Université de Bourgogne

Systèmes Intelligents Avancés TD

Auteurs :
Clément GHYS
Benjamin MILHET

Professeur :
M. BROUSSE

clement_ghys@etu.u-bourgogne.fr
benjamin_milhet@etu.u-bourgogne.fr

2023 - 2024

Table des matières

Introduction

La conversion de modèles entre différents frameworks de deep learning constitue un élément clé pour comprendre la flexibilité et les nuances spécifiques à chaque plateforme. Dans ce rapport, nous explorons la traduction des modèles Keras vers PyTorch, en utilisant les codes fournis dans le support de cours comme point de départ.

L'objectif principal de ce rapport est de détailler les étapes de conversion entreprises, les défis rencontrés, ainsi que les performances comparatives des modèles obtenus. Nous mettrons en lumière les différences conceptuelles et techniques entre Keras et PyTorch, offrant ainsi un aperçu clair du processus de traduction des modèles de deep learning.

Le code des différents exercices sont disponible sur notre dépôt : Github

MNIST solved using an MLP

2.1 Introduction

L'objectif de ce premier exercice est de réaliser un premier réseau de neurone artificiel appelé MLP. Pour entraîner notre réseau de neurone, nous allons utiliser le dataset MNIST comprenant un lot d'image représentant les chiffres allant de 0 à 9. Pour réaliser cette exercice, nous nous basons sur un programme Keras afin de le réaliser avec la bibliothèque PyTorch. Ce programme permet de classifier les images des différents chiffres.

2.2 Explications du programme

2.2.1 Les constantes

```
batch_size = 128
num_classes = 10
epochs= 20
learning_rate = 0.001
```

Pendant la phase de conception du programme, nous avons choisit 5 epochs pour réduire le temps d'exécution assez long mais pas non plus trop court pour pouvoir analyser nos résultats. Une fois le programme terminé, nous avons augmenter ce nombre jusqu'à 20 pour obtenir un résultats qui n'évolue plus ou presque plus. Une 'epoch' représente un passage complet sur l'ensemble du jeu de données d'entraînement

2.2.2 Chargement des données

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
mnist_train_set = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
mnist_test_set = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
mnist_train_loader = torch.utils.data.DataLoader(mnist_train_set, batch_size=batch_size, shuffle=True)
mnist_test_loader = torch.utils.data.DataLoader(mnist_test_set, batch_size=batch_size, shuffle=False)
```

Nos images de nombre proviennent du dataset MNIST et nous les séparons en deux dataset distinct. Le premier est utilisé pour l'entraînement de notre auto-encodeur, le second pour réaliser la phase de test. Pour la phase de test, nous ne mélangeons pas les données pour pouvoir avoir l'ensemble des nombres dans l'ordre pour le rendu final.

2.2.3 MLP

```
class Net(nn.Module): # On definit le reseau
    def __init__(self): # On definit les couches
        super(Net, self).__init__() # On herite de la classe Module
        self.fc1 = nn.Linear(784, 500) # On definit la premiere couche lineaire (500 neurones)
        self.fc2 = nn.Linear(500, 500) # On definit la deuxieme couche lineaire (500 neurones)
        self.fc3 = nn.Linear(500, num_classes) # On definit la troisieme couche lineaire (10 neurones)

    def forward(self, x): # On definit la fonction forward qui calcule les sorties du reseau
        out = self.fc1(x) # On calcule la sortie de la premiere couche
        out = F.relu(out) # On applique la fonction d'activation relu
        out = F.dropout(out, p=0.2, training=self.training) # On applique le dropout
        out = self.fc2(out) # On calcule la sortie de la deuxieme couche
        out = F.relu(out) # On applique la fonction d'activation relu
        out = F.dropout(out, p=0.2, training=self.training) # On applique le dropout
        out = self.fc3(out) # On calcule la sortie de la troisieme couche
        return F.log_softmax(out, dim=1) # On applique la fonction d'activation softmax
```

Le MLP est un réseau de neurone artificiel comprenant une couche en entrée, une couche en sortie et composé d'une ou plusieurs couche cachée entre ces deux parties. Dans notre cas, nous l'utilisons pour la classification de chiffres entre 0 et 9.

2.2.4 Initialisation du réseau de neurones et choix de l'optimiseur

```
    MLP = Net() # Initialisation du reseau de neurones MLP

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.RMSprop(MLP.parameters(), lr=learning_rate) # Choix de l'optimiseur
        RMSprop
```

2.2.5 Le programme

```
list_loss_train = [] # Liste de loss pour l'entrainement
list_accuracy_train = [] # Liste d'accuracy pour l'entrainement
list_loss_test = [] # Liste de loss pour le test
list_accuracy_test = [] # Liste d'accuracy pour le test

y_pred_list = []
y_true_list = []

for epoch in range(epochs): # On parcourt les differentes epochs
    MLP.train() # On met le reseau en mode train
    sous_list_loss_train = [] # On initialise une liste pour stocker les loss
    sous_valeur_accuracy_train = 0 # On initialise une variable pour stocker le nombre de bonnes
        predictions
    for inputs_train, labels_train in mnist_train_loader: # On parcourt les batchs du dataset
        optimizer.zero_grad() # On met les gradients a zero
        outputs = MLP(inputs_train.view(-1, 784)) # On calcule les sorties du reseau
        loss = criterion(outputs, labels_train) # On calcule la loss
        loss.backward() # On retropropage la loss
        optimizer.step() # On met a jour les parametres

        sous_list_loss_train.append(loss.item()) # On ajoute la loss a la liste
        _, predicted = torch.max(outputs, 1) # On recupere les predictions
        correct = (predicted == labels_train).sum().item() # On calcule le nombre de bonnes predictions
        sous_valeur_accuracy_train += correct # On ajoute ce nombre a la variable

    list_accuracy_train.append(sous_valeur_accuracy_train / len(mnist_train_set)) # On ajoute la
        precision moyenne a la liste

    list_loss_train.append(np.mean(sous_list_loss_train)) # On ajoute la loss moyenne a la liste

        MLP.eval() # On met le reseau en mode evaluation
        with torch.no_grad(): # On desactive le calcul des gradients
            sous_list_loss_test = [] # On initialise une liste pour stocker les loss
            sous_valeur_accuracy_test = 0 # On initialise une variable pour stocker le nombre de
                bonnes predictions
            for inputs_test, labels_test in mnist_test_loader: # On parcourt les batchs du dataset de
                test
                test_outputs = MLP(inputs_test.view(-1, 784)) # On calcule les sorties du reseau
                loss = criterion(test_outputs, labels_test) # On calcule la loss
                sous_list_loss_test.append(loss.item()) # On ajoute la loss a la liste
                _, predicted = torch.max(test_outputs, 1) # On recupere les predictions
                correct = (predicted == labels_test).sum().item() # On calcule le nombre de bonnes
                    predictions

                y_pred_list.extend(predicted) # On ajoute les predictions a la liste
                y_true_list.extend(labels_test) # On ajoute les vraies valeurs a la liste

                sous_valeur_accuracy_test += correct # On ajoute ce nombre a la variable
            list_accuracy_test.append(sous_valeur_accuracy_test / len(mnist_test_set)) # On ajoute la
                precision moyenne a la liste
            list_loss_test.append(np.mean(sous_list_loss_test)) # On ajoute la loss moyenne a la liste
```

2.3 Explications des résultats

2.3.1 Loss

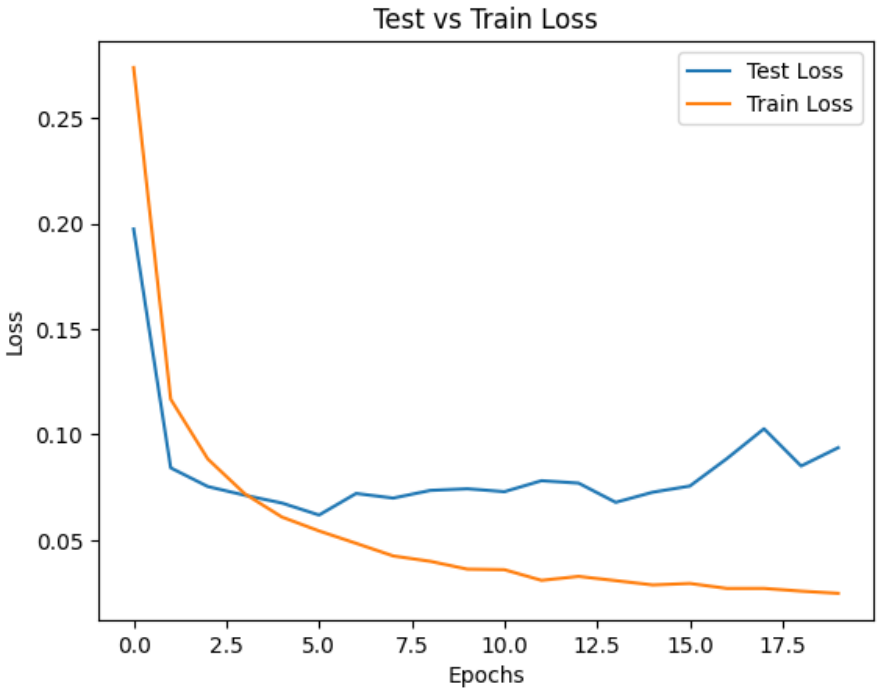

FIGURE 2.1 – Loss

La courbe orange représente l'évolution de la "loss" pour le jeu d'entraînement et la courbe bleu pour le jeu de test. On remarque pour les deux courbes une diminution qui correspond à la période d'apprentissage, puis une période de stagnation qui montre que l'auto-encodeur converge vers une solution. Plus la "loss" est faible, plus l'auto-encodeur a réussi a reproduire l'image d'origine. On remarque que la courbe du jeu de test a un meilleur démarrage que celle du jeu d'entraînement mais converge sur une solution moins précise que lors de l'entraînement.

2.3.2 Accuracy

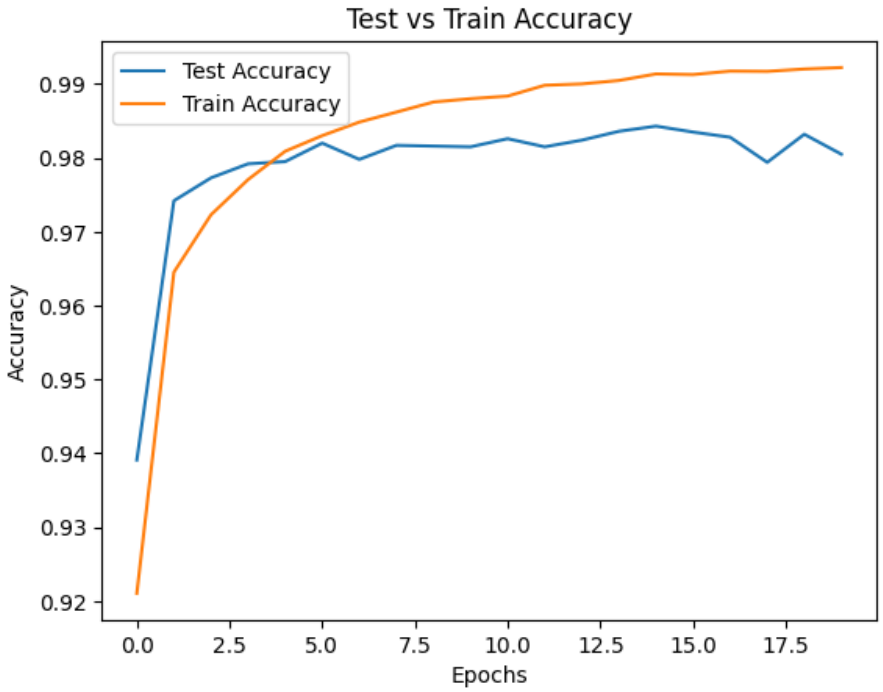

FIGURE 2.2 – Accuracy

La courbe orange représente l'évolution de l'"accuracy" pour le jeu d'entraînement et la courbe bleue pour le jeu de test. On remarque pour les deux courbes une augmentation qui correspond à la période d'apprentissage, puis une période de stabilisation qui montre que l'auto-encodeur converge vers une solution. Plus l'"accuracy" est élevée, plus l'auto-encodeur a réussi à reproduire fidèlement l'image d'origine. Comme pour la "loss", la courbe représentant le jeu test à un meilleur démarrage mais converge vers un moins bonne solution.

2.3.3 Matrice de confusion obtenue sur le dataset de test de MNIST

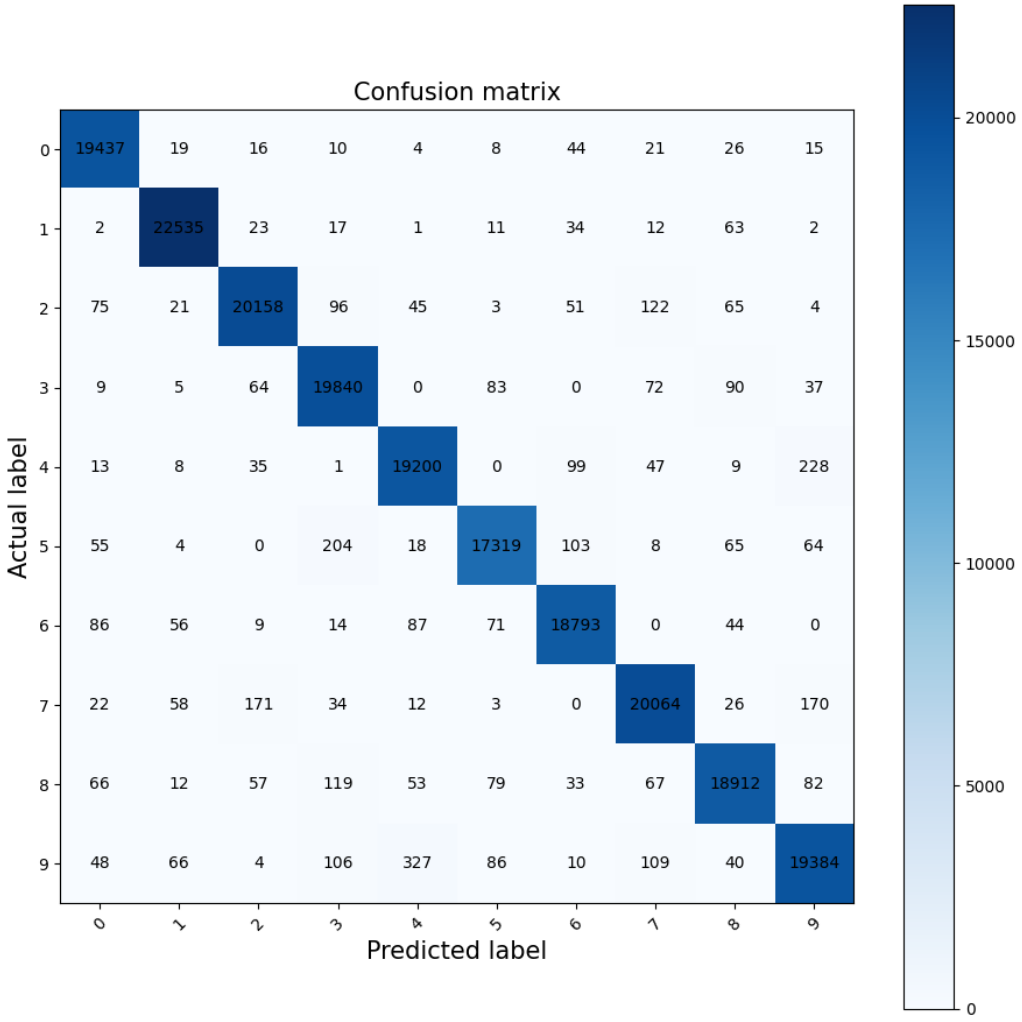

FIGURE 2.3 – Matrice de confusion

La matrice de confusion est un outil pour évaluer les performances d'un modèle de classification, comme dans notre cas pour un réseau de neurones entraîné sur le dataset MNIST. Plus les valeurs dans la diagonales sont élevée et plus les autres sont faibles, plus notre modèle est réussi et minimise les erreurs de classifications. Dans notre cas, certain chiffre on un meilleur taux de classification que d'autre. En combinant cette matrice à nos courbes précédentes on remarque que notre modèle pourrait être encore améliorer pour converger vers une meilleur solution et diminuer les erreurs.

MNIST solved using a CNN

3.1 Introduction

Le reseau de neurone convolutif est un type de reseau de neurone qui est tres utilisé pour la reconnaissance d'image. Il est composé de plusieurs couches de convolution qui permettent de detecter des motifs dans une image. Dans ce TP nous allons utiliser un reseau de neurone convolutif pour reconnaitre des chiffres manuscrits. Pour cela nous allons utiliser le dataset MNIST qui est un dataset de 60000 images de chiffres manuscrits de 28x28 pixels.

3.2 Code pytorch permettant d'entraîner un modèle équivalent

Comme que dans le cours, on défini notre réseau de neurones de la manière suivante :

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential( # Sequential, permet de construire un reseau de neurones couche
            par couche
            nn.Conv2d( # Conv2d, permet de faire une convolution sur une image 2D
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2), # Deuxieme couche de convolution
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.out = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x): # Enchainement des couches de convolution
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x  # return x for visualization

cnn = CNN()
optimizer = optim.Adam(cnn.parameters(), lr = learning_rate) # Definition de l'optimiseur
```

3.3 Graphe d'évolution des métriques de l'entraînement loss et accuracy pour les datasets d'entraînement et de test

3.3.1 Entrainement du réseau

```
for epoch in range(num_epochs):
    cnn.train() # Mode entrainement
    sous_list_loss_train = []
    sous_valeur_accuracy_train = 0

    for i, (inputs_train, labels_train ) in enumerate(loaders['train']):
        b_x = Variable(inputs_train) # batch x
        b_y = Variable(labels_train)
        outputs = cnn(b_x)[0]
        loss = loss_func(outputs, b_y)
        optimizer.zero_grad()
        loss.backward()         # backpropagation, compute gradients
        optimizer.step()

        sous_list_loss_train.append(loss.item()) # ajout de la valeur de la loss pour chaque batch
        _, predicted = torch.max(outputs, 1)
        correct = (predicted == labels_train).sum().item() # Nombre de prediction correcte
        sous_valeur_accuracy_train += correct

    list_accuracy_train.append(sous_valeur_accuracy_train/len(loaders['train'])) # ajout de la valeur
        de l'accuracy pour chaque epoch
    list_loss_train.append(np.mean(sous_list_loss_train)) # ajout de la valeur de la loss pour chaque
        epoch
```

3.3.2 Évaluation du réseau

Le code de l'évaluation du réseau de neurone est assez similaire et est disponible sur GitHub.

3.3.3 Affichages des graphes

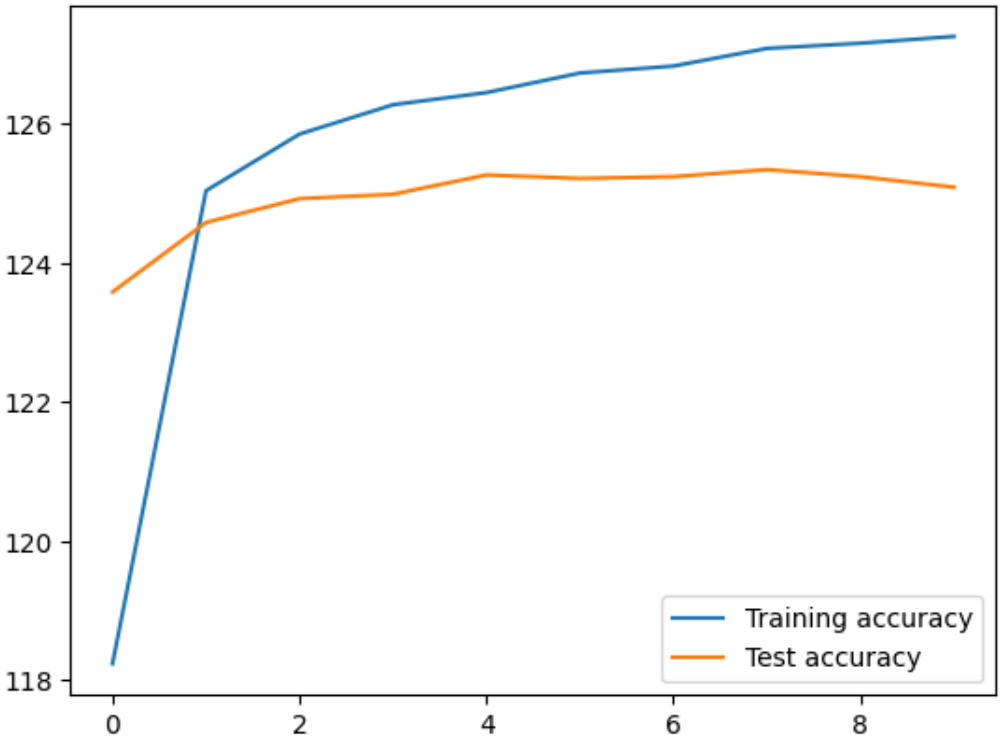

FIGURE 3.1 – Accuracy

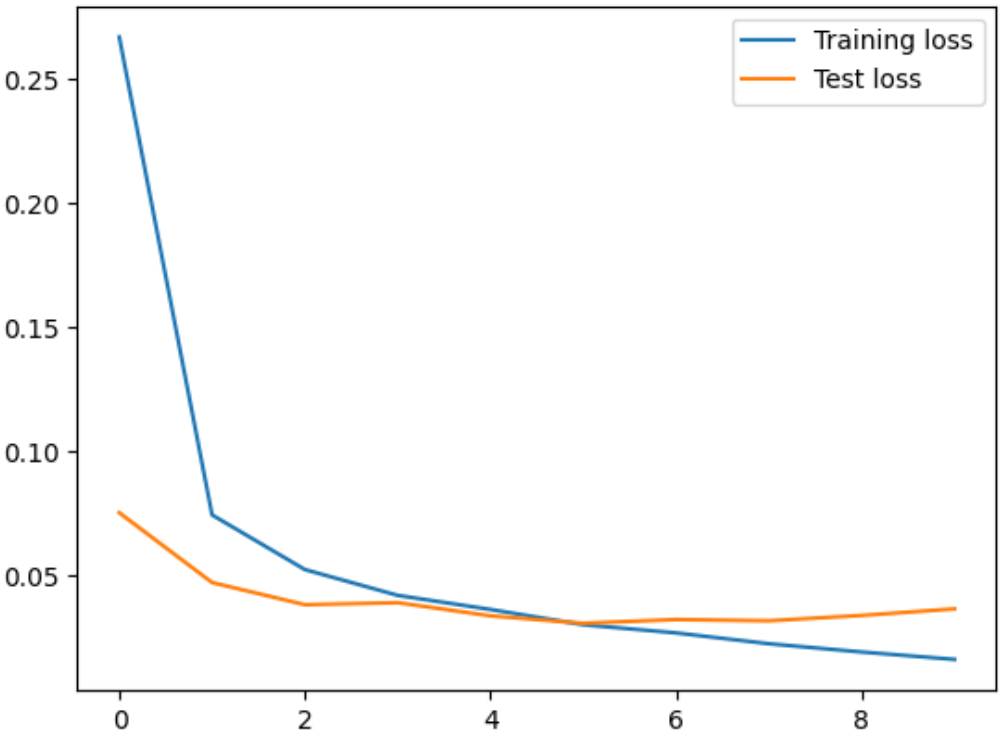

FIGURE 3.2 – Loss

Pour la loss comme pour l'accuracy on observe que les courbes d'entrainement et de test sont relativement proche que cui montre la fiabilité de notre reseau.

3.4 Matrice de confusion obtenue sur le dataset de test de MNIST

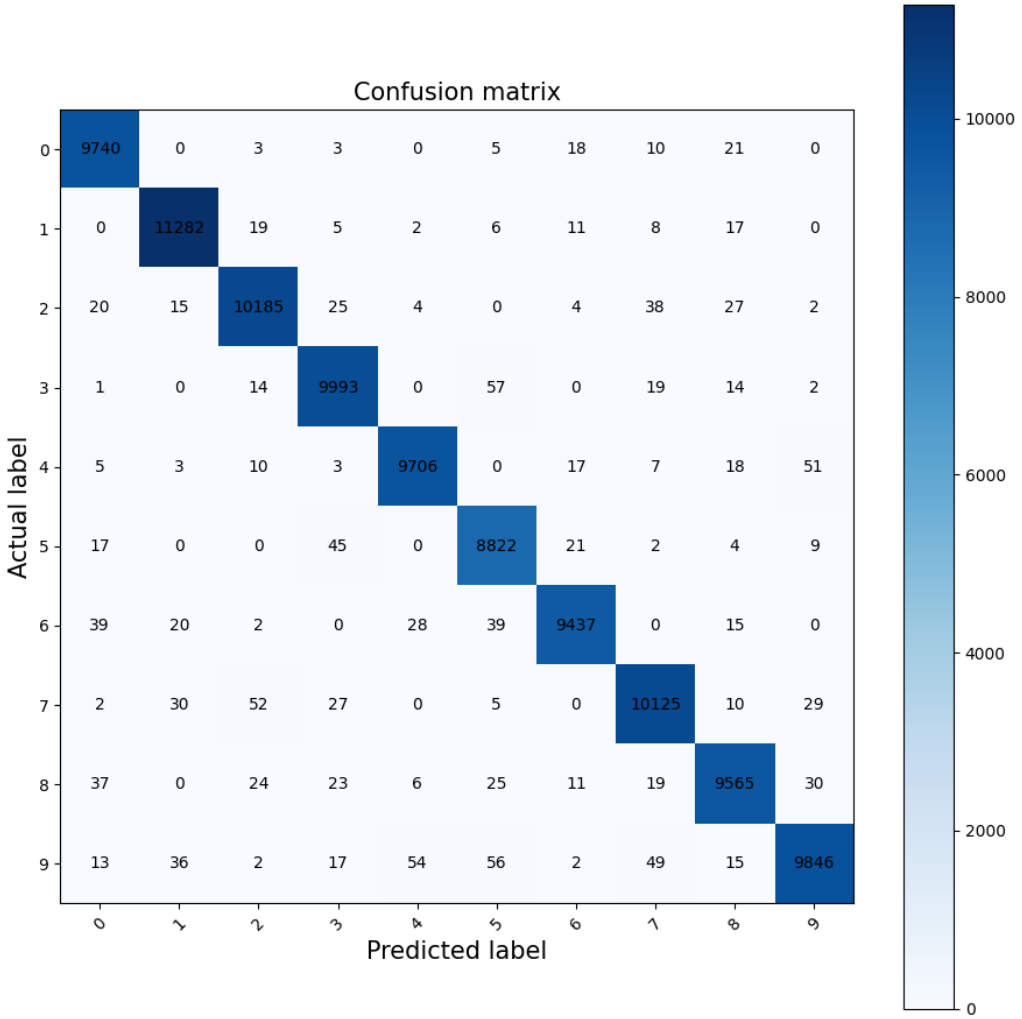

FIGURE 3.3 – Matrice de confusion

Simple AE for MNIST

4.1 Introduction

L'auto-encodage est un algorithme de compression de données où les fonctions de compression et de décompression sont spécifiques aux données. Il faut adapter notre algorithme en fonction des données que nous voulons traités. De plus, Il faut que les données utilisées lors de la phase d'entraînement soit le même type de données que celle sur lesquels nous souhaitons travaillé. Ce type d'algorithme de compression possède de la perte d'information comme pour les algorithmes de compression MP3 ou JPEG. La principale caractéristique de cette algorithme est qu'il apprend par lui-même a faire la compression et la décompression d'après un jeu d'entraînement et un réseau de neurones.

4.2 Explications du programme

4.2.1 Les constantes

```
batch_size = 256
num_classes = 10
epochs= 20
learning_rate = 0.001
encoding_dim = 32
```

Pendant la phase de conception du programme, nous avons choisit 5 epochs pour réduire le temps d'exécution assez long mais pas non plus trop court pour pouvoir analyser nos résultats. Une fois le programme terminé, nous avons augmenter ce nombre jusqu'à 20 pour obtenir un résultats qui n'évolue plus ou presque plus. Une 'epoch' représente un passage complet sur l'ensemble du jeu de données d'entraînement

4.2.2 Chargement des données

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
mnist_train_set = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
mnist_test_set = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
mnist_train_loader = torch.utils.data.DataLoader(mnist_train_set, batch_size=batch_size, shuffle=True)
mnist_test_loader = torch.utils.data.DataLoader(mnist_test_set, batch_size=batch_size, shuffle=False)
```

Nos images de nombre proviennent du dataset MNIST et nous les séparons en deux dataset distinct. Le premier est utilisé pour l'entraînement de notre auto-encodeur, le second pour réaliser la phase de test. Pour la phase de test, nous ne mélangeons pas les données pour pouvoir avoir l'ensemble des nombres dans l'ordre pour le rendu final.

4.2.3 L'Auto-encodeur

```
class Autoencoder(nn.Module): # Autoencoder
    def __init__(self): # Constructeur
        super(Autoencoder, self).__init__() # Heritage de la classe nn.Module
        self.encoder = nn.Sequential( # Definition de l'encodeur
            nn.Linear(28, encoding_dim), # Couche lineaire de 28 neurones en entree et encoding_dim en
                sortie
            nn.ReLU() # Fonction d'activation ReLU
        )
        self.decoder = nn.Sequential( # Definition du decodeur
            nn.Linear(encoding_dim, 28), # Couche lineaire de encoding_dim neurones en entree et 28 en
                sortie
            nn.Sigmoid() # Fonction d'activation Sigmoid
        )

    def forward(self, x):
        encoded = self.encoder(x) # Encodage de l'entree
        decoded = self.decoder(encoded) # Decodage de l'entree encodee
        return decoded # Renvoie de la sortie decodee
```

Cette auto-encodeur simple permet d'encodé une image possédant une couche linéaire de 28 neurones en entrés a 32 en sortie, puis de le décodée en faisant l'inverse. Il apprend à compresser les données (encodage) puis à les reconstruire (décodage) à partir de la représentation compressée.

4.2.4 Initialisation du réseau de neurones et choix de l'optimiseur

```
    AE = Autoencoder().to(device) # Initialisation du reseau de neurones de l'autoencodeur

    criterion = nn.MSELoss() # Fonction de cout (erreur quadratique moyenne)
    optimizer = optim.Adam(AE.parameters(), lr=learning_rate) # Choix de l'optimiseur Adam
```

4.2.5 Le programme

```
list_loss_train = [] # Liste des loss pour l'entrainement
list_accuracy_train = [] # Liste des accuracy pour l'entrainement
list_loss_test = [] # Liste des loss pour le test
list_accuracy_test = [] # Liste des accuracy pour le test
seuil = 0.5 # Seuil pour la binarisation

for epoch in range(epochs): # Boucle sur les epochs
    AE.train() # Mode entrainement
    sous_list_loss_train = [] # Liste des loss pour l'entrainement
    sous_valeur_accuracy_train = 0 # Valeur de l'accuracy pour l'entrainement
    for data in mnist_train_loader: # Boucle sur les donnees d'entrainement
        img, _ = data # Recuperation des images
        optimizer.zero_grad() # Remise a zero des gradients
        img = img.to(device) # Passage des images sur le GPU
        output = AE(img) # Passage des images dans le reseau
        loss = criterion(output, img) # Calcul de la loss
        loss.backward() # Calcul des gradients
        optimizer.step() # Mise a jour des poids

        sous_list_loss_train.append(loss.item()) # Ajout de la loss a la liste
        sous_valeur_accuracy_train += ((output > seuil) == (img > seuil)).sum().item() # Ajout de
            l'accuracy

    total_pixels = img.view(img.size(0), -1).size(1) # Calcul du nombre de pixels
    list_accuracy_train.append(sous_valeur_accuracy_train / (len(mnist_train_set) * total_pixels)) #
        Ajout de l'accuracy a la liste
    list_loss_train.append(np.mean(sous_list_loss_train)) # Ajout de la loss a la liste

    AE.eval() # Mode evaluation
    with torch.no_grad(): # Pas de calcul de gradient
        sous_list_loss_test = [] # Liste des loss pour le test
        sous_valeur_accuracy_test = 0 # Valeur de l'accuracy pour le test
        for data in mnist_test_loader: # Boucle sur les donnees de test
            img, _ = data # Recuperation des images
            img = img.to(device) # Passage des images sur le GPU
            output = AE(img) # Passage des images dans le reseau
            loss = criterion(output, img) # Calcul de la loss
            sous_list_loss_test.append(loss.item()) # Ajout de la loss a la liste
            sous_valeur_accuracy_test += ((output > seuil) == (img > seuil)).sum().item() # Ajout de
                l'accuracy

        total_pixels = img.view(img.size(0), -1).size(1) # Calcul du nombre de pixels
        list_accuracy_test.append(sous_valeur_accuracy_test / (len(mnist_test_set) * total_pixels)) #
            Ajout de l'accuracy a la liste
        list_loss_test.append(np.mean(sous_list_loss_test)) # Ajout de la loss a la liste

    img = img.cpu().data.numpy() # Passage des images sur le CPU
    output = output.cpu().data.numpy() # Passage des images sur le CPU
    print('epoch [{}/{}], loss train:{:.4f}, loss test:{:.4f}, accuracy train:{:.4f}, accuracy
        test:{:.4f}'.format(epoch + 1, epochs, list_loss_train[-1], list_loss_test[-1],
        list_accuracy_train[-1], list_accuracy_test[-1]))
```

4.3 Explications des résultats

4.3.1 Loss

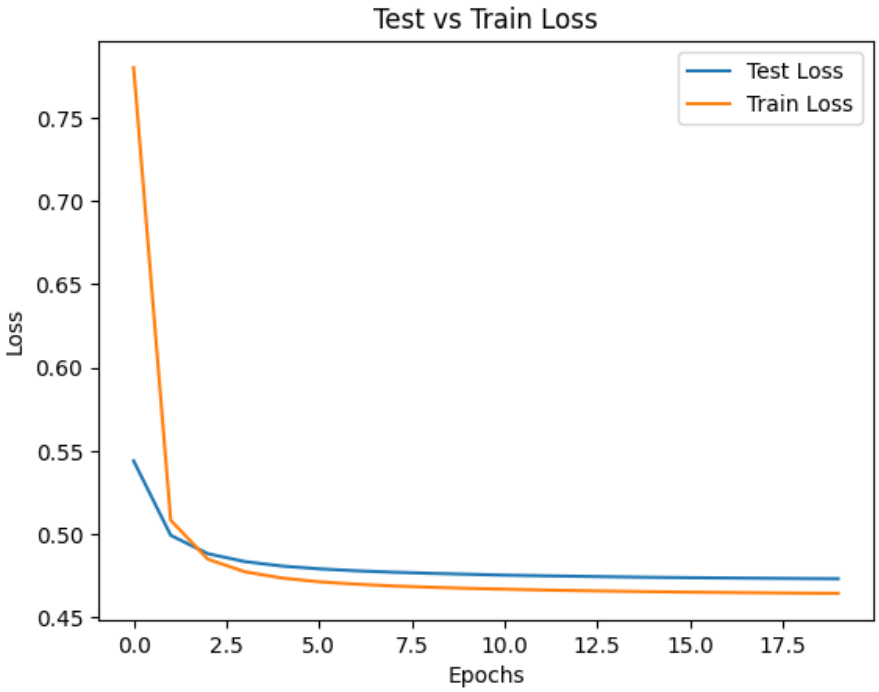

FIGURE 4.1 – Loss

La courbe orange représente l'évolution de la "loss" pour le jeu d'entraînement et la courbe bleu pour le jeu de test. On remarque pour les deux courbes une diminution qui correspond à la période d'apprentissage, puis une période de stagnation qui montre que l'auto-encodeur converge vers une solution. Plus la "loss" est faible, plus l'auto-encodeur a réussi a reproduire l'image d'origine. On remarque une très faible différence entre les deux courbes et une convergence vers une loss similaire.

4.3.2 Accuracy

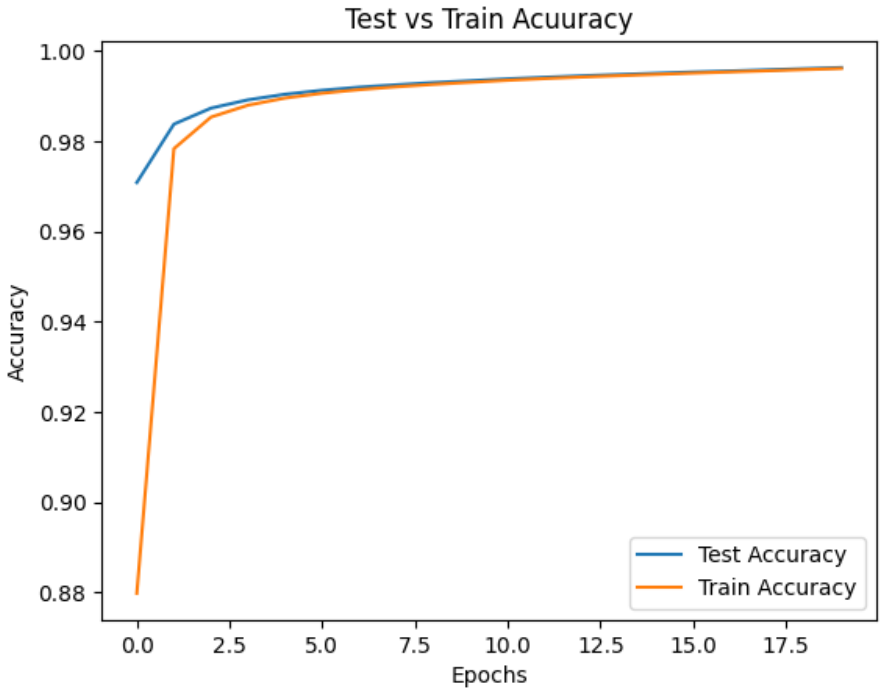

FIGURE 4.2 – Accuracy

La courbe orange représente l'évolution de l'"accuracy" pour le jeu d'entraînement et la courbe bleue pour le jeu de test. On remarque pour les deux courbes une augmentation qui correspond à la période d'apprentissage, puis une période de stabilisation qui montre que l'auto-encodeur converge vers une solution. Plus l'"accuracy" est élevée, plus l'auto-encodeur a réussi à reproduire fidèlement l'image d'origine. On remarque une très faible différence entre les deux courbes et une convergence vers une accuracy similaire.

4.3.3 Comparaisons de 10 images originales et reconstituées

FIGURE 4.3 – 10 images originales et reconstituées

On remarque que les images des chiffres obtenues après l'utilisation de l'auto-encodeur sont plus nette mais elles sont plus pixélisé et les chiffres sont moins lisse que les images d'origines.

Convolutional Auto Encoder for MNIST

5.1 Introduction

Dans ce chapitre nous allons implémenter un autoencodeur convolutionnel pour le dataset MNIST. Nous allons ensuite comparer les images originales et les images reconstituées. L'autoencodeur est un réseau de neurones qui apprend à reconstruire une entrée en sortie. Il est composé de deux parties, un encodeur et un décodeur. L'encodeur va réduire la dimension de l'entrée et le décodeur va reconstruire l'entrée à partir de la sortie de l'encodeur. L'autoencodeur est donc un réseau de neurones non supervisé.

5.2 Code pytorch permettant d'entraîner un modèle équivalent

5.2.1 Encodeur

Dans un premier temps on défini un Encodeur de la manière suivante :

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        # Definition des differentes couches qui vont composer notre encodeur
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=8, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        # Definition de la propagation avant
        # Assemblage des differentes couches
        x = self.conv1(x)
        x = nn.ReLU()(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = nn.ReLU()(x)
        x = self.pool(x)
        x = self.conv3(x)
        x = nn.ReLU()(x)
        x = self.pool(x)
        return x # Retourne la sortie du reseau
```

5.2.2 Décodeur

On défini ensuite le décodeur :

```
class Decoder (nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        # Definition des differentes couches qui vont composer notre decodeur
        self.conv1 = nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=3,stride=2,
            padding=1, output_padding=1)
        self.conv2 = nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=3,stride=2,
            padding=1, output_padding=1)
```

```
        self.conv3 = nn.ConvTranspose2d(in_channels=8, out_channels=16, kernel_size=3,stride=2,
            padding=0, output_padding=1)
        self.conv4 = nn.ConvTranspose2d(in_channels=16, out_channels=1, kernel_size=3,padding=0)
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear')
        self.batch_norm = nn.BatchNorm2d(1)

    def forward(self, x):
        # Definition de la propagation avant
        # Assemblage des differentes couches
        x = self.conv1(x)
        x = nn.ReLU()(x)
        x = self.conv2(x)
        x = nn.ReLU()(x)
        x = self.conv3(x)
        x = nn.ReLU()(x)
        x = self.conv4(x)
        x = nn.Sigmoid()(x) # Ajout de la fonction d'activation finale
        return x # Retourne la sortie du reseau
```

5.2.3 AutoEncodeur

On assemble ensuite des deux réseau en un seul :

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Move your model to the device (GPU if available, else CPU)
model = Autoencoder().to(device)
```

5.3 Graphe d'évolution des métriques de l'entraînement loss et accuracy pour les datasets d'entraînement et de test

Le code pour l'entraînement et le test ainsi que pour le calcul des métriques et quasi identique aux exercices précédents.

5.3.1 Loss

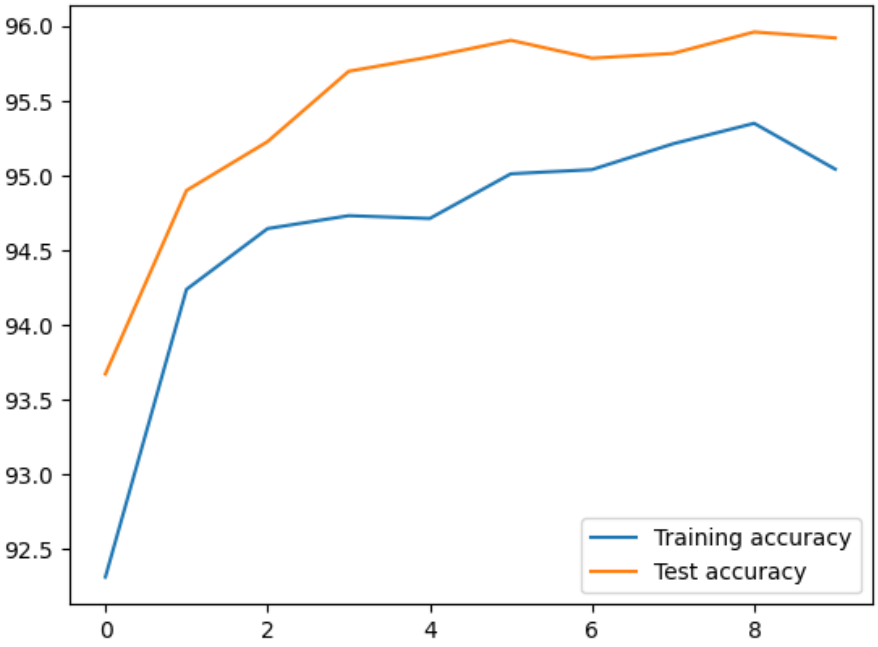

FIGURE 5.1 – Loss

5.3.2 Accuracy

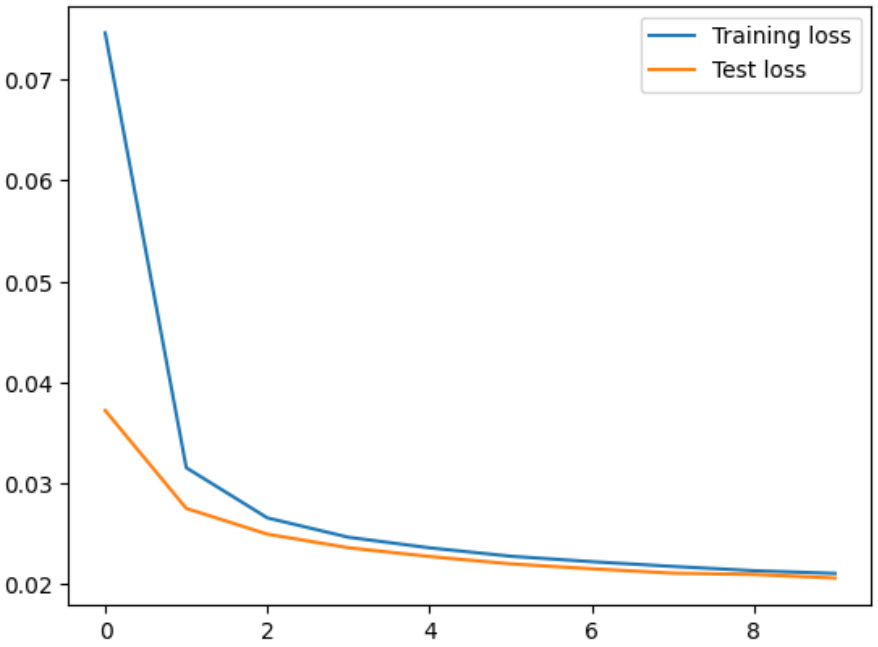

FIGURE 5.2 – Accuracy

On remarque sur les deux courbes que les valeurs de test sont meilleurs que les valeurs d'entraînement ce qui montre que nous n'avons pas d'overfitting et montre la performance du modele.

5.4 Comparaisons de 10 images originales et reconstituées

FIGURE 5.3 – 10 images originales et reconstituées

On vois que les image sont plutôt bien reconstituées même si les boucles on tendance à rester bouchés.

Denoising AE for MNIST

6.1 Introduction

Dans cette exercice, nous allons utiliser notre auto-encodeur pour dé-bruiter des images. Pour cela, nous allons entraîner notre auto-encodeur avec des images de chiffres puis d'entraîner notre modèle avec ces images. Ensuite nous allons ajouter du bruit sur l'ensemble des images représentant des chiffres. La dernière étape de notre programme et de notre auto-encodeur est de retrouver les images d'origines a partir des images bruités.

6.2 Explications du programme

6.2.1 Les constantes

```
batch_size = 128
num_classes = 10
epochs= 10
learning_rate = 0.001
encoding_dim = 32
noise_factor= 0.5
seuil = 0.5
```

Pendant la phase de conception du programme, nous avons choisit 5 epochs pour réduire le temps d'exécution assez long mais pas non plus trop court pour pouvoir analyser nos résultats. Une fois le programme terminé, nous avons augmenter ce nombre jusqu'à obtenir un résultats qui n'évolue plus ou presque plus. Une 'epoch' représente un passage complet sur l'ensemble du jeu de données d'entraînement

La constante noise_factor nous permet de fixer un seuil de bruitage, plus celui-ci est élevé, plus l'image sera bruité.

Nos images bruités ne sont plus binarisés, ils nous faut donc choisir un seuil pour savoir le pixel sera blanc ou noir dans l'image dé-bruité par l'auto-encodeur.

6.2.2 Chargement des données

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
mnist_train_set = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
mnist_test_set = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
mnist_train_loader = torch.utils.data.DataLoader(mnist_train_set, batch_size=batch_size, shuffle=True)
mnist_test_loader = torch.utils.data.DataLoader(mnist_test_set, batch_size=batch_size, shuffle=False)
```

Nos images de nombre proviennent du dataset MNIST et nous les séparons en deux dataset distinct. Le premier est utilisé pour l'entraînement de notre auto-encodeur, le second pour réaliser la phase de test. Pour la phase de test, nous ne mélangeons pas les données pour pouvoir avoir l'ensemble des nombres dans l'ordre pour le rendu final.

6.2.3 L'Auto-encodeur

```
class Autoencoder(nn.Module): # Autoencoder
```

```
def __init__(self): # Constructeur
    super(Autoencoder, self).__init__() # Heritage de la classe Module
    self.encoder = nn.Sequential( # Encodeur
        nn.Conv2d(1, 32, kernel_size=3, padding=1), # Convolutions 3x3 avec 32 filtres
        nn.ReLU(), # Fonction d'activation ReLU
        nn.MaxPool2d(2, 2), # Max pooling 2x2
        nn.Conv2d(32, 32, kernel_size=3, padding=1), # Convolutions 3x3 avec 32 filtres
        nn.ReLU(), # Fonction d'activation ReLU
        nn.MaxPool2d(2, 2) # Max pooling 2x2
    )
    self.decoder = nn.Sequential( # Decodeur
        nn.Conv2d(32, 32, kernel_size=3, padding=1), # Convolutions 3x3 avec 32 filtres
        nn.ReLU(), # Fonction d'activation ReLU
        nn.UpsamplingNearest2d(scale_factor=2), # Upsampling 2x2
        nn.Conv2d(32, 32, kernel_size=3, padding=1), # Convolutions 3x3 avec 32 filtres
        nn.ReLU(), # Fonction d'activation ReLU
        nn.UpsamplingNearest2d(scale_factor=2), # Upsampling 2x2
        nn.Conv2d(32, 1, kernel_size=3, padding=1), # Convolutions 3x3 avec 1 filtre
        nn.Sigmoid() # Fonction d'activation Sigmoid
    )

def forward(self, x): # Propagation avant
    x = self.encoder(x) # Encodeur
    x = self.decoder(x) # Decodeur
    return x # Retourne la sortie du decodeur
```

Cette classe représente notre auto-encodeur spécialisé dans le dé-bruitage d'image représentant des nombres blancs sur fond noir.

6.2.4 Ajout du bruit sur les images

```
img, _ = data
img = torch.clamp(img, 0., 1.)
noisy_img = img + torch.randn(img.size()) * noise_factor
noisy_img = torch.clamp(noisy_img, 0., 1.)
```

On récupère l'image du nombre dans data, puis nous vérifions que la valeurs de ces pixels soit bien compris entre 0 et 1 avec la fonction "torch.clamp". Nous ajoutons ensuite le bruit a l'image original, pour cela nous créons un tensor de bruit que nous obtenons en multipliant notre noise_factor avec la taille de notre image. Après l'ajout du bruit, nous réutilisons la fonction "torch.clamp" pour s'assurer que les pixels bruiter reste bien entre 0 et 1.

6.2.5 Initialisation du réseau de neurones et choix de l'optimiseur

```
AE = Autoencoder() # Initialisation du reseau de neurones de l'autoencodeur

criterion = nn.BCELoss() # Fonction de cout (entropie croisee binaire)
optimizer = optim.Adam(AE.parameters(), lr=learning_rate) # Choix de l'optimiseur Adam
```

6.2.6 Le programme

```
list_loss_train = [] # Liste de loss pour l'entrainement
list_accuracy_train = [] # Liste d'accuracy pour l'entrainement
list_loss_test = [] # Liste de loss pour le test
list_accuracy_test = [] # Liste d'accuracy pour le test
seuil = 0.5 # Seuil pour la binarisation

for epoch in range(epochs): # On parcourt les epochs
    AE.train() # Mode entrainement
    sous_list_loss_train = [] # Liste de loss pour l'entrainement
    sous_valeur_accuracy_train = 0 # Valeur d'accuracy pour l'entrainement
    for data in mnist_train_loader: # On parcourt les donnees d'entrainement
        img, _ = data # On recupere l'image
        img = torch.clamp(img, 0., 1.) # On borne l'image entre 0 et 1
        noisy_img = img + torch.randn(img.size()) * noise_factor # On ajoute du bruit a l'image
        noisy_img = torch.clamp(noisy_img, 0., 1.) # On borne l'image entre 0 et 1
        optimizer.zero_grad() # On met a zero les gradients
        output = AE(noisy_img) # On calcule la sortie du reseau
        loss = criterion(output, img) # On calcule la loss
        loss.backward() # On calcule les gradients
        optimizer.step() # On met a jour les poids

        sous_list_loss_train.append(loss.item()) # On ajoute la loss a la liste
        sous_valeur_accuracy_train += ((output > seuil) == (img > seuil)).sum().item() # On ajoute
            l'accuracy a la valeur

    total_pixels = img.view(img.size(0), -1).size(1) # Nombre de pixels dans l'image
    list_accuracy_train.append(sous_valeur_accuracy_train / (len(mnist_train_set) * total_pixels)) #
        On ajoute l'accuracy a la liste
    list_loss_train.append(np.mean(sous_list_loss_train)) # On ajoute la loss a la liste

    AE.eval() # Mode evaluation
    with torch.no_grad(): # Pas de calcul de gradient
        sous_list_loss_test = [] # Liste de loss pour le test
        sous_valeur_accuracy_test = 0 # Valeur d'accuracy pour le test
        for data in mnist_test_loader: # On parcourt les donnees de test
            img, _ = data # On recupere l'image
            img = torch.clamp(img, 0., 1.) # On borne l'image entre 0 et 1
            noisy_img = img + torch.randn(img.size()) * noise_factor # On ajoute du bruit a l'image
            noisy_img = torch.clamp(noisy_img, 0., 1.) # On borne l'image entre 0 et 1
            output = AE(img) # On calcule la sortie du reseau
            loss = criterion(output, img) # On calcule la loss
            sous_list_loss_test.append(loss.item()) # On ajoute la loss a la liste
            sous_valeur_accuracy_test += ((output > seuil) == (img > seuil)).sum().item() # On ajoute
                l'accuracy a la valeur

        total_pixels = img.view(img.size(0), -1).size(1) # Nombre de pixels dans l'image
        list_accuracy_test.append(sous_valeur_accuracy_test / (len(mnist_test_set) * total_pixels)) #
            On ajoute l'accuracy a la liste
        list_loss_test.append(np.mean(sous_list_loss_test)) # On ajoute la loss a la liste


    print('epoch [{}/{}], loss train:{:.4f}, loss test:{:.4f}, accuracy train:{:.4f}, accuracy
        test:{:.4f}'.format(epoch + 1, epochs, list_loss_train[-1], list_loss_test[-1],
        list_accuracy_train[-1], list_accuracy_test[-1]))
```

6.3 Explications des résultats

6.3.1 Loss

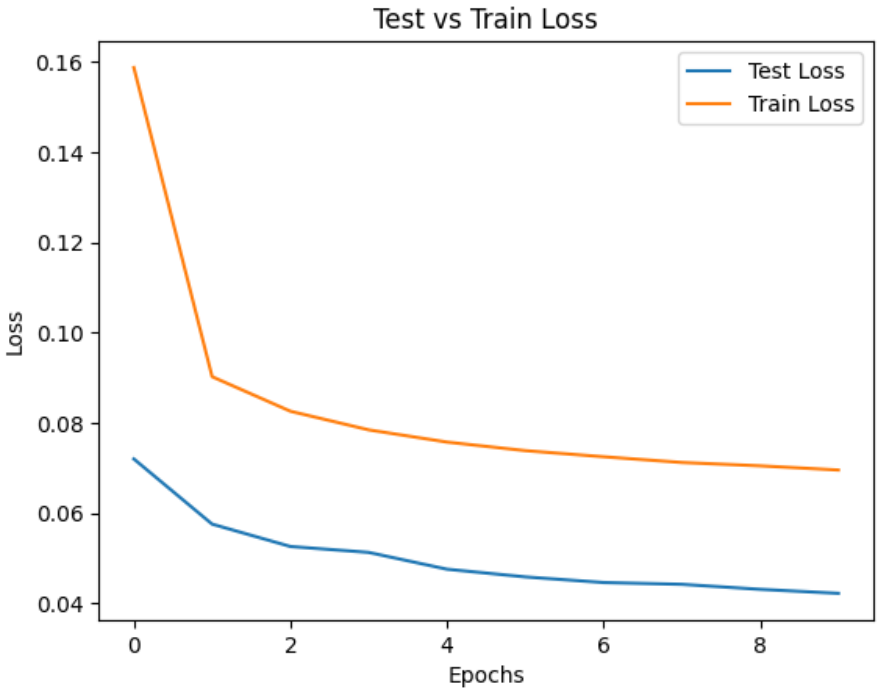

FIGURE 6.1 – Loss

La courbe orange représente l'évolution de la "loss" pour le jeu d'entraînement et la courbe bleu pour le jeu de test. On remarque pour les deux courbes une diminution qui correspond à la période d'apprentissage, puis une période de stagnation qui montre que l'auto-encodeur converge vers une solution. Plus la "loss" est faible, plus l'auto-encodeur a réussi a reproduire l'image d'origine.

6.3.2 Accuracy

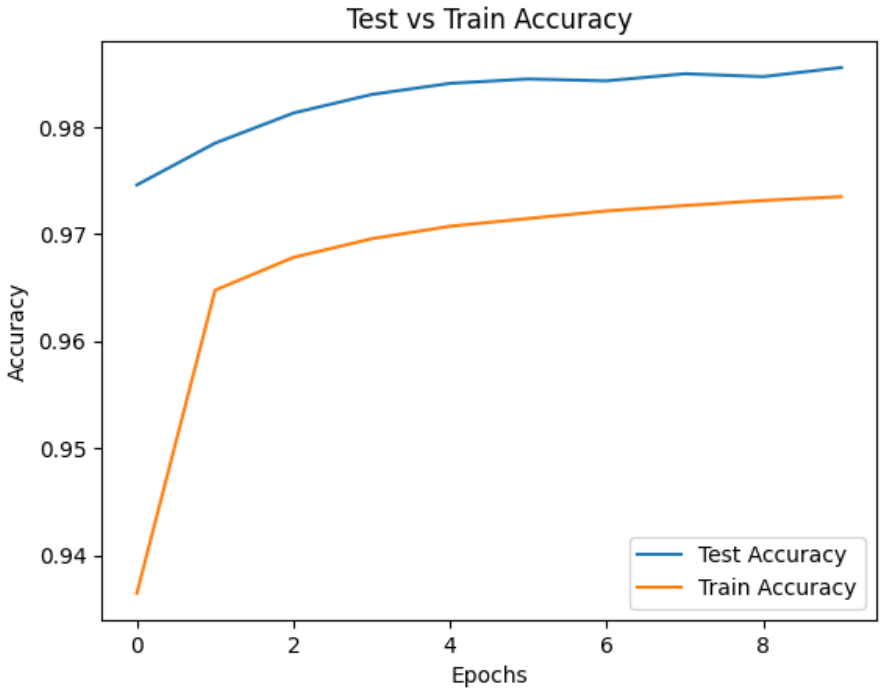

FIGURE 6.2 – Accuracy

La courbe orange représente l'évolution de l'"accuracy" pour le jeu d'entraînement et la courbe bleue pour le jeu de test. On remarque pour les deux courbes une augmentation qui correspond à la période d'apprentissage, puis une période de stabilisation qui montre que l'auto-encodeur converge vers une solution. Plus l'"accuracy" est élevée, plus l'auto-encodeur a réussi à reproduire fidèlement l'image d'origine.

6.3.3 Comparaisons de 10 images originales et reconstituées

FIGURE 6.3 – 10 images originales, avec du bruits et reconstituées

On remarque les nombres obtenus après reconstitutions sont fidèles aux images d'origines malgré l'ajout de bruit. Les images reconstituées sont légèrement flou mais cela est du à la compression de l'auto-encodeur. Seul le trou dans le chiffre 6 est légèrement visible, mais cela est du au petit trou de l'image d'origine et l'ajout de bruit qui a du le faire disparaître lors des phases de traitement.

Conclusion

En conclusion de ce projet, nous avons pu mettre en place un système de reconnaissance de caractères manuscrits. Nous avons pu voir que les réseaux de neurones sont des outils très puissants pour ce genre de problèmes. Nous avons également pu voir que les réseaux de neurones sont très sensibles à l'architecture. En effet, nous avons pu obtenir des résultats très différents en changeant l'architecture du réseau de neurones en particulier sur l'agencement des différentes couches de convolution et de pooling. Nous avons rencontré quelques difficulté à la conception de l'architecture du réseau de neurone notamment pour configurer les bonne tailles dans les couches de convolution.