Introduction to Machine Learning
Winter 2020
Lab 2: Neural Networks

Team member	Contribution	Contribution percentage
Zibo Wen	1 (1.1-1.4)	50%
Yunyi Zhu	2 (2.1-2.3)	50%

0 Introduction

The purpose of this assignment is to investigate the classification performance of neural networks. In the first part, a neural network model using Numpy is implemented, and in the second part another model with the same function is built in Tensorflow.

The neural network models developed in both parts will be used for letter recognition. The dataset that we will use in this assignment is a permuted version of notMNIST1 used in the Linear and Logistic Regression study, which contains 28-by-28 images of 10 letters (A to J) in different fonts. This dataset has 18720 instances, which are divided into different sets for training, validation and testing and fed into the models.

1 Neural Networks using Numpy

The neural network implemented for this assignment has the following structure:

Layer Number: 3. One input, one hidden and one output.

Input Layer: x

Hidden Layer: $h = ReLU(W_h x + b_h)$

Output Layer: p = softmax(o), where $o = W_o h + b_o$

Loss Function: Cross Entropy Loss $L = -\sum_{k=1}^{K} y_k log(p_k)$

1.1 Helper Functions

Below are the snippets of our Python code implementation of the helper functions:

```
def relu(x):
    # Return x if x>0
    return np.maximum(0, x)


def softmax(x):
    # Subtract the maximum value of z from
    # all its elements to prevent overflow
    exp_x = np.exp(x - np.nanmax(x, axis=0))
    return exp_x / np.sum(exp_x, axis=0)
```

```python
def computeLayer(X, W, b):
    # Prediction of a given layer
    return (W @ X) + b


def averageCE(target, prediction):
    # Cross Entropy Loss
    N = target.shape[1]
    ce = -np.sum(target * np.log(prediction)) / N
    return ce


def gradCE(target, prediction):
    # Gradient of CE with respect to the softmax function outcome
    return prediction - target
```

Here is the analytical expression of the gradient of CE: $\frac{\partial L}{\partial o} = \sigma - y$

Steps:
For i = j in the softmax function:

$$\frac{\partial \sigma}{\partial o_j} = \frac{e^{o_i}\sum_{k=1}^{K} e^{o_k} - e^{o_j}e^{o_i}}{\left(\sum_{k=1}^{K} e^{o_k}\right)^2} = \frac{e^{o_i}\left(\sum_{k=1}^{K} e^{o_k} - e^{o_j}\right)}{\left(\sum_{k=1}^{K} e^{o_k}\right)^2} = \frac{e^{o_i}}{\sum_{k=1}^{K} e^{o_k}} \times \frac{\sum_{k=1}^{K} e^{o_k} - e^{o_j}}{\sum_{k=1}^{K} e^{o_k}}$$

$$= \sigma_{o_i} \cdot (1 - \sigma_{o_j})$$

For i ≠ j:

$$\frac{\partial \sigma}{\partial o_j} = \frac{0 - e^{o_j}e^{o_i}}{\left(\sum_{k=1}^{K} e^{o_k}\right)^2} = \frac{e^{o_j}}{\sum_{k=1}^{K} e^{o_k}} \times \frac{-e^{o_i}}{\sum_{k=1}^{K} e^{o_k}}$$

$$= -\sigma_{o_i} \cdot \sigma_{o_j}$$

Now we compute $\frac{\partial L}{\partial o}$:

$$\frac{\partial L}{\partial o_i} = \frac{\partial - \sum_{k=1}^{K} y_k^i \ log(\sigma_k^n)}{\partial o_i} = \ - \sum_{k=1}^{K} y_k^i \frac{\partial log(\sigma)}{\partial \sigma} \times \frac{\partial \sigma}{\partial o}$$

$$= \ - \sum_{k=1}^{K} y_k^i \frac{1}{\partial \sigma} \times \frac{\partial \sigma}{\partial o}$$

Then substitute $\frac{\partial \sigma}{\partial o}$ with the calculation result from last step:

$$= \ - [y_i \ (1 - \sigma_{o_i}) - \sum_{k \neq i}^{K} y_k \ (- \sigma_i)]$$

$$= \ [- y_i \ + y_i \cdot \sigma_{o_i} + \sum_{k \neq i}^{K} y_k \cdot \sigma_i]$$

$$= \ [- y_i \ + \sigma_{o_i} (y_i \ + \sum_{k \neq i}^{K} y_k)]$$

Note that y is one hot encoded, so: $(y_i \ + \sum_{k \neq i}^{K} y_k) = 1$

$$= \ - y_i \ + \sigma_{o_i}$$

$$\Rightarrow \frac{\partial L}{\partial o} = \frac{1}{N} (\sigma - y)$$

1.2 Backpropagation Derivation

- $\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial W_o} = \frac{1}{N}(\sigma - y) \cdot h^T$
- $\frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial b_o} = \frac{1}{N}\sum_{i=1}^{N}(\sigma_{o_i} - y_i)$

Define $S_h = W_h x + b_h$, then:

$\frac{\partial L}{\partial S_h} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial h} \cdot \frac{\partial h}{\partial S_h}$, where

$$\frac{\partial L}{\partial o} = \frac{1}{N}(\sigma - y),$$

$$\frac{\partial o}{\partial h} = \frac{\partial}{\partial h}(W_o h + b_o) = W_o,$$

$$\frac{\partial h}{\partial S_h} = 1 \; if \; S_h > 0; \frac{\partial h}{\partial S_h} = 0 \; if \; S_h \leq 0,$$

- $\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial S_h} \cdot \frac{\partial S_h}{\partial W_h} = \frac{\partial L}{\partial S_h} \cdot \frac{\partial (W_h x + b_h)}{\partial W_h} = \frac{\partial L}{\partial S_h} \cdot x^T$
- $\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial S_h} \cdot \frac{\partial S_h}{\partial b_h} = \frac{\partial L}{\partial S_h} \cdot \frac{\partial (W_h x + b_h)}{\partial b_h} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L}{\partial S_{h\,i}}$

The Python implementation of the calculations are here:

```
# Calculate gradients
dE_dS2 = gradCE(Y, y_hat)
dE_dw_2 = dE_dS2 @ x_1.T / N
```

```
    dE_db_2 = np.sum(dE_dS2, axis=1, keepdims=True) / N
    dE_dS1  = w_2.T @ dE_dS2 * (x_1 > 0)
    dE_dw_1 = dE_dS1 @ X.T / N
    dE_db_1 = np.sum(dE_dS1, axis=1, keepdims=True) / N

def gradCE(target, prediction):
   # Gradient of CE with respect to the softmax function outcome
   return (prediction - target)
```

1.3 Learning

After successful implementing the steps described in the preceding sections, I constructed the neural network model in Numpy. Here are the running results of the neural network trained with 1000 hidden units in 200 epochs:

Final Training Accuracy	0.9403
Final Validation Accuracy	0.9
Final Test Accuracy	0.9008810572687225

Table 1.3.A - Numpy Neural Network Performance

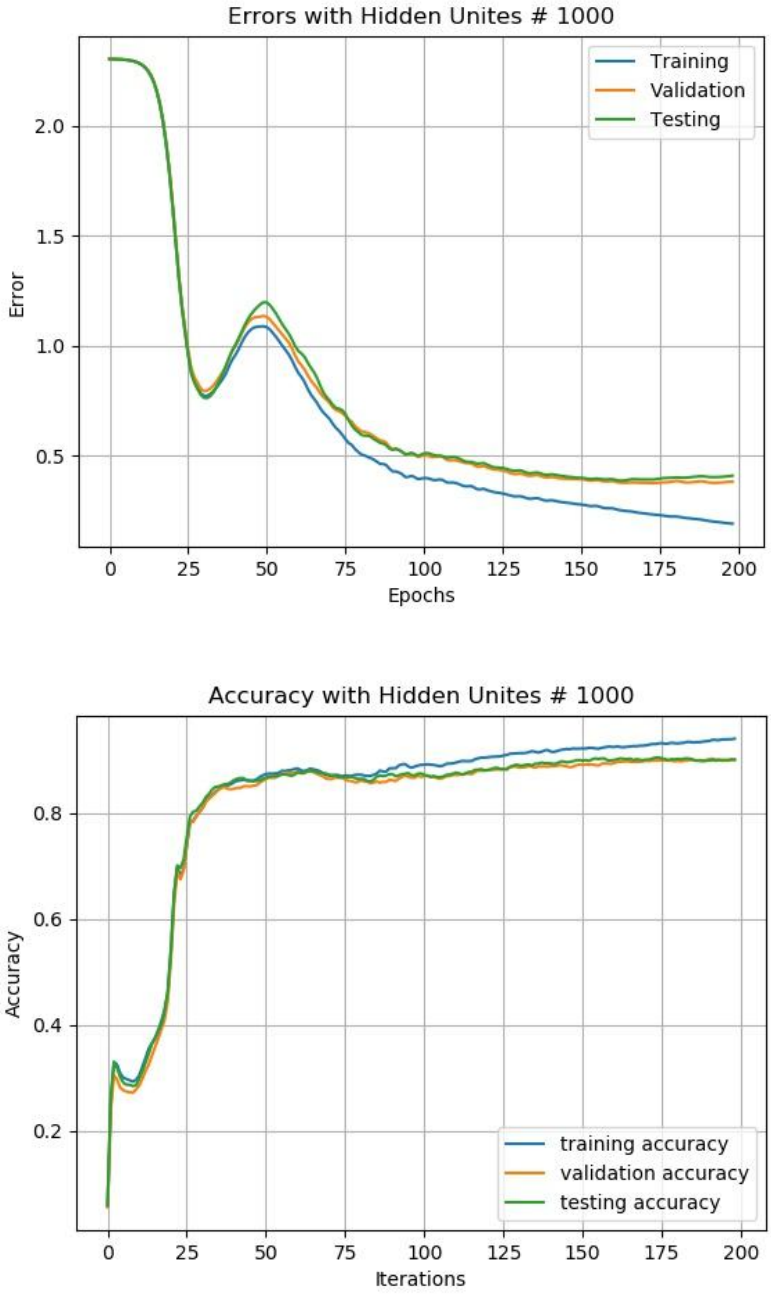

Figure 1.3.A - Numpy Neural Network Performance

1.4 Hyperparameter Investigation

1.4.1 Number of hidden units

From the running results listed below, we can see that as the number of hidden units increases, the final training accuracy increases correspondingly. Also note that as the number of hidden units increases from 100 to 500, the final validation accuracy is increased by almost 5% while the validation accuracy merely stays the same as the number grows from 500 to 2000. This suggests that the optimized number of hidden units might be around 500 in this case.

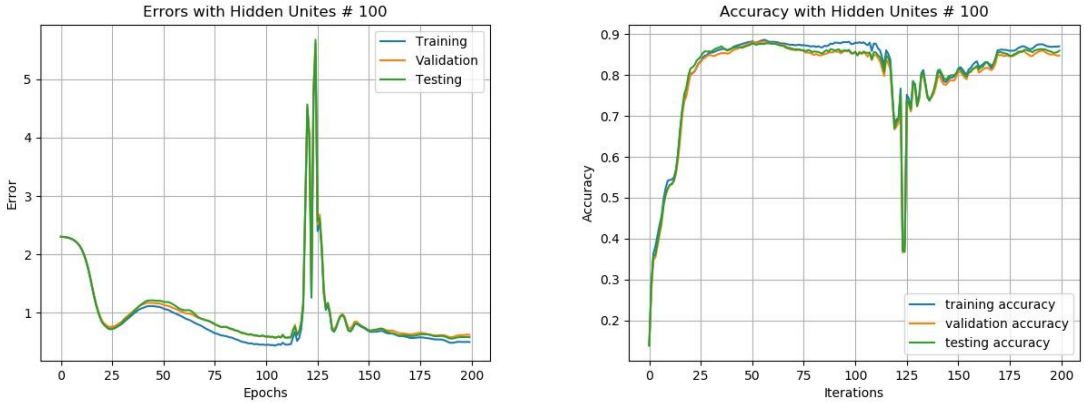

Figure 1.4.1.A - Numpy Neural Network Performance with 100 Hidden Units

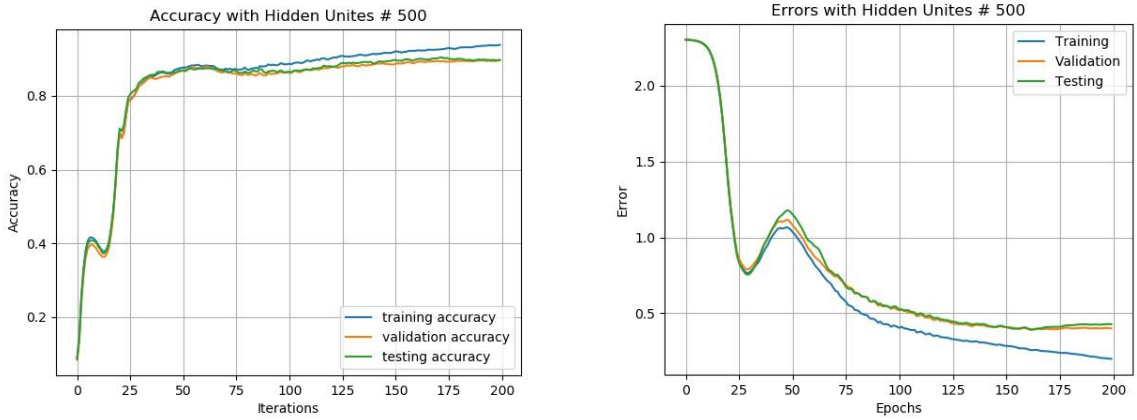

Figure 1.4.1.B - Numpy Neural Network Performance with 500 Hidden Units

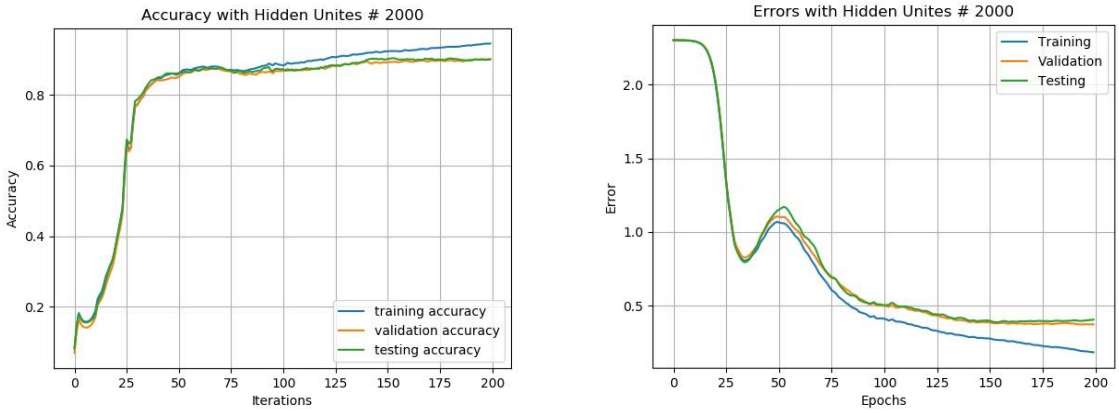

Figure 1.4.1.C - Numpy Neural Network Performance with 2000 Hidden Units

Number of Hidden Units	100	500	2000
Final Training Accuracy	0.8691	0.9371	0.9446
Final Validation Accuracy	0.847333333	0.8955	0.9006666
Final Test Accuracy	0.855726872	0.897209985	0.8997797

Table 1.4.1.A - Numpy Neural Network Performance with Different Number of Hidden Units

1.4.2 Early stopping

Early stopping can be used to reduce overfitting in neural networks. As the training gets iterated for too many times with the training data, the validation accuracy can drop while the training accuracy still increases.

From the running result with 1000 hidden units, we can see that it is not heavily influenced by overfitting. The highest validation accuracy happens at epoch number 191, and the validation accuracy slightly drops after that. This suggests that #191 epoch can be the early stopping point. At that point, the Training Accuracy is 0.9382, the Validation Accuracy is 0.9025, and the Accuracy is 0.9001468. The validation accuracy is slightly better than the final value, which is 0.900881.

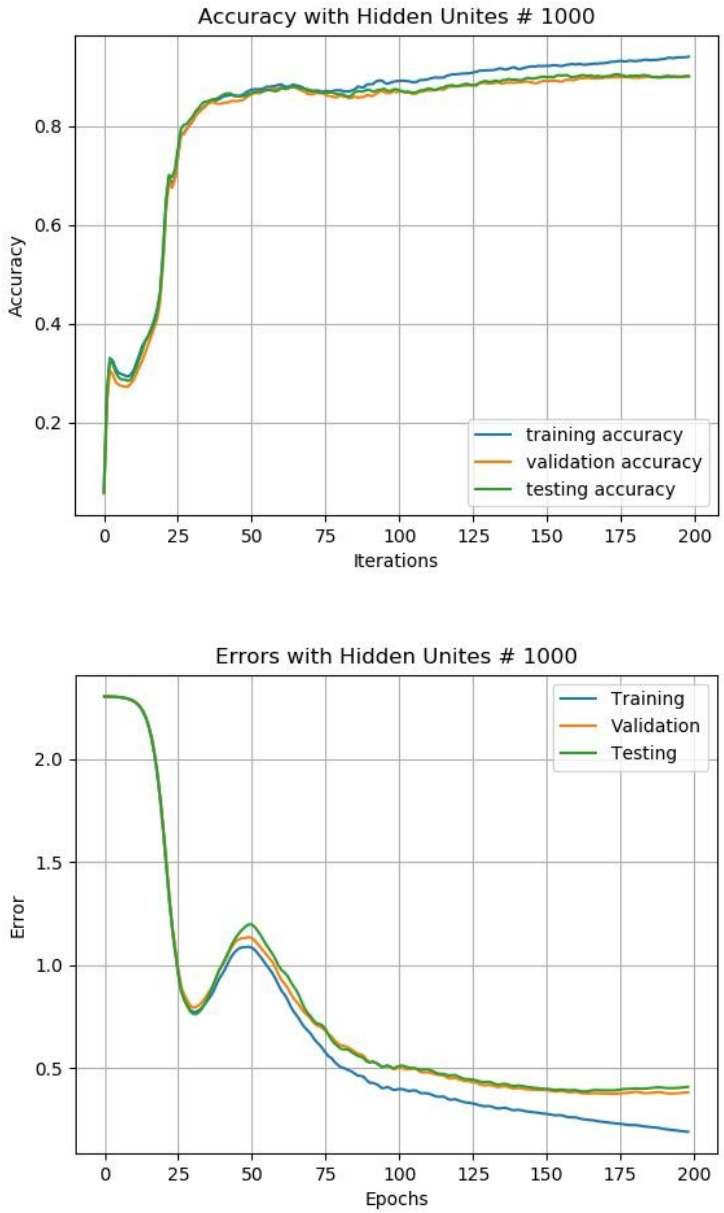

Figure 1.4.2.A - Numpy Neural Network Performance with 2000 Hidden Units

Final Training Accuracy	0.9
Final Validation Accuracy	0.9008810572687225
Final Test Accuracy	0.9025
Early Stop Point	#191
Early Stop Point Training Accuracy	0.9382
Early Stop Point Validation Accuracy	0.9025
Early Stop Point Test Accuracy	0.9001468428781204

Table 1.4.2.A - Numpy Neural Network Performance with 500 Hidden Units

2. Neural Networks in Tensorflow

2.1 Model implementation

With the help of the tutorial web page here https://www.datacamp.com/community/tutorials/cnn-tensorflow-python, I implemented the set of helper functions. According to the requirements, I implemented the 3 layers of neural network in 11 steps. Hyperparameters from 2.3 are also included but either commented out or set to zero in this code snippet.

```
#https://www.datacamp.com/community/tutorials/cnn-tensorflow-python
epoch = 50
learning_rate = 1e-04
batchSize = 32

# MNIST total classes (0-9 digits)
n_classes = 10

#Creating wrappers for simplicity
def conv2d(x, W, b, strides=1):
      # Conv2D wrapper, with bias and relu activation
      x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1],
padding='SAME')
      x = tf.nn.bias_add(x, b)
      return tf.nn.relu(x)

def maxpool2d(x, k=2):
      return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k,
1],padding='SAME')

'''
1. Input Layer
2. A 3 x 3 convolutional layer, with 32 filters, using vertical and
horizontal strides of 1.
3. ReLU activation
4. A batch normalization layer
5. A 2 x 2 max pooling layer
```

```
6. Flatten layer
7. Fully connected layer (with 784 output units, i.e. corresponding
to each pixel)
8. ReLU activation
9. Fully connected layer (with 10 output units, i.e. corresponding to
each class)
10. Softmax output
11. Cross Entropy loss
'''
def conv_net(x, weights, biases, labels):

      # 2 & 3
      conv1 = conv2d(x, weights['wc1'], biases['bc1'])

      # 4
      mean, variance = tf.nn.moments(conv1, axes=[0, 1, 2])
      batchNormalization = tf.nn.batch_normalization(conv1, mean,
variance, None, None, 1e-09)
      # 5
      # Max Pooling (down-sampling), this chooses the max value from
a 2*2 matrix window and outputs a 14*14 matrix.
      maxPoolingLayer = maxpool2d(batchNormalization)

      # 6 & 7 & 8
      fc1 = tf.reshape(maxPoolingLayer, [-1,
weights['wc2'].get_shape().as_list()[0]])
      fc1 = tf.add(tf.matmul(fc1, weights['wc2']), biases['bc2'])

      #2.3.2 dropout
      #fc1 = tf.nn.dropout(fc1,rate=0.1)
      #fc1 = tf.nn.dropout(fc1,rate=0.25)
      #fc1 = tf.nn.dropout(fc1,rate=0.5)

      fc1 = tf.nn.relu(fc1)

      #9
      fc2 = tf.reshape(fc1, [-1,
weights['out'].get_shape().as_list()[0]])
```

```
      fc2 = tf.add(tf.matmul(fc2, weights['out']), biases['out'])

      #10
      softmaxLayer = tf.nn.softmax(fc2)

      #11
      ceLost =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels,
softmaxLayer))

      #2.3.1 regularization
      ceLost = ceLost + 0*(tf.nn.l2_loss(weights['wc1']) +
tf.nn.l2_loss(weights['wc2']) + tf.nn.l2_loss(weights['out']))


      optimizer =
tf.train.AdamOptimizer(learning_rate).minimize(ceLost);

      #Here you check whether the index of the maximum value of the
predicted image is equal to the actual labelled image. and both will
be a column vector.
      correct_prediction = tf.equal(tf.argmax(softmaxLayer, 1),
tf.argmax(labels, 1))

      #calculate accuracy across all the given images and average
them out.
      accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))

      return ceLost, optimizer, accuracy
```

After the helper functions are written, the following scheme is used to initiate the weights and biases, and train the convolutional neural network.

```
trainData, validData, testData, trainTarget, validTarget, testTarget =
loadData();
# -1 means let the reshape decide on its own
trainData = trainData.reshape(-1, 28, 28, 1)
validData = validData.reshape(-1, 28, 28, 1)
```

```
testData = testData.reshape(-1, 28, 28, 1)
trainTarget, validTarget, testTarget = convertOneHot(trainTarget,
validTarget, testTarget);

#both placeholders are of type float
x = tf.placeholder("float", [None, 28,28,1])
y = tf.placeholder("float", [None, n_classes])

weights = {
      'wc1': tf.get_variable('W0', shape=(3,3,1,32),
initializer=tf.contrib.layers.xavier_initializer()),
      'wc2': tf.get_variable('W1', shape=(14*14*32,784),
initializer=tf.contrib.layers.xavier_initializer()),
      'out': tf.get_variable('W2', shape=(784,n_classes),
initializer=tf.contrib.layers.xavier_initializer())
}
biases = {
      'bc1': tf.get_variable('B0', shape=(32),
initializer=tf.contrib.layers.xavier_initializer()),
      'bc2': tf.get_variable('B1', shape=(784),
initializer=tf.contrib.layers.xavier_initializer()),
      'out': tf.get_variable('B2', shape=(n_classes),
initializer=tf.contrib.layers.xavier_initializer())
}

ceLost, optimizer, accuracy = conv_net(x, weights, biases, y);

init_op = tf.global_variables_initializer()

trainingError = []
validationError = []
testError = []
trainingAccuracy = []
validationAccuracy = []
testAccuracy = []

with tf.Session() as sess:
      sess.run(init_op)
      for i in range(epoch):
      trainData, trainTarget = shuffle(trainData, trainTarget);

      for j in range(int(len(trainData)/batchSize)):
```

```
            dataBatch = trainData[j*batchSize:(j+1)*batchSize]
            labelBatch = trainTarget[j*batchSize:(j+1)*batchSize]

            feed_dict={y: labelBatch,x: dataBatch}
            feed_dict_train={y: trainTarget,x: trainData}
            feed_dict_validation={y: validTarget,x: validData}
            feed_dict_test={y: testTarget,x: testData}

            sess.run(optimizer, feed_dict=feed_dict);

      trainingE, trainingA = sess.run([ceLost, accuracy],
feed_dict=feed_dict_train);
      validationE, validationA = sess.run([ceLost, accuracy],
feed_dict=feed_dict_validation);
      testE, testA = sess.run([ceLost, accuracy],
feed_dict=feed_dict_test);

      if 1:
            trainingError.append(trainingE)
            validationError.append(validationE)
            testError.append(testE)
            trainingAccuracy.append(trainingA)
            validationAccuracy.append(validationA)
            testAccuracy.append(testA)

      plt.figure(1, figsize=(10, 10))
      trainingErrorLine, = plt.plot(trainingError, label='trainingError')
      validationErrorLine, = plt.plot(validationError,
label='validationError')
      testErrorLine, = plt.plot(testError, label='testError')
      plt.ylabel('Error')
      plt.xlabel('Epoch')
      plt.title('Errors in Epoches')
      plt.legend([trainingErrorLine, validationErrorLine, testErrorLine],
['trainingError', 'validationError', 'testError'])

      plt.figure(2, figsize=(10, 10))
      trainingAccuracyLine, = plt.plot(trainingAccuracy,
label='trainingAccuracy')
      validationAccuracyLine, = plt.plot(validationAccuracy,
label='validationAccuracy')
      testAccuracyLine, = plt.plot(testAccuracy, label='testAccuracy')
```

```
      plt.ylabel('Accuracy')
      plt.xlabel('Epoch')
      plt.title('Accuracy in Epoches')
      plt.legend([trainingAccuracyLine, validationAccuracyLine,
testAccuracyLine], ['trainingAccuracy', 'validationAccuracy',
'testAccuracy'])

      plt.show()
```

2.2 Model Training

With the above implementation, the following results are observed.

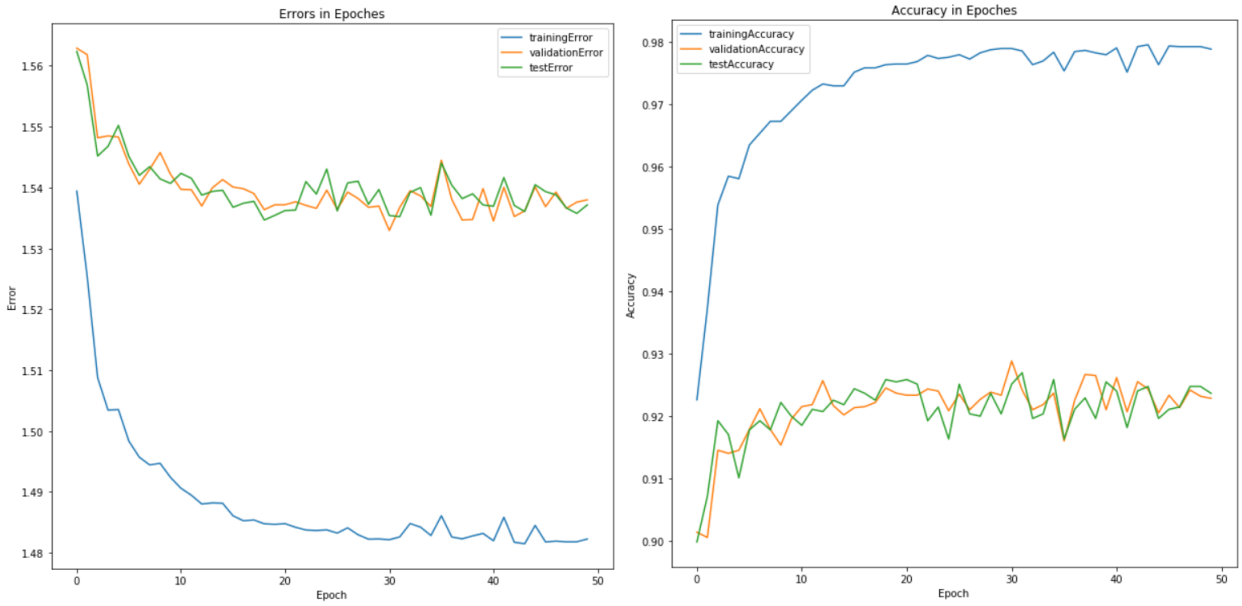

Figure 2.2.A - Tensorflow Neural Network Performance

Training Accuracy	0.9789
Validation Accuracy	0.9228333
Testing Accuracy	0.9236417
Training Error	1.4822401
Validation Error	1.5379802
Testing Error	1.5371443

Table 2.2.A - Tensorflow Neural Network Performance

2.3 Hyperparameter Investigation

2.3.1 L2 Regularization

With $\lambda = 0.01$:

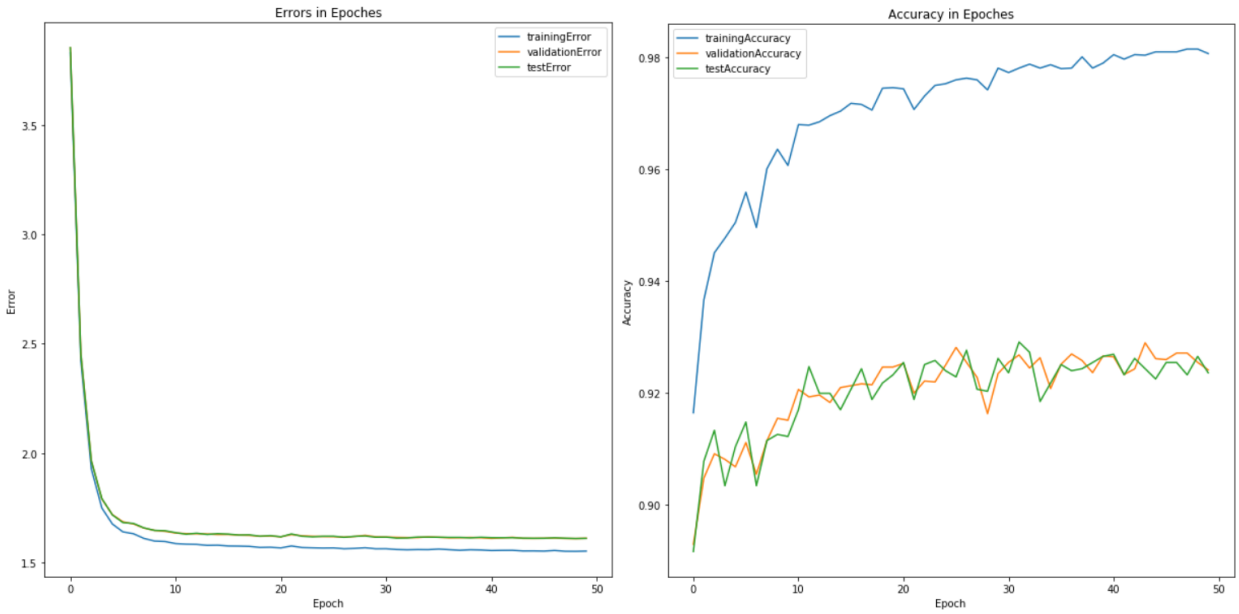

Figure 2.3.1.A - Tensorflow Neural Network Performance with $\lambda = 0.01$

With $\lambda = 0.1$:

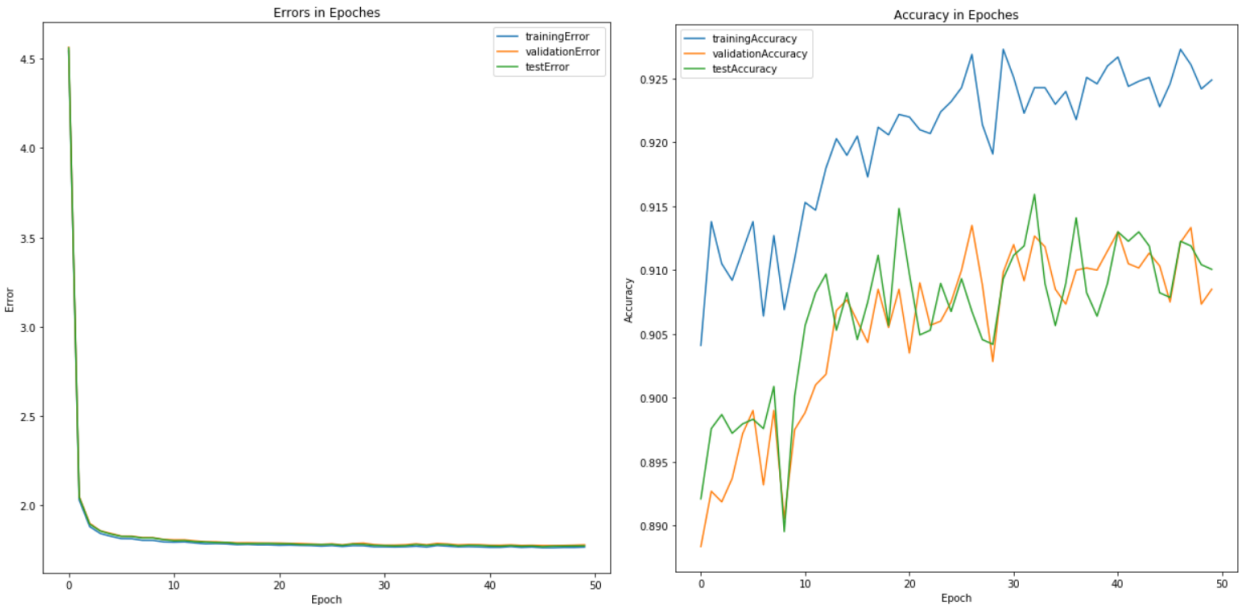

Figure 2.3.1.B - Tensorflow Neural Network Performance with $\lambda = 0.1$

With $\lambda = 0.5$:

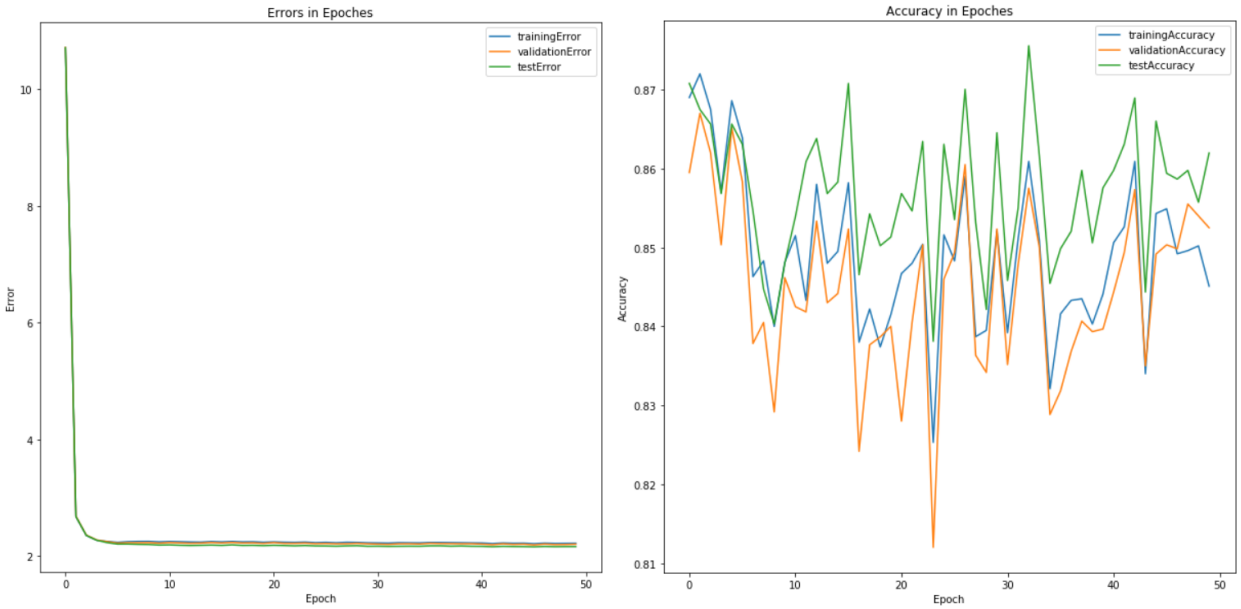

Figure 2.3.1.C - Tensorflow Neural Network Performance with $\lambda = 0.5$

L2 Regularization	λ=0.01	λ=0.1	λ=0.01
Training Accuracy	0.9807	0.9249	0.8451
Validation Accuracy	0.9241667	0.9085	0.8525
Testing Accuracy	0.9236417	0.91005874	0.8619677
Training Error	1.5514107	1.7674717	2.2223663
Validation Error	1.6100879	1.7801408	2.2062726
Testing Error	1.60982	1.7744949	2.1656213

Table 2.3.1.D - Tensorflow Neural Network Performance under Different L2 Regularization Settings

It can be seen that with a λ given, errors in validation and test are highly conforming with the error in training. This is quite different from the case without such regularization where testing error is 1.48, but validation and testing error are at 1.53. However, the differences in accuracy are still seen, and the overall accuracy is not increased when regularization is used. Moreover, the overall accuracy drops considerably with larger λ.

2.3.2 Dropout

With a keep probability of 0.9, the rate of 0.1 is fed into tf.nn.dropout.

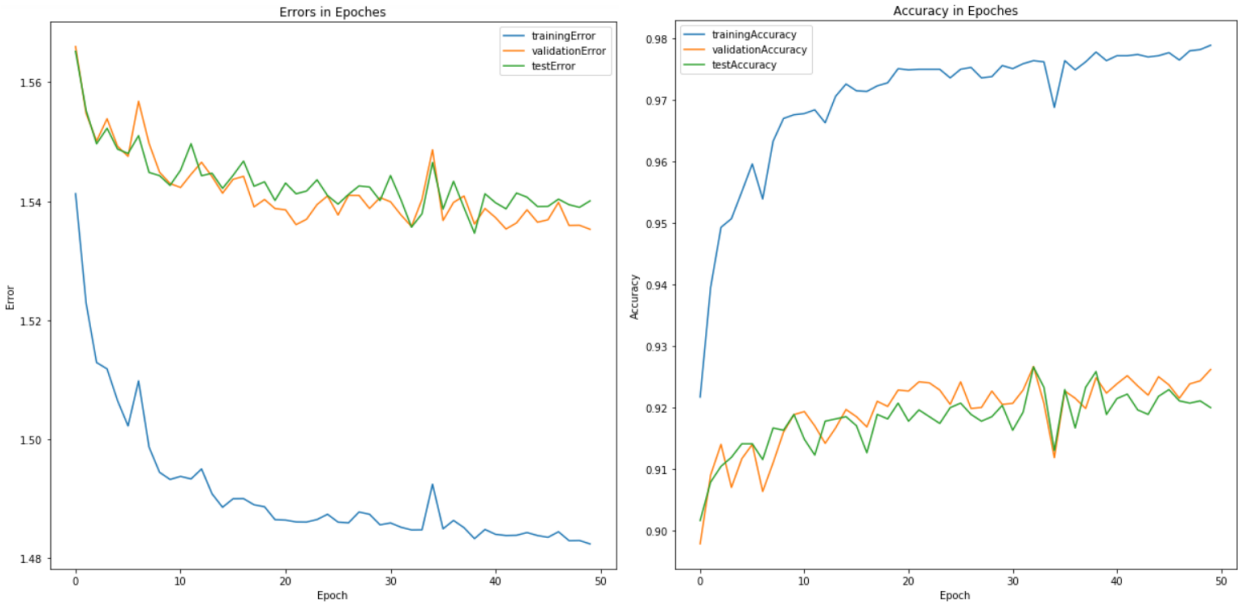

Figure 2.3.2.A - Tensorflow Neural Network Performance with $p = 0.9$

With a keep probability of 0.75:

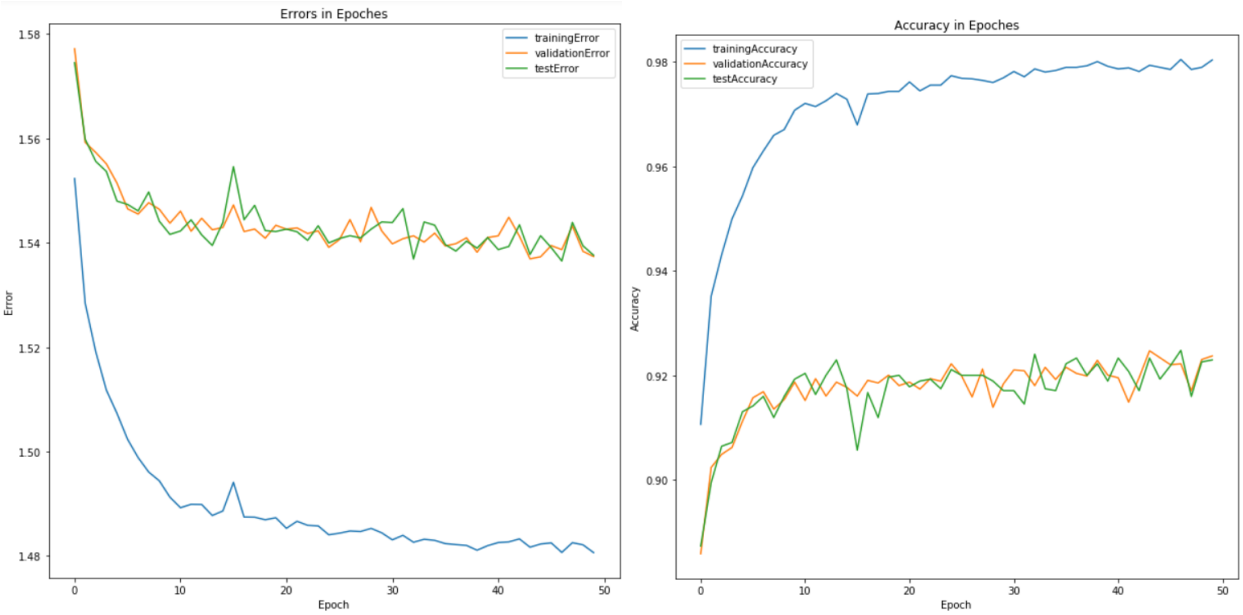

Figure 2.3.2.B - Tensorflow Neural Network Performance with $p = 0.75$

With a keep probability of 0.5:

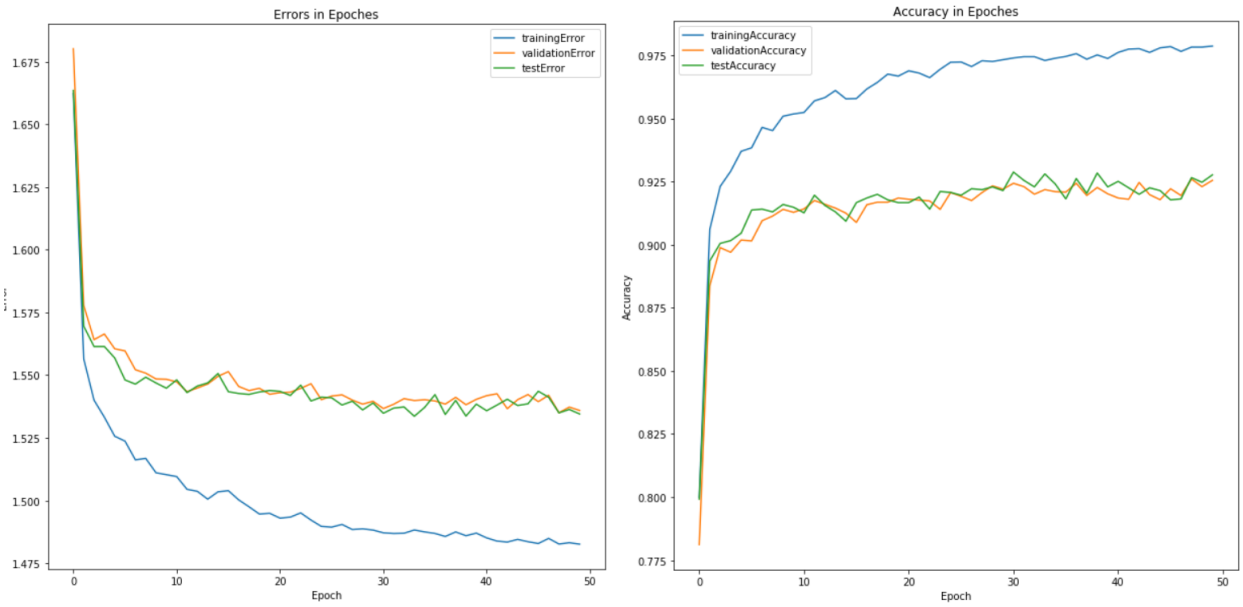

Figure 2.3.2.C - Tensorflow Neural Network Performance with $p = 0.5$

Probabilities of Keeping	*p*=0.9	*p*=0.75	*p*=0.5
Training Accuracy	0.9789	0.9803	0.9787
Validation Accuracy	0.92616665	0.92366666	0.9255
Testing Accuracy	0.91997063	0.9229075	0.9276799
Training Error	1.4823995	1.4806646	1.4826415
Validation Error	1.5352894	1.5373615	1.5359231
Testing Error	1.5400543	1.537609	1.5345093

Table 2.3.2.C - Tensorflow Neural Network Performance with Different Dropout Probabilities

It is observed that overfitting is prevented as dropout is introduced with all three probabilities; however, the overall accuracy and the shapes of the plots are not drastically affected by different rates of dropout. The impact of different dropout probabilities on the performance of neural networks is not clearly shown in this case.