Special Topics

MSDA 3440-01-F23

Project 1

Code Explanation

Submitted By:

Kabi Raj Tiruwa (C70292740)

Rahul Thapa Magar (C70293419)

Step 1: Importing Libraries

In this part all the important libraries were installed and imported.

```python
import os
# !pip install torch
# !pip install torchvision
# !pip install matplotlib
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision as tv
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
```

✓ 16.2s

Step 2:

"get_loaders_MNIST" function prepares data loaders for the MNIST dataset using PyTorch's torchvision library which will be using for our machine learning part. In this function we:

- The function takes an optional batch_size argument, with a default value of 100, specifying the number of data samples processed in each iteration.
- It applies data transformations to the MNIST dataset, converting images to PyTorch tensors and normalizing their values.
- It creates two dataset objects: one for training and one for testing. These datasets are stored in the "./data/" directory and include the specified transformations.
- For the training dataset, a data loader is created with settings for batch size, shuffling, and parallel data loading. The last batch is dropped if its size is less than the specified batch size.
- A similar data loader is created for the test dataset, but shuffling is disabled since there's no need for randomness during testing.
- The function returns the training and testing data loaders, which can be used for efficiently loading and processing data in machine learning or deep learning tasks.

```python
def get_loaders_MNIST(batch_size=100):
    transforms = tv.transforms.Compose([
        tv.transforms.ToTensor(),
        tv.transforms.Normalize((0.1307,), (0.3081,))
    ])
    train_data = tv.datasets.MNIST(root="./data/", train=True, download=True, transform=transforms)
    test_data = tv.datasets.MNIST(root="./data/", train=False, download=True, transform=transforms)

    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, drop_last=True, num_workers=2)
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)
    return train_loader, test_loader
```

✓ 0.0s

Step 3:

This code makes a scatter plot of data points in 2D space, color-coded by class, using the Python function Plot_Graph. Here's a quick rundown of what it does:

- The function accepts feature and target data as input, as well as an epoch number and a directory path to save the plot to.
- It specifies a set of colors as well as a set of class labels (usually representing numbers 0 to 9).
- The code creates an interactive charting environment.
- It iterates through each class, extracting data points for that class and plotting them in 2D space as a scatter plot with the provided color.
- The plot also contains a legend, and the title involves the epoch number.

```
def Plot_Graph(feature, targets, epoch, save_path):
    color = ["red", "black", "yellow", "green", "pink", "gray", "lightgreen", "orange", "blue", "teal"]
    cls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    plt.ion()
    plt.clf()
    for j in cls:
        mask = targets == j
        feature_ = feature[mask].numpy()
        x = feature_[:, 1]
        y = feature_[:, 0]
        plt.plot(x, y, ".", color=color[j])

    plt.legend(cls, loc="upper right")
    plt.title("epoch={}".format(str(epoch+1)))
    plt.savefig('{}/{}.jpg'.format(save_path, epoch+1))
    plt.show()
✓ 0.0s
```

Step 4:

This code defines the Model_Accuracy function, which evaluates the accuracy of a neural network model on a test dataset. It performs this by running the test data through the model, comparing projected labels to actual labels, and then estimating calculating the accuracy as a percentage. As inputs, the function accepts a model to be assessed, a classification network (usually with log-probabilities), and a test data loader.

```
def Model_Accuracy(net, arcnet, test_loader):
    acc = 0
    for i, (x, y) in enumerate(test_loader):
        latent_out = net(x)
        arc_out = torch.log(arcnet(latent_out))
        value = torch.argmax(arc_out, dim=1)
        acc += torch.sum((value == y).float())
    print('test accuracy = ', acc.item() / len(test_loader.dataset))
✓ 0.0s
```

Step 5:

This code creates the "Network" PyTorch neural network class for processing picture input and generating a lower-dimensional latent representation. Convolutional layers, batch normalization, and a linear layer are all included. The model is intended for feature extraction from images, with the "latent_dim" parameter controlling the latent representation dimension.

The neural network architecture is defined in two parts within the constructor:

- "self.cnn_layers" contains a sequence of convolutional and batch normalization layers for processing input picture data.
    - Convolutional Layer 1 consists of 64 filters, a 3x3 kernel, stride 2, and padding 1.
    - Activation of the Batch Normalization Layer and the ReLU.
    - Convolutional Layer 2: 256 similar-parameter filters.
    - Activation of the Batch Normalization Layer and the ReLU.
    - Convolutional Layer 3 consists of 256 filters stride 1 with normalizing and ReLU.
    - Convolutional Layer 4: 64 normalized and ReLU filters.
    - Convolutional Layer 5 consists of 16 filters, a 3x3 kernel, stride 2 and a ReLU.
    - "self.linear_layer" specifies a fully connected linear layer that flattens the output of convolutional layers and maps it to "latent_dim."
- The forward pass is provided via the "forward(self, xs)" method:
    - "cnn_out" stores the output of the convolutional layers that were applied to the input data.
    - "flatten" flattens "cnn_out" into a vector.
    - By running the flattened vector through the linear layer, "latent_out" computes the final latent representation.

```python
class Network(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(1, 64, 3, 2, 1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 256, 3, 2, 1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, 3, 1, 1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 64, 3, 1, 1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 16, 3, 2, 1),
            nn.ReLU()
        )
        self.linear_layer = nn.Linear(16*4*4, latent_dim)

    def forward(self, xs):
        cnn_out = self.cnn_layers(xs)
        flatten = cnn_out.reshape(-1, 16*4*4)
        latent_out = self.linear_layer(flatten)
        return latent_out
```

✓ 0.0s

Step 6:

This ArcNet class is utilized in neural networks for face recognition or similar tasks, with the goal of improving embedding quality and maximizing separation between distinct classes.

- The code defines the "ArcNet" class, which is used to implement the ArcFace loss function, which is commonly used in facial recognition.
- The class constructor accepts parameters such as the number of classes, the dimension of the input embeddings, and the scaling factor ("s") and margin ("m").
- The "forward" approach performs multiple mathematical operations on input embeddings:
- Normalize the weight matrix and the input embeddings.
- Calculate the cosine similarity between embeddings and class centers while taking the scaling factor into account.
- Calculate sine values and adjust cosine similarity to the specified margin.
- Exponentiate the cosine values.
- For each example, add the sum of the scaled cosine values.
- As a ratio of specific exponential terms, compute the ArcFace output ("arcout").

```python
class ArcNet(nn.Module):
    def __init__(self, num_classes, latent_dim, s=20, m=0.1):
        super().__init__()
        self.s = s
        self.m = torch.tensor(m)
        self.w = nn.Parameter(torch.rand(latent_dim, num_classes))

    def forward(self, embedding):
        embedding = F.normalize(embedding, dim=1)
        w = F.normalize(self.w, dim=0)
        cos_theta = torch.matmul(embedding, w) / self.s
        sin_theta = torch.sqrt(1.0 - torch.pow(cos_theta, 2))
        cos_theta_m = cos_theta * torch.cos(self.m) - sin_theta * torch.sin(self.m)
        cos_theta_scaled = torch.exp(cos_theta * self.s)
        sum_cos_theta = torch.sum(torch.exp(cos_theta * self.s), dim=1, keepdim=True) - cos_theta_scaled
        top = torch.exp(cos_theta_m * self.s)
        arcout = top / (top + sum_cos_theta)
        return arcout
```

✓ 0.0s

Step 7:

- It is used in machine learning training to implement early stopping.
- It keeps track of a performance metric, usually test accuracy.
- It determines whether the test accuracy improves by a specific delta.
- When the number of successive non-improvements exceeds a predefined patience level, early stopping is initiated.
- The class keeps track of the best observed performance.
- It tracks the number of times the performance metric does not improve in a row.
- A flag is used to signal when the training process should be terminated.
- Early halting is a method used to avoid overfitting and save time during training.

```
class EarlyStopping:
    def __init__(self, patience=5, delta=0):
        self.patience = patience
        self.counter = 0
        self.best_score = None
        self.delta = delta
        self.stop = False

    def __call__(self, test_accuracy):
        score = test_accuracy

        if self.best_score is None:
            self.best_score = score
        elif score < self.best_score + self.delta:
            self.counter += 1
            print(f"EarlyStopping counter: {self.counter} out of {self.patience}")
            if self.counter >= self.patience:
                self.stop = True
        else:
            self.best_score = score
            self.counter = 0
```

✓ 0.0s

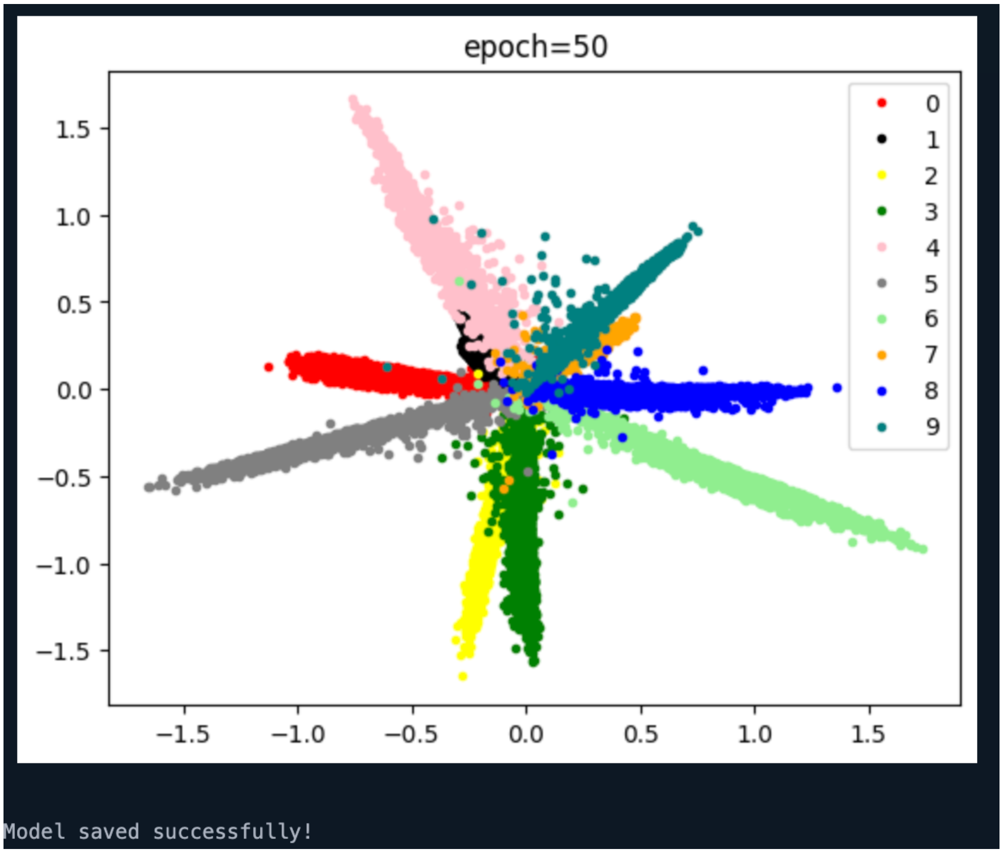
Step 8:

- It sets up model parameters such as latent dimensions and the number of classes.
- The model architecture, loss function, optimizer, and data loaders are all set up by the code.
- It enters a training loop for the number of epochs supplied.
- It trains the model, calculates training loss and accuracy, and evaluates the model's correctness on a test dataset at each epoch.
- Early stopping is used to halt training if test accuracy does not improve by a specific threshold after a set number of epochs.
- After each epoch, the embedded model is shown, and the trained model is stored to a file.
- The main function is called to complete the training and assessment procedure on the MNIST dataset.

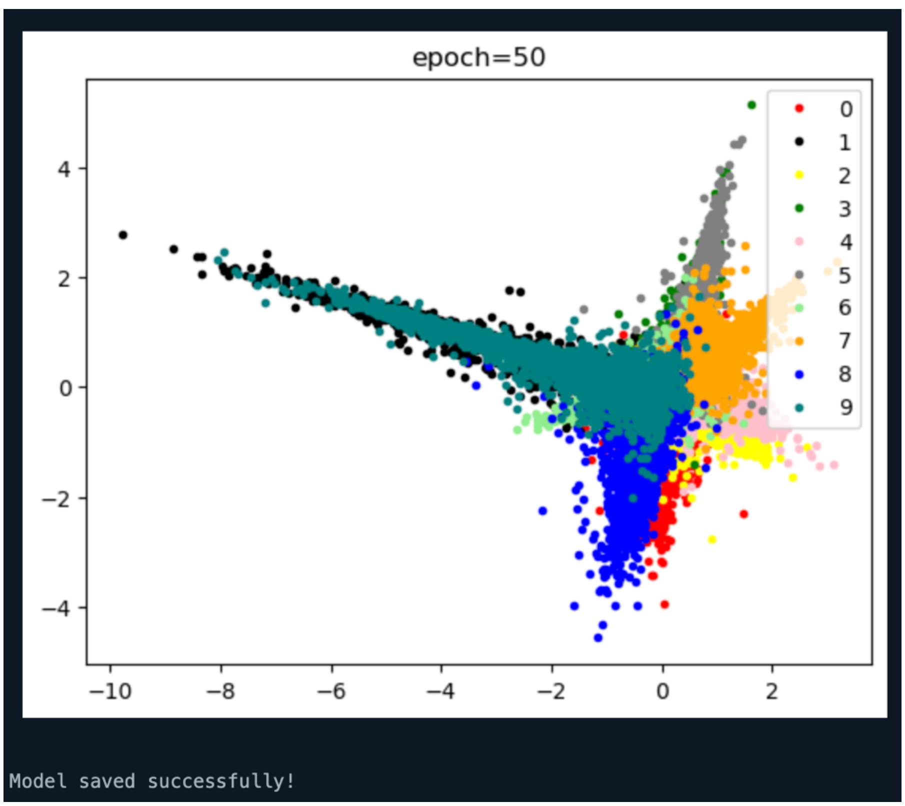
```python
def main():
    latent_dim = 3
    num_classes = 10
    net = Network(latent_dim)
    arcnet = ArcNet(num_classes, latent_dim)
    arcloss = nn.NLLLoss(reduction="mean")
    optimizerarc = torch.optim.SGD([{'params': net.parameters()}, {'params': arcnet.parameters()}], lr=0.01, momentum=0.9, weight_decay=0.0005)
    save_pic_path = "./Images"
    train_loss = []
    test_accuracy = []
    num_epochs = 50
    train_loader, test_loader = get_loaders_MNIST()

    os.makedirs(save_pic_path, exist_ok=True)

    early_stopping = EarlyStopping(patience=5, delta=0.01)  # Stop if accuracy hasn't improved by 1% in the last 10 epochs

    for epoch in range(num_epochs):
        net.train()
        total_loss = 0
        total_correct = 0
        total_samples = 0
        embeddings = []
        targets = []

        for i, (x, y) in enumerate(train_loader):
            latent_out = net(x)
            arc_out = torch.log(arcnet(latent_out))
            loss = arcloss(arc_out, y)
            optimizerarc.zero_grad()
            loss.backward()
            optimizerarc.step()

            total_loss += loss.item()
            predictions = torch.argmax(arc_out, dim=1)
            total_correct += torch.sum(predictions == y).item()
            total_samples += y.size(0)

            embeddings.append(latent_out)
            targets.append(y)

        train_accuracy = total_correct / total_samples
        train_loss.append(total_loss / len(train_loader))
        test_accuracy_val = Model_Accuracy(net, arcnet, test_loader)
        early_stopping(test_accuracy_val)

        if early_stopping.stop:
            print("Early stopping triggered!")
            break


        print('Epoch [{}/{}], Training Loss: {:.4f}, Training Accuracy: {:.2f}%'.format(epoch + 1, num_epochs, total_loss / len(train_loader), train_accuracy * 100))

        # Visualizing the embeddings
        all_embeddings = torch.cat(embeddings, 0)
        all_targets = torch.cat(targets, 0)
    Plot_Graph(all_embeddings.data.cpu(), all_targets.data.cpu(), epoch, save_pic_path)

    # Saving the trained model
    PATH = "model.pth"
    torch.save(net.state_dict(), PATH)
    print("Model saved successfully!")


# Run the main function
main()
```

216m 41.1s   Python

```
test accuracy =  0.9593
Epoch [1/50], Training Loss: 3.4062, Training Accuracy: 81.89%
test accuracy =  0.9677
Epoch [2/50], Training Loss: 3.2741, Training Accuracy: 96.79%
test accuracy =  0.9701
Epoch [3/50], Training Loss: 3.2651, Training Accuracy: 97.63%
test accuracy =  0.9797
Epoch [4/50], Training Loss: 3.2620, Training Accuracy: 97.81%
test accuracy =  0.9745
Epoch [5/50], Training Loss: 3.2592, Training Accuracy: 98.22%
test accuracy =  0.9833
Epoch [6/50], Training Loss: 3.2562, Training Accuracy: 98.45%
test accuracy =  0.9803
Epoch [7/50], Training Loss: 3.2554, Training Accuracy: 98.56%
```



Distributions of the features in MNIST Dataset.



Distributions of the features in CIFAR Dataset.