Rapport

Partie 2 : Shallow network

Implémentation du réseau

Nous avons utilisé l'exemple fourni sur le site PyTorch dans la section "Build the Neural Network" (lien) pour démarrer la création du réseau. Dans cette partie, pour implémenter le réseau nous avons utilisé une seule couche cachée. Notre MLP est instancié avec les hyperparamètres ce qui permet de changer facilement de modèle pour nos tests.

Voici la structure de notre MLP :

- Couche d'entrée : 784 neurones correspondant aux pixels des images MNIST aplaties de taille 28×28.
- Couche cachée : à une taille ajustable afin d'optimiser les performances du modèle.
- Fonction d'activation : ReLU appliquée sur la couche cachée pour introduire de la non-linéarité.
- Couche de sortie : 10 neurones chacun correspondant à un chiffre de 0 à 9, permettant de prédire le label de l'image.

Notre modèle hérite de nn.Module. Cela permet d'utiliser les fonctionnalités de PyTorch pour créer et gérer les couches, comme nn.Linear et nn.ReLU. Les poids W et les biais B de chaque couche sont gérés automatiquement avec model.parameters() et peuvent être passés directement à l'optimiseur. On peut envoyer les données dans le MLP en appelant simplement model(entrée). Les méthodes model.train() et model.eval() permettent d'améliorer l'entraînement et la qualité des prédictions. nn.Module offre aussi de nombreuses autres fonctionnalités comme pour sauvegarder et charger les poids du modèle et pour utiliser le GPU.

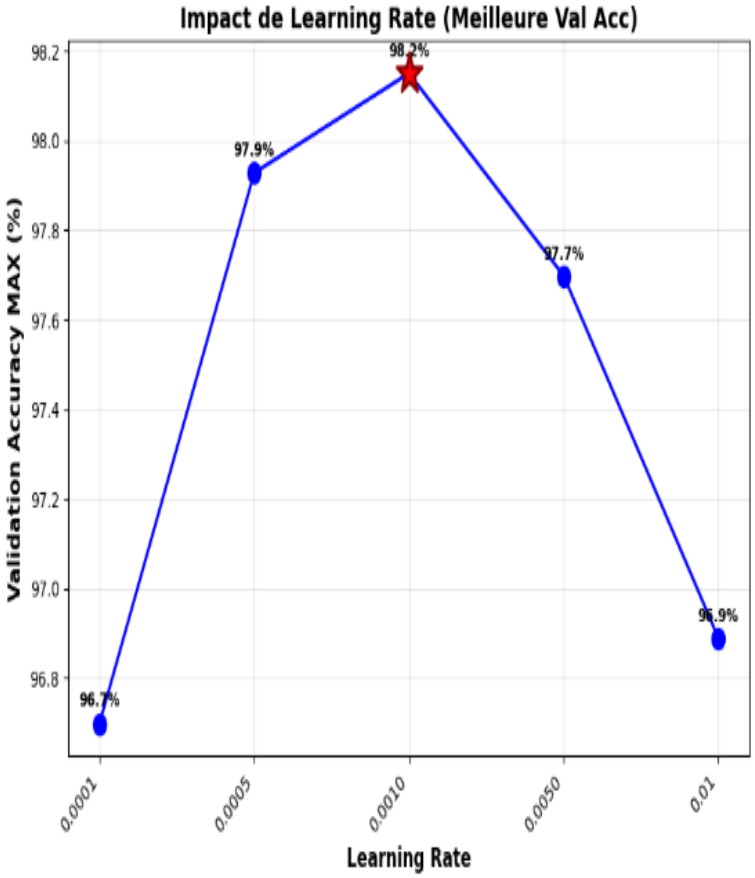
Méthodologie pour chercher les hyperparamètres

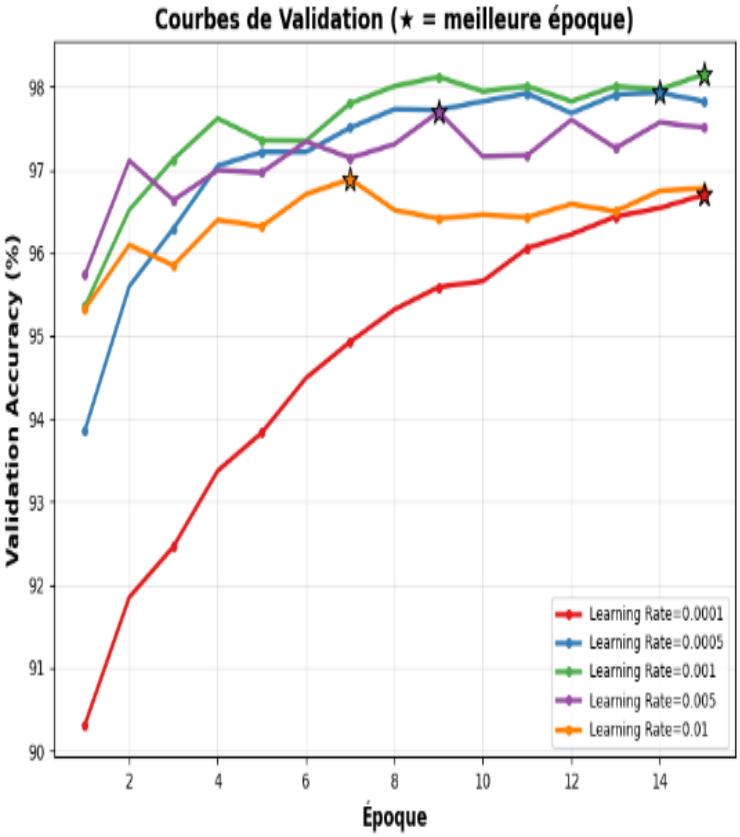
Nous avons choisi de tester chaque hyperparamètre indépendamment des autres afin d'observer plus clairement leur impact sur l'entraînement et la validation, tout en conservant des valeurs par défaut pour les paramètres non testés. Nous n'avons donc pas utilisé la méthode grid search, car tester toutes les combinaisons aurait été trop long (par exemple, 5 choix pour chacun des 4 hyperparamètres donneraient $5^4 = 625$ combinaisons). Cette approche nous permet aussi de mieux comprendre l'effet de chaque hyperparamètre individuellement.

Analyse des résultats learning rate

On commence par tester le learning rate sur les données de validation avec les valeurs suivantes : [0.0001, 0.0005, 0.001, 0.005, 0.01].

Validation accuracy

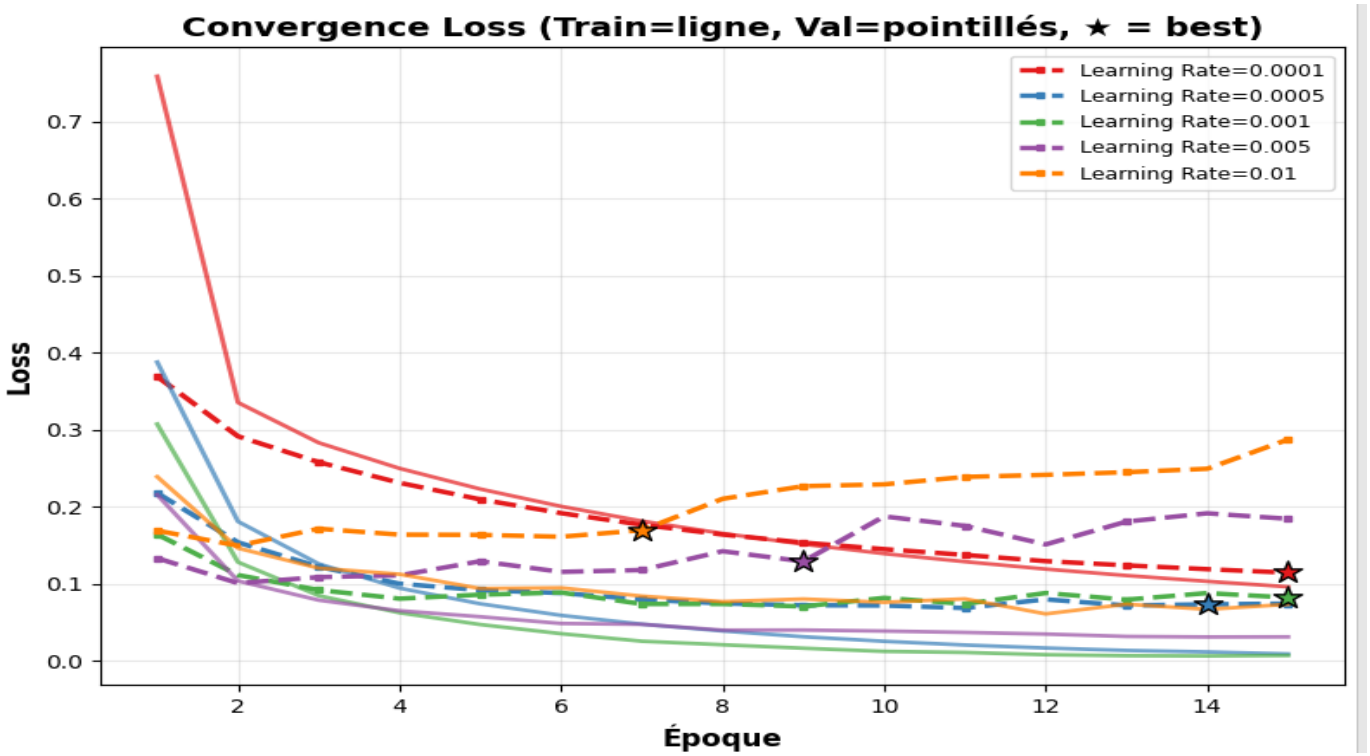




On observe sur le graphique de gauche que le meilleur taux d'accuracy est obtenu avec un learning rate de 0.001. Plus on s'éloigne de cette valeur plus les performances diminuent ce qui forme une courbe en cloche.

Sur le graphique de gauche on voit qu'avec un learning rate très faible 0.0001, l'apprentissage est lent et progressif, il faut plus d'époques pour converger. À l'inverse, un learning rate trop élevé 0.01 fait converger rapidement (dès la 7e époque), mais la performance stagne et n'atteint pas le niveau des autres courbes.

Loss training et validation



On observe un légé sur-apprentissage pour un learning rate de 0.01, et dans une moindre mesure pour 0.005. La loss de validation (courbes pointillées violet et orange) est beaucoup plus élevée que la loss d'entraînement (courbes continues). Cela indique que le modèle a trop appris les détails des données d'entraînement et n'arrive pas à bien généraliser aux nouvelles données. Même si l'accuracy reste relativement correcte, le modèle fait parfois de grosses erreurs de prédictions sur certaines images de validation, ce qui fait augmenter fortement la loss.

Performance temps

Concernant le temps d'entraînement, nous avons observé une très faible augmentation (de l'ordre de quelques millisecondes) lorsque le learning rate augmente. L'impact sur le temps total est donc négligeable.

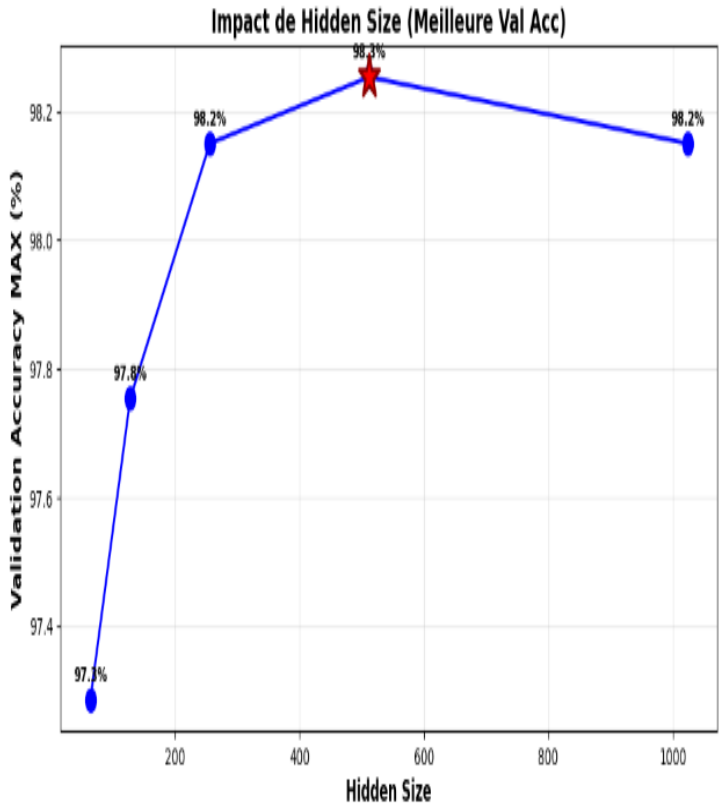
Conclusion learning rate

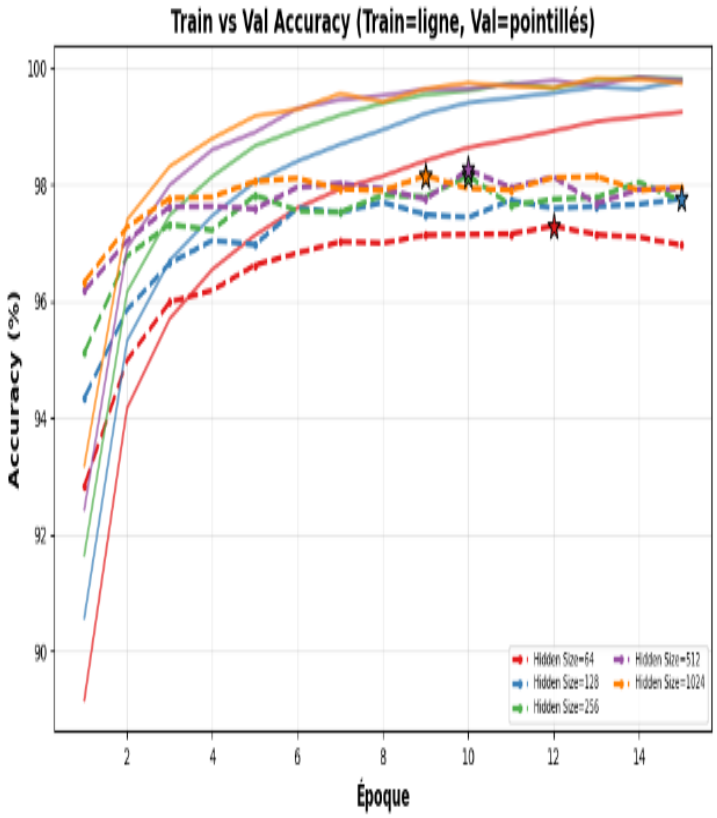
Pour conclure, pour obtenir de meilleures prédictions et apprendre rapidement tout en gardant un faible écart entre la loss d'entraînement et de validation (assurant une bonne généralisation), le meilleur learning rate à retenir est 0.01. Il permet un apprentissage pas trop long, sans sur-apprentissage flagrant, et maintient une accuracy élevée. Ces tests ont permis de constater que le learning rate influence à la fois la capacité d'apprentissage et la stabilité : un learning rate trop faible ralentit la convergence, tandis qu'un learning rate trop élevé peut rendre l'apprentissage instable.

Analyse des résultats hidden_size

On teste hidden size sur les données de validation et d'entrainement avec les valeurs suivantes : [64, 128, 256, 524, 1028].

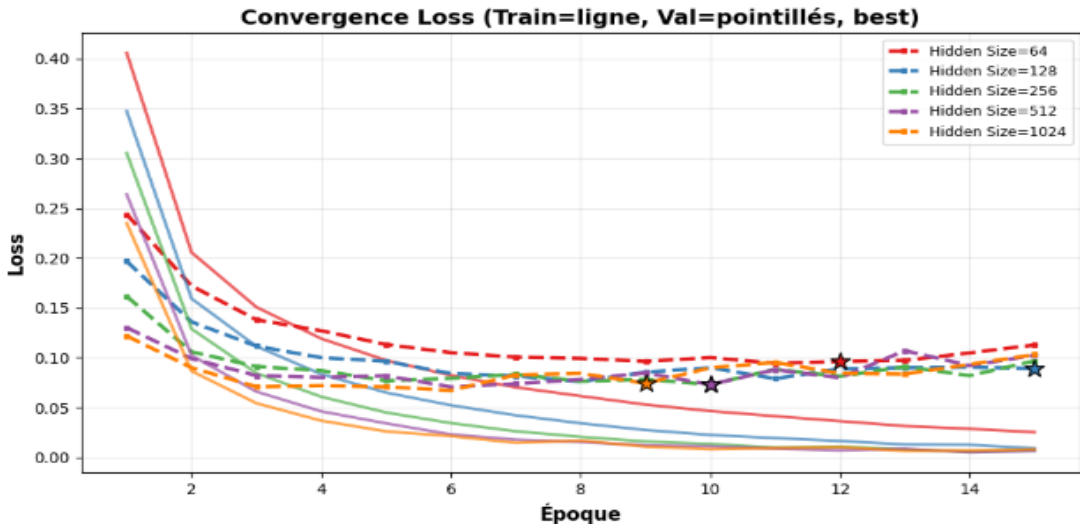
Validation accuracy





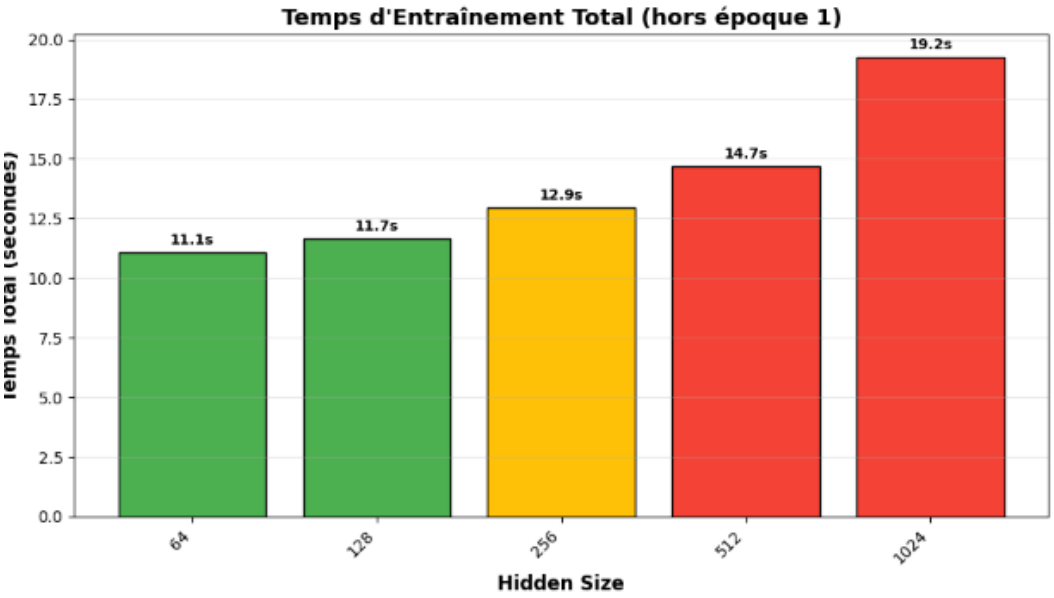
On observe qu'il n'y a pas de grande différence d'accuracy à partir d'une taille de couche cachée de 256 neurones. Une couche trop petite (par exemple 64) limite la capacité d'apprentissage du modèle, ce qui réduit logiquement l'accuracy. En revanche, augmenter la taille au-delà d'un certain point n'apporte presque plus d'amélioration, tout en augmentant le temps d'entraînement. Il faut donc choisir une taille de couche cachée ni trop petite, ni trop grande. 256 semble être le bon compromis.

Loss training et validation



On peut voir qu'une couche trop grande n'a pas beaucoup d'impact sur le sur-apprentissage, contrairement à un learning rate trop élevé.

Performance temps



On voit que plus la couche cachée est grande, plus le temps d'entraînement augmente. À partir de 1024 neurones, le temps monte rapidement, avec environ 8 secondes d'écart entre une couche de 64 et une de 1024 neurones. Il faut donc trouver un bon compromis entre la taille de la couche et le temps d'entraînement, surtout si le dataset est plus grand et nécessite plus d'itérations. Une couche trop grande n'apporte pas forcément plus d'efficacité et peut augmenter inutilement le temps d'apprentissage.

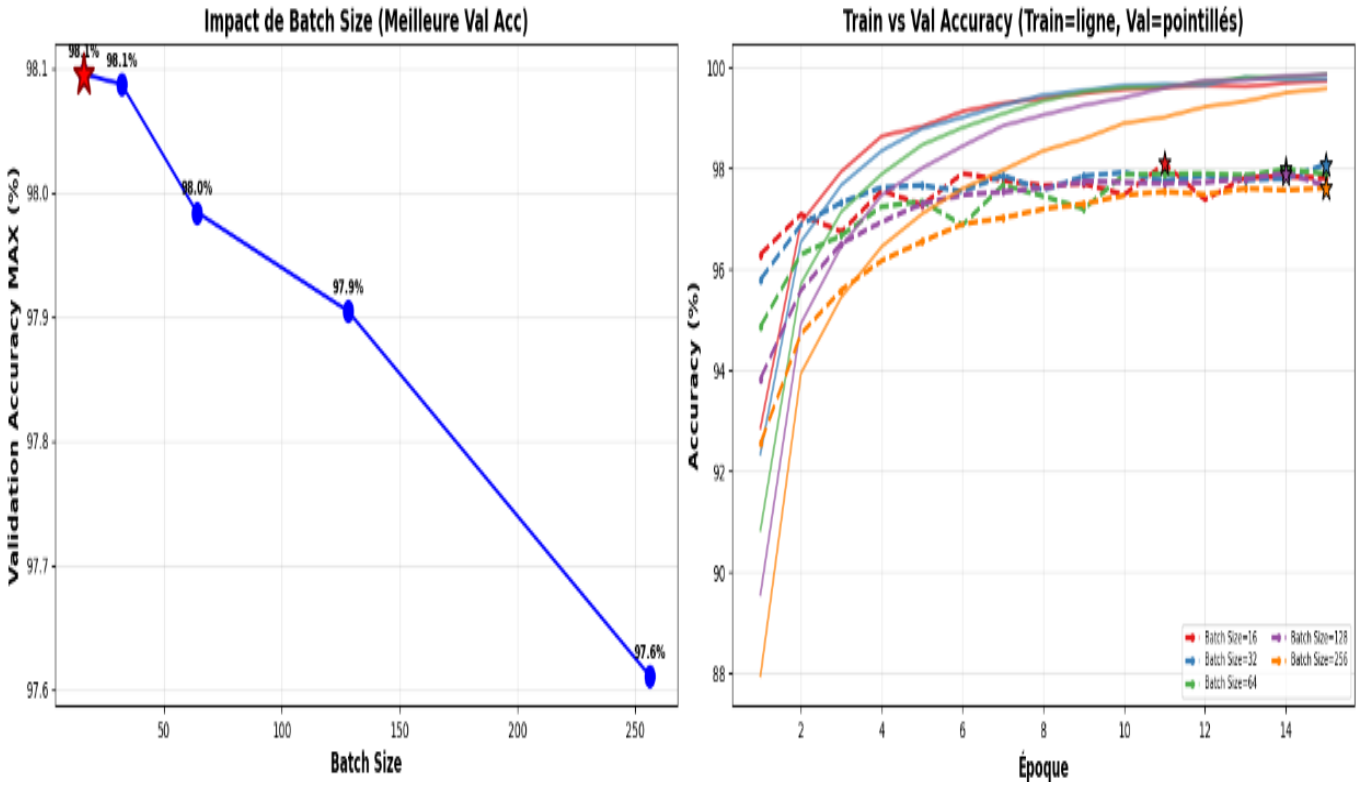
Conclusion hidden_size

La couche de taille 256 est le meilleur compromis performance/temps. Même si des couches plus grandes peuvent donner des résultats légèrement meilleurs, cela ne vaut pas le coup en raison du temps d'entraînement supplémentaire. Nous avons appris qu'une taille de couche trop grande n'impacte pas le sur-apprentissage comme le fait un learning rate trop élevé, mais elle augmente le temps de calcul. À l'inverse, une couche trop petite limite l'apprentissage, comme avec la couche de 64 neurones qui obtient des résultats inférieurs aux autres.

Analyse des résultats batch_size

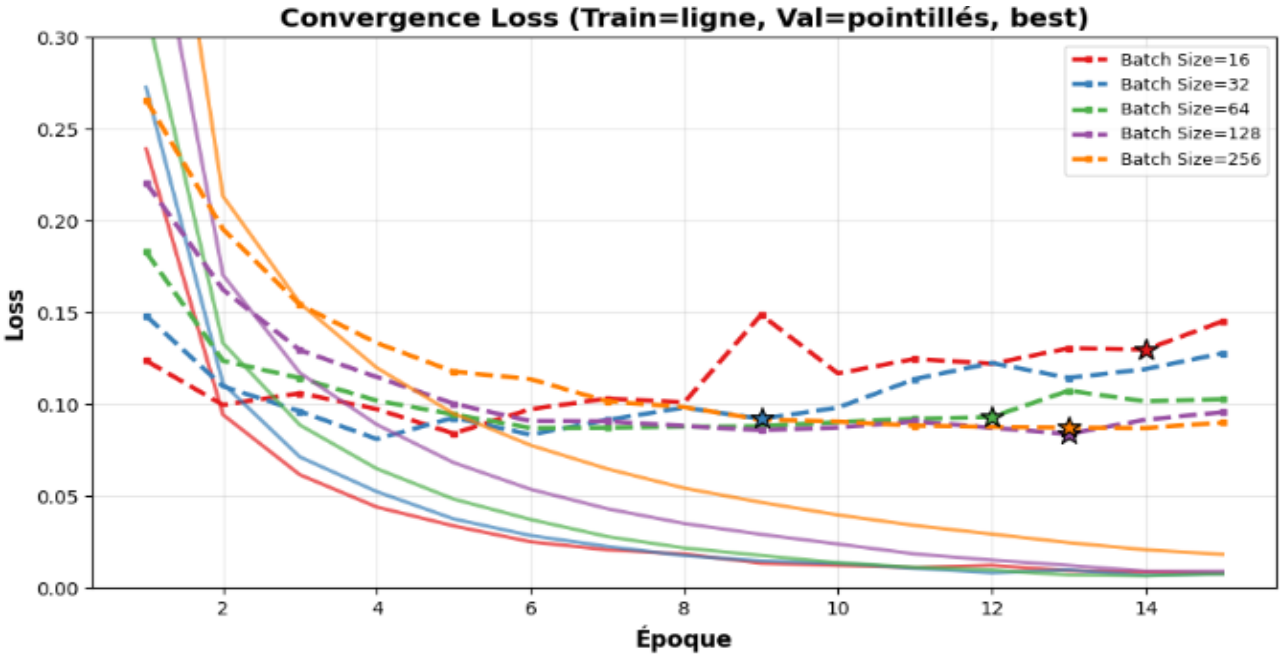
On teste batch_size sur les données de validation et d'entrainement avec les valeurs suivantes : [16, 32, 64, 128, 256]

Accuracy validaton et training



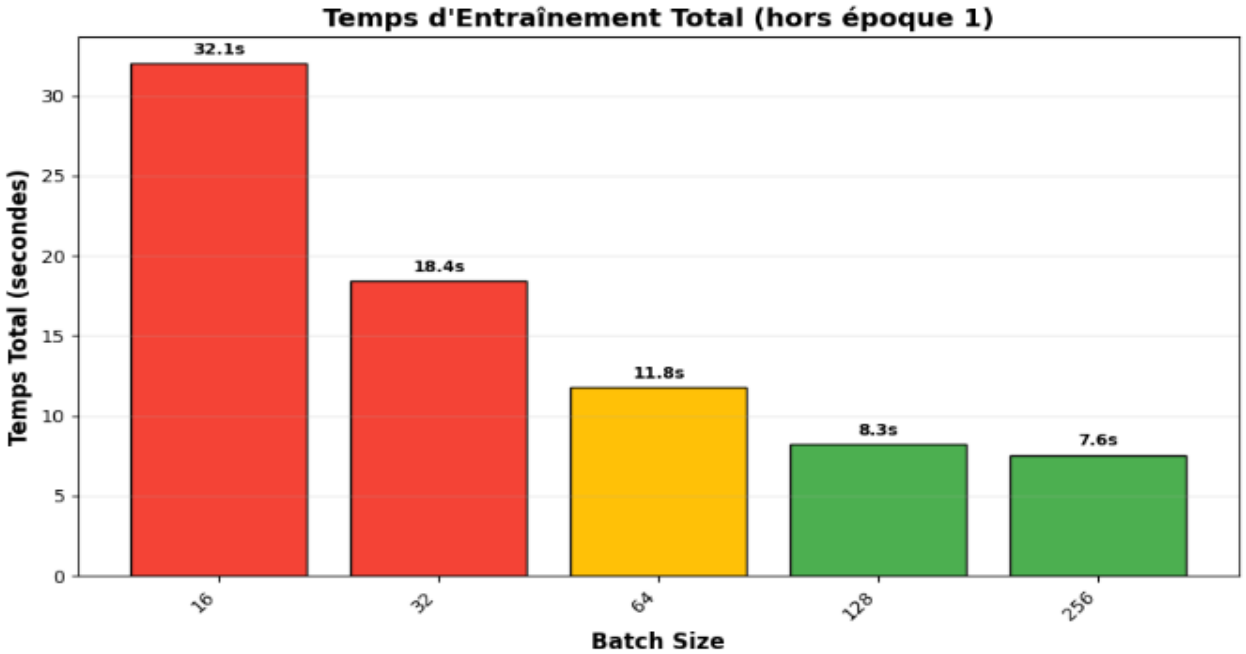
On peut voir que le batch size influence très peu l'accuracy, sûrement parce que le dataset MNIST est petit. Les résultats sont très proches quelle que soit la taille du batch. On remarque une légère amélioration avec les petits batchs, car les gradients sont mis à jour plus souvent et sont plus bruités, ce qui aide le modèle à mieux généraliser. En revanche, cela rend l'apprentissage un peu plus instable, comme on peut le voir sur les courbes pointillées(validations), les petits batchs qui présentent de petites fluctuations, contrairement aux courbes des grands batchs, plus stables.

Loss training et validation



On observe, comme mentionné plus haut, que les petits batchs sont plus instables contrairement aux grands batchs.

Performance temps



On observe que plus le batch est petit, plus le temps d'entraînement augmente, car le nombre de calculs de gradients par époque est plus élevés. Il est donc judicieux de choisir un batch ni trop petit, afin d'assurer à la fois stabilité et bonne généralisation.

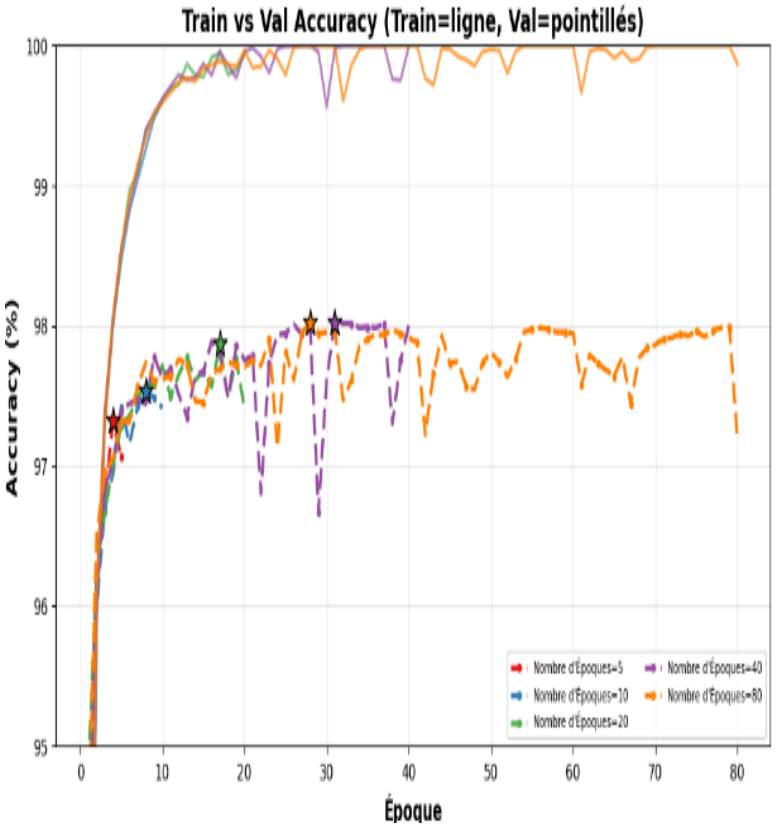
Conclusion batch_size

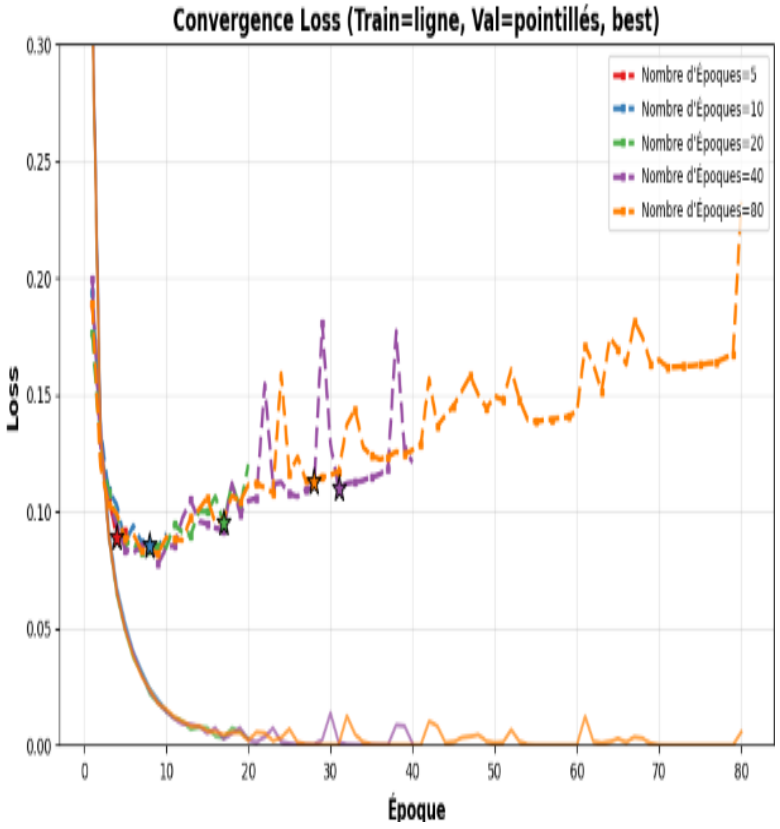
Le meilleur compromis est un batch size de 64. Il offre stabilité, bonne accuracy et un temps d'entraînement raisonnable.

Analyse des résultats nb_epochs

On teste nb_epochs sur les données de validation et d'entrainement avec les valeurs suivantes : [5, 10, 20, 40, 80].

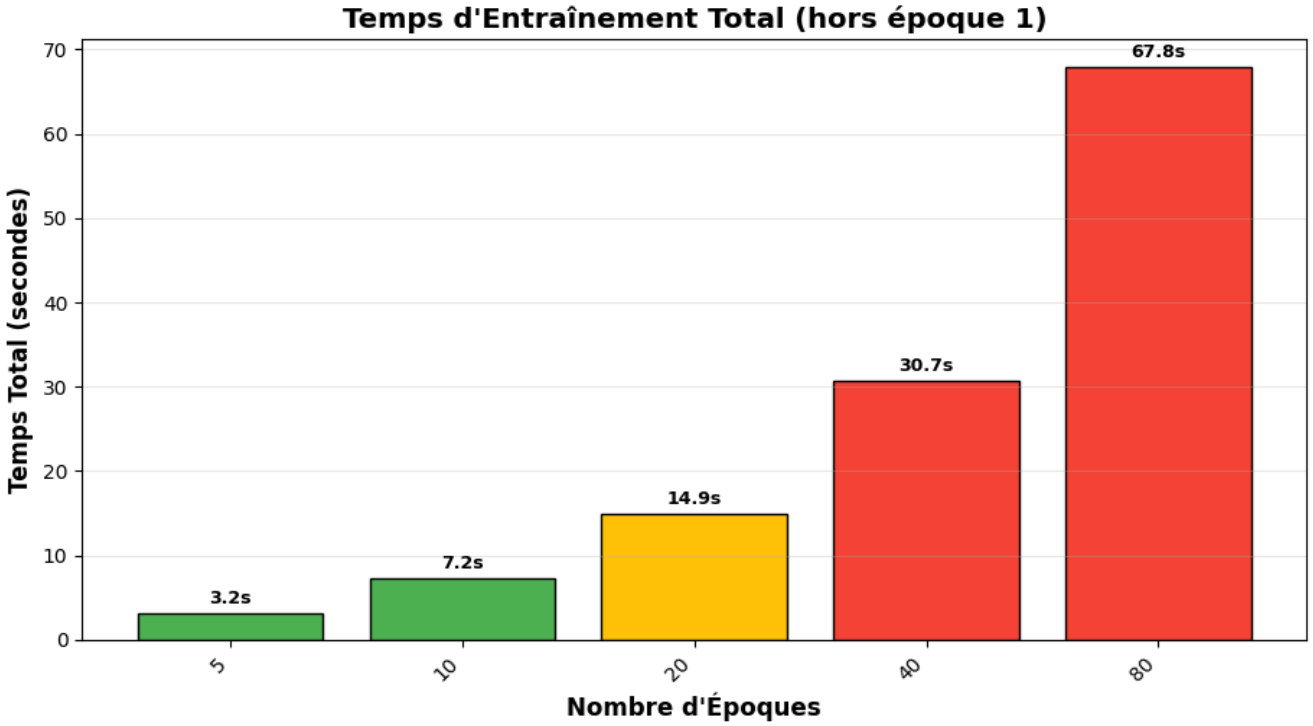
Accuracy validation et training





On observe qu'à partir de 20 époques, la loss de validation augmente lentement et devient plus instable, tandis que l'accuracy reste élevée. Cela montre que le modèle devient trop confiant. il fait peu d'erreurs, mais celles-ci sont plus importantes, ce qui fait augmenter la loss. Comme l'accuracy reste stable (voire s'améliore très légèrement), on peut conclure que le modèle a déjà convergé avant la 20e époque.

Performance temps



On constate qu'à chaque fois qu'on double le nombre d'époques, le temps d'entraînement double également. Il est donc inutile d'en faire trop, au risque de perdre du temps et de favoriser le sur-apprentissage.

Conclusion

Le meilleur compromis entre le temps d'entraînement et la convergence du modèle, sans risque de sur-apprentissage, est 15 époques. En effet, entre 10 et 20 époques, l'accuracy reste quasiment identique, tandis qu'à 20 époques, la loss commence légèrement à augmenter.

Conclusion finale

Les paramètres que nous avons choisis sont un learning rate de 0.01, une taille cachée (hidden size) de 256, une taille de lot (batch size) de 64 et 15 époques d'entraînement. Ces paramètres permettent d'obtenir de bons résultats en tenant compte du temps d'exécution, de l'accuracy et en évitant à la fois le surapprentissage et le sous-apprentissage.

Les résultats obtenus avec ces paramètres sont les suivants :

```
Meilleure Val Acc: 98.02% (époque 14)

--------------------------------------------------------------------------------
Résultats train et val
--------------------------------------------------------------------------------
Train Acc (best): 99.90% | Val Acc (best): 98.02%
Train Acc (avg): 98.39% | Val Acc (avg): 97.39%
Train Loss (avg): 0.0567 | Val Loss (avg): 0.0891
--------------------------------------------------------------------------------

Résultat du test
Test Loss: 0.08 | Test : 98.07%
--------------------------------------------------------------------------------
```