Homework #6
EE 541: Fall 2023

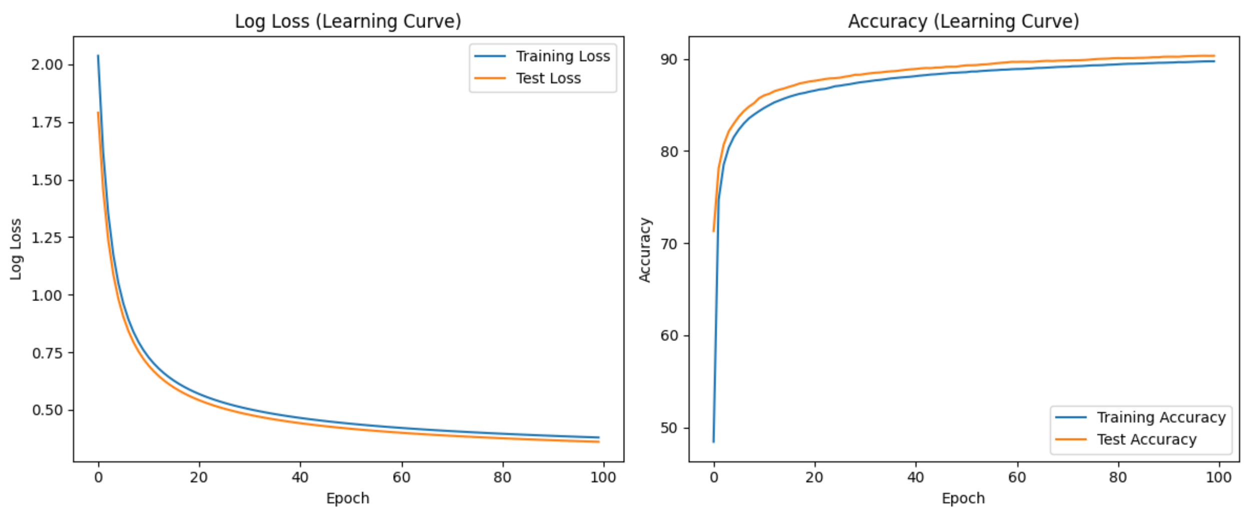
Name: Nissanth Neelakandan Abirami
USC ID: 2249203582
Instructor: [Dr. Franzke]

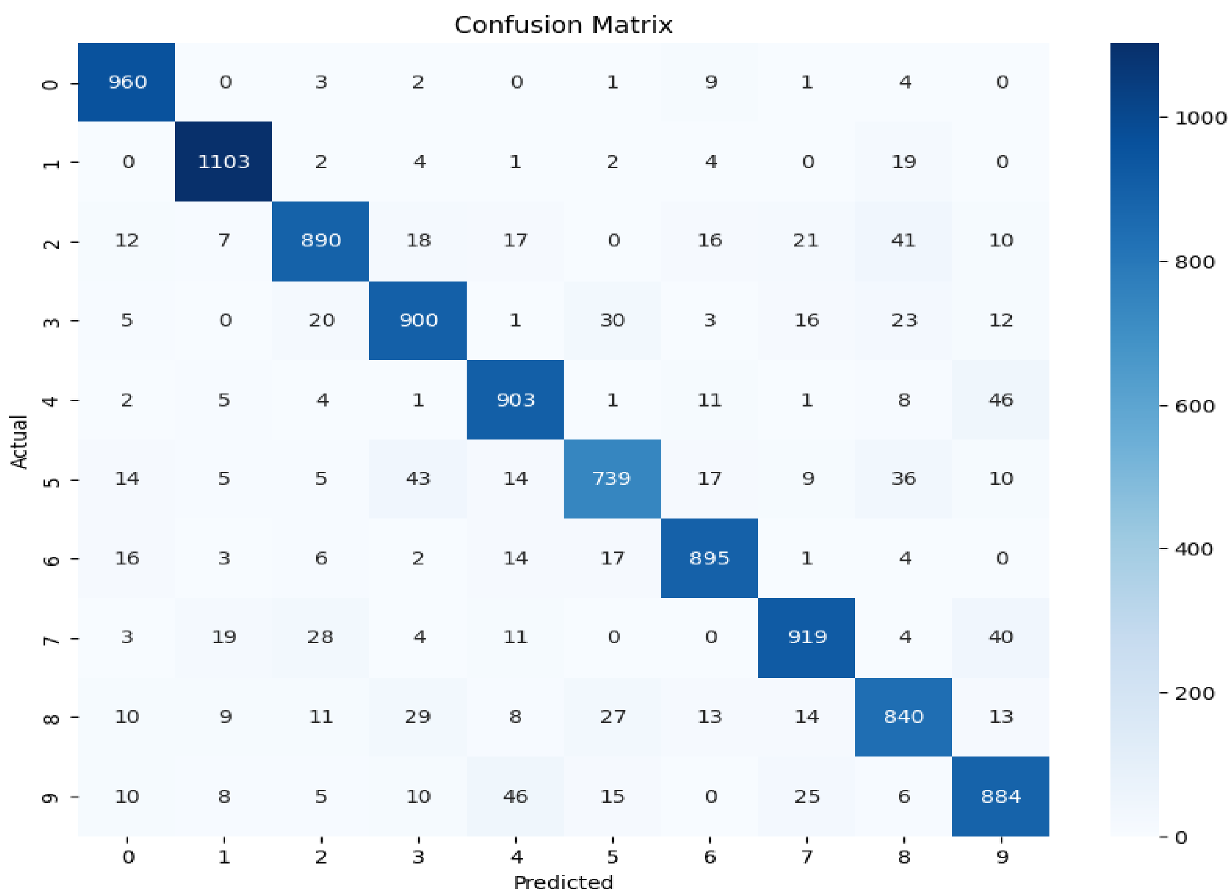
1. Use PyTorch to train a logistic-classifier for the MNIST dataset of handwritten digits. Use the provided PyTorch DataSet to read from the earlier MNIST data files — mnist traindata.hdf5 and mnist testdata.hdf5. Create a nn.Sequential network with a single fully-connected (i.e., linear) layer. Choose dimensions to match the multi-class classifier from Homework 3. Omit the final soft-max activation (PyTorch implicitly computes this when calculating the loss).
Train your model using a multi-category cross-entropy loss. PyTorch provides the CrossEntropyLoss to numerically stabilize the combined SoftMax-NLLikelihood calculation. Optimize with (standard) stochastic gradient descent (SGD) and with a mini-batch size of 100. Experiment with l1- and/or l2-regularization to stabilize training and improve generalization. You may need to experiment the learning rate to improve performance. Record the log-loss and accuracy of your model on the training set and test set at the end of each epoch. Plot log-loss (i.e., learning curve) of the training set and test set on the same figure. On a separate figure plot the accuracy of your model on the training set and test set. Plot each as a function of the epoch number. Evaluate the fully trained model on the test data and generate a confusion matrix – i.e., find the classification rate conditioned on the true class. Element (i, j) of the confusion matrix is the rate at which the network decides class j when class i is the correct label (ground truth). Use seaborn or similar python package to generate a heatmap showing the confusion matrix.

The graphs after visualization and training yields maximum accuracy in Learning rate = 0.01 and decay of 1e-5.
It gives Training accuracy of 92.21 percent and Testing accuracy of 92.23 percent the corresponding losses are 0.2801 and 0.2781

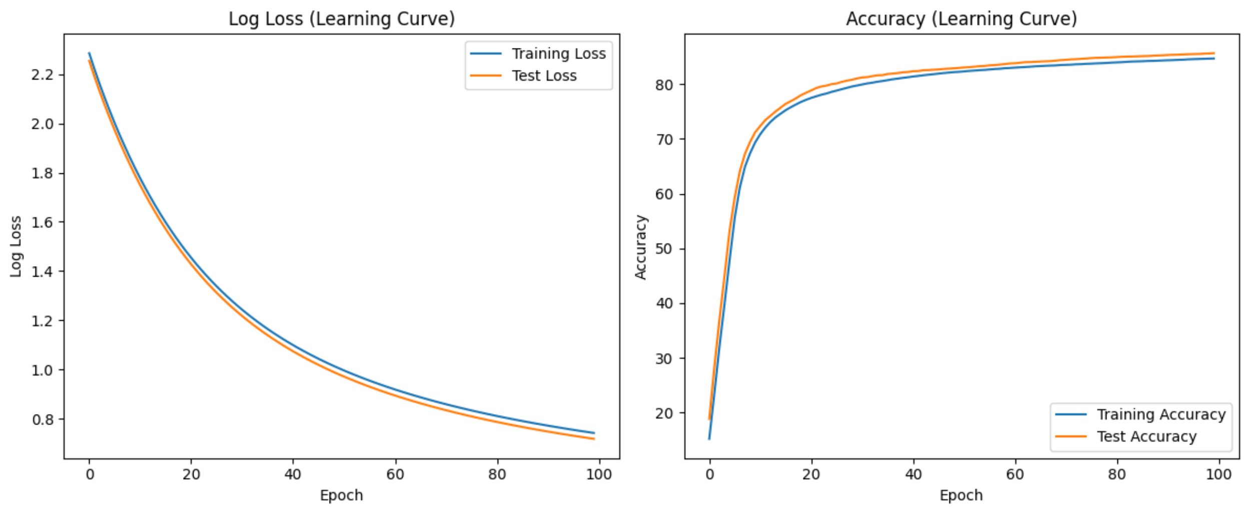
Learning rate 0.001
Decay - 1e-5

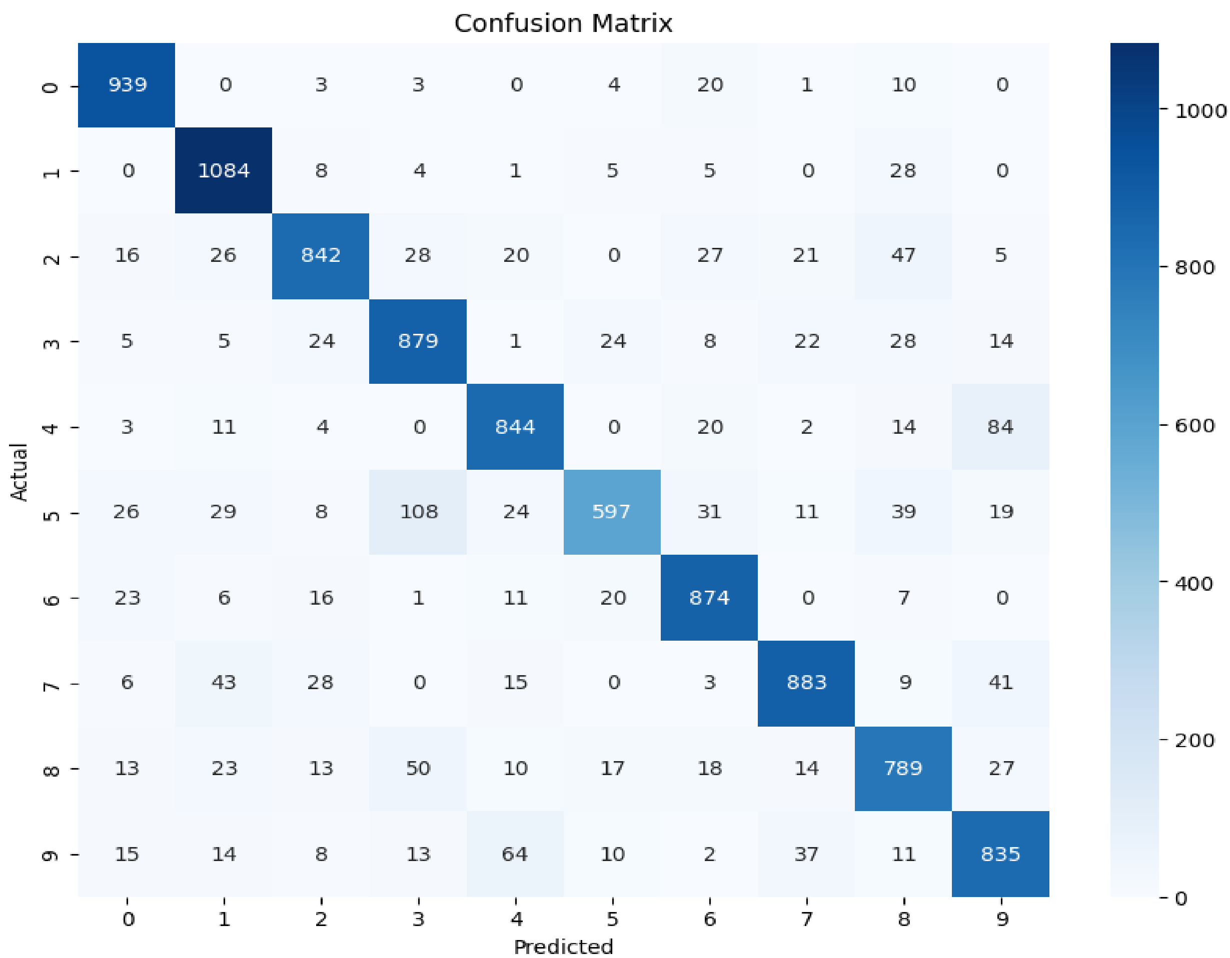
```
Epoch [100/100], Training Loss: 0.3803, Training Accuracy: 89.73%, Test Loss: 0.3617, Test Accuracy: 90.33%
```





Learning rate- 0.0001
Decay - 1e-5
`Epoch [100/100], Training Loss: 0.7428, Training Accuracy: 84.69%, Test Loss: 0.7190, Test Accuracy: 85.66%`

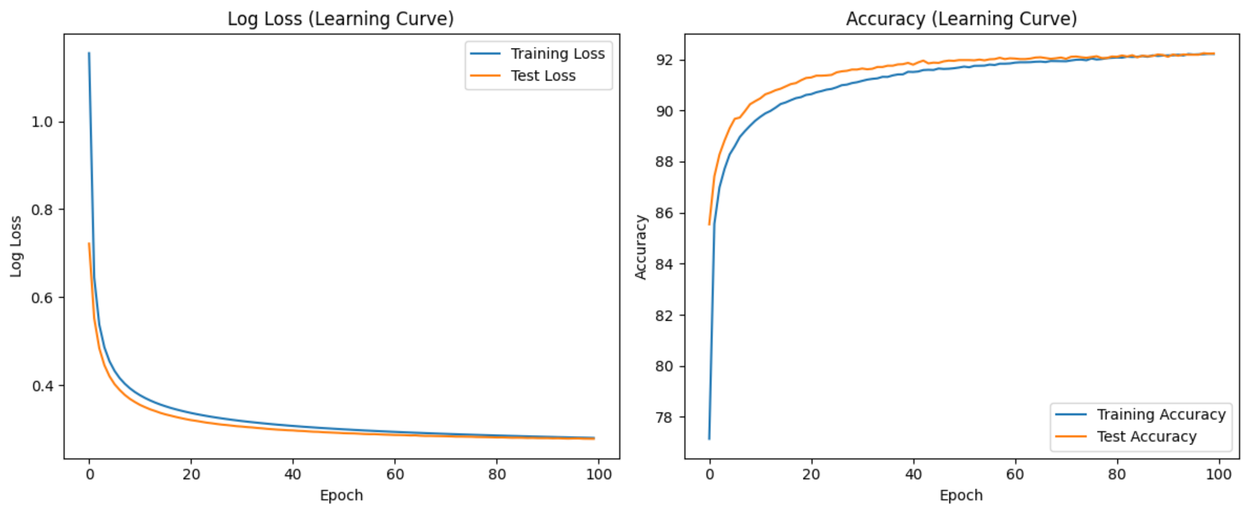


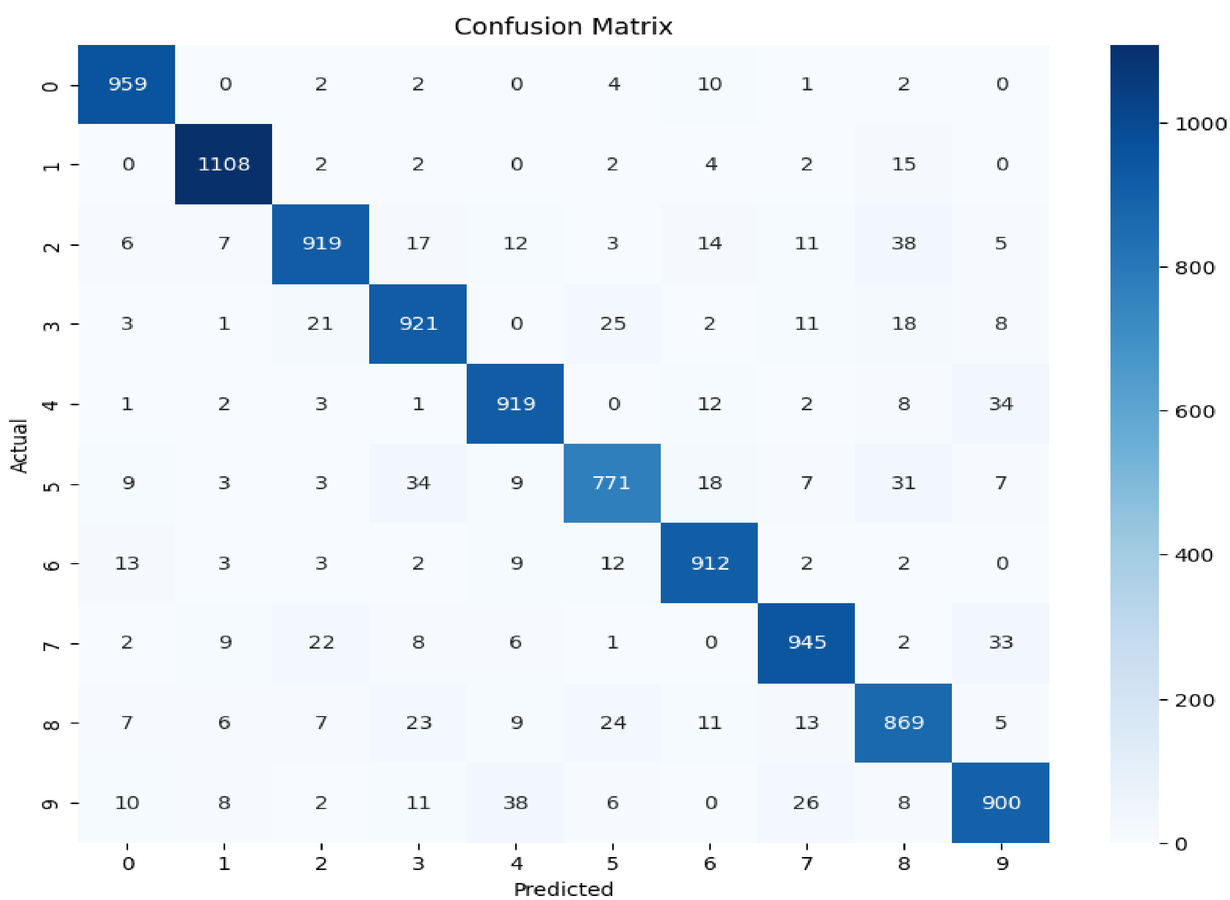


Learning rate - 0.01

Decay- 1e-5

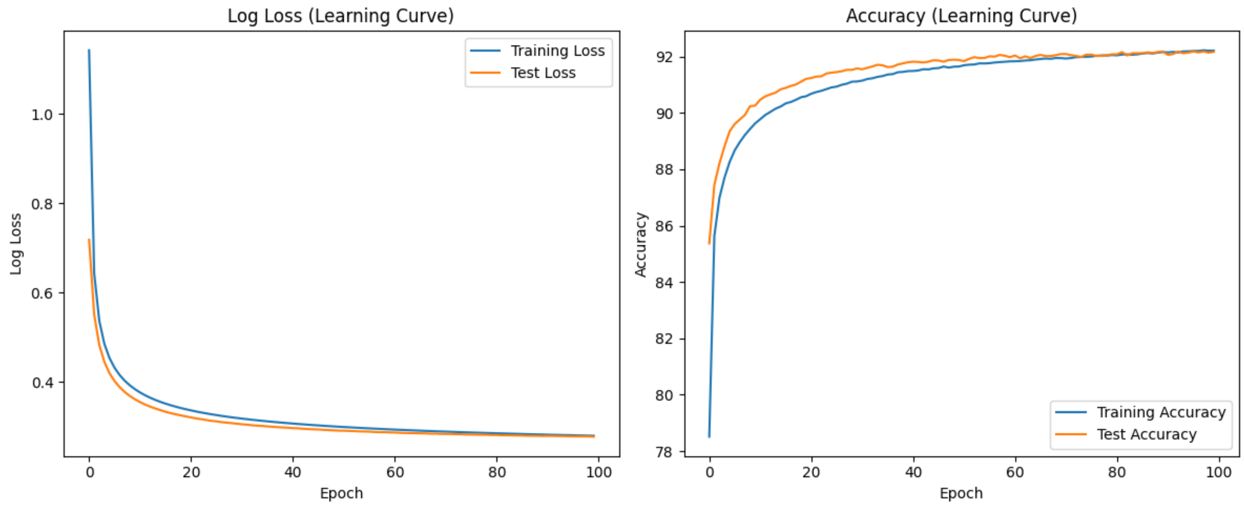
`Epoch [100/100], Training Loss: 0.2801, Training Accuracy: 92.21%, Test Loss: 0.2781, Test Accuracy: 92.23%`

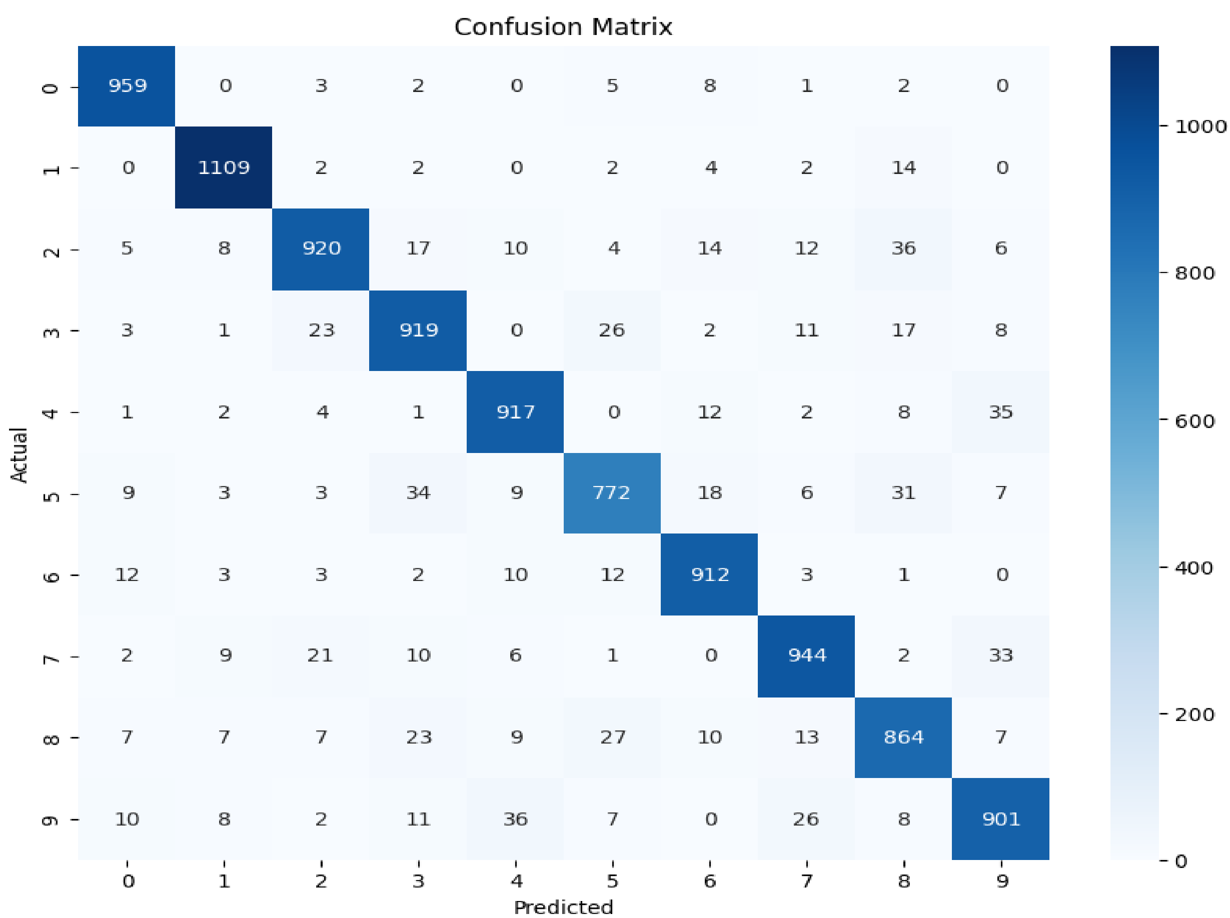




Learning rate - 0.01
Deacy- 1e-10

`Epoch [100/100], Training Loss: 0.2800, Training Accuracy: 92.21%, Test Loss: 0.2779, Test Accuracy: 92.17%`

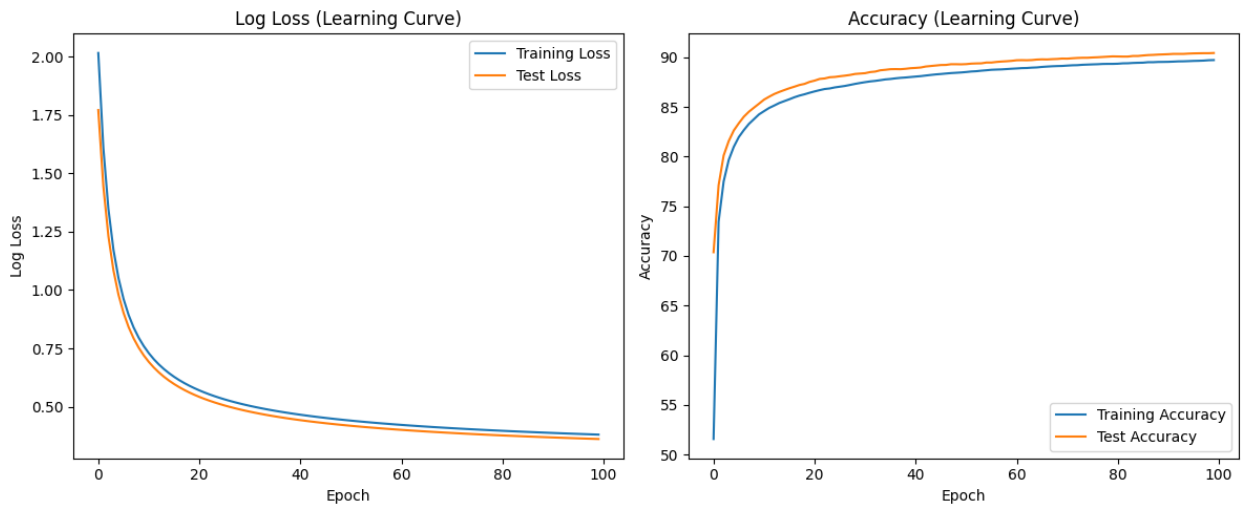


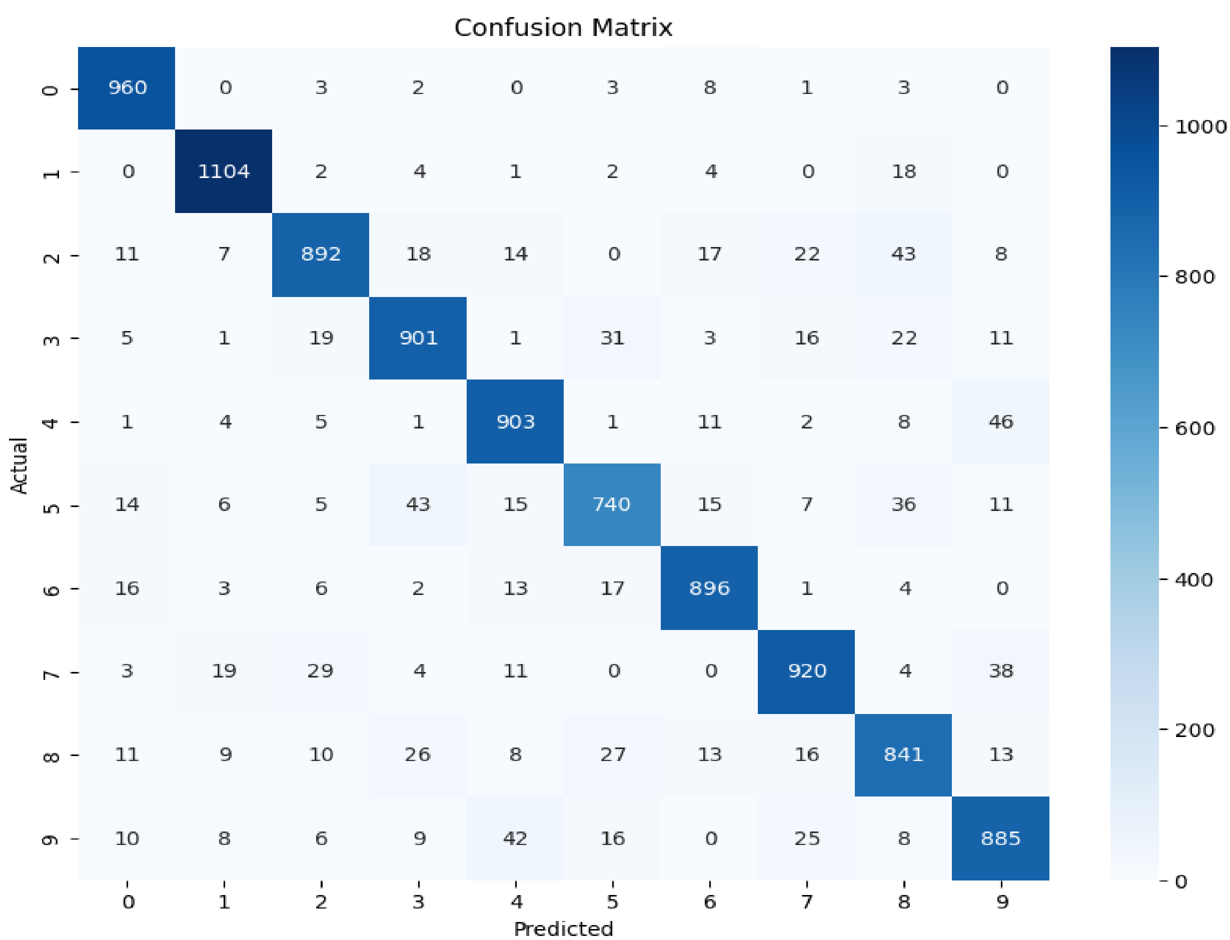


Learning rate- 0.001

Decay- 1e-10

`Epoch [100/100], Training Loss: 0.3802, Training Accuracy: 89.71%, Test Loss: 0.3613, Test Accuracy: 90.42%`

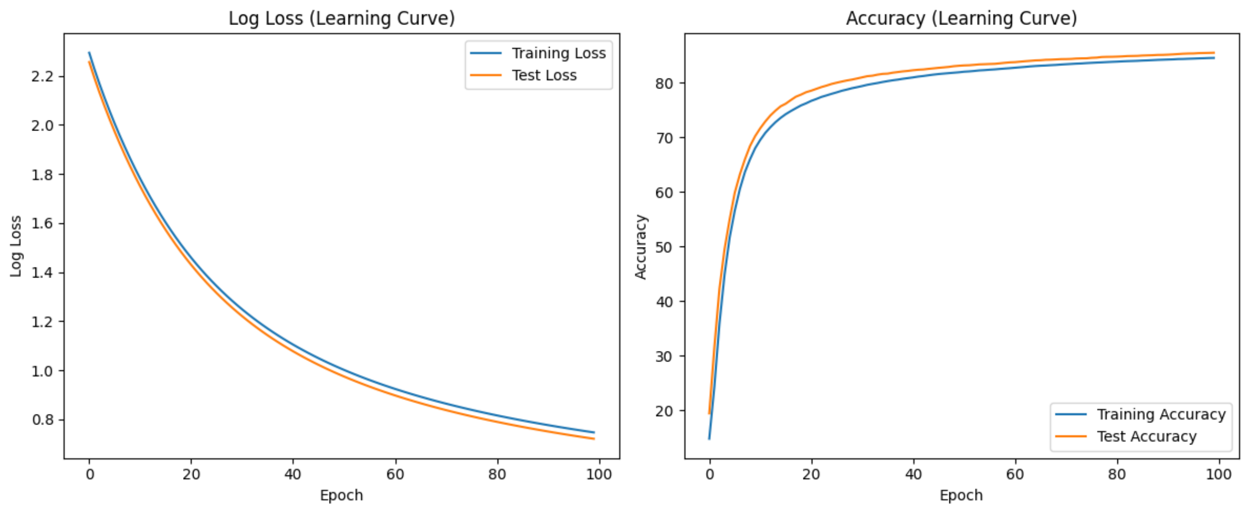


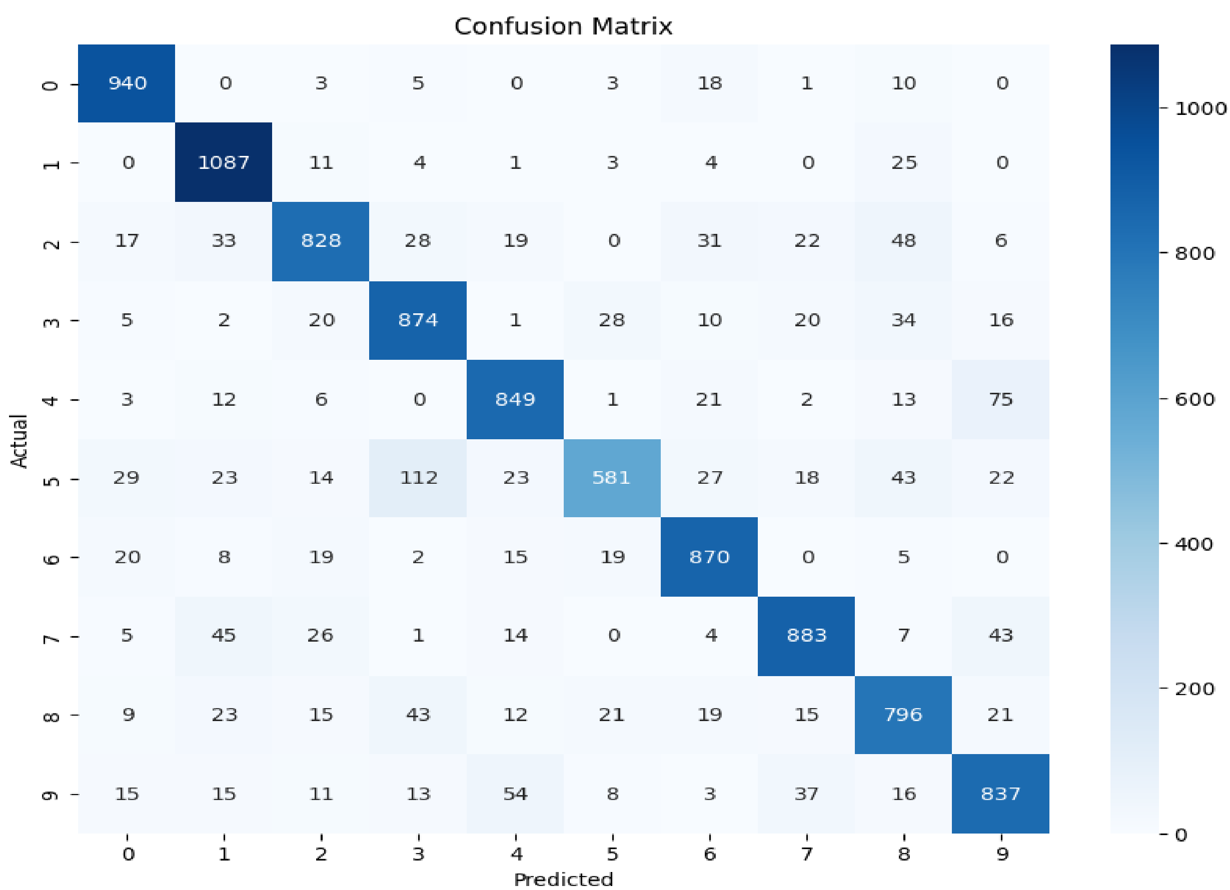


Learning rate - 0.0001

Decay - 1e-10

```
Epoch [100/100], Training Loss: 0.7471, Training Accuracy: 84.50%, Test
Loss: 0.7213, Test Accuracy: 85.45%
```





APPENDIX:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sn
from pathlib import Path
import h5py

# Define the HDF5Dataset class
class HDF5Dataset(torch.utils.data.Dataset):
    """Abstract HDF5 dataset"""
    def __init__(self, file_path, data_name, label_name):
        super().__init__()
        self.data = {}
        self.data_name = data_name
        self.label_name = label_name

        h5dataset_fp = Path(file_path)
        assert(h5dataset_fp.is_file())

        with h5py.File(file_path) as h5_file:
            # iterate datasets
            for dname, ds in h5_file.items():
                self.data[dname] = ds[()]

    def __getitem__(self, index):
        # get data
        x = self.data[self.data_name][index]
        x = torch.from_numpy(x)

        # get label
        y = self.data[self.label_name][index]
        y = torch.from_numpy(y)
        return x, y

    def __len__(self):
        return len(self.data[self.data_name])
```

```python
# Create an instance of the HDF5Dataset for training and testing
train_set = HDF5Dataset(file_path="/content/mnist_traindata.hdf5",
data_name='xdata', label_name='ydata')
train_loader = DataLoader(train_set, batch_size=100, shuffle=True)

test_set = HDF5Dataset(file_path="/content/mnist_testdata.hdf5",
data_name='xdata', label_name='ydata')
test_loader = DataLoader(test_set, batch_size=len(test_set))

# Define the model
num_pixels = 28 * 28
num_classes = 10

model = nn.Sequential(
    nn.Linear(in_features=num_pixels, out_features=num_classes)
)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, weight_decay=1e-5)

# Training the model
num_epochs = 100  # Adjust the number of epochs as needed
loss_train_list = []
accuracy_train_list = []
loss_test_list = []
accuracy_test_list = []

for epoch in range(num_epochs):
    model.train()
    correct_train = 0
    total_train = 0
    epoch_loss_train = 0

    for x_batch, y_batch in train_loader:
        x_batch = x_batch.view(x_batch.size(0), -1)

        # Forward pass
        outputs = model(x_batch)
        loss = criterion(outputs, torch.argmax(y_batch, dim=1))
```

```
        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate training accuracy and log-loss
        predictions_train = torch.argmax(outputs, 1)
        true_labels = torch.argmax(y_batch, 1)
        correct_train += (true_labels == predictions_train).sum().item()
        total_train += len(y_batch)
        epoch_loss_train += loss.item()

    # Calculate training accuracy and log-loss for the epoch
    accuracy_train = correct_train * 100 / total_train
    loss_train_list.append(epoch_loss_train / len(train_loader))
    accuracy_train_list.append(accuracy_train)

    # Evaluate on the test set
    model.eval()
    correct_test = 0
    total_test = 0
    epoch_loss_test = 0

    for x_test, y_test in test_loader:
        x_test = x_test.view(x_test.size(0), -1)
        output_test = model(x_test)
        loss_test = criterion(output_test, torch.argmax(y_test, dim=1))

        # Calculate test accuracy and log-loss
        predictions_test = torch.argmax(output_test, 1)
        true_labels_test = torch.argmax(y_test, 1)
        correct_test += (true_labels_test ==
predictions_test).sum().item()
        total_test += len(y_test)
        epoch_loss_test += loss_test.item()

    # Calculate test accuracy and log-loss for the epoch
    accuracy_test = correct_test * 100 / total_test
    loss_test_list.append(epoch_loss_test / len(test_loader))
```

```
    accuracy_test_list.append(accuracy_test)

    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Training Loss: {epoch_loss_train / len(train_loader):.4f}, '
          f'Training Accuracy: {accuracy_train:.2f}%, '
          f'Test Loss: {epoch_loss_test / len(test_loader):.4f}, '
          f'Test Accuracy: {accuracy_test:.2f}%')

# Plotting the learning curves
plt.figure(figsize=(12, 5))

# Plot log-loss (learning curve)
plt.subplot(1, 2, 1)
plt.plot(loss_train_list, label='Training Loss')
plt.plot(loss_test_list, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Log Loss')
plt.title('Log Loss (Learning Curve)')
plt.legend()

# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(accuracy_train_list, label='Training Accuracy')
plt.plot(accuracy_test_list, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy (Learning Curve)')
plt.legend()

plt.tight_layout()
plt.show()

# Evaluate on the test data and generate a confusion matrix
model.eval()
all_predictions = []
all_targets = []

for x_test, y_test in test_loader:
    x_test = x_test.view(x_test.size(0), -1)
    output = model(x_test)
```

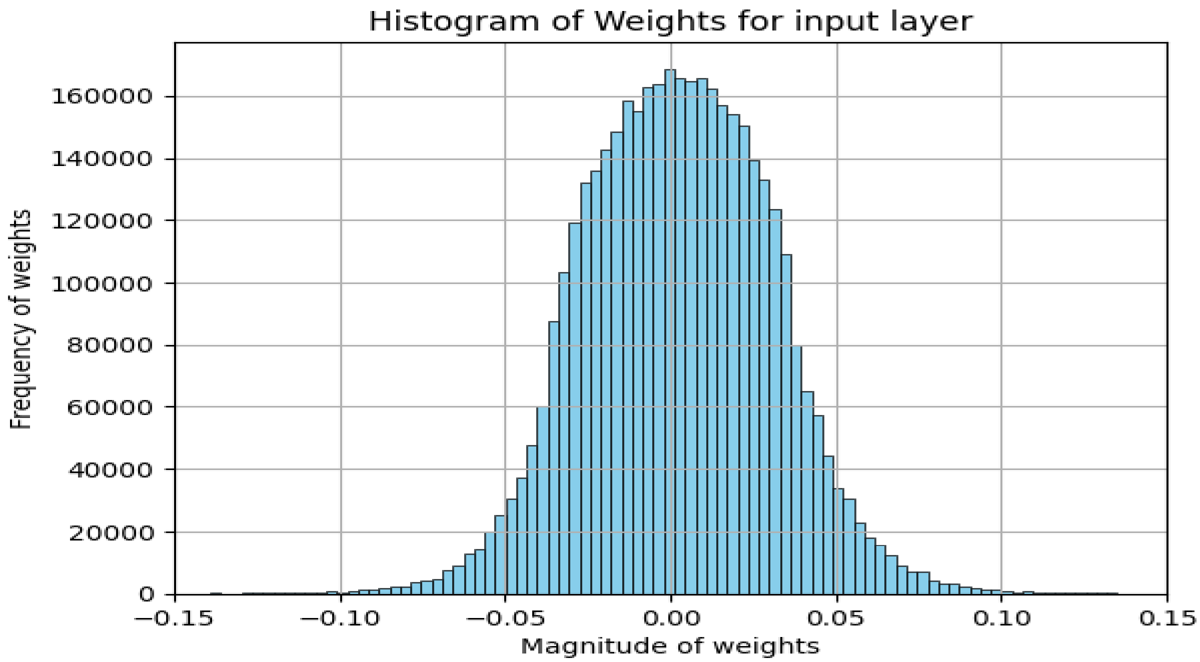
```
    prediction = torch.argmax(output, 1)

    # Accumulate predictions and ground truth labels
    all_predictions.extend(prediction.numpy())
    all_targets.extend(torch.argmax(y_test, 1).numpy())

# Computing the confusion matrix
cf_matrix = confusion_matrix(all_targets, all_predictions)

# Generating a heatmap of the confusion matrix
plt.figure(figsize=(10, 8))
sn.heatmap(cf_matrix, fmt='0', annot=True, cmap="Blues",
xticklabels=range(num_classes), yticklabels=range(num_classes))
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

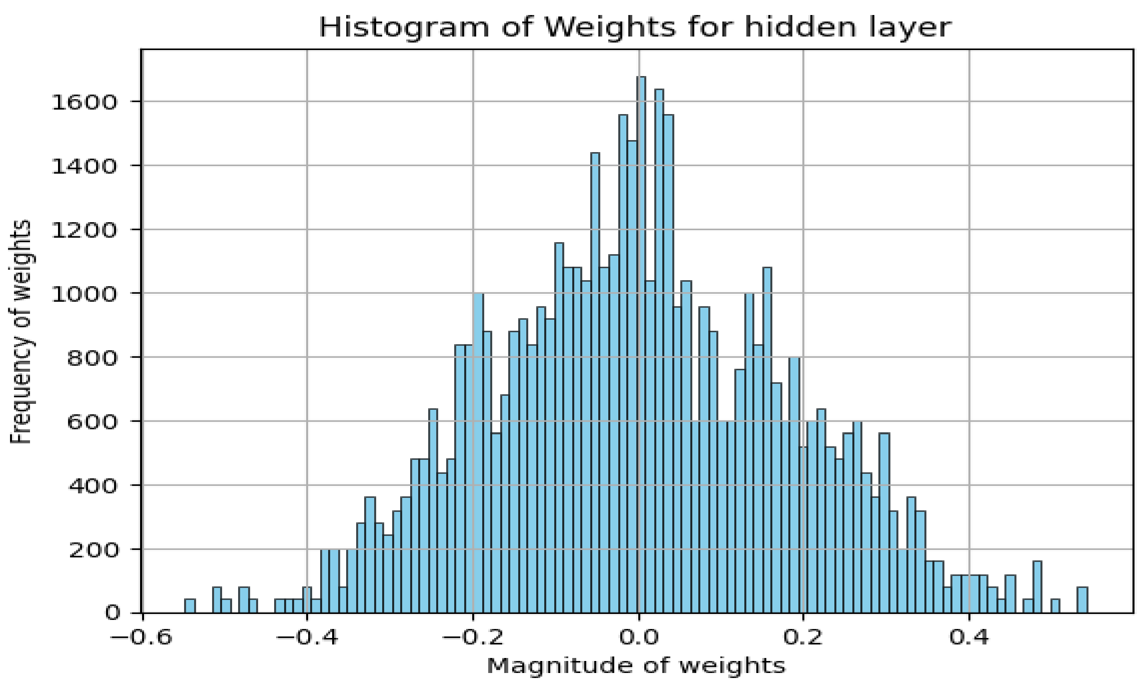
2. Train two models on Fashion MNIST for 40 epochs:
(1). One hidden layer with ReLU activation, 128 nodes. No regularization, no dropout.
(2). One hidden layer with ReLU activation, 48 nodes. L2 regularization with coefficient λ = 0.0001 and dropout with rate 0.2 at the hidden layer.
You may use the demo scripts from lecture and discussion. Produce histograms for the weights of these two networks – a separate histogram for the two layers (the input and hidden layers) in each case. Describe the qualitative differences between these histograms. What effect does regularization have on the distribution of weights?

Solutions:

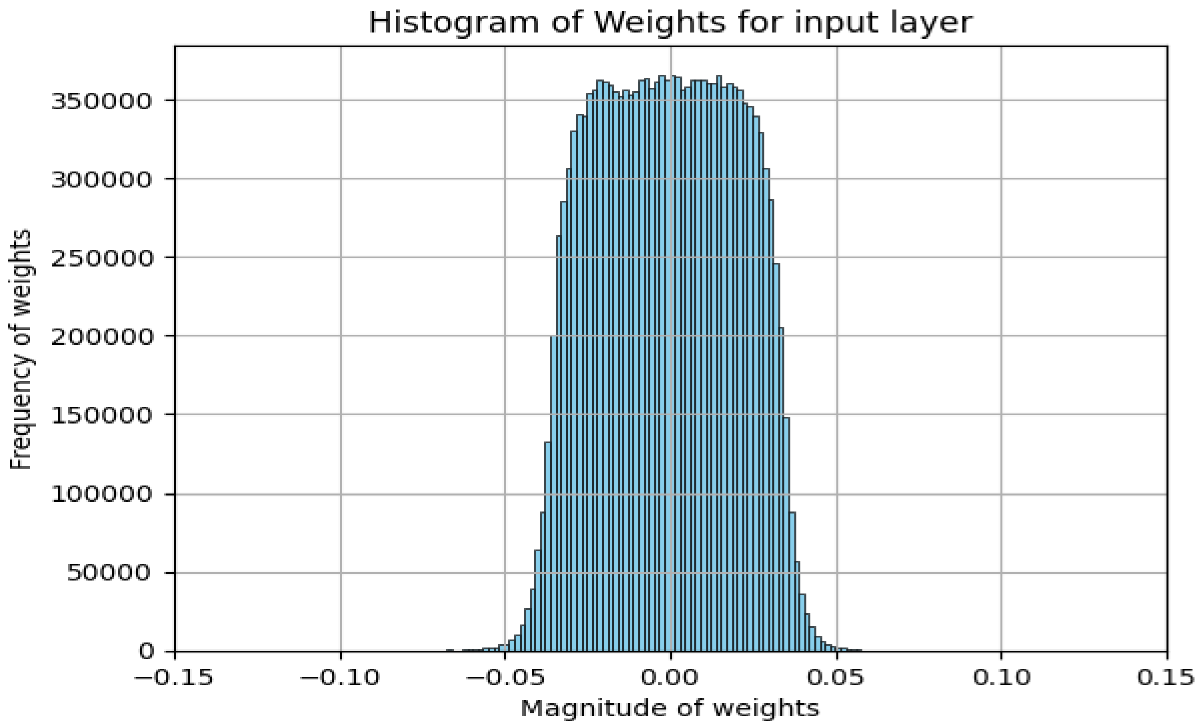
The histograms of weights in both the input and hidden layers will likely exhibit a more concentrated distribution in the model with L2 regularization (= 0.0001) and dropout (rate = 0.2) applied to the hidden layer, compared to the model without regularization.
L2 regularization tends to reduce weights to zero, discouraging excessively large weight values.
Dropout increases unpredictability during training by temporarily removing certain units, which might result in a sparser representation.
As a result, weights in the histograms for the model with regularization may be more centered around zero and may have a smaller spread than in the model without regularization.
The regularization strategies are designed to keep the model from becoming overly specialized to the training data, allowing for greater generalization to new data.

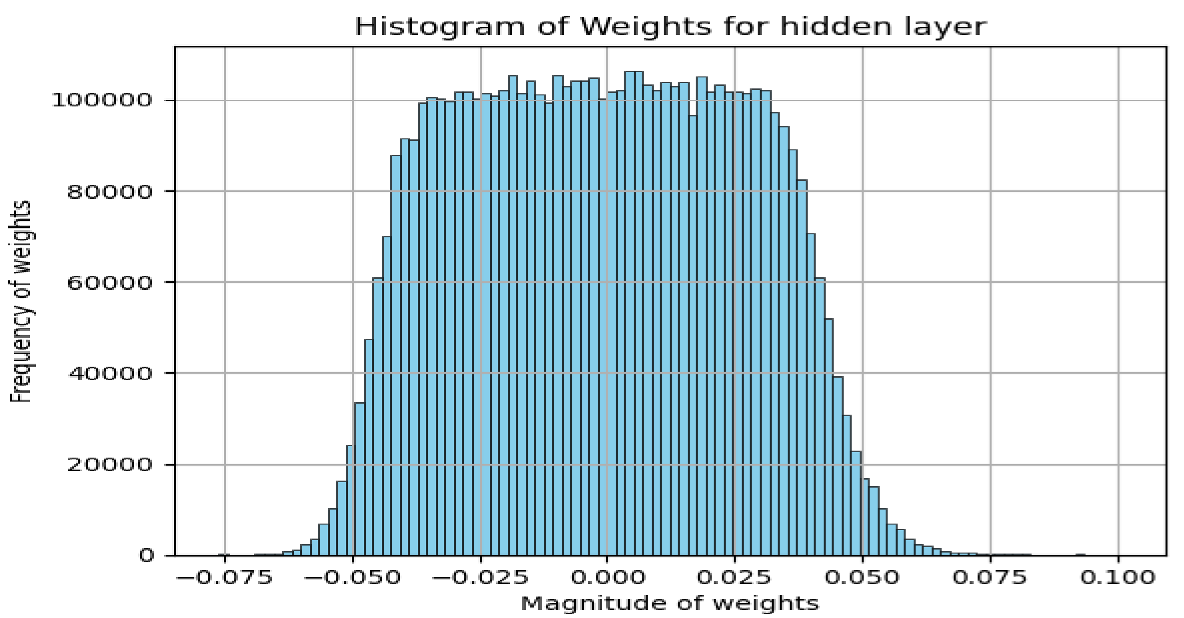
```
Epoch: 40, Training Loss: 0.3671,
Training Accuracy: 87.228%,
Test Accuracy: 85.650%
Training Completed
```





```
Epoch: 40,
Training Loss: 0.3271,
Training Accuracy: 88.115%,
Test Accuracy: 87.580%
Training Completed
```





APPENDIX:

1)

```
import torch
import torch.nn as nn
import torchvision
import matplotlib.pyplot as plt
import numpy as np

# Define the dataset and data loaders
transform =
torchvision.transforms.Compose([torchvision.transforms.ToTensor()])
train_set = torchvision.datasets.FashionMNIST(root="./data", train=True,
download=True, transform=transform)
test_set = torchvision.datasets.FashionMNIST(root="./data", train=False,
download=True, transform=transform)

batch_size = 100
train_loader = torch.utils.data.DataLoader(train_set,
batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
shuffle=False)

# Define the model with one hidden layer, ReLU activation, and 128 nodes
model = nn.Sequential(
    nn.Linear(in_features=28 * 28, out_features=128),
    nn.ReLU(),
    nn.Linear(in_features=128, out_features=10)
)

# Define the loss function and the optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

```
num_epochs = 40  # Set the number of epochs

# Training the model
train_acc_list = []
test_acc_list = []
train_loss_list = []
test_loss_list = []

input_weights_hist = []
hidden_weights_hist = []

for epoch in range(num_epochs):
    correct_train = 0
    running_loss_train = 0.0

    model.train()  # Set the model to training mode

    for x_batch, y_batch in train_loader:
        optimizer.zero_grad()

        x = x_batch.view(batch_size, -1)
        y_hat = model(x)

        loss_train = criterion(y_hat, y_batch)
        loss_train.backward()
        optimizer.step()

        running_loss_train += loss_train.item()
        correct_train += (y_batch == torch.max(y_hat, 1)[1]).sum().item()

    train_accuracy = 100 * correct_train / len(train_loader.dataset)
    train_loss_list.append(running_loss_train / len(train_loader))
    train_acc_list.append(train_accuracy)

    # Validation
    model.eval()  # Set the model to evaluation mode
    correct_test = 0
    running_loss_test = 0.0

    with torch.no_grad():
```

```
        for x_test, y_test in test_loader:
            output = model(x_test.view(batch_size, -1))

            loss_test = criterion(output, y_test)
            running_loss_test += loss_test.item()
            correct_test += (torch.max(output, 1)[1] ==
y_test).sum().item()

    test_accuracy = correct_test * 100 / len(test_loader.dataset)
    test_loss_list.append(running_loss_test / len(test_loader))
    test_acc_list.append(test_accuracy)

    # Store weights for histograms
    input_weights_hist.append(model[0].weight.detach().numpy().reshape(-1,
1))

hidden_weights_hist.append(model[1].weight.detach().numpy().reshape(-1,
1))

    print(f'Epoch: {epoch + 1:02d}, '
          f'Training Loss: {running_loss_train / len(train_loader):.4f}, '
          f'Training Accuracy: {train_accuracy:.3f}%, '
          f'Test Loss: {running_loss_test / len(test_loader):.4f}, '
          f'Test Accuracy: {test_accuracy:.3f}%')

print('Training Completed')

# Plotting histograms for the weights of the input and hidden layers
plt.hist(np.concatenate(input_weights_hist), bins=100, color='skyblue',
ec='black', lw=0.5)
plt.grid()
plt.xlim([-0.15, 0.15])
plt.xlabel('Magnitude of weights')
plt.ylabel('Frequency of weights')
plt.title('Histogram of Weights for input layer')

plt.figure()
plt.hist(np.concatenate(hidden_weights_hist), bins=100, color='skyblue',
ec='black', lw=0.5)
plt.grid()
```

```
plt.xlabel('Magnitude of weights')
plt.ylabel('Frequency of weights')
plt.title('Histogram of Weights for hidden layer')

plt.show()
```

2)

```
import torch
import torch.nn as nn
import torchvision
import matplotlib.pyplot as plt
import numpy as np

# Define the dataset and data loaders
transform =
torchvision.transforms.Compose([torchvision.transforms.ToTensor()])
train_set = torchvision.datasets.FashionMNIST(root="./data", train=True,
download=True, transform=transform)
test_set = torchvision.datasets.FashionMNIST(root="./data", train=False,
download=True, transform=transform)

batch_size = 100
train_loader = torch.utils.data.DataLoader(train_set,
batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
shuffle=False)

# Define the model with L2 regularization and dropout
model = nn.Sequential(
    nn.Linear(in_features=28 * 28, out_features=48),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(in_features=48, out_features=10)
)

# Define L2 regularization strength
l2_reg = 0.0001
# Add L2 regularization to linear layers
```

```
for layer in model.children():
    if isinstance(layer, nn.Linear):
        layer.weight.data.add_(layer.weight.data, alpha=-l2_reg)  # Use
negative alpha for subtraction

# Define the loss function and the optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

num_epochs = 40  # Set the number of epochs
max_stale_epochs = 10  # Set the maximum number of consecutive epochs with
no improvement on validation loss

# Training the model
train_acc_list = []
test_acc_list = []
train_loss_list = []
test_loss_list = []

input_weights_hist = []
hidden_weights_hist = []

best_validation_loss = float('inf')
stale_epochs = 0

for epoch in range(num_epochs):
    correct_train = 0
    running_loss_train = 0.0

    model.train()  # Set the model to training mode

    for x_batch, y_batch in train_loader:
        optimizer.zero_grad()

        x = x_batch.view(batch_size, -1)
        y_hat = model(x)

        loss_train = criterion(y_hat, y_batch)
        loss_train.backward()
        optimizer.step()
```

```
        running_loss_train += loss_train.item()
        correct_train += (y_batch == torch.max(y_hat, 1)[1]).sum().item()

    train_accuracy = 100 * correct_train / len(train_loader.dataset)
    train_loss_list.append(running_loss_train / len(train_loader))
    train_acc_list.append(train_accuracy)

    # Validation
    model.eval()  # Set the model to evaluation mode
    correct_test = 0
    running_loss_test = 0.0

    with torch.no_grad():
        for x_test, y_test in test_loader:
            output = model(x_test.view(batch_size, -1))

            loss_test = criterion(output, y_test)
            running_loss_test += loss_test.item()
            correct_test += (torch.max(output, 1)[1] ==
y_test).sum().item()

    test_accuracy = correct_test * 100 / len(test_loader.dataset)
    test_loss_list.append(running_loss_test / len(test_loader))
    test_acc_list.append(test_accuracy)

    # Store weights for histograms
    input_weights_hist.append(model[0].weight.detach().numpy().reshape(-1,
1))

hidden_weights_hist.append(model[1].weight.detach().numpy().reshape(-1,
1))

    print(f'Epoch: {epoch + 1:02d}, '
          f'Training Loss: {running_loss_train / len(train_loader):.4f}, '
          f'Training Accuracy: {train_accuracy:.3f}%, '
          f'Test Loss: {running_loss_test / len(test_loader):.4f}, '
          f'Test Accuracy: {test_accuracy:.3f}%')

    # Early stopping check
```

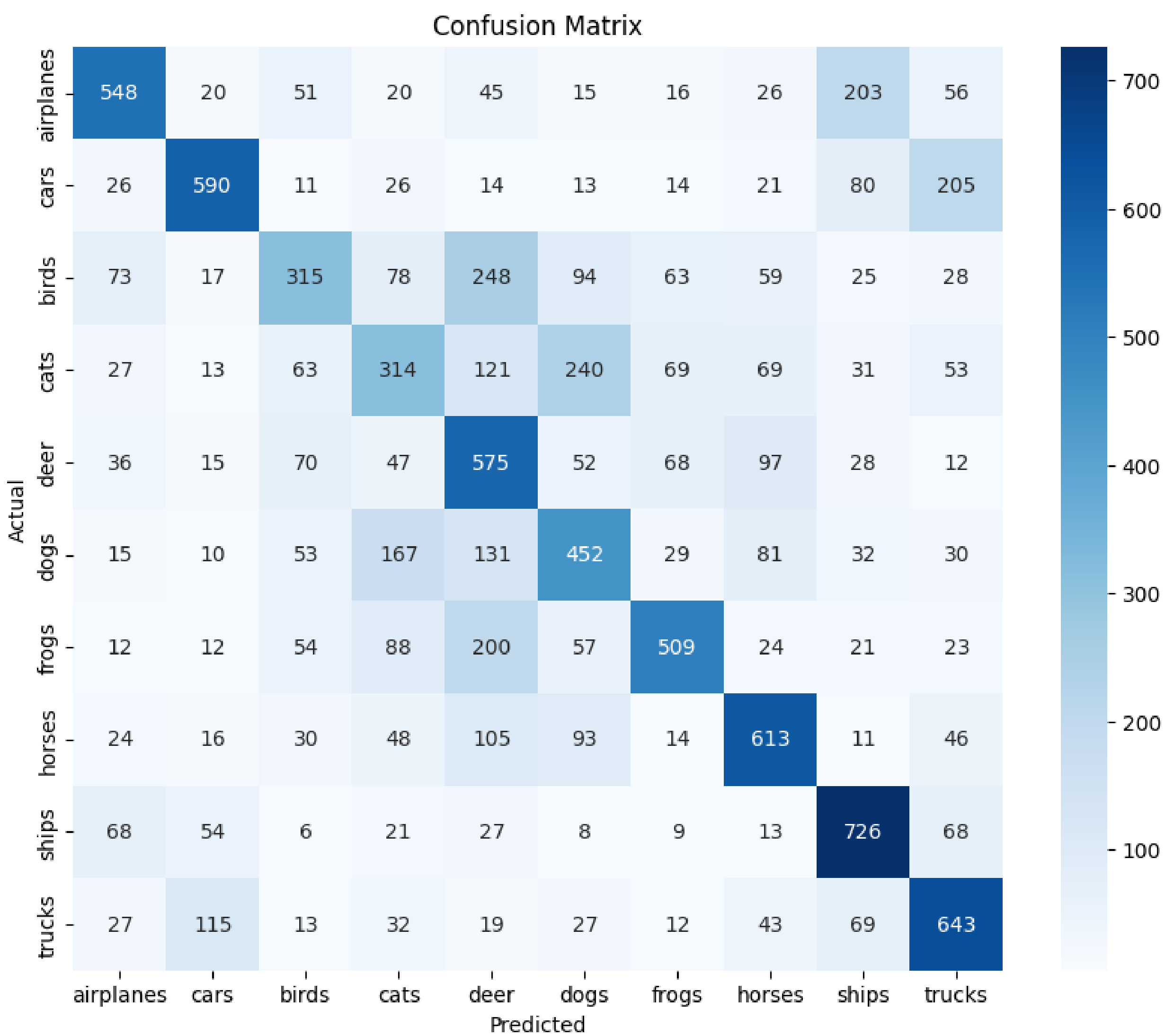
```
    if running_loss_test < best_validation_loss:
        best_validation_loss = running_loss_test
        stale_epochs = 0
    else:
        stale_epochs += 1

    if stale_epochs >= max_stale_epochs:
        print(f'Early stopping: No improvement on validation loss for
{max_stale_epochs} consecutive epochs.')
        break

print('Training Completed')

# Plotting histograms for the weights of the input and hidden layers
plt.hist(np.concatenate(input_weights_hist), bins=100, color='skyblue',
ec='black
```

3. Train an MLP for CIFAR-10 (https://www.cs.toronto.edu/~kriz/cifar.html). The CIFAR-10 dataset consists of 60000 32 × 32 colour images in 10 classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each 32 × 32 image consists of three color channels (Red, Green, and Blue) for a total of 3 × 32 × 32 = 3072 pixels. There are 50000 training images and 10000 test images. Use two hidden layers and ReLU activations. Use 256 nodes in the first hidden layer and 128 nodes in the second hidden layer. Use a dropout layer for each hidden layer output with 30% dropout rate and use an L2 regularizer with coefficient $\lambda = 0.0001$. You may use the demo scripts from lecture and discussion. Evaluate this trained model on the test data and compute a confusion matrix. Use seaborn or similar python package to generate a heatmap showing the confusion matrix.
(a) Consider class m. List the class most likely confused for class m for each object type.
(b) Which two classes (object types) are most likely to be confused overall?

a)
Class 0 - Airplanes - The class airplanes got most confused with class ships and birds
Class 1 - Cars - The class cars got most confused with class trucks
Class 2 - Birds - The class birds got most confused with class deer
Class 3 - Cats - The class cats got most confused with class dogs
Class 4 - Deer - The class deer got most confused with class birds
Class 5 - Dogs - The class dogs got most confused with class cats
Class 6 - Frogs - The class frogs got most confused with class deer

Class 7 - Horses - The class horses got most confused with class deer
Class 8 - Ships - The class ships got most confused with class airplanes
Class 9 - Trucks - The class trucks got most confused with class cars

b)
The two classes that got confused overall are birds and deer



APPENDIX:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
```

```
import torch.nn.functional as F
import matplotlib.pyplot as plt
import seaborn as sn
from sklearn.metrics import confusion_matrix

# Class labels
labels = ['airplanes', 'cars', 'birds', 'cats', 'deer', 'dogs', 'frogs',
'horses', 'ships', 'trucks']

# Importing Dataset
transform = transforms.ToTensor()
train_data = torchvision.datasets.CIFAR10(root="./data", train=True,
download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root="./data", train=False,
download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=100,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100,
shuffle=False)

# Define the model
model = nn.Sequential(
    nn.Linear(in_features=3 * 32 * 32, out_features=256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(in_features=256, out_features=128),
    nn.ReLU(),
    nn.Dropout(0.3)
)

# Define loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01,
weight_decay=0.0001)

# Number of epochs
num_epochs = 100

# Training the model
```

```python
count = 0
accuracy_train_list = []
accuracy_test_list = []
loss_train_list = []
loss_test_list = []

for epoch in range(num_epochs):
    correct = 0
    model.train()

    for x_batch, y_batch in train_loader:
        count += 1
        x = x_batch.view(x_batch.size(0), -1)
        y_hat = model(x)
        loss_train = loss_fn(y_hat, y_batch)

        optimizer.zero_grad()
        loss_train.backward()
        optimizer.step()

        predictions = torch.max(y_hat, 1)[1]
        correct += (y_batch == predictions).sum().item()

    with torch.no_grad():
        total_test = 0
        correct_test = 0

        model.eval()
        for x_test, y_test in test_loader:
            x_test = x_test.view(x_test.size(0), -1)
            output = model(x_test)
            prediction = torch.max(output, 1)[1]
            loss_test = loss_fn(output, y_test)
            correct_test += (prediction == y_test).sum().item()
            total_test += len(y_test)

    accuracy_test = correct_test * 100 / total_test

    loss_train_list.append(loss_train.item())
    loss_test_list.append(loss_test.item())
```

```
    accuracy_train_list.append(100 * correct / len(train_loader.dataset))
    accuracy_test_list.append(accuracy_test)

    print(f'Epoch: {epoch+1:02d}, Iteration: {count: 5d}, Loss:
{loss_train.item():.4f}, '
          f'Accuracy: {100*correct/len(train_loader.dataset):2.3f}%')

print('Completed')

# Computing the confusion matrix
all_preds = []
all_targets = []

for x_test, y_test in test_loader:
    model.eval()
    output2 = model(x_test.view(x_test.size(0), -1))
    prediction2 = torch.max(output2, 1)[1]

    # Accumulate predictions and ground truth labels
    all_preds.extend(prediction2.numpy())
    all_targets.extend(y_test.numpy())

# Generating heat map of confusion matrix
cf_matrix = confusion_matrix(all_targets, all_preds)

plt.figure(figsize=(10, 8))
sn.heatmap(cf_matrix, fmt='0', annot=True, cmap="Blues",
xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```