NANYANG TECHNOLOGICAL UNIVERSITY

NEURAL NETWORKS AND DEEP LEARNING

Assignment 2

# YE HTET MYAT- U1621116b#
YEHT0005@e.ntu.edu.sg

Contents

Part A: Object Recognition

Introduction

The dataset is subsampled from the CIFAR-10 dataset which originally has 50,000 training images and 10,000 testing images. In this subsample there are 10,000 training images and testing is done on 2,00 testing images. Each image is RGB colored and is of size 32x32 and they area labelled from 0 to 9. The labels 0 to 9 represents objects which are airplane, automobile, bird, cat, deer, dog, frog, horses, ship and truck and in that order. The classes are also mutually exclusive.

Convolutional neural network consisting of:

• An Input layer of 3x32x32 dimensions

• A convolution layer $C1$ of 50 filters of window size 9x9, VALID padding, and ReLU neurons. A max pooling layer $S1$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.

• A convolution layer $C2$ of 60 filters of window size 5x5, VALID padding, and ReLU neurons. A max pooling layer $S2$with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.

• A fully connected layer $F3$ of size 300. • A softmax layer $F4$of size 10.

Is used throughout this part of the assignment.

1.(a) Train the network by using mini-batch gradient descent learning. Set batch size =128, and learning rate α = 0.001. Images should be scaled. Plot the training cost and the test accuracy against learning epochs

Implementation

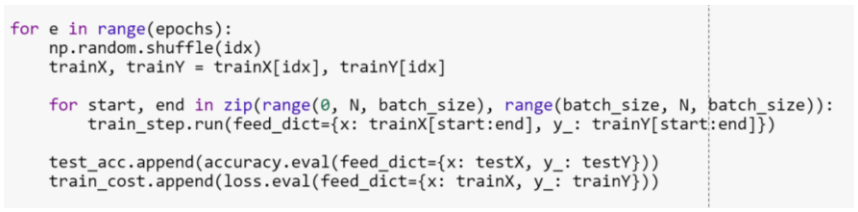
Setting batch size to 128 with learning rate of 0.001

```
NUM_CLASSES = 10
IMG_SIZE = 32
NUM_CHANNELS = 3
learning_rate = 0.001
epochs = 1000
batch_size = 128
```

Scaling the Images

```
# Scale the images
trainX = (trainX - np.min(trainX, axis=0)) / np.max(trainX, axis=0)
testX = (testX - np.min(testX, axis=0)) / np.max(testX, axis=0)
```

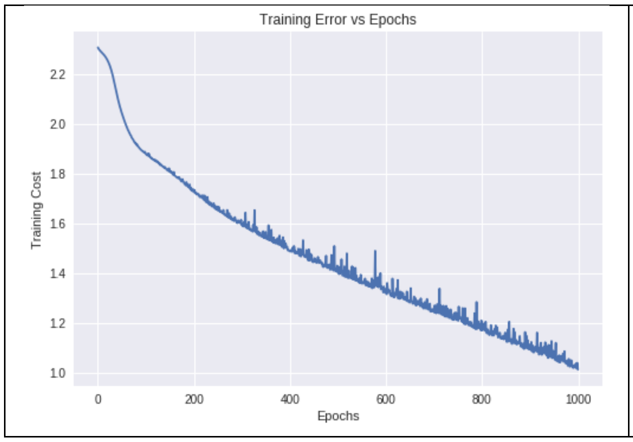
Training by mini-batch gradient descent learning

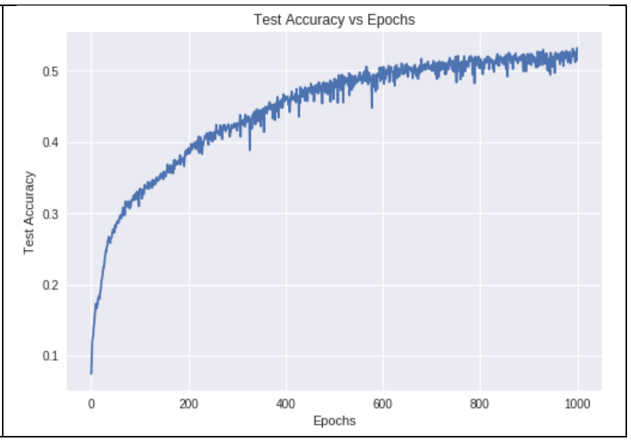


```
for e in range(epochs):
    np.random.shuffle(idx)
    trainX, trainY = trainX[idx], trainY[idx]

    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]})

    test_acc.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
    train_cost.append(loss.eval(feed_dict={x: trainX, y_: trainY}))
```

Above code snippets shows the implementation of requirements. For scaling, the test images were scaled with respect to the training data's minimum and maximum as test data is treated as unseen data.

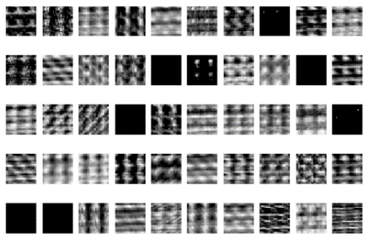
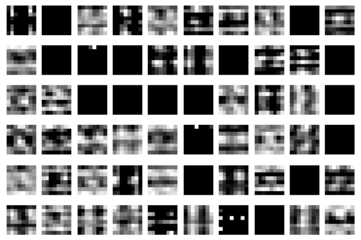
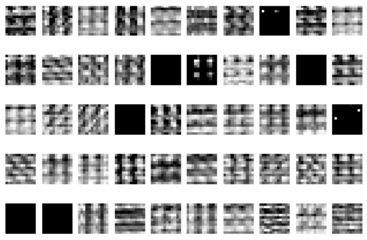
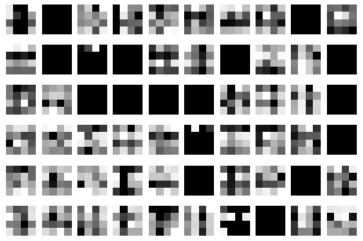
Experiment and Results



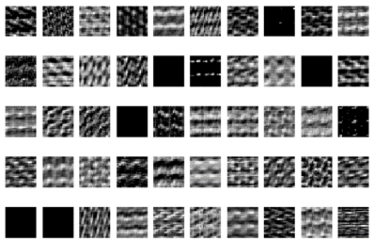
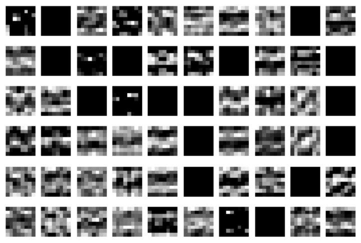
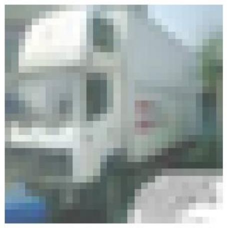
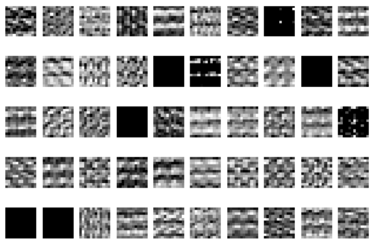
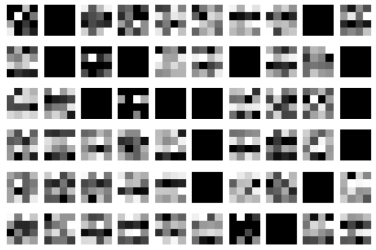


Shown above are the plots of training cost and test accuracies against epochs for the network implemented with the specified specifications. Gradually over time, the training cost decreases over time and test accuracies increases as the network learns. The network hits convergence at around 900th epoch when the test accuracy is estimated to be about 55% as indicated by the plateau.

1.(b) For any two test patterns, plot the feature maps at both convolution layers ($C1$ and $C2$) and pooling layers ($S1$ and $S2$) along with the test patterns.

Test Pattern 1		
		

The above diagram shows the test pattern 1's image (bottom left) which is a ship, C1's feature map(center up), C2's feature map(right above), S1's feature map(center below) and S2's feature map(bottom right). The images at the convolutional layers represents the features that the networks learn to recognize the object involved. If observed carefully, the feature maps at C1 and P1 as well as C2 and P2 are identical to the naked eye. This is because the pooling layers are simply just down sampling from the convolutional layers.

Test Pattern 2		
		

The above diagram shows the test pattern 2's image (bottom left) which is a truck, C1's feature map(center up), C2's feature map(right above), S1's feature map(center below) and S2's feature map(bottom right).

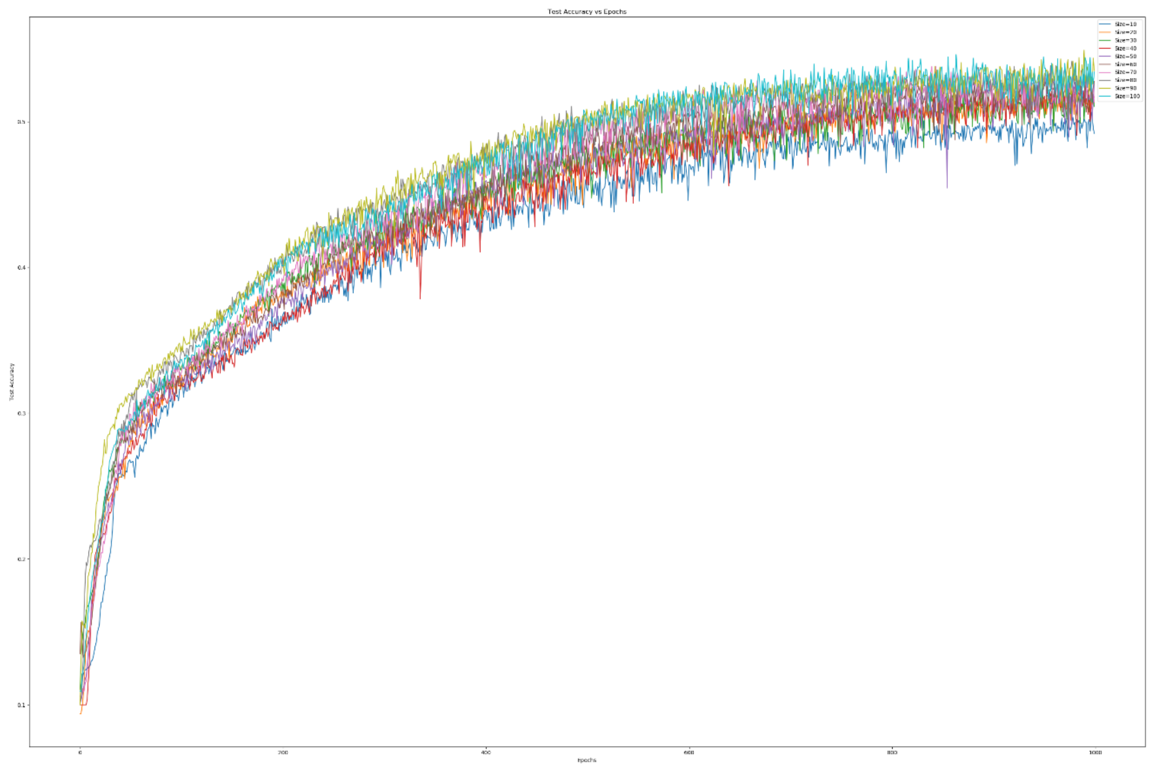
2. Using a grid search, find the optimal numbers of feature maps for part (1) at the convolution layers. Use the test accuracy to determine the optimal number of feature maps.

Implementation

```
grid_search = {'feature_map':[10,20,30,40,50,60,70,80,90,100]}
grid = ParameterGrid(grid_search)

for parameters in grid:
    print("FEATURE MAP SIZE %s" % parameters['feature_map'])
```

Grid search is done with feature map size starting from 10 to 100. For instance, when the feature map size is 10, the 1st convolutional layer has a size of 10 and 2nd convolutional layer has a size of 20 and so on and so forth. Using the sci-kit learn's sklearn.model_selection import ParameterGrid, the model is trained with different feature map sizes and its test accuracies were evaluated as follow.

Experiment and results



The above plot shows the test accuracies against epochs for feature map sizes specified in the grid search which ranges from 10-100. It is noted that going below feature map size 50 with a second feature map size of 60 which is the starting value resulted in decrement of the test accuracies at convergence. However, as the number of feature maps increases, the test accuracies and convergence doesn't improve any further beyond the feature map size 80 and 90 at convolutional layer 1 and 2 respectively. Therefore, the optimal number of feature map sizes is taken to be 80 and 90 respectively at the convolutional layers 1 and 2.

3. Using the optimal number of filters found in part (2), train the network by:

a. Adding the momentum term with momentum $\gamma = 0.1$.

b. Using RMSProp algorithm for learning

c. Using Adam optimizer for learning

d. Adding dropout to the layers

Plot the training costs and test accuracies against epochs for each case.

Implementation

Specifications

```
NUM_CLASSES = 10
IMG_SIZE = 32
NUM_CHANNELS = 3
learning_rate = 0.001
epochs = 1000
batch_size = 128
FEATURE_SIZE1 = 80
FEATURE_SIZE2 = 90
momentum = 0.1
```

Training with 4 different optimizers

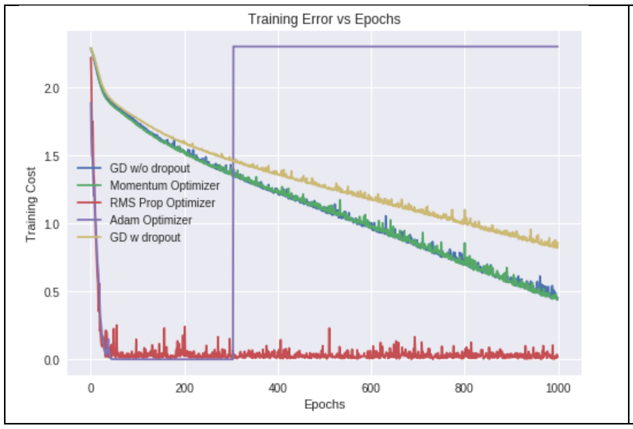
```
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
train_step1 = tf.train.MomentumOptimizer(learning_rate,momentum).minimize(loss)
train_step2 = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)
train_step3 = tf.train.AdamOptimizer(learning_rate).minimize(loss)
train_step4 = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss_drop)
```

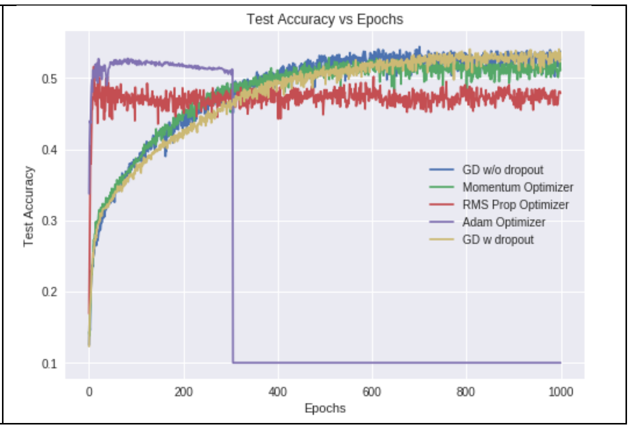
Dropout with 0.5 keep probability

```
train_step4.run(feed_dict={x: trainX[start:end], y_: trainY[start:end], keep_prob: 0.5})
```

The above code snippets show the implementation of training with different optimizers such as gradient descent, momentum optimizer, RMSProp and AdamOptmizer. Dropouts were added to training with gradient descent optimizer and evaluated.

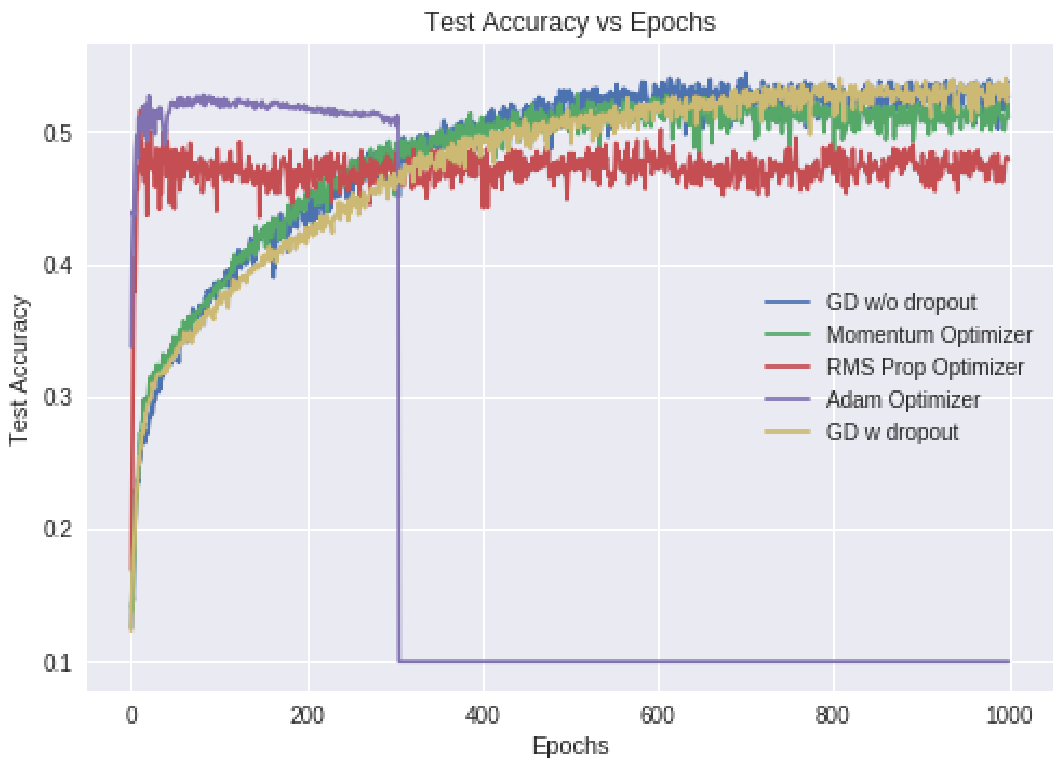
Experiment and result





The 2 plots shown above are the training costs and test accuracies against epochs for all the different optimizers used to train the network. It also includes gradient descent optimizer trained with and without the dropouts. Overtime as the network learns, the training cost generally decreases. The highest rate of decrease would be Adam optimizer followed by the RMS Prop Optimizer, Momentum optimizer with very similar to gradient descent without weights dropping and lastly gradient descent with weights dropped. As for Adam Optimizer, at 300th epoch, the training cost can be seen immediately increased. Likewise, the test accuracy also decreased sharply at the same epoch. This indicates that Adam optimizer allows us to determine more accurately when the overfitting has occurred.

4. 4. Compare the accuracies of all the models from parts (1) - (3) and discuss their performances.



In the above plot, the test accuracies against epochs from 1 to 3 can be observed. At the highest level we can see gradient descent and gradient descent without dropout with similar convergent test accuracies.

The close second would be the Adam Optimizer and the Momentum Optimizer followed by RMS Prop Optimizer.

The Momentum Optimizer when compared with the traditional gradient descent optimizer, it is able to converge at a slightly higher rate. The 2 most prominent optimizers that achieves convergence at highest rate is both the RMSProp and Adam Optmizer. This is due to the momentum terms and adaptive learning rates. The RMS prop achieves a lower convergence test accuracy as compared to adam optimizer which could be due to the momentum term causing it not to be able to reach the true minimum.

Adam optimizer also allows us to determine when overfitting has occurred more accurately as seen by a sharp drop in test accuracy around the 300th epoch. Therefore, Adam optimizer is the best suited optimizer out of all the optimizers used here.

Part B: Text Classification

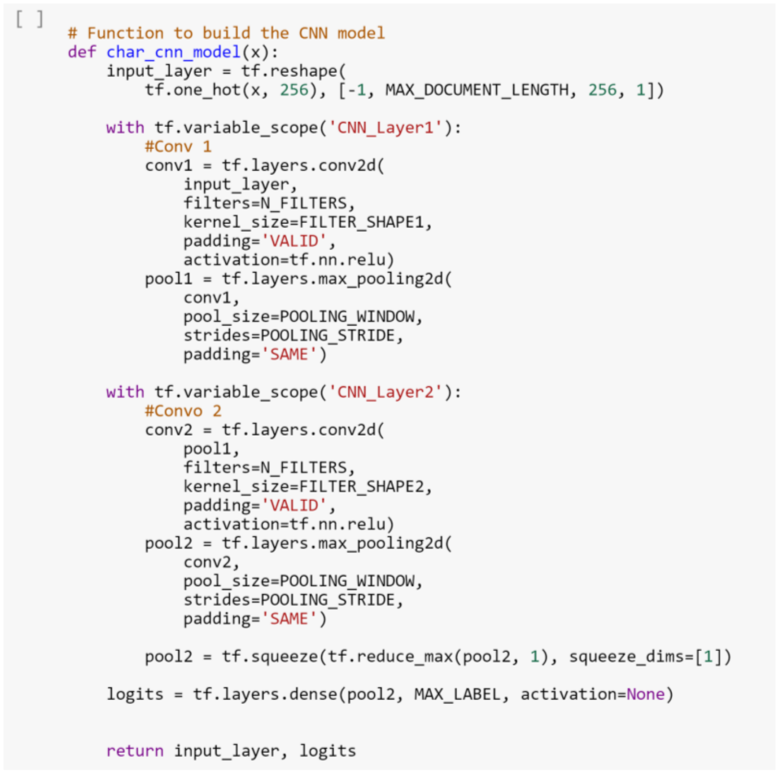
Introduction

The dataset used for training and predictions in this part of the assignment is the fist paragraphs of texts from wikipages. The paragraphs are then classified into their respective categories. The training data consists of 5600 entries and the testing dataset consists of 700 entries and there are a total of 15 categories / labels such as people, company, schools etc.

For all networks trained in this part of the assignment, maximum document length is limited to 100 characters/words with a batch size of 128 and learning rate of 0.01.

1. Design a Character CNN Classifier that receives character ids and classifies the input.

Implementation



```
[ ]
    # Function to build the CNN model
    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'):
            #Conv 1
            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')

        with tf.variable_scope('CNN_Layer2'):
            #Convo 2
            conv2 = tf.layers.conv2d(
                pool1,
                filters=N_FILTERS,
                kernel_size=FILTER_SHAPE2,
                padding='VALID',
                activation=tf.nn.relu)
            pool2 = tf.layers.max_pooling2d(
                conv2,
                pool_size=POOLING_WINDOW,
                strides=POOLING_STRIDE,
                padding='SAME')

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

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


        return input_layer, logits
```

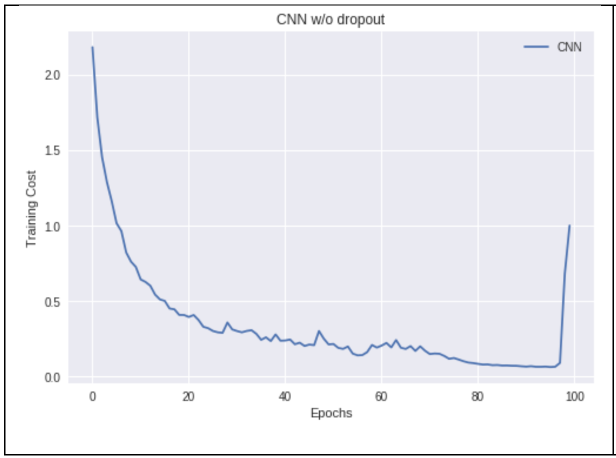
The code snippet shown above demonstrates the implementation of a 2 layer character convolutional network with 1st Convolutional Layer C1 having 10 filters of window size 20x256, VALID padding and ReLu neurons, a max pooling layer with window size 4x4 and stride =2 and padding = 'SAME'.

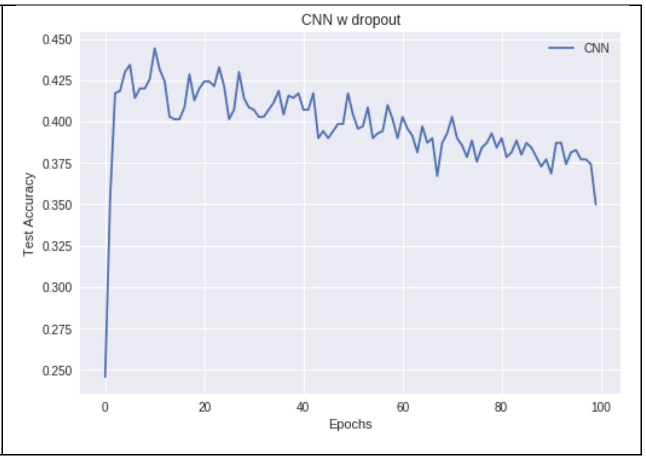
There is also a 2nd Convolutional Layer C2 having 10 filters ofwindow size 20x1, VALID padding and Relu neurons, a second max pooling layer with window size 4x4 with stride = 2 and padding ='SAME'.

Finally there is a final fully connected softmax layer connected to the previous pooling layer.

```
MAX_DOCUMENT_LENGTH = 100
N_FILTERS = 10
FILTER_SHAPE1 = [20, 256]
FILTER_SHAPE2 = [20, 1]
POOLING_WINDOW = 4
POOLING_STRIDE = 2
MAX_LABEL = 15
batch_size = 128

no_epochs = 100
lr = 0.01
```

Experiment Results

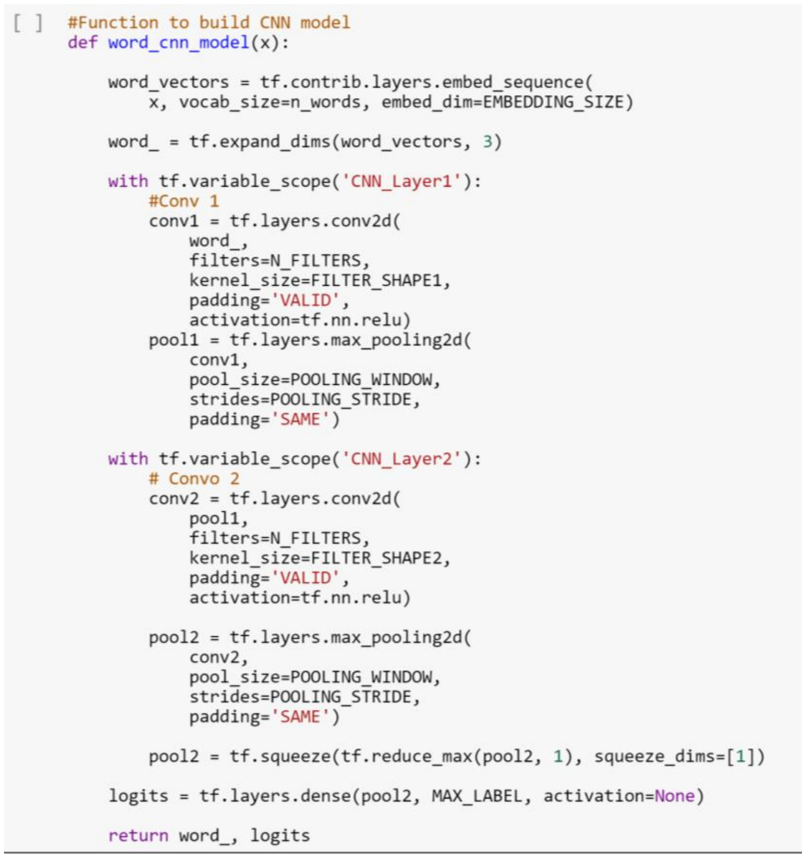




The figure on the left shows the plot of Training Cost against the number of epochs for the convolutional neural network implemented. Generally, the training cost decreases over time as the network learns. The network learns quickly as observed by a sharp increase in test accuracy from 0 – 5 epochs from 25 to 40%. The test accuracy peeks at estimated 45% after which the general graph of the test accuracy decreases over the epochs which could possibly indicate that overfitting has occurred.

2. Design a Word CNN Classifier that receives word ids and classifies the input. Pass the inputs through an embedding layer of size 20 before feeding to the CNN

Implementation



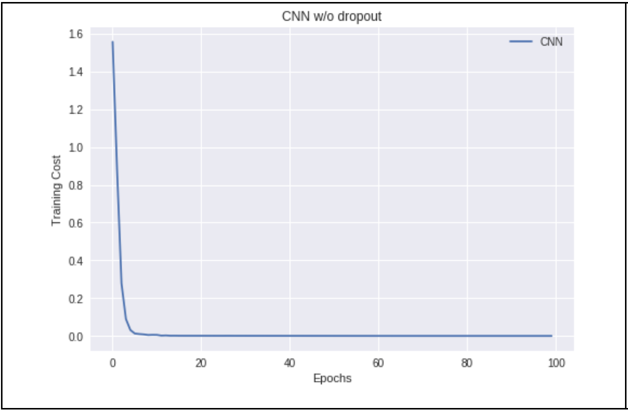
```
[ ]  #Function to build CNN model
     def word_cnn_model(x):

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

         word_ = tf.expand_dims(word_vectors, 3)

         with tf.variable_scope('CNN_Layer1'):
             #Conv 1
             conv1 = tf.layers.conv2d(
                 word_,
                 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')

         with tf.variable_scope('CNN_Layer2'):
             # Convo 2
             conv2 = tf.layers.conv2d(
                 pool1,
                 filters=N_FILTERS,
                 kernel_size=FILTER_SHAPE2,
                 padding='VALID',
                 activation=tf.nn.relu)

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

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

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

         return word_, logits
```

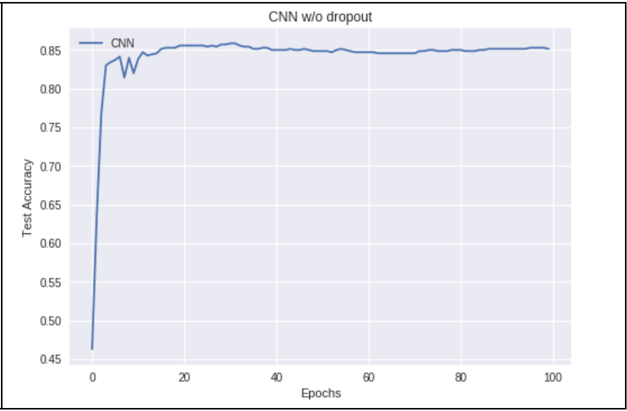
The code snippet shown above demonstrates the implementation of the 2 layer word convolutional network with 1st convolutional layer having 10 filters of window size 20x20, VALID padding, and ReLU neurons followed by a 1st pooling layer with window size 4x4 and stride 2 with ‘SAME’ pooling.

The 2nd convolutional layer consists of 10 filters of window size 20x1, VALID padding and ReLU neurons followed by the a 2nd pooling layer with window size 4x4 and stride 2 with ‘SAME’ pooling. Finally the pooling layer is connected to a fully connected output softmax layer for classification.

The significant difference between the Character CNN and the Word CNN is that the input layer for word CNN is an embedding layer. In the case of character CNN, we notice that there are a total of only 256 different characters whereas in word CNN we observe that there are a total of 38658 different words. It would be dimensionally inefficient to create a one hot vector of 38658 different words which would mostly consists of 0s making it very sparse. Hence by creating an embedding layer, it creates a dense vector of lower dimension which also classifies them by contextual similarity.

Experiment Results

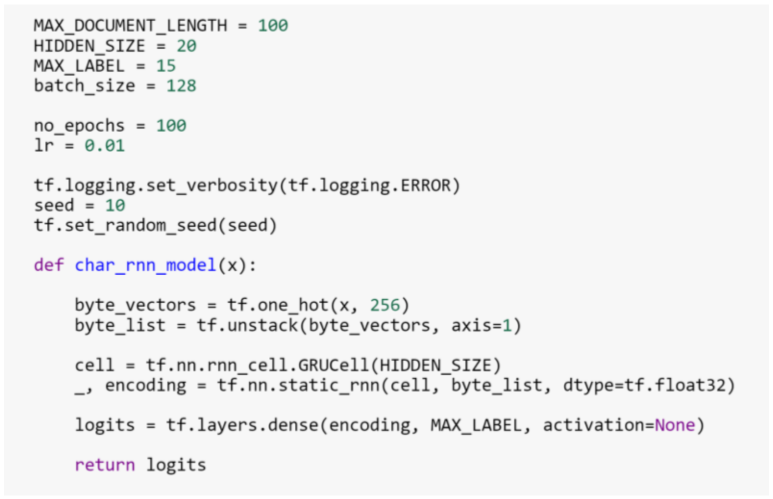




In the above 2 diagrams, it shows the Training Cost against epochs and Test Accuracies against epochs. Generally, the training cost decreases over the first few epochs. A similar trend can also be observed with the test accuracies. A sharp decrease and increase in cost and accuracies over the initial epochs indicates that the network's learning of 0.01 rate is high. At convergence it can be observed that a test accuracy of estimated 85% is achieved.

3. Design a Character RNN Classifier that receives character ids and classify the input. The RNN is GRU layer and has a hidden-layer size of 20.

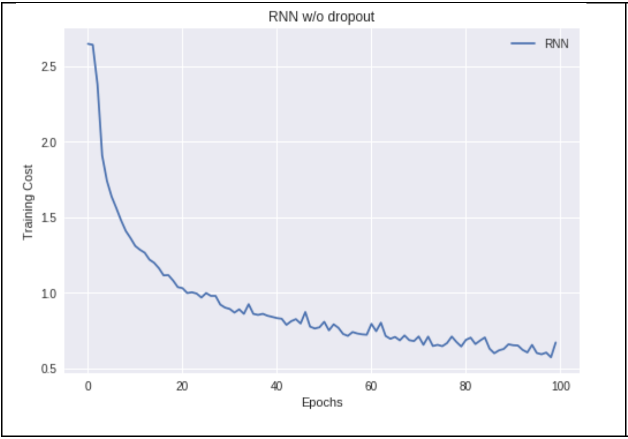
Implementation

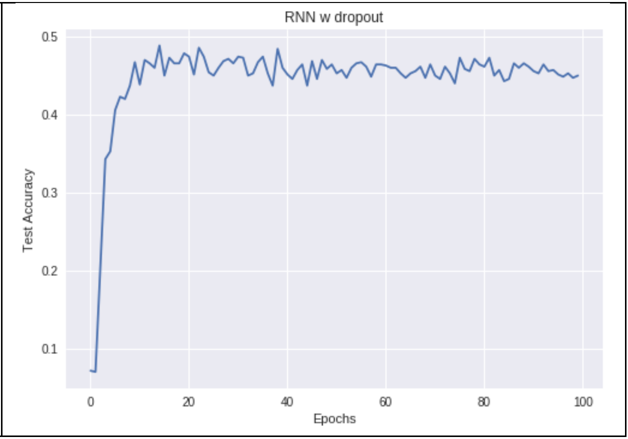


```
MAX_DOCUMENT_LENGTH = 100
HIDDEN_SIZE = 20
MAX_LABEL = 15
batch_size = 128

no_epochs = 100
lr = 0.01

tf.logging.set_verbosity(tf.logging.ERROR)
seed = 10
tf.set_random_seed(seed)

def char_rnn_model(x):

    byte_vectors = tf.one_hot(x, 256)
    byte_list = tf.unstack(byte_vectors, axis=1)

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

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

    return logits
```

The code snippet above shows a simple implementation of the Character RNN model. The cells in the network are Gated Recurrent Units (GRU) of hidden layer neurons size of 20. A one hot vector for 256 different character classifications are unstacked along yaxis into a list and fed as input tensors. After which, the encoding or final states of the cell is then connected to the fully connected softmax layer.

Experiment Results

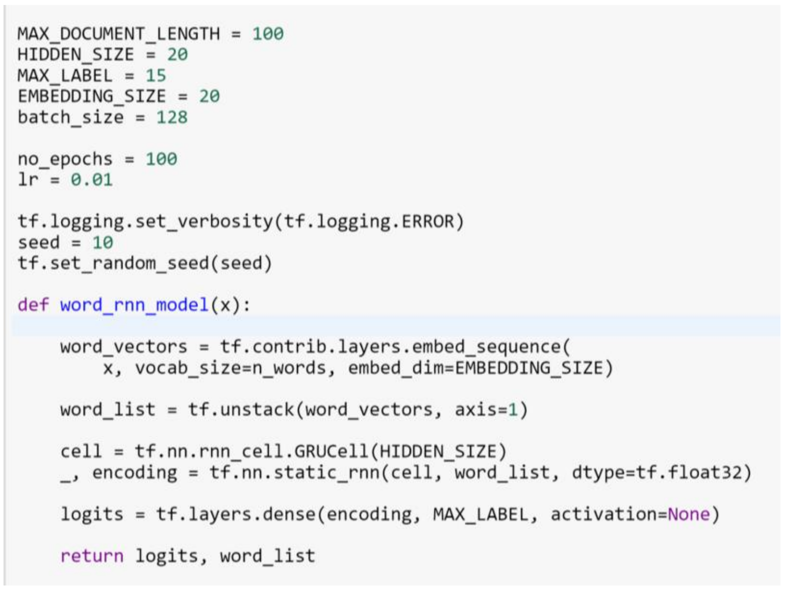




In the above 2 diagrams it shows the training cost (left) and the test accuracies (right) against epochs. Generally, as the network learns over the epochs, the training cost and test accuracies decreases and increases respectively. Again, the network learns fast as it can be observed that over 0-10 epochs there is an increase in test accuracy from 10% to 47%. At convergence the network's accuracy is estimated to be 45%.

4. Design a word RNN classifier that receives word ids and classify the input. The RNN is GRU layer and has a hidden-layer size of 20. Pass the inputs through an embedding layer of size 20 before feeding to the RNN.

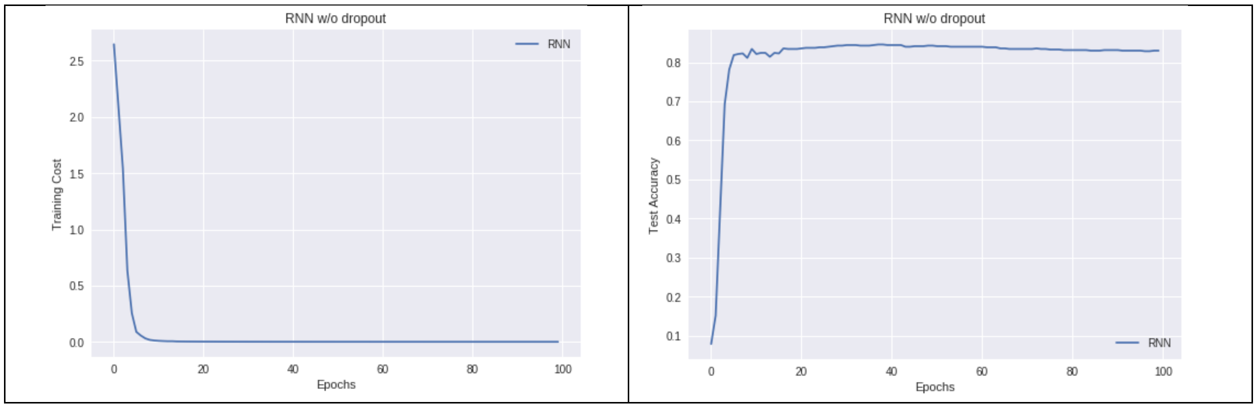
Implementation



```
MAX_DOCUMENT_LENGTH = 100
HIDDEN_SIZE = 20
MAX_LABEL = 15
EMBEDDING_SIZE = 20
batch_size = 128

no_epochs = 100
lr = 0.01

tf.logging.set_verbosity(tf.logging.ERROR)
seed = 10
tf.set_random_seed(seed)

def word_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
```

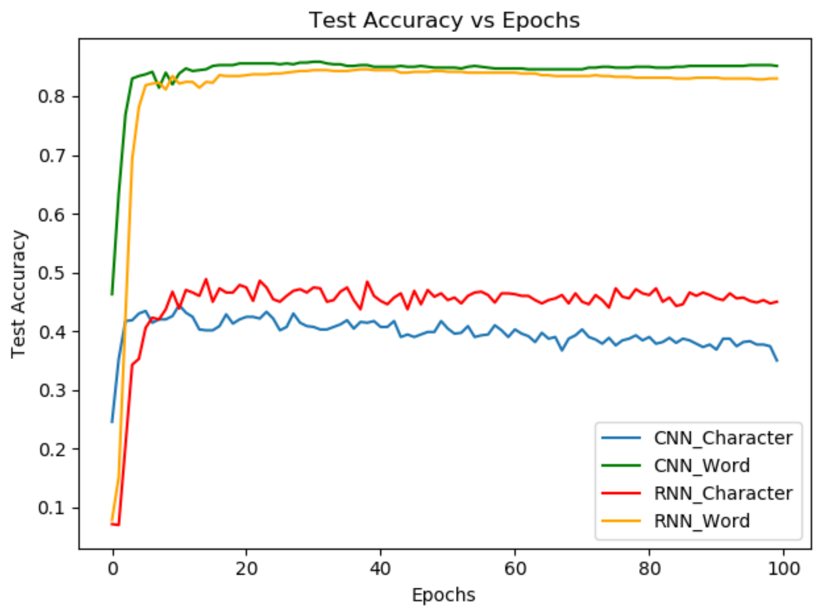
In the above code snippet, an implementation of the word recurrent neural network using a GRU cell of hidden layer size 20 is shown. Likewise, as in the equivalent convolutional network, inputs are embedded into embedded layer which creates a denser and lower dimensional vector as compared to a sparse and high dimension one hot vector. Finally, the final states of the cell is then fully connected to a softmax output layer.

Experiment Result



The above 2 diagrams show the training cost (left) and the test accuracies (right) against number of epochs respectively. Generally, as the training cost decreases over the epochs the test accuracies increase as the network trains. Attributing to a high learning factor, it is evident that the network learns fairly fast as we observe a sharp increment in test accuracy from initial few epochs (0-8). At convergence the network is able to achieve 84% of test accuracy.

5.(i) Compare the test accuracies and the running times of the networks implemented in parts (1) – (4).

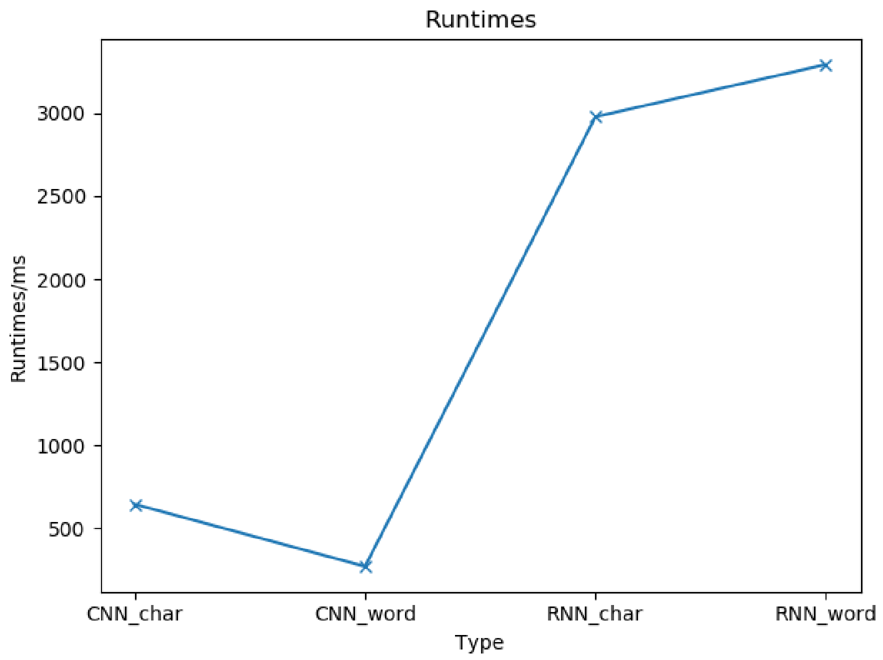


The figure above shows the plot of test accuracies against the epochs ran for different networks. These networks include character level convolutional neural network, character level recurrent neural network, word level convolutional network and word level recurrent neural network.

When we compare convergence test accuracies, we observe that the test accuracies for the RNN model is able to achieve higher accuracies than their CNN model counter parts. For instance, character RNN is at estimated to attain a 45% accuracy where as the character CNN is about 5% lower. This could be attributed to the fact that gated recurrent units implemented in the RNN models are able to learn sequential data better. As the network trains, during the backpropagation the gradient gets multiplied a significantly great number of times which causes the small gradients to either diminish or large gradients to explode. This is known as the exploding and vanishing gradient problems which can in turn affect the accuracies. In the GRU cells, there are gates implemented to forget previous hidden states and when to update hidden states when exposed to new information. However, at word level classification this is not true as the word CNN and word RNN has very similar convergence test accuracies. Hence the RNN models are able to achieve higher accuracies than the CNN models at character level.

Second feature that is most prominent is that character level classifications achieves lower test accuracies than the word level classifications. This could be attributed to the embedded layer which is present in the word level classification. By embedding the words into an embedding vector which also learns a weight matrix, the word level classification is able to achieve a higher convergent accuracy than the character level classifications. For instance, the word RNN is able to achieve test accuracies of 85% at convergence compared to character RNN which achieves test accuracies of 75%, a discrepancy of 10%. In conclusion, the word level classifications of the text are able to achieve higher levels of accuracies than that of the character level classifications.

Lastly, the RNN models tends to learn slower than the CNN models. This is true at both the character and word level as both the CNN accuracy graphs (green and blue) has steeper slopes at initial epochs than the RNN accuracy graphs(red and orange).

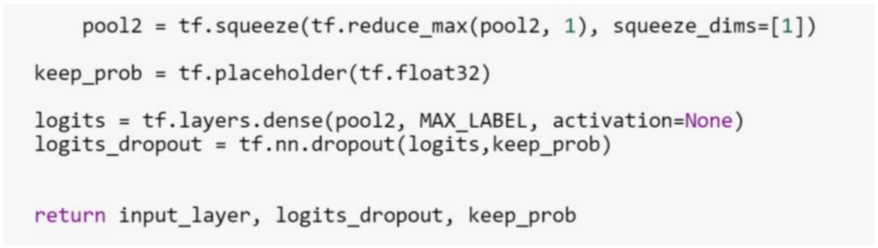


The above graph shows the run times in milli seconds of different networks which consists of character level CNN, character level RNN, word level CNN and word level RNN. Generally, the RNN models have higher runtimes than CNN models indicating that they are slower to run each epoch. This could be due to the fact that in the RNN model there are more parameters to be updated per epoch than that in the CNN models as there are gates such as forget gate's weight parameters and input gate' parameters. The interesting trend to note here is that in the CNN model the word level classification's runtime is lower than that of the character's but in the RNN model the opposite is observed.

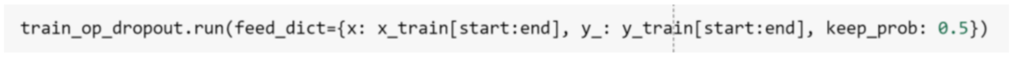
5.(ii) Compare and comment on the accuracies of the networks with/without dropout.

Implementation

CNN

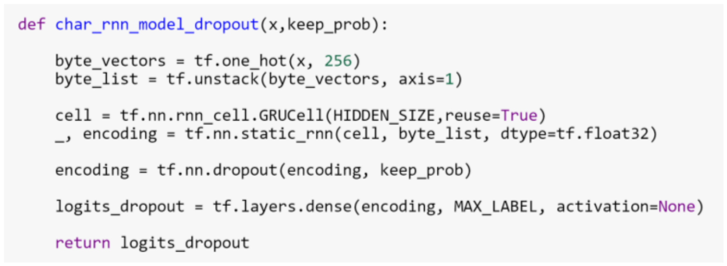


```
    pool2 = tf.squeeze(tf.reduce_max(pool2, 1), squeeze_dims=[1])

keep_prob = tf.placeholder(tf.float32)

logits = tf.layers.dense(pool2, MAX_LABEL, activation=None)
logits_dropout = tf.nn.dropout(logits,keep_prob)


return input_layer, logits_dropout, keep_prob
```

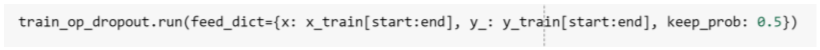


```
train_op_dropout.run(feed_dict={x: x_train[start:end], y_: y_train[start:end], keep_prob: 0.5})
```

RNN



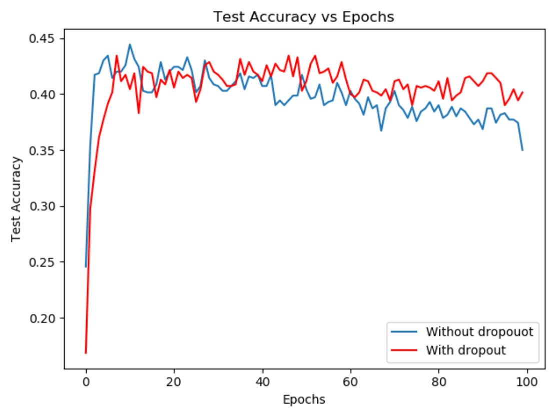
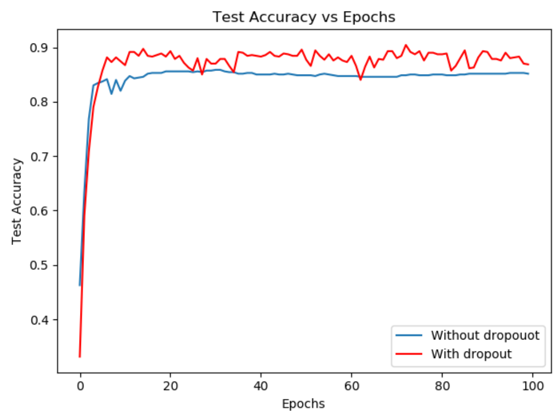
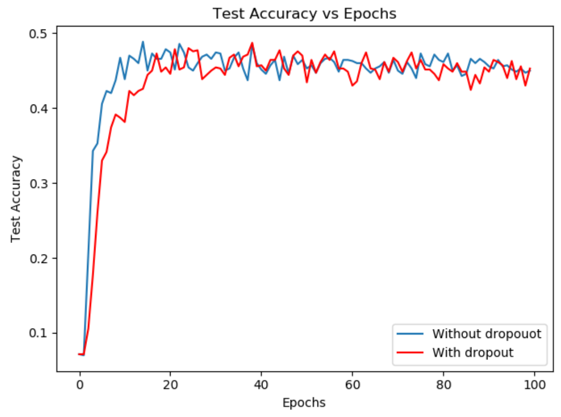
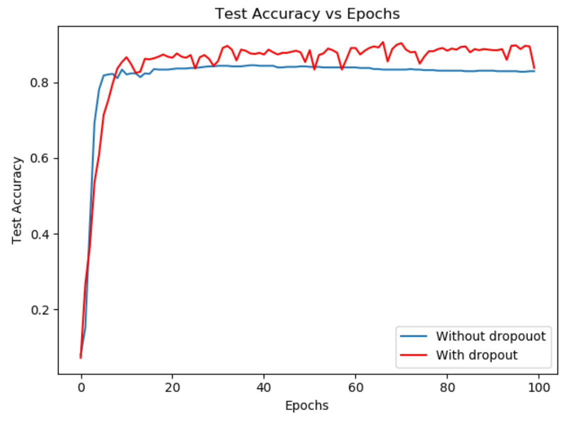
```
def char_rnn_model_dropout(x,keep_prob):

    byte_vectors = tf.one_hot(x, 256)
    byte_list = tf.unstack(byte_vectors, axis=1)

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

    encoding = tf.nn.dropout(encoding, keep_prob)

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

    return logits_dropout
```



```
train_op_dropout.run(feed_dict={x: x_train[start:end], y_: y_train[start:end], keep_prob: 0.5})
```

For both the CNN and RNN models, dropouts are only introduced at the fully connected layer (i.e output softmax layer). This is because this layer has the highest number of weight parameters to be updated. In this experiment keep_prob is kept to 0.5 which follows the first paper by Hinton(2012) that originally proposed dropout layers. [1]

Experiment and Results

	Character Level	Word Level
CNN		
RNN		

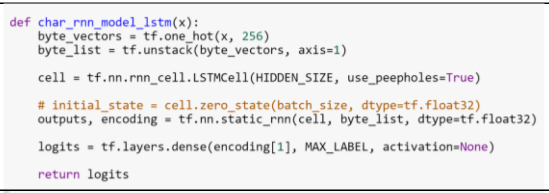
The four diagrams above represents the plot of test accuracies against epochs with and without epochs for Character CNN, Word CNN, Character RNN and Word RNN. Generally, across all the models it is observed that the models with dropouts introduced, learns the data faster than the models without the dropouts introduced indicated by the steeper curve in the initial 0-10 epochs.

In the character CNN, the model with dropouts introduced is able to achieve a higher convergence test accuracy than the model without dropouts. However, in the character RNN, a similar test accuracy plots were observed for both the models with and without the dropouts introduced.

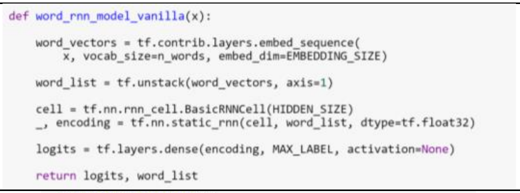
At the word level, for both the CNN model and RNN, the model with the dropouts introduced are able to achieve a higher convergence test accuracy than the one without dropouts. For instance, in the word CNN, model with dropout achieves nearly 90% test accuracies compared to 85% of the model without dropouts. Similarly in the word RNN model the model with dropouts introduced are able to achieve a higher test accuracy by estimated 5%. Therefore, introducing dropouts generally allows the network to achieve higher convergence test accuracies as compared to models without dropouts.

6.(a) Replace the GRU layer in 3 and 4 with (i) a vanilla RNN layer and (ii) a LSTM layer and compare the test accuracies.

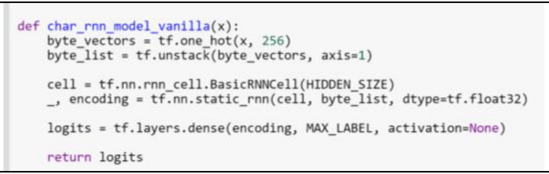
Implementation



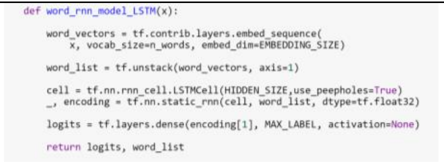
```python
def char_rnn_model_lstm(x):
    byte_vectors = tf.one_hot(x, 256)
    byte_list = tf.unstack(byte_vectors, axis=1)

    cell = tf.nn.rnn_cell.LSTMCell(HIDDEN_SIZE, use_peepholes=True)

    # initial_state = cell.zero_state(batch_size, dtype=tf.float32)
    outputs, encoding = tf.nn.static_rnn(cell, byte_list, dtype=tf.float32)

    logits = tf.layers.dense(encoding[1], MAX_LABEL, activation=None)

    return logits
```



```python
def word_rnn_model_vanilla(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.BasicRNNCell(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
```



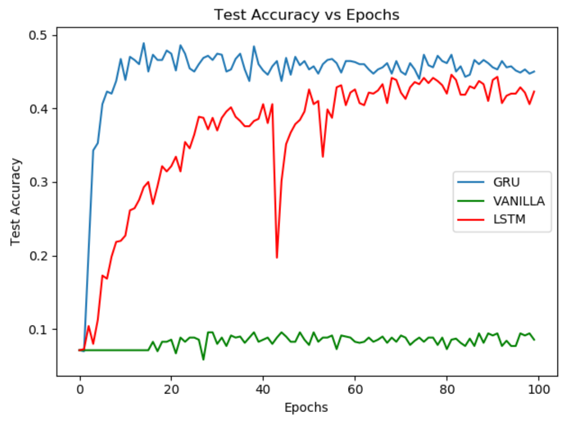
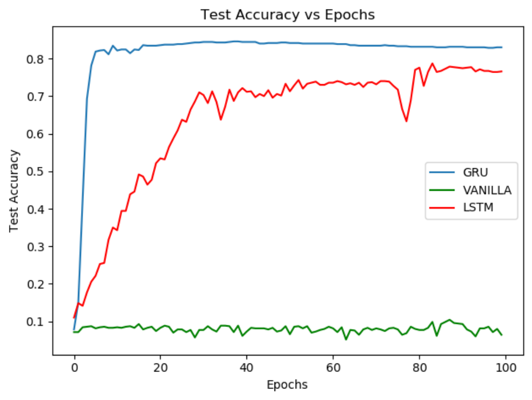
```python
def char_rnn_model_vanilla(x):
    byte_vectors = tf.one_hot(x, 256)
    byte_list = tf.unstack(byte_vectors, axis=1)

    cell = tf.nn.rnn_cell.BasicRNNCell(HIDDEN_SIZE)
    _, encoding = tf.nn.static_rnn(cell, byte_list, dtype=tf.float32)

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

    return logits
```



```python
def word_rnn_model_LSTM(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.LSTMCell(HIDDEN_SIZE,use_peepholes=True)
    _, encoding = tf.nn.static_rnn(cell, word_list, dtype=tf.float32)

    logits = tf.layers.dense(encoding[1], MAX_LABEL, activation=None)

    return logits, word_list
```

The above 4 code snippets shows the implementation of RNN models using vanilla and LSTM cells respectively at both the character and word level.

Experiment and results

Character	Word
	

Shown above are the plots of test accuracies against epochs for recurrent neural networks of 3 different cells: gated recurrent unit, basic recurrent unit (vanilla) and long short-term memory. The diagram on the left represents those at the character level while the diagram on the right represents those at the word level.

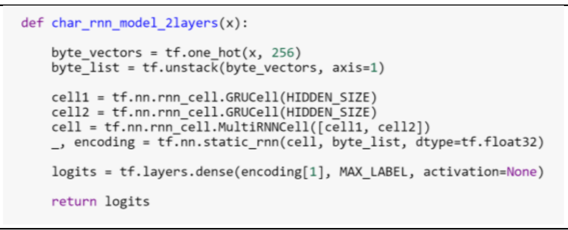
Across both the diagrams, it can be observed that the test accuracies at convergence are highest for the RNN models with GRU cells followed by LSTM cells and lastly by the BasicRNN cells. It is prominent that in both the plots the vanilla cells' fairs very badly for the test accuracies as it just stays fluctuates below the 10% mark. This could be due to the fact that BasicRNN cells does not solve the issue of exploding and vanishing gradients are they are just simple recurrent units with a tanh activation functions without the forget gates and input gates to learn when to forget and when to update new information.

Though there is no generalized fact that RNN with GRU cells tends to fair better in terms of accuracy compared to the LSTM cells, in this context of text classification it is evident that the GRU cells are able to achieve higher accuracies than those of the LSTM. This indicates that in the context of text classification, GRU cells which has one gating unit which decides the forgetting and updating factor simultaneously is more beneficial than LSTM cells which has separate gating units for the forgetting and updating factor.

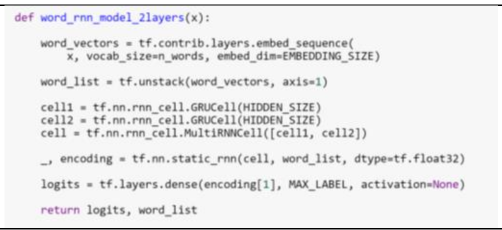
In conclusion, the GRU cells at convergence can achieve test accuracies of 47% for character and 84% for word, the LSTM cells at convergence can achieve test accuracies of 43% for character and 75% for word while the vanilla RNN can only achieve test accuracies lower than 10% in both cases.

6.(b) Replace the GRU layer in 3 and 4 with 2-layer RNN and compare the test accuracies.

Implementation



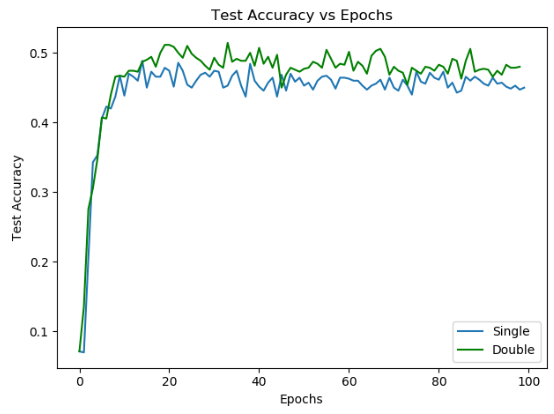
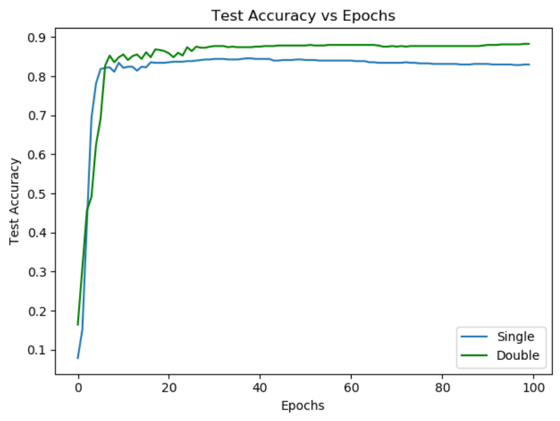
```
def char_rnn_model_2layers(x):

    byte_vectors = tf.one_hot(x, 256)
    byte_list = tf.unstack(byte_vectors, axis=1)

    cell1 = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    cell2 = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    cell = tf.nn.rnn_cell.MultiRNNCell([cell1, cell2])
    _, encoding = tf.nn.static_rnn(cell, byte_list, dtype=tf.float32)

    logits = tf.layers.dense(encoding[1], MAX_LABEL, activation=None)

    return logits
```



```
def word_rnn_model_2layers(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)

    cell1 = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    cell2 = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    cell = tf.nn.rnn_cell.MultiRNNCell([cell1, cell2])

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

    logits = tf.layers.dense(encoding[1], MAX_LABEL, activation=None)

    return logits, word_list
```

The above 2 code snippets shows the implementation of 2-layers RNN at both character level and word level.

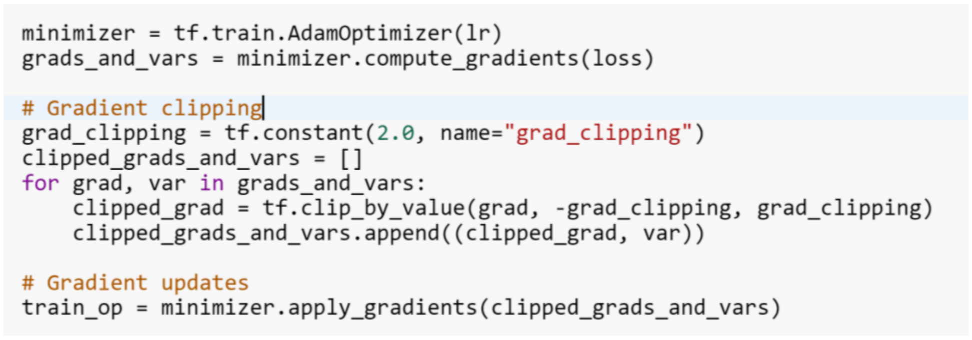
Experiments and Results

Character	Word
	

The above 2 diagrams shown are the plots of test accuracies against epochs for single layer RNN and multi layered RNN (2-layers) implemented with GRU cells. In both the character level and word level classifications, it is observed that the multilayered RNN is able to achieve higher test accuracies than the single layer RNN. As such there is about 5% discrepancy in both the character level and word level for double layer and single layer. Hence in terms of accuracy, although the multi-layer does better, it might not be a good trade off as the runtime for multi-layer is higher than that of the single layer due to more parameters involved to be updated.

6.(c) Add gradient clipping to RNN training with clipping threshold = 2 for 3-4 and compare the test accuracies.

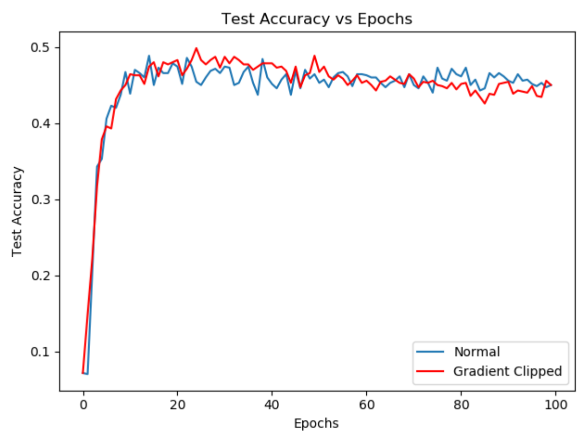
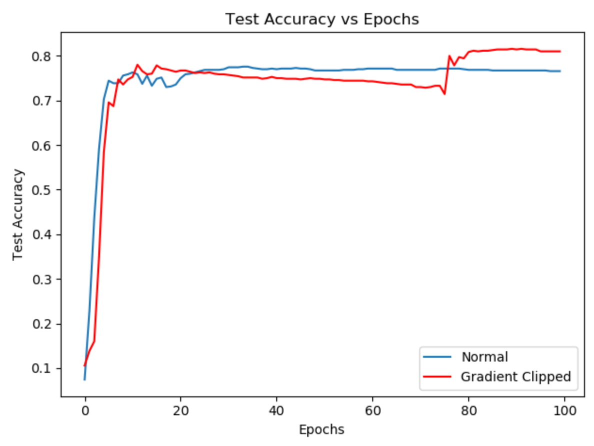
Implementation



```
minimizer = tf.train.AdamOptimizer(lr)
grads_and_vars = minimizer.compute_gradients(loss)

# Gradient clipping
grad_clipping = tf.constant(2.0, name="grad_clipping")
clipped_grads_and_vars = []
for grad, var in grads_and_vars:
    clipped_grad = tf.clip_by_value(grad, -grad_clipping, grad_clipping)
    clipped_grads_and_vars.append((clipped_grad, var))

# Gradient updates
train_op = minimizer.apply_gradients(clipped_grads_and_vars)
```

The above code snippet shows the implementation of the gradient clipping with a threshold of 2.0.

Experiments and Results

Character	Word
	

The above diagrams show the plots of test accuracies against epochs for RNN models with GRU cells with and without gradient clipping. At both the character level and the word level, it is observed that the test accuracies achieved are of similar percentage on average at convergence. This could indicate that gradient clipping does not make much difference for the GRU cell RNN model as the GRU cells are able to handle the exploding and vanishing gradients with the forget gates and update gates. Having additional gradient clipping which is to put a threshold on the gradients would not improve the accuracies further in this case. However, if implemented in basic RNN cells, it could have a different result which could possibly indicate that basic RNN model with gradient clipping would achieve a better test accuracy.

References

[1] - Hinton(2012) Improving neural networks by preventing co-adaptation of feature detectors https://arxiv.org/pdf/1207.0580.pdf