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 des méthodes d'Apprentissage profond qui l'implémente. Puis nous finirons sur une mise en application de DeepRL.

1 Apprentissage profond (DL)

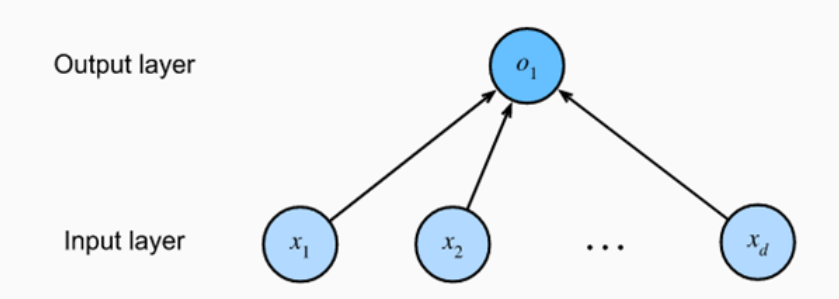
Dans cette section, nous allons étudier comment fonctionne l'apprentissage profond, quels sont les problèmes que l'on peut rencontrer lors de l'implémentation de ce type d'algorithmes, ainsi que quelques solutions.

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 représentée 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 pixels 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 :

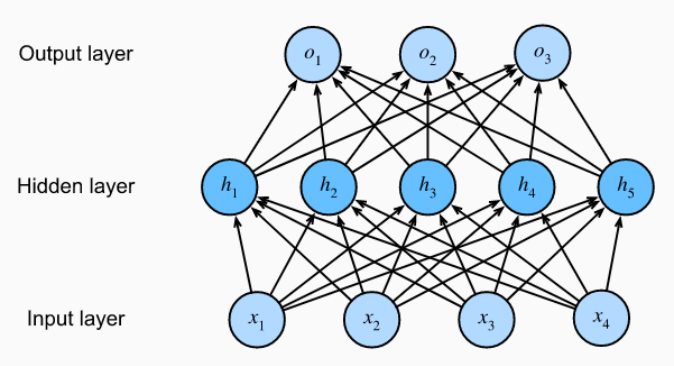
$$o_1 = b_1 + \sum_{i=1}^{d} \omega_{i,1} x_i.$$

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 problèmes sont non-linéaires. 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 neurones formels (perceptron) tous entièrement reliés entre eux. Un tel réseau neuronal est appelé Perceptron Multi-couche, et c'est le plus simple réseau 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 possédant 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 $x^k \in \mathbb{R}^d$ correspond à une nouvelle représentation de la donnée 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 et 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'entrée $X = (x_k)_{k=1,...,n} \in \mathbb{R}^{n\times d}$. Donc le jeu de données de sortie sera $O \in \mathbb{R}^{n\times q}$, déterminé par :

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

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 complexe. C'est pourquoi dans de nombreux cas d'autres architectures de Réseaux de neurones pourront être choisies car plus pratiques (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 réseau 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 prédire 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 paramétrage du réseau de neurones $(\mathbf{w},b)$ 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 :

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

Enfin, les valeurs optimales des poids se déterminent par :

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

Étant 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 réseau, il faut pouvoir propager une dérivée des outputs aux inputs. 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 minis-batch restants;
5. On peut refaire d'autres Epochs (= apprentissage sur tous les minis-batchs) pour s'approcher au mieux de la solution ($\mathbf{w}^*$,b^*).

Remarque : Les minis-batchs 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'erreurs 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 pourquoi le choix des fonctions d'activations σ est si important. 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 futures.
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 accès à la totalité des données de la population. On va approximer l'erreur de généralisation par l'erreur sur une poignée de données. Pour cela, on collecte le maximum de données possible sur le problème traité, puis on sépare cet ensemble de données en un ensemble d'entraînement (de loi $\mathbb{P}_{train}$) avec lequel on construira le modèle f, et en un ensemble de test (de loi $\mathbb{P}_{test}$) avec lequel on approximera l'erreur de généralisation,

$$\mathbb{E}_{gen} \approx \mathbb{E}_{test} = \frac{1}{n_{test}} \sum_{i \in test} l(Y_i,f(X_i)).$$

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

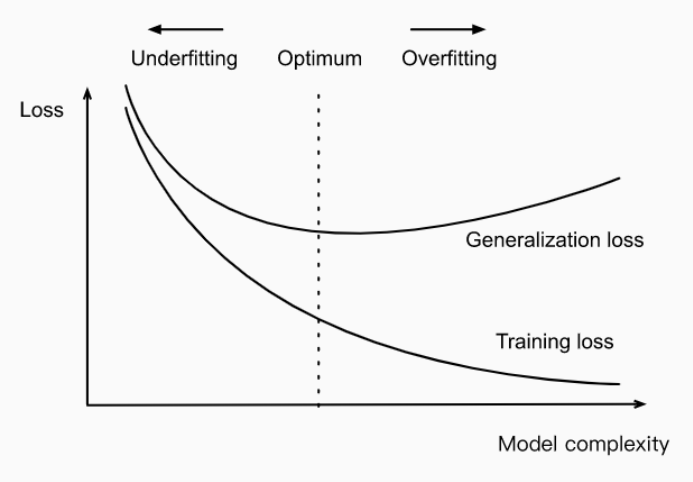
Ainsi, une démarche générale se dégage :

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

Malgré tout, il se peut qu'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 qu'il y a trop peu de données d'entrainement.

La figure 3 schématise 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 à 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, voire 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 ignoré avec probabilité p.

Il a été montré expérimentalement que la régularisation Dropout était une des méthodes les 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

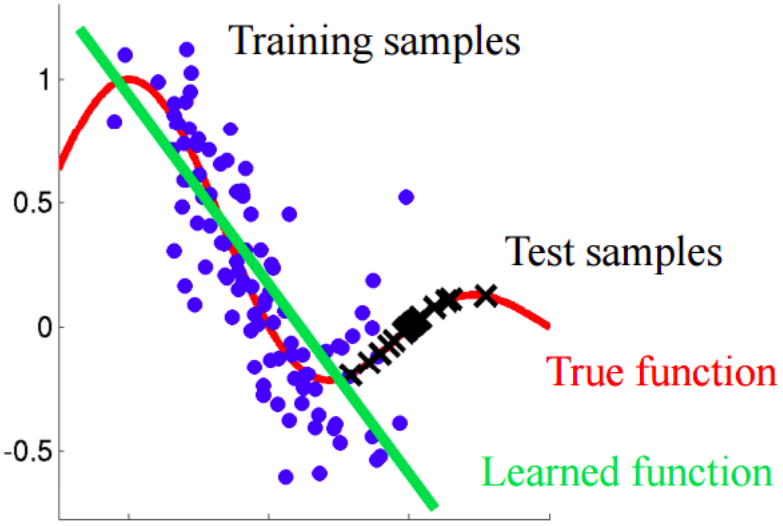
Durant l'entraînement du modèle nous faisons plus ou moins l'hypothèse que la distribution des données d'entraînement $\mathbb{P}_{train}(X,Y)$ est la même que la distribution cible $\mathbb{P}(X,Y)$ qui génère la totalité des données de la population Ω (en particulier les données futures). Bien sûr, 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. $q(Y|X) = p(Y|X)$).
La figure 4 présente un exemple de covariate shift :

Figure 4 – Covariate Shift



Pour éviter ce type de problème, il faut bien faire attention aux données que l'on manipule. Également, on peut essayer de mieux randomiser le tirage des données du trainset et du testset parmi les données du dataset.

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. $q(X|Y) = p(X|Y)$).
Par exemple dans le cas d'une application de diagnostic 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 ($q(x|y) = p(x|y)$).
Les préventions de ce type de problème sont les mêmes que celles du covariate shift.

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 réseau neuronal sur des nouvelles données bien labellisées (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 brièvement 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 codée par un array de dictionnaires, et enfin, le réseau neuronal 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 commenté 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 cachés, 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)

Cette section est consacrée à l'apprentissage par renforcement. Dans un premier temps nous allons définir ce contexte d'apprentissage, puis nous allons voir les principales méthodes de résolution.

2.1 Processus Décisionnel Markovien

Un problème d'apprentissage par renforcement peut être modélisé par un Processus Décisionnel Markovien (PDM) particulier.

Un PDM est un modèle stochastique où un système se trouve dans un certain état et dans lequel un agent peut effectuer certaines actions pour que le système puisse changer d'état. Ce modèle est stochastique car les choix des actions et les transitions d'états suivent des probabilités. De plus, des récompenses sont associées aux couples état-action choisis.

Plus précisément, un PDM est un processus stochastique satisfaisant la propriété de Markov et défini par un quintuplet du type :

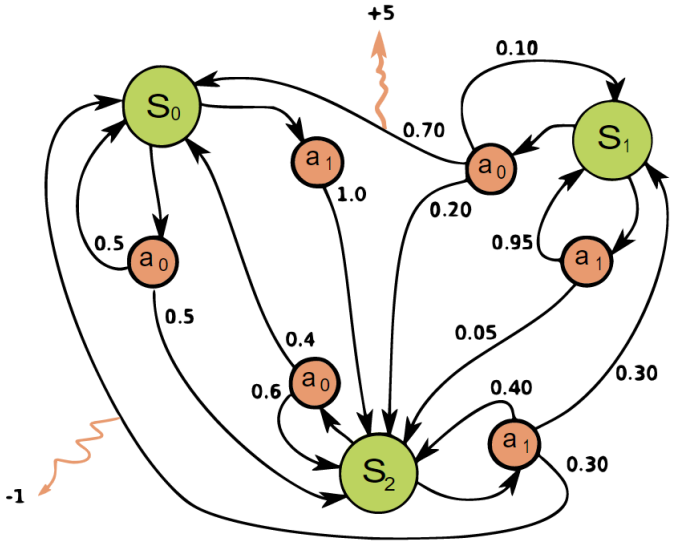
$$(S, A, T, p, r),$$

où :

- S est l'espace d'états dans lequel évolue le processus ;
- A est l'espace des actions que le système peut prendre et qui contrôle la dynamique de l'état ;
- T est l'axe temporel du processus (car contrairement aux autres contextes d'apprentissage, le RL s'effectue dans le temps), souvent $T = \mathbb{N}$;
- p représente les probabilités de transition, par exemple $p(s'|s,a)$ est la probabilité que le système passe dans l'état s' après avoir exécuté l'action a dans l'état s ;
- r est la fonction de récompense sur les transitions entre états, $r(s,a) \in \mathbb{R}$.

La figure 5 donne un exemple de PDM avec 3 états (S_0, S_1 et S_2), les deux mêmes actions par état (a_0 et a_1) et 2 récompenses (+5 et -1) :

Figure 5 – Exemple de PDM



En particulier, l'apprentissage par renforcement est un Processus Décisionnel Markovien dont on ne connaît pas à priori les probabilités de transition $p()$ et les récompenses $r()$. De ce fait, un des enjeux de ce type de problème est de réussir à découvrir/estimer ces deux fonctions par essais-erreurs en interagissant avec l'environnement du système.

Notons également que l'apprentissage par renforcement est de l'apprentissage non supervisé. Car les données auxquelles l'agent a accés ne sont pas labelisées et donc il doit identifier par lui-même la meilleure réponse possible. Cependant, les données possèdent une récompense qui permet une évaluation par l'agent. Ce qui amène donc à un mode de fonctionnement par essais et erreurs pour chercher la meilleure récompense possible.

De plus, l'apprentissage par renforcement porte sur des séquences temporelles. En effet, lorsque l'on cherche à faire de la classification (par exemple), le temps importe peu car on peut présenter les données d'entraînement dans l'ordre que l'on veut. Tandis qu'en apprentissage par renforcement, tout choix d'une action exécutée dans un état peut avoir des conséquences à plus ou moins long terme et donc la donnée de la récompense immédiate n'est généralement pas suffisante (il faudrait y ajouter toutes les récompenses suivantes).

2.2 Politiques et fonctions de valeur

La notion de PDM étant définie, voyons maintenant comment l'agent choisit ses actions à effectuer.

À chaque état $s \in S$, l'agent doit donc choisir une action $a \in A$ (ou A_s si les actions possibles dépendent de l'état dans lequel se trouve le système). Pour faire ce choix, l'agent suit une politique (notée π) qui est une fonction qui à tout état $s \in S$ renvoie l'action à effectuer (ça peut aussi être une politique aléatoire qui, à tout couple (s,a), renvoie la probabilité que a soit l'action à prendre).
Notons que, comme un PDM est markovien, la politique π est également markovienne (càd qu'il suffit de connaître l'état courant s pour déterminer l'action a, pas besoin du passé du processus avant s).
Ainsi, la politique est de la forme $\pi : s \in S \rightarrow \pi(s) \in A$.

L'objectif est donc de trouver la politique π qui convient le mieux selon le problème rencontré.

Pour mesurer la performance d'une politique, on utilise un critère de performance. Il en existe plusieurs, mais le plus répandu est le critère γ-pondéré qui calcule la moyenne de la somme pondéré des récompenses obtenues en partant de l'état s_0 et en suivant la politique π :

$$\mathbb{E}_\pi[r_0 + \gamma r_1 + \gamma^2 r_2 + ... + \gamma^t r_t + ...|s_0],$$

avec $0 < \gamma < 1$ appelé "facteur d'actualisation" et qui permet de donner plus d'importance aux récompenses les plus proches.

Dans la suite, nous utiliserons toujours le critère de performance γ-pondéré.
Une fois le critère de performance choisi, on définit les fonctions de valeur V et Q qui permettent d'avoir une relation d'ordre entre les politiques.

V^π est la fonction de valeur d'état associée à la politique π qui, à tout état s associe la valeur de performance du critère considéré en suivant π à partir de s :

$$V^\pi(s) = \mathbb{E}_\pi[\sum_{t \in T} \gamma^t r_t|s].$$

Cepdendant étudier la récompense totale avec comme unique information que l'on part de l'état s n'est pas suffisant, car on ne sait pas quelle est l'action (suivant l'état s) qui donne la meilleure performance, et donc il n'est pas possible de déterminer la meilleure politique.

D'où la nécessité de la fonction de valeur d'état-action Q^π qui associe à tout état s et l'action a, la valeur de performance du critère considéré du processus

partant de s, exécutant l'action a, puis suivant la politique π par la suite :

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{t \in T} \gamma^{t} r_t | s, a].$$

Ainsi, avec ces fonctions nous pouvons comparer des politiques entre elles et, par conséquent, trouver la politique optimale π^* qui est telle que $\forall \pi, Q^{\pi} \leqslant Q^{\pi^*}$ (notée aussi Q^*).

Pour finir, notons que trouver la politique optimale π^* est équivalent à trouver la fonction de valeur d'état-action optimale Q^*.
En effet, si on a Q^*, on déduit π^* en prenant à chaque état s l'action a qui a le plus grand $Q^*(s,a)$.

2.3 Méthode de différence temporelle

Comme on l'a étudié précédemment, l'agent prend itérativement des décisions, change d'état et gagne des récompenses, il suit donc une trajectoire du type :

$$(s_0, a_0, r_0; s_1, a_1, r_1; ...; s_t, a_t, r_t; ...).$$

Comment exploiter ces "données d'entraînement" pour déterminer une bonne politique π ?

Pour cela, nous nous aidons de l'équation de Bellman qui permet (dans le cas du RL) d'écrire la fonction de valeur au temps t de deux façons différentes :

$$Q(s_t,a_t) = \mathbb{E}[r_t + \gamma Q(s_{t+1}, a_{t+1})],$$

où γ est le facteur d'actualisation, r_t est la récompense immédiate reçue de l'action a_t (récompense que l'on peut observer), $Q(s_t,a_t)$ est l'estimation du critère de performance sur la totalité du processus à partir du temps t et, de même, $Q(s_{t+1}, a_{t+1})$ est l'estimation du critère de performance sur la totalité du processus à partir du temps $(t+1)$.

Nous pouvons donc estimer de deux façons différentes la même quantité, et donc obtenir l'erreur d'estimation suivante (nommée erreur de différence temporelle) :

$$\delta_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t,a_t).$$

Finalement, nous pouvons mettre à jour les estimations des coefficients de Q le long de la trajectoire d'interaction agent/environnement grâce à cette formule de mise à jour :

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t * \delta_t$$

càd,

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t * [r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t,a_t)],$$

où α_t est le taux d'apprentissage à l'instant t.

Cependant, comme a_{t+1} ne nous est pas connu à l'instant t, la valeur $Q(s_{t+1}, a_{t+1})$ nous est également inconnue. Il faut donc la remplacer par une valeur connue et utile (par ex. l'algorithme Q-Learning la remplace par $\{\max_a[Q(s_{t+1}, a)]\}$ qu'il possède en mémoire).

2.4 Le dilemme exploitation/exploration

Au cours de l'apprentissage, pour régler la politique π (ou ce qui est équivalent, la fonction de valeur Q) de façon à maximiser la récompense sur le long terme, la phase d'apprentissage se trouve confrontée à choisir entre l'exploitation, qui consiste à refaire les actions dont on connaît déjà la récompense, ou l'exploration, qui consiste à parcourir de nouveaux couples (état, action) à la recherche d'une récompense cumulée plus grande, mais au risque d'adopter parfois des actions sous-optimales. Car tant que l'agent n'a pas exploré toutes les possibilités, il peut n'avoir accès qu'à un maximum local (et non global).

Dans le cas où on choisit l'action optimale courante à tout instant t (c'est-à-dire celle qui rapporte la plus grande récompense immédiate), cela peut amener à une récompense globale sous-optimale ou encore à une divergence du temps de calcul de l'algorithme. Cette action est appelée action gloutonne (resp, politique gloutonne).

Une solution à ce dilemme est de choisir de suivre la politique optimale courante dans la majorité des cas (exploitation) et de choisir aléatoirement l'action dans la minorité restante des cas (exploration). Comme par ex. l'algorithme *ϵ-greedy* qui choisit de suivre la meilleure politique connue avec probabilité $(1 - \epsilon)$ ou tire uniformément l'action dans A avec probabilité ϵ, avec $\epsilon \in [0; 1]$ proche de 0.

3 Implémentation et DeepRL

La formule de mise à jour de la partie 2.3 est à la base d'un grand nombre d'algorithmes d'apprentissage par renforcement. Citons par exemple l'algorithme SARSA (State Action Reward State Action), l'algorithme TD (Temporal Difference) ou encore l'algorithme Q-Learning (qui se base sur l'optimisation de la fonction de valeur d'état-action).

Nous allons étudier l'implémentation de l'algorithme Q-Learning sur un exemple basique et répandu de problème d'apprentissage par renforcement : le problème du pendule inversé.

L'implémentation se fera en Python avec TensorFlow, et elle reprendra l'exemple donné dans un GitHub, accessible ici, qui permet d'implémenter rapidement tous les composants de l'exemple (l'environnement, l'agent, etc.).

L'algorithme Q-Leraning consiste à approximer la fonction de valeur d'état-action optimale Q^* en s'appuyant sur cette formule de mise à jour :

$$Q_{t+1}(s,a) \leftarrow \mathbb{E}[r + \gamma \max_{a \in A} Q_t(s', a)].$$

Le problème du pendule inversé consiste à faire en sorte qu'un pendule inversé en mouvement sur un segment horizontal réussisse à garder sa barre droite.
Pour cela, l'environnement est décrit par 4 valeurs : la position et la vitesse du pendule; l'angle et la vitesse angulaire du pendule. L'agent est le pendule lui-même, et il a accès à deux actions : aller à gauche (codé par la valeur 0) et aller à droite (valeur 1). Pour finr, un épisode de vie du système se termine si le pendule dépasse un certain angle avec la verticale (il commence à tomber), si il sort des limites du segment ou si l'épisode dépasse 200 étapes.

Voici un exemple d'état du système : [0.02774083, -0.00539378, 0.00493613, 0.0424618]. Étant donné que les états sont trop nombreux (les valeurs doivent être précises pour que l'apprentissage soit efficace, en plus il y a 4 valeurs par état), la fonction de valeur $Q(s,a)$ aura beaucoup trop de composantes pour pouvoir tous les mettres à jour un par un. C'est pourquoi il est préférable de plutôt chercher à approximer Q^* par une fonction paramétrique $Q(s,a;\theta)$ qui sera déterminée avec un réseau de neurones.

Cette méthode s'appelle Deep Q-Learning. Elle consiste à utiliser un réseau de neurones qui minimise la fonction de perte suivante :

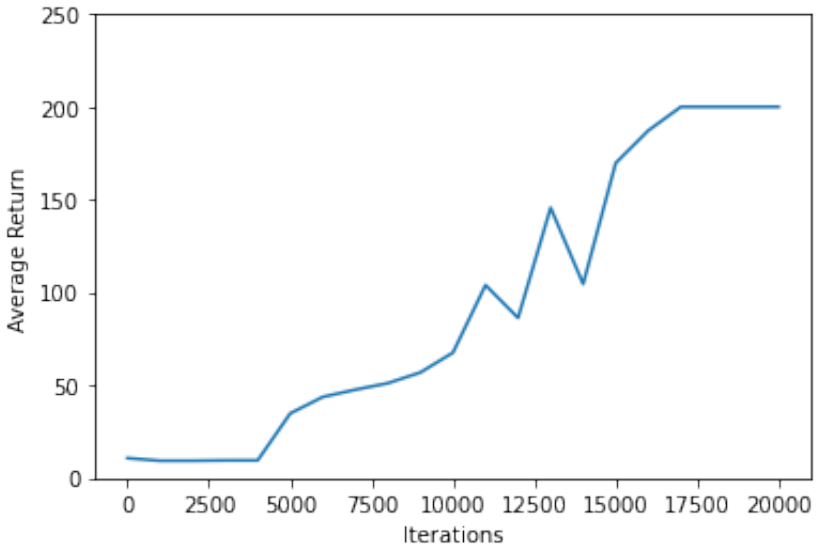
$$L(\theta_t) = \mathbb{E}[(y_t - Q(s,a;\theta_t))^2], \text{ où } y_t = (r + \gamma \max_{a \in A} Q(s', a; \theta_t)).$$

La méthode ϵ-*greedy* est appliquée pour avoir un bon ration eploitation/exploration.

Après entraînement (qui dure plusieurs minutes) sur 20 000 épisodes de aux plus 200 étapes, on obtient une approximation des composantes $Q^*(s,a), \forall a,s$. Ensuite, pour obtenir l'approximation de la politique optimale π^*, il suffit d'utiliser : $\forall s$, $\pi^*(s) = \max_a Q^*(s,a)$.

Pour visualiser graphiquement l'apprentissage, la figure 6 montre, pour chacun des 20 000 épisodes, le nombre d'étapes consécutives où le pendule réussit à respecter l'objectif. On observe effectivement que la courbe croît, ce qui prouve que l'agent améliore sa politique d'action.

Figure 6 – Évolution de la durée de vie du pendule



Conclusion

Au cours de ce projet nous avons étudié d'une part l'apprentissage profond et d'autre part l'apprentissage par renforcement.

Concernant l'apprentissage profond, nous sommes d'abord partis de la présentation du neurone formel pour ensuite aller à la description du MLP. Nous avons décrit le fonctionnement de son apprentissage (rétro-propagation, descente de gradient stochastique, etc.), ainsi que les différents problèmes que l'on pouvait rencontrer (qui ne sont pas propres à l'apprentissage profond mais peuvent avoir lieu dans tous les domaines de l'apprentissage automatique) tels que le sous/sur-apprentissage, la mauvaise qualité des données d'entraînement ou encore le distribution shift.
Puis nous avons achevé cette partie par l'implémentation Python d'un petit réseau de neurones sur un problème de classification.

Est ensuite venue l'étude de l'apprentissage par renforcement. Nous nous sommes d'abord attardés sur la définition de ce contexte d'apprentissage (à l'aide de la notion de processus décisionnel de Markov), puis nous avons vu la méthode de différence temporelle sur laquelle beaucoup d'algorithmes se basent.

Enfin, nous avons étudié un exemple d'implémentation de problème d'apprentissage par renforcement qui utilise l'algorithme Q-Learning amélioré avec l'utilisation d'un réseau de neurones.

References

[1] Dive Into Deep Learning - Chapitres 1, 2, 3 et 4
https://d2l.ai/

[2] Difference between Batch and Epoch in a Neural Network
https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

[3] Weight Initialization in Neural Networks
https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79

[4] Understanding Xavier Initialization In Deep Neural Networks
https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/

[5] Understanding Dataset Shift
https://towardsdatascience.com/understanding-dataset-shift-f2a5a262a766

[6] Covariate Shift
https://www.analyticsvidhya.com/blog/2017/07/covariate-shift-the-hidden-problem-of-real-world-data-science/

[7] How to Code a Neural Network with Backpropagation In Python (from scratch)
https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

[8] Reinforcement Learning Made Simple - Parties 1, 2, 3, 4 et 5
https://towardsdatascience.com/reinforcement-learning-made-simple-part-1-intro-to-basic-concepts-and-terminology-1d2a87aa060

[9] Reinforcement Learning - L42Project
https://www.youtube.com/playlist?list=PLALfJegMRGp3_H_eiOoxqkCEKnSsAf9cg

[10] PDM en IA - Chapitres 1 & 2
http://researchers.lille.inria.fr/munos/papers/files/bouquinPDMIA.pdf

[11] Introduction to RL and Deep Q Networks
https://www.tensorflow.org/agents/tutorials/0_intro_rl