Documentation of PyTorch MNIST Example Code

1 Introduction

This document provides an overview and explanation of a Python script used for loading the MNIST dataset, defining a neural network model, and training and testing the model using PyTorch. The MNIST dataset is a classic benchmark dataset for image classification tasks consisting of handwritten digits.

2 Imports

The script begins by importing the necessary libraries:

Listing 1: Imports

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

- `torch`: The core PyTorch library for tensor operations and deep learning.
- `datasets` from `torchvision`: Provides access to common datasets like MNIST.
- `ToTensor` from `torchvision.transforms`: Converts images to PyTorch tensors.
- `matplotlib.pyplot`: Used for visualizing images and results.

3 Data Loading

The MNIST dataset is loaded for training and testing. The dataset is transformed into tensors using the `ToTensor` transform to prepare it for model training.

Listing 2: Loading MNIST Data

```
# Load the MNIST dataset for training
training_data = datasets.MNIST(root=".", train=True, download
    =True, transform=ToTensor())

# Load the MNIST dataset for testing
test_data = datasets.MNIST(root=".", train=False, download=
    True, transform=ToTensor())
```

3.1 Data Visualization

A 5x5 grid of random samples from the training dataset is visualized to understand the data distribution.

Listing 3: Visualizing Data

```
figure = plt.figure(figsize=(8, 8))
cols, rows = 5, 5

# Loop through and plot random samples
for i in range(1, cols * rows + 1):
    # Randomly select an image from the training data
    sample_idx = torch.randint(len(training_data), size=(1,))
        .item()
    img, label = training_data[sample_idx]
    # Add subplot to the figure
    figure.add_subplot(rows, cols, i)
    plt.axis("off")  # Hide axis labels
    plt.imshow(img.squeeze(), cmap="gray")  # Display image
        in grayscale
plt.show()
```

4 Data Loaders

Data loaders are created to manage mini-batches of the dataset, which is essential for efficient training and evaluation.

Listing 4: Creating DataLoaders

```python
from torch.utils.data import DataLoader

# Create data loaders for training and testing datasets
loaded_train = DataLoader(training_data, batch_size=64, shuffle=True)
loaded_test = DataLoader(test_data, batch_size=64, shuffle=True)
```

4.1 Explanation

- `DataLoader`: Provides an iterable over the dataset with support for batching and shuffling.
- `batch_size=64`: Specifies the number of samples per batch.
- `shuffle=True`: Shuffles the data at every epoch to improve training.

5 Model Definition

A simple feedforward neural network is defined using PyTorch's `nn.Module`. The network consists of three linear layers with ReLU activations.

Listing 5: Defining Neural Network

```python
from torch import nn

# Define the neural network architecture
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()  # Flatten the 2D images into 1D vectors
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),  # First fully connected layer
            nn.ReLU(),              # ReLU activation
            nn.Linear(512, 512),    # Second fully connected layer
            nn.ReLU(),              # ReLU activation
            nn.Linear(512, 10),     # Output layer with 10 classes
        )

    def forward(self, x):
        x = self.flatten(x)       # Flatten the input tensor
        logits = self.linear_relu_stack(x)  # Forward pass through the network
```

```
        return logits

# Instantiate and print the model
model = NeuralNetwork()
print(model)
```

6 Training and Testing Functions

Functions for training and testing the model are defined. These functions handle the optimization and evaluation of the model.

Listing 6: Training Function

```
# Define loss function and optimizer
loss_function = nn.CrossEntropyLoss()  # Loss function for
    classification
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)  #
    Stochastic Gradient Descent optimizer

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        pred = model(X)              # Forward pass: Compute
            predicted values
        loss = loss_fn(pred, y)     # Compute the loss

        optimizer.zero_grad()        # Zero gradients before
            backward pass
        loss.backward()             # Backward pass: Compute
            gradient
        optimizer.step()            # Update model parameters

        if batch % 100 == 0:        # Print loss every 100
            batches
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5
                d}]")
```

Listing 7: Testing Function

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():  # Disable gradient computation for
        evaluation
```

```
        for X, y in dataloader:
            pred = model(X)                    # Forward pass
            test_loss += loss_fn(pred, y).item()  # Accumulate loss
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()  # Count correct predictions

    test_loss /= num_batches  # Average loss
    correct /= size  # Accuracy
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

7 Training the Model

The model is trained for 5 epochs, and its performance is evaluated after each epoch.

Listing 8: Training Loop

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(loaded_train, model, loss_function, optimizer)  # Train the model
    test(loaded_test, model, loss_function)  # Evaluate the model
print("Done!")
```

8 Conclusion

This script demonstrates the process of training a simple neural network on the MNIST dataset using PyTorch. It includes data loading, model definition, training, and testing. The provided functions and structure offer a solid foundation for understanding and experimenting with neural network training in PyTorch.