Rapport de Travaux Pratiques

De la Conception au Déploiement de Modèles de Deep Learning

Votre Nom
Département Génie Informatique, ENSPY
bouiyodajoseph@gmail.com
Lien vers le dépôt GitHub (Branche TP1)

22 janvier 2026

Table des matières

1 Introduction

Ce document constitue le rapport pour les Travaux Pratiques portant sur le cycle de vie complet d'un modèle de Deep Learning. L'objectif est de mettre en œuvre les concepts fondamentaux de l'apprentissage profond, de l'entraînement d'un premier modèle à son déploiement en tant qu'application conteneurisée. Ce rapport détaille chaque étape du processus, des aspects théoriques aux implémentations pratiques, en répondant aux questions posées dans le cadre du TP.

Première partie
Fondations du Deep Learning

2 Concepts Théoriques

2.1 Rappel des modèles linéaires et de l'optimisation stochastique

2.1.1 Différence entre Descente de Gradient (GD) et Descente de Gradient Stochastique (SGD)

La principale différence entre la Descente de Gradient (GD) et la Descente de Gradient Stochastique (SGD) réside dans la quantité de données utilisées pour calculer le gradient de la fonction de coût à chaque mise à jour des poids du modèle.

- **Descente de Gradient (GD)**, ou "Batch Gradient Descent", calcule le gradient en utilisant **l'intégralité du jeu de données d'entraînement**. Cette méthode est précise mais extrêmement lente et coûteuse en mémoire pour de grands jeux de données.
- **Descente de Gradient Stochastique (SGD)** met à jour les poids en utilisant le gradient calculé sur **un seul exemple** de données à la fois, choisi aléatoirement. C'est beaucoup plus rapide mais peut produire une convergence bruitée.
- **Mini-Batch SGD** est le compromis le plus utilisé en pratique. Il calcule le gradient sur un petit sous-ensemble de données (un "lot" ou "batch"), offrant un bon équilibre entre vitesse de calcul et stabilité de la convergence.

La SGD (et ses variantes mini-batch) est préférée en Deep Learning car elle est beaucoup plus efficace sur les jeux de données massifs et le bruit inhérent à ses mises à jour peut aider l'algorithme à échapper aux minima locaux.

2.2 Compréhension des réseaux de neurones modernes

2.2.1 Rôles des couches

Un réseau de neurones est typiquement structuré en trois types de couches :

- **Couche d'entrée (Input Layer) :** C'est la porte d'entrée du réseau. Elle reçoit les données brutes (pour MNIST, un vecteur de 784 pixels) et les transmet à la première couche cachée. Elle n'effectue aucun calcul.
- **Couches cachées (Hidden Layers) :** Ce sont les couches intermédiaires où les calculs ont lieu. Leur rôle est d'apprendre des représentations de plus en plus abstraites et complexes des données. Chaque couche apprend des motifs à partir des sorties de la couche précédente.
- **Couche de sortie (Output Layer) :** C'est la dernière couche qui produit le résultat final du modèle. Sa structure dépend de la tâche (par exemple, 10 neurones avec une activation `softmax` pour la classification des 10 chiffres de MNIST).

2.2.2 Rétropropagation du gradient (Backpropagation)

C'est l'algorithme fondamental qui permet d'entraîner un réseau de neurones. Il fonctionne en deux phases :

1. **Passe avant (Forward Pass) :** Les données d'entrée traversent le réseau de couche en couche jusqu'à la sortie pour produire une prédiction.
2. **Passe arrière (Backward Pass) :** L'erreur entre la prédiction et la valeur attendue est calculée. La rétropropagation consiste à propager cette erreur "vers l'arrière", de la couche de sortie jusqu'à la première couche, en utilisant la règle de dérivation en chaîne pour calculer la contribution de chaque poids à l'erreur finale (le gradient). Les poids sont ensuite ajustés dans la direction opposée du gradient pour minimiser l'erreur.

3 Exercice 1 : Construction d'un réseau de neurones avec Keras

Cet exercice consiste à implémenter un classifieur pour le jeu de données MNIST. Le code ci-dessous a été utilisé.

```
# Importation des bibliotheques
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Chargement du jeu de donnees MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Pre-traitement des donnees
# Normalisation des pixels entre 0 et 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Aplatissement des images 28x28 en vecteurs 1D de 784 pixels
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# Construction du modele Sequential
model = keras.Sequential([
    # Premiere couche Dense avec 512 neurones et activation ReLU
    keras.layers.Dense(512, activation='relu', input_shape=(784,)),
    # Couche de Dropout pour la regularisation (eviter le surapprentissage)
    keras.layers.Dropout(0.2),
    # Couche de sortie avec 10 neurones (un par classe) et activation softmax
    keras.layers.Dense(10, activation='softmax')
])

# Compilation du modele
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Entrainement du modele
history = model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=128,
    validation_split=0.1
)

# Evaluation de la performance finale sur le jeu de test
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Precision sur les donnees de test: {test_acc:.4f}")

# Sauvegarde du modele entraine
model.save("mnist_model.h5")
```

Listing 1 – Script d'entraînement `train_model.py`

3.1 Question 1 : Utilité des couches `Dense`, `Dropout` et `softmax`

Couche `Dense` : C'est une couche entièrement connectée où chaque neurone est connecté à tous les neurones de la couche précédente. Son rôle est d'apprendre des relations complexes et des motifs dans les données.

Couche `Dropout` : C'est une technique de régularisation qui combat le surapprentissage. Durant l'entraînement, elle désactive aléatoirement une fraction des neurones (ici 20%), forçant le réseau à apprendre des caractéristiques plus robustes et moins interdépendantes.

Activation `softmax` : Elle est utilisée dans la couche de sortie pour les problèmes de classification multi-classe. Elle transforme les scores bruts des 10 neurones en une distribution de probabilités, où la somme des probabilités est égale à 1. Cela permet d'interpréter la sortie comme le degré de confiance du modèle pour chaque chiffre.

3.2 Question 2 : Améliorations de l'optimiseur `adam`

L'optimiseur `Adam` (Adaptive Moment Estimation) est une amélioration de la SGD simple car il combine deux concepts clés :

1. **Taux d'Apprentissage Adaptatif :** Adam maintient un taux d'apprentissage individuel pour chaque poids du réseau et l'ajuste dynamiquement, permettant une convergence plus rapide et plus stable.
2. **Momentum :** Adam utilise une moyenne mobile des gradients passés pour accélérer la convergence dans les directions pertinentes et amortir les oscillations.

3.3 Question 3 : "Vectorisation" et "Calculs par lots"

Vectorisation : Ce concept est appliqué lors du pré-traitement. L'opération `x_train / 255.0` est une opération vectorisée qui divise chaque pixel de l'ensemble des 60 000 images en une seule fois, sans utiliser de boucle, ce qui est extrêmement efficace.

Calcul par Lots : Ce concept est défini par le paramètre `batch_size=128` dans `model.fit()`. Le modèle met à jour ses poids non pas après chaque image, mais après avoir traité un lot de 128 images. Il calcule l'erreur moyenne sur ce lot et effectue une seule mise à jour, ce qui constitue un excellent compromis entre vitesse et stabilité.

Deuxième partie
Ingénierie du Deep Learning

4 Exercice 2 : Versionnement avec Git et GitHub

Les instructions pour le versionnement du projet ont été suivies. Un dépôt Git local a été initialisé, et un dépôt distant public a été créé sur GitHub. Le code source du projet, ainsi que ce rapport, sont versionnés et disponibles à l'adresse suivante : https://github.com/BouiyodaJoseph/DeepLearning_TP/tree/TP1

5 Exercice 3 : Suivi des expérimentations avec MLflow

Le script `train_model.py` a été modifié pour intégrer MLflow, un outil de gestion du cycle de vie des modèles. Cela permet de suivre et de comparer les expérimentations de manière rigoureuse. Le code d'entraînement a été encapsulé dans un bloc `with mlflow.start_run():` pour enregistrer automatiquement les hyperparamètres (époques, taille du lot) et les métriques de performance (précision).

6 Exercice 4 : Conteneurisation avec Docker et création d'une API

Pour rendre le modèle déployable et accessible, une API web a été créée avec Flask et l'ensemble de l'application a été conteneurisé avec Docker. Trois fichiers ont été créés :

— `requirements.txt` pour lister les dépendances.
— `app.py` pour créer le serveur Flask qui charge le modèle et expose un point de terminaison `/predict`.
— `Dockerfile` pour définir les instructions de construction de l'image Docker.

7 Exercice 5 : Déploiement et CI/CD

7.1 Question 1 : Pipeline de CI/CD

Un pipeline de CI/CD (Intégration Continue / Déploiement Continu) avec GitHub Actions automatise entièrement le déploiement. Le processus est le suivant :

1. **Déclenchement :** Un `git push` sur la branche principale déclenche le workflow.
2. **Construction :** GitHub Actions exécute la commande `docker build` en utilisant le `Dockerfile` du projet pour créer l'image Docker de l'application.
3. **Publication :** L'image est ensuite "poussée" vers un registre de conteneurs (ex : Google Artifact Registry, Docker Hub).
4. **Déploiement :** Enfin, le pipeline exécute une commande (ex : `gcloud run deploy`) pour déployer la nouvelle version de l'image sur un service cloud comme Google Cloud Run.

Ce processus garantit des déploiements rapides, fiables et reproductibles.

7.2 Question 2 : Indicateurs de Monitoring en Production

Une fois un modèle déployé, il est crucial de le surveiller. Voici trois types d'indicateurs clés :

Indicateurs Opérationnels : Ils mesurent la santé de l'API.
— **Latence :** Le temps de réponse pour chaque prédiction.
— **Taux d'Erreur :** Le pourcentage de requêtes qui échouent (erreurs 500).
— **Utilisation des Ressources :** Consommation de CPU, RAM, etc.

Indicateurs de Performance du Modèle : Ils mesurent la qualité des prédictions.
— **Distribution des Prédictions :** Surveiller si la répartition des chiffres prédits change de manière anormale.
— **Score de Confiance :** Suivre la probabilité moyenne des prédictions.

Indicateurs de Dérive des Données (Data Drift) : Ils détectent si les données reçues en production diffèrent des données d'entraînement.
— **Distribution des Caractéristiques d'Entrée :** Suivre les statistiques des données d'entrée (ex : moyenne des pixels). Une dérive peut indiquer qu'il est temps de réentraîner le modèle.

8 Conclusion

Ce Travail Pratique a permis de couvrir de manière exhaustive le cycle de vie d'un projet de Deep Learning. Partant des fondements théoriques, nous avons implémenté, entraîné et évalué un réseau de neurones fonctionnel. Par la suite, nous avons appliqué des pratiques d'ingénierie logicielle modernes, incluant le versionnement avec Git, le suivi d'expériences avec MLflow, et la conteneurisation avec Docker, jusqu'à la planification conceptuelle du déploiement automatisé et du monitoring. Cette approche holistique fournit une base solide pour le développement de projets d'IA robustes et prêts pour la production.