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 :

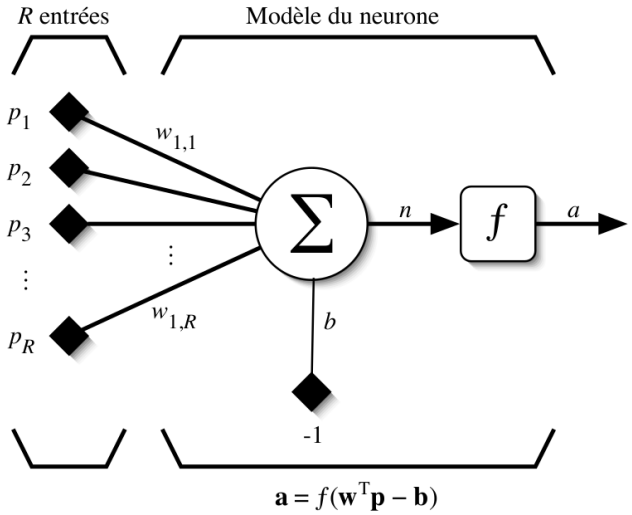


Figure 1: Modèle d'un neurone artificiel

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.

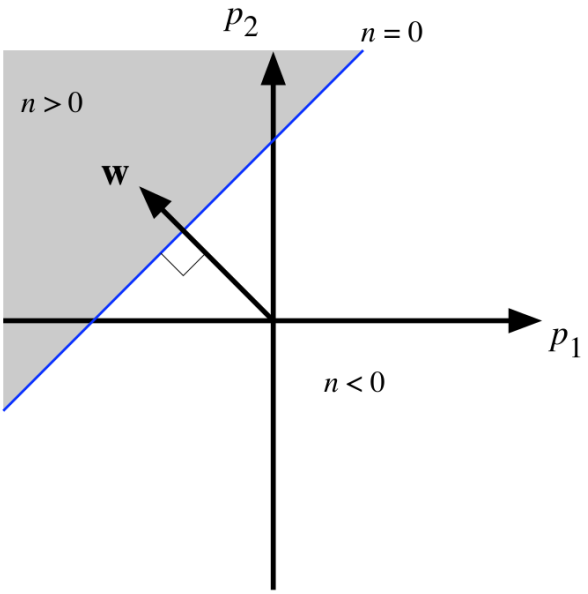


Figure 2: Domaine de séparation du neurone

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 :

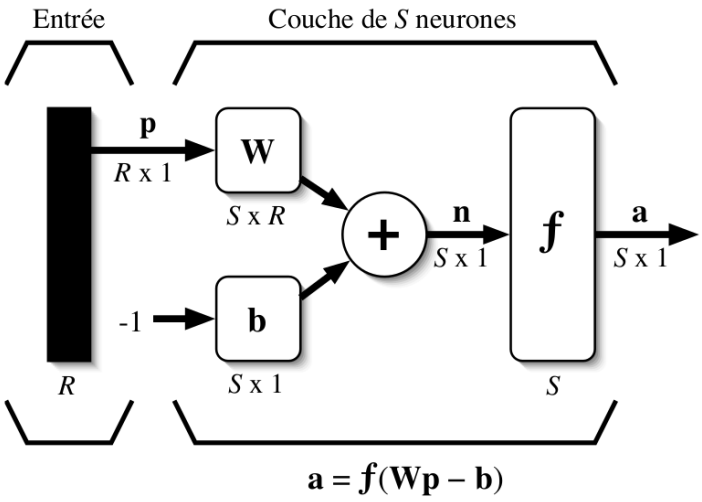


Figure 3: Représentation matricielle d'une couche de S neurones recevant R entrées

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.

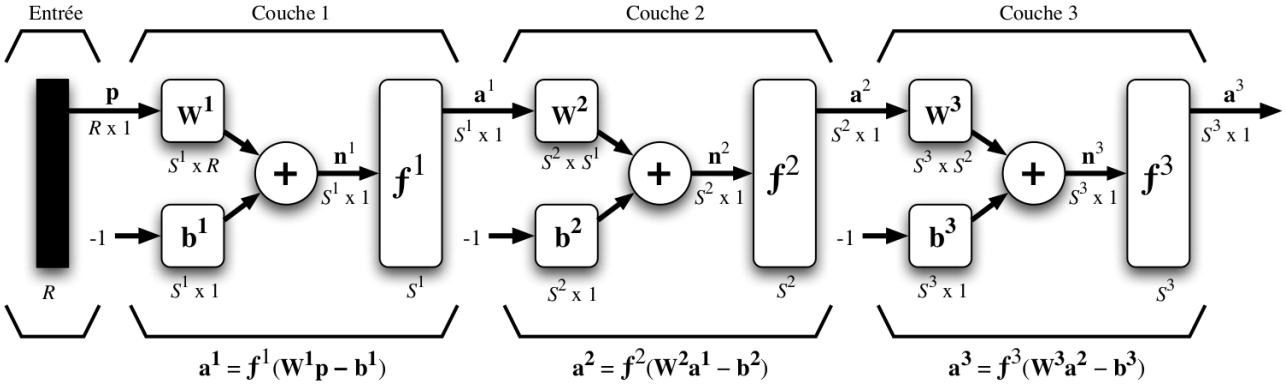


Figure 4: Modèle du perceptron multicouche

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 La fonction d'erreur

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

La formule de l'entropie croisée est la suivante :

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

1.3.1 Comportement pour une erreur quadratique

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

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

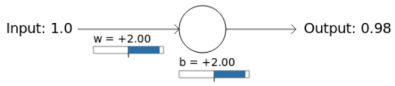
(a) Réseau utilisé et son initialisation

(b) Courbe d'apprentissage du neurone

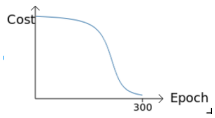
Figure 5: Première initialisation du réseau d'exemple utilisant l'erreur quadratique

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

On considère maintenant un second exemple. Le même réseau est utilisé mais en ayant une initialisation différente, de sorte que la sortie soit plus proche de 1, donc que l'on est plus "tort" puisqu'on cherche à retrouver 0. Les données et résultats de cet exemple sont montrés figure 6b. On remarque que l'apprentissage est de qualité moindre. En effet, juste après l'initialisation, le neurone commettait une erreur plus importante, mais l'apprentissage est beaucoup plus lent. On doit donc réaliser un nombre d'itérations suffisant pour retomber dans le cas 5 et pouvoir apprendre correctement.



(a) Réseau utilisé et son initialisation



(b) Courbe d'apprentissage du neurone

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

1.3.2 D'où provient ce comportement ?

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

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

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

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

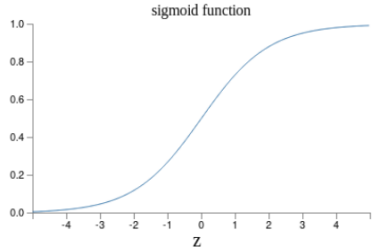


Figure 7: Graphe de la fonction sigmoide

1.3.3 Une solution à ce problème d'apprentissage

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

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

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

La formule de la chaîne nous donne

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

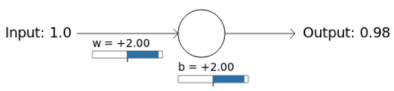
En utilisant l'expression voulue (équation 16), on obtient

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

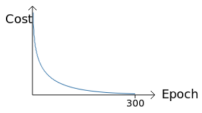
$$C = -y\ln(a) - (1-y)\ln(1-a) + Const \tag{19}$$

On comprend alors que la formule de l'entropie croisée n'est pas simplement une formule qui se trouve avoir des propriétés intéressantes, mais que l'on peut construire cette fonction de coût pour respecter les conditions des équations 15 et 16. L'expression 12 est alors une condition suffisante et quasiment nécessaire au respect desdites

contraintes. Le “quasiment” provient de la constante d’intégration. Puisque ce n’est pas la fonction de coût en elle-même qui est intéressante mais plutôt ses variations et ses dérivées partielles, alors on peut se contenter d’une constante nulle, ce qui conserve la positivité de C. On s’attend alors à une amélioration considérable de l’apprentissage observé figure 6. La courbe d’apprentissage 8b montre 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 qui est moins sensible à l’initialisation et qui apprend d’autant plus qu’il commet une grosse erreur. Ce comportement est parfois obtenu en faisant varier le taux d’apprentissage au cours du temps. Sur ces exemples, le taux d’apprentissage est constant. Ce comportement souhaité étant un artéfact de la fonction de coût, on s’attend à des performances supérieures de la part des réseaux utilisant l’entropie croisée.



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



(b) Courbe d’apprentissage du neurone

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

1.3.4 Une explication intuitive

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

1.4 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.

1.5 Résultats