AMAL: Advanced MAchine Learning & Deep Learning

Rapport TME 1,2,3

Kevin Meetooa

Année 2020-2021

Table des matières

Chapitre 1

TME 1

Dans ce TME, nous avons implémenté une descente de gradient sur un problème de régression linéaire. Nous avons implémenté une régression linéaire générale prenant un batch de q exemples en entrée et p sorties.
La figure 1.1 montre les résultats obtenus pour 50 exemples, 3 sorties et 13 features.
Remarque : Ici, on a appliqué la descente de gradient en optimisant la loss sur l'ensemble des données en même temps.

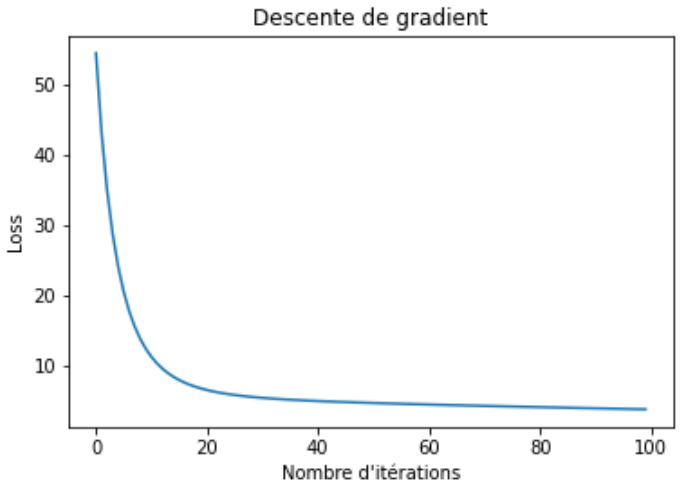

FIGURE 1.1 – Descente de gradient

Nous avons ensuite essayé de voir l'impact de l'hyperparamètre ε (learning rate).

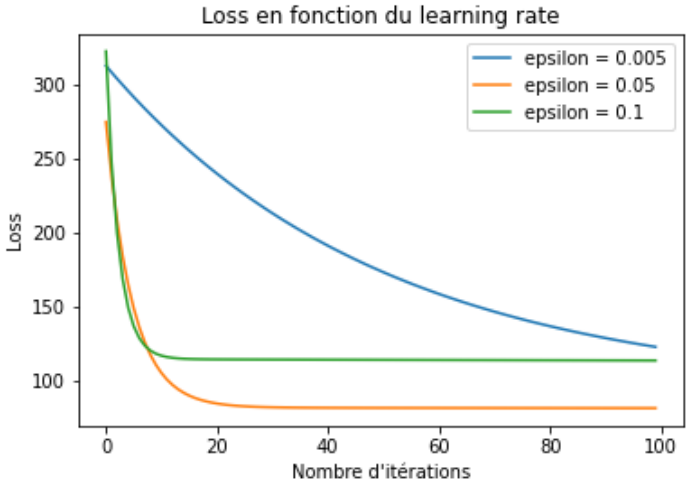

FIGURE 1.2 – Impact de l'hypermaramètre ε

La figure 1.2 montre que les résultats dépendent du ε choisi.

- Pour un ε trop faible, la convergence est trop lente.
- Pour un ε trop grand, on converge rapidement mais vers un minimum non optimal
- Il faut donc choisir un optimum qui soit ni trop grand ni trop faible afin de converger rapidement vers un minimum optimal.

Chapitre 2

TME 2

Dans ce TME, nous tentons de résoudre un problème de régression linéaire sur les données Boston Housing. Pour cela, nous avons implémenté un algorithme de descente de gradient mini-batch/stochastique en utilisant la différenciation automatique, puis nous avons implémenté un réseau de neurones à deux couches.

Pour commencer, nous avons comparé les trois types possibles de descente de gradient :

- Batch : Optimisation sur l'ensemble des données
- Mini-batch : Optimisation sur des sous-ensembles
- Stochastique : Optimisation sur un élément à la fois

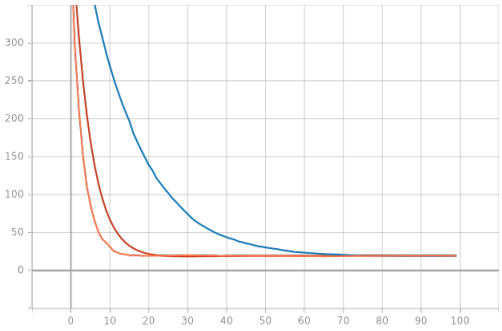

FIGURE 2.1 – Erreur sur les données en train

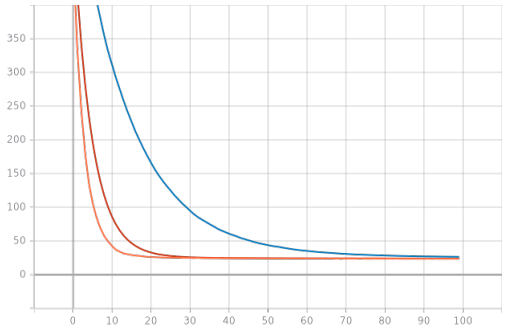

FIGURE 2.2 – Erreur sur les données en test

Les figures 2.1 et 2.2 montrent respectivement les performances en apprentissage et en test. Ces figures ont été réalisées à l'aide de TensorBoard.

Les courbes oranges, rouges et bleues correspondent respectivement aux algorithmes mini-batch, batch et stochastique.

Nos trois modèles semblent tous converger vers le même optimum, seule la vitesse de convergence diffère. Nous pouvons voir que le mini-batch converge plus rapidement que les autres algorithmes.

L'algorithme stochastique converge lentement, ce qui est cohérent car on optimise notre modèle sur un élément à la fois.

Nous avons ensuite mené une campagne d'expériences afin de sélectionner la bonne valeur pour le nombre de neurones dans la couche cachée du réseau.

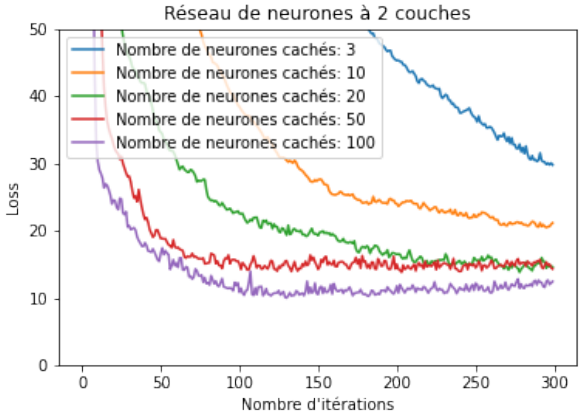

FIGURE 2.3 – Erreur en test pour différents nombre de neurones cachés

La figure 2.3 montre qu'un nombre de neurones cachés trop faible conduit à une convergence trop lente.

On voit que la vitesse de convergence est meilleure lorsque le nombre de neurones cachés augmente. Cependant, on peut se contenter de 20 neurones cachés. En effet, il n'y a pas d'amélioration notable entre 20 et 100 neurones cachés. Ainsi, dans la suite de cette partie, quand nous parlerons du réseau de neurones, nous considèrerons un réseau à 20 neurones cachés.

Nous avons ensuite évalué les performance de nos algorithmes (descente de gradient et réseau de neurones) en entraînement ainsi qu'en test.

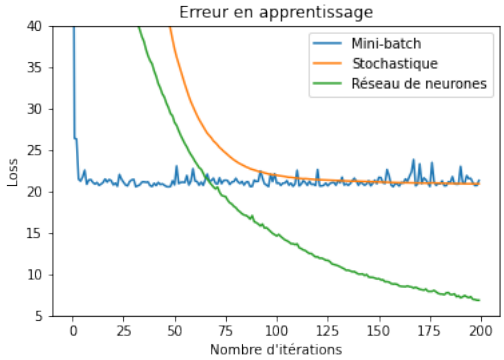

FIGURE 2.4 – Erreur sur les données train

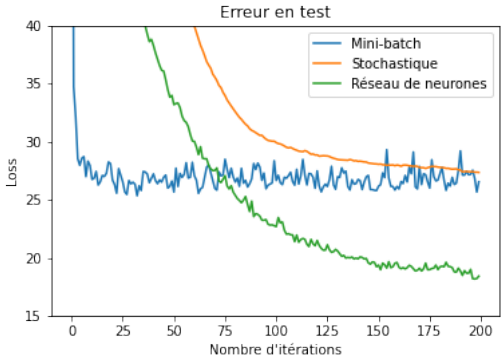

FIGURE 2.5 – Erreur sur les données test

Les figures 2.4 et 2.5 montrent que le réseau de neurones converge vers un minimum beaucoup plus bas que ceux atteint par une simple descente de gradient.

On peut noter que l'algorithme de descente de gradient mini-batch converge beaucoup plus rapidement que les autres algorithmes.

On compare maintenant les performances de la descente de gradient mini-batch en fonction de la taille des batchs.

Sur la figure 2.6, on peut voir que lorsque la taille des batchs augmente, la convergence est beaucoup plus lente en termes de nombre d'itérations.

En zoomant sur la courbe de loss, on peut voir sur la figure 2.7 que le minimum atteint semble être le même, peu importe la taille de batch.

Cependant, on voit (en légende) que plus la taille de batch est grande, plus l'algorithme s'exécute rapidement, cela est dû à la parallélisation des calculs qui est plus efficace lorsqu'on a des batchs de grande taille.

Ainsi, il y a un compromis à faire entre taille de batch et nombre d'itérations afin d'avoir un algorithme qui s'exécute rapidement sans effectuer trop d'itérations.

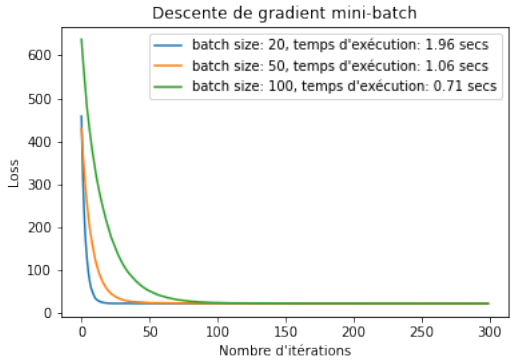

FIGURE 2.6 – Test avec différentes tailles de batchs

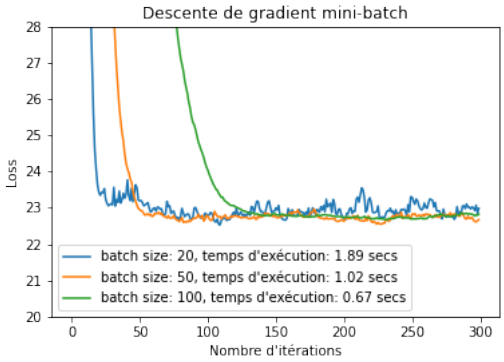

FIGURE 2.7 – Zoom sur la loss

Chapitre 3

TME 3

Dans ce TME, nous avons développé un AutoEncoder. Nous l'avons appliqué sur la base de données MNIST composée de 70000 images, chacune de taille 28 x 28 pixels.

Le principe est le suivant :

- Nos images comportent $28 * 28 = 784$ features.
- On encode les données en dim_{enc} features avec $dim_{enc} < 784$ en utilisant une transformation linéaire, puis on applique une fonction d'activation sur le résultat obtenu
- Ensuite, on décode les données afin de leur redonner leur forme initiale (784 features) en utilisant à nouveau une transformation linéaire

L'objectif étant de pouvoir reconstruire des images à partir d'images encodées.

Remarque : Nous effectuons deux transformations linéaires. Ces deux transformations linéaires partagent le même paramètre W. L'encodage utilise W et le décodage utilise $W^\intercal$.

Cependant, les paramètres b des deux transformations linéaires sont différents.

Nous avons entraîné notre AutoEncodeur puis nous avons évalué ses performances en fonction de la dimension d'encodage (le nombre de neurones dans la couche cachée).

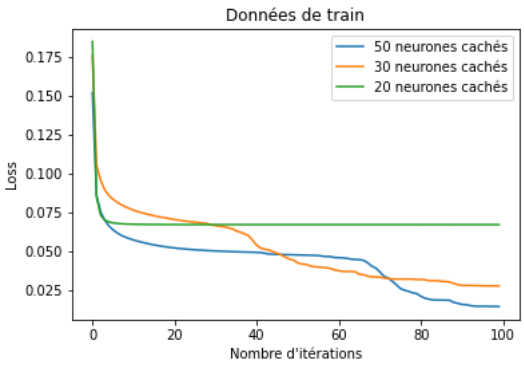

FIGURE 3.1 – Erreur en apprentissage

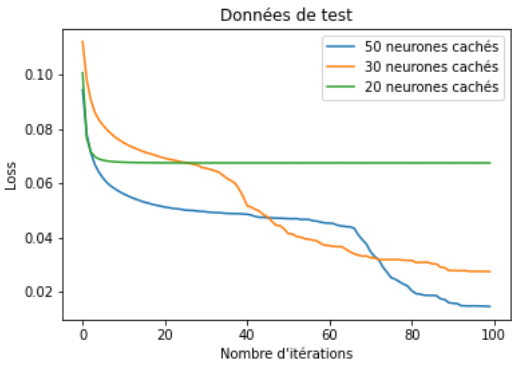

FIGURE 3.2 – Erreur en test

Les figures 3.1 et 3.2 montrent respectivement l'évolution de la loss en apprentissage et en test. On voit que plus la dimension d'encodage dim_{enc} croît, plus la loss est faible, ce qui est cohérent. En effet, lorsque plus dim_{enc} est faible, plus on risque de perdre de l'information.

De même, plus dim_{enc} est grand, plus on a de chances de pouvoir restituer l'information de manière fidèle aux données.

Nous avons ensuite visualisé la reconstruction des images sur les données de test.

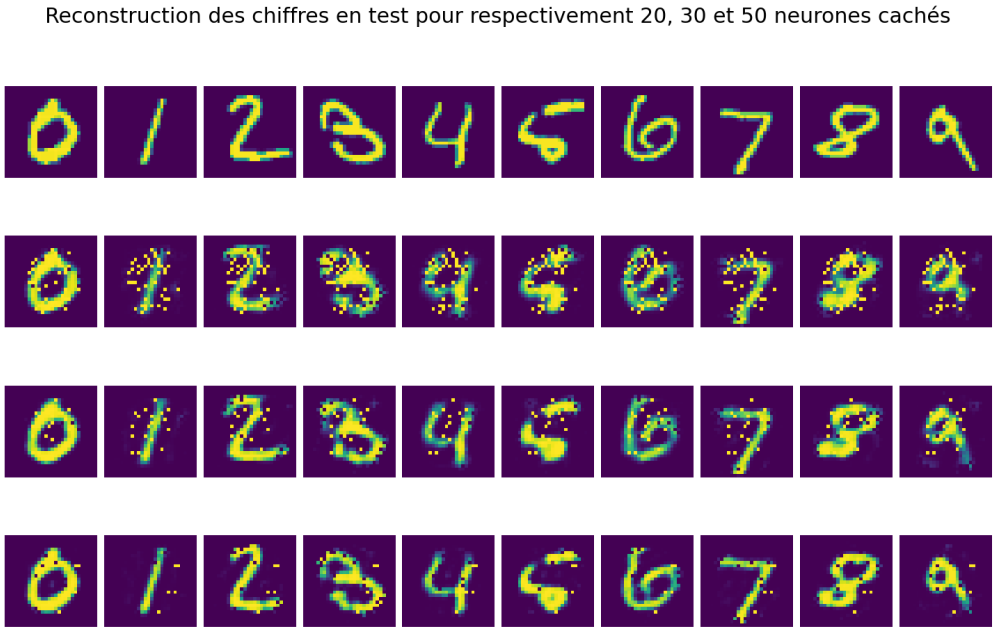

FIGURE 3.3 – Reconstruction d'images

La figure 3.3 montre les images de test en première ligne puis les images reconstruites avec respectivement 20, 30 et 50 neurones cachés.

Nous voyons que pour 20 neurones cachés, les images reconstruites comportent des "artéfacts" bruitants. Plus le nombre de neurones cachés augmente, plus ces artéfacts disparaissent, ce qui est cohérent avec les courbes de loss observées en 3.2.

Finalement, nous avons implémenté un Highway Network que nous avons appliqué au problème de régression sur les données Boston Housing.

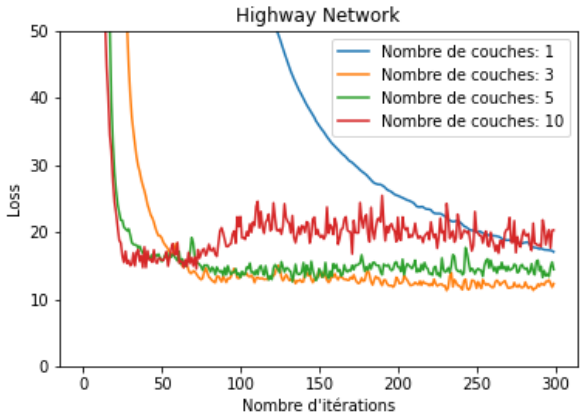

FIGURE 3.4 – Highway Network

La figure 3.4 montre les résultats obtenus selon différents nombres de couches. Nous pouvons voir que 3 couches suffisent pour ce problème : Un nombre de couches plus élevé n'améliore pas les performances du modèle.