Rapport du tp1 de ML Avancé

NOUNDJEU Franck

September 2025

Chapter 1

Introduction

L'objectif de ce travail pratique est de concevoir et déployer un modèle d'apprentissage profond basé sur le jeu de données **MNIST**. Nous utilisons **TensorFlow/Keras** pour entraîner un réseau de neurones fully-connected capable de reconnaître les chiffres manuscrits.

Le modèle est ensuite évalué afin de mesurer sa précision sur un jeu de test indépendant. Dans un second temps, nous intégrons un suivi expérimental avec **MLflow**, ce qui permet d'enregistrer les hyperparamètres, les métriques et le modèle entraîné pour une meilleure reproductibilité.

Enfin, le modèle est exposé sous forme d'API grâce au framework **Flask**, ce qui permet d'effectuer des prédictions via des requêtes HTTP. Pour rendre le déploiement portable, une image **Docker** est créée afin d'exécuter l'application dans un environnement isolé et reproductible.

Chapter 2

Fondations du Deep learning

2.1 Exercice 1

2.1.1 Question 1 : Expliquez l'utilité des couches Dense et Dropout. Pourquoi la fonction d'activation softmax est-elle utilisée dans la couche de sortie pour ce problème de classification ?

Couche Dense La couche Dense est une couche de neurones entièrement connectée (fully-connected), où chaque neurone de la couche est connecté à chaque entrée. Dans le code du tp, la première couche

```
Dense(512, activation='relu', input_shape=(784,))
```

prend les 784 pixels d'entrée (images MNIST aplaties de 28x28) et produit 512 sorties. Elle effectue une transformation linéaire suivie d'une activation non linéaire (`relu`), ce qui permet au modèle d'apprendre des relations complexes dans les données. Elle est essentielle pour apprendre des caractéristiques abstraites à partir des pixels bruts. La fonction d'activation `relu` (Rectified Linear Unit) introduit de la non-linéarité, permettant au réseau de modéliser des relations non linéaires entre les pixels et les classes.

Couche Dropout La couche Dropout(0.2) désactive aléatoirement 20 % des neurones de la couche précédente pendant l'entraînement. Cela réduit le risque de surapprentissage (overfitting) en empêchant le modèle de trop dépendre de neurones spécifiques.

Dans le code du tp, elle est placée après la couche Dense(512) pour régulariser le modèle, améliorant sa généralisation sur les données de

test. Elle agit comme une forme d'ensemble implicite, forçant le réseau à apprendre des représentations plus robustes.

Fonction d'activation Softmax Pourquoi dans la couche de sortie ? : La couche de sortie Dense(10, activation='softmax') utilise softmax car il s'agit d'un problème de classification multiclasse (10 chiffres de 0 à 9 dans MNIST). La fonction softmax transforme les sorties brutes (logits) en probabilités normalisées pour chaque classe, où la somme des probabilités est égale à 1.

Exemple : Pour une image donnée, softmax peut produire [0.1, 0.05, 0.7, ..., 0.02], où chaque valeur représente la probabilité que l'image appartienne à une classe (par exemple, 70 % de chance d'être un "2"). Cela facilite l'interprétation des prédictions et est compatible avec la fonction de perte

`sparse_categorical_crossentropy`

, qui compare les probabilités prédites aux étiquettes réelles.

2.1.2 Question 2 : L'optimiseur Adam est utilisé. Faites une recherche et expliquez brièvement en quoi il s'agit d'une amélioration par rapport à la SGD simple.

SGD (Stochastic Gradient Descent) simple : SGD met à jour les poids du modèle en calculant le gradient de la fonction de perte par rapport aux paramètres, puis en ajustant ces paramètres dans la direction qui réduit la perte, avec un taux d'apprentissage fixe.

Limites :

- Sensible au choix du taux d'apprentissage (trop grand : instabilité ; trop petit : convergence lente).
- Peut rester coincé dans des minima locaux ou des plateaux.
- Ne gère pas bien les gradients de magnitudes variables (problème des échelles).

Optimiseur Adam (Adaptive Moment Estimation) : Adam combine les avantages de deux extensions de SGD : AdaGrad (adaptation du taux d'apprentissage par dimension) et RMSProp (moyenne mobile des gradients pour gérer les oscillations).

Améliorations :

- Taux d'apprentissage adaptatif : Adam ajuste dynamiquement le taux d'apprentissage pour chaque paramètre en utilisant des estimations du premier moment (moyenne des gradients) et du second moment (moyenne des carrés des gradients). Cela permet une convergence plus rapide et plus stable.
- Moins sensible aux hyperparamètres : Les valeurs par défaut d'Adam (par exemple, learning_rate=0.001, beta_1=0.9, beta_2=0.999) fonctionnent bien dans la plupart des cas, contrairement à SGD où le taux d'apprentissage doit être ajusté manuellement.
- Robustesse aux gradients bruités : Adam gère mieux les gradients instables ou épars, fréquents dans les réseaux neuronaux profonds.
- Convergence rapide : En combinant l'élan (momentum) et l'adaptation du taux d'apprentissage, Adam converge souvent plus vite que SGD, surtout pour des problèmes comme MNIST avec des données normalisées.

2.1.3 Question 3 : Comment les concepts de "vectorisation" et de "calculs par lots" sont-ils appliqués dans le code ci-dessus ?

Vectorisation La vectorisation consiste à effectuer des opérations sur des ensembles de données (par exemple, tableaux ou matrices) en une seule opération, plutôt que d'utiliser des boucles explicites. Cela exploite les optimisations des bibliothèques comme NumPy et TensorFlow pour des calculs rapides sur des structures de données entières.

Application dans le code du tp :

- Normalisation des données :

```
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
```

  Ici, la division par 255.0 est appliquée simultanément à tous les pixels de toutes les images dans

  `x_train (forme : 60000x28x28) et x_test (10000x28x28)`

  grâce à la vectorisation de NumPy. Sans vectorisation, il faudrait une boucle sur chaque pixel.

- Redimensionnement :

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

La méthode reshape transforme les tableaux en une seule opération, convertissant les images 28x28 en vecteurs de 784 éléments pour toutes les images simultanément.

- Calculs dans le modèle : Dans model.fit, les opérations des couches Dense (produits matriciels, biais, activation relu ou softmax) sont vectorisées. Par exemple, la couche Dense(512) effectue un produit matriciel entre la matrice d'entrée (batch_size x 784) et la matrice des poids (784 x 512) en une seule opération, grâce à TensorFlow.

Calculs par lots (Batch Processing) Les calculs par lots consistent à traiter les données par groupes (batches) plutôt que toutes à la fois ou une par une. Cela équilibre l'efficacité computationnelle et la stabilité de l'apprentissage.

Application dans le code du tp :

- Entraînement par lots :

```
history = model.fit(x_train,
    y_train, epochs=5,
    batch_size=128,
    validation_split=0.1)
```

Le paramètre batch_size=128 indique que le modèle traite 128 images à la fois par itération. Les 60000 images d'entraînement sont divisées en lots de 128 (environ 469 lots par époque, car $60000 * 0.9/128 \approx 421$ après validation_split=0.1). Pour chaque lot, TensorFlow calcule les gradients, met à jour les poids avec l'optimiseur Adam, et passe au lot suivant. Cela réduit la mémoire nécessaire par rapport à un traitement de toutes les données simultanément et accélère l'entraînement par rapport à un traitement image par image.

- Évaluation :

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

Lors de l'évaluation, TensorFlow traite également les données de test par lots (taille par défaut ou définie en interne) pour calculer la perte et la précision, exploitant ainsi les calculs par lots pour des performances optimales.

- Vectorisation dans les lots : Les calculs par lots sont combinés avec la vectorisation. Par exemple, pour un lot de 128 images, la couche Dense(512) effectue un produit matriciel sur une matrice d'entrée de forme (128, 784), ce qui est optimisé par TensorFlow pour des calculs parallèles sur GPU/TPU.

La **vectorisation** réduit le temps de calcul en exploitant les optimisations des processeurs (CPU/GPU) et des bibliothèques comme NumPy/TensorFlow. Sans cela, boucler sur 60000 images ou 784 pixels serait extrêmement lent. Le **calcul par lots** permet un entraînement stable (gradients moyens sur un lot plutôt qu'une seule image) et efficace (moins de mémoire que tout traiter d'un coup). Le choix de batch_size=128 est un compromis courant pour équilibrer vitesse et stabilité.

Chapter 3

Ingénierie du Deep Learning

3.1 Exercice 5 : Déploiement et CI/CD

3.1.1 Question 1 : Expliquez comment un pipeline de CI/CD (par exemple, avec GitHub Actions) pourrait automatiser la construction et le déploiement de votre image Docker sur un service comme Google Cloud Run ou Amazon Elastic Container Service.

Un pipeline de **CI/CD** (Continuous Integration / Continuous Deployment) avec **GitHub Actions** permet d'automatiser la construction et le déploiement d'une application. Le fonctionnement est le suivant :

- **Intégration continue (CI)** : à chaque commit ou pull request sur GitHub, un workflow YAML est déclenché. Ce workflow vérifie le code (lint, tests), construit l'image Docker (`docker build -t mon_image .`) et exécute des tests unitaires à l'intérieur du conteneur.
- **Déploiement continu (CD)** : si la CI est validée, l'image est poussée dans un registre de conteneurs (par exemple Docker Hub, GitHub Container Registry, Google Artifact Registry ou Amazon ECR) via la commande :

  ```
  docker push gcr.io/mon-projet/mon_image:latest
  ```

  Ensuite, l'application est déployée automatiquement :

– sur **Google Cloud Run** via `gcloud run deploy mon-service --image gcr.io/mon-projet/mon_image:latest`,

– ou sur **Amazon ECS** via `aws ecs update-service --cluster monCluster --service monService --force-new-deployment`.

Ainsi, toute nouvelle version du code déclenche automatiquement la construction, le test et le déploiement de l'application sans intervention manuelle.

3.1.2 Question 2 : Une fois le modèle déployé, quels sont les indicateurs clés que vous mettriez en place pour le monitoring et le débogage en production? Citez au moins trois types d'indicateurs.

Une fois le modèle déployé en production, il est essentiel de définir des **indicateurs clés** pour assurer le monitoring et le débogage. On distingue principalement trois catégories d'indicateurs :

1. **Indicateurs système (infrastructure)** :
   - Utilisation CPU, RAM, GPU.
   - Latence des requêtes (temps de réponse).
   - Nombre de requêtes traitées par seconde (QPS).
   - Taux d'erreurs HTTP (4xx, 5xx).
2. **Indicateurs de performance du modèle (métier / ML)** :
   - Accuracy, précision, rappel, F1-score sur des échantillons en production.
   - Taux de prédictions nulles ou invalides.
   - Distribution des classes prédites (pour détecter un éventuel *data drift*).
3. **Indicateurs de robustesse et de qualité des données** :
   - Présence de valeurs manquantes ou d'anomalies dans les entrées.
   - Détection de *data drift* ou *concept drift*.
   - Suivi des entrées hors domaine (non représentées lors de l'entraînement).

Chapter 4

Conclusion

Ce TP illustre un cycle complet de développement en **Machine Learning** : de la construction du modèle jusqu'à son déploiement en production. Nous avons pu mettre en place un modèle simple mais performant pour la classification de chiffres manuscrits, tout en assurant sa traçabilité via **MLflow**.

L'utilisation de **Flask** et de **Docker** a permis de rendre l'application accessible sous forme de service web, offrant ainsi une base solide pour un déploiement dans le cloud. Ce travail met en évidence l'importance non seulement de l'entraînement du modèle, mais aussi de l'ingénierie autour de celui-ci (suivi, monitoring, conteneurisation), qui sont des étapes essentielles dans tout projet de **MLOps**.