1 Neural Networks using Numpy

1.1 Helper Function:

```
def Relu(x):
    x = np.maximum(0,x)
    return x

def Reludiff(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def softmax(z):
    expo = np.exp(z - np.max(z))
    sum = np.sum(expo, axis = 1)
    sum = sum.reshape(sum.shape[0], 1)
    return np.divide(expo, sum)

def inputreshape(x):
    x_0 = x.shape[0]
    x_size = x.shape[1]*x.shape[2]
    return x.reshape(x_0, x_size)

def compute(Weight, Input, Bias):
    return np.matmul(Input, Weight) + Bias

def averageCE(label, prediction):
    N = prediction.shape[0]
    avg_CE =  np.sum(np.multiply(label, np.log(prediction)))
    return -avg_CE/N

def gradCE(label, prediction):
    gradCE = np.divide(label, prediction)
    return -np.sum(gradCE)
```

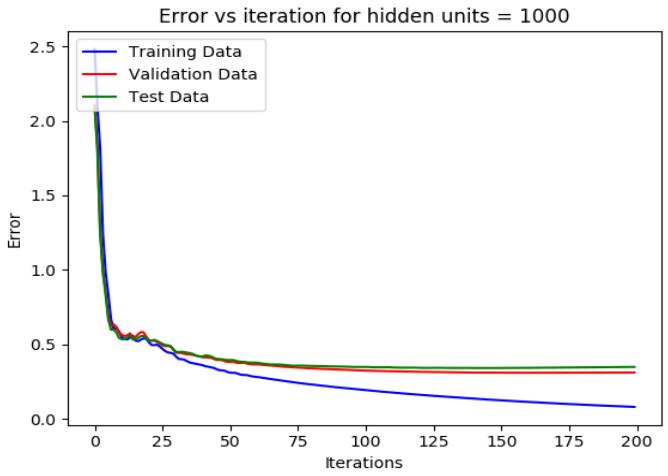
1.2 Backpropagation:

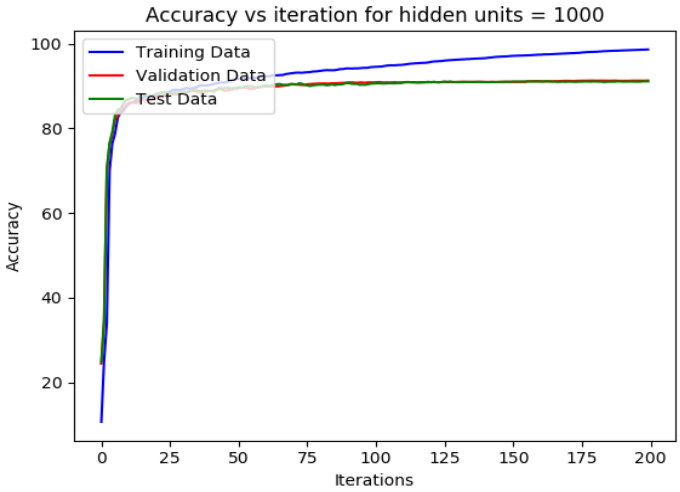
```
def backwardpropogation(Input, trainingLabels, W_hidden, W_out, Bias_hidden, Bias_out, Output_h, output_o, relu_diff):
    b_msg = backwardMessageOut(trainingLabels, output_o)
    row = np.multiply((np.matmul(b_msg, np.transpose(W_out))), relu_diff)
    dL_bo = (1/Output_h.shape[0])*np.sum(b_msg, axis = 0)
    dL_wo = (1/Output_h.shape[0])*np.matmul(np.transpose(Output_h), b_msg)
    dL_bh = (1/Input.shape[0])*np.sum(row, axis = 0)
    dL_wh = (1/Input.shape[0])*np.matmul(np.transpose(Input), row)
    return dL_bo, dL_wo, dL_bh, dL_wh
```

1.3 Learning:

We have trained our neural net with 1000 hidden units for 200 epochs using following parameters: Alpha = 0.07, Gamma=0.95

We notice that our Training Data has the highest accuracy compared to the other two. Also, at certain epoch the validation loss slightly increases due to overfitting.





Training outcomes:

```
Training loss for 1000 units=  0.08087903913539508
valid Loss for 1000 units= =  0.311624757124739
test loss for 1000 units=  0.3492205865971844
Training accuracy for 1000 units=  98.66
Valid accuracy for 1000 units=  91.31666666666666
Test accuracy for 1000 units=  91.11600587371512
```

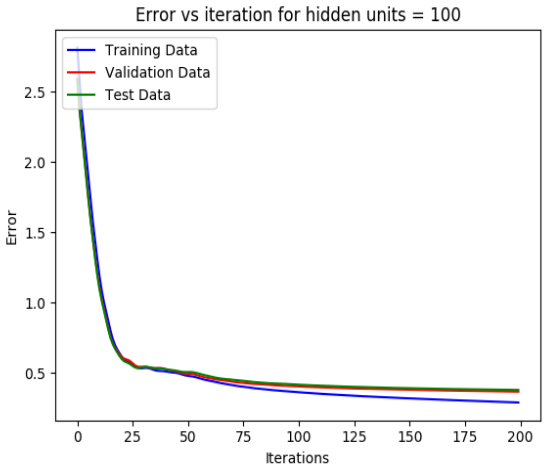
1.4 Hyperparameter Investigation:

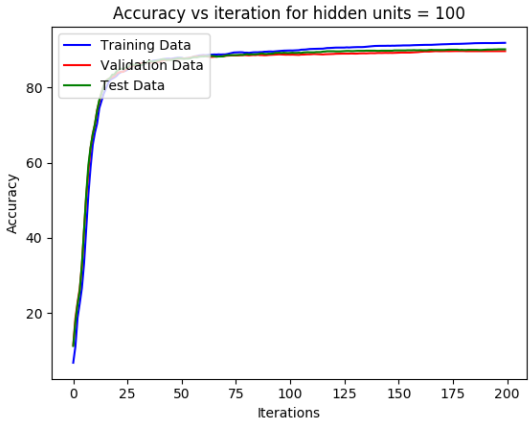
1. Number of hidden units:

We have trained our neural net for 200 epochs using following parameters:

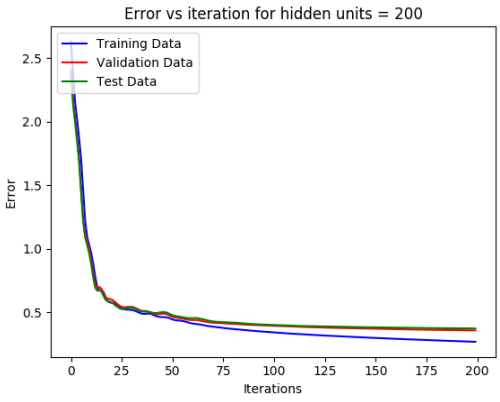
Alpha = 0.07, Gamma=0.95

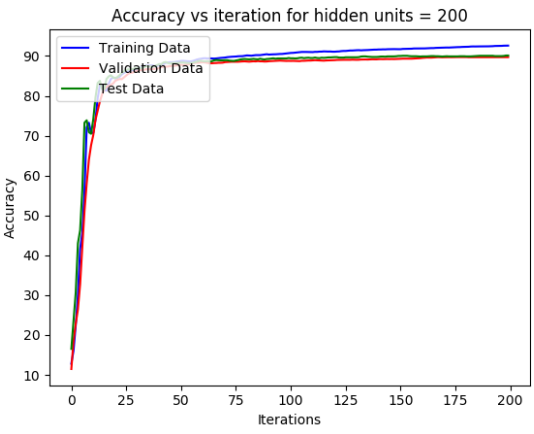
1.Hidden units = 100



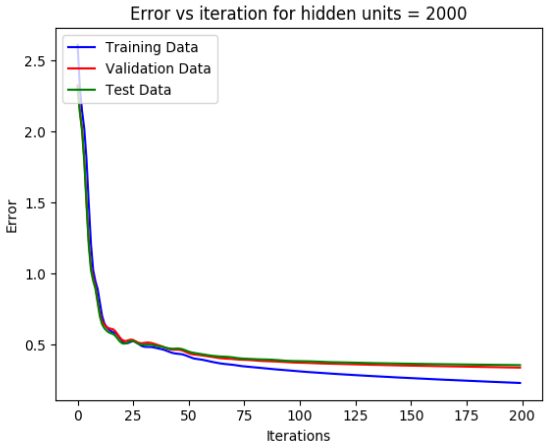


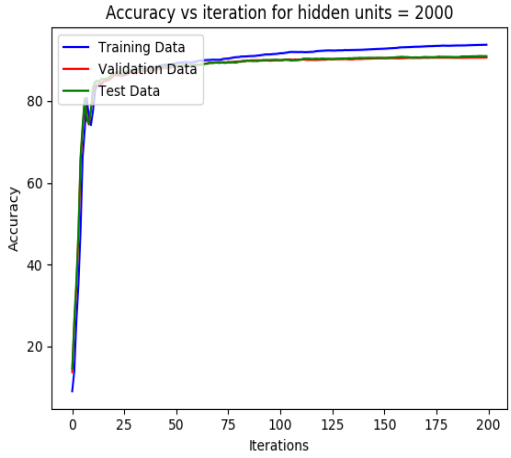
2.Hidden units = 200





3.Hidden units = 2000





Hidden Units	Training Loss	Validation Loss	Test Loss	Train Accuracy	Valid Accuracy	Test Accuracy
100	0.265	0.357	0.365	92.59	90.133	89.9
200	0.238	0.343	0.366	93.23	90.4	90.34
2000	0.068	0.299	0.332	99.03	91.6	91.37

We notice that on increasing hidden units the overall accuracy for all the three data sets (Test, train, Valid) our accuracy is improving. Although more hidden units make the training process slow. Also, we see the test and valid accuracy do not improve much due to overfitting, which increases with more hidden unit. Overfitting would be more evident in hidden units greater than 2000.

2. Early Stopping:

From the graph Loss vs Iteration graph in section 1.3 we observe that 168 iteration onwards the Validation loss starts to increase due to overfitting. So, 168 could be our early stopping point. We observe at this point our 'Training accuracy' is slightly lower than the final accuracy but our 'Validation accuracy' is slightly better than the final one and our 'test accuracy' is almost same as the final one.

At 168^{th} iteration we are getting following values:

Training Loss and Accuracy: 0.096 and 98.08%

Valid Loss and Accuracy: 0.31 and 91.3166%

Test Loss and Accuracy: 0.34 and 91.07%

Part 2 Neural Networks in Tensorflow

2.1 Model implementation

Code:

```
def shuffle(trainData, trainTarget):
    np.random.seed(421)
    randIndx = np.arange(len(trainData))
    target = trainTarget
    np.random.shuffle(randIndx)
    data, target = trainData[randIndx], target[randIndx]
    return data, target

trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
train_y, valid_y, test_y = convertOneHot(trainTarget, validTarget, testTarget)
#Reshape trainData, validData and test
train_x = trainData.reshape(-1, 28, 28, 1)
validation_x = validData.reshape(-1, 28, 28, 1)
test_x = testData.reshape(-1, 28, 28, 1)
```

```
training_epoch = 50 #epoch
learning_rate = 0.0001 #learning rate
mini_batch_size = 32 #batch size

x = tf.placeholder("float", [None, 28,28,1])
y = tf.placeholder("float", [None, 10])
W1 = tf.get_variable("W1", shape=[3,3,1,32],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
b1 = tf.get_variable("b1", shape=[32],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
W2 = tf.get_variable("W2", shape=[784*8,784],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
b2 = tf.get_variable("b2", shape=[784],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
W3 = tf.get_variable("W3", shape=[784,10],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
b3 = tf.get_variable("b3", shape=[10],initializer=tf.contrib.layers.xavier_initializer(),dtype=tf.float32)
```

```
# convolutional layer
# 1*1 strides
conv = tf.nn.conv2d(x, W1, strides=[1, 1, 1, 1], padding='SAME')
conv = tf.nn.bias_add(conv, b1)
conv = tf.nn.relu(conv)
batch_mean, batch_var = tf.nn.moments(conv,[0, 1, 2])
# batch normalization layer
Batch_normalization = tf.nn.batch_normalization(conv,batch_mean,batch_var,None,None,1e-5)
# 2x2 max pooling layer, 2x2 strides
max_pool = tf.nn.max_pool(Batch_normalization, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
# Fully connected layer 1
fc1 = tf.reshape(max_pool, [-1, W2.get_shape()[0]])
fc1 = tf.add(tf.matmul(fc1, W2), b2)
#dropout layers
#training = tf.placeholder_with_default(False, shape=(), name='training')
#dropout = tf.layers.dropout(fc1, 0.5, training=training)
#fc1 = tf.reshape(dropout, [-1, W3.get_shape()[0]])
fc1 = tf.nn.relu(fc1)
# Fully connected layer 2
fc2 = tf.reshape(fc1, [-1, W3.get_shape()[0]])
fc2 = tf.add(tf.matmul(fc2, W3), b3)
# Softmax cross entropy
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=fc2, labels=y))
# L2 regularization
#meanSquaredError = tf.losses.mean_squared_error(predictions=fc2, labels=y)
#weight_loss = (tf.nn.l2_loss(W1) + tf.nn.l2_loss(W2) + tf.nn.l2_loss(W3)) * 0.01 #lambda value
#cost = meanSquaredError + weight_loss
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
correct_prediction = tf.equal(tf.argmax(fc2, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
    sess.run(init)
    train_loss = []
    test_loss = []
    validation_loss = []
    train_accuracy = []
    validation_accuracy = []
    test_accuracy= []
    total_epoch=[]

    for epoch in range(training_epoch):
        total_epoch.append(epoch)
        sum_acc = 0
        sum_loss = 0
        for iter in range (int(10000 / mini_batch_size)):
            x_batch = train_x[(iter*mini_batch_size):(iter*mini_batch_size) + mini_batch_size]
            y_batch = train_y[(iter*mini_batch_size):(iter*mini_batch_size) + mini_batch_size]
            opt = sess.run(optimizer, feed_dict={x: x_batch,y: y_batch})
            trainloss, trainaccuracy = sess.run([cost, accuracy], feed_dict={x: x_batch,y: y_batch})
            sum_acc += trainaccuracy
            sum_loss += trainloss
        average_acc = sum_acc / int(10000 / mini_batch_size)
        average_loss = sum_loss / int(10000 / mini_batch_size)
        validloss,validaccuracy = sess.run([cost, accuracy], feed_dict={x: validation_x,y : valid_y})
        testloss, testaccuracy = sess.run([cost, accuracy], feed_dict={x: test_x,y: test_y})
        train_loss.append(average_loss)
        train_accuracy.append(average_acc)
        validation_loss.append(validloss)
        validation_accuracy.append(validaccuracy)
        test_loss.append(testloss)
        test_accuracy.append(testaccuracy)
        print("Epoch " + str(epoch+1) + ": Train Loss:" + "{:.3f}".format(average_loss) + ", Train Acc:" + "{:.2f}".fo
        #reshuffle training data
        train_x ,  train_y = shuffle(train_x, train_y)
```

```
def figPlot(figureNumber,datas, title,yLabel):
    f = plt.figure(figureNumber)
    plt.xlabel('epoch')
    title = title
    plt.title(title)
    plt.ylabel(yLabel)
    labels =[]
    for Value, Data in datas.items():
        plt.plot(Data)
        labels.append(Value)
    plt.legend(labels, ncol = len(labels))
    f.show()

figPlot(1,{"trainData" : train_loss, "validData" : validation_loss, "testData" : test_loss},"SGD with adam","Error")
figPlot(2,{"trainData" : train_accuracy, "validData" : validation_accuracy, "testData" : test_accuracy},"SGD with adam
```

2.2 Model training

Setting: SGD with batch size of 32, 50 epochs and Adam optimizer for learning rate of $1*10^4$

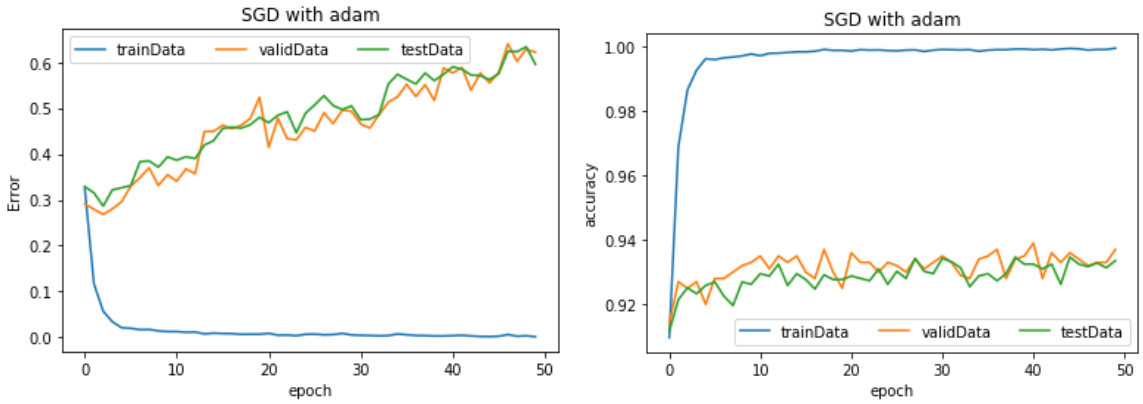


Figure 1, 2: SGD with Adam

Analysis:

By using SGD with Adam optimizer, we can figure from the graph that the test loss for trainData is converging to zero at early epochs while the test lost for validData and testData goes higher while training because of overfitting. We could use early stop condition to stop training in this case to prevent overfitting. From the accuracy graph, we can see that the trainData reach approximately 100% during training. The accuracy of validData nad testData increases with the number of epochs. These fluctuate a lot and also become very stagnant after certain epochs(due to overfitting). Also the accuracy of validData is slightly higher than testData overall to 93.50%

2.3 Hyperparameter Investigation

1. L2 Normalization

λ=0.01

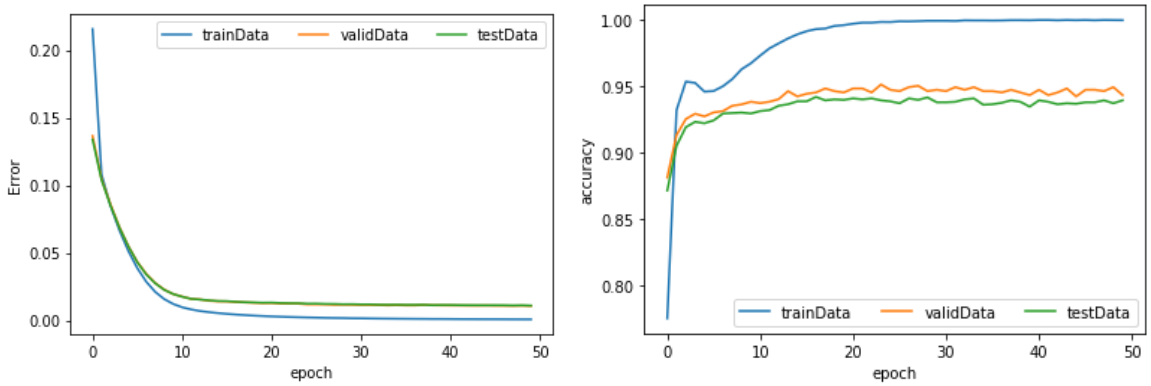


Figure 3, 4: SGD with L2, λ=0.01

λ=0.1

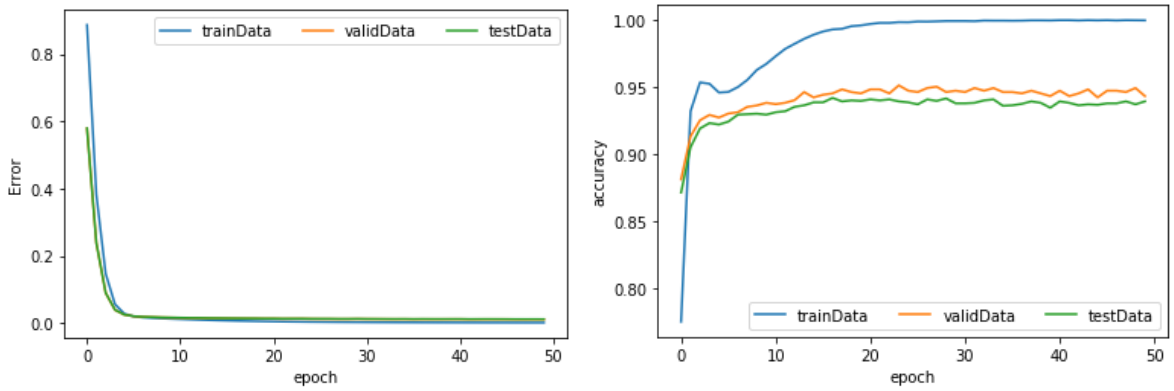


Figure 5, 6: SGD with L2, λ=0.1

λ=0.5

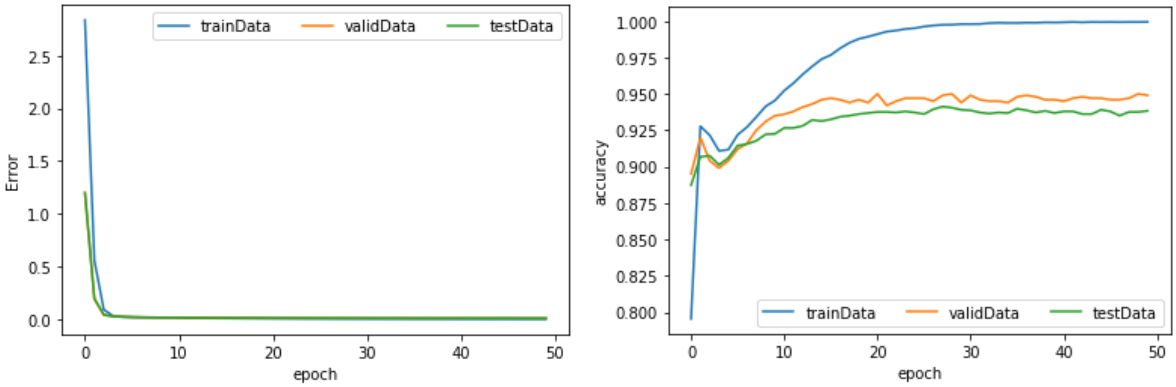


Figure 7, 8: SGD with L2, λ=0.5

Table of final lost and accuracy

Lost	λ=0.01	λ=0.1	λ=0.5
Training	0.0001	0.0001	0.0002
Validation	0.011	0.011	0.011
Test	0.011	0.011	0.012

Accuracy	λ=0.01	λ=0.1	λ=0.5
Training	99.93%	99.94%	99.95%
Validation	94.20%	94.80%	94.70%
Test	93.54%	93.91%	93.83%

Analysis:

After implementing L2 regularization for the weights, we can figure from the lost graphs that all three curves converge to zero during training. Also, the validation and test accuracies are increasing compared to SGD with Adam optimizer. The validation accuracy is always higher than test accuracy.

For different lambda values, we get the highest validation and test accuracy with lambda is 0.1 with 94.8% final validation accuracy and 93.9% final test accuracy. When lambda equals to 0.01(there is most overfitting), it may need more epochs to train as we can see the accuracy is still increasing. When lambda equals to 0.5, the model is getting lower accuracy as the overfitting decreases since the lambda value is getting large(Here is kind of underfits).

2. Dropout

p=0.9

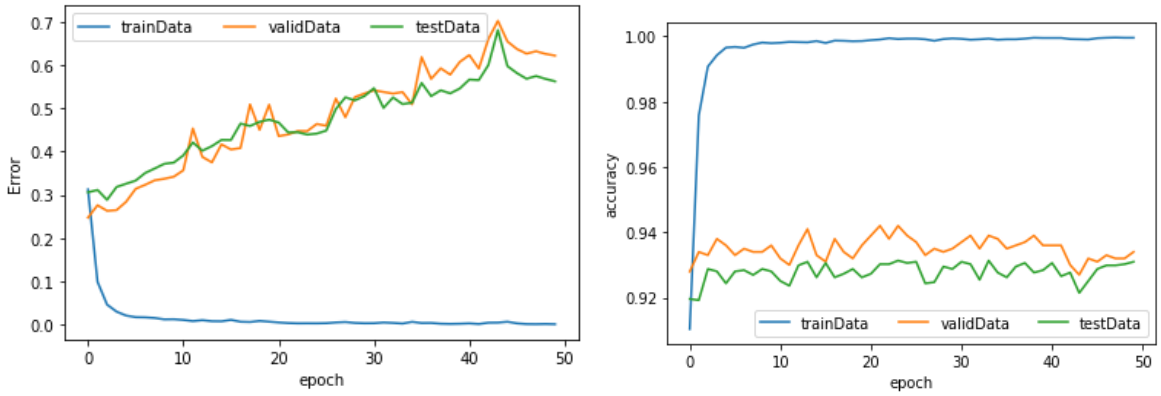


Figure 9, 10: Dropout with p=0.9

p=0.75

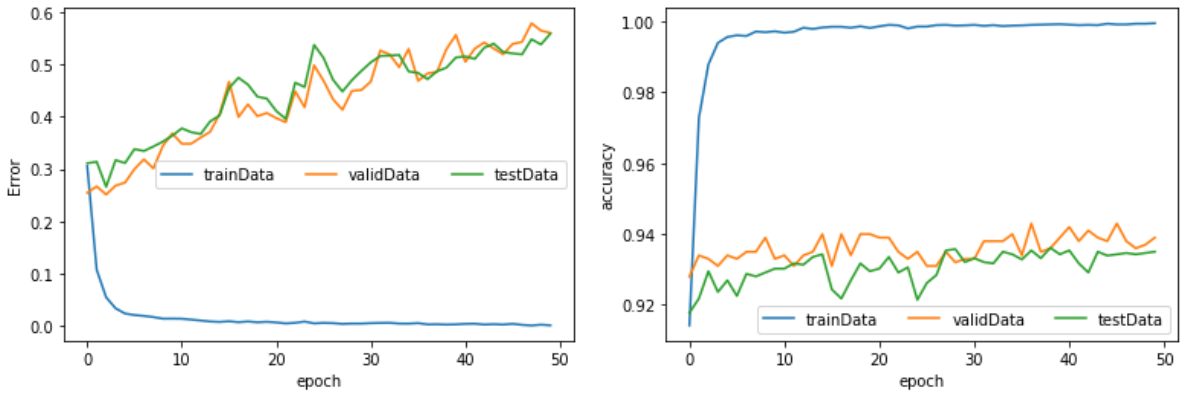


Figure 11, 12: Dropout with p=0.75

p=0.5

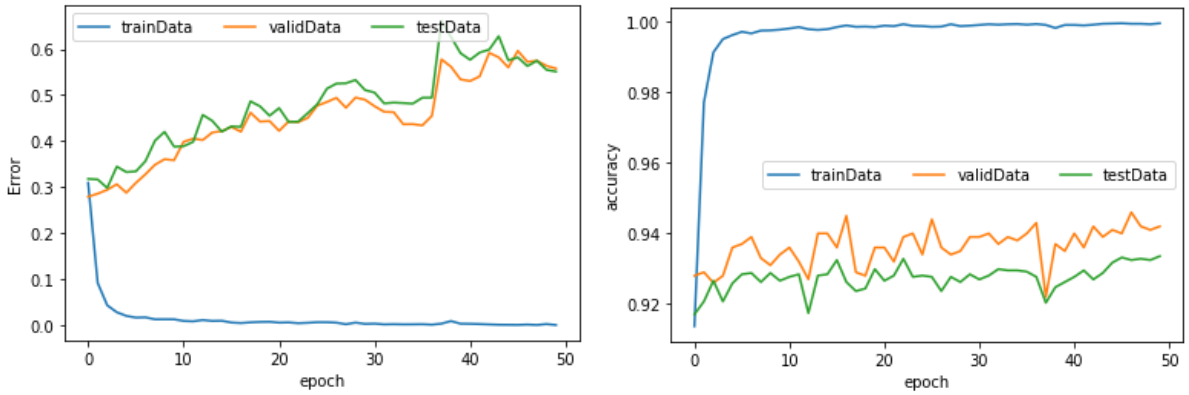


Figure 13, 14: Dropout with p=0.5

Table of final lost and accuracy

Lost	p=0.9	p=0.75	p=0.5
Training	0.001	0.002	0.001
Validation	0.622	0.560	0.558
Test	0.562	0.559	0.551

Accuracy	p=0.9	p=0.75	p=0.5
Training	99.95%	99.95%	99.96%
Validation	93.40%	93.90%	94.20%
Test	93.10%	93.41%	93.36%

Analysis:

By applying dropout layers in the model, we can see from the lost graphs that the test loss for validation and test data increases slower as the dropout layers control the overfitting. Also, the validation and test accuracies are getting higher compared to SGD with Adam optimizer. The validation accuracy is always higher than test accuracy.

For different p values, we get the highest validation and test accuracy with p equals to 0.5, as the dropout rate is 50%, it has 94.2% final validation accuracy and 93.36% final test accuracy(as overfitting decreases here). With higher p values, the dropout rate increases which cause the valid and test loss increases and the final accuracy decreases.