CZ4042 Neural Network & Deep Learning

Assignment 2

Name	Wang Xiaoyu	Bryan Leow Xuan Zhen
Matriculation Number	U1721992H	U1721837L

Part A: Object Recognition

Introduction

In this project, we are required to build a convolutional neural network consisting of an input layer, several convolutional and pooling layers, a fully connected layer, as well as a softmax output layer. The model is used to classify RGB images from the CIFAR-10 dataset.

Dataset[1]

There are 10,000 training images of size 32 x 32, with labels ranging from 0 to 9. Testing is done on 2,000 images of the same size.

The raw data is given in the form of a 10000 x 3072 numpy array. Each row of the array stores a 32 x 32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image. Given this information, we performed some transformation to the raw data, before feeding it into the network. The details are elaborated below.

The labels are given as a list of 10,000 numbers in the range 0 to 9. These numbers represent the categories listed below (not in order):

airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

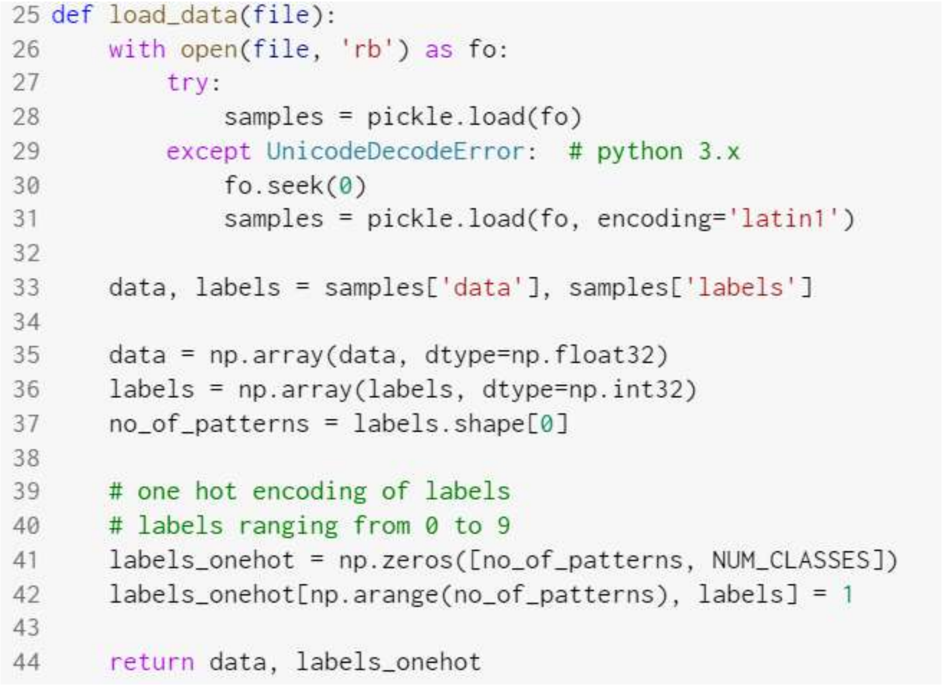
[1] https://www.cs.toronto.edu/~kriz/cifar.html

Method

Image loading

From the dataset documentation, we can see that after loading the data file using *pickle*, the file contains a dictionary with keys *data* and *labels*. We extracted out the values of *data* and *labels* and converted them into numpy arrays. For the labels, we further converted them into one hot labels, which can be directly fed into the neural networks model.

Here, we did a minor modification to the sample code given in *start_project_2a.py*. In line 42 below, we changed *labels-1* to *labels*. Because the labels are already in the range 0~9, there is no need to minus 1 anymore.

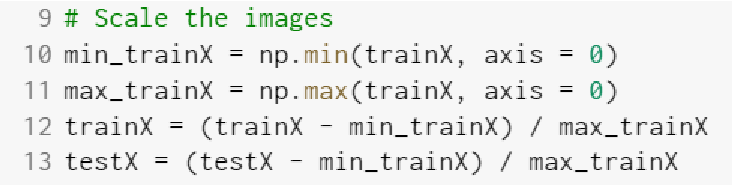


```
25 def load_data(file):
26     with open(file, 'rb') as fo:
27         try:
28             samples = pickle.load(fo)
29         except UnicodeDecodeError:  # python 3.x
30             fo.seek(0)
31             samples = pickle.load(fo, encoding='latin1')
32 
33     data, labels = samples['data'], samples['labels']
34 
35     data = np.array(data, dtype=np.float32)
36     labels = np.array(labels, dtype=np.int32)
37     no_of_patterns = labels.shape[0]
38 
39     # one hot encoding of labels
40     # labels ranging from 0 to 9
41     labels_onehot = np.zeros([no_of_patterns, NUM_CLASSES])
42     labels_onehot[np.arange(no_of_patterns), labels] = 1
43 
44     return data, labels_onehot
```

Input feature scaling

Before feeding in the images, we also normalised the image inputs. It is an important step and ensures that each input parameter (pixel in this case) has a similar data distribution. This makes convergence faster while training the network[2]. Here, normalisation is done by subtracting the minimum value of the pixel across all training images, and dividing by the maximum value of that pixel across all training images. After normalisation, the pixel values range from 0 to 1.

[2] https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258



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

One assumption we made is that the test images follow roughly the same distribution as the training images. Given this, we can use the minimum and maximum pixel values from the training images to normalise the test images as well. This is supposed to be less biased because the number of training images is significantly larger than that of the test images.

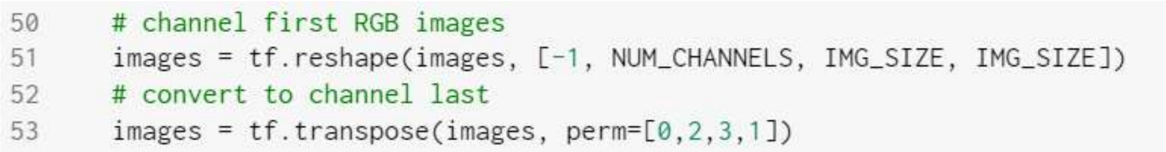
Raw data transformation

As mentioned above, the raw data is given as a 10000 x 3072 numpy array. In order to feed it into the convolutional layer correctly, we need to reshape it to a 4-dimensional tensor.

The usual way to achieve this is to simply use

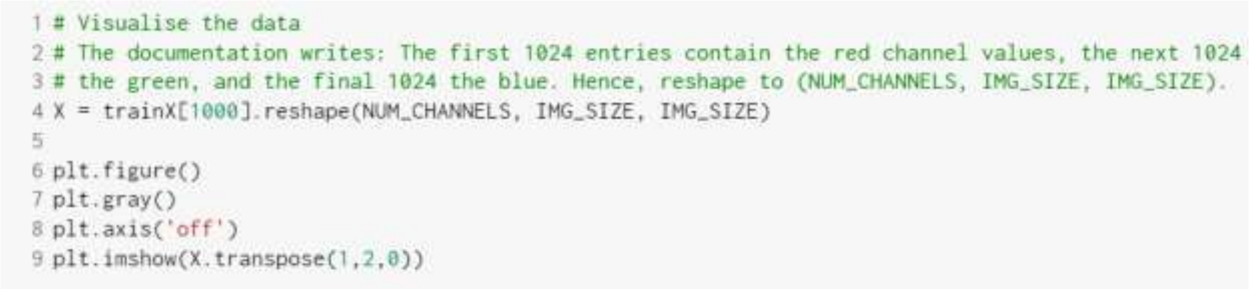
tf.reshape(images, [-1, IMG_SIZE, IMG_SIZE, NUM_CHANNELS])

However, in this case, we need more effort because the dataset is in a channels-first format, i.e. “the first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue”. To deal with this, we first reshaped the images into a *[-1, NUM_CHANNELS, IMG_SIZE, IMG_SIZE]* shape so that the array contains the correct image information. We then used transpose to convert it to channels-last format so as to input to the convolutional layer.



```
    # channel first RGB images
    images = tf.reshape(images, [-1, NUM_CHANNELS, IMG_SIZE, IMG_SIZE])
    # convert to channel last
    images = tf.transpose(images, perm=[0,2,3,1])
```

Using the same method, we visualised one of the images below. This double confirms that we did the correct reshape and transformation.



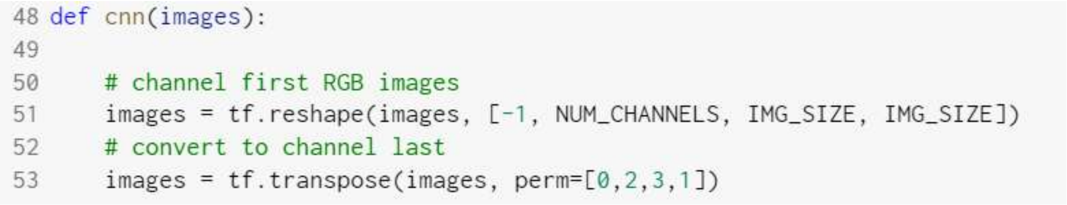
```
# Visualise the data
# The documentation writes: The first 1024 entries contain the red channel values, the next 1024
# the green, and the final 1024 the blue. Hence, reshape to (NUM_CHANNELS, IMG_SIZE, IMG_SIZE).
X = trainX[1000].reshape(NUM_CHANNELS, IMG_SIZE, IMG_SIZE)

plt.figure()
plt.gray()
plt.axis('off')
plt.imshow(X.transpose(1,2,0))
```

```
<matplotlib.image.AxesImage at 0x7f31c92d4780>
```

Model building

We started with a simple convolutional neural network using TensorFlow package in Python. The network consists of an input layer of 3 x 32 x 32 dimensions, a convolutional layer of 50 filters and a max pooling layer, a convolutional layer of 60 filters and a max pooling layer, a fully connected layer, and a softmax output layer of size 10.

The input layer takes in a batch of vectors of dimension 3 x 32 x 32, because the size of the images is 32 x 32 and these images have 3 colour channels (RGB). After taking in the images as 2-dimensional vectors, these vectors are then reshaped and transposed as described in the above *raw data transformation* section.



```
def cnn(images):

    # channel first RGB images
    images = tf.reshape(images, [-1, NUM_CHANNELS, IMG_SIZE, IMG_SIZE])
    # convert to channel last
    images = tf.transpose(images, perm=[0,2,3,1])
```

After transformation, we now have a 4-dimensional tensor ready to be fed into the convolutional neural networks. The first convolutional layer *conv_1* has 50 filters of size 9 x 9 with ReLU as the activation function. This is followed by a max pooling layer *pool_1* of non-overlapping window size 2 x 2. After the pooling, the feature maps are passed to another convolutional layer *conv_2* which has 60 filters of size 5 x 5 with ReLU as the activation function. This is again followed by a max pooling layer *pool_2* of non-overlapping window size 2 x 2. The feature maps are then flattened into a 2-dimensional vector and passed into a fully connected layer of 300 neurons with ReLU activation. Finally, the results are passed into a softmax layer to compute the final output.

As this is a classification problem, the cost function is defined as the multi-class cross entropy loss, and the accuracy is defined by the percentage correct prediction. The training process and updating of parameters is handled by in-built tensorflow functions such as *tf.train.GradientDescentOptimizer* to minimise the cost function.

The parameters in the model are initialized to the following:

- *NUM_CLASSES = 10*
- *IMG_SIZE = 32*
- *NUM_CHANNELS = 3*
- *learning_rate = 0.001*
- *epochs = 2000*
- *batch_size = 128*

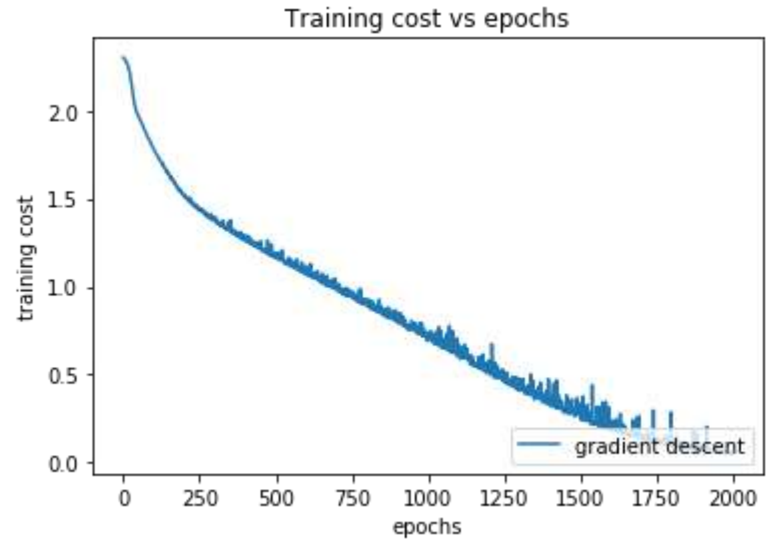
To assess the performance during the training procedure, we plot the training loss and test accuracy against epochs.

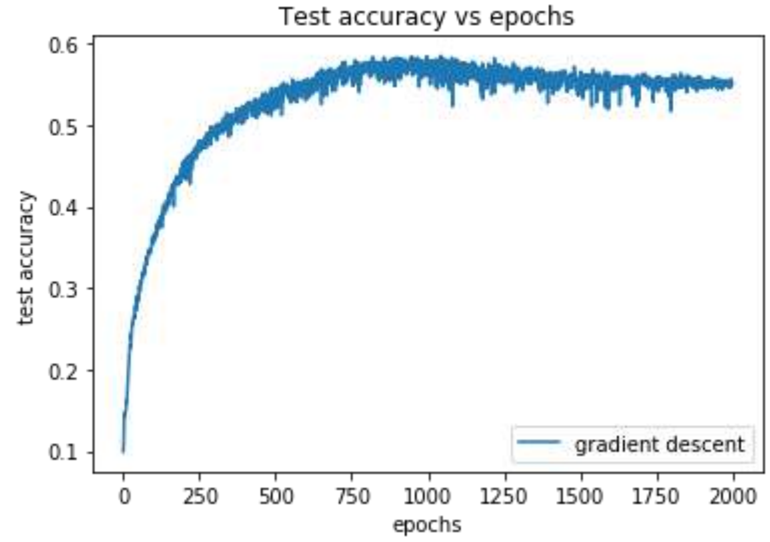
Experiments and Results

Question 1

a) Plot the training cost and the test accuracy against learning epochs.

For this question, we use the base model with 50 filters for *conv_1* and 60 filters for *conv_2*. The optimiser used is mini-batch gradient descent.





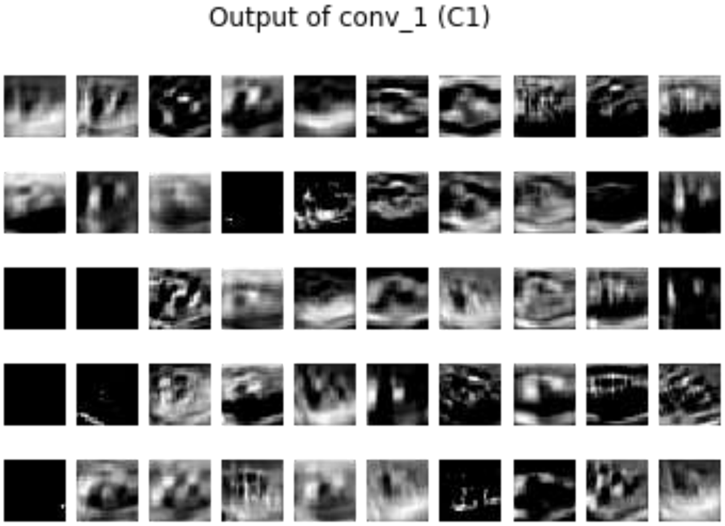
As shown above, the first graph shows the training cost against epochs, while the second graph shows the test accuracy against epochs.

We trained the model to 2000 epochs, in order to let it train sufficiently. From the second graph, we can see that the test accuracy starts to drop after about 1000 epochs, whereas in the first graph, the training cost continues to decrease. This means that after 1000 epochs, over-fitting has occurred.

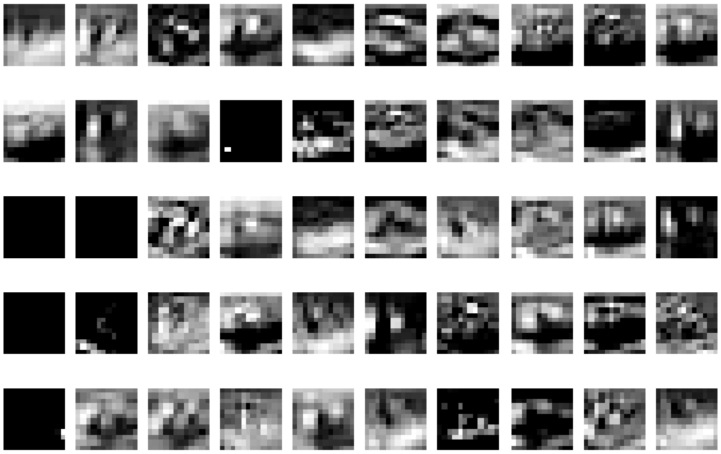
We also observe that the best test accuracy is below 60%. This might be due to the complex nature of the input images. Given the simple base model with only 2 convolutional layers, these complex features cannot be captured precisely for decision-making. In the following questions, we can explore some ways to improve the performance.

a) **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.**

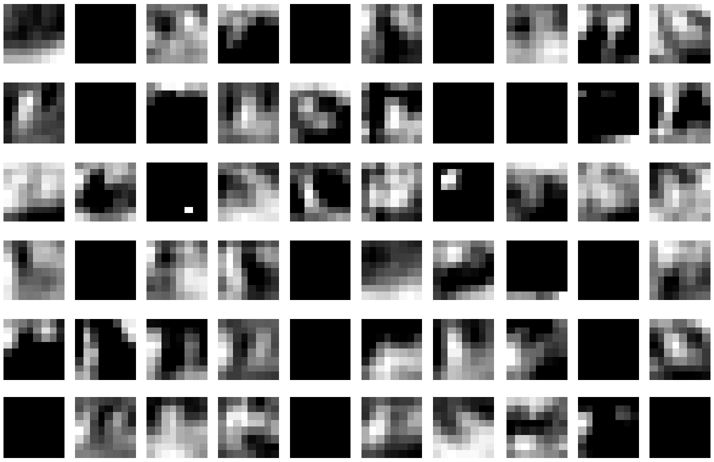
After training the model, we randomly selected two images from the test dataset to visualise the feature maps at different layers. The first image and its feature maps are shown below.



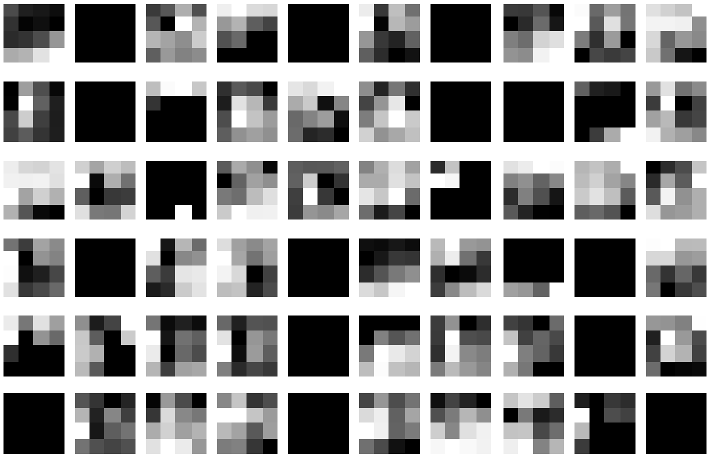
Output of pool_1 (S1)



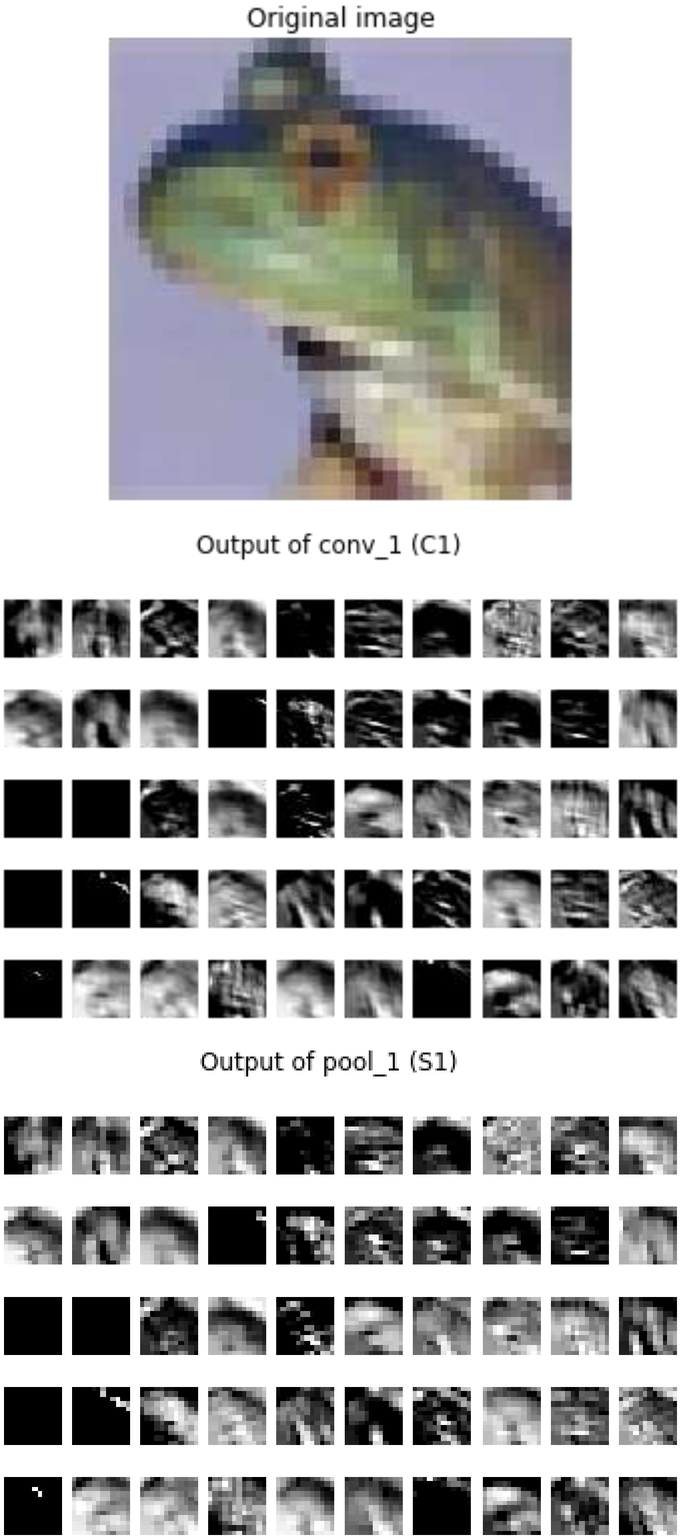
Output of conv_2 (C2)



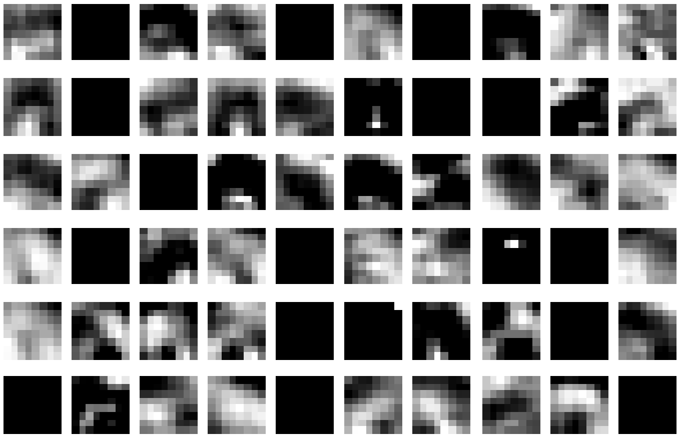
Output of pool_2 (S2)



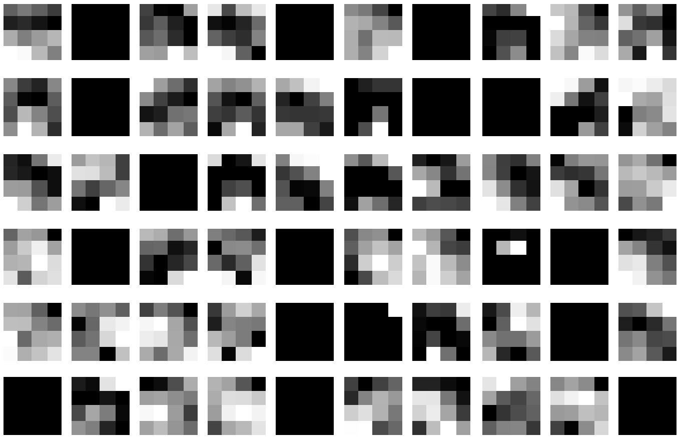
The second image from the test dataset and its corresponding feature maps are shown below:


Original image
Output of conv_1 (C1)
Output of pool_1 (S1)

Output of conv_2 (C2)



Output of pool_2 (S2)

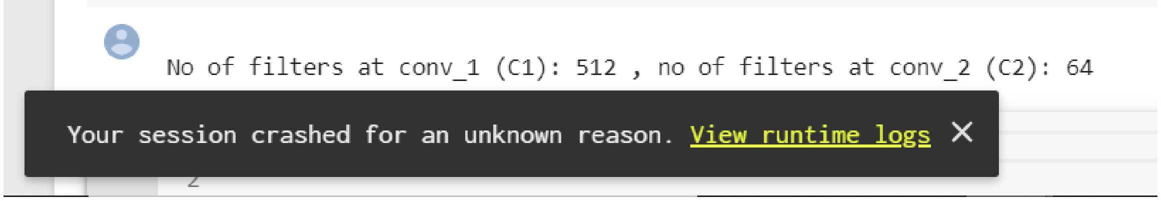


Question 2

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.

For efficiency, we limit the search space to *[32, 64, 128, 256]* for the grid search. This means that we try different combinations of number of feature maps in the search space at the two convolutional layers.

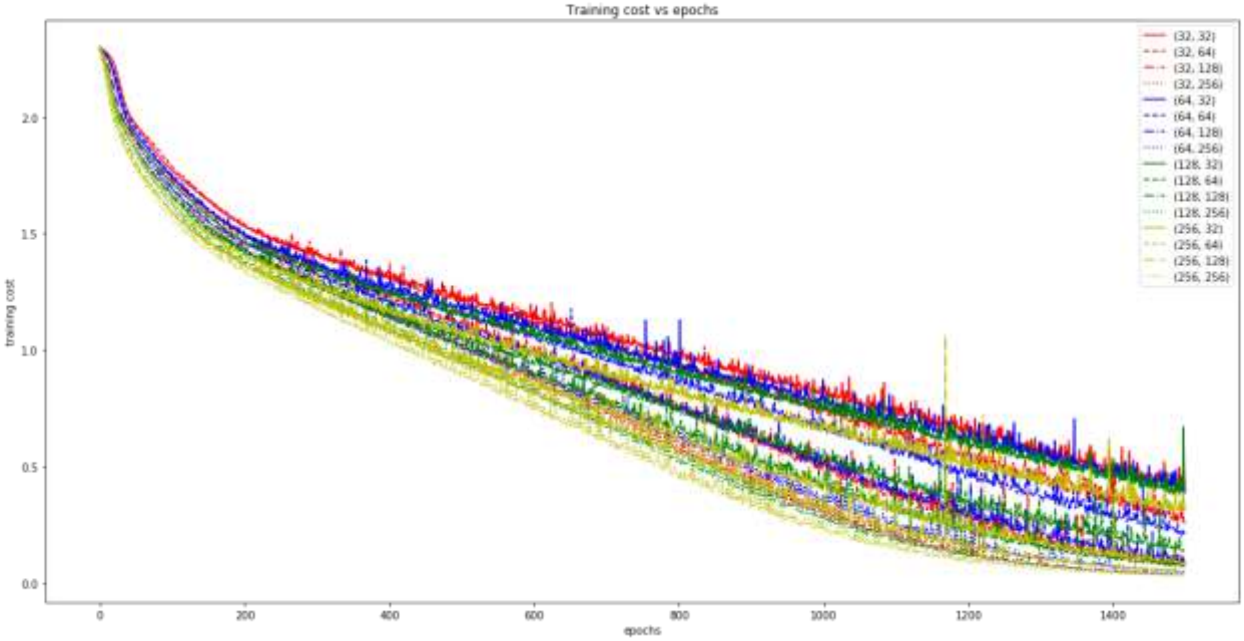
We also attempted to use 512 feature maps for the grid search. However, as we are using the GPU provided by Google Colab, we faced the difficulty that when we tried to train the model, the Colab session crashed as shown below.



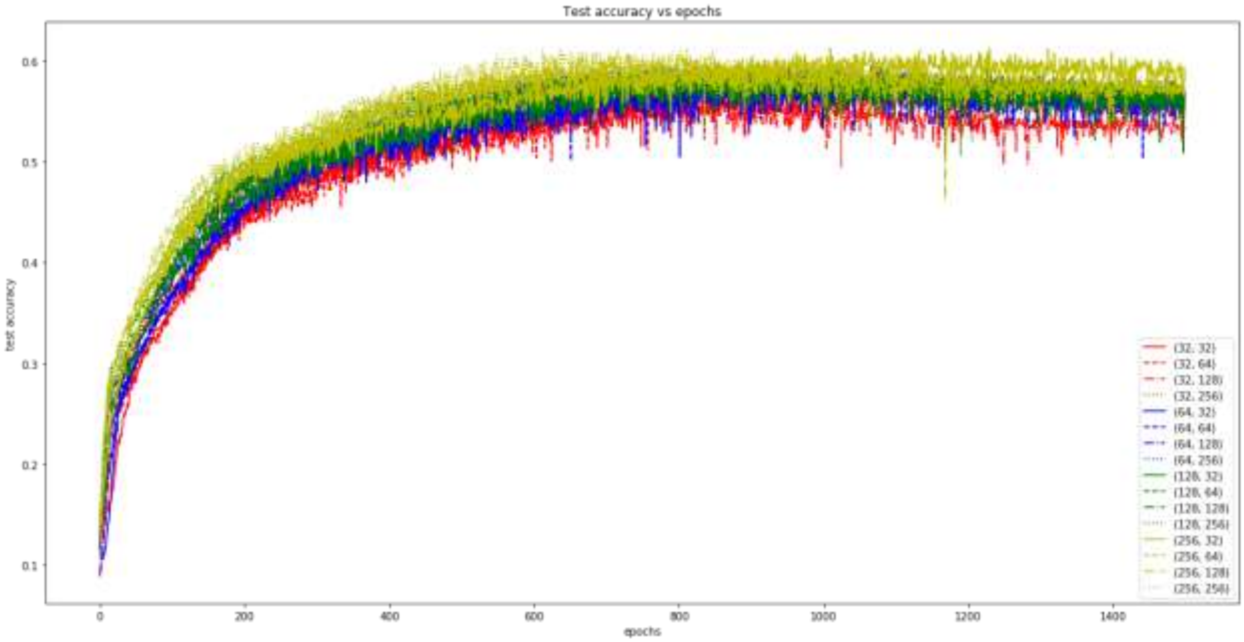
```
No of filters at conv_1 (C1): 512 , no of filters at conv_2 (C2): 64
```

```
Your session crashed for an unknown reason. View runtime logs
```

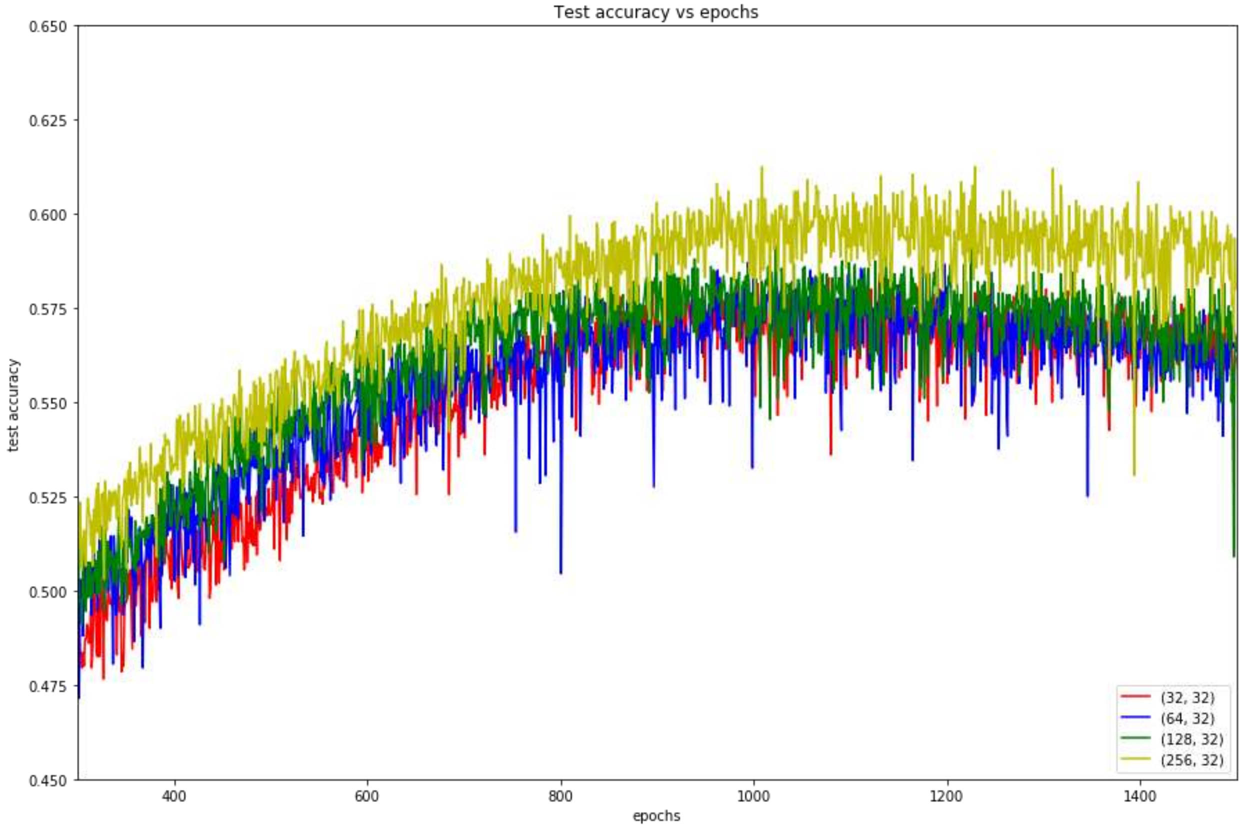
Nevertheless, with the above defined search space, we conducted 16 experiments with different combinations of the number of filters at *conv_1* and *conv_2*. The detailed training log can be found in the notebook attached. The consolidated results are as follows:



The above plot shows the training cost against epochs for the 16 models.

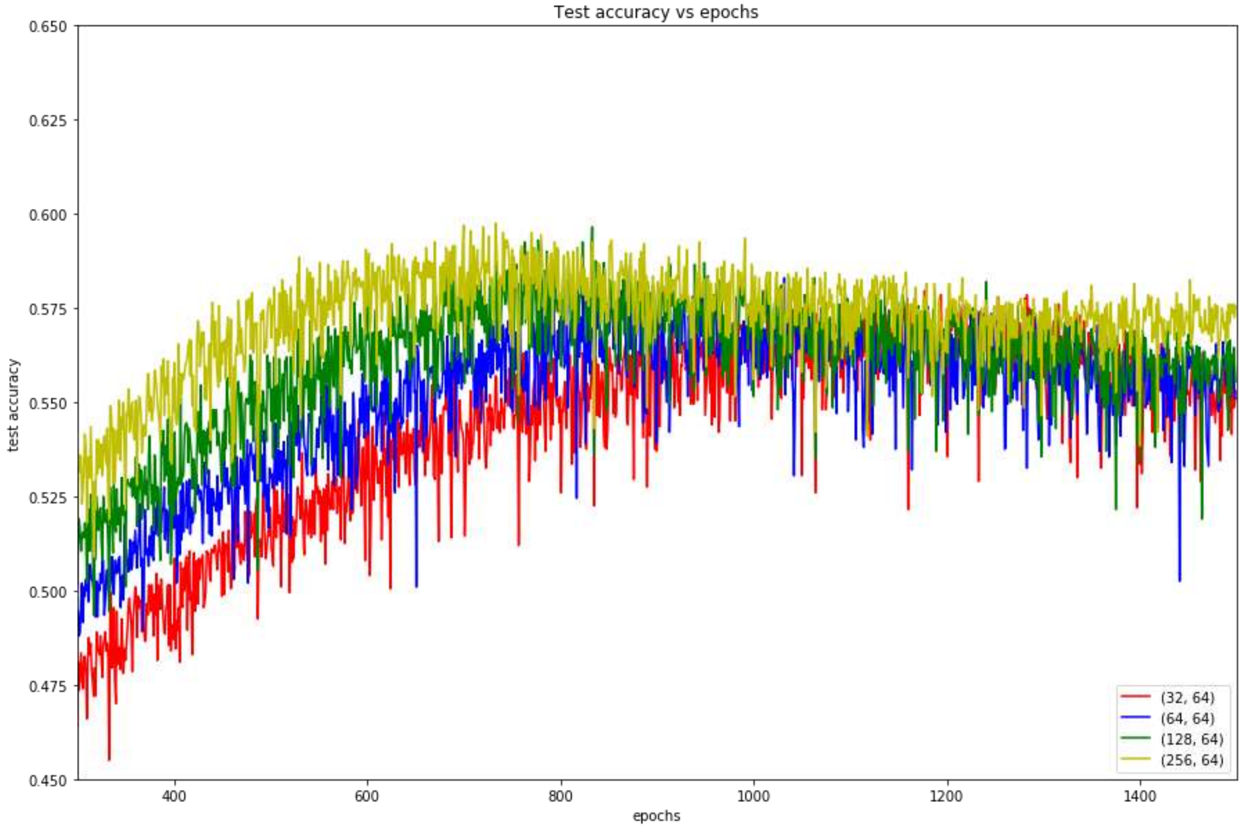


The above plot shows the test accuracies against epochs for the 16 models. However, it is hard to see which model performs the best. As such, we plotted the models separately as 4 groups to compare the relative performance.



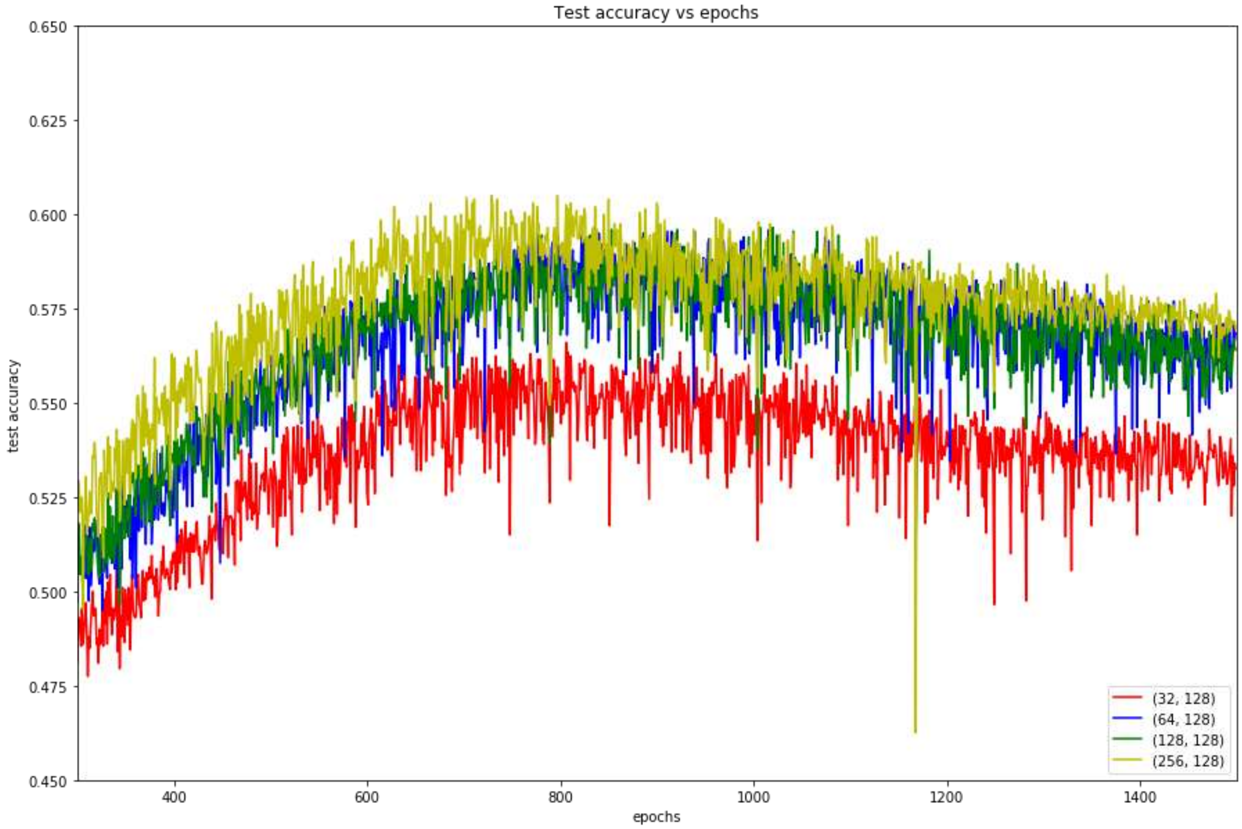
In the above plot, among models with 32 filters in *conv_2*, the yellow curve performs the best with the highest test accuracy all along. Thus, the parameters (256, 32) is used in the final comparison.

We can also see that, generally, as the number of filters increases, the test accuracy also increases. This is probably because more filters / feature maps are able to capture the complex structure in the images better. For example, more feature maps can capture a variety of different edges and other image features, which can help in the prediction of the labels.



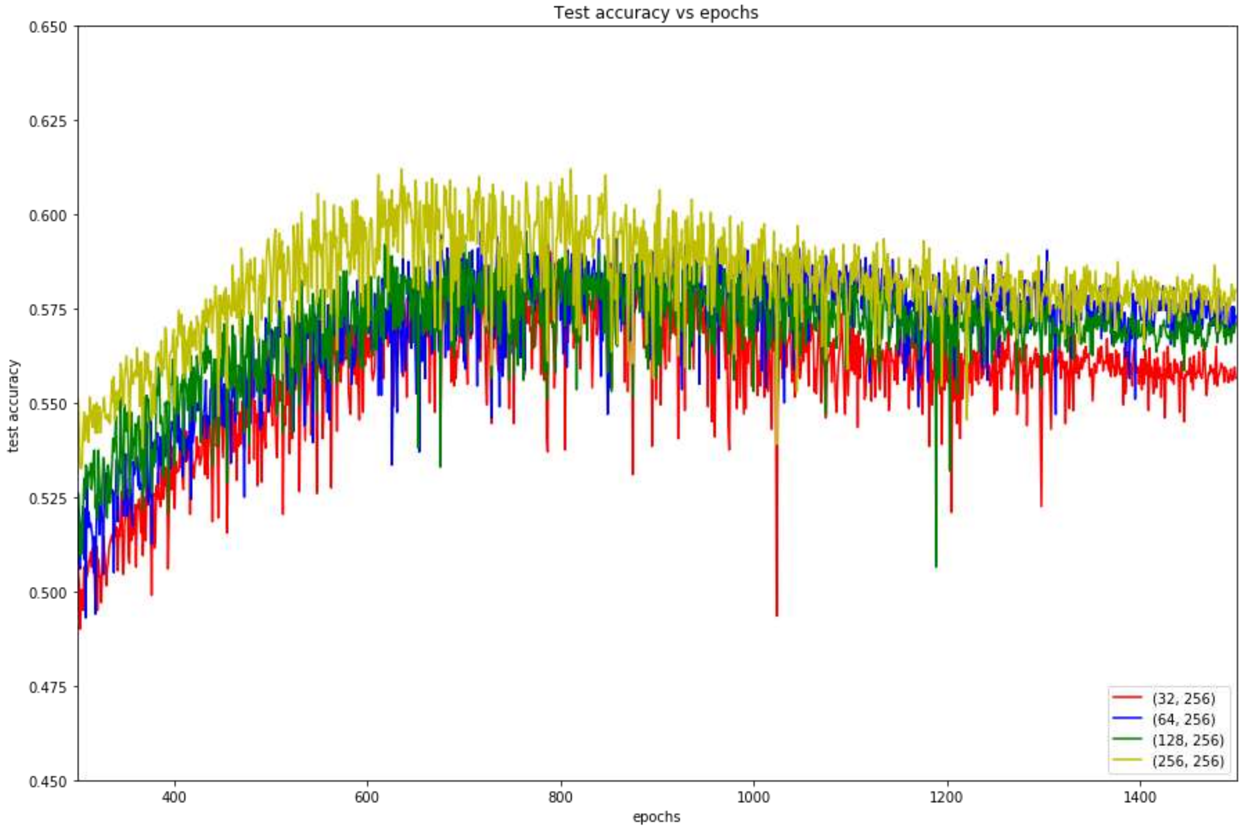
In the above plot, among models with 64 filters in *conv_2*, the yellow curve performs the best with the highest test accuracy. Thus, the parameters (256, 64) is used in the final comparison.

Our previous observation still holds - as the number of filters increases, the test accuracy generally also increases.



In the above plot, among models with 128 filters in *conv_2*, still, the yellow curve performs the best with the highest test accuracy. Thus, the parameters (256, 128) is used in the final comparison.

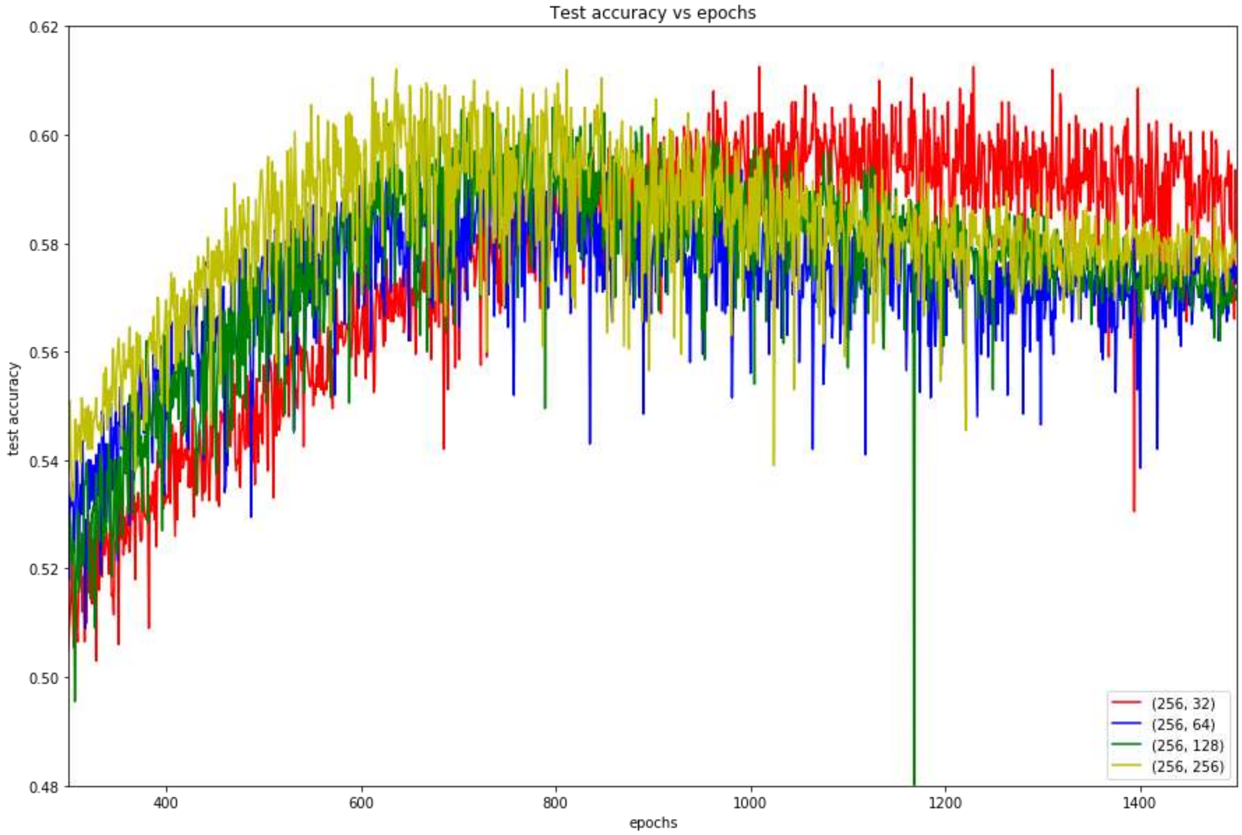
Our previous observation still holds - as the number of filters increases, the test accuracy generally also increases.



In the above plot, among models with 256 filters in *conv_2*, still, the yellow curve performs the best with the highest test accuracy. Thus, the parameters (256, 256) is used in the final comparison.

Our previous observation still holds - as the number of filters increases, the test accuracy generally also increases.

After selecting the 4 best models, we then compared the performance of these 4 models.



In the above plot, among the 4 best models, we can see that the red and yellow curves are better than the other two because they have higher test accuracies. However, both of them can reach similar test accuracies, and hence we need to consider some other factors in making a decision.

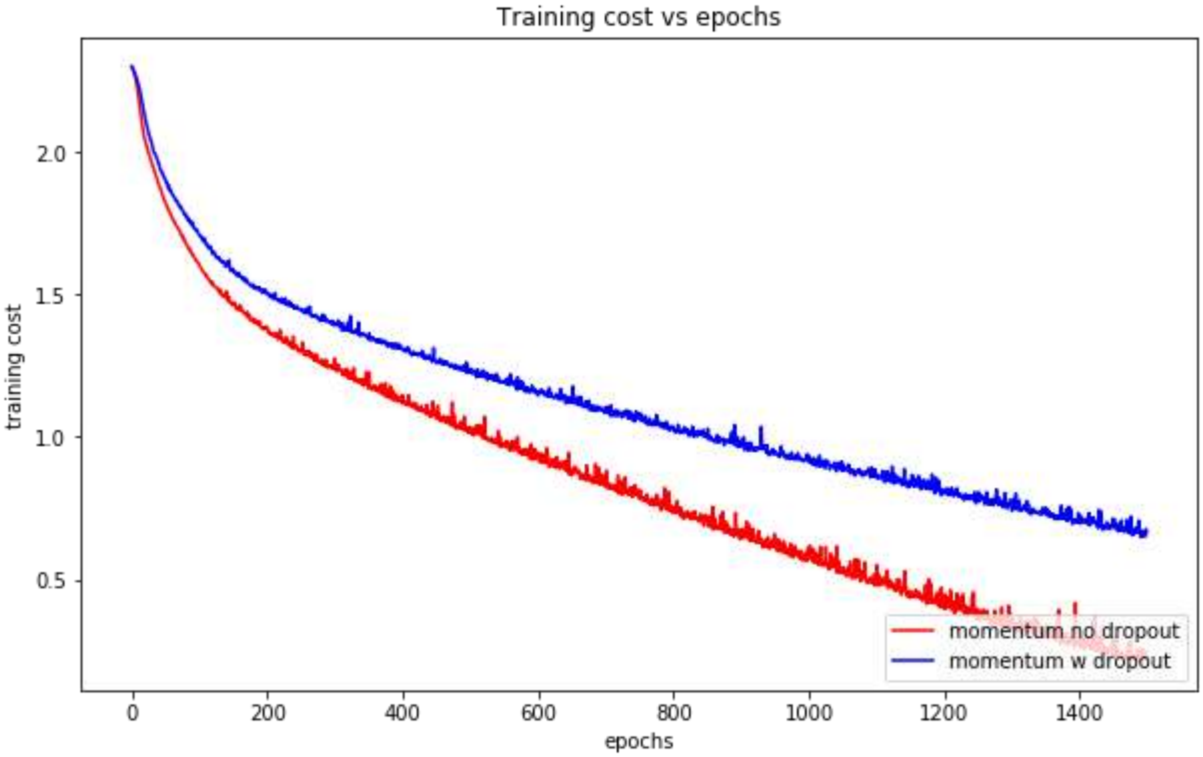
As a rule of thumb, typically we choose the simpler model if the performance is similar. This is an attempt to reduce the effect of over-fitting. In fact, the yellow curve starts to drop at about 700 epochs, whereas the red curve seems stable until over 1200 epochs. This might be attributed to the fact that with more filters, more features can be captured by the network, and hence the network is more likely to fit the training images and also over-fit the training images at earlier epochs.

Therefore, we choose the parameter set (256, 32), meaning that we choose 256 filters in the first convolutional layer and 32 filters in the second convolutional layer as the optimal.

Question 3

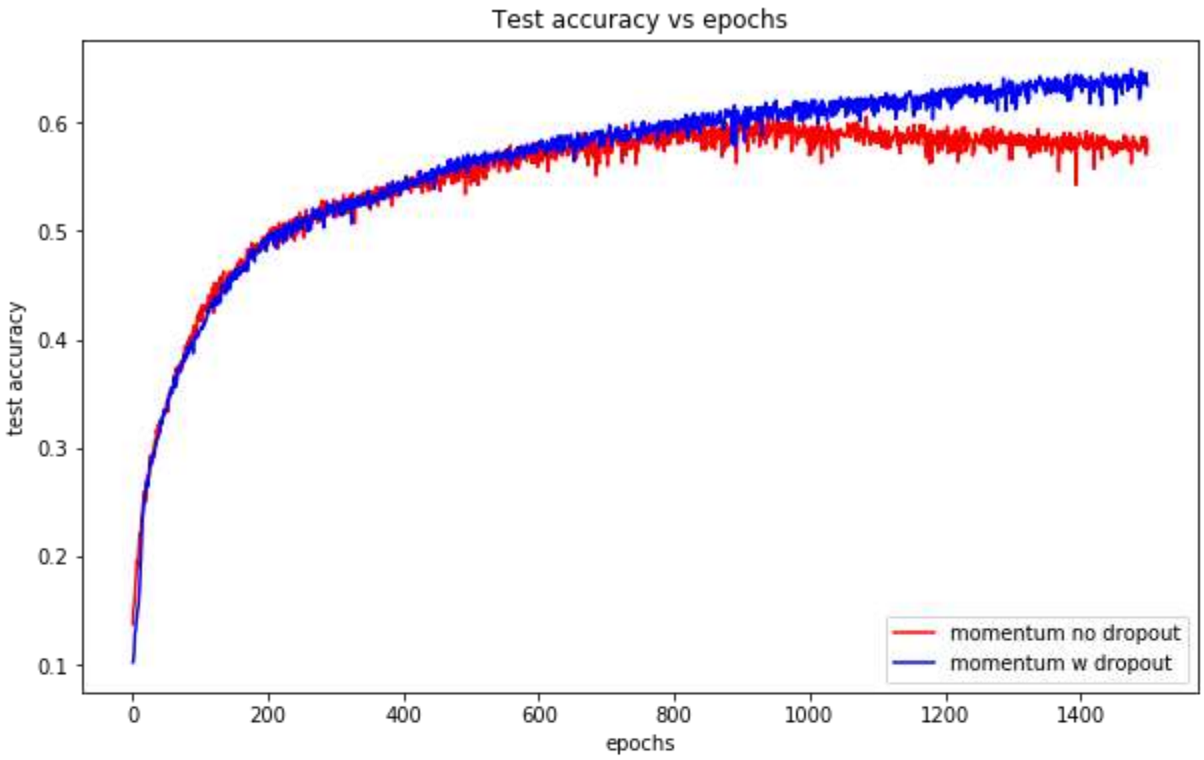
a) Adding the momentum term with momentum term = 0.1.

To add a momentum term, we changed our optimiser to *tf.train.MomentumOptimizer*. We also attempted to add dropout between the fully connected layer and the softmax output layer, and compared the training performance with and without the dropout.



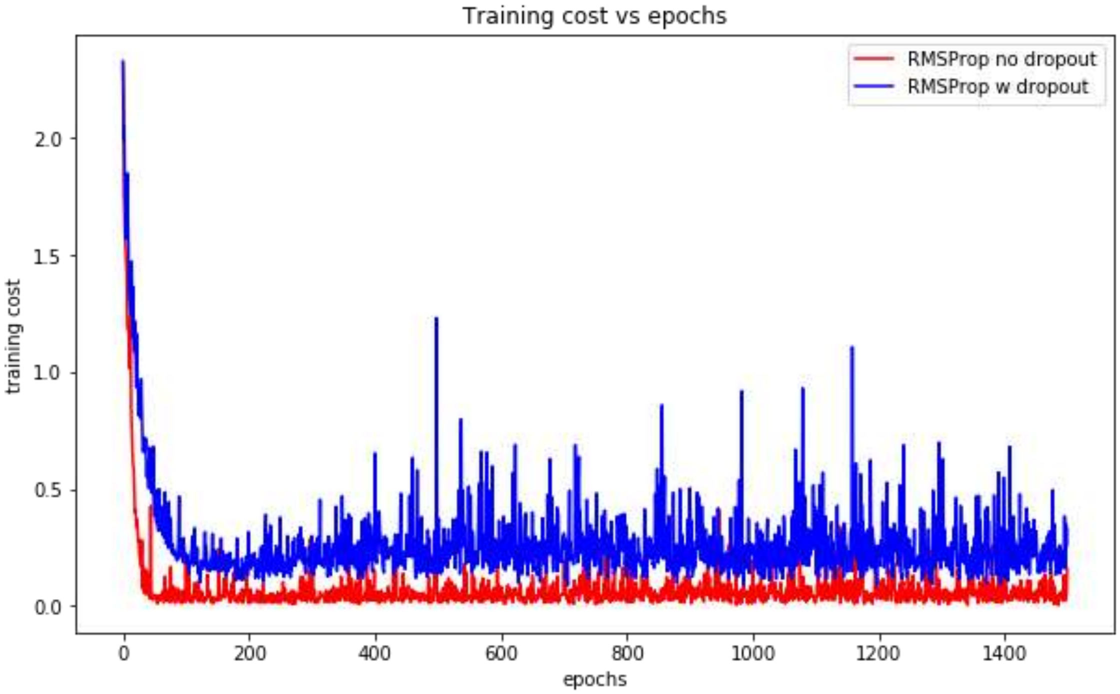
The above plot shows the training cost against epochs. It is clear that with dropout, the training cost decreases much slower than without dropout. This is expected because dropout prevents a layer from relying on a few strong inputs. This might cause the training to take longer because the layer cannot always make use of its "strong predictor" inputs. However, because of dropout, the layer is forced to use all of its inputs, improving generalization[3].

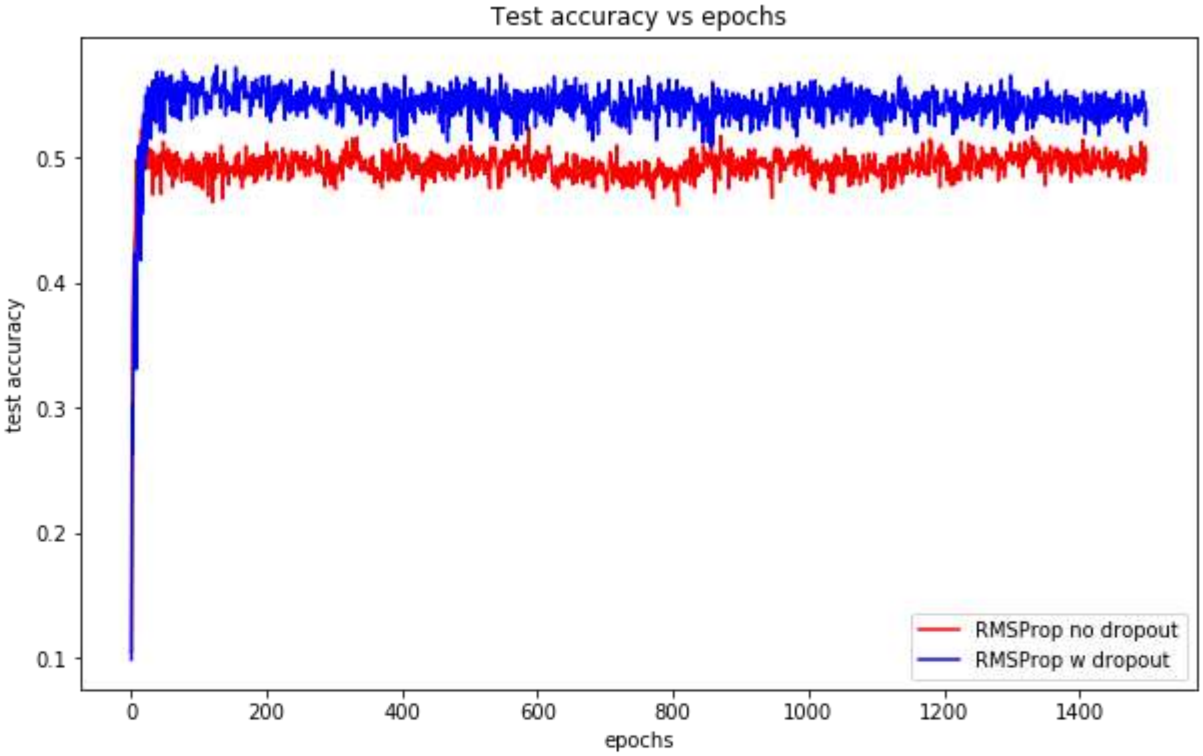
[3] https://stats.stackexchange.com/questions/374742/does-dropout-regularization-prevent-overfitting-due-to-too-many-iterations



b) Using RMSProp algorithm for learning

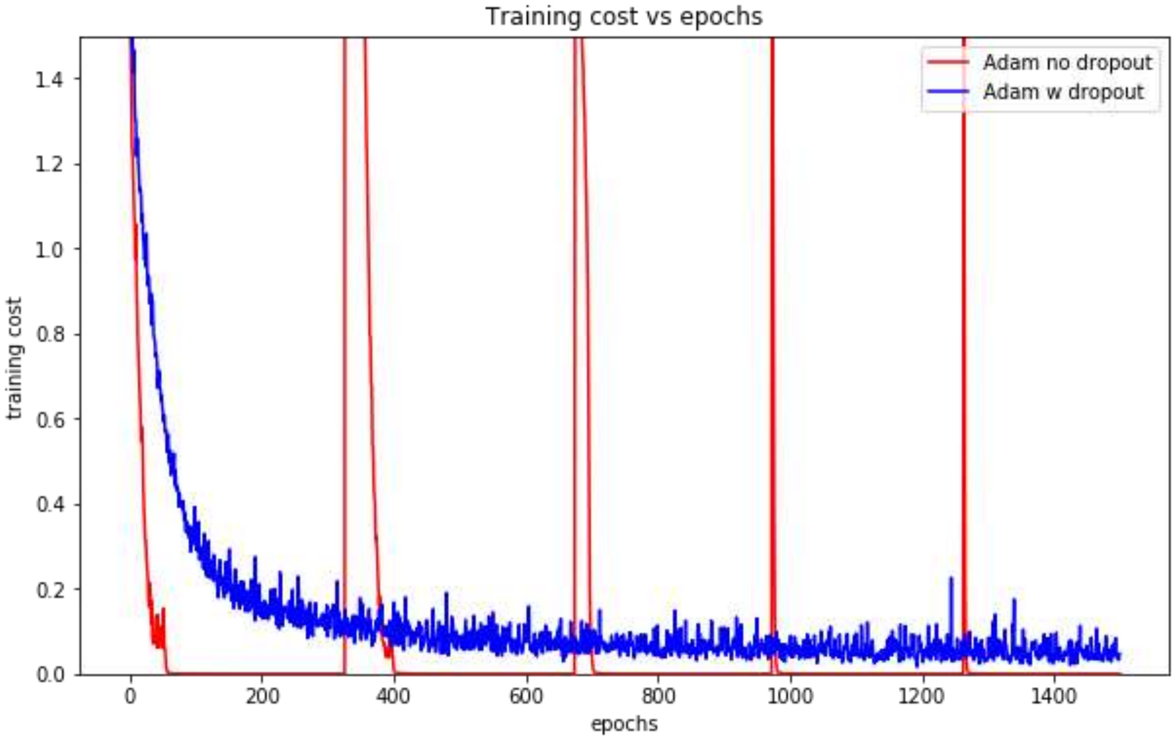
Here, we changed our optimiser to *tf.train.RMSPropOptimizer*. We also attempted to add dropout between the fully connected layer and the softmax output layer, and compared the training performance with and without the dropout.

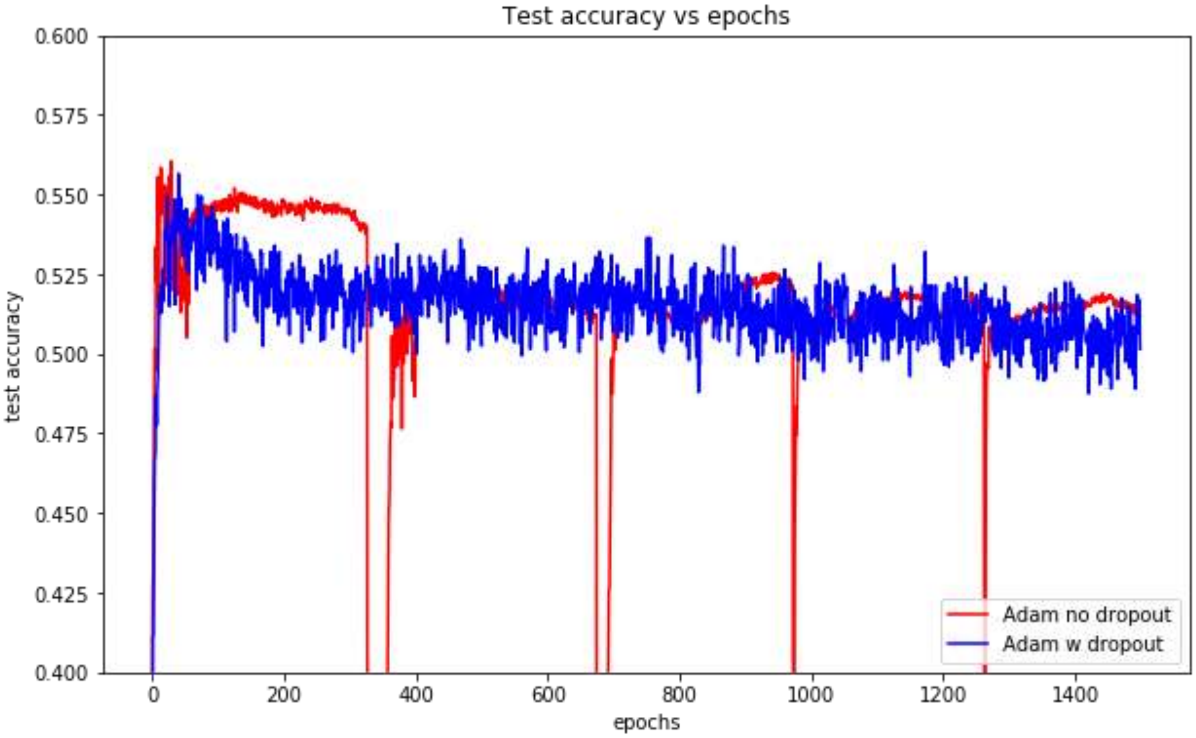



Test accuracy vs epochs
test accuracy
0.1
0.2
0.3
0.4
0.5
epochs
0
200
400
600
800
1000
1200
1400
RMSProp no dropout
RMSProp w dropout

c) Using Adam optimizer for learning

Here, we changed our optimiser to *tf.train.AdamOptimizer*. We also attempted to add dropout between the fully connected layer and the softmax output layer, and compared the training performance with and without the dropout.





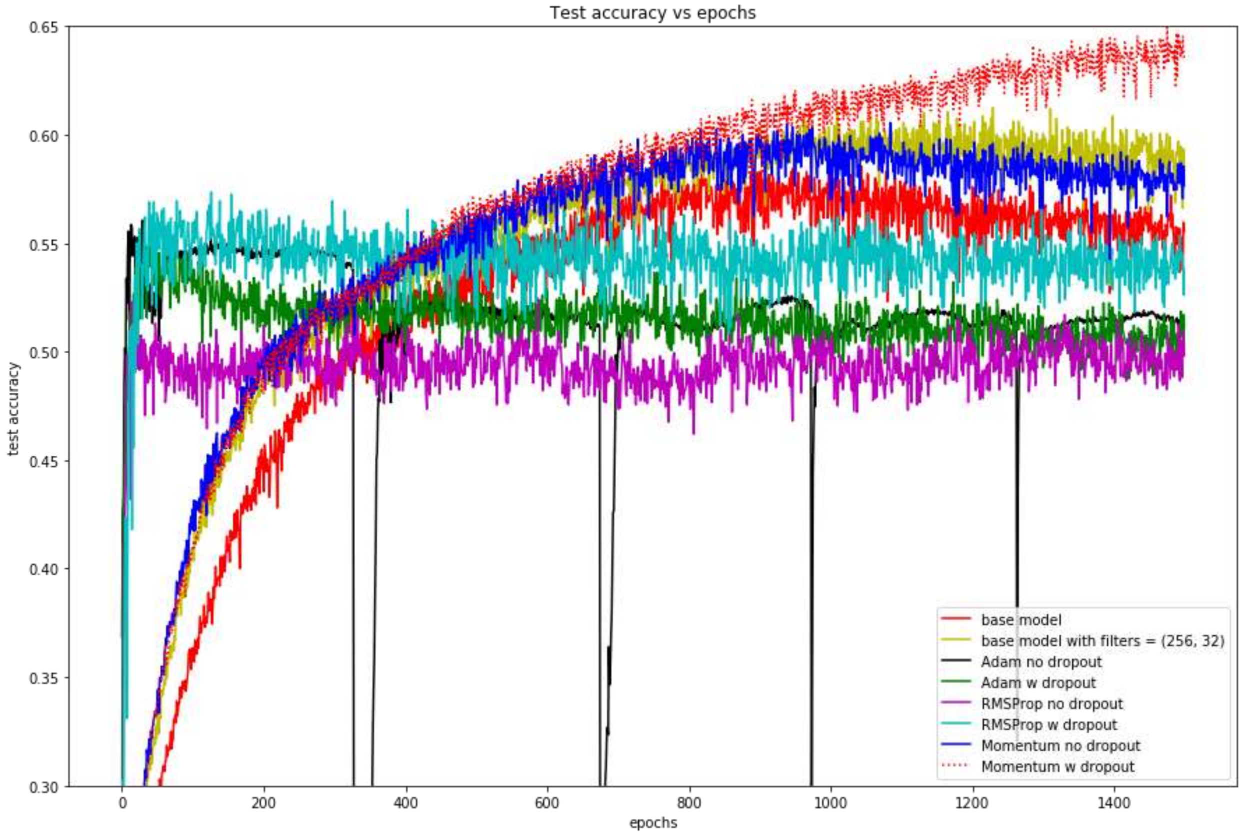
d) Adding dropout to the layers

This part is integrated into part (a), (b), (c) above. Please refer to the previous parts for the plots.

Question 4

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

We saved all the test accuracies as *.npy* files and plotted them on the same graph to compare their performances.



In the above plot, we plotted a total of 8 models from the previous sections. There is 1 model from question 1, 1 model from question 2 and 6 models from question 3.

Note that for question 2, we only plotted the final chosen model, because we do not want to make the plot so messy by plotting all 16 grid search models. For the comparison and discussion about the performances of these 16 models, please refer to question 2 above.

Effect of dropout

From the plot, we can see that with dropout, the models generally performs better. This better performance is evident in three ways.

Firstly, dropout reduces the degree of over-fitting. Comparing between momentum with and without dropout (the dotted red curve and the blue curve), the model without dropout starts to over-fit after 1000 epochs, and test accuracy starts to drop. However, with dropout, the accuracy continues to improve even at epoch 1500 and no over-fitting is observed yet.

Secondly, comparing between Adam with and without dropout (the green curve and the black curve), dropout significantly reduces the fluctuations in the test accuracy. While the black curve has a number of sharp “cliff” in it, the green curve is much smoother.

Thirdly, comparing between RMSProp with and without dropout (the cyan curve and the purple curve), dropout simply improves the test accuracy. As mentioned in the previous question, because of dropout, the layer is forced to use all of its inputs and train all the weights properly, improving generalisation. As a result, the performance on test dataset can be improved.

Effect of optimiser

From the plot, we can see that different optimisers have quite different performances on the given dataset. The performance can be evaluated in three dimensions - the efficiency in improving test accuracy quickly, the ability to achieve high test accuracy, as well as the speed of training.

For the first aspect, Adam and RMSProp (the black, green, purple, cyan curves) perform much better than gradient descent and gradient descent with momentum. This can be seen from their sharp rise in test accuracies in the first few epochs. This could be attributed to the design of these two optimisers to use adaptive learning rate, such that they can “take big steps” at the early stage of the training to drive down the training cost and improve the test accuracy quickly.

The second aspect is related to the maximum test accuracy achieved by the different optimisers. Here, gradient descent and gradient descent with momentum (the red, yellow, blue, dotted red curves) perform better because their best accuracies are higher than the rest. This could be due to the small learning rate present in these four models, such that they are able to slowly step to the optimal point. The Adam and RMSProp optimisers, on the other hand, eventually lose to the simpler optimisers, despite having a very good starting performance.

The last aspect is not evident in the graph, but rather observed during our training process. While all the other models take similar time to train, RMSProp models take particularly longer time to finish training. This could be related to the implementation of RMSProp by TensorFlow, such that there are actually more computations to be done.

The best model

From the above plot, the best model from our empirical experiments is the gradient descent model with momentum and dropout (red dotted curve). While all other models reach a maximum of 60% accuracy, the gradient descent with momentum and dropout model is able to achieve 65% accuracy in 1500 epochs. Since there is no over-fitting observed yet, we can actually continue training and the test accuracy might be even higher.

Conclusion

Answers to the queries

Answers are elaborated in the above Experiments and Results section.

Summary of findings

After several experiments, we realised that there are quite a number of hyper-parameters which can affect the training results of a convolutional neural network. For example:

- The number of filters in convolutional layers affects how well the model can learn the features in the images. Generally, the higher the number of filters, the better the model can learn the features and reduce the training cost. However, if there are too many filters, the test accuracy may not improve even though the training cost decreases. This is when over-fitting occurs.
- The type of optimiser affects the training process as well as the training results. Some optimisers can achieve a good performance within a few epochs, while others take a long time to train. Some optimisers can arrive at a better result eventually, while others might get stuck at a local optimal point.
- Adding dropout can improve the model performance on test dataset, because it helps the model to be more generalised.

Experience of experiments

Apart from the above-mentioned, there are a lot of other hyper-parameters in neural networks. Although there are some collated pros and cons for each of the hyper-parameter choices based on experience, the final training results are still largely data-dependent. Hence, a large number of experiments are required to really fine-tune a model. High computational power is required as well.

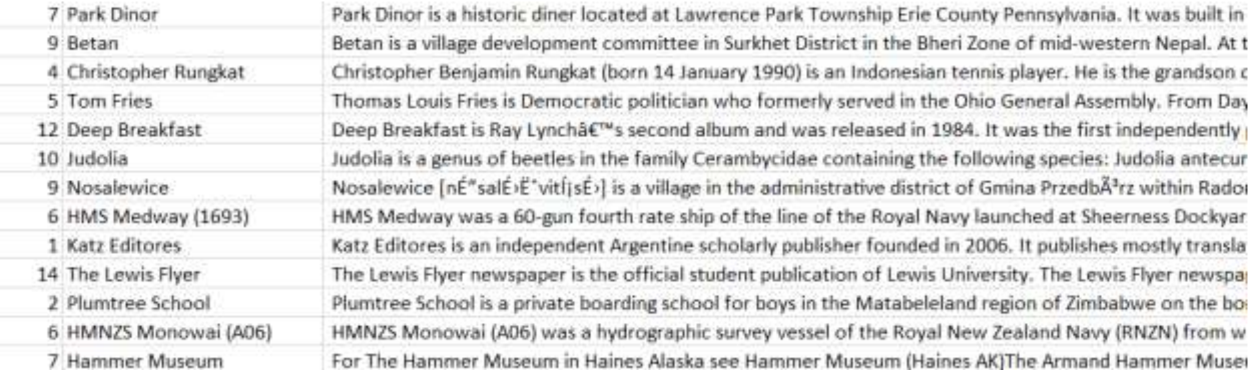
Part B: Text Classification

Introduction

In this project, we are required to perform text classification given some paragraphs and their correspondings labels about their category. We are required to perform this task with the help of convolutional neural networks and recurrent neural networks. Specifically, we will build character and word classifiers, for both convolutional neural networks and recurrent neural networks. The output layer of the networks is a softmax layer.

Dataset

The dataset used in this project contains the first paragraphs collected from Wikipage entries and the corresponding labels about their categories. The training and test datasets are given in ‘*train_medium.csv’* and ‘*test_medium.csv’* The training dataset contains 5600 entries while the testing dataset contains 700 entries. There are a total of 15 categories which is represented by the first column represents the labels for the dataset. The second column represents the inputs for the character classifiers, while the third column represents the inputs for the word classifiers. Below is a snapshot of the dataset.



7	Park Dinor	Park Dinor is a historic diner located at Lawrence Park Township Erie County Pennsylvania. It was built in
9	Betan	Betan is a village development committee in Surkhet District in the Bheri Zone of mid-western Nepal. At t
4	Christopher Rungkat	Christopher Benjamin Rungkat (born 14 January 1990) is an Indonesian tennis player. He is the grandson c
5	Tom Fries	Thomas Louis Fries is Democratic politician who formerly served in the Ohio General Assembly. From Day
12	Deep Breakfast	Deep Breakfast is Ray Lynchâ€™s second album and was released in 1984. It was the first independently
10	Judolia	Judolia is a genus of beetles in the family Cerambycidae containing the following species: Judolia antecur
9	Nosalewice	Nosalewice [nÉ”salÉ›Ë ˆvitÍ¡sÉ›] is a village in the administrative district of Gmina PrzedbÃ³rz within Rado
6	HMS Medway (1693)	HMS Medway was a 60-gun fourth rate ship of the line of the Royal Navy launched at Sheerness Dockyar
1	Katz Editores	Katz Editores is an independent Argentine scholarly publisher founded in 2006. It publishes mostly transla
14	The Lewis Flyer	The Lewis Flyer newspaper is the official student publication of Lewis University. The Lewis Flyer newspa
2	Plumtree School	Plumtree School is a private boarding school for boys in the Matabeleland region of Zimbabwe on the bo
6	HMNZS Monowai (A06)	HMNZS Monowai (A06) was a hydrographic survey vessel of the Royal New Zealand Navy (RNZN) from w
7	Hammer Museum	For The Hammer Museum in Haines Alaska see Hammer Museum (Haines AK)The Armand Hammer Muse

Method

Data Loading and Preprocessing

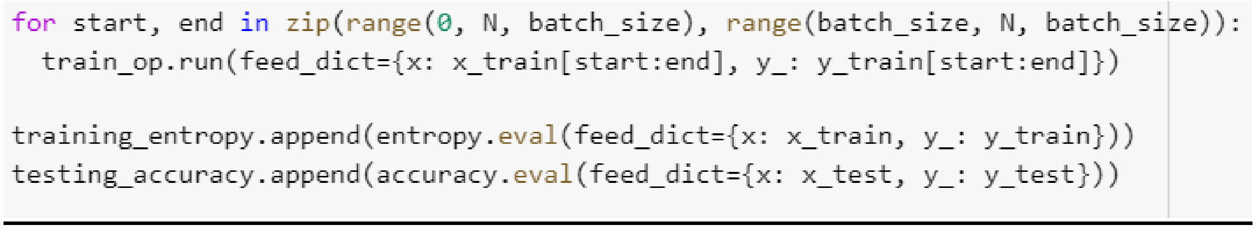
As mentioned in the previous section, the labels for the dataset is at the first column of the dataset.The second column represents the inputs for the character classifiers, while the third column represents the inputs for the word classifiers. Below is a snapshot of the dataset.

After reading in the csv files and getting the label and input from the respective columns,we need to convert the text input data to character IDs or word IDs to feed into the characters classifier and word classifier respectively. To do this, we make use of the following:

tf.contrib.learn.preprocessing.ByteProcessor : to convert text to character IDs

tf.contrib.learn.preprocessing.VocabularyProcessor : to convert text to word IDs

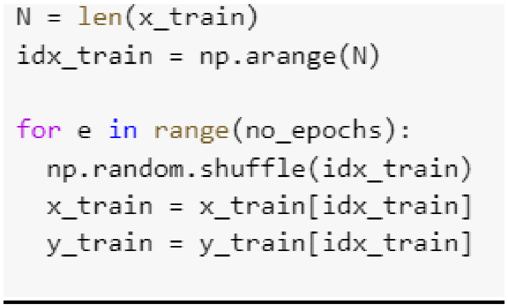
Mini-Batch Gradient Descent



```
for start, end in zip(range(0, N, batch_size), range(batch_size, N, batch_size)):
  train_op.run(feed_dict={x: x_train[start:end], y_: y_train[start:end]})

training_entropy.append(entropy.eval(feed_dict={x: x_train, y_: y_train}))
testing_accuracy.append(accuracy.eval(feed_dict={x: x_test, y_: y_test}))
```

Mini-batch gradient descent of batch 128 as specified in the question is used rather than stochastic gradient since it is faster.

Shuffling of Dataset at the beginning of each epoch



```
N = len(x_train)
idx_train = np.arange(N)

for e in range(no_epochs):
  np.random.shuffle(idx_train)
  x_train = x_train[idx_train]
  y_train = y_train[idx_train]
```

To avoid the risk of creating batches that are not representative of the overall dataset which may cause the estimation of gradient to be off, we shuffle our data at the beginning of each epoch.

Model Building

Character Convolutional Neural Network models:

To build a character convolutional neural network classifier, we first create a one-hot input layer, followed by the first convolutional layer, first pooling layer, second convolutional layer, second pooling layer before we flattened it and connect it to a fully connected-layer. The fully connected layer is then connected to a softmax output layer.

Word Convolutional Neural Network models:

To build a word convolutional neural network classifier, the input is first pass through an embedding layer to form word vectors which is reshaped before feeding into the CNN. This reshaped word vector is passed to the first convolutional layer, followed by first pooling layer, followed by second convolutional layer, followed by second pooling layer before we flattened it and connect it to a fully connected layer. The fully connected layer is then connected to a softmax output layer.

Character Recurrent Neural Network models:

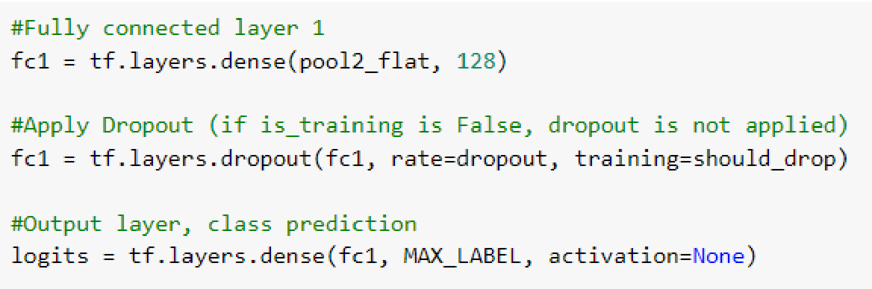
To build a character recurrent neural network classifier, we first convert the input into a one hot byte vector which is then unstacked into a byte list. Then, we select the type of RNN to use and passed the byte list inside so as to get the output of the last time-step of the RNN which is then connected to a softmax output layer.

Word Recurrent Neural Network models:

To build a word recurrent neural network classifier, the input is first pass through an embedding layer to form word vectors which is the unstacked into a word list. Then, we select the type of RNN to use and passed the word list inside so as to get the output of the last time-step of the RNN which is then connected to a softmax output layer.

Implementation of dropout at final layer

Even though we could work with both fully connected and convolutional layers, we followed the convention of adding dropout to the last layer of neural network and implement dropout only after the fully connected layer. This will reduce the complexity of the problem at hand.



```
#Fully connected layer 1
fc1 = tf.layers.dense(pool2_flat, 128)

#Apply Dropout (if is_training is False, dropout is not applied)
fc1 = tf.layers.dropout(fc1, rate=dropout, training=should_drop)

#Output layer, class prediction
logits = tf.layers.dense(fc1, MAX_LABEL, activation=None)
```

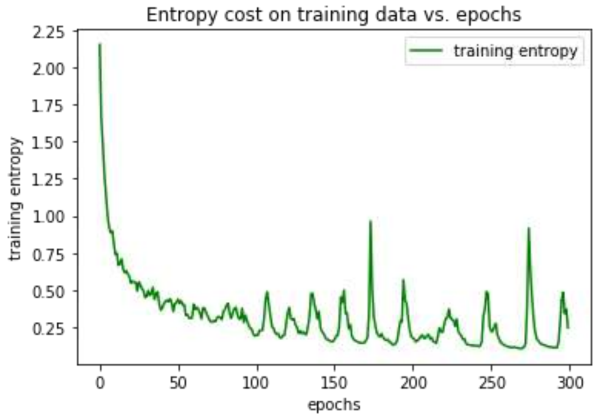
Experiments and Results

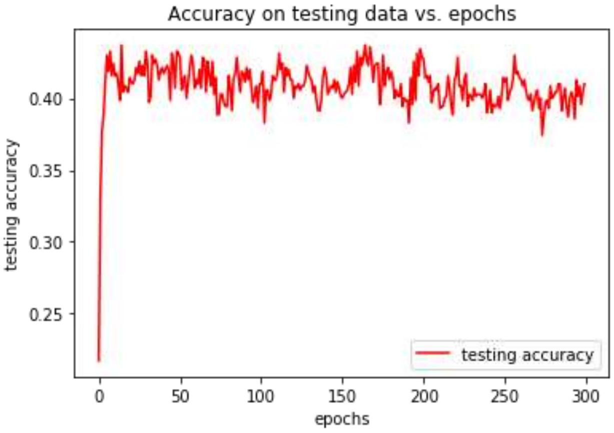
Question 1

Plot the entropy cost on the training data and the accuracy on the testing data against training epochs.

For the full code implementation of Question 1, please refer to *Part B Question 1 (Character CNN).ipynb.*

For this question, we have built a character CNN classifier that takes in character ids and classifies the input. The specification of the CNN classifier built fulfils the requirements of the questions. However, we have made a slight modification to it by adding a fully-connected layer between the flattened layer and output layer. Below are the plots required by the question.





The first plot shows the entropy cost on the training data against the training epochs, while the second plot shows the accuracy on the testing data against the training epochs,

From the first plot, we can see that the training entropy decreases steadily until the 80th epoch before it starts to fluctuate wildly.

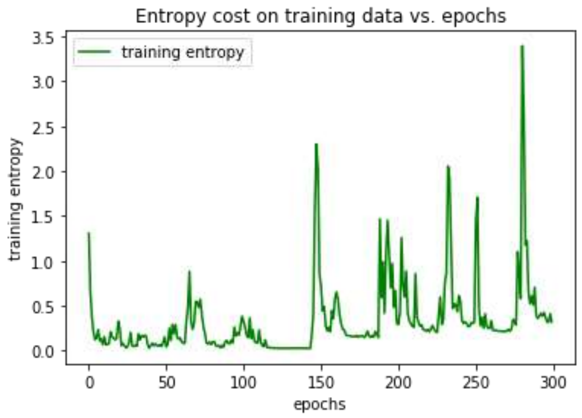
On the other hand, we can see that the testing accuracy increases sharply in the first few epochs before it starts to fluctuate at around 40%.

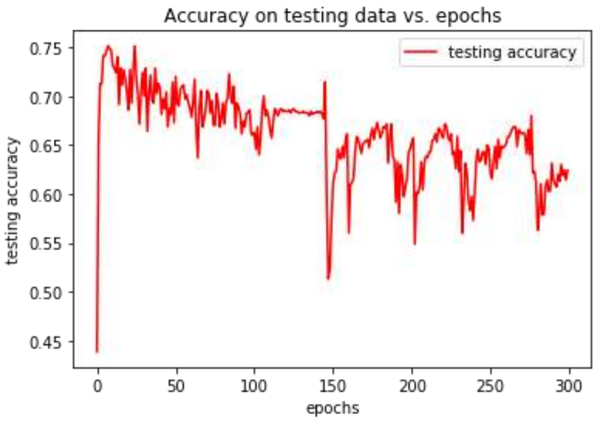
Question 2

Plot the entropy cost on the training data and the accuracy on the testing data against training epochs.

For the full code implementation of Question 2, please refer to *Part B Question 2 (Word CNN).ipynb.*

For this question, we have built a word CNN classifier that takes in the word ids and classifies the input. The inputs is passed in through an embedding layers of size 20 before feeding to the CNN. The specification of the CNN classifier built fulfils the requirements of the questions. However, we have made a slight modification to it by adding a fully-connected layer between the flattened layer and output layer. Below are the plots required by the question.





The first plot shows the entropy cost on the training data against the training epochs, while the second plot shows the accuracy on the testing data against the training epochs,

From the first plot, we can see that the training entropy decreases sharply in the first few epochs and maintained in the low region (<0.1) before it hits the 50th epoch. Beyond the 50th epoch, there is a sudden increase in the training entropy but it quickly decrease and remain low until around the 140th epoch. Beyond the 150th epoch, it could be observed that fluctuations are occurring more often and the training entropy is on an increasing trend.

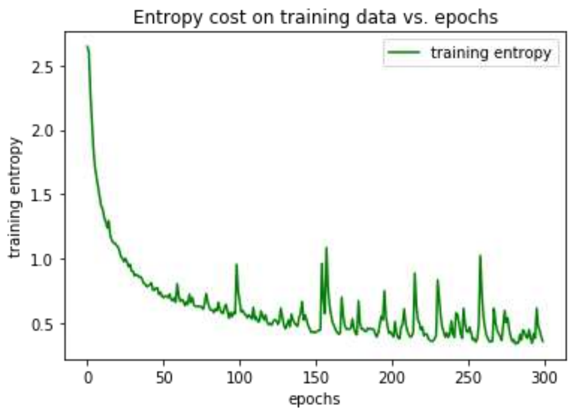
For the second plot, we can see the the testing accuracy increases sharply in the first few epochs. Beyond that, we can actually see a general decreasing trend with more epochs.

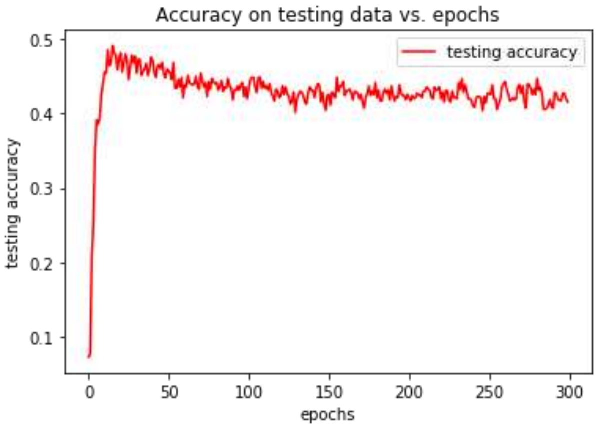
<u>Question 3</u>

Plot the entropy cost on training data and the accuracy on testing data against training epochs.

For the full code implementation of Question 3, please refer to *Part B Question 3 (Character RNN(GRU)).ipynb.*

For this question, we have built a character RNN classifier that takes in character ids and classifies the input. The specification of the RNN classifier built fulfils the requirements of the questions.





The first plot shows the entropy cost on the training data against the training epochs, while the second plot shows the accuracy on the testing data against the training epochs,

From the first plot, we can see that the training entropy decreases steadily until around the 150th epoch. Beyond the 150th epoch, it starts to fluctuate. However, we could still see a decreasing trend.

From the second plot, we can see that the testing accuracy increases sharply in the first few epochs before it starts to fluctuate mildly at around 42%.

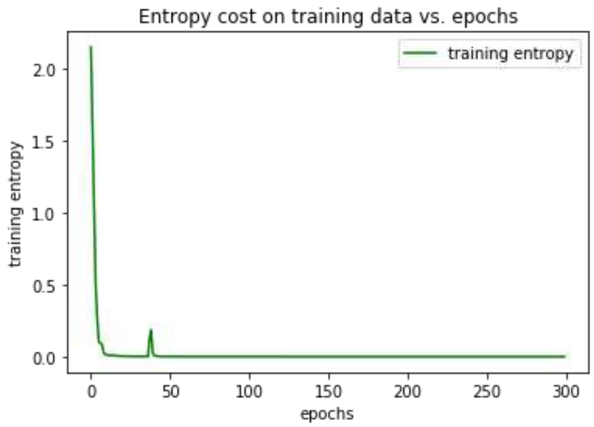
Perhaps due to that fact that both are character classifiers, the 2 plots in Question 3 looks somewhat similar to the 2 plots in question 1. However, we can see that the training entropy and testing accuracy against epochs in question 3 and fewer and milder fluctuation compared to that in question 1.

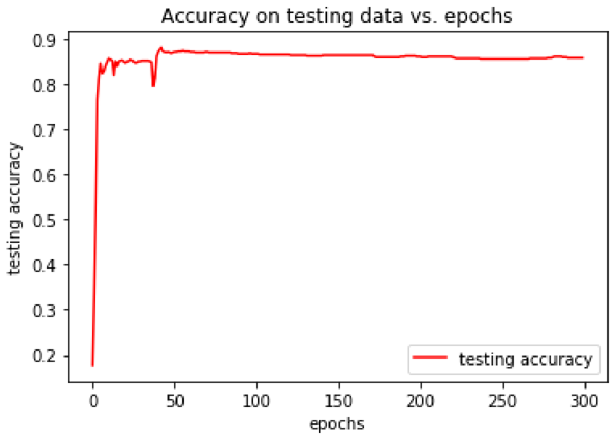
Question 4

Plot the entropy on training data and the accuracy on testing data versus training epochs.

For the full code implementation of Question 4, please refer to *Part B Question 4 (Word RNN(GRU)).ipynb*

For this question, we have built a word RNN classifier that takes in the word ids and classifies the input. The inputs is passed in through an embedding layers of size 20 before feeding to the RNN. The specification of the RNN classifier built fulfils the requirements of the questions.





The first plot shows the entropy cost on the training data against the training epochs, while the second plot shows the accuracy on the testing data against the training epochs,

From the first plot, we observe that the training entropy decreases sharply in the first few epochs and remained low throughout the 300 epochs,

From the second plot, we observe that the testing accuracy increase sharply in the first few epochs and remained high throughout the 300 epochs.

It is worthy to note that this model has the least fluctuation and is the most stable of all the models built so far. Unlike question 1 and 3 which looks somewhat similar (apart from the fluctuation), this model is a notch above the rest, showing the might of using embedding with GRU which is capable of remembering long sequence from the past. This could probably explain why the accuracy of the word CNN decreases with more epochs but the accuracy of this model still remain somewhat unchanged.

Question 5

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

Experiment with adding dropout to the layers of networks in parts (1) – (4), and report the test accuracies. Compare and comment on the accuracy of the networks with/without dropout.

For the full code implementation of Question 5, please refer to the following files:

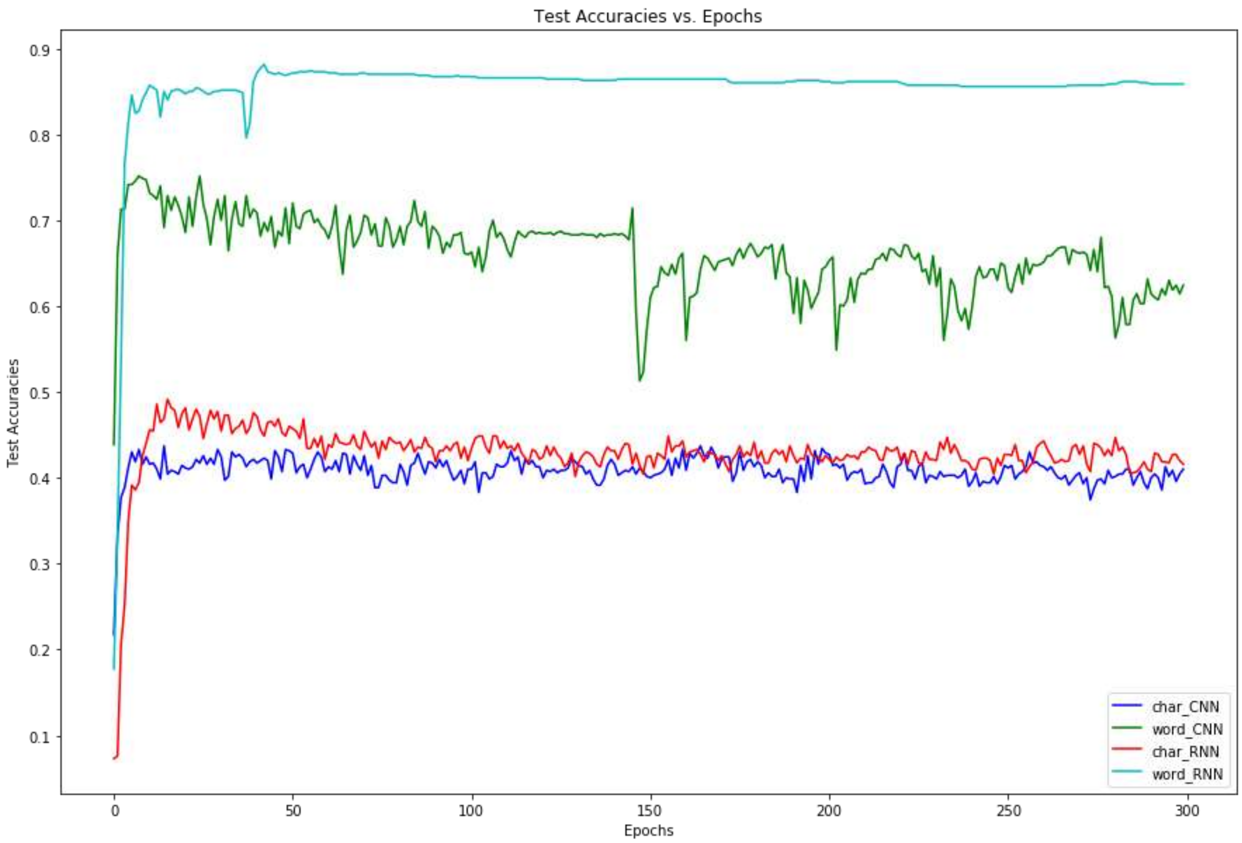
Part B Question 5a (Comparison of test accuracy and running time).ipynb
Part B Question 5b (Character CNN_with dropout).ipynb
Part B Question 5c (Word CNN_with dropout).ipynb
Part B Question 5d (Character RNN(GRU)_with dropout).ipynb
Part B Question 5e (Word RNN(GRU)_with dropout).ipynb
Part B Question 5f (Comparison of dropout vs non-dropout networks).ipynb

From Question 1 to 4, we have discovered the running time for 300 epochs for the different models.

1. **Character CNN**: 92.701461815 s
2. **Word CNN**: 35.63326258400002 s
3. **Character RNN (GRU)** : 339.3416120890006 s
4. **Word RNN (GRU)** : 576.2670882979999 s

From here, we can see that Word CNN took the shortest time to complete the run, followed by Character CNN and Character RNN, Word RNN took the longest time. From this example, we can conclude that RNN has longer running time than CNN.

Next, we will compare the accuracies for the 4 networks implemented.

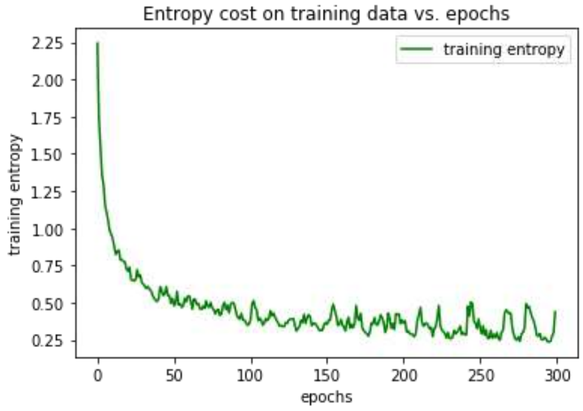


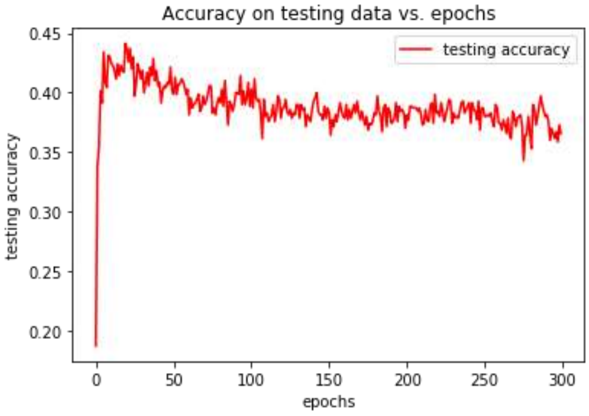
A few observations could be made:

1. For the word models, RNN model outperformed the CNN model.
2. For the character models, RNN model outperformed the CNN model.
3. The word models outperformed the character models.
4. Word RNN model is the most stable model.
5. Word CNN model is the most unstable model.
6. Accuracies from the best to the worst are: Word RNN, Word CNN, Character RNN, Character CNN.

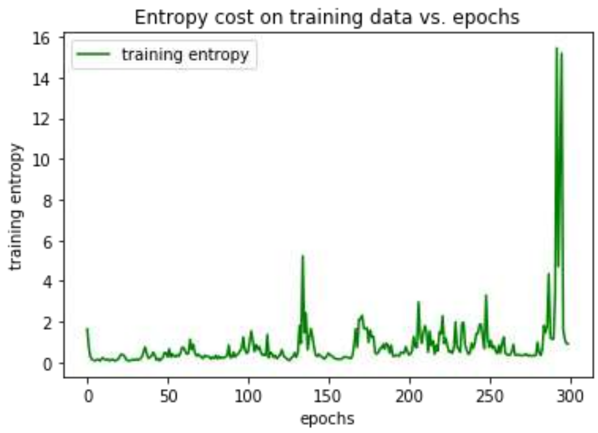
The following parts show our experimentations by adding to a dropout rate of 0.2 to the four models.

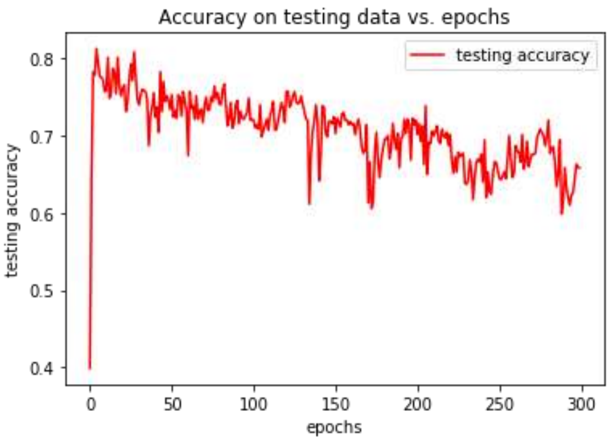
Character CNN model with dropout



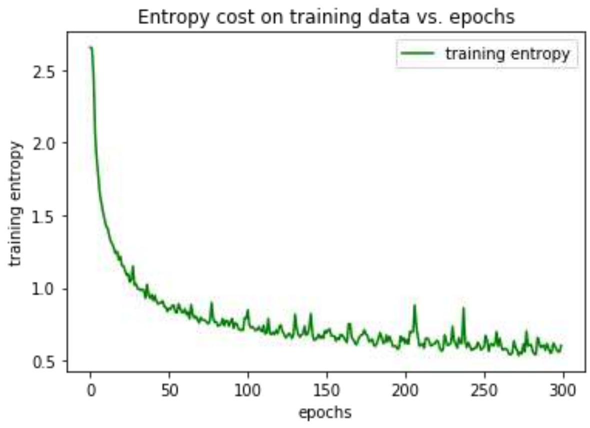


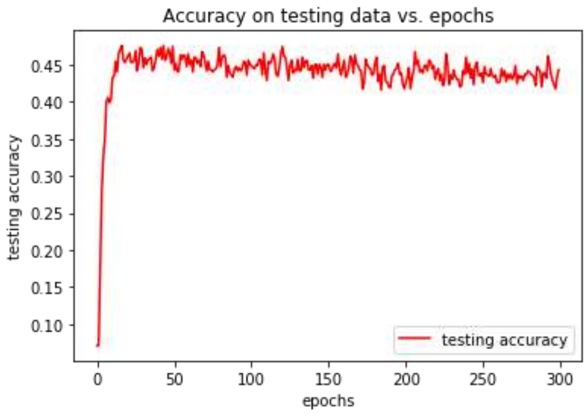
Word CNN model with dropout



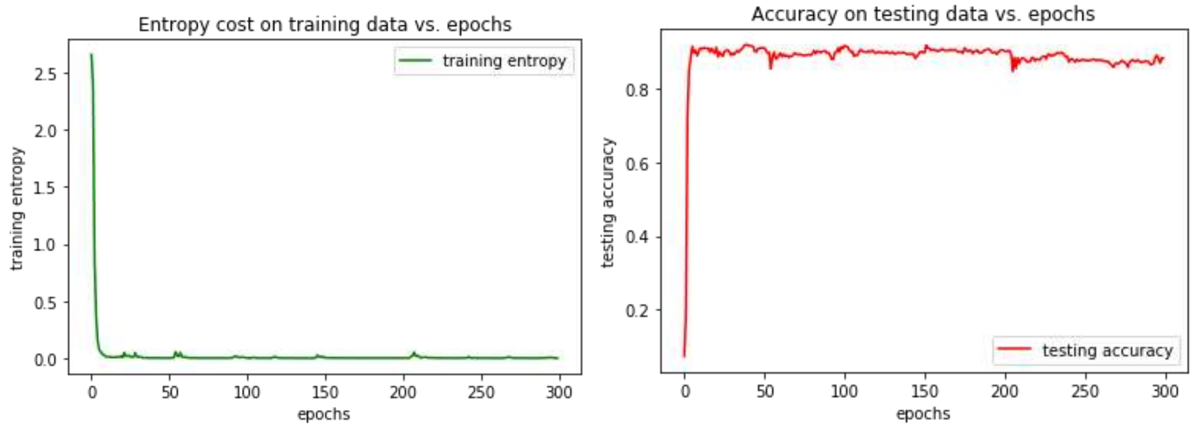


Character RNN model with dropout





Word RNN model with dropout

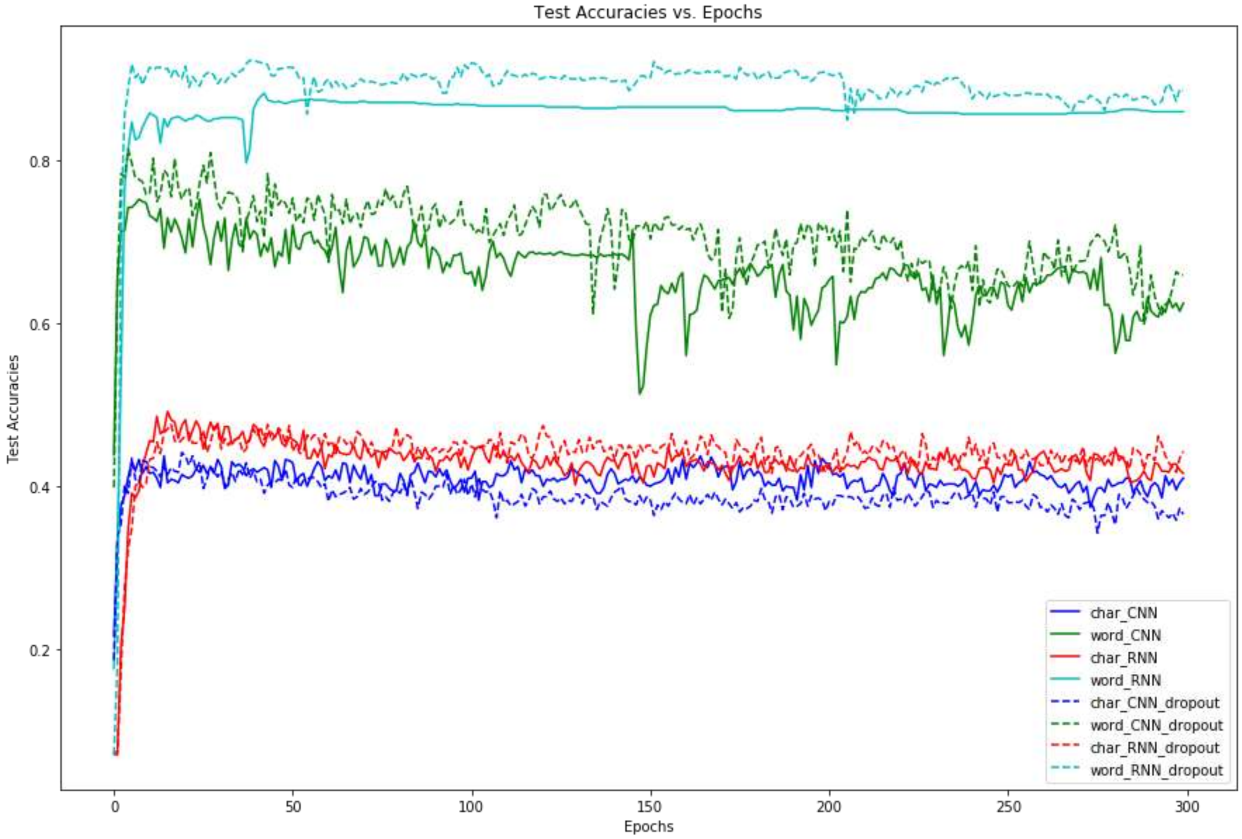


Next, we compare the running time and accuracy for the networks with and without dropout. We then plot them on a graph.

Running time for:

1. **Character CNN**:
   - Without dropout: 92.701461815 s
   - With dropout: 196.06870266300007
2. **Word CNN**:
   - Without dropout: 35.63326258400002 s
   - With dropout: 37.688433374 s
3. **Character RNN (GRU)** :
   - Without dropout: 339.3416120890006 s
   - With dropout: 635.086576849 s

4. **Word RNN (GRU)** :
   - Without dropout: 576.2670882979999 s
   - With dropout: 560.0884795449999



A few observations could be made:

1. For the word models, RNN model outperformed the CNN model.
2. For the character models, RNN model outperformed the CNN model.
3. The word models outperformed the character models.
4. Word RNN model is the most stable model.
5. Word CNN model is the most unstable model.
6. Accuracies from the best to the worst are: Word RNN, Word CNN, Character RNN, Character CNN.
7. **With the exception for the character CNN model, the dropout model will generally give us better test accuracies than the non-dropout model, with the exception for the character CNN model.**

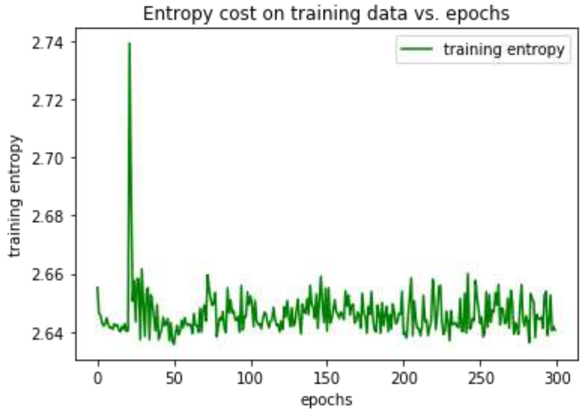
Question 6

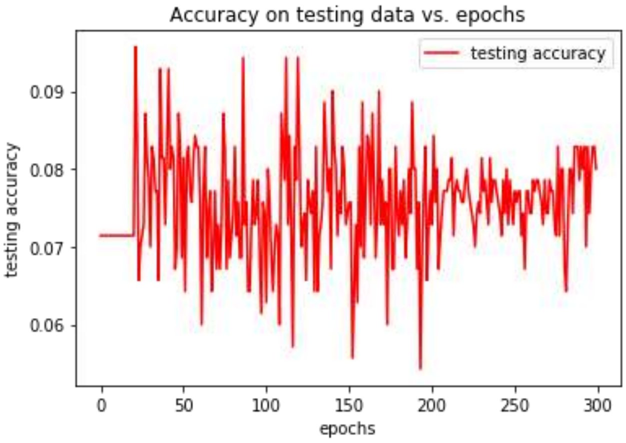
For RNN networks implemented in (3) and (4), perform the following experiments with the aim of improving performances, compare the accuracies and report your findings:

For the full code implementation of Question 6, please refer to the following files:

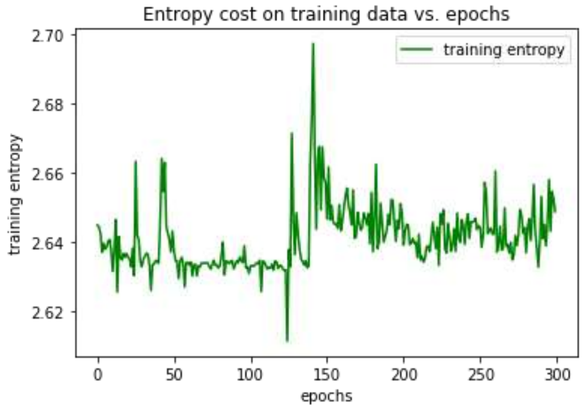
Part B Question 6 (Word RNN(Vanilla RNN)).ipynb
Part B Question 6 (Character RNN(Vanilla RNN)).ipynb
Part B Question 6 (Gradient Clipped Word RNN(GRU)).ipynb
Part B Question 6 (Gradient Clipped Character RNN(GRU)).ipynb
Part B Question 6 (Comparison across the different RNN networks).ipynb

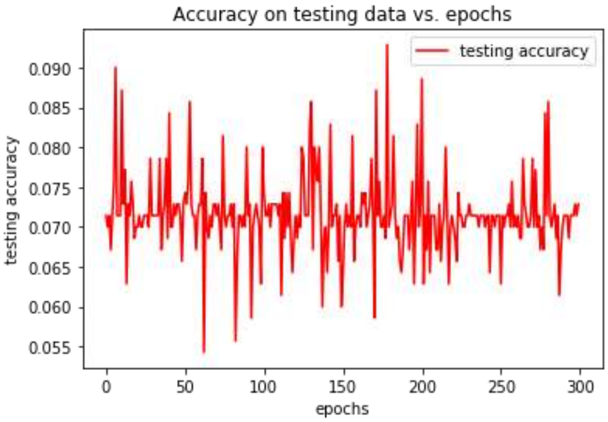
Character Vanilla RNN



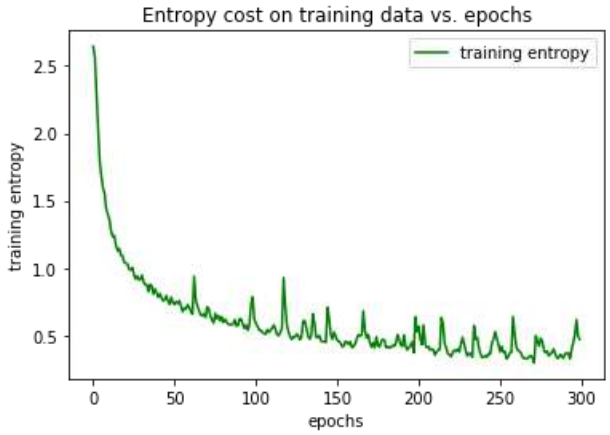


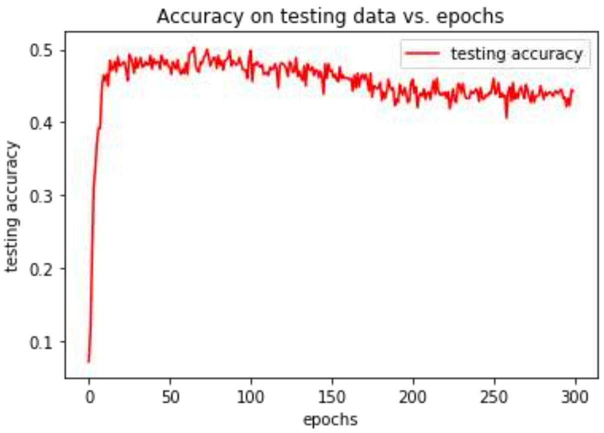
Word Vanilla RNN



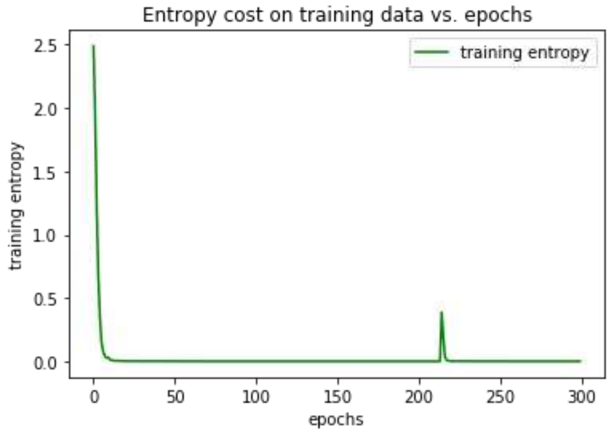


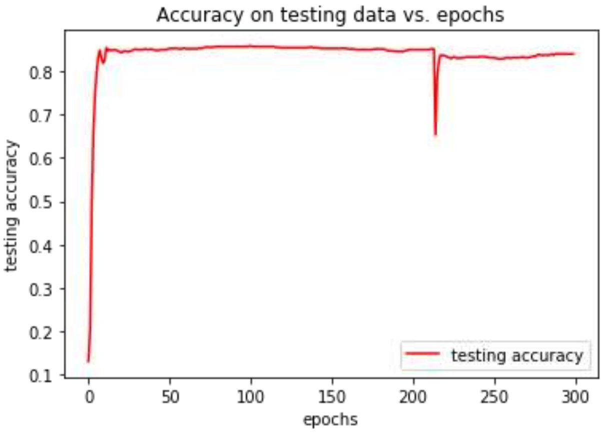
Gradient Clipped Character GRU


Entropy cost on training data vs. epochs

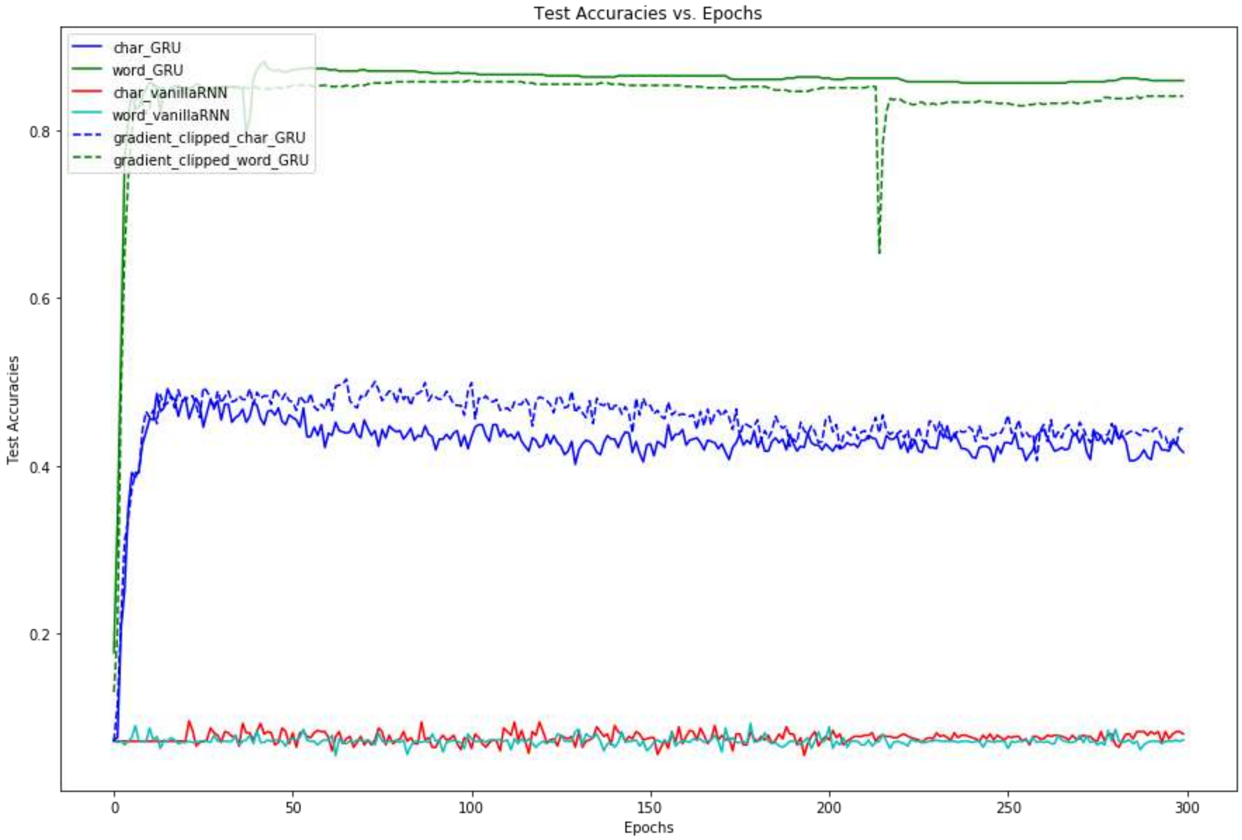

Accuracy on testing data vs. epochs

Gradient Clipped Word GRU


Entropy cost on training data vs. epochs


Accuracy on testing data vs. epochs

Comparison across the different RNN



A few observations could be made:

1. The accuracy of vanilla RNN is very low be it with the character or word
2. The accuracy of word GRU is better than character GRU be it gradient clipped or not.
3. The accuracy of word GRU is better than gradient clipped word GRU
4. The accuracy of gradient clipped character GRU is better than character GRU

Conclusion

Answers to the queries

Answers are elaborated in the above Experiments and Results section.

Summary of findings

After several experiments, we came out with some key observations:

- The word model is more accurate than character models. While we cannot explain why the word model is more accurate, it is our belief that embedding plays a key factor in the higher accuracy.
- Generally, RNN model is more accurate than CNN model for sequential data such as text.
- GRU and LSTM will give a better result compared to vanilla RNN model which is even worse than CNN model in terms of accuracy, This is because GRU and LSTM models can learn long term dependencies better than CNN model.
- Adding dropout can improve the model performance on test dataset, because it helps the model to be more generalised.

Interesting experience

Apart from the above-mentioned, we have also experimented with adding 2 fully connected layers to CNN models. We realised that more layers do not translate to better accuracy. From experiment, with 2 fully connected layers, the CNN models' accuracy increased to a point and has a sudden drop to near zero accuracy.