Session 2: Guided Exercise
Handwritten Digit Recognition with Feedforward Neural Networks

Neural Networks Course - Computer Engineering

May 9, 2025

Abstract

In this guided exercise, you will implement a complete feedforward neural network for recognizing handwritten digits from the MNIST dataset using PyTorch. You will learn how to load and preprocess data, define a neural network architecture, train the model, and evaluate its performance. This exercise will reinforce theoretical concepts from the lecture and give you hands-on experience with neural network implementation.

Contents

1 Introduction

The MNIST dataset (Modified National Institute of Standards and Technology) is a large collection of handwritten digits commonly used for training and testing machine learning algorithms. It contains 70,000 grayscale images of handwritten digits (0-9), each of size 28×28 pixels.

In this guided exercise, you will build a feedforward neural network (also known as a multilayer perceptron) to recognize these handwritten digits. By the end of this exercise, you will have implemented a complete deep learning pipeline and learned how to:

- Load and preprocess image data
- Define a neural network architecture
- Train a model using backpropagation and gradient descent
- Evaluate model performance
- Visualize results and model predictions

Note

This exercise assumes basic familiarity with Python and PyTorch. If you are new to PyTorch, please refer to the documentation at https://pytorch.org/docs/stable/index.html.

2 Setting Up the Environment

Before starting, ensure that you have all the necessary libraries installed. You will need PyTorch, torchvision, matplotlib, and numpy. If you are using Google Colab, these libraries are pre-installed.

```
pip install torch torchvision matplotlib numpy
```

Listing 1: Installing required packages

Alternatively, you can use the provided Jupyter notebook that already has all dependencies configured.

Tip

If you encounter CUDA-related errors and don't have a GPU, you can force CPU usage with: `device = torch.device("cpu")`

3 MNIST Dataset Overview

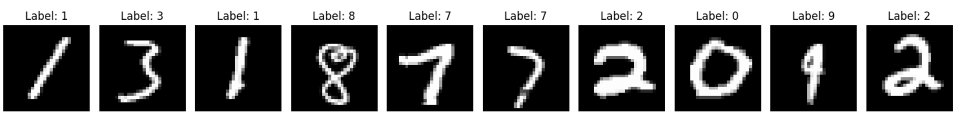

Figure 1: Sample images from the MNIST dataset

The MNIST dataset has the following characteristics:

- 60,000 training images and 10,000 test images
- 10 classes (digits 0-9)
- Grayscale images with size 28×28 pixels
- Pixel values range from 0 (white) to 255 (black)

Figure 1 shows sample images from the dataset. Notice the variations in handwriting styles, which make this a good challenge for classification algorithms.

4 Loading and Preprocessing Data

The first step is to load the MNIST dataset and prepare it for training. PyTorch provides convenient utilities through the `torchvision.datasets` module.

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.1307,), (0.3081,))  # Normalize with mean and std of
     MNIST
])

# Download and load training data
train_dataset = datasets.MNIST(root='./data',
                              train=True,
                              download=True,
                              transform=transform)

# Download and load test data
test_dataset = datasets.MNIST(root='./data',
                             train=False,
                             download=True,
                             transform=transform)

# Create data loaders for batch processing
batch_size = 64
train_loader = DataLoader(train_dataset,
                          batch_size=batch_size,
                          shuffle=True)

test_loader = DataLoader(test_dataset,
                         batch_size=batch_size,
                         shuffle=False)
```

Listing 2: Loading the MNIST dataset

Note

The `Normalize` transform standardizes the pixel values using the mean (0.1307) and standard deviation (0.3081) of the MNIST dataset. This helps in faster convergence during training.

5 Defining the Neural Network Architecture

Now, let's define our feedforward neural network architecture. We'll create a simple network with one hidden layer.

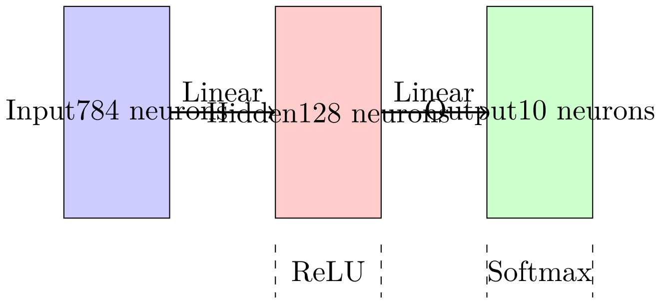

Figure 2: Architecture of our feedforward neural network

```python
import torch.nn as nn
import torch.nn.functional as F

class FeedforwardNeuralNet(nn.Module):
    """A simple feedforward neural network with one hidden layer"""

    def __init__(self, input_size, hidden_size, output_size):
        """Initialize the network architecture

        Args:
            input_size: Number of input features
            hidden_size: Number of neurons in the hidden layer
            output_size: Number of output classes
        """
        super(FeedforwardNeuralNet, self).__init__()

        # First fully connected layer (input $\rightarrow$ hidden)
        self.fc1 = nn.Linear(input_size, hidden_size)

        # Second fully connected layer (hidden $\rightarrow$ output)
        self.fc2 = nn.Linear(hidden_size, output_size)

        # Dropout layer for regularization (preventing overfitting)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        """Forward pass through the network

        Args:
            x: Input tensor of shape [batch_size, 1, 28, 28]

        Returns:
            Output tensor of shape [batch_size, output_size]
        """
        # Reshape input: [batch_size, 1, 28, 28] $\rightarrow$ [batch_size, 784]
        x = x.view(-1, 28*28)

        # First layer with ReLU activation
        x = F.relu(self.fc1(x))

        # Apply dropout
        x = self.dropout(x)

        # Output layer (no activation yet - will use softmax with loss function)
```

```python
        x = self.fc2(x)

        return x

# Initialize the model
input_size = 28 * 28  # MNIST images are 28x28 pixels
hidden_size = 128     # Number of neurons in the hidden layer
output_size = 10      # 10 digits (0-9)
model = FeedforwardNeuralNet(input_size, hidden_size, output_size)
```

Listing 3: Defining the neural network model

Tip

The choice of hidden layer size (128 neurons) is a hyperparameter. You can experiment with different values to see how it affects model performance.

6 Loss Function and Optimizer

To train our model, we need to define:

- A loss function to measure how well the model is performing
- An optimizer to update the model parameters based on the gradients

```python
import torch.optim as optim

# Define the loss function (Cross-Entropy Loss)
criterion = nn.CrossEntropyLoss()

# Define the optimizer (Stochastic Gradient Descent)
learning_rate = 0.01
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
```

Listing 4: Setting up loss function and optimizer

Note

The Cross-Entropy Loss combines softmax activation with negative log-likelihood loss, making it suitable for multi-class classification problems.

7 Training the Model

Now, let's implement the training loop. We'll train the model for multiple epochs (complete passes through the training dataset).

```python
# Training parameters
num_epochs = 5

# Lists to store metrics for plotting
train_losses = []
train_accuracies = []

def train(model, train_loader, criterion, optimizer, epoch):
    """Train the model for one epoch"""
    # Set model to training mode
    model.train()

```

```
    running_loss = 0.0
    correct = 0
    total = 0

    # Iterate over batches
    for batch_idx, (data, target) in enumerate(train_loader):
        # Clear gradients from previous step
        optimizer.zero_grad()

        # Forward pass
        outputs = model(data)

        # Calculate loss
        loss = criterion(outputs, target)

        # Backward pass
        loss.backward()

        # Update weights
        optimizer.step()

        # Accumulate loss
        running_loss += loss.item()

        # Calculate accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

        # Print statistics every 100 batches
        if (batch_idx + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], '
                  f'Step [{batch_idx+1}/{len(train_loader)}], '
                  f'Loss: {loss.item():.4f}, '
                  f'Accuracy: {100 * correct / total:.2f}%')

    # Calculate average metrics
    epoch_loss = running_loss / len(train_loader)
    epoch_acc = 100 * correct / total

    # Store for plotting
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_acc)

    return epoch_loss, epoch_acc

# Train for multiple epochs
for epoch in range(num_epochs):
    epoch_loss, epoch_acc = train(model, train_loader, criterion, optimizer, epoch)
    print(f'Epoch {epoch+1}/{num_epochs} completed - '
          f'Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')
```

Listing 5: Training loop

Warning

Training deep learning models can be computationally intensive. If you're running this on a CPU, it might take several minutes to complete.

8 Evaluating the Model

After training, we need to evaluate how well our model performs on unseen data (the test set).

```
def evaluate(model, test_loader):
    """Evaluate model performance on the test set"""
    # Set model to evaluation mode
    model.eval()

    correct = 0
    total = 0

    # Disable gradient calculation
    with torch.no_grad():
        for data, target in test_loader:
            # Forward pass
            outputs = model(data)

            # Get predictions
            _, predicted = torch.max(outputs.data, 1)

            # Update statistics
            total += target.size(0)
            correct += (predicted == target).sum().item()

    # Calculate and return accuracy
    accuracy = 100 * correct / total
    print(f'Test Accuracy: {accuracy:.2f}%')

    return accuracy

# Evaluate on test set
test_accuracy = evaluate(model, test_loader)
```

Listing 6: Model evaluation

Note

We set the model to evaluation mode (`model.eval()`) during testing. This disables dropout and batch normalization layers, which behave differently during training and evaluation.

9 Visualizing Results

Let's visualize the training progress and model predictions to better understand how our model performs.

9.1 Training Metrics

First, let's plot the loss and accuracy during training:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(train_losses)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')

```

```python
plt.subplot(1, 2, 2)
plt.plot(train_accuracies)
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')

plt.tight_layout()
plt.savefig('training_metrics.png')
plt.show()
```

Listing 7: Plotting training metrics

Figure 3: Training loss and accuracy over epochs

9.2 Visualizing Predictions

Now, let's visualize how well our model predicts on the test set:

```python
def visualize_predictions(model, test_loader, num_samples=10):
    """Visualize model predictions on sample test images"""
    # Set model to evaluation mode
    model.eval()

    # Get a batch of test data
    examples = iter(test_loader)
    samples, labels = next(examples)

    # Make predictions
    with torch.no_grad():
        outputs = model(samples)
        _, predicted = torch.max(outputs, 1)

    # Plot results
    plt.figure(figsize=(15, 3))
    for i in range(num_samples):
        plt.subplot(1, num_samples, i+1)
        plt.imshow(samples[i][0], cmap='gray')

        # Green title for correct predictions, red for incorrect
        if predicted[i] == labels[i]:
            plt.title(f'Pred: {predicted[i]}\nTrue: {labels[i]}', color='green')
        else:
            plt.title(f'Pred: {predicted[i]}\nTrue: {labels[i]}', color='red')

        plt.axis('off')

    plt.tight_layout()
    plt.savefig('model_predictions.png')
    plt.show()

# Visualize predictions
visualize_predictions(model, test_loader)
```

Listing 8: Visualizing model predictions

10 Saving and Loading the Model

Once you've trained a model that performs well, you'll want to save it for future use.

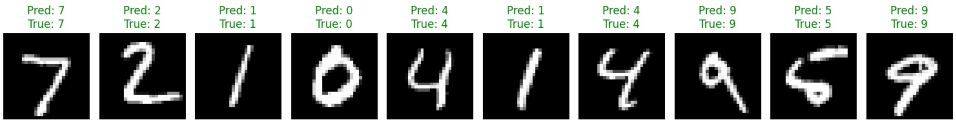

Figure 4: Model predictions on test images (green: correct, red: incorrect)

```
# Save the model
torch.save(model.state_dict(), 'mnist_feedforward_model.pth')
print("Model saved successfully!")

# Later, to load the model:
loaded_model = FeedforwardNeuralNet(input_size, hidden_size, output_size)
loaded_model.load_state_dict(torch.load('mnist_feedforward_model.pth'))
loaded_model.eval()  # Set to evaluation mode
```

Listing 9: Saving and loading the model

11 Extending the Exercise

Now that you have implemented a basic neural network for MNIST classification, you can extend your learning by trying the following challenges:

Challenge

1. Add more hidden layers to create a deeper network
2. Experiment with different activation functions (Sigmoid, Tanh, Leaky ReLU)
3. Implement learning rate scheduling to improve convergence
4. Try different optimizers (Adam, RMSprop)
5. Add batch normalization for faster and more stable training
6. Implement early stopping to prevent overfitting
7. Use data augmentation to improve generalization
8. Convert your model to a Convolutional Neural Network (CNN)

Here's an example of how to create a deeper network:

```
class DeepFeedforwardNet(nn.Module):
    """A deeper feedforward neural network with multiple hidden layers"""

    def __init__(self, input_size, hidden_sizes, output_size):
        """Initialize the network architecture

        Args:
            input_size: Number of input features
            hidden_sizes: List of sizes for each hidden layer
            output_size: Number of output classes
        """
        super(DeepFeedforwardNet, self).__init__()

        # Create a list to hold all layers
```

```python
        layers = []

        # Input layer
        layers.append(nn.Linear(input_size, hidden_sizes[0]))
        layers.append(nn.ReLU())
        layers.append(nn.BatchNorm1d(hidden_sizes[0]))
        layers.append(nn.Dropout(0.2))

        # Hidden layers
        for i in range(len(hidden_sizes) - 1):
            layers.append(nn.Linear(hidden_sizes[i], hidden_sizes[i+1]))
            layers.append(nn.ReLU())
            layers.append(nn.BatchNorm1d(hidden_sizes[i+1]))
            layers.append(nn.Dropout(0.2))

        # Output layer
        layers.append(nn.Linear(hidden_sizes[-1], output_size))

        # Combine all layers into a sequential model
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        """Forward pass through the network"""
        x = x.view(-1, 28*28)  # Flatten the input
        return self.model(x)

# Example usage:
deeper_model = DeepFeedforwardNet(
    input_size=28*28,
    hidden_sizes=[256, 128, 64],
    output_size=10
)
```

Listing 10: Implementing a deeper network

12 Comparison with State-of-the-Art

The feedforward neural network we implemented achieves reasonable performance on MNIST (typically around 97-98% accuracy), but it's worth noting that more advanced architectures like Convolutional Neural Networks (CNNs) can achieve over 99.5% accuracy on this dataset.

Model Architecture	**Typical Accuracy on MNIST**
Feedforward Neural Network (1 hidden layer)	~97-98%
Deep Feedforward Network (3+ hidden layers)	~98-99%
Convolutional Neural Network (CNN)	~99-99.5%
CNN with data augmentation	>99.5%

Table 1: Comparison of different architectures on MNIST classification

13 Conclusion

In this guided exercise, you implemented a complete feedforward neural network for handwritten digit recognition using the MNIST dataset. You learned how to:

- Load and preprocess image data
- Define a neural network architecture

- Implement the training and evaluation loops
- Visualize training progress and model predictions
- Save and load trained models

This implementation serves as a foundation for more complex neural network architectures and applications. The concepts and techniques you've learned are directly applicable to other image classification tasks and can be extended to more challenging datasets.

14 References

1. LeCun, Y., Cortes, C., & Burges, C. (2010). MNIST handwritten digit database. Retrieved from http://yann.lecun.com/exdb/mnist/
2. Paszke, A., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems 32 (NeurIPS 2019).
3. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. http://www.deeplearningbook.org

15 Appendix: Complete Code

The complete code for this exercise is available in the accompanying Python file `mnist_feedforward.py` and can also be found in the course GitHub repository. You can run it directly or use it as a reference for your own implementation.