L'apprentissage profond appliqué à la bataille navale

I. Objectifs

L'objectif de notre projet était de créer une intelligence artificielle et de lui apprendre à jouer la bataille navale. Pour cela on a construit et entraîné un réseau de neurones artificiels. Pour obtenir un temps d'entraînement raisonnable on utilise une version 6x6 de la bataille navale à la place 10x10; et le réseau comporte 3 couches de 36 neurones artificiels.

II. Historique

L'apprentissage automatique (machine learning) et plus précisément l'apprentissage profond (deep learning) ont été théorisés dans la seconde moitié du XXème siècle. Au lendemain de la seconde guerre mondiale, des informaticiens et scientifiques américains se penchent sur la question de l'intelligence artificielle. Ce terme désigne alors la capacité d'un algorithme accumuler des connaissances et prendre des décisions sans intervention humaine. Dans les années 50 la possibilité d'une IA qui égale les performances humaine paraît réalisable dans un avenir proche.Notamment grâce aux premiers travaux sur les modèles de neurones artificiels. Mais dans les années 60, la capacité de stockage, la puissance de calcul et le manque d'exemples probants mettent en évidences le fait que celles-ci ne peuvent pas réellement raisonner.
Plus tard, l'arrivée du Big Data permet de régler le problème du manque d'exemples et le machine learning est relancé. Le développement de la théorie l'apprentissage automatique met en évidence plusieurs modèles d'apprentissages: le supervisé, non supervisé, renforcé Ici nous allons utiliser l'apprentissage par renforcement: le programme implémenter va s'exécuter un très grand nombre de fois en analysant ses résultats pour apprendre de ses erreurs.

III. Modélisation et outils mathématiques

III.1. Perceptron multicouche

Le perceptron multicouche est le type de réseau de neurones artificiels utilisé dans ce projet informatique. Il est composé de plusieurs couches de neurones où chaque neurone est connecté à tous les neurones de la couche inférieure. La première et dernière couche sont respectivement appelée "couche d'entrée" et "couche de sortie", et toutes les autres sont appelées "couches cachées". Chacune de ces connexions est représenté par un poids (nombre réel) qui mesure l'importance celle-ci.

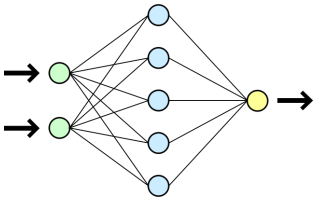

Figure 1: Perceptron à 3 couches
(source : wikimédia.org)

III.2. Neurone formel

Un neurone formel est constitué d'une fonction activation (on utilisera la fonction sigmoïde) prenant en argument la somme pondérée des valeurs des neurones de la couche inférieure connectés à celui-ci .

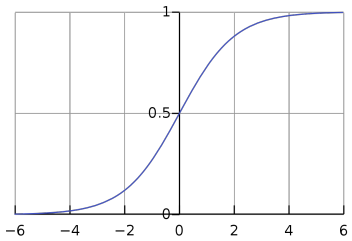

Figure 2: Graphique de la fonction sigmoïde

III.3. Propagation directe

En notant :

- $a_i^{(j)}$ la valeurs du i-ième neurone de la j-ième couche
- $p_{ki}^{(j)}$ le poids de la connexion entre le i-ième neurone de la j-ème couche et le k-ième neurone de la j+1-ième couhe
- $z_k^{(j+1)} = \sum_{i=1}^{n} p_{ki}^{(j)} \cdot a_i^{(j)}$ l'entrée du k-ième neurone de la j+1-ième couhe

On a la formule suivante $a_k^{(j+1)} = f(z_k^{(j+1)}) = f(\sum_{i=1}^{n} p_{ki}^{(j)} \cdot a_i^{(j)})$ permettant de propager l'information vers les couches supérieures.

Et si on adopte la notation matricielle suivante :

- $A^{(j)}$ la matrice colonne des neurones de la j-ième couche
- $P^{(j)}$ la matrice de poids entre la j-ème couche et la j+1-ième couhe
- $Z^{(j+1)} = P^{(j)} \times A^{(j)}$ la matrice des entrées des neurones de la j+1-ième couhe

La formule précèdente devient :

$$A^{(j+1)} = f(Z^{(j+1)}) = f(P^{(j)} \times A^{(j)})$$

où "$f(Z^{(j+1)})$" veut dire qu'on applique f sur chaque élément de $Z^{(j+1)}$.

III.4. La fonction objectif

La fonction objectif (aussi appelée fonction coût) est l'outil permettant de mesurer l'erreur du réseau :

$$C : ((y_i), (\hat{y}_i)) \mapsto \frac{1}{2} \cdot \sum_i (y_i - \hat{y}_i)^2$$

y_i étant les valeurs souhaitées et $\hat{y}_i$ les valeurs des neurones de la couche de sortie.

III.5. L'entraînement

Le but de l'entraînement est de modifier les poids du réseau de façon à minimiser la fonction objectif, c'est un problème d'optimisation. Pour savoir comment modifier les poids on peut calculer le gradient de la fonction objectif dont les poids sont les variables.

III.5.1. La rétropropagation

Pour connaître le gradient il faut calculer toutes les dérivées partielles de la fonction objectif C par rapport à chaque poids, pour faire cela on utiliser la règle de la chaîne.

Calcul des dérivées par rapport aux poids entre la deuxième et la dernière couche :

$$\frac{\partial C}{\partial p_{ki}^{(2)}} = \frac{\partial C}{\partial \hat{y_k}} \cdot \frac{\partial \hat{y_k}}{\partial p_{ki}^{(2)}} = -(y_k - \hat{y}_k) \cdot \frac{\partial \hat{y_k}}{\partial z_k^{(3)}} \cdot \frac{\partial z_k^{(3)}}{\partial p_{ki}^{(2)}} = -(y_k - \hat{y}_k) \cdot f'(z_k^{(3)}) \cdot a_i^{(2)}$$

On peut retrouver ces termes par le aussi par le calcul matriciel :

$$\delta^{(3)} = -(Y - A^{(3)}) \circ f'(Z^{(3)})$$

$$D^{(2)} = \delta^{(3)} \times {A^{(2)}}^T$$

L'opération $\circ$ est le produit matriciel d'Hadamard (produit terme à terme de deux matrices de mêmes dimensions). $D^{(2)}$ est une matrice 36x36 contenant tous les termes $\frac{\partial C}{\partial p_{ki}^{(2)}}$.

On peut faire la même chose pour les $\frac{\partial C}{\partial p_{ki}^{(1)}}$:

$$\delta^{(2)} = ({P^{(2)}}^T \times \delta^{(3)}) \circ f'(Z^{(2)})$$

$$D^{(1)} = \delta^{(2)} \times {A^{(1)}}^T$$

Exemple de produit d'Hadamard :

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \circ \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} \end{pmatrix}$$

III.5.2. Le gradient de C

En récupérant tous les éléments des matrices $D^{(1)}$ et $D^{(2)}$ on obtient toutes les composantes de $\nabla f(x)$. x étant le vecteur dont les composantes sont tous les poids du réseau.

Le vecteur $-\nabla f$ indique la direction dans laquelle la fonction objectif C diminue.
L'algorithme du gradient se résume à ces étapes :

- Calcul de $-\nabla f(x_n)$
- $x_{n+1} = x_n - \alpha_n \nabla f(x_n)$, α_n est un coefficient représentant la taille des pas effectué par l'algorithme.

On arrête l'algorithme quand $||\nabla f(x_n)|| < \varepsilon$, ε étant un nombre petit que l'on fixe.
L'algorithme du gradient possède plusieurs défauts majeurs : il peut ne jamais s'arrêter, s'arrêter dans un minimum local, s'arrêter alors que x n'est même pas un minimum local de C.

IV. Informatique

IV.1. Le réseau

Pour construire le réseau de neurones on utilise la programmation orientée objet. On commence par créer l'objet Réseau en définissant une classe Réseau avec comme attributs (variables que possède l'objet) : la taille des 3 couches, les 2 matrices de poids qu'on remplie avec des nombres aléatoires. On lui crée les méthodes (fonction que possède l'objet) suivantes :

- propagation : effectue les calculs matricielles détaillés dans la partie III.3. à l'aide de NumPy.
- fonction_objectif : effectue le calcul détaillé dans la partie III.4. .
- getParams : renvoie le vecteur ayant comme composante tous les poids du réseau.
- setParams : permet de modifier les poids du résau.
- fonction_objectifPrime : effectue les calculs de $D^{(1)}$ et $D^{(2)}$ détaillé dans la partie III.5.1. .
- gradient : renvoie le vecteur $\nabla f(x)$ à partir des calculs de $D^{(1)}$ et $D^{(2)}$.

IV.2. Le jeu de la bataille navale

On crée la classe BatailleNavale de la manière suivante : lors de la création d'un objet BatailleNavale, on crée trois attributs (taille du plateau de jeu, un tableau représentant le plateau de jeu visible par le joueur, un tableau représentant le plateau adverse).
On place aléatoirement les "bâteaux" en choisissant d'abord leur sens (horizontal ou vertical), on choisie ensuite un emplacement en faisant attention qu'il soit libre. On ajoute à l'objet la méthode torpille : modifie le plateau visible en indiquant si le joueur a touché ou non un navire ennemi. Renvoie aussi si le tir était possible (case non attaquée).

IV.3. Entraînement

On crée la classe Trainer qui permet d'appliquer l'algorithme BFGS (un algorithme plus efficace que celui du gradient) de la bibliothèque scipy.optimize.
On crée aussi une fonction exemple qui permet de sélectionner des exemples intéressants de parties de la bataille navale et d'entraîner le réseau dessus : on fait jouer le réseau des parties entières jusqu'à qu'il en finisse une en beaucoup de tours, c'est-à-dire que le résau a mal joué puis on prend un tours de jeu au hasard dans cette partie et on entraîne le réseau sur celui-ci.
On enregistre le réseau grâce à la bibliothèque Pickle pour pouvoir l'utiliser plus tard.

IV.4. Interface

On crée une fenêtre Tkinter dans laquelle on y ajoute un caneva où l'on dessine les deux grilles de jeu (plateau visible par le joueur, et le plateau adverse). On lie l'événement d'un clic gauche de souris avec une fonction qui se charge de jouer la case visée si le clic est dans la grille supérieure, de faire jouer l'ordinateur si le clic est dans la partie intermédiaire et de révéler l'emplacement des bâteaux si le clic est dans la grille inférieure.