

Rapport TP Deep Learning

De la conception au déploiement

FOLONG TAFOUKEU ZIDANE

8 octobre 2025

1 Introduction

Ce rapport présente la réalisation du TP de Deep Learning portant sur la conception, l'entraînement et le déploiement d'un modèle de classification des chiffres manuscrits MNIST.

2 Partie 1 : Fondations du Deep Learning

2.1 Concepts Théoriques

Question 1 : Différence entre descente de gradient classique et SGD

La descente de gradient classique utilise l'ensemble du dataset pour calculer le gradient à chaque itération, ce qui est coûteux en mémoire et temps de calcul. La SGD (Stochastic Gradient Descent) utilise un échantillon aléatoire (batch) pour estimer le gradient, permettant des mises à jour plus fréquentes et une convergence plus rapide, particulièrement adaptée aux grands datasets du deep learning.

Rétropropagation du gradient

La rétropropagation calcule les gradients des poids en propageant l'erreur de la sortie vers l'entrée du réseau, utilisant la règle de dérivation en chaîne pour ajuster chaque poids proportionnellement à sa contribution à l'erreur finale.

2.2 Exercice 1 : Construction du réseau de neurones

Question 1 : Utilité des couches Dense et Dropout

Les couches Dense (fully-connected) connectent chaque neurone à tous les neurones de la couche suivante, permettant l'apprentissage de relations complexes. La couche Dropout désactive aléatoirement des neurones pendant l'entraînement pour éviter le surapprentissage. La fonction softmax normalise les sorties en probabilités pour la classification multi-classes.

Question 2 : Optimiseur Adam

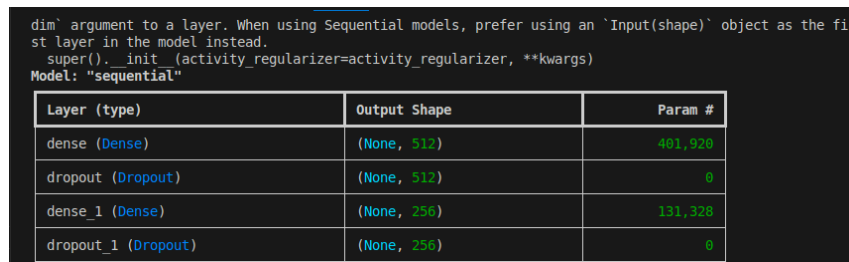
Adam combine les avantages de RMSprop et Momentum en adaptant le taux d'apprentissage pour chaque paramètre individuellement et en utilisant des moyennes mobiles des gradients et de leurs carrés, offrant une convergence plus stable et rapide que la SGD simple.

Question 3 : Vectorisation et calculs par lots

La vectorisation permet de traiter plusieurs échantillons simultanément via des opérations matricielles optimisées. Le paramètre `batch_size=128` traite 128 images à la fois, exploitant le parallélisme des GPU et améliorant l'efficacité computationnelle.

3 Partie 2 : Resultats Obtenus

Nous avons réalisé l'entraînement du modèle tout en suivant la méthodologie donnée dans le TP et les images ci-dessous présentent les logs de l'entraînement :



```

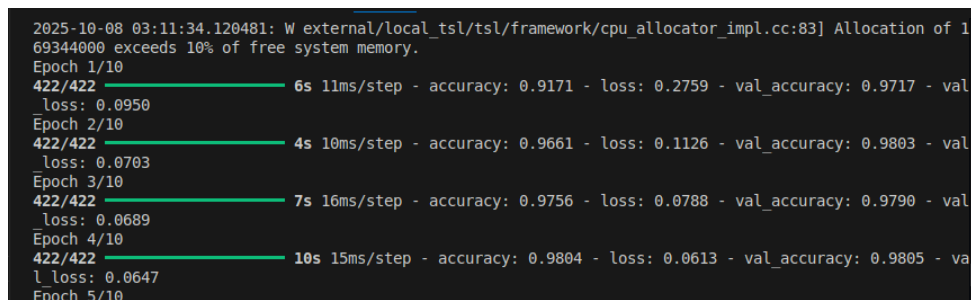
dim' argument to a layer. When using Sequential models, prefer using an 'Input(shape)' object as the first layer in the model instead.
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"

```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	401,920
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0

FIGURE 1 – Architecture d'un Réseau de Neurones Séquentiel

Le tableau précédent nous présente donc la structure du modèle de type **Sequential**. Sur l'image précédente, nous avons une vue sur les epochs lors de l'entraînement du modèle.



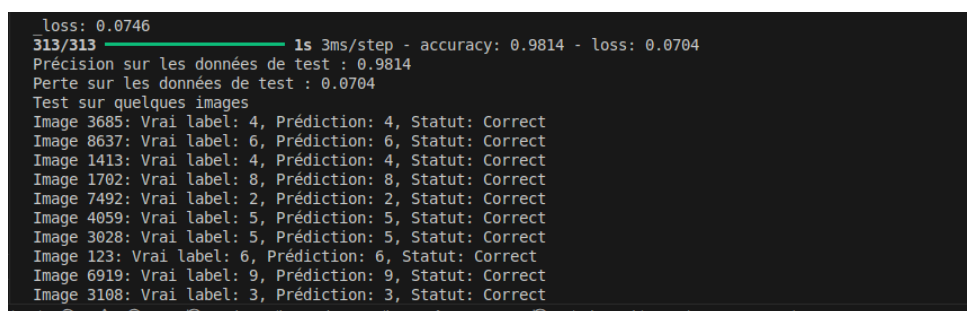
```

2025-10-08 03:11:34.120481: W external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of 169344000 exceeds 10% of free system memory.
Epoch 1/10
422/422 ————— 6s 11ms/step - accuracy: 0.9171 - loss: 0.2759 - val_accuracy: 0.9717 - val_loss: 0.0950
Epoch 2/10
422/422 ————— 4s 10ms/step - accuracy: 0.9661 - loss: 0.1126 - val_accuracy: 0.9803 - val_loss: 0.0703
Epoch 3/10
422/422 ————— 7s 16ms/step - accuracy: 0.9756 - loss: 0.0788 - val_accuracy: 0.9790 - val_loss: 0.0689
Epoch 4/10
422/422 ————— 10s 15ms/step - accuracy: 0.9804 - loss: 0.0613 - val_accuracy: 0.9805 - val_loss: 0.0647
Epoch 5/10

```

FIGURE 2 – Epochs

Afin de tester le modèle, nous avons ajouté le code de sorte à intégrer un test de reconnaissance des chiffres à partir des données d'entraînement et voici les résultats : Nous



```

loss: 0.0746
313/313 ————— 1s 3ms/step - accuracy: 0.9814 - loss: 0.0704
Précision sur les données de test : 0.9814
Perte sur les données de test : 0.0704
Test sur quelques images
Image 3685: Vrai label: 4, Prédiction: 4, Statut: Correct
Image 8637: Vrai label: 6, Prédiction: 6, Statut: Correct
Image 1413: Vrai label: 4, Prédiction: 4, Statut: Correct
Image 1702: Vrai label: 8, Prédiction: 8, Statut: Correct
Image 7492: Vrai label: 2, Prédiction: 2, Statut: Correct
Image 4059: Vrai label: 5, Prédiction: 5, Statut: Correct
Image 3028: Vrai label: 5, Prédiction: 5, Statut: Correct
Image 123: Vrai label: 6, Prédiction: 6, Statut: Correct
Image 6919: Vrai label: 9, Prédiction: 9, Statut: Correct
Image 3108: Vrai label: 3, Prédiction: 3, Statut: Correct

```

FIGURE 3 – Résultats tests

constatons que le modèle reconnaît avec précision les chiffres des données de test avec une précision moyenne de 0.98.

Integration du serveur Flask et Docker Dans la logique du Tp nous avons realiser un serveur Flask pour exposer le modele et permettre les test. Afin de tester le fonctionnement du serveur nous avons ajouter un fichier de test qui permet de generer des images ded chiffre aleatoirement et de les envoyer au serveur. D'apres l'image precedente

```
* Running on http://127.0.0.1:5000
* Running on http://10.30.238.41:5000
INFO:werkzeug:Press CTRL+C to quit
1/1 0s 77ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:42] "POST /predict HTTP/1.1" 200 -
1/1 0s 52ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:43] "POST /predict HTTP/1.1" 200 -
1/1 0s 45ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:43] "POST /predict HTTP/1.1" 200 -
1/1 0s 37ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:43] "POST /predict HTTP/1.1" 200 -
1/1 0s 33ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:43] "POST /predict HTTP/1.1" 200 -
1/1 0s 38ms/step
INFO:werkzeug:127.0.0.1 - - [08/Oct/2025 02:25:43] "POST /predict HTTP/1.1" 200 -
```

FIGURE 4 – Log sur serveur Flask lors de la reception des requettes

```
-----
Différence maximale: 0.000000
✓ Tout est cohérent entre le test local et l'API

=====
[4] TEST SUR 5 IMAGES SUPPLÉMENTAIRES
=====
✓ Image 1: Vrai=2, Local=2, API=2
✓ Image 2: Vrai=1, Local=1, API=1
✓ Image 3: Vrai=0, Local=0, API=0
✓ Image 4: Vrai=4, Local=4, API=4
✓ Image 5: Vrai=1, Local=1, API=1
=====
```

FIGURE 5 – Caption

fort est a constater que les reponse du modele par le serveur flask realiser sont tres precises, sur les 5 donnees transmises le modele reconnait exactement ces 5 chiffres donc les le serveur est fonctionnel et precis dans sa prediction.

4 Partie 3 : Ingénierie du Deep Learning

4.1 Versionnement avec Git

Le projet a été versionné avec Git, permettant le suivi des modifications et la collaboration. Les commandes utilisées :

```
1 git init
2 git add .
3 git commit -m "Initial commit"
4 git remote add origin <URL>https://github.com/Folong-zidane/
  machine_learning-FOLONGZIDANE.git
5 git push -u origin master
```

4.2 Suivi avec MLflow

MLflow a été intégré pour tracer les paramètres (epochs, batch_size, dropout_rate) et métriques (test_accuracy) de chaque expérimentation, facilitant la comparaison des modèles.

4.3 Conteneurisation Docker

L'application Flask a été conteneurisée avec Docker, incluant :

- Image de base Python 3.9-slim
- Installation des dépendances
- Exposition du port 5000
- API REST pour les prédictions

4.4 Déploiement et CI/CD

Question 1 : Pipeline CI/CD

Un pipeline GitHub Actions pourrait automatiser :

1. Construction de l'image Docker à chaque push
2. Tests automatisés du modèle
3. Déploiement sur Google Cloud Run
4. Mise à jour automatique en production

Question 2 : Indicateurs de monitoring

Trois types d'indicateurs clés :

1. **Performance** : Latence des prédictions, throughput, temps de réponse
2. **Qualité** : Accuracy en production, distribution des prédictions, détection de drift
3. **Infrastructure** : Utilisation CPU/mémoire, disponibilité du service, erreurs HTTP

5 Conclusion

Ce TP a permis de maîtriser le cycle complet d'un projet de Deep Learning, de la conception du modèle jusqu'au déploiement en production, en intégrant les bonnes pratiques d'ingénierie logicielle (versionnement, conteneurisation, monitoring).