Part 1 . Neural Networks using NumPy

1.1 Helper Functions

1. ReLU( ):

```
def relu(x):

    return x*(x>0)
```

2. Softmax( ):

```
def softmax(x):

    return np.exp(x)/np.sum(np.exp(x),axis=0)
```

3. compute( ):

```
def computeLayer(X, W, b):

    S = np.dot(np.tranpose(W),X) + b

    return S
```

4. averageCE( ):

```
def CE(target, prediction):

    return -np.sum(np.sum(np.transpose(target)*np.log(prediction),axis=0))/len(target)
```

5. gradCE( ):

$$\text{aveage CE} = -\frac{1}{\mathrm{N}}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \log s_k^n$$

compute gradient of Cross Entropy loss E with respect to activation function: $\frac{\partial \mathrm{E}}{\partial \mathrm{a}_j}$

where E is the average CE, $\mathrm{a_j}$ is the activation function input.

For simplicity set N =1, then $\mathrm{E} = -\sum_{k=1}^{K} t_k \log \mathrm{s_k}$

Then softmax function output is $\mathrm{s_k} = \frac{e^{a_k}}{\sum_{k=1}^{k} e^{a_k}}$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{a}_j} = \frac{\partial \mathrm{E}}{\partial s_k}\frac{\partial s_k}{\partial \mathrm{a}_j} = -\sum_{\mathrm{k}=1}^{\mathrm{k}} \frac{t_k}{s_k}\frac{\partial s_k}{\partial \mathrm{a}_j} \quad \mathrm{j} = 1, \ldots . \mathrm{K}$$

$$\text{For j = k, } \frac{\partial s_k}{\partial \mathrm{a}_j} = \frac{e^{a_j}(\sum_{k=1}^{k} e^{a_k} - e^{a_j})}{(\sum_{k=1}^{k} e^{a_k})^2} = \mathrm{s_k}(1 - \mathrm{s_j})$$

$$\text{For j} \neq \text{ k}, \frac{\partial s_k}{\partial a_j} = \frac{-e^{a_j} e^{a_k}}{\left(\sum_{k=1}^{k} e^{a_k}\right)^2} = -s_j s_k$$

Combine all the equations above,

$$\frac{\partial E}{\partial a_j} = -t_j(1 - s_j) + \sum_{k \neq j} t_k s_j = s_j - t_j \quad j = 1, \ldots\ldots K$$

```
def gradCE(target, prediction):

    gradientCE = np.transpose(prediction - np.transpose(target))

    return gradientCE
```

1.2 Backward Propagation

1. The gradient of the loss with respect to the outer layer weights

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_0} = \bar{x} \cdot (\bar{s} - \bar{t})^T$$

   where:
   x is the output data from the first layer
   t is the data label
   s is the output from softmax

2. The gradient of the loss with respect to the outer layer biases:

$$\frac{\partial L}{\partial b_0} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial b_0} = (\bar{s} - \bar{t})^T$$

   where:
   t is the data label
   s is the output from softmax

3. The gradient of the loss with respect to the hidden layer weights:

$$\frac{\partial L}{\partial w_h} = x^{(0)} \cdot (w^{(2)} \cdot \delta^{(2)} \otimes \theta'(s^{(1)}))^T$$

$$\theta'(s^{(1)}) = \text{derivative of ReLu} = \begin{cases} 0 & \text{if } s^{(1)} < 0, \\ 1 & \text{if } s^{(1)} > 0 \end{cases}$$

   where:
   $x^{(0)}$ is the input data
   $w^{(2)}$ is the weight matrices for the outer layer
   $\delta^{(2)}$ is the derivative of loss with respect to softmax input
   θ' is the derivative of the ReLu

4. The gradient of the loss with respect to the hidden layer biases:

$$\frac{\partial L}{\partial b_h} = (w^{(2)} \cdot \delta^{(2)} \otimes \theta'(s^{(1)}))^T$$

   where:
   $w^{(2)}$ is the weight matrices for the outer layer

$\delta^{(2)}$ is the derivative of loss with respect to softmax input

θ' is the derivative of the ReLu

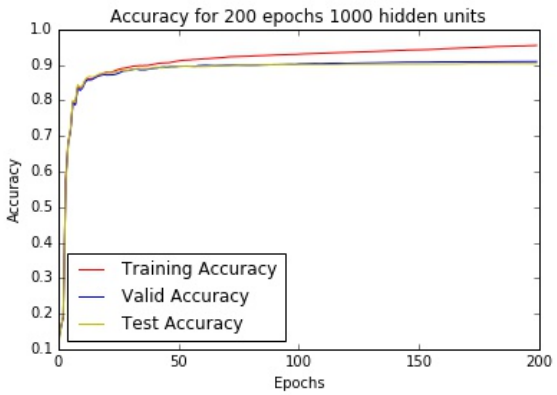
For neural network update implementation, the gradients are the averages of all sample data gradients.

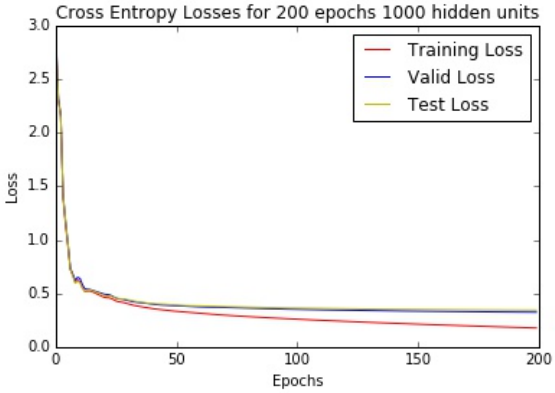
```
def grad_weight2(X,Y,S):
delta = gradCE(Y,S)
return np.dot(X,delta)/len(Y)

def grad_bias2(Y,S):
b = np.sum((S-np.transpose(Y)),axis=1)
return np.transpose(np.reshape(b,(len(b),1)))/len(Y)

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

def grad_weight1(X,W,delta,dtheta):
delta0 = np.transpose(np.dot(W,np.transpose(delta))*dtheta)
return np.dot(X,delta0)/len(X[1,:])

def grad_bias1(W,delta,dtheta):
b = np.sum(np.transpose(np.dot(W,np.transpose(delta))*dtheta),axis=0)
return np.transpose(np.reshape(b,(len(W),1)))/len(delta)
```

1.3 Learning

For implementation of the model, epoch is set to 200 with 1000 hidden units. The learning rate is set to 0.05 with with a momentum parameter 0.9. The weights and biases are initialized via Xavier initilization scheme.

	Train	Validation	Test
Loss	0.1787	0.3255	0.3451
Accuracy	0.9547	0.9095	0.9035



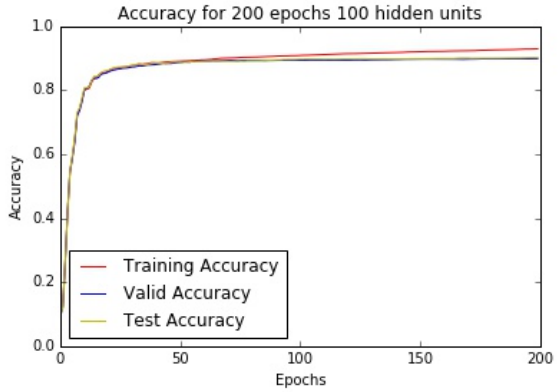


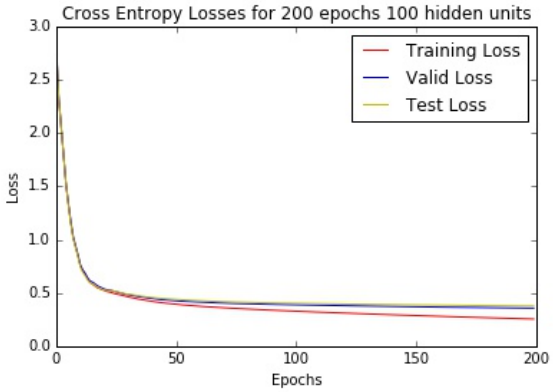
1.4 Hyperparameter Investigation

1. Number of Hidden Units

1. Hidden units = 100

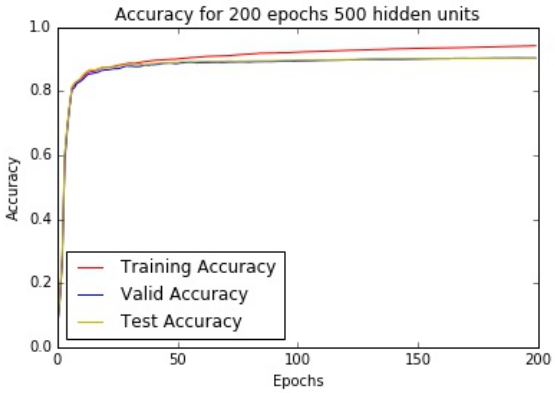
	Train	Validation	Test
Loss	0.2562	0.3584	0.3784
Accuracy	0.9298	0.8990	0.9023

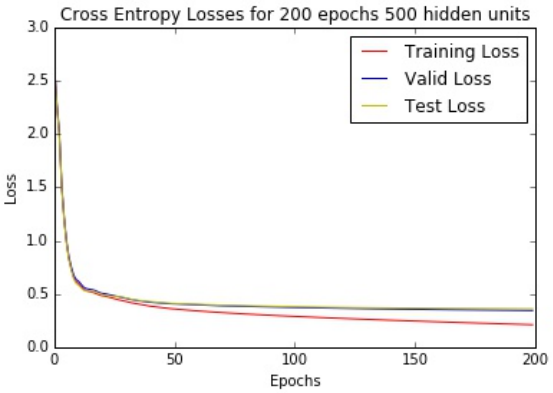




2. Hidden units = 500

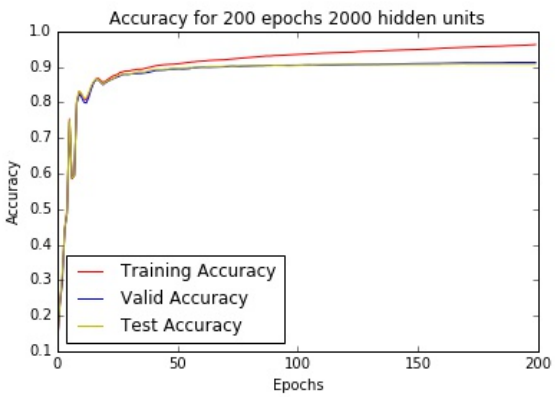
	Train	Validation	Test
Loss	0.2123	0.3456	0.3614
Accuracy	0.9422	0.9042	0.9031

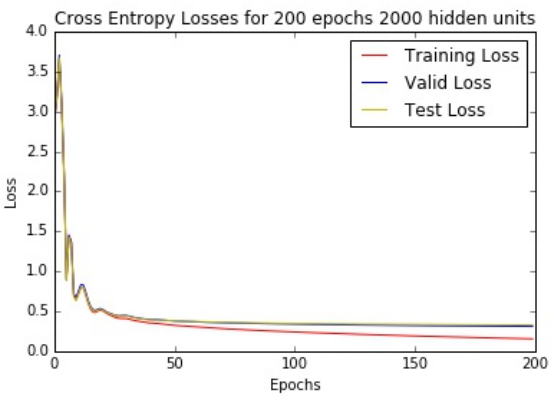




3. Hidden units = 2000

	Train	Validation	Test
Loss	0.1566	0.3126	0.3309
Accuracy	0.9628	0.9123	0.9075

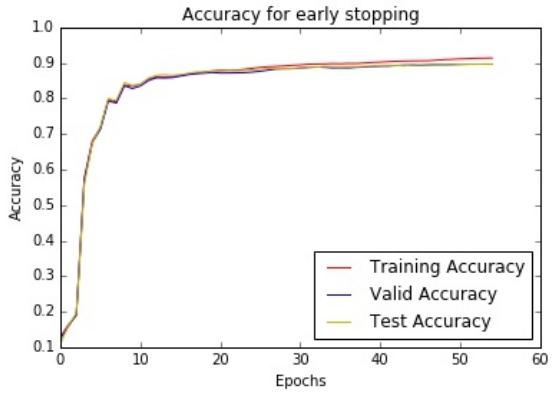


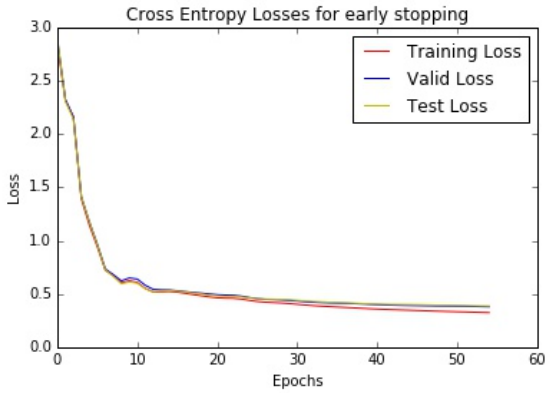


2. Early Stopping

If the accuracy for validation set starts to decrease, then it means that there is overfitting in the training process. In order to implement early stopping in the model, a condition is added to the model. As the training error becomes steady (ie. after 50 epochs), if the validation accuracy starts decreasing, then stop the training process. In this case, for a hidden units of 1000, learning rate of 0.05, the training stops at 55 epochs.

	Train	Validation	Test
Loss	0.3253	0.3804	0.3877
Accuracy	0.9137	0.8962	0.8965





Part 2. Neural Network in Tensorflow

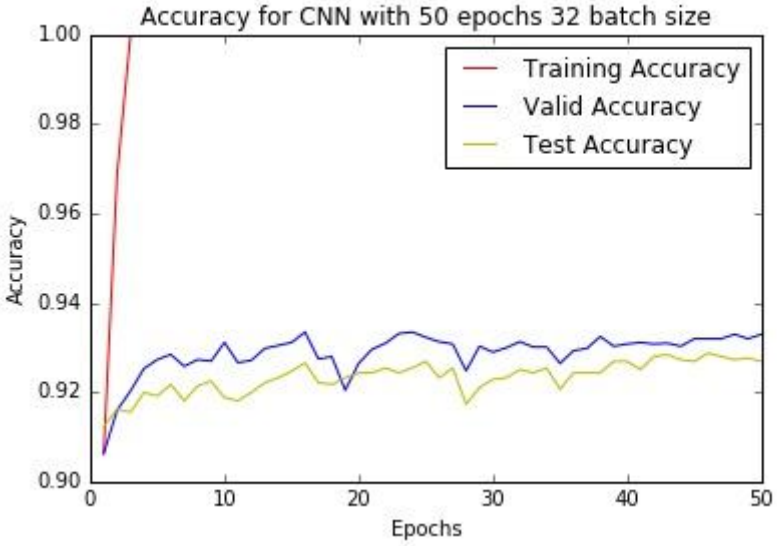
2.1 Model Implementation

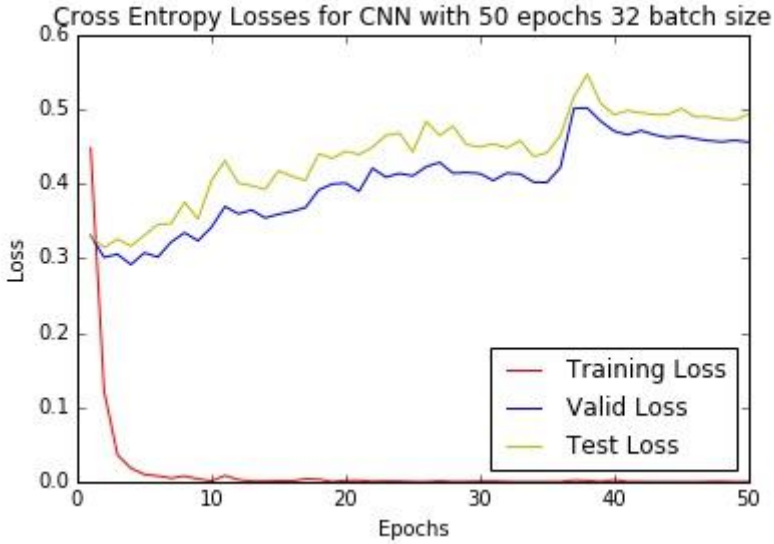
```
# Define convolutional layer
def conv_layer(X,W,b,stride=1):
    X = tf.nn.conv2d(X,W,strides=[1,stride,stride,1],padding = 'SAME')
    X = tf.nn.bias_add(X,b)
    return tf.nn.relu(X)
# Define max pooling layer
def max_pooling(X,K=2):
    return tf.nn.max_pool(X,ksize = [1,K,K,1], strides = [1,K,K,1], padding = 'SAME')
# Define batch normalization and flatten layer
def batch_norm_flatten(X):
    mean,variance = tf.nn.moments(X,axes=[0,1,2])
    X = tf.nn.batch_normalization(X,mean,variance,offset=None,scale=None,variance_epsilon=1e-3)
    size = X.get_shape().as_list()
    X = tf.reshape(X,[-1,size[1]*size[2]*size[3]])
    return X
 # Define fully connected layer
def fully_c(X,W,b):
    return tf.add(tf.matmul(X,W),b)
 # Combine all the layers together
def con_net(X,Y,W,b):
```

```
    conv = conv_layer(X,W['wc1'],b['bc1'],stride=1)
    pool = max_pooling(conv,K=2)
    flatpool = batch_norm_flatten(pool)
    fc1=tf.nn.relu(fully_c(flatpool,W['wfc1'],b['bfc1']))
    fc2=fully_c(fc1,W['wfc2'],b['bfc2'])
    return fc2

def reshape_data(X):
    return X.reshape(-1,28,28,1)

trainData = trainData.reshape(-1,28,28,1)
validData = validData.reshape(-1,28,28,1)
testData = testData.reshape(-1,28,28,1)

epoch = 50
batch_size = 32
learning_rate = 1e-4
reg=0.5
p= 0.25

tf.reset_default_graph()
weights = {
           'wc1': tf.get_variable('W0', shape=(3,3,1,32), initializer=tf.contrib.layers.xavier_initializer()),
           'wfc1': tf.get_variable('W1', shape=(14*14*32,784), initializer=tf.contrib.layers.xavier_initializer()),
           'wfc2': tf.get_variable('W2', shape=(784,10), initializer=tf.contrib.layers.xavier_initializer())
}

biases = {
          'bc1': tf.get_variable('B0', shape=(32), initializer=tf.contrib.layers.xavier_initializer()),
          'bfc1': tf.get_variable('B1', shape=(784), initializer=tf.contrib.layers.xavier_initializer()),
          'bfc2': tf.get_variable('B2', shape=(10), initializer=tf.contrib.layers.xavier_initializer()),
}
X = tf.placeholder("float",[None, 28,28,1])
Y = tf.placeholder("float",[None, 10])
keep_prob = tf.placeholder("float")
pred = con_net(X,Y,weights,biases,keep_prob)
last_layer = tf.nn.softmax(pred)
# Compute the cost through softmax cross entropy function
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=pred,labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
correct_prediction = tf.equal(tf.argmax(last_layer, 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 = []
    valid_loss = []
    test_loss = []
    train_accuracy = []
    valid_accuracy = []
    test_accuracy = []
    iterations = []
    curr_itr = 0
    summary_writer = tf.summary.FileWriter('./Output', sess.graph)
    for i in range(epoch):
        trainData, newtrain = shuffle(trainData,newtrain)
        for batch in range(len(trainData)//batch_size):

            batch_x = trainData[batch*batch_size:(batch+1)*batch_size]
            batch_y = newtrain[batch*batch_size:(batch+1)*batch_size]
            opt = sess.run(optimizer, feed_dict={X: batch_x,
                                                 Y: batch_y, keep_prob:p})

        loss, acc = sess.run([cost, accuracy], feed_dict={X: batch_x,
                                                          Y: batch_y, keep_prob:p})
        print("Iter " + str(i) + ", Loss= " + \
                      "{:.6f}".format(loss) + ", Training Accuracy= " + \
                      "{:.5f}".format(acc))
        print("Optimization Finished!")

        # Calculate accuracy
        test_acc,test_cost = sess.run([accuracy,cost], feed_dict={X: testData,Y : newtest, keep_pr
ob:p})
        valid_acc,valid_cost = sess.run([accuracy,cost], feed_dict={X: validData,Y : newvalid, kee
p_prob:p})
        train_loss.append(loss)
        test_loss.append(test_cost)
         valid_loss.append(valid_cost)
         train_accuracy.append(acc)
         test_accuracy.append(test_acc)
         valid_accuracy.append(valid_acc)
         curr_itr+=1
         iterations.append(curr_itr)
         print("Validation Accuracy:","{:.5f}".format(valid_acc))
         print("Testing Accuracy:","{:.5f}".format(test_acc))
    summary_writer.close()
```

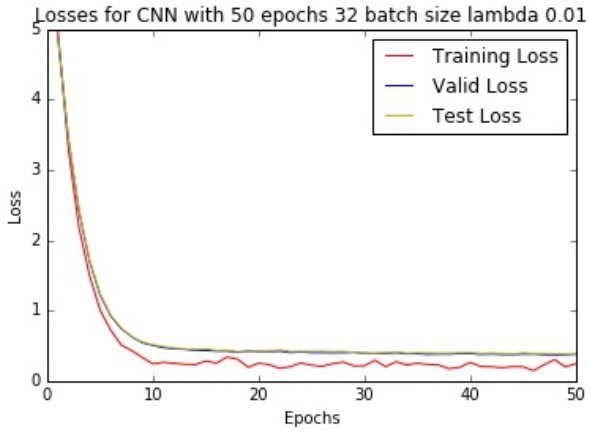
2.2 Model Training

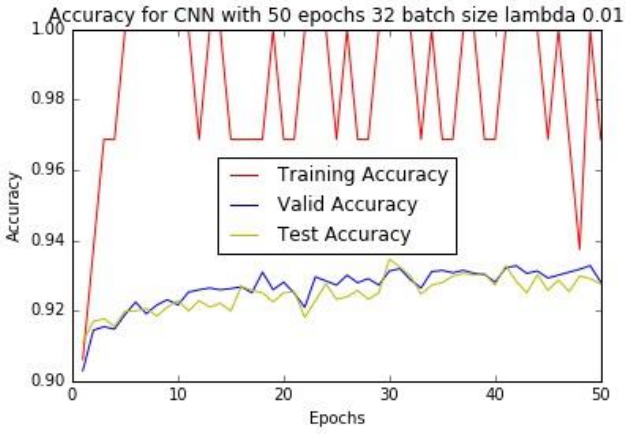


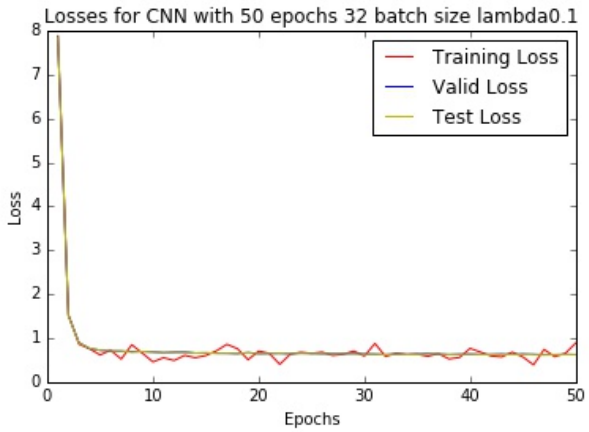


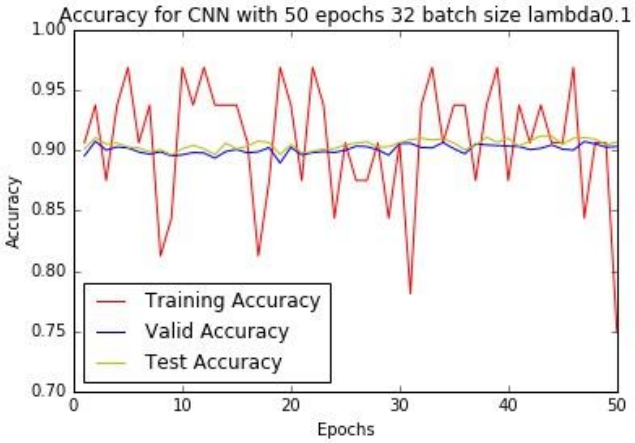
2.3 Hyperparameter Investigation

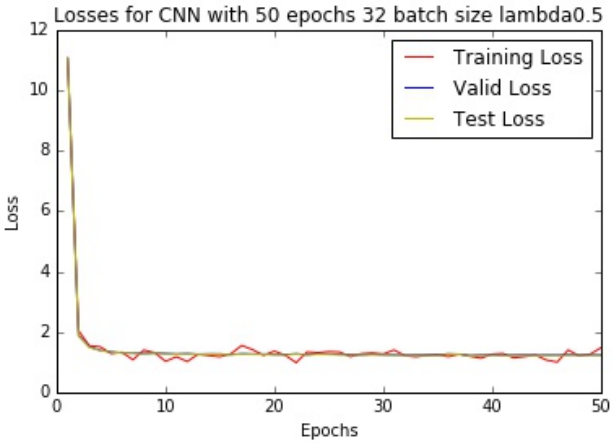
a) L2 Normalization

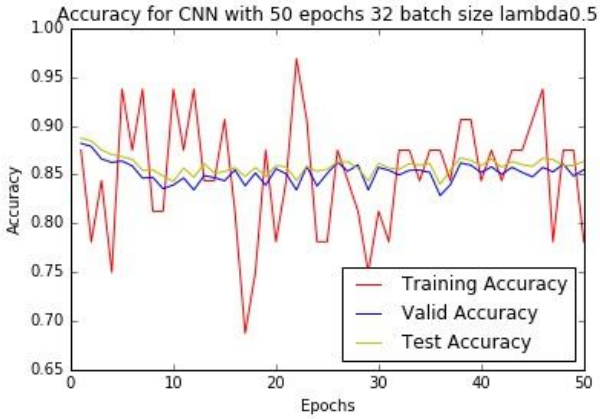








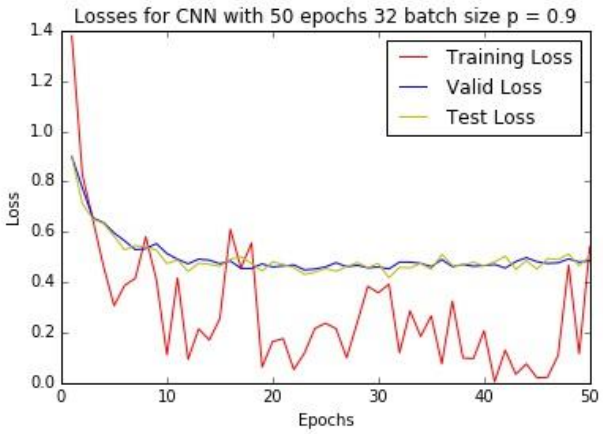


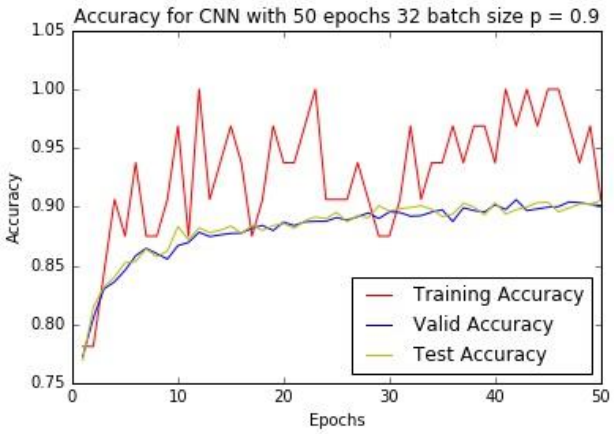


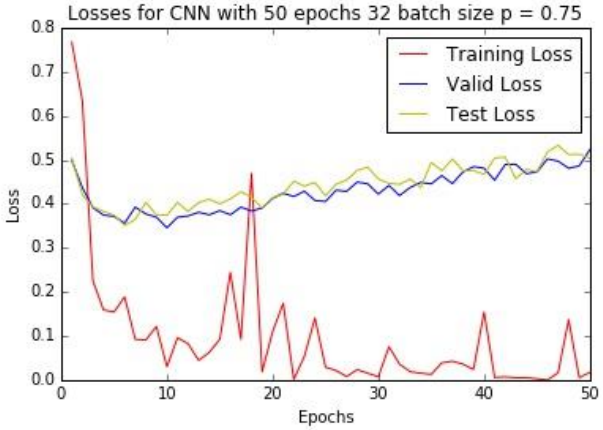
	λ = 0.01	λ=0.1	λ=0.5
Training Loss	0.2489	0.8927	1.4771
Validation Loss	0.3858	0.6288	1.2376
Test Loss	0.3952	0.6251	1.2272
Training Accuracy	0.9688	0.7500	0.7813
Validation Accuracy	0.9282	0.9032	0.8550
Test Accuracy	0.9277	0.9068	0.8634

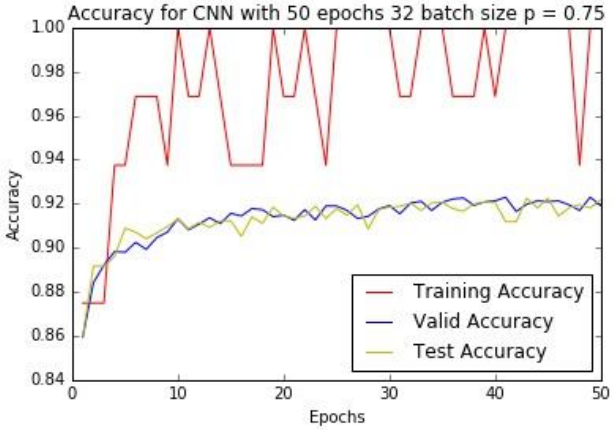
Without L2 normalization, there is overfitting in the CNN model implemented above. In section 2.2, the plots have shown that there is a huge gap between the training losses and validation losses. As the training loss gets increasingly smaller, the validation and test losses are getting larger. These are the signals of overfitting of the data. While using L2 regularization parameter of 0.01, the model works well. The are only small differences between the training loss and validation loss, and the validation and test accuracies are improving. The final accuracies are higher, and the losses are smaller when using L2 regularization of 0.01. Therefore, the overfitting is reduced. However, the model is underfitted when using parameter of 0.1 and 0.5 where the losses for validation and training data are almost the same and the training accuracies are highly fluctuated. The final accuracies are worse for parameter of 0.1 and 0.5.

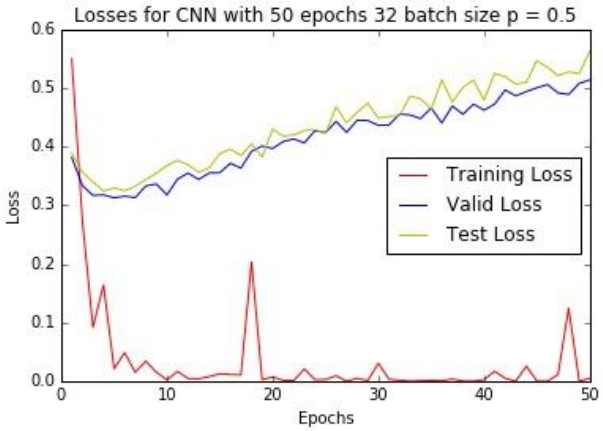
b) Dropout

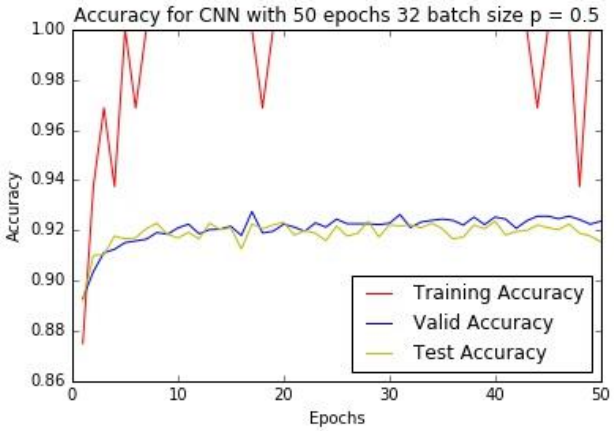












	P = 0.9	P =0.75	P = 0.5
Training Loss	0.5415	0.5415	0.016531
Validation Loss	0.4851	0.5226	0.5187901
Test Loss	0.5059	0.5031	0.53257513
Training Accuracy	0.9063	1.0	1.0
Validation Accuracy	0.9010	0.9010	0.9262
Test Accuracy	0.9053	0.9053	0.9222