Rapport de Travaux Pratiques

De la Conception au Déploiement de Modèles de Deep Learning

Joseph BOUIYODA
Département Génie Informatique, ENSPY
bouiyodajoseph@gmail.com

21 janvier 2026

Table des matières

1 Introduction

Ce document constitue le rapport pour la première partie des Travaux Pratiques de Deep Learning. L'objectif est de consolider les bases théoriques de l'apprentissage profond et de mettre en pratique la construction, l'entraînement et l'évaluation d'un premier réseau de neurones pour un problème de classification d'images. Le rapport couvrira également une analyse conceptuelle des enjeux de déploiement et de monitoring, qui feront l'objet de la suite des TPs.

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 descente de gradient classique (GD)**, ou "Batch Gradient Descent", calcule le gradient de la fonction de coût en utilisant l'intégralité du jeu de données d'entraînement à chaque itération. C'est une méthode très précise mais extrêmement coûteuse en calculs avec de grands jeux de données.
- **La descente de gradient stochastique (SGD)** met à jour les poids du modèle en utilisant le gradient calculé sur un seul exemple de données à la fois, choisi aléatoirement.
- **La "Mini-Batch SGD"** est un compromis qui calcule le gradient sur un petit sous-ensemble de données (un "lot" ou "batch"). C'est la variante la plus utilisée en pratique.

2.1.2 Préférence de la SGD en Deep Learning

La SGD (et ses variantes) est préférée pour deux raisons majeures :

1. **Efficacité calculatoire :** Le Deep Learning utilise des jeux de données massifs. Calculer le gradient sur des millions d'exemples à chaque mise à jour (GD) est irréalisable. La SGD est beaucoup plus rapide.
2. **Échapper aux minima locaux :** Le "bruit" introduit par l'estimation du gradient sur un petit lot de données peut aider l'algorithme à "sauter" hors des minima locaux de la fonction de coût, ce qui peut potentiellement l'amener vers un meilleur minimum global.

2.2 Compréhension des réseaux de neurones modernes

2.2.1 Rôles des couches

- **Couche d'entrée (Input Layer) :** 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) :** Cœur du réseau où les calculs ont lieu. Chaque couche apprend à reconnaître des caractéristiques de plus en plus complexes à partir des sorties de la couche précédente.
- **Couche de sortie (Output Layer) :** Produit le résultat final. 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 d'entraînement des réseaux de neurones, qui fonctionne en deux temps :

Passe avant (Forward Pass) : Les données traversent le réseau de l'entrée à la sortie pour produire une prédiction.

Passe arrière (Backward Pass) : L'erreur entre la prédiction et la vraie valeur est calculée. Cette erreur est ensuite propagée "vers l'arrière" à travers le réseau pour calculer la contribution de chaque poids à l'erreur. Les poids sont ensuite ajustés pour réduire cette erreur.

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

3.1 Implémentation du Modèle

Le code ci-dessous implémente un réseau de neurones simple pour la classification des chiffres manuscrits de la base de données MNIST.

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

# 2. Chargement du jeu de donnees MNIST
# Keras fournit un acces direct a ce jeu de donnees classique.
# Il est divise en un ensemble d'entrainement et un ensemble de test.
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# 3. Pre-traitement des donnees
# Les pixels des images sont initialement des entiers de 0 a 255.
# On les normalise en flottants entre 0 et 1 pour aider le modele a converger.
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Les images 28x28 sont "aplaties" en vecteurs de 784 pixels
# pour etre traitees par des couches denses.
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# 4. Construction du modele Sequential
# Un modele Sequential est une pile lineaire de couches.
model = keras.Sequential([
    # Premiere couche cachee, entierement connectee (Dense).
    # 512 neurones, fonction d'activation ReLU.
    # input_shape specifie la taille des donnees d'entree.
    keras.layers.Dense(512, activation='relu', input_shape=(784,)),

    # Couche de Dropout pour la regularisation.
    # Elle desactive aleatoirement 20% des neurones a chaque etape
    # pour eviter le surapprentissage.
    keras.layers.Dropout(0.2),

    # Couche de sortie avec 10 neurones (un par chiffre).
    # L'activation softmax convertit les sorties en distribution de probabilites
    .
    keras.layers.Dense(10, activation='softmax')
])

# 5. Compilation du modele
# On definit l'optimiseur, la fonction de perte et les metriques a suivre.
model.compile(
    optimizer='adam', # Adam est un optimiseur efficace et populaire.
```

```
    loss='sparse_categorical_crossentropy', # Adapte a la classification multi-classe.
    metrics=['accuracy'] # On veut suivre la precision.
)

# 6. Entrainement du modele
# Le modele est entraine sur les donnees d'entrainement.
history = model.fit(
    x_train, y_train,
    epochs=5,                # 5 passages complets sur les donnees.
    batch_size=128,          # Mise a jour des poids tous les 128 images.
    validation_split=0.1     # 10% des donnees sont mises de cote pour la validation.
)

# 7. Evaluation de la performance finale
# Le modele est evalue sur le jeu de test qu'il n'a jamais vu.
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Precision sur les donnees de test: {test_acc:.4f}")

# 8. Sauvegarde du modele entraine
model.save("mnist_model.h5")
print("Modele sauvegarde sous mnist_model.h5")
```

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

3.2 Résultats de l'Entraînement

L'exécution du script a produit la sortie suivante, montrant la progression de l'entraînement sur 5 époques et le résultat final de l'évaluation.

```
Epoch 1/5
422/422 [==============================] - 12s 19ms/step - accuracy: 0.9150 -
    loss: 0.3007 - val_accuracy: 0.9653 - val_loss: 0.1248
Epoch 2/5
422/422 [==============================] - 6s 13ms/step - accuracy: 0.9623 -
    loss: 0.1292 - val_accuracy: 0.9745 - val_loss: 0.0905
Epoch 3/5
422/422 [==============================] - 6s 14ms/step - accuracy: 0.9736 -
    loss: 0.0896 - val_accuracy: 0.9777 - val_loss: 0.0779
Epoch 4/5
422/422 [==============================] - 5s 13ms/step - accuracy: 0.9797 -
    loss: 0.0678 - val_accuracy: 0.9790 - val_loss: 0.0723
Epoch 5/5
422/422 [==============================] - 5s 13ms/step - accuracy: 0.9840 -
    loss: 0.0539 - val_accuracy: 0.9805 - val_loss: 0.0713

313/313 [==============================] - 1s 4ms/step - accuracy: 0.9781 - loss
    : 0.0699
Precision sur les donnees de test: 0.9781
Modele sauvegarde sous mnist_model.h5
```

On observe que la précision sur l'ensemble de validation (`val_accuracy`) augmente à chaque époque, et le modèle atteint une précision finale de **97.81%** sur l'ensemble de test.

3.3 Analyse et Questions

3.3.1 Question 1 : Utilité des couches `Dense`, `Dropout` et de l'activation `softmax`

Couche `Dense` : C'est une couche entièrement connectée, dont le rôle est d'apprendre des relations complexes entre les caractéristiques. La première couche `Dense` (512 neurones)

apprend des motifs à partir des pixels bruts, tandis que la seconde (10 neurones) apprend à classer ces motifs.

Couche `Dropout` : C'est une technique de régularisation pour lutter contre le surapprentissage. En désactivant aléatoirement 20% des neurones à chaque étape de l'entraînement, elle force le réseau à apprendre des caractéristiques plus robustes et moins dépendantes de neurones spécifiques.

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 de sortie 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.

3.3.2 Question 2 : Améliorations de l'optimiseur `adam` par rapport à la SGD

L'optimiseur `Adam` (Adaptive Moment Estimation) améliore la SGD simple sur deux points principaux :

— **Taux d'Apprentissage Adaptatif :** Adam maintient un taux d'apprentissage individuel pour chaque poids du réseau et l'ajuste dynamiquement, ce qui permet une convergence plus rapide et plus stable.

— **Conservation d'un "Momentum" :** Adam utilise une moyenne mobile des gradients passés pour accélérer la convergence dans les directions pertinentes et amortir les oscillations.

Ces caractéristiques en font un optimiseur très efficace et facile à utiliser pour la plupart des problèmes.

3.3.3 Question 3 : Application de la "vectorisation" et des "calculs par lots"

Vectorisation : Ce concept est appliqué lors du pré-traitement des données. L'opération `x_train / 255.0` est une opération vectorisée qui divise chaque pixel de l'ensemble des 60 000 images par 255 en une seule fois, sans utiliser de boucle `for`, ce qui est extrêmement efficace.

Calcul par Lots (Batch Processing) : Ce concept est défini par le paramètre `batch_size=128` dans la méthode `model.fit()`. Le modèle ne met pas à jour ses poids 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 des poids, ce qui est un excellent compromis entre vitesse de calcul et stabilité de l'apprentissage.

Deuxième partie
Ingénierie du Deep Learning (Analyse Conceptuelle)

4 Exercice 5 : Déploiement et CI/CD

4.1 Question 1 : Pipeline de CI/CD avec GitHub Actions

Un pipeline de CI/CD (Intégration Continue / Déploiement Continu) avec GitHub Actions automatise la mise en production d'une application à partir du code source. Le processus se déroule comme suit :

1. **Déclenchement :** Un `git push` sur la branche principale du projet déclenche automatiquement le pipeline.

2. **Authentification :** Le pipeline se connecte de manière sécurisée à un registre de conteneurs (ex : Docker Hub) et au fournisseur Cloud (ex : Google Cloud).
3. **Construction de l'image Docker :** La commande `docker build` est exécutée, utilisant le `Dockerfile` du projet pour créer une image contenant l'application.
4. **Publication de l'image :** L'image nouvellement construite est "poussée" vers le registre de conteneurs.
5. **Déploiement :** Le pipeline exécute une commande pour déployer la nouvelle image sur un service comme Google Cloud Run, qui se charge de rendre l'application accessible en ligne.

Ce processus garantit des déploiements rapides, fiables et reproductibles.

4.2 Question 2 : Indicateurs de Monitoring en Production

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

Type 1 : Indicateurs Opérationnels (Santé de l'API) Ils mesurent la performance technique du service.
- **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.

Type 2 : Indicateurs de Performance du Modèle Ils mesurent la qualité des prédictions du modèle en production.
- **Distribution des Prédictions :** Surveiller si la répartition des chiffres prédits change de manière anormale.
- **Score de Confiance :** Suivre la probabilité moyenne des prédictions pour détecter une baisse de "confiance" du modèle.

Type 3 : 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 significative peut indiquer qu'il est temps de ré-entraîner le modèle.

5 Conclusion

Ce premier travail pratique a permis de balayer les concepts fondamentaux du Deep Learning, de la théorie sur l'optimisation et les réseaux de neurones jusqu'à l'implémentation concrète d'un classifieur d'images avec Keras. Les résultats obtenus sont satisfaisants et démontrent une bonne compréhension du processus d'entraînement. Enfin, l'analyse conceptuelle des problématiques de déploiement et de monitoring offre une vision claire des défis de l'ingénierie du Deep Learning qui seront abordés dans la suite du projet.