Proceedings of Machine Learning Research 1–14

Étude des réseaux de neurones comme outil d'apprentissage supervisé

Malik Algelly MALIK.ALGELLY@ETU.UNIGE.CH
Luca Perone LUCA.PERONE@ETU.UNIGE.CH
Juraj Rosinsky JURAJ.ROSINSKY@ETU.UNIGE.CH
Samuel Simko SAMUEL.SIMKO@ETU.UNIGE.CH

Abstract

Nous proposons une nouvelle librairie de création, modification et entraînement de réseaux de neurones, perceptrons et classifieurs k-NN, nommée **battelle**. Nous utilisons cette librairie pour étudier l'emploi des réseaux de neurones comme outil d'apprentissage supervisé, à travers la confection de classifieurs entrainés sur des bases de données connues de UCI (Dua and Graff [2017]). Nous comparons les résultats obtenus avec d'autres librairies de réseaux de neurones tels que Scikit-learn (Pedregosa et al. [2011]), ainsi que le classifieur k-NN.

Keywords: Neural Networks, Supervised Learning, Machine Learning, Artificial Intelligence.

1. Introduction

Les réseaux de neurones forment la pierre angulaire de l'intelligence artificielle moderne et de l'apprentissage profond.

Des librairies python telles que Scikit-Learn (Pedregosa et al. [2011]) proposent des interfaces modernes pour construire des réseaux de neurones. Nous allons tout d'abord expliquer les fondements théoriques derrière l'apprentissage des réseaux de neurones, pour ensuite pouvoir analyser leur performance.

2. Théorie des Réseaux de Neurones

2.1. Définitions et notations

Soit l'échantillon $x \in \mathbb{R}^n$, auquel est associé le label $y \in \mathbb{R}^m$.

Soit un réseau de neurones possèdant l couches numérotées, ou "layers". Soient $w_{ji}^{(k)}$ les poids du neurone entre le $i$ème neurone de la couche $k-1$ et le $j$ième neurone de la couche k. Soient $b_{ji}^{(k)}$ le biais du $i$ème neurone de la couche $k-1$. Soit $g^{(k)}$ la fonction d'activation associée à la couche k.

2.2. Algorithmes

Le réseau forme une prédiction grâce à l'algorithme de propagation (1).

Au fur et à mesure des prédictions, le réseau est entrainé grâce à l'algorithme de ré-

Algorithm 1 Propagation

1. Pour $n = 1$ jusqu'à l
   (a) $h_j^{(n)} := \sum_i w_{ji}^{(n)} x_i^{(n-1)} + b_j^{(n-1)}$
   (b) $x_j^{(n)} := g^{(n)}(h_j^{(n)})$
2. Retourner le vecteur $x^{(l)}$

tropropagation (2) On utilise un facteur λ, que l'on nomme le "learning rate").

Algorithm 2 Rétropropagation

1. $e_i^l := g^{(l)\prime}(h_i^l)(y_i - x_i^{(l)})$
2. (a) Pour $k = l$ jusqu'à 2
      i. $e_j^{(k-1)} := g'^{(k-1)}(h_j^{k-1}) \sum_i w_{ij}^{(k)} e_i^{(k)}$
   (b) Pour $k = 1$ jusqu'à l
      i. $w_{ij}^{(l)} = w_{ij}^{(l)} - \lambda e_i^{(l)} x_j^{(l-1)}$
      ii. $b_{ij}^{(l)} = b_{ij}^{(l)} - \lambda e_i^{(l)}$

Cet algorithme modifie les poids de façon à minimiser la perte, ou "loss", définie comme

$$\mathcal{L} := \frac{1}{2} \sum_i (x^{(l)} - y_i)^2.$$

Avec y le vecteur label, et $x^{(l)}$ le vecteur retournée par l'algorithme de propagation.

2.3. Explication de l'algorithme de rétropropagation

Selon le cours (Marchand-Maillet [2021]), on a

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \frac{\partial \mathcal{L}}{\partial x_j^{(l)}} \frac{\partial x_j^{(l)}}{\partial h_j^{(l)}} \frac{\partial h_j^{(l)}}{\partial w_{ji}^{(l)}}.$$

Et

$$\frac{\partial h_j^{(l)}}{\partial w_{ji}^{(l)}} = \sum_k w_{jk}^{(l)} \frac{\partial x_k^{(l-1)}}{\partial w_{ji}^{(l)}} + b_j^{(l)} = x_j^{(l-1)}.$$

$$\frac{\partial \mathcal{L}}{\partial x_j^{(l)}} = (x_j^{(l)} - y_j).$$

$$\frac{\partial x_j^{(l)}}{\partial h_j^{(l)}} = g'^{(l)}(h_j^{(l)}).$$

Donc, par définition de $e_i^l := g^{(l)\prime}(h_i^l)(y_i - x_i^{(l)})$,

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = e_j^{(l)} x_j^{(l-1)}.$$

Remarquons que

$$\frac{\partial h_j^{(l)}}{\partial b_j^{(l)}} = e_j^{(l)}$$

car

$$\frac{\partial h_j^{(l)}}{\partial b_j^{(l)}} = 1$$

.

Par principe de descente en gradient, nous pouvons ainsi soustraire une valeur proportionnelle à $e_j^{(l)} x_j^{(l-1)}$ au poids $w_{ji}^{(l)}$, et faire ainsi baisser la perte $\mathcal{L}$.

Pour les couches inférieures, remarquons que

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \frac{\partial h_j^{(l)}}{\partial w_{ji}^{(l)}} \frac{\partial \mathcal{L}}{\partial h_j^{(l)}} = x_i^{(l-1)} \frac{\partial \mathcal{L}}{\partial h_j^{(l)}} \quad (1)$$

Et

$$\frac{\partial \mathcal{L}}{\partial h_j^{(l)}} = \frac{\partial x_j^{(l)}}{\partial h_j^{(l)}} \frac{\partial \mathcal{L}}{\partial x_j^{(l)}} = g'^{(l)}(h_j^{(l)}) \frac{\partial \mathcal{L}}{\partial x_j^l} \quad (2)$$

Et finalement

$$\frac{\partial \mathcal{L}}{\partial x_j^{(l)}} = \sum_k \frac{\partial h_k^{(l-1)}}{\partial x_j^{(l)}} \frac{\partial \mathcal{L}}{\partial h_k^{(l-1)}} = \sum_k w_{kj}^{(l)} \frac{\partial \mathcal{L}}{\partial h_k^{(l-1)}} \quad (3)$$

On peut donc calculer les gradients $\frac{\partial \mathcal{L}}{\partial w_{jk}^{(n)}}$ pour toutes les couches, en chaînant les équations 2 et 3 pour calculer les gradients itérativement pour $(l-1), (l-2), \ldots$

En remplaçant avec

$$e_j^{(k-1)} := g'^{(k-1)}(h_j^{k-1}) \sum_i w_{ij}^{(k)} e_i^{(k)}$$

On tombe sur

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(k)}} = x_i^{(l-1)} e_j^k$$

Car $e_j^{(l)} = \frac{\partial \mathcal{L}}{\partial h_j^{(l)}}$.

Pour les gradients en fonction des biais, on peut oublier le facteur $x_i^{l-1} \frac{\partial h_j^{(l)}}{\partial b_j^{(l)}} = 1$.

L'algorithme 2 va donc correctement entraîner les poids et bias.

3. Implémentation

Nous avons développé, dans le langage Python, une librairie de réseaux de neurones, nommée **Battelle** conçue pour faciliter la construction de réseaux simples. La syntaxe de création de réseaux a été inspirée de celle de Keras (Chollet et al. [2015]).

L'utilisateur peut spécifier le nombre de couches, ainsi que le nombre de neurones de chaque couche. Il peut également choisir la dimension d'entrée du réseau, ainsi que la fonction d'agrégation utilisée au sein de chaque couche (entre la fonction sigmoïde et la fonction tanh). Il peut aussi définir la sienne.

Il peut ensuite entrainer son réseau grâce à l'algorithme de rétropropagation, et effectuer des mesures de performance avec quatre types de métriques (accuracy, recall, precision, F1-score).

Battelle possède trois sous-modules :

- **Battelle.nn**, pour le réseaux de neurones ;
- **Battelle.knn**, pour le classifieur k-nn ;
- **Battelle.perceptron**, pour le perceptrons.

Une documentation a été crée et est disponible sur le site `battelle.readthedocs.io`.

4. Visualisation des poids du réseau sous forme de matrices

Pour chaque test, on va préciser tous les paramètres utilisés, c'est à dire : Dataset / Neural-Net shape / Train size / Learning rate / Activation function / Min step. Voici ce que chacun de ces termes veut dire :

- Dataset est le dataset que l'on a utilisé pour le test.
- Neural-Net shape est la forme du réseau de neurones que l'on a utilisée.
- Train size est la proportion du dataset que l'on a utilisée pour entraîner le réseau de neurone.

- Learning rate est le learning rate utilisé dans le réseau de neurone.

- Activation function est la fonction d'activation que l'on a utilisée dans le modèle. Cette fonction change en fonction du type de problème que l'on veut résoudre. Pour les problèmes de classifications , il est recommandé d'utiliser la fonction sigmoid ($f(x) = \frac{1}{1+e^{-x}}$) ou tanh ().

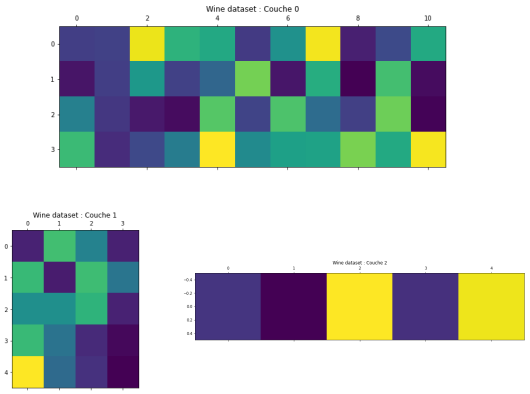
- Min step est le pas minimal que l'on demande pour continuer à entraîner le réseau de neurones. Si le pas devient trop petit, cela veut dire que le réseau de neurone se trouve très près d'un minimum local, donc très proche du meilleur résultat obtenable de cette manière.

- Total EPOCH est le nombre total d'itérations exécutées.

- Final error est l'erreur finale du réseau de neurones.

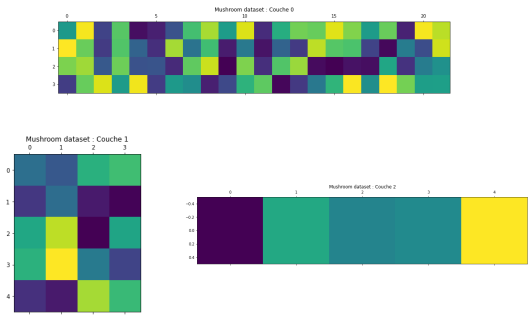
Pour visualiser les poids des neurones, les valeurs absolues des poids ont été organisées en matrices par couche du réseau de neurone. Ensuite, à chaque poids a été assignée une couleur : plus la couleur est bleue, plus le poids est petit (et donc le neurone n'a pas été considéré comme important) et plus la couleur est jaune, plus le poids est grand (et donc important). Voici les matrices pour le dataset des vins blancs :

Dataset	Mushroom
Neural-Net shape	(4,5,1)
Train size	0.8
Learning rate	0.3
Activation function	sigmoid
Min step	0.1
Total EPOCH	36
Final error	$4,423 \cdot 10^{-6}$



Ici, on peut voir que le premier neurone de la Couche 0 a trouvé l'acidité citrique ainsi que la densité des vins très pertinente. En revanche, le quatrième neurone de la Couche 0 a donné beaucoup d'importance à la teneur en chlorures et en alcool.

Dataset	Mushroom
Neural-Net shape	(4,5,1)
Train size	0.8
Learning rate	0.3
Activation function	sigmoid
Min step	0.1
Total EPOCH	20
Final error	$6,287 \cdot 10^{-5}$



Ici, on peut voir que tous les neurones de la Couche 0 ont trouvé important de prendre en compte le type de surface du chapeau des champignons.

5. Perceptron

5.1. Implémentation

Pour l'implémentation du perceptron, nous avons suivi l'algorithme proposé en cours (Marchand-Maillet [2021]). Cependant, vous avons décidé d'utiliser une autre fonction de prédiction. La fonction Φ_Θ est remplacé par une fonction renvoyant 1 si Φ_Θ est positive et 0 sinon. Nous avons constaté que la majorité des implémentations sur le net font de cette manière (par exemple Brownlee [2016])

Concernant l'implémentation, nous avons réalisé une classe python nommée Perceptron qui contient un champ weight, avec un getter et un setter, ainsi que deux méthodes :

- train : cette méthode nous servira à entraîner notre Perceptron. Elle prend en argument un tableau contenant nos données, un tableau des labels associés à ces données, un scalaire représentant le pas d'apprentissage, et un nombre d'itération d'apprentissage. Elle modifira les weights au fur et à mesure des itérations.
- predict : cette méthode nous permet de prédire un label en fonction d'une donnée de test. Elle prend en argument une donnée sous forme de tableau et renvoie la prédiction 1 ou 0.

Comme vu au cours, le biais est pris en compte grâce à la méthode des données augmentées. En effet, en rajoutant une caractéristique égale à 1 dans les données, le poids correspondant servira de biais et le seuil de la fonction de prédiction pourra ainsi être égal à 0.

5.2. Vérification de l'implémentation

Comme expliqué dans le cours, un Perceptron permet uniquement de séparer des données linéairement. Pour tester notre implémentation, on peut modéliser différents ensembles étant linéairement ou non-linéairement séparables :

- Soit les x_i, un ensemble fini de points appartenant à l'intervalle $[-1, 1]^2$, et y_i les labels associés tel que si $x_i = (x_1, x_2)$ avec $x_1 > -0.5$ alors $y_i = 1$ et $y_i = 0$ sinon.

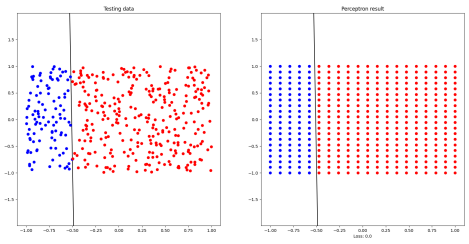


FIGURE 1: Résultat du Perceptron sur un ensemble linéairement séparable

Sur la figure 1, on constate que le perceptron permet de séparer ses données correctement car elle sont linéairement séparées. De plus, on constate une loss égal à 0.

Prenons maintenant un ensemble n'étant pas linéairement séparable :

- Soit les x_i, un ensemble fini de points appartenant à l'intervalle $[-1, 1]^2$, et y_i les labels associés tel que si $x_i = (x_1, x_2)$ avec ($x_1 > 0$ et $x_2 > 0$) alors $y_i = 1$ et $y_i = 0$ sinon.

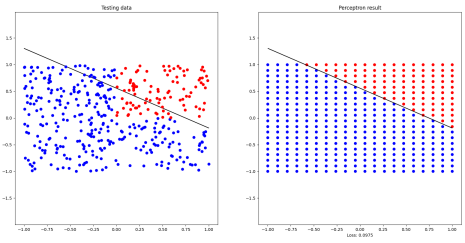


FIGURE 2: Résultat du Perceptron sur l'ensemble AND

Sur la figure 2, il est clair que ces données ne sont pas complétement linéairement séparables et que la séparation calculée par le Perceptron n'est pas optimal. Cependant, on constate que l'algorithme converge vers une séparation avec une loss de 0.0975.

Prenons maintenant un ensemble ne permettant pas au Perceptron de converger :

- Soit les x_i, un ensemble fini de points appartenant à l'intervalle $[-1, 1]^2$, et y_i les labels associés tel que si $x_i = (x_1, x_2)$ avec ($x_1 > 0$ et $x_2 < 0$) ou ($x_1 < 0$ et $x_2 > 0$) alors $y_i = 1$ et $y_i = 0$ sinon.

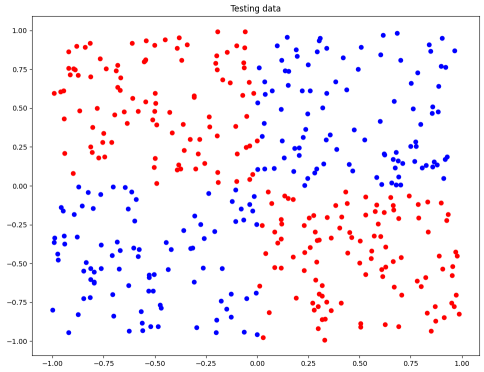


FIGURE 3: Dataset de l'ensemble XOR

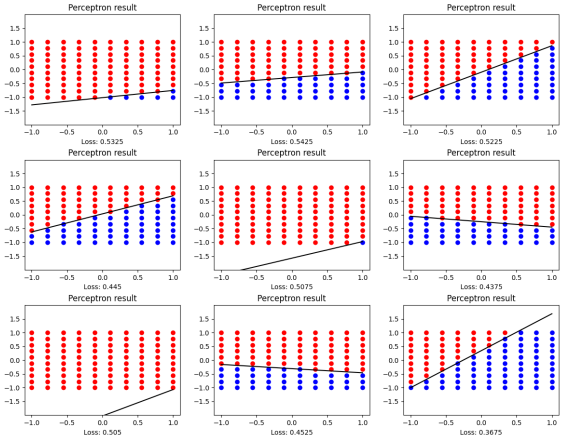


FIGURE 4: Résultat du Perceptron sur l'ensemble XOR

Sur la figure 4, on constate que le Perceptron ne permet pas de séparer ces données et qu'il ne converge pas.

6. Evolution des mesures d'évaluations

Pour répondre aux questions concernant l'évolution des mesures d'évaluation en fonction du bruit dans les données d'apprentissage ou dans les données de test, ainsi que du volume de données utilisées pour l'apprentissage, nous avons utilisé un réseau de neurones de trois couches. La première couche étant constituée de 4 neurones, la deuxième de 5 neurones et la dernière d'un neurone. De plus, nous avons utilisé la fonction d'activation 'sigmoid'. Pour finir, nous avons fixé le nombre d'itération d'apprentissage à 20.

Les différents graphiques qui suivent illustrent la moyenne des valeurs d'évaluation de cinq exécutions. A chaque exécution, les données sont remélangées afin de modifier les données sélectionnées pour l'apprentissage et celles pour le test. Les données utilisées sont celle de l'ensemble de données "Mushrooms".

6.1. Volume de données utilisées pour l'apprentissage

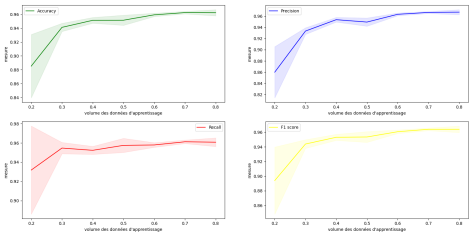


FIGURE 5: Evolution des mesures en fonction du volume de données d'apprentissage

Comme on peut le constater sur la figure 5, la valeur des mesures d'évaluation augmente en fonction de la quantité de données utilisées pour l'apprentissage. En effet, plus il y a de données à disposition lors de l'apprentissage, plus le Perceptron donnera des prédictions correctes par la suite.

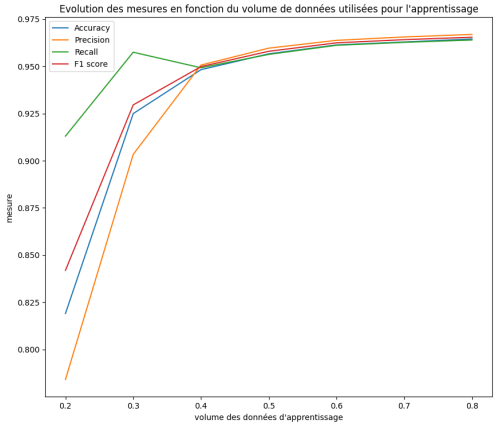


FIGURE 6: Evolution des mesures en fonction du volume de données d'apprentissage

6.2. Bruit dans les données d'apprentissage

Pour ce point, nous avons ajouté un bruit aléatoire gaussien en faisant varier l'écart-type de 10^{-3} à 10^{1}.

Sur la figure 7, on constate qu'en augmentant le bruit dans les données d'apprentissage, la valeur des mesures d'évaluation diminue. Cela s'explique par le mélange des données entre les différentes catégories. En effet, une donnée ayant une valeur à la limite de sa

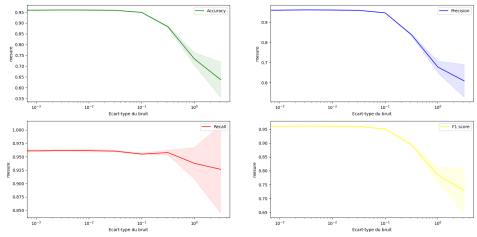


FIGURE 7: Evolution des mesures en fonction du bruit dans les données d'apprentissage

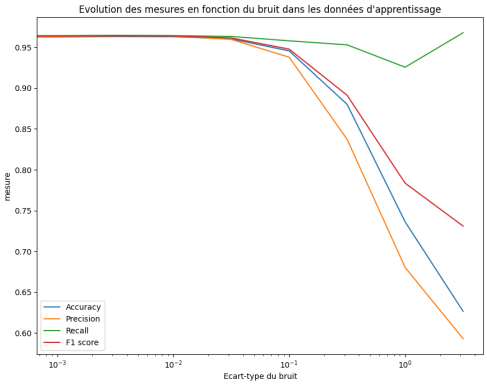


FIGURE 8: Evolution des mesures en fonction du bruit dans les données de Tests

catégorie basculera dans une autre catégorie créant ainsi de faux positif ou faux négatif. De plus, on peut voir une augmentation de la déviation standard lors des différentes exécutions lorsque l'écart-type du bruit augmente.

6.3. Bruit dans les données de Test

Comme pour le point précédant, nous avons ajouté un bruit aléatoire gaussien en faisant varier l'écart-type de 10^{-3} à 10^{1}.

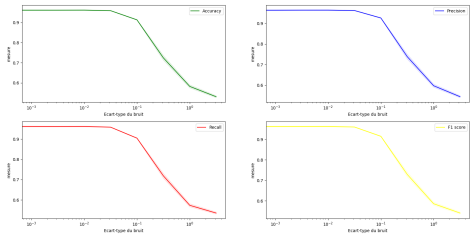


FIGURE 9: Evolution des mesures en fonction du bruit dans les données de Tests

Sur la figure 9, on constate qu'en augmentant le bruit dans les données de test, la valeur des mesures d'évaluation diminue. Cependant, on peut voir que la déviation standard lors des différentes exécutions reste du même ordre.

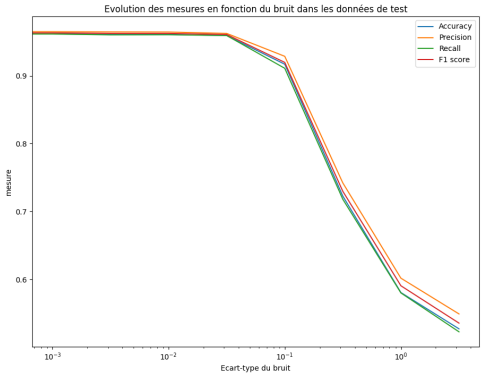


FIGURE 10: Evolution des mesures en fonction du bruit dans les données de tests

7. Impact de la complexité du réseau sur l'apprentissage

Nous avons entrainé quatre réseaux de neurones avec un taux d'apprentissage de 0.04 et la fonction d'agrégation tanh à l'aide des données représentées sur la figure 11

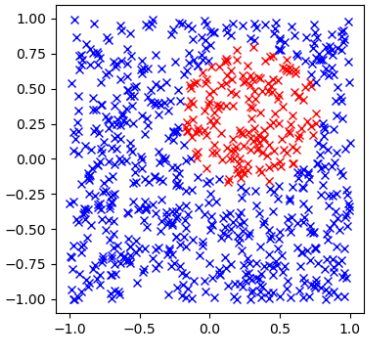


FIGURE 11: Données d'entraînement

Nous avons mesuré la précision (Precision), le rappel (Recall), l'accuracy et le F1-score entre chaque Epoch de l'entraînement, pour des réseaux possèdant 3 couches avec un nombre de neurones variable. Nous notons $[x, y]$ le réseau possèdant 3 couches de x, y et 1 neurone respectivement. 5 mesures ont été prises pour des réseaux $[1, 1]$, $[2, 2]$, $[4, 4]$, $[8, 8]$ et $[16, 16]$.

Sur la figure 12, nous avons représenté l'écart-type des mesures effectuées pour chaque réseau avec des aires de couleur. Nous pouvons apercevoir que le réseau $[1, 1]$ n'a pas su s'améliorer après la première epoch. Il classifie toutes les données dans la classe la plus présente, et obtient donc une accuracy de 0.8.

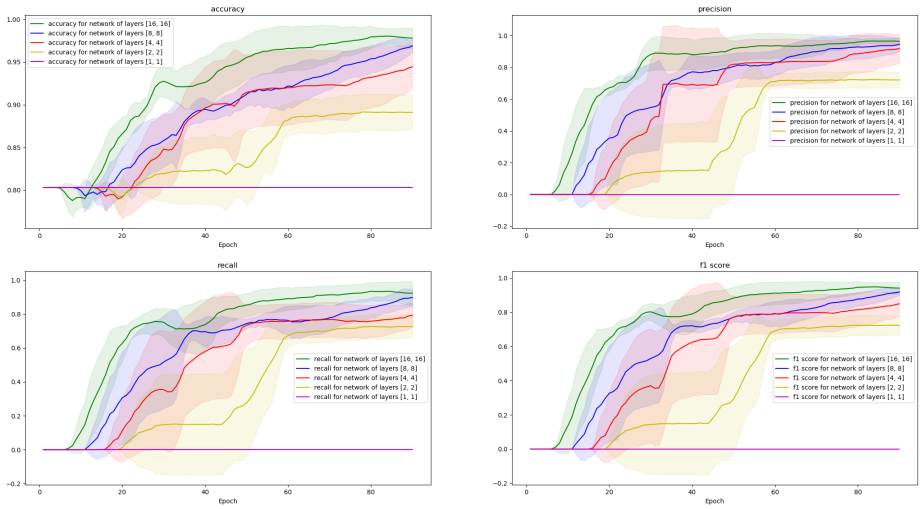


FIGURE 12: Évolution des mesures pour des réseaux de complexité variable

Nous remarquons que plus le réseau est complexe, plus les métriques s'améliorent rapidement. Un réseau complexe s'adapte plus rapidement à nos données.

Un réseau plus complexe obtient également des meilleures performances. Par exemple, le réseau $[2, 2]$ obtient une accuracy finale de 0.89 ± 0.02, tandis que le réseau $[16, 16]$ obtient une accuracy finale de 0.97 ± 0.02.

Les écarts-types des valeurs mesurées pour chaque réseau sont également moins importantes pour de plus grands réseaux.

8. Phénomène de sur-apprentissage

Pour cette section, nous utilisons un réseau de neurones avec 3 couches de 5, 4 et 11 neurones respectivement. Le taux d'apprentissage est de 0.04, et nous utilisons la fonction d'agrégation tanh.

Nous utilisons la base de données Wine Quality (P. Cortez and Reis. [2009]).

Après avoir mis à l'échelle nos données avec la méthode MinMax, Nous avons réparti nos données de façon à avoir un faible nombre de données d'entraînement (30% des données totales). Voici les spécifités du réseau de neurones :

Dataset	Wine quality
Neural-Net shape	(4,5,11)
Train size	0.3
Learning rate	0.04
Activation function	tanh
Min step	0.1
Total EPOCH	20

Nous avons ensuite entrainé notre réseau de neurones à classifier les attributs des vins

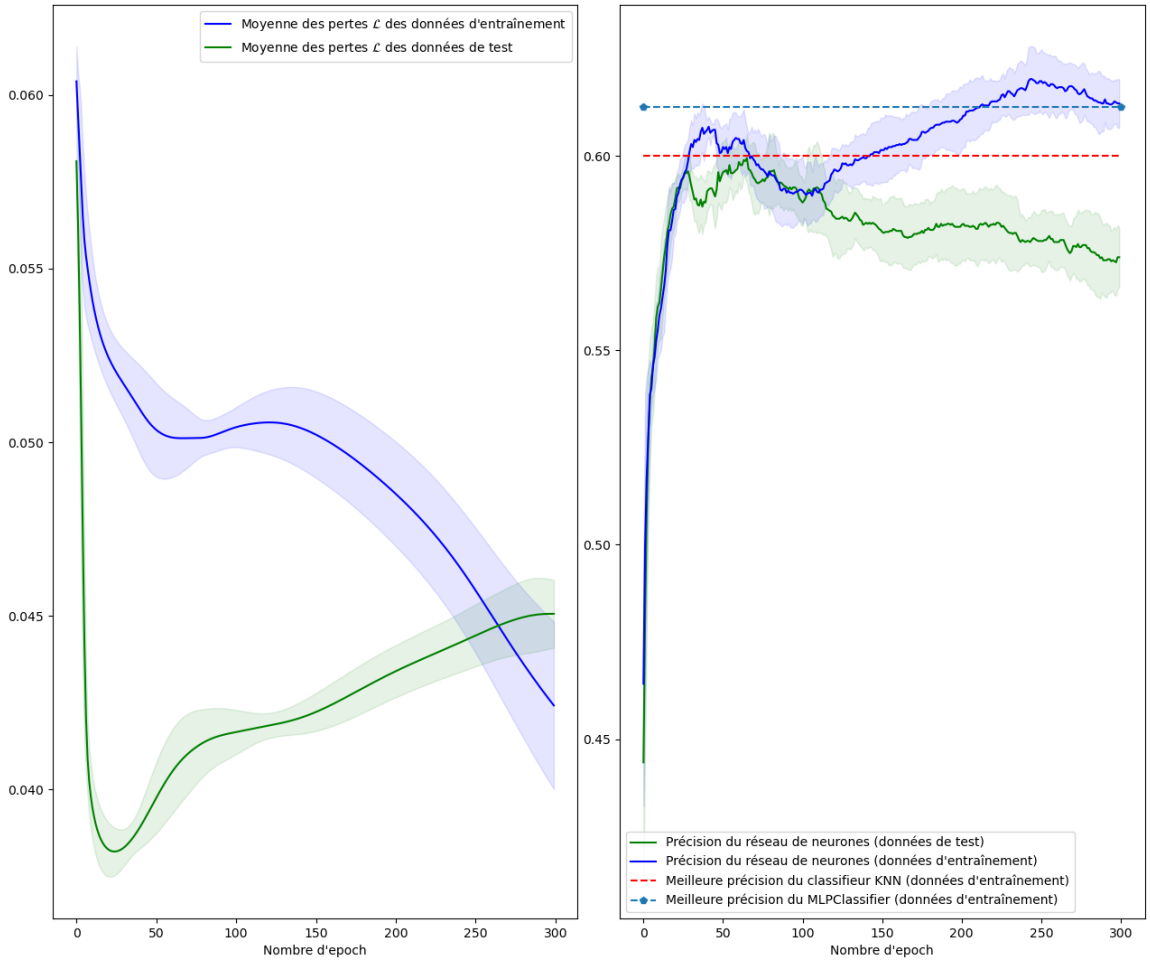


FIGURE 13: Évolution des métriques avec 30% de données d'entraînement

rouges en qualité. Nous avons effectué 8 mesures et analysé le résultat des 8 mesures.

La figure 13 représente l'évolution de la perte $\mathcal{L}$ ainsi que de la précision des données d'apprentissage et de validation, en fonction du nombre d'epoch (un epoch étant un passage complet des données dans le réseau de neurone).

Nous observons sur la figure 13 que la perte des données d'apprentissage ne cesse de baisser, mais la perte des données de test augmente après un certain nombre d'epochs, et passe de 0.038 ± 0.02 à l'epoch numéro 65 environ, à 0.045 ± 0.02 à l'epoch numéro 300. De façon similaire, la précision du réseau baisse pour les données de test après un certain point, tandis que celle des données d'entraînement est croissante.

Nous en déduisons donc que le réseau s'est adapté pour donner les meilleurs métriques pour les données d'apprentissage, mais elles ne sont pas représentatives des données de validation, car l'échantillon pris est trop petit. Les performances pour les données de validation en sont donc atteintes négativement.

Cela illustre le phénomène de surentraînement.

Dataset	Wine quality
Neural-Net shape	(4,5,11)
Train size	de 0.3 à 0.01
Learning rate	0.03
Activation function	sigmoid
Min step	0.1
Total EPOCH	20

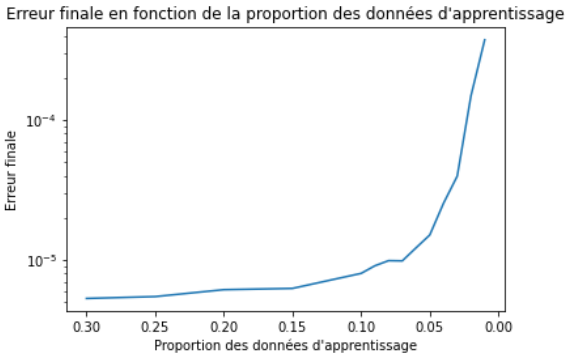


FIGURE 14: Erreur finale dans les données tests pour le dataset Winequality

9. Comparaison avec sklearn

On va comparer notre réseau de neurones avec sklearn sur le dataset des qualités de vins. C'est un problème de classification, c'est pourquoi on va utiliser sklearn.MLPClassifier avec les paramètres suivants :

Dataset	Wine quality
Neural-Net shape	(4,5,1)
Train size	0.2
Learning rate	0.03
Activation function	tanh
Max EPOCH	300

Cependant, pour faire marcher le Classifier de sklearn, il fallait laisser les notes de vins entre 0 et 10 et ne pas les normaliser. Du coup pour comparer les données de sklearn avec les notres, on a normalisé les résultats à la fin de la même manière que pour notre réseau de neurones.

Les résultats finaux pour sklearn et notre réseau de neurones sont :

Erreur Réseau de Neurones	$7,78 \cdot 10^{-6}$
Erreur sklearn	$6,33 \cdot 10^{-6}$

On peut voir que les deux modèles ont convergé vers des erreurs très proches. On peut donc conclure que notre méthode est presque aussi efficace que celle de sklearn.

Comparons également ces deux modèles sur le dataset des champignons, les paramètres utilisés sont :

Dataset	Mushroom
Neural-Net shape	(4,5,1)
Train size	0.2
Learning rate	0.03
Activation function	tanh
Max EPOCH	300

et on peut voir que notre réseau de neurone et la librairie sklearn nous donnent des résultats très similaires :

Erreur Réseau de Neurones	$6,18 \cdot 10^{-5}$
Erreur sklearn	$3,99 \cdot 10^{-5}$

10. Contributions

- **Malik Algelly** : développement (battelle.nn, battelle.perceptron, créations de scripts pour battelle.nn, battelle.perceptron), écriture du rapport (sections 5 ,6)
- **Luca Perone** : développement (battelle.nn, battelle.knn, créations de tests, de scripts et d'exemples pour battelle et sklearn), structure de la librairie battelle.
- **Juraj Rosinsky** : développement (méthodes de la librairie battelle.nn, scripts d'entraînements pour battelle.nn et sklearn, scripts de visualisation de données), écriture du rapport (sections 4, 8, 9)
- **Samuel Simko** : développement (battelle.nn, scripts d'entraînements pour battelle et sklearn), structure de la librairie battelle, documentation en ligne, écriture du rapport (sections 2, 7, 8)

11. Citations and Bibliography

Acknowledgments

Nous remercions le corps enseignant du cours *Intelligence Artificielle* de l'Université de Genève pour l'aide apportée.

Références

Jason Brownlee. How to implement the perceptron algorithm from scratch in python, 2016. URL https://machinelearningmastery.com/implement-perceptron-algorithm-scratch-python/.

Francois Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Stéphane Marchand-Maillet. Réseaux de neurones artificiels, 2021.

F. Almeida T. Matos P. Cortez, A. Cerdeira and J. Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47 :547–553, 2009.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.