TP : Implementation d'un MLP en PyTorch pour MNIST

Master GBM

March 27, 2025

1 Introduction

Dans ce TP, vous allez apprendre a implementer une Multi-Layer Perceptron (MLP) avec PyTorch pour classifier les chiffres manuscrits du dataset MNIST.

En particulier, nous allons implémenter l'architecture suivante:

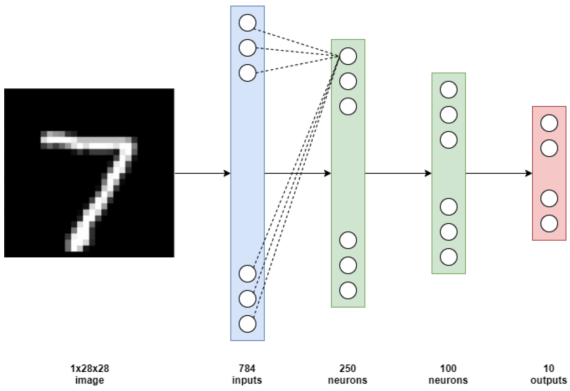

Figure 1: Architecture du MLP

—

2 Pre-requis

Avant de commencer, assurez-vous que :

- Vous avez installé PyTorch.
- Vous avez installé les bibliothèques nécessaires (`torch`, `torchvision`, etc.).

—

3 Chargement du dataset MNIST

Le dataset MNIST contient des images de chiffres manuscrits (0-9), chaque image ayant une taille de 28×28 pixels. Voici le code pour charger ce dataset :

```
import torch
from torchvision import datasets, transforms

# Transformation des donnees : normalisation et
    conversion en tenseurs
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Chargement des datasets d'entrainement et de test
train_dataset = datasets.MNIST(root='./data', train=
    True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=
    False, download=True, transform=transform)

# Creation des DataLoaders pour gerer les batches
train_loader = torch.utils.data.DataLoader(dataset=
    train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=
    test_dataset, batch_size=64, shuffle=False)
```

—

4 Definition de la MLP

La MLP est un reseau de neurones avec plusieurs couches entierement connectees. Voici comment definir une MLP simple :

```
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(, )  # Premiere couche
            cachee
        self.fc2 = nn.Linear(, )      # Deuxieme
            couche cachee
        self.fc3 = nn.Linear(, )       # Couche de
            sortie

    def forward(self, x):
        batch_size = x.shape[0]
        x = x.view(batch_size, )            # Flatten
            de l'image (28x28 -> 784)
```

```
        # Activation ReLU sur la premiere couche
        # Activation ReLU sur la deuxieme couche
        # Pas d'activation sur la couche de sortie (
            logits)
        return y_pred
```

—

5 Entrainement du modele

Pour entrainer le modèle, nous avons besoin d'une fonction de cout (CrossEntropyLoss) et d'un optimiseur (Adam). Voici le code d'entrainement :

6 Rappels

La fonction **softmax** est utilisée pour transformer les sorties d'un modèle en une distribution de probabilité. Elle est définie comme suit :

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

où :

- $\mathbf{x}$ est le vecteur des sorties du modèle (appelé logits),
- x_i est la valeur associée à la classe i,
- j parcourt toutes les classes.

Cette fonction garantit que :

- Toutes les valeurs sont comprises entre 0 et 1,
- La somme des probabilités pour toutes les classes est égale à 1.

La transformation en une distribution de probabilité permet d'utiliser des fonctions de perte adaptées, comme la log-vraisemblance négative (*Negative Log Likelihood Loss*). Cette fonction de perte est définie par :

$$\text{NLL}(\hat{\mathbf{y}}, y) = -\log(\text{softmax}(\hat{\mathbf{y}})[y])$$

où :

- $\hat{\mathbf{y}}$ est le vecteur des logits produit par le modèle,
- y est l'indice de la classe correcte.

Supposons que les sorties du modèle soient :

$$\hat{\mathbf{y}} = [5, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

Après application de la fonction softmax :

$$\text{softmax}(\hat{\mathbf{y}}) = [0.8585, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157]$$

Si l'étiquette correspond à la classe 0, la perte sera :

$$\text{NLL}(\hat{\mathbf{y}}, 0) = -\log(0.8585) = 0.153$$

Si l'étiquette correspond à la classe 5, la perte sera :

$$\text{NLL}(\hat{\mathbf{y}}, 5) = -\log(0.0157) = 4.154$$

Ainsi, lorsque le modèle attribue une probabilité élevée à la classe correcte, la perte diminue.

6.1 Entrainement typique d'un modèle en Pytorch

1. **Boucle sur les mini-batches** : Les données d'entraînement sont chargées par le `DataLoader`, qui fournit des mini-batches composés d'images et de labels.
2. **Réinitialisation des gradients** : Les gradients accumulés des étapes précédentes sont réinitialisés avec `optimizer.zero_grad()`.
3. **Passage avant (forward pass)** : Les données du mini-batch sont passées dans le modèle pour obtenir les prédictions (`outputs = model(images)`).
4. **Calcul de la perte** : La fonction de perte (`criterion`) compare les prédictions (`outputs`) aux labels réels (`labels`) pour mesurer l'erreur (`loss = criterion(outputs, labels)`).
5. **Passage arrière (backward pass)** : La rétropropagation calcule les gradients de la perte par rapport aux paramètres du modèle avec `loss.backward()`.
6. **Mise à jour des poids** : L'optimiseur utilise les gradients pour ajuster les poids du modèle avec `optimizer.step()`.

```
import torch.optim as optim

# Initialisation du modele, de la fonction de cout et
   de l'optimiseur
model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
num_epochs =
# Boucle d'entrainement sur 5 epoques
for epoch in range(num_epochs):
    for images, labels in train_loader:
         # Reinitialisation des gradients
         # Forward pass
         # Calcul de la perte
         # Backward pass (calcul des gradients)
         # Mise a jour des poids
    print(f'Epoque{epoch+1},Perte:{loss.item()}')
```

—

7 Évaluation du modèle

Après l'entraînement, évaluez la performance du modèle sur le dataset de test :

```
correct = 0
total = 0

with torch.no_grad(): # Desactivation du calcul des
    gradients pour l'evaluation
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Precision:{100*correct/total}%')
```

—

8 Pour aller plus loin

Dans ce TP, vous avez appris a implementer une MLP pour classifier les chiffres manuscrits du dataset MNIST. Voici quelques idees pour aller plus loin :

- Ajouter plus de couches cachées ou augmenter leur taille.
- Tester differentes fonctions d'activation comme LeakyReLU ou Sigmoid.
- Experimenter avec differents optimiseurs comme SGD ou RMSprop.
- Appliquer ce modele a un autre dataset comme CIFAR-10.