CentraleSupélec

Rapport de projet

Ready Player One : Deep Reinforcement Learning

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

2018 – 2019

Nathan Cassereau
Paul Le Grand Des Cloizeaux
Thomas Estiez
Raphaël Bolut

Professeurs encadrants :
Joanna Tomasik
Arpad Rimmel

Établissement :
CentraleSupélec cursus Supélec (promotion 2020)

Table des matières

Table des figures

Introduction

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

Le projet est mené par deux groupes de quatre élèves, afin de pouvoir comparer les performances des deux produits finaux. Notre groupe, le groupe « Eponge », est composé de Raphaël Bolut, Nathan Cassereau, 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 à pour but de nous apprendre les mécanismes du machine learning, nous étudierons plusieurs algorithmes différents au cours de l'année, avec lesquels nous expérimenterons. Les différents codes utilisés lors de ce projet sont consultables sur GitHub.

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

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

Nous allons ensuite rentrer dans le vif du sujet : Le Q-learning, appliqué au jeu vidéo Pong. Nous allons pour cela réaliser une interface grâce à laquelle notre algorithme pourra intéragir avec le jeu, pour lui permettre d'apprendre. 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

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 a été abandonné alors, son exécution étant 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 adaptés aux calculs matriciels.

Le modèle du neurone est le suivant :

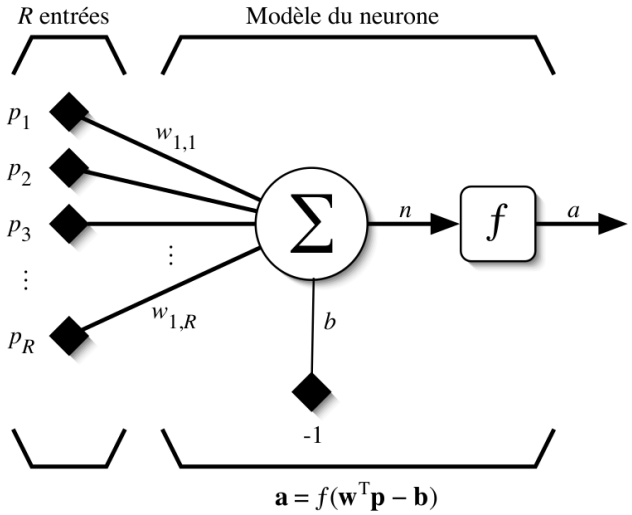

Figure 1: Modèle d'un neurone artificiel

Le neurone est composé de différents élements :

- p_1, p_2, ..., p_R constituent les R variables d'entrées du perceptron. Le nombre d'entrée 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. Il mesure l'importance accordée à chaque entrée. Un poids plus important signifie que l'entrée associée est plus pertinente pour ce neurone que les autres.
- Le biais b
- Le niveau d'activation n
- La fonction d'activation f
- La sortie a

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

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

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

On a alors :

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

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

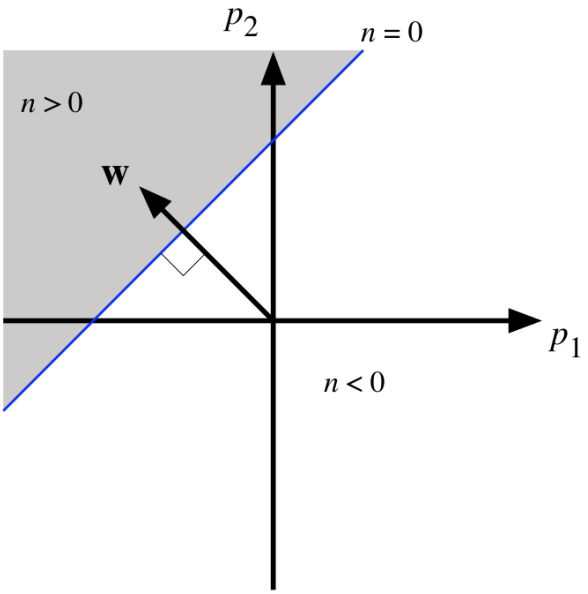

Figure 2: Domaine de séparation du neurone

Ce neurone n'est capable de traiter 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 élaborées. Pour cela on utilise plusieurs neurones sur une même couche. Chacun d'entre eux reçoit la même entrée mais 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 associé une ligne de la matrice). De la même manière, on retrouve un vecteur de biais (qui sont essentiellement des poids dont l'entrée est constante à -1), et un vecteur de niveaux d'activation. Finalement, on retrouve ce modèle :

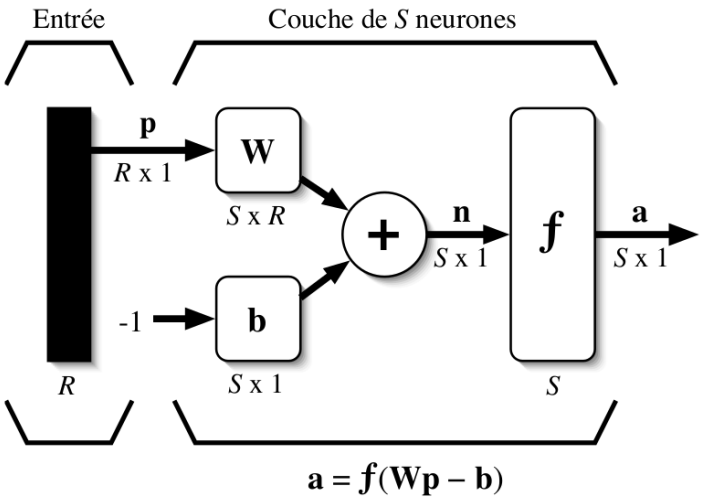

Figure 3: Représentation matricielle d'une couche de S neurones recevant R entrées

Pour pouvoir définir des ensembles de solution plus complexes, on ajoute d'autres couches de neurone. 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.

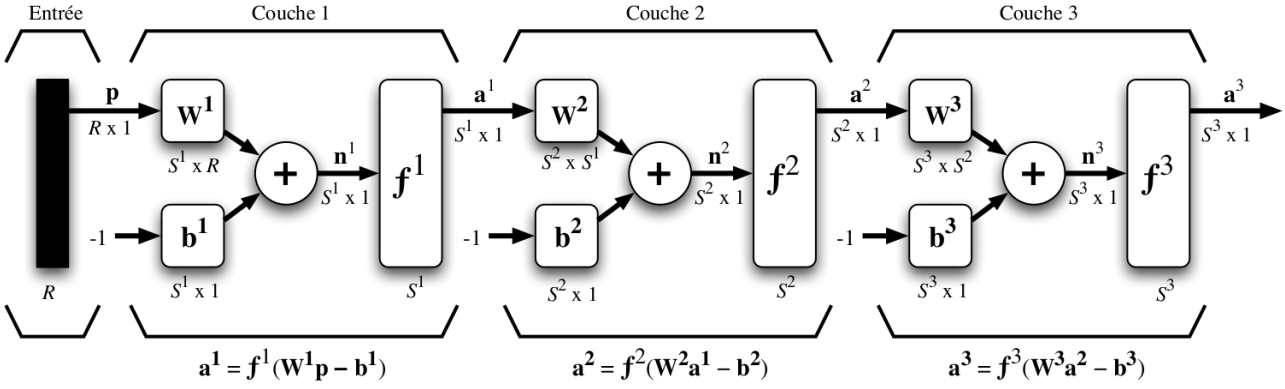

Figure 4: Modèle du perceptron multicouche

1.2 Apprentissage

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 qu'avec les données collectées, il y ait également la "bonne" réponse pour que le réseau puisse apprendre en conséquence. D'autres méthodes comme l'apprentissage non supervisé existent, cette dernière reposant uniquement sur les jeux d'entrées, le réseau devant alors les discriminer lui-même sans connaître la "bonne" réponse.

Pour réaliser cela, on présente à notre réseau une entrée et, puisqu'on dispose de la sortie attendue, on peut mesurer à quel point le réseau s'est trompé. C'est le rôle de la fonction d'erreur. Plus celle-ci est importante, moins le réseau est adapté pour cette donnée. Il existe différentes fonctions d'erreur. Une des plus utilisées est la somme des carrés des écarts entre la valeur attendue et la valeur calculée :

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

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

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

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

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

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

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

On note également l'utilisation du raccourci suivant :

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

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

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

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

- On calcule les sensibilités

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

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

- On calcule les changements de poids

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

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

1.3 Implémentation

Afin de pouvoir comprendre en détail le fonctionnement du perceptron, nous avons commencé dans un premier temps à implémenter un version de celui-çi chacun de notre côté. Cela nous a permis de commencer à réfléchir à l'architecture du code que nous voulions, et de pouvoir comparer les performances des différentes implémentations.

Nous avons testé dans un premier temps les résultats de nos perceptrons sur la fonction XOR. Cette fonction est un bon départ pour pouvoir avoir un code fonctionnel, car il s'agit d'une fonction ne pouvant pas être répliquée par une fonction linéaire : il faut au moins une couche cachée afin de pouvoir l'implémenter grâce à un perceptron. Cette prémière étape nous a permis de comparer les résultats et les performances de nos algorithmes, et de pouvoir choisir l'implémentation du perceptron que nous avons utilisé par la suite.

1.4 Résultats

2 Les CNN

2.1 Le modèle du CNN

Bien qu'étant un modèle efficace, le perceptron reste cependant coûteux en calcul : il faut réaliser les opérations matricielles sur tous les neurones de chaque couche. Les réseaux neuronaux convolutifs (convolutional neural network (CNN) en anglais) permettent de réduire le nombre de calculs réalisés. Ils se basent sur la structure de 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 (Convolutionnal Neural Networks) 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 couche de convolution
2) Une couche d'activation
3) Une couche 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 channels (pour une image RGB, on aura N = 3). On applique en parallèle sur cette images M filtres de dimension identique (h, l, n) avec n = N. La sortie de chaque filtre correspondra à un des M channels de l'image de sortie.

Figure 5: 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 :

$$J(x,y) = I \star H(x,y) = \sum_{i=0}^{h-1} \sum_{j=0}^{l-1} \sum_{k=0}^{N-1} I(x+i, y+j, k) \times H(i,j,k) \tag{12}$$

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 :

$$reLU(x) = x \tag{13}$$

si x > 0

$$reLU(x) = 0 \tag{14}$$

sinon

Cette partie 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, qui est plus utilisé en pratique : pour une image J, on choisi un paramètre S (S $<$ L, H), et on otient la sortie K telle que :

$$K(x, y) = max_{i,j \in [0,S-1]^2} J(x \times S + i, y \times S + j) \tag{15}$$

On peut ainsi enchainer les couches de convolution jusqu'à une couche finale, qui sera ensuite reliée à un réseau complètement connecté (qui n'est rien d'autre qu'un simple perceptron), pour pouvoir exploiter les résultats obtenus jusque là. Les poids à optimiser sont alors les poids des matrices "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 avec un perceptron), à l'exception que certains poids sont 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 :

Figure 6: Image à reconnaître

2.3 Utilisation de TensorFlow

TensorFlow a été utilisé pour coder le CNN. Un codage à la main du CNN aurait été possible et très instructive 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 coder 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.

TensorFlow est facile d'installation sous Linux et sa prise en main est plutôt aisée. Il existe de nombreux tutoriels pour construire des CNN. Cependant, le développement de TensorFlow évolue rapidement et la documentation n'est pas forcément très complète. L'implémentation de fonctionnalités précises est donc parfois difficile. Il est nécessaire de faire de longues recherches pour trouver les fonctions désirées.

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

L'exploitation des résultats a posteriori s'est avérée plus compliquée. L'interface de TensorFlow offrant des fonctionnalités plutôt limitées, l'utilisation d'un post-traitement s'est avéré indispensable. Il est par exemple impossible de réaliser des moyennes sur plusieurs runs. Après quelques recherches, nous avons réussi a 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 avéré 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.

Le post-traitement nous a permis de travailler sur l'évolution du taux de réussite en fonction du nombre d'apprentissage et en fonction du temps.