Logistic Regression Analysis

Yazdan Bayat

May 25, 2025

1. Importing Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import plotly.express as px

from sklearn.linear_model import LogisticRegression, LassoCV
from sklearn.feature_selection import RFE
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    confusion_matrix, ConfusionMatrixDisplay,
    accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, classification_report,
    roc_curve
)
```

Listing 1: Importing required Python libraries

Explanation

This code imports all the essential libraries for data analysis, visualization, modeling, and evaluation:

- **NumPy, Pandas**: Data manipulation and numerical computing.
- **Matplotlib, Seaborn, Plotly**: Visualization libraries for static and interactive plots.
- **Scikit-learn Modules**:
  - `LogisticRegression`, `LassoCV`: Regression and regularization techniques.
  - `RFE`: Recursive Feature Elimination for feature selection.
  - `StandardScaler`: Standardizes features to zero mean and unit variance.
  - `train_test_split`: Splits dataset into training and testing sets.
  - `metrics`: Tools to evaluate model performance, such as accuracy, precision, recall, F1, AUC, and ROC.

2. Device Configuration

```
# Define device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Listing 2: Setting the computation device (CPU or GPU)

Explanation

This line checks whether a CUDA-enabled GPU is available and sets the computation device accordingly. If a GPU is available, the model and tensors will be moved to it, allowing faster computation. Otherwise, the CPU will be used. This abstraction ensures the code runs correctly on both GPU-equipped and standard systems.

3. Loading and Transforming the MNIST Dataset

```
# Define the transformation: convert to tensor and normalize
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # mean and std of MNIST
])

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

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

print("    MNIST dataset downloaded successfully.")
```

Listing 3: Downloading and transforming the MNIST dataset

Explanation

This block prepares the MNIST dataset for training and evaluation:

- **Transformation:** The images are converted to PyTorch tensors and normalized using the mean and standard deviation specific to MNIST: $(0.1307, 0.3081)$.
- **Dataset Download:**
  - `train=True` downloads the training set (60,000 samples).
  - `train=False` downloads the test set (10,000 samples).
- The `transform` is applied to both training and test sets to ensure consistent preprocessing.

4. Visualizing Sample Images

```
# Visualize a few samples
def show_samples(dataset, num_samples=6):
    plt.figure(figsize=(10, 2))
    for i in range(num_samples):
        image, label = dataset[i]
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(image.squeeze(), cmap='gray')
        plt.title(f"Label: {label}")
        plt.axis('off')
    plt.show()

# Call the visualization function
show_samples(train_dataset)
```

Listing 4: Function to visualize sample digits from the MNIST dataset

Explanation

This function is used to display a few example images from the MNIST dataset:

- **Function `show_samples`:** Takes a dataset and the number of samples to visualize.
- **Plotting:** Uses `matplotlib` to display a horizontal row of digit images.
- **Image Display:** Each image is squeezed to remove extra dimensions and shown in grayscale.
- **Labeling:** The title above each subplot shows the corresponding digit label.
- **Call:** `show_samples(train_dataset)` invokes the function to show sample images from the training set.

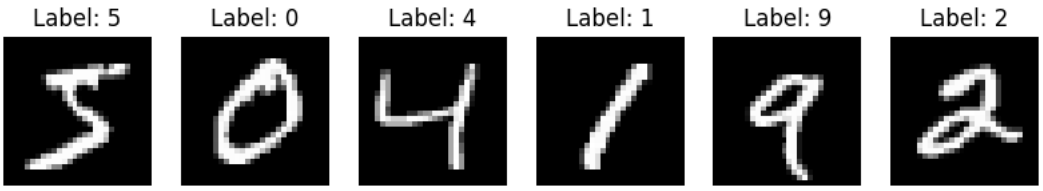

Figure 1: Sample MNIST digits displayed using the `show_samples` function.

5. Dataset Summary and Class Distribution

```
# Basic info
print(f"\u2705 Number of training samples: {len(train_dataset)}")
print(f"\u2705 Number of test samples: {len(test_dataset)}")

# Check balance of classes in the training set
labels = [label for _, label in train_dataset]
label_counts = Counter(labels)

print("\nClass distribution in training set:")
for digit in sorted(label_counts):
    print(f"Digit {digit}: {label_counts[digit]} samples")

# Optional: Plot class distribution
plt.bar(label_counts.keys(), label_counts.values())
plt.xlabel('Digit')
plt.ylabel('Count')
plt.title('Class Distribution in MNIST Training Set')
plt.grid(True)
plt.show()

# Sample image shape and range
image, label = train_dataset[0]
print(f"\nImage shape: {image.shape}")
print(f"Pixel value range: min={image.min():.4f}, max={image.max():.4f}")
```

Listing 5: Printing dataset statistics and visualizing class distribution

Explanation

This block provides descriptive statistics for the MNIST dataset:

- It prints the number of training and test samples.
- It computes the class distribution using Python's `Counter` to confirm dataset balance.
- A bar plot shows the number of samples per digit class (0–9).
- Finally, it prints the shape and pixel value range of a sample image.

Output

```
Number of training samples: 60000
Number of test samples: 10000

Class distribution in training set:
Digit 0: 5923 samples
Digit 1: 6742 samples
Digit 2: 5958 samples
Digit 3: 6131 samples
Digit 4: 5842 samples
```

```
Digit 5: 5421 samples
Digit 6: 5918 samples
Digit 7: 6265 samples
Digit 8: 5851 samples
Digit 9: 5949 samples

Image shape: torch.Size([1, 28, 28])
Pixel value range: min=0.0000, max=1.0000
```

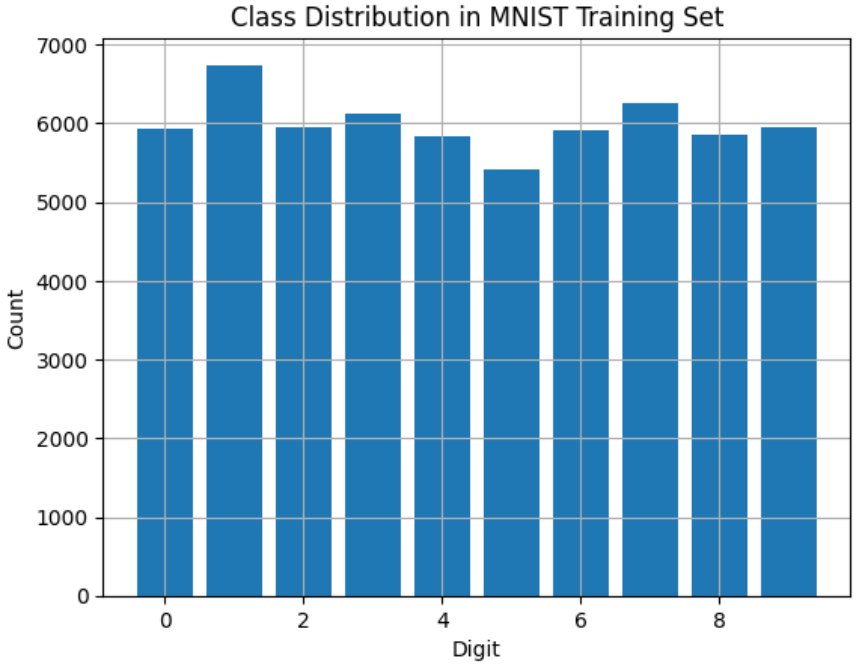

Figure 2: Bar chart showing the distribution of digit classes in the MNIST training set.

6. Creating Data Loaders for Training and Validation

```
from torch.utils.data import DataLoader  # Make sure this import is
    included

# Define transform (if not already defined)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

dataset = datasets.MNIST(root='./data', train=True, download=True,
    transform=transform)
val_size = int(0.15 * len(dataset))
train_size = len(dataset) - val_size
```

```
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=1000, shuffle=False)
```

Listing 6: Splitting the dataset and creating PyTorch data loaders

Explanation

This code prepares the MNIST data for model training and validation:

- The full training dataset (60,000 samples) is split into:
  - **Training set**: 85% of the original data
  - **Validation set**: 15%
- `random_split` is used to create the two sets, preserving class randomness.
- **DataLoader** objects:
  - `train_loader` shuffles data in mini-batches of 64 for stochastic training.
  - `val_loader` loads 1000 samples per batch without shuffling for consistent validation.

7. Defining the Neural Network Model

```
class Model_V1(nn.Module):
    def __init__(self):
        super(Model_V1, self).__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        return self.net(x)
```

Listing 7: Deep neural network architecture for MNIST classification

Explanation

This class defines a deep fully connected neural network for MNIST digit classification:

- **Input**: Each 28×28 image is flattened into a 784-dimensional vector.
- **Architecture**:
  - Dense layer with 512 neurons, followed by batch normalization, ReLU, and dropout.
  - Dense layer with 256 neurons, again followed by batch normalization, ReLU, and dropout.
  - A third dense layer with 128 neurons and ReLU.
  - Final output layer with 10 units, one for each digit class (0–9).
- **Activation**: ReLU non-linearity is used throughout.
- **Regularization**: Dropout layers with 30% rate reduce overfitting.

8. Model Initialization

```
# Initialize model, loss function, and optimizer
model = Model_V1().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
```

Listing 8: Initializing the model with regularization

Explanation

This block sets up the model and its training components with L2 regularization:

- `Model_V1()` instantiates the deep neural network and moves it to the specified device (GPU or CPU).
- `CrossEntropyLoss()` is appropriate for multi-class classification tasks and will compare the model's predicted logits to the integer-encoded target labels.
- `Adam` optimizer is initialized with:
  - A learning rate of `0.001` for stable convergence.
  - A `weight_decay=1e-4` term, which applies L2 regularization (also known as Ridge regularization) to reduce overfitting by penalizing large weights.

9. Training and Validation with Loss Monitoring

```
epochs = 15
train_losses = []
val_losses = []

for epoch in range(epochs):
    # Training
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(train_loader)
    train_losses.append(train_loss)

    # Validation
    model.eval()
    val_running_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_running_loss += loss.item()

    val_loss = val_running_loss / len(val_loader)
    val_losses.append(val_loss)

    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{epochs}] - Train Loss: {train_loss:.4f}
   - Val Loss: {val_loss:.4f}")

# Plotting losses
plt.plot(range(1, epochs+1), train_losses, label='Train Loss')
plt.plot(range(1, epochs+1), val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training vs Validation Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 9: Training and validation with tracked losses

Explanation

This section introduces tracking of both training and validation losses across epochs:

- **Training Loop**:
  - The model is set to training mode with `model.train()`.
  - For each batch, loss is computed, backpropagated, and accumulated.
  - Average loss per epoch is stored in `train_losses`.
- **Validation Loop**:
  - Performed with `model.eval()` and `torch.no_grad()` to avoid gradient computation.
  - Average validation loss is calculated and stored in `val_losses`.
- **Visualization**: After training, a plot compares training and validation losses to assess model performance and overfitting.

Output

```
Epoch [5/15] - Train Loss: 0.0876 - Val Loss: 0.0748
Epoch [10/15] - Train Loss: 0.0653 - Val Loss: 0.0731
Epoch [15/15] - Train Loss: 0.0575 - Val Loss: 0.0718
```

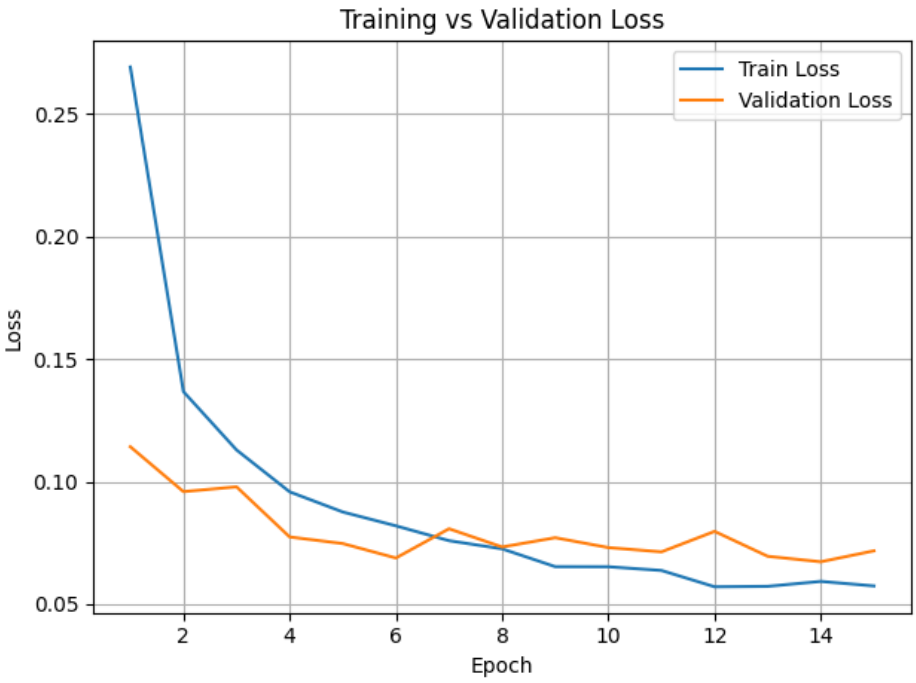

Figure 3: Line plot showing training vs validation loss over 15 epochs.

10. Test Set Evaluation

```
# Prepare test data loader
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)

# Evaluate on test dataset
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"\nTest Accuracy of Model_V1: {100 * correct / total:.2f}%")
```

Listing 10: Evaluating the model on the test dataset

Explanation

This block evaluates the trained model on the MNIST test set:

- A `DataLoader` is created for the test dataset with a batch size of 1000 and no shuffling.
- The model is set to evaluation mode using `model.eval()`.
- Gradient tracking is disabled with `torch.no_grad()` to improve efficiency.
- For each batch, predictions are compared to the true labels to compute total accuracy.

Output

```
Test Accuracy of Model_V1: 98.14%
```

11. Visualizing One Prediction per Digit

```
#     Collect 1 example per digit
samples = {}
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predictions = torch.max(outputs, 1)

        for img, label, pred in zip(images, labels, predictions):
            label = label.item()
```

```
            if label not in samples:
                samples[label] = (img.cpu(), pred.item())
            if len(samples) == 10:
                break
        if len(samples) == 10:
            break

#     Plot 1 image per class
fig, axes = plt.subplots(2, 5, figsize=(12, 5))
axes = axes.flatten()
for i, digit in enumerate(sorted(samples.keys())):
    image, pred = samples[digit]
    axes[i].imshow(image.squeeze(), cmap='gray')
    axes[i].set_title(f"Label: {digit}, Pred: {pred}")
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```

Listing 11: Plotting one prediction per digit from the test set

Explanation

This block displays one representative example from each digit class (0–9) along with the model's prediction:

- The script iterates through the test set and collects one image per class using a dictionary.
- After collecting 10 distinct digits, it stops further evaluation.
- The selected images are then plotted in a 2×5 grid with their true label and predicted label shown in the title.

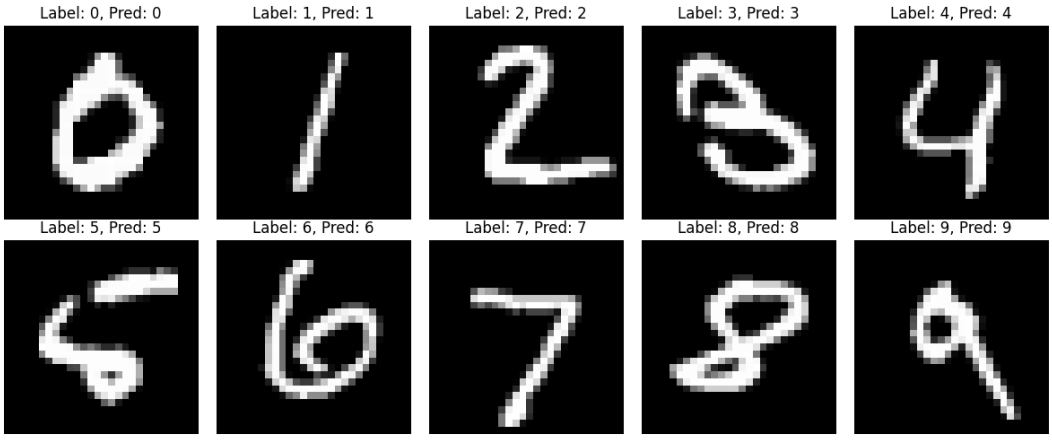

Figure 4: One predicted example per digit class from the MNIST test set.

12. Training a Lightweight Neural Network with Dropout and Regularization

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Transform: normalize MNIST images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

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

# Split training set into training and validation sets (90% train, 10% val
   )
train_size = int(0.9 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(train_dataset, [train_size,
   val_size])

# Data loaders
train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_subset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define simple model
class SimpleMNISTModel(nn.Module):
    def __init__(self):
        super(SimpleMNISTModel, self).__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 32),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 10)
        )

    def forward(self, x):
```

```
        return self.net(x)

# Initialize model, loss function, and optimizer
model = SimpleMNISTModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)

# Train the model and record losses
epochs = 15
train_losses = []
val_losses = []

for epoch in range(epochs):
    # Training
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_loss = running_loss / len(train_loader)
    train_losses.append(train_loss)

    # Validation
    model.eval()
    val_running_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_running_loss += loss.item()

    val_loss = val_running_loss / len(val_loader)
    val_losses.append(val_loss)

    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {train_loss:.4f},
   Val Loss: {val_loss:.4f}")

# Final Validation Accuracy
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in val_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
```

```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"\nValidation Accuracy of SimpleMNISTModel: {100 * correct / total
   :.2f}%")

# Plot training and validation loss
plt.plot(range(1, epochs+1), train_losses, label="Train Loss")
plt.plot(range(1, epochs+1), val_losses, label="Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 12: Simplified neural network with dropout and weight decay

Explanation

In this version, a smaller network with two hidden layers and dropout regularization is trained:

- **Model Architecture**:
  - First dense layer with 32 neurons, followed by ReLU and dropout (20%).
  - Second layer with 16 neurons and ReLU activation.
  - Output layer maps to 10 classes (digits 0–9).
- **Regularization**:
  - `Dropout` prevents overfitting by randomly deactivating neurons during training.
  - `weight_decay` applies L2 regularization to discourage large weights.
- **Training Procedure**: Losses are recorded for both training and validation sets across 15 epochs.
- **Evaluation**: Final validation accuracy is computed, and loss curves are visualized.

Output

```
Epoch [5/15], Train Loss: 0.2208, Val Loss: 0.1946
Epoch [10/15], Train Loss: 0.1797, Val Loss: 0.1692
Epoch [15/15], Train Loss: 0.1581, Val Loss: 0.1570

Validation Accuracy of SimpleMNISTModel: 95.63%
```

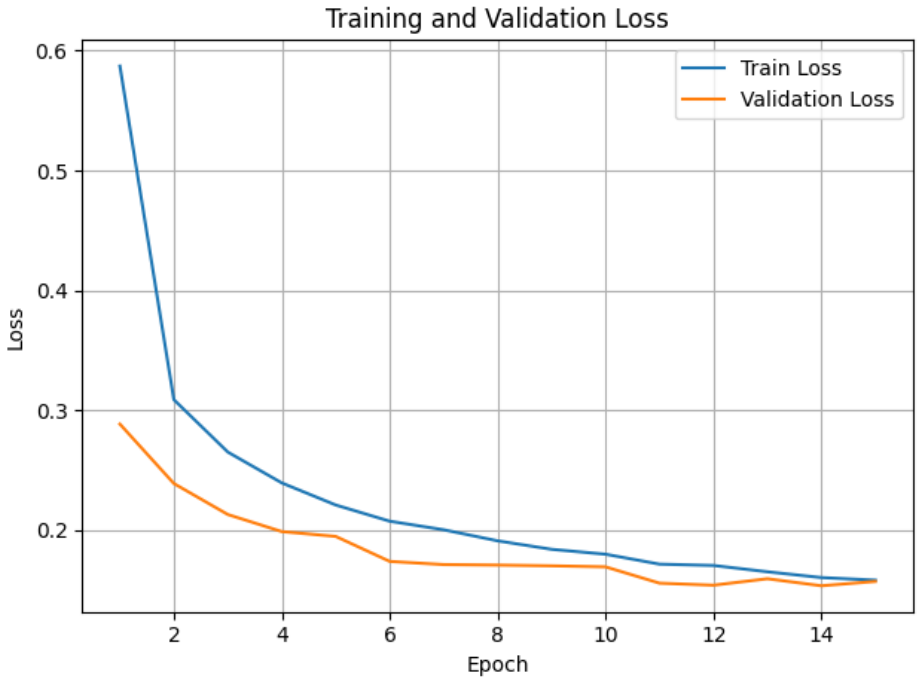

Figure 5: Loss curves of training and validation sets for SimpleMNISTModel with dropout and L2 regularization.

13. Test Set Evaluation for Simple Model

```
# Prepare test data loader (if not already defined)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)

# Evaluate on test dataset
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"\nTest Accuracy of SimpleMNISTModel: {100 * correct / total:.2f
   }%")
```

Listing 13: Evaluating the SimpleMNISTModel on the test dataset

Explanation

This code evaluates the performance of the trained `SimpleMNISTModel` on unseen test data:

- The test dataset is loaded in large batches (size 1000) for faster evaluation.
- The model is switched to evaluation mode with `model.eval()`.
- `torch.no_grad()` disables gradient tracking for memory and speed efficiency.
- Predictions are obtained using `torch.max`, and total correct classifications are counted to compute the final test accuracy.

Output

```
Test Accuracy of SimpleMNISTModel: 96.35%
```