NANYANG TECHNOLOGICAL UNIVERSITY

CZ4042 Neural Networks
Project 2 Report

Chen Zhiwei (U1620741J)

Zeng Jinpo (U1620575J)

Table of Contents

Part A - Object Recognition

1. Introduction

The project aims at building convolutional neural networks to recognize the CIFAR object label from the inputs. The original dataset contains 60,000 32*32 colour images in 10 classes, with 6,000 images per class; and there are 50,000 training images and 10,000 test images. And the provided dataset is a subset of the original CIFAR-10 dataset, which consists of 10,000 training images and 2,000 test images.

2. Method

2.1 Data Pre-processing: Normalization of inputs

Initially, we scaled all input attributes of both train and test data into [0,1] by the following formula:

$$\tilde{x}_i = \frac{x_i - x_{i,min}}{x_{i,max} - x_{i,min}}$$

Here, the maximum and the minimum were only calculated over the training data, as the model would not know the test data in advance.

This scaling step was introduced to improve the model performance and convergence.

2.2 Model Development

For this assignment, we train the convolutional neural network to optimize the cross-entropy cost function. We had tried to determine the optimal hyper-parameters for the convolutional neural network by conducting exhaustive controlled experiments with grid search, whereby each time only one hyper-parameter is explored. The hyper-parameter we had experimented for the convolutional neural network are:

- Number of feature maps in convolutional layer C1,
- Number of feature maps in convolutional layer C2,
- Drop out

Moreover, we had also experimented with different model optimization algorithms:

- Mini-batch gradient descent (GD) [default]
- GD with momentum $\gamma = 0.1$
- GD with adaptive learning rates: (1) RMSProp (2) Adam

Furthermore, we had applied early stopping for the convolutional neural network, so as to prevent model overfitting and to assess the impact of different hyper-parameters or optimization algorithms on model convergence time.

2.2.1 Architecture

For Q1 to Q4 of part A, we developed the convolutional neural network, with the following architecture:

- 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, and
- A output softmax layer of size 10

2.2.2 Learning Goal

In this assignment, the above-mentioned neural models aim to minimize the cross-entropy loss.

The cross-entropy is the cost function for neural network models learning classification tasks, it is the negative likelihood of the data given by the model:

$$-log\big(p(data|model)\big) = -\sum_{k=1}^{K} n_k log\, p_k$$

2.2.3 Model optimization algorithms

2.2.3.1 Mini-batch gradient descent

Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent, with the introduction of the random shuffling of the pairs of the inputs and outputs in each epoch, and the efficiency of batch gradient descent. We trained the models batch by batch in an effort to minimize the loss function.

2.2.3.2 GD with momentum γ = 0.1

Momentum update is given by:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

Here, V is known as the velocity term and has the same direction as the weight vector W.

The momentum parameter γ ∈ [0,1] indicates how many iterations the previous gradients are incorporated into the current update. With momentum, the current update of the weight not only depends on the gradient estimation of your current batch, but also depends on the estimations done in the previous batch. Thus, the convergence process is going to take an "inertial speed", that is going to make it resilient against the noise that it will encounter in the initial transient phase. If the direction of the gradient estimation of the current batch is the same as that of the previous batch, the change in weight will be greater; if the direction is opposite, the change in weight will be partially offset, and the magnitude of change will be reduced. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. In this assignment, γ is kept at 0.1.

2.2.3.3 GD with Root Mean Square Propagation (RMSProp)

RMSProp uses an exponentially decaying average to discard the history from extreme past so that it can converge after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\boldsymbol{r}+\varepsilon}} \cdot (\nabla_{\boldsymbol{W}} J)$$

Here, the parameter ρ controls the length of the moving average of gradients; it sets the weight between the average of previous values and the square of current value to calculate the new weighted average. It exponentially decays average to decay from the extreme past gradient. By restricting the oscillations in the vertical direction, we can increase the learning rate, and larger steps in the horizontal direction, which can converge faster.

2.2.3.4 GD with Adam Optimizer

Adam optimizer realizes the benefits of both RMSProp and momentum methods. The algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters ρ1 and ρ2 control the decay rates of these moving averages (Brownlee, 2017). It is generally regarded as fairly robust to hyperparameters. The formula applied is as follows:

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1}$$
$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

2.2.4 Grid Search Method

In order to optimize the hyper parameters, we have designed controlled experiments, by conducting gridsearch for the 2 hyperparameters below:

(a) N1 ∈ [40, 45, 50, 55, 60, 65, 70], where N1 is the number of feature maps for C1

(b) N2 ∈ [40, 45, 50, 55, 60, 65, 70], where N2 is the number of feature maps for C2

As there are 7 choices for each hyperparameter, if we brutally apply gridsearch on all the possible combinations, we will end up building 7^2 = 49 models, which is very time and resource consuming. Thus, we had decided to apply the grid search in 2 phases, so as to reduce the resources needed for training:

1) Firstly, we conduct gridsearch on N1 ∈ [40, 50, 60, 70] and N2 ∈ [40, 50, 60, 70]. This will result in 4^2 = 16 combinations.
2) Next, among the 16 models developed in phase 1, we will pick the one with the best performance, for example (N1 = 60, N2 = 50). This will give us a hint that the best combination of of N1 and N2 probably lies around (60, 60). As such, we will narrow down the gridsearch to N1 ∈ [55, 60, 65] and N2 ∈ [45, 55, 55], which will lead to 3^2 = 9 models.

As a result, we only need to develop 16 + 9 = 25 models, rather than 49 models.

2.2.5 Early Stopping

By default, we set the number of training epochs as 2000. However, in order to prevent overfitting and to improve generalization of the mode, as well as to assess the impact of different hyper-parameters on model convergence time, we had also experimented with early stopping.

When early stopping is applied, 20% of the original training data was randomly sampled as the validation data (the validation data will not be trained) before training. At the end of each training epoch, we kept track of the validation error using the validation data. To decide when to early stop, we introduced another 2 parameters:

(a) Patience (default: 20) - Number of epochs with no improvement after which training will be stopped.

(b) Min_delta (default: 0.0005) - Minimum improvement in the monitored quantity to qualify as an improvement.

For example, if the validation error did not improve by min_delta of 0.0005 for consecutive 20 epochs (patience), the training will be early stopped.

3. Experiments and Results

In this section, we present the experiment findings for different hyper-parameters. The default hyper-parameters and optimization method are, unless otherwise stated:

- Batch size = 128
- Convolution layer C1:
    - Number of filters : 50
    - Kernel size : 9 x 9
    - Padding method : VALID
    - Stride : 1
    - Activation : ReLu
- Pooling layer P1:
    - Kernel size : 2 x 2
    - Padding method : VALID
    - Stride : 2
- Convolution layer C2:
    - Number of filters : 60
    - Kernel size : 5 x 5
    - Padding method : VALID
    - Stride : 1
    - Activation : ReLu
- Pooling layer P2:
    - Kernel size : 2 x 2
    - Padding method : VALID
    - Stride : 2
- Size of fully connected layer F3 = 300
- Size of softmax layer = 10
- Learning Rate = 0.01
- Patience = 20
- Min_delta = 0.0005

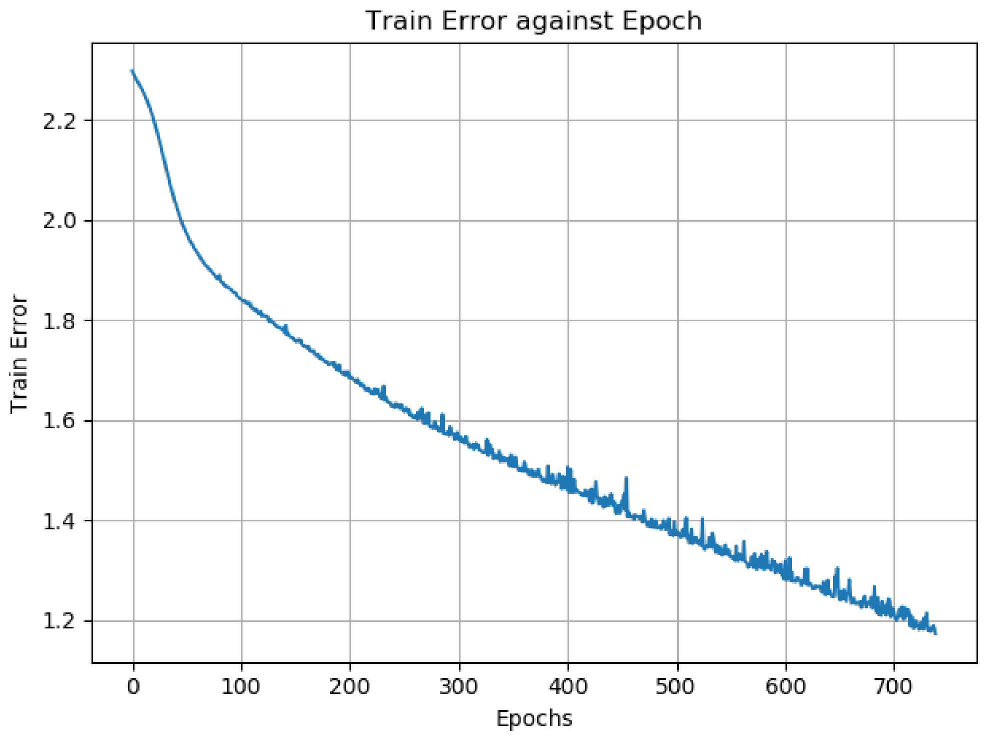
3.1 CNN with default parameters

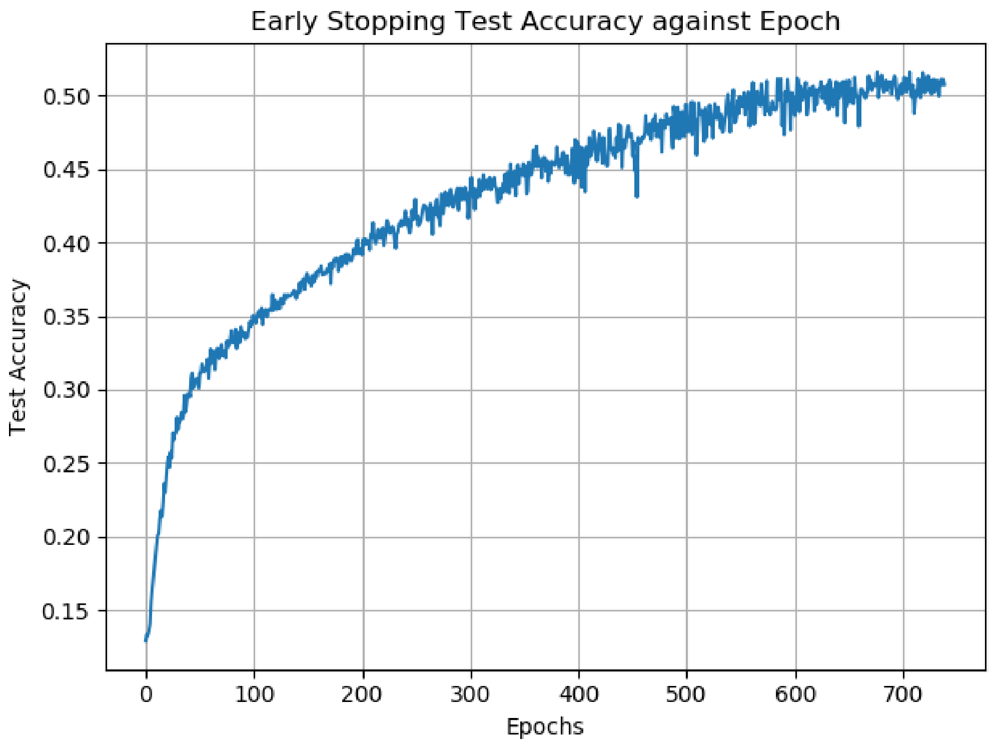
This section answers Q1 of part A.

The table below summarises our findings:

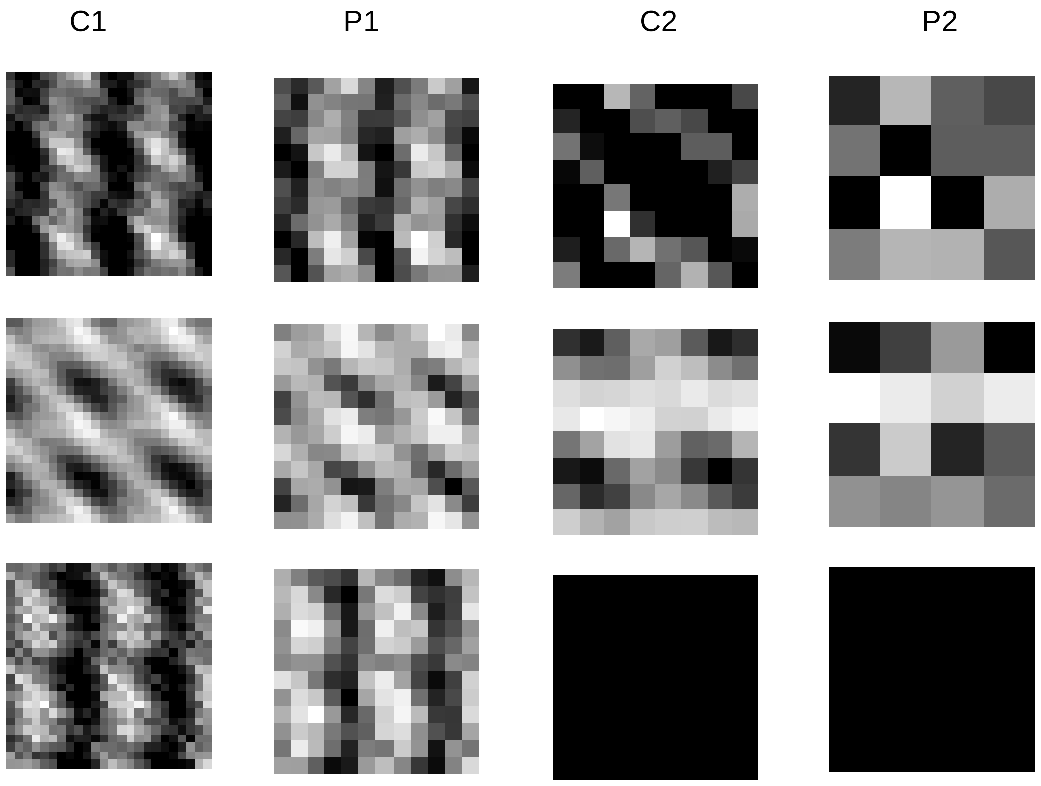
Test Accuracy	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
0.5075	933.2135	739	689644.7765

We next present the training cost and test accuracy against epochs:

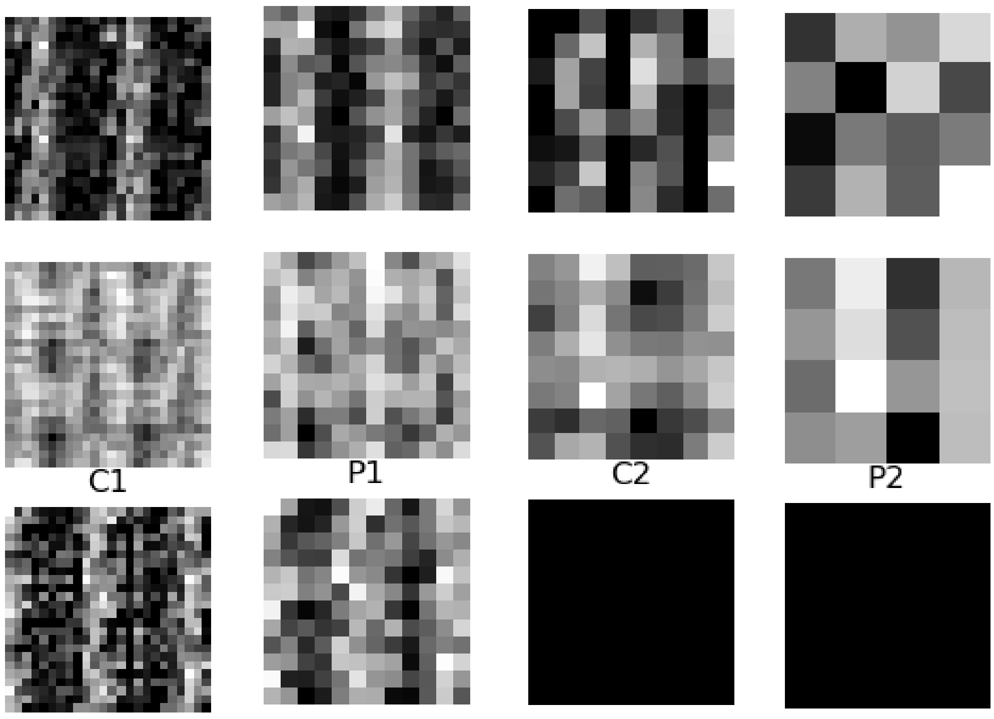




Next, we randomly selected 2 pictures from the test set, and present their corresponding feature maps at C1, P1, C2 and P2:



C1 P1 C2 P2



3.2 Optimal numbers of feature maps

This section answers Q2 of part A. The parameters used are the same as the default parameters, except: (1) number of filters for C1 (2) number of filters for C2

We applied grid search as mentioned in Section 2.2.4, to find the optimal number of feature maps. The results of the first grid search on N1 ∈ [40, 50, 60, 70] and N2 ∈ [40, 50, 60, 70]:

N1	**N2**	**Test Accuracy**
40	40	0.4850

40	50	0.4875
40	60	0.4845
40	70	0.4865
50	40	0.4845
50	50	0.4770
50	60	0.5115
50	70	0.4960
60	40	0.4965
60	50	0.5000
60	**60**	**0.5120**
60	70	0.5035
70	40	0.4960
70	50	0.5090
70	60	0.4950
70	**70**	**0.5120**

Then, when (N1, N2) = (60, 60) and (N1, N2) = (70, 70), the converged test accuracy is the minimum. According to principle of parsimony, as well as to prevent the model's robustness, we will prefer (N1, N2) = (60, 60), which is simpler than its counterparts. As such, we will narrow down the gridsearch to N1 ∊ [55, 60, 65] and N2 ∊ [45, 55, 55]. The results are summarised as follows:

N1	**N2**	**Test Accuracy**
55	55	0.4950
55	60	0.4900
55	65	0.5051
60	55	0.4980
60	**60**	**0.5120**
60	65	0.4895
65	55	0.4930

65	60	0.4865
65	65	0.5025

For the second round of grid search, when number of filters in C1 = 60, number of filters in C2 = 60, the converged test accuracy is the minimum. Hence, the optimal number of feature maps:

- N1 = 60
- N2 = 60

We will apply the optimal number of filters for Q3.

3.3 CNN with different training methods

This section answers Q3 of part A.

The hyperparameters used in this question is the same as the default parameters, except that we are using the optimal number of filters for C1 and C2 found in previous section:

- N1 = 60
- N2 = 60

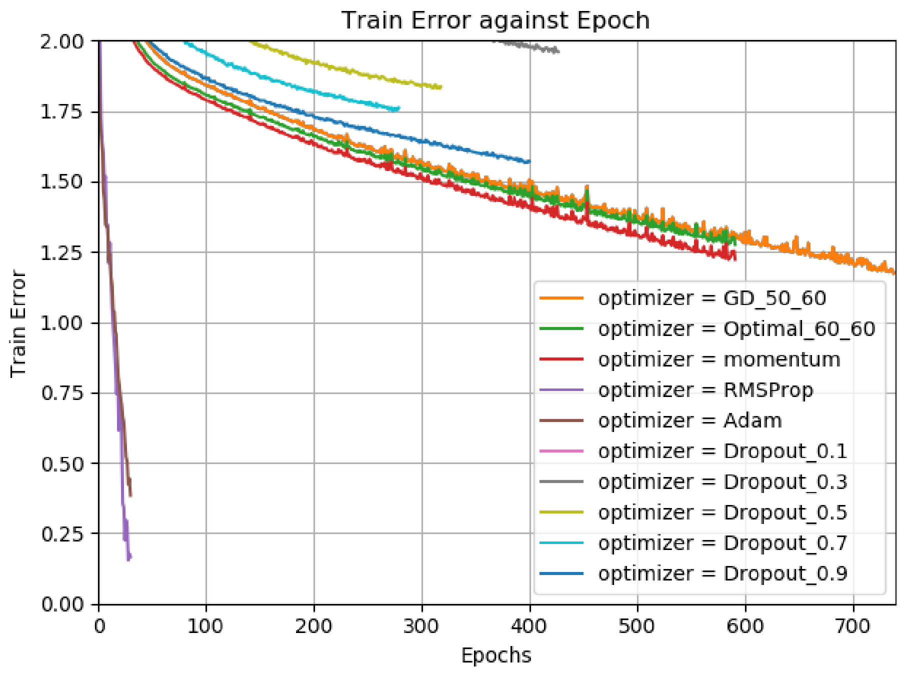
We trained the network with different methods:

(a) Added the momentum term with momentum γ = 0.1
(b) Use RMSProp algorithm for learning
(c) Use Adam optimizer for learning
(d) Add dropout to the layers (2 pooling layers and output layer)

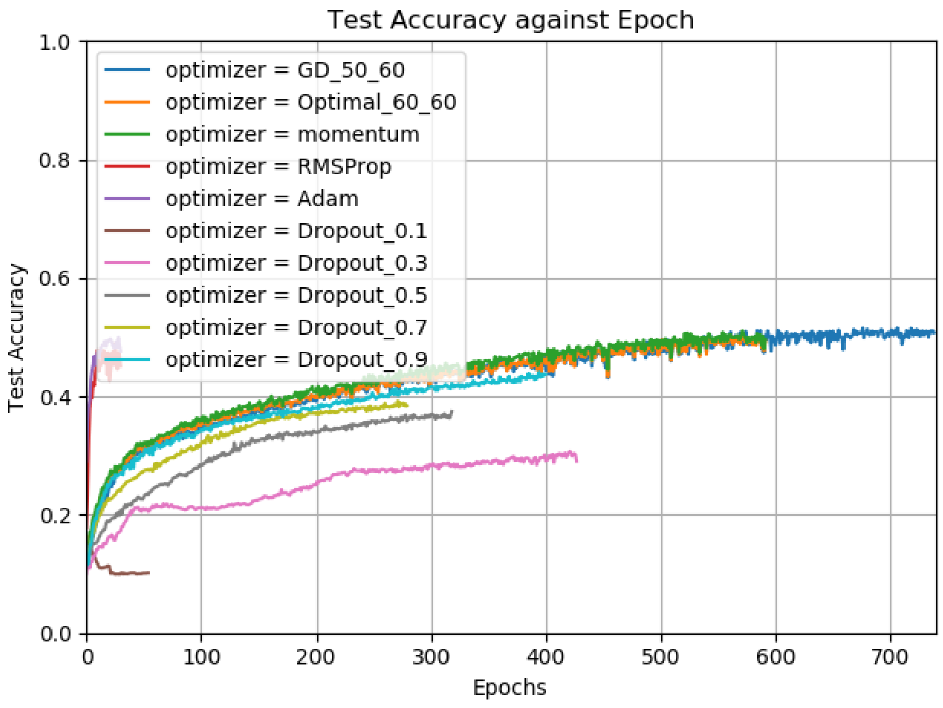
For (d), we experimented with different keep probability values for drop out, the keep probability values experimented are: 0.1, 0.3, 0.5, 0.7, 0.9.

Model	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
momentum	0.4970	1057.8750	592	626262.0000
RMSProp	0.4515	1105.0839	31	34257.6009
Adam	0.4760	1095.0193	31	33945.5983
Dropout_0.1	0.1020	1181.9127	55	65005.1985
Dropout_0.3	0.2900	1202.0748	428	514488.0144
Dropout_0.5	0.3745	1173.5699	319	374368.7981
Dropout_0.7	0.3840	1187.7729	280	332576.4120
Dropout_0.9	0.4370	1202.7561	401	482305.1961

The plot of training cost against epochs:



The plot of test accuracies against epochs:



3.4 Evaluation of model performance

This section answers Q4 of part A. The question requested us to apply dropout, but the keep_prob to apply was not specified. Thus, we had decided to experiment with keep_prob = [0.1, 0.3, 0.5, 0.7, 0.9].

The results from section 3.1 to 3.3 can be summarised into the table below:

Model	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
GD_50_60	0.5075	933.2135	739	689644.7765
Optimal_60_60	**0.5120**	**1061.5378**	**592**	**628430.3776**
momentum	0.4970	1057.8750	592	626262.0000
RMSProp	0.4515	1105.0839	31	34257.6009
Adam	0.4760	1095.0193	31	33945.5983
Dropout_0.1	0.1020	1181.9127	55	65005.1985
Dropout_0.3	0.2900	1202.0748	428	514488.0144
Dropout_0.5	0.3745	1173.5699	319	374368.7981
Dropout_0.7	0.3840	1187.7729	280	332576.4120
Dropout_0.9	0.4370	1202.7561	401	482305.1961

Here,

(1) GD_50_60 refers to gradient descent model with 50 C1 filters and 60 C2 filter
(2) Optimal_60_60 refers to gradient descent model with 60 C1 filters and 60 C2 filter
(3) Momentum refers to model with momentum, 60 C1 filters and 60 C2 filter
(4) RMSProp refers to model with RMSProp algorithm, 60 C1 filters and 60 C2 filter
(5) Adam refers to model with Adam algorithm, 60 C1 filters and 60 C2 filter
(6) Dropout_0.1 refers to model with drop out (keep probability is 0.1), 60 C1 filters and 60 C2 filter
(7) Dropout_0.3 refers to model with drop out (keep probability is 0.3), 60 C1 filters and 60 C2 filter
(8) Dropout_0.5 refers to model with drop out (keep probability is 0.5), 60 C1 filters and 60 C2 filter
(9) Dropout_0.7 refers to model with drop out (keep probability is 0.7), 60 C1 filters and 60 C2 filter
(10) Dropout_0.9 refers to model with drop out (keep probability is 0.9), 60 C1 filters and 60 C2 filter

Next, we will evaluate the model performances with 2 different metrics - test accuracy and total training time.

Test Accuracy:
From the table, it is shown that optimal_60_60 has test accuracy of 0.5120, which is the best

among all models.

(1) <u>Optimal_60_60 vs. GD_50_60</u>
When the number of C1 filters increases from 50 to 60, the test accuracy improved from 0.5075 to 0.5120. This is probably due to the model capacity is still insufficient in producing a high test accuracy (underfitting), when number of C1 filters is 50. Hence, when we used a higher number of C1 filters, the model became more complexed and less underfit; the test accuracy improved for Optimal_60_60.

(2) <u>Optimal_60_60 vs. Models with drop out</u>
The model without dropout (optimal_60_60) has a test accuracy of 0.5120, much higher than those models with drop out. Moreover, for this particular dataset, drop out had backfired and hurt the model performance. The test accuracies for models with drop out are significantly lower than without dropout (optimal_60_60). This indicates that the model has yet reached the overfitting stage, and may still be underfitting. The intention for the application of dropout is to limit the model capacity so as to prevent overfitting. Since model without dropout is probably not yet overfitted, dropout shall not be applied here to further lower the model capacity.

(3) <u>Optimal_60_60 vs. Models with optimization methods</u>
The test accuracy of models with adaptive optimization methods such as momentum, RMSProp and Adam have lower test accuracy than optimal_60_60, which uses the conventional gradient descent method. According to Wilson (Wilson, n.d.), adaptive optimization methods, which perform local optimization with a metric constructed from the history of iterates, can lead to drastically different findings as compared to using gradient descent or stochastic gradient descent. Wilson also pointed out that the performance of models using adaptive optimization methods are generally worse (and significantly worse) than the performance of models using GD or SGD, even though adaptive optimization methods often have better training performances.

Nonetheless, it is worth noting that the models with Adam/RMSProp optimization methods converged significantly faster than conventional gradient descent method (refer to the table above for number of training epochs). Thus, such optimization methods will be very helpful when the dataset becomes large, so that the model can converge to a minimum with significant training cost savings.

<u>Total Time Taken:</u>
Total time = Time Taken per Epoch x Number of Epochs

From the table, it is shown that the model with Adam optimizer has the shortest total training time (33945.6ms), as it has the short time per epoch and smallest early_stop epochs.

(1) Adam vs. RMSProp
It can be shown that Adam has the total training time of 33945.6ms, shorter than that of RMSProp (34257.6009ms). This is because Adam combines the benefits of both RMSProp and momentum methods. With momentum, if the current gradient direction is same as the previous, the gradient will be updated with a higher value than it would for RMSProp; if the current gradient direction is opposite of the previous gradient direction (which is more frequent in the case of narrow ravine), momentum will help dampen the oscillation and allows faster convergence to the minimum.

(2) Adam and RMSProp vs. the remaining models
It can be shown that both RMSProp and Adam have significantly lower total training time than that of the other models. This is primarily due to the fact that both can adjust learning rates automatically during training, and use an exponentially decaying average to discard the history from extreme past, so that both can converge rapidly after finding a convex region.

Overall, while Adam has the shortest total training time, its test accuracy (0.4760) is significantly lower than that of Optimal_60_60 (0.5120). As the dataset is comparatively small and does not require much train cost even for the case of Optimal_60_60, test accuracy will be a more important criterion than total training time in the model assessment. Thus overall, **Optimal_60_60 is the best model among all**.

The plots below show the visual comparisons of different models' performances:

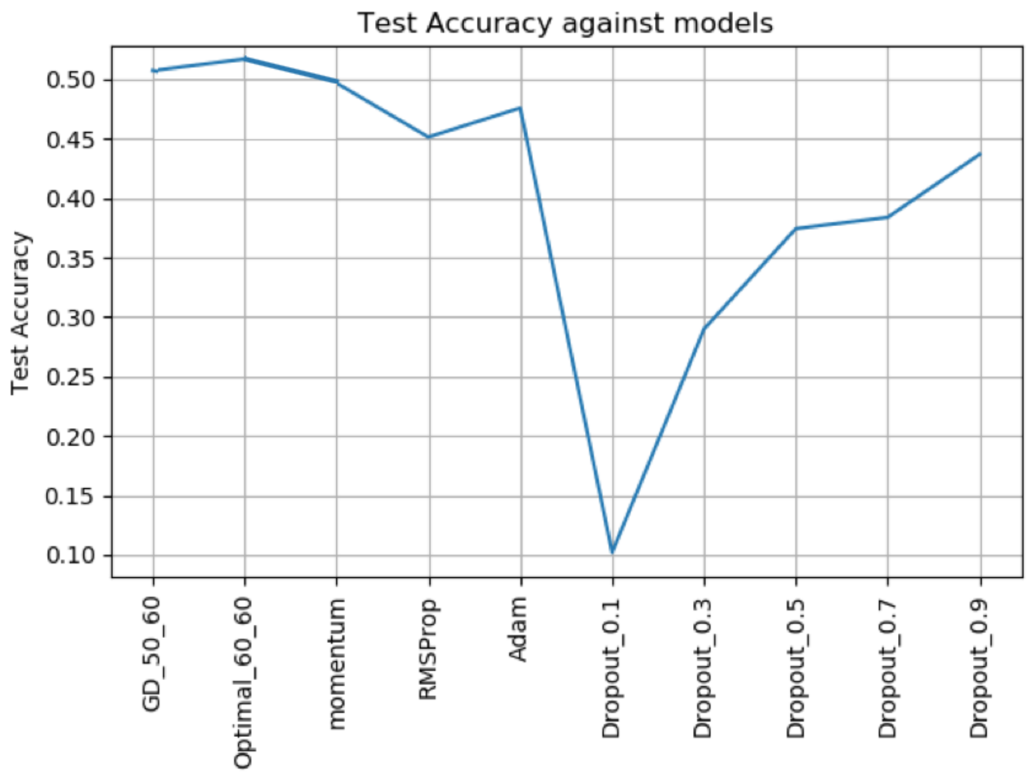


Fig1. Test accuracy against models

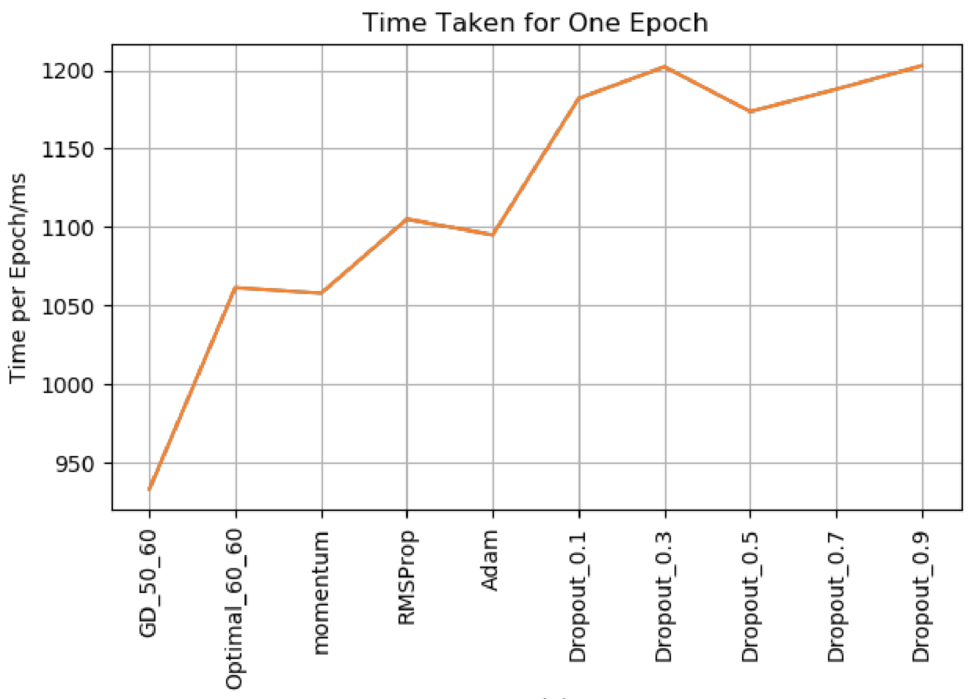


Fig2. Time Taken for one epoch

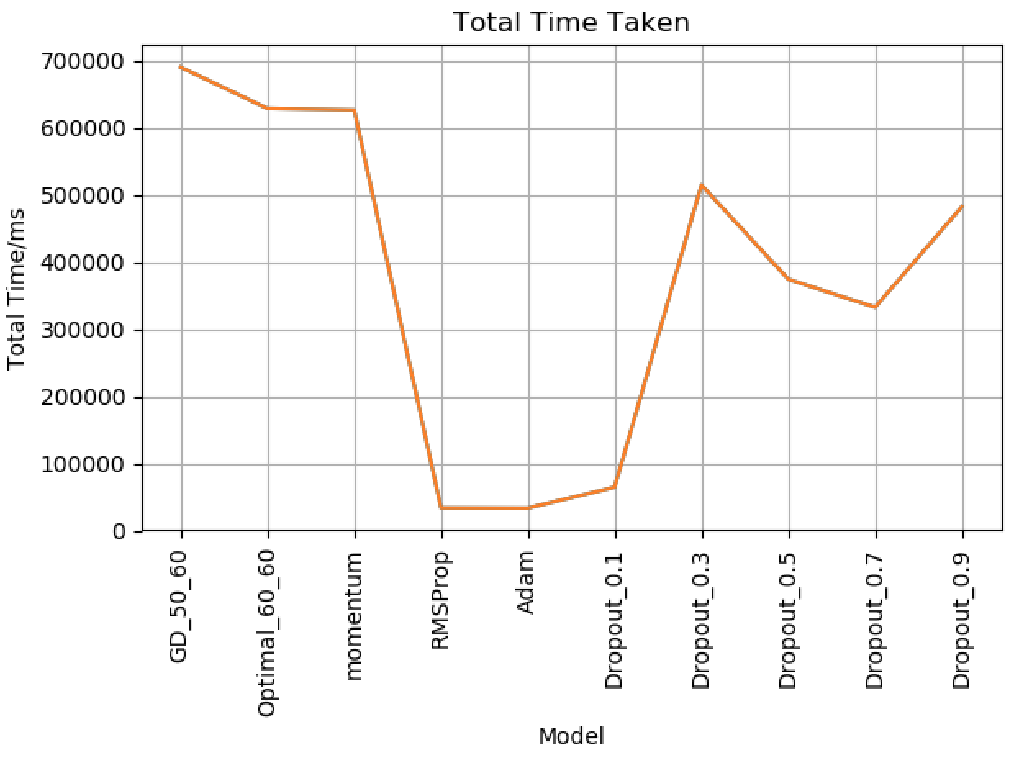


Fig3. Total Time Taken

Part B - Text Classification

1. Introduction

The project aims at building convolutional neural networks/recurrent neural networks at char and word level, so as to classify the text into different categories. The dataset used in this project contains the first paragraphs collected from Wikipage entries and the corresponding labels about their category. The training dataset contains 5600 entries and test dataset contains 700 entries. The label of an entry is one of the 15 categories such as people, company, schools, etc.

2. Method

2.1 Data pre-processing: Encoding

The input data is in text, which need be converted to character/word IDs to feed to the networks by using 'tf.contrib.learn.preprocessing.ByteProcessor' and 'tf.contrib.learn.preprocessing.VocabularyProcessor'. The maximum length of the characters/word inputs is restricted to 100.

2.2 Model Development

For this assignment, we train the convolutional neural networks (CNN) and recurrent neural networks (RNN) aiming to optimize their cross-entropy cost function.

Furthermore, we had applied early stopping for both CNN and RNN, so as to prevent model overfitting and to assess the model convergence time.

2.2.1 Architecture

2.2.1.1 Char CNN

For Q1 of part B, we developed the char CNN, with the following architecture:

- An Input layer of 100 dimensions (since we had restricted the maximum length of char/word inputs to 100).
- A convolution layer $C1$ of 10 filters of window size 20x256, VALID padding, and ReLU neurons. A max pooling layer $S1$ with a pooling window of size 4x4, with stride = 2 and padding = SAME.
- A convolution layer $C2$ of 10 filters of window size 20x1, VALID padding, and ReLU neurons. A max pooling layer $S2$ with a pooling window of size 4x4, with stride = 2 and padding = SAME.
- An output softmax layer of size 15

2.2.1.2 Word CNN

For Q2 of part B, we developed the word CNN, with the following architecture:

- An Input layer of 100 dimensions (since we had restricted the maximum length of char/word inputs to 100).
- An embedding layer with embed_size = 20
- A convolution layer $C1$ of 10 filters of window size 20x20, VALID padding, and ReLU neurons. A max pooling layer $S1$ with a pooling window of size 4x4, with stride = 2 and padding = SAME.
- A convolution layer $C2$ of 10 filters of window size 20x1, VALID padding, and ReLU neurons. A max pooling layer $S2$ with a pooling window of size 4x4, with stride = 2 and padding = SAME.
- An output softmax layer of size 15

2.2.1.3 Char RNN

For Q3 of part B, we developed the char RNN, with the following architecture:

- An Input layer of 100 dimensions (since we had restricted the maximum length of char/word inputs to 100).
- A GRU layer with a hidden-layer size of 20
- An output softmax layer of size 15

2.2.1.4 Word RNN

For Q4 of part B, we developed the char RNN, with the following architecture:

- An imbedding layer of size 20
- An embedding layer with embed_size = 20
- A GRU layer with a hidden-layer size of 20
- An output softmax layer of size 15

2.2.2 Learning Goal

In this assignment, the above-mentioned neural models aim to minimize the cross-entropy loss.

The cross-entropy is the cost function for neural network models learning classification tasks, it is the negative likelihood of the data given by the model:

$$-log\big(p(data|model)\big) = -\sum_{k=1}^{K} n_k log\, p_k$$

2.2.3 Early Stopping

In order to prevent overfitting and to improve generalization of the mode, as well as to assess the impact of different hyper-parameters on model convergence time, we had also experimented with early stopping.

When early stopping is applied, 20% of the original training data was randomly sampled as the validation data (the validation data will not be trained) before training. At the end of each training epoch, we kept track of the validation error using the validation data. To decide when to early stop, we introduced another 2 parameters:

(c) Patience (default: 20) - Number of epochs with no improvement after which training will be stopped.
(d) Min_delta (default: 0.0005) - Minimum improvement in the monitored quantity to qualify as an improvement.

For example, if the validation error did not improve by min_delta of 0.0005 for consecutive 20 epochs (patience), the training will be early stopped.

3. Experiments and Results

In this section, we present the experiment findings for different hyper-parameters. The default hyper-parameters and optimization method are, unless otherwise stated:

- Batch size = 128
- Maximum length of the characters / word inputs = 100
- Learning rate = 0.01
- Character vocabulary size = 256
- Optimization = Adam

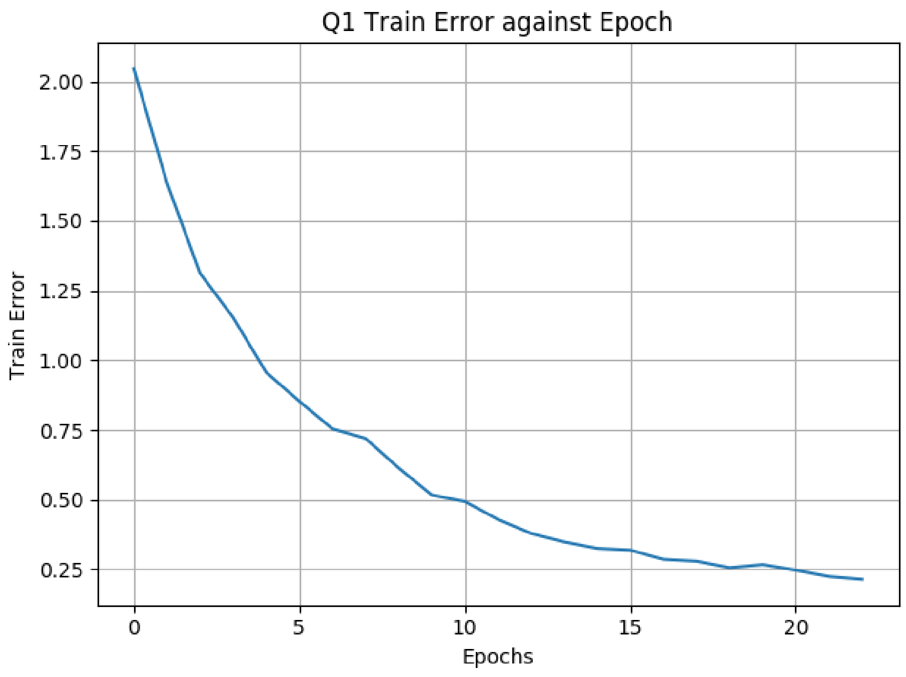
3.1 Character CNN

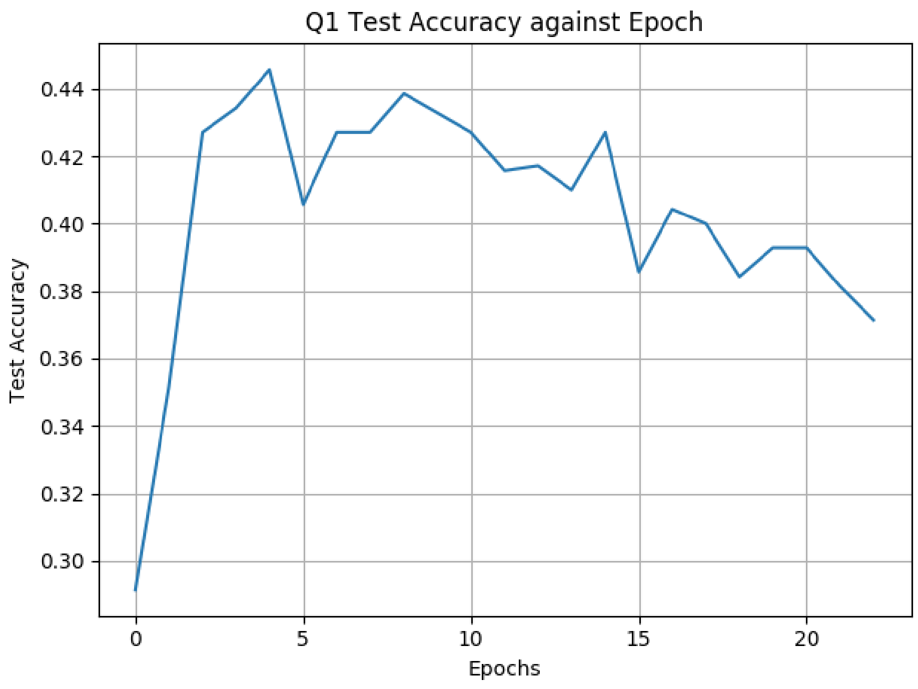
This section answers Q1 of part B. In this section, we applied the architecture defined in section 2.2.1.1.

The table below summarises our findings:

Test Accuracy	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
0.3714	816.6261	23	18782.4003

We next present the training cost and test accuracy against epochs:





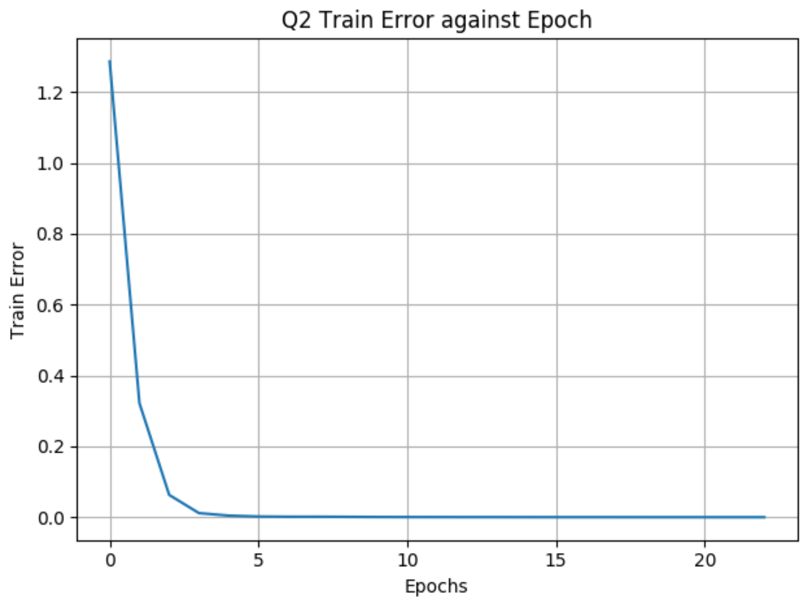
3.2 Word CNN

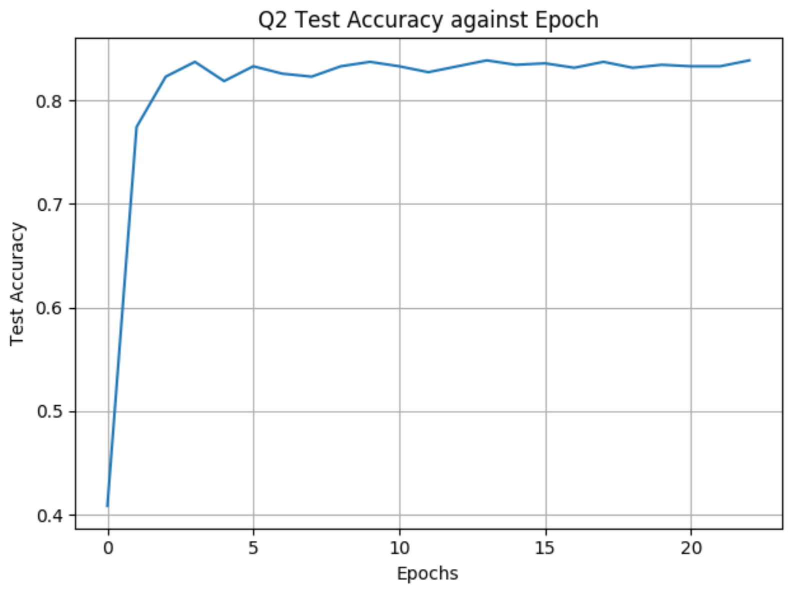
This section answers Q2 of part B. In this section, we applied the architecture defined in section 2.2.1.2.

The table below summarises our findings:

Test Accuracy	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
0.8386	246.2087	23	5622.7996

Next, we present the plots for train error and test accuracy against epoch:





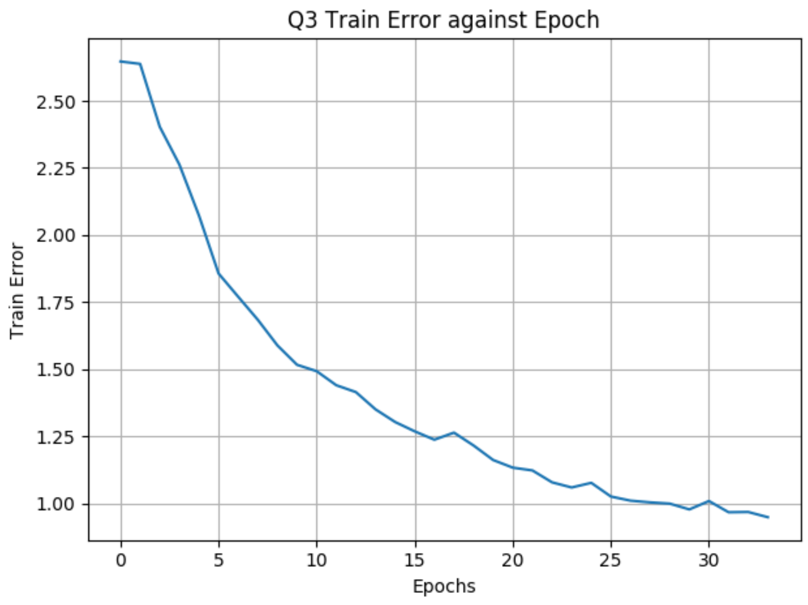
3.3 Character RNN

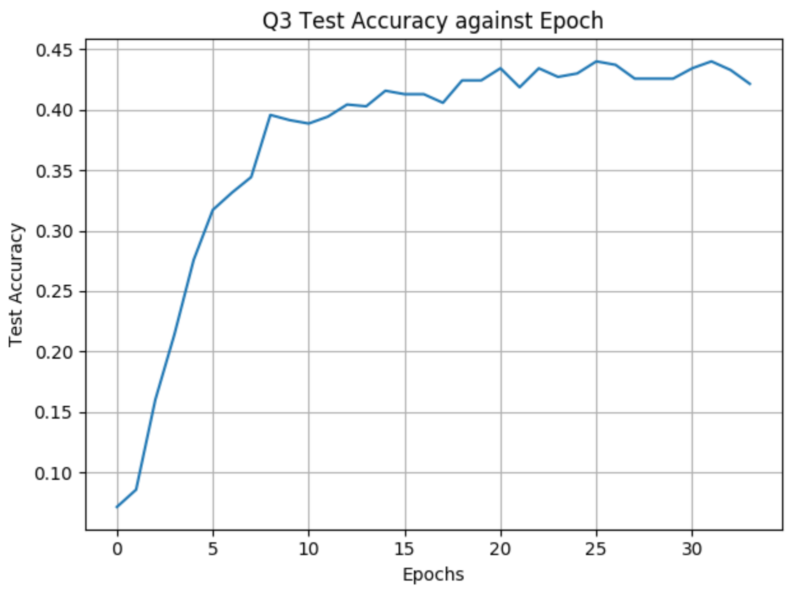
This section answers Q3 of part B. In this section, we applied the architecture defined in section 2.2.1.3.

The table below summarises our findings:

Test Accuracy	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
0.4214	6687.8118	34	227385.5996

Next, we present the plots for train error and test accuracy against epoch:





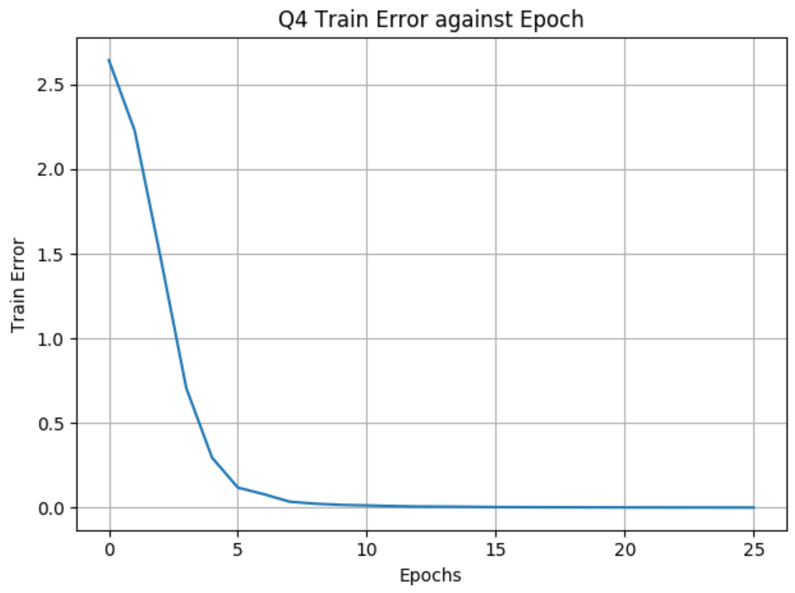
3.4 Word RNN

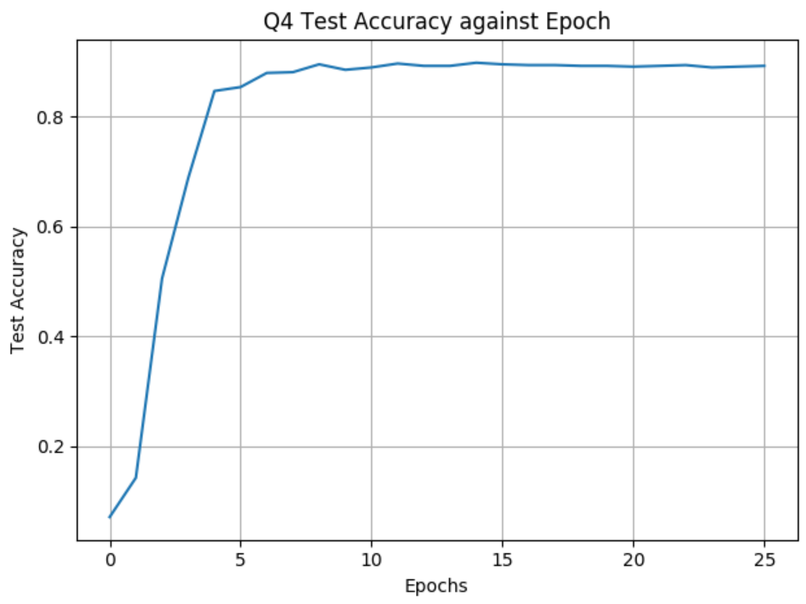
This section answers Q4 of part B. In this section, we applied the architecture defined in section 2.2.1.4.

The table below summarises our findings:

Test Accuracy	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
0.8929	7071.6000	26	183861.6

Next, we present the plots for train error and test accuracy against epoch:





3.5 Model comparison

This section answers Q5 of part B. There are 2 parts in Q5, one is to compare model performance in Q1 to Q4 (section 3.5.1), and the other is to compare model performances with and without dropout (section 3.5.2).

3.5.1 Comparison of model performance in Q1 - Q4

Character model:

First, we would compare the performance of Character CNN in 3.1 with Character RNN in 3.3. The table below summarises our findings:

Model	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
q1_char_cnn	0.3714	816.6261	23	18782.4003
q3_char_rnn	0.4214	6687.8118	34	227385.5996

Comparison of test accuracies:

It is found that the test accuracy of Character RNN (0.4214) is higher than that of Character CNN (0.3714).

Comparison of running times of the networks:

In terms of time per epoch, Character RNN takes 6687.8118ms per epoch, longer than that of Character CNN (816.6261ms per epoch).

In terms of epochs, Character RNN requires 34 epochs to early stop, more than that of Character CNN (23 epochs).

In terms of total time, Character RNN requires 227385.5996ms, much longer than that of Character CNN of 18782.4003ms.

Word model:

Next, we would compare the performance of Word CNN in 3.2 with Word RNN in 3.4. The table below summarises our findings:

Model	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
q2_word_cnn	0.8386	246.2087	23	5622.7996

q4_word_rnn	0.8929	7071.6000	26	183861.6000

<u>Comparison of test accuracies:</u>

It is found that the test accuracy of Word RNN (0.8929) is higher than that of Word CNN (0.8386).

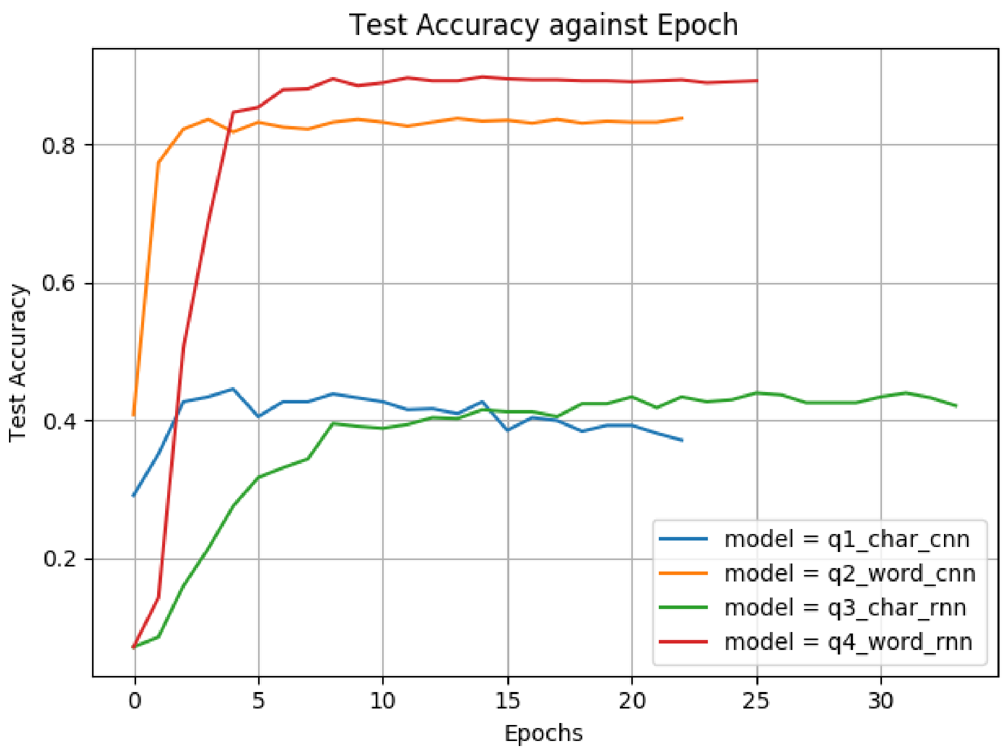
<u>Comparison of running times of the networks:</u>

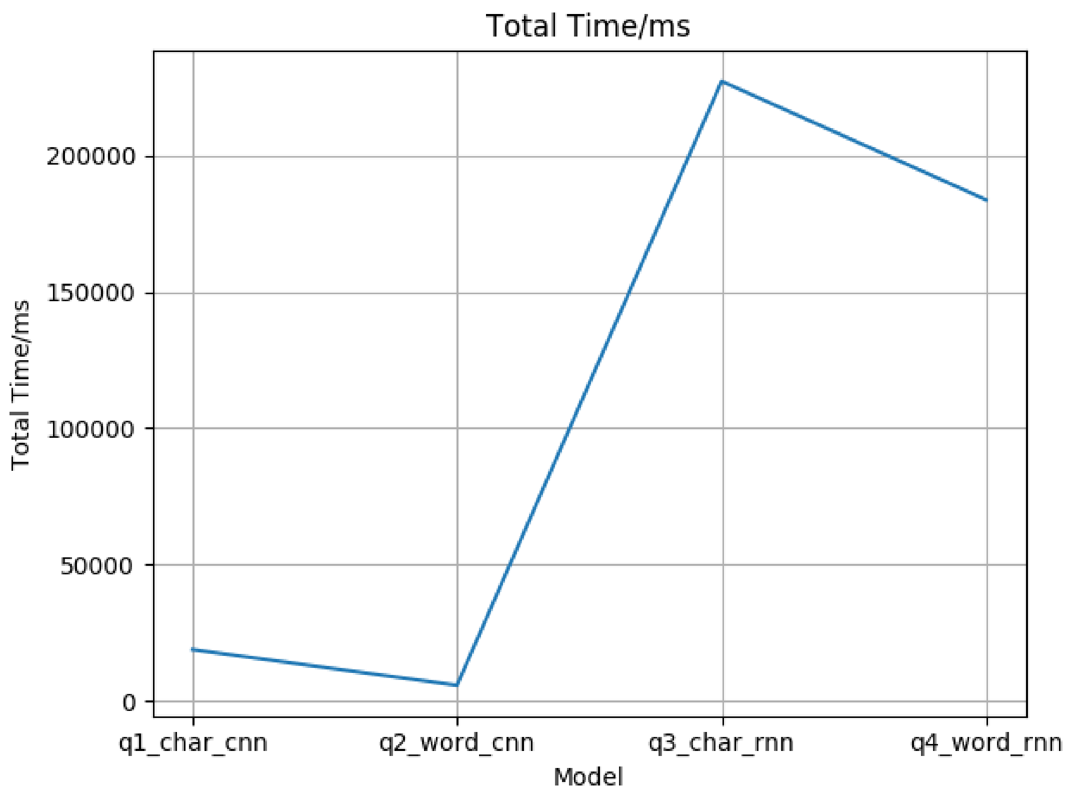
In terms of time per epoch, Word RNN takes 7071.6ms per epoch, longer than that of Word CNN (246.2087ms per epoch).

In terms of epochs, Word RNN requires 26 epochs to early stop, more than that of Word CNN (23 epochs).

In terms of total time, Word RNN requires 183861.6ms, much longer than that of Word CNN of 5622.7996ms.

Next, we present the plot for test accuracy and total training time against different models from section 3.1 to 3.4:





3.5.2 Comparison of model performance in parts (1) - (4) with / without dropout

We would compare the performance of models in 3.1-3.4 with / without dropout. Q5 did not specify which keep_prob to apply, and thus, we tried to experiment with keep_prob = [0.1, 0.3, 0.5, 0.7, 0.9]. The table below summarises our findings:

Model	**Drop out**	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
char_cnn	No drop out	0.3714	816.6261	23	18782.4003
	Keep_prob 0.1	0.1429	728.1677	62	45146.4000
	Keep_prob 0.3	0.2857	731.6903	31	22682.3988
	Keep_prob 0.5	0.3871	732.0444	27	19765.1997
	Keep_prob 0.7	**0.4614**	**730.0800**	**25**	**18252.0006**

	Keep_prob 0.9	0.4243	733.8501	24	17612.4012
word_cnn	No drop out	0.8386	246.2087	23	5622.7996
	Keep_prob 0.1	0.1429	249.6000	34	8486.4004
	Keep_prob 0.3	0.7271	230.7500	24	5538.0001
	Keep_prob 0.5	0.8129	247.1040	25	6177.5992
	Keep_prob 0.7	**0.8586**	**250.3091**	**22**	**5506.7992**
	Keep_prob 0.9	0.8014	236.0348	23	5428.7994
char_rnn	No drop out	0.4214	6687.8118	34	227385.5996
	Keep_prob 0.1	0.36	7699.0775	49	377254.7987
	Keep_prob 0.3	0.4014	6895.6334	36	248242.8010
	Keep_prob 0.5	0.4057	7008.9882	34	238505.5995
	Keep_prob 0.7	0.4086	6706.8300	40	268273.1998
	Keep_prob 0.9	**0.4371**	**7585.3946**	**37**	**280659.6002**
word_rnn	**No drop out**	**0.8929**	**7071.6000**	**26**	**183861.6000**
	Keep_prob 0.1	0.7871	7736.0903	62	479637.6002
	Keep_prob 0.3	0.5657	7199.4000	28	201583.2002

	Keep_prob 0.5	0.8657	7238.4001	31	224390.4021
	Keep_prob 0.7	0.88	7330.9935	31	227260.7996
	Keep_prob 0.9	0.7614	7673.0483	29	222518.4000

Character CNN:
The char model with drop out and keep probability of 0.7 has the highest test accuracy of 0.4616, much higher than that without drop out (0.3714). This implies that the original char cnn model may have overfitted, such that by applying dropout to limit the model capacity helped to improve the model generalization on unseen test data.

Word CNN:
The word model with drop out and keep probability of 0.7 has the highest test accuracy of 0.8586, higher than that without drop out (0.8386). This implies that the original word cnn model may have overfitted, such that by applying dropout to limit the model capacity helped to improve the model generalization on unseen test data.

Character RNN:
The char model with drop out and keep probability of 0.9 has the highest test accuracy of 0.4371, higher than that without drop out (0.4214). This implies that the original char rnn model may have overfitted, such that by applying dropout to limit the model capacity helped

Word RNN:
Word RNN model without drop out has the highest test accuracy of 0.8929, higher than those with drop out. This shows that the current model capacity is appropriate, we do not need to limit the model capacity by further applying drop out.

3.6 Char/Word RNN with modifications

This section answers Q6 of part B. In this section, we explore with the following modifications with regards to the char/word RNN in section 3.3 and 3.4:

- Vanilla RNN layer/LSTM layer
- Increase number of RNN layers to 2
- Gradient clipping with threshold = 2

The results of our experiments as well as the results from section 3.3 and 3.4 can be summarised into the table below:

Model	**Document Length**	**Test Accuracy**	**Time per Epoch (ms)**	**Epochs**	**Total Time (ms)**
q3_char_rnn	100	0.4214	6687.8118	34	227385.5996

q4_word_rnn	**100**	**0.8929**	**7071.6000**	**26**	**183861.6000**
q6a_char_rnn_BASIC (vanilla)	100	0.07143	5411.7375	32	173175.5990
	50	0.08000	1145.1888	30	35500.8528
	20	0.3514	448.3338	52	23313.3576
q6a_word_rnn_BASIC (vanilla)	100	0.07143	5921.2540	37	219086.3990
	50	0.2000	337.2047	20	6744.0940
	20	0.4829	184.5156	23	4243.8588
q6a_char_rnn_LSTM	100	0.07143	7681.0500	32	245793.6010
	50	0.4429	1573.7551	49	77113.9999
q6a_word_rnn_LSTM	100	0.7243	7880.8080	50	394040.4000
	50	0.7929	667.4366	38	25362.5908
q6b_char_rnn_2_layer	100	0.4714	14120.6634	41	578947.2000
q6b_word_rnn_2_layer	100	0.7771	15175.8750	48	728442.0000
q6c_char_rnn_gradient_clipped	100	0.4514	14476.3784	37	535626
q6c_word_rnn_gradient_clipped	100	0.7371	15218.3571	28	426113.999

Vanilla and LSTM

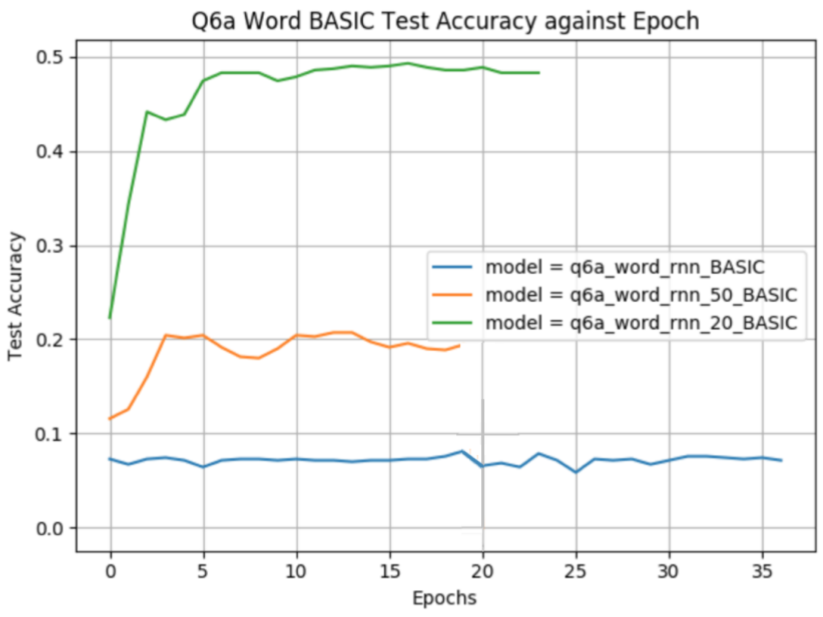
It is worth noting that when we set the maximum length of the characters/words input as 100 (as requested by this assignment), the performance of BASIC and LSTM character models, and vanilla word model are performing terribly bad (only about 7% test accuracy). This is a strong indication that the models are under-fitted. We had identified 2 possible causes for the poor performance:

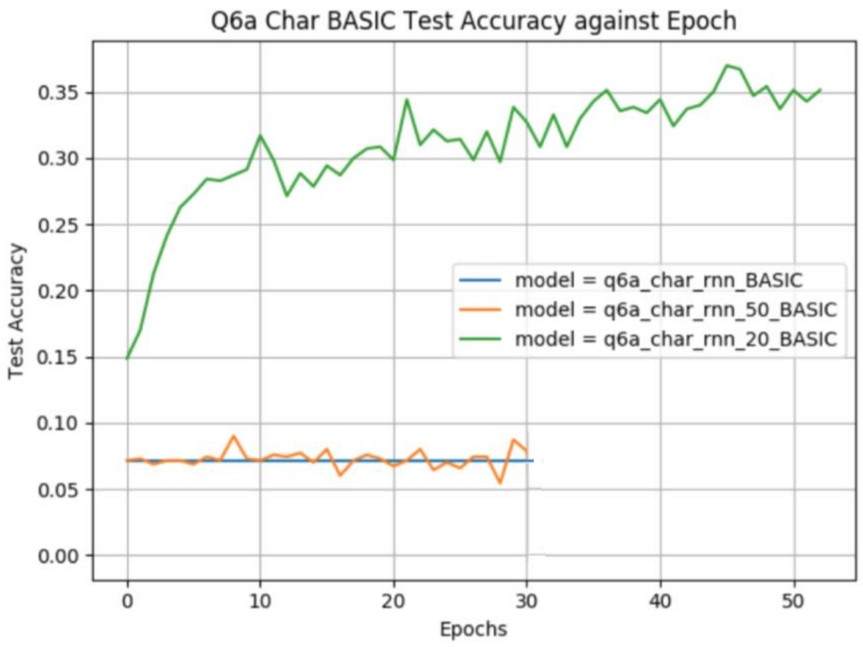
1. Limited data: the training dataset is too small, such that the models have difficulty learning the variations among the data points. However, this problem cannot be addressed as we cannot get more data given the context of this assignment.

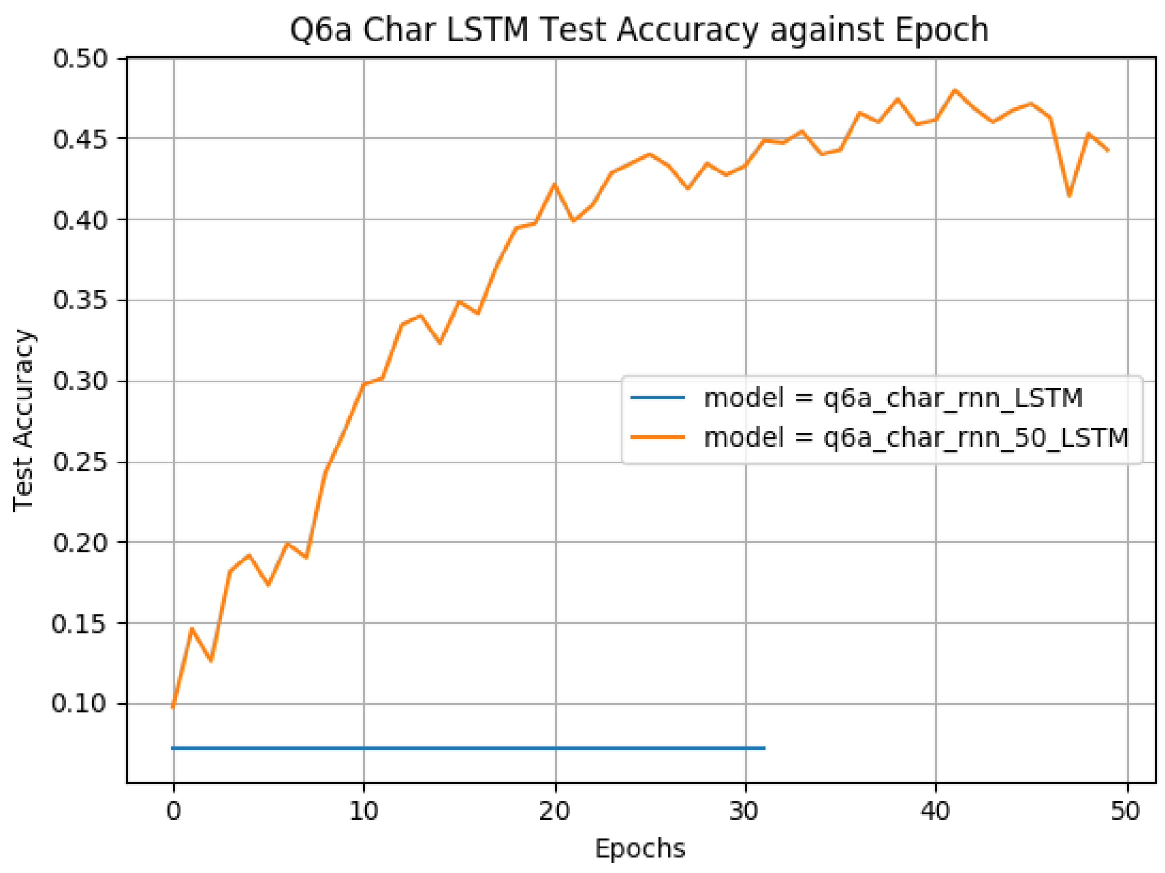
2. The the number of time steps for each data sequence is too big for BASIC and LSTM learn, such that it resulted in **vanishing gradients**. During gradient backpropagation, the gradient ended up being multiplied a large number of times (in this case, 100 times), leading to vanishing gradients, which is a situation where the learning either become very slow or stops working altogether. It is worth noting that although LSTM model is designed such that the vanishing gradients problem should be mitigated, in this assignment, the dataset is probably too small such that when time steps equals to 100, vanishing gradients problem arises even for LSTM models.
    a. And for this problem, we address it by first lowering the maximum length of the characters/words to 50. The results, as indicated in the table above, have shown significant improvement in term of test accuracy for both char/word LSTM models, especially so for the char LSTM model (where the test accuracy shot up to 44% from original 7%).
    b. However, the test accuracy for both char/word vanilla models, though improved, are still bad (8% and 20% respectively). This is probably due to vanilla model's inability to learn long term dependencies. Thus, we further lowered the input dimension to 20 for the vanilla models. This change in parameter had significantly improved the performances of both char/word vanilla models to 35% and 48% test accuracy respectively.

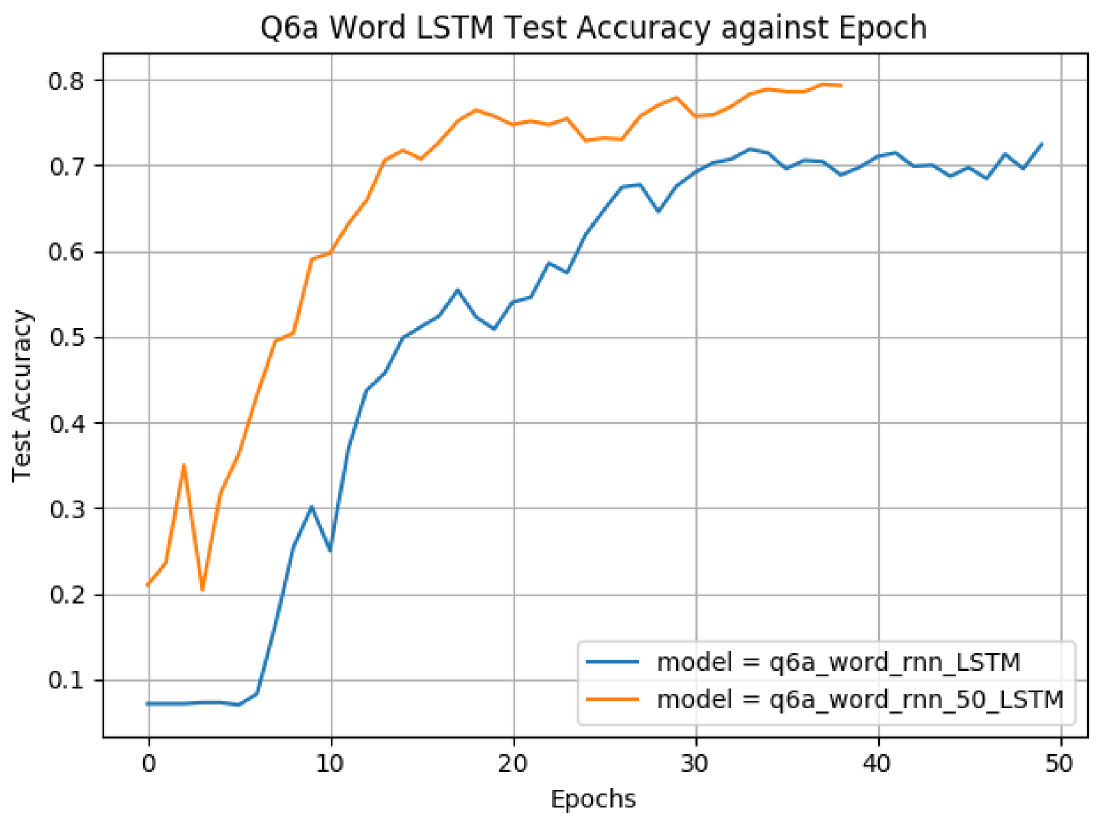
Moreover, from the summary table above, we can also see that with smaller number of time steps for each data sequence, it not only improve the model performance, but also significantly reduce the total training cost by reducing the training time per epoch. The reason for the huge saving in training cost is because the computation in recurrent neural network cannot be easily parallelised due to the fact that forward propagation is inherently sequential.

The 4 plots below show the visual comparisons of different models' performances before and after changing the timesteps:






Q6a Char LSTM Test Accuracy against Epoch
Test Accuracy
Epochs
model = q6a_char_rnn_LSTM
model = q6a_char_rnn_50_LSTM
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
10
20
30
40
50


Q6a Word LSTM Test Accuracy against Epoch
Test Accuracy
Epochs
model = q6a_word_rnn_LSTM
model = q6a_word_rnn_50_LSTM
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50

2-layer GRU

- Character model:
  From the table above, the 2 hidden layers char model (q6b_char_rnn_2_layer) has test accuracy of 47%, while the original 1 hidden layer char model (q3_char_rnn) has test accuracy of 42%. Thus, the capacity of the original 1 hidden layer model is probably still low, such that by expanding the model architecture to 2 hidden layers, the char model's performance was improved.
- Word model:
  From the table above, the 2 hidden layers word model (q6b_word_rnn_2_layer) has test accuracy of 77%, while the original 1 hidden layer word model (q4_word_rnn) has test accuracy of 89%. Thus, the capacity of the original 1 hidden layer model is probably already high, such that by expanding the model architecture to 2 hidden layers, the word model's performance was worsened.

Gradient Clipping

Due to long term dependencies, RNN tends to have gradients having very large or very small magnitudes. The large gradients resemble cliffs in the error landscape and when the gradient descent encounters the gradient updates, it can move the parameters away from true minimum. This is known as the gradient explosion problem. As such, gradient clipping is employed to avoid the gradients from becoming too large. In this experiment, the gradient clipping threshold is set as 2, such that whenever the gradient exceeds a magnitude of 2, it will be clipped to 2 instead.

- Character model: 45%
  From the table above, the gradient-clipped char model (q6c_char_rnn_gradient_clipped) has test accuracy of 45%, while the original char model (q3_char_rnn) has test accuracy of 42%. This improvement shows that without gradient clipping, the model may have moved away from the true minimum, and thus performed worse than with gradient clipping in term of test accuracy.

- Word model: 73%
  From the table above, the gradient-clipped word model (q6c_word_rnn_gradient_clipped) has test accuracy of 73%, while the original word model (q4_word_rnn) ahs test accuracy of 89%. Gradient clipping was applied in hope that it will prevent the model learning process to miss out true minimum due to large gradient changes. However, this may also cause the model to converge to a non-optimal local minima, and it is evident from our experiment with the word models, where the application of gradient clipping worsen the model performance.

References:

1. Wilson, A. C., Roelofs, R., Stern, M., Srebro†, N., & Recht, B. (n.d.). *The Marginal Value of Adaptive Gradient Methods in Machine Learning*(Rep.).
2. Brownlee, J. (2017, July 05). Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Retrieved from https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/