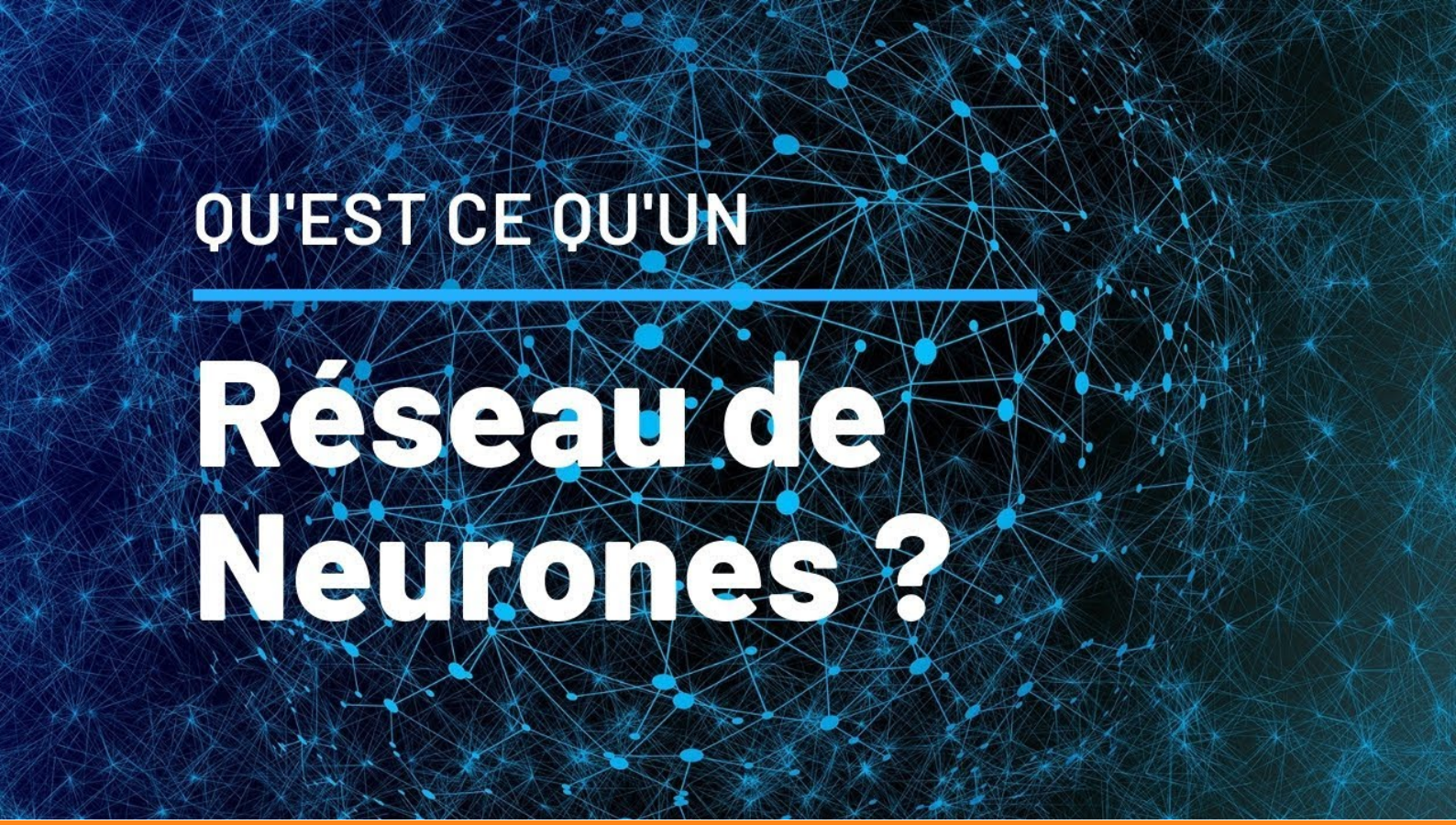

Réseaux de Neurones Artificiels : Exemples d'applications industrielles

Rapport ISDS2

Author : Cyprien Ferraris

Institute : ISUP

Date : 19 avril 2020

Version : 1

What we want is a machine that can learn from experience — A. Turing

Table des matières

Chapitre Présentation des réseaux de neurones artificiels

Les réseaux de neurones artificiels sont aujourd'hui une des méthodes les plus en vogue dans le domaine de l'Intelligence Artificielle. L'idée originelle est de reproduire le fonctionnement du cerveau humain. De fait sa construction est assez complexe comme on le verra par la suite. En échange de ce "coût", elle présente l'avantage de pouvoir approximer n'importe quelle fonction mesurable Hornik et al. (1989).

Grâce à cette propriété, les réseaux de neurones sont utilisés dans de nombreux domaines que ce soit dans des problèmes de régression (par exemple : prédiction de la puissance de production d'une centrale électrique) ou de classification (prédiction du nombre de semaines avant la panne d'un composant en fiabilité).

Comme son nom l'indique, un réseau de neurones est composé de plusieurs neurones artificiels reliés entre eux. On va donc commencer par regarder le fonctionnement d'un neurone artificiel.

1.1 Le neurone artificiel

Un neurone artificiel est composé d'une fonction d'entrée (notons la h) et d'une fonction d'activation (a). Comme son nom l'indique, la fonction d'entrée prend comme argument les entrées du neurones. Elle est souvent linéaire ou bien affine.

La fonction d'activation quant à elle prend en argument la sortie de la fonction d'entrée et se comporte à la manière des neurones biologique : si la réponse à l'entrée statisfait un certain critère, alors la fonction d'activation renvoie un signal, ce qui correspond à la sortie du neurones. Sinon, la sortie est nulle et le neurone est "désactivé". Le choix de la fonction d'activation est très important et sera discuté dans une autre section.

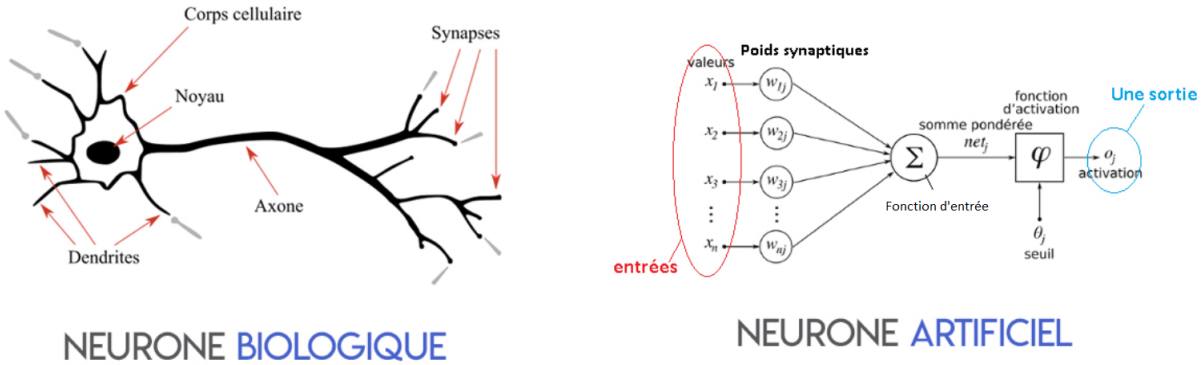

Figure 1.1 : Représentation d'un neurone (deeplylearning.fr)

On voit très clairement l'inspiration du neurone biologique dans cette représentation. Un neurone biologique reçoit les signaux transmis par les autres neurones via ses dendrites. Puis le corps cellulaire du neurone traite ces signaux. Si le résultat obtenu est superieur à un seuil d'excitabilité, alors il envoie un potentiel d'action le long de l'axone et est transmis à d'autres neurones via les synapses.

1.2 Fonctionnement d'un réseau de neurones

D'un point de vue mathématique, un réseau de neurone s'écrit sous la forme d'un graphe orienté pondéré.

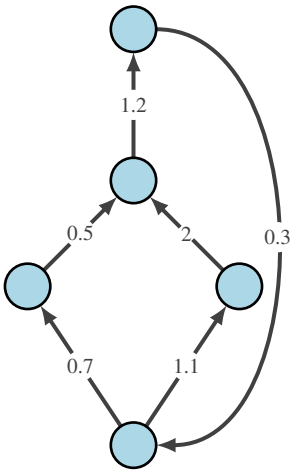

Figure 1.2 : Exemple de graphe

Afin d'utiliser un réseaux de neurones dans un but prédictif, il faut d'abord que le réseau apprenne la structure liant les données à prédire aux régresseurs.

Pour cela, on se place dans un contexte d'apprentissage supervisé, c'est-à-dire que l'on dispose d'observations $(X_i, Y_i)_{1 \leq i \leq n}$ qui vérifie $Y_i = f(X_i)$ et l'on souhaite construire une estimation de f. La phase d'apprentissage du réseau consiste donc à estimer les poids des arêtes entre les neurones de sorte que si l'on note $\hat{f}$ la sortie du réseau, on ait $Y_i \approx \hat{f}(X_i)$.

Une fois cette phase d'apprentissage réalisée, on peut étudier les propriétés de généralisation du modèle, ie sa capacité à prédire juste de nouvelles observations.

On peut aussi voir un réseau de neurones comme une modélisation d'un système dynamique. On est alors dans un cas où les poids sont connus et l'on souhaite étudier l'évolution des trajectoires. Dans la suite on se concentrera sur l'utilisation statistique des réseaux de neurones.

1.3 Bref historique

Bien que l'âge dort actuel des réseaux de neurones (aussi appelé Deep Learning et parfois par abus de langage intelligence artificielle) soit assez récent, cette technique reste relativement ancienne.

En effet, si l'on considère la machine de Turing (1940), comme le premier ordinateur, le neurone artificiel a été inventé peu de temps après (1943) par McCulloh et Pitts Strayer (1943). Leur but étant de représenter de manière mathématique le fonctionnement d'un neurone. Finalement, les premiers réseaux sont dus à Von Neuman en 1948 et sont suivis une année plus tard par les premières règles d'apprentissage Hebb (1949). En 1958, Rosenblatt propose comme réseau de neurone le Perceptron Rosenblatt (1958). Avec le modèle de l'époque, ce réseau est capable d'estimer n'importe quelle fonction continue ce qui en fait une méthode très interessante. On rentre alors dans une phase de grand intérêt pour les réseaux de neurones.

Cependant, on se rend compte en 1969 que le perceptron a des limites. Minsky and Papert (1969) montrent qu'il n'est pas capable d'apprendre des fonctions comme le "ou exclusif". On entre alors en quelque sorte dans une période de vide pour ces méthodes. Malgré cette période de vide de nombreuses architectures particulières de réseaux de neurnones (choix des poids non nuls dans le réseaux) apparaissent. On peut notamment citer le Perceptron multicouche (PMC) Rumelhart et al. (1986) qui consiste à combiner plusieurs perceptron entre eux (1986), l'architecture Long Short Time Memory (1997, très adapté aux données temporelles) Hochreiter and Schmidhuber (1997) ou encore l'architecture convolutive comme on la connait aujourd'hui (2003) Matsugu et al. (2003). Cependant ces méthodes sont très couteuse en temps de calcul et en données, ce qui avec les performances informatiques de l'époque rend leur utilisation très limitée. De plus dans les années 90, se développe les Machines à vecteurs de support (avec ressort) qui on les mêmes capacités d'apprentissage que le PMC mais qui sont plus simple à mettre en place et pour lesquelles on dispose d'un meilleur bagage théorique.

Cependant les capacité de calculs grandissent et en 2009, l'idée d'utiliser des GPU (processeurs graphiques) pour accélérer les calculs commence à être mise en place, ce qui couplé avec l'apparition des grandes masses de données permet en 2010 le développement du Deep Learning, c'est à dire à l'utilisation de grand réseaux de neurones (PMC avec beaucoup de couches et de neurones au départ), puis généralisation à d'autres architectures.
Enfin, on peut noter en 2014 le développement d'une architecture novatrice : les GAN (réseaux antagonistes génératifs) Goodfellow et al. (2014) qui consiste à mettre en compétition deux réseaux de neurones, l'un ayant pour but d'apprendre à simuler des données selon une loi de probabilité complexe et inconnue (typiquement la simulation d'image) et le deuxième apprenant à reconnaitre si une image est simulée ou si elle est issue du jeu de données d'entrainement.

1.4 Conception d'un réseau de neurones

Jusqu'ici on a regardé les réseaux de neurone en considérant comme connu leur architecture ainsi que le choix des diverses fonctions utilisées. Or les performances des réseaux sont très dépendantes de ces choix. Par exemple choisir un réseau trop compliqué peut, en plus du problème de temps de calcul, conduire à une situation de sur-apprentissage (mauvaise capacités

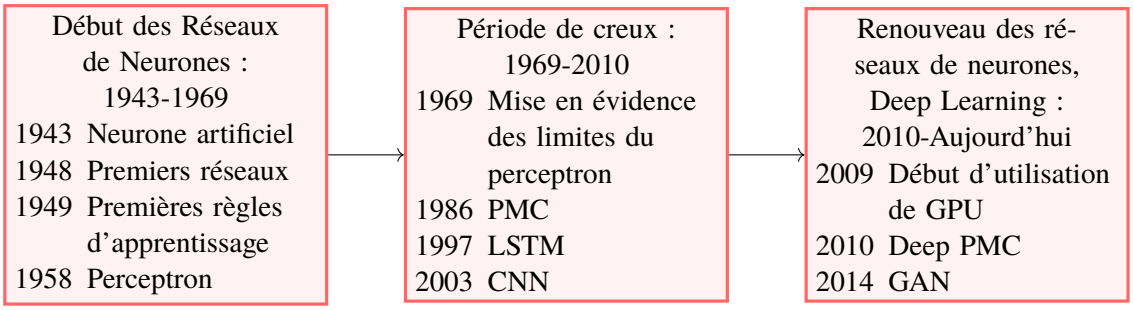

Figure 1.3 : Historique des Réseaux de neurones

à généraliser).

Dans cette section on va donc discuter du choix de l'architecture du réseau ainsi que le choix des fonctions d'activation et des fonction de perte qui permettent de mesurer les capacités du réseau.

A ma connaissance il n'existe pas de véritables règles théoriques pour la construction des réseaux de neurones. Je me contenterai donc de présenter les règles empiriques les plus utilisées.

1.4.1 Architecture du réseau

On ne parlera ici que des réseaux de types perceptron multicouche. L'architecture LSTM sera développée plus loin et les réseaux convolutifs ainsi que les GAN ne seront pas traités ici.

Le perceptron multicouche est un réseau composé de plusieurs couches de neurones accollées les unes aux autres. Les neurones de deux couches adjacentes sont tous reliés entre eux.

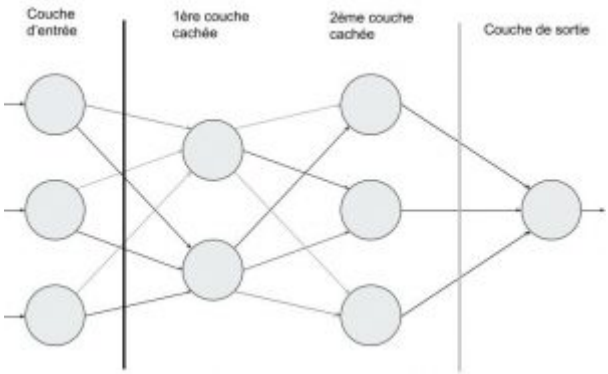

Figure 1.4 : Perceptron multicouche (penseeartificielle.fr)

Pour un réseau efficace, il ne suffit pas de rajouter des neurones à tout va. En effet on observe très clairement qu'il est en général meilleur de privilégier beaucoup de couche avec peu de neurones (augmenter la profondeur du réseau) plutôt que de construire des réseaux avec peu de couches mais beaucoup de neurones par couche (augmenter la taille des couches).

Dans le cas d'un PMC à une couche cachée on retrouve 3 "règles" pour décider de la taille de la couche cachée :

égale à la taille de la couche dentrée (Wierenga et Kluytmans, 1994)

égale à 75% de celle-ci (Venugopal et Baets, 1994)

égale à la racine carrée du produit du nombre de neurones dans la couche dentrée et de sortie (Shepard, 1990).

Dans le cas de petites applications (jeu de données pas trop importants, pas beaucoup de neurones), il est possible de choisir la taille du réseau par validation croisée. C'est à dire en selectionnant des échantillons du jeu de données sur lesquels tester les capacités du réseaux, puis on choisit celui qui a les meilleurs performances. Cependant cela n'est pas possible pour de grosses applications. Dans ce cas, pour éviter le sur-apprentissage on ajoute des phases de "dropout". Elles consistent à supprimer certaines connexion entre neurones. On parle aussi de neurochirurgien optimal. A nouveau déterminer la taille des dropout n'est pas évidente. Cette phase peut se faire de façon aléatoire mais aussi en fonction de l'évolution des poids du réseau.

De plus, le choix des fonctions d'activation (souvent la même au moins pour chaque couche du réseau) est très important. Elle a longtemps été linéaire. Cependant ce type d'activation limite les performances du réseau, en effet les fonction apprises sont alors limiter à un sous ensemble des fonctions mesurables mais on l'avantage de faciliter l'apprentissage des poids. Il en existe beaucoup, mais parmis les plus utilisée, on peut citer :

- la sigmoïde exponentielle $\frac{1}{1+e^{-x}}$
- la fonction tangente hyperbolique
- la fonction softmax dans le cas d'un vecteur de sortie z : $f(z_i) = \frac{e^{z_i}}{\sum_j e^{-z_j}}$
- la fonction ReLu (rectified linear unit) : $max(0, x)$

Récement on peut aussi citer une nouvelle fonction d'activation nommée SELU (scaled exponential linear units) :

$$SELU(x) = \lambda \begin{cases} x & si\ x > 0 \\ \alpha e^x - \alpha & sinon \end{cases}$$

Elle présente la particularité de conserver la normalisation lors de la propagation Klambauer et al. (2017).

Usuellement, la fonction d'activation vérifie les propriétés suivantes :

monotonie : elle est généralement croissante

seuillage : elle dépend d'un seuil θ tel que $x < \theta \Rightarrow f(x) = 0$

saturation : elle dépend d'un seuil M tel que $f(x) > M \Rightarrow f(x) = M$

dérivabilité : elle est dérivable pour permettre de bonne propriété lors de l'apprentissage

En pratique on voit que ce n'est pas toujour le cas (par exemple la fonction ReLu n'est pas dérivable en 0) cependant il a été montré Glorot et al. (2010) qu'elle permet un meilleur entraînement des réseaux.

Devant la complexité de choisir une architecture efficace et le cas échéant de réaliser la phase d'apprentissage de tâches difficiles (par exemples détecter ce qu'il y a sur une image) il est nécessaire d'avoir un réseau avec plusieurs millions de paramètres. Entraîner ce type de réseau n'est pas toujours possible. Pour résoudre ce problème deux méthodologies ressortent.

La première consiste à utiliser un réseau existant déjà entrainé, comme les meilleurs réseaux de diverses compétitions sur internet. Cependant, ces réseaux ne sont pas toujours adaptés au problème que l'on a. Une solution est alors de couper la poire en deux plutôt que d'utiliser un réseau déjà entrainer directement, on va supprimer les dernières couches du réseau et en rajouter de nouvelles qui vont permettre de s'adapter à notre problème. On parle alors de Transfer-Learning. C'est une méthode devenue assez utilisée ces dernières années et qui montre de bon résultats.

1.4.2 Choix de la fonction de perte

En statistique supervisée, on utilise ce que l'on appelle une fonction de perte pour mesurer la performance d'un méthode d'estimation.

L'apprentissage des poids du réseau dépend donc du choix de la fonction de perte. Tout comme la fonction d'activation, il en existe un grand nombre.

On distingue ici deux type : le cas de régression (la sortie est continue) et le cas de classification (la sortie est dénombrable).

Pour un exemple de régression la plus courante est l'erreur moyenne quadratique $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$. Où y_i est la vraie valeur et $\hat{y}_i$ est celle prédite.

Pour de la classification, si on considère k classes et que l'on note y_{ij} la vraie valeur de la classe de l'observation i ($y_{ij} = 1$ si l'observation i appartient à la classe j et 0 sinon) et de même on note $\hat{y}_{ij}$ la valeur prédite par l'algorithme. On peut alors choisir comme fonction de perte l'entropie croisée (catégorielle) : $-\sum_{j=1}^{k}\sum_{i=1}^{n}\left(y_{ij}\ log(\hat{y}_{ij})\right)$

Remarque : En réseau de neurone on manipule souvent des vecteurs et des matrices. De fait, il est courant que l'on note [1,0] lorsqu'une donnée est dans la classe 1 et [0,1] lorsqu'elle appartient à la classe 2. De fait, dans keras (cf plus loin) la fonction codant l'entropie croisée est sous cette forme. Celle correspondant à l'encodage [1],[2] se nomme alors "sparse categorical entropy".

Ainsi la phase d'apprentissage consiste à déterminer les poids qui minimisent la fonction de perte.

1.4.3 Apprentissage

Regardons maintenant plus en détatil le fonction de l'apprentissage des poids du réseau de neurone.

Commençons par regarder le cas d'un neurone. La fonction d'entrée du neurone i à l'instant "t" est de la forme

$$h_t(i) = \sum_{j=1}^{m} w_{ij} a_j(t)$$

Avec a_j la sortie du neurone j et $W = (w_{ij})$ la matrice des poids à l'instant t (W dépend de t mais, sauf en cas de confusion, on ne l'écrira pas pour ne pas alourdir les notations). On note

$h_t = (h_t(1), \ldots, h_t(m))^t$ et $a(t) = (a_1(t), \ldots, a_m(t))^t$. En considérant que la fonction d'activation f est la même partout, on a comme sortie du neurone i à l'instant t+1,

$$\begin{aligned} a(t+1) &= f(h_t) \\ &= f(Wa(t)) \end{aligned}$$

On peut alors écrire pour chaque neurone i l'équation : $a(t+1) = a(t) + g(h_t)$, en posant $g(x) = f(x) - a(t)$. Ce qui se réécrit aussi sous la forme :

$$\frac{dx}{dt} = l(x, W)$$

Si l'on souhaite maintenant que nos poids minimisent une fonction de perte L, l'utilisation de résultat sur l'étude de la stabilité des systèmes dynamiques continue et plus particulièrement de la fonction de Lyapunov nous permet de dire que :

$$\frac{\partial w_{ij}}{\partial t} = -\frac{\partial L}{\partial w_{ij}}$$

Finalement on peut introduire un pas d'apprentissage λ qui correspond à une approximation discrete des systèmes continus, et on obtient :

$$W_{t+1} = W_t = -\lambda \frac{\partial L}{\partial W}$$

On se retrouve alors avec une mise à jour des poids selon un algorithme de type descente de gradient. On converge donc vers un minimum local de la fonction L. Si de plus la fonction L est convexe et que son minimum est unique, on converge vers le minimum global. On connait de nombreux résultats dans le cas d'optimisation des fonctions convexes. Le fait que l'approche par système dynamique et celle par analyse convexe coïncident permet de profiter de nombreuses propriétés notament une généralisation dans le cas ou la fonction à optimiser n'est pas différentiable partout.

Néanmoins, en pratique L n'est pas toujours convexe. Cepedant on observe que les méthodes d'optimisation convexe offrent tout de même de bonnes performances.

Il existe d'autres variantes plus performantes de la descente de gradient, notamment dans le cas de réseaux profonds comme la descente de gradient stochastique. De plus, la détermination du pas d'apprentissage est importante pour avoir un apprentissage efficace. Pour le lecteur intéressé, des détails sur la descente de gradient et le choix du pas d'apprentissage sont donnés dans l'annexe.

Prenons maintenant le cas d'un réseau en entier on ne va pouvoir calculer la fonction de perte qu'à la sortie du réseau. Cependant, avec la méthode décrite précédement on ne sait mettre à jour que la dernière partie du réseau. Il faut donc utiliser l'erreur en sortie du réseau pour mettre à jour tous les poids du réseau et non plus seulement la dernière couche. Et donc plus exactement, pour mettre à jour les poids d'un neurones, il faut connaitre les erreurs sur les poids de neurones en aval qui lui sont reliés. On parle alors de rétropropagation.

Pour ce faire, cela fait appel à la règles des dérivées chaînées (aussi appelé Théorème de

dérivation des fonctions composées), ie

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Si l'on note h^p l'entrée de la couche p, et que l'on considère k couches, on a donc

$$\begin{aligned}\frac{\partial L}{\partial W} &= \frac{L}{\partial h^k}\frac{\partial h^k}{\partial W} \\ &= \frac{L}{\partial h^k}\frac{\partial h^k}{\partial h^{k-1}}\cdots\frac{h^{p+1}}{\partial h^p}\frac{\partial h^p}{\partial W}\end{aligned}$$

$$\begin{aligned}\frac{\partial h^p}{\partial W} &= \frac{h^{p+1}}{\partial h^p}\frac{\partial h^k}{\partial W} \\ &= \frac{L}{\partial h^k}\frac{\partial h^k}{\partial h^{k-1}}\cdots\frac{h^{p+1}}{\partial h^p}\frac{\partial h^p}{\partial W}\end{aligned}$$

1.5 Dropout

Comme dit précédement, il est n'est en général pas possible de connaitre à l'avance l'architecture du réseaux qui permettra de bien modéliser les données sans qu'apparaisse un réel problème de sur apprentissage. De fait deux approches pour arriver de façon empirique à une bonne architecture existe. La première consiste à rajouter au fur et à mesure des neurones (des couches). La deuxième consiste au contraire à supprimer des connexions neuronales. Aujourd'hui il semble que c'est la deuxième approche qui est la plus privilégiée, on ne discutera donc pas de la première ici.

Les premières versions de "sélection de modèles" consistant à supprimer des connexions neuronales sont du à LeCun et al. (1990) (Optimal Brain Damage) et sa version généralisée par Hassibi and Stork (1993) (Optimal Brain Surgeon). L'idée est assez simple. On considère que le réseau est bien entrainé pour des simplifications théoriques de la hessienne puis il s'agit de trouver le lien tel que si on le retire, le terme d'erreur n'augmente que très peu. On répète cette procédure jusqu'à ce que le fait de retirer une connexion augmente trop l'erreur. Cependant, il est nécessaire de calculer l'inverse de la hessienne. Aujourd'hui la plupart des logiciels de calculs font en sorte si nécessaire d'optimiser ce genre de calcul, mais avec la taille grandissante des réseaux qui comportent parfois des millions de paramètres et dont l'entrainement prend parfois plusieurs jours, rajouter cette étape risque de rajouter un important temps de calcul ce qui n'est pas toujours souhaitable.

De plus on peut noter que les connexions sont retirées en se basant sur la perte et non pas sur la pertinence du lien. De fait Zaprnanis and Haramis (2001)propose un technique appelée "Irrelevant Connection Elimination" (ICE). Le principe est le même que l'OBS mais la manière de déterminer quel lien supprimé est différente. Cette approche en plus de profiter du fait de ne pas nécessiter de calculer l'inverse de la hessienne se montre en pratique plus efficace.

Cependant les améliorations se montrent parfois toujours trop lourde en temps de calcul. Ainsi, on peut se demander ce qu'il se passe si on supprime des connexions de manière aléatoire.

C'est ce qu'on appelle le "Dropout" (au moins dans le cas du logiciel Keras présenté plus tard). Bien qu'on pourrait douter de la pertinence de cette approche, elle semble en pratique donner de bons résultats. On peut citer notamment Hinton et al. (2012) qui préconise de fixer un pourcentage de lien à supprimer de 50 %.
C'est cette méthode qui sera utilisée dans les exemples.

1.6 Remarques

Aujourd'hui, lors de l'apprentissage sur de grandes masses de données, pour des limitations de stockage mémoire et de temps de calcul il est rarement possible d'utiliser à chaque itération l'ensemble du jeu de données. Une solution à ce problème qui est très utilisée en pratique et qui sera mise en oeuvre dans la suite est le "Batch". Il s'agit simplement à chaque étape de choisir un sous ensemble du jeu de données (appelé batch) et d'entraîner le réseau sur ce sous ensemble.

Les logiciels utilisées par la suite étant en anglais, le code montré dans la suite sera lui aussi en anglais.

Chapitre NNET pour la construction de réseaux de neurones et comparaison avec d'autres méthodes statistiques en R

2.1 Présentation de NNET

Le package nnet de R est un package simple permettant de construire un réseau à une couche cachée. Ainsi il est assez limité (pas de véritable choix de la méthode d'optimisation, des fonctions de perte, ...). En contre partie il a l'avantage d'offrir, couplé à la fonction tune du package e1071 une méthode d'optimisation sur les paramètres du réseaux à savoir dans ce cas, la taille de la couche cachée et le taux d'apprentissage.

Ainsi, la liste des principaux paramètre de la méthodes sont : les données d'apprentissage (x pour les entrées, y pour les sorties), la taille de la couche cachée (size), le pas d'apprentissage (decay), si on est dans un problème de régression (linout =T) ou non (linout =F), si on veut un fonction de perte de type entropy (entropy = T) ou moindre carré (entropy =F). La majorité des paramètres restant correspondent à des paramètres sur le critère d'arrêt de l'optimisation.

Ainsi si l'on souhaite prédire Y (variable continue du jeu de données data_app) il faut écrire :

```
modele1 <- nnet(Y ~ ., data = data_app, size = 10, decay = 0.1,
   linout = T, entropy = F )

modele2 <- nnet(Y ~ X1 + X2, data = data_app, size = 10, decay =
   0.1, linout = T, entropy = F )
```

Le modèle 1 correspond à la régression de Y en fonction de toutes les autres variables de data_app. Le modèle 2 correspond à la régression de Y seulement sur les variables X1 et X2 du jeu de données.

2.2 Comparaison avec d'autres méthodes statistiques : Application à la prévision de la production électrique d'une centrale

Dans cette section nous allons comparer les résultats d'un perceptron à une couche cachée avec un modèle ARMA et un modèle de gradient boosting.

A titre d'exmple, je vais utiliser un jeu de données disponible à l'adresse https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant. Il consiste en environ 9500 points collectés sur une centrale électrique pendant une période de 6 ans (2006-2011).

Afin de ne pas fausser l'analyse, les données ont été conservées uniquement lorsque la centrale fonctionnait à plein régime. Chaque observation consiste en 5 mesures :

- la température moyenne par heure (en degrés)
- la pression ambiante (en milibar)
- l'humidité relative (en pourcentage)
- la vapeur rejetée (en cm Hg)
- la production d'électricité dans l'heure suivante (en MW)

Un des grands défis actuel est le stockage de l'électricté. En effet, il est aujourd'hui très difficile de garder longtemps l'électricité. Il est donc nécessaire d'avoir une bonne connaissance des quantités produites pour notamment éviter que certaines zones se retrouvent sans électricité. Mais aussi pour une entreprise, éviter de produire à perte.

2.2.1 ARMA

Pour la modélisation de série temporelle, une des modélisations les plus classiques est la modélisation par des modèles ARMA (modèles autorégressifs et moyenne mobile, ou en anglais Autoregressive Moving Average). Cette modélisation a été décrite dans la thèse de Peter Whittle en 1951 et popularisée par le livre de Box et Jenkins en 1970 Box George E. P. (1970).

Ainsi un modèle ARMA d'ordre (p,q) est un processus aléatoire $(X_t)_{t\in\mathbb{N}}$ qui vérifie l'équation :

$$X_t = \epsilon_t + \sum_{i=1}^{p} \phi_i X_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j}$$

Où les (ϕ_i) et les (θ_j) sont les paramètres du modèle et les (ϵ_t) sont les termes aléatoires qui sont indépendants identiquement distribués selon un bruit blanc centré en 0 (dans notre cas $\epsilon_t \overset{iid}{\sim} N(0,\sigma^2)$.

L'objectif ici n'étant pas une étude approfondie des séries temporelles, on ne détaillera donc pas plus. Pour plus d'information sur les processus ARMA et les série temporelles plus généralement, on peut citer notamment le livre très complet *Introduction to Time Series and Forecasting* Pete J. Brockwell (2016).

2.2.2 Gradient Boosting

Le boosting est une technique d'apprentissage ensembliste introduite de façon claire par Breiman Breiman (1997). Elle consiste à agréger des modèles (ou classifieurs) qui sont élaborés de façon séquentiel sur un échantillon d'apprentissage dont les poids des individus sont corrigés au fur et à mesure. De plus, les poids des modèles sont pondérés selon leurs performances.

L'idée est de condirer pleins de modèles simples pour obtenir une bonne approximation. On peut penser dans un style assez proche à l'estimateur de Nadaraya-Watson Nadaraya (1964), qui conssite à combiner plusieurs gaussiennes bien choisies.

Ainsi dans un contexte de régression et lorque l'on considère la perte quadratique, on cherche $F(x) = \hat{y}$ qui minimise

$$L(y, F(x)) = \frac{1}{n}\sum_i (\hat{y}_i - y_i)^2$$

Si on considère M étapes, à chaque étapes m, on va construire un nouveau modèle F_{m+1} basé sur l'ajout d'une correction simple h et du modèle de l'étape précédente F_m de sorte que :

$$F_{m+1}(x) = F_m(x) + h(x)$$

Ainsi, si on note H l'ensemble des modèles simples possibles, on a

$$F_{m+1}(x) = F_m(x) + \gamma_m \underset{h_{m+1}\in H}{argmin} \left[\sum_{i=1}^{n} L(y_i,\ F_{m-1}(x_i) + h_{m+1}(x_i))\right]$$

où γ_m est un paramètre de poids sur la correction ajoutée.

D'une manière plus détaillée, l'algorithme de Gradient Boosting classique Friedman (2000) s'écrit comme :

- $F_0(x) = \underset{\gamma\in\mathbb{R}}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$
- Pour m = 1 jusqu'à M :
  1. pour tout $1 \leq i \leq n$ calculer $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$
  2. calculer $h_m = \underset{h_m\in H}{argmin} \left[\sum_{i=1}^{n} L(y_i,\ F_{m-1}(x_i) + h_m(x_i))\right]$
  3. calculer le poids $\gamma_m = \underset{\gamma\in\mathbb{R}}{argmin} \left[\sum_{i=1}^{n} L(y_i,\ F_{m-1}(x_i) + \gamma h_m(x_i))\right]$
  4. $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$
- renvoyer $F_M(x)$

Ce type de méthode est comme les réseaux de neurones très facilement sujet au sur apprentissage. De fait on rajoute souvent un coefficient de shrinkage (rétrecissement) Telgarsky (2013) $0 < \nu \leq 1$ lors de la mise à jour du modèle selon l'équation :

$$F_m(x) = F_{m_1}(x) + \nu\ \gamma_m h_m(x)$$

2.2.3 Comparaison des approches

Afin de mesurer les écarts entre prédiction et réalité, on choisit la fonction de perte quadratique.

Une étude rapide des données ne montre pas de saisonnalité ou de tendance non constante. On va donc ne pas faire de prétraitement particulier des données.

Les données ayant un dépendance temporelle, on va regarder la fonction d'auto-corrélation pour déterminer l'ordre (p,q) du modèle ARMA

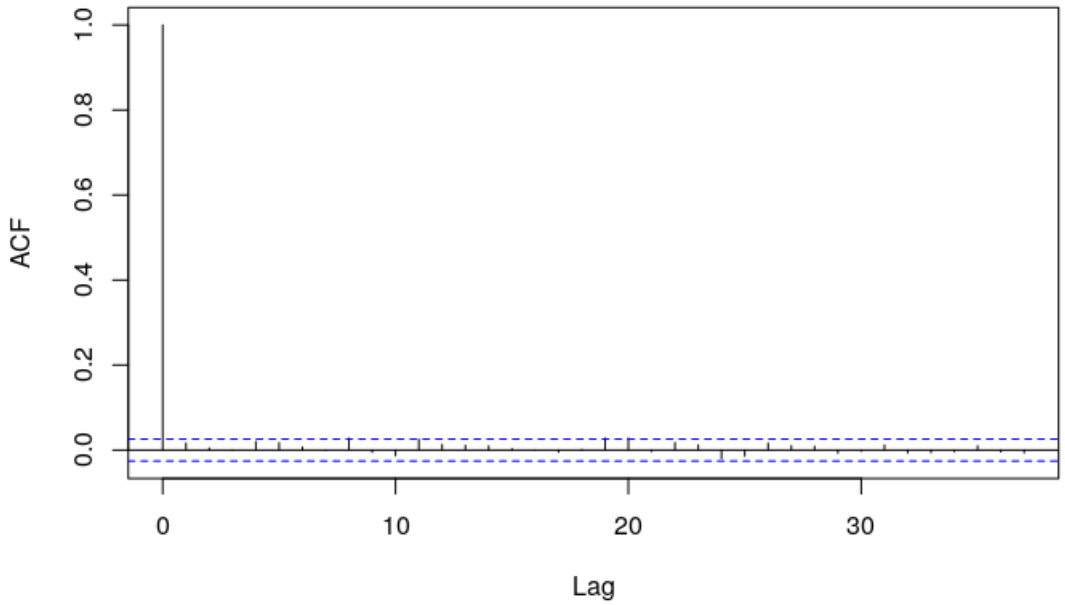

Figure 2.1 : Graphe d'auto-corrélation du jeu de données

On observe qu'un processus ARMA(1,1) devrait convenir. On peut utiliser le package 'asta' de R pour construire le modèle. On obtient alors comme valeur de la fonction de perte 22.

Ce package permet aussi de visualiser différents résultats sur les résidus :

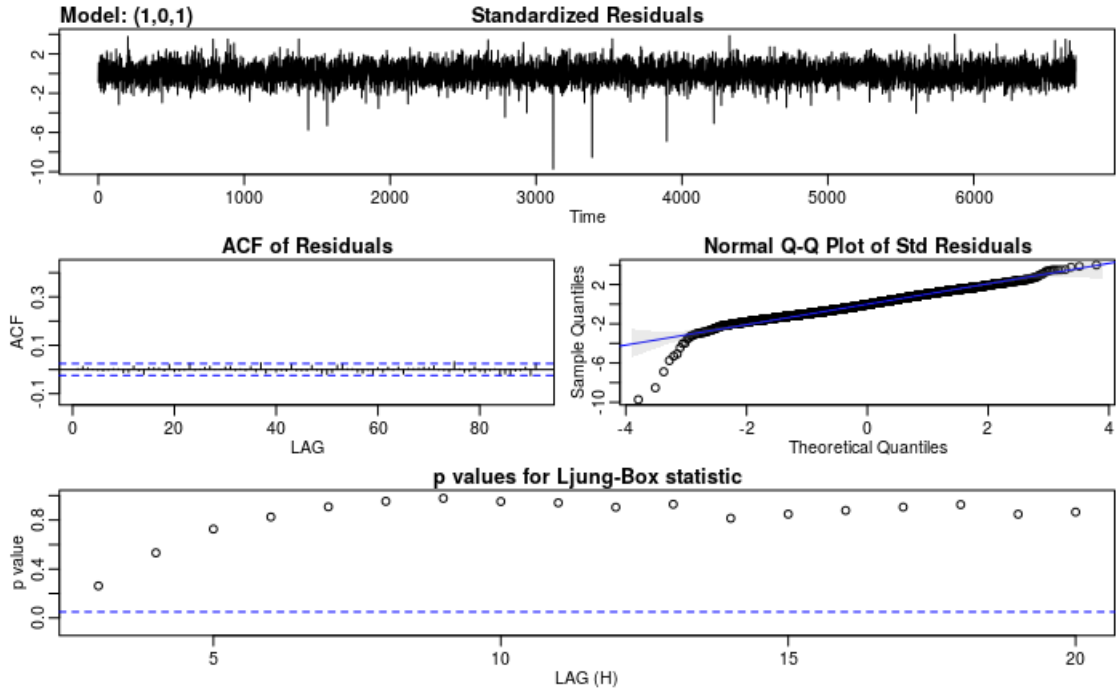

Figure 2.2 : Résultats du modèle ARMA(1,1)

On observe que la normalité des résidus n'est pas vérifiée, tout comme l'hypothèse d'homocédasticité du bruit ($\forall t,\ Var(\epsilon_t) = \sigma^2$). Une idée pour obtenir de meilleurs résultats serait donc d'utiliser un modèle plus complexe, par exemple un modèle GARCH Bollerslev (1986).

Comparaison néanmoins le résultat avec le modèle de gradient boosting (package 'gbm'

de R). Cela permet de réaliser une estimation via boosting avec comme modèles des arbres de régressions.

Il faut choisir quelque paramètre du modèles :

- distribution = "gaussian", le fonction de perte est l'erreur quadatratique
- n.trees = 50, le nombre d'arbres à construire
- interaction.depth = 5, la taille maximale des arbres
- shrinkage = 0.1, le coefficient de shrinkageq

On obtient alors une erreur quadratique de 17. On voit que c'est meilleur que le modèle ARMA.

On va maintenant comparer aux performance d'un perceptron à une couche cachée. Pour ce faire, on va utiliser la package 'nnet' de R et la fonction tune de 'e1071' afin de déterminer la taille de la couche cachée et le pas d'apprentissage (constant) optimaux.

Pour ce faire, plusieurs tests ont été réalisée en amont afin de construire un affichage intéressant, i.e. que l'on puisse voir une vraie différence entre un modèle peu efficace et un modèle efficace. Le défaut de tune et de nnet en général est que le temps de calcul est très important (en particulier par rapport à d'autres packages / logiciels). On se contentera donc de peu de neurones. On obtient finalement :

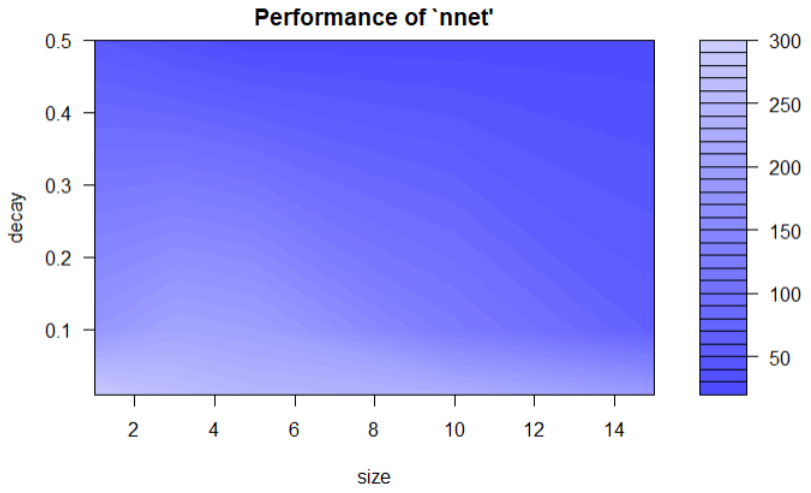

Figure 2.3 : Résultat de la fonction tune

On obtient alors une erreur quadratique pour le réseau avec les meilleurs paramètres de 19.

Finalement, on voit qu'ici le réseau de neurones performe mieux que le modèle ARMA mais moins bien que celui de boosting. Cela peut s'expliquer par le fait que le boosting est assez complexe tandis-que le réseau est ici relativement simple. Cependant, tous les modèles performent ici plutôt bien. Ce n'est pas si étonnant étant donné que les données ont une structure relativement simple (linéaire). On peut donc conclure cette partie en disant qu'il n'est pas toujours nécessaire d'utiliser un réseau de neurones qui est un modèle très complexe lorsque cela n'est pas nécessaire.

2.2.4 Code R

A l'origine code sous forme d'un fichier R markdown

```
#packages
library(nnet) # pour le PMC
library(MASS) #pour le jeu de données iris
library(ggplot2) #pour l'affichage
library(e1071) #pour les SVM
library(rpart)
library(readxl)
library(cowplot)
library(astsa) # ARMA(1,1)
library(gbm)

#importation des données
data <- read_excel("C:/Users/cypri/OneDrive/Documents/Reseaux de neurones
    2019-2020/projet/CCPP/Folds5x2_pp.xlsx")
nb_obs <- length(data$AT)

ggplot.hist <- function(data){
  nb_bins <- 1 + log2(length(data))
  g <- ggplot() + geom_histogram( aes(x = data, y = ..density.. ), bins = nb_
      bins )
  return(g)
}

g1 <- ggplot.hist(data$AT) + labs(title = "Temperature")
g2 <- ggplot.hist(data$V) + labs(title = "Exhaust Vacuum")
g3 <- ggplot.hist(data$AP) + labs(title = "Ambient Pressure")
g4 <- ggplot.hist(data$RH) + labs(title = "Relative Humidity")
g5 <- ggplot.hist(data$PE) + labs(title = "Net hourly electrical energy output"
    )

# visualisation des données
plot_grid(g1,g2,g3,g4,g5, ncol = 2)

# fonction de perte
quad_error <- function(x,y){
  return( mean( (x-y)**2 ) )
}
```

```
# sélection des paramètres du réseau
set.seed(0)

nb_app <- ceiling(0.7 * nb_obs)
nb_test <- nb_obs - nb_app

ind_app <- 1:nb_app

data_app <- data[ind_app,]
x_data_test <- data[-ind_app,-5]
y_data_test <- data[-ind_app,5]

tune.model <- tune.nnet(PE ~ ., data = data_app, size = c(1, 3, 5, 10, 15),
    decay = c(0.5, 0.1, 0.01), linout = T )

print(tune.model)
plot(tune.model)

# entrainement d'un réseau et affichage de ses performances
pmc_model <- nnet(PE ~ ., data = data_app, size = 5, decay = 0.5, linout = T, )
y_pmc_pred <- c(predict(pmc_model, x_data_test, type = "raw"))

print(paste("quad␣error␣for␣the␣optimal␣PMC(5)␣is", quad_error(y_pmc_pred, y_
    data_test$PE)))


pmc_model <- nnet(PE ~ ., data = data_app, size = 10, decay = 0.5, linout = T,
    )
y_pmc_pred <- c(predict(pmc_model, x_data_test, type = "raw"))

print(paste("quad␣error␣for␣the␣optimal␣PMC(10)␣is", quad_error(y_pmc_pred, y_
    data_test$PE)))

#Approche série temporelle


acf(data_app$V) # fonction d'autocorrélation


model_sa <- sarima(data_app$PE, 1,0,1, xreg = data_app[,-5])

pred_sa <- sarima.for(data_app$PE, n.ahead = nb_test, 1,0,1, xreg = data_app
    [,-5], newxreg = x_data_test)
```

```
print(paste("quad␣error␣for␣the␣ARMA(1,1)␣is", quad_error(pred_sa$pred, y_data_
    test$PE)))

#Approche Boosting

# create the training formula

set.seed(1234)
gbm_model1 <- gbm(formula = PE ~ ., data = data_app,
                  distribution = "gaussian", n.trees = 50,
                  interaction.depth = 5, shrinkage = 0.1)

summary(gbm_model1)

y_gbm <-predict.gbm(gbm_model1, n.trees = 50, newdata = x_data_test)
print(paste("quad␣error␣for␣the␣GBM␣is", quad_error(y_gbm, y_data_test$PE)))
```

Chapitre Un autre approche logiciel pour les réseaux de neurones : Keras

De nos jours, le logiciel le plus utilisé dans le domaine des réseaux de neurones est Python. De fait c'est celui que j'utiliserai dans la suite. De plus, il a l'avantage d'être gratuit et disponible sur la plateforme clood : google colab, qui offre de très bonnes performances de calcul et notamment le calcul sur GPU nécessaire aujourd'hui.

Aujourd'hui, il existe deux principales librairies utilisées sont Keras (sur-couche de TensorFlow, développée par Google) et PyTorch (dévellopée par Facebook). La principale différence entre les deux est la gestion des ressources machine. Tandis que PyTorch nécessite une gestion des ressources manuelles, il faut indiquer manuellement la mémoire (CPU ou GPU nécessaire pour des temps de calculs résonnables) sur laquelle les données sont stockées et les calculs sont exécutés. Alors que pour Keras, cela est fait automatiquement. Bien que cela offre moins de flexibilité, j'utilise dans la suite Keras car cela facilite grandement l'implémentation.

Keras est une librairie open-source de python développée pour l'utilisation de réseaux de neurones. Elle a été crée en 2015 par François Chollet un ingénieur de chez Google. Son objectif est d'offrir une interface facile d'utilisation pour la manipulation de réseau de neurones.

En 2017, elle est intégrée dans la librairie TensorFlow, librairie développée par Google pour les réseaux de neurones, permettant ainsi de profiter de la vitesse de calcul de TensorFlow avec une interface simplier. Cette facilité d'utilisation est son avantage par rapport à l'autre librairie majeure pour le développement de réseaux de neurones à savoir PyTorch développée par Facebook mais plus difficile d'utilisation notamment de par le fait qu'il faut spécifier pour chaque étape si les calculs sont faits sur GPU (carte graphique) ou non.

A l'heure actuelle, il existe trois façon d'implementer des réseaux de neurones à l'aide de keras :

- la méthode "sequential" : elle permet d'implémenter de simples architectures couche par couche. C'est la méthode basique d'utilisation de Keras. Je n'en parlerai pas par la suite car elle est très détaillée sur internet et n'apporte rien de plus par rapport aux deux autres et est assez proche de la méthode "functional".
- la méthode "functional" : c'est le type d'API (interface de programmation) la plus populaire de Keras. Elle permet de construire tous les réseaux réalisable via la méthode "sequential" mais facilite de plus la réalisation d'architectures plus complexes, par exemple qui incluent des entrées et/ou sorties multiples et des branches. Elle reste tout de même claire et facile d'utilisation comme nous le verrons. Cette API se retrouve aussi sous R.
- la méthode "Subclassing" : c'est la plus compliquée des trois. Elle consiste a créer une sous classe d'une classe déjà existante de Keras. Elle est particulièrement utile lorsque

l'on souhaite un contrôle total sur le réseau car cette API permet de redéfinir des fonctions native de Keras. Du fait de ca complexité, la programmation est plus lourde et difficile autant à écrire qu'à debboguer.

La suite de ce chapitrebest principalement inspirée du tutorial de François Chollet disponible à l'adresse `https://colab.research.google.com/drive/14CvUNTaX1OFHDfaKaaZzrBsvMfhCOHIR`.

3.1 Quelques méthodes classiques de Keras

Avant de voir comment utiliser Keras à proprement parler, je vais expliquer rapidement quelques méthodes classiques qui seront utilisées par la suite.

- keras.layers.Dense(size = ..., activation = '..') : qui représente une couche de neurones complètement connectée à laquelle on peut ajouter une fonction d'activation telle que 'relu' ou 'softmax'.
- keras.layers.Flatten() : s'utilise avant une couche complètement connectée, permet de linéariser la couche suivante par exemple au lieu d'avoir une matrice de neurone 28x28, on obtient un vecteur de neurones de taille 784.
- keras.optimizers.SGD : représente la méthode d'optimisation 'Descente de Gradient Stochastique'.
- keras.losses. ... : classe qui contient les différentes fonctions de perte présente nativement dans keras.

Dans les exemples de code des parties suivantes, TensorFlow sera referencé comme tf et keras sera équivalent à tf.keras (version 2 de Tensorflow).

3.2 Création de réseaux de neurones avec Keras

Comme indiqué en introduction, l'une des vocations de Keras est de rendre très facile la conception de réseaux de neurones performant d'un point de vue computationnel.

Commençons par présenter la méthode la moins évidente.

3.2.1 Subclassing

Cette façon de coder les réseaux de neurones est celle qui permet de rédéfinir manuellement des fonctions codées de manière automatique en python dans TensorFlow. Pour cela, on utilise l'héritage : principe de la programmation objet qui introduit une hierarchie entre les différents objets (classe, sous-classe, etc) et permet à la sous classe d'utiliser les méthodes de la classe dont elle hérite. On peut donc se contenter de redéfinir les fonctions que l'on souhaite dans le réseau.

L'utilisation première est l'héritage de la classe qui permet de définir les réseaux en entier.

Listing 3.1 : Définition de la classe de modèles Perceptron à 2 couches cachées

```
class MLP(keras.Model) :
```

```
    def __init__(self, hidden1_size, hidden2_size, output_size) :
        #il n'y a pas besoin de donner la taille d'entrée des
           couches
        super(MLP, self).__init__() :
        self.layer1 = keras.layers.Dense(hidden1_size, activation='
           relu')
        self.layer2 = keras.layers.Dense(hidden2_size activation='
           relu')
        self.layer3 = keras.layers.Dense(output_size, activation='
           softmax')

    def call(self, input_data) :
        output_data = self.layer1(input_data)
        output_data = self.layer2(output_data)
        return self.layer3(output_data)

# Afin de construire un objet de cette classe, il suffit de faire
    la chose suivante
model = MLP(hidden1_size, hidden2_size, output_size)
```

Comme le montre l'exemple si dessus, il faut définir la méthode __init__ qui définit la structure voulue du réseau et la méthode call qui correspond à la façon dont les données d'entrées se propagent dans le réseau. En effet suivant si l'on est en phase d'apprentissage ou en phase de test, le fait de supprimer des liaisons entre neurones n'a pas le même fonctionnement ce qui n'est pas pris en compte par la méthode call de base. Mais on peut également rédéfinir d'autres fonctions comme la méthode : fit, qui effectue l'apprentissage des poids du réseau.

Mais on peut aussi se contenter de construire une nouvelle type de couche.

Listing 3.2 : Redéfinition d'une couche completement connectée

```
class Linear(keras.Layer) :
  #y = w.x + b

  def __init__(self, units) :
      super(Linear, self).__init__()
      self.units = units

  def build(self, input_shape) :
      self.w = self.add_weight(shape=(input_shape[-1],
        self.units), initializer='random_normal',
```

```
                                trainable=True)
        self.b = self.add_weight( shape=(self.units,),
                                initializer='random_normal',
                                trainable=True)

    def call(self, inputs) :
        return tf.matmul(inputs, self.w) + self.b
```

La fonction build permet de définir les poids de la couche.

On peut alors utiliser cette nouvelle couche comme une couche de base de keras lors de la construction de réseau. A titre d'exemple, on va entrainer ce réseau sur le jeu de données MNIST qui consiste en des images de nombres entre 0 et 9 écrit à la main. Pour cela dans l'esprit de tout réécrire à la main, on va définir la fonction qui permet d'entrainer le modèle.

Listing 3.3 : Entrainement d'une couche complètement connectée

```
# on importe les données (comme on regarde juste la partie
    apprentissage, on ne récupère pas les données de test)
(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
mnist_dataset = tf.data.Dataset.from_tensor_slices( (x_train.
    reshape(60000, 784).astype('float32') / 255, y_train) )
#mise en forme des données sous forme de batch
mnist_dataset = mnist_dataset.shuffle(buffer_size=1024).batch(64)

# On initialise un réseau à compètement connecté à une couche avec
     10 neurones
linear_layer = Linear(10)

# On doit définir une fonction de perte, ici on choisit une
    version parcimonieuse (en terme de calculs) de l'entropie-
    croisée
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True)
# de même on doit choisir un solveur, ici la descente de gradient
    stochastique
optimizer_sgd = tf.keras.optimizers.SGD(learning_rate=1e-3)

def train_model(model, loss_function, optimizer, dataset, verbose
    = True) :
  # On itère sur chaque lot (batch) du jeu de données
```

```
  for step, (x, y) in enumerate(dataset) :

    # On ouvre un GradientTape, cela permet de garder en mémoire
       les calculs des gradients
    # utile pour accélérer l'étape de rétropropagation
    with tf.GradientTape() as tape :

      # On propage les entrées dans le réseau
      logits = model(x)

      # On calcule la perte associée
      loss = loss_function(y, logits)

    # On récupère les gradients associés aux poids du modèle
    gradients = tape.gradient(loss, model.trainable_weights)

    # On met à jour les poids du modèle
    optimizer.apply_gradients(zip(gradients, model.
       trainable_weights))

    # on affiche l'évolution de la perte en fonction des
       itérations
    if step % 100 == 0 and verbose  :
      print(step, float(loss))

# entraine le réseau
train_model(linear_layer, loss_fn, optimizer_sgd, mnist_dataset)
```

Malgré l'avantage de pouvoir redéfinir les fonctions de native de Keras, cela n'est pas toujours nécessaire. De plus, du fait de recoder certaines fonctions, le modèle est donc écrit en python plutôt que dans la structure native de TensorFlow (graphe statique de couches), ce qui diminue les performances de calculs et rend certaines méthodes et attributs inaccessibles pour ce genre de modèles.

Cependant pour avoir de meilleures performances de calcul, il est nécessaire de créer soi-même une fonction que l'on va encapsuler avec @tf.function pour effectuer l'apprentissage du réseau.

Listing 3.4 : Utilisation de @tf.function

```
mlp = MLP(64, 64, 10)
```

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True)
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)

@tf.function
def train_on_batch(x, y) :
  with tf.GradientTape() as tape :
    logits = mlp(x)
    loss = loss_fn(y, logits)
    gradients = tape.gradient(loss, mlp.trainable_weights)
  optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
  return loss

for step, (x, y) in enumerate(mnist_dataset) :
  loss = train_on_batch(x, y)
  if step % 100 == 0 :
    print(step, float(loss))
```

Afin de pouvoir utiliser toute l'étendue des capacités de TensorFlow, on pourra préférer la méthode fonctionnelle.

3.2.2 Functionnal

Comme indiqué dans la partie précédentes, le principale défaut de la méthode subclassing est la complexité du code pour les réseaux sans architecture trop complexe.

Ainsi on va reprendre la définition précédente et la réécrire via l'approche fonctionnelle.

Listing 3.5 : Définition de la classe de modèles Perceptron à 2 couches cachées

```
(x_train, y_train), _ = tf.keras.datasets.mnist.load_data()

x_train = x_train.astype(np.float32) / 255
y_train = keras.utils.to_categorical(y_train, 10)

hidden1_size = 64
hidden2_size = 64
output_size = 10

inputs = keras.Input(shape=(28,28), name='entry')
x = keras.layers.Dense(hidden1_size, activation='relu')(inputs)
x = keras.layers.Dense(hidden2_size, activation='relu')(x)
```

```
x = keras.layers.Flatten()(x)
outputs = keras.layers.Dense(output_size, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs, name='MLP')

# pour spécifier la méthode d'optimization et la fonction de perte
    , on fait appel à la méthode compile
model.compile(optimizer='sgd', loss='categorical_crossentropy',
    metrics=['accuracy'])

#permet d'afficher les différentes couches du réseau avec leurs
    tailles et le nombre de paramètres
model.summary()

# pour entrainer le réseau, il suffit d'utiliser la méthode fit
# epochs représente le nombre d'itérations
model.fit(x_train, y_train, epochs= 10, batch_size= 64)
```

Listing 3.6 : Résultat de la méthode summary

```
Model: "MLP"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
entry (InputLayer)           [(None, 28, 28)]          0
_________________________________________________________________
dense_21 (Dense)             (None, 28, 64)            1856
_________________________________________________________________
dense_22 (Dense)             (None, 28, 64)            4160
_________________________________________________________________
flatten_6 (Flatten)          (None, 1792)              0
_________________________________________________________________
dense_23 (Dense)             (None, 10)                17930
=================================================================
Total params: 23,946
Trainable params: 23,946
Non-trainable params: 0
```

On observe que cette fois-ci, il est nécessaire de définir la taille d'entrée du réseau.

3.2.3 Apprentissage et Keras

Lors de la description de l'apprentissage du réseau, on a vu qu'il est nécessaire d'être capable de calculer la dérivée des différentes fonctions utilisées. Cependant, si on regarde la fonction d'activation relu qui est l'une des plus utilisées actuellement, on observe qu'elle n'est pas dérivable en 0. De fait, l'optimisation est théoriquement impossible si on doit dériver en 0. Pour surpasser ce problème, tensorflow considère que relu'(0) = 0. Cependant avec ce résultat, Pauwels (n.d.) montre qu'on peut réécrire la fonction relu de telle sorte à changer la valeur de sa dérivée ce qui pose problème. Cette video de conférence explique que pour faire cela proprement, il faudrait en réalité utiliser une formulation basée sur le sous différentiel. Dans le cas convexe on peut montrer de bonnes propriétés sur ce genre de fonctions et donc sur la convergence de la méthode. Mais cela ralentirait les calculs et en pratique, on observe que les réseaux ont de bonnes performances, cela n'est donc pas forcement intéressant d'un point de vue pratique.

Chapitre Exemple d'utilisation d'une carte de Kohonen

Les cartes de Kohonen, du nom de son concepteur, aussi apppelées carte auto-adaptative ou SOM (self-organized map) Kohonen (1982) sont un type de réseaux de neurones non supervisés, qui ont pour objectif de reprodiure en dimension faible (usuellement deux ou trois pour que l'on puisse visualiser la structure) en réalisant une grille.

4.1 Algorithme d'apprentissage

L'objectif de l'apprentissage de la méthode SOM est de faire en sorte que différentes partie du réseau répondent de manière similaire à certaines entrées. Cette idée est en partie motivée par le fait que les informations visuelles, auditives et sensorielles sont gérées par des parties séparées du cortex cérébral du cerveau humain.

Les poids des neurones sont initialisés soit par de faibles valeurs aléatoires ou tirés au hasard dans un sous espace engendré par les composantes principales Ciampi and Lechevallier (2000). Cette dernière alternative permet une converge plus rapide dans le cas ou les données sont linéaire mais elle est déconseillée dans le cas non linéaire Akinduko and Mirkes (2016).

L'entraînement à proprement parlé fonctionne grâce à l'apprentissage par compétition. Lorsqu'un exemple d'apprentissage est donné au réseau, sa distance euclidienne à chaque vecteur de poids est calculée. Le neurone dont le vecteur de poids est le plus proche de l'entrée est appelé BMU (best matching unit). Les poids du BMU et des neurones proches de lui dans la grille SOM sont mis àjour via la formule suivante :

$$W_v(s+1) \quad = \quad W_v(s) + \theta(u, v, s) \cdot \alpha(s) \cdot (D(t) - W_v(s))$$

Où $W_v(s)$ représente les poids du neurones v à l'instant s, D(t) est l'exemple d'apprentissage, u est le neurone BMU. $\alpha(s)$ est un coefficient décroissant d'apprentissage et $\theta(u, v, s)$ est la fonction de voisinnage qui donne la distance entre le neurone u et le neurone v à l'instant s.

Usuellement, θ est un noyau gaussien d'écart type s, mais la fonction peut être différente. Cependant, dans tous les cas, le voisinnage diminue avec le temps s.

Bien que les cartes de Kohonen soient avant tout une méthode de réduction de dimension, on peut les utiliser pour faire de la classification. En effet, pour cela, il suffit de considérer pour une cellule de la grille la classe qui réagit le plus à cette cellule.

4.2 SOM : capacités et limites

Contrairement à de nombreuses méthodes, la méthode somme permet de représenter de manière non linéaire des données. Afin de représenter ces capacités, on va utiliser un exemple sur des données générées par make_circles de scikit-learn.

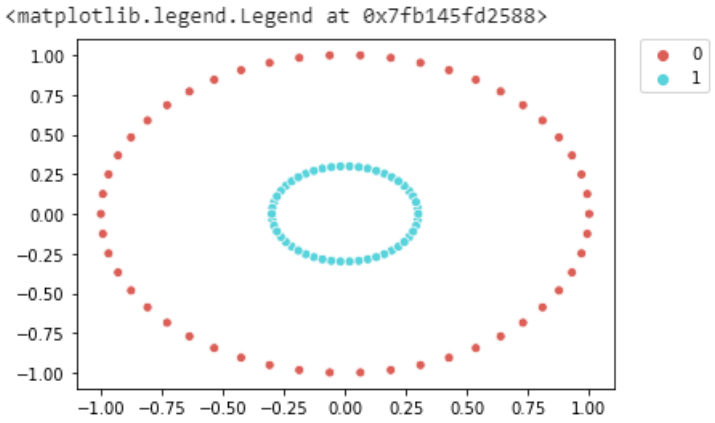

Figure 4.1 : Données d'exemple SOM

On peut alors considerer une grille hexagonal de taille 10x10 et observer le résultat de l'algorithme (en bleu) par rapport aux vrais données (en rouge).

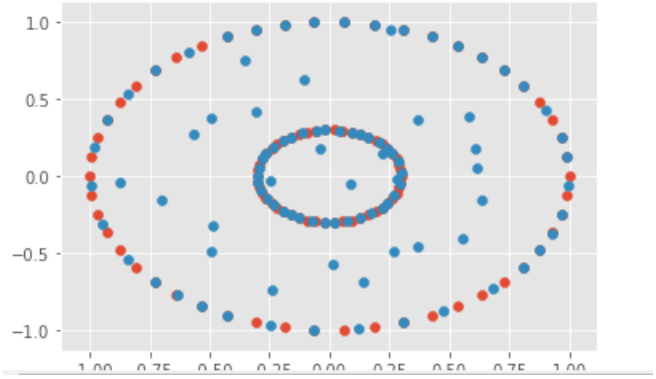

Figure 4.2 : Résultat SOM sur cercles

Ce code a été réaliser avec le package neupy de python.

On voit très clairement la capacité du réseaux à apprendre des structures de données complexes. Cependant, cet algorithme a ses limites. C'est pour ces raisons qu'il est moins utilisés aujourd'hui au profit d'autres méthodes comme la t-sne van der Maaten and Hinton (2008).

Afin de voir les limitations de la méthode, prenont un autre jeu de données. Les données que je vais utilisée sont issue du jeu de données Télimétrie. Le jeu de données est composé de cinq fichiers : l'historique des défaillances, l'historique des maintenances, des mesures en fonctionnement des détails sur la machine de production et sur l'opérateur utilisant la machine. Les données ont été rassemblées suivant la procédure décrite à l'adresse `https://gallery.azure.ai/`

Notebook/Predictive-Maintenance-Modelling-Guide-R-Notebook-1. Elles sont composées des mesures des conditions de fonctionnement (tension, pression, etc) et des meusures liées à des défaillainces et celà pour une centaine de machines. Ici par mesure de simplicité, on va regarder 5 machines pour lesquels on ne garde que 700 observations. L'objectif est trouver une représentation qui permette de séparer les différentes machines et ce en faible dimension (2,3) alors que les données d'entrées comportent une vingtaine de variables.

Pour ces données, on utilisera le package minisom de python.

La sortie de l'algorithme nous donne :

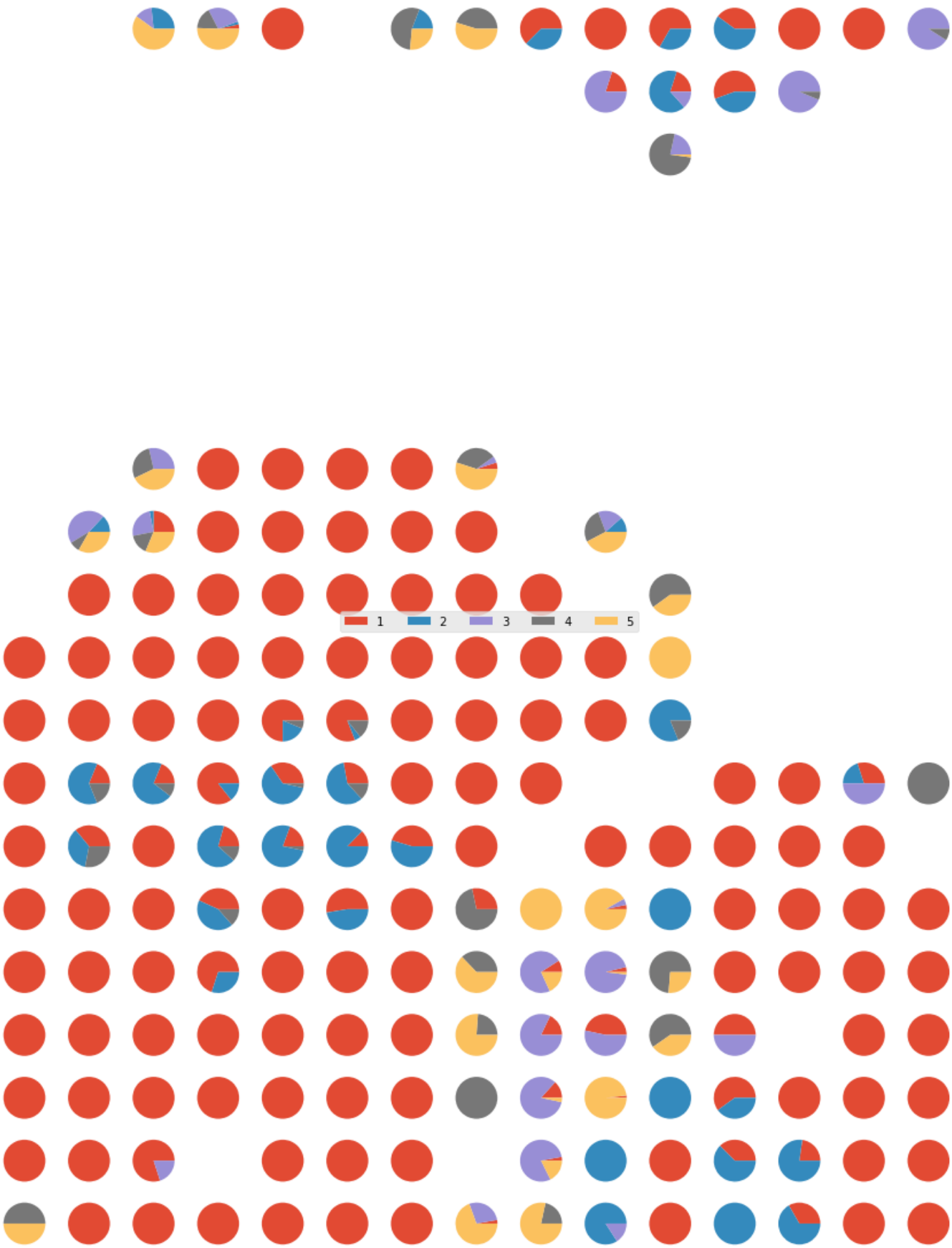

FIGURE 4.3 : Résultat SOM sur les données télémetrie

On voit que l'algorithme ,n'a pas réussi à bien apprendre une représentation des données en faible dimension. En effet, on voit qu'il y a beaucoup de mélanges parmis les données. A titre d'exemple, le résultat de la méthode t-sne donne

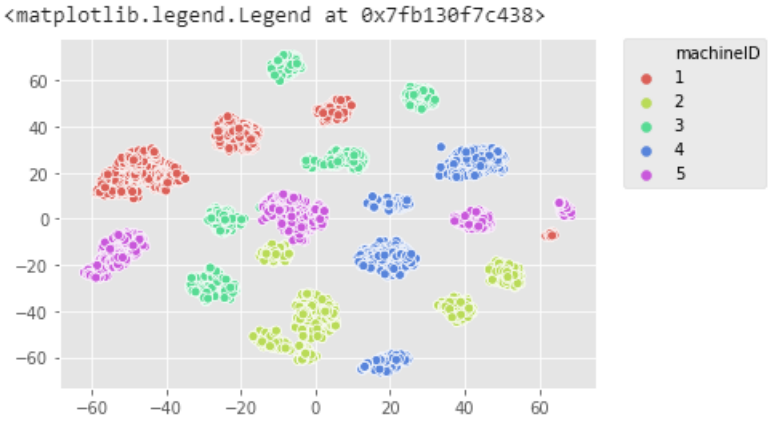

Figure 4.4 : Résultat tsne sur données télémetrie

On voit que dans ce cas, on arrive bien à séparer les données. On observe de plus que pour une machine, il y a plusieurs nuages de points. Il faudrait une analyse pour justifier de cela, mais on peut imaginer des modes de fonctionnement différents.

Chapitre Introduction à l'Apprentissage par Renforcement

5.1 Présentation de l'apprentissage par renforcement

L'apprentissage par renforcement (reinforcement learning en anglais) est devenu ces dernières années une partie de l'intelligence artificielle avec un intérêt grandissant. Cependant, contrairement à ce que l'on pourrait croire, l'apprentissage par renforcement est une approche assez ancienne. Son émergence à deux motivations distinctes. La première est basée sur le contrôle optimal. C'est un concept de la fin des années 50 afin de décrire le problème de la conception d'un contrôleur pour minimiser une mesure du comportement d'un système dynamique. L'une des approches à ce problème est la résolution de l'équation de Bellman Bellman (1957). La classe des méthodes de résolution de cette équation s'appelle programmation dynamique. Dans la partie suivante, on regardera la version stochastique discrète du problème de contrôle optimal qui est connu sous le nom de processus de décision markoviens (MDP pour Markov Decision Process) Beranek and Howard (1961).

L'autre approches est celle de l'apprentissage via la méthode essai-erreur développée dans la psychologie de l'apprentissage des animaux Thorndike (1898). C'est une méthode d'apprentissage qui consiste à associer des comportements particuliers aux conséquences qu'ils produisent, i.e. le comportement est susceptible de se répéter si les conséquences sont agréables, mais pas si elles sont désagréables.

C'est cette dernière partie qui a conduit au renouveau de l'apprentissage par renforcement dans les années 80. Bien qu'au départ ces deux approches faisaient route à part, l'apprentissage par renforcement moderne consiste à une réunion de ces deux approches.

Malgré leur ancienneté, tout comme les réseaux de neurones, ces méthodes ont connu une période de vide à la fin des années 80 avant d'avoir un regain d'intérêt récemment. On peut citer notamment les grands succès dans le monde du jeu vidéo : il est possible de battre les meilleurs joueurs de jeux vidéo comme Dota ou League of Legends en 2016 (aujourd'hui on peut citer notamment OpenAI), mais aussi AlphaGo qui a battu pour la première fois un professionel du jeu de Go en 2015 avant de battre la légende du jeu en 2016. Un des développement d'application étudié actuellement pour l'apprentissage par renforcement est la conduite autonome.

Afin de comprendre un peu mieux le fonctionnement de l'apprentissage par renforcement, prenont un exemple. Imaginons un enfant qui commence à marcher. Il doit apprendre à ne pas se cogner dans les meubles pour aller rejoindre ses parents sur le canapé en traversant le salon. A

force de se cogner dans les meubles, il va y associer un sentiment négatif et va donc apprendre à les éviter. C'est cette approche que l'on va essayer d'appliquer.

Dans un contexte plus général, un problème d'apprentissage par renforcement se modélise comme suit : On considère un agent (dans notre exemple l'enfant) qui va intéragir avec un espace d'états (ici le salon) appelé environnement. On appelle action le fait de se déplacer dans l'espace d'état. A chaque action, on associe une récompence (par exemple se cogner contre un meuble entraine une récompense négative, atteindre le canapé une récompense positive, ni l'un ni l'autre une récompense nulle). Enfin on appele politique la fonction de sélection de l'action à réaliser en fonction de l'état de l'agent dans l'environnement.

L'objectif de l'apprentissage par renforcement est de trouver une politique qui maximise les récompenses obtenues.

Dans l'exemple, on considère un seul objectif (atteindre le canapé) et que l'environnement est connu et ne change pas au cours de l'apprentissage. Ces hypothèses peuvent être élargies : plusieurs agents, environnement inconnu ou qui évolue. On parle d'état terminal lorsque le jeu s'arrête une fois l'état atteint. Dans notre exemple le canapé. On peut aussi considérer le fait de se cogner contre un meuble comme un état terminal.

Ainsi plus généralement, on peut utiliser l'apprentissage par renforcement pour résoudre les problèmes suivant :

- Apprendre à évaluer une action en fonction d'un état
- Trouver la meilleure séquence d'actions pour réaliser un objectif
- Explorer un espace d'état

5.2 Formalisme mathématique

Dans cette partie nous allons regarder la modélisation mathématique de ce type de méthode. Plus particulièrement, on va s'intéresser à une modélisation à l'aide des processus de décision markoviens.

On note

- S : l'espace d'états
- A : l'espace d'actions
- T : S x A $\mapsto$ S, la fonction de transition
- r : S x A $\mapsto \mathbb{R}$, la fonction de récompense

Pour simplifier la modélisation fait l'hypothèse suivante :

Hypothse Markovienne : la récompense et la fonction de transition dépendent que de l'état (et action) en cours, pas de l'historique.

On définit de plus

- $\pi \mapsto A$

- V^π : $S \mapsto \mathbb{R}$: une fonction de valeur d'états
- Q^π : S x A $\mapsto \mathbb{R}$: une fonction de valeur d'actions
- une séquence / scénario qui est une liste de triplet d'éléments de S x A x $\mathbb{R}$
- sur une séquence de temps t_0 à T, la récompense globale comme $R_{t_0} = \sum_{t=0}^{T-t_0} \gamma^t r_{t_0+t}$ avec $0 < \gamma \leq 1$, un paramètre d'oubli

Plus exactement, on considère que $V^\pi(s) = E_\pi[R_{t_0}|s_{t_0} = s]$ et $Q^\pi(s,a) = E_\pi[R_{t_0}|s_{t_0} = s, a_{t_0} = a]$.

L'objectif est de trouver la politique qui maximise V^π. Elle s'écrit selon les cas comme suit :

- $r(s,\pi(s)) + \gamma V^\pi(s')$ dans le cas où π et le MDP sont déterministes
- $r(s,\pi(s)) + \gamma \sum_{s'} p(s'|s,\pi(s))V^\pi(s')$ si le MDP n'est pas déterministe
- $\sum_a \pi(s,a)\,[r(s,\pi(s)) + \gamma \sum_{s'} p(s'|s,\pi(s))\, V^\pi(s')]$ si la politique n'est pas déterministe

L'équation de maximisation de V^π s'appelle Equation de Bellman.

Notons π^* la politique optimale. Elle vérifie les équations

$$\begin{aligned} V^* &= \max_\pi V^\pi(s) \\ &= \max_a r(s,a) + \gamma \sum_{s'} p(s'|s,a)V^*(s') \end{aligned}$$

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) \\ &= E[r_{t+1} + \gamma V^*(s_{t+1}|s_t = s, a_t = a)] \\ &= r(s,a) + \gamma \max_a \sum_{s'} p(s'|s,a)Q^*(s',a') \end{aligned}$$

Dans le cas où Q^* est connue, la politique optimale est alors

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

Ce problème peut être résolu de manière exact grâce à des algorithmes de programmation dynamiques. On peut notamment citer :

L'opérateur de Bellman T^π Bellman (1957) qui consiste à itérer

$$\begin{aligned} V_{t+1}(s) &= (T^\pi V_t)(s) \\ &= r(s,\pi(s)) + \gamma \sum_{s'} p(s'|s,\pi(s))V_t(s') \end{aligned}$$

Du fait que ce soit un opérateur contractant, on a l'existence d'un point fixe. Cette procédure permet donc d'estimer V^π par itération successive et donc on peut en déduire la politique recherchée.

Une autre méthode est l'algorithme Policy Iteration : Beranek and Howard (1961). Tant qu'il n'y a pas convergence, on répète

- On évalue V^{π_t}
- On améliore la solution $\pi_{t+1}(s) = \operatorname*{argmax}_a [r(s,a) + \gamma \sum_{s'} p(s'|s,a)V^{\pi_t}(s')]$

Enfin on peut aussi citer l'algorithme Value Iteration Shapley (1953) qui consiste à alterner

évaluation de V et amélioration de la politique :

On boucle $V_{t+1} = \max_a [r(s,a) + \gamma \sum_{s'} p(s'|s,a)V_t(s')]$ et on en dérive la politique optimale, $\forall s,\ \pi'(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)[r(s,a) + \gamma V^{\pi}(s')]$. On parle alors de politique greedy (gloutonne).

5.3 Algorithmes d'apprentissage par renforcement

L'apprentissage par renforcement en temps que tel est un problème plus général. En effet, dans ce problème, le MDP est inconnu. Seules les actions sont connues.

On distingue dans ce cas deux approches :

model-free : on cherche uniquement π

model-based : on estime le MDP puis on en déduit π

Pour résoudre ce problème, on doit donc utiliser des approches différentes.

5.3.1 Monte Carlo

La première approche, la plus simple est une approche par Monte Carlo. On va donc estimer $V^{\pi}(s)$ par moyenne empirique de la récompense sur un grand nombre d'échantillons (que l'on va simuler).

Malgré sa simplicité, il présente plusieurs avantages. Le premier est que l'estimateur est non biaisé pour peu que les échantillons soient indépendants. Le deuxième avantage est que c'est une approche Model free : il n'y a aucun besoin du modèle ou d'approximation de ce dernier. En contre-partie, la principale limite de cette approche est qu'elle nécessite que les épisodes se terminent.

L'agorithme Monte Carlo Singh and Sutton (1996) consiste à répeter les étapes suivantes :

- Générer un épisode $(s_i, a_i, r_i)_{1 \leq i \leq T}$ selon la politique π
- Fixer R = 0
- Pour tout t = T-1 à t = 1, $R = \gamma R + r_t$ et $V(s_t) = V(s_t) + \alpha(R - V(s_t))$

La même approche peut être utilisée pour calculer $Q^{\pi}(s,a)$

Afin de ne plus dépendre du problème de terminaison des épisodes, une nouvelle méthode appelée Temporal-Difference Learning a été développée.

5.3.2 Temporal-Difference Learning

Son principe est de combiner une approche Monte-Carlo et une approche de Programmation dynamique. Ainsi l'algorithme est capable d'apprendre à chaque étape plutôt qu'à la fin d'un scénario. De plus, l'algorithme peut fonctionner même avec des scénarii incomplets, des séquences incomplètes ou des séquences sans fin.

Comme exemple d'algorithme, un des plus utiliser est l'algorithme TD-λ Sutton (1988) :
Initialiser $V^{\pi}(s)$ de façon arbitraire (sauf si s est terminal, dans quel cas $V^{\pi}(s)$ =0) Pour chaque

épisode, répeter :

- Initialiser E(s) = 0 Pour chaque étape d'un épisode, répeter :
  - Soit S l'état actuel, A l'action donnée par π, la récompense observée R et le nouvel état après action S'
  - $\delta \leftarrow R + \gamma V^{\pi}(S') - V^{\pi}(S)$
  - E(S) $\leftarrow (1 - \alpha)E(S) + 1$
  - $\forall s : V^{\pi}(s) \leftarrow V(s) + \alpha\delta E(s)$ et $E(s) \leftarrow \gamma E(s)$

Il existe d'autres procédures de Temporal-Difference Learning très utilisées en pratique. Plutôt que de chercher à calculer V^{π}, ces méthodes cherchent à calculer Q^{π}. On parle alors de Q-learning Watkins and Dayan (1992).

Un des algorithme les plus connus est l'algorithme SARSA (Stateactionrewardstateaction) Rummery and Niranjan (1994) : Pour chaque épisode, on répète :
On choisit l'état initial S, on choisit A une action selon une politique issue de Q et on itère

- Réaliser l'action A (soit R la récompense et S' le nouvel état)
- Choisir une action A' selon une politique issue de Q.
- $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
- $S \leftarrow S', A \leftarrow A'$

On s'arrête lorsque S est terminal.

5.4 Dilemme Exploration/ Exploitation

Avec les méthodes évoquées précédement, on a vu qu'il est possible d'évaluer une politique π. Cependant, on considère ci-dessus des approches "naïves" qui parcourent l'ensemble des possibilités de la même manière dans lesquelles notre politique π qui est estimée de manière greedy. L'idée est alors de rendre non déterministe notre politique. On peut citer dans ce genre les méthodes ϵ-greedy, UCB, etc (présentées rapidement plus loin). Ou alors alterner entre estimation de π et de V^{π}. On parle alors de Genearlized Policy Iteration (GPI).

Pour une structure GPI, on répète alors les trois étapes suivantes :

- Jouer un scénario selon la politique π avec une dose d'exploration
- Evaluer la politique : mettre à jour V et Q
- Améliorer la politique : mettre à jour de manière greedy π

L'idée est alors non plus de regarder seulement la récompense associée à chaque action de manière individuelle, mais aussi de regarder les récompenses futures. On va alors introduire une fonction de valeurs d'états (aussi appelée d'état/action) qui va servir d'indication sur le long terme des récompenses attendues plutôt que de regarder uniquement la récompense imédiate.

Dès lors, on se retrouve à devoir décider de l'action à réaliser à partir de l'estimation de la fonction de valeurs. On se pose alors la question : a-t'on confiance ou non en notre estimation ? Si oui, alors on suit la fonction de valeur. On appelle cela exploitation. Si ce n'est pas le cas on

va essayer une des autres actions possibles. Cette phase s'appelle quand à elle exploration.

Il s'agit alors de trouver un compromis entre l'exploitation (l'utilisation de notre fonction de décicsion) et l'exploration (on considère un autre choix que celui fait par notre fonction de décision). En effet dans le cas ou le MDP est inconnu on doit apprendre la fonction de décision et par conséquent faire de l'exploration. De plus dans bien des cas, il n'est pas possible d'explorer tout l'espace d'état, une fois que l'on a appris une bonne politique, on se retrouve à devoir faire un choix : la suivre ou explorer en espérant en trouver une meilleure. C'est ce qu'on appelle dilemme Exploitation / Exploration.

Pour cela, on peut utiliser un algorithme ce que l'on appelle ϵ-greedy Watkins (1989), qui est une méthode gloutonne, mais avec une probabilité 1-ϵ on écoute la fonction de valeur et avec probabilité ϵ on explore. On peut aussi citer la famille d'algorithme UCB (Upper Confidence Bound) Auer et al. (2002). C'est une famille d'algorithmes utilisés à l'origine pour la résolution du problème du Bandit Manchot (choisir entre plusieurs machines à sous pour maximiser ses gains). L'idee est d'ajouter un biais dans l'équation à maximiser pour obtenir une meilleure solution (à l'image du compromis biais variance en statistique).

5.5 Deep reinforcement Learning

Comme on l'a vu en introduction, aujourd'hui l'apprentissage par renforcement est utilisé pour résoudre des problématiques très complexes avec des espaces d'états gigantesque (par exemple de l'ordre de 10^{20} pour le backgammon et 10^{170} pour le jeu de Go). Cela couplé au fait que l'on peut avoir plusieurs agents et non plus un seul avec des interactions entre eux, il est impossible de visiter tous les états. Ce à quoi on peut aussi rajouter le fait que le temps n'est pas forcément discret et peut être continu (conduite autonome ou encore contrôle de drône).

5.5.1 Passage à l'échelle

La solution à ces difficultés : approximer V et Q. Dans ce but, on va choisir un famille de fonctions paramétrée par un vecteur de $\mathbb{R}^d$. On va alors approximer $V^\pi(s)$ par $\hat{v}(s,w)$ et $Q^\pi(s,a)$ par $\hat{q}(s,a,w')$. Les paramètres (w,w') sont alors optimisés soient par Monte Carlo soit par TD-learning.

En plus de permettre l'utilisation de l'apprentissage par renforcement à des situations complexes, cette approches rend aussi possible de généraliser sur des états qui n'ont pas été vus et plus généralement d'estimer les couples (état, action). Cela permet aussi l'utilisation de méthodes d'apprentissage on-line (les données sont reçues une par une) ce qui est souvent le cas en pratique.

D'une manière un peu plus formelle la procédure est la suivante :

- Objectif : Trouver $w^* = \underset{w}{argmin}\ E_\pi[(V^\pi(s) - \hat{v}(s,w))^2]$

- Pour cela on utilise une descente de gradient (souvent stochastique) ce qui donne $\Delta_w = \alpha(V^\pi(s) - \hat{v}(s,w))\nabla_w \hat{v}(s,w)$
- On represente un état comme une fonction de projection dans $\mathbb{R}^d$: $x(s) = (x_i(s))_{1 \leq i \leq d}$
- Suivant la famille de fonction choisi (linéaires, RBF, ...), on obtient une forme différente de gradient, par exemple pour le cas linéaire : $\Delta_w = \alpha(V^\pi(s) - \hat{v}(s,w))x(s)$

5.5.2 Lien avec le deep learning

Comme on l'a vu ci-dessus pour pouvoir appliquer l'apprentissage par renforcement à des situations très complexes, on utilise une approximation de la fonction de décision. On peut notamment utiliser un réseau de neurones comme famille de fonction pour approximer. Dans le cas où l'on utilise un réseau profond comme fonction d'approximation on parle alors de deep reinforcement learning. Le principal avantage à utiliser un réseau de neurone pour modéliser la politique est que c'est une famille qui bien que paramétrique, peut approximer la plupart des fonctions même très complexes et même dans notre cas le couple action/récompense. De plus des architectures de réseaux de neurones particulières (réseaux convolutionnels notamment) surpassent toutes les autres méthodes pour la reconnaissance de formes ce qui est très utile notamment pour la conduite autonome. Mais on peut aussi citer d'autres applications de ces méthodes notamment pour les chatbots (systèmes de conversions intelligents).

Chapitre Un exemple de réseau profond : utilisation de l'architecture LSTM

La plupart des réseaux de neurones actuels (PMC) sont ce que l'on appelle des réseaux feedforwards. C'est à dire que l'information en entrée se propage dans un seul sens.

Cela peut conduire à plusieurs limitations. La première qui ne sera pas traitée ici est que les pour les réseaux feedforwards, les entrées et les sorties sont toutes de même tailles. Cela est assez contraignant dans des cas comme le traitement du langage. Par exemple dans un problème de traduction, on peut vouloir donner en entrée au réseau des phrases de tailles différentes ce qui n'est pas possible avec les réseaux classiques. La deuxième est que ces reseaux ne gardent pas en mémoire les sorties des couches autres que celle juste avant eux. Or si l'on se rappelle du modèle ARMA pour les séries temporelles, on voit clairement que l'estimation au temps t ne dépend pas uniquement de l'estimation au temps t-1 mais aussi des estimations avant. Cela est impossible avec une architecture de réseau feedforward. Ces réseaux ne semblent donc pas adaptés à la gestion de données temporelles.

Ainsi pour combler ces problèmes, il est nécessaire d'utiliser des réseaux de neurones que l'on appelle : réseau de neurones récurrent (RNN).

6.1 Les réseaux récurrents

Les réseaux neuronaux récurrents forment une classe de réseaux neuronaux artificiels où les connexions entre les nuds forment un graphique dirigé le long d'une séquence temporelle, i.e. il y a toujours une propagation de l'entrée vers la sortie dans un seul sens, mais on ajoute alors des connexions supplémentaires entres des couches qui ne seraient pas connectées en temps "normale". Un des plus célèbre est le réseau d'Hopfield.

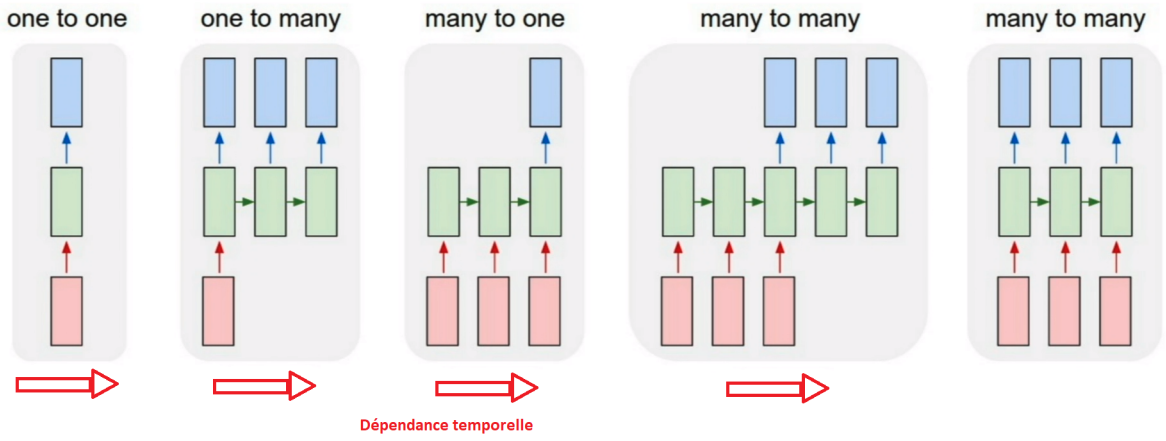

Figure 6.1 : Exemples de réseaux récurrents : Vanilla RNN (https ://calvinfeng.gitbook.io/)

Cela permet d'avoir un comportement selon une dynamique temporelle. Contrairement aux réseaux neuronaux feedforwards, les RNN peuvent donc utiliser leur état interne (mémoire) pour traiter des séquences d'entrées de longueur variable. On retrouve à nouveau leur utilité pour des tâches telles que la reconnaissance d'écriture manuscrite non segmentée, la reconnaissance vocale ou encore l'étude de données temporelles.

Le terme "réseau neuronal récurrent" est utilisé indifféremment pour désigner deux grandes classes de réseaux ayant une structure générale similaire, où l'une est à impulsion finie et l'autre à impulsion infinie. Les deux classes de réseaux présentent un comportement dynamique temporel. Un réseau récurrent est dit à impulsion finie si c'est un graphe acyclique dirigé qui peut être déroulé et remplacé par un réseau neuronal feedforward, tandis qu'un réseau récurrent est dit à impulsion infinie si c'est un graphe cyclique dirigé qui ne peut pas être déroulé.

Les réseaux récurrents à impulsion finie et à impulsion infinie peuvent avoir des états stockés supplémentaires, et le stockage peut être sous le contrôle direct du réseau neuronal. Le stockage peut également être remplacé par un autre réseau ou graphique, s'il intègre des retards ou comporte des boucles de rétroaction. Ces états contrôlés sont appelés état fermé ou mémoire fermée, et font partie de réseaux de mémoire à court terme à long terme (comme les LSTM) et d'unités récurrentes fermées. On parle également de reseaux de neurones feedback.

Les réseaux de neurones récurrents sont comparables à des réseaux de neurones classiques avec des contraintes d'égalité entre les poids du réseau. Les techniques d'entraînement du réseau sont les mêmes que pour les réseaux classiques (rétropropagation du gradient), néanmoins les réseaux de neurones récurrents se heurtent au problème de disparition du gradient pour apprendre à mémoriser des évènements passés. Des architectures particulières répondent à ce dernier problème, on peut citer en particulier les réseaux Long short-term memory (LSTM) . On peut étudier les comportements des réseaux de neurones récurrents avec la théorie des bifurcations (systèmes dynamiques), mais la complexité de cette étude augmente très rapidement avec le nombre de neurones.

Pour ces réseaux de neurones on peut aussi rencontrer un problème qui n'arrive pas avec les réseaux classiques : l'explosion du gradient. De fait pour ce genre de réseau il est en général déconseillé d'utiliser la fonction d'activation RELU, car bien qu'elle fonctionne bien contre les problèmes de disparitions de gradient, elle a tendance à favoriser les problèmes d'explosion de gradient.

Aujourd'hui il est courant de combiner des couches convolutionnelles en entrée du réseau avec des couches récurrentes en sortie du réseau.

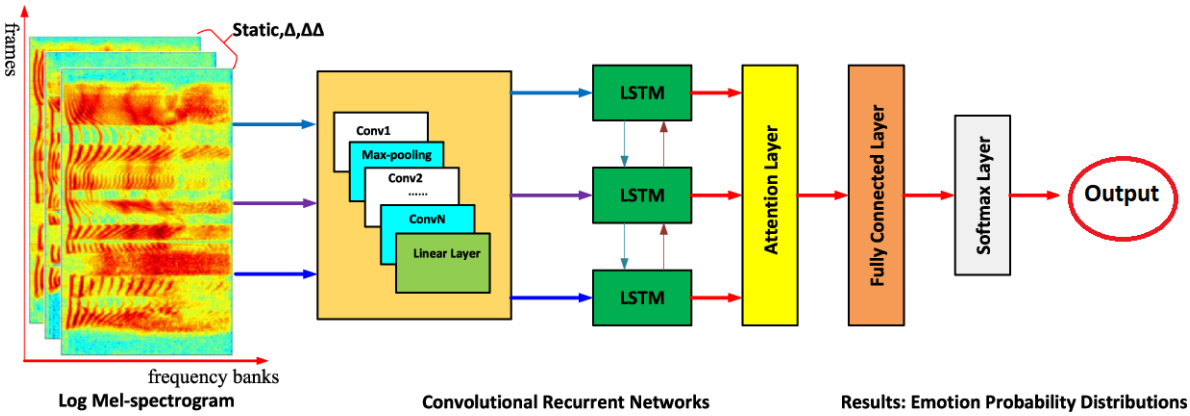

FIGURE 6.2 : Exemples d'architecture complexe Chen et al. (2018)

Nous ne regarderons pas ce genre de réseaux. Nous nous contenterons de l'architecture LSTM.

6.2 L'architecture Long Short Time Memory (LSTM)

Un réseau Long short-term memory (LSTM), en français réseau récurrent à mémoire court et long terme ou plus explicitement réseau de neurones récurrents à mémoire court-terme et long terme, est l'architecture de réseau de neurones récurrents la plus utilisée en pratique qui permet de répondre au problème de disparition de gradient. Le réseau LSTM a été proposé par Sepp Hochreiter et Jürgen Schmidhuber en 1997 Hochreiter and Schmidhuber (1997). L'idée associée au LSTM est que chaque unité computationnelle est liée non seulement à un état caché h mais également à un état c de la cellule qui joue le rôle de mémoire. Le passage de c_{t-1} à c_t se fait par transfert à gain constant et égal à 1. De cette façon les erreurs se propagent dans le passé sans phénomène de disparition de gradient. L'état de la cellule peut être modifié à travers une porte qui autorise ou bloque la mise à jour (input gate). De même une porte contrôle si l'état de cellule est communiqué en sortie de l'unité LSTM (output gate). La version la plus répandue des LSTM utilise aussi une porte permettant la remise à zéro de l'état de la cellule (forget gate).

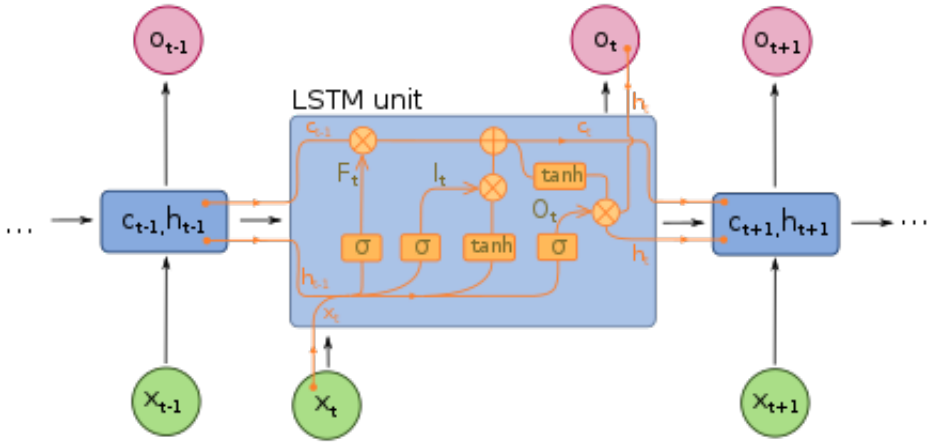

FIGURE 6.3 : Fonctionnement d'une unité LSTM (Wikipedia)

Ainsi, d'un point de vue plus mathématiques, le fonctionnement d'une cellule LSTM suit les équations suivantes.

Soient les valeurs initiales : $c_0 = 0$ et $h_0 = 0$. L'opérateur $\circ$ symbolise le produit matriciel de Hadamard (produit terme à terme). Les symboles σ et tanh représentent respectivement la fonction sigmoïde et la fonction tangente hyperbolique, bien que d'autres fonctions d'activation soient possibles. On obtient

$$
\begin{aligned}
F_t &= \sigma(W_F x_t + U_F h_{t-1} + b_F) && \text{(forget gate)} \\
I_t &= \sigma(W_I x_t + U_I h_{t-1} + b_I) && \text{(input gate)} \\
O_t &= \sigma(W_O x_t + U_O h_{t-1} + b_O) && \text{(output gate)} \\
c_t &= F_t \circ c_{t-1} + I_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= O_t \circ \tanh(c_t) \\
o_t &= f(W_o h_t + b_o)
\end{aligned}
$$

Il existe plusieurs approches pour apprendre les poids d'une cellule LSTM. La plus courrante est la méthode BackPropagation Through Time (BPTT) Williams and Zipser (1995).

L'idée est "d'applatir" (unfold en anglais) le réseaux et de considérer que le réseau "applati" partage les mêmes poids, alors d'utiliser la méthode de rétropropagation du gradient comme pour les perceptrons multicouches.

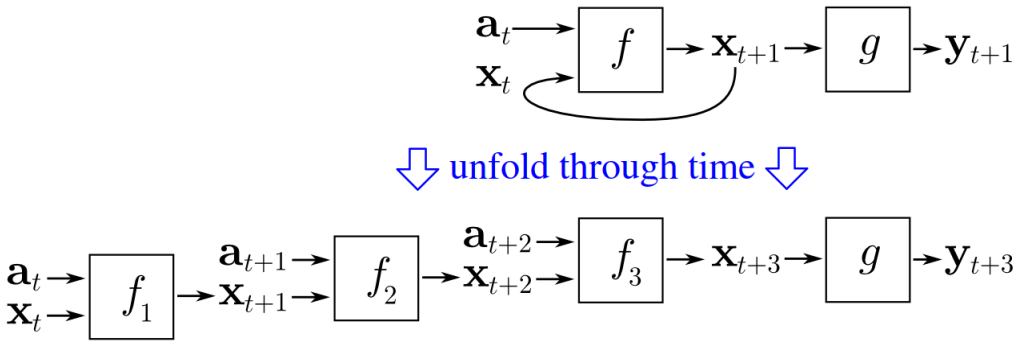

Figure 6.4 : Fonctionnement de la backpropagation through time (Wikipedia)

Afin d'illustrer le fonction on se place dans le cas ci dessus. La fonction de cout est considérée comme étant la somme des fonctions de couts intermédiaires. Ainsi pour déterminer la mise à jour des poids de f, il suffit de sommer les mises à jours des poids de f_1, f_2, f_3.

6.3 Exemple d'application à la maintenance prédictive

On va maintenant regarder comment entrainer les modèles que l'on a construit précédement. On regarde toujours un exemple de maintenance prédictive, mais on va tester un autre jeu de données. A titre d'exemple, on va utiliser un jeu de donnée qui représente une simulation du fonctionnement d'un moteur d'avion Saxena and Goebel (2008). L'objectif est de prédire s'il va

y avoir une panne dans la labs de temps prédéfini en fonction cette fois-ci, des paramètres de la machine et des valeurs de capteurs (ce à quoi ils correspondent n'est pas connu, mais on peut supposer que ce sont des quantités telles que la température, la pression, etc).

L'architecture du réseaux LSTM est issu de https://github.com/Azure/lstms_for_predictive_maintenance.

6.3.1 Utilisation des réseaux LSTM avec Keras

Comme vu dans la partie sur Keras, on pourrait définir manuellement une cellule LSTM et une couche de cellule LSTM. Néanmoins cela est assez long et est déjà mieux fait dans la bibliothèque keras de tensorflow. Le code source est accessible à l'adresse https://github.com/keras-team/keras/blob/master/keras/layers/recurrent.py

On peut noter deux manière de coder des réseaux LSTM avec keras. La première est en utilisant LSTMCell en l'englobant dans un réseaux récurrent comme le montre l'exemple suivant issu de la documentation de Tensorflow :

Listing 6.1 : Exemple de LSTMCell

```
inputs = np.random.random([32, 10, 8]).astype(np.float32)
rnn = tf.keras.layers.RNN(tf.keras.layers.LSTMCell(4))

output = rnn(inputs) # The output has shape `[32, 4]`.

rnn = tf.keras.layers.RNN(
    tf.keras.layers.LSTMCell(4),
    return_sequences=True,
    return_state=True)

# whole_sequence_output has shape `[32, 10, 4]`.
# final_memory_state and final_carry_state both have shape `[32,
    4]`.
whole_sequence_output, final_memory_state, final_carry_state = rnn
    (inputs)
```

Cependant toujours d'après la documentation, il est plutôt conseillé d'utiliser la fonction LSTM directement qui correspond à peut près à RNN(LSTMCell(...)...) mais peut offrir des gains de perfomance.

Listing 6.2 : Réseau LSTM a deux couches

```
# build the network (version fonctionnelle)
from tensorflow import keras
```

```
inputs = keras.Input(shape = (sequence_length, nb_features),name =
    'entry')
x = keras.layers.LSTM(
    units=100,
    return_sequences=True,
    name = 'first_LSTM_layer')(inputs)

x = keras.layers.Dropout(0.2, name = 'first_dropout_layer')(x)

x = keras.layers.LSTM(
    units=50,
    return_sequences=False,
    name = 'second_LSTM_layer')(x)

x = keras.layers.Dropout(0.2, name = 'second_dropout_layer')(x)

outputs = keras.layers.Dense(nb_out, activation='sigmoid', name =
    'Dense_layer')(x)

model = keras.Model(inputs = inputs, outputs = outputs , name = '2
    _layers_LSTM')

model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])

print(model.summary())

Model : "2_layers_LSTM"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
entry (InputLayer)           [(None, 50, 25)]          0
_________________________________________________________________
first_LSTM_layer (LSTM)      (None, 50, 100)           50400
_________________________________________________________________
first_dropout_layer (Dropout (None, 50, 100)           0
_________________________________________________________________
```

```
second_LSTM_layer (LSTM) (None, 50)            30200
_________________________________________________________________
second_dropout_layer (Dropou (None, 50)        0
_________________________________________________________________
Dense_layer (Dense)      (None, 1)             51
=================================================================
Total params : 80,651
Trainable params : 80,651
Non-trainable params : 0
```

On peut alors entrainer ce réseau

Listing 6.3 : Entrainement d'un réseau LSTM

```
model.fit(seq_array, label_array, epochs=10, batch_size=200,
   validation_split=0.05, verbose=1)

Epoch 1/10
75/75 [==============================] - 2s 20ms/step - loss :
   0.2557 - accuracy : 0.8857 - val_loss : 0.1762 - val_accuracy
    : 0.9399
Epoch 2/10
75/75 [==============================] - 1s 12ms/step - loss :
   0.1071 - accuracy : 0.9562 - val_loss : 0.0837 - val_accuracy
    : 0.9706
 ...

Epoch 10/10
75/75 [==============================] - 1s 12ms/step - loss :
   0.0542 - accuracy : 0.9777 - val_loss : 0.0321 - val_accuracy
    : 0.9872
```

Pour évaluer les performances du réseau on peut regarder la matrice de confusion (le nombre et le type des bonnes et mauvaises classification) et le nombre de bonnes classifications.

Listing 6.4 : Résultat du réseau LSTM

```
Confusion matrix
- x-axis is true labels.
- y-axis is predicted labels
array([[12310, 221],
```

```
      [   78,  3022]])

98.0871319770813 pourcents de bonnes classifications
```

On observe que le réseau aini construit est vraiment très performant. Rajouter des couches risquerait de conduire à du sur apprentissage sans réel gain de performance.

6.3.2 Comparaison avec le PMC

Dans les motivations des réseaux LSTM, on a que ces réseaux sont plus adaptés aux données temporelles afin de voir si cela se vérifie ici, on va comparer les performances à des PMC.

Dans un premier temps, on considère un PMC avec des couches de faibles tailles mais beaucoup de couche. :

Listing 6.5 : PMC

```
hidden1_size = 64
hidden2_size = 64

inputs = keras.Input(shape=(sequence_length, nb_features), name='
    entry')
x = keras.layers.Dense(hidden1_size, activation='relu')(inputs)
x = keras.layers.Dense(hidden2_size, activation='relu')(x)
x = keras.layers.Flatten()(x)
outputs = keras.layers.Dense(nb_out, activation='sigmoid')(x)

model2 = keras.Model(inputs=inputs, outputs=outputs, name='
    Small_MLP')
model2.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

model2.summary()

Model : "Small_MLP"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
entry (InputLayer)           [(None, 50, 25)]          0
_________________________________________________________________
dense_43 (Dense)             (None, 50, 64)            1664
_________________________________________________________________
```

```
dense_44 (Dense)          (None, 50, 64)          4160
_________________________________________________________________
flatten_9 (Flatten)       (None, 3200)            0
_________________________________________________________________
dense_45 (Dense)          (None, 1)               3201
=================================================================
Total params : 9,025
Trainable params : 9,025
Non-trainable params : 0
```

L'entrainement de ce réseau donne un ratio de bonnes prédictions / nombre de prédictions est de 0.32 ce qui est très mauvais. En comparaison du réseau LSTM. Cependant, en regardant le nombre de paramètres des deux résaux, on voit qu'il y a un très grande différence (environ 9000 ici contre 80000 dans le réseau LSTM). Les différences de performances viennent donc peut être de là et non de l'architecture en elle même.

De fait on construit un réseau avec environ 70000 paramètres :

Listing 6.6 : Grand PMC

```
hidden1_size = nb_features # 25
hidden2_size = 64

inputs = keras.Input(shape=(sequence_length, nb_features), name='
    entry')
x = keras.layers.Dense(hidden1_size, activation='relu')(inputs)
x = keras.layers.Dense(2 * hidden1_size, activation='relu')(x)
x = keras.layers.Dense(3 * hidden1_size, activation='relu')(x)
x = keras.layers.Dropout(0.5)(x)
x = keras.layers.Dense(5 * hidden1_size, activation='relu')(x)
x = keras.layers.Dense(7 * hidden1_size, activation='relu')(x)
x = keras.layers.Dropout(0.5)(x)
x = keras.layers.Dense(hidden2_size * 2, activation='relu')(x)
x = keras.layers.Dense(hidden2_size , activation='relu')(x)
x = keras.layers.Dense(hidden2_size / 2, activation='relu')(x)
x = keras.layers.Flatten()(x)
x = keras.layers.Dropout(0.5)(x)
outputs = keras.layers.Dense(nb_out, activation='sigmoid')(x)

model3 = keras.Model(inputs=inputs, outputs=outputs, name='
    large_MLP')
```

```
model3.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

model3.summary()

Model : "large_MLP"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
entry (InputLayer)           [(None, 50, 25)]          0
_________________________________________________________________
dense_46 (Dense)             (None, 50, 25)            650
_________________________________________________________________
dense_47 (Dense)             (None, 50, 50)            1300
_________________________________________________________________
dense_48 (Dense)             (None, 50, 75)            3825
_________________________________________________________________
dropout_12 (Dropout)         (None, 50, 75)            0
_________________________________________________________________
dense_49 (Dense)             (None, 50, 125)           9500
_________________________________________________________________
dense_50 (Dense)             (None, 50, 175)           22050
_________________________________________________________________
dropout_13 (Dropout)         (None, 50, 175)           0
_________________________________________________________________
dense_51 (Dense)             (None, 50, 128)           22528
_________________________________________________________________
dense_52 (Dense)             (None, 50, 64)            8256
_________________________________________________________________
dense_53 (Dense)             (None, 50, 32)            2080
_________________________________________________________________
flatten_10 (Flatten)         (None, 1600)              0
_________________________________________________________________
dropout_14 (Dropout)         (None, 1600)              0
_________________________________________________________________
dense_54 (Dense)             (None, 1)                 1601
=================================================================
Total params : 71,790
```

```
Trainable params : 71,790
Non-trainable params : 0
```

On obtient alors 80 % de réussite. C'est toujours assez significativement moins bien que le réseau LSTM, mais on observe de bonnes performances. Cependant, lorsque l'on s'intéresse à la matrice de confusion, on obtient :

Listing 6.7 : Matrice de confusion grand PMC

```
Confusion matrix
- x-axis is true labels.
- y-axis is predicted labels
[[12525    6]
 [ 3100    0]]
```

On voit que le réseau prédit presque tout dans une même classe ce qui est donc en fait très mauvais. Dans cet exemple au moins, on voit donc bien l'intéret des réseaux LSTM par rapport aux réseaux PMC.

Chapitre Conclusion

Les réseaux de neurones sont aujourd'hui des méthodes très populaires ces dernières années. Bien que cette classe de méthodes présente plusieurs défauts importants :

- On ne maitrise pas bien la théorie des réseaux de neurones. En dehors des PMC, il est très compliqué de vérifier théoriquement que le réseau va converger vers un minimum. De plus mal utilisés ils peuvent très rapidement conduire à une situation de sûr-apprentissage.
- Il n'existe pas de régles pour déterminer l'architecture du réseau adaptée à chacun des cas. Bien que l'on puisse considérer que les CNN semblent adaptés au traitement d'image, les réseaux récurrents pour les séquences temporelles et le langage naturel. Mais pas de réelles règles sur le nombre de couches / neurones. Ce qui rend leur confection parfois délicate.
- Contrairement à la plupart des méthodes statistiques il n'existe pas de réelle procédure d'intervalles de confiance / prévision, il n'y a donc pas de maitrise de l'erreur d'approximation en dehors d'approches par simulations qui sont très couteuses.

Cependant ces défauts sont contrebalancés par de très intéressant avantages. On peut notamment citer :

- La capacité d'approximer n'importe quelle fonction mesurable.
- Bien que théoriquement il n'y a que très peu de résultats et que ces derniers ne sont pas toujours adaptés, on observe en pratique que ces méthodes dans le cas d'applications complexes n'ont pas d'égales en terme de résultats parmis les autres méthodes existantes.
- Avec les avancées en terme de puissance de calcul informatique le désavantages de temps de calcul est de moins en moins important. Cela couplé avec de très bon outils open-source comme TensorFlow ou Pytorch il est très facile de créer son propre réseau.

Ainsi malgré ces limites, les réseaux de neurones n'ont pas d'équivalent dans de nombreuses applications et c'est notamment pour ces raisons qu'ils ont été choisis comme approximateurs dans l'apprentissage par renforcement.

Avec tout l'engoument (à juste titre) et leurs preformances ces méthodes ont encores un bel avenir devant elles.

Bibliographie

Akinduko, Ayodeji and Evgeny Mirkes, "SOM : Stochastic initialization versus principal components," *Information Sciences*, 10 2016, *364365*, 213–221.

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, 05 2002, *47*, 235–256.

Bellman, Richard, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, 1957, *6* (5), 679–684.

Beranek, William and Ronald Howard, "Dynamic Programming and Markov Processes," *Technometrics*, 02 1961, *3*, 120.

Bollerslev, Tim, "Generalized autoregressive conditional heteroskedasticity," *Journal of Econometrics*, 1986, *31* (3), 307 – 327.

Breiman, L., "Arcing The Edge," Technical Report 486 06 1997.

Brockwell, Richard A. Davis Pete J., *Introduction to Time Series and Forecasting*, Springer, 2016.

Chen, Mingyi, Xuanji He, Jing Yang, and Han Zhang, "3-D Convolutional Recurrent Neural Networks With Attention Model for Speech Emotion Recognition," *IEEE Signal Processing Letters*, 2018, *25*, 1440–1444.

Ciampi, Antonio and Yves Lechevallier, "Clustering Large, Multi-level Data Sets : An Approach Based on Kohonen Self Organizing Maps," *Lecture Notes in Computer Science*, 01 2000, *1910*, 161–177.

Duchi, John, Elad Hazan, and Yoram Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 07 2011, *12*, 2121–2159.

Fausett, Laurene V., "Fundamentals of neural networks : architectures, algorithms, and applications," in "in" 1994.

Friedman, Jerome, "Greedy Function Approximation : A Gradient Boosting Machine," *The Annals of Statistics*, 11 2000, *29*.

Glorot, Xavier, Antoine Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in "in," Vol. 15 01 2010.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Networks," 2014.

Gupta, Maya, Samy Bengio, and Jason Weston, "Training Highly Multiclass Classifiers," *Journal of Machine Learning Research*, 04 2014, *15*, 1461–1492.

Hassibi, Babak and David G. Stork, "Second order derivatives for network pruning : Optimal Brain Surgeon," in S. J. Hanson, J. D. Cowan, and C. L. Giles, eds., *Advances in Neural Information Processing Systems 5*, Morgan-Kaufmann, 1993, pp. 164–171.

Hebb, D. O., *The organization of behavior ; a neuropsychological theory*, Oxford, England : Wiley, 1949.

Hinton, Geoffrey, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint*, 07 2012, *arXiv*.

Hochreiter, Sepp and Jürgen Schmidhuber, "Long Short-term Memory," *Neural computation*, 12 1997, *9*, 1735–80.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, 1989, *2* (5), 359 – 366.

Kingma, Diederik and Jimmy Ba, "Adam : A Method for Stochastic Optimization," *International Conference on Learning Representations*, 12 2014.

Klambauer, Günter, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter, "Self-Normalizing Neural Networks," 06 2017.

Kohonen, Teuvo, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, 01 1982, *43* (1), 59–69.

LeCun, Yann, John S. Denker, and Sara A. Solla, "Optimal Brain Damage," in D. S. Touretzky, ed., *Advances in Neural Information Processing Systems 2*, Morgan-Kaufmann, 1990, pp. 598–605.

Man, Zhihong, Hong Ren Wu, Sophie Liu, and Xinghuo Yu, "A New Adaptive Backpropagation Algorithm Based on Lyapunov Stability Theory for Neural Networks," *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 12 2006, *17*, 1580–91.

Matsugu, Masakazu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network," *Neural Networks*, June 2003, *16* (5-6), 555–559.

Minsky, M. and S. Papert, "Perceptrons : An Introduction to Computational Geometry," *MIT Press*, 1969.

Nadaraya, E. A., "On Estimating Regression," *Theory of Probability & Its Applications*, January 1964, *9* (1), 141–142.

P., Jenkins Gwilym M. Box George E., *Time series analysis*, Holden-Day, 1970.

Pauwels, Edouard, "What does back propagation compute."

Pierre, Gillot, Zemmari Akka, Benois-Pineau Jenny, and Nesterov Yurii, "Algorithmes de Descente de Gradient Stochastique avec le filtrage des paramètres pour lentraînement des réseaux à convolution profonds," in "in" 2018.

Robbins, Herbert and Sutton Monro, "A Stochastic Approximation Method," *Ann. Math. Statist.*, 09 1951, *22* (3), 400–407.

Rosenblatt, F., "The perceptron : A probabilistic model for information storage and organization in the brain.," *Psychological Review*, 1958, *65* (6), 386–408.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*, Cambridge, MA, USA : MIT Press,

Rummery, G. A. and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Technical Report 1994.

Saxena, A. and K. Goebel, "Turbofan Engine Degradation Simulation Data Set," 2008. NASA Ames Research Center, Moffett Field, CA.

Shapley, L. S., "Stochastic Games," *Proceedings of the National Academy of Sciences*, 1953, *39* (10), 1095–1100.

Singh, Satinder P. and Richard S. Sutton, "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning*, 1996.

Strayer, Joseph R., "WALLIS, WILSON D. Messiahs : Their Role in Civilization. Pp. 217. Wash ington : American Council on Public Af fairs, 1943. Paper Ed. : $2.50 ; Cloth Ed. : $3.00," *The ANNALS of the American Academy of Political and Social Science*, September 1943, *229* (1), 198–198.

Sutton, Richard, "Learning to Predict by the Method of Temporal Differences," *Machine Learning*, 08 1988, *3*, 9–44.

Telgarsky, Matus, "Margins, Shrinkage, and Boosting," 2013.

Thorndike, Edward L., "Animal intelligence : An experimental study of the associative processes in animals," *The Psychological Review : Monograph Supplements*, 1898.

van der Maaten, Laurens and Geoffrey E. Hinton, "Visualizing Data using t-SNE," in "in" 2008.

Watkins, Christopher and Peter Dayan, "Technical Note : Q-Learning," *Machine Learning*, 05 1992, *8*, 279–292.

Watkins, Christopher John Cornish Hellaby, "Learning from Delayed Rewards." PhD dissertation, King's College, Cambridge, UK May 1989.

Williams, Ronald J. and David Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in "in" 1995.

Zaprnanis, A. D. and G. Haramis, "Obtaining locally identified models : The irrelevant connection elimination scheme," in "HERCMA" 2001.

Annexe : Mise à jour des poids

Approche via les systèmes dynamiques

Considèrons le réseau de neurones suivant :

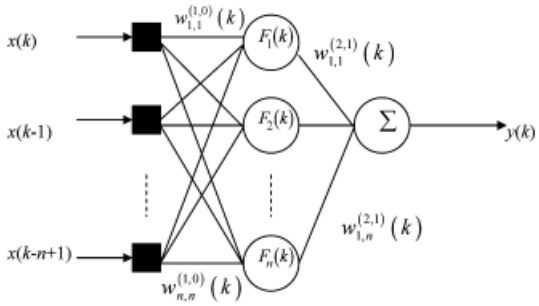

Figure 1 : Exemple de réseau Man et al. (2006)

On va étudier la stabilité de ce réseau. Il représente la structure classique des réseaux de neurones. Pour simplifier les notations on ne notera pas toujours la dépendance en le temps k.

Notons $S_j(k)$ la sortie de la couche cachée à l'étape k :

$$S_j(k) \quad = \quad F\left(\sum_{i=1}^{n} w_{j,i}^{(1,0)} x_i\right)$$

Avec $\forall\ j\ :\ F_j(x) = \frac{1}{1+e^{-\alpha x}}$, où $\alpha > 0$ (a priori on peut considérer n'importe quelle fonction d'activation).

En prenant la sortie d'un seul neurone de la couche cachée on retrouve un perceptron.

On note y(k) la sortie du réseau à l'étape k, d le signal de sortie attendu et e(k) = y(k) - d l'erreur du réseau.

L'objectif est de mettre à jour les poids du réseau afin que

$$e(k) \underset{k\to\infty}{\longrightarrow} O$$

S'il existe une fonction de Lyapunov V(k) telle que V(k) est positive avec un minimum unique. Si on met à jour les poids tels que $\Delta V(k) = \Delta V(k) - \Delta(k-1) < 0$, alors d'après le thèorème de Lyapunov, l'erreur temps vers 0. De plus, le minmum de la fonction de Lyapunov coïncide avec le point fixe du système dynamique associé.

Cette dernière propriété indique que pour obtenir la procédure de mise à jour des poids, il suffit donc de choisir une fonction de l'erreur, i.e. V(k) = g(e(k)), qui vérifie les propriétés d'une fonction de Lyapunov. Cette fonction correspondra à la perte que l'on cherche à minimiser.

En particulier, on peut prendre (perte quadratique)

$$V(k) \quad = \quad \frac{1}{2}e(k)^2$$

Par un théorème de discrétisation du temps et l'équation différentielle vérifiée par les fonctions de Lyapunov, il existe un pas η tel que

$$\begin{aligned}\Delta W &= \frac{W(k+1)-W(k)}{(k+1)-k} \\ &= -\eta\frac{\partial V}{\partial W}\end{aligned}$$

Il faut donc s'assurer que l'on a $V(k) < 0$.

On va regarder cela dans le cadre du perceptron pour plus de simplicité.

On a les équations suivantes

$$\Delta V(k) = V(k+1) - V(k) = \frac{1}{2}\left[e(k+1)^2 - e(k)^2\right]$$

$$e(k+1) = e(k)\ \Delta e(k) \Rightarrow e(k+1)^2 = e(k)^2 + \Delta^2 e(k) + 2e(k)\Delta e(k)$$

On en déduit

$$\Delta V(k) = \Delta e(k)\left[e(k) + \frac{1}{2}\Delta e(k)\right]$$

En considérant de plus que

$$\Delta e(k) = e(k+1) - e(k) \approx \left[\frac{\partial e(k)}{\partial W}\right]^T \cdot \Delta W$$

$$\begin{aligned}\Delta V(k) &= \left[\frac{\partial e(k)}{\partial W}\right]^T \cdot \Delta W \cdot \left[e(k) + \frac{1}{2}\left[\frac{\partial e(k)}{\partial W}\right]^T \cdot \Delta W\right] \\ &= \left[\frac{\partial e(k)}{\partial W}\right]^T \cdot \left(-\eta\frac{\partial V}{\partial W}\right) \cdot \left[e(k) + \frac{1}{2}\left[\frac{\partial e(k)}{\partial W}\right]^T \cdot \left(-\eta\frac{\partial V}{\partial W}\right)\right] \\ &= \left[\frac{\partial e(k)}{\partial W}\right]^T \cdot (-\eta) \cdot e(k) \cdot \frac{\partial y(k)}{\partial W} \cdot \left[e(k) + \frac{1}{2}\left[\frac{\partial e(k)}{\partial W}\right]^T \cdot (-\eta) \cdot e(k) \cdot \frac{\partial y(k)}{\partial W}\right] \\ &= -\frac{1}{2}e(k)^2 \cdot \eta \cdot \left(\frac{\partial y(k)}{\partial W}\right)^2 \cdot \left[2 - \left(\frac{\partial y(k)}{\partial W}\right)^2\right]\end{aligned}$$

Finalement, on obtient donc en posant $\lambda = \frac{\eta}{2} \cdot \left(\frac{\partial y(k)}{\partial W}\right)^2 \cdot \left[2 - \left(\frac{\partial y(k)}{\partial W}\right)^2\right]$

$$\Delta V(k) = -\lambda \cdot e(k)^2$$

Pour vérifier $\Delta V(k) < 0$, il faut donc que $\lambda > 0$. Comme $\frac{\eta}{2}\left(\frac{\partial y(k)}{\partial W}\right)^2 > 0$, on cherche

$$2 - \eta \cdot \left(\frac{\partial y(k)}{\partial W}\right)^2 > 0 \Rightarrow \eta \cdot \left(\frac{\partial y(k)}{\partial W}\right)^2 < 2 \Rightarrow \eta < \frac{2}{\left(\frac{\partial y(k)}{\partial W}\right)^2}$$

Donc pour avoir convergence, il faut que

$$0 < \eta < \frac{2}{max\ \left(\frac{\partial y(k)}{\partial W}\right)^2}$$

Donc si η vérifie la condition ci-dessus, la formule de mise à jour des poids suivante, les poids du réseau converge vers la solution optimale :

$$W(k+1) = W(k) - \eta \frac{\partial V}{\partial W}(k)$$

Approche via l'analyse convexe

Comme vu précédement, on peut utiliser des propriétés de stabilité des systèmes dynamiques pour déterminer une façon de calculer les poids.

Cependant, si la fonction de perte que l'on cherche à minimiser est "bien choisie", on peut retrouver l'équation de mise à jour des poids en appliquant ce que l'on appelle une descente de gradient, voir par exemple Fausett (1994). En particulier, on a le théorème suivant :

Soit $f : \mathbb{R}^d \to \mathbb{R}$ une fonction convexe et différentiable telle que son gradient est une fonction de Lipschitz de constante L > 0, i.e. $\forall\ x, y\ :\ \|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$. Alors pour $t \leq 1/L$, on a

$$f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2tk}$$

Où x^* réalise le minimum de f et $x^{(k+1)} = x^{(k)} - t\nabla f(x^{(k)})$.

Démonstration : ∇f est Lipschitz continue donc $\nabla^2 f \leq LI$. Les hypothèses de régularité sur f (convexe et gradient Lipschitz) permettent d'écrire pour tout x, y :

$$\begin{aligned} f(y) &\leq f(x) + \nabla f(x) \cdot (y - x) + \frac{1}{2}\nabla^2 \|y - x\|_2^2 \\ &\leq f(x) + \nabla f(x) \cdot (y - x) + \frac{1}{2} L \|y - x\|_2^2 \end{aligned}$$

Maintenant, si $y = x^+ = x - t\nabla f(x)$, on obtient

$$\begin{aligned} f(x^+) &\leq f(x) + \nabla f(x) \cdot (x^+ - x) + \frac{1}{2} L \|y - x\|_2^2 \\ &= f(x) + \nabla f(x) \cdot (x - t\nabla f(x) - x) + \frac{1}{2} L \|x - t\nabla f(x) - x\|_2^2 \\ &= f(x) - \nabla f(x) \cdot t\nabla f(x) + \frac{1}{2} L \|t\nabla f(x)\|_2^2 \\ &= f(x) - t\|\nabla f(x)\|_2^2 + \frac{1}{2} L t^2 \|\nabla f(x)\|_2^2 \\ &= f(x) - (1 - \frac{1}{2} Lt) t \|\nabla f(x)\|_2^2 \end{aligned}$$

Finalement, comme $t \leq 1/L$, $-(1 - \frac{1}{2}Lt) \leq -\frac{1}{2}$

D'où

$$f(x^+) \leq f(x) - \frac{1}{2} t \|\nabla f(x)\|_2^2$$

Enfin,

$$\begin{aligned}\sum_{i=1}^{k} f(x^{(i)}) - f(x^*) &\leq \sum_{i=1}^{k} \frac{1}{2t}\left(\|x^{(i_1)} - x^*\|_2^2 - \|x^{(i)} - x^*\|_2^2\right) \\ &\leq \frac{1}{2t}\left(\|x^{(0)} - x^*\|_2^2 - \|x^{(k)} - x^*\|_2^2\right) \\ &\leq \frac{1}{2t}\|x^{(0)} - x^*\|_2^2\end{aligned}$$

et comme $f(x^{(i)})$ est décroissant en i :

$$\begin{aligned}f(x^{(k)}) - f(x^*) &\leq \frac{1}{k}\sum_{i=1}^{k} f(x^{(i)}) - f(x^*) \\ &\leq \frac{\|x^{(0)} - x^*\|_2^2}{2tk}\end{aligned}$$

Ainsi dans le cas de fonctions convexes (ce qui est le cas pour la perte quadratique notamment) on a un moyen de déterminer une valeur adaptée pour le pas d'apprentissage.

Aujourd'hui néanmoins, avec les grandes masses de données utilser cet algorithme de mise à jour des poids n'est pas assez efficace, la convergence est trop lente.

Pour cela, la méthode la plus courrante actuellement est la descente de gradient stochastique Robbins and Monro (1951). Elle s'applique à ce qui s'appelle en statistique les M-estimateurs. Ce sont des cas ou l'estimateur recherché réalise le minimum d'une fonction de la forme :

$$L(W) = \sum_{i=1}^{n} L_i(W)$$

A nouveau la perte quadratique vérifie ces conditions. La méthode de descente de gradient prend alors la forme suivante

$$W^+ = W - t\nabla L(W) = W - t\sum_{i=1}^{n} \nabla L_i(W)$$

L'algorithme de descente du gradient stochastique s'écrit alors

- Répeter jusqu'à convergence
  - Mélanger au hasard les indices des gradients
  - pour i allant de 1 à n mettre à jour les poids W

Cette méthode converge plus vite et est plus rapide à calculer. En effet contrairement à la descente de gradient classique, ici on ne calcule pas le gradient de la fonction en entier mais le gradient d'une "sous fonction" ce qui est à priori plus simple.

L'algorithme de descente de gradient stochastique est un algorithme parmis d'autres, voir par exemple Pierre et al. (2018). Ils ont chacun leurs avantages et inconvénients et faire une liste même partielle n'apporterait pas grand chose.

Néanmoins, une version de la descente de gradient stochastique nommée AdaGrad Duchi et al. (2011) est très intéressante. En effet bien que destinée à résoudre des problèmes convexes, elle a montré des résultats dans des problèmes non-convexes Gupta et al. (2014). De plus contrairement aux méthodes présentés précédement, elle a un taux d'apprentissage qui n'est pas constant.

Si on conserve les notations précédentes, on obtient que

$$W^{+} = W - tdiag(G)^{-\frac{1}{2}} \cdot diag(G)$$

Avec pour la t-ième itération $G = \sum_{\tau}^{t} g_\tau g_\tau^t$ et $g_\tau = \nabla L_i(W^{(\tau)})$

De plus en plus de nouvelles méthodes sont développé et l'une de celle qui apportent dans certains cas de réel progrès dans la phase d'optimisation est la méthode ADAM Kingma and Ba (2014).