Apprentissage par renforcement avec de l'apprentissage profond

Auteur:

Masini Tom

Responsable:

Clausel Marianne

2020 / 2021

Contents

Introduction

L'objectif de ce projet est d'étudier comment l'Apprentissage profond (Deep Learning en Anglais) peut être utilisé pour faire de l'Apprentissage par renforcement (Reinforcement Learning).

Sommairement, l'Apprentissage profond représente une famille de méthodes d'Apprentissage Machine (Machine Learning) mettant en œuvre des « Réseaux de Neurones » et dans lesquelles les données sont représentées sous forme de vecteur ou matrice.

Tandis que l'Apprentissage par renforcement désigne une situation d'apprentissage dans laquelle l'algorithme ajuste séquentiellement son modèle d'apprentissage en fonction des nouvelles données qu'il reçoit.

Durant ce projet, nous allons tout d'abord étudier l'Apprentissage profond (DL) en faisant un point sur le Perceptron Multi-Couche (Multi-Layer Perceptron). Ensuite nous allons nous concentrer sur l'Apprentissage par Renforcement (RL), et étudier les méthodes d'Apprentissage profond qui l'implémente (RNN). Puis, nous finirons sur une mise en application de DeepRL sur des données réelles.

1 Apprentissage profond (DL)

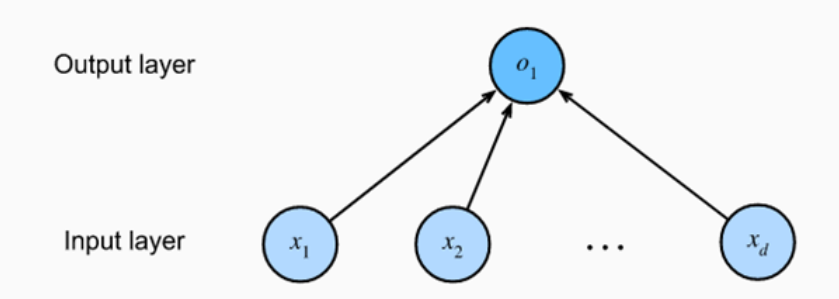
Dans cette section, nous allons étudier comment fonctionne l'apprentissage profond, quels sont les problèmes que l'on peut rencontrer lors de l'implémentation de ce type d'algorithme, ainsi que quelques solutions.

1.1 Présentation du Percreptron Multi-couche

Un modèle d'apprentissage profond peut être visualisé par ce que l'on appelle un Réseaux de Neurones dans lequel une donnée est représenté par un vecteur de $\mathbb{R}^d$ (ex. les features d'une image sont ses pixels qui seront codés par un vecteur dont le i-ème élément est la couleur du i-ème pixel, ainsi la dimension d des features de cette donnée sera le nombre de pixel de l'image).

Pour comprendre ce qu'est un réseau neuronal, il faut d'abord considérer sa brique élémentaire qui est le neurone Formel (figure 1).

Figure 1 – Neurone formel



Un neurone formel est composé d'une couche d'entrée à d inputs (d étant la dimension des features des données considérées), et d'une couche de sortie à un output $o_1 \in \mathbb{R}$ (qui est le label sur lequel on travaille).
Les inputs sont reliés à l'output par des "synapses" contenant chacun un poids $\omega_{i,1} \in \mathbb{R}$ (poids de la i-ème synapse reliant l'input i à l'output 1) de telle sorte que l'output soit la combinaison linéaire des inputs pondérés par les poids plus le biais b_1 du neurone :

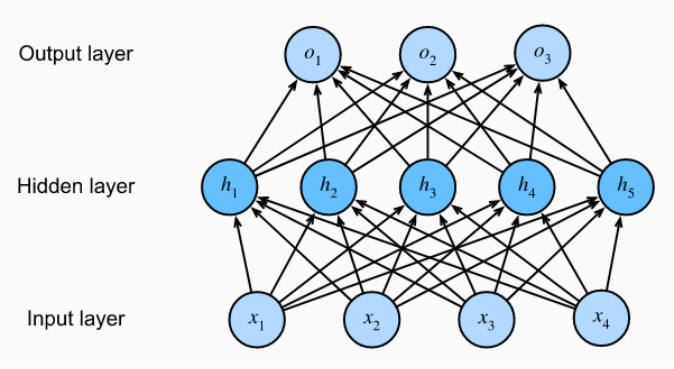
$$o_1 = b_1 + \sum_{i=1}^{d} \omega_{i,1} x_i.$$

Ainsi, ce neurone formel (aussi nommé Perceptron) équivaut exactement au modèle de régression linéaire.
Cependant, ce modèle (qui est le plus simple possible) ne peut que traiter des problèmes linéaires. Or, énormément de problème sont non-linéaire. Donc pour passer du linéaire au non-linéaire, nous introduisons le calcul de la sortie du neurone par une fonction d'activation $\sigma : \mathbb{R} \to [0; 1]$ possiblement non linéaire (ex. cette fonction peut être $ReLU(x) = max(x,0)$, la fonction échelon ou encore la fonction *sigmoid*). Puis nous empilons des couches de neurone formel (perceptron) tous entièrement reliés entre eux. Un tel réseaux neuronal est appelé Perceptron Multi-couche, et c'est le plus simple réseaux profond possible.

De façon générale, un Perceptron Multi-Couche est composé de C couches où la première couche est la couche des entrées (inputs) à d neurones, la C-ème couche est la couche des sorties (outputs) à q neurones, et les $C-2$ couches intermédiaires sont les couches cachées (qui sont composées de neurones possèdant une fonction d'activation σ).
Plus exactement, la dimension des inputs est la dimension d des features, la dimension q de la couche de sortie est souvent plus petite que d car le vecteur de sortie obtenu après le passage d'une donnée $x^k \in^d$ correspond à une nouvelle représentation de la données dans le nouvel espace $\mathbb{R}^q$. Enfin, chaque couche cachée $H^{(c)}$ (avec $c = 2, ..., C-1$) possède sa propre quantité r_c de neurones intermédiaires, où le neurone h de la couche c possède sa propre fonction d'activation $\sigma_{c,h}$.

Pour illustrer cela, la figure 2 montre un MLP à $C = 3$ couches, $d = 4$ neurones d'entrées, $q = 3$ neurones de sorties et $r_2 = 5$ neurones intermédiaires de même fonction d'activation σ.

Figure 2 – MLP



Supposons que l'on dispose du jeu de données d'entré $X = (x_k)_{k=1,...,n} \in \mathbb{R}^{n\times d}$. Donc le jeu de données de sortie sera $O \in \mathbb{R}^{n\times q}$, déterminé par :

$$H = \sigma(XW^{(1)} + b^{(1)}),$$
$$O = HW^{(2)} + b^{(2)}.$$

(Demander à Marianne Clausel si cette remarque est vraie, et si la laisser est pertinent) Voici une dernière remarque importante. Théoriquement, peu importe la complexité de la relation f qui existe entre les inputs et les labels, il existera toujours un MLP qui pourra apprendre cette relation f. Cette propriété fait du MLP un approximateur universel.
Cependant, ce MLP peut nécessiter énormément de couches cachées, de neurones ou de fonction d'activation plus ou moins complexes. C'est pourquoi dans de nombreux cas d'autres architectures de Réseaux de neurones pourrons être choisies car plus pratique (ex. CNN ou RNN).

1.2 Apprentissage - Propagation Avant et Rétro-propagation

Dans cette section, nous allons voir comment s'effectue l'apprentissage des poids du Réseaux de neurones MLP.

Nous nous plaçons dans le cadre de l'apprentissage supervisé, ainsi considérons un jeu de n données $(x^k,y^k)_{k=1,...,n}$ où les features sont $x^k = (x_1^k, ..., x_d^k) \in \mathbb{R}^d$ et le label à prèdire est $y^k \in \mathbb{R}$.

L'apprentissage des poids $(\mathbf{w},b)$ des neurones se fait de la même façon que dans la plupart des algorithmes d'apprentissage automatique supervisé, c'est à dire en cherchant à minimiser l'erreur d'entrainement sur le jeu de n données (appelé données d'entrainement) que l'on aura séparé en sous-ensemble (mini-batch β).

Soit l une fonction de perte qui, pour un certain paramètrage du réseau de neurone $(\mathbf{w},b)$ et une certaine donnée x^k, calcule l'erreur entre la prédiction $\hat{y^k}$ effectuée avec ce paramétrage et la vraie valeur à prédire y^k.
Souvent la fonction de perte est celle des moindres carrés :

$$l^{(k)}(\mathbf{w},b) = \frac{1}{2}(\hat{y^k} - y^k)^2.$$

Puis, on obtient l'erreur d'entrainement sur un mini-batch β :

$$L_\beta(\mathbf{w},b) = \frac{1}{n}\sum_{k\in\beta} l^{(k)}(\mathbf{w},b).$$

Avec,

$$L(\mathbf{w},b) = \sum_{\beta} L_{\beta}(\mathbf{w},b).$$

Enfin, les valeurs optimales des poids se déterminent par :

$$(\mathbf{w}^*,b^*) = \arg\min_{\mathbf{w},b} L(\mathbf{w},b).$$

Etant donné que dans la plupart des cas, la solution analytique au problème d'optimisation n'est pas trouvable, on l'approxime grace à l'algorithme d'optimisation de la Descente de Gradient (qui a chaque itération, mets à jour les poids dans la direction où le gradient diminue) que voici :

$$\begin{cases} \textit{Initialisation aléatoire} : (\mathbf{w}^0,b^0) \\ \textit{Mise à jour des poids} : \\ (\mathbf{w}^{j+1},b^{j+1}) \leftarrow (\mathbf{w}^j,b^j) - \eta.\partial_{(\mathbf{w}^j,b^j)}L \end{cases}$$

Remarques : Pour que la descente de gradient fonctionne bien, il faut certaines conditions sur la fonction L à optimiser, en particulier qu'elle soit Convexe.
De plus, comme nous le verrons à la section suivante, il faut faire très attention à l'initialisation des poids et bien choisir le taux d'apprentissage η (qui contrôle à quel point on modifie les poids à chaque itération), au risque d'aboutir à une instabilité numérique.

On remarque donc que pour mettre à jour les poids à travers le réseaux, il faut pouvoir propager une dérivée des outpouts aux inpouts. Cela se fait grâce à l'algorithme de la rétro-propagation du gradient. En fait, cet algorithme se base sur la formule de la dérivée d'une fonction composée (chain rule).

En reprenant le cas du MLP de la figure 2, la fonction suivante :

$$L = L(b^{(2)} + W^{(2)}.\sigma(XW^{(1)} + b^{(1)})) = L(O(W^{(2)}, H)), \quad o\text{ù } H = \sigma(XW^{(1)} + b^{(1)})$$

a pour dérivées partielles :

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial O}.\frac{\partial O}{\partial W^{(2)}},$$

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial O}.\frac{\partial O}{\partial b^{(2)}},$$

Ce qui permet donc de mettre à jour les poids des neurones de la deuxième couche (qui équivaut à la première couche en partant de la fin).

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial O}.\frac{\partial O}{\partial H}.\frac{\partial H}{\partial W^{(1)}},$$
$$\frac{\partial L}{\partial b^{(1)}} = \frac{\partial L}{\partial O}.\frac{\partial O}{\partial H}.\frac{\partial H}{\partial b^{(1)}}.$$

Ce qui permet donc de mettre à jour les poids des neurones de la première couche (qui équivaut à la deuxième couche en partant de la fin).

Ainsi, tous les poids des neurones du MLP de la figure 2 peuvent être mis à jour via la rétro-propagation du gradient.
Il en va de même pour tous les poids d'un MLP de profondeur L quelconque.

En somme, l'apprentissage se fait de la façon suivante :

1. On initialise les poids à ($\mathbf{w}^0$,b^0);
2. Propagation Avant du mini-batch β_1 : On donne en inputs les données du mini-batch qui vont parcourir tout le réseau pour générer les sorties dont on va calculer l'erreur $L_{\beta_1}(\mathbf{w}^0,b^0)$;
3. Rétro-propagation du mini-batch β_1 : Via l'optimisation par descente du gradient et la rétro-propagation, on calcule les nouveaux poids ($\mathbf{w}^1$,b^1));
4. Et ainsi de suite pour tous les mini-batch restants;
5. On peut refaire d'autres Epochs (= apprentissage sur tous les mini-batch) pour s'approcher au mieux de la solution ($\mathbf{w}^*$,b^*).

Remarque : Les mini-batch peuvent être de taille 1, ainsi la mise à jour des poids se fait une donnée après l'autre.

1.3 Stabilité Numérique et Initialisation

Bien qu'en théorie l'apprentissage des poids devrait toujours plus ou moins bien fonctionner, dans la pratique il y a de nombreuses sources d'erreur qui peuvent faire échouer l'apprentissage.

Par exemple, lors de la propagation du gradient, de nombreuses multiplications informatiques (donc peut être avec des arrondies) sont effectuées.
Si on multiplie de trop petites quantités, il peut y avoir un "évanouissement du gradient"; tandis que si les quantités sont trop grandes, il peut y avoir une "explosion du gradient". Et ces deux situations peuvent se répercuter sur les futurs poids mis à jour.

C'est pouquoi le choix des fonctions d'activations σ est si importante. Elles permettent non seulement de passer du linéaire au non linéaire, mais également de ramener la valeur de l'output du neurone courant dans l'intervalle $[0;1]$ et, ainsi, diminuer les trop grands écarts de valeur (c'est comme une normalisation).

Il y a aussi deux autres hyper-paramètres dont il faut faire attention pour éviter une instabilité numérique. Ce sont deux paramètres de l'algorithme de la descente de gradient : le taux d'apprentissage $\eta \in [0;1]$ et l'initialisation des poids ($\mathbf{w}^0$,b^0).

Le taux d'apprentissage η adéquat dépend du problème étudié, il est donc parfois estimé par essais-erreurs, mais il est souvent proche de 0 (ex. $\eta = 0{,}1$ ou encore 0,01).

L'initialisation des poids assez répandue dans la pratique est l'initialisation de Xavier : Pour éviter que le signal à travers les couches cachées ne s'estompe ou ne s'amplifie de trop, nous voulons que la variance des données reste la même à chaque passage d'une couche neuronal.

C'est-à-dire,

$$Var(entrée) = Var(sortie),$$

donc,

$$Var(X) = Var(Y),$$

or,

$$Var(Y) = \frac{(n_{in} + n_{out})}{2}.Var(X).Var(W^0),$$

on obtient donc,

$$Var(W^0) = \frac{2}{(n_{in} + n_{out})},$$

Finalement, l'initialisation de Xavier consiste à initialiser les poids selon la loi

$$(\mathbf{w}^0,b^0) \sim \mathcal{N}(0,\frac{2}{(n_{in} + n_{out})}).$$

1.4 Sélection de modèle, Sur & Sous-apprentissage

L'objectif de l'apprentissage automatique est de trouver le meilleur modèle f (en général f est paramétrique de paramètre $\mathbf{w}$) qui simule ou prédit au mieux les données futurs.
Pour construire un tel modèle, on cherche à minimiser l'erreur de généralisation $\mathbb{E}_{gen}$ que fait notre modèle sur la prédiction des données de la population entière $\Omega \sim \mathbb{P}$,

$$\mathbb{E}_{gen} = \mathbb{E}_{\mathbb{P}}[l(Y,f(X))].$$

Cependant, comme nous n'avons pas accés à la totalité des données de la population. On va approximer l'erreur de généralisation par l'erreur sur une poignée de données d'entrainement (obtenues par la loi $\mathbb{P}_{train}$). Pour cela, on collecte le maximum de données possibles sur le problème traité, puis on calcul l'erreur de prédiction faite sur les données d'entrainement,

$$\mathbb{E}_{train} = \sum_{i \in train} l(Y_i,f(X_i)).$$

Bien sûr, pour approximer au mieux $\mathbb{E}_{gen}$, il faut que la loi $\mathbb{P}_{train}$ soit la plus proche possible de $\mathbb{P}$. Pour cela, il faut souvent un très grand nombre n de données (des données représentatives qui plus est !) et tester plusieurs modèle ou famille de modèle différents.

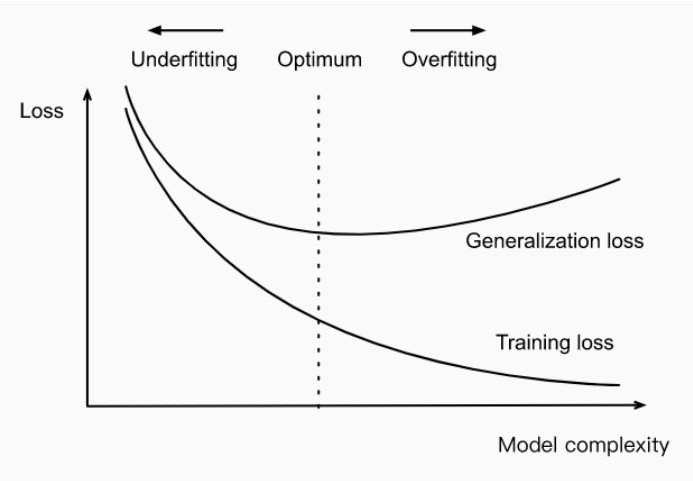
Ainsi, une démarche générale se dégage :

1. Train : Entraîner différents modèles sur le TrainSet;
2. Model Selection : Sélectionner le meilleur modèle sur le ValidationSet;
3. Test : Vérifier la généralisation du modèle sur le TestSet.

Malgré tout, il se peut que un modèle soit très performant durant l'entrainement, mais qu'il fasse une piètre performance au moment du test. Ce problème s'appelle le Sur-Apprentissage. Il survient souvent quand le modèle construit a une trop grande "complexité", ou que il y a trop peu de données d'entrainement.

La figure 3 shématise ces problèmes de sur-apprentissage et sous-apprentissage.

Figure 3 – Sous/Sur-apprentissage



1.5 Régularisation-L_2 (Weight Decay) & Dropout

Pour résoudre le problème de la sur-interprétation évoquée précédemment, il existe diverses méthodes telles que la Régularisation-L_2 et le Dropout.

La régularisation est une méthode générale qui s'applique à l'apprentissage automatique (pas que à l'apprentissage profond). Elle consiste a pénaliser les modèles trop complexes dans le processus d'apprentissage. Plus précisément, la régularisation revient à rajouter une pénalité $g(\mathbf{w},b)$ sur les valeurs des paramètres dans la fonction à minimiser,

$$(\mathbf{w}^*,b^*) = \arg\min_{\mathbf{w},b} \{L(\mathbf{w},b) + g(\mathbf{w},b)\}$$

Dans le cas de l'apprentissage profond, une régularisation souvent employée est la régularisation-L_2 (= weight decay) qui consiste à poser comme pénalité $g = \frac{\lambda}{2}\|.\|_2$. Ainsi, la nouvelle fonction objective à minimiser est :

$$(\mathbf{w}^*,b^*) = \arg\min_{\mathbf{w},b} \{L(\mathbf{w},b) + \frac{\lambda}{2}\|(\mathbf{w},b)\|_2\}$$

Pénaliser par la norme L_2 permet de garder une fonction différentiable (et donc de pouvoir calculer le gradient) et de pénaliser les poids trop grands en valeur absolue. Donc ça tend à les rapprocher de 0, voir même à les annuler, et donc à diminuer la complexité du modèle (ça rend le modèle plus "smooth" en supprimant les grandes variations) et à diminuer le phénomène de sur-apprentissage.

La seconde méthode de régularisation est l'Abandon de Neurone (= Dropout). Le Dropout consiste à ignorer aléatoirement certains neurones (et ses connexions) pendant chaque phase du processus d'apprentissage. Ainsi, on peut voir cette méthode comme l'entraînement en parallèle d'un grand nombre de réseaux neuronaux d'architecture différente.
Cet abandon de neurone a pour effet d'ajouter un "bruit" forçant les neurones au sein d'une couche à probabilistiquement prendre la responsabilité des inputs à la place des neurones abandonnés.

Pour implémenter le dropout, on introduit un nouvel hyperparamétre p qui spécifie la probabilité avec laquelle la sortie d'un neurone est abandonnée. Puis durant l'entraînement, à chaque étape de l'apprentissage, chaque neurone est ignorée avec probabilité p.

Il a été montré expérimentalement que la régularisation Dropout était une des plus efficaces et moins coûteuses. De plus, on peut combiner l'utilisation de plusieurs méthodes de régularisation pour diminuer au mieux le sur-apprentissage et, ainsi, optimiser l'efficacité du modèle.

1.6 Le problème du Distribution Shift

Durant l'entraînement du modèle nous faisons plus ou moins l'hypothèse que la distribution des données d'entraînement $\mathbb{P}_{train}(X,Y)$ est la même que la distribution cible $\mathbb{P}(X,Y)$ qui génère la totalité des données de la population Ω (en particulier les données futurs). Bien sûr, cela n'est jamais le cas dans la réalité car il faudrait toutes les données de la population Ω, mais on arrive quand même à s'en approcher suffisamment pour que l'apprentissage automatique soit possible.

Cependant, il existe de nombreux cas où l'apprentissage ne fonctionne pas car on observe un phénomène de Saut de Distribution (= Distribution Shift). Ce phénomène se traduit par le fait que la distribution des données d'entraînement

$\mathbb{P}_{train}(X,Y)$ est significativement différente de la distribution des données de test $\mathbb{P}_{test}(X,Y)$ (qui joue le rôle de la distribution cible $\mathbb{P}(X,Y)$).

Il existe trois types principaux de distribution shift : le covariate shift, le label shift et le concept shift.

Le plus répandu des trois est le covariate shift. Il se traduit par le fait que la distribution cible des features (= covariables) $p(X)$ change en une nouvelle distribution $q(X)$, tandis que la distribution cible des "labels sachant les features" $p(Y|X)$ ne change pas (càd. $q(Y|X) = p(Y|X)$). Ainsi, ce problème n'apparait que dans les cas où les features impliquent les labels ($X \implies Y$).
Par exemple dans le cas d'une application de reconnaissance d'image de chien et de chat, il se peut que les images soit d'abord des photos puis des dessins. Ainsi on a donc bien un changement des features des données (on passe de photo à dessin, ce qui peut drastiquement modifier les pixels, donc les features).
Une des méthodes pour résoudre ce type de problème est la suivante. On effectue l'apprentissage sur les données (X,Y) que l'on pondères par le coefficient $\beta_i = \frac{\hat{p(x_i)}}{q(x_i)}$ (où $\hat{p(x_i)}$ est estimé par une régression logistique), et la fonction objective à minimiser devient donc :

$$\min_f \{\frac{1}{n}\sum_{i=1}^{n} \beta_i.l(f(x_i),y_i)\}$$

Le label shift quant à lui signifie que la distribution cible des labels $p(Y)$ change en une nouvelle distribution $q(Y)$, tandis que la distribution cible des "features sachant les labels" $p(X|Y)$ ne change pas (càd. $q(X|Y) = p(X|Y)$).
Ainsi, ce problème n'apparait que dans les cas où les labels impliquent les features ($Y \implies X$).
Par exemple dans le cas d'une application de diagnostique de maladie, il se peut que la prévalence d'une certaine maladie y change de $p(y)$ à $q(y)$ sans que ses symptômes x ne changent ($q(x|y) = p(x|y)$).
Une bonne méthode pour résoudre ce type de problème est la même que pour le covariate shift, à la différence près que $\hat{p(x_i)}$ est estimé plus simplement en résolvant un simple système linéaire.

Pour ce qui est du concept shift, il apparaît quand la distribution cible des "labels sachant les features" $p(Y|X)$ change en $q(Y|X)$.
Par exemple dans le cas d'une application cherchant la définition des mots, il se peut qu'un mot x ait une signification différente d'un bout à l'autre d'un même pays ($y_1 \neq y_2$).

Pour ce type de problème, il faut remarquer que dans la plupart des cas il n'y a pas un changement abrupt de concept, mais plutôt un changement progressif qu'il nous est possible de suivre. Ainsi, pour résoudre ce problème il suffit souvent de mettre progressivement à jour les poids de l'ancien réseaux neuronal sur des nouvelles données bien labelisées (avec le nouveau label dû au concept shift).

1.7 Implémentation Python d'un MLP

Nous allons implémenter un MLP à 3 couches pour résoudre un problème de classification à partir d'un jeu de données de Semence de Blé. En fonction des caractéristiques de la semence (features), il faudra prédire l'espèce de la semence (label).

Pour décrire briévement le jeu de données, il comporte 201 individus décrits par 7 caractéristiques (donc 7 inputs), et le label à prédire comporte trois classes (donc 3 outputs pour de la 3-classification). Ajoutons que pour utiliser l'algorithme Back-prop, il faudra normaliser les données dans [0;1].

Le MLP que nous allons créer aura trois couches : la couche d'entrée (à 7 neurones), une couche cachée, et la couche de sortie (à 3 neurones).
Un neurone sera codé par un dictionnaire (pour pouvoir coder plus simplement ses propriétés : poids, fonction d'activation, etc.), une couche de neurones sera codé par un array de dictionnaires, et enfin, le réseau de neurone entier sera codé par une liste d'arrays.
De nombreuses fonctions seront implémentées (initialisation du réseau, activation des neurones, màj des poids, training, propagation avant/arrière, etc...), le code commanté est consultable en annexe (ou sur github).

Pour tester le réseau neuronal, nous allons utiliser la Validation Croisée à k-blocs. Dans notre code, pour un MLP à 3 couches avec 5 neurones cachées, un taux d'apprentissage de 0.3, un apprentissage sur 50 époques, et en testant avec une validation croisée à 5 blocs, nous obtenons une précision moyenne de 93.3%. C'est-à-dire que les prédictions de notre MLP sur les données de notre TestSet sont bonnes en moyenne dans 93.3% des cas.

2 Apprentissage par renforcement (RL)

Théorie : lecture les liens 3 à 11 de la prof (Teams).
Application : Implémentation rapide d'un RL.
Deadline : Environ fin Janvier.

3 Méthodes de DL pour le RL : RNN

Théorie : Lecture du chapitre 8 du livre "Dive Into Deep Learning".
Application : Implémentation rapide d'un RNN (lien 2).
Deadline : Environ fin Février.

4 Mise en application du DeepRL

Mise en application (de ce qui précède) sur un cas réel de problème de RL avec des méthodes de DL (peut être du NLP avec un algo du type Word2Vec !).
Deadline : jusqu'à la fin du projet long (environ mi-Mars).

Conclusion