Réseau de neurones en actuariat

Projet de recherche

Nicolas Bellemare

Baccalauréat en actuariat

Québec, Canada

Résumé

<Texte du résumé en français. Obligatoire.>

Table des matières

Liste des tableaux

Liste des figures

Remerciements

<Texte des remerciements en prose.>

Introduction

<Texte de l'introduction. La thèse ou le mémoire devrait normalement débuter par une introduction. Celle-ci est traitée comme un chapitre normal, sauf qu'elle n'est pas numérotée.>

Chapitre 1

Théorie des réseaux de neurones

1.1 Apprentissage supervisé

La section présente est basée sur le chapitre 2 de James et collab. (2013) et le chapitre 2 de Hastie et collab. (2009).

Les réseaux de neurones sont des modèles qui peuvent s'appliquer tant à l'apprentissage supervisé qu'à l'apprentissage non-supervisé. On résume dans cette section les bases de l'apprentissage supervisé pour la suite.

Soit $X = (X_1, \dots X_p)$, un vecteur de p variables aléatoires et Y, une variable que l'on veut prédire. On suppose qu'il existe une relation entre Y et X telle que,

$$Y = f(X) + \varepsilon,$$

où f est une fonction quelconque et ε est un terme d'erreur qui est indépendant de X. L'apprentissage supervisé vise à estimer f pour pouvoir prédire et/ou expliquer Y à partir de X. On veut utiliser les données pour estimer une fonction f qui soit utile, c'est-à-dire, une fonction qui permet de bien estimer Y à partir de nouvelles données.

Un algorithme d'apprentissage supervisé est capable de modifier ses paramètres internes en réponse à une fonction objective. En d'autres termes, il apprend à partir des erreurs commises. La fonction objective mesure donc la différence entre la prédiction, $\hat{f}(x)$, et la vraie valeur de Y. On dénote cette fonction par

$$\mathscr{L}\left(Y, \hat{f}(X)\right) = \left(\hat{f}(X) - Y\right)^2.$$

1.1.1 Compromis biais-variance

Un des enjeux principaux en apprentissage statistique est le compromis biais-variance. On veut un modèle qui soit le plus précis possible tout en étant le moins variable possible. Par contre, lorsqu'on diminue le biais, la variance de l'estimateur finit toujours par augmenter : on doit trouver le compromis. On illustre cet enjeu par l'erreur quadratique espérée :

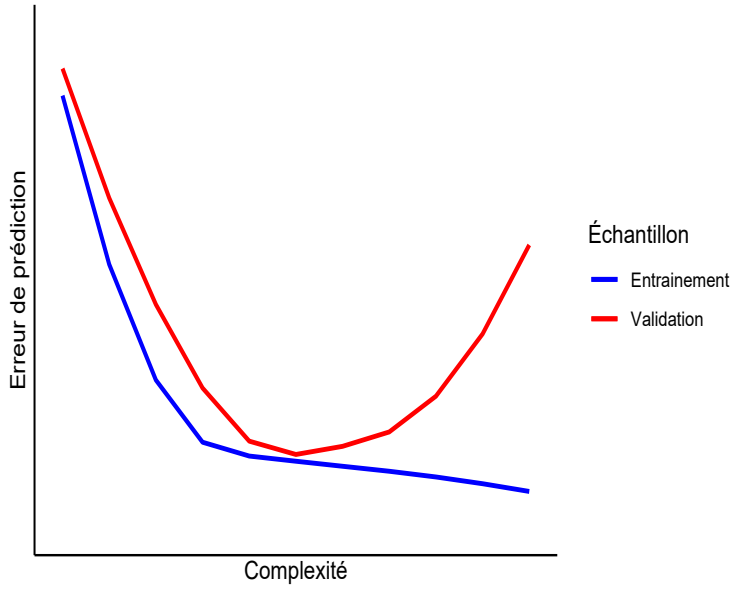
$$\mathrm{E}\left[\left\{Y_0 - \hat{f}(X)\right\}^2\right] = \mathrm{Biais}^2\left\{\hat{f}(X)\right\} + \mathrm{var}\left\{\hat{f}(X)\right\} + \sigma^2,$$

où $\sigma^2 = \mathrm{var}(\varepsilon)$, l'erreur irréductible.

On ne peut pas diminuer l'erreur irréductible, comme son nom l'indique, puisqu'elle fait partie de tout processus aléatoire. Toutefois, on peut diminuer l'erreur réductible (biais + variance) en diminuant ou en augmentant la complexité du modèle. La complexité du modèle se traduit par le nombre de paramètres qu'il faut estimer. Plus on a de paramètres, plus le modèle est en mesure de faire des prédictions exactes sur les données d'entrainement. En revanche, la variance sera plus élevée, car le modèle aura appris les caractéristiques des observations qui sont propres à celles-ci.

Pour être en mesure de surveiller le compromis, on sépare les données en trois échantillons distincts : l'échantillon d'entrainement, l'échantillon de validation et l'échantillon de test. L'échantillon d'entrainement $\mathscr{D}$ est celui sur lequel on entraine le modèle ; il sert à ajuster les paramètres. L'échantillon de test $\mathscr{T}$ permet de mesurer la performance du modèle et ainsi le comparer avec d'autres. L'échantillon de validation $\mathscr{V}$ permet de faire la sélection d'hyperparamètres et c'est à partir de celui-ci qu'on surveille si le modèle ne surajuste pas les données d'entrainement.

FIGURE 1.1 – Erreur de prédiction en fonction de la complexité du modèle

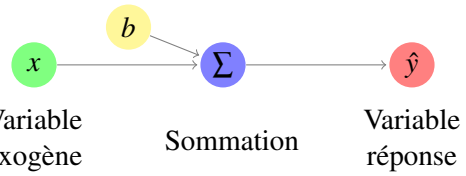


La Figure 1.1 montre un example de l'erreur de prédiction en fonction de la complexité d'un modèle. On voit que plus la complexité augmente, plus l'erreur de prédiction sur les données d'entrainement diminue. On remarque, aussi, que l'erreur de prédiction sur les données de validation diminue lorsqu'on augmente la complexité pour ensuite augmenter. Lorsque le modèle atteint le point où l'erreur de prédiction sur l'échantillon de validation augmente, on dit que le modèle surajuste l'échantillon d'entrainement. Il ne généralise pas bien pour des données qu'il n'a pas vu. L'objectif devient donc de trouver une combinaison des paramètres à estimer et de la complexité du modèle pour minimiser la fonction objective sur l'échantillon de validation.

1.2 Architecture d'un réseau de neurones

Malgré qu'il existe plusieurs types de réseau de neurones, la section présente ainsi que l'analyse de données subséquente sont basées sur la forme la plus simple de réseau de neurones, soit le perceptron ou réseau de neurones à propagation directe (« feedforward neural network »). Il existe deux formes de perceptron : le perceptron simple et le perceptron multicouche. La différence entre les deux formes réside dans le nombre de couche cachée. Le perceptron simple a une seule couches cachées, tandis que le perceptron multicouche en a plusieurs. Pour bien comprendre le perceptron, on compare son fonctionnement avec des modèles plus simples. On illustre les modèles de régression linéaire simple et multiple et le modèle de régression logistique pour expliquer la progression qui nous amènera au réseau de neurones.

FIGURE 1.2 – Modèle de régression linéaire simple



Un modèle de régression linéaire simple est une somme de la variable exogène. On suppose que la relation entre Y et X est de la forme suivante :

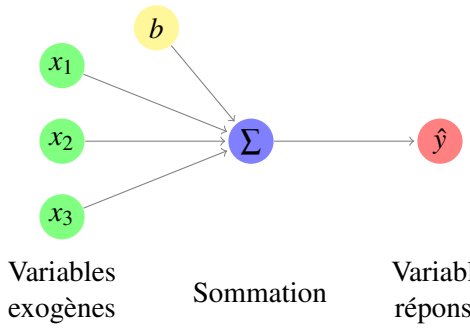
$$Y = b + \omega X + \varepsilon.$$

On estime Y par

$$\hat{Y} = \hat{f}(X) = \hat{b} + \hat{\omega} X.$$

Ce modèle utilise l'information qui est contenue dans la variable exogène pour prédire la variable réponse. Pour ce faire, on calcule $\hat{b}$ et $\hat{\omega}$ de façon à minimiser l'erreur quadratique de prévision. On illustre ce modèle à l'aide de la Figure 1.2. La partie verte de la figure illustre l'information qui entre dans le modèle, la partie bleue est le traitement de l'information et la partie rouge est la prévision du modèle. On voit que la partie bleue est en fait la fonction $\hat{f}$. Le terme de la partie jaune est le biais.

FIGURE 1.3 – Modèle de régression multiple



Dans le cas de la régression linéaire multiple, on a plusieurs variables exogènes. On fait une sommation pondérée de toutes ces variables pour pouvoir prédire la variable réponse. La relation entre Y et X est

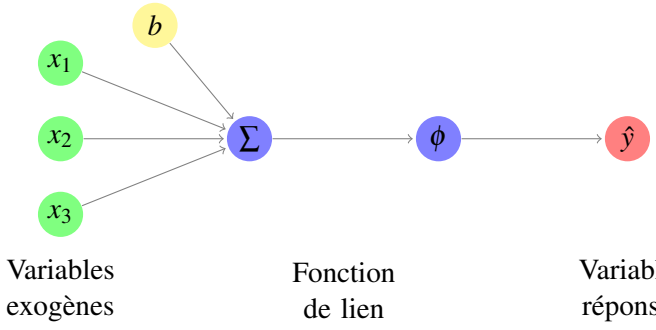
$$Y = b + \sum_{i=1}^{n} \omega_i X_i + \varepsilon.$$

On estime Y par

$$\hat{Y} = \hat{f}(X) = \hat{b} + \sum_{i=1}^{n} \hat{\omega}_i X_i.$$

La Figure 1.3 illustre un cas avec 3 variables exogènes. On y voit le même mécanisme qu'avec la régression linéaire simple, c'est-à-dire, on combine l'information, on la traite et puis on prédit un résultat. La différence est que l'information a plus d'une dimension. On doit donc estimer un paramètre pour chaque variable exogène et un paramètre pour le biais. Il s'en dégage une structure qui est bien présente dans les modèles d'apprentissage statistique.

FIGURE 1.4 – Régression logistique

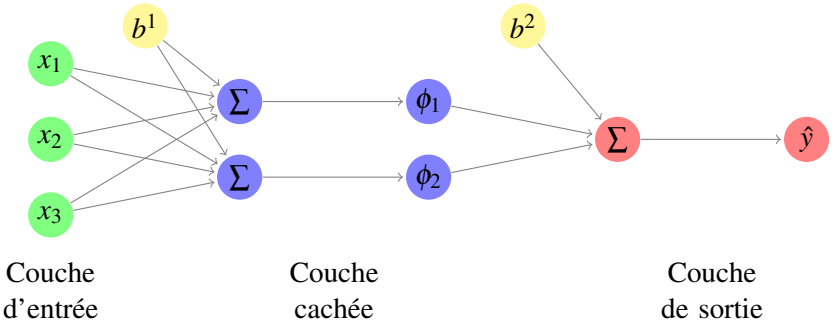


La régression logistique, quant à elle, ajoute une transformation non-linéaire à la combinaison linéaire des variables exogènes. Ainsi, la variable prédite peut être contenue dans l'espace $[0,1]$. La Figure 1.4 ajoute donc ϕ au traitement de l'information. Le modèle est

$$Y = \frac{e^{b+\sum_{i=1}^{n} \omega_i X_i}}{1 + e^{b+\sum_{i=1}^{n} \omega_i X_i}}.$$

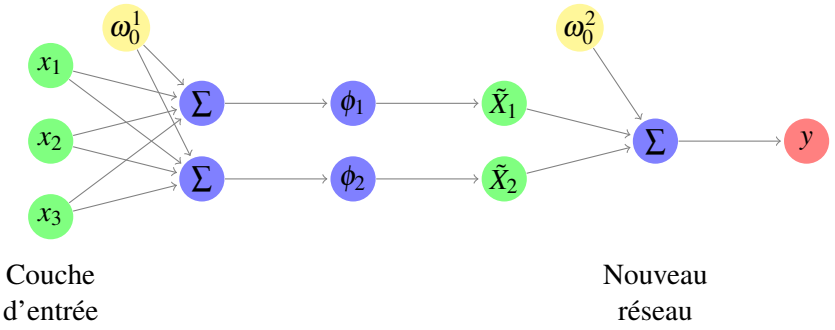
La relation entre Y et X est ainsi non-linéaire. On peut voir la régression logistique comme étant un réseau de neurones avec une seule couche cachée et un seul neurone dans la couche cachée.

FIGURE 1.5 – Réseau de neurones avec une couche cachée [1]



Les trois exemples précédents permettent de voir que les réseaux de neurones ne sont en fait qu'une généralisation de la régression linéaire. En effet, comme le montre la Figure 1.5, la partie de traitement de l'information(en bleu) est constituée de plusieurs neurones(deux neurones dans cet exemple). À chaque neurone, on combine linéairement les variables exogènes et on applique la fonction d'activation ϕ. Ensuite, la combinaison linéaire de la réponse de chacun de ces neurones est utilisée pour prédire la variable réponse du réseau. Chaque résultat des fonctions ϕ deviennent à leur tour de nouvelles variables entrant dans un réseau. Illustrons ceci par un exemple.

FIGURE 1.6 – Décomposition d'un réseau en une suite de deux réseaux

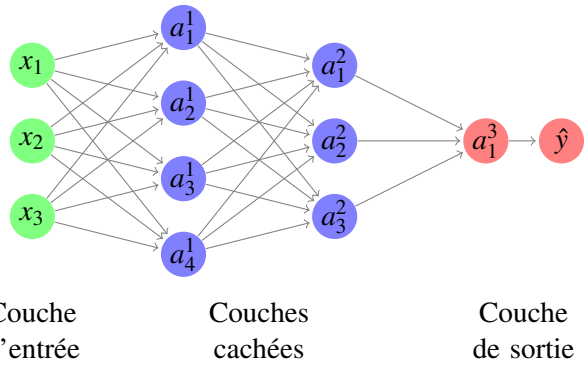


La Figure 1.6 montre que les fonctions non-linéaires ϕ_j créent une nouvelle représentation de l'information. Cette représentation est alors combinée linéairement et traitée par un autre réseau. On voit que l'on peut reproduire cette séquence un nombre indéfini de fois. Par mesure de simplicité et pour ne pas saturer les graphiques, on résume chaque neurone par $a_j^1 = \phi_j\left(b^1 + \sum_{i=1}^{n} \omega_i^1 X_i\right)$, pour le $j$ième neurone de la première couche cachée, et par $a_j^l = \phi\left(b_j^l + \sum_{k=1}^{q_{k-1}} \omega_{j,k}^l a_k^{l-1}\right)$, pour le $j$ième neurone de

1. Ce graphique est une adaptation de la réponse de l'utilisateur *gvgramazio* sur https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz

la $l$ième couche cachée. On a q_k neurones pour la $k$ième couche cachée. La Figure 1.7 illustre un réseau de neurones en considérant cette notation avec deux couches cachées de quatre et trois neurones, respectivement. On remarque une neurone supplémentaire directement lié à la prévision, soit a_1^3 dans notre exemple, ou z_1^{l+1} pour un réseau avec k couches cachées et une seule prédiction. Ce dernier neurone permet de faire le lien entre entre le réseau et la prévision. C'est la fonction de régression.

FIGURE 1.7 – Réseau de neurones avec deux couches cachées



On obtient la formule suivante pour un réseau de neurones avec une couche cachée de q_1 neurones et avec la fonction identité pour la fonction de régression, $z_1^3(x)$:

$$Y = b^2 + \sum_{j=1}^{q_1} \omega_{1,j}^2 \, \phi_j \left(b_j^1 + \sum_{k=1}^{n} \omega_{j,k}^1 \, X_k \right)$$

1.3 Phase de propagation directe(« feedforward »)

On résume la notation :

- $\omega_{j,k}^1$, le poids du lien entre le $k$ème neurone de la $l-1$ème couche au $j$ème neurone de la $l$ème couche
- b_j^l, le biais associé au $j$ème neurone de la $l$ème couche
- ϕ, une fonction d'activation quelconque
- a_j^l, l'activation du $j$ème neurone de la $l$ème couche
- z_j^l, l'intrant du $j$ème neurone de la $l$ème couche

On utilise une notation sous forme de matrice pour la suite. Ainsi, on a la matrice de poids suivante pour relier la couche $l-1$ à la couche l

$$\omega^l = \begin{bmatrix} \omega^l_{1,1} & \omega^l_{1,2} & \cdots & \omega^l_{1,q_l} \\ \omega^l_{2,1} & \omega^l_{2,2} & \cdots & \omega^l_{2,q_l} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^l_{q_l,1} & \omega^l_{q_l,2} & \cdots & \omega^l_{q_l,q_l} \end{bmatrix}.$$

On a que

$$z^l_j = \sum_k \omega^1_{j,k} a^{l-1}_j + b^l_j$$

et

$$z^l = \omega^l a^{l-1} + b^l.$$

Alors, la matrice des intrants de la $l$ème couche avec q_l neurones est

$$z^l = \begin{bmatrix} z^l_1 \\ z^l_2 \\ \vdots \\ z^l_{q_l} \end{bmatrix}.$$

Ainsi, on peut réécrire

$$a^l_j = \phi\left(\sum_k \omega^1_{j,k} a^{l-1}_j + b^l_j\right)$$
$$a^l = \phi\left(\omega^l a^{l-1} + b^l\right).$$

Ce qui donne pour la matrice des activations de la $l$ème couche de q_l neurones

$$a^l = \begin{bmatrix} a^l_1 \\ a^l_2 \\ \vdots \\ a^l_{q_l} \end{bmatrix}.$$

1.4 Fonction d'activation

La fonction d'activation permet de créer une représentation non-linéaire de l'information. Elle doit absolument être non-linéaire, sinon le réseau serait une combinaison linéaire de combinaisons linéaires.

Elle permet ainsi d'apprendre des interactions complexes entre les variables exogènes. Les fonctions d'activation les plus communes sont

$$\phi(x) = \begin{cases} \frac{e^x}{1+e^x} & \text{, sigmoid} \\ \tanh(x) & \text{, tangente hyperbolique} \\ \mathbf{1}_{\{x \geq 0\}} & \text{, escalier} \\ x\mathbf{1}_{\{x \geq 0\}} & \text{, « Rectified Linear unit », Relu} \end{cases}$$

1.5 Rétro-propagation(« backpropagation »)

Cette section est inspirée de la série sur les réseaux de neurones de la page Youtube Sanderson et du chapitre sur le « backpropagation » de Nielsen (2015).

On aborde maintenant le mécanisme d'apprentissage du réseau de neurone à propagation directe. On dit propagation directe(« feedforward ») puisque l'information ne fait que se propager vers l'avant. Il n'y a pas de cycle dans le modèle où l'information repasse dans une partie du réseau plusieurs fois via une boucle. Un réseau dont l'information circule de cette façon est appelé un réseau de neurones récurent.

Toutefois, le réseau à propagation directe utilise la méthode de rétro-propagation (« backpropagation ») pour diffuser le signal qui lui permet d'ajuster ses paramètres. On dit rétro-propagation puisque cette méthode diffuse le signal en faisant le chemin inverse de la phase de propagation directe.

On veut que le réseau apprenne la bonne combinaison de poids et de biais pour la tâche à effectuer. Dans notre cas, la tâche est une régression. On utilise une fonction objective pour mesurer la performance du modèle à cette tâche. C'est à partir de cette fonction que le réseau va apprendre.

La méthode de rétro-propagation permet de trouver une combinaison de poids et de biais qui puisse minimiser la fonction objective. L'astuce est de calculer le gradient de cette fonction en appliquant successivement la règle de dérivation en chaîne.

La première étape consiste à calculer la sortie du réseau, $\hat{y}_i$, pour chaque observation i. Pour ce faire, on doit initialiser les paramètres de façon aléatoire. On calcule ensuite le résultat de la fonction de perte. Pour la suite du document, on utilise la déviance de Poisson :

$$\mathscr{L}(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^{n} 2y_i \left(\frac{\hat{y}_i}{y_i} - 1 - \log\left(\frac{\hat{y}_i}{y_i} \right) \right).$$

Cette fonction varie seulement selon la valeur des paramètres du réseau. En effet, elle prend comme argument y_i et $\hat{y}_i$, mais y_i est fixe. On rappelle aussi que $\hat{y}_i = \hat{f}(\mathbf{X}_i)$. Ainsi, pour faire varier $\mathscr{L}$, il

faut faire varier $\hat{f}$. Considérant que $\mathbf{X}_i$ est fixe, la seule façon de faire varier $\hat{f}$ est de faire varier ses paramètres internes $w_{j,k}^l$ et b_j^l.

Le paragraphe précédent décrit l'intuition derrière la méthode de rétro-propagation. On détermine comment la fonction de perte varie par rapport aux paramètres indirectement. $\mathscr{L}$ est fonction des paramètres du réseau. Le gradient de cette fonction, $\nabla\mathscr{L}$, détermine la direction dans laquelle un changement aux paramètres permet d'augmenter le plus rapidement la valeur de $\mathscr{L}$. On veut donc déterminer chaque élément de $\nabla\mathscr{L}$,

$$\nabla\mathscr{L} = \begin{bmatrix} \dfrac{\partial\mathscr{L}}{\partial\omega^{(1)}} \\ \dfrac{\partial\mathscr{L}}{\partial b^{(1)}} \\ \vdots \\ \dfrac{\partial\mathscr{L}}{\partial\omega^{(L)}} \\ \dfrac{\partial\mathscr{L}}{\partial b^{(L)}} \end{bmatrix}.$$

On veut trouver les dérivées partielles

$$\frac{\partial\mathscr{L}}{\partial\omega_{j,k}^{(l)}} \quad \text{et} \quad \frac{\partial\mathscr{L}}{\partial b_j^{(l)}}, \quad \text{pour chaque} \quad j,k,l.$$

Celles-ci nous informent sur la sensibilité de $\mathscr{L}$ par rapport à un petit changement de $\omega_{j,k}^{(l)}$ et de $b_j^{(l)}$, respectivement. Ainsi, à l'aide de chaque élément de $\nabla\mathscr{L}$, on peut ajuster les paramètres dans la direction qui permet de diminuer le plus rapidement $\mathscr{L}$. On applique

$$\omega_{j,k}^{(l)} \mapsto \omega_{j,k}^{(l)} - \eta\frac{\partial\mathscr{L}}{\partial\omega_{j,k}^{(l)}}$$

et

$$b_j^{(l)} \mapsto b_j^{(l)} - \eta\frac{\partial\mathscr{L}}{\partial b_j^{(l)}},$$

où η est le taux d'apprentissage. Celui-ci permet d'ajuster les paramètres proportionnellement à leur dérivée partielle. Autrement dit, on ajuste chacun des paramètres par un pas proportionnel à leur importance relative d'une variation de $\mathscr{L}$ dans la direction qui mène le plus rapidement au minimum local.

Une méthode qui permet de calculer chaque $\dfrac{\partial \mathscr{L}}{\partial \omega_{j,k}^{(l)}}$ est de calculer

$$\frac{\partial \mathscr{L}}{\partial \omega_{j,k}^{l}} \approx \frac{\mathscr{L}\left(\omega + \varepsilon e_{j,k}^{l}\right) - \mathscr{L}(\omega)}{\varepsilon}.$$

où $\varepsilon > 0$ est petit, et $e_{j,k}^{l}$ est un vecteur unitaire dans la direction, j,k,l. Cependant, cette méthode requiert de calculer $\mathscr{L}\left(\omega + \varepsilon e_{j,k}^{l}\right)$ pour chaque paramètre. Ainsi, si on a q paramètres à estimer dans le réseau, on doit faire $q+1$ phases de propagation directe. De plus, on doit ajuster plusieurs fois les paramètres avant d'atteindre un minimum local. Évidemment, le temps de calcul est très élevé pour cette méthode.

Le méthode de rétro-propagation permet de calculer efficacement ces dérivées en passant dans le réseau seulement deux fois pour ajuster tous les paramètres. Elle s'appuie sur la règle de dérivation en chaîne. Soit $a_i^L = \phi(z_i^L)$, le vecteur des activations de la couche de sortie d'un réseau avec $L-1$ couches cachées pour la $i$ème observation, et $z_i^L = (\omega_L a_i^{L-1} + b^L)$, l'intrant de la fonction d'activation a^L, alors on a que

$$\frac{\partial \mathscr{L}}{\partial \omega_{j,k}^{L}} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial z_i^L}{\partial \omega_{j,k}^{L}} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial \mathscr{L}_i}{\partial a_i^L}.$$

1.6 Hyperparamètres

Chapitre 2

<Titre du chapitre ou de l'article>

<Texte du chapitre ou de l'article.>

Conclusion

<Texte de la conclusion. Une thèse ou un mémoire devrait normalement se terminer par une conclusion placée avant les annexes, le cas échéant. La conclusion est traitée comme un chapitre normal, sauf qu'elle n'est pas numérotée.>

Annexe A

<Titre de l'annexe>

<Texte de l'annexe.>

Bibliographie

Hastie, T., R. Tibshirani et J. Friedman. 2009, *The Elements of Statistical Learning : Data mining, Inference, and Prediction*, 2e éd., Springer, New York.

James, G., D. Witten, T. Hastie et R. Tibshirani. 2013, *An introduction to statistical learning*, vol. 112, Springer.

Nielsen, M. A. 2015, *Neural networks and deep learning*, vol. 2018, Determination press San Francisco, CA, USA :.

Sanderson, G. «3blue1brown», URL \url{https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw/featured}, [En-ligne ; consulté 23 - Mars -2020].

Wuthrich, M. V. et C. Buser. 2019, «Data analytics for non-life insurance pricing», *Swiss Finance Institute Research Paper*, , no 16-68.