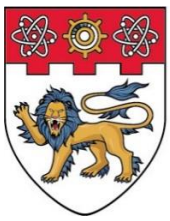

CE/CZ4042

Neural Networks and Deep Learning

Assignment 2 Report

SCSE Yew Wei Chee

U1820962G

Assignment 2 Report

CE/CZ4042 Neural Networks and Deep Learning

Yew Wei Chee

INTRODUCTION

This assignment is to help us better understand Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). In part A, an image classification model is built to predict the 10 classes of the images in CIFAR-10 dataset. Several experiments are carried out on the number of channels for each convolutional layer and on different optimizers, to compare the model performance. In part B, a text classification model is built to predict the 15 categories of the first paragraphs collected from Wikipage entries. The performance of models implemented using CNN and RNN on character and word levels are compared.

1. Part A: Object Detection

1.1 Methods

1.1.1 Data Preprocessing

The original CIFAR-10 dataset [1] consists of 50000 training images and 10000 testing images, labelled with classes from 0 to 9. In this assignment, we extract and use only 20% of the dataset. The batch_1 of the dataset is extracted out, which consists of 10000 training samples and is stored in a pickle file *'data_batch_1'*. 2000 test samples are also extracted and stored in another pickle file *'test_batch_trim'*.

When images are loaded from the pickle file, they are in the form of 1D array with a length of 3072. For each image, the 1D array is then reshaped and transposed into (32, 32, 3) where 32 is the height and width of the image, and 3 is the RGB channels of the image. The visualization is shown in Figure 1.1.1(a). Each value of the array ranges from 0 to 255, which is then scaled by dividing them by 255 so that the new values ranges from 0 to 1.

1.1.2 Convolutional and Pooling Layers

The model created have 2 convolutional layers. The first convolutional layer has a window of size 9x9 whereas the second convolutional layer has a window size of 5x5. Both layers used valid padding and ReLU activation.

Each convolutional layer is immediately followed by a pooling layer with a pooling window of size 2x2 with a stride 2, and valid padding used.

1.2 Experiments and Results

In the following experiments, the batch size used is 128 and the learning rate is 0.001.

1.2.1 Base Model

Firstly, we create a model with 50 channels for the first convolutional layer and 60 channels for the second convolutional layer. Then we flatten the output of the second max pooling layer and pass it into a dense hidden layer of 300 units before passing it to the output layer. The base model summary is shown in Figure 1.2.1(a) and Table 1.2.1(a). From the summary we can see that most of the parameters come from the dense layers.

Layer	Output Shape	Param #
Input	(32, 32, 3)	0
C1 (Conv2D)	(24, 24, 50)	12200
S1 (MaxPooling2D)	(12, 12, 50)	0
C2 (Conv2D)	(8, 8, 60)	75060
S2 (MaxPooling2D)	(4, 4, 60)	0
Flatten	960	0
Dense	300	288300
Output (Dense)	10	3010

Table 1.2.1(a): Base model summary

The training and testing accuracies and loss plots are shown in Figure 1.2.1(b) and Figure 1.2.1(c). From the plots we can see that the test accuracy and test loss converge around 800 epochs, reaching a value of around 0.55 for test accuracy and 1.3 for test loss.

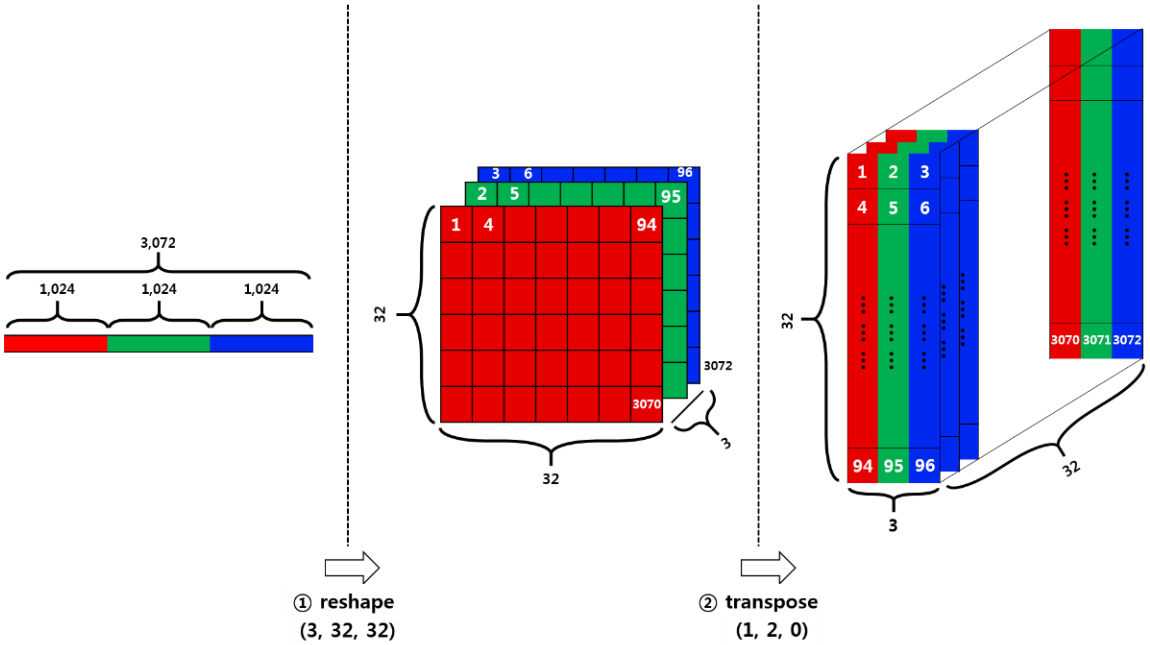

Figure 1.1.1(a): Reshape and transposing image to correct shape [2]

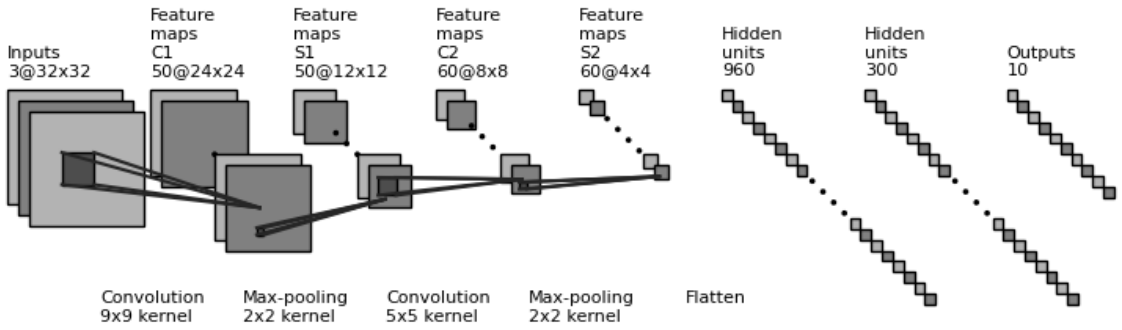

Figure 1.2.1(a): Base model architecture [3]

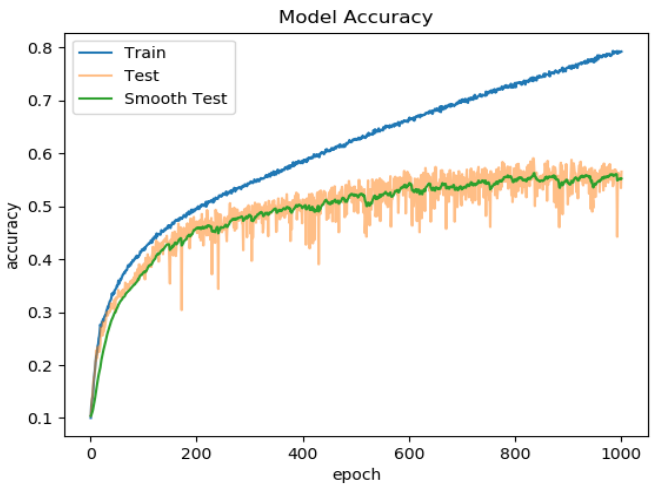

Figure 1.2.1(b): Model accuracy against epochs

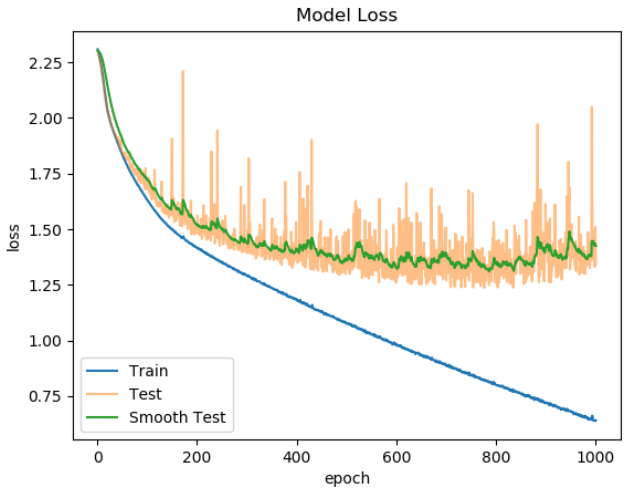

Figure 1.2.1(c): Model loss against epochs

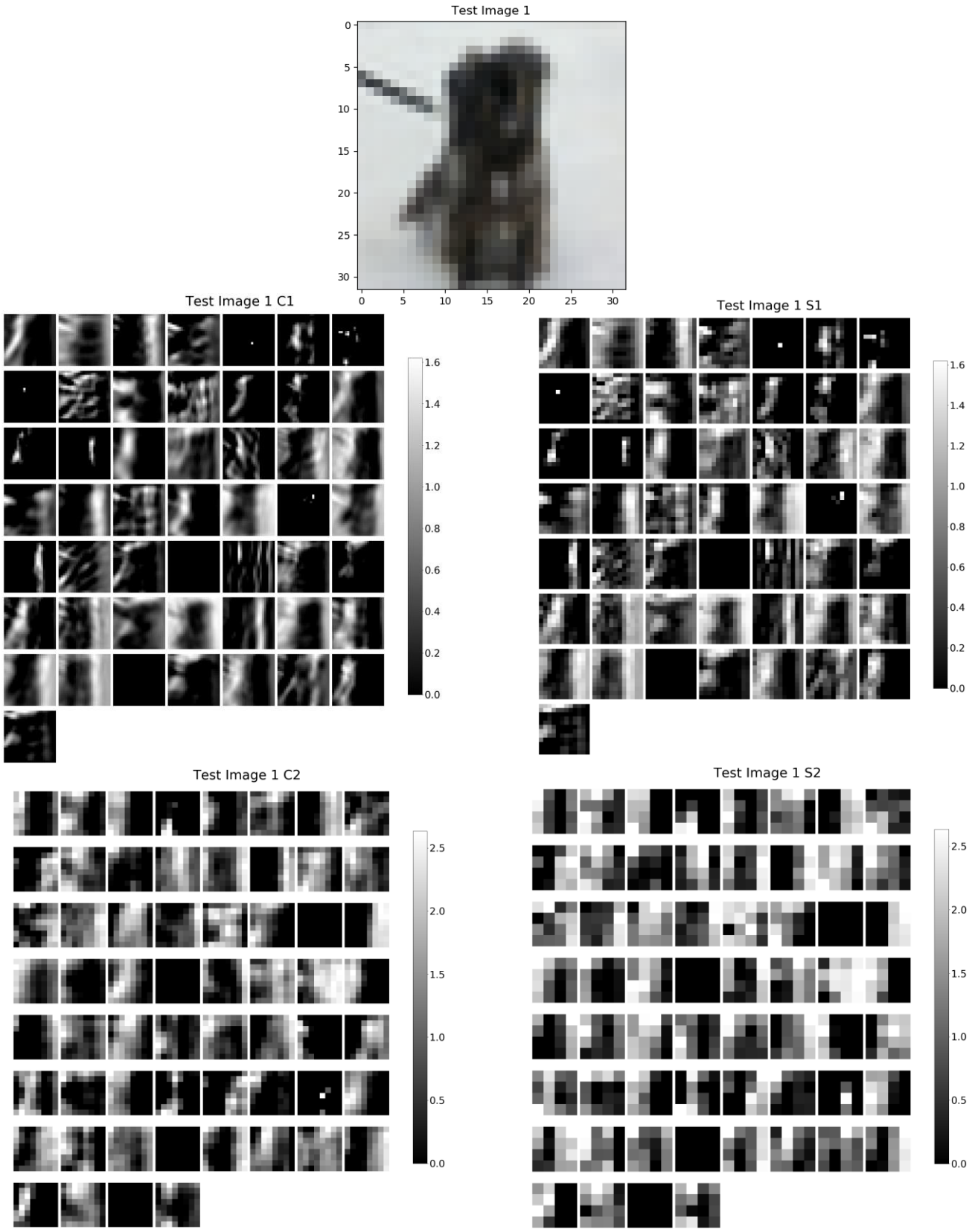

Figure 1.2.1(d): Test image 1 and its feature maps

Figure 1.2.1(e): Test image 2 and its feature maps

From Figure 1.2.1(b) and Figure 1.2.1(c), we can see that the model shows overfitting behaviour as the training accuracies keep increasing but the testing accuracies remain at around 0.55, also the training losses keep decreasing but the testing losses decrease until around 1.3 and slightly increments after that. This might be due to the lack of training data as we are only using one batch of the original dataset 12000 images

The features maps of the first 2 test images at both the convolutional layers and the pooling layers are shown in Figure 1.2.1(d) and Figure 1.2.1(e). [4] We can observe some similarities between the feature maps of the first convolutional layer C1 and the test images, showing that C1 managed to capture some low level features such as edges and shapes of the object. The receptive field at this layer is 9x9. The max pooling layer S1 then reduce the dimension of the output of C1 from (24, 24, 50) to (12, 12, 50), which helps the model to be slightly invariant to small translations of the input. The second convolutional layer C2 takes in the output from S1. The feature maps of C2 is not so similar to the test images as it began to capture some high level features from the low level features captured by C1. The receptive field at this layer is 18x18. The max pooling layer S2 then again reduce the dimension of the output of C2 from (8, 8, 60) to (4, 4, 60). The whole combination of the 2 convolutional layers and the 2 max pooling layers manage to reduce the number of neurons from 3072 to 960, which reduces computational time a lot.

1.2.2 Grid Search

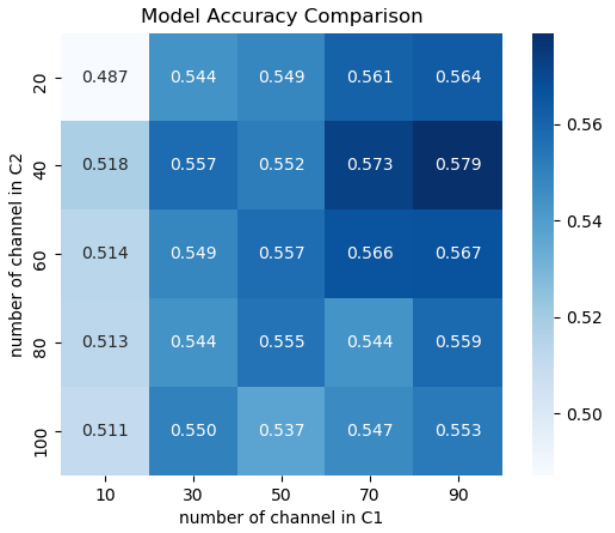

Figure 1.2.2(a): Model accuracies for different combinations of the number of channels in C1 and C2

A grid search is performed to find out the optimal combination of the number of channels at the convolution layers. For C1, the number of channels to be tested is {10, 30, 50, 70, 90}, whereas for C2, the number of channels to be tested is {20, 40, 60, 80, 100}. The results are shown in Figure 1.2.2(a).

From Figure 1.2.2(a), we can see that as the number of channels in C1 increases, the model accuracy also increases. Whereas for the number of channels in C2, 40 achieves the highest accuracy across different number of channels in C1. The optimal combination is chosen as C1=90 and C2=40 as the test accuracy is the highest among all other combinations.

1.2.3 Optimizer, Momentum and Dropout

Next, model is created using the optimal number of channels for C1 and C2 found previously. Several experiments are carried out using different optimizers, momentum and dropout, and their performance are compared.

1.2.3.1 SGD with Momentum

A momentum term of $\gamma = 0.1$ is added to the original mini-batch gradient descent. The accuracy and loss against epoch is plotted as shown in Figure 1.2.3.1(a) and Figure 1.2.3.1(b) respectively.

Similar to the base model, the test accuracy and loss converges at around 800 epochs, reaching a value of 0.55 for test accuracy and 1.3 for test loss.

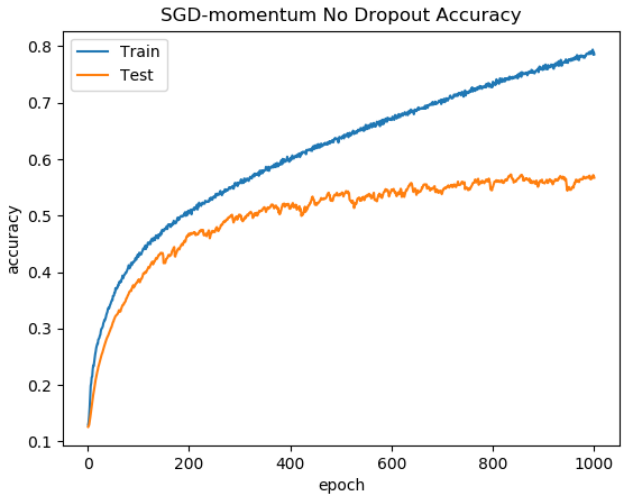

Figure 1.2.3.1(a): Accuracy against epoch for model trained using SGD optimizer with a momentum term of 0.1

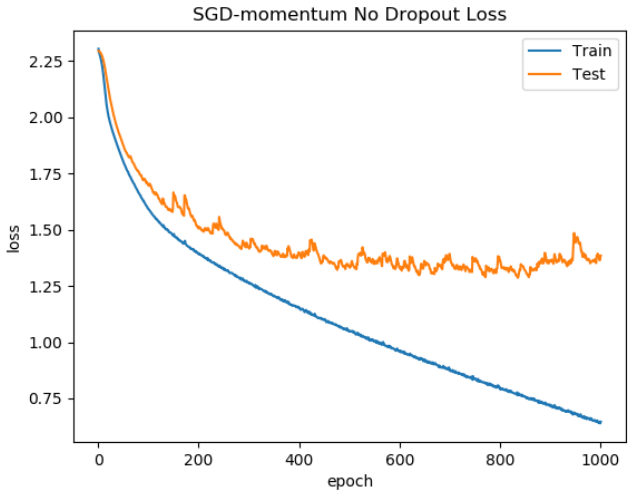

Figure 1.2.3.1(b): Loss against epoch for model trained using SGD optimizer with a momentum term of 0.1

1.2.3.2 RMSprop

RMSprop optimizer is used to train the model and the accuracy and loss against epoch is plotted as shown in Figure 1.2.3.2(a) and Figure 1.2.3.2(b) respectively.

We can see that the test accuracy converges in less than 100 epochs to a value of around 0.5. However, the test loss exhibits abnormal behaviour of keep increasing after only a few epochs. This shows that the network is strongly overfitted. The train accuracy has already approached 1 but the test accuracy is only 0.5. As the model keeps training, the model overfits more severely, causing the test loss to keep increasing.

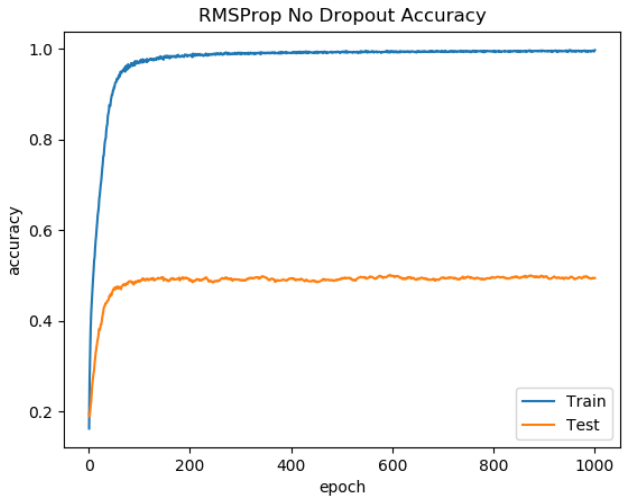

Figure 1.2.3.2(a): Accuracy against epoch for model trained using RMSprop optimizer

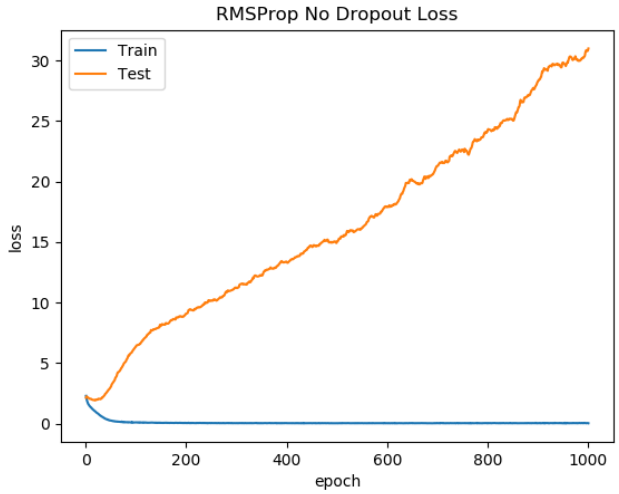

Figure 1.2.3.2(b): Loss against epoch for model trained using RMSprop optimizer

1.2.3.3 Adam

Adam optimizer is used to train the model and the accuracy and loss against epoch is plotted as shown in Figure 1.2.3.3(a) and Figure 1.2.3.3(b) respectively.

From the plots, we can see that the final test accuracy is around 0.47. Whereas the test loss also exhibits abnormal behaviour of keep increasing and fluctuating. We can also observe spikes in the training loss and accuracies, these might be due to gradient explosions when the optimizer is traversing the loss curve.

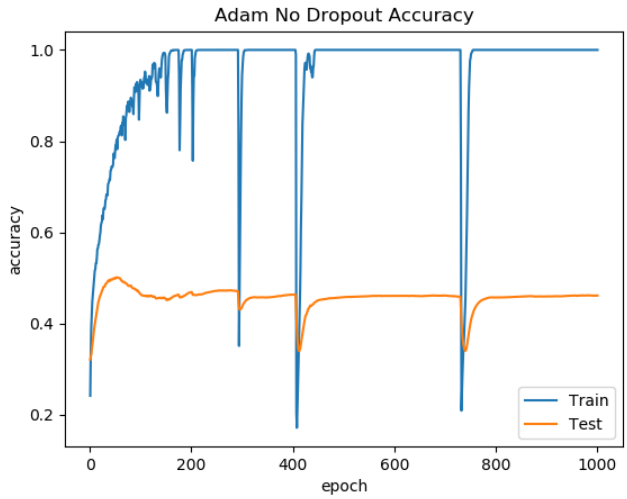

Figure 1.2.3.3(a): Accuracy against epoch for model trained using Adam optimizer

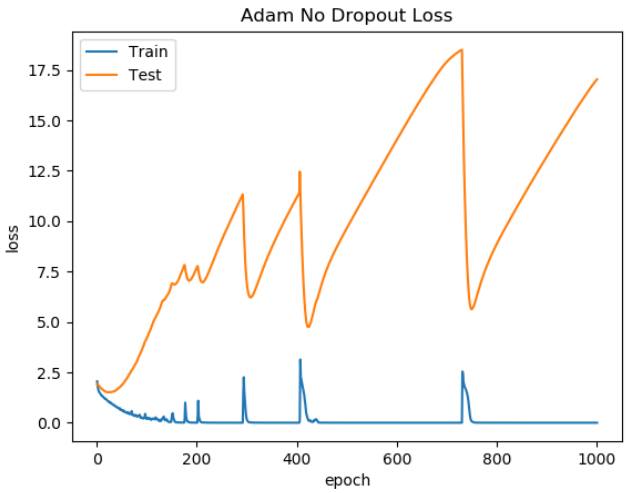

Figure 1.2.3.3(b): Loss against epoch for model trained using Adam optimizer

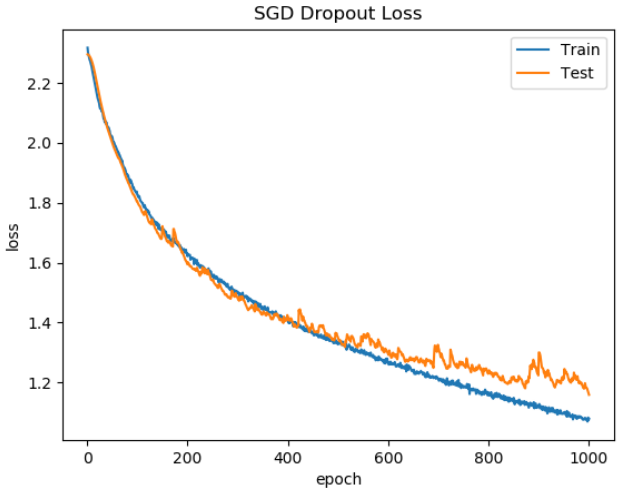

Figure 1.2.3.4(b): Loss against epoch for model trained with dropout layers

1.2.3.4 Dropout

Without changing the optimizer used previously (SGD), we now add a dropout layer with a drop rate of 0.5 to both the fully connected layers. The accuracy and loss against epoch are plotted as shown in Figure 1.2.3.4(a) and Figure 1.2.3.4(b) respectively.

We can observe that after adding the dropout layers, the train accuracy and test accuracy increases at the same rate, also the train loss and test loss decreases at the same rate. This proves that the dropout layer manages to reduce the overfitting problem in the previous model effectively. The test accuracy manages to rise until around 0.59 whereas the test loss manages to drop until around 1.2, which is an improvement compared to the previous models.

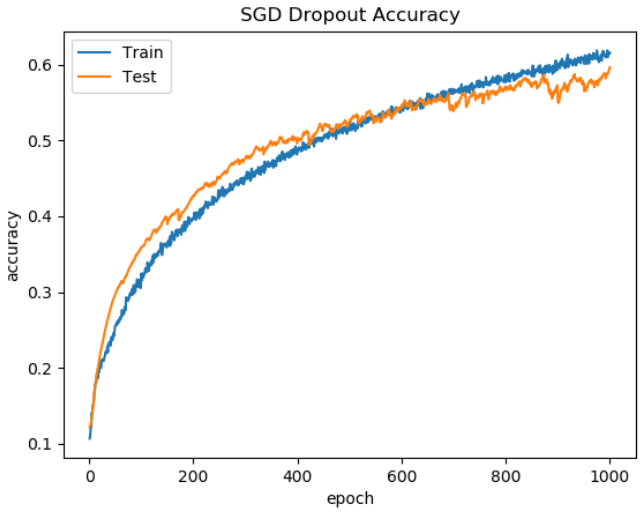

Figure 1.2.3.4(a): Accuracy against epoch for model trained with dropout layers

1.2.3.5 Comparison

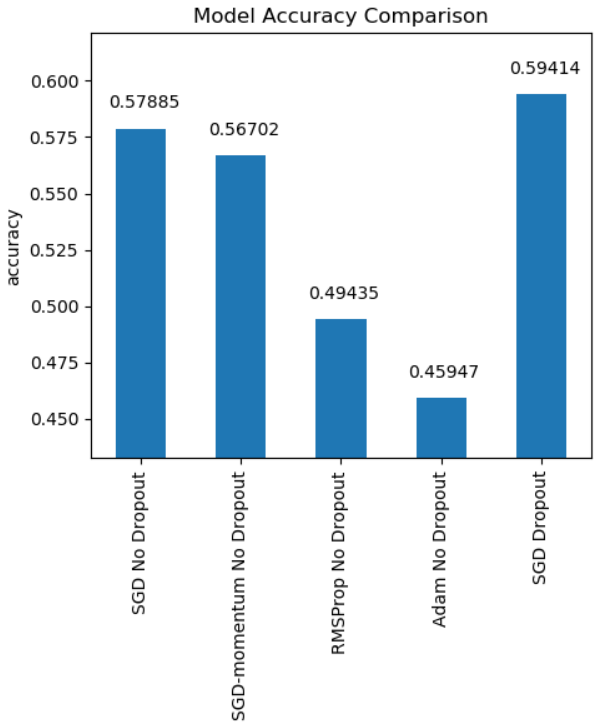

Figure 1.2.3.5(a): Model accuracy comparison

Comparing between all optimizers, we observed that SGD optimizer produces model that gives the highest test accuracy compared to RMSprop and Adam in this case. RMSprop is more stable than Adam in this case as it manages to avoid gradient exploding situations leading to smoother testing accuracy curve.

SGD with momentum in this case does not help the model to converge faster, as shown in the plot in Figure 1.2.3.5(b), this may be due to the small value of momentum term, which is insufficient to accelerate gradient vectors in the right direction.

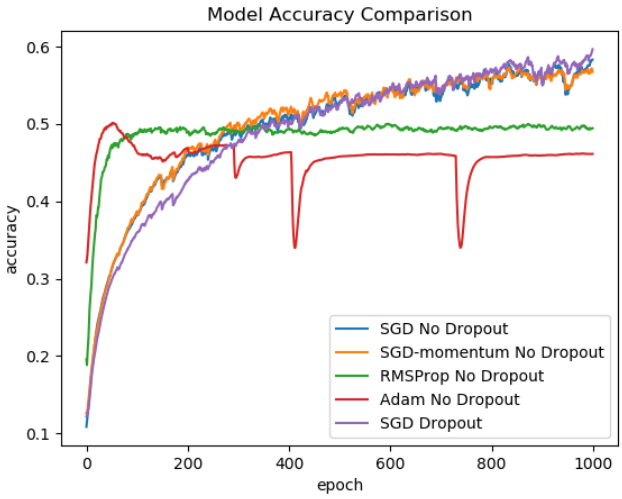

Figure 1.2.3.5(b): Model accuracy against epoch

Besides, we can see that model trained with dropout performs slightly better than other models. However, the convergence rate of the model using dropout is much slower, from the plots we can see that the accuracies and loss for model using dropout haven't fully converged.

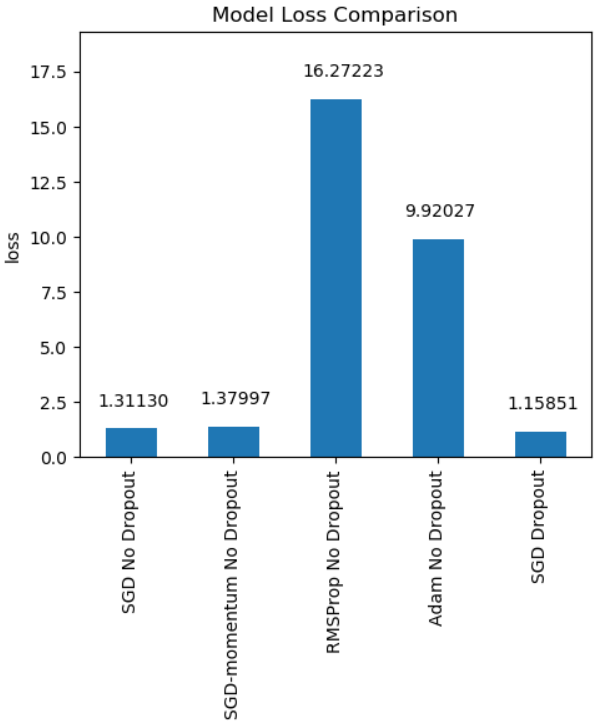

Figure 1.2.3.5(c): Model loss comparison

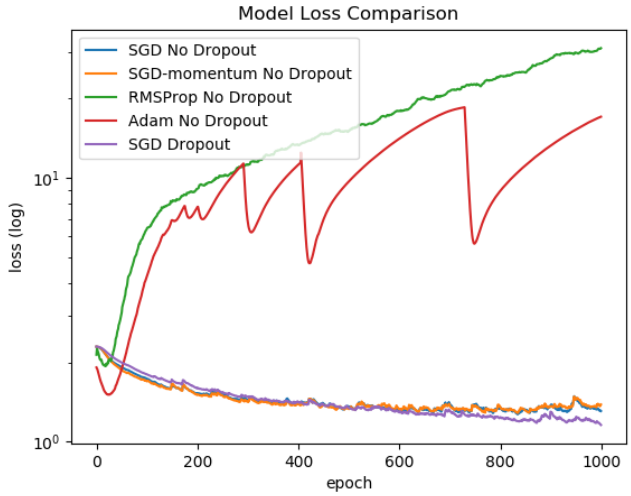

Figure 1.2.3.5(d): Model loss against epoch

Hence, we can conclude that dropout does help improve model performance for model trained using small dataset. Also, we can see that no matter which optimizer is used, the model trained using small dataset overfits easily. Therefore, to further improve on the generalization of the model, we need to use a larger dataset.

Once we have a larger dataset, we can improve on the model by having a deeper network by increasing the number of convolutional layers and max pooling layers. Also having large kernels increased the number of parameters and computation needed to update the weights. We can reduce the kernel size of each layer so as to reduce the number of parameters and make training faster. A 5x5 convolution can be replaced with two 3x3 convolution stacked together, resulting in the same receptive field but with lower number of parameters and a deeper network [5] [6].

2. Part B: Text Classification

2.1 Methods

2.1.1 Data Preprocessing

We are given a dataset of paragraphs of different length collected from Wikipage entries, each of them is labelled with one of the 15 labels. In total, there are 5600 paragraphs for training and 700 paragraphs for testing. The paragraphs are preprocessed at character level and at word level.

For character level preprocessing, firstly we remove any other characters other than lowercase and uppercase letters, numbers, commas, periods, single quotes (apostrophe) and spaces. Next, we set the maximum number of characters to

be 100. For each paragraph that has more than 100 characters, we truncate the characters that are after the 100th character. For paragraphs that has less than 100 characters, we append space character to the end of the paragraph until the number of characters is 100. Then we encode each of the characters into an integer that represent that character.

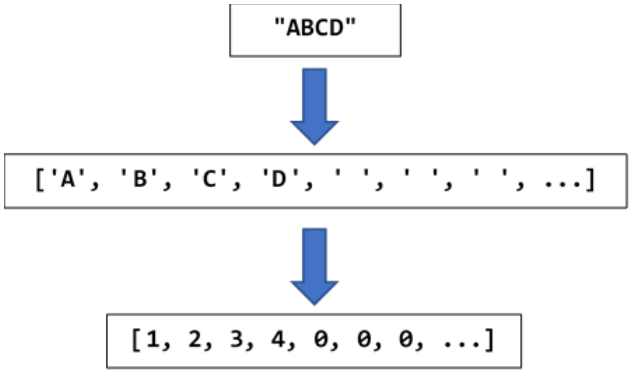

Figure 2.1.1(a): Preprocessing at character level

For word level processing, firstly we remove any unwanted characters from the string and convert all letters to lowercase. Then we tokenize the string into word tokens using the *nltk.word_tokenize* function from the *nltk* library [7]. Next, we encode each word into a unique integer according to the frequency of occurrence of that word. A word that appears more frequently gets encode with a smaller integer. Besides, we introduced 3 special tokens: *"unk"* token to represent words not in the dictionary; *"pad"* token to represent padded tokens to fulfil the required length; *"eos"* token that is added to the end of each paragraphs representing the end of a string. Similar to character level preprocessing, we set the maximum number of tokens of each paragraph to be 100. For those paragraphs that have more than 100 tokens, the tokens that appeared after the 100th token are removed. For paragraphs that have less than 100 tokens, then encoding of the *"pad"* token is added to the end until the length is met.

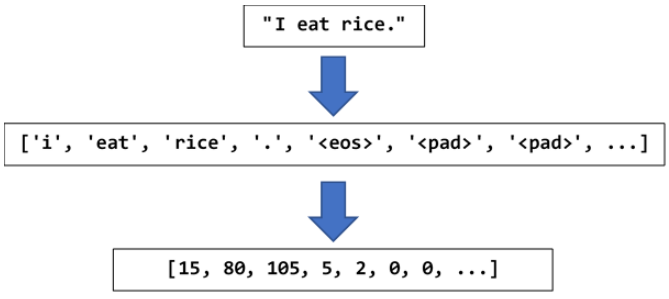

Figure 2.1.1(b): Preprocessing at word level

2.1.2 Word Embeddings

For character level neural network, the characters are encoded as integers. When passed as input to the neural network, the integers are first converted to one-hot vectors of size 256 first before being feed into the neural network.

For word level neural network, instead of converting the integers that representing words into one-hot vectors, we convert them into word embeddings which are trainable instead. Word embeddings are vectors that represent words such that words that have similar context will appear close to each other in the embedding space. In this assignment, the embedding layer used have a size of 20.

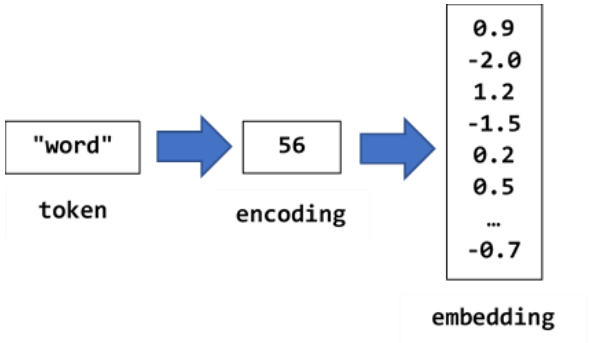

Figure 2.1.2(a): Word embedding

2.1.3 Recurrent Layer

Recurrent layer is composed of cell units that stores internal states and they can use these states to process variable length sequences of inputs. In this assignment, we will use 3 types of recurrent layer which are Vanilla RNN, LSTM and GRU.

Vanilla RNN is the basic RNN cell whereby the output of the hidden state is pass back in as the input to hidden state at the next time step. The problem with vanilla RNN is that it suffers from gradient vanishing problem during backpropagation through time, which causes the weights to be updated based on the recent few timesteps only and not able to learn the dependencies between current context and context from longer timesteps away.

LSTM and GRU reduces the gradient vanishing problem by introducing gates which selectively preserve or remove information from the previous timesteps. This enables the gradient to be able to propagate through many timesteps, allowing the weights to be updated correctly to learn the dependencies between current context and context from longer timesteps away. LSTM is controlled by 3 gates whereas GRU is controlled with 2 gates. Therefore, LSTM has more parameters and train slower than GRU.

2.2 Experiments and Results

In the following experiments, the models are trained using Adam optimizer, with a batch size of 128 and learning rate of 0.1.

2.2.1 Character Level CNN

A CNN classifier is created to take in the character ids as input and classify them to one of the 15 categories. The first convolution layer has a filter window of size 20x256, whereas the second convolution layer has a filter window of size 20x1. Both convolution layers have valid padding and ReLU activation, each is followed by a max pooling layer with window size 4x4, stride 2 and same padding. The accuracy and loss of the model are plotted in Figure 2.2.1(a) and Figure 2.2.1(b).

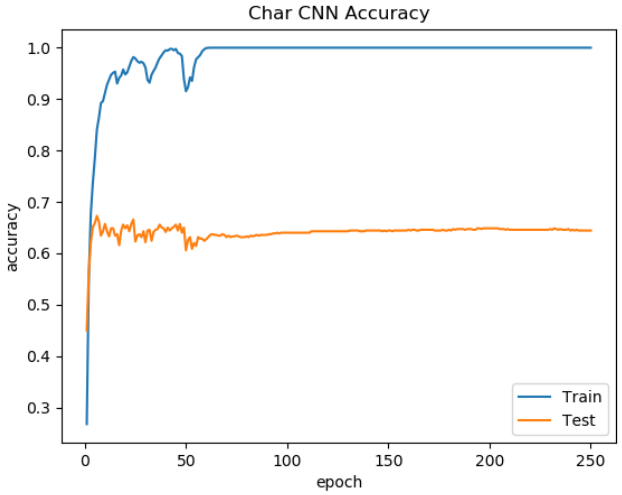

Figure 2.2.1(a): Character CNN accuracy against epoch

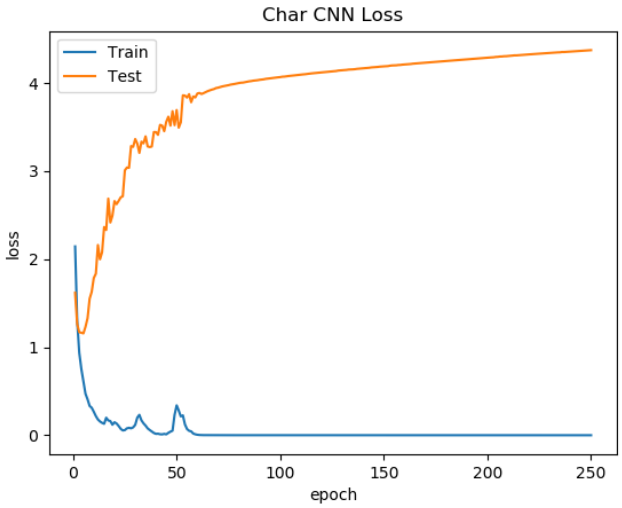

Figure 2.2.1(b): Character CNN loss against epoch

2.2.2 Word Level CNN

A CNN classifier is created to take in the word ids as input and classify them to one of the 15 categories. The first convolution layer has a filter window of size 20x20, whereas the second convolution layer has a filter window of size 20x1. Both convolution layers have valid padding and ReLU activation, each is followed by a max pooling layer with window size 4x4, stride 2 and same padding. The accuracy and loss of the model are plotted in Figure 2.2.2(a) and Figure 2.2.2(b).

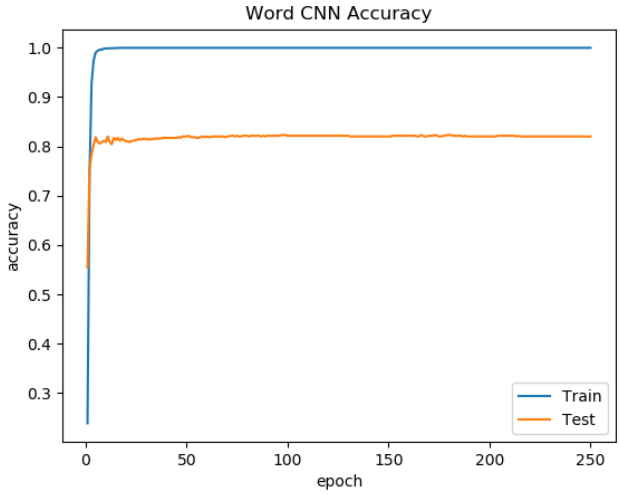

Figure 2.2.2(a): Word CNN accuracy against epoch

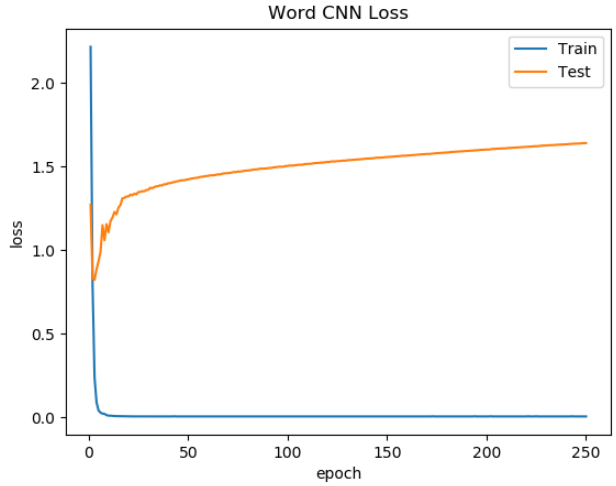

Figure 2.2.2(b): Word CNN loss against epoch

2.2.3 Character Level RNN (GRU)

An RNN classifier is created to take in the character ids as input and classify them to one of the 15 categories. The RNN uses GRU cells and has a hidden layer of size 20. The accuracy and loss of the model are plotted in Figure 2.2.3(a) and Figure 2.2.3(b).

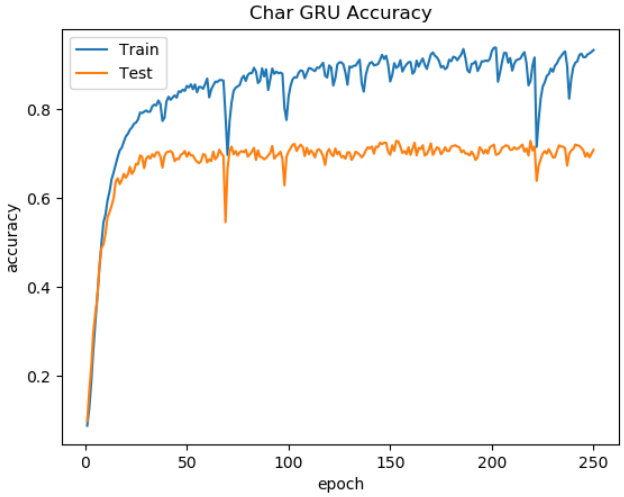

Figure 2.2.3(a): Character RNN accuracy against epoch

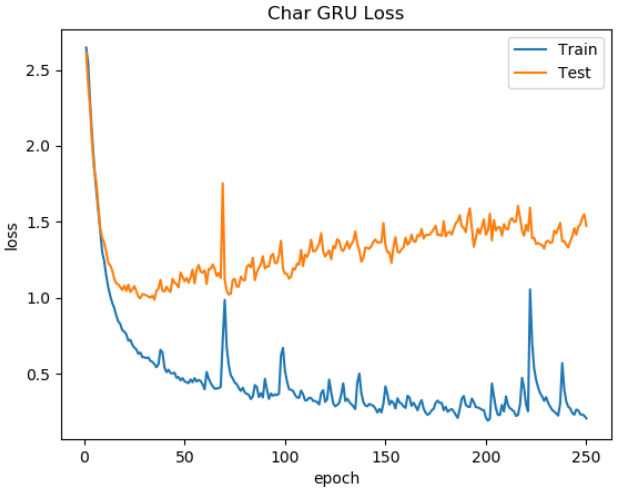

Figure 2.2.3(b): Character RNN loss against epoch

2.2.4 Word Level RNN (GRU)

An RNN classifier is created to take in the word ids as input and classify them to one of the 15 categories. The RNN uses GRU cells and has a hidden layer of size 20. The accuracy and loss of the model are plotted in Figure 2.2.4(a) and Figure 2.2.4(b).

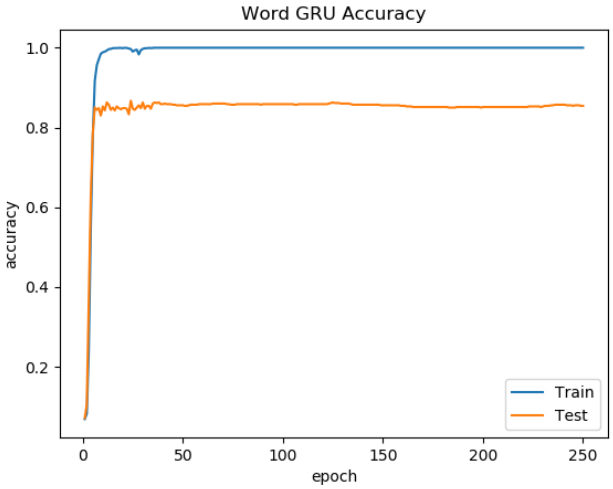

Figure 2.2.4(a): Word RNN accuracy against epoch

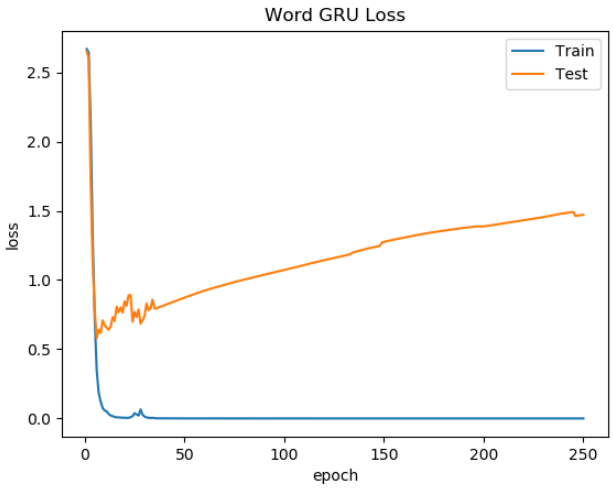

Figure 2.2.4(b): Word RNN loss against epoch

2.2.5 CNN and RNN (GRU) Model Comparison

The 4 models above are trained again but with dropout layers added after every convolution or recurrent layers. The drop rate used is 0.5.

From the plots previously, we can observe that all models are experiencing overfitting. This is because the testing loss began to increase after dropping for a few epochs. We can also see that CNN models overfit much more compared to RNN models as the test losses increased much more severely after reaching a minimum point.

We can see that models taking input as word ids has test accuracies of around 0.8 – 0.9, performing much better than the models taking input as character ids, which has test accuracies of only around 0.6 – 0.7. This might be due to character tokens are not able to capture dependencies within the paragraphs as good as word tokens. Besides,

even though both word level and character level models are taking input of 100 tokens, character level model only managed to take the first 100 characters as inputs, whereas the word level model is able to take the first 100 words as input, which is able to capture dependencies within a longer sequence. In addition, word level models are trained using word embeddings instead of one-hot vectors, which better represents the context of the words.

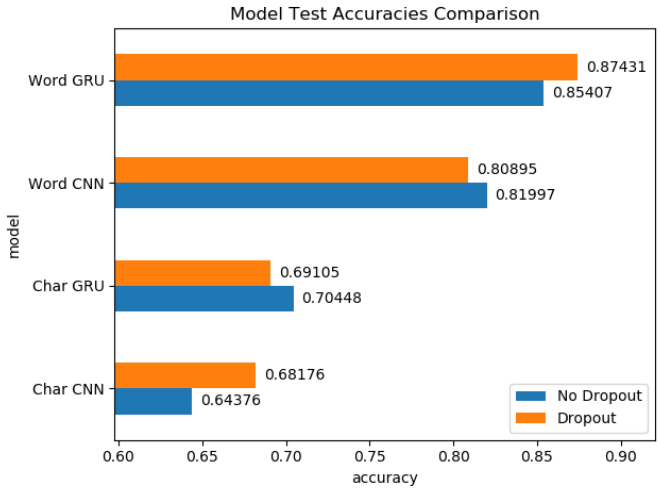

Figure 2.2.5(a): Model test accuracies comparison

For both character level and word level models, RNN performs better than CNN. This might be due to CNN can only capture the context within a specific window size, in our case is 20. This window size is not enough to capture dependencies between tokens that are more than 20 steps apart. Also, CNN treats each token in a window differently, which causes a lack of symmetry in how the inputs are processed. RNN with GRU cells are able to propagate information capture from previous timesteps for longer time and can also selectively choose information to be updated or to be forgot, which better capture dependencies within a sequence of tokens.

In general, the addition of dropout layer into the models helps to prevent overfitting. This can be clearly seen from Figure 2.2.5(c) where the test losses for models trained using dropout layer are generally lower than the models trained without dropout layer.

The time per epoch used to trained each of the models are shown in Figure 2.2.5(e). We can see that training RNN takes a much longer time compared to training CNN. This is because the computations in CNN can happen in parallel, whereas for RNN, it need to be processed sequentially. Besides, time taken to train word level model is longer than character level model, this is because word level model has an additional embedding layer to train.

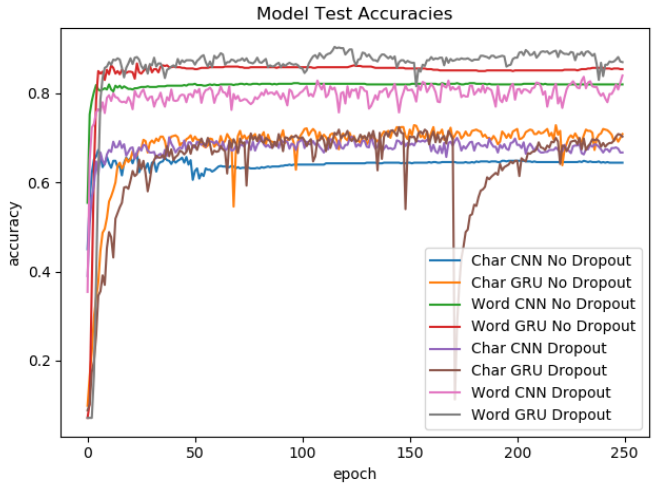

Figure 2.2.5(b): Model test accuracies against epochs

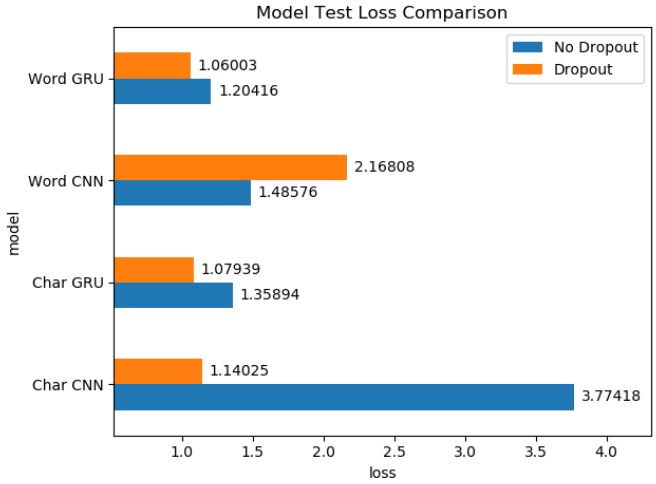

Figure 2.2.5(c): Model test losses comparison

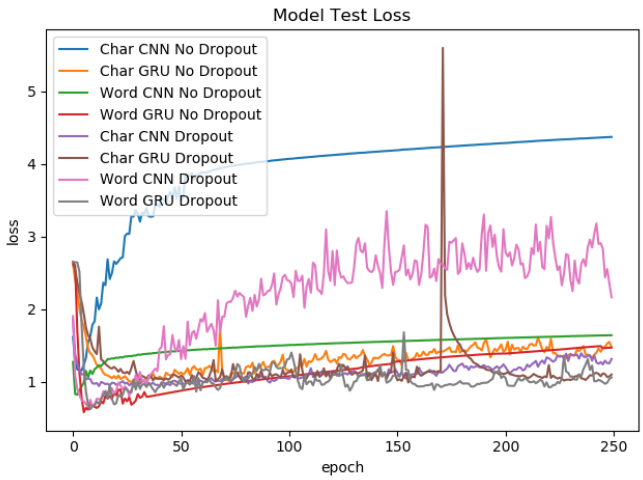

Figure 2.2.5(d): Model test losses against epochs

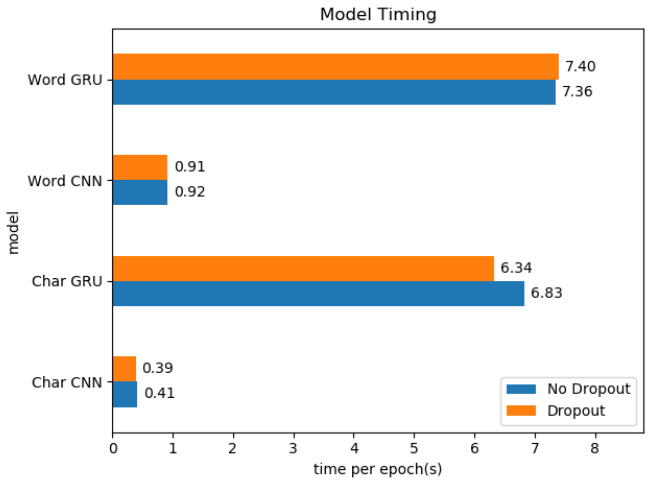

Figure 2.2.5(e): Model training time comparison

2.2.6 RNN Model Comparison

2.2.6.1 LSTM and Vanilla RNN

For both character level and word level RNN, the GRU cell are replaced with LSTM and Vanilla RNN. The size of the hidden layer remains 20. The results are shown in the figures below.

From the plots below, we can see that Vanilla RNN barely learn during training as the test accuracy remains low and the test loss remains high throughout the training period. LSTM performs better than Vanilla RNN but is not as good as GRU.

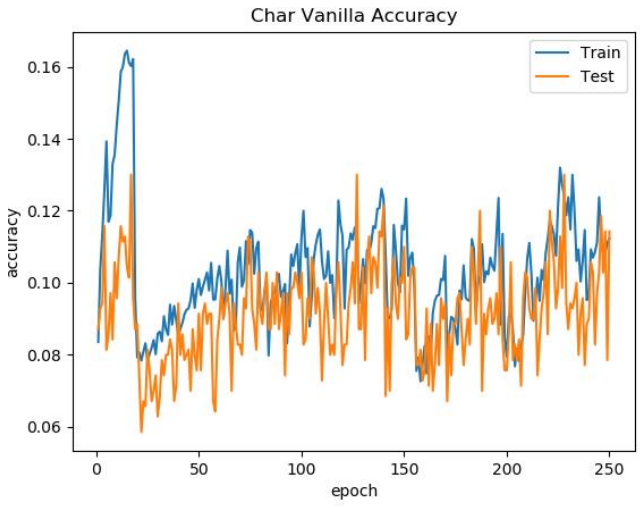

Figure 2.2.6.1(a): Character Vanilla RNN accuracy against epoch

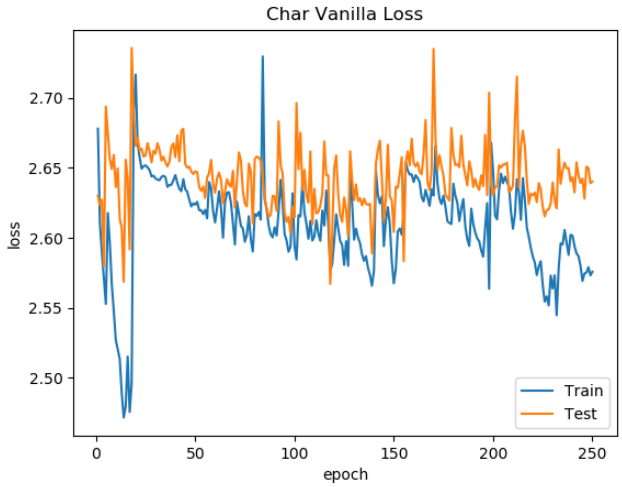

Figure 2.2.6.1(b): Character Vanilla RNN loss against epoch

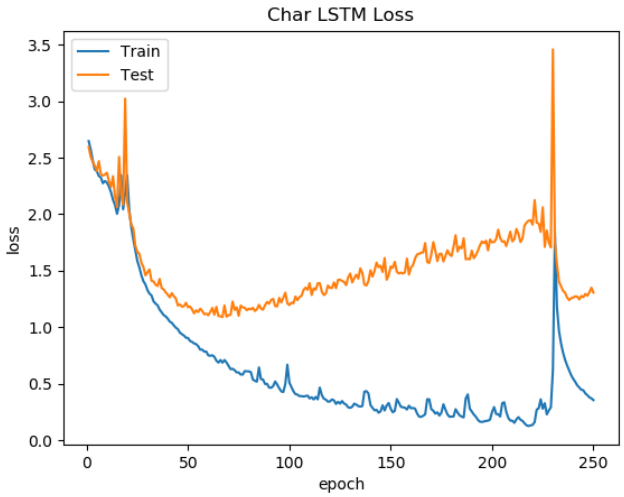

Figure 2.2.6.1(c): Character LSTM accuracy against epoch

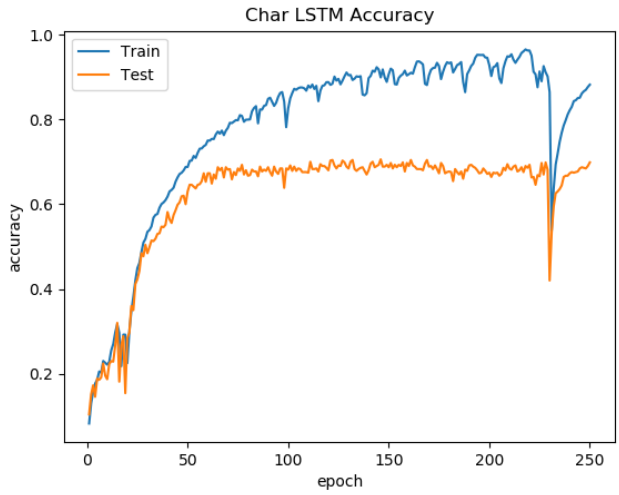

Figure 2.2.6.1(d): Character LSTM loss against epoch

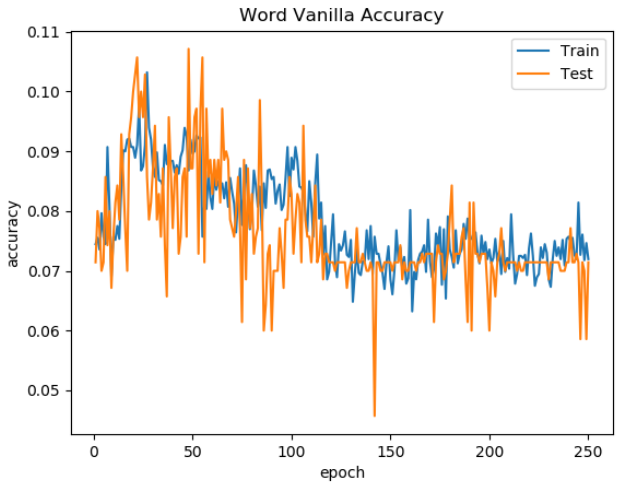

Figure 2.2.6.1(e): Word Vanilla RNN accuracy against epoch

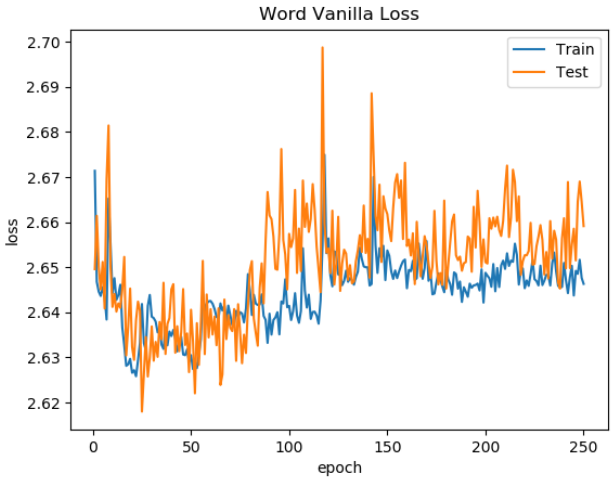

Figure 2.2.6.1(f): Word Vanilla RNN loss against epoch

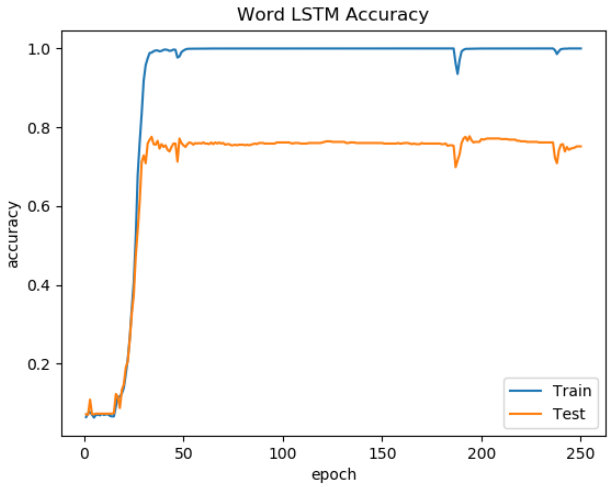

Figure 2.2.6.1(g): Word LSTM accuracy against epoch

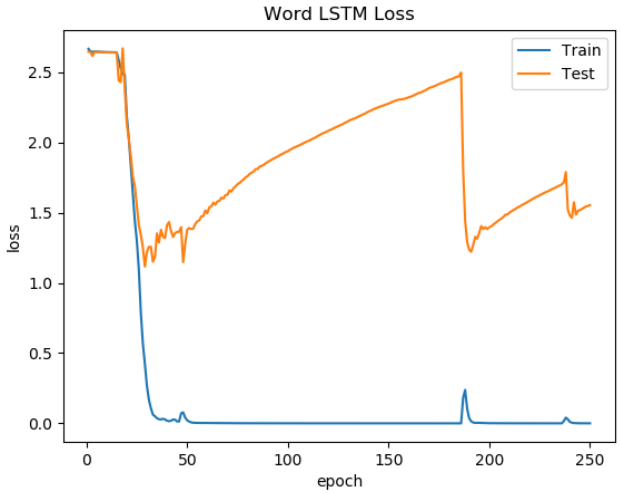

Figure 2.2.6.1(h): Word LSTM loss against epoch

2.2.6.2 2-Layer GRU

For both character level and word level RNN, we stack 2 layers of GRU together, each layer remains the size of 20. The results are shown below.

From the plots below, we can see that both character level and word level models with 2 layers of GRU performs better than with only 1 layer of GRU. Stacking layers of GRU together is just like having multiple layers of CNN, the purpose is to make the network deeper. The higher level hidden layers recombine information learned from previous layers and create higher level abstraction that better represents the information. However, stacking too much layers of GRU may cause drop in performance due to gradient exploding or vanishing during training.

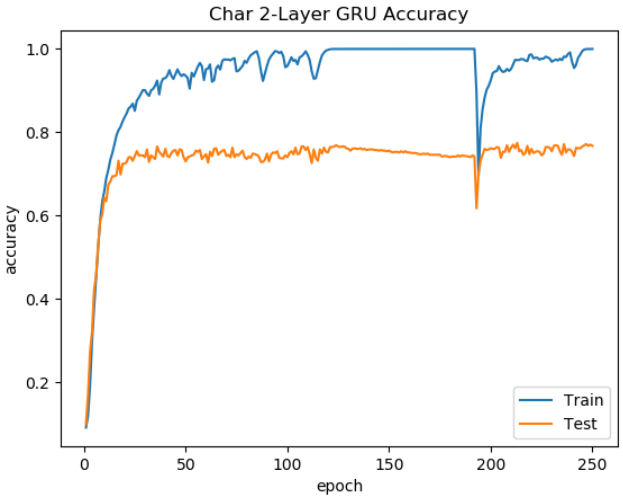

Figure 2.2.6.2(a): Character 2-layer GRU accuracy against epoch

Figure 2.2.6.2(b): Character 2-layer GRU loss against epoch

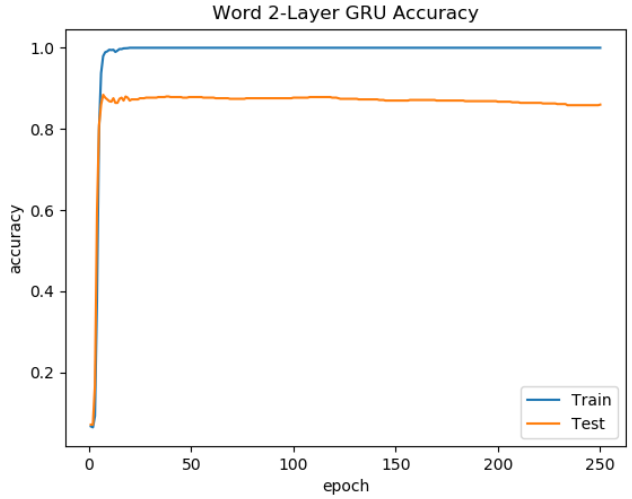

Figure 2.2.6.2(c): Word 2-layer GRU accuracy against epoch

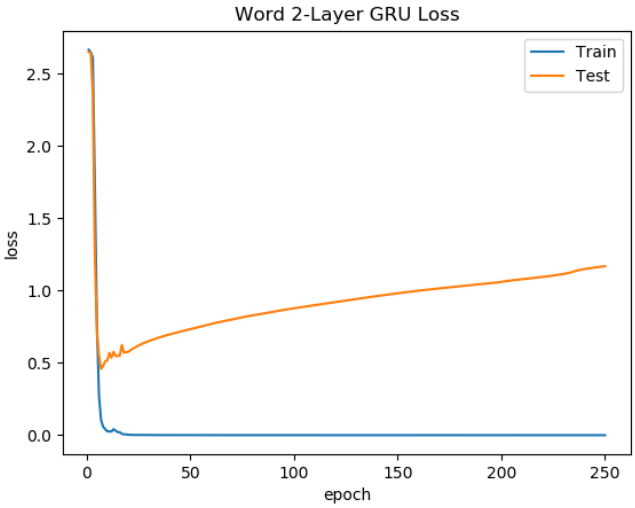

Figure 2.2.6.2(d): Word 2-layer GRU loss against epoch

2.2.6.3 Gradient Clipping

Using the same character level and word level RNN model with GRU cells, we now train with a gradient clipping threshold of 2.0. The results are plotted as below.

As compared with the previous training and testing curve of the model trained without using gradient clipping, we can observe that now the training and testing curve becomes much smoother with lesser spikes. This can be obviously seen by comparing Figure 2.2.3(a) with Figure 2.2.6.3(a) or Figure 2.2.3(b) and Figure 2.2.6.3(b).

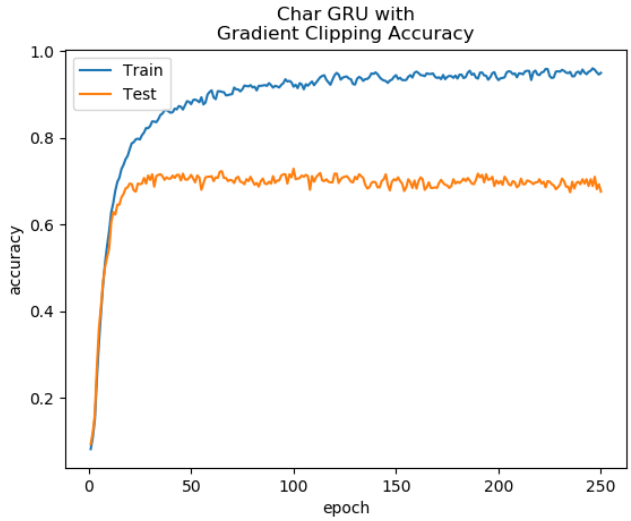

Figure 2.2.6.3(a): Accuracy of character RNN (GRU) trained with gradient clipping against epoch

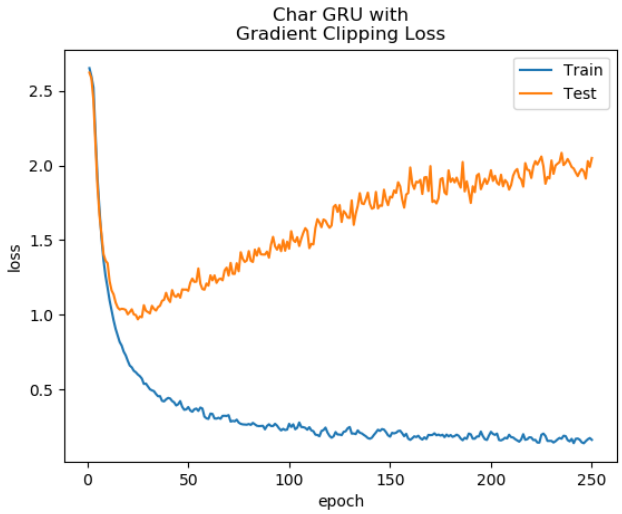

Figure 2.2.6.3(b): Loss of character RNN (GRU) trained with gradient clipping against epoch

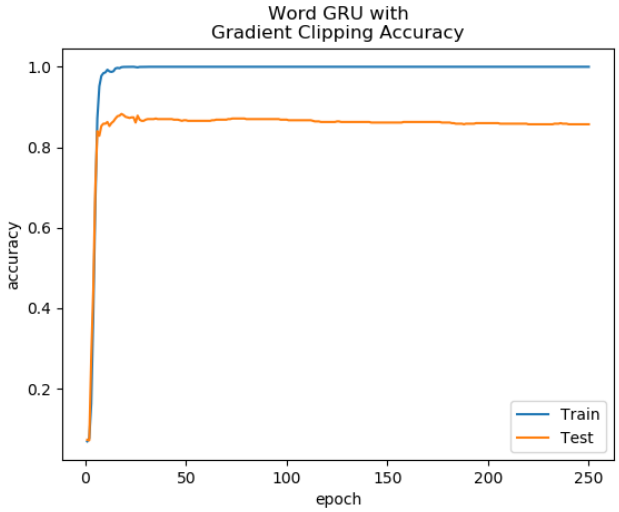

Figure 2.2.6.3(c): Accuracy of word RNN (GRU) trained with gradient clipping against epoch

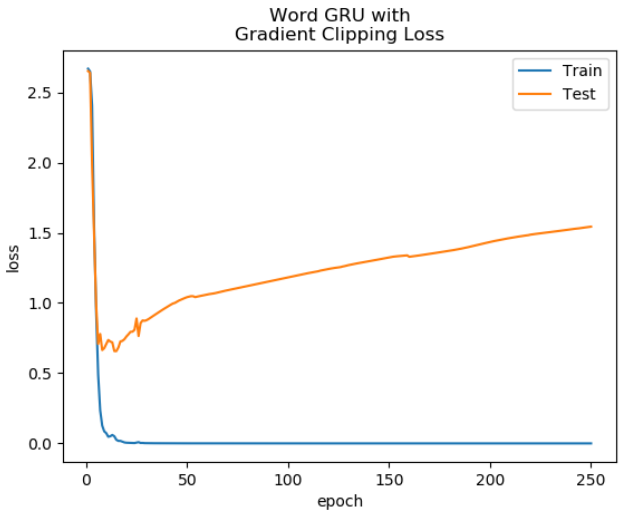

Figure 2.2.6.3(d): Loss of word RNN (GRU) trained with gradient clipping against epoch

Gradient clipping helps to prevent gradient exploding issues and makes training much smoother. The performance of the model remains almost the same.

2.2.6.4 Model Comparison

In general, we can see that 2-layer GRU performs the best out of other models whereas Vanilla RNN fails to learn anything during training. Gradient clipping helps to reduce the spikes that occur due to gradient exploding problems. Comparing word level models and character level models, we can say that word level models performed better than character level models.

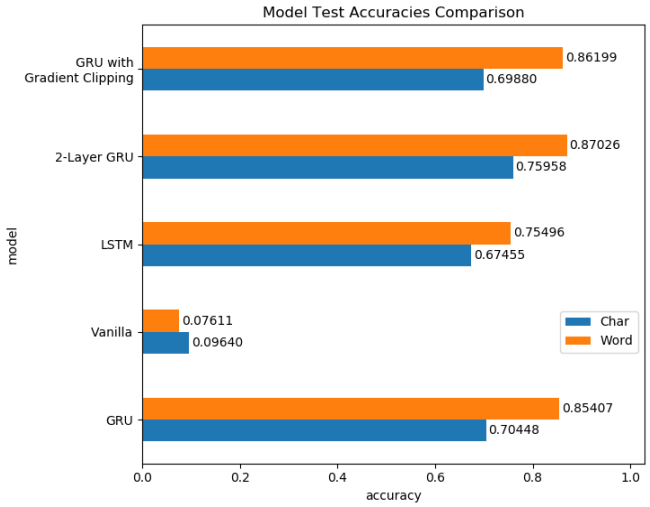

Figure 2.2.6.4(a): Model test accuracies comparison

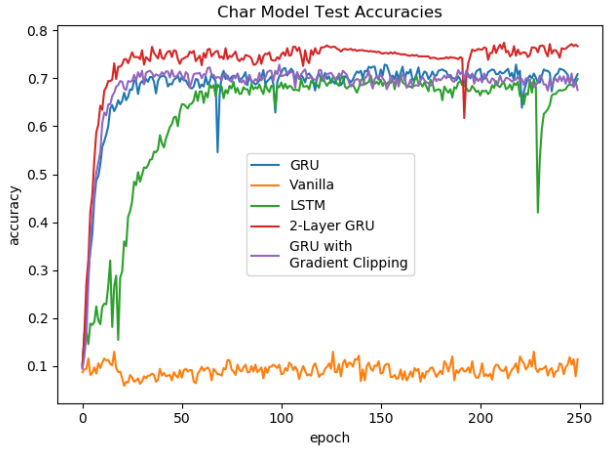

Figure 2.2.6.4(b): Character model test accuracies against epoch

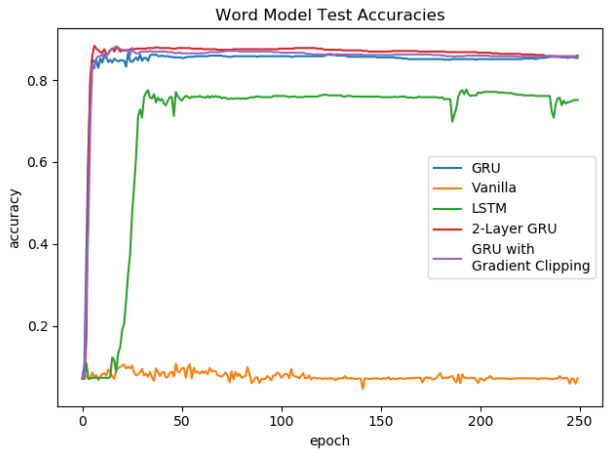

Figure 2.2.6.4(c): Word model test accuracies against epoch

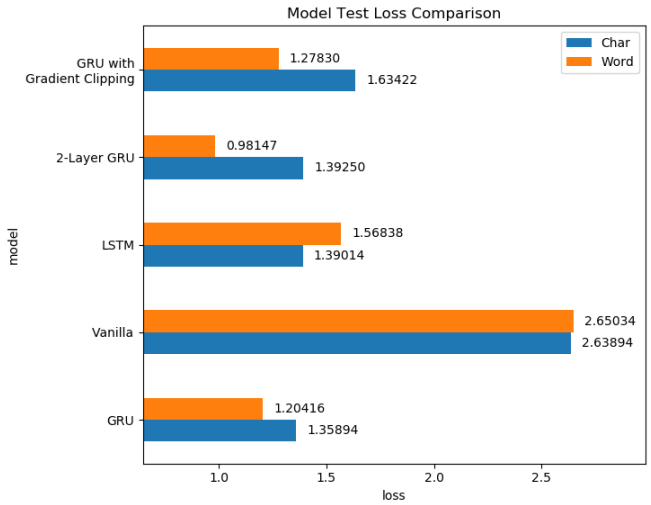

Figure 2.2.6.4(d): Model test losses comparison

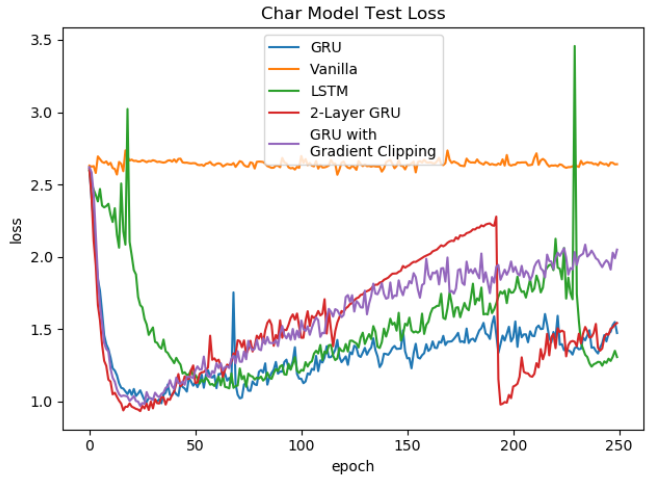

Figure 2.2.6.4(e): Character model test losses against epoch

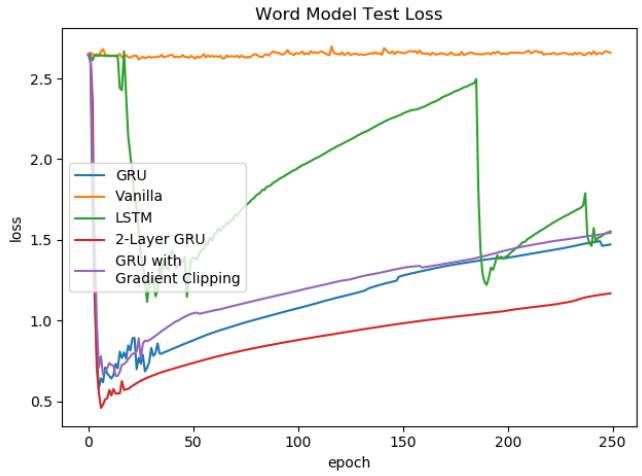

Figure 2.2.6.4(e): Word model test losses against epoch

CONCLUSION

For part A, the optimal number of channels for the first convolutional layer is 90 and for the second convolutional layer is 40. SGD optimizer performs the best in this case, followed by RMSprop and lastly Adam. Adding dropout to the dense layer can help reduce overfitting problem and improve model performance.

For part B, word level models perform better than character level models and RNN performs better than CNN in this case. Among all RNNs, GRU gives the best model performance, followed by LSTM and lastly Vanilla RNN which does not show any improvement throughout the training. Stacking GRUs together makes the RNN network deeper and helps improve model performance. Gradient clipping can help smooth the training curve and avoid gradient exploding problems.

REFERENCES

[1] *CIFAR-10 and CIFAR-100 datasets. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 30-Oct-2020].*

[2] *P. Chansung, "CIFAR-10 Image Classification in TensorFlow," Medium, 15-Jun-2018. [Online]. Available: https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c. [Accessed: 30-Oct-2020].*

[3] *Gwding, "gwding/draw_convnet," GitHub. [Online]. Available: https://github.com/gwding/draw_convnet. [Accessed: 30-Oct-2020].*

[4] *Philipperemy, "philipperemy/keract," GitHub. [Online]. Available: https://github.com/philipperemy/keract/blob/master/keract/keract.py. [Accessed: 30-Oct-2020].*

[5] *A. Chazareix, "About Convolutional Layer and Convolution Kernel," Sicara. [Online]. Available: https://www.sicara.ai/blog/2019-10-31-convolutional-layer-convolution-kernel. [Accessed: 30-Oct-2020].*

[6] *"Why are neural networks becoming deeper, but not wider?," Cross Validated [Online]. Available: https://stats.stackexchange.com/questions/222883/why-are-neural-networks-becoming-deeper-but-not-wider. [Accessed: 30-Oct-2020].*

[7] *"Natural Language Toolkit," Natural Language Toolkit - NLTK 3.5 documentation. [Online]. Available: https://www.nltk.org/. [Accessed: 30-Oct-2020].*