

Deep Learning Report

Unsupervised Data Augmentation on MNIST Dataset

Jaad BELHOUARI, Léos COUTROT, Luc YAO

May 23, 2025

Contents

1	Introduction	2
2	Contexte du projet	2
3	Unsupervised Data Augmentation (UDA)	2
3.1	Formulation Mathématique	3
3.2	Rôle de l'Augmentation des Données	3
4	Approche et méthodologie	4
4.1	Méthodologie générale	4
4.2	Optimisation des hyperparamètres et fine-tuning du modèle	5
5	Resultats	5
5.1	Modèle "témoin"	5
5.2	Modèle avec Unsupervised Data Augmentation (UDA)	7
5.3	Modèle finale : "State of the art"	8
6	Conclusion	10
7	Annexe	12

1 Introduction

Le deep learning a révolutionné de nombreux domaines, notamment la reconnaissance des images, en permettant aux modèles d'apprendre des représentations riches à partir de données. Cependant, la réussite de ces modèles repose souvent sur la disponibilité de larges ensembles de données labélisées. Or, l'annotation manuelle des données est coûteuse en temps et en ressources, ce qui constitue une barrière importante dans de nombreuses applications.

Pour surmonter ce défi, les approches d'apprentissage semi-supervisé (Semi-Supervised Learning, SSL) ont gagné en popularité. Ces méthodes exploitent à la fois un petit ensemble de données labélisées et un grand volume de données non labélisées, maximisant ainsi les performances tout en minimisant les besoins en annotations.

Dans ce projet, nous explorons une approche d'apprentissage semi-supervisé : l'augmentation des données non supervisées (UDA - Unsupervised Data Augmentation). L'objectif est de tirer parti de la robustesse et de la généralisation offertes par les techniques d'augmentation des données appliquées au grand ensemble non labélisé, tout en utilisant efficacement les informations limitées provenant des données labélisées.

Pour tester cette approche, nous utilisons le célèbre dataset MNIST, qui contient 60 000 images de chiffres manuscrits en niveaux de gris, divisées en un petit ensemble labélisé de 100 échantillons et un ensemble non labélisé de 59 900 échantillons. L'objectif principal est d'évaluer dans quelle mesure l'apprentissage semi-supervisé avec UDA peut améliorer les performances du modèle, même en présence de données labélisées extrêmement limitées.

2 Contexte du projet

Comme expliqué plus tôt, le projet va porter sur le jeu de données MNIST, mais avec la particularité que l'on va se concentrer sur uniquement 100 données labélisées, et le reste des données seront donc non labélisées. Afin de pouvoir attester ou non de l'efficacité de la méthode présentée (Unsupervised Data Augmentation), nous allons tout d'abord nous concentrer sur un modèle de base qui va simplement apprendre sur les données labélisées. Puis nous utiliserons la même architecture mais avec UDA afin de pouvoir comparer les différentes performances. De plus, nous allons comparer différentes architectures de réseaux de neurones pour essayer de trouver la meilleure pour notre projet.

Ce projet est donc un projet de classification d'images, ainsi comme les données sont bien réparties équitablement dans nos 10 classes, notre métrique d'évaluation sera l'accuracy. C'est donc cette métrique qui nous permettra de choisir entre différents modèles.

3 Unsupervised Data Augmentation (UDA)

L'idée principale derrière UDA repose sur l'entraînement par consistance (*Consistency Training*), qui impose que les prédictions du modèle soient invariantes face à des modifications appliquées aux exemples d'entrée. Cette approche est représentée formellement par la minimisation d'une perte de consistance entre les prédictions effectuées sur des données originales et augmentées. Cette perte de consistance va alors être combinée à notre perte (des données labélisées, généralement la Cross Entropy) pour obtenir une nouvelle perte globale. Ainsi le modèle va pouvoir gagner en robustesse grâce à cette approche. Voici ci-dessous un schéma détaillant les idées principales se trouvant derrière UDA.

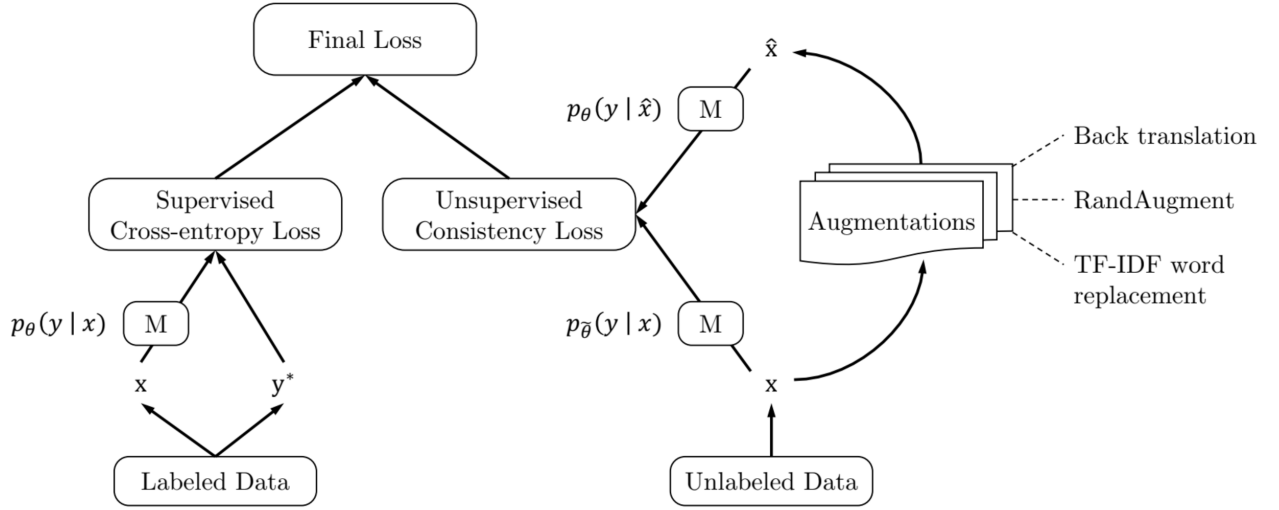


Figure 1: Schéma du fonctionnement générale d'UDA

3.1 Formulation Mathématique

Soit x une entrée et y^* son label associée. Pour rappel, l'objectif est d'apprendre un modèle $p_{\theta}(y|x)$, où θ désigne les paramètres du modèle. Les ensembles de données labélisées et non labélisées suivent les distributions respectives $p_L(x)$ et $p_U(x)$. UDA minimise la perte globale définie comme suit :

$$\mathcal{J}(\theta) = \mathbb{E}_{x \sim p_L(x)} [-\log p_{\theta}(y^*|x)] + \lambda \mathbb{E}_{x \sim p_U(x)} \mathbb{E}_{\hat{x} \sim q(\hat{x}|x)} [\text{CE}(p_{\tilde{\theta}}(y|x), p_{\theta}(y|\hat{x}))],$$

où :

- $q(\hat{x}|x)$ représente une distribution de transformation (augmentation des données),
- CE est l'entropie croisée

$$\text{CE} = \text{KL}(P \parallel Q) + \mathcal{H}(P)$$

- $\tilde{\theta}$ est une copie des paramètres du modèle, utilisée pour stabiliser l'apprentissage en empêchant la rétropropagation des gradients à travers $\tilde{\theta}$,
- λ est un facteur de pondération entre la perte supervisée et la perte non supervisée.

Ainsi on comprend donc que nous allons introduire un nouvel hyperparamètre λ qui devra être choisi avec précaution afin qu'on puisse tirer le meilleur de l'aspect supervisé et non supervisé de notre apprentissage.

De plus on remarque que dans notre cas, on peut limiter l'entropie croisée à la divergence de Kullback–Leibler, qui est la partie qui nous intéresse vraiment dans notre contexte.

3.2 Rôle de l'Augmentation des Données

Les transformations $q(\hat{x}|x)$ appliquées dans UDA sont issues de techniques avancées, comme RandAugment pour la vision (qui est le cas qui nous intéresse ici) ou le back-traduction pour la NLP. Ces augmentations doivent respecter les propriétés suivantes :

- **Validité des labels :** Les transformations produisent des exemples augmentés \hat{x} partageant la même label que x .

- **Diversité** : Elles génèrent une large variété d'exemples augmentés, augmentant ainsi l'efficacité d'apprentissage.
- **Biais inductifs** : Elles introduisent des biais inductifs pertinents pour la tâche considérée.

On remarque bien ici que certains de ces aspects vont déjà poser problème. En effet la partialité des labels va nous empêcher d'appliquer certaines augmentations comme la rotation (que ce soit à 90° ou 180°) car elle va entraîner des conflits. En effet le seul point permettant de faire la différence entre un 6 et un 9 est l'orientation de la photo. Ainsi, pour les données non labélisées nous allons uniquement utiliser la transformation RandAugment (comme conseillé dans l'article). On voit dans le schéma ci-dessous que les transformations effectuées avec RandAugment ne risquent pas de créer des images qui pourraient prêter à confusion.



Figure 2: Fonctionnement de RandAugment

Enfin, l'article mentionne qu'il est toujours aussi crucial d'utiliser de la data augmentation sur les données labélisées. Dans notre cas, nous avons choisi d'effectuer les augmentations suivantes :

- Recadrage aléatoire d'une région dans l'image d'entrée
- Normalisation des valeurs des pixels
- Rotation aléatoire entre -15 et 15 degrés
- Translation aléatoire
- Modification aléatoirement des propriétés de couleur

Ainsi, grâce à ces augmentations, nous avons pu passer de 100 données labélisées à plus de 5000.

4 Approche et méthodologie

4.1 Méthodologie générale

Pour ce projet nous allons comparer différentes approches. Tout d'abord, notre référence et cas "témoin" sera un modèle avec une architecture CNN qui sera entraîné uniquement grâce

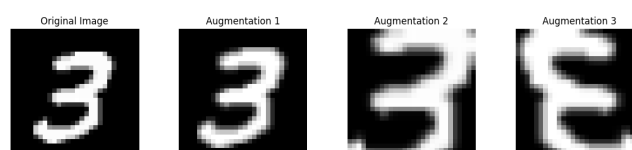


Figure 3: Exemple de données augmentées

aux 100 données labélisées puis avec les données labélisées augmentées. Puis avec la même architecture nous entraînerons un second modèle en utilisant l’approche UDA afin de pouvoir évaluer l’efficacité de la méthode. Enfin, une fois ceci fait, nous entraînerons d’autres modèles avec des architectures plus complexes et en utilisant UDA afin d’obtenir les meilleurs résultats possibles.

4.2 Optimisation des hyperparamètres et fine-tuning du modèle

Afin d’obtenir les meilleures performances possibles, il est important d’optimiser au mieux tous les hyperparamètres disponibles. Dans notre cas nous avons utilisé :

- **Un learning rate scheduler** : Ce dernier va faire varier notre learning rate afin qu’il permette à notre modèle d’apprendre le plus vite possible. Dans notre cas, nous avons choisi le cosine learning rate scheduler (Figure 4)
- **Un algorithme de descente de gradient optimisé** : Pour cet aspect nous avons choisi l’algorithme de descente de gradient stochastique (SGD).
- **Le coefficient λ d’UDA** : Pour ce coefficient, nous avons commencé avec une valeur de 1 (comme recommandé dans l’article), puis nous avons ajusté empiriquement ce coefficient pour éviter de passer par un GridSearch qui aurait été trop coûteux en terme de temps.
- **La température de la divergence de KL** : Là aussi, nous sommes parti d’une base de 0.4, puis nous avons ajusté empiriquement pour trouver la valeur optimale.

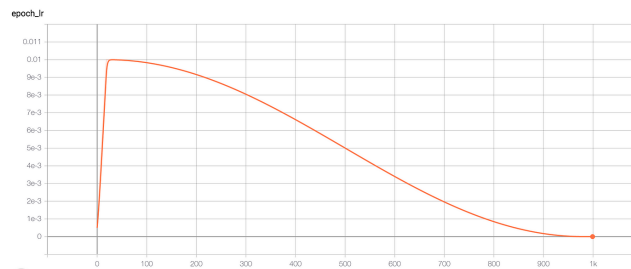


Figure 4: Schéma du fonctionnement du cosine learning rate scheduler

5 Resultats

5.1 Modèle "témoin"

Pour établir une base de comparaison solide, nous avons d’abord entraîné un modèle de convolution (CNN) simple sur les 100 données labélisées fournies, sans utiliser d’augmentation de données. L’objectif ici est de voir dans quelle mesure l’approche UDA améliore les performances par rapport à un modèle purement supervisé utilisant un ensemble de données très restreint.

L’architecture du modèle de convolution est la suivante :

Couche	Taille
Conv2D (32, 3, 'relu', padding='same')	(28, 28, 32)
Conv2D (32, 3, 'relu', padding='same')	(28, 28, 32)
MaxPooling2D (2, 2)	(14, 14, 32)
Dropout (0.25)	(14, 14, 32)
Flatten	(6272)
Dense (512, 'relu')	(512)
Dropout (0.5)	(512)
Dense (10, 'softmax')	(10)

Table 1: Résumé de l'architecture du réseau

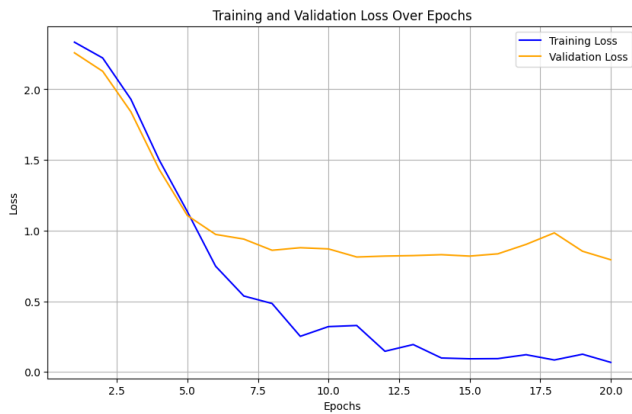


Figure 5: Loss en fonction des epochs

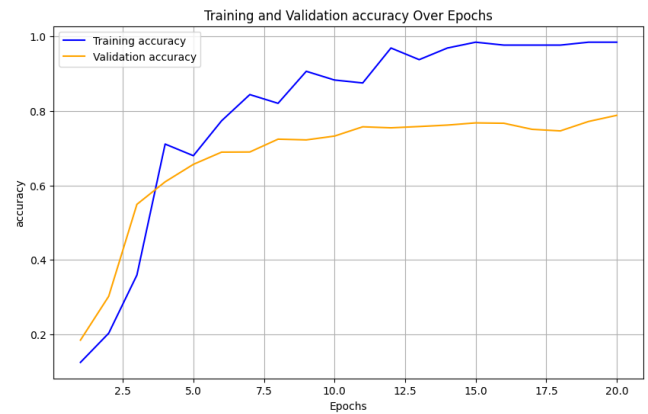


Figure 6: Accuracy en fonction des epochs

Nous obtenons une accuracy finale de **0.79**, et la matrice de confusion présentée à la [Figure 7](#) illustre les bonnes performances de ce modèle. Les erreurs de classification restent cohérentes, notamment dans des cas tels que la confusion entre un 9 prédit et un 4 réel.

Compte tenu du fait que ce modèle a été entraîné sur seulement 100 données labelisées, ces résultats sont satisfaisants et constituent une base de référence pertinente.

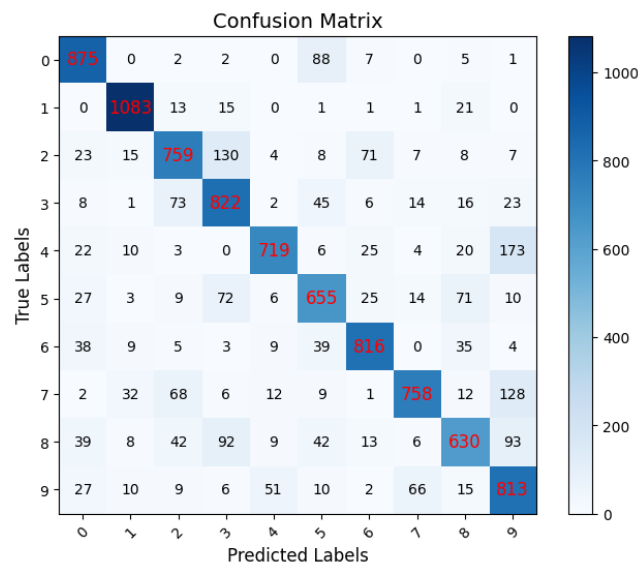


Figure 7: Matrice de confusion du modèle

5.2 Modèle avec Unsupervised Data Augmentation (UDA)

Dans cette section, nous appliquons l'approche UDA (Unsupervised Data Augmentation). Cette méthode utilise à la fois les données labélisées et un grand ensemble de données non labélisées, renforçant ainsi la capacité du modèle à généraliser.

Nous avons utilisés la même architecture que le modèle témoin. Les détails de la configuration sont présentés dans le [Table 2](#) pour les hyperparamètres.

Variable	Valeur	Signification
train_batch_size	10	Taille des batchs utilisés pour les données supervisées
unsup_batch_size	64	Taille des batchs utilisés pour les données non supervisées
eval_batch_size	64	Taille des batchs utilisés lors de l'évaluation
learning_rate	5×10^{-5}	Initialisation du taux d'apprentissage utilisé par l'optimiseur
train_steps	5000	Nombre total d'étapes d'entraînement
uda_temp	0.3	Température utilisée dans la fonction Softmax pour l'approche UDA
unsup_coeff	1.1	Coefficient de pondération pour la perte non supervisée
log_steps	100	Fréquence d'affichage des logs durant l'entraînement
eval_steps	100	Fréquence d'évaluation du modèle durant l'entraînement

Table 2: Configuration des hyperparamètres

La configuration repose sur un total de 5000 étapes d'entraînement, avec un coefficient de perte non supervisée `unsup_coeff` fixé à 1.1 et `uda_temp` = 0.3. Les valeurs initiales ont été inspirées de l'article, qui a servi de référence pour déterminer les réglages de départ. Cette configuration a été établie après plusieurs ajustements des hyperparamètres. On obtient le graphique suivant :

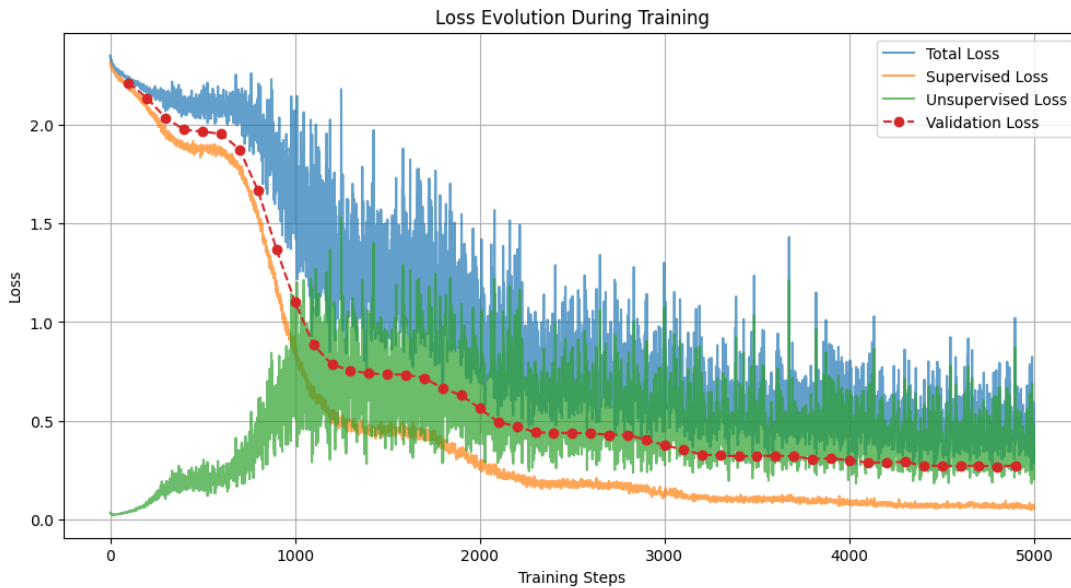


Figure 8: Graphique des loss

Le modèle avec UDA converge correctement et semble bien apprendre à partir des données étiquetées et non étiquetées : la faible loss supervisée montre que le modèle s’ajuste bien aux données supervisées et la perte non supervisée diminue également de manière satisfaisante, ce qui prouve que l’approche semi-supervisée améliore la performance globale.

Nous obtenons une accuracy finale de **0.88**, et la matrice de confusion présentée à la Figure 9 illustre les bonnes performances de ce modèle. Les erreurs de classification restent cohérentes, notamment dans des cas tels que la confusion entre un 8 prédit et un 5 ou un 3 réel.

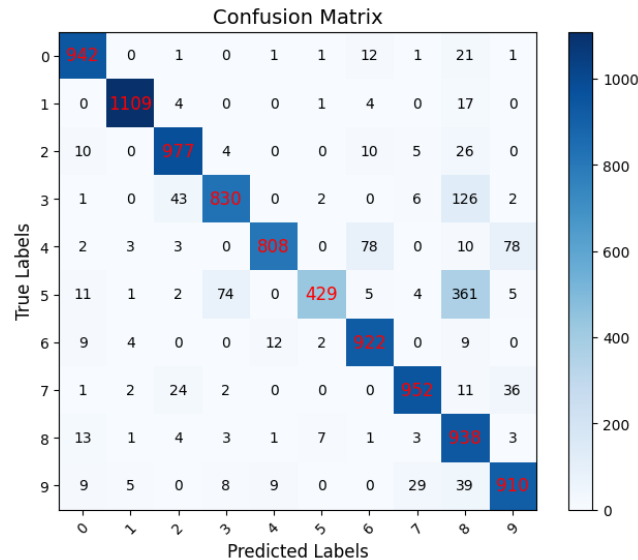


Figure 9: Matrice de confusion du modèle

Les résultats montrent que l’approche UDA améliore significativement les performances du modèle comparé au modèle témoin.

5.3 Modèle finale : ”State of the art”

Dans le cadre de la recherche d’un modèle plus complexe et performant pour la reconstruction des données en apprentissage non supervisé, l’architecture suivante (Table 3) se révèle particulièrement adaptée pour résoudre le problème de classification des données MNIST comme en témoignent les benchmarks sur le site KAGGLE.

(Source : <https://www.kaggle.com/code/cdeotte/how-to-choose-cnn-architecture-mnist>)

Couche	Taille
Conv2D (32, 3, 'relu', padding=1)	(28, 28, 32)
BatchNorm2D	(28, 28, 32)
Conv2D (32, 3, 'relu', padding=1)	(28, 28, 32)
BatchNorm2D	(28, 28, 32)
Conv2D (32, 5, 'relu', stride=2, padding=2)	(14, 14, 32)
BatchNorm2D	(14, 14, 32)
Dropout (0.3)	(14, 14, 32)
Conv2D (64, 3, 'relu', padding=1)	(14, 14, 64)
BatchNorm2D	(14, 14, 64)
Conv2D (64, 3, 'relu', padding=1)	(14, 14, 64)
BatchNorm2D	(14, 14, 64)
Conv2D (64, 5, 'relu', stride=2, padding=2)	(7, 7, 64)
BatchNorm2D	(7, 7, 64)
Dropout (0.4)	(7, 7, 64)
Flatten	(3136)
Dense (128, 'relu')	(128)
Dropout (0.5)	(128)
Dense (10, 'softmax')	(10)

Table 3: Résumé de l'architecture du réseau

Dans une configuration initiale où le modèle est entraîné uniquement sur un sous-ensemble de 100 labels, ses performances restent limitées, avec une précision modeste de 0.23, comme illustré par la matrice de confusion [Figure 10](#).

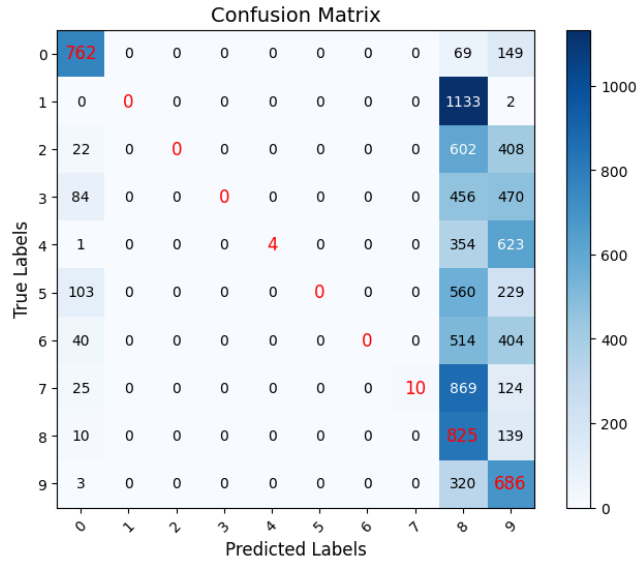


Figure 10: Matrice de confusion avec 100 labels

Cependant, en intégrant ce modèle dans un cadre d'apprentissage semi-supervisé à l'aide de l'approche UDA, ses performances se sont considérablement améliorées. En utilisant des données non labellisées augmentées, l'erreur supervisée diminue rapidement, comme en témoigne l'évolution des loss sur l'ensemble de validation et des pertes non supervisées, illustrée dans [Figure 11](#).

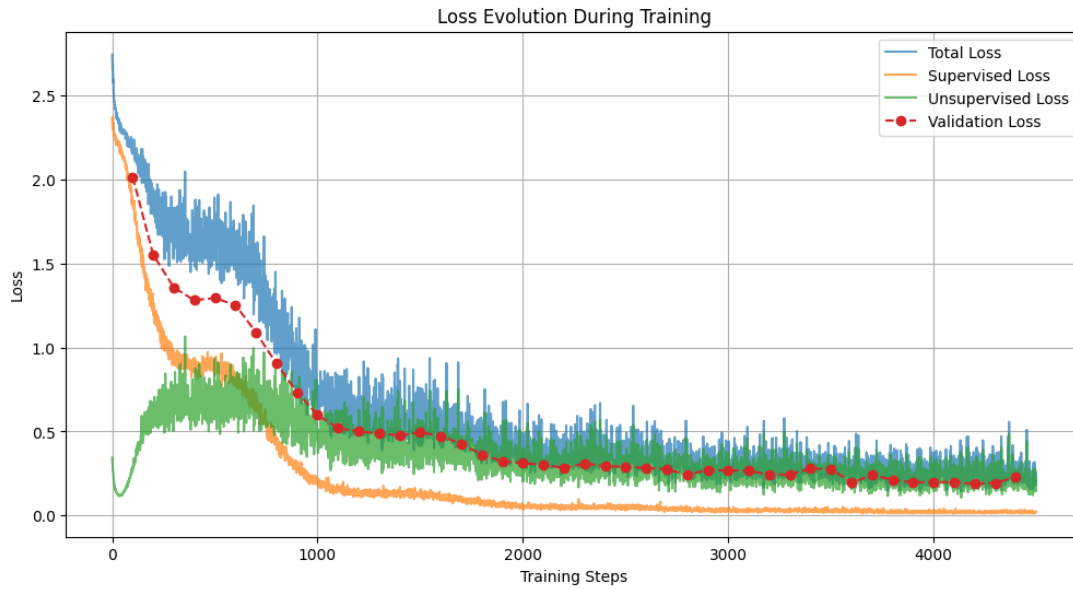


Figure 11: Évolution des pertes lors de l'apprentissage avec UDA

La performance finale du modèle montre une accuracy significativement améliorée égale à **0.97**, confirmée par la matrice de confusion finale [Figure 12](#).

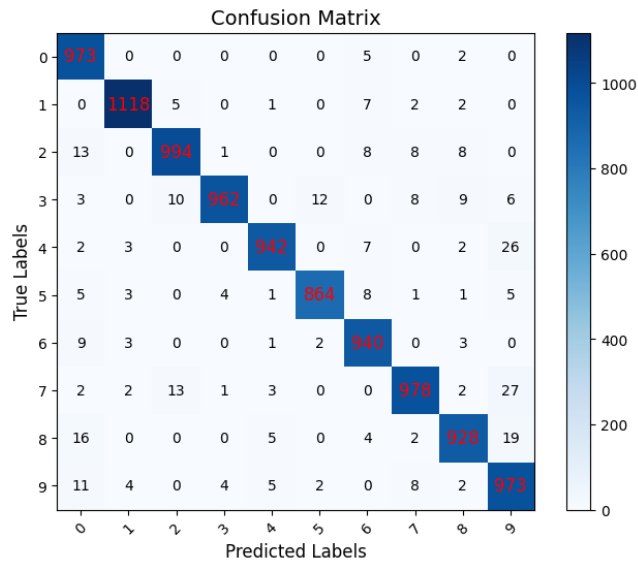


Figure 12: Matrice de confusion finale avec UDA

Ces résultats mettent en évidence la pertinence de l'utilisation de modèles complexes et de stratégies comme UDA pour exploiter efficacement des données non labellisées, ce qui est crucial dans des scénarios où les données annotées sont rares.

6 Conclusion

Le tableau suivant résume les performances des différentes approches explorées dans ce projet :

Modèle	Utilisation d'UDA	Architecture	Accuracy
Modèle témoin	Non	CNN simple	0.79
Modèle témoin	Oui	CNN simple	0.88
Modèle avancé	Non	Architecture complexe	0.23
Modèle avancé	Oui	Architecture complexe	0.97

Table 4: Résumé des performances des modèles explorés

Ce projet a permis de démontrer l'efficacité de l'approche UDA dans un cadre d'apprentissage semi-supervisé. Les résultats obtenus montrent que l'approche UDA améliore significativement les performances des modèles, même avec un ensemble limité de données étiquetées.

7 Annexe

```
1 # -*- coding: utf-8 -*-
2 """projet_deep_finale.ipynb
3
4 Automatically generated by Colab.
5
6 Original file is located at
7     https://colab.research.google.com/drive/1LQt_fo0l4YpAJPMbVSK-jnEn6Ye-Ul4f
8
9 # Projet Deep Learning : Luc Yao, L os Coutrot, Jaad Belhouari
10
11 ### UNSUPERVISED DATA AUGMENTATION FOR CONSISTENCY TRAINING
12
13 ### 0. Settings
14
15 ##### 0.1. Imports
16 """
17
18 import scipy
19 import numpy as np # manipulate N-dimensional arrays
20 import pandas as pd # data frame
21 import matplotlib.pyplot as plt # data plotting
22 import seaborn # advanced data plotting
23 from sklearn import preprocessing # basic ML models
24 # import scipy # scientific computing library
25
26 # Metrics and confusion matrix
27 from sklearn.metrics import classification_report
28 from sklearn.metrics import confusion_matrix
29
30 # Train/test split
31 from sklearn.model_selection import train_test_split
32 from sklearn.model_selection import StratifiedShuffleSplit
33
34 # To count the number of element in each class
35 from collections import Counter
36
37 # make our loops show a smart progress
38 from tqdm import tqdm
39
40 import os
41
42 # Pytorch libraries
43
44 import torch
45 import torchvision
46 import torchvision.transforms as transforms
47
48 import torch.nn as nn
49 import torch.nn.functional as F
50
51 from torch.utils.data import DataLoader, Subset
52
53 import torch.optim as optim
54
55 from torch.optim import Adam
56
57 ##### 0.2. Connexion to GPU"""
58
59 # Check if is using GPU acceleration
60 print("GPU available:", torch.cuda.is_available())
61
62 print(torch.cuda.current_device()) # Prints the index of the current device
63 print(torch.cuda.get_device_name(torch.cuda.current_device())) # Prints the name of the active device
64
65 print(torch.version.cuda)
66
67 ##### 0.3. Usefull functions"""
68
69 # Function to get the accuracy
70 def get_accuracy(y_true, y_pred):
71     """
72     Get accuracy from classification report.
73
74     Args:
75     - y_true (array-like): True labels.
76     - y_pred (array-like): Predicted labels.
77
78     Returns:
79     - float: Accuracy.
80     """
81     report = classification_report(y_true, y_pred, output_dict=True)
82     accuracy = report['accuracy']
83     return accuracy
84
85 # Function to plot confusion matrix in a proper manner
86 def plot_confusion_matrix(y_true, y_pred):
87     """
88     Plot a confusion matrix in a sexy manner.
89
90     Args:
91     - y_true (array-like): True labels.
92     - y_pred (array-like): Predicted labels.
93
94     Returns:
95     - None
96     """
97
98     # getting min and max values
99     min_label = min([np.min(y_true), np.min(y_pred)])
```

```

101 max_label = max([np.max(y_true), np.max(y_pred)])
102 # list of our test labels
103 labels_test = [x for x in range(int(min_label), int(max_label)+1)]
104
105 # Calculate confusion matrix
106 conf_matrix = confusion_matrix(y_true, y_pred)
107
108 # Plot confusion matrix as heatmap
109 plt.figure(figsize=(8, 6))
110 plt.imshow(conf_matrix, interpolation='nearest', cmap=plt.cm.Blues)
111 plt.title('Confusion Matrix', fontsize=14)
112 plt.colorbar()
113
114 classes = np.unique(labels_test)
115 tick_marks = np.arange(len(classes))
116 plt.xticks(tick_marks, classes, rotation=45)
117 plt.yticks(tick_marks, classes)
118
119 thresh = conf_matrix.max() / 2.
120 for i in range(conf_matrix.shape[0]):
121     for j in range(conf_matrix.shape[1]):
122         color = "white" if conf_matrix[i, j] > thresh else "black"
123         if i == j: # Highlight diagonal cells
124             plt.text(j, i, format(conf_matrix[i, j], 'd'),
125                     ha="center", va="center", color="red", fontsize=12)
126         else:
127             plt.text(j, i, format(conf_matrix[i, j], 'd'),
128                     ha="center", va="center", color=color)
129 plt.ylabel('True Labels', fontsize=12)
130 plt.xlabel('Predicted Labels', fontsize=12)
131
132 # Function to reverse normalization for display
133 def denormalize(tensor, mean=(0.5,), std=(0.5,)):
134     """
135     Denormalizes a normalized tensor for visualization.
136
137     Args:
138         tensor (Tensor): Normalized tensor.
139         mean (tuple): Mean used for normalization.
140         std (tuple): Std deviation used for normalization.
141
142     Returns:
143         Tensor: Denormalized tensor.
144     """
145     for t, m, s in zip(tensor, mean, std):
146         t.mul_(s).add_(-m) # reverse normalization
147     return tensor
148
149 # Visualize a sample image
150 def visualize_transformed_data(augmented_data, i):
151     """
152     Visualizes a transformed image from the augmented data.
153
154     Args:
155         augmented_data (list): List of (image, label) pairs.
156
157     # Get the first image-label pair
158     transformed_image, label = augmented_data[i] # Assuming it's a tensor
159     denorm_image = denormalize(transformed_image.clone()) # Clone to avoid modifying the original tensor
160     np_image = denorm_image.numpy() # Convert to numpy array
161
162     # Remove the channel dimension for grayscale image
163     np_image = np.squeeze(np_image)
164
165     # Display the image
166     plt.imshow(np_image, cmap="gray")
167     plt.title(f"Label: {label}")
168     plt.axis("off")
169     plt.show()
170
171 def evaluate_test_data(model, test_loader, device):
172     """
173     Evaluates the trained model on the test dataset and prints a classification report.
174
175     Args:
176         model: Trained PyTorch model.
177         test_loader: DataLoader for the test dataset.
178         device: Device to run the evaluation on (e.g., "cuda" or "cpu").
179
180     """
181     model.eval() # Set the model to evaluation mode
182     all_predictions = []
183     all_labels = []
184
185     # Define the output activation function
186     output_fn = nn.Softmax(dim=1) # Assuming classification task with softmax outputs
187
188     with torch.no_grad():
189         for inputs, labels in test_loader:
190             # Move data to the correct device
191             inputs, labels = inputs.to(device), labels.to(device)
192
193             # Forward pass
194             outputs = model(inputs)
195
196             # Get predictions
197             predictions = output_fn(outputs).argmax(dim=1) # Get the index of the max log-probability
198             all_predictions.extend(predictions.cpu().numpy())
199             all_labels.extend(labels.cpu().numpy())
200
201     # Compute and print detailed classification report
202     print("Classification Report:")
203     print(classification_report(all_labels, all_predictions))

```

```

204 # Confusion matrix
205 plot_confusion_matrix(all_labels, all_predictions)
206
207 # Function to create augmented dataset
208 def create_augmented_dataset(dataset, indices, augment_transform, target_size=1000):
209     augmented_images = []
210     augmented_labels = []
211
212     original_subset = Subset(dataset, indices)
213     while len(augmented_images) < target_size:
214         for image, label in original_subset:
215             # Convert tensor back to PIL Image for augmentation
216             image_pil = transforms.ToPILImage()(image)
217             augmented_image = augment_transform(image_pil) # Apply the transformation pipeline
218             augmented_images.append(augmented_image)
219             augmented_labels.append(label)
220             if len(augmented_images) >= target_size:
221                 break
222
223     return list(zip(augmented_images, augmented_labels))
224
225 """### 1. Baseline model
226
227 ### 1.1. Data Loading
228 """
229
230 transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=(0.5), std=(0.5))])
231
232 batch_size = 64
233
234 dataset = torchvision.datasets.MNIST(root='./data', train=True,
235                                     download=True, transform=transform)
236
237 testset = torchvision.datasets.MNIST(root='./data', train=False,
238                                     download=True, transform=transform)
239
240 # Visualisation of the test data
241 testset
242
243 """#### Number of elements in each class"""
244
245 classes = dataset.classes
246
247 class_count = {}
248 for _, index in dataset:
249     label = classes[index]
250     if label not in class_count:
251         class_count[label] = 0
252     class_count[label] += 1
253 class_count
254
255 # Splitting the training set into training and validation data
256 random_state = 42 #for reproducible results
257 train_indices, val_indices = train_test_split(list(range(len(dataset.targets))),
258                                             test_size=10000,
259                                             stratify=dataset.targets,
260                                             random_state=random_state)
261
262 trainset = torch.utils.data.Subset(dataset, train_indices)
263 valset = torch.utils.data.Subset(dataset, val_indices)
264
265 """#### 1.2. Data Loader"""
266
267 trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
268                                         shuffle=True, num_workers=2)
269 valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
270                                         shuffle=False, num_workers=2)
271 testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
272                                         shuffle=False, num_workers=2)
273
274 """#### 1.3. Model Building
275
276 As our data are images, we decide to build a Convolutional Neural Network for our classification task.
277 Therefore, let's first train a baseline "naive" model, only using 100 labeled images with no :
278 - data augmentation
279 - semi-supervised learning techniques
280
281 To see how our model perform in this case to prove that the previous techniques are indeed needed to improve significantly the
282 performances.
283
284 """
285 class Net(nn.Module):
286     def __init__(self):
287         super().__init__() # always subclass
288         self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding="same") # first conv layer
289         # the three arguments in_channels, out_channels, kernel_size must be filled, the others are optional and have default
290         # values
291         # out_channels correspond to the number of filters
292         # if height=width in the kernel size, just set one value instead of a tuple
293         # stride is set to 1 by default
294         # padding='same' pads the input so the output has the shape as the input. However, this mode doesn't support any
295         # stride values other than 1.
296         self.conv2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding="same") # second conv layer
297         self.pool = nn.MaxPool2d(kernel_size=2) # maxpooling
298         self.dp1 = nn.Dropout(p=0.25)
299         self.fc1 = nn.Linear(in_features=32 * 14 * 14, out_features=512) # linear layer after flattening
300         self.dp2 = nn.Dropout(p=0.5)
301         self.fc2 = nn.Linear(in_features=512, out_features=10) # we have 10 probability classes to predict so 10 output
302         # features
303
304     def forward(self, x):
305         x = F.relu(self.conv1(x))
306         x = self.pool(F.relu(self.conv2(x)))

```

```

303         x = self.dpi(x)
304         x = torch.flatten(x, 1) # flatten all dimensions except batch
305         x = self.dp2(F.relu(self.fc1(x)))
306         x = self.fc2(x)
307         return x
308
309 device = torch.device("cuda")
310 net = Net().to(device) # train on GPU if available
311
312 print(net) # similar to 'model.summary' in keras
313 print("(model mem allocation) - Memory available : {:.2e}".format(torch.cuda.memory_reserved(0)-torch.cuda.memory_allocated(0)
314 ))
315
316 # Extract labels from the dataset
317 labels = [trainset[i][1] for i in range(len(trainset))] # Extract class labels
318
319 # Use StratifiedShuffleSplit to get stratified indices
320 num_samples = 100 # Total number of samples
321 sss = StratifiedShuffleSplit(n_splits=1, test_size=num_samples, random_state=42)
322 _, indices = next(sss.split(X=trainset, y=labels)) # Split indices
323
324 # Create a subset with the stratified indices
325 stratified_subset = Subset(trainset, indices)
326
327 # Create a DataLoader for the subset
328 batch_size = 32
329 stratified_loader = DataLoader(stratified_subset, batch_size=batch_size, shuffle=True)
330
331 print("The number of labeled data fro training is :", len(stratified_subset))
332
333 class_counts = Counter([stratified_subset[i][1] for i in range(len(stratified_subset))])
334 class_counts
335
336 # Visualization with barplot
337
338 # Extracting the classes and their counts
339 classes = list(class_counts.keys())
340 counts = list(class_counts.values())
341
342 # Creating the barplot
343 plt.bar(classes, counts)
344
345 # Adding labels and title
346 plt.xlabel('Classes')
347 plt.ylabel('Number of Elements')
348 # Adding a horizontal line at y=5 (for example)
349 plt.axhline(y=len(stratified_subset)/len(classes), color='r', linestyle='--')
350 plt.title('Number of Elements in Each Class')
351
352 # Display the plot
353 plt.show()
354
355 """Our labels are balancely distributed over all classes.
356 """
357
358 ##### 1.4. Model training
359
360 # Check if GPU is available
361 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
362
363 # Init
364 epochs = 20
365 output_fn = torch.nn.Softmax(dim=1) # we instantiate the softmax activation function for the output probabilities
366 criterion = nn.CrossEntropyLoss() # we instantiate the loss function
367 optimizer = optim.Adam(net.parameters(), lr=0.001) # we instantiate Adam optimizer that takes as inputs the model parameters
368 and learning rate
369
370 # Move the model to the GPU
371 net.to(device)
372
373 loss_valid, acc_valid = [], []
374 loss_train, acc_train = [], []
375
376 for epoch in tqdm(range(epochs)):
377
378     # Training loop
379     net.train() # always specify that the model is in training mode
380     running_loss = 0.0 # init loss
381     running_acc = 0.
382
383     # Loop over batches returned by the data loader
384     for idx, batch in enumerate(stratified_loader):
385
386         # get the inputs; batch is a tuple of (inputs, labels)
387         inputs, labels = batch
388         inputs = inputs.to(device) # put the data on the same device as the model
389         labels = labels.to(device)
390
391         # put to zero the parameters gradients at each iteration to avoid accumulations
392         optimizer.zero_grad()
393
394         # forward pass + backward pass + update the model parameters
395         out = net(x=inputs) # get predictions
396         loss = criterion(out, labels) # compute loss
397         loss.backward() # compute gradients
398         optimizer.step() # update model parameters according to these gradients and our optimizer strategy
399
400         # Iteration train metrics
401         running_loss += loss.view(1).item()
402         t_out = output_fn(out.detach()).cpu().numpy() # compute softmax (previously instantiated) and detach predictions from the
403 model graph
404         t_out=t_out.argmax(axis=1) # the class with the highest energy is what we choose as prediction
405         ground_truth = labels.cpu().numpy() # detach the labels from GPU device

```

```

403     running_acc += get_accuracy(ground_truth, t_out)
404
405     ### Epochs train metrics ###
406     acc_train.append(running_acc/len(stratified_loader))
407     loss_train.append(running_loss/len(stratified_loader))
408
409     # compute loss and accuracy after an epoch on the train and valid set
410     net.eval() # put the model in evaluation mode (this prevents the use of dropout layers for instance)
411
412     ### VALIDATION DATA ###
413     with torch.no_grad(): # since we're not training, we don't need to calculate the gradients for our outputs
414         idx = 0
415         for batch in valloader:
416             inputs, labels = batch
417             inputs = inputs.to(device)
418             labels = labels.to(device)
419             if idx == 0:
420                 t_out = net(x=inputs)
421                 t_loss = criterion(t_out, labels).view(1).item()
422                 t_out = output_fn(t_out).detach().cpu().numpy() # compute softmax (previously instantiated) and detach predictions
423             from the model graph
424             t_out = t_out.argmax(axis=1) # the class with the highest energy is what we choose as prediction
425             ground_truth = labels.cpu().numpy() # detach the labels from GPU device
426         else:
427             out = net(x=inputs)
428             t_loss = np.hstack((t_loss, criterion(out, labels).item()))
429             t_out = np.hstack((t_out, output_fn(out).argmax(axis=1).detach().cpu().numpy()))
430             ground_truth = np.hstack((ground_truth, labels.detach().cpu().numpy()))
431         idx += 1
432
433     acc_valid.append(get_accuracy(ground_truth, t_out))
434     loss_valid.append(np.mean(t_loss))
435
436     print('| Epoch: {}/{} | Train: Loss {:.4f} Accuracy : {:.4f} '\
437           '| Val: Loss {:.4f} Accuracy : {:.4f}\n'.format(epoch+1, epochs, loss_train[epoch], acc_train[epoch], loss_valid[epoch],
438                                                         acc_valid[epoch]))
439
440     # Plot training and validation loss
441     plt.figure(figsize=(10, 6))
442     plt.plot(range(1, epochs + 1), loss_train, label='Training Loss', color='blue')
443     plt.plot(range(1, epochs + 1), loss_valid, label='Validation Loss', color='orange')
444     plt.xlabel('Epochs')
445     plt.ylabel('Loss')
446     plt.title('Training and Validation Loss Over Epochs')
447     plt.legend()
448     plt.grid(True)
449     plt.show()
450
451     # Plot training and validation accuracy
452     plt.figure(figsize=(10, 6))
453     plt.plot(range(1, epochs + 1), acc_train, label='Training accuracy', color='blue')
454     plt.plot(range(1, epochs + 1), acc_valid, label='Validation accuracy', color='orange')
455     plt.xlabel('Epochs')
456     plt.ylabel('accuracy')
457     plt.title('Training and Validation accuracy Over Epochs')
458     plt.legend()
459     plt.grid(True)
460     plt.show()
461
462     """### 1.5. Model Evaluation """
463
464     # Evaluate on test data
465     net.eval() # Set the model to evaluation mode
466     test_loss = 0.0
467     correct = 0
468     total = 0
469
470     output_fn = torch.nn.Softmax(dim=1) # Instantiate the softmax function for predictions
471
472     with torch.no_grad(): # No need to calculate gradients
473         for inputs, labels in testloader:
474             inputs, labels = inputs.to(device), labels.to(device) # Move data to GPU/CPU
475
476             # Forward pass
477             outputs = net(inputs)
478             loss = criterion(outputs, labels)
479             test_loss += loss.item()
480
481             # Compute predictions and accuracy
482             predictions = output_fn(outputs).argmax(dim=1) # Get predicted class
483             correct += (predictions == labels).sum().item()
484             total += labels.size(0)
485
486     # Compute final metrics
487     average_test_loss = test_loss / len(testloader)
488     test_accuracy = correct / total
489
490     print(f'Test Loss: {average_test_loss:.4f}')
491     print(f'Test Accuracy: {test_accuracy:.4f}')
492
493     # After collecting all predictions and ground truth
494     all_predictions = []
495     all_labels = []
496
497     with torch.no_grad():
498         for inputs, labels in testloader:
499             inputs, labels = inputs.to(device), labels.to(device)
500             outputs = net(inputs)
501             predictions = output_fn(outputs).argmax(dim=1)
502             all_predictions.extend(predictions.cpu().numpy())
503             all_labels.extend(labels.cpu().numpy())
504
505     # Compute detailed classification report

```



```

504 print(classification_report(all_labels, all_predictions))
505
506 # Confusion matrix
507 plot_confusion_matrix(all_labels, all_predictions)
508
509 """### 2. RandAugment method
510
511 ### 2.1. Building new class for semi-supervised technics
512 """
513
514 import matplotlib.pyplot as plt
515
516 class SemiSupervisedTrainer:
517     def __init__(self, model, device, config):
518         self.model = model.to(device)
519         self.device = device
520         self.config = config
521
522         # Define the optimizer, scheduler, and loss functions
523         self.optimizer = optim.SGD(
524             model.parameters(),
525             lr=config["learning_rate"],
526             momentum=0.9,
527             weight_decay=config["weight_decay"]
528         )
529         self.scheduler = optim.lr_scheduler.CosineAnnealingLR(
530             self.optimizer,
531             T_max=config["train_steps"]
532         )
533         self.supervised_criterion = nn.CrossEntropyLoss()
534         self.kl_divergence = nn.KLDivLoss(reduction='batchmean')
535
536         # Initialize lists to store losses
537         self.training_loss_history = []
538         self.supervised_loss_history = []
539         self.unsupervised_loss_history = []
540         self.validation_loss_history = []
541
542     def train(self, labeled_dataset, unlabeled_dataset, val_dataset):
543         labeled_loader = DataLoader(labeled_dataset, batch_size=self.config["train_batch_size"], shuffle=True)
544         unlabeled_loader = DataLoader(unlabeled_dataset, batch_size=self.config["unsup_batch_size"], shuffle=True)
545         val_loader = DataLoader(val_dataset, batch_size=self.config["eval_batch_size"], shuffle=False)
546
547         unlabeled_iter = iter(unlabeled_loader)
548
549         for step in range(self.config["train_steps"]):
550             self.model.train()
551
552             # Initialize accumulators for step losses
553             total_loss_accumulator = 0.0
554             supervised_loss_accumulator = 0.0
555             unsupervised_loss_accumulator = 0.0
556             batch_count = 0
557
558             for labeled_data in labeled_loader:
559                 images_sup, labels_sup = labeled_data
560                 images_sup, labels_sup = images_sup.to(self.device), labels_sup.to(self.device)
561
562                 # if unlabeled_iter is None or not hasattr(unlabeled_iter, "__next__"):
563                 #     unlabeled_iter = iter(unlabeled_loader)
564
565                 try:
566                     images_unsup = next(unlabeled_iter)
567                 except StopIteration:
568                     unlabeled_iter = iter(unlabeled_loader)
569                     images_unsup = next(unlabeled_iter)
570
571                 if isinstance(images_unsup, (list, tuple)):
572                     images_unsup = images_unsup[0]
573
574                 images_unsup = images_unsup.to(self.device)
575
576                 # Supervised predictions
577                 logits_sup = self.model(images_sup)
578                 supervised_loss = self.supervised_criterion(logits_sup, labels_sup)
579
580                 # Unsupervised predictions
581                 with torch.no_grad():
582                     logits_ori = self.model(images_unsup)
583
584                 images_unsup_uint8 = (images_unsup * 255).clamp(0, 255).to(torch.uint8)
585                 augmented_images = transforms.RandAugment(num_ops=2, magnitude=9)(images_unsup_uint8)
586                 augmented_images = augmented_images.to(torch.float32) / 255.0
587                 augmented_images = transforms.Normalize((0.5,), (0.5,))(augmented_images)
588
589                 logits_aug = self.model(augmented_images)
590
591                 unsup_loss = self.kl_divergence(
592                     torch.log_softmax(logits_aug / self.config["uda_temp"], dim=-1),
593                     torch.softmax(logits_ori / self.config["uda_temp"], dim=-1)
594                 )
595
596                 # Combine losses
597                 total_loss = supervised_loss + self.config["unsup_coeff"] * unsup_loss
598
599                 # Accumulate losses
600                 total_loss_accumulator += total_loss.item()
601                 supervised_loss_accumulator += supervised_loss.item()
602                 unsupervised_loss_accumulator += unsup_loss.item()
603                 batch_count += 1
604
605                 # Backpropagation
606                 self.optimizer.zero_grad()

```

```

607         total_loss.backward()
608         self.optimizer.step()
609
610         # Adaptative learning rate
611         if step == 2000:
612             self.optimizer.param_groups[0]['lr'] = 5e-5
613             print(self.optimizer.param_groups[0]['lr'])
614
615         if step == 2500:
616             self.optimizer.param_groups[0]['lr'] = 1e-5
617             print(self.optimizer.param_groups[0]['lr'])
618
619         if step == 3000:
620             self.optimizer.param_groups[0]['lr'] = 5e-6
621             print(self.optimizer.param_groups[0]['lr'])
622
623         if step == 3500:
624             self.optimizer.param_groups[0]['lr'] = 1e-6
625             print(self.optimizer.param_groups[0]['lr'])
626
627         if step == 4000:
628             self.optimizer.param_groups[0]['lr'] = 8e-7
629             print(self.optimizer.param_groups[0]['lr'])
630
631         if step == 4500:
632             self.optimizer.param_groups[0]['lr'] = 5e-7
633             print(self.optimizer.param_groups[0]['lr'])
634
635         #self.scheduler.step()
636
637         # Compute average losses for the step
638         avg_total_loss = total_loss_accumulator / batch_count
639         avg_supervised_loss = supervised_loss_accumulator / batch_count
640         avg_unsupervised_loss = unsupervised_loss_accumulator / batch_count
641
642         # Store the losses
643         self.training_loss_history.append(avg_total_loss)
644         self.supervised_loss_history.append(avg_supervised_loss)
645         self.unsupervised_loss_history.append(avg_unsupervised_loss)
646
647         if step % self.config["log_steps"] == 0:
648             print(
649                 f"Step [{step}/{self.config['train_steps']}]: "
650                 f"Total Loss = {avg_total_loss:.4f}, "
651                 f"Supervised Loss = {avg_supervised_loss:.4f}, "
652                 f"Unsupervised Loss = {avg_unsupervised_loss:.4f}"
653             )
654
655         if step % self.config["eval_steps"] == 0 and step > 0:
656             val_loss = self.evaluate(val_loader)
657             self.validation_loss_history.append(val_loss)
658             print(
659                 f"Step [{step}/{self.config['train_steps']}]: "
660                 f"Validation Loss = {val_loss:.4f}"
661             )
662
663         print("Training complete.")
664         self.plot_losses()
665
666     def evaluate(self, val_loader):
667         self.model.eval()
668         total_loss = 0
669         total = 0
670
671         with torch.no_grad():
672             for images, labels in val_loader:
673                 images, labels = images.to(self.device), labels.to(self.device)
674                 outputs = self.model(images)
675                 loss = self.supervised_criterion(outputs, labels)
676                 total_loss += loss.item() * labels.size(0)
677                 total += labels.size(0)
678
679         avg_loss = total_loss / total
680         self.model.train()
681         return avg_loss
682
683     def plot_losses(self):
684         # Plot the loss curves
685         steps = range(len(self.training_loss_history))
686         plt.figure(figsize=(12, 6))
687         plt.plot(steps, self.training_loss_history, label="Total Loss", alpha=0.7)
688         plt.plot(steps, self.supervised_loss_history, label="Supervised Loss", alpha=0.7)
689         plt.plot(steps, self.unsupervised_loss_history, label="Unsupervised Loss", alpha=0.7)
690
691         # Dynamically calculate eval steps based on recorded validation loss
692         eval_steps = [
693             step for step in range(self.config["eval_steps"], self.config["train_steps"] + 1, self.config["eval_steps"])
694             if len(self.validation_loss_history) >= step
695         ]
696
697         plt.plot(eval_steps, self.validation_loss_history, label="Validation Loss", marker='o', linestyle='--')
698
699         plt.xlabel("Training Steps")
700         plt.ylabel("Loss")
701         plt.title("Loss Evolution During Training")
702         plt.legend()
703         plt.grid(True)
704         plt.show()
705
706     """#### 2.2. Preprocessing of the data
707
708     Let's split our training data into :
709     - 100 labeled data
710     - the rest as unlabeled

```

```

710 """
711
712 # Transformations for MNIST
713 transform = transforms.Compose([
714     transforms.ToTensor(),
715     transforms.Normalize((0.5,), (0.5,))
716 ])
717
718 # Load the full MNIST training dataset
719 dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
720
721 # Define the number of labeled samples
722 num_labeled = 100
723 num_unlabeled = len(dataset) - num_labeled
724
725 # Generate random indices for labeled and unlabeled data
726 indices = torch.randperm(len(dataset))
727 labeled_indices = indices[:num_labeled]
728 unlabeled_indices = indices[num_labeled:]
729
730 # Create subsets
731 labeled_dataset = Subset(dataset, labeled_indices)
732 unlabeled_dataset = Subset(dataset, unlabeled_indices)
733
734 # Verify sizes
735 print(f"Labeled dataset size: {len(labeled_dataset)}")
736 print(f"Unlabeled dataset size: {len(unlabeled_dataset)}")
737
738 # Load test dataset
739 testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
740
741 """Making data augmentation on our 100 labeled examples"""
742
743 import torchvision.transforms as transforms
744 from torch.utils.data import random_split, DataLoader, Subset
745 import random
746
747 # Data augmentation: Apply random zoom and crop
748 # Data augmentation: Apply random zoom and crop, then normalize
749 augment_transform = transforms.Compose([
750     transforms.RandomResizedCrop(28, scale=(0.8, 1.0)), # Random zoom and crop
751     transforms.ToTensor(), # Convert back to tensor
752     transforms.Normalize((0.5,), (0.5,)), # Normalize
753     transforms.RandomRotation(degrees=15), # Rotation alatoire entre -15 et 15 degr s
754     transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)), # D placement jusqu'  10% de la taille
755     transforms.ColorJitter(brightness=0.2, contrast=0.2)
756 ])
757
758 # Create 5000 augmented training samples
759 augmented_labeled_data = create_augmented_dataset(dataset, labeled_indices, augment_transform, target_size=5000)
760
761 # Shuffle the augmented data
762 random.shuffle(augmented_labeled_data)
763
764 # Split the augmented data into training (90%) and validation (10%)
765 train_size = int(0.9 * len(augmented_labeled_data)) # 90% for training
766 val_size = len(augmented_labeled_data) - train_size # 10% for validation
767
768 training_labeled_data = augmented_labeled_data[:train_size]
769 validation_data = augmented_labeled_data[train_size:]
770
771 # Convert to PyTorch Dataset objects
772 class CustomDataset(torch.utils.data.Dataset):
773     def __init__(self, data):
774         self.data = data
775
776     def __len__(self):
777         return len(self.data)
778
779     def __getitem__(self, idx):
780         image, label = self.data[idx]
781         return image, label
782
783 training_dataset = CustomDataset(training_labeled_data)
784 validation_dataset = CustomDataset(validation_data)
785
786 # DataLoaders for training and validation
787 train_loader = DataLoader(training_dataset, batch_size=32, shuffle=True)
788 val_loader = DataLoader(validation_dataset, batch_size=32, shuffle=False)
789
790 # Verify sizes
791 print(f"Training dataset size: {len(training_dataset)}")
792 print(f"Validation dataset size: {len(validation_dataset)}")
793
794 """### 2.3. Visualization of the augmented labeled data"""
795
796 # Example of augmented data
797 visualize_transformed_data(augmented_labeled_data, i=550)
798
799 """### 2.4. Model Training
800
801 ### Test 1 :
802 - "train_batch_size": 10, # smaller than unsup batch size
803 - "unsup_batch_size": 64,
804 - "eval_batch_size": 64,
805 - "learning_rate": 1e-4,
806 - "weight_decay": 0,
807 - "train_steps": 1000,
808 - "uda_temp": 0.5,
809 - "unsup_coeff": 1, # 1 as suggested in the article
810 - "log_steps": 100,
811 - "eval_steps": 200
812 """

```

```

813
814 # Configuration
815 config = {
816     "train_batch_size": 10, # smaller than unsup batch size
817     "unsup_batch_size": 64,
818     "eval_batch_size": 64,
819     "learning_rate": 5e-5,
820     "weight_decay": 0,
821     "train_steps": 4000,
822     "uda_temp": 0.5, # according to the article : setting the Softmax temperature to 0.7, 0.8 or 0.9 leads to the best
                        # performances.
823     "unsup_coeff": 1, # 1 as suggested in the article
824     "log_steps": 100,
825     "eval_steps": 200
826 }
827
828 # Initialize the model
829 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
830 model = Net()
831
832 # Initialize the trainer and train the model
833 trainer = SemiSupervisedTrainer(model, device, config)
834 trainer.train(labeled_dataset, unlabeled_dataset, validation_dataset)
835
836 """#### Test 1 : Model evaluation"""
837
838 # Create DataLoader for test set
839 test_loader = DataLoader(testset, batch_size=64, shuffle=False)
840
841 # Evaluation of our model
842 evaluate_test_data(model, test_loader, device)
843
844 """#### 2.5. Hyperparameter Finetuning
845
846 ##### Note
847 - According to the article : We use a cosine learning rate decay schedule:  $\cos(\frac{7t}{8T})$  where t is the current step and T is
  the total number of steps.
848 However, we decided to use our own cooked adaptative learning rate based on the evolution of the loss on the validation data.
849
850 ##### Test with unsup_coeff = 1.1
851 """
852
853 # Configuration
854 config2 = {
855     "train_batch_size": 10, # smaller than unsup batch size
856     "unsup_batch_size": 64,
857     "eval_batch_size": 64,
858     "learning_rate": 5e-5,
859     "weight_decay": 0,
860     "train_steps": 5000,
861     "uda_temp": 0.3, # according to the article : setting the Softmax temperature to 0.7, 0.8 or 0.9 leads to the best
                        # performances.
862     "unsup_coeff": 1.1, # 1 as suggested in the article
863     "log_steps": 100,
864     "eval_steps": 100
865 }
866
867 # Initialize the model
868 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
869 model5 = Net()
870
871 # Initialize the trainer and train the model
872 trainer4 = SemiSupervisedTrainer(model5, device, config2)
873 trainer4.train(labeled_dataset, unlabeled_dataset, validation_dataset)
874
875 test_loader = DataLoader(testset, batch_size=64, shuffle=False)
876 evaluate_test_data(model5, test_loader, device)
877
878 """##### We keep this final model achieving an accuracy of 0.88
879
880 ### 3. New model and new analysis
881
882 ### 3.1. Data Preprocessing
883 """
884
885 transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=(0.5), std=(0.5))])
886
887 batch_size = 64
888
889 dataset = torchvision.datasets.MNIST(root='./data', train=True,
890                                     download=True, transform=transform)
891
892 testset = torchvision.datasets.MNIST(root='./data', train=False,
893                                     download=True, transform=transform)
894
895 """Splitting the training set into labeled and unlabeled data with :
896 - labeled data : 100
897 - unlabeled data : 59 900 = 60 000 - 100
898 """
899
900 from collections import defaultdict
901
902 # Collect indices for each class
903 class_indices = defaultdict(list)
904 for idx, (_, label) in enumerate(dataset):
905     class_indices[label].append(idx)
906
907 # Ensure reproducibility
908 random.seed(42)
909
910 # Sample exactly 10 indices per class for labeled data
911 labeled_indices = []
912 for label in range(10): # Iterate through each class

```

```

913     labeled_indices.extend(random.sample(class_indices[label], 10))
914
915 # Remaining indices are for unlabeled data
916 all_indices = set(range(len(dataset)))
917 unlabeled_indices = list(all_indices - set(labeled_indices))
918
919 # Create subsets
920 labeled_data = torch.utils.data.Subset(dataset, labeled_indices)
921 unlabeled_data = torch.utils.data.Subset(dataset, unlabeled_indices)
922
923 # Check class counts in labeled data
924 from collections import Counter
925 class_counts = Counter([labeled_data[i][1] for i in range(len(labeled_data))])
926 print("Class counts in labeled data:", class_counts)
927
928 print("Size of the labeled data:", len(labeled_data))
929 print("Size of the unlabeled data:", len(unlabeled_data))
930
931 class_counts = Counter([labeled_data[i][1] for i in range(len(labeled_data))])
932 class_counts
933
934 # Visualization with barplot
935
936 # Extracting the classes and their counts
937 classes = list(class_counts.keys())
938 counts = list(class_counts.values())
939
940 # Creating the barplot
941 plt.bar(classes, counts)
942
943 # Adding labels and title
944 plt.xlabel('Classes')
945 plt.ylabel('Number of Elements')
946 # Adding a horizontal line at y=5 (for example)
947 plt.axhline(y=len(stratified_subset)/len(classes), color='r', linestyle='--')
948 plt.title('Number of Elements in Each Class')
949
950 # Display the plot
951 plt.show()
952
953 """Splitting the labeled set into training labeled and validation data with :
954 - labeled_train : 90
955 - validation_data : 10
956 """
957
958 # Ensure reproducibility
959 random.seed(42)
960
961 # Group labeled data indices by class
962 labeled_class_indices = defaultdict(list)
963 for idx, (_, label) in enumerate(labeled_data):
964     labeled_class_indices[label].append(idx)
965
966 # Create validation set with one example per class
967 validation_indices = []
968 for label in range(10): # Iterate through each class
969     validation_indices.append(random.choice(labeled_class_indices[label]))
970     # Remove selected index from labeled_class_indices
971     labeled_class_indices[label].remove(validation_indices[-1])
972
973 # Remaining indices are for the labeled_train set
974 labeled_train_indices = []
975 for label, indices in labeled_class_indices.items():
976     labeled_train_indices.extend(indices)
977
978 # Create subsets
979 labeled_train = torch.utils.data.Subset(labeled_data, labeled_train_indices)
980 validation_data = torch.utils.data.Subset(labeled_data, validation_indices)
981
982 # Check class counts in validation data
983 validation_class_counts = Counter([validation_data[i][1] for i in range(len(validation_data))])
984 print("Class counts in validation data:", validation_class_counts)
985
986 print("Size of the labeled_train data:", len(labeled_train))
987 print("Size of the validation data:", len(validation_data))
988
989 """Data augmentation
990 - RandomRotation(10): Rotates images randomly by 10 degrees.
991 - RandomAffine(0, translate=(0.1, 0.1)): Applies random translation to images.
992 - Normalize((0.5,), (0.5,)): Normalizes pixel values to have mean 0 and standard deviation 1.
993 """
994
995 # Define augmentation transforms
996 train_transform = transforms.Compose([
997     transforms.RandomRotation(10),
998     transforms.RandomAffine(0, translate=(0.1, 0.1)),
999     transforms.Normalize((0.5,), (0.5,))
1000 ])
1001
1002 val_transform = transforms.Compose([
1003     transforms.Normalize((0.5,), (0.5,))
1004 ])
1005
1006 # Custom dataset to handle augmentation and replication
1007 class AugmentedDataset():
1008     def __init__(self, subset, transform, target_size):
1009         self.subset = subset
1010         self.transform = transform
1011         self.target_size = target_size
1012         self.num_repeats = target_size // len(subset) + 1 # Number of times to repeat each item
1013
1014     def __getitem__(self, index):
1015         # Map index to original subset and repeat augmentation

```

```

1016         original_idx = index % len(self.subset)
1017         image, label = self.subset[original_idx]
1018
1019         # Apply augmentation
1020         if self.transform:
1021             image = self.transform(image)
1022         return image, label
1023
1024     def __len__(self):
1025         return self.target_size
1026
1027 # Create augmented datasets
1028 target_train_size = 5000
1029 target_val_size = 500
1030
1031 augmented_labeled_train = AugmentedDataset(labeled_train, train_transform, target_train_size)
1032 augmented_validation = AugmentedDataset(validation_data, val_transform, target_val_size)
1033
1034 # Create data loaders
1035 labeled_train_loader = DataLoader(augmented_labeled_train, batch_size=64, shuffle=True)
1036 val_loader = DataLoader(augmented_validation, batch_size=64, shuffle=False)
1037
1038 # Test the loaders
1039 for images, labels in labeled_train_loader:
1040     print("Batch of images shape:", images.shape)
1041     print("Batch of labels:", labels)
1042     break
1043
1044 for images, labels in val_loader:
1045     print("Validation batch shape:", images.shape)
1046     print("Validation batch labels:", labels)
1047     break
1048
1049 # Check class counts in augmented data
1050 augmented_class_counts = Counter([augmented_labeled_train[i][1] for i in range(len(augmented_labeled_train))])
1051 print("Class counts in validation data:", augmented_class_counts)
1052
1053 """#### 3.2. Building new model (more complex)"""
1054
1055 import torch
1056 import torch.nn as nn
1057 import torch.nn.functional as F
1058
1059 import torch.nn as nn
1060 import torch.nn.functional as F
1061
1062 class BestMNISTCNN(nn.Module):
1063     def __init__(self):
1064         super(BestMNISTCNN, self).__init__()
1065         # First Block
1066         self.conv1_1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
1067         self.conv1_2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1)
1068         self.conv1_3 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=5, stride=2, padding=2)
1069         self.bn1_1 = nn.BatchNorm2d(32)
1070         self.bn1_2 = nn.BatchNorm2d(32)
1071         self.bn1_3 = nn.BatchNorm2d(32)
1072         self.dropout1 = nn.Dropout(0.3) # Dropout for first block
1073
1074         # Second Block
1075         self.conv2_1 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
1076         self.conv2_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
1077         self.conv2_3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=5, stride=2, padding=2)
1078         self.bn2_1 = nn.BatchNorm2d(64)
1079         self.bn2_2 = nn.BatchNorm2d(64)
1080         self.bn2_3 = nn.BatchNorm2d(64)
1081         self.dropout2 = nn.Dropout(0.4) # Dropout for second block
1082
1083         # Fully Connected Layers
1084         self.fc1 = nn.Linear(64 * 7 * 7, 128) # After downsampling, feature map size is 7x7
1085         self.fc2 = nn.Linear(128, 10)
1086         self.dropout_fc = nn.Dropout(0.5) # Dropout for fully connected layer
1087
1088     def forward(self, x):
1089         # First Block
1090         x = F.relu(self.bn1_1(self.conv1_1(x)))
1091         x = F.relu(self.bn1_2(self.conv1_2(x)))
1092         x = F.relu(self.bn1_3(self.conv1_3(x)))
1093         x = self.dropout1(x) # Apply dropout
1094
1095         # Second Block
1096         x = F.relu(self.bn2_1(self.conv2_1(x)))
1097         x = F.relu(self.bn2_2(self.conv2_2(x)))
1098         x = F.relu(self.bn2_3(self.conv2_3(x)))
1099         x = self.dropout2(x) # Apply dropout
1100
1101         # Flatten for fully connected layers
1102         x = torch.flatten(x, 1)
1103
1104         # Fully Connected Layers
1105         x = F.relu(self.fc1(x))
1106         x = self.dropout_fc(x) # Apply dropout
1107         x = self.fc2(x)
1108         return x
1109
1110 """#### 3.3. Baseline : Naive Training Model (only using 100 labeled data)"""
1111
1112 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
1113
1114 # Hyperparameters
1115 epochs = 20
1116 learning_rate = 0.0001
1117
1118 # Initialize model, loss, and optimizer

```

```

1119 net = BestMNISTCNN().to(device)
1120 criterion = nn.CrossEntropyLoss() # CrossEntropyLoss for multi-class classification
1121 optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate) # Adam optimizer
1122 output_fn = torch.nn.Softmax(dim=1) # Softmax for probabilities
1123
1124 # Track loss and accuracy
1125 loss_train, acc_train = [], []
1126 loss_valid, acc_valid = [], []
1127
1128 # Training Loop
1129 for epoch in tqdm(range(epochs), desc="Training Progress"):
1130     # ---- Training ----
1131     net.train() # Set model to training mode
1132     running_loss = 0.0
1133     running_acc = 0.0
1134
1135     for batch in labeled_train_loader: # Loop over training batches
1136         inputs, labels = batch
1137         inputs, labels = inputs.to(device), labels.to(device)
1138
1139         # Zero the parameter gradients
1140         optimizer.zero_grad()
1141
1142         # Forward pass
1143         outputs = net(inputs)
1144         loss = criterion(outputs, labels)
1145
1146         # Backward pass and optimize
1147         loss.backward()
1148         optimizer.step()
1149
1150         # Accumulate metrics
1151         running_loss += loss.item()
1152         preds = output_fn(outputs).argmax(dim=1).cpu().numpy()
1153         ground_truth = labels.cpu().numpy()
1154         running_acc += (preds == ground_truth).mean()
1155
1156     # Compute epoch-level metrics
1157     train_loss = running_loss / len(train_loader)
1158     train_acc = running_acc / len(train_loader)
1159     loss_train.append(train_loss)
1160     acc_train.append(train_acc)
1161
1162     # ---- Validation ----
1163     net.eval() # Set model to evaluation mode
1164     val_loss = 0.0
1165     val_acc = 0.0
1166
1167     with torch.no_grad(): # Disable gradient calculations
1168         for batch in val_loader:
1169             inputs, labels = batch
1170             inputs, labels = inputs.to(device), labels.to(device)
1171
1172             # Forward pass
1173             outputs = net(inputs)
1174             loss = criterion(outputs, labels)
1175
1176             # Accumulate metrics
1177             val_loss += loss.item()
1178             preds = output_fn(outputs).argmax(dim=1).cpu().numpy()
1179             ground_truth = labels.cpu().numpy()
1180             val_acc += (preds == ground_truth).mean()
1181
1182     # Compute epoch-level validation metrics
1183     val_loss /= len(val_loader)
1184     val_acc /= len(val_loader)
1185     loss_valid.append(val_loss)
1186     acc_valid.append(val_acc)
1187
1188     # Print metrics for the epoch
1189     print(f"| Epoch: {epoch + 1}/{epochs} | "
1190           f"Train Loss: {train_loss:.4f} | Train Accuracy: {train_acc:.4f} | "
1191           f"Validation Loss: {val_loss:.4f} | Validation Accuracy: {val_acc:.4f} |")
1192
1193 # Plot training and validation loss
1194 plt.figure(figsize=(10, 6))
1195 plt.plot(range(1, epochs + 1), loss_train, label='Training Loss', color='blue')
1196 plt.plot(range(1, epochs + 1), loss_valid, label='Validation Loss', color='orange')
1197 plt.xlabel('Epochs')
1198 plt.ylabel('Loss')
1199 plt.title('Training and Validation Loss Over Epochs')
1200 plt.legend()
1201 plt.grid(True)
1202 plt.show()
1203
1204 # Plot training and validation accuracy
1205 plt.figure(figsize=(10, 6))
1206 plt.plot(range(1, epochs + 1), acc_train, label='Training Accuracy', color='blue')
1207 plt.plot(range(1, epochs + 1), acc_valid, label='Validation Accuracy', color='orange')
1208 plt.xlabel('Epochs')
1209 plt.ylabel('Accuracy')
1210 plt.title('Training and Validation Accuracy Over Epochs')
1211 plt.legend()
1212 plt.grid(True)
1213 plt.show()
1214
1215 """#### Baseline : Evaluation of the naive model"""
1216
1217 test_loader = DataLoader(testset, batch_size=64, shuffle=False)
1218 evaluate_test_data(net, test_loader, device)
1219
1220 """#### 3.3. UDA technique"""
1221

```

```

1222 # Configuration
1223 config = {
1224     "train_batch_size": 10, # smaller than unsup batch size
1225     "unsup_batch_size": 64,
1226     "eval_batch_size": 64,
1227     "learning_rate": 5e-5,
1228     "weight_decay": 0,
1229     "train_steps": 4500,
1230     "uda_temp": 0.4, # according to the article : setting the Softmax temperature to 0.7, 0.8 or 0.9 leads to the best
                        # performances.
1231     "unsup_coeff": 1.1, # 1 as suggested in the article
1232     "log_steps": 100,
1233     "eval_steps": 100
1234 }
1235
1236 # Initialize the model
1237 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
1238 model_best4 = BestMNISTCNN()
1239
1240 # Initialize the trainer and train the model
1241 trainer2 = SemiSupervisedTrainer(model_best4, device, config)
1242 trainer2.train(labeled_data, unlabeled_data, validation_dataset)
1243
1244 test_loader = DataLoader(testset, batch_size=64, shuffle=False)
1245 evaluate_test_data(model_best4, test_loader, device)
1246
1247 """#### New try
1248
1249 ### 4. Additional training techniques
1250
1251 #### 4.1. Sharpening prediction class
1252 """
1253
1254 import torch
1255 import torch.nn as nn
1256 import torch.optim as optim
1257 from torch.utils.data import DataLoader
1258 import torchvision.transforms as transforms
1259 import matplotlib.pyplot as plt
1260 from itertools import cycle
1261
1262 class SemiSupervisedTrainerWithBeta:
1263     def __init__(self, model, device, config):
1264         self.model = model.to(device)
1265         self.device = device
1266         self.config = config
1267
1268         # Define the optimizer, scheduler, and loss functions
1269         self.optimizer = optim.SGD(
1270             model.parameters(),
1271             lr=config["learning_rate"],
1272             momentum=0.9,
1273             weight_decay=config["weight_decay"],
1274         )
1275         self.scheduler = optim.lr_scheduler.CosineAnnealingLR(
1276             self.optimizer, T_max=config["train_steps"]
1277         )
1278         self.supervised_criterion = nn.CrossEntropyLoss()
1279         self.kl_divergence = nn.KLDivLoss(reduction="batchmean")
1280
1281         # Initialize lists to store losses
1282         self.training_loss_history = []
1283         self.supervised_loss_history = []
1284         self.unsupervised_loss_history = []
1285         self.validation_loss_history = []
1286
1287     def train(self, labeled_dataset, unlabeled_dataset, val_dataset):
1288         labeled_loader = DataLoader(
1289             labeled_dataset,
1290             batch_size=self.config["train_batch_size"],
1291             shuffle=True
1292         )
1293         unlabeled_loader = DataLoader(
1294             unlabeled_dataset,
1295             batch_size=self.config["unsup_batch_size"],
1296             shuffle=True
1297         )
1298         val_loader = DataLoader(
1299             val_dataset,
1300             batch_size=self.config["eval_batch_size"],
1301             shuffle=False
1302         )
1303
1304         unlabeled_iter = cycle(unlabeled_loader)
1305
1306         for step in range(self.config["train_steps"]):
1307             self.model.train()
1308
1309             total_loss_accumulator = 0.0
1310             supervised_loss_accumulator = 0.0
1311             unsupervised_loss_accumulator = 0.0
1312             batch_count = 0
1313
1314             for labeled_data in labeled_loader:
1315                 images_sup, labels_sup = labeled_data
1316                 images_sup = images_sup.to(self.device), labels_sup.to(self.device)
1317
1318                 # Fetch a batch of unlabeled data
1319                 images_unsup = next(unlabeled_iter)
1320                 if isinstance(images_unsup, (list, tuple)):
1321                     images_unsup = images_unsup[0]
1322                 images_unsup = images_unsup.to(self.device)

```



```

1324         # Supervised loss
1325         logits_sup = self.model(images_sup)
1326         supervised_loss = self.supervised_criterion(logits_sup, labels_sup)
1327
1328         # Unsupervised loss with thresholding
1329         with torch.no_grad():
1330             logits_ori = self.model(images_unsup)
1331             probs_ori = torch.softmax(logits_ori, dim=-1)
1332             max_probs, _ = torch.max(probs_ori, dim=-1)
1333
1334         # Apply the threshold condition
1335         mask = max_probs > self.config["Beta"]
1336         selected_indices = torch.nonzero(mask).squeeze()
1337         if selected_indices.numel() > 0:
1338             logits_selected = logits_ori[selected_indices]
1339             probs_selected = probs_ori[selected_indices]
1340
1341         # Data augmentation for selected examples
1342         images_selected = images_unsup[selected_indices]
1343         images_selected_uint8 = (images_selected * 255).clamp(0, 255).to(torch.uint8)
1344         augmented_images = transforms.RandomAugment(num_ops=2, magnitude=9)(images_selected_uint8)
1345         augmented_images = augmented_images.to(torch.float32) / 255.0
1346         augmented_images = transforms.Normalize((0.5,), (0.5,))(augmented_images).to(self.device)
1347
1348         logits_aug = self.model(augmented_images)
1349
1350         # Compute unsupervised loss
1351         unsup_loss = self.kl_divergence(
1352             torch.log_softmax(logits_aug / self.config["uda_temp"], dim=-1),
1353             probs_selected
1354         )
1355         else:
1356             unsup_loss = torch.tensor(0.0, device=self.device)
1357
1358         # Combine losses
1359         total_loss = supervised_loss + self.config["unsup_coef"] * unsup_loss
1360
1361         # Accumulate losses
1362         total_loss_accumulator += total_loss.item()
1363         supervised_loss_accumulator += supervised_loss.item()
1364         unsupervised_loss_accumulator += unsup_loss.item()
1365         batch_count += 1
1366
1367         # Backpropagation
1368         self.optimizer.zero_grad()
1369         total_loss.backward()
1370         self.optimizer.step()
1371         self.scheduler.step()
1372
1373         # Compute average losses for the step
1374         avg_total_loss = total_loss_accumulator / batch_count
1375         avg_supervised_loss = supervised_loss_accumulator / batch_count
1376         avg_unsupervised_loss = unsupervised_loss_accumulator / batch_count
1377
1378         # Store the losses
1379         self.training_loss_history.append(avg_total_loss)
1380         self.supervised_loss_history.append(avg_supervised_loss)
1381         self.unsupervised_loss_history.append(avg_unsupervised_loss)
1382
1383         if step % self.config["log_steps"] == 0:
1384             print(
1385                 f"Step [{step}/{self.config['train_steps']}] : "
1386                 f"Total Loss = {avg_total_loss:.4f}, "
1387                 f"Supervised Loss = {avg_supervised_loss:.4f}, "
1388                 f"Unsupervised Loss = {avg_unsupervised_loss:.4f}"
1389             )
1390
1391         if step % self.config["eval_steps"] == 0 and step > 0:
1392             val_loss = self.evaluate(val_loader)
1393             self.validation_loss_history.append(val_loss)
1394             print(
1395                 f"Step [{step}/{self.config['train_steps']}] : "
1396                 f"Validation Loss = {val_loss:.4f}"
1397             )
1398
1399         print("Training complete.")
1400         self.plot_losses()
1401
1402     def evaluate(self, val_loader):
1403         self.model.eval()
1404         total_loss = 0
1405         total = 0
1406
1407         with torch.no_grad():
1408             for images, labels in val_loader:
1409                 images, labels = images.to(self.device), labels.to(self.device)
1410                 outputs = self.model(images)
1411                 loss = self.supervised_criterion(outputs, labels)
1412                 total_loss += loss.item() * labels.size(0)
1413                 total += labels.size(0)
1414
1415         avg_loss = total_loss / total
1416         self.model.train()
1417         return avg_loss
1418
1419     def plot_losses(self):
1420         steps = range(len(self.training_loss_history))
1421         plt.figure(figsize=(12, 6))
1422         plt.plot(steps, self.training_loss_history, label="Total Loss", alpha=0.7)
1423         plt.plot(steps, self.supervised_loss_history, label="Supervised Loss", alpha=0.7)
1424         plt.plot(steps, self.unsupervised_loss_history, label="Unsupervised Loss", alpha=0.7)
1425
1426         eval_steps = [

```

```

1427         step for step in range(self.config["eval_steps"], self.config["train_steps"] + 1, self.config["eval_steps"])
1428     ][:len(self.validation_loss_history)]
1429
1430     plt.plot(eval_steps, self.validation_loss_history, label="Validation Loss", marker="o", linestyle="--")
1431
1432     plt.xlabel("Training Steps")
1433     plt.ylabel("Loss")
1434     plt.title("Loss Evolution During Training")
1435     plt.legend()
1436     plt.grid(True)
1437     plt.show()
1438
1439     """#### Unfortunately we didn't find how to make this class works. We will keep our final model achieving 0.97 accuracy"""

```