5241_Project

May 8, 2022

1 Stat 5241 Statistical Machine Learning Final Project

1.1 Author: Gexin Chen Uni: gc2936

2 I. BackGround

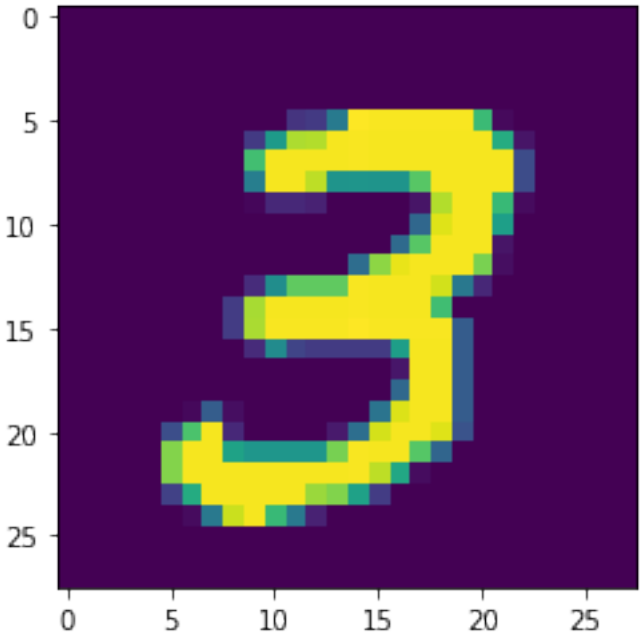
The MNIST database of handwritten digits is one of the most commonly used dataset for training various image processing systems and machine learning algorithms. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. MNIST is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bilevel) images from NIST were size normalized. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28 $\times$28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28 $\times$28 field.

3 II. Data Processing

3.1 1.(a)

[ ]:
```
# Plot The Feature Parameters
imshow(X_train[7]);
# Return the Output
print(Y_train[7])
```

```
3
```



From above image, we can see the number is '3' which matches the label in Y_train.

3.2 1.(b)

[ ]:
```
# Data Reshape
X_train = X_train.reshape(-1,28*28)
X_test = X_test.reshape(-1, 28*28)
# Data Normalization
X_train = preprocessing.normalize(X_train)
X_test = preprocessing.normalize(X_test)
```

3.3 1.(c)

[ ]:
```
# One Hot Encoding
onehot = preprocessing.OneHotEncoder(sparse = False)
onehot.fit(Y_train.reshape(-1,1))
Y_train_oh = onehot.transform(Y_train.reshape(-1,1))
Y_test_oh = onehot.transform(Y_test.reshape(-1,1))
```

One hot embedding transfer every labels to 1*N matrix containing only 0 and 1. In this way, each label can be treated equally by machine learning algorithm, which is difficult to be achieved by using numerical values to represent classes.

4 III. Before Deep Learning

4.1 2.(a)

[ ]:
```
# KNN Training With K = 3
KNN = KNeighborsClassifier(n_neighbors=3).fit(X_train, Y_train_oh)
1 - KNN.score(X_test, Y_test_oh)
```

[ ]: 0.02729999999999999

[ ]:
```
# Adaboost Training
ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=20),␣
↪n_estimators=30).fit(X_train, Y_train)
1 - ada.score(X_test, Y_test)
```

[ ]: 0.03720000000000001

[ ]:
```
# Decision Tree with Max_depth = 20
dt = DecisionTreeClassifier(max_depth = 20).fit(X_train, Y_train)
1 - dt.score(X_test, Y_test)
```

[ ]: 0.11280000000000001

[ ]:
```
# SVM with Gaussian Kernel
svm = SVC().fit(X_train, Y_train)
1 - svm.score(X_test, Y_test)
```

[ ]: 0.018900000000000028

From above, we can see that the test errors of all three algorithms are different from the given result.
KNN: 2.73% Test Error vs. 5% Desired Test Error
AdaBoost.M1: 3.72% Test Error vs. 4.05% Desired Test Error
Decision Tree: 11.3% Test Error vs. 4.05% Desired Test Error
SVM with Gaussian Kernel: 1.9% Test Error vs. 1.4% Test Error

It can be caused by multiple reasons:
1. The data we used above may be different from the data authors used; 2. The preprocessing procedures are different (We have done normalization in Part1); 3. The hyperparameters for the four algorithms may be different (KNN with K = 3, AdaBoost & Decision Tree with maximum depth = 20, SVM with C = 1);
4. The train test split method applied on the dataset may be different.

4.2 2.(b)

```
[ ]: #SVM with Gaussian Kernel
svm1 = SVC(C = 2).fit(X_train, Y_train)
1 - svm1.score(X_test, Y_test)
```

```
[ ]: 0.015800000000000036
```

From above, we can see that the testing error for SVM with Gaussian Kernel with C = 2 is 1.5%, which is lower than the test errors of all classifiers in part (a).

5 IV. Deep Leaning

5.1 3.(a) & (b)

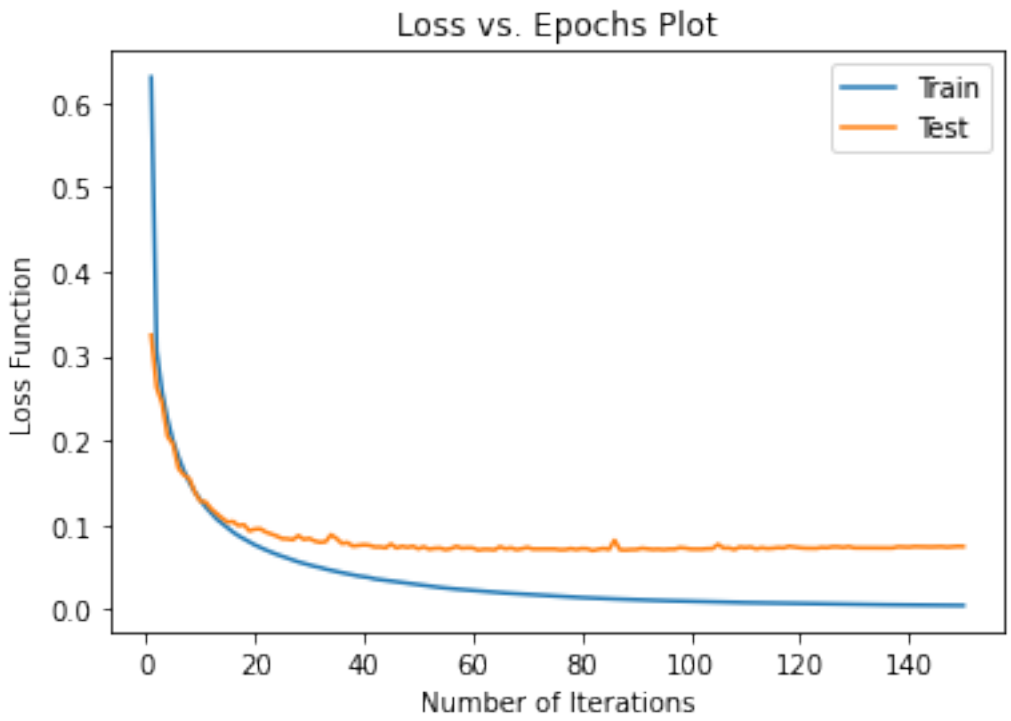
The first model is a single layer neural network with 100 hidden units. You can see the structure of algortithm as below.

```
[ ]: # Create a Fully Connected Neural Network With Linear Layer
class ANN(nn.Module):
    def __init__(self, input_size, num_classes):
        super(ANN, self).__init__()
        self.flatten = nn.Flatten()
        # 100 Hidden Neurons
        self.layer1 = nn.Linear(input_size, 100)
        # 10 Output Classes
        self.layer2 = nn.Linear(100, num_classes)
    def forward(self, x):
        x = self.flatten(x)
        # Sigmoid As Activation Function
        x = torch.sigmoid(self.layer1(x))
        x = self.layer2(x)
        return x
```

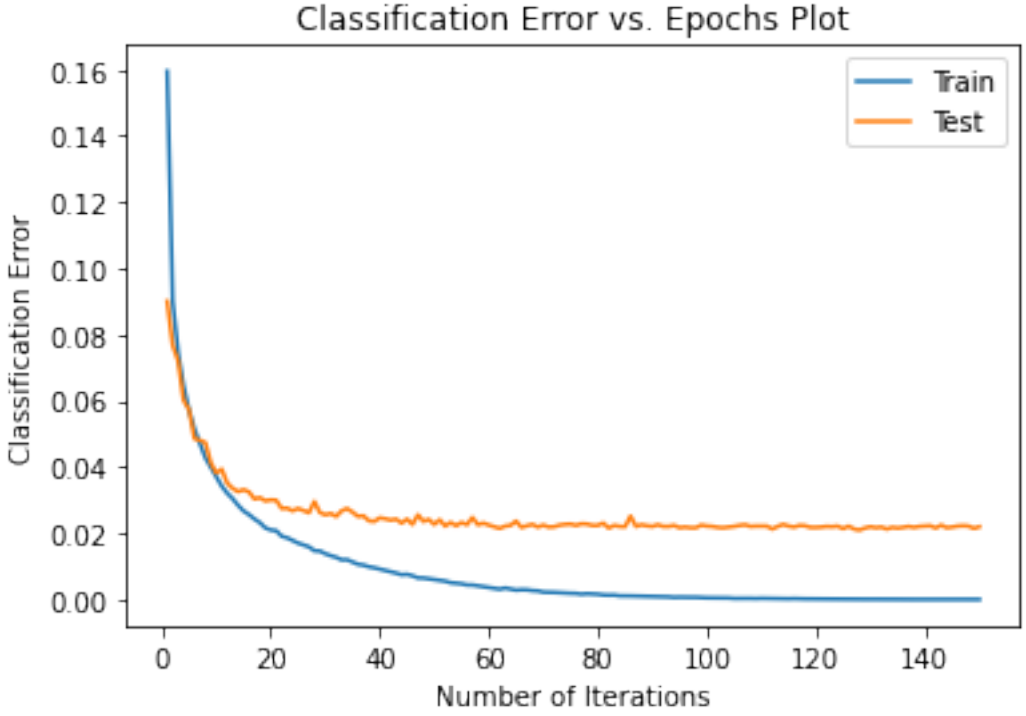
5.1.1 Seed 1

```
[ ]: # Define HyperParameters
input_size = 784
num_classes = 10
learning_rate = 0.1
num_epochs = 150
model1 = ANN(input_size, num_classes)

# Define Loss & Optimizer
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model1.parameters(), lr = learning_rate)
```

```
[ ]: loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



5.1.2 Seed 2

```
[ ]: # Define HyperParameters
learning_rate = 0.1
model2 = ANN(input_size, num_classes)
optimizer = optim.SGD(model2.parameters(), lr = learning_rate)
```

5.1.3 Seed 3

```
[ ]: # Define HyperParameters
learning_rate = 0.1
model3 = ANN(input_size, num_classes)
optimizer = optim.SGD(model3.parameters(), lr = learning_rate)
```

5.1.4 Seed 4

```
[ ]: # Define HyperParameters
learning_rate = 0.1
model4 = ANN(input_size, num_classes)
optimizer = optim.SGD(model4.parameters(), lr = learning_rate)
```

5.1.5 Seed 5

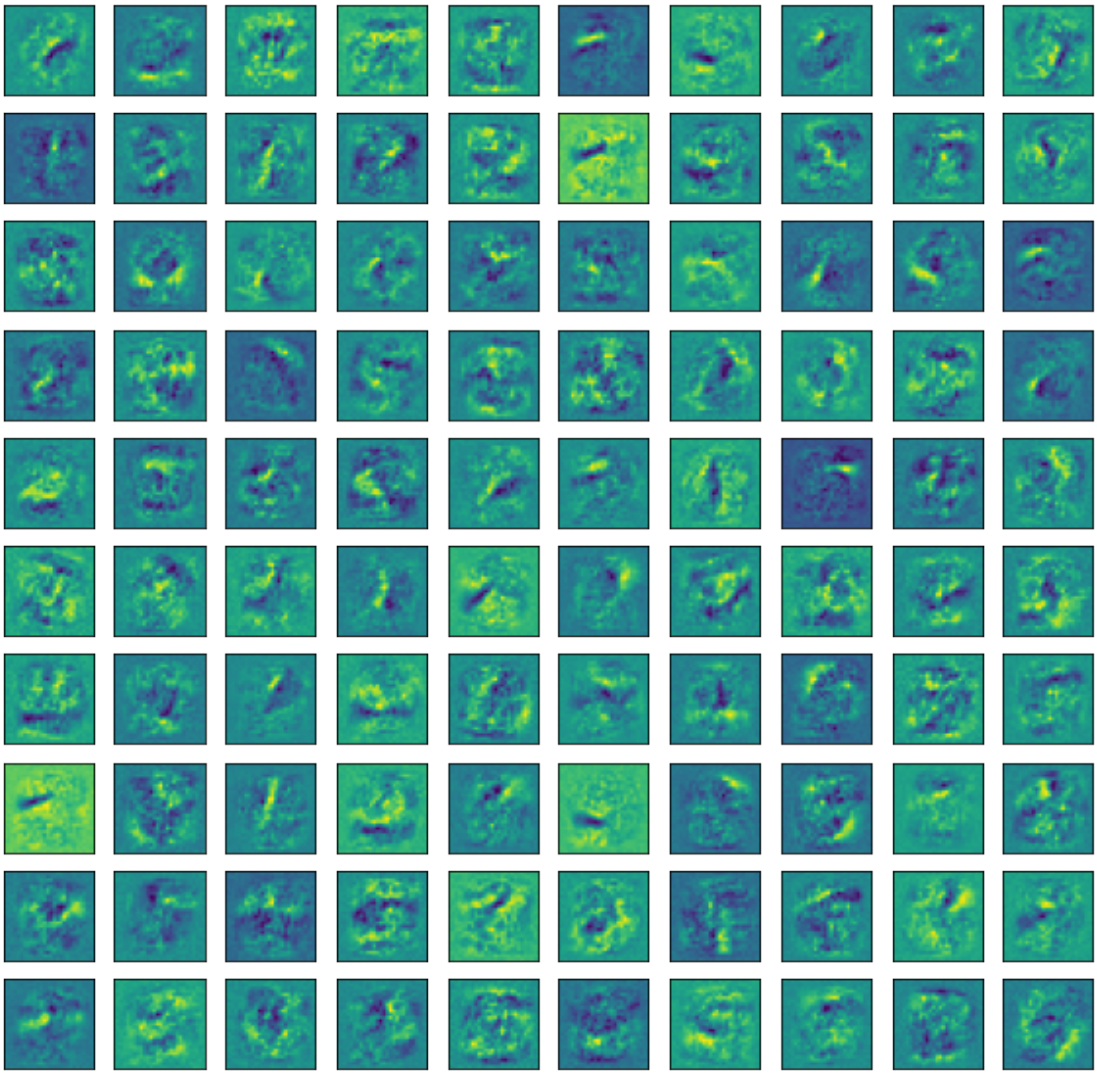
```
[ ]: # Define HyperParameters
learning_rate = 0.1
model5 = ANN(input_size, num_classes)
optimizer = optim.SGD(model5.parameters(), lr = learning_rate)
```

5.2 3.(c)

We initialize five different random seeds and randomly shuffle our training set. It can be observed that the training progress of five different models are similar.
We choose our best paramaters based on the highest testing accuracy and lowest loss on test set, which is the parameters set from seed 3.
Testing Accuracy:
Seed 1: 97.8%;
Seed 2: 97.85%;
Seed 3: 97.91%;
Seed 4: 97.78%;
Seed 5: 97.93%.

```
[ ]: # Return the Weights for First Linear Layers (784 * 100)
weights = model3.state_dict()['layer1.weight']
plt.figure(figsize = (10,10))
for i in range(weights.shape[0]):
```

```
plt.subplot(10,10,i+1)
# Total of 100 figures of dimension 28 * 28
plt.imshow(weights[i].reshape(28,28))
plt.xticks([])
plt.yticks([])
```



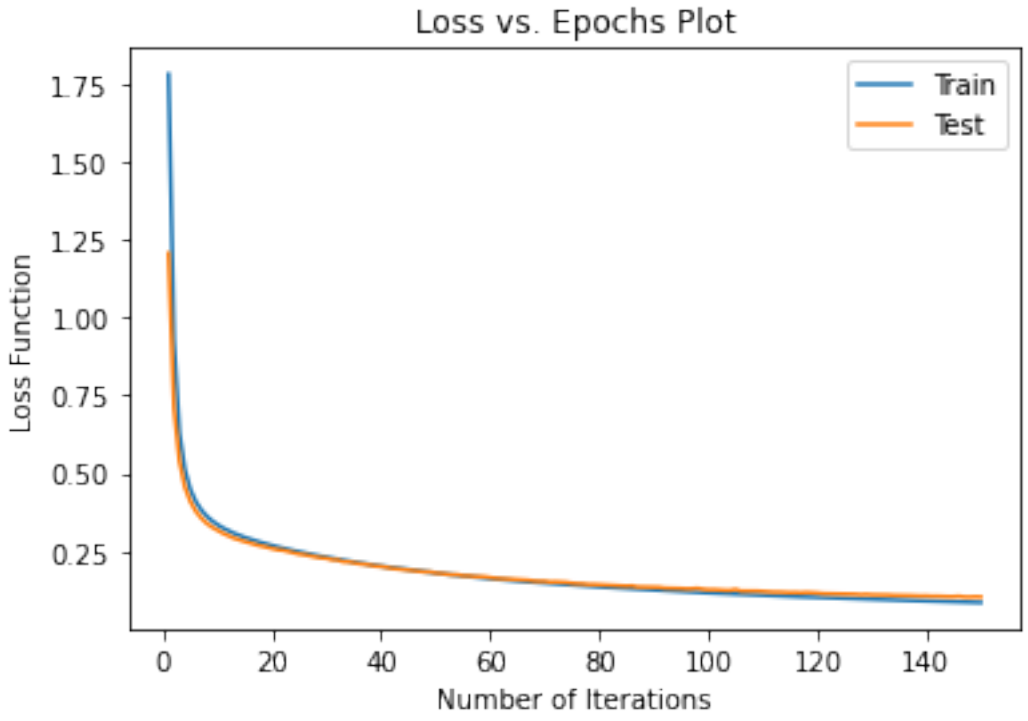
WE can see that the weight visualization is neither too noisy nor too correlated.

5.3 3.(d)

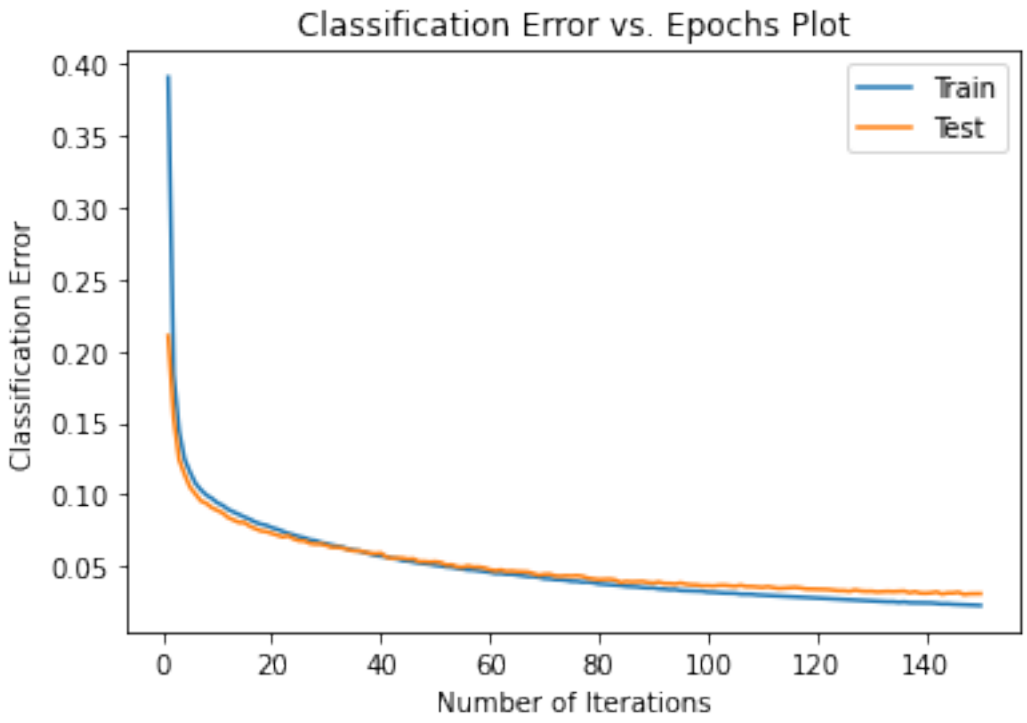
5.3.1 Learning Rate = 0.01

[ ]:
```
# Define HyperParameters
learning_rate = 0.01
model_lr01 = ANN(input_size, num_classes)
optimizer = optim.SGD(model_lr01.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



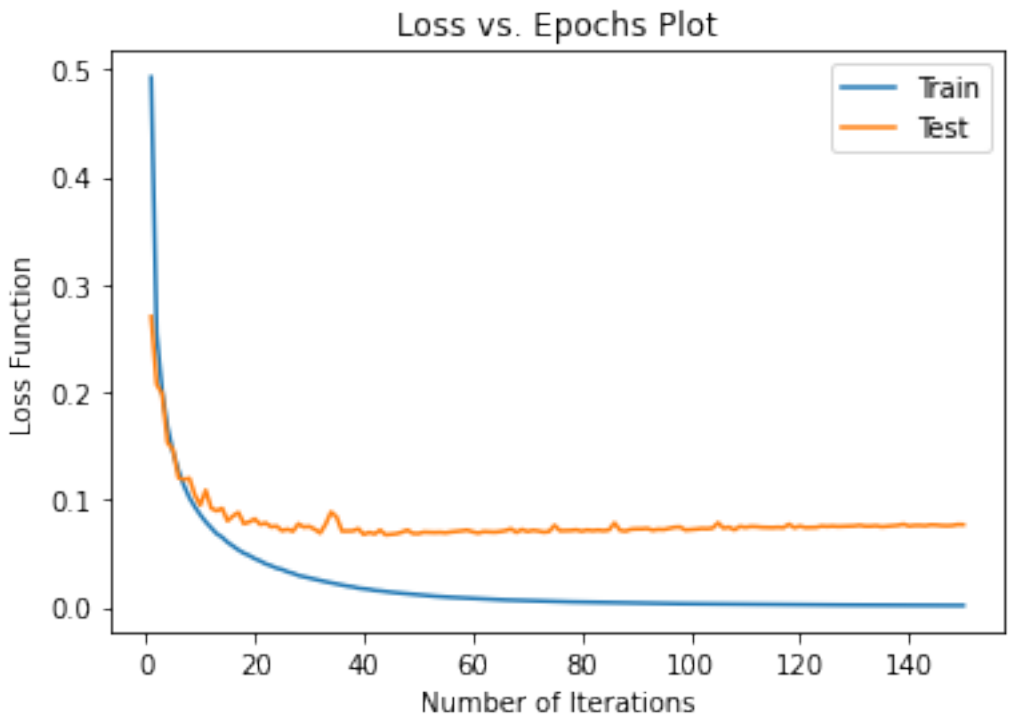
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



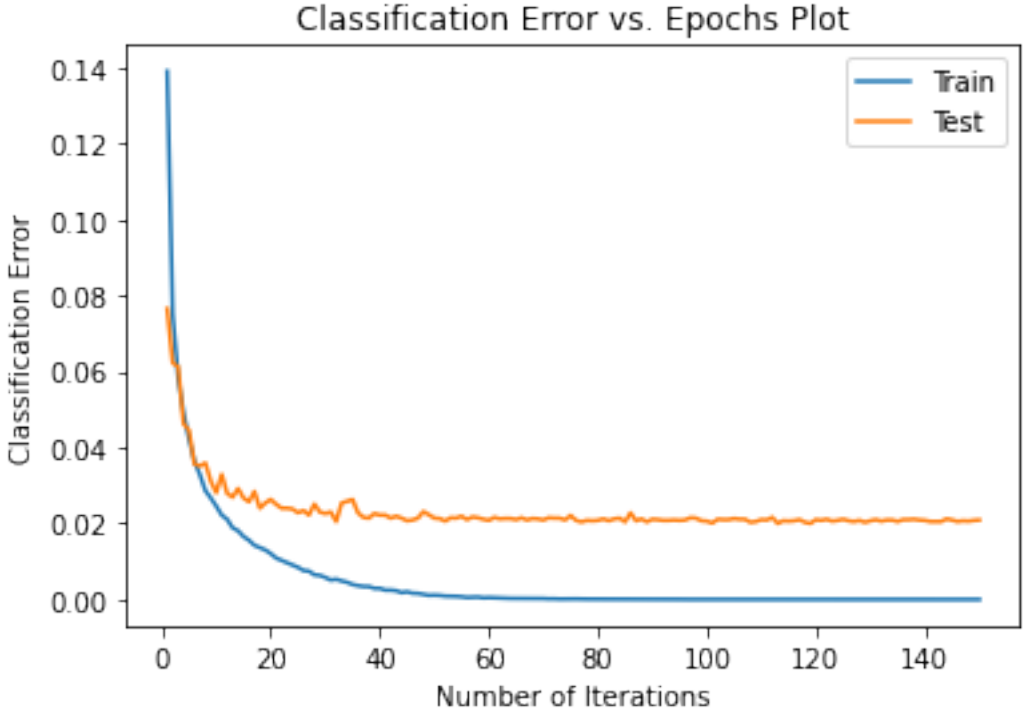
5.3.2 Learning Rate = 0.2

[ ]:
```
# Define HyperParameters
learning_rate = 0.2
model_lr2 = ANN(input_size, num_classes)
optimizer = optim.SGD(model_lr2.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



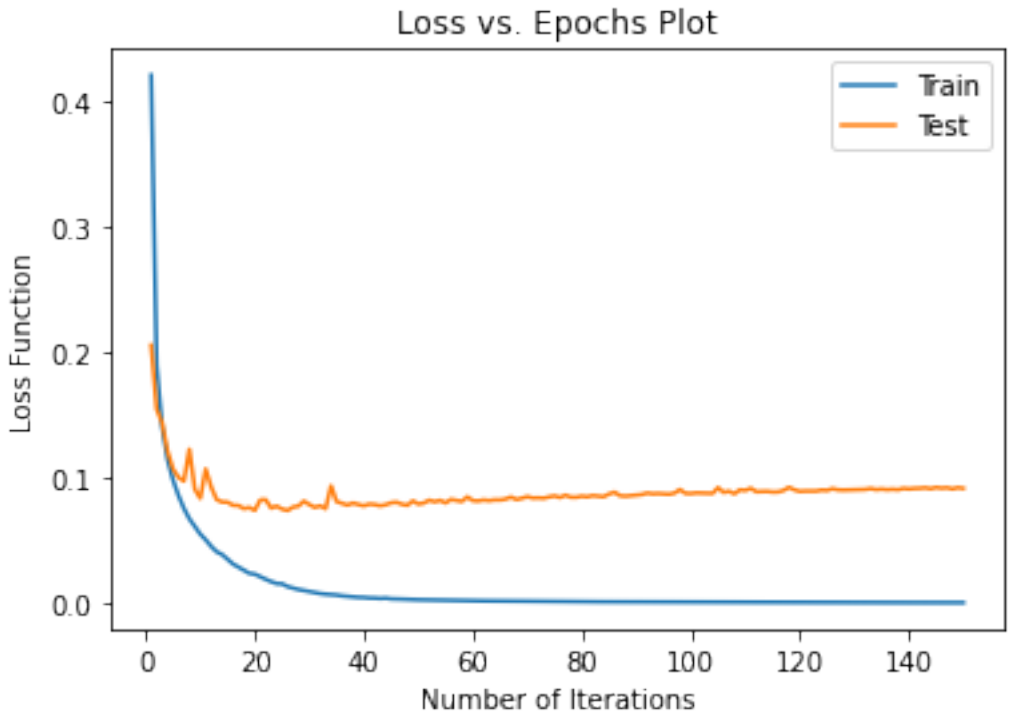
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



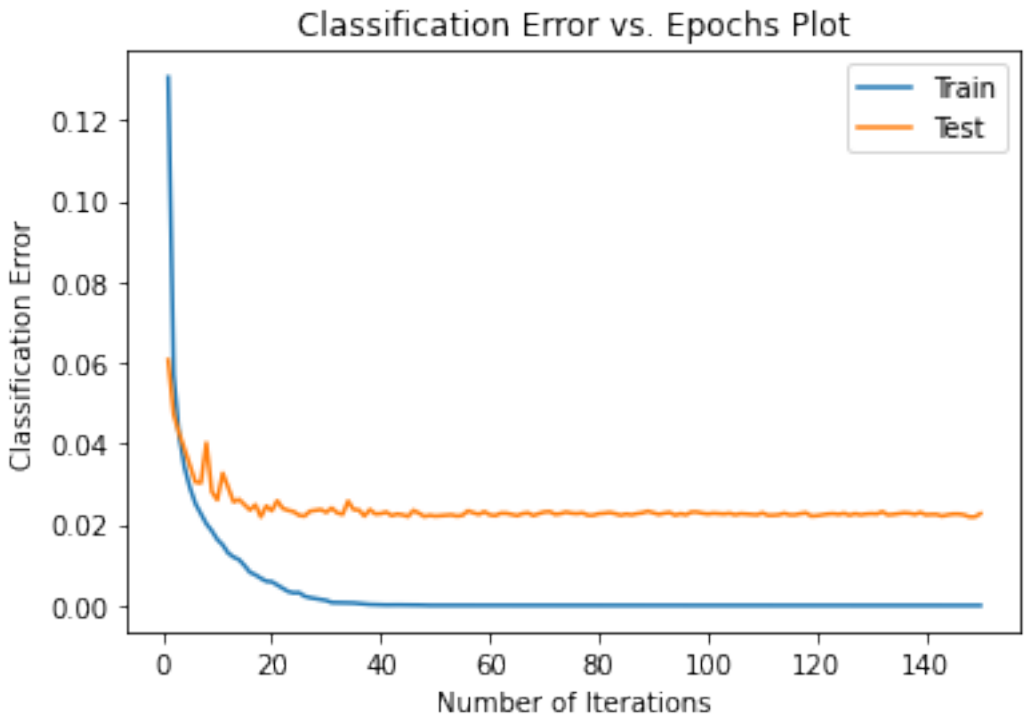
5.3.3 Learning Rate = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.5
model_lr5 = ANN(input_size, num_classes)
optimizer = optim.SGD(model_lr5.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



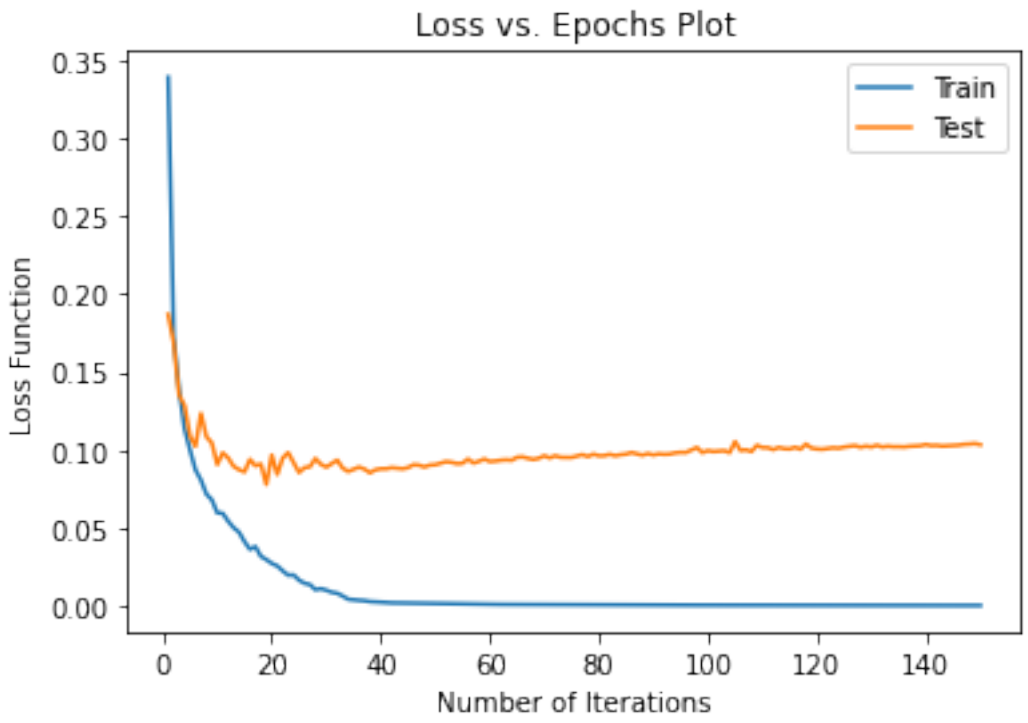
5.3.4 Momentum = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.5
model_m5 = ANN(input_size, num_classes)
optimizer = optim.SGD(model_m5.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

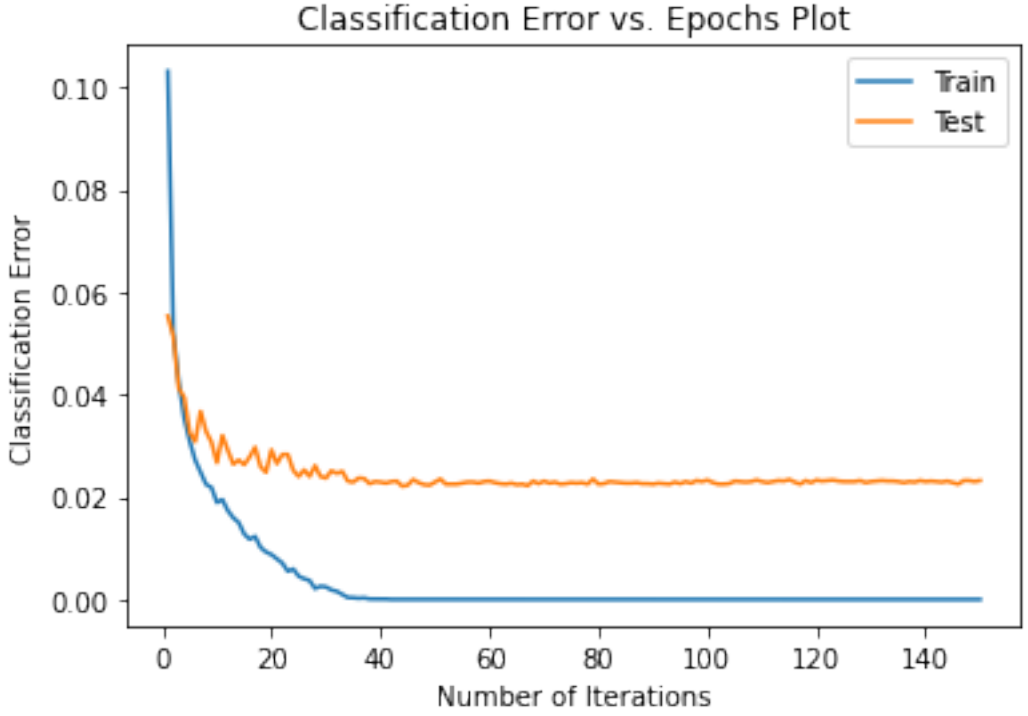
5.3.5 Momentum = 0.9

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.9
model_m9 = ANN(input_size, num_classes)
optimizer = optim.SGD(model_m9.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```

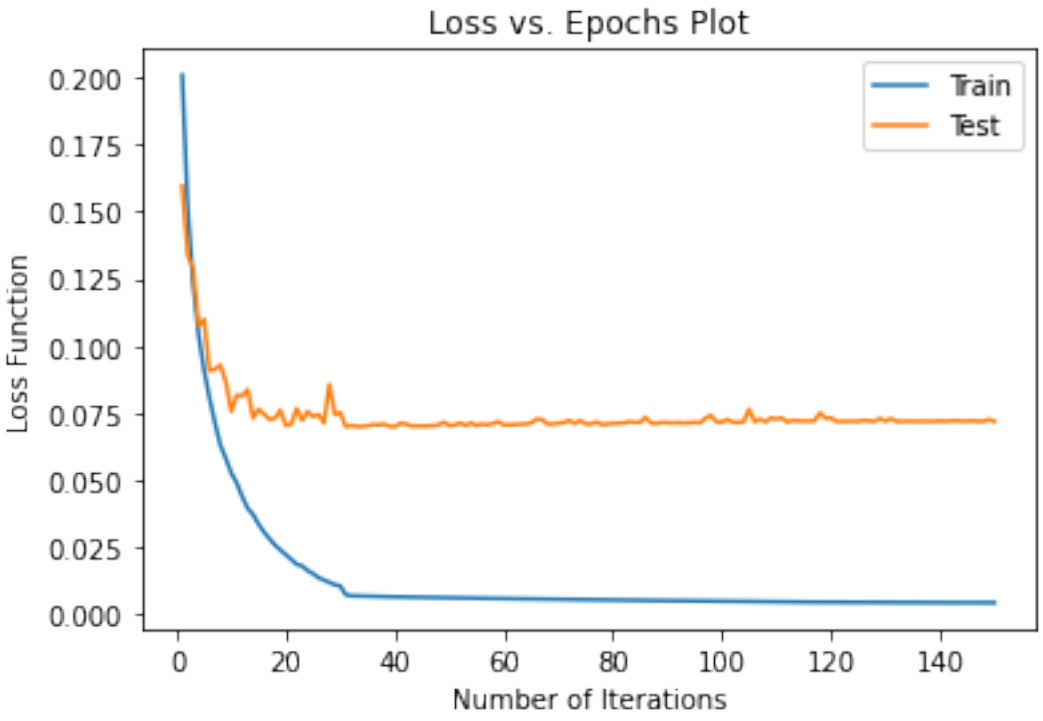


5.3.6 Best Combination of Hyperparameters

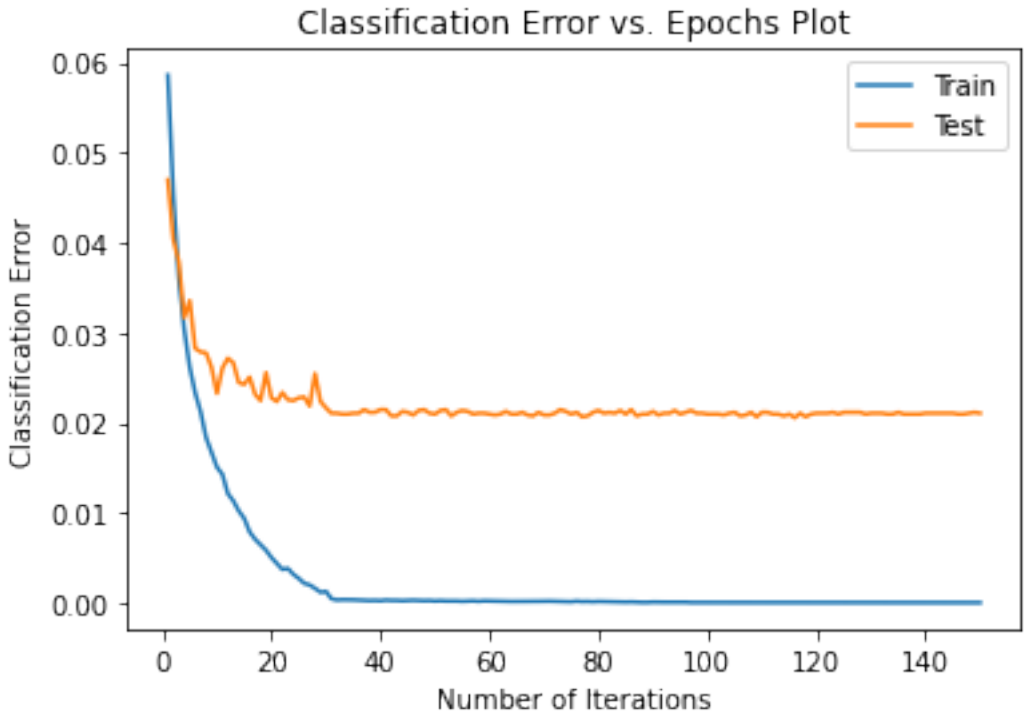
From above plots, it can be observed that:
1.With a total number of 150 epochs, learning rate = 0.2, 0.5, momentum = 0.5, 0.9 combined with learning rate = 0.1, the models all suffer from overfitting. In another word, the testing loss and testing missclassification increase at the end of training process;
2.With learning rate = 0.01, the convergence process is slow and inefficient.
Therefore, in terms of efficiency and stability in local minimum, we choose a decaying learning rate which start at 0.2 at the beginning and decay to 0.02 after 30 epochs and 0.002 after 120 epochs to ensure both efficiency and stability.

[ ]:
```
# Define HyperParameters
learning_rate = 0.2
momentum = 0.5
best_ANN = ANN(input_size, num_classes)
optimizer = optim.SGD(best_ANN.parameters(), lr = learning_rate, momentum =␣
↪momentum)
scheduler1 = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30,120],␣
↪gamma=0.1)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



Test Accuracy As Follow:
Learning Rate = 0.01: 96.92%;
Learning Rate = 0.2: 97.91%;
Learning Rate = 0.5: 97.73%;
Momentum = 0.5 & Learning Rate = 0.1: 97.69%;
Momentum = 0.9 & Learning Rate = 0.1: 97.68%;
Decaying Learning Rate 0.2/0.02/0.002: 97.89%.
The highest accuracy is obtained from the model with learning rate = 0.2 desipite of the sudden increase in loss and misclassification error. In contrast, decaying learning rate avoid sudden increase in loss and result in high enough accuracy.

5.4 4.(a) & (b)

Secondly, we construct a convolutional neural network with two convolutional layers and some linear layers for dimension reduction. The number of filters in the first convolutional layer is 64, is 16 for the sencond layer.

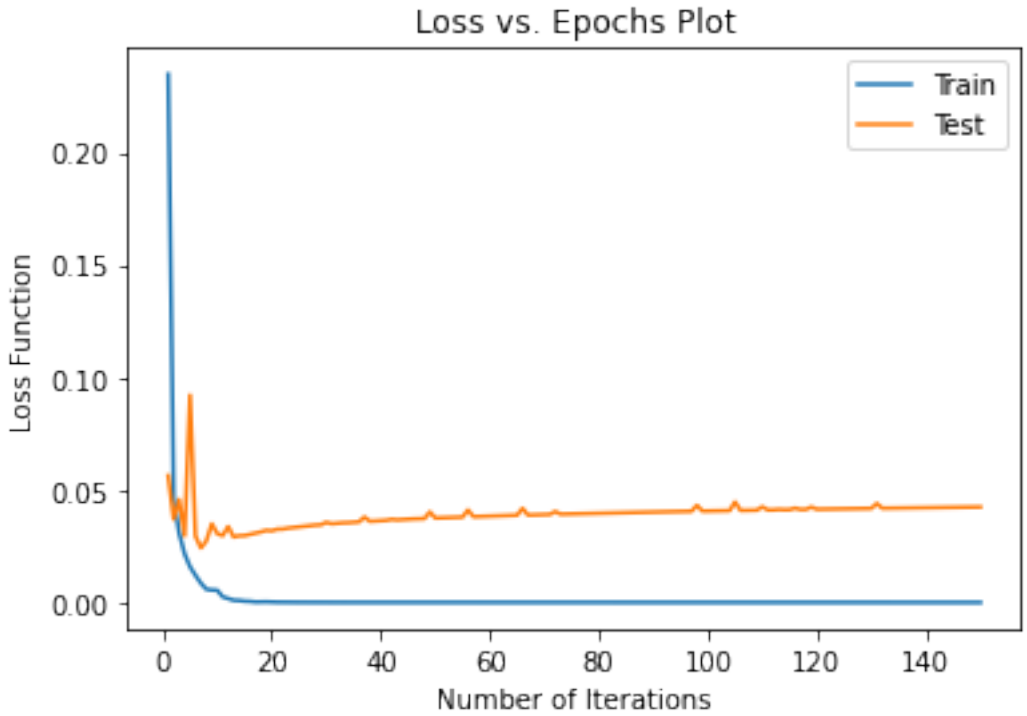
[ ]:
```
# Create a Convolutional Neural Network
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__() # Input Shape 1 * 28 * 28
        self.cv1 = nn.Conv2d(1, 64, 5) # Ouput Shape 64 * 24 * 24
```

```
        self.cv2 = nn.Conv2d(64, 16, 3) # Output Shape 16 * 22 * 22
        self.pool1 = nn.MaxPool2d(3, stride = 2) # Output Shape 16 * 10 * 10
        self.fc1 = nn.Linear(1600, 800) # Output Shape 800
        self.fc2 = nn.Linear(800, 256) # Output Shape 256
        self.fc3 = nn.Linear(256, num_classes) # Output Shape: num_classes

    def forward(self, x):
        x = F.relu(self.cv1(x))
        x = F.relu(self.cv2(x))
        x = self.pool1(x)
        x = x.reshape(-1, 1600)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

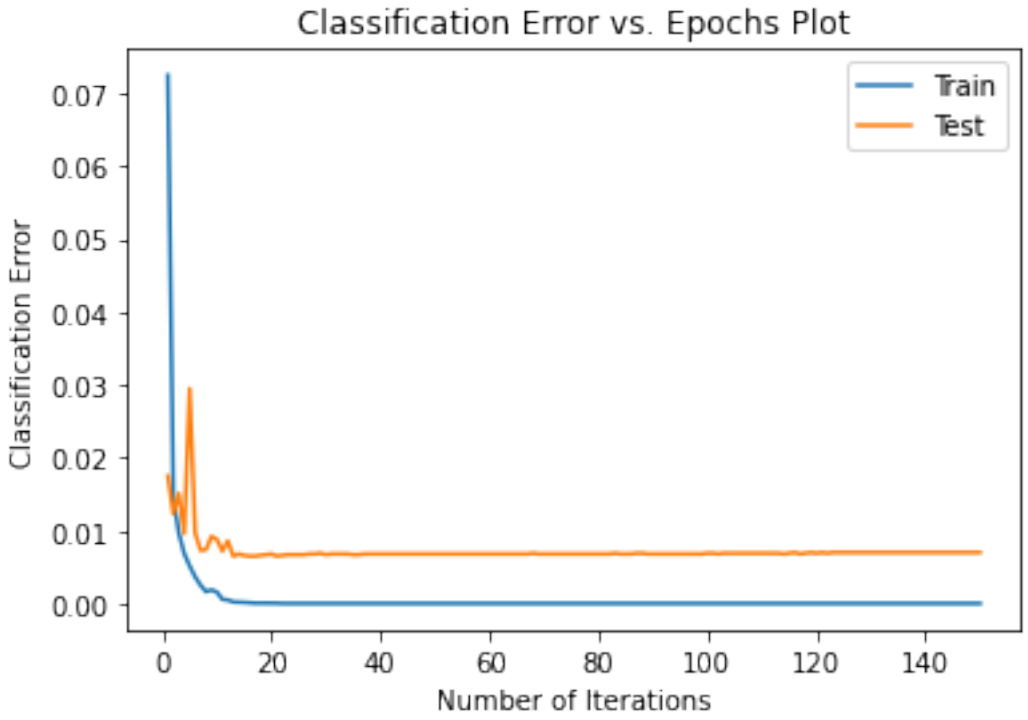
5.4.1 Seed 1

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
CNN1 = CNN(num_classes)
optimizer = optim.SGD(CNN1.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```

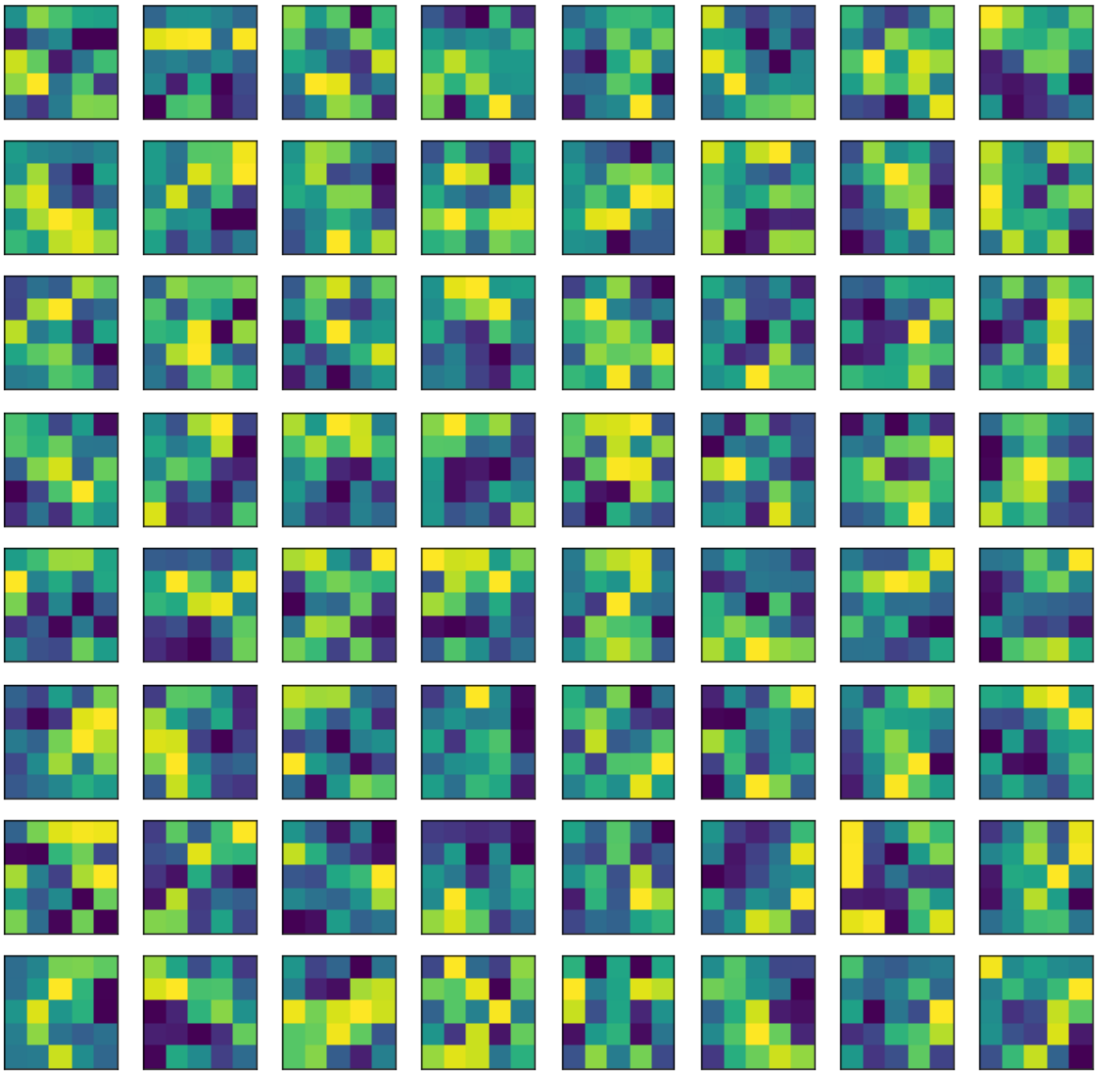


[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



5.5 4.(c)

[ ]:
```
weights = CNN1.state_dict()['cv1.weight']
plt.figure(figsize = (10,10))
for i in range(len(weights)):
  plt.subplot(8,8,i+1)
  plt.imshow(weights[i].reshape(5,5))
  plt.xticks([])
  plt.yticks([])
```



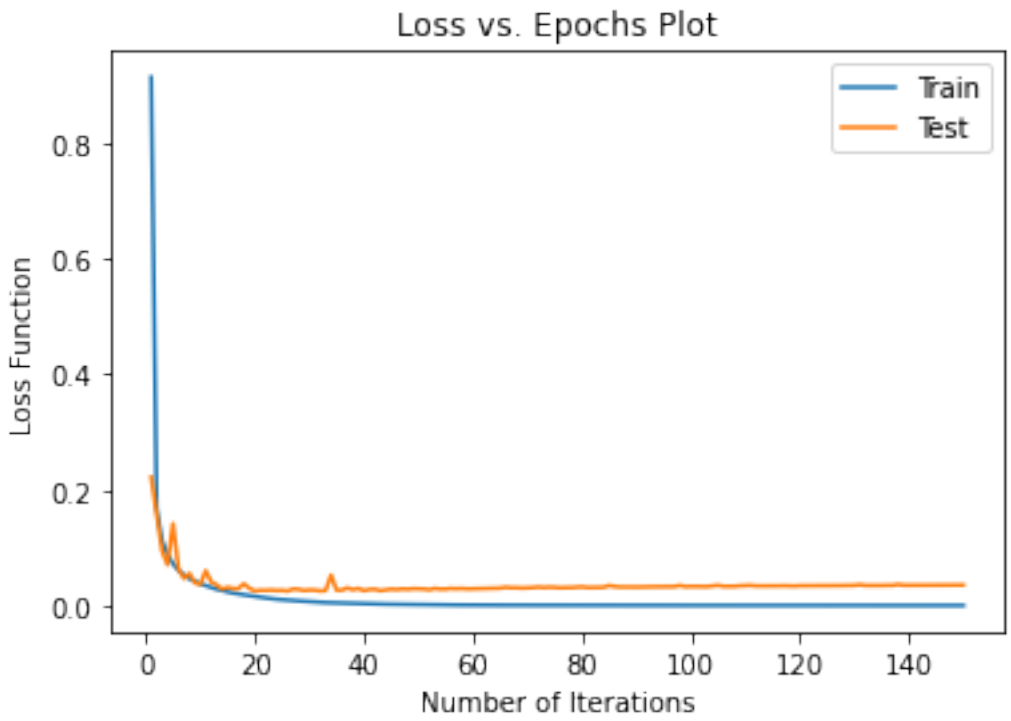
We can see that the weight visualization is neither noisy nor correlated.

5.6 4.(d)

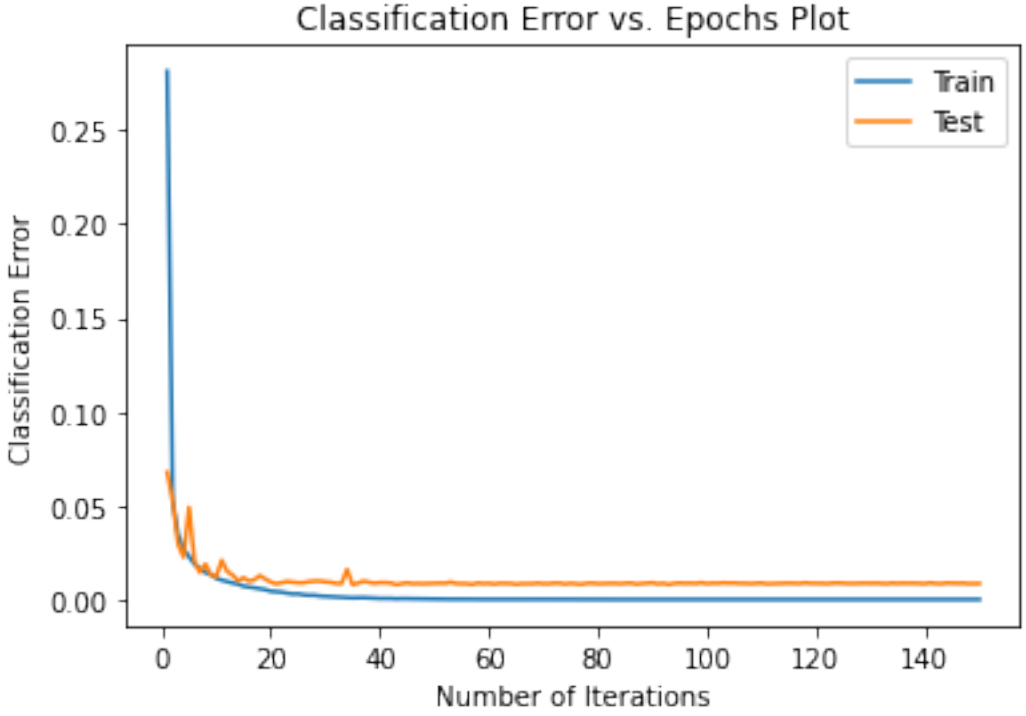
5.6.1 Learning Rate = 0.01

[ ]:
```
# Define HyperParameters
learning_rate = 0.01
CNN_LR01 = CNN(num_classes)
optimizer = optim.SGD(CNN_LR01.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



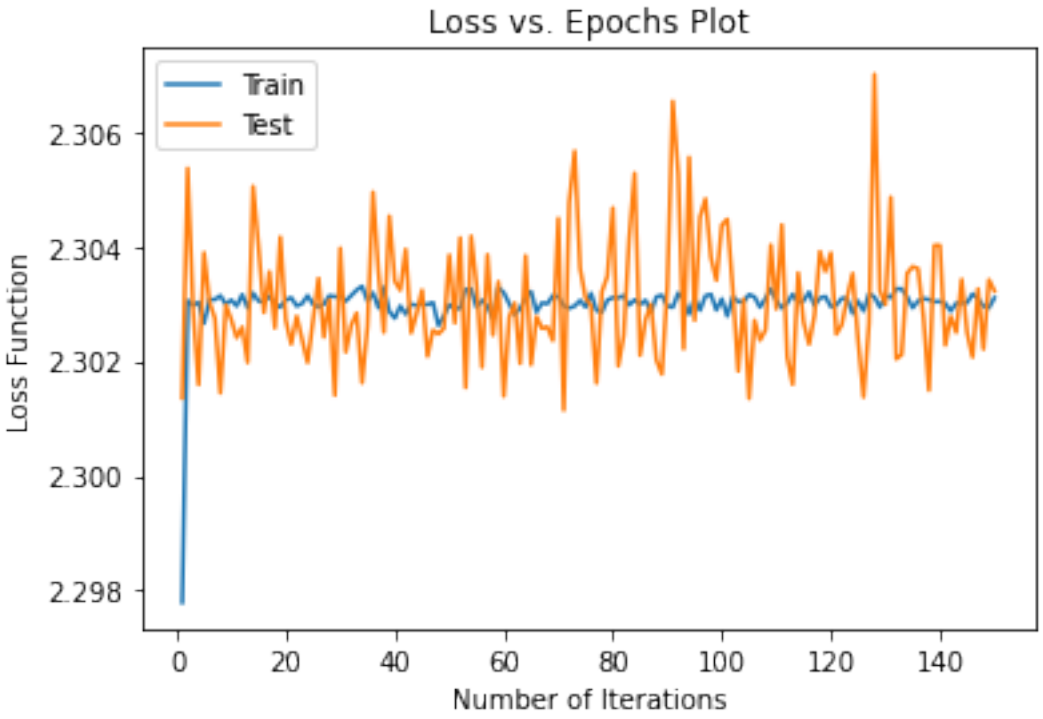
5.6.2 Learning Rate = 0.2

[ ]:
```
# Define HyperParameters
learning_rate = 0.2
CNN_LR2 = CNN(num_classes)
optimizer = optim.SGD(CNN_LR2.parameters(), lr = learning_rate)
```

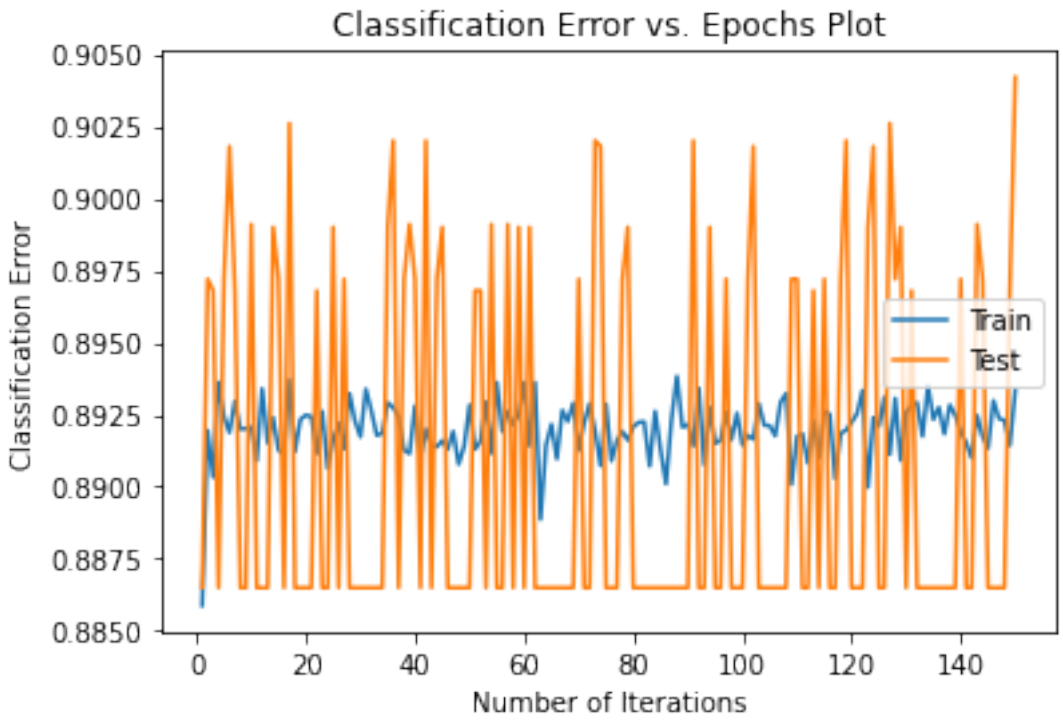
5.6.3 Learning Rate = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.5
CNN_LR5 = CNN(num_classes)
optimizer = optim.SGD(CNN_LR5.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



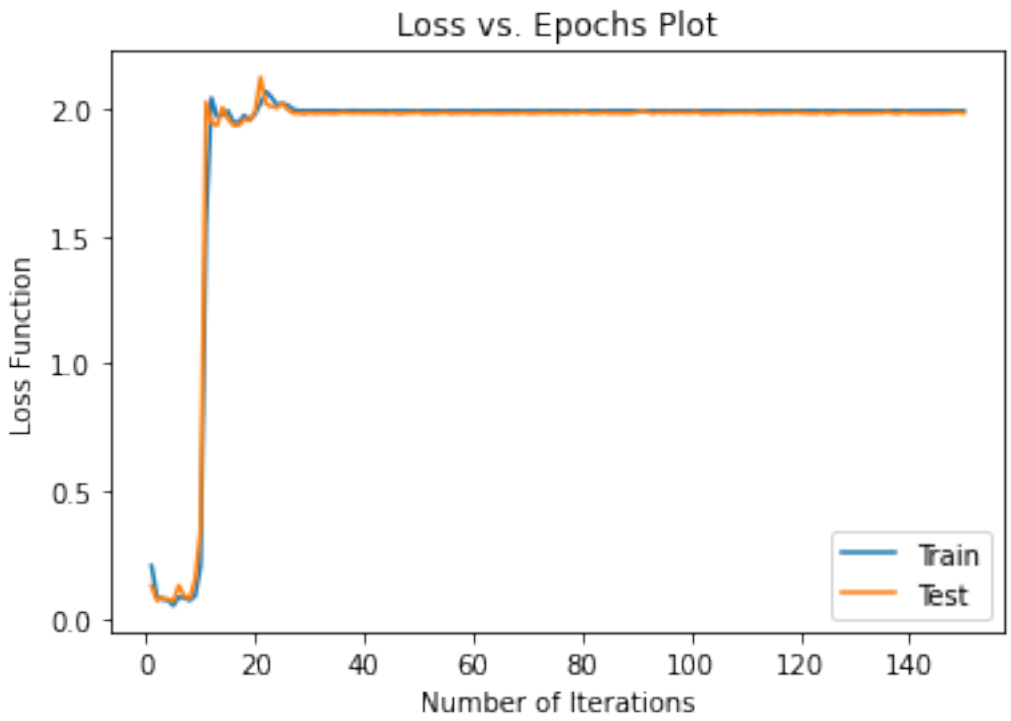
5.6.4 Momentum = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.5
CNN_M5 = CNN(num_classes)
optimizer = optim.SGD(CNN_M5.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

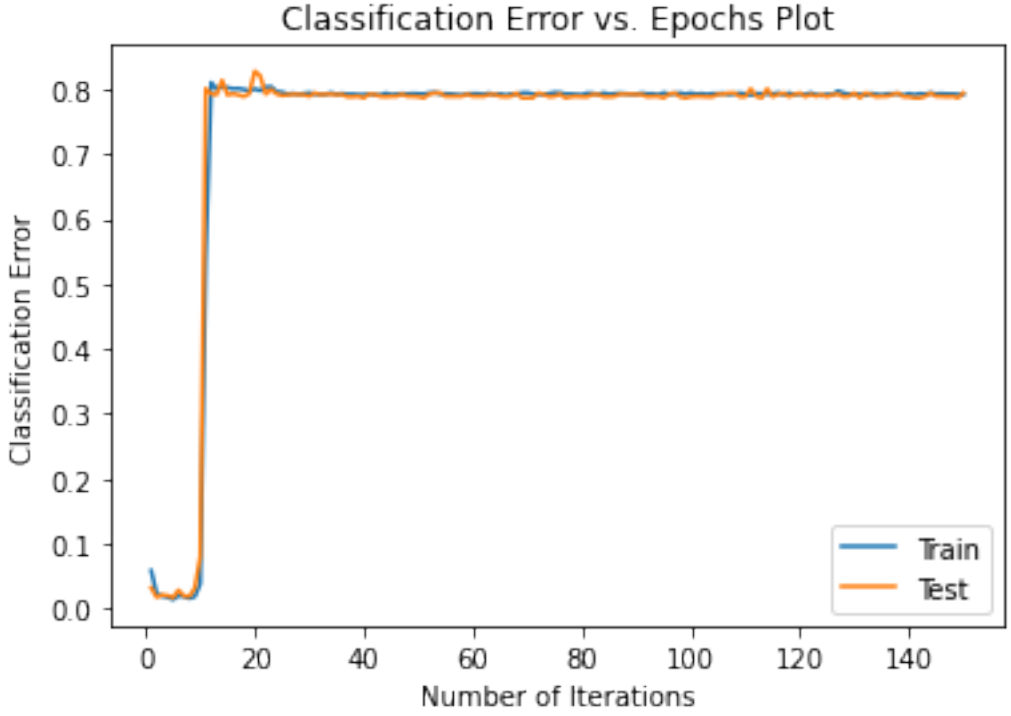
5.6.5 Momentum = 0.9

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.9
CNN_M9 = CNN(num_classes)
optimizer = optim.SGD(CNN_M9.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



5.6.6 Best Combination of Hyperparameters

Testing Accuracy As Follow:
Learning Rate = 0.1: 99.3%;
Learning Rate = 0.01: 99.15%;
Learning Rate = 0.2: 99.26%;
Learning Rate = 0.5: 9.58%;
Momentum = 0.5 & Learning Rate = 0.1: 99.33%;
Momentum = 0.9 & Learning Rate = 0.1: 2.06%.
From above, we can see that models with learning rate = 0.2 or 0.1 suffer from the problem of overfitting. The model with learning rate = 0.5 and model with momentum = 0.9 cannot reach to local minumin. Therefore, we can conclude that learning rate = 0.1 & momentum = 0.5 is the best hyperparameters choice in terms of efficiency and accuracy.

5.7 5.(a) & (b)

Lastly, we choose to construct a LeNet-5 model. At first, we need to reshape our input to 32 * 32 to fit LeNet-5. It includes two convolutional layers, batch normalizations, max pooling layers. The structure of neural network is shown below.

[ ]:
```
# Data Type Convert
transform = transforms.Compose(
    [transforms.Resize((32,32)), # Resize image to 32 * 32 to fit LeNet-5
     transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))])
# Read Data
train_data = MNIST(root='../data/mnist/', train=True, download=True, transform␣
 ↪= transform)
test_data = MNIST(root='../data/ mnist/', train=False, download=True, transform␣
 ↪= transform)
```

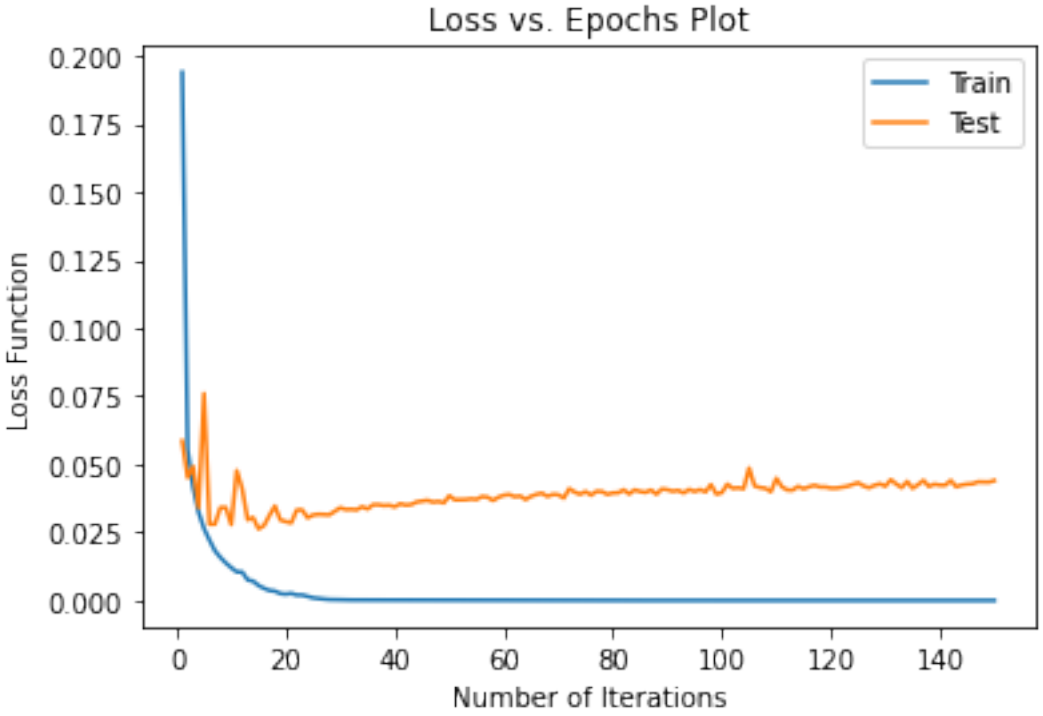
[ ]:
```
# Create a LeNet-5
class LeNet5(nn.Module):
    def __init__(self, num_classes):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(1, 6, 5), nn.BatchNorm2d(6), nn.
 ↪ReLU(), nn.MaxPool2d(2))
        self.conv2 = nn.Sequential(nn.Conv2d(6, 16, 5), nn.BatchNorm2d(16), nn.
 ↪ReLU(), nn.MaxPool2d(2))
        self.linear1 = nn.Linear(400, 120)
        self.linear2 = nn.Linear(120,84)
        self.linear3 = nn.Linear(84, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
```

```
        x = x.reshape(x.size(0),-1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

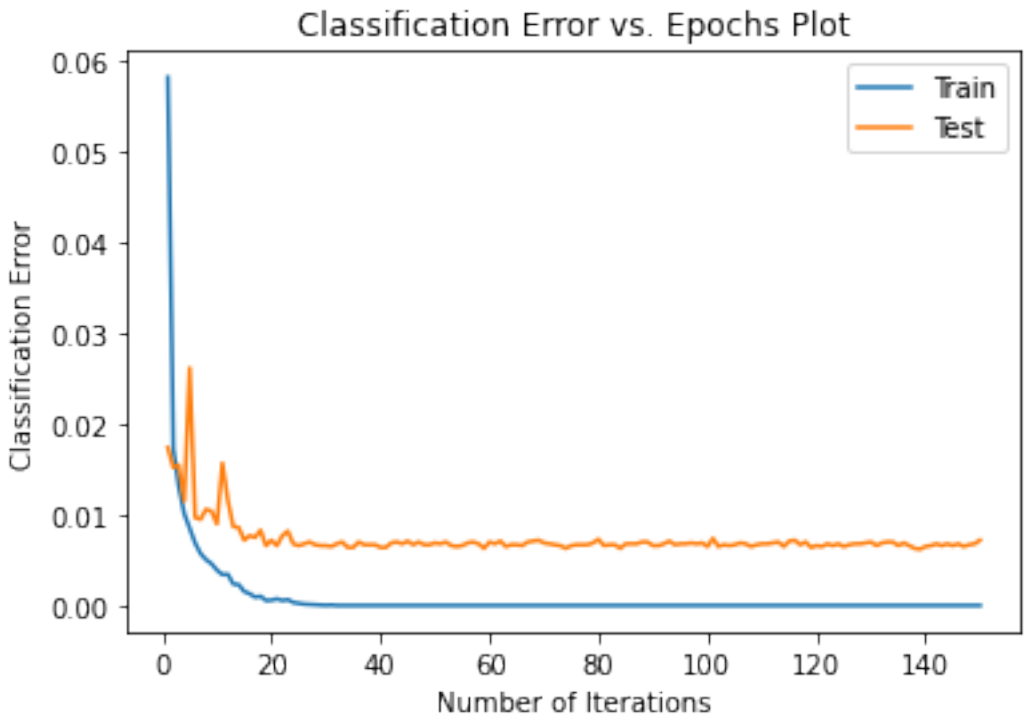
5.7.1 Seed 1

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
lenet1 = LeNet5(num_classes)
optimizer = optim.SGD(lenet1.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



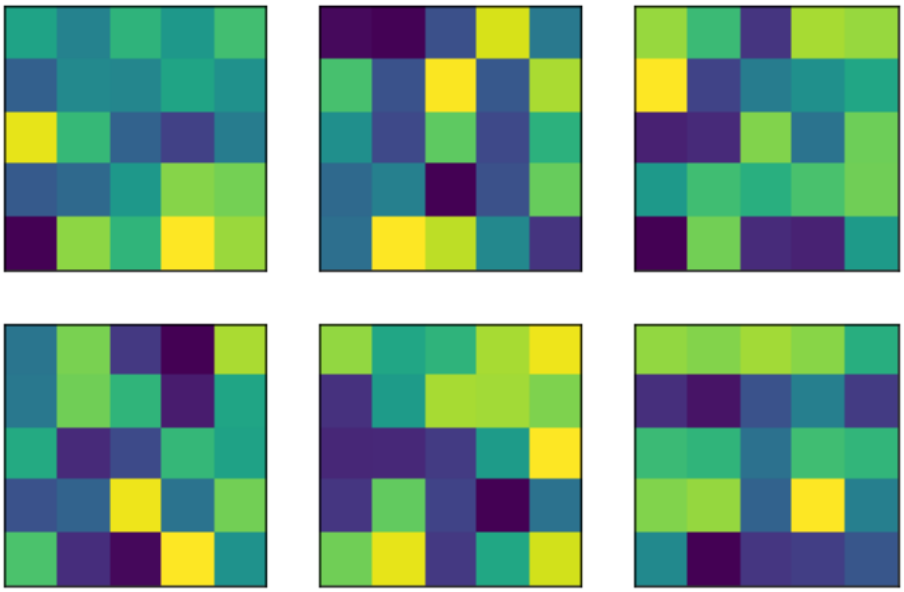
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



5.8 5.(c)

There are 6 filters for the first Convolutional Layer, which is corresponding to the number of output channel. The dimension for each filter is 5 * 5.

[ ]:
```
weights = lenet1.state_dict()['conv1.0.weight']
for i in range(len(weights)):
  plt.subplot(2,3,i+1)
  plt.imshow(weights[i].reshape(5,5))
  plt.xticks([])
  plt.yticks([])
```



We can see that the weight visualization is neither too noisy nor too correlated.

5.9 5.(d)

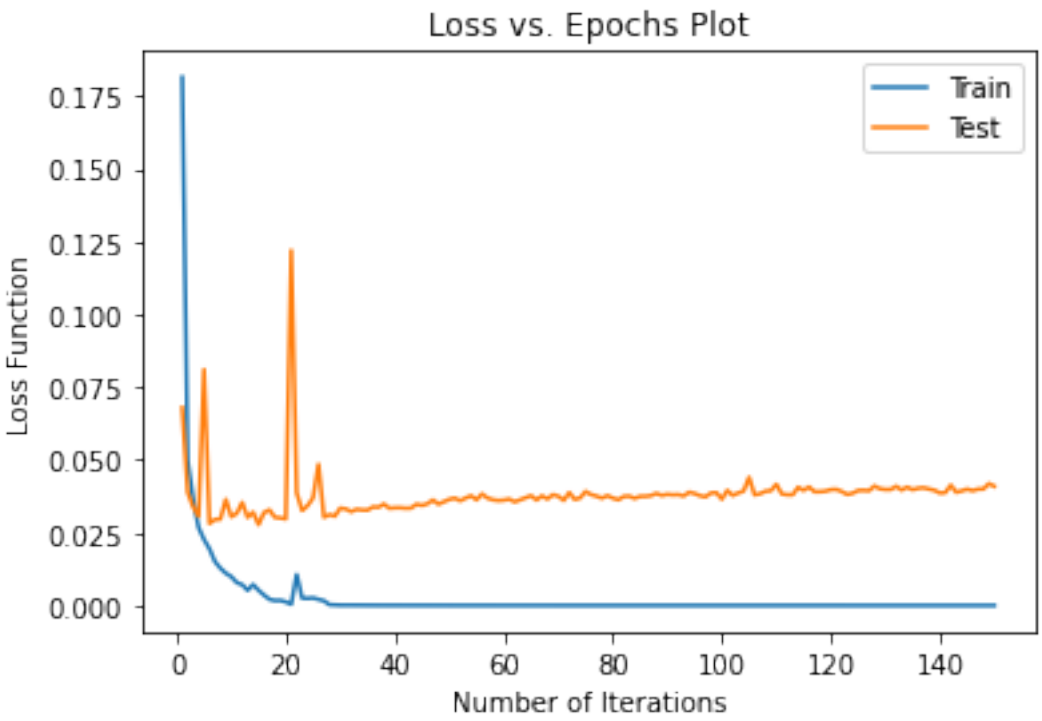
5.9.1 Learning Rate = 0.01

[ ]:
```
# Define HyperParameters
learning_rate = 0.01
lenet_lr01 = LeNet5(num_classes)
optimizer = optim.SGD(lenet_lr01.parameters(), lr = learning_rate)
```

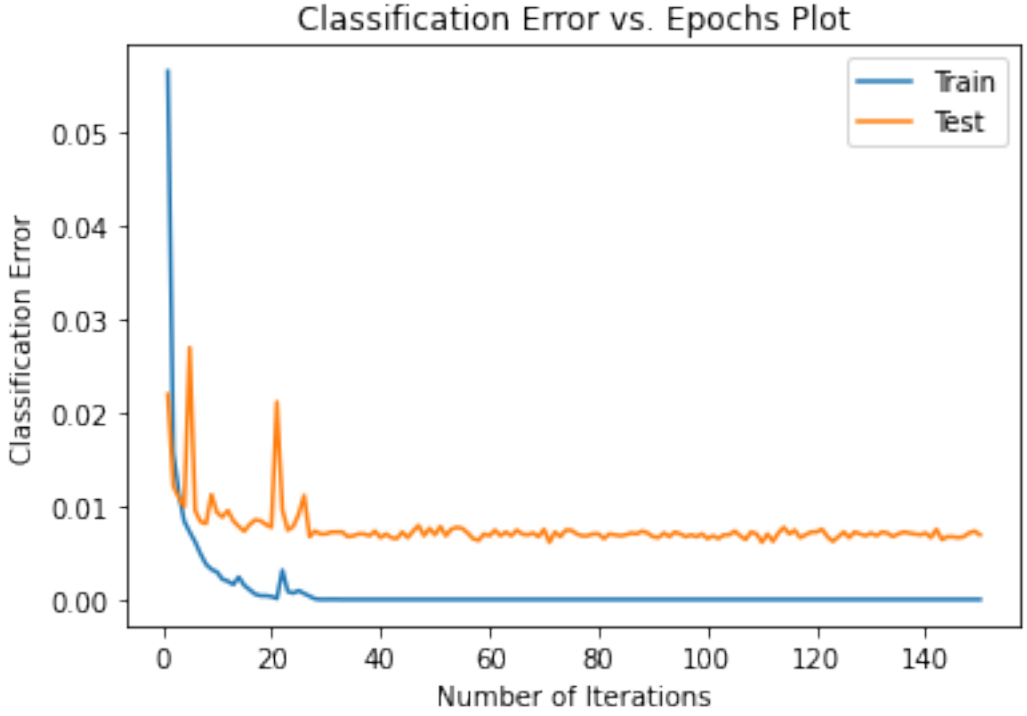
5.9.2 Learning Rate = 0.2

[ ]:
```
# Define HyperParameters
learning_rate = 0.2
lenet_lr2 = LeNet5(num_classes)
optimizer = optim.SGD(lenet_lr2.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



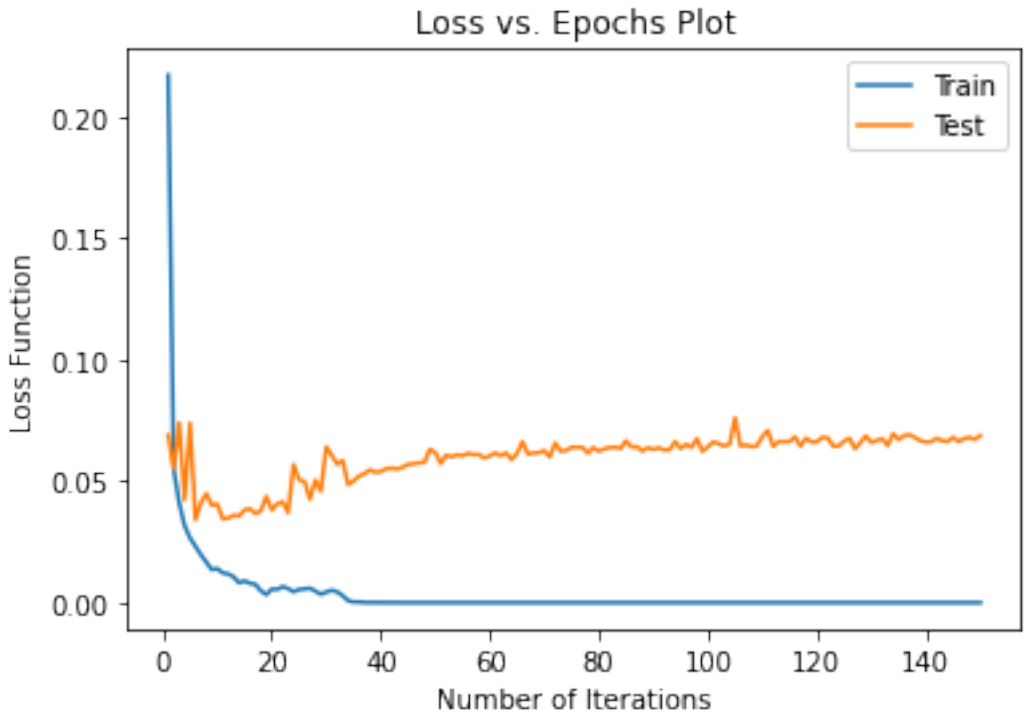
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



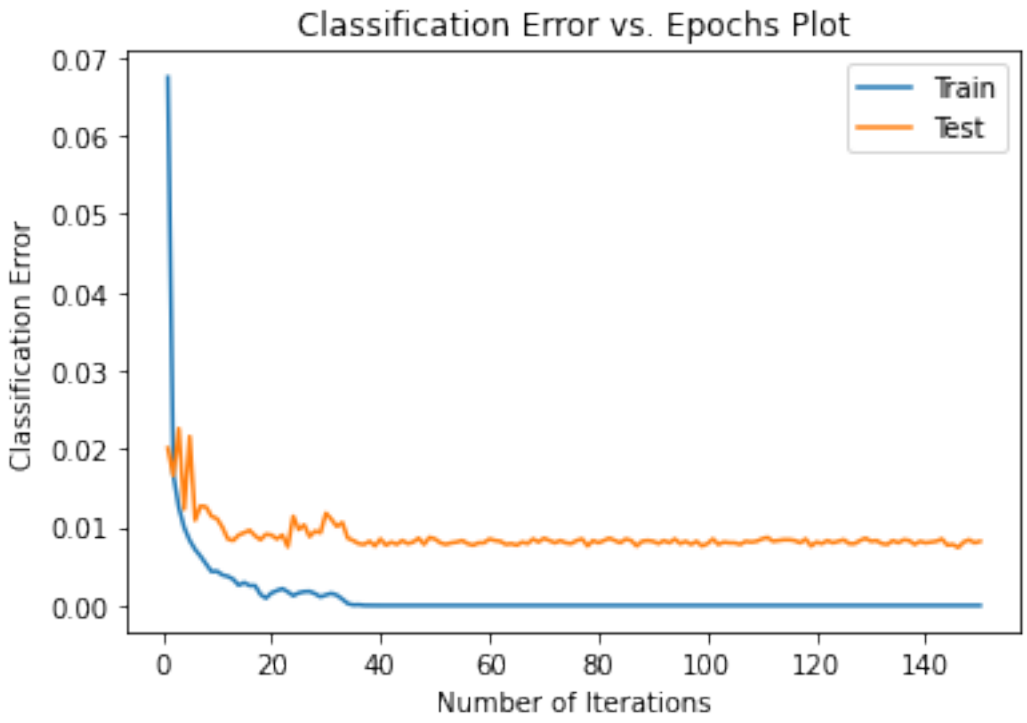
5.9.3 Learning Rate = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.5
lenet_lr5 = LeNet5(num_classes)
optimizer = optim.SGD(lenet_lr5.parameters(), lr = learning_rate)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



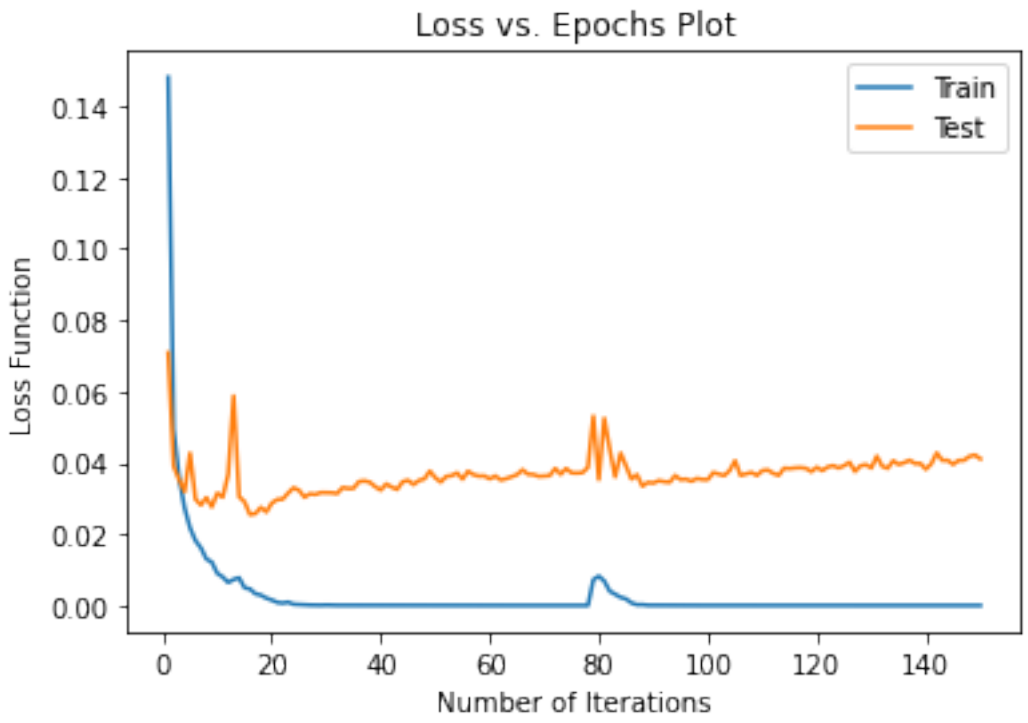
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



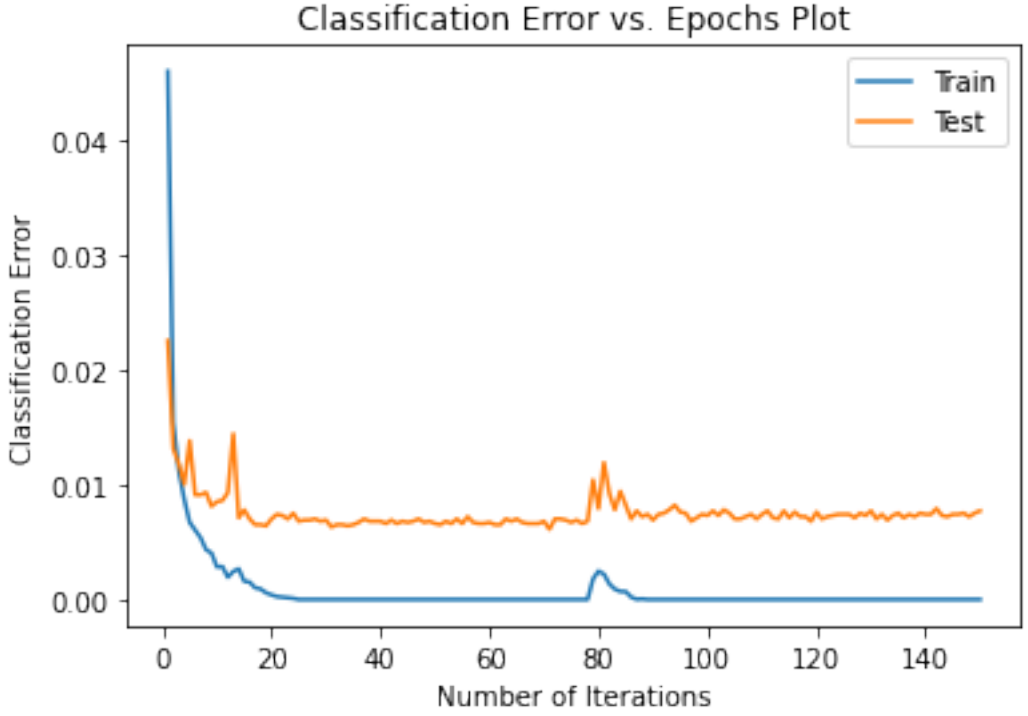
5.9.4 Momentum = 0.5

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.5
lenet_m5 = LeNet5(num_classes)
optimizer = optim.SGD(lenet_m5.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



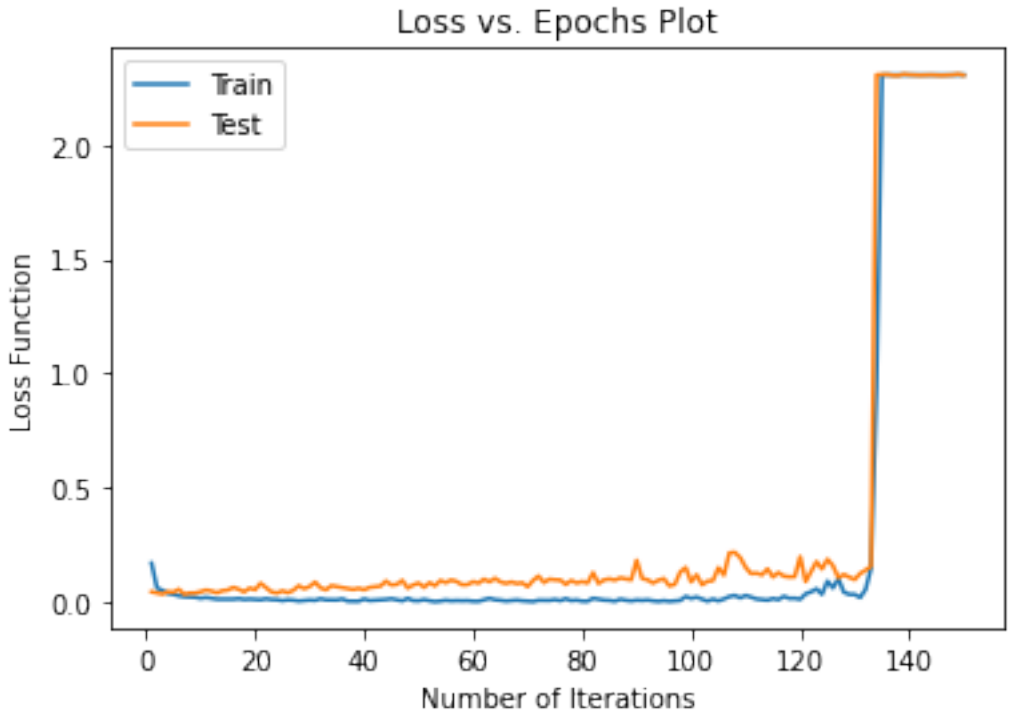
[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



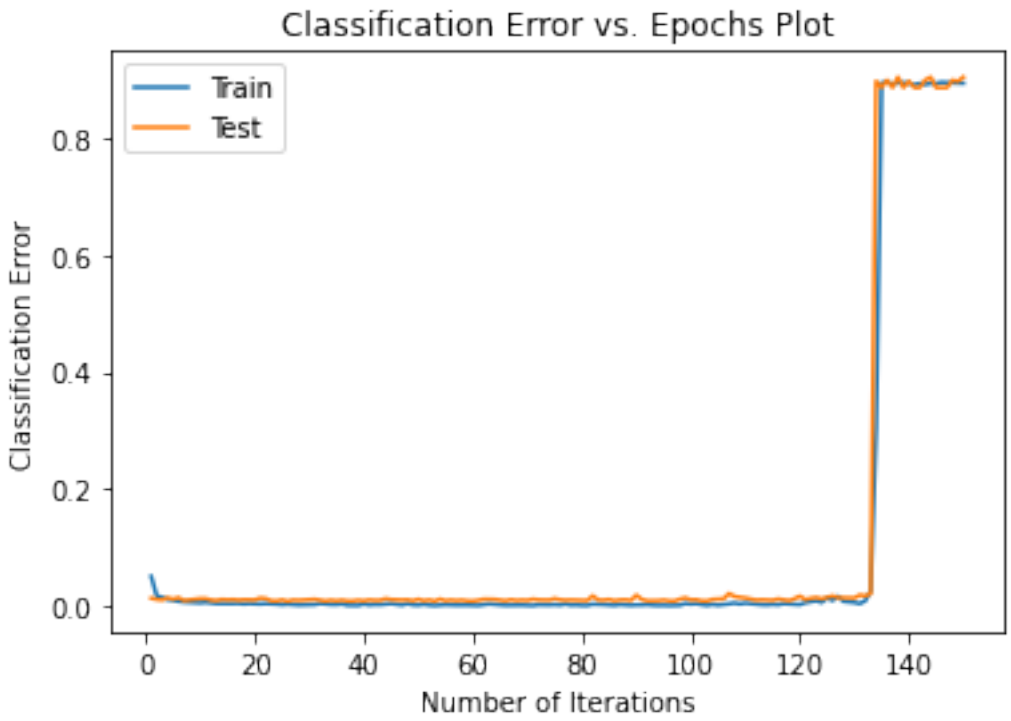
5.9.5 Momentum = 0.9

[ ]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.9
lenet_m9 = LeNet5(num_classes)
optimizer = optim.SGD(lenet_m9.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

[ ]:
```
loss_plot(train_loss_record, test_loss_record)
```



[ ]:
```
error_plot(train_loss_record, test_loss_record)
```



5.9.6 Best Combination of Hyperparameters

Testing Accuracy As follow:
Learning Rate = 0.1: 99.28%;
Learning Rate = 0.01: 99.05%;
Learning Rate = 0.2: 99.31%;
Learning Rate = 0.5: 99.18%;
Momentum = 0.5 & Learning Rate = 0.1: 99.23%;
Momentum = 0.9 & Learning Rate = 0.1: 9.58%.

From above, it can be observed that the loss function of LeNet5 can always converge quickly to local minimum with different hyperparameters. However, all models suffer from the problem of overfitting, while for the models with learning rate = 0.2, learning rate = 0.5 and momentum = 0.5, we can observe a sudden surge in loss and misclassification error, which indicates loss function moves away from local minimum. In, addition, it can be observed that the model with momentum = 0.9 moves away from local minimum and cannot return to local minimum. Therefore, we can conclude that momentum is unnecessary for LeNet5 and a small learning rate like 0.1 and 0.01 is good enough.

6 V. More about Deep Learning

6.1 6.

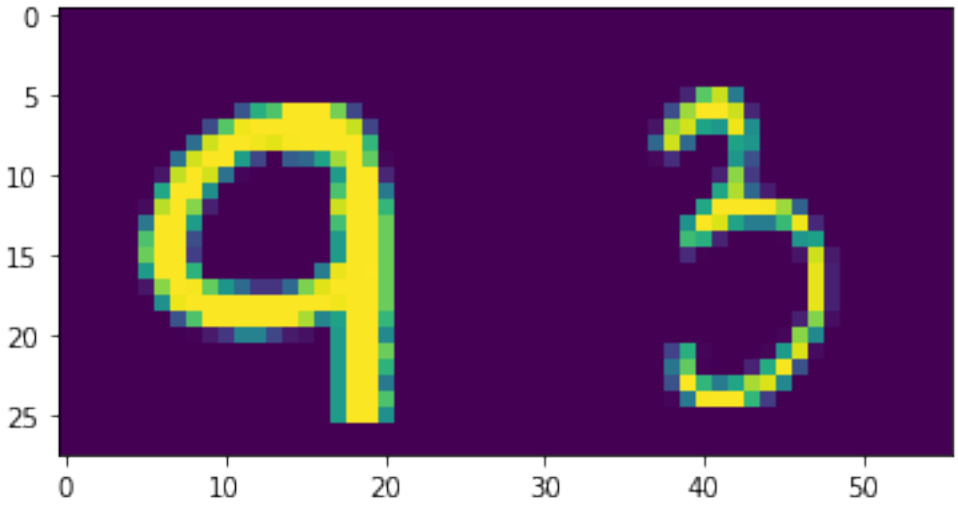
[ ]:
```
# Load Data
train_data = pd.read_csv('/train.txt', header = None)
test_data = pd.read_csv('/test.txt', header = None)
val_data = pd.read_csv('/val.txt', header = None)

train_data.shape
```

[ ]:
```
(20000, 1569)
```

From above, we can see that the number of rows is 20000. We know that each sample is composed of a total of 1568 pixels and one label. Therfore, we know that the pixels are scanned out in row-major order.

[ ]:
```
sampleplot(4)
```

```
1568    12.0
Name: 4, dtype: float64
```



From above, we can see that the sum of two digits equal to the last coordinate of each line.

6.2 7.(a)

[7]:
```
def train_loop7(feature,label, model, loss_fn, optimizer):
    running_loss, correct = 0, 0
    size = len(feature)
    for index, X in enumerate(feature):
        # Compute prediction and loss
```

```
        y = label[index]
        pred = model(X)
        loss = loss_fn(pred, y)
        running_loss += loss.item()
        # Use Round to Classify
        correct += (torch.round(pred) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    running_loss = running_loss / size
    correct /= size
    print('Training loss: {}'.format(running_loss))
    print('Training accuracy: {}'.format(correct))
    return running_loss, correct
def test_loop7(feature,label, model, loss_fn):
    size = len(feature)
    test_loss, correct = 0, 0
    with torch.no_grad():
        for index, X in enumerate(feature):
            pred = model(X)
            y = label[index]
            test_loss += loss_fn(pred, y).item()
            correct += (torch.round(pred) == y).type(torch.float).sum().item()
    test_loss /= size
    correct /= size
    print('Testing loss: {}'.format(test_loss))
    print('Testing accuracy: {}'.format(correct))
    return test_loss, correct
```

6.3 Replication From Paper

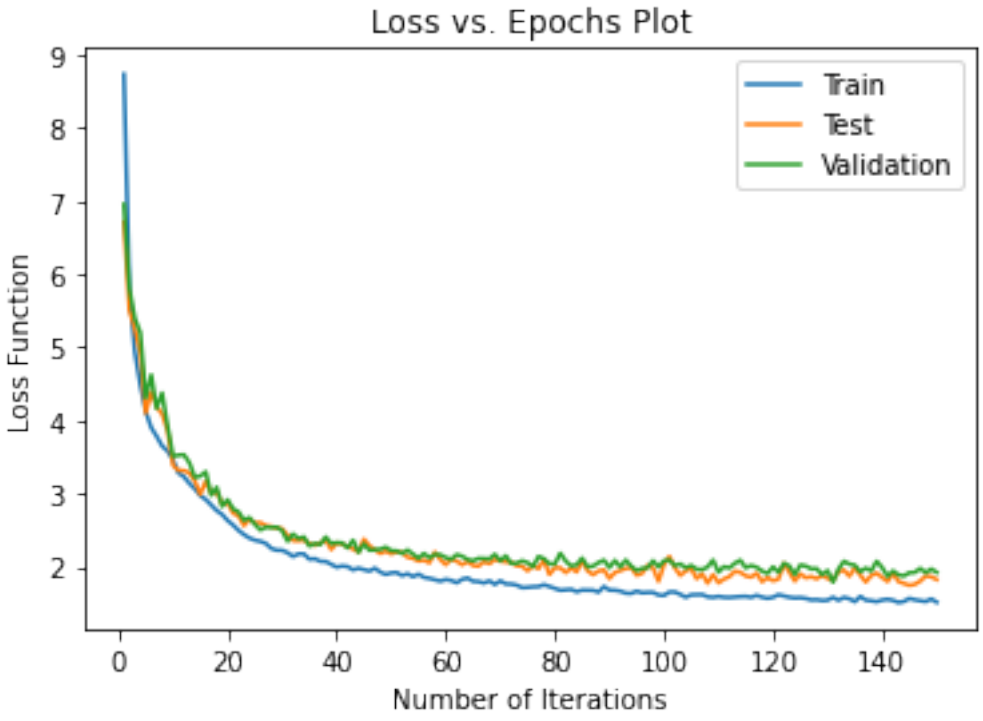
[8]:
```
class sum_model(nn.Module):
    def __init__(self):
        super(sum_model, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3) # Output Shape 26 * 54 * 32
        self.conv2 = nn.Conv2d(32, 16, 3) # Output Shape 24 * 52 * 16
        self.pool1 = nn.MaxPool2d(2) # Output Shape 12 * 26 * 16
        self.do = nn.Dropout(0.25)
        self.flatten = nn.Flatten() # Output Shpae 16 * 312
        self.linear1 = nn.Linear(312, 16) # ouput shape 16 * 16
        self.do2 = nn.Dropout(0.5)
        self.linear2 = nn.Linear(256, 100)
        self.linear3 = nn.Linear(100, 1) # Output Shape 1 Make it Regression
 ↪
    def forward(self, x):
        x = F.relu(self.conv1(x))
```

```
        x = F.relu(self.conv2(x))
        x = self.pool1(x)
        x = self.do(x)
        x = self.flatten(x)
        x = F.relu(self.linear1(x))
        x = x.reshape(-1)
        x = self.do2(x)
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

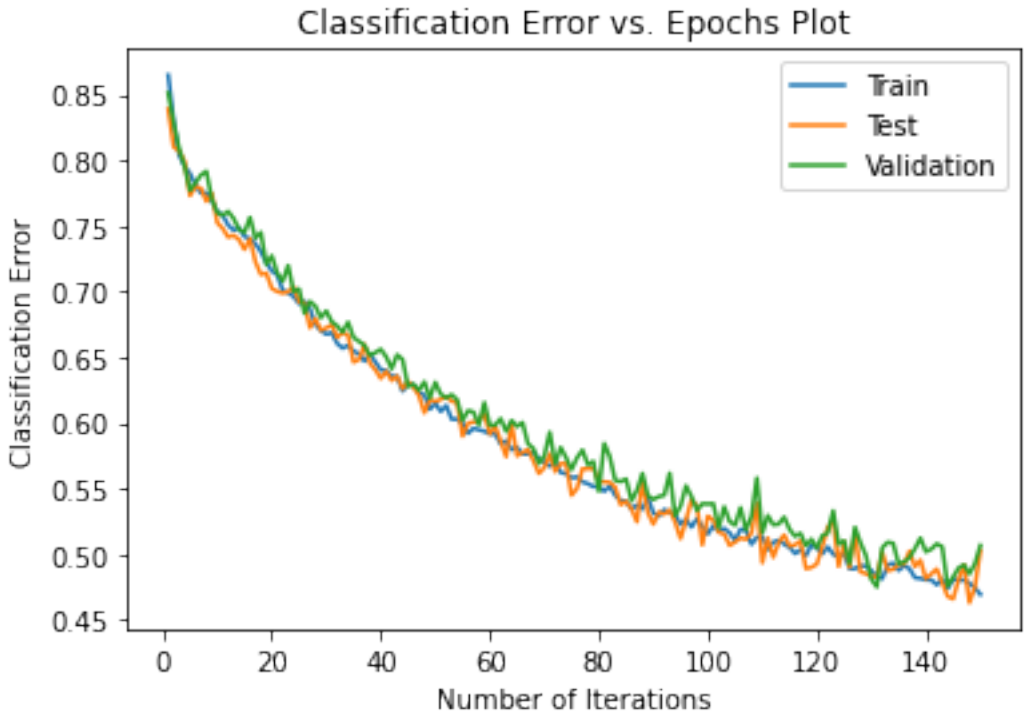
6.3.1 Learning Rate = 1

[ ]:
```
# Define HyperParameters
learning_rate = 1
net5 = sum_model().double()
# Use the same optimizer Adadelta As in the paper
optimizer = optim.Adadelta(net5.parameters(), lr = learning_rate, rho = 0.95,␣
 ↪eps = 1e-08, weight_decay = 0)
# Mean Squared Error Loss Function Used
loss = nn.MSELoss()
```

[ ]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```

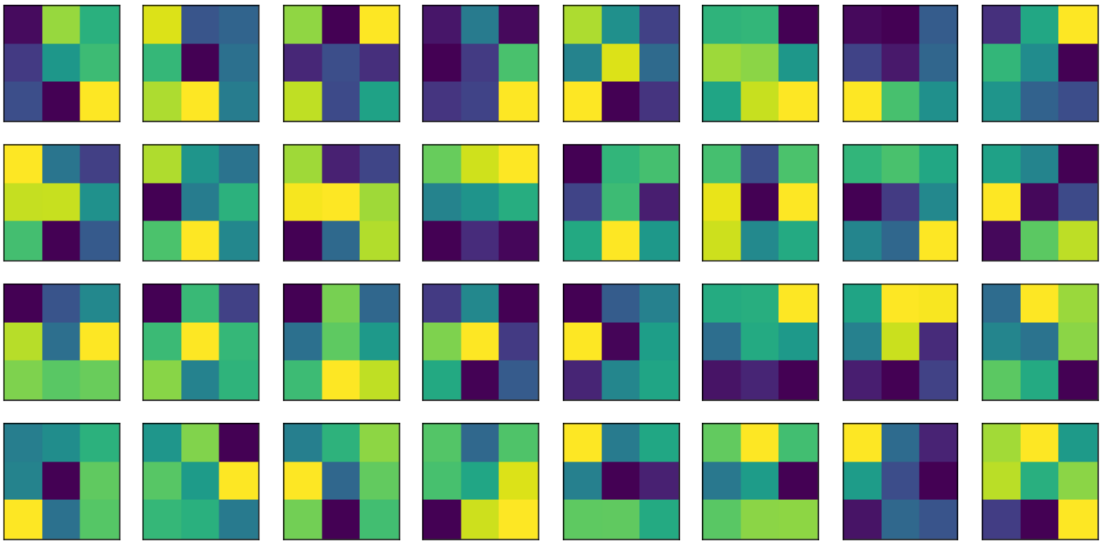


[ ]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



6.3.2 Weight Visualization

[ ]:
```
# Visualize the Weights from First Convolutional Network
weights = net5.state_dict()['conv1.weight']
plt.figure(figsize = (12,6))
for i in range(len(weights)):
  plt.subplot(4,8,i+1)
  plt.imshow(weights[i].reshape(3,3))
  plt.xticks([])
  plt.yticks([])
```

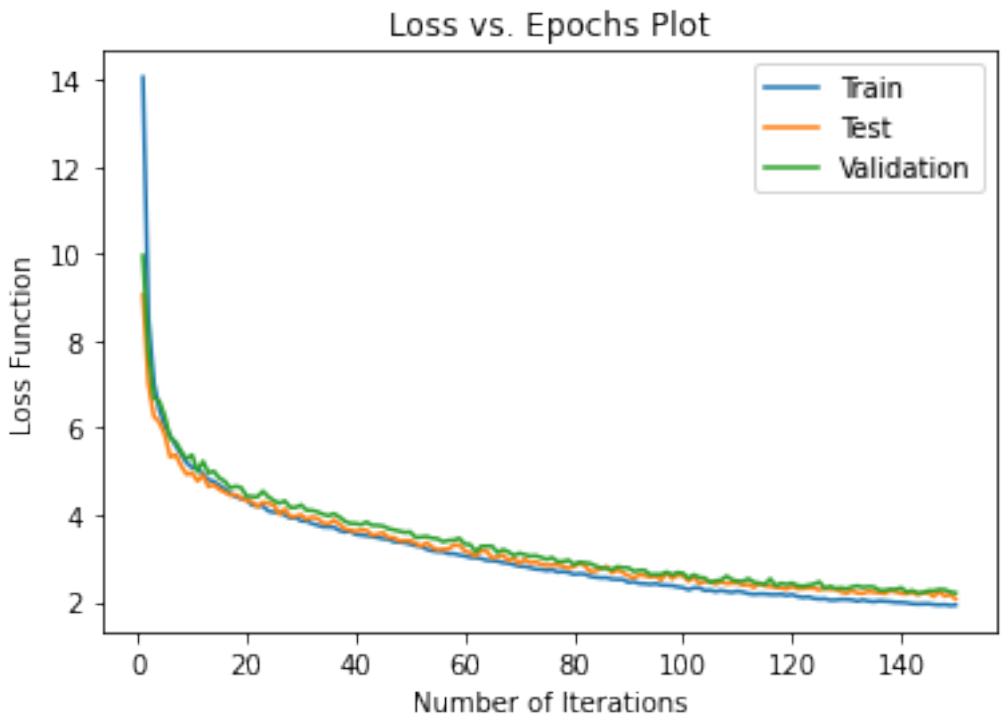


We can see that the weight visulization is neither too noisy nor too correlated.

6.3.3 Learning Rate = 0.2

[ ]:
```
# Define HyperParameters
learning_rate = 0.2
sum_2 = sum_model().double()
optimizer = optim.Adadelta(sum_2.parameters(), lr = learning_rate, rho = 0.95,␣
↪eps = 1e-08, weight_decay = 0)
```

[ ]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```

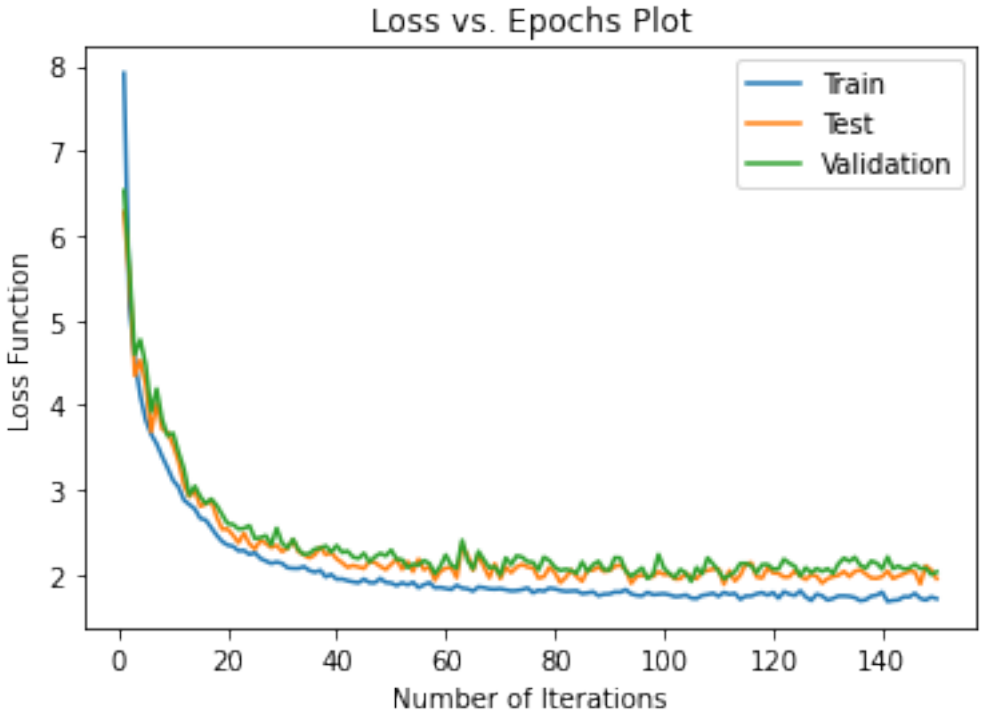


[ ]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```

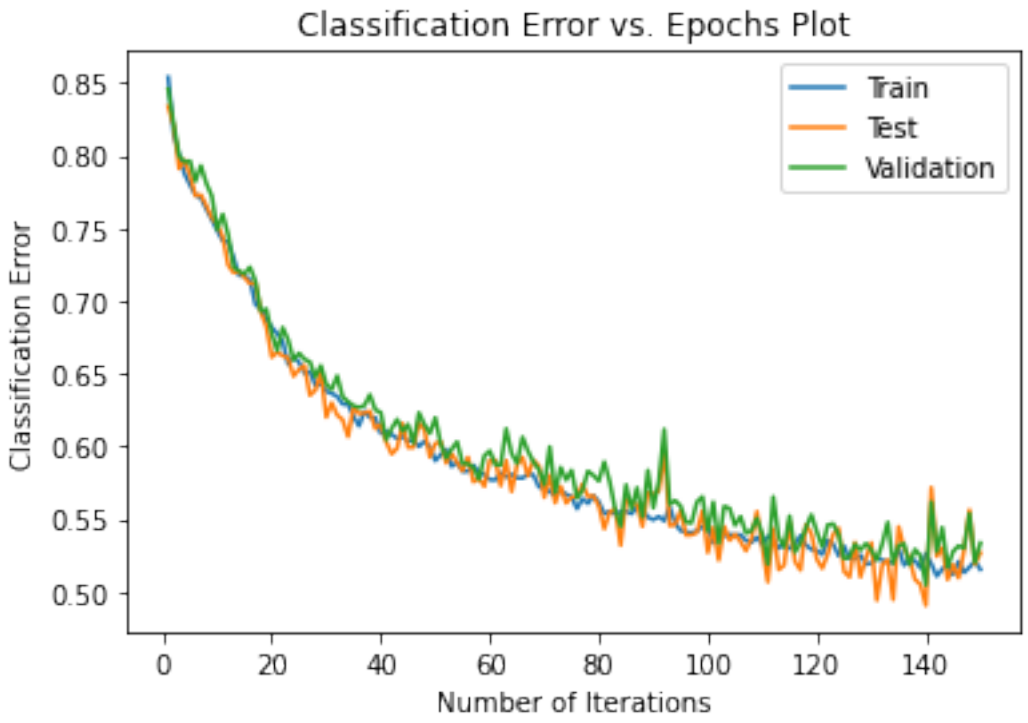
6.3.4 Learning Rate = 2.5

[9]:
```
# Define HyperParameters
learning_rate = 2.5
sum_5 = sum_model().double()
optimizer = optim.Adadelta(sum_5.parameters(), lr = learning_rate, rho = 0.95,␣
 ↪eps = 1e-08, weight_decay = 0)
```

[11]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```



[12]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



From above, we can observe the validation error as follow:
Learning Rate = 1: 50.66%;
Learning Rate = 0.2: 62.94%;
Learning Rate = 2.5: 53.34%.
Therefore, we choose the model with learning rate = 1. In addition, it can be observed that the loss function and misclassification keep decreasing with the number of epochs increasing. In another word, they do not suffer from the problem of overfitting and we can increase the number of epochs to obtain better result. However, due to the limited calculation power of my computer, I decide not to do it.
The testing error of our chosen model is 50.26% which is significantly larger than the test errors with respect to the original MNIST data. The bad performance is probably because I decrease the number of output channels of the second convolutional layer due to the limited calculation power of my computer. However, we can see that this algorithm replicated from paper is generalizable. Therefore, I believe we can train a good enough model with a better computer.

6.4 3-Layer 2D Convolutional Neural Network

[60]:
```
# Create a Convolutional Neural Network
class CNN1(nn.Module):
    def __init__(self, num_classes):
        super(CNN1, self).__init__() # Input Shape 1 * 28 * 56
        self.conv1 = nn.Sequential(nn.Conv2d(1, 32, 3), nn.BatchNorm2d(32), nn.
↪ReLU(), nn.MaxPool2d(2))
        # Ouput Shape 32 * 13 * 27
```

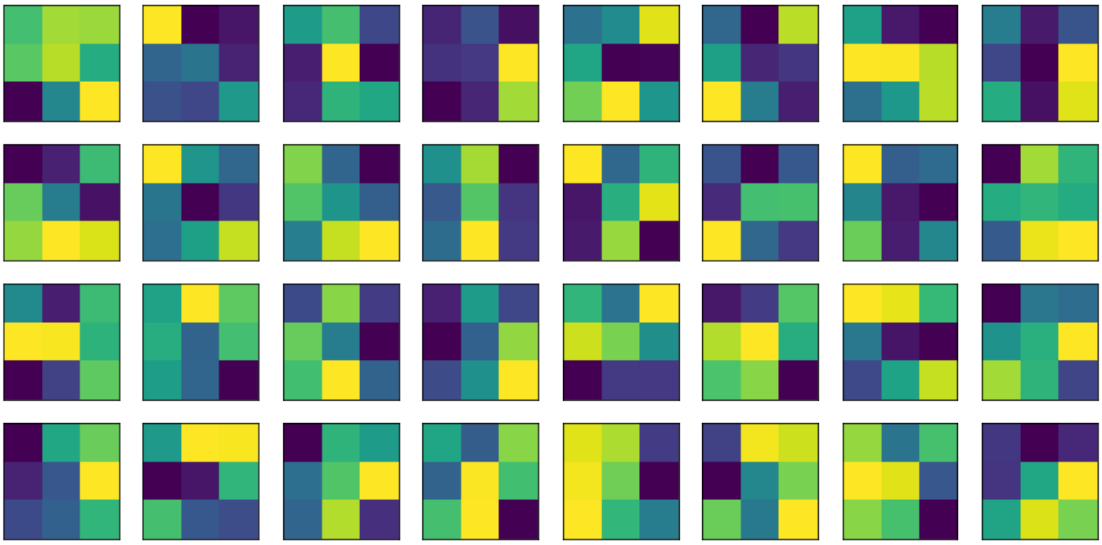
```
        self.conv2 = nn.Sequential(nn.Conv2d(32, 64, 3), nn.BatchNorm2d(64), nn.
↪ReLU(), nn.MaxPool2d(2))
        # Output Shape 64 * 5 * 12
        self.conv3 = nn.Conv2d(64,128,3) # Output Shape 128 * 3 * 10
        self.fc1 = nn.Linear(3840, 800) # Output Shape 800
        self.fc2 = nn.Linear(800, 256) # Output Shape 256
        self.fc3 = nn.Linear(256, num_classes) # Output Shape: num_classes
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = F.relu(self.conv3(x))
        x = x.reshape(-1, 3840)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

6.4.1 Learning Rate = 0.1

[66]:
```
# Define HyperParameters
# Number of Classes = 19
num_classes = 19
learning_rate = 0.1
CNN_1 = CNN1(num_classes).double()
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(CNN_1.parameters(), lr = learning_rate)
```

6.4.2 Weight Visualization

[75]:
```
# Visualize the Weights from First Convolutional Network
weights = CNN_1.state_dict()['conv1.0.weight']
plt.figure(figsize = (12,6))
for i in range(len(weights)):
  plt.subplot(4,8,i+1)
  plt.imshow(weights[i].reshape(3,3))
  plt.xticks([])
  plt.yticks([])
```

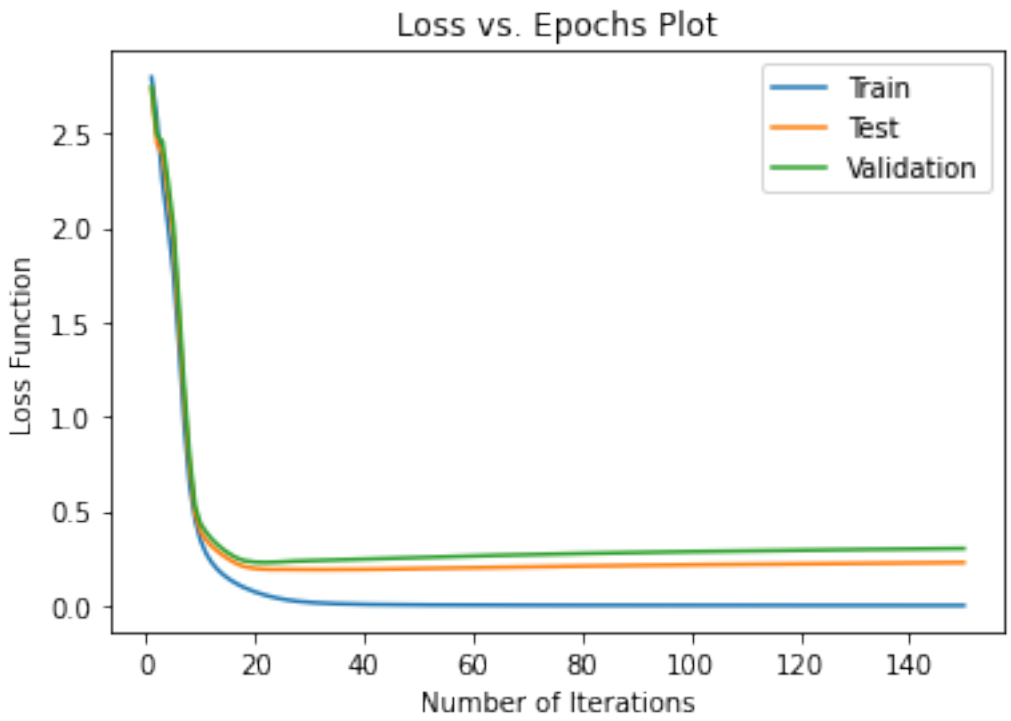


We can see that the weight visualization is neither too noisy nor too correlated.

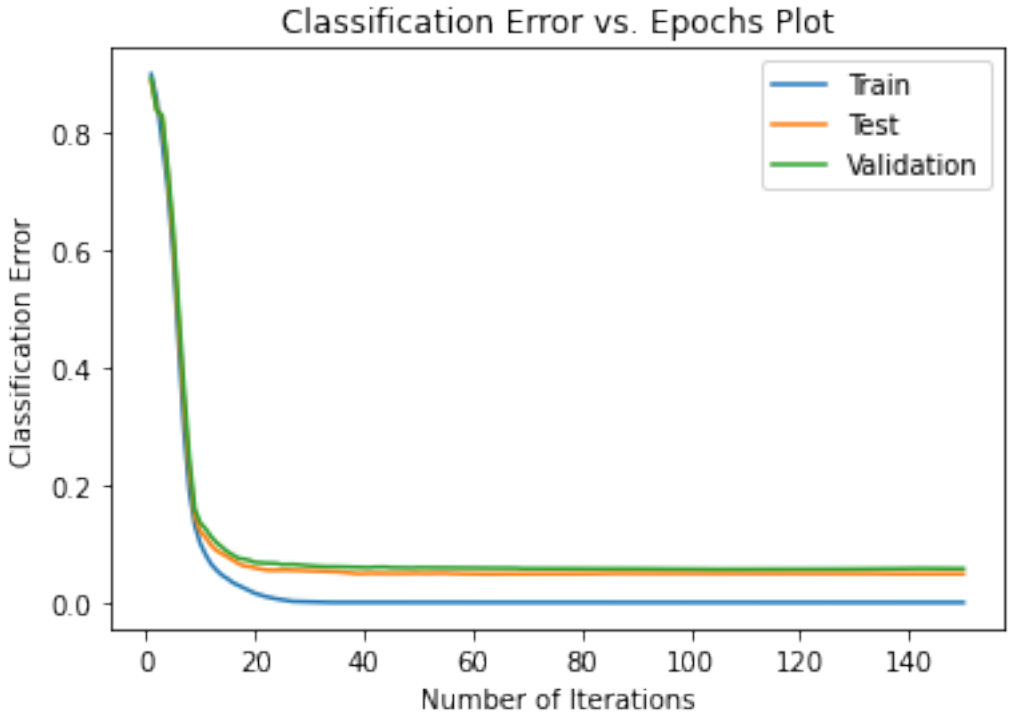
6.4.3 Learning Rate = 0.01

[77]:
```
# Define HyperParameters
learning_rate = 0.01
CNN_01 = CNN1(num_classes).double()
optimizer = optim.SGD(CNN_01.parameters(), lr = learning_rate)
```

[79]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```



[80]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



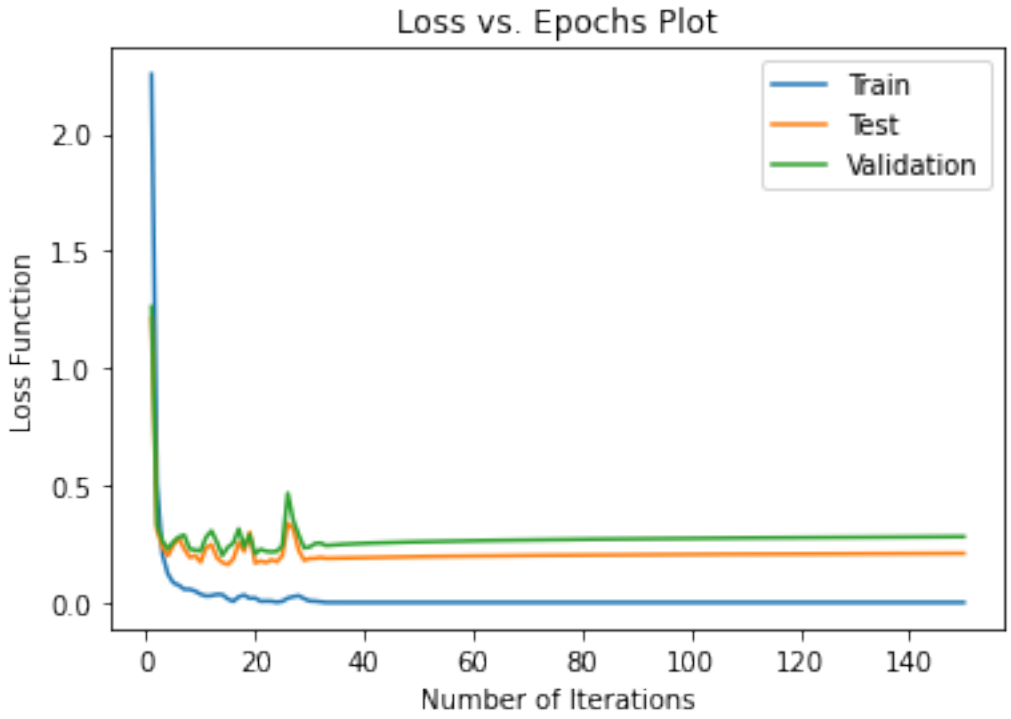
6.4.4 Learning Rate = 0.2

[81]:
```
# Define HyperParameters
learning_rate = 0.2
CNN_2 = CNN1(num_classes).double()
optimizer = optim.SGD(CNN_2.parameters(), lr = learning_rate)
```

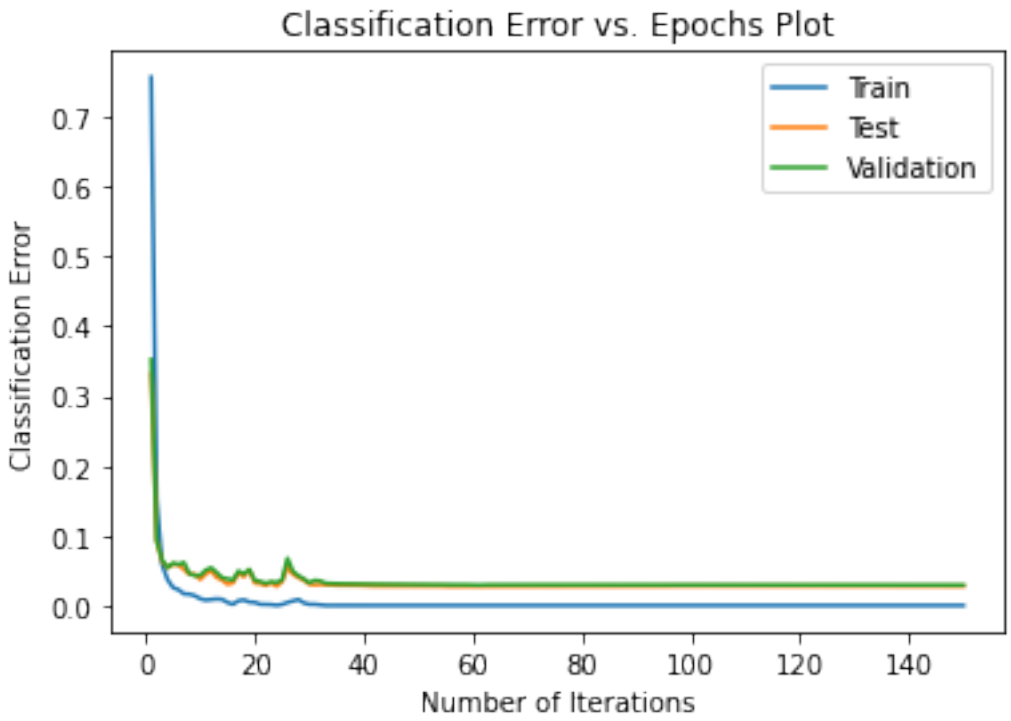
6.4.5 Learning Rate = 0.5

[85]:
```
# Define HyperParameters
learning_rate = 0.5
CNN_5 = CNN1(num_classes).double()
optimizer = optim.SGD(CNN_5.parameters(), lr = learning_rate)
```

[87]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```



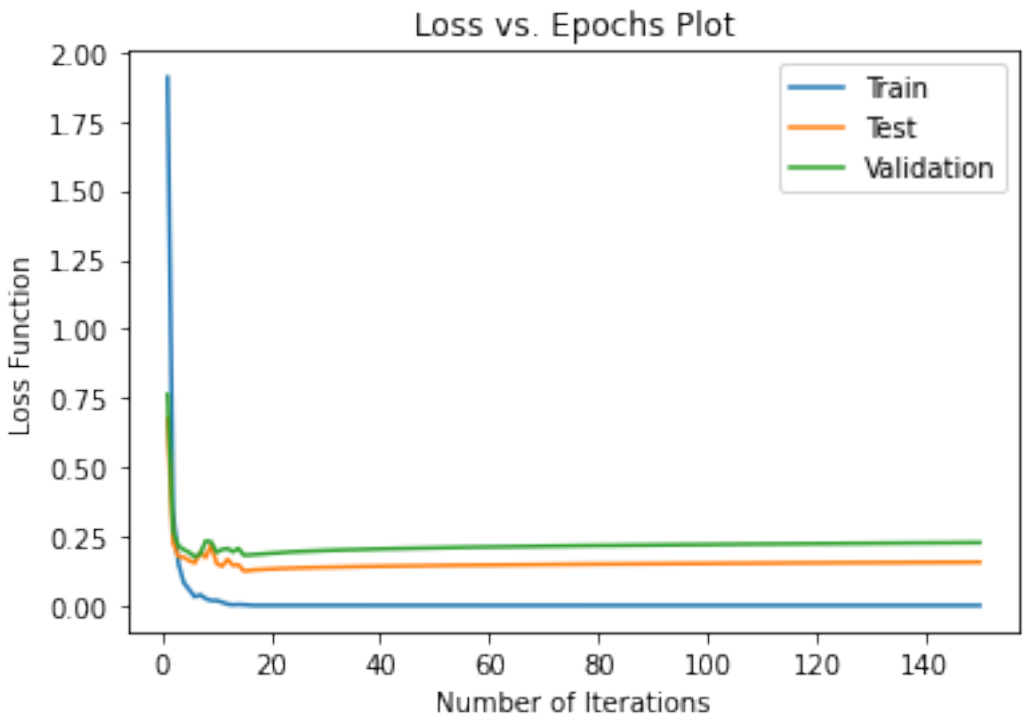
[88]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



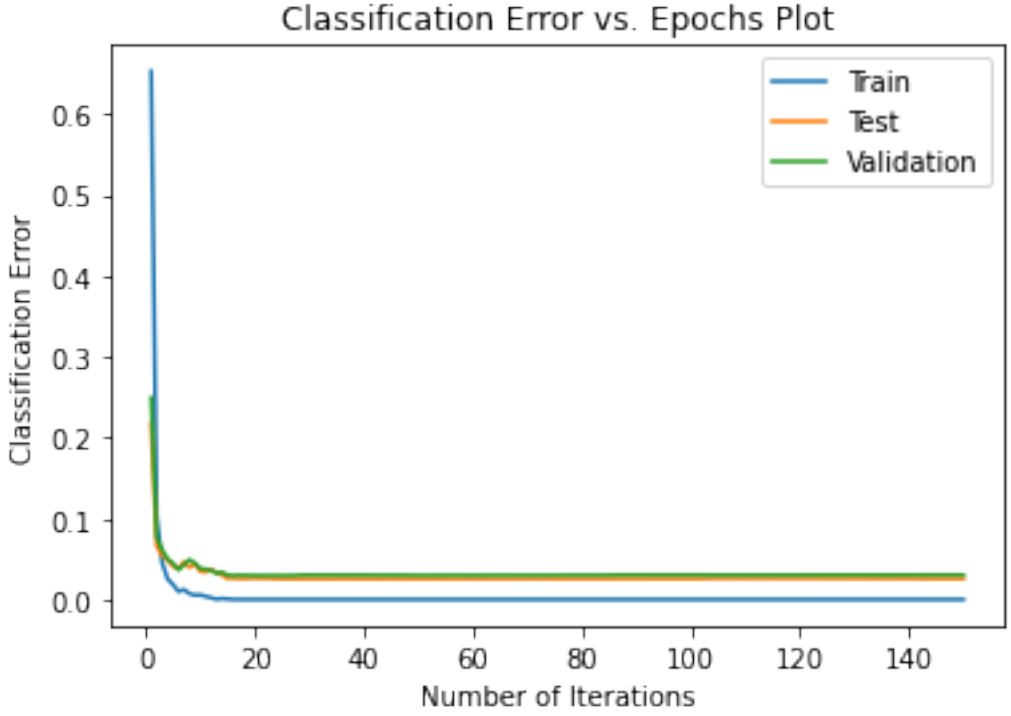
6.4.6 Momentum = 0.5

[89]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.5
CNN_M5 = CNN1(num_classes).double()
optimizer = optim.SGD(CNN_M5.parameters(), lr = learning_rate, momentum =␣
↪momentum)
```

[91]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```



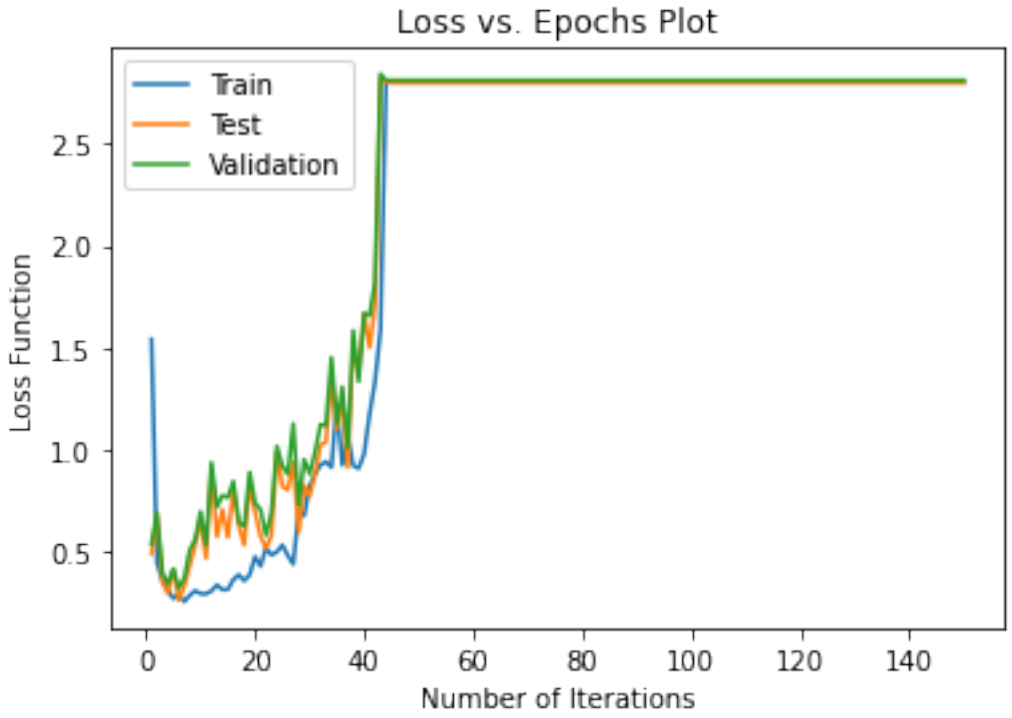
[92]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



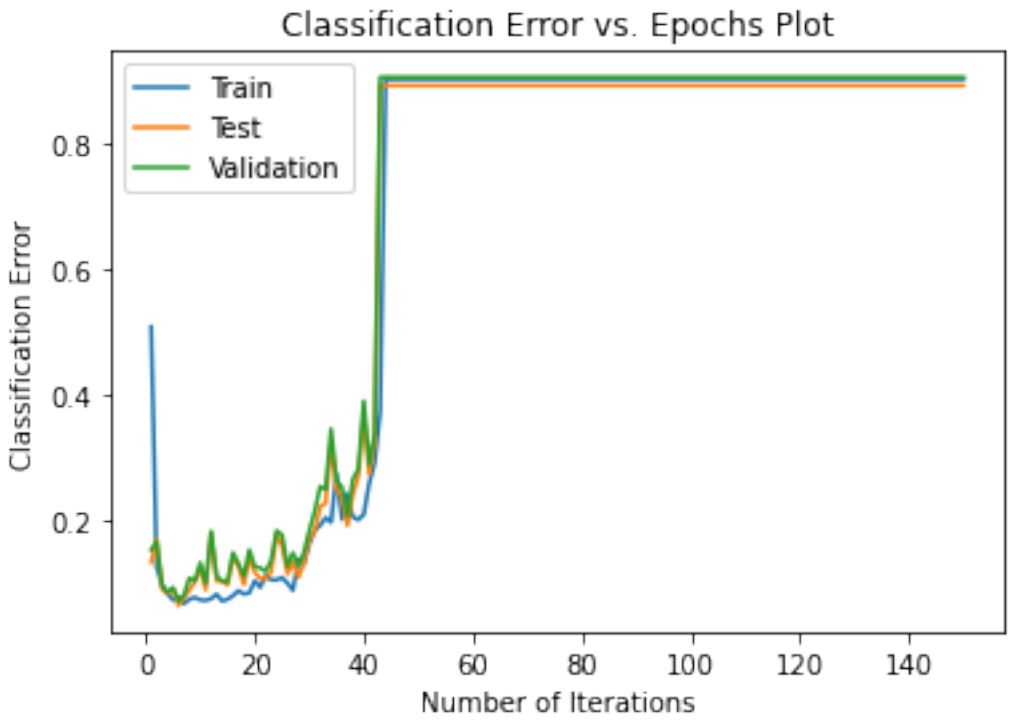
6.4.7 Momentum = 0.9

[93]:
```
# Define HyperParameters
learning_rate = 0.1
momentum = 0.9
CNN_M9 = CNN1(num_classes).double()
optimizer = optim.SGD(CNN_M9.parameters(), lr = learning_rate, momentum = 0.9)
```

[95]:
```
loss_plot1(train_loss_record, test_loss_record, val_loss_record)
```



[96]:
```
error_plot1(train_loss_record, test_loss_record, val_loss_record)
```



6.5 7.(b)

From above, we can observe the validation error as follow:
Learning Rate = 0.1: 3.28%;
Learning Rate = 0.01: 5.76 %;
Learning Rate = 0.2: 3.04%;
Learning Rate = 0.5: 2.93%;
Momentum = 0.5 & Learning Rate = 0.1: 3.02%;
Momentum = 0.9 & Learning Rate = 0.1: 90.64%.
Therefore, we choose the model with Learning Rate = 0.5.

6.6 7.(c)

From above, we can observe the generalization error as follow:
Learning Rate = 0.1: 3.3%;
Learning Rate = 0.01: 4.88%;
Learning Rate = 0.2: 2.54%;
Learning Rate = 0.5: 2.98%;
Momentum = 0.5 & Learning Rate = 0.1: 2.58%;
Momentum = 0.9 & Learning Rate = 0.1: 89.3%.
Let's assume that the neural network perfectly figure out the summation relationship between the two image digits and label and there is no influence caused by the concatenation of two images. The highest generalization accuracy we can get is 99.3% from convolution neural network in part 4 with momentum = 0.5 & learning rate = 0.1. The probability of two correct classification is 0.993

* 0.993 = 0.986. Therefore, even if our above assumption hold, we still cannot obtain a test error lower than 1%.

7 Reference:

Bloice, M. D., Roth, P. M., & Holzinger, A. (2019, December 6). Performing arithmetic using a neural network trained on digit permutation pairs. arXiv.org. Retrieved April 26, 2022, from https://arxiv.org/abs/1912.03035