Documentation for MNIST Dataset Exploration and Model Development

Introduction

This project involves exploring and processing the MNIST dataset for building deep learning models. Various preprocessing techniques, data augmentation, and model architectures are employed to improve accuracy and robustness.

---

1. Dataset Overview

The MNIST dataset contains grayscale images of handwritten digits (0-9) with dimensions 28×2828 \times 28. It is divided into training (60,000 images) and test (10,000 images) sets.

---

2. Libraries Used

- **PyTorch**: Framework for building and training models.
- **Torchvision**: Dataset utilities and transformations.
- **Matplotlib**: Visualization.
- **NumPy**: Data manipulation.
- **Scikit-learn**: Metrics like precision and accuracy.

---

3. Data Loading and Preprocessing

1. **Loading MNIST Dataset:**

   - torchvision.datasets.MNIST is used to load the MNIST dataset.
   - root argument specifies the directory where the downloaded data will be stored.
   - train set to True loads the training dataset, False for testing.
   - download set to True downloads the dataset if not already present.
   - transform defines how the images will be preprocessed before feeding them to the network.

3.1 Transformations

Transformations applied:

- **Normalization**: Centering data to have mean 00 and standard deviation 11.
- **Random Augmentations**: Rotation, shearing, resizing, flipping, and brightness adjustments for data augmentation.
   - transforms.Compose combines multiple transformations into a single pipeline.

- transforms.ToTensor: Converts the PIL images (0-255) to PyTorch tensors (0.0-1.0).
- transforms.Normalize: Normalizes the pixel values based on the mean and standard deviation of the dataset (typically used for better convergence).

•

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

3.2 Data Loaders

Data is loaded using DataLoader, with batch size 64 for efficient mini-batch training.

- torch.utils.data.DataLoader creates iterators for accessing the dataset in batches.
- batch_size defines the number of images processed at once.
- shuffle set to True randomly shuffles the data during training (improves generalization).

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

---

4. Data Visualization

Batch Visualization

Visualize random batches of images and their labels.

- show_batch function iterates through a batch and displays the images and their corresponding labels using Matplotlib.

```
def show_batch(images, labels):
    fig, ax = plt.subplots(5, 5, figsize=(12, 12))
    for i in range(25):
        ax[i].imshow(images[i].numpy().squeeze(), cmap='gray')
        ax[i].set_title(f'Label: {labels[i].item()}')
        ax[i].axis('off')
    plt.tight_layout()
    plt.show()
```

---

5. Models

5.1 Multilayer Perceptron (MLP)

A fully connected neural network with three layers:

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

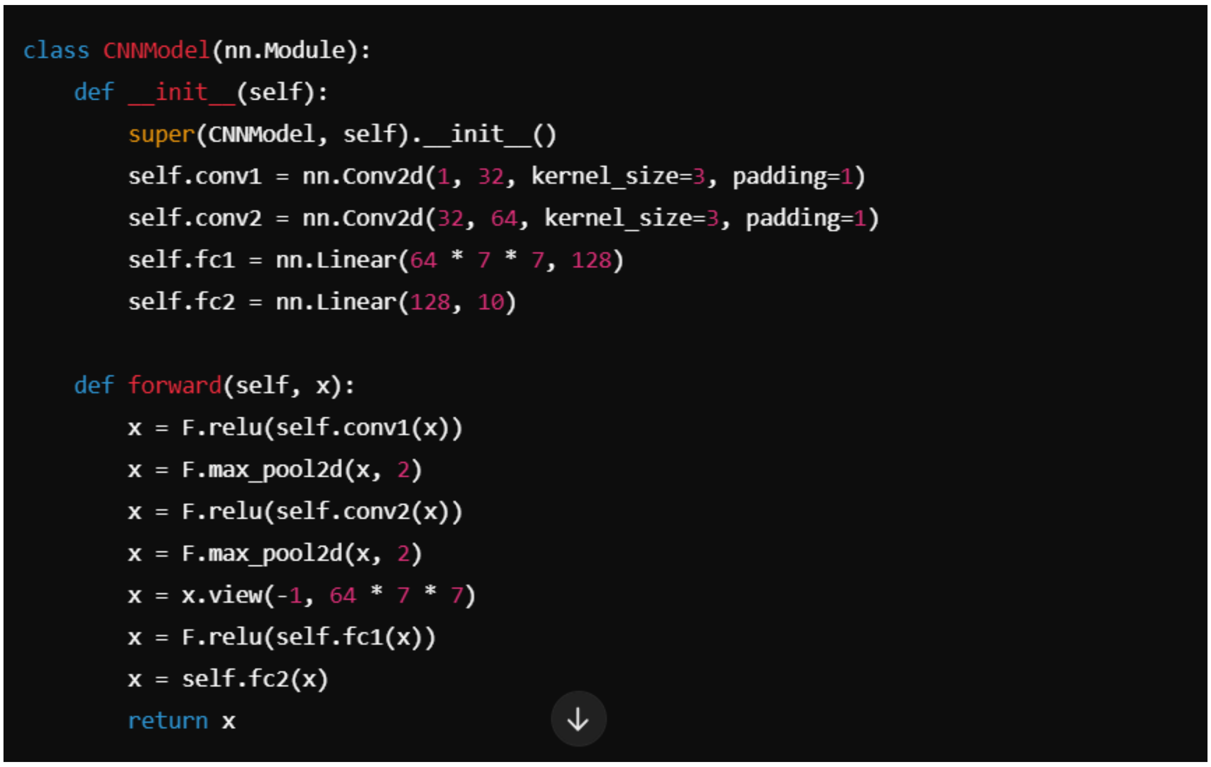
5.2 Advanced MLP

Includes Batch Normalization and Dropout for better generalization.

5.3 Convolutional Neural Network (CNN)

A simple CNN with two convolutional layers and fully connected layers.

1. **Multi-Layer Perceptron (MLP):**
   - A simple feed-forward neural network with fully connected layers.
   - nn.Linear defines a linear layer with a specified input and output size.
   - nn.ReLU is the activation function used between layers (introduces non-linearity).
   - The forward pass defines how data propagates through the network.
2. **Advanced MLP:**
   - Similar to the basic MLP but with additional features.
   - Batch normalization layers (nn.BatchNorm1d) are added after convolutional layers to improve training stability.
   - Dropout layer (nn.Dropout) is introduced for regularization (prevents overfitting).



```python
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

---

6. Training and Testing

6.1 Training Loop

The model is trained for a specified number of epochs, and the loss is printed.

1. **Training Function (train):**
   - Iterates through training epochs.
   - Calculates the loss for each batch using the chosen criterion (usually cross-entropy for classification).
   - Updates the model weights using the optimizer (e.g., Adam).
   - Prints the training loss after each epoch.
2. **Training different architectures:**
   - The code allows for training the basic MLP, advanced MLP, and LeNet (a convolutional neural network) by simply changing the model definition.
3. **Training Loop:**
   - The script defines a training loop that iterates for a specified number of epochs.
   - During each epoch, it trains the model using the training data and evaluates it on the test data.

```python
def train(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}')
```

6.2 Testing Loop

Calculates accuracy and precision on the test set.

Testing Function (test):

- Evaluates the model on the test set.
- Calculates accuracy and optionally other metrics like precision.
- Prints the evaluation results.

```python
def test(model, test_loader, criterion):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f'Accuracy: {100 * correct / total:.2f}%')
```

---

7. Evaluation

- Models were evaluated based on **Loss**, **Accuracy**, and **Precision**.
- Visualization plots for these metrics over epochs were created.

```
plt.figure(figsize=(12, 5))
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss')
plt.legend()
plt.show()
```

---

8. Unseen Data

Fashion-MNIST Testing

Tested models on Fashion-MNIST to evaluate generalization.

Conclusion

This project provided hands-on experience in:

1. Data preprocessing and augmentation.
2. Building and evaluating various neural network architectures.
3. Visualizing model performance with accuracy, loss, and precision metrics.

Each model exhibited strengths in different aspects, and the CNN model achieved the best accuracy.