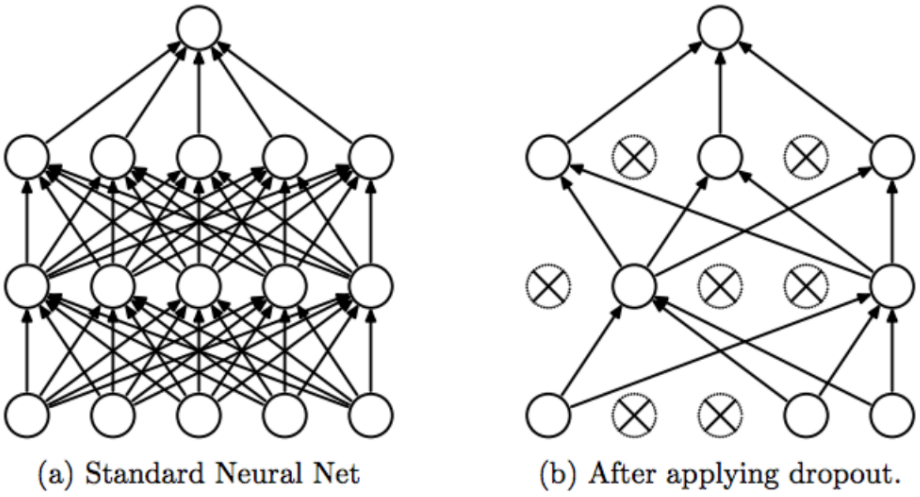


Figure 9 – *Description du dropout dans le cas simple d'un perceptron*

Nous avons eu l'occasion de discuter avec un ancien élève de Supélec lors de l'une des nombreuses réunions de projet, qui travaillait également avec tensorflow sur de l'analyse de signaux. Cet élève nous a gracieusement fourni un code tensorflow, utilisant des fonctions actuellement à jour, et permettant la visualisation en direct de

tensorboard et des différents graphiques liés à nos réseaux de neurones. Si la partie graphique de tensorboard est assez intuitive à prendre en main, la déconcertante complexité du code nécessaire à son bon fonctionnement a été une embûche à notre avancement.

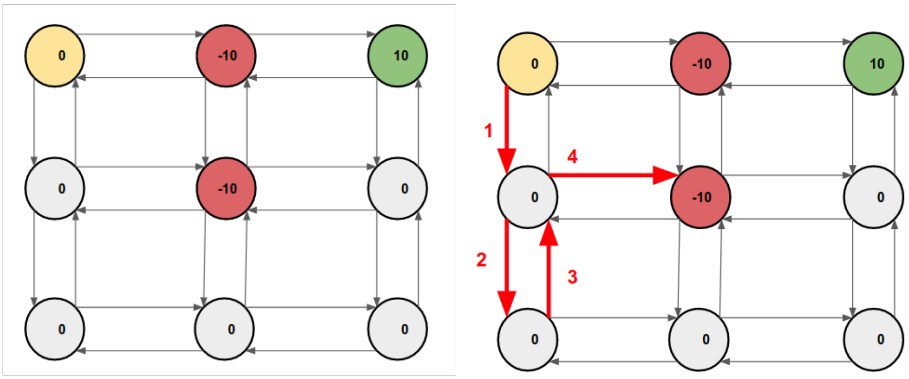
Une fois l'étape des CNN terminée, les profs encadrants nous ont donné pour but de travailler sur le Q-learning : une théorie de l'apprentissage par machine tout à fait différente des perceptrons, avec des cas d'applications tout aussi différents.

3 Q-LEARNING

L'objectif du Q-Learning est de calculer une fonction permettant au programme de choisir une action dans un ensemble d'actions $a \in A$ en présence d'un environnement ou état $s \in S$. Pour cela, on définit une fonction $Q : S \times A \to \mathbb{R}$ qui permet d'estimer la qualité du choix de a dans l'état s. Il existe une fonction $R : S \to \mathbb{R}$ donnant la récompense associée à un état s. Cette fonction, qui présente très peu de valeurs non nulles, est la seule manière d'avoir un retour sur la qualité des choix effectués. Elle annonce notamment les cas d'échec ou de succès pour le programme. Si l'espace d'état est de dimension assez faible, on peut estimer la fonction Q par itérations, en connaissant toutes les valeurs possibles. Cette fonction est donc définie par ce que l'on appelle une Q-table.

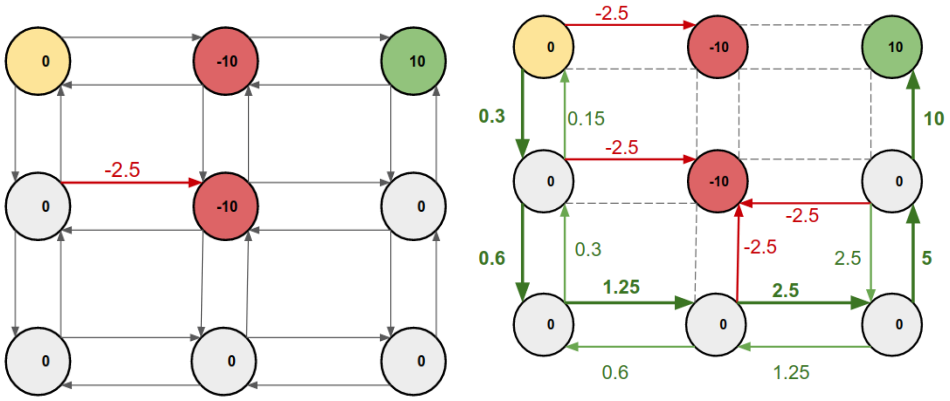
La Q-table enregistre tous les scores présumés de toutes les transitions (ou actions) entre les différents états. Après avoir effectué une action a et obtenu une récompense r, on peut mettre à jour la valeur que nous avions précédemment pour Q dans la Q-table, avec une valeur que nous estimons plus proche de la réalité et calculée par l'expression suivante :

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a'}Q(s',a'))$$



(a) état initial du plateau **(b)** exploration du plateau

Figure 10 – *Schéma décrivant le principe du Q learning*



(a) mise à jour des valeurs du tableau **(b)** exploitation après apprentissage

Figure 11 – *Schéma décrivant le principe du Q learning*

Comme on peut le voir dans la figure 10 (a), le plateau de jeu initial ne contient que des récompenses, et les valeurs de la table de qualité sont inconnues (elles

peuvent être initialisées à 0 par exemple). Lors de cette phase, l'algorithme peut être amené à explorer aléatoirement des actions, plutôt que de suivre la meilleure stratégie actuelle. En (b), l'algorithme explore le plateau de jeu en choisissant une action à partir de chaque état, et en prenant connaissance de la récompense associée. Après avoir fini de jouer une partie, l'algorithme va mettre à jour les valeurs dans la Q-table (figure 11 (a)) grâce à la formule vue précédemment. Enfin, après s'être assez entrainé, l'algorithme ne cherche plus à apprendre, mais seulement à utiliser la meilleure stratégie découverte pour gagner un maximum, voir (b).

Nous avons donc décidé d'appliquer cette théorie à un problème bien plus simple que le but final qu'est Pong. Nous avons donc écrit un algorithme de jeu proche de celui de Nim : un ensemble de 11 batons est présenté à chaque joueur, et chacun son tour, chaque joueur doit retirer 1 ou 2 batons. Le but étant de ne pas retirer le dernier baton.

Nous avons entraîné notre "intelligence" avec différents adversaires : un adversaire jouant "aléatoirement", un adversaire "modéré", un adversaire utilisant la même table d'apprentissage que notre intelligence, et un adversaire parfait. Les résultats sur les courbes d'apprentissages ont suggéré que dans la suite, il serait plus judicieux d'entrainer notre algorithme avec un adversaire modéré, ou dont la difficulté varie dans le temps (un parallèle peut être effectué avec l'apprentissage chez l'être humain : confronter un novice à un expert ne fera pas avancer le novice lors de jeux compliqués).

L'étape suivante fut l'application sur un simili-labyrinthe, très simplifié. Encore une fois, les résultats étaient concluants, et la relative complexité du problème par rapport au jeu de Nim nous a permis pour la première d'introduire dans le Q-learning un réseau de neurones. Nous ne pouvions le faire précédemment car l'entrée de l'algorithme n'était en fait qu'un nombre : le nombre de batons restants. Il était donc impossible de donner "plus" ou "moins" de poids à différentes composantes de l'entrée, puisque celle-ci n'en avait qu'une.

Malheureusement pour notre utilisation, l'état consiste en une image à 160×160 dimensions. Il est impossible d'énumérer tous les états, et donc de définir la fonction Q par un tableau. Il est donc nécessaire de faire appel à un système de réseau de neurones afin d'estimer cette fonction.

L'étude préliminaire théorique du jeu de Pong a révélé plusieurs problématiques propres au jeu de pong :

— L'utilisation d'un réseau de neurones est absolument nécessaire : utiliser simplement l'observation comme indexeur d'états donnerait un ensemble d'états possible bien trop grand (position de la balle, des joueurs, changement du score, etc...)

— ce réseau de neurones sera a priori un réseau à convolution, puisque la problématique de la detection de "patterns" et de leur positionnement est primordiale en jouant à Pong.

— L'utilisation des 3 couleurs (RGB) n'est pas forcément utile, on se limitera par exemple à une combinaison linéaire du rouge et du bleu, qui donnent le plus de contraste entre les différents éléments de l'écran.

D'autres problèmes d'un autre genre se sont présentés à nous lors de l'utilisation des différents modules :

— Le module atari_py, module principal de nos travaux, n'est pas compatible avec Windows.

— Il semblait au premier abord que le module gym nécessitait absolument une sortie graphique.

— La bibliothèque atari de gym n'a que très peu de documentation en ligne : il nous a été nécessaire de regarder le code source, heureusement disponible en libre accès, pour pouvoir manipuler la bilbliothèque.

Une fois ces quelques problèmes surmontés, les premiers tests d'intelligence artificielle pour pong ont été effectués. Et les résultats sont les suivants :

Les principes du Q-Learning et des réseaux de neurones étant expliqués, il est temps pour l'équipe de mélanger les deux algorithmes. La problématique est la suivante : nous aimerions appliquer l'apprentissage par Q-Learning à pong où un état serait représenté par une image. Toutefois, la taille de l'espace des états étant tellement grande (de l'ordre de quelques centaines de millions d'états), qu'une Q-Table basée sur l'état du jeu serait inutile : la probabilité de retomber sur un état déjà visité étant proche de 0.

Nous introduisons alors un réseau de neurones, prenant la représentation du jeu en entrée, et qui a pour but d'estimer la valeur de Q. Dans notre cas, deux types d'implémentation sont possibles :

— Le réseau prend en entrée l'état et l'action, et renvoie la qualité de l'association état/action (elle fait office à proprement parler de fonction de qualité Q)

— Le réseau prend en entrée l'état, et possède N sorties (N étant le nombre d'actions possibles), qui correspondent aux qualités des arcs respectifs. Le cas de pong est plutôt simple puisque l'on n'autorise que deux actions : haut et bas.

La première chose à faire est d'essayer cette implémentation à l'aide d'un perceptron classique, sans convolution. Avec un perceptron à une couche de 1000 neurones, l'apprentissage est plutôt « rapide », mais quelque chose cloche : le réseau a en fait appris tous les chemins comme étant des chemins perdants, puisqu'il a passé bien plus de temps à perdre qu'à gagner, et que les états gagnants et perdants sont visuellement très proches (le perceptron n'arrive pas à les distinguer).

La chose qui semble alors logique afin d'éviter ce problème est de passer à des CNN, afin d'ajouter une dimension de « localité » aux patterns détectés.

Pour vérifier l'implémentation correcte de notre Q-Learning par réseau de neurones, nous tentons d'abord de l'appliquer sur un jeu « trivial » qu'est le jeu de Nim. Le jeu de Nim est un jeu très simple qui se joue à deux joueurs. Un certain nombre d'allumettes sont alignées, et les joueurs jouent chacun leur tour, en retirant entre 1 et 3 allumettes. Le joueur gagnant est celui qui retire la dernière allumette.

Nous sommes donc en présence d'un jeu, avec l'un des joueurs étant le réseau et l'autre étant une intelligence pré-codée, avec différents états (le nombre d'allumettes faisant office d'états) et différentes actions (prendre 1,2 ou 3 allumettes). Un perceptron a été mis à l'épreuve au jeu de Nim, et les résultats ont été tout à fait concluants (mettre des graphes, pipoter, etc...)

Vient maintenant le tour de pong, et sa complexité infiniment supérieure à celle du jeu de Nim.

L'équipe se heurte alors à un obstacle : les temps de calcul. La convergence du réseau à convolution sur l'image de pong semble être trop lente pour aboutir à des résultats concluants sans avoir à faire tourner le programme pendant plusieurs jours voire semaines (les simulations étaient effectuées sur une carte graphique NVIDIA GTX 1050 Ti). Des solutions alternatives commencent alors à être envisagées :

— Récompenser le réseau non plus seulement sur son taux de victoire mais également sur son temps de survie. Effectivement, on peut imaginer qu'avec ce système de récompense, le réseau aura tendance à enregistrer les différents mouvements à effectuer en fonction de l'image pour pouvoir renvoyer la balle. Ce système de « récompenses intermédiaires » a été pensé en analogie avec une voiture sur un circuit qui devrait apprendre au fur et à mesure comment progresser sur ce circuit, et aurait pour but d'accompagner le réseau sur son apprentissage.

— Utiliser non pas l'image du jeu, mais des « features », c'est-à-dire des valeurs scalaires décrivant l'état actuel de l'image de façon parfaite. Un exemple de features utilisables sur pong est : position des raquettes, position de la balle, vitesses horizontales et verticales de la balle, ce qui fait un total de 6 features,

décrivant un unique état. Nous aurions donc en entrée du réseau 6 valeurs, et en sortie une ou deux valeurs, en fonction de la manière dont a été codée le réseau.

— Pré-traiter l'image afin d'appliquer un gros contraste sur les éléments intéressants du jeu sans en perdre l'information. Le pré-traitement que l'on trouve généralement dans les implémentations de réseaux de neurones pour l'apprentissage de pong est le suivant : donner la valeur 0 à tous les éléments d'arrière-plan, donner la valeur 1 à tous les « sprites » (balle, raquettes), et effectuer la différence de deux images consécutives (les raquettes étant obligées de se déplacer, et la balle étant en mouvement), afin d'obtenir une image qui contient la vitesse et la position de la balle, ainsi que les directions et les positions des deux raquettes. Eventuellement réduire la taille de l'image pour simplifier le réseau.

Le problème moral de ces implémentations intermédiaires est le fait d'aider le réseau. Plus le réseau a besoin d'aide, moins il est indépendant, et moins le sentiment d'avoir « fait apprendre à la machine » est présent. Par exemple, la décision de prendre des features arbitraires donne un sentiment de « triche » qui va à l'encontre du but final du projet.

L'une des solutions serait justement de ne pas choisir de features arbitraires, mais de laisser un réseau de neurones « choisir » ces features, à l'aide d'un auto-encodeur. Le principe est le suivant : l'auto-encodeur est un réseau de neurones comportant des couches en « sablier », il prend l'image en entrée et a pour but de retransmettre cette image en sortie. La couche de « features » est une des couches intermédiaires du réseau.

Les différentes implémentations ont été essayées… sans succès. Vous trouverez les courbes correspondantes à différentes implémentations ci-dessous.

Le modèle semblait souvent converger vers une mauvaise solution, dans laquelle la raquette se bloquait en haut ou en bas de l'écran. Les scores étaient le plus souvent dus au hasard.

Cette étape du projet a conduit à quelques semaines de désespoir pour toute l'équipe, l'atteinte du but du projet semblait bel et bien impossible. Les semaines s'enchaînaient, en s'interrogeant toujours plus sur les résultats obtenus par les différents articles de référence, leur puissance de calcul nécessaire et disponible…

Les professeurs encadrants constatant l'exaspération des élèves décidèrent de mettre de côté le problème visiblement trop complexe qu'est pong, pour se pencher sur un autre environnement fourni par le module python gym, appelé cartpole, ainsi que sur un mini-jeu codé par un élève de l'équipe Eponge, une variante du jeu mobile « Flappy Bird ».

PRINCIPE DE FLAPPY BIRD : Un oiseau soumis à la gravité doit passer au travers d'obstacles consécutifs en passant par une ouverture. Les deux actions possibles sont « aller vers le haut » ou « ne rien faire ».

Le réseau peut prendre en entrée l'image ou des features adéquates (nous nous sommes contentés des features pour l'apprentissage afin de vérifier l'implémentation de nos réseaux de neurones associés au Q-Learning), et le résultat était plutôt convaincant.

PRINCIPE DE CARTPOLE : Un pendule retourné attaché à un chariot doit rester en équilibre le plus longtemps possible. Le réseau a deux actions possibles : diriger le chariot vers la gauche ou vers la droite. Encore une fois, le réseau dispose de features associées à l'état du chariot.

L'une des particularités des implémentations de Cartpole est l'utilisation de deux réseaux de neurones synchronisés périodiquement. En effet, l'apprentissage du réseau « principal » met en jeu les résultats du réseau principal (la sortie théorique dépend d'autres sorties du même réseau, etc…), et peut provoquer l'oscillation de tout le système. Le moyen de contrebalancer ce problème est de produire un réseau

« secondaire » duquel dépendront les sorties théoriques du réseau principal, et qui sera synchronisé périodiquement.

Pour notre part, nous avons essayé l'implémentation avec un seul réseau. Il nous a permis de dépasser le score de 200 à CartPole, mais pas le maximum. Nous n'avons donc pas encore observé d'instabilité à ce jour.

CONCLUSION

Le projet a sans aucun doute été une expérience extrêmement enrichissante pour tous les étudiants des deux équipes. Certes, le but final qui était de produire une intelligence artificielle pouvant jouer à pong n'a pas été atteint, toutefois nous avons réussi à intégrer cette solution à des systèmes moins complexes comme cartpole ou Flappy Bird. Nous manquions de temps et de moyens techniques pour pouvoir reproduire les résultats de nos articles de référence.

L'une des difficultés du Q-learning est de réaliser un algorithme totalement séparé de la logique du jeu, alors que notre instinct naturel nous pousse à donner de l'aide à notre intelligence artificielle afin de la rendre plus performante. Ce n'est pas forcément un problème dans l'absolu, mais cela ne permet pas de comparer des implémentations, qui peuvent *in fine* différer légèrement dans leur logique.

Toutefois, le temps nous ayant manqué, nous avions encore pas mal de pistes à explorer :

— Modifier la multitude d'hyperparamètres (taille des filtres, nombre de filtres, pas des filtres, nombre de couches, nombre de neurones, vitesse d'apprentissage, taux d'exploration, taux de rafraîchissement, etc. . .) de notre réseau afin de trouver un ensemble de paramètres permettant une performance suffisante pour obtenir une convergence d'un réseau qui gagne à pong en un temps correct.

— Trouver un différent pré-traitement de l'image tel que les informations n'en soient pas perdues, tout en ne la déformant pas trop.