Projet de Deep Learning

Master 2 Data science : Santé, assurance, finance

2024 - 2025

Rédigé à
L'Université de Paris-Saclay

par
ATHOUMANI Ibroihima et AH-MOUCK Laetitia

Apprentissage semi-supervisé par prédiction des rotations d'image
Application sur le dataset MNIST

Table des matières

1 Introduction

L'apprentissage profond (Deep Learning) est devenu une méthode incontournable pour résoudre des problèmes complexes dans divers domaines tels que la vision par ordinateur, le traitement du langage naturel et bien d'autres. Parmi les techniques les plus prometteuses, l'apprentissage par transfert (Transfer Learning) se distingue par sa capacité à réutiliser les connaissances acquises sur une tâche pour en résoudre une autre, réduisant ainsi le besoin en données et en ressources de calcul.

Dans ce projet, nous souhaitons utiliser les réseaux de neurones afin de classifier correctement les différentes images. Nous utiliserons notamment les réseaux de neurones convolutifs, dont une méthode semi-supervisée basée sur l'article de Gidaris et al.[1] qui utilise l'apprentissage par transfert.

Les résultats obtenus permettront d'évaluer les avantages et les limites de cette méthode dans le cadre de tâches de classification d'images.

Ce projet vise à démontrer que l'apprentissage par transfert permet non seulement de réduire les besoins en données annotées, mais aussi d'améliorer les performances globales du modèle grâce à une meilleure initialisation des paramètres.

1.1 Présentation de la base de données MNIST

La base de données MNIST (Modified ou Mixed National Institute of Standards and Technology) est composée d'images de chiffres écrits à la main. Les images sont en niveau de gris, normalisées centrées de 28 pixels de côté. Il y a 60 000 images d'apprentissage et 10 000 images de test.

FIGURE 1 – Aperçu de la base de données MNIST

1.2 Objectif

L'objectif est d'implémenter des méthodes de deep learning pour classer les images et d'évaluer les résultats de la base de donnée MNIST. Cependant, nous avons pour contrainte d'utiliser uniquement 100 images labélisées au lieu des 60 000 disponibles. C'est ici qu'intervient le choix d'un algorithme semi-supervisé pour pallier au manque de données.

2 Description de la méthode

Dans ce projet, la méthode utilisée est une méthode semi-supervisée. Cette approche nous permet de contourner le problème lié au manque de données étiquetées. Dans notre cas d'application, l'idée est de faire des rotations sur les images et d'entraîner le réseau, que l'on nomme Rotnet, à prédire l'angle de rotation de l'image.

Ainsi, nous obtenons quatre classes :

— 0 : image originale,
— 1 : image tournée à 90°,
— 2 : image tournée à 180°,
— 3 : image tournée à 270°.

Après avoir entraîné ce modèle, nous appliquons l'apprentissage par transfert en fixant les poids du modèle déjà entraîné pour prédire les rotations des images, sauf pour la dernière couche. Cette dernière sera réinitialisée, puis entraînée et évaluée sur les 100 images étiquetées avec une sortie de 10 classes. Cette technique repose sur l'hypothèse que si le modèle est entraîné à prédire les rotations des images, il apprend également la sémantique de ces images.

Nous terminerons par une comparaison entre ce modèle Rotnet et un modèle de convolution entraîné sur la même quantité d'images (100 images étiquetées) afin d'évaluer l'efficacité de cette méthode dans un cas similaire où les données étiquetées sont limitées.

3 Traitement des données

Nous avons pris 100 images labélisées au hasard en faisant bien attention à garder les mêmes proportions que la base de données initiale pour chaque label afin de ne pas avoir de résultats biaisés. Nous avons ainsi sélectionné 10 images de chaque chiffre pour le set d'entraînement.

3.1 Qualité des images

Nous avons modifié les images de base afin d'améliorer la qualité des images. En effet, les images étant en niveau de gris, nous les avons normalisées en utilisant la moyenne et l'écart-type du set d'entraînement de base pour avoir uniquement des pixels en noir et blanc.

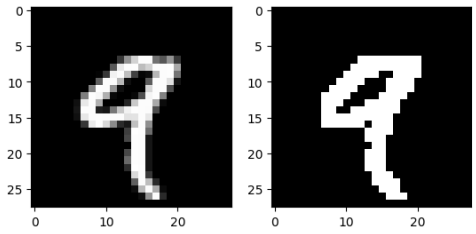


FIGURE 2 – A gauche l'image originale, à droite l'image améliorée

3.2 Rotation des images

Pour appliquer la méthode RotNet, nous avons besoin des images pivotées à 90°, 180° et 270°, issues de l'image originale.

Voici un exemple des images obtenues par rotation :

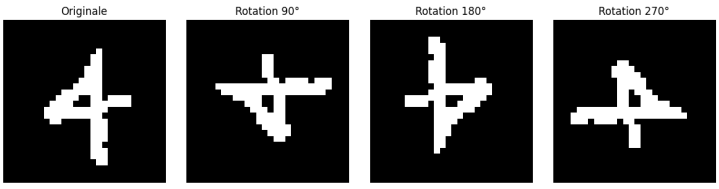


FIGURE 3 – Image originale et ses 3 rotations

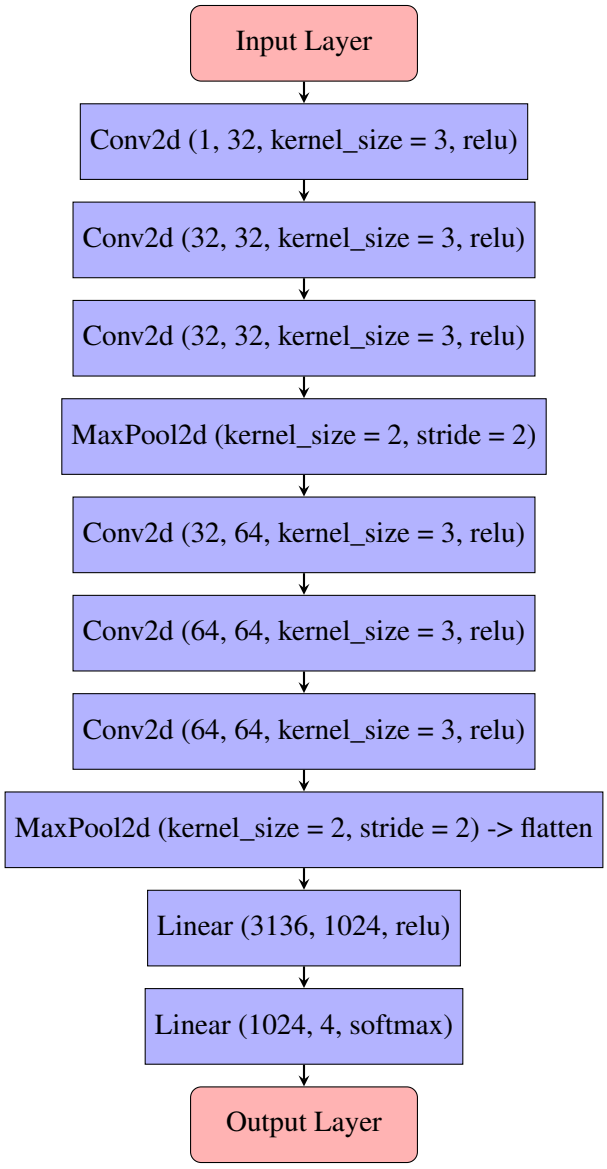
4 Modélisation

Dans cette partie, nous allons expliciter l'architecture des modèles Rotnet et CNN, ainsi que les résultats obtenus lors de l'entraînement et de l'évaluation de ces derniers.

4.1 Modèle Rotnet

Rotnet est un réseau de neurones convolutifs. Il est composé de deux blocs de convolution, chacun contenant des couches de convolution et une couche de max pooling, permettant de réduire les dimensions tout en conservant les caractéristiques essentielles des données.

4.1.1 Architecture du modèle



Notons que ce modèle comporte uniquement 4 classes pour la sortie, comme décrit dans la partie consacrée à la description. Cela correspond à la première étape de l'entraînement : nous allons l'entraîner à prédire l'angle de rotation. Dans un second temps, nous fixerons les poids associés aux blocs

de convolution et réentraînerons le reste du modèle sur les 100 images annotées pour reconnaître les chiffres, ce qui impliquera 10 classes en sortie.

4.1.2 Courbe d'apprentissage

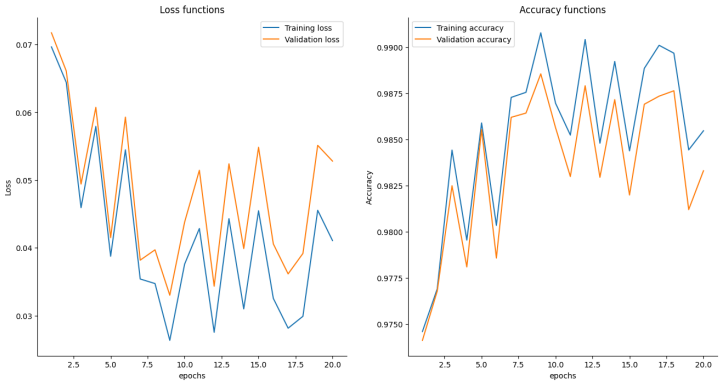


FIGURE 4 – Courbe d'apprentissage pour la prédiction des rotations

Le modèle semble bien apprendre, comme en témoigne la diminution de la loss et l'augmentation de l'accuracy. Cependant, les oscillations dans les courbes de validation (loss et accuracy) suggèrent une certaine instabilité.

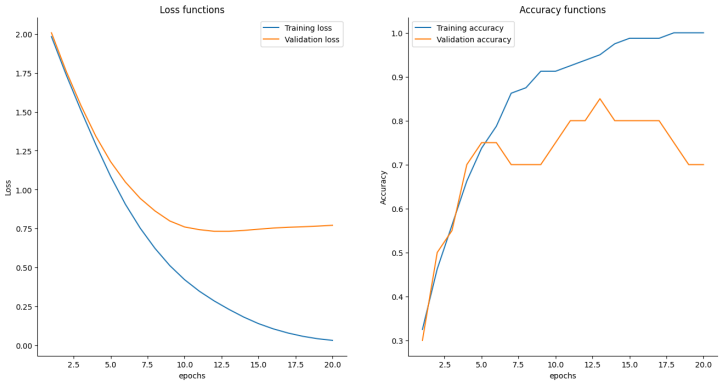


FIGURE 5 – Courbe d'apprentissage pour la prédiction des classes originales

Le modèle montre une bonne capacité d'apprentissage initiale, mais il commence à sur-ajuster les données d'entraînement, ce qui limite ses performances sur les données de validation. Des ajustements supplémentaires, comme l'utilisation de régularisation ou l'augmentation des données, pourraient améliorer ses capacités de généralisation.

4.1.3 Évaluation du modèle

Classification report :

	precision	recall	f1-score	support
Class 0	0.97	0.99	0.98	10000
Class 1	0.98	0.99	0.98	10000
Class 2	0.99	0.97	0.98	10000
Class 3	0.99	0.98	0.98	10000
accuracy			0.98	40000
macro avg	0.98	0.98	0.98	40000
weighted avg	0.98	0.98	0.98	40000

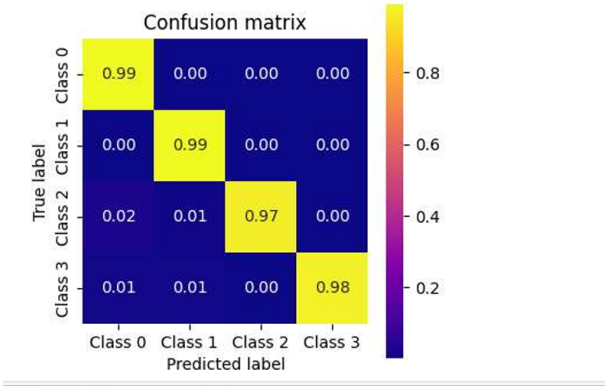


FIGURE 6 – Évaluation du modèle Rotnet sur la prédiction des rotations des images

Le modèle montre une performance très élevée (98% de précision, rappel et F1-score). Les erreurs sont rares et bien réparties. Cependant, la classe 2 semble légèrement plus difficile à prédire, avec un rappel légèrement inférieur (0.97). On remarque que le modèle a bien appris à prédire les rotations des images avec une accuracy de 98%. Il reste à observer la seconde phase sur la prédiction des classes originales avec les 100 images labellisées.

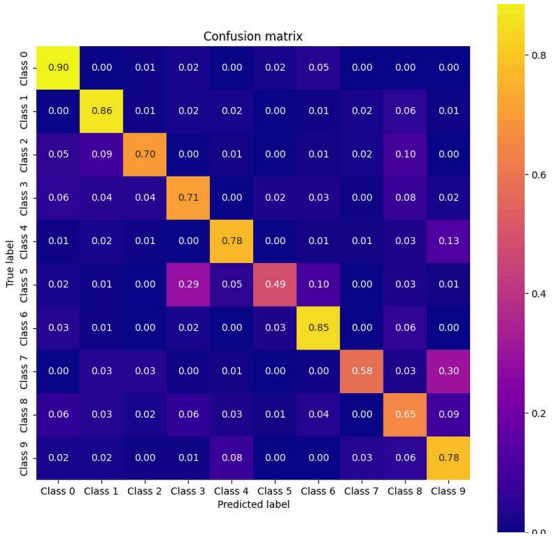


FIGURE 7 – Matrice de confusion pour la prédiction des classes originales

Le modèle montre un bon potentiel global. Il est performant pour certaines classes mais a des faiblesses significatives pour d'autres, notamment pour la Classe 5 où les erreurs sont dominantes.

Les valeurs diagonales montrent les prédictions correctes pour chaque classe. Par exemple, pour la

Classe 0, le modèle a correctement prédit 90% des échantillons (valeur de 0.90 sur la diagonale). Une valeur élevée sur la diagonale indique que le modèle est performant pour cette classe.

Classification report :

	precision	recall	f1-score	support
Class 0	0.78	0.90	0.84	980
Class 1	0.79	0.86	0.82	1135
Class 2	0.86	0.70	0.77	1032
Class 3	0.65	0.71	0.68	1010
Class 4	0.78	0.78	0.78	982
Class 5	0.83	0.49	0.62	892
Class 6	0.77	0.85	0.81	958
Class 7	0.86	0.58	0.69	1028
Class 8	0.59	0.65	0.62	974
Class 9	0.58	0.78	0.66	1009
accuracy			0.73	10000
macro avg	0.75	0.73	0.73	10000
weighted avg	0.75	0.73	0.73	10000

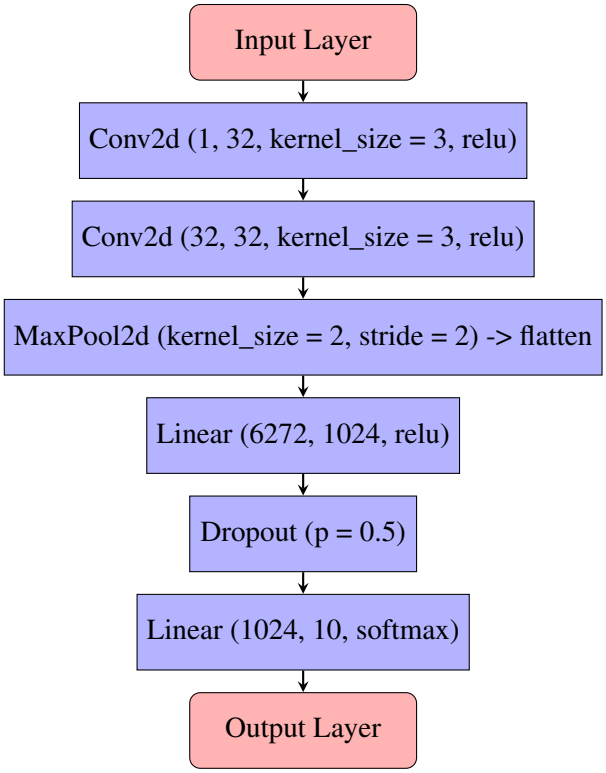
FIGURE 8 – Classification des classes originales

L'accuracy du modèle est de 73%, ce qui indique une performance moyenne. Les scores moyens de précision, rappel et F1-score sont de 0.75, 0.73, et 0.73, respectivement. Cela montre que le modèle ne favorise pas de manière significative une classe particulière mais reste limité dans sa capacité globale à bien classer toutes les classes.

4.2 Modèle CNN

Nous construisons un CNN moins profond car nous ne disposons que de 100 images pour entraîner ce modèle. Il serait possible de concevoir un modèle plus profond, comme RotNet, mais étant donné la limitation à 100 images, il est préférable d'opter pour un CNN moins profond.

4.2.1 Architecture du modèle



Ce modèle contient un bloc de convolution comprenant deux couches de convolution, suivi d'une couche de pooling pour réduire les dimensions tout en conservant les caractéristiques essentielles des données. Il intègre également deux couches entièrement connectées et une couche de dropout pour contrôler le sur-apprentissage. Enfin, la sortie comporte 10 unités correspondant aux 10 classes.

4.2.2 Courbe d'apprentissage

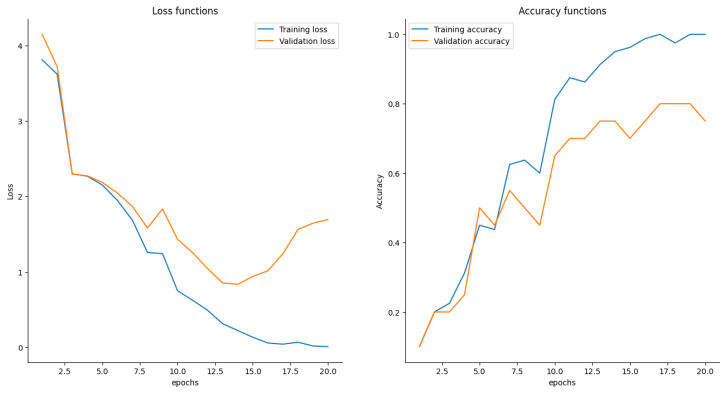


FIGURE 9 – Courbe d'apprentissage du CNN pour la prédiction des classes originales

La fonction de perte de l'entraînement (ligne bleue) diminue régulièrement, ce qui indique que le modèle apprend bien sur les données d'entraînement. La fonction de perte de la validation (ligne orange) diminue initialement, mais commence à augmenter après environ 10 époques c'est peut-être à cause du nombre limité de données.

4.2.3 Évaluation du modèle

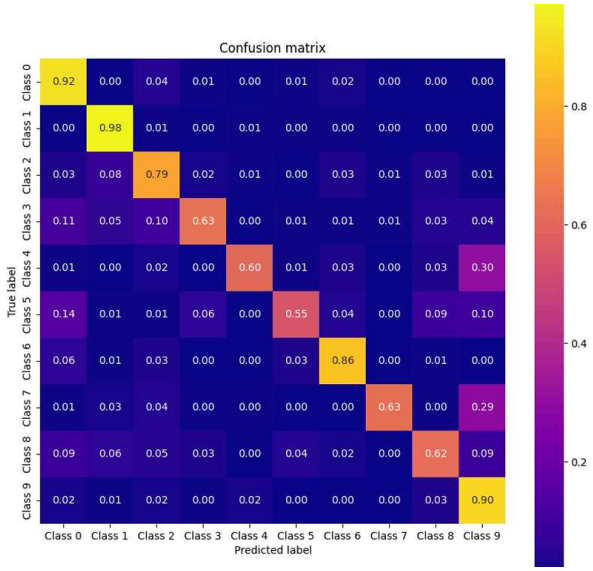


FIGURE 10 – Matrice de confusion

Certaines confusions semblent récurrentes, comme celles entre les classes 4 et 5 ou les classes 5 et 6. Cela pourrait indiquer que les caractéristiques discriminantes entre ces classes ne sont pas suffisamment bien apprises par le modèle.

```
Classification report :
              precision    recall  f1-score   support

     Class 0       0.67      0.92      0.77       980
     Class 1       0.82      0.98      0.89      1135
     Class 2       0.73      0.79      0.76      1032
     Class 3       0.84      0.63      0.72      1010
     Class 4       0.93      0.60      0.73       982
     Class 5       0.83      0.55      0.66       892
     Class 6       0.84      0.86      0.85       958
     Class 7       0.97      0.63      0.76      1028
     Class 8       0.73      0.62      0.67       974
     Class 9       0.52      0.90      0.66      1009

    accuracy                           0.75     10000
   macro avg       0.79      0.75      0.75     10000
weighted avg       0.79      0.75      0.75     10000
```

FIGURE 11 – Classification des classes originales

La précision mesure le pourcentage de prédictions correctes parmi toutes les prédictions faites pour une classe. Les classes avec une précision élevée incluent la classe 4 (93%) et la classe 1 (82%). La classe 9 a une précision faible (52%), ce qui indique un taux élevé de faux positifs pour cette classe.

Le rappel mesure le pourcentage d'instances d'une classe correctement identifiées parmi toutes les instances réelles de cette classe. Les classes avec un rappel élevé incluent la classe 1 (98%) et la classe 0 (92%). La classe 5 a un rappel particulièrement faible (55%), ce qui montre qu'une grande partie des instances de cette classe n'est pas correctement détectée.

5 Conclusion

Rotnet offre une solution intéressante dans les cas où les données labélisées sont rares. En utilisant l'apprentissage semi-supervisé pour apprendre les invariances de rotation, Rotnet peut fournir une base utile pour la classification, même si les performances restent légèrement inférieures à un modèle CNN classique. CNN, en revanche, nécessite des données labélisées en grande quantité pour atteindre une performance optimale, mais démontre une meilleure capacité à généraliser.

Annexe A

Code Python[2]

1 Boite noire

Le code boite noire contient toutes les fonctions qui nous seront utiles pour charger, traiter les données, et évaluer nos résultats.

```
import pandas as pd
import numpy as np
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchvision import datasets
from tqdm import tqdm
from torchvision.transforms.functional import rotate
from torch.utils.data import Dataset, DataLoader
from torch.utils.data import TensorDataset, Subset, ConcatDataset
from sklearn.metrics import classification_report, confusion_matrix,
    ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import os
import warnings
warnings.filterwarnings("ignore")
device = "cuda:0" if torch.cuda.is_available() else "cpu"


# Evaluation du modèle
def get_accuracy(y_true, y_pred) :
  """
  y_true: valeurs des vrais labels
  y_pred: valeurs des labels prédits
  Retourne l'accuracy du modèle
  """

  return int(np.sum(np.equal(y_true, y_pred))) / y_true.shape[0]


# Architecture et mémoire
def get_model_memory(model) :
  """Retourne l'allocation de mémoire, le nombre de paramètres et
```

```
  l'achitecture du modèle"""

  # Architecture du modèle
  print(model)
  print("Model memory allocation : {:.2e}".format(torch.cuda.memory_reserved(0)
  - torch.cuda.memory_allocated(0)))

  # Paramètres
  total_params = sum(p.numel() for p in model.parameters())
  print(f"\nNb total de paramètres : {total_params}")
  total_trainable_params = sum(p.numel() for p in model.parameters() if p.
    requires_grad)
  print(f"Nb total de paramètres d'entrainement : {total_trainable_params}")

# Importation MNIST
def load_mnist(cleared = False) :
    """Importer les données MNIST
    Si cleared = True, alors l'image est en noir et blanc"""

    if cleared :
        transformer = transforms.Compose([transforms.ToTensor(),
                                          transforms.Normalize((0.1307, ),
    (0.3081, )),
                                          lambda x: x > 0,
                                          lambda x: x.float(), ])
    else :
        transformer = transforms.Compose([transforms.ToTensor(),
                                          transforms.Normalize((0.5, ), (0.5, )
    )])

    # Importation des données MNIST
    train_data = datasets.MNIST(root = './data', train = True, download = True,
     transform = transformer) # Train set
    test_data = datasets.MNIST(root = './data', train = False, download = True,
     transform = transformer) # Test set

    return train_data, test_data


# Sélection des 100 données labélisées
def reduce_mnist(data, nb_label = 10) :
    """Retourne un sous-ensemble aléatoire de 10 images par label"""

    reduced_data = []

    for i in range(10) :
        label_indices = torch.where(data.targets == i)[0]
        subset_indices = random.sample(label_indices.tolist(), nb_label)
        reduced_data.append(Subset(data, subset_indices))

    return ConcatDataset(reduced_data)


# Rotation des images à 90  , 180   et 270
def rotate_mnist(data) :
    """Ajout de nouvelles images par rotation de 90  , 180   et 270   de
    l'image originales"""

    rotated_data = []
    rotated_labels = []
```

```
94
95      for i in range(len(data)) :
96          img, _ = data[i]
97
98          rotated_data.append(img)
99          rotated_labels.append(0)
100
101         # 3 nouvelles images par rotation
102         for angle in [90, 180, 270] :
103             img_rotated = rotate(img, angle)
104             rotated_data.append(img_rotated)
105
106         # Mise à jour de la liste des lables pour la rotation
107         # 1:rotation à 90  , 2: rotation à 180  , 3: rotation à  270
108         rotated_labels.extend([1, 2, 3])
109
110     # Conversion en tenseur
111     rotated_data = torch.stack(rotated_data)
112     rotated_labels = torch.tensor(rotated_labels)
113
114     return TensorDataset(rotated_data, rotated_labels)
115
116 def load_data(data, batch_size, shuffle = False, num_workers = 2) :
117     """Charge le dataset dans un dataloader"""
118
119     return DataLoader(data, batch_size = batch_size, shuffle = shuffle,
        num_workers = num_workers)
120
121
122 def train_model(train_loader, val_loader, model = None, output_fn = None,
123                 epochs:int = None, optimizer = None, criterion = None,
124                 device = None) :
125     """Entraine le modèle pytorch et calcule l'accuracy et la loss pour chaque
        epoch"""
126
127     loss_valid, acc_valid = [], []
128     loss_train, acc_train = [], []
129
130     for epoch in tqdm(range(epochs)) :
131
132         # Entrainement
133         model.train()
134         running_loss = 0.0
135         for _, batch in enumerate(train_loader) :
136
137             # Entrainement sur le GPU
138             inputs, labels = batch
139             inputs = inputs.to(device)
140             labels = labels.to(device)
141
142             # Initialisation du gradient à 0
143             optimizer.zero_grad()
144
145             # Forward, backward et optimizer
146             out = model(x = inputs)
147             loss = criterion(out, labels)
148             loss.backward()
149             optimizer.step()
150
```

```
151         # Calcul de la  loss et de l'accuracy sur l'ensemble de validation aprè
    s chaque epoch
152         model.eval()
153         with torch.no_grad() : # Pas besoin de calculer le gradient pour l'é
    valuation
154             idx = 0
155 
156             for batch in val_loader :
157                 inputs, labels = batch
158                 inputs = inputs.to(device)
159                 labels = labels.to(device)
160 
161                 if idx==0 :
162                     t_out = model(x = inputs)
163                     t_loss = criterion(t_out, labels).view(1).item()
164                     t_out = output_fn(t_out).detach().cpu().numpy()
165                     t_out = t_out.argmax(axis = 1)
166                     ground_truth = labels.detach().cpu().numpy()
167 
168                 else :
169                     out = model(x = inputs)
170                     t_loss = np.hstack((t_loss, criterion(out, labels).item()))
171                     t_out = np.hstack((t_out, output_fn(out).argmax(axis = 1).
    detach().cpu().numpy()))
172                     ground_truth = np.hstack((ground_truth, labels.detach().cpu
    ().numpy()))
173                 idx += 1
174 
175             acc_valid.append(get_accuracy(ground_truth, t_out))
176             loss_valid.append(np.mean(t_loss))
177 
178         # Calcul de la loss et de l'accuracy sur le training set après chaque
    epoch
179         with torch.no_grad() :
180             idx = 0
181 
182             for batch in train_loader :
183                 inputs, labels = batch
184                 inputs = inputs.to(device)
185                 labels = labels.to(device)
186 
187                 if idx==0 :
188                     t_out = model(x = inputs)
189                     t_loss = criterion(t_out, labels).view(1).item()
190                     t_out = output_fn(t_out).detach().cpu().numpy()
191                     t_out = t_out.argmax(axis = 1)
192                     ground_truth = labels.detach().cpu().numpy()
193 
194                 else :
195                     out = model(x = inputs)
196                     t_loss = np.hstack((t_loss, criterion(out, labels).item()))
197                     t_out = np.hstack((t_out, output_fn(out).argmax(axis = 1).
    detach().cpu().numpy()))
198                     ground_truth = np.hstack((ground_truth, labels.detach().cpu
    ().numpy()))
199                 idx += 1
200 
201         acc_train.append(get_accuracy(ground_truth, t_out))
202         loss_train.append(np.mean(t_loss))
203 
```

```
204         print("| Epoch: {}/{} | Train: Loss {:.4f} Accuracy : {:.4f} "\
205         "| Val: Loss {:.4f} Accuracy : {:.4f}\n".format(epoch + 1, epochs,
    loss_train[epoch], acc_train[epoch], loss_valid[epoch], acc_valid[epoch]))
206
207     return model, (loss_train, acc_train, loss_valid, acc_valid)
208
209
210 def split_train_valid(data, valid_size, random_state = None, is_image = True) :
211     """Séparation du jeu de données en set d'entrainemente et de validation"""
212
213     if is_image :
214
215         labels = []
216         for i in range(len(data)) :
217             _, label = data[i]
218             labels.append(label)
219
220         # Séparation des indices pour l'entrainement et la validation si on a
    une image
221         train_indices, val_indices = train_test_split(list(range(len(labels))),
     test_size = valid_size, stratify = labels, random_state = random_state)
222
223     else :
224         train_indices, val_indices = train_test_split(list(range(len(data
    [:][1]))), test_size = valid_size, stratify = data[:][1], random_state =
    random_state)
225
226     return Subset(data, train_indices), Subset(data, val_indices)
227
228
229 def plot_accuracy(epochs, loss_train, loss_valid, acc_train, acc_valid) :
230     """Trace la courbe de l'accuracy et de la loss"""
231
232     fig = plt.figure(figsize = (16, 8))
233
234     def plot_metric(epochs, metric_train, metric_valid, metric_name) :
235         """Trace les métriques pour le set d'entrainement et de validation"""
236
237         plt.plot(range(1, epochs + 1), metric_train, label = f"Training {
    metric_name.lower()}")
238         plt.plot(range(1, epochs + 1), metric_valid, label = f"Validation {
    metric_name.lower()}")
239         plt.xlabel("epochs")
240         plt.ylabel(metric_name)
241         plt.title(f"{metric_name} functions")
242         plt.legend()
243
244     # Tracer la courbe de la loss
245     ax = fig.add_subplot(121)
246     for side in ["right", "top"]:
247         ax.spines[side].set_visible(False)
248     plot_metric(epochs, loss_train, loss_valid, "Loss")
249
250     # Tracer la courbe de l'accuracy
251     ax = fig.add_subplot(122)
252     for side in ["right", "top"]:
253         ax.spines[side].set_visible(False)
254     plot_metric(epochs, acc_train, acc_valid, "Accuracy")
255
256
```

```
def evaluate_model(model, test_loader, device, num_classes = 10) :
    """Evaluation du modèle sur le set de test"""

    y_true, y_pred = [], []
    with torch.no_grad() :

        for inputs, labels in test_loader :
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

    # Affiche le rapport de classification
    target_names = [f"Class {str(i)}" for i in range(num_classes)]
    print("Classification report :")
    print(classification_report(y_true, y_pred, target_names = target_names))

    # Affiche la matrice de confusion
    cm = confusion_matrix(y_true, y_pred)
    cm = cm.astype("float") / cm.sum(axis = 1)[:, np.newaxis]
    plt.subplots(figsize = (num_classes,num_classes))
    sns.heatmap(cm, annot = True, fmt = ".2f", cmap = "plasma", square = True,
                xticklabels = target_names, yticklabels = target_names)
    plt.ylabel("True label")
    plt.xlabel("Predicted label")
    plt.title("Confusion matrix")
    plt.show()


def save_model(model, path) :
    """sauvegarde le modèle dans un fichier.pth"""

    return torch.save(model.state_dict(), path)
```

2 Main code

```
from boitenoire import*

device = "cuda:0" if torch.cuda.is_available() else "cpu" # GPU use

#Importation & preprocessing
# MNIST Importation
train_data, test_data = load_mnist()

### Cleared MNIST Importation
traindata_cleared, testdata_cleared = load_mnist(True)

# Différence de qualité pour une image donnée
plt.subplot(1, 2, 1)
plt.imshow(train_data[4][0].view(28, 28), cmap = "gray")
plt.title("A gauche l'image originale et à droite l'image améliorée :", loc = "
    left")
plt.subplot(1, 2, 2)
plt.imshow(traindata_cleared[4][0].view(28, 28), cmap = "gray")
plt.show()

```

```
# Réduction de la taille du jeu de données
traindata_reduced = reduce_mnist(traindata_cleared)

# Affichage du jeu de données réduit

fig, ax = plt.subplots(nrows = 10, ncols = 10, figsize = (10, 10))

for i in range(10) :
    for j in range(10) :
        img, label = traindata_reduced[i*10 + j]
        ax[i, j].imshow(img.squeeze(), cmap = 'gray')
        ax[i, j].axis('off')
        ax[i, j].set_title('Label: {}'.format(label))

plt.tight_layout()
plt.show()

# Afficher l'image originale et ses rotations pour une image donnée
rotated_dataset = rotate_mnist(traindata_reduced)
for i in range(0,100,10):
  display_rotated_images(rotated_dataset, index=i)

class Rotnet(nn.Module) :
    """Rotnet for digit recognition task"""

    def __init__(self) :
        super().__init__()

        # Premier bloc de convolution
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size = 3, padding = "same")
        self.conv2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, padding = "same")
        self.conv3 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3, padding = "same")
        self.pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        # Second bloc de convolution
        self.conv4 = nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 3, padding = "same")
        self.conv5 = nn.Conv2d(in_channels = 64, out_channels = 64, kernel_size = 3, padding = "same")
        self.conv6 = nn.Conv2d(in_channels = 64, out_channels = 64, kernel_size = 3, padding = "same")
        self.pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        # Couches entièrement connectées
        self.fc1 = nn.Linear(in_features = 3136, out_features = 1024)
        self.fc2 = nn.Linear(in_features = 1024, out_features = 4)

    def forward(self, x) :

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool1(F.relu(self.conv3(x)))
        x = F.relu(self.conv4(x))
        x = F.relu(self.conv5(x))
        x = self.pool2(F.relu(self.conv6(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
```

```
74          x = self.fc2(x)
75
76          return x
77
78  class CNN(nn.Module) :
79      """CNN for classification purpose"""
80
81      def __init__(self) :
82          super().__init__()
83
84          self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size
        = 3, padding = "same")
85          self.conv2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size
         = 3, padding = "same")
86          self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
87
88          self.fc1 = nn.Linear(in_features = 6272, out_features = 1024)
89          self.dropout = nn.Dropout(p = 0.5)
90          self.fc2 = nn.Linear(in_features = 1024, out_features = 10)
91
92      def forward(self, x) :
93
94          x = F.relu(self.conv1(x))
95          x = self.pool(F.relu(self.conv2(x)))
96          x = torch.flatten(x, 1)
97          x = F.relu(self.fc1(x))
98          x = self.dropout(x)
99          x = self.fc2(x)
100
101         return x
102
103 get_model_memory(CNN())
104
105 get_model_memory(Rotnet())
106
107 # MNIST dataset
108 traindata, testdata = load_mnist(True)
109
110 ### Dataset and préparation du modèle
111 # Jeux de données avec rotation des images
112 traindata_rotated = rotate_mnist(traindata)
113 testdata_rotated = rotate_mnist(testdata)
114
115 # Séparation en set d'entrainement et de tes
116 traindata_rotated, validata_rotated = split_train_valid(traindata_rotated, 0.2,
         random_state=None) #80% pour l'entrainement et 20% pour la validation
117
118 # Taille des jeux de données
119 print("Size of the train dataset :", len(traindata_rotated), ",",
        traindata_rotated[0][0].shape,
120       "\nSize of the validation dataset :", len(validata_rotated), ",",
        validata_rotated[0][0].shape,
121       "\nSize of the test dataset :", len(testdata_rotated), ",",
        testdata_rotated[0][0].shape, "\n")
122
123 # Chargement des données dans un dataloader
124 trainloader = load_data(traindata_rotated, 256, True, 0)
125 validloader = load_data(validata_rotated, 256, False, 0)
126 testloader = load_data(testdata_rotated, 256, False, 0)
127
```

```
### Entrainement du modèle
rotnet = Rotnet().to(device)

# Définition des paramètres
output_fn = nn.Softmax(dim = 1)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(rotnet.parameters(), lr = 0.01)

# Entrainement du ROTNET
epochs = 20
args_train = {"train_loader" : trainloader,
              "val_loader" : validloader,
              "model" : rotnet,
              "output_fn" : output_fn,
              "epochs" : epochs,
              "optimizer" : optimizer,
              "criterion" : criterion,
              "device" : device}

rotnet, (loss_train, acc_train, loss_valid, acc_valid) = train_model(**
    args_train)

plot_accuracy(epochs, loss_train, loss_valid, acc_train, acc_valid)

evaluate_model(rotnet, testloader, device, 4)

import os
os.makedirs("save_models", exist_ok=True) # Créé la direction si elle n'existe
    pas

# Sauvegarde du modèle
save_model(rotnet, "save_models/rotnet.pth")

### Fine tuning du modèle pré entrainé
rotnet = Rotnet().to(device)
rotnet.load_state_dict(torch.load("save_models/rotnet.pth", map_location =
    device))

# Paramètres du modèle
params = rotnet.state_dict()
params.keys()

# Figer les paramètres sauf ceux de la dernière couche

for name, param in rotnet.named_parameters() :
    if param.requires_grad and 'conv'  in name :
        param.requires_grad = False

# Vérification des paramètres figés

for name, param in rotnet.named_parameters() :
    print(name, param.requires_grad)

### Mise à jour des dimensions de la dernière couche

# Récupérer le nombre d'entités en entrée dans la dernière couche entièrement
    connectée
number_features_last_layer = rotnet.fc2.in_features

# Réinitialiser la dernière couche avec le nb correct de sorties
```

```
rotnet.fc2 = nn.Linear(number_features_last_layer, 10)

# Récapitulatif du modèle
rotnet = rotnet.to(device)
get_model_memory(rotnet)

### Dataset et préparation du modèle

# Réduction du nombre d'images
traindata_reduced = reduce_mnist(traindata)
traindata_reduced, validata_reduced = split_train_valid(traindata_reduced, 0.2,
    False)

# Taille des données
print("Size of the train dataset :", len(traindata_reduced), ",",
    traindata_reduced[0][0].shape,
      "\nSize of the validation dataset :", len(validata_reduced), ",",
    validata_reduced[0][0].shape,
      "\nSize of the test dataset :", len(testdata), ",", testdata[0][0].shape,
     "\n")

# Chargement des données
trainloader = load_data(traindata_reduced, 80, True, 0)
validloader = load_data(validata_reduced, 20, False, 0)
testloader = load_data(testdata, 128, False, 0)

### Entrainement du modèle

# Paramètres
output_fn = nn.Softmax(dim = 1)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad, rotnet.parameters()),
    lr = 0.01)

# Entrainement
epochs = 20
args_train = {"train_loader" : trainloader,
              "val_loader" : validloader,
              "model" : rotnet,
              "output_fn" : output_fn,
              "epochs" : epochs,
              "optimizer" : optimizer,
              "criterion" : criterion,
              "device" : device}

rotnet, (loss_train, acc_train, loss_valid, acc_valid) = train_model(**
    args_train)

plot_accuracy(epochs, loss_train, loss_valid, acc_train, acc_valid)

evaluate_model(rotnet, testloader, device, 10)

### Entrainement du modèle CNN
cnn= CNN().to(device)

# Paramètres
output_fn = nn.Softmax(dim = 1)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)

```

```
# Entrainement pour le CNN
epochs = 20
args_train = {"train_loader" : trainloader,
              "val_loader" : validloader,
              "model" : cnn,
              "output_fn" : output_fn,
              "epochs" : epochs,
              "optimizer" : optimizer,
              "criterion" : criterion,
              "device" : device}

cnn, (loss_train, acc_train, loss_valid, acc_valid) = train_model(**args_train)

plot_accuracy(epochs, loss_train, loss_valid, acc_train, acc_valid)

evaluate_model(cnn, testloader, device, 10)
```

Bibliographie

[1] S.Gidaris, P.Singh, and N.Komodakis, "Unsupervised representation learning by predicting image rotations," Mars 2018.

[2] DataSaiyentist, "Rotnet," *http//github.com/DataSaiyentist/RotNet/tree/main*, 2023.