Apprentissage par renforcement profond

Nathan Barloy
Telecom Nancy
Universite de Lorraine
Email : nathan.barloy@telecomnancy.eu

Aurélien Delage
Telecom Nancy
Universite de Lorraine
Email: aurelien.delage@telecomnancy.eu

Olivier Buffet
et Vincent Thomas
INRIA
Equipe : Larsen
E-mail : olivier.buffet@loria.fr
et vincent.thomas@loria.fr

***Résumé*—Ce document est une trace des recherches que nous avons mené sur le thème de l'apprentissage par renforcement profond.**

L'objectif premier de ce projet était de créer une intelligence artificielle qui apprend seule à jouer à un jeu vidéo. Cependant, nous n'avons pas réussi (par manque de temps entre autres) à créer cette intelligence artificielle. Cet article présente différentes notions importante dans la création de cette I.A : les réseux de neurones et les réseaux convolutifs, qui ont permis de reconnaitre des chiffres manuscrits, ainsi que le Q-learning, qui a permis de créer une I.A. performante dans certains cas.Le DQN (qui n'a pas pu être mis en place). Tous les réseaux que nous avons créés ont été fait grâce à la bibliothèque TensorFlow, développée par Google.

I. Introduction

Les prémices de l'intelligence artificielle datent d'il y a plus de 60 ans. Depuis, de nombreux progrès ont été réalisés grâce à des avancées théoriques majeures [3].

Ces progrès se sont traduits par des réussites des algorithmes d'intelligence artificielle. On peut par exemple citer la célèbre victoire de l'algorithme *Deep Blue* contre M.Garry Kasparov, champion du monde d'échec, en 1996. On peut aussi citer la victoire d'*AlphaGo* battant en 2017 le champion du monde du jeu de Go Lee Sedol. Sa version adaptée au jeu d'échec est aussi particulièrement impressionnante tant dans l'originalité du jeu proposé que dans l'incapacité reconnue des maîtres d'échec à comprendre réellement, et à anticiper les coups de l'algorithme.

AlphaGo (et AlphaZero,son successeur étendu à d'autres jeux) utilisent des méthodes qui seront entre autres le sujet de cet article. En effet, AlphaGo utilise des réseaux de neurones profonds [4], et AlphaZero utilise le principe de l'apprentissage par renforcement profond.

Néanmoins, les réseaux de neurones ne sont qu'une petite partie de toute la recherche en intelligence artificielle [11]. On peut par exemple citer les Processus décisionnels de Markov [13], les arbres de décisions [14] ou bien encore la programmation dynamique [9]. Ce qui explique l'omniprésence des algorithmes utilisant des réseaux de neurones est l'avancée théorique du Deep Learning datée d'il y a quelques années (2014-2015).

Dans ce PIDR, nous nous proposons d'appliquer certaines de ces techniques, entre autres, à des jeux de type Atari.

Nous utiliserons la librairie Tensorflow [16] pour python, et un livre d'initiation [22].

Nous étudieront dans un premier temps les techniques de Deep Learning en expliquant ce que sont les réseaux de neurones, puis nous aborderons une des techniques d'apprentissage par renforcement qu'est le Q-learning. Enfin, nous essayerons d'associer ces deux méthodes afin d'appliquer la théorie de l'apprentissage par renforcement profond (ou Deep Reinforcement Learning).

II. Etat de l'art

A. Les réseaux de neurones

Les réseaux de neurones sont une des méthodes utilisées dans l'apprentissage automatique (machine learning en anglais), c-à-d. qu'ils permettent à un ordinateur "d'apprendre" des choses par elle-même, que ce soit à partir d'une base de données ou non.

De manière simplifiée, un réseau de neurones est un système composé de couches et de noeuds, qui effectue des opérations arithmétiques sur des données en entrée. Le résultat final de ces opérations constitue la sortie du réseau. Ces derniers réalisent des performances particulièrement impresionnantes sur certains problèmes, notamment de classification. Pour appuyer cette affirmation, on peut citer une étude des résultats d'un réseau de convolution sur la classification d'images [2] Dans ce type de problème, les données en entrée sont des points dans un espace pouvant être de grande dimension, et la sortie du réseau est une décision d'appartenance du point à une classe de points. Un des champs de recherche actuel est d'essayer de réduire le nombre de dimension des données en entrée [15].

On peut néanmoins craindre un certain effet boîte noire des réseaux de neurone [20]. En effet, le maillage des différentes couches peut être compliqué et le nombre de couches très grandes. Les caractéristiques du réseau, à savoir les poids des noeuds (nous l'expliquerons juste après) ne réprésente plus grand chose, et il devient difficile de savoir comment le réseau a raisonné pour traiter les données qu'il a reçues.

Avant d'expliquer ce qu'est un réseau de neurones, il faut déjà expliquer ce qu'est un neurone. C'est simplement une structure qui récupère des signaux en entrée, les traite de manière simple, puis renvoie un signal en sortie. (voir Fig.1)

Un neurone artificiel s'inspire des neurones humains. En effet, un neurone humain est une cellule du cerveau dont la membrane reçoit des impulsions électrique au cours du temps.

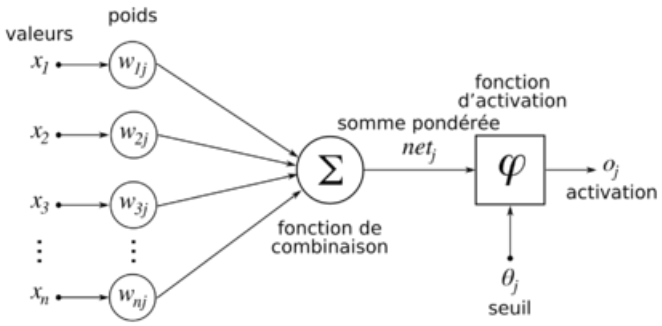

FIG. 1 –. *représentation d'un neurone*

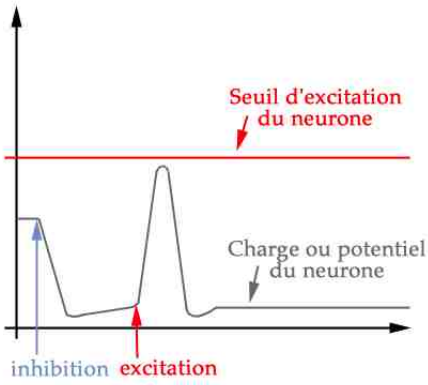

FIG. 2 –. *Potentiel électrique de la membrane neuronale (en mV) en fonction du temps*

Si le potentiel électrique est supérieur à un seuil, ce neurone est alors excité [1]. (voir Fig.2)

Ces neurones intéragissent alors entre eux pour créer des fonctionnements plus complexes.

Dans notre cas, un neurone va simplement faire un somme pondérée des entrées, puis appliquer une fonction d'activation pour donner une valeur de sortie. De manière simplifiée, on peut dire qu'à l'instar du neurone humain, et grâce à la fonction d'activation, si ce résultat est supérieur à un seuil, le neurone est activé, sinon il ne l'est pas. Une fonction d'activation est une fontion de $\mathbb{R} \mapsto [0,1]$ *ou* $[-1,1]$. Différentes fonctions existent [9], la plus utilisée est la fonction sigmoïde : $s_a(x) = \frac{1}{1+e^{-a}}$. Ces neurones sont ensuites regroupés en couches, et plusieurs couches forment un réseau (voir Fig.3)

On remarque que toutes les valeurs de la couche i-1 sont les entrées de tous les neurones de la couche i ($i \neq 1$) : c'est ce qu'on appelle un réseau entièrement connecté. A chacune de ces connexions est associé un poids.

Mathématiquement, cela s'écrit comme suit:

Notons n_i^j le neurone i de la couche j, $\forall i \in [0,p], \forall j \in [0,n]$, en supposant qu'il y a n couches, et p neurones à la couche i.

Alors on a :

$$\boxed{n_i^{j+1} = \sum_{k=1}^{p}(n_k^j \cdot W_{k,i}^j) + b} \quad (1)$$

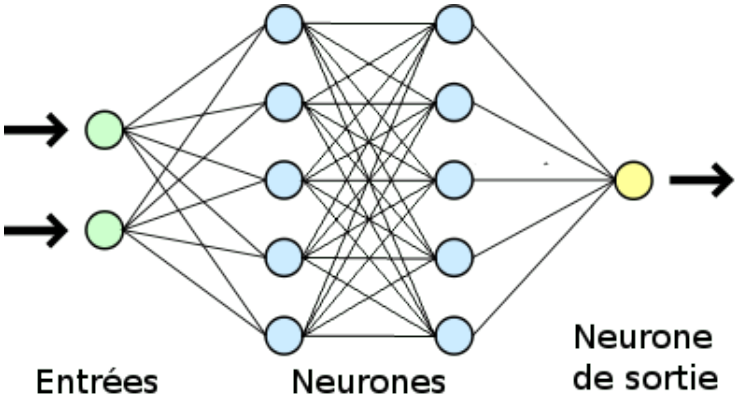

FIG. 3 –. *Représentation schématique d'un réseau de neurone*

où on note $W_{k,i}^j$ les poids, $k \in [0,p]$ associés aux neurones $i \in [0,p]$ de la couche j.

On voit que l'on peut réecrire cette formule sous forme matricielle. Prenons soin de définir correctement les vecteurs et les matrices utilisés.

- $\forall j \in [0,n]$, $n^j \in \mathbb{R}^p$
- $\forall j \in [0,n]$, $W^j \in \mathbb{M}_{n,p}$
- $\forall j \in [0,n]$, $b \in \mathbb{R}^p$

et on a :

$$\boxed{n^{j+1} = W^j \cdot n^j + b} \quad (2)$$

Les poids pouvant être grand, $\forall j \in [0,n]$, $n^j \subset \mathbb{R}^p$. On aimerait qu'à l'instar du neurone humain, son état soit activé ou non activé, ie que l'on ait :

$$\forall j \in [0,n],\ n^j \subset [0,1]^p. \quad (3)$$

Ceci se fait en utilisant une fonction d'activation, telle que définie précedemment.

On remarque que la fonction f pour passer d'une couche à une autre est une fonction affine, de partie linéaire $\mathcal{L}(f) = W$ qui n'est pas nécessairement bijective. f ne l'est alors pas. Ceci montre que l'on peut obtenir des mêmes résultats tout en ayant des poids sur les neurones totalement différent. Cela a une importance quand on teste la sensibilité du réseau à des petites pertubations non prévues : si les poids sont trop grands, il y a un risque de propagation de l'erreur et de divergence par rapport à la solution attendue.

Les techniques d'Intelligence Artificielle à base de réseaux de neurones sont des techniques dîtes *supervisées*. "L'aprentissage supervisé est une tâche d'apprentissage automatique consistant à apprendre une fonction de prédiction à partir d'exemples annotés [17]". C'est à dire que l'on donne au réseau, en même temps que l'entrée, la sortie qu'il devrait obtenir. *L'apprentissage non supervisé* est une tâche où les données d'entrée ne sont pas étiquetée de leurs sorties attendues.

Les neurones humains ont la faculté de renforcer leurs interconnexions. En effet, quand on apprend une nouvelle langue, certaines connexions entre neurones se renforcent, et deviennent

de plus en plus facile à emprunter, d'où l'apprentissage de la langue. Les réseaux de neurones utilisent le même procédé. L'idée est de renforcer certaines connexions en donnant un poids plus important entre neurones, ie un coefficient de la matrice W fort.

Pour apprendre l'Anglais, il faut pratiquer. Pour un réseau de neurones, c'est la même chose! On nourrit le réseau avec un grand nombre de données en entrée et de la sortie attendue, ce qui lui permet d'apprendre les poids de ses matrices W de chaque couche, afin d'avoir une prédiction la plus fiable possible.

1) Technique de rétro-propagation de l'erreur, du gradient: À chaque étape (chaque fois qu'on donne un jeu de donnée au réseau de neurones), le réseau doit mettre à jour ses poids. Une des techniques les plus connues pour faire ceci - et que nous utiliserons dans notre projet - est la *Rétro-propagation du gradient*.

Pour entraîner le réseau, on lui fourni en entrée des jeux de données ainsi que les résultats attendus pour chacuns. On notera y_{connu} et y_{pred} respectivement les étiquettes des données d'entrée et les sorties calculées par le réseau.

Le résultat de sortie sera nécessairement un vecteur $y_{pred} \subset \mathbb{R}^p$, p pouvant valoir 1.
On définit alors *la fonction de perte* $\mathcal{L}$ comme étant une mesure de la différence entre y_{connu} et y_{pred}. On peut alors utiliser plein de fonctions différentes, comme $||y_{connu} - y_{pred}||_1$, $||y_{connu} - y_{pred}||_2$, $||y_{connu} - y_{pred}||_\infty$ ... Ce sont des applications différentiables. En effet, si on leur associe un produit scalaire, alors $||x||^2 =< x,x >$, or le produit scalaire étant une application bilinéaire de $\mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$, elle est différentiable.

La fonction de perte est alors différentiable, sauf en 0, on peut donc ***suivre son gradient*** pour minimiser la fonction de perte, ie $\tilde{x} = x - \alpha\nabla_x(f)$, avec α un coefficient arbitraire. En effet, géométriquement, la différentielle d'une fonction dans un espace vectoriel (ici $\mathbb{R}^p$) correspond à l'hyperplan tangent à la surface.

Dans $\mathbb{R}^p$, $\nabla_x(f)(h) = Jac_x(f) \cdot h$, où $Jac_x(f)$ est la jacobienne de f en x définie par (pour $f : \mathbb{R}^p \mapsto \mathbb{R}$) :

$$\nabla_x(f) = (\frac{\partial f}{\partial x_1}(x),..,\frac{\partial f}{\partial x_n}(x)) \tag{4}$$

et on a l'équation suivante de l'hyperplan tangeant à $f(x)$:

$$\boxed{y(h) = f(x) + \nabla_x(f) \cdot h} \tag{5}$$

Et alors, $-\nabla_x(f)$ donne la direction dans laquelle la fontion est plus faible. En se déplaçant alors selon cette direction, on minimise la fonction f (il faut régulièrement mettre à jour le gradient, si la fonciton est irrégulière).

Pour expliquer ce mécanisme de descente de gradient, plaçons nous dans un cas simple. On cherche par exemple à minimiser une fonction $f : \mathbb{R}^p \mapsto \mathbb{R}$).

Or, la dérivée de f est en chaque point, la droite tangente à la courbe. En se déplaçant dans la direction de la pente négative de cette courbe, on se déplace vers des valeurs plus faible de la fonction [10].

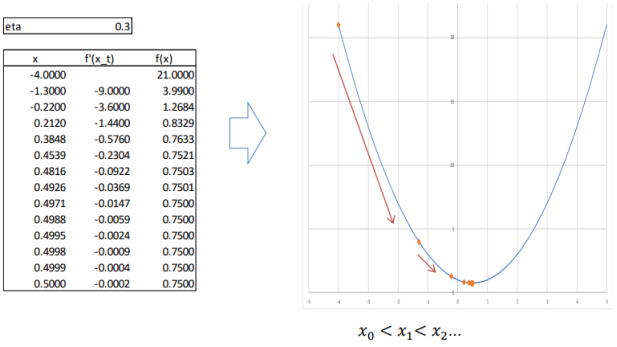

FIG. 4 –. *Illustration de la méthode de descente de gradient pour une fonction réelle.*

L'algorithme de descente de gradient est alors le suivant , en notant $(x_k)_{k\in\mathbb{N}}$ la suite des valeurs des abscisses lors de la descente:

algorithme 1 Algorithme de descente de gradient

Require: f fonction différentiable ,ϵ seuil de tolérence, α coefficient de déplacement le long du gradient
$k = 0$
while $||\nabla(f)(x_k)|| \leq \epsilon$ **do**
Mise à jour éventuelle du coefficien α
Calculer le gradient de f en x_k
$x_{k+1} = x_k - \alpha \cdot \nabla(f)(x_k)$
$k = k + 1$
end while

On comprend alors la nécessité que la fonction soit dérivable (différentiable si elle est dans $\mathbb{R}^p$)

La mise à jour des poids par algorithme de rétropropagation du gradient est un peu complexe et lourde en notation, nous envoyons le lecteur sur cette page [7].

L'apprentissage repose sur cette technique qui permet de mettre à jour les poids par rétropropagation du gradient.

B. Les réseaux de neurones convolutifs

Les réseaux convolutifs sont utilisés quand il y a trop de données en entrée, et que celles-ci sont liées entre elles : c'est notamment le cas pour des images en entrée, où des pixels proches dans l'espace sont fortement liés, contrairement à 2 pixels éloignés. Dans ce cas, on va appliquer un filtre local, appelé noyau de convolution, sur toute l'image, ce qui permet d'obtenir une autre image, un peu plus petite, nommée carte de caractéristique. (voir Fig.5)

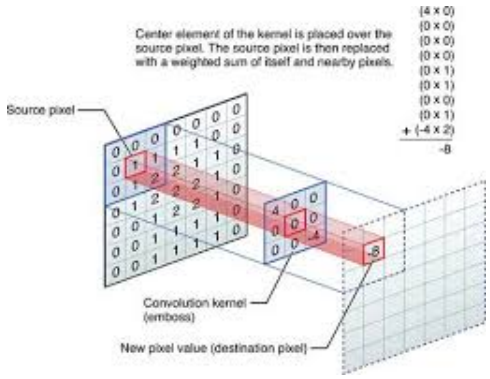

FIG. 5 –. *Noyau de convolution*

Si on utilise plusieurs noyaux de convolutions ensemble, on obtient ce qu'on appelle un couche de convolution, qui extrait donc plusieurs cartes de caractéristique. On applique ensuite sur ces cartes de caractériqtique un oppération de pooling, qui permet de réduire la taille des données, puis on peut réappliquer une opération de convolution sur ces cartes. La Fig.6 illustre ce à quoi peut ressembler un réseau convolutif.

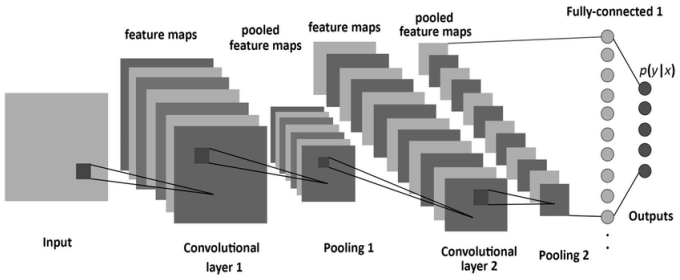

FIG. 6 –. *Architecture convolutive*

On a une première couche de convolution qui extrait 7 cartes de caractéristiques, puis un couche de pooling, puis une autre couche de convolution qui extrait 12 cartes de caractéristiques, une autre couche de pooling, et on fini avec un réseau entièrement connecté (les réseaux vus précédemment). Ici, les poids que les réseau va apprendre sont ceux des différents noyaux de convolution, il n'y a aucun apprentissage au niveau du pooling.

C. Le Q-Learning

Le Q-learning est un des algorithmes d'apprentissage par renforcement. Dans l'apprentissage par renforcement, un agent se déplace dans un ensemble d'états S grace à des actions d'un ensemble A. A chaque instant, il obtient ou non une récompense R. La spécificité du Q-learning est d'introduire une fonction de valeur, notée Q, qui donne une valeur à chaque action dans un état donné.Formellement, on a

$$\begin{array}{rcccc} \mathcal{Q} & : & \mathcal{S}\times\mathcal{A} & \to & \mathbb{R} \\ & & s,a & \mapsto & \mathcal{Q}(s,a) \end{array}$$

Cette Q-valeur est souvent initialisée à 0, et on la met à jour lorsque l'agent parcours les différents états grace à la formule suivante :

$$\mathcal{Q}(s,a) := (1-\alpha)\cdot \overbrace{Q(s,a)}^{\text{ancienne valeur}} + \overbrace{\alpha}^{\text{coeff. d'apprentissage}} \cdot \left(\underbrace{r}_{\text{récompense immédiate}} + \gamma \cdot \underbrace{max_{a'\in\mathcal{A}}\mathcal{Q}(s',a')}_{\text{valeur espérée}} \right) \quad (6)$$

où s' est l'état atteint à partir de s avec l'action a, α le coefficient correpondant à l'importance que l'on veut donner à l'aprentissage par rapport à la valeur actuelle, et où γ est une contante inférieure à 1, ce qui assure la convergence de cette fonction.

Le coefficient α est un coefficient qui doit être grand au début, et diminuer au fil du temps. En effet, au début de l'expérience, l'agent n'a rien apprit, donc l'apprentissage est très fort. Cependant, une nouvelle expérience ne doit pas être plus importante que les anciennes, et c'est pourquoi plus on a apprit, moins une nouvelle expérience est importante. Cependant, si ce facteur α décroît trop vite, alors au bout d'un moment, le reste des expériences futures sera négligeable devant les expériences passées. Ainsi, il existe un critère sur α indiquant que la série des α doit être divergeante, et la série des α^2 doit être convergeante. Typiquement, on a $\alpha = \frac{1}{n}$, avec n le nombre d'expériences réalisées.

Se pose alors la question : comment faire évoluer mon agent lors de ces expériences ? En effet, s'il choisit toujours une action au hasard, il n'utilise jamais ce qu'il a apprit, et ne pourra pas se déplacer efficacement. Au contraire, s'il suit toujours l'action qui maximise la Q-valeur actuelle, il y a peu de chance qu'il trouve les chemins optimisés. C'est le dilemme exploration-exploitation.

Dans les fait, on va choisir au hasard l'une des 2 stratégies à chaque fois. La probabilité de choisir l'exploration au début doit être grande, car on n'a encore rien appris, et plus le temps passe, plus on va faire confiance à la Q-valeur apprise, et faire de l'exploitation.

La preuve de la convergence de l'algorithme du Q-learning nécessite des connaissances de probabilité (lemme de Borel-Cantelli pour la convergence presque-sûre) et d'analyse (théorème du point fixe pour les applications contractante) qui prendraient trop de temps à définir ici, et ce n'est pas le but de ce PIDR. Nous renvoyons le lecteur vers la première preuve de convergence de ce théorème [5] et qui donne aussi les conditions sur α et sur γ pour qu'il y ait convergence.

D. le DQN

L'apprentissage par renforcement profond est un mélange des deux concepts précédents. Ici, nous allons parler du DQN, qui adapte le Q-learning avec un réseau de neurones.En effet, il y a beacoup de situations où il n'y a pas un nombre fini d'états, et construire un fonction de valeur Q est alors impossible. C'est pourquoi on essaye alors d'approximer cette Q-valeur avec un réseau de neurones. Ce réseau prend en entrée tous les paramètres qui décrivent un état (typiquement des positions, des vitesses,...), et ressortent une valeur pour chacune des

actions possibles (ces valeurs représentes la recommandation de ces actions pour les entrées données, ce sont les Q-valeurs). Il faut aussi adapter la formule de mise à jour de la Q-valeur, puisqu'on ne dispose plus que d'une approximation, et qu'elle ne peut pas s'appeler elle-même pour se mettre à jour (voir Fig.7).

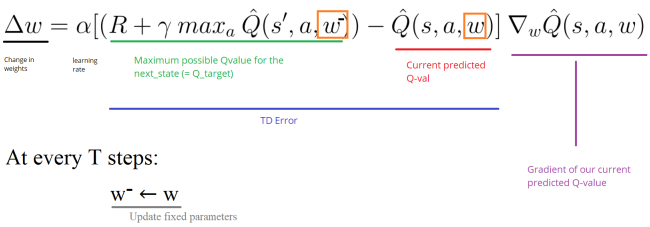

FIG. 7 –. *Formule pour le DQN*

Il faut alors entrainer ce réseau avec des batch, qui prennent au hasard un nombre donné d'expériences enregistrées dans le passé. De plus, il y a 2 réseaux qui existent en même temps : un qui évolue constamment, et un qui est un copie du premier, faite régulièrement, afin d'avoir un base pour les formules utilisées.

III. MÉTHODOLOGIE

A. *Les réseaux de neurones*

On l'a dit précédemment, un neurone se calcule de la façon suivante (1) $n_l^{j+1} = \sum_{k=1}^{p}(n_k^j \cdot W_k^{j,l}) + b$. Il s'agit d'un produit scalaire auquel on ajoute la constante b. Ceci est *exactement* l'équation d'un hyperplan de dimension p-1 dans un espace de dimension p. On peut alors interpréter le travail d'un neurone dans un réseau de neurone comme la délimitation d'un hyperplan séparant en deux les données en entrée de la couche.

Pour visualiser ceci, prennons l'exemple du xor en nombre flottants. Nous disposons donc de nombres flottants entre 0 et 1. On représente alors le résultat (jaune correspond à 1 et bleu correspond à 0) d'un point (x,y) qui est le résultat de l'opération x xor y.

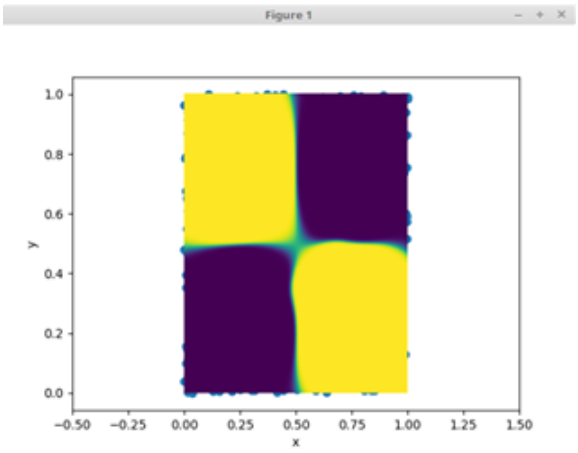

FIG. 8 –. *Illustration de la classification correspondant au xor en nombres flottants*

Et l'on voit se dessiner des délimitations de zone en fonction du résultat de l'opération. On peut donc supposer que c'est ce qu'a fait le réseau : il a délimité l'espace en hyperplan grâce à ses noeuds.

On voit qu'à la limite entre deux zones, la frontière est floue. Celà s'explique par le fait qu'à cette limite, le réseau a du mal à distinguer l'appartenance à une zone ou à une autre, dû à la proximité des données en entrée.

La compléxité des réseaux de neurones (nombre de neurones, nombre de couches cachées) dépend naturellement de la complexité de délimiter des zones dans un espace de dimension n. Les surfaces de séparation sont d'ailleurs plus compliquées en dimension n. Par exemple, en dimension 3, ce sont des surfaces.
Ainsi, la reconnaissance de chiffres entre 0 et 9 peut se faire avec le nombre de pixels comme nombre de neurones d'entrée, deux couches cachées de 16 neurones, et 10 neurones de sortie qui correspondent chacun à l'activation pour le chiffre 0, le chiffre 2,..., le chiffre 9. En utilisant un réseau de neurones avec plus de couches et plus de neurones (3 couches à 16 neurones par couche), nous sommes arrivés aux taux de reconnaissance suivants :

0	reconnu à	93.55%
1	reconnu à	96.19%
2	reconnu à	83.69%
3	reconnu à	84.58%
4	reconnu à	84.56%
5	reconnu à	78.95%
6	reconnu à	86.42%
7	reconnu à	90.62%
8	reconnu à	76.18%
9	reconnu à	84.46%
total	reconnu à	86.17%

Ces valeurs ont été obtenues avec un coefficient de descente de gradient $\alpha = 0.01$, 5000 entrainement, fournis chacuns

de 60 000 images. L'algorithme utilisé pour la descente de gradient est AdamOptimizer [6], un outil disponible dans la librairie tensorflow [16]

B. Les réseaux de neurones convolutifs

On a vu précédemment que pour les images, utiliser des réseaux convolutifs est souvent plus efficace. C'est donc ce que nous avons essayé de faire, pour améliorer les test de reconnaissance, toujours sur la base de données MNIST.

Tensorflow inclue déjà des structures qui permettent de faire de la convolution facilement, créer un réseau convolutif n'est donc pas très compliqué une fois que la théorie est en place.

Nous avons utilisé la structure LeNet5 (voir Fig.9), qui a déjà fait ses preuves sur cet exemple[21].

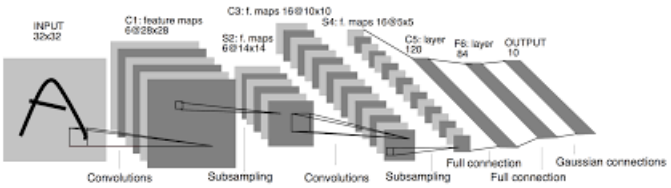

FIG. 9 –. *structure LeNet5*

Dans cette structure, les images de base font du 28*28, mais on rajoute 2 couches de 0 tout autours (du padding), ce qui donne du 32*32. On applique une première couche de convolution, avec des filtres 5*5, ce qui donne 6 cartes de 28*28. On applique ensuite un maxpooling, qui coupe la carte en morceaux de 4 pixel, et garde le maximum des 4. On obtient donc 6 cartes de 14*14. On utilise alors une autre couche de convolution, avec des filtres 5*5, et qui donne 16 cartes de 10*10, sur lesquelles ont réapplique le maxpooling pour avoir 16 cartes de 5*5, donc 400 entrées en tout. On utilise alors 2 couches de neurones entièrement connectés, de 120 et 84 neurones, puis enfin les 10 neurones de sortie (les chiffres de 0 à 9).

Avec cette structure, le taux d'erreur sur les tests a atteint les 0.7% au bout de 3 minutes d'apprentissage, ce qui est beaucoup mieux que les 10% obtenus avec uniquement des couches entièrement connectés.

C. Le Q-Learning

Pour appliquer le Q-learning, nous avons pris un exemple très commun et simple à réaliser : il s'agit d'un jeu où un personnage doit se déplacer de case en case pour atteindre une arrivée qui rapporte un certain nombre de points. Il y a des obstacles sur le chemin tels que des murs et des malus (voir Fig.10). Le but est alors d'obtenir le maximum de points. Cependant, lorsque le personnage fait un choix, il y a un risque (ici de 10%) pour qu'il glisse vers la droite ou vers la gauche, et n'aille donc pas dans la direction voulue.

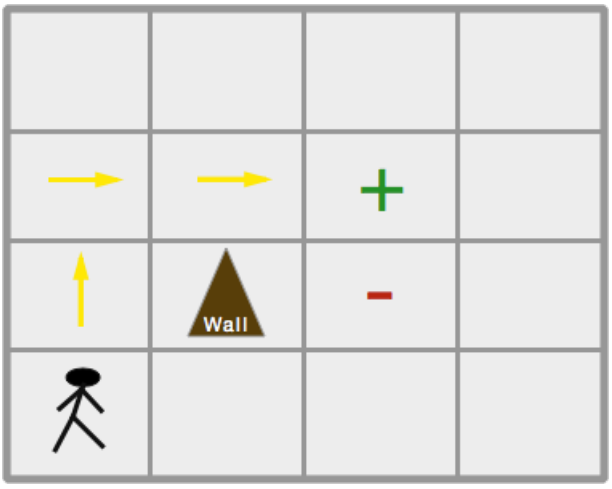

FIG. 10 –. *plateau du jeu*

Pour cela, on va expliciter la Q-valeur, qui consiste en une matrice avec sur les lignes les différents états, et sur les colonnes les actions (voir Fig.11). Il n'y a ici que 4 actions (se déplacer dans une des 4 directions), et n*m états (le nombre de cases du plateau, pour un plateau de taille n*m).

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	327	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

FIG. 11 –. *représentation de la Q-valeur*

Dans la suite, nous avous représenté notre plateau de jeu avec de l'ASCII,avec les significations suivantes :

- . → case vide
- # → mur
- o → trou (malus)
- @ → arrivée (bonus et fin)
- x → point de départ

En Fig.12, voici le 1er plateau que nous avons utilisé (à gauche le plateau de départ, à droite la solution trouvée par notre algorithme). On peut voir que le chemin trouvé est parfaitement celui attendu : le personnage atteint l'aarrivée sans aucun risque de tomber.

```
x....        0....
..o..        1.o..
.....        23...
#...@        #456@
```

FIG. 12 –. *1er plateau*

On peut aussi voir (Fig.13) que si on trace la somme de toutes les Q-valeurs en fonction du temps, on obtient quelque chose qui se stabilise, et c'est donc qu'une solution a été trouvé.

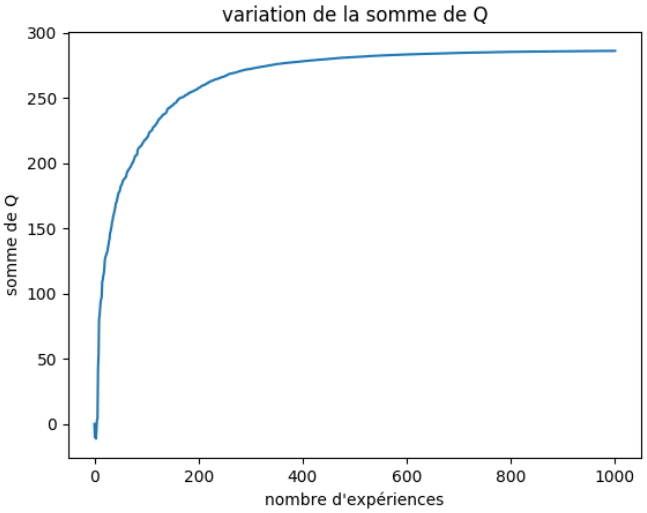

FIG. 13 –. *courbe de la somme des Q-valeurs*

Lors d'un 2e essai (Fig.14), nous avons voulu forcer notre I.A. à prendre des risques : il n'y a que 2 chemins possibles : un sans risque mais avec une petite récompense, et un avec un malus sur le chemin, mais une très grosse récompense à la fin. On se rend compte qu'en effet, l'I.A. choisi de prendre un malus, pour aller chercher un plus gros bonus derrière.

```
x...@        0...@
.###.        1###.
o#...        o#...
.#...        3#...
@....        @....
```

FIG. 14 –. *2d plateau*

On a ensuite fait un 3e essai, plus complexe, avec 3 arrivées possibles, et plus d'obstacles (Fig.15). Malheureusement, notre I.A ne va pas chercher ici la meilleur récompense, la meilleure (à l'opposé du plateau) est trop éloignée du point de départ, et une autre arrivée est sur le chemin. Cela est du à notre représentation de la fonction de Q-valeur, qui commence à être trop grosse. Pour voir de meilleurs résultats, il faudrait faire beaucoup plus d'expériences, ce qui serait trop long.

```
x...#.@      0...#.@
..o..o.      1.o..o.
....#..      234.#..
#...@..      #.56@..
....#..      ....#..
......@      ......@
```

FIG. 15 –. *3e plateau*

C'est un peu décevant, mais cela montre les limites de cette technique.

D. le DQN

Pour mettre en place l'algorithme de DQN, nous sommes partis du jeu flappy bird (code disponible sur internet), et nous avons essayé d'y adapter une I.A. qui joue à la place de l'humain. Nous avons réussi à obtenir un oiseau qui bougeait tout seul, mais malheureusement, ces mouvements étaient chaotiques, et il n'apprenait pas grand chose. Pire encore, il lui arrivait parfois de ne plus rien faire, et de mourir en boucle (il apprenait donc de mauvaises choses). Nous n'avons donc pas de résultats à présenter pour cette partie.

IV. CONCLUSION

Bien que nous ayons fait de nombreuses avancées, et maîtrisé certains domaines, nous n'avons pas pu atteindre notre objectif, qui était de créer une intelligence artificielle qui apprenne seule à jouer à un jeu, grâce à un algorithme d'apprentissage par renforcement profond.

Nous avons réussi à mettre en place un réseau de neurones entièrement connecté dans un premier temps, convolutif dans un second temps, qui arrive à reconnaitre des chiffres écrits à la main. Cela nous a permis de comprendre comment fonctionnait vraiment un réseau de neurones, et l'importance des différents paramètres utilisés (quelle fonction de loss utiliser, comment faire la rétropropagation, combien de couches mettre, avec combien de neurones).

Nous avons aussi réussi à mettre en place un algorithme de Q-learning, et ainsi créer une intelligence artificielle qui trouve les stratégies optimales, pour des problèmes pas trop gros. En effet, s'il y a trop d'états, notre I.A s'arrête sur un solution qui est bonne, mais qui n'est pas la meilleure, ce qui est un assez gros défaut.

ACKNOWLEDGMENT

Nous souhaiterions tout d'abord remercier nos 2 encadrants, M. Vincent Thomas et M.Olivier Buffet, qui nous ont accompagnés durant tout ce projet, nous ont conseillés et nous ont accueillis. Ils nous ont permis de découvrir des domaines de

l'informatique qui nous intéressent énormement. Nous souhaiterions aussi remercier notre école, Télécom Nancy, pour nous avoir permis de collaborer avec ces chercheurs du LORIA.

RÉFÉRENCES

[1] http://tpe.epilepsie.free.fr/gaba-intro.html.

[2] Marek Dabrowski and Tomasz Michalik. How effective is transfer learning method for image classification. In *Position Papers of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017, Prague, Czech Republic, September 3-6, 2017.*, pages 3–9, 2017.

[3] Université de Picardie Frédéric Fürst. Histoire de l'Intelligence Artificielle. https://home.mis.u-picardie.fr/~furst/docs/3-Naissance_IA.pdf.

[4] Google. AlphaGo Zero: Learning from scratch. https://deepmind.com/blog/alphago-zero-learning-scratch/.

[5] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. Convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[6] Diederik Kingma and Jimmy Ba. Adam: a method for stochastic optimization (2014). *arXiv preprint arXiv:1412.6980*, 15, 2015.

[7] Université de Lille Marc Tommasi. L'algorithme de rétropropagation du gradient. 1997. http://www.grappa.univ-lille3.fr/~gilleron/PolyApp/node28.html.

[8] Charles EDOU NZE. https://charlesen.fr/reseaux-de-neurones-en-javascript-avec-brain-js/.

[9] Université de Paris 6 Philippe Chrétienne. Programmation dynamique. www-poleia.lip6.fr/~spanjaard/articles/beosdp.pdf.

[10] Université de Lyon 2 Ricco Rakotomalala. Principe de la descente de gradient pour l'apprentissage supervisé. https://eric.univ-lyon2.fr/~ricco/cours/slides/gradient_descent.pdf.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[12] Inria Lille Rémi Munos. Principe de la descente de gradient pour l'apprentissage supervisé. http://researchers.lille.inria.fr/~munos/master-mva/lecture01.pdf.

[13] Olivier Sigaud and Olivier Buffet. *Processus décisionnels de Markov en intelligence artificielle*, volume 1 - principes généraux et applications of *IC2 - informatique et systèmes d'information*. Lavoisier - Hermes Science Publications, 2008.

[14] Université de Paris Dauphine Stéphane Airiau. Apprendre un arbre de décision. https://www.lamsade.dauphine.fr/~airiau/Teaching/ia/2017/ia-10-dt.pdf.

[15] Concours Centrale Supélec. Aplatissement aléatoire d'un ensemble de points en grande dimension. 2018. https://www.concours-centrale-supelec.fr/CentraleSupelec/2018/MP/sujets/M007.pdf.

[16] Tensorflow. https://www.tensorflow.org/.

[17] Wikipédia. https://fr.wikipedia.org/wiki/Apprentissage_non_supervis.

[18] Wikipédia. Apprentissage automatique. https://fr.wikipedia.org/wiki/Apprentissage_automatique.

[19] Wikipédia. Deep Blue. https://fr.wikipedia.org/wiki/Deep_Blue.

[20] Wikipédia. Réseaux de neurones : Points faibles et limites. https://fr.wikiversity.org/wiki/R%C3%A9seaux_de_neurones/Points_faibles_et_limites.

[21] Yoshua Bengio Patrick Haffner Yann Lecun, Léon Bottou. Gradient-based learning applied to document recognition. 1998. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[22] Bharath Ramsundar & Reza Bosagh Zadehn. *Tensorflow pour le deeplearning : de la régression linéaire à l'apprentissage par renforcement*. O'reilly edition, 2018.