NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ4042 - Neural Networks and Deep Learning

Project 2

Done By:
Janaki H Nair (U1622879J)
Dodda Sharon Olivia (U1621792K)

Part A: Object Recognition

Introduction:

We were given a dataset that contains RGB coloured images of size 32 X 32 with labels 0-9.

It deals with the design of a convolutional neural network (CNN) consisting of an input layer, 2 convolutional layers, 2 pooling layers, a fully connected layer and a softmax layer, to recognize objects.

The training dataset contains 10,000 samples and test set contains 2000 samples.

Methods:

Convolutional Layers

Convolution layer is the first layer to extract features from an input image. We perform a convolution function with the given stride and padding on the input and filters to obtain the feature maps.

Pooling Layers

The output of the convolutional layers is fed into the max pooling layer where the output dimensions are reduced by choosing the maximum element within the stride and then outputs it.

Same Padding vs Valid Padding

Padding is an operation to improve performance by keeping information at the borders. VALID padding leaves the input data as it is, without any padding. SAME padding pads the input in such a way that the produced output is of the same dimension as the input.

Dropouts

Dropout help prevent the neural network from overfitting on the training data by ignoring units during training phase of a certain set of neurons which are chosen at random.

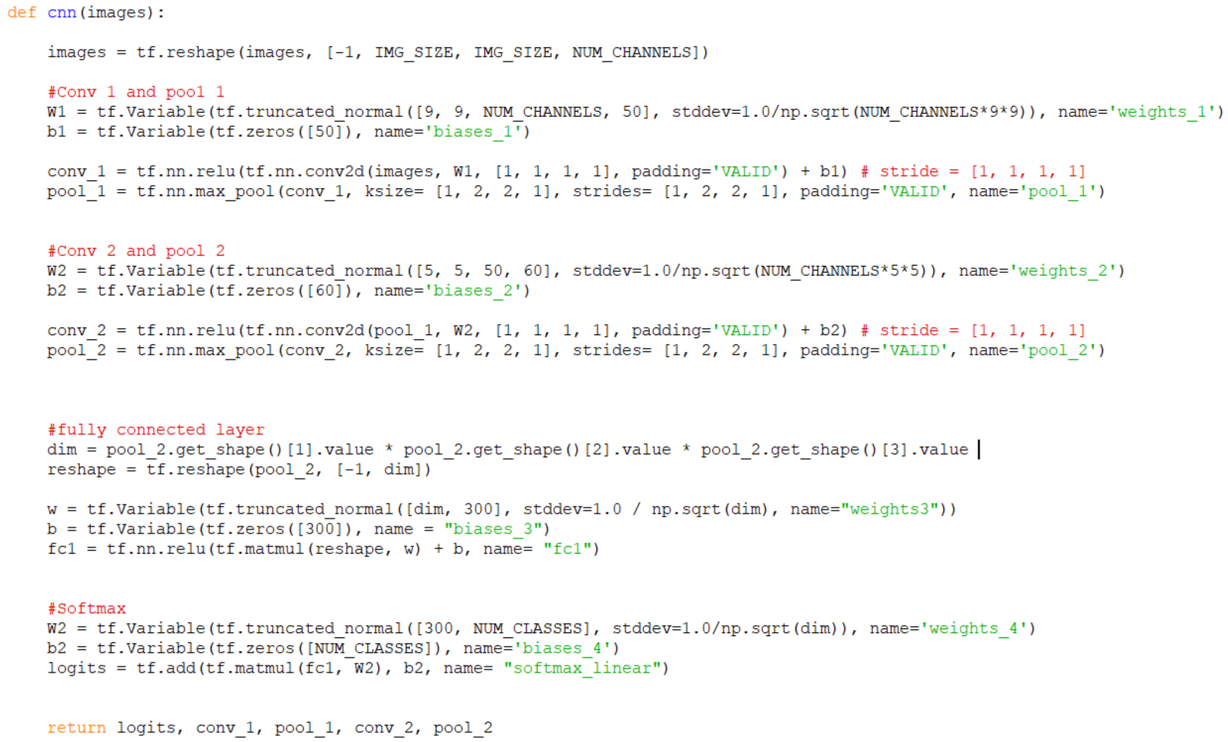
GridSearch

A model hyperparameter is a characteristic of a model that is external to the model and whose value cannot be estimated from data. The value of the hyperparameter has to be set before the learning process begins. Grid Search is used to find the optimal hyperparameters which can make predictions more accurate.

Results:

Question 1:

The first step is to preprocess the data by extracting it from the files and modifying the labels column into a one-hot matrix to be used in the model.

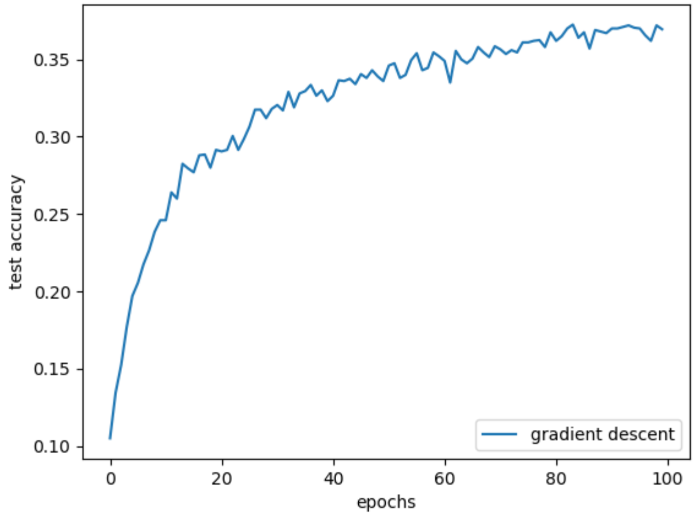


```
def cnn(images):

    images = tf.reshape(images, [-1, IMG_SIZE, IMG_SIZE, NUM_CHANNELS])

    #Conv 1 and pool 1
    W1 = tf.Variable(tf.truncated_normal([9, 9, NUM_CHANNELS, 50], stddev=1.0/np.sqrt(NUM_CHANNELS*9*9)), name='weights_1')
    b1 = tf.Variable(tf.zeros([50]), name='biases_1')

    conv_1 = tf.nn.relu(tf.nn.conv2d(images, W1, [1, 1, 1, 1], padding='VALID') + b1) # stride = [1, 1, 1, 1]
    pool_1 = tf.nn.max_pool(conv_1, ksize= [1, 2, 2, 1], strides= [1, 2, 2, 1], padding='VALID', name='pool_1')


    #Conv 2 and pool 2
    W2 = tf.Variable(tf.truncated_normal([5, 5, 50, 60], stddev=1.0/np.sqrt(NUM_CHANNELS*5*5)), name='weights_2')
    b2 = tf.Variable(tf.zeros([60]), name='biases_2')

    conv_2 = tf.nn.relu(tf.nn.conv2d(pool_1, W2, [1, 1, 1, 1], padding='VALID') + b2) # stride = [1, 1, 1, 1]
    pool_2 = tf.nn.max_pool(conv_2, ksize= [1, 2, 2, 1], strides= [1, 2, 2, 1], padding='VALID', name='pool_2')



    #fully connected layer
    dim = pool_2.get_shape()[1].value * pool_2.get_shape()[2].value * pool_2.get_shape()[3].value
    reshape = tf.reshape(pool_2, [-1, dim])

    w = tf.Variable(tf.truncated_normal([dim, 300], stddev=1.0 / np.sqrt(dim), name="weights3"))
    b = tf.Variable(tf.zeros([300]), name = "biases_3")
    fc1 = tf.nn.relu(tf.matmul(reshape, w) + b, name= "fc1")


    #Softmax
    W2 = tf.Variable(tf.truncated_normal([300, NUM_CLASSES], stddev=1.0/np.sqrt(dim)), name='weights_4')
    b2 = tf.Variable(tf.zeros([NUM_CLASSES]), name='biases_4')
    logits = tf.add(tf.matmul(fc1, W2), b2, name= "softmax_linear")


    return logits, conv_1, pool_1, conv_2, pool_2
```

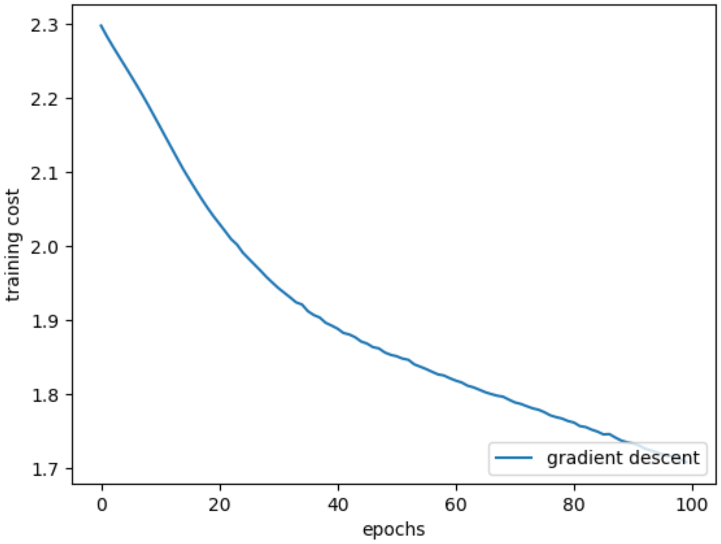
The weights of the CNN are a four-dimensional tensor where each dimension corresponds to the length and width of the convolutional layer as well as the number of input and output channels. The biases are initially set to zero.

ReLU is used as the activation function and loss is calculated based on the softmax entropy. The model was trained using 100 epochs and a mini-batch size of 128.

Part (a):
The graph below shows the test accuracy vs epoch. It converges at about 0.3695.



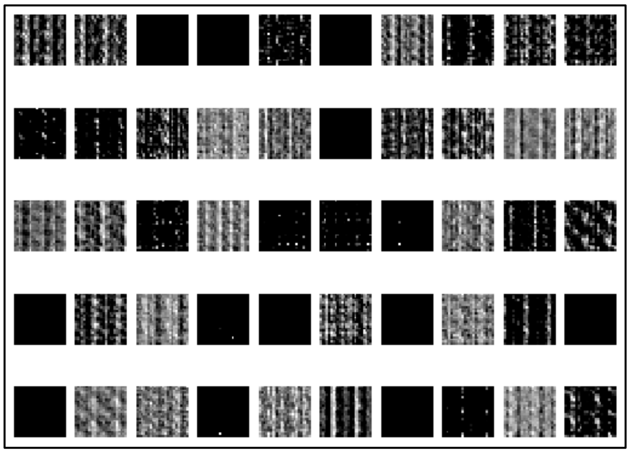
The graph below shows the training cost vs epochs. We obtained a final training error of about 1.70875.



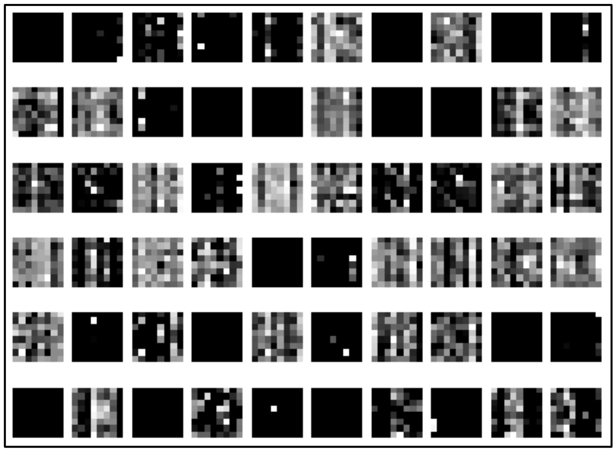
Part (b):

Pattern 1

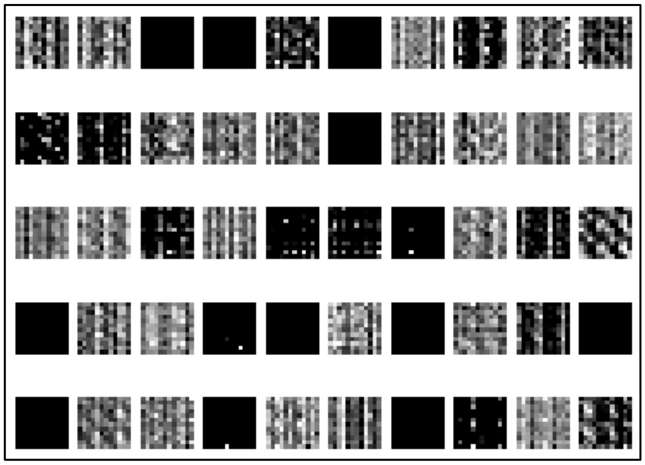
Feature maps convolution layer 1:



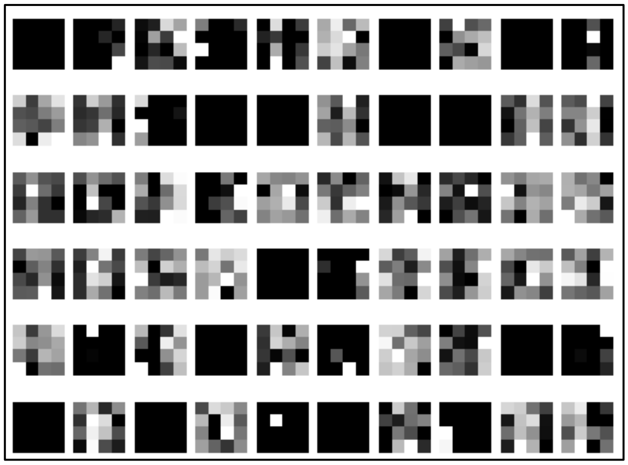
Feature maps at convolution layer 2:



Feature maps at pooling layer 1:

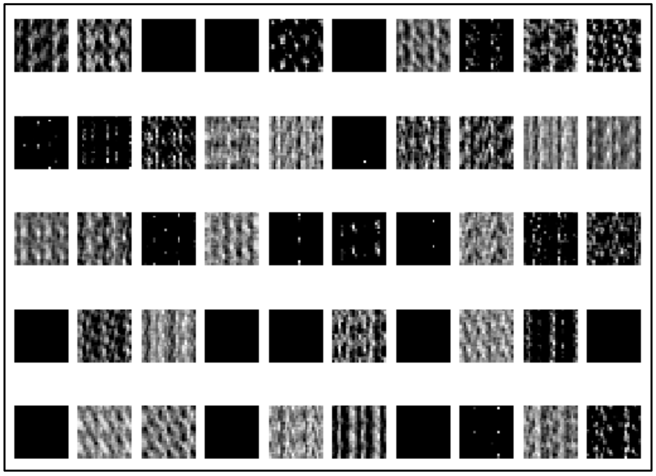


Feature maps at pooling layer 2:

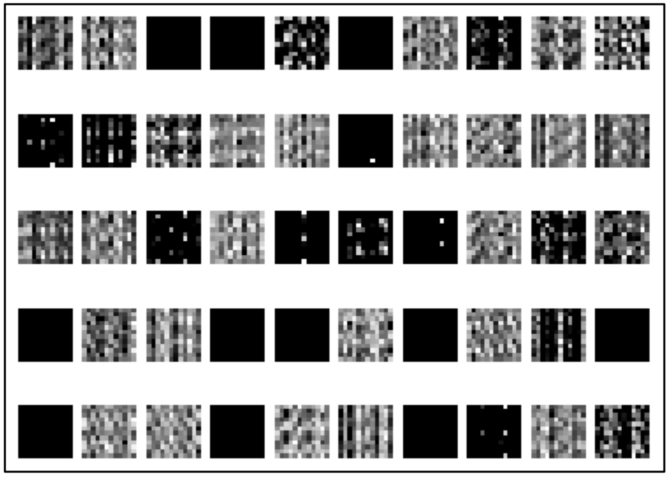


Pattern 2

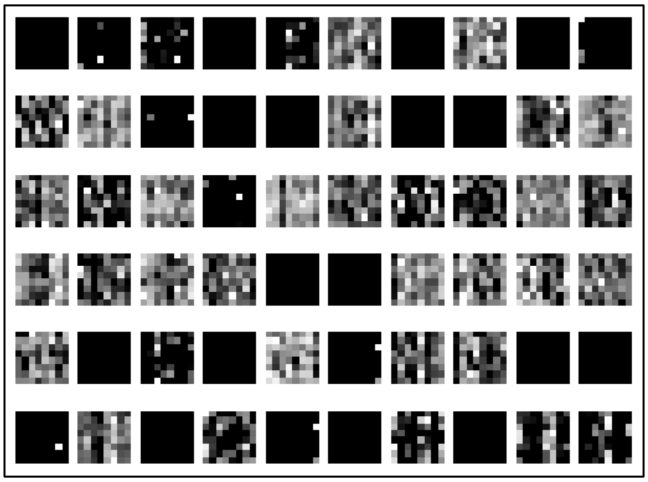
Feature maps at convolution layer 1:



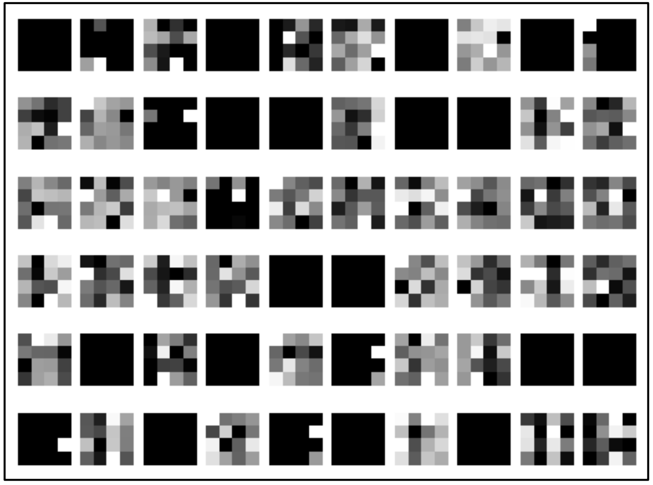
Feature maps at convolution layer 2:



Feature maps at pooling layer 1:

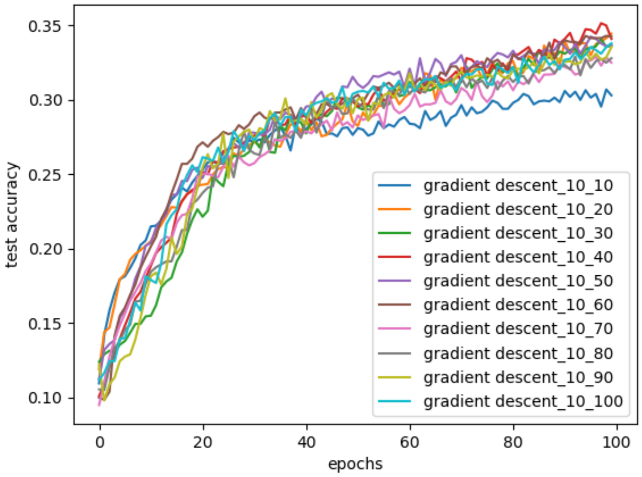


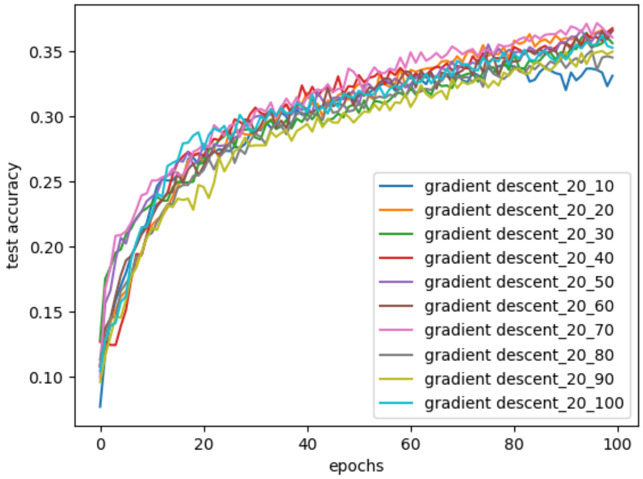
Feature maps at pooling layer 2:

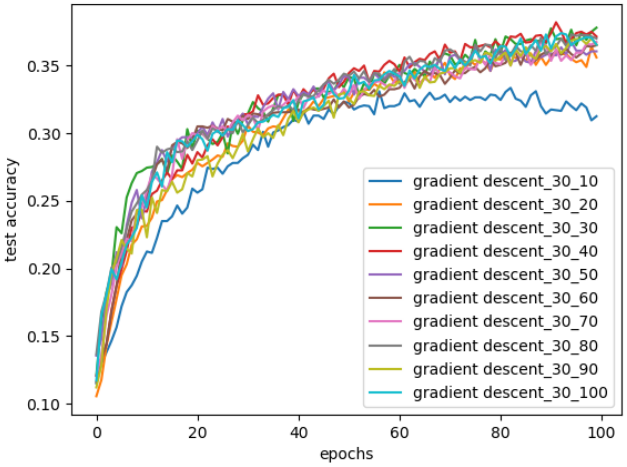


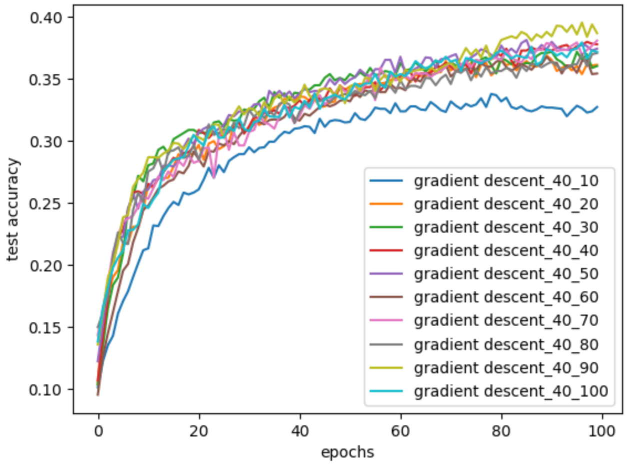
Question 2:

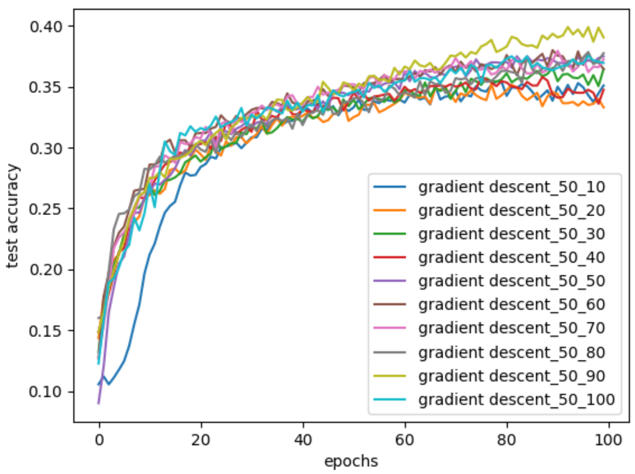
In this question, we use the concept of grid search to find the optimal number of feature maps at the convolutional layers based on the test accuracies. The search was performed from 10 to 70 at each convolutional layer with an interval of 10. The test accuracy graphs were plotted with the label as '*gradient descent_NumofFeatureMapsAtLayer1_NumofFeatureMapsAtLayer2*'. We have decided to break up the graphs into 10 different plots as it would not be too cluttered and it would be easier to view the lines. With the results obtained, as seen from the graphs below, it was found that the optimal number of feature maps at convolutional layer 1 is 70 and the optimal number of feature maps at convolutional layer 2 is 90.

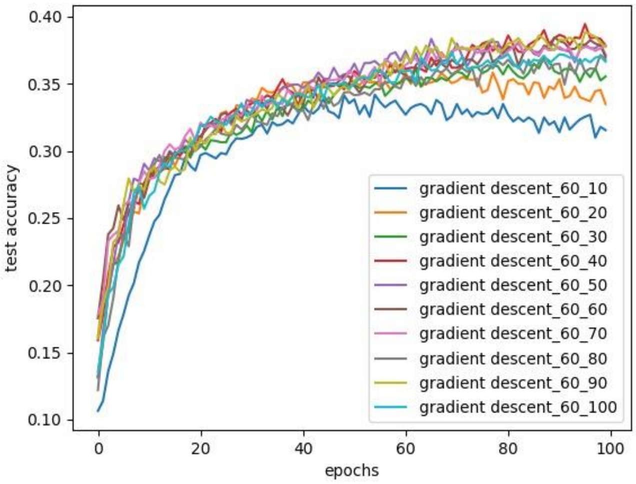


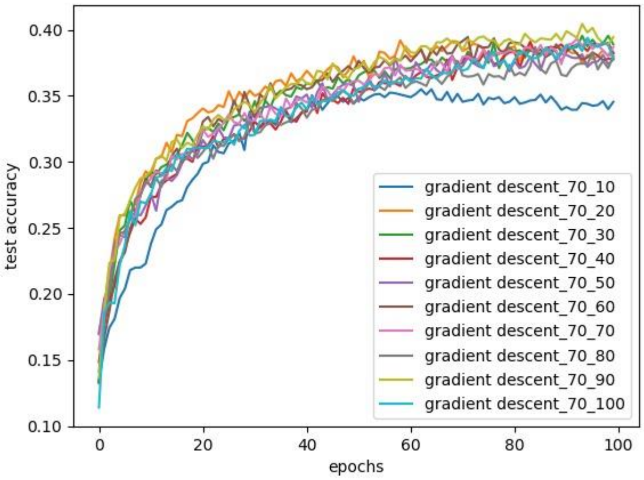







test accuracy
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
20
40
60
80
100
epochs
gradient descent_50_10
gradient descent_50_20
gradient descent_50_30
gradient descent_50_40
gradient descent_50_50
gradient descent_50_60
gradient descent_50_70
gradient descent_50_80
gradient descent_50_90
gradient descent_50_100


test accuracy
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
20
40
60
80
100
epochs
gradient descent_60_10
gradient descent_60_20
gradient descent_60_30
gradient descent_60_40
gradient descent_60_50
gradient descent_60_60
gradient descent_60_70
gradient descent_60_80
gradient descent_60_90
gradient descent_60_100

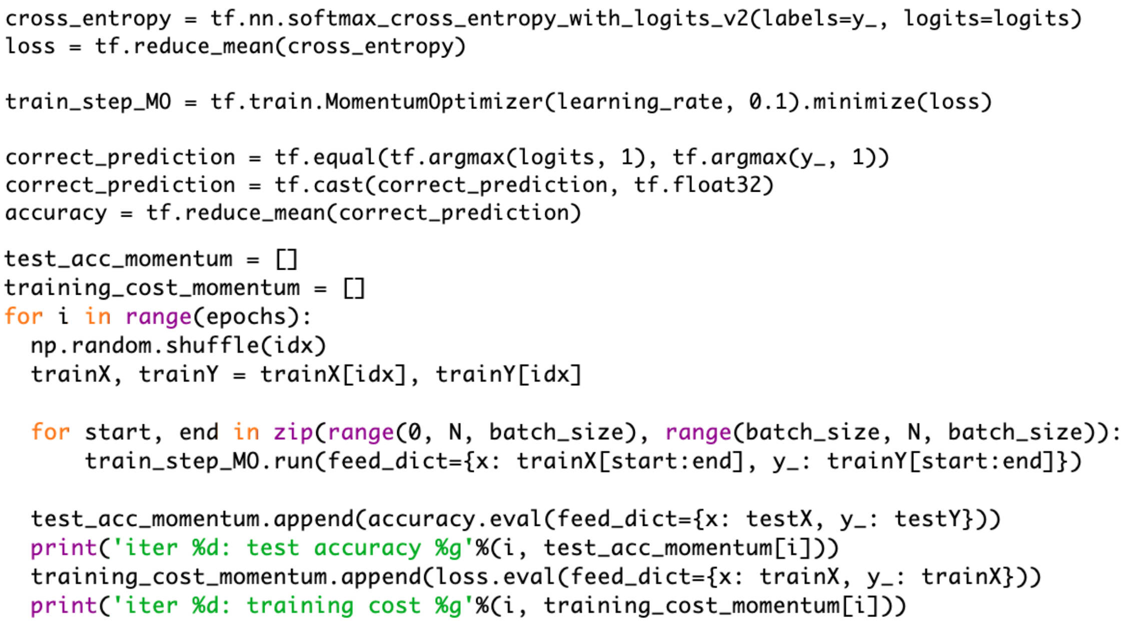

test accuracy
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
20
40
60
80
100
epochs
gradient descent_70_10
gradient descent_70_20
gradient descent_70_30
gradient descent_70_40
gradient descent_70_50
gradient descent_70_60
gradient descent_70_70
gradient descent_70_80
gradient descent_70_90
gradient descent_70_100

Question 3:

We now have manually set the number of feature maps of the first convolutional layer to 70 and of the second convolutional layer to 90.

Part (a):

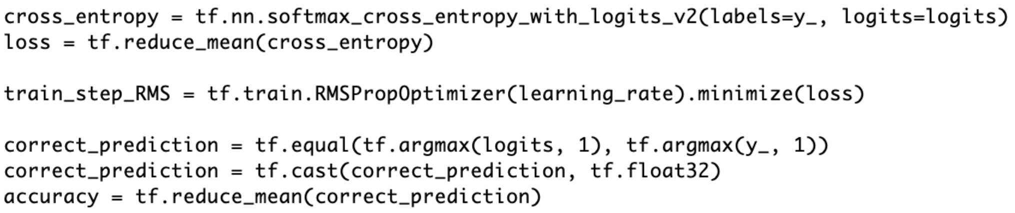
The model is trained with Momentum Optimizer with the optimum parameter set to 0.1. The plots below show the training error and test accuracies against the epochs. Below is our code snippet for this part:



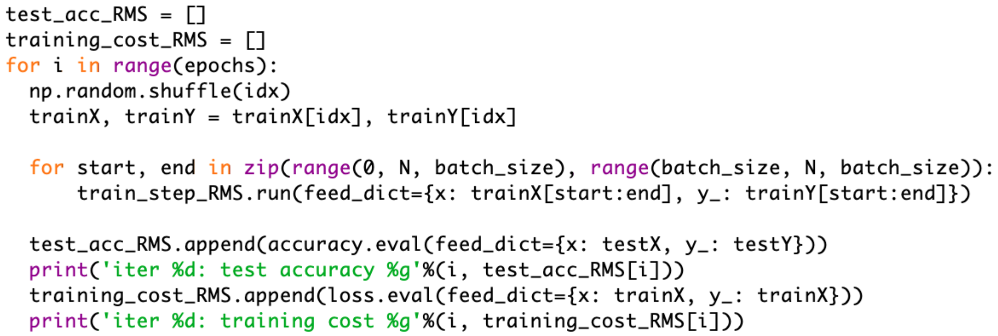
```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=logits)
loss = tf.reduce_mean(cross_entropy)

train_step_MO = tf.train.MomentumOptimizer(learning_rate, 0.1).minimize(loss)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y_, 1))
correct_prediction = tf.cast(correct_prediction, tf.float32)
accuracy = tf.reduce_mean(correct_prediction)

test_acc_momentum = []
training_cost_momentum = []
for i in range(epochs):
  np.random.shuffle(idx)
  trainX, trainY = trainX[idx], trainY[idx]

  for start, end in zip(range(0, N, batch_size), range(batch_size, N, batch_size)):
      train_step_MO.run(feed_dict={x: trainX[start:end], y_: trainY[start:end]})

  test_acc_momentum.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
  print('iter %d: test accuracy %g'%(i, test_acc_momentum[i]))
  training_cost_momentum.append(loss.eval(feed_dict={x: trainX, y_: trainX}))
  print('iter %d: training cost %g'%(i, training_cost_momentum[i]))
```

Part (b):

Next, the model is trained using RMSProp Optimizer with the above set hyperparameters. RMSProp restricts the oscillations in the vertical direction to prevent weights from blowing up. Below is our code snippet for this part:



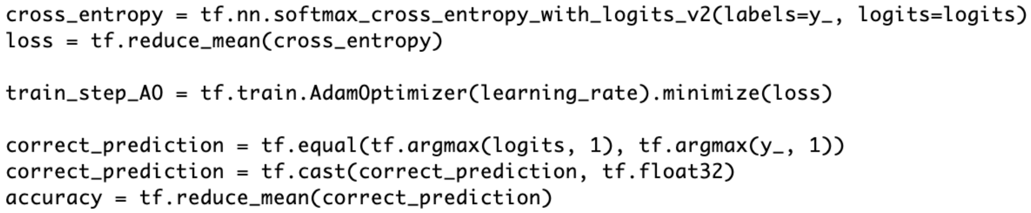
```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=logits)
loss = tf.reduce_mean(cross_entropy)

train_step_RMS = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y_, 1))
correct_prediction = tf.cast(correct_prediction, tf.float32)
accuracy = tf.reduce_mean(correct_prediction)
```



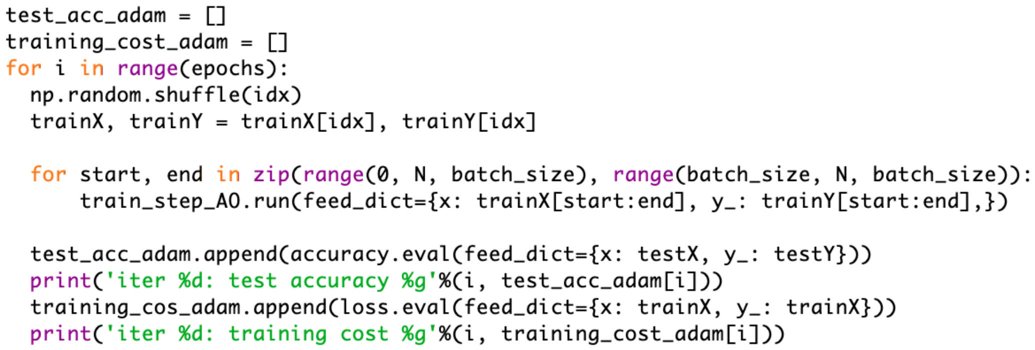
```
test_acc_RMS = []
training_cost_RMS = []
for i in range(epochs):
  np.random.shuffle(idx)
  trainX, trainY = trainX[idx], trainY[idx]

  for start, end in zip(range(0, N, batch_size), range(batch_size, N, batch_size)):
      train_step_RMS.run(feed_dict={x: trainX[start:end], y_: trainY[start:end]})

  test_acc_RMS.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
  print('iter %d: test accuracy %g'%(i, test_acc_RMS[i]))
  training_cost_RMS.append(loss.eval(feed_dict={x: trainX, y_: trainX}))
  print('iter %d: training cost %g'%(i, training_cost_RMS[i]))
```

Part (c):

The model is now trained using the Adam Optimizer. Adam Optimizer is a combination of the momentum and the RMSProp Optimizer and is said to be more stable than them as it does not suffer any major decreases in accuracy. Below is our code snippet for this part:



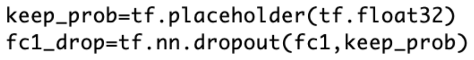
```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=logits)
loss = tf.reduce_mean(cross_entropy)

train_step_AO = tf.train.AdamOptimizer(learning_rate).minimize(loss)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y_, 1))
correct_prediction = tf.cast(correct_prediction, tf.float32)
accuracy = tf.reduce_mean(correct_prediction)
```



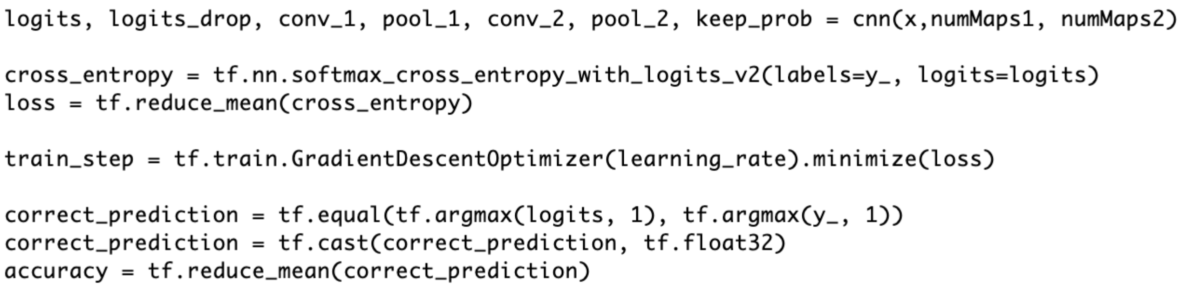
```
test_acc_adam = []
training_cost_adam = []
for i in range(epochs):
  np.random.shuffle(idx)
  trainX, trainY = trainX[idx], trainY[idx]

  for start, end in zip(range(0, N, batch_size), range(batch_size, N, batch_size)):
      train_step_AO.run(feed_dict={x: trainX[start:end], y_: trainY[start:end],})

  test_acc_adam.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
  print('iter %d: test accuracy %g'%(i, test_acc_adam[i]))
  training_cos_adam.append(loss.eval(feed_dict={x: trainX, y_: trainX}))
  print('iter %d: training cost %g'%(i, training_cost_adam[i]))
```

Part (d):

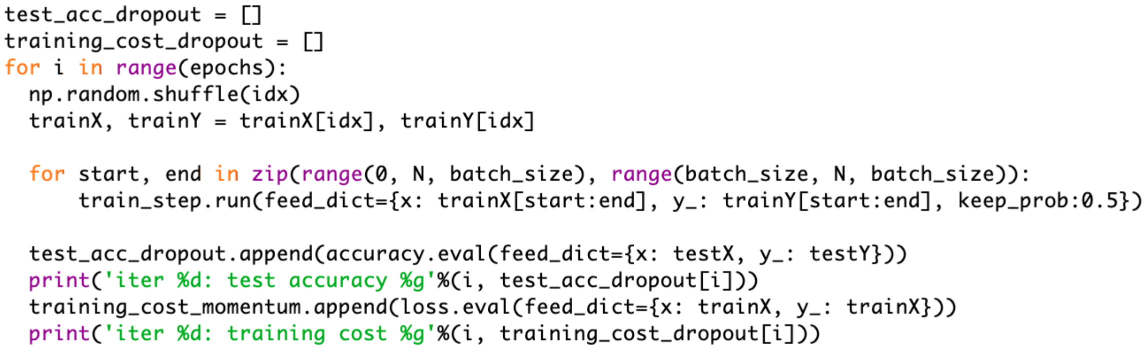
Lastly, the model was trained using dropouts at the convolutional layers. We created a *keep_prob* tensor and added dropout to the fully connected layer. The keep-probability used was 0.5. Below is our code snippet for this part:



```
keep_prob=tf.placeholder(tf.float32)
fc1_drop=tf.nn.dropout(fc1,keep_prob)
```

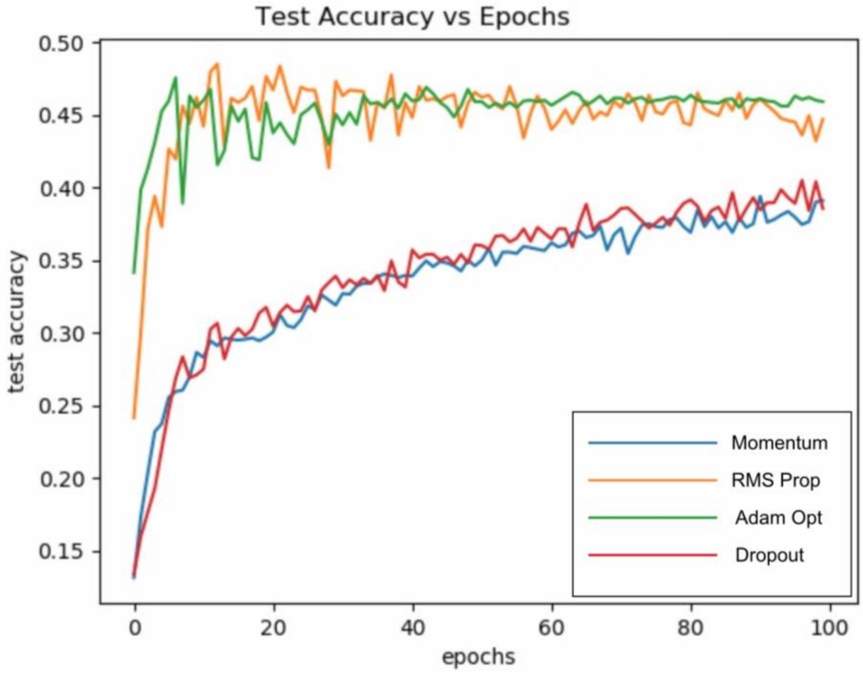


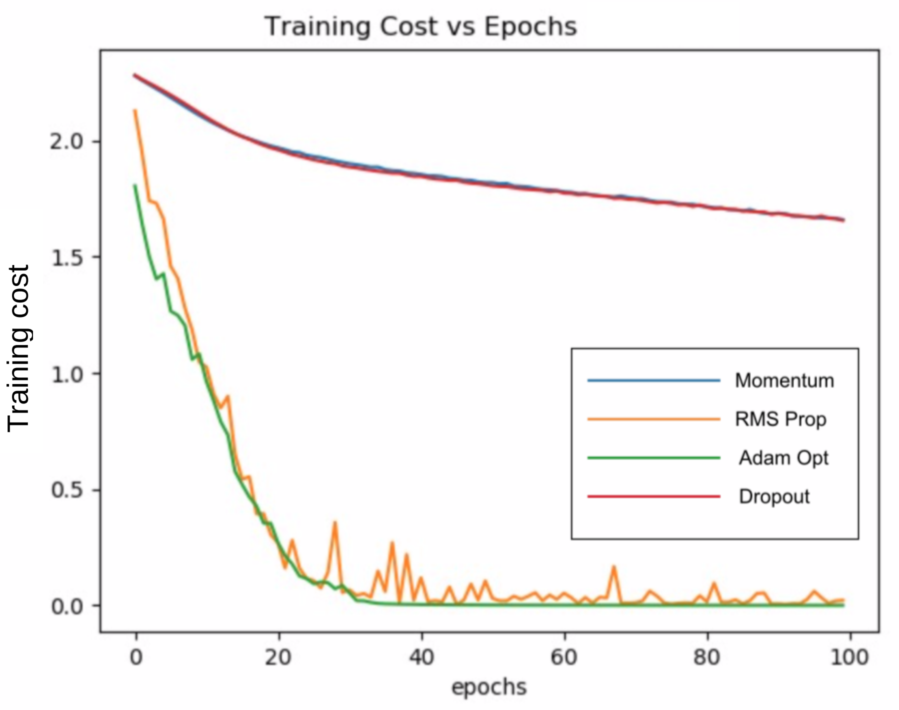
```
logits, logits_drop, conv_1, pool_1, conv_2, pool_2, keep_prob = cnn(x,numMaps1, numMaps2)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=logits)
loss = tf.reduce_mean(cross_entropy)

train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y_, 1))
correct_prediction = tf.cast(correct_prediction, tf.float32)
accuracy = tf.reduce_mean(correct_prediction)
```



```
test_acc_dropout = []
training_cost_dropout = []
for i in range(epochs):
  np.random.shuffle(idx)
  trainX, trainY = trainX[idx], trainY[idx]

  for start, end in zip(range(0, N, batch_size), range(batch_size, N, batch_size)):
      train_step.run(feed_dict={x: trainX[start:end], y_: trainY[start:end], keep_prob:0.5})

  test_acc_dropout.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
  print('iter %d: test accuracy %g'%(i, test_acc_dropout[i]))
  training_cost_momentum.append(loss.eval(feed_dict={x: trainX, y_: trainX}))
  print('iter %d: training cost %g'%(i, training_cost_dropout[i]))
```

Below are the Test Accuracy and Training Cost graphs respectively for all the 4 different optimizers:





Question 4:

Dropout and momentum optimizers have similar test accuracies, while RMS Prop and Adam optimizers have similar test accuracies. The test accuracies of dropout and momentum optimizers have a significantly lower test accuracy compared to RMS Prop and Adam optimizers.

Dropout and momentum optimizers also have similar training costs, while RMS Prop and Adam optimizers have similar training costs. The test accuracies of dropout and momentum optimizers have a significantly higher training cost compared to RMS Prop and Adam optimizers.

Hence, we conclude that RMS Prop and Adam optimizers are better than dropout and momentum.

As seen from the graphs, Adam Optimizer is the better than RMS Prop as it has the highest accuracy (~ 0.45) and lowest training cost. It converges at about 40 epochs.

Greater accuracy may be achieved by increasing the number of epochs.

Part B: Text Classification

Introduction:

We were given a dataset that contains first paragraphs collected from Wikipage entries and the corresponding labels about their category.

Part B dealt with the implementation of CNNs and RNNs at both the character and word levels for text classification.

The training set contains 5600 samples whereas test set contains 700 entries. The maximum length of the document was restricted to characters/words input of 100.

Methods:

Embedding Layer

Embedding is mainly applied to the input of the word classifiers before feeding into the RNN. It is used to create word vectors for incoming word inputs. In an embedding, words are represented by dense vectors where a vector represents projection of words into a continuous vector space.

Gradient Clipping:

It is very important in RNNs to prevent the gradients from either vanishing(diminishingly low values) or exploding. It clips the gradient between 2 numbers to regularize the weights.

Long Short-Term Memory (LSTM) network:

Long short-term memory networks are an altered version of RNN. They are capable of learning long-term dependencies as they can remember previous data easily, including the gradients. An LSTM network trains a model using back propagation and is suitable for predicting a time series, for example text classification.

Vanilla RNN:

Vanilla RNN is a feedforward network where the connections between layers do not form cycles.

Gated Recurrent Unit (GRU):

GRU is a gating mechanism in RNN. It is an LSTM with an additional forget gate.

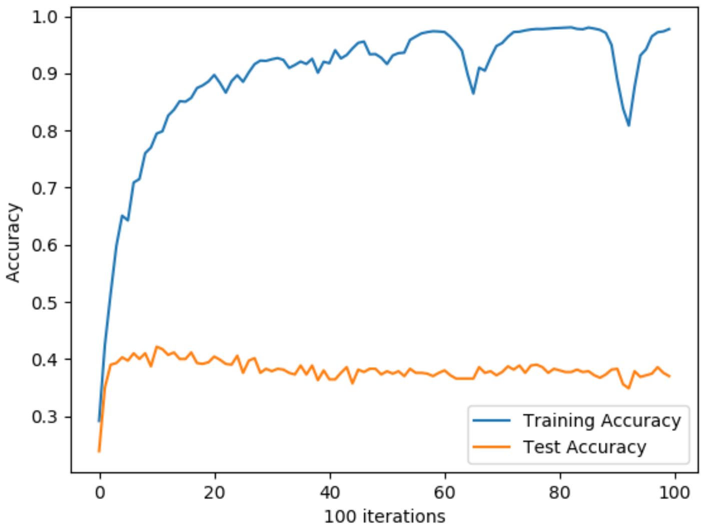
Results:

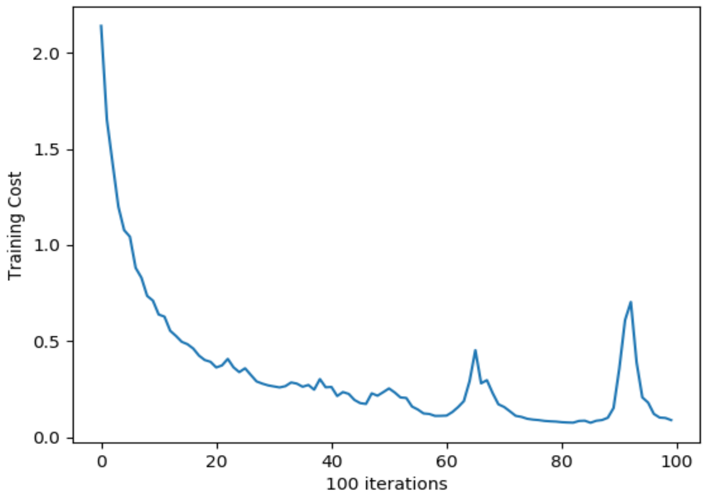
Question 1:

In this problem, we have designed a character CNN classifier consisting of two convolutional layers as well as two pooling layers.

A batch-size was 128 was used along with learning rate of 0.01. The following graphs were obtained after running for 100 epochs.

```
def char_cnn_model(x):

  input_layer = tf.reshape(
      tf.one_hot(x, 256), [-1, MAX_DOCUMENT_LENGTH, 256, 1])

  with tf.variable_scope('CNN_Layer1'):
    conv1 = tf.layers.conv2d(
        input_layer,
        filters=N_FILTERS,
        kernel_size=FILTER_SHAPE1,
        padding='VALID',
        activation=tf.nn.relu)
    pool1 = tf.layers.max_pooling2d(
        conv1,
        pool_size=POOLING_WINDOW,
        strides=POOLING_STRIDE,
        padding='SAME')
    conv2 = tf.layers.conv2d(
        pool1,
        filters=N_FILTERS,
        kernel_size=FILTER_SHAPE2,
        padding='VALID',
        activation=tf.nn.relu,
        name="conv2")
    pool2 = tf.layers.max_pooling2d(
        conv2,
        pool_size=POOLING_WINDOW,
        strides=POOLING_STRIDE,
        padding='SAME',
        name="pool2")


    pool2 = tf.squeeze(tf.reduce_max(pool2, 1), squeeze_dims=[1])

    logits = tf.layers.dense(pool2, MAX_LABEL, activation=None,name="logits")
```



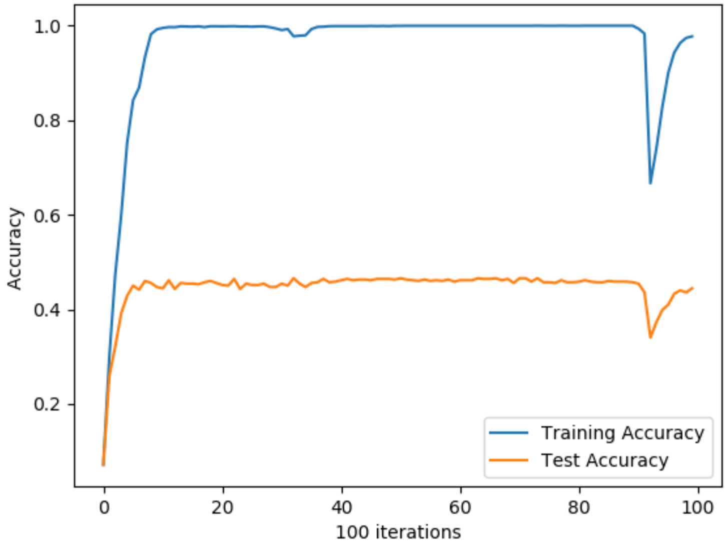


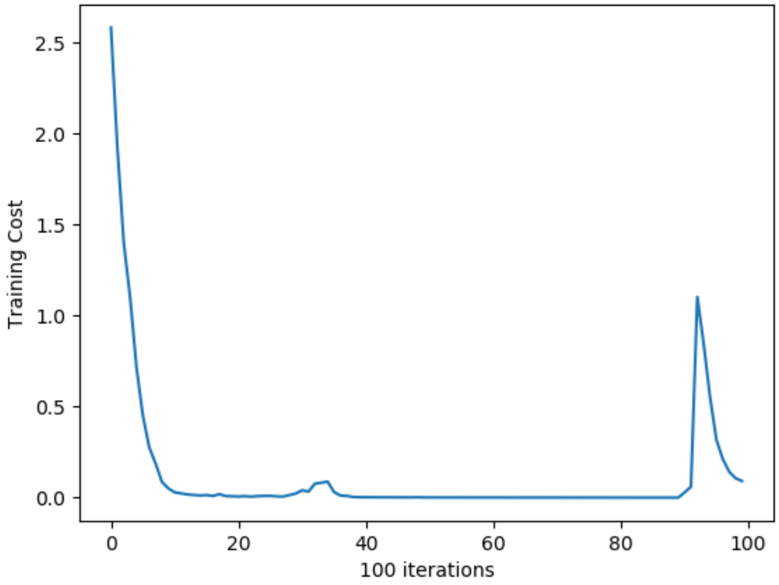
We notice there are spikes in the training cost as the number of iterations increase. This is apparently due to the unavoidable consequence of the Mini_Batch Gradient Descent in the Adam Optimizer and our code uses the Adam Optimizer. Our test accuracy is around 40.5%

Question 2:

This question implements a word CNN classifier with the use of embedding layer of size 20. The CNN has 2 convolutional and 2 pooling layers.

```
def word_cnn_model(x):

  word_vectors = tf.contrib.layers.embed_sequence(x, vocab_size=n_words, embed_dim=EMBEDDING_SIZE)

  input_layer = tf.reshape(word_vectors, [-1, MAX_DOCUMENT_LENGTH, EMBEDDING_SIZE, 1])

  with tf.variable_scope('CNN_Layer1'):
    conv1 = tf.layers.conv2d(
        input_layer,
        filters=N_FILTERS,
        kernel_size=FILTER_SHAPE1,
        padding='VALID',
        activation=tf.nn.relu)

    pool1 = tf.layers.max_pooling2d(
        conv1,
        pool_size=POOLING_WINDOW,
        strides=POOLING_STRIDE,
        padding='SAME')

    conv2 = tf.layers.conv2d(
        pool1,
        filters=N_FILTERS,
        kernel_size=FILTER_SHAPE2,
        padding='VALID',
        activation=tf.nn.relu,
        name="conv2")

    pool2 = tf.layers.max_pooling2d(
        conv2,
        pool_size=POOLING_WINDOW,
        strides=POOLING_STRIDE,
        padding='SAME',
        name="pool2")


    pool2 = tf.squeeze(tf.reduce_max(pool2, 1), squeeze_dims=[1])

    logits = tf.layers.dense(pool2, MAX_LABEL, activation=None,name="logits")
```

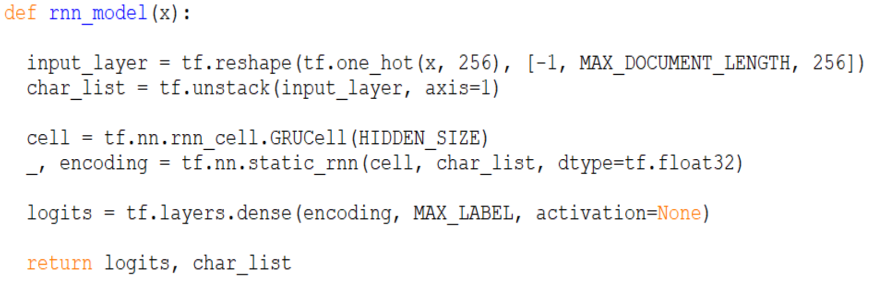




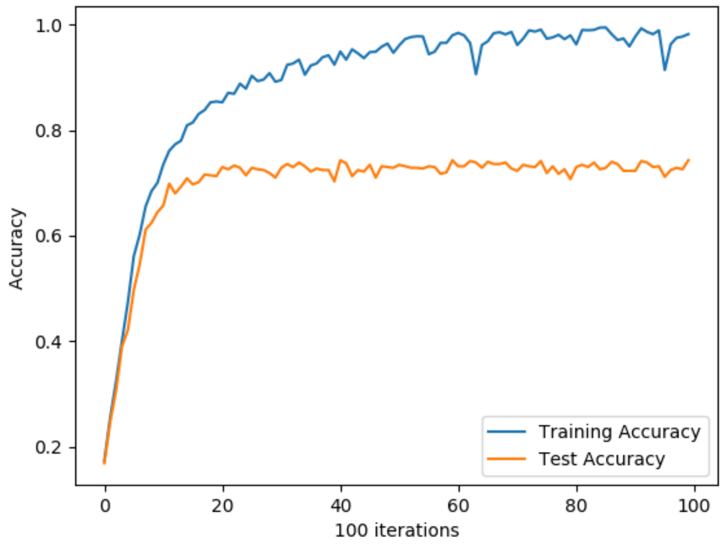
The test accuracy approaches around 47%.

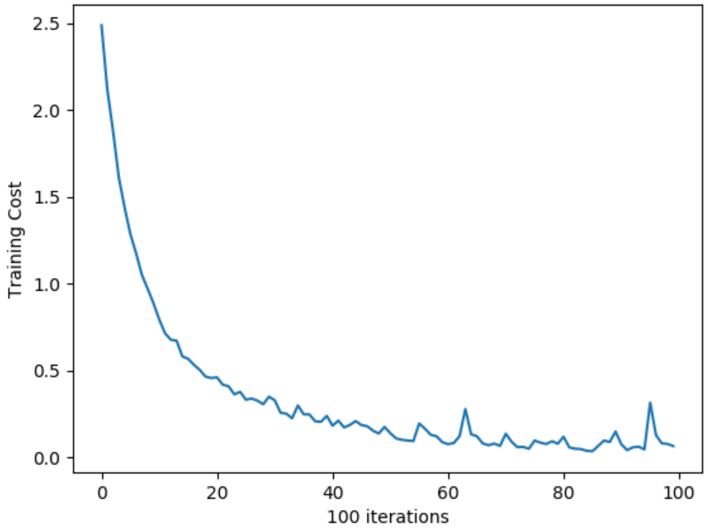
Question 3:

For this question, we designed a character RNN classifier. The RNN has a GRU(Gated Recurrent Unit) layer and a hidden layer of size 20.



```
def rnn_model(x):

  input_layer = tf.reshape(tf.one_hot(x, 256), [-1, MAX_DOCUMENT_LENGTH, 256])
  char_list = tf.unstack(input_layer, axis=1)

  cell = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
  _, encoding = tf.nn.static_rnn(cell, char_list, dtype=tf.float32)

  logits = tf.layers.dense(encoding, MAX_LABEL, activation=None)

  return logits, char_list
```



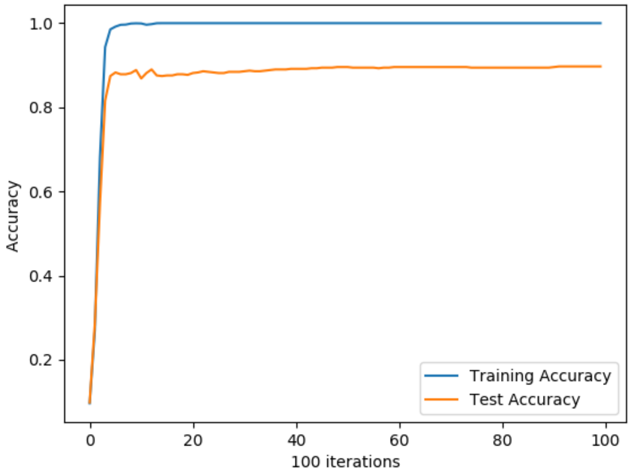


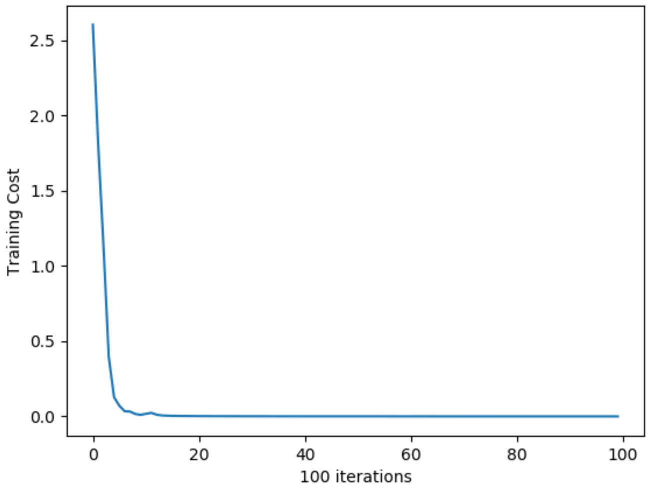
The test accuracy approaches around 75%. From this result, we can deduce that the accuracy for a RNN classifier is more as compared to CNN for character.

Question 4:

This question implements a word RNN classifier. The RNN again consists of a GRU layer and a hidden layer of size 20. The inputs are fed through an embedding layer of size 20 first before feeding it into the RNN.

```
def rnn_model(x):

  word_vectors = tf.contrib.layers.embed_sequence(
      x, vocab_size=n_words, embed_dim=EMBEDDING_SIZE)

  word_list = tf.unstack(word_vectors, axis=1)

  cell = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
  _, encoding = tf.nn.static_rnn(cell, word_list, dtype=tf.float32)

  logits = tf.layers.dense(encoding, MAX_LABEL, activation=None)

  return logits, word_list
```





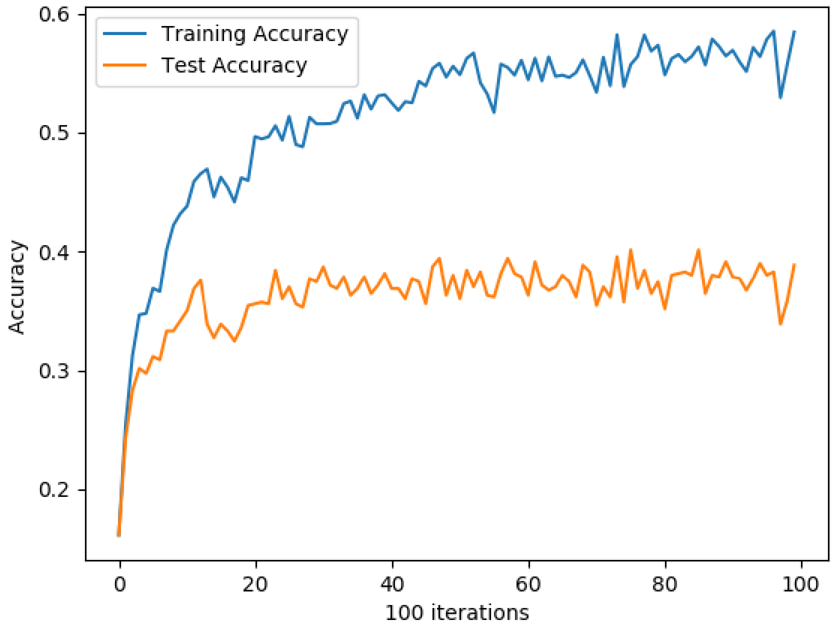
The test accuracy obtained from the results is around 89%.

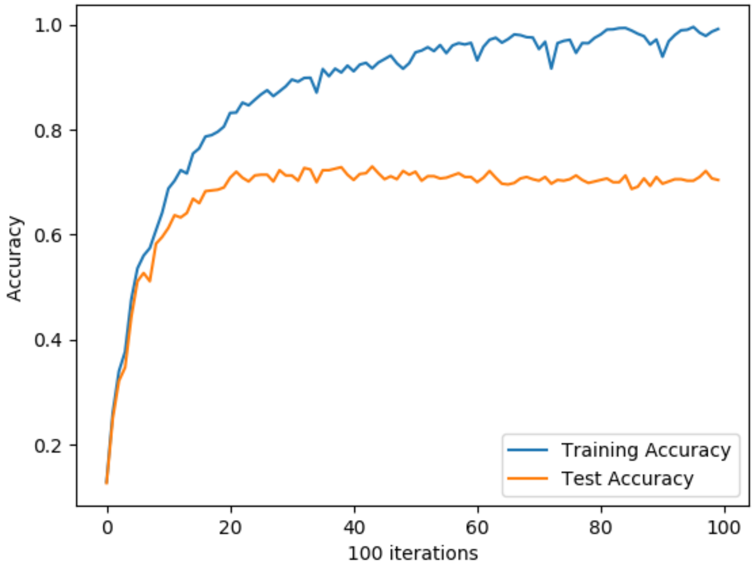
Question 5:

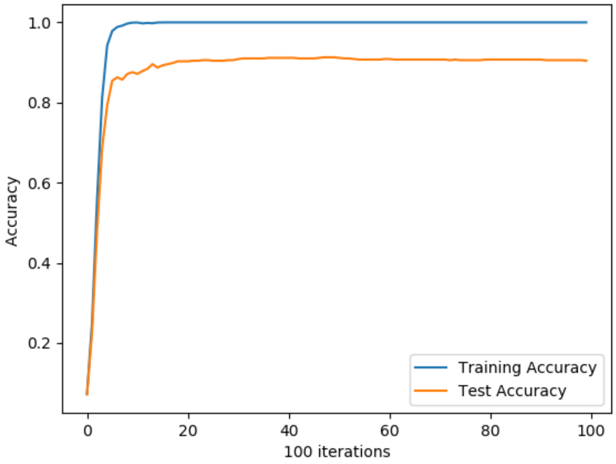
Model	Running time	Test Accuracy (%)
Char CNN	0.001048	40.5
Word CNN	0.0142	47
Char RNN	0.0645	75
Word RNN	0.0165	89

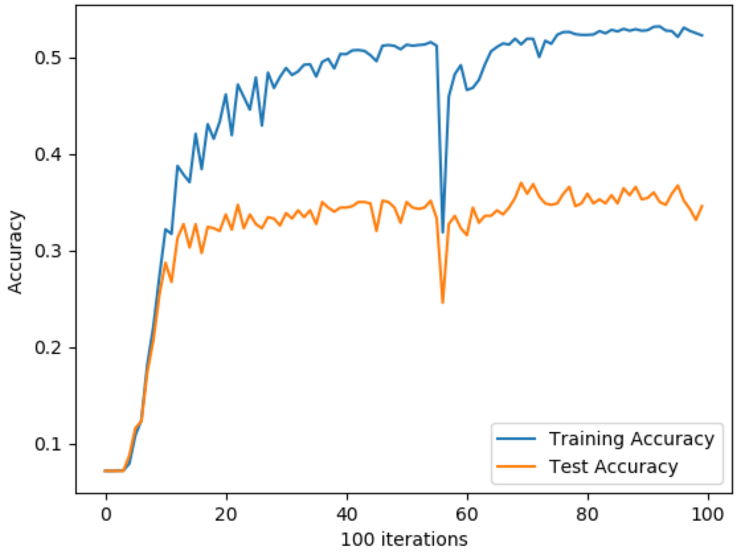
From the above graph it is seen that, the RNN classifiers provide better accuracies for text classification. However, they take longer time to train as compared to the CNN classifiers. Therefore, if the classification problem requires higher accuracy, then RNN models can be used and if the problem needs to be run in a shorter time, CNN models can be used.

Comparison using dropouts




1.0
0.8
0.6
0.4
0.2
Accuracy
0
20
40
60
80
100
100 iterations
Training Accuracy
Test Accuracy


1.0
0.8
0.6
0.4
0.2
Accuracy
0
20
40
60
80
100
100 iterations
Training Accuracy
Test Accuracy

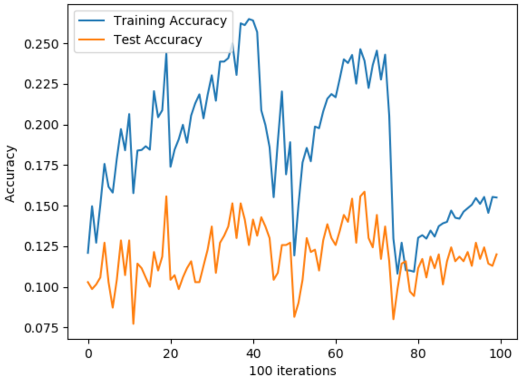


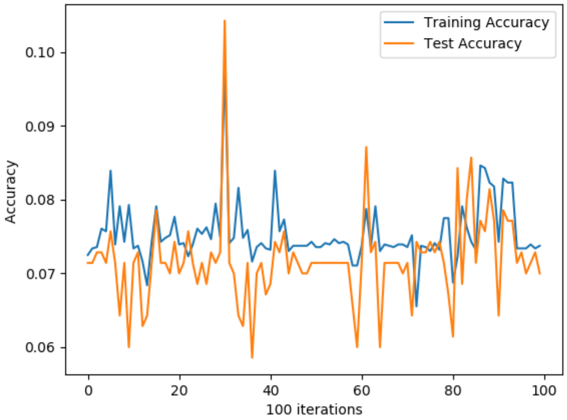
These graphs show more spikes as compared to the graphs without dropouts. This may suggest that this fits the model better and the graphs without dropout may show signs of overfitting.

Question 6:

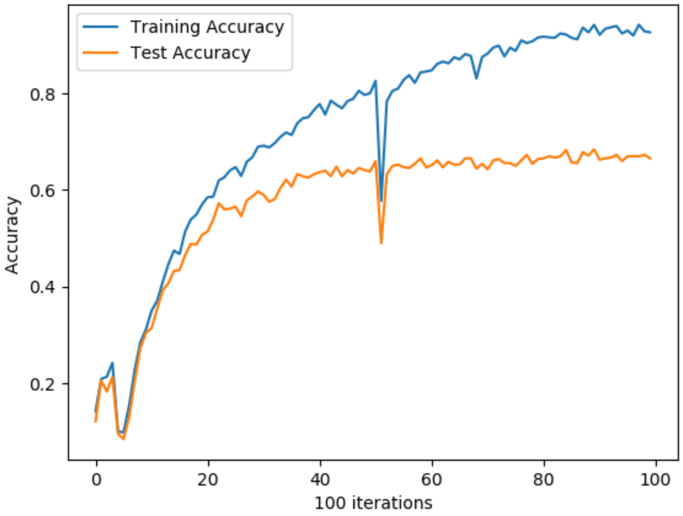
Part (a)

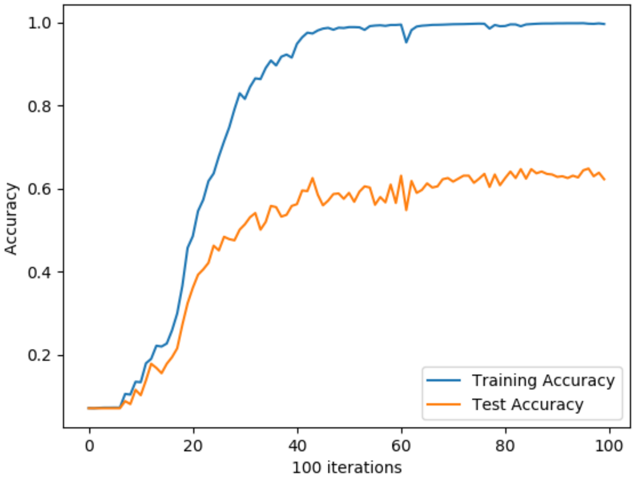
In the RNN networks implemented above, we replaced the GRU layer with Vanilla RNN layer. The findings are as below:





The results given above as seen is quite unstable. This may be due to the issue of exploding gradients.

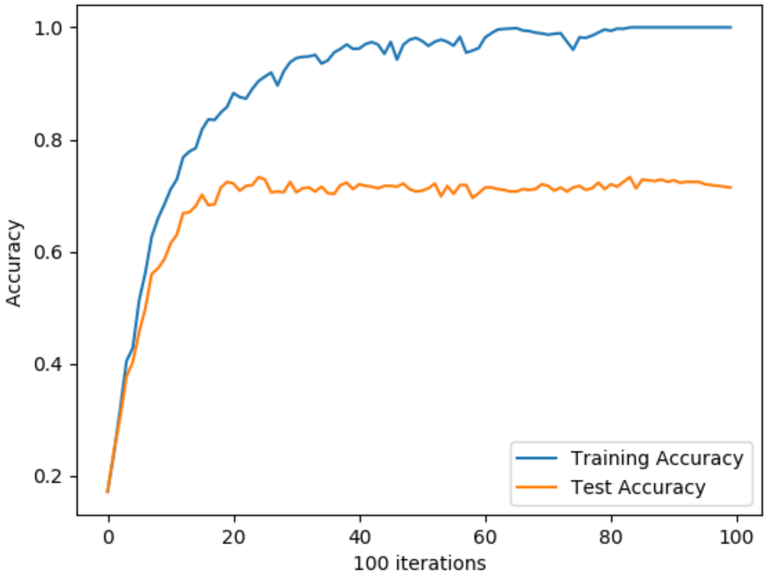


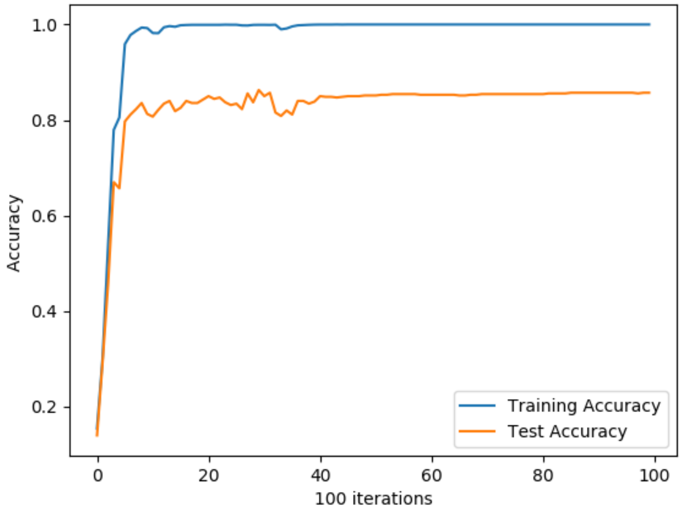


For the LSTM model, the results seem to be better as compared to the previous which we got from vanilla RNN. It does show that as the training accuracy gets higher, the test accuracy also increases.

Part (b)

We now increased the number of RNN layers to two in the RNN networks implemented initially.

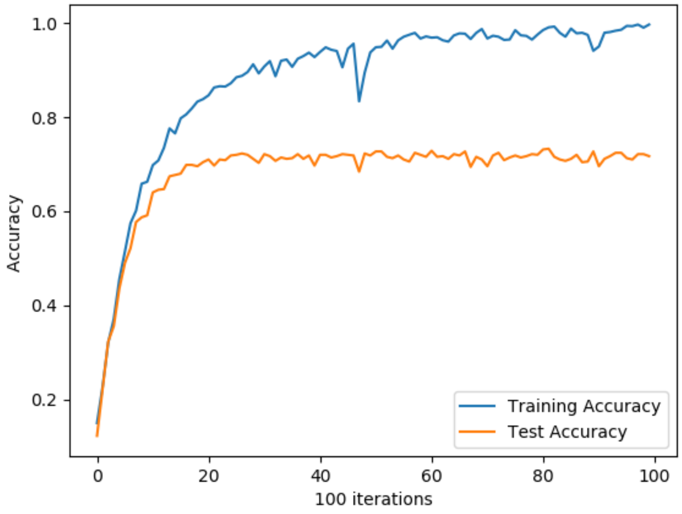


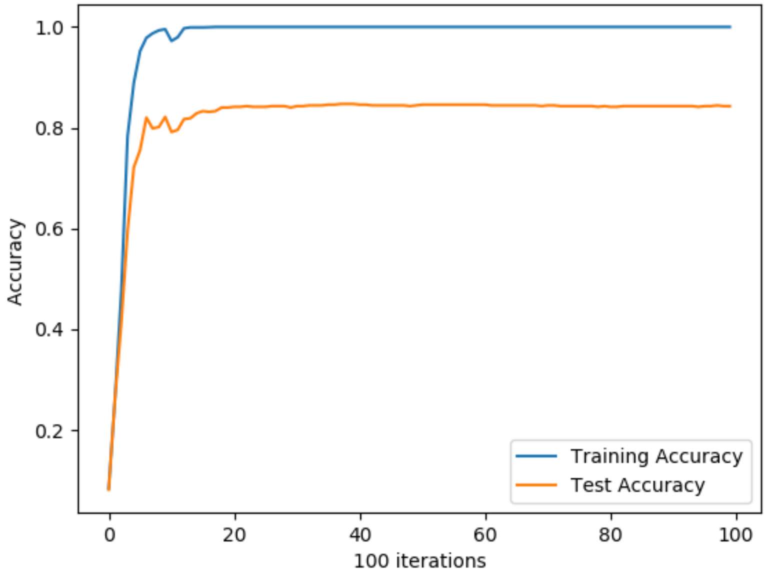


These plots show much better results as compared to the above LSTM and Vanilla plots. Even the training and testing accuracies are better shown and produced without much variations.

Part (c)

In this part, we added gradient clipping to the RNN networks with a threshold of 2.





Adding gradient clipping also shows more accurate results when compared to the RNN networks with vanilla layer or the LSTM layer.

Models with parameter	Running time	Test Accuracy
Char Vanilla RNN layer	0.038	0.12
Word Vanilla RNN layer	0.012	0.08
Char LSTM layer	0.094	0.67
Word LSTM layer	0.016	0.64
Char RNN Multi-layer	0.0017	0.72
Word RNN Multi-layer	0.034	0.86
Char RNN with gradient clipping	0.0010	0.74
Word RNN with gradient clipping	0.031	0.85

The above graph summarizes the test accuracies and the running times of the various parameters. Multi-layered models and models with gradient clipping have higher accuracies.