université de BORDEAUX

2024/2025

Architectures de réseaux de neurones avancées Projet : Apprentissage continu

Meriem Kadiri

Université de Bordeaux M2 IA

31/12/2024

Table de matière

1. Introduction

1.1 Contexte

Ce projet s'inscrit dans le cadre de l'étude des architectures de réseaux de neurones avancées, avec un focus particulier sur l'apprentissage continu, appliqué à la classification d'images des datasets MNIST et FMNIST. L'objectif principal est d'explorer et de comparer différentes approches d'apprentissage continu pour la mise à jour des modèles d'intelligence artificielle dans un contexte de flux de données constant.

1.2 Objectifs du Projet

- Implémenter et comparer les approches Brute Force (BF) et Naive Fine-Tuning (NFT).

- Implémenter et évaluer l'approche Move-To-Data (MTD).

- Étudier l'adaptation des modèles aux nouvelles classes.

2. Méthodologie

2.1 Préparation des Données

2.1.1 Datasets Utilisés

MNIST : images de chiffres manuscrits (60,000 images d'entraînement, 10,000 images de test)

Fashion-MNIST : images de vêtements (60,000 images d'entraînement, 10,000 images de test)

Pour les expérimentations, utilisation de sous-ensembles :

- Données initiales : 5,000 images MNIST
- Nouvelles classes : 2,000 images Fashion-MNIST
- Test : 2,000 images (1,000 de chaque dataset)

2.1.2 Prétraitement

Normalisation des images (division par 255)

Ajustement des étiquettes Fashion-MNIST (+10) pour éviter le chevauchement avec MNIST

Création de batches de taille contrôlée pour l'apprentissage incrémental

2.2 Architecture du Modèle de Base

Réseau convolutif (CNN) avec :

- 3 couches convolutives (32, 64, 64 filtres)
- 2 couches de max pooling

- Dense layer de 128 neurones (features)
- Couche de sortie adaptative (10 ou 20 classes selon le cas)

```python
def create_base_model():
    inputs = Input(shape=(28, 28, 1))
    x = layers.Conv2D(32, (3, 3), activation='relu')(inputs)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Conv2D(64, (3, 3), activation='relu')(x)
    x = layers.MaxPooling2D((2, 2))(x)
    x = layers.Conv2D(64, (3, 3), activation='relu')(x)
    x = layers.Flatten()(x)
    x = layers.Dense(128, activation='relu', name='features')(x)
    outputs = layers.Dense(20, activation='softmax')(x)
    model = models.Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

3. Expérimentations et Résultats

3.1 Comparaison BF vs NFT

3.1.1 Protocole Expérimental

- Test sur 10 batches de données

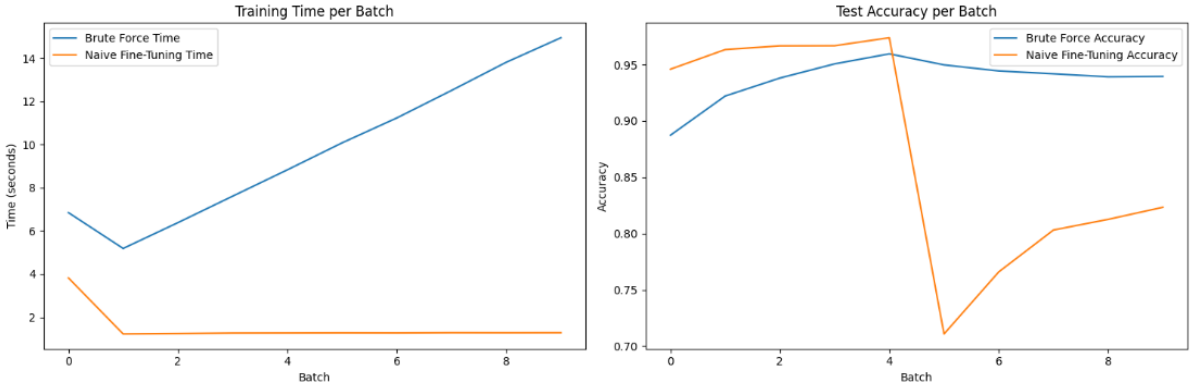
- Mesure du temps d'entraînement, de la mémoire utilisée et de la précision

- Batch size de 500 échantillons (on peut l'augmenter, ça peut prendre plus de temps ça dépends de la performance de la machine)

- Évaluation sur un ensemble de test fixe

3.1.2 Résultats Comparatifs

```
Brute Force (BF) - Avg Time: 9.74s, Avg Memory: 482.27 MB, Avg Accuracy: 93.72%
Naive Fine-Tuning (NFT) - Avg Time: 1.52s, Avg Memory: 30.54 MB, Avg Accuracy: 87.32%
```



Graphique 1 : Temps d'entraînement par lot

Observations générales :

1. **Brute Force (BF) :**

Le temps d'entraînement augmente de manière linéaire au fur et à mesure que les lots arrivent, avec un temps moyen de 9.74 secondes.

Cela peut s'expliquer par le fait que la méthode BF réentraîne le modèle sur l'ensemble des données à chaque lot, ce qui explique la valeur moyenne de la mémoire utilisée 482.27 MB. Cette approche exhaustive garantit une bonne précision avec une valeur moyenne de 93.72% mais est extrêmement coûteuse en temps de calcul.

2. **Naive Fine-Tuning (NFT) :**

Le temps d'entraînement reste constant tout au long des lots au moyen 1.52 secondes. Cela indique que NFT ne s'entraîne que sur les nouvelles données, (d'où la consomation faible en mémoire 30.54 MB) ce qui est beaucoup plus rapide.

Cependant, ce gain en rapidité pourrait entraîner des compromis en termes de précision 87.32%, notamment à cause du **catastrophic forgetting**.

Graphique 2 : Test Accuracy per Batch (Précision par lot)

Observations générales :

1. **Brute Force (BF) :**

La précision reste constamment élevée. Cela confirme que BF est robuste aux nouvelles classes, car il s'entraîne sur l'ensemble des données, anciennes et nouvelles, à chaque lot.

Bien que coûteuse, cette méthode assure une stabilité dans les performances.

2. **Naive Fine-Tuning (NFT) :**

Initialement, la précision est comparable à BF, car le modèle est performant sur les premières classes.

Cependant, à partir du lot 4, on observe une chute brutale de la précision : NFT probablement oublie les classes précédentes en se concentrant uniquement sur les nouvelles données.

Après la chute, la précision augmente progressivement, ce qui peut indiquer que le modèle commence à mieux s'adapter aux nouvelles classes. Cependant, la perte d'informations sur les anciennes classes reste un problème majeur.

3.1.3 Analyse

Impact de l'entraînement sur toutes les données (BF) :

- BF maintient une précision élevée parce qu'il intègre systématiquement toutes les données disponibles lors de chaque phase d'entraînement.
- Cela évite le catastrophic forgetting, mais au prix d'un temps d'entraînement beaucoup plus élevé.

Impact de l'entraînement uniquement sur les nouvelles données (NFT) :

- NFT est rapide et efficace en termes de temps, car il ne s'entraîne que sur les nouvelles données.
- Cependant, cette approche est très sensible au catastrophic forgetting, car elle ne conserve pas les connaissances acquises sur les anciennes classes.

Explications possibles pour la chute de précision de NFT :

- Lorsque de nouvelles classes apparaissent, le modèle NFT réaffecte mal ses poids, ce qui dégrade ses performances sur les anciennes classes.
- Cette situation illustre un compromis classique entre rapidité et stabilité dans les approches d'apprentissage incrémental.

3.2 Implémentation et Évaluation MTD en comparaison avec BF /NFT

3.2.1 Détails d'Implémentation

Chargement et Préparation des Données

- Les datasets **MNIST** (chiffres manuscrits) et **Fashion-MNIST** (articles de mode) sont utilisés.
- Les classes des deux datasets sont séparées :

Les classes 0–9 sont issues de MNIST.

Les classes 10–19 sont issues de Fashion-MNIST (les labels sont décalés de 10 pour éviter les conflits avec MNIST).

- Données divisées en :

Classes initiales : Un sous-ensemble de MNIST pour entraîner le modèle de base.

Nouvelles classes : Un sous-ensemble de Fashion-MNIST ajouté au modèle via MTD.

Données de test : Une combinaison des deux pour évaluer les performances du modèle.

Création du Modèle de Base

Un modèle CNN (Convolutional Neural Network) simple est conçu avec deux parties principales :

- **Extraction de caractéristiques** : Les couches convolutives extraient des caractéristiques significatives des images.
- **Classification** : Une couche dense (avec 20 neurones) effectue la classification finale. Le modèle est initialement entraîné avec 10 classes. « model = create_base_model(num_classes=20) »

Entraînement Initial

Le modèle est entraîné sur les données initiales (MNIST, classes 0–9).

« train_initial_model(model, x_initial, y_initial) »

Ajout Progressif des Nouvelles Classes avec MTD

Le **MTD** modifie directement les poids de la couche de classification pour intégrer de nouvelles classes, sans réentraîner complètement le modèle.

Pour implémenter MTD :

$$w_j' = w_j + (\|w_j\| * \frac{h_i}{\|h_i\|} - w_j)\epsilon,$$

```
h = features_layer.predict(x)
h_norm = h / np.linalg.norm(h)
weights[:, y] += epsilon * (np.linalg.norm(weights[:, y]) * h_norm[0] - weights[:, y])
```

Étapes principales dans update_weights_mtd :

1. **Extraction des caractéristiques** :
    - Les représentations des nouvelles images (vecteurs de caractéristiques) sont extraites de la couche intermédiaire features.
2. **Normalisation** :
    - Les vecteurs de caractéristiques extraits sont normalisés pour s'assurer qu'ils ont une magnitude uniforme.
3. **Mise à jour des poids** :
    - Les poids correspondants à une nouvelle classe dans la couche de classification sont mis à jour proportionnellement aux vecteurs de caractéristiques extraits.

Évaluation et Mesure des Performances

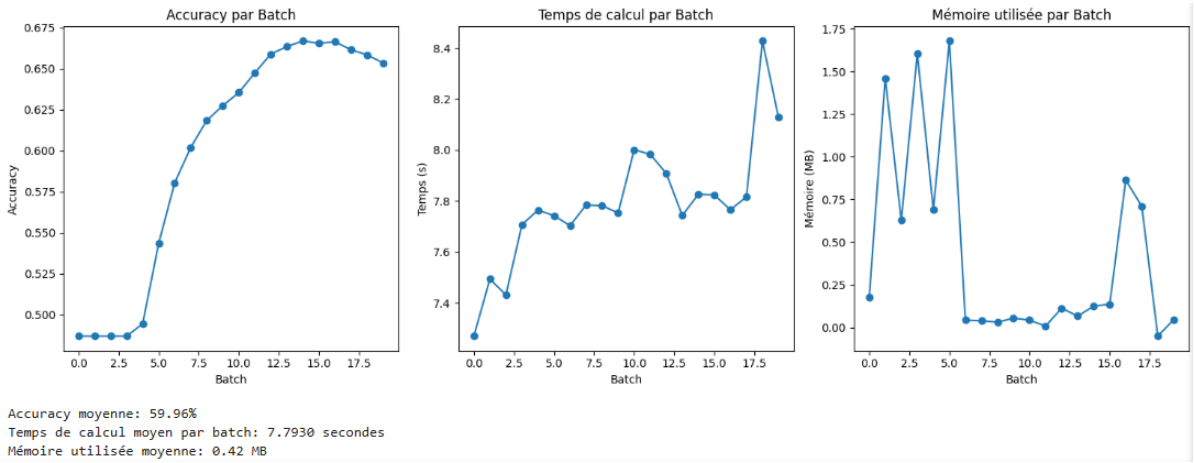
Pour chaque batch de nouvelles classes :

1. Temps d'exécution et mémoire utilisée sont mesurés avant et après l'appel à update_weights_mtd.
2. Le modèle est évalué sur les données de test pour vérifier son **accuracy**.

Visualisation des Résultats

- Les résultats des performances sont affichés sous forme de courbes :
    - Accuracy par batch.
    - Temps de calcul par batch.
    - Mémoire utilisée par batch.

3.2.2 Résultats MTD



- L'approche semble avoir une efficacité modérée en termes d'accuracy (59.96%) mais reste peu coûteuse en mémoire (0.42 MB).
- Le temps de calcul est raisonnable avec une moyenne de 7.7930 secondes par batch, bien que des optimisations pourraient être explorées pour améliorer les performances.

3.3 Adaptation aux Nouvelles Classes

3.3.1 Modifications Apportées

Notre objectif est de mettre en place un scénario d'apprentissage incrémental où le modèle doit apprendre progressivement de nouvelles classes qui n'étaient pas présentes dans l'entraînement initial. Le code implémente et compare trois approches différentes pour gérer ce défi, le processus expérimental sera donc :

1. Entraînement initial sur les 5 premières classes
2. Ajout séquentiel des trois batches de nouvelles classes
3. Évaluation continue des performances pour chaque méthode
4. Comparaison des résultats via des métriques et visualisations

Préparation des données (load_data_sequential_classes) :

1. Chargement et combinaison de deux datasets : MNIST (chiffres) et Fashion-MNIST (vêtements)
2. Division des données en 4 groupes :
    - Données initiales : 5 premières classes de MNIST (0-4)
    - Batch 1 : 5 classes suivantes de MNIST (5-9)
    - Batch 2 : 5 premières classes de Fashion-MNIST (10-14)
    - Batch 3 : 5 dernières classes de Fashion-MNIST (15-19)

Architecture du modèle (create_expandable_model) :

- CNN classique avec une couche de sortie de 20 neurones (pour gérer toutes les classes futures)
- La couche "features" sert d'extracteur de caractéristiques
- La couche finale utilise softmax pour la classification

Implémentation de trois approches différentes :

1. Brute Force (brute_force_new_classes) :
    - Réentraîne le modèle sur toutes les données (anciennes + nouvelles)
    - Simple mais coûteux en mémoire et temps
2. NFT - New Fine Tuning (nft_new_classes):
    - Entraîne uniquement sur les nouvelles classes
    - Plus rapide mais risque d'oublier les anciennes classes
3. MTD - Modified Target Distribution (mtd_new_classes):
    - Approche plus sophistiquée utilisant la couche de features
    - Ajuste les poids de manière itérative pour les nouvelles classes
    - Utilise un momentum pour stabiliser l'apprentissage
    - Epsilon adaptatif pour contrôler l'apprentissage

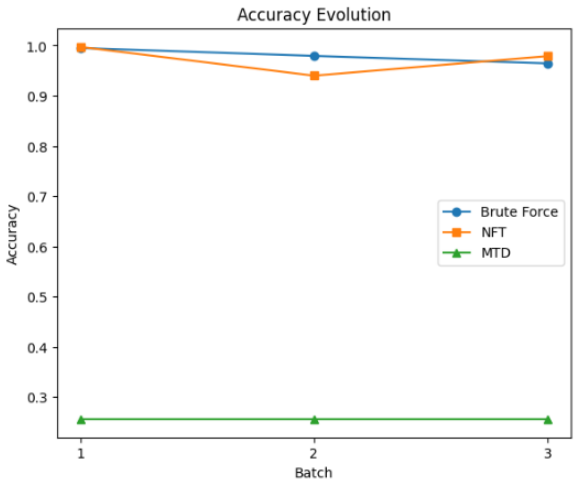
Évaluation et comparaison :

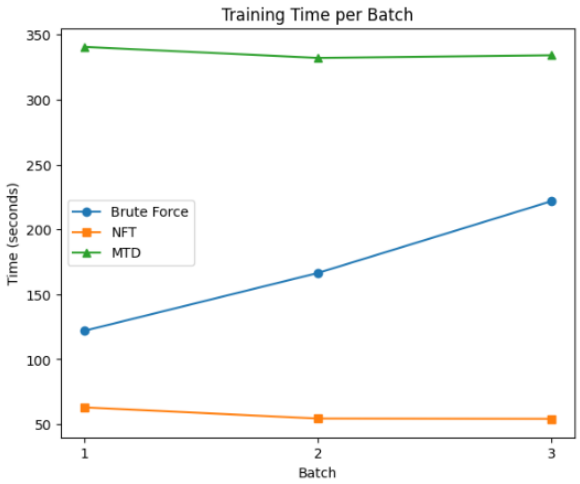
Mesure de plusieurs métriques :

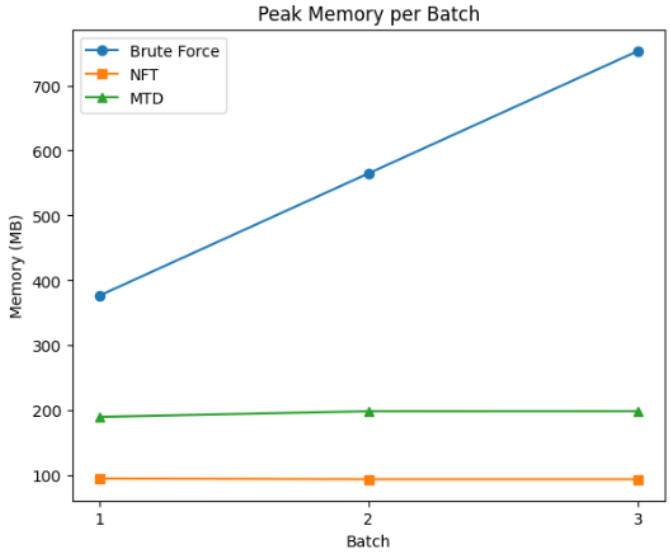
- Précision par classe
- Temps d'entraînement
- Utilisation mémoire
- Précision globale

3.3.2 Résultats et Analyse

```
Résultats finaux:

Brute Force:
Temps moyen: 170.00s
Mémoire moyenne: 564.78MB
Accuracy finale: 96.45%

NFT:
Temps moyen: 57.06s
Mémoire moyenne: 94.63MB
Accuracy finale: 97.90%

MTD:
Temps moyen: 335.65s
Mémoire moyenne: 195.99MB
Accuracy finale: 25.62%
```







Évolution de la précision :

- Brute Force (BF) et NFT montrent des précisions élevées (près de 96.45% et 97.90%, respectivement).
- La méthode MTD, en revanche, offre une précision nettement inférieure (25.62%), ce qui indique qu'elle n'est pas efficace pour cette tâche.

Interprétation : NFT surpasse légèrement BF en termes de précision, suggérant qu'elle exploite mieux les données ou les ressources pour apprendre des caractéristiques utiles. En revanche, MTD semble ne pas bien généraliser les données.

Temps d'entraînement par lot :

- NFT est nettement plus rapide, avec un temps moyen d'entraînement de 57.06 secondes par batch.
- BF prend plus de temps (170.00 secondes), et MTD est la méthode la plus lente (335.65 secondes).

Interprétation : NFT est non seulement plus précis mais aussi plus rapide, ce qui en fait une solution efficace pour ce problème. MTD, malgré son temps plus long, n'apporte pas d'avantage significatif en termes de précision.

Mémoire maximale utilisée :

- BF consomme le plus de mémoire, avec une moyenne de 564.78 MB.
- NFT est la méthode la plus économique en mémoire, à 94.63 MB.
- MTD utilise une mémoire intermédiaire de 195.99 MB.

Interprétation : NFT se distingue à nouveau par son efficacité, à la fois en termes de mémoire et de temps, ce qui la rend idéale pour des systèmes avec des ressources limitées.

4. Discussion

4.1 Synthèse des Résultats

Méthode	Précision	Temps de calcul (seconde)	Mémoire utilisée (MB)
Brute Force (BF)	93% - 96%	9 – 170	428 - 564
Native Fine-Tuning (NFT)	87% - 67%	1 - 57	30 – 94
Move-To-Data (MTD)	25% - 59%	7 – 335	0.42 – 195

Points forts et limites de chaque méthode :

Brute Force (BF) :

- **Points forts** : Précision élevée, robuste aux nouvelles classes, évite le problème du *catastrophic forgetting*.
- **Limites** : Temps d'entraînement très long, forte consommation mémoire, ce qui rend la méthode peu viable pour de grands ensembles de données ou des systèmes avec des ressources limitées.

Naive Fine-Tuning (NFT) :

- **Points forts** : Meilleure précision que BF, efficace en termes de temps et de mémoire, ce qui en fait une option attrayante pour des systèmes limités en ressources.
- **Limites** : Risque de *catastrophic forgetting* important, car la méthode ne tient compte que des nouvelles données, laissant de côté l'ancienne information.

Move-To-Data (MTD) :

- **Points forts** : Méthode innovante qui ajuste les poids de la couche de classification sans réentraîner entièrement le modèle, permettant de gérer les nouvelles classes de manière plus flexible.

- **Limites** : Mauvaise généralisation sur les nouvelles classes, faible précision, et performances globales moins compétitives en comparaison avec BF et NFT, malgré une consommation mémoire modérée.

4.2 Défis Rencontrés

Difficultés techniques :

- L'un des défis majeurs était la gestion de l'oubli catastrophique avec NFT, où le modèle ne parvenait pas à maintenir une bonne précision sur les anciennes classes lorsqu'il se concentrait sur l'ajout de nouvelles classes.
- L'implémentation de l'approche MTD a également posé des problèmes de convergence, notamment en raison des ajustements des poids proportionnels aux nouvelles classes, ce qui a perturbé l'apprentissage stable du modèle.

Limitations identifiées :

- La méthode MTD ne semble pas adaptée aux datasets avec des classes très hétérogènes et un grand nombre de classes.
- NFT a montré des signes évidents d'oubli catastrophique, ce qui pourrait être problématique dans des applications réelles où les anciennes classes sont cruciales.
- Les performances globales de MTD en termes de précision ne sont pas compétitives, ce qui peut en limiter l'applicabilité.

4.3 Améliorations Possibles

Suggestions d'optimisation :

- Pour BF, l'optimisation pourrait inclure l'usage de techniques de parallélisation ou de GPU pour accélérer le temps d'entraînement.
- NFT pourrait bénéficier de techniques avancées comme la régularisation basée sur la mémoire (Memory Replay) pour éviter l'oubli catastrophique.
- MTD pourrait être amélioré en affinant l'algorithme d'ajustement des poids pour augmenter la précision, notamment en introduisant des mécanismes de *momentum* et en ajustant les hyperparamètres de manière plus précise.

Pistes d'exploration futures :

- L'intégration de méthodes hybrides combinant BF et NFT pourrait permettre de conserver les avantages de chaque approche tout en atténuant leurs inconvénients.
- L'utilisation de modèles pré-entraînés sur des bases de données plus larges (par exemple, des réseaux pré-formés sur ImageNet) pourrait aider à mieux gérer les nouvelles classes tout en réduisant les risques de *catastrophic forgetting*.

- L'application de mécanismes d'apprentissage par transfert pourrait être explorée pour faciliter l'intégration de nouvelles classes sans sacrifier les performances sur les anciennes.

5. Conclusion

Résumé des principales découvertes : Ce projet a exploré trois approches d'apprentissage continu pour la classification d'images, chacune ayant ses avantages et ses limitations. Le Brute Force s'est avéré robuste mais coûteux en temps et en mémoire, tandis que le Naive Fine-Tuning a offert un bon compromis entre rapidité et précision, mais avec un risque de *catastrophic forgetting*. L'approche Move-To-Data, bien qu'intéressante, n'a pas montré des résultats convaincants en termes de performance.

Recommandations pratiques :

- Pour des systèmes à ressources limitées, NFT peut être une approche intéressante, à condition de trouver des solutions au problème de l'oubli catastrophique.
- Le Brute Force reste une bonne option si les ressources et le temps d'entraînement ne sont pas des contraintes majeures.
- MTD, bien que prometteuse, nécessite des améliorations importantes avant d'être viable pour des applications pratiques.

Perspectives : L'exploration de méthodes hybrides et de mécanismes de mémorisation avancés pour NFT et MTD pourrait ouvrir de nouvelles perspectives dans le domaine de l'apprentissage continu, permettant de mieux gérer l'intégration de nouvelles classes tout en conservant les performances sur les anciennes données.

6. Références

[1] Poursanidis, M., et al. (2020). "Move-to-Data: A new Continual Learning approach with Deep CNNs, Application for image-class recognition."

Annexes

A. Code Source

- Liens vers le repository: kadirimeriem/Continuous_Learning_Project: This project demonstrates continuous learning using advanced neural network architectures.