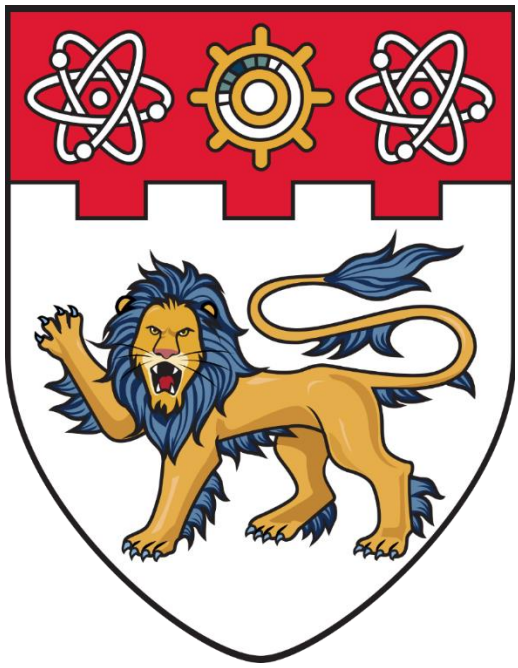


CZ 4042

Neural Networks and Deep Learning

Assignment 2 Report

Tan Chuan Xin

U1821755B

Table of Contents

Prologue

This assignment was done through Google Colaboratory Notebooks. The repository has been hosted on Github, and cloning it will be the best way to get a copy of the code to run. It can be found at the following link: https://github.com/tanchuanxin/cz4042_assignment_2

Each individual question has a notebook dedicated to it. Please refer to the relevant notebooks alongside this project report for a better understanding.

This report will be written as a companion to the notebook. Code blocks will be omitted as much as possible, as this report shall center on the rationale of various decisions made that led to the code, rather than the actual code itself.

Source code

The source codes can be found under the folders titled 2a and 2b

Part A: Object Recognition

Introduction

Part A is an object recognition problem on the CIFAR-10 dataset, which contains images in 10 classes. This is also a classification problem. The end goal is to perform a multi-class classification and output a class label corresponding to the input image.

There are four tasks given, namely

1. Implement a Convolutional Neural Network (CNN) based on parameters that were provided
2. Perform a grid search to find optimal channels for the CNN
3. Experiment with various optimization techniques to improve performance
4. Compare every model created and discuss their performance

The primary goal of this problem is to discover a set of hyperparameters that are optimal to classify the images in the CIFAR-10 dataset when using CNNs.

Methods

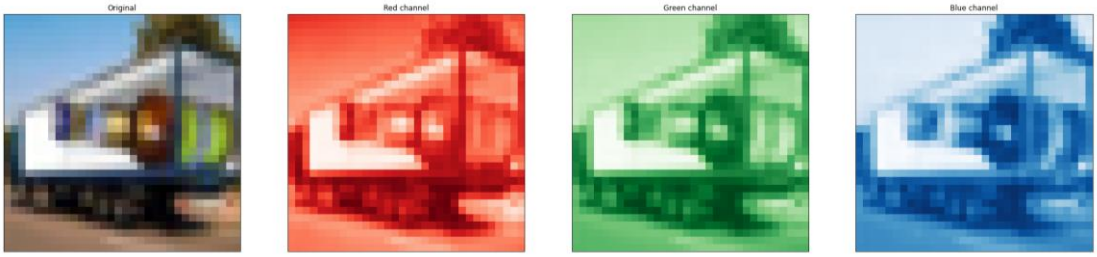
There are several primary techniques that are common throughout the notebooks used for Part A. We shall address why these methods were necessary, and hence chosen.

Data Preparation

Some changes were made to the `load_data(file)` function in the starter code. When trying to plot out the source image for inspection, random noise was plotted out instead. It was discovered that there was a problem with the reshaping of the source data into proper images. Therefore, this line was added to ensure proper reshaping into a 32x32x3 dimension image.

```
data = data.reshape(len(data), 3, 32, 32).transpose(0, 2, 3, 1)
```

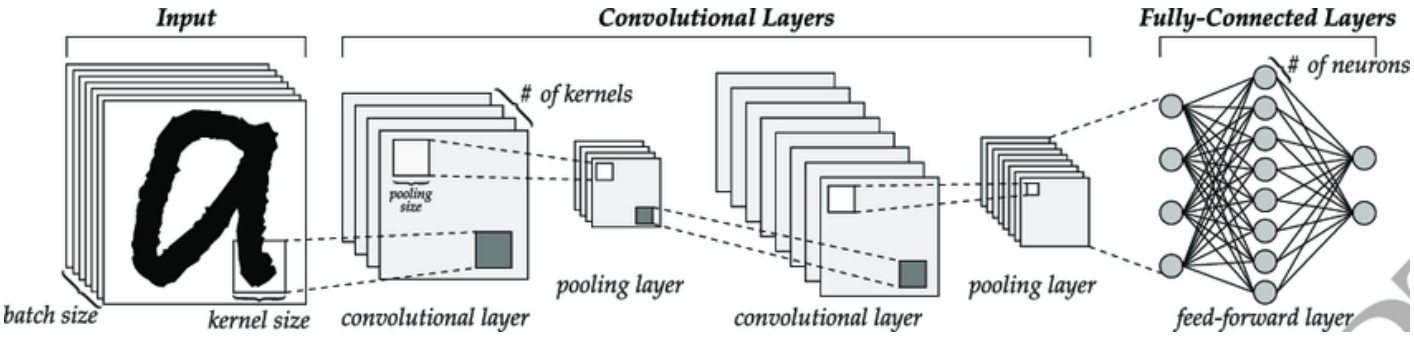
Before, the numpy array representations of the images were incorrectly prepared. Feeding these into our model provided incoherent predictions and very random intermediate layer activations. Therefore, it was important that the data was prepared correctly. Now, we are able to output the images as they were meant to be.



Convolutional Neural Networks (CNN)

CNNs are a powerful form of feedforward neural networks, especially when it comes to tasks related to computer vision. CNNs are defined in a large part with convolutional layers and pooling layers.

Convolutional layers perform convolutional operations that involve taking a kernel (essentially a set of weights) and passing it across the entire input matrix to produce a feature map. Multiple kernels of varying sizes can be passed through the input matrix to produce various feature maps, that may correspond to certain characteristics of the input matrix. Feature maps are then passed through pooling layers, which perform local interpolation on feature maps to select just a few values from them, such as maxpooling, which takes the maximum value from a window of values.



The number of channels in a convolutional layer represent the number of kernels that are applied on the input matrix, which produces an equal number of feature maps for the image. Given that 'feature maps' are nothing but characteristics of a given input matrix (commonly images) represented as a matrix/vector, we expect that earlier convolutional layers should have more channels to capture various simple characteristics of the input, and later convolutional layers will have fewer channels that capture overarching themes of the earlier feature maps produced by earlier layers.

We can capture the feature maps at each convolutional layer and pooling layer by simply getting the outputs of each layer in a fully trained model, instead of just the terminal output at the softmax layer. This allows us to visualize what the convolutional and pooling layers are doing.

Optimization

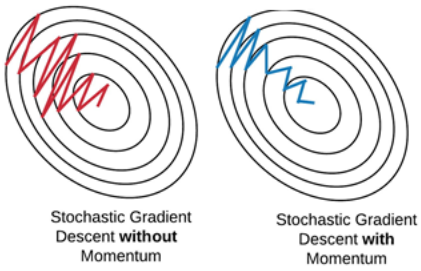
To enhance the performance of our model, various optimization techniques were used. The following were utilized in this question.

1. Stochastic Gradient Descent (SGD) / Mini-Batch Gradient Descent

SGD is one of the simplest learning algorithms that we can use. In essence, we compute the gradient of the cost function with respect to the parameter that we are optimizing (usually the weights of the model) to discover the optimal set of values for the weights that minimizes our cost function. Stochastic Gradient Descent (formally) will update the weights with every training input. In our case, we implemented mini-batch gradient descent.

Mini-Batch Gradient Descent updates the weights after a specified number of inputs have been processed, termed as the batch size. This speeds up the model, as SGD is very slow due to weights updating for every single training input.

2. Mini-Batch Gradient Descent with momentum term



Gradient Descent aims to find an optimal set of weights that allows us to achieve the global/local minima for the cost function. However, these minimum points are often associated with steep gradient drops. Vanilla SGD tends to oscillate across the slopes of these 'ravines' and take much longer to reach the minima. Therefore, the model will take a longer time to approach convergence at minima.

Momentum terms nudge the gradient towards the ravine. Essentially, the momentum term accelerates the gradient in the direction that previous gradients also imply. If the history of the past gradients has vectors that point in a certain direction, then there is momentum in that direction, and the momentum terms accounts for this direction, adding the momentum to the current gradient. Thus, if the direction of past gradients points toward the minima, the gradient will be accelerated towards the minima instead of taking small steps and bouncing across the ravine.

More formally,

γ momentum term	θ optimization parameter	η learning rate	v_t update vector at time t
$v_t=\gamma v_{t-1}+\eta\nabla_\theta J(\theta)$ the momentum from update vector at previous time (t-1) carries into the update vector at time t, and joins with actual gradient of loss function with respect to optimization parameter to form new update vector			

We expect that momentum terms will help navigate gradients better and converge faster and more accurately.

3. RMSProp algorithm

Complex neural networks like that pass gradients through multiple layers in back propagation might face vanishing gradients if the gradients are small and undergo repeated matrix multiplications with other small gradients, effectively causing them to go to zero. The inverse is true for gradient explosions, where large gradients keep getting bigger.

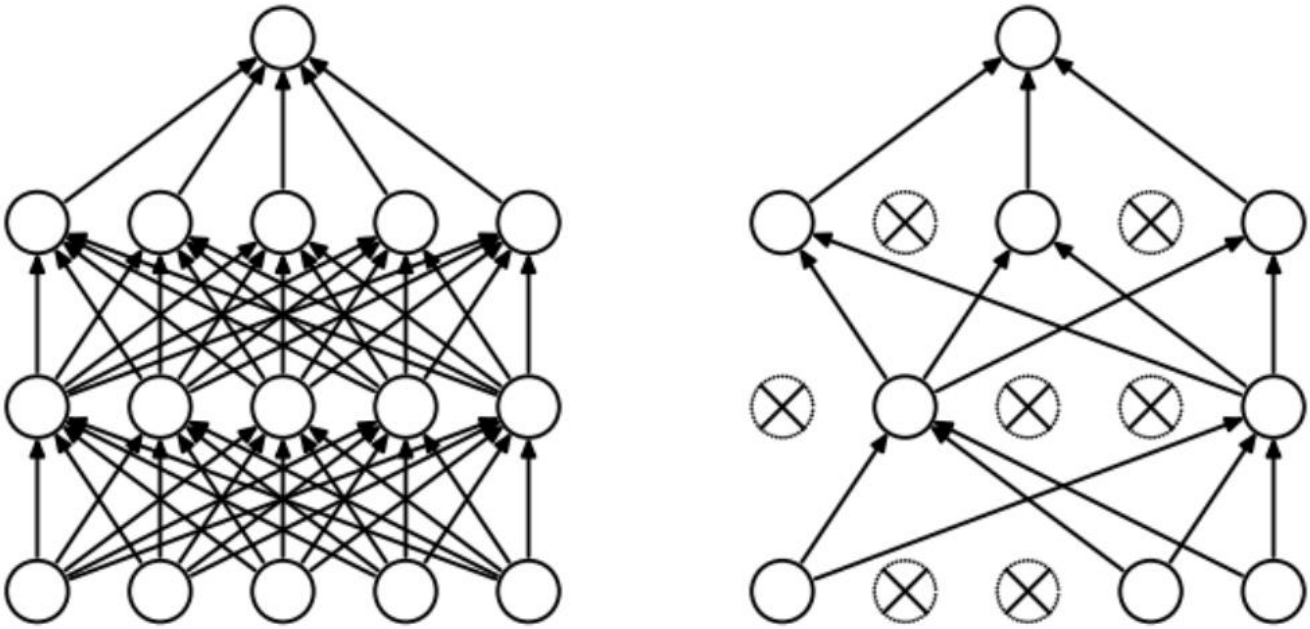
RMSProp normalizes gradients by using a running average of the magnitudes of recent gradients. Therefore, it places restrictions on how big and how small the gradients can end up being, which helps avoid exploding gradients and vanishing gradients. This keeps the model weights under control, and training at a reasonable pace.

4. Adam optimizer

The Adam optimizer differs from SGD where learning rates are concerned. In SGD, a single learning rate is applied to the model. Adam uses adaptive moments to calculate a suitable learning rate for individual parameters, thus learning rates do not remain the same throughout training, and it is adapted by calculating exponential moving average of the gradient and squared gradients (controlled by two parameters β_1 and β_2). Adam optimizer shows good performance in many tasks.

5. Dropout

Neural networks function using layers of neurons. Dropout is a regularization technique that will randomly turn off some of the neurons in a neural network. Turning off a neuron simply renders its synaptic output invalid, and therefore the weights that are learned by neurons that have been turned off do not propagate through the model. Dropout is usually used to prevent overfitting by randomly turning off certain weights, which might include very large weights learned in overfitting. It is expected that dropout will help models generalize better in cases of overfitting.



(a) Standard Neural Net (b) After applying dropout.

Saving Data

The history of each model generated by the model.fit( ) method will be saved. Logging these outputs is necessary, to analyze them afterwards without having to keep the notebook instance running constantly. It will allow us to easily plot any graph that we want without having to run the model and go through the model training every time.

We save the history of the model, which includes loss and accuracy information, as a .json file, which we can easily retrieve through helper functions that have been defined.

Question 1 – Create a specified CNN, visualize feature maps

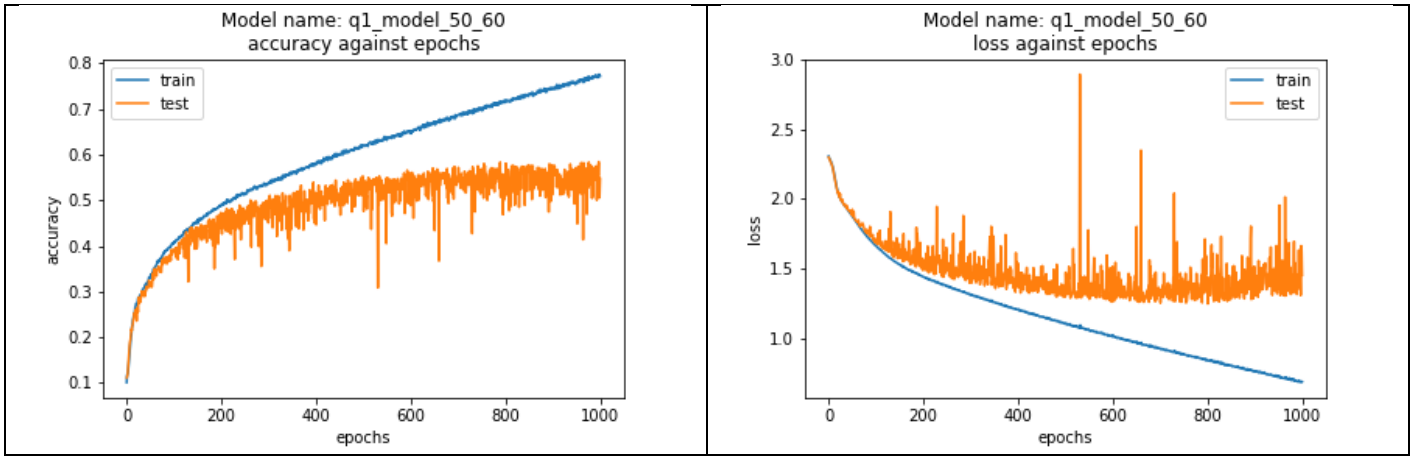
Experiments

We were given specifications for a CNN, and tasked to build this CNN, train it, and visualize the feature maps at each convolution and pooling layer. Also, the loss and accuracy should be plotted.

Notebook title: 2a/2a1.ipynb

Results

Q1a – cost and accuracy plots



We can see that the model has achieved convergence for the test set.

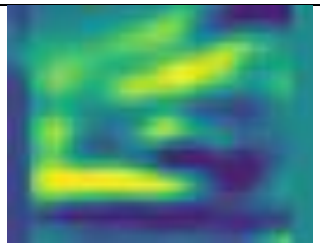
Q1b – feature maps

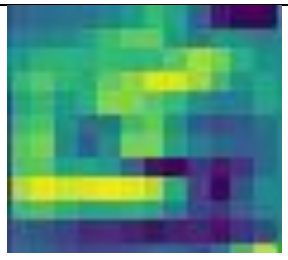
1. Convolutional layer C1, Maxpool layer S1 – 50 channels
2. Convolutional layer C2, Maxpool layer S2 – 60 channels

Therefore, we expect 50 feature maps for the first pair of layers, and 60 feature maps for the second pair of layers. We will plot the original image, and the feature maps for the first two test images. C1, S1 will be a grid of 5x10 images, each grid square representing a feature map, and C2, S2 will be a grid of 6x10 images, each grid square representing a feature map as well.

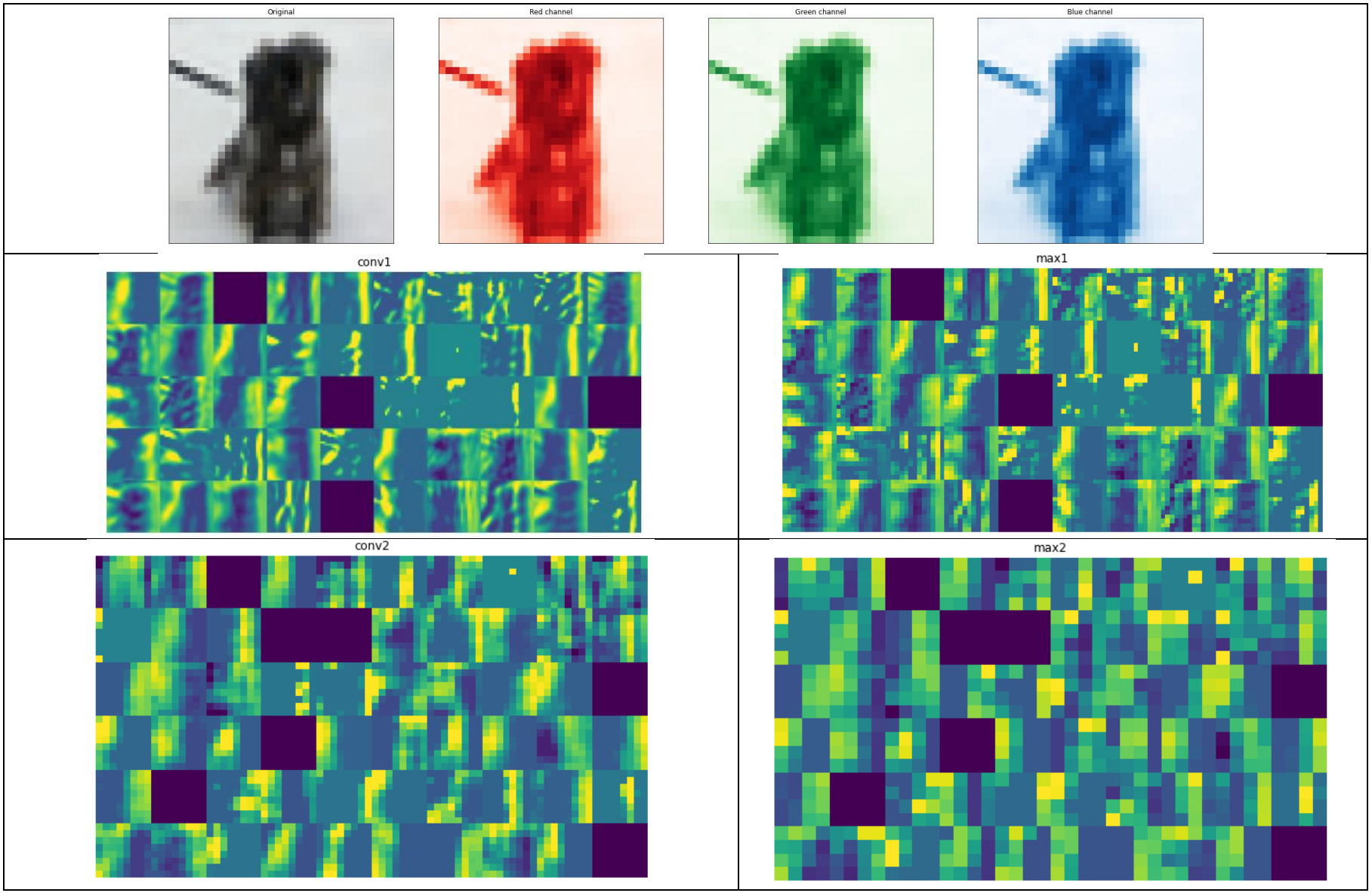
We can observe some edge detection from the C1 and S1, but nothing too significant or obvious. C2 and S2 simply look like noise, or rather the features are not discernable to us.


Original

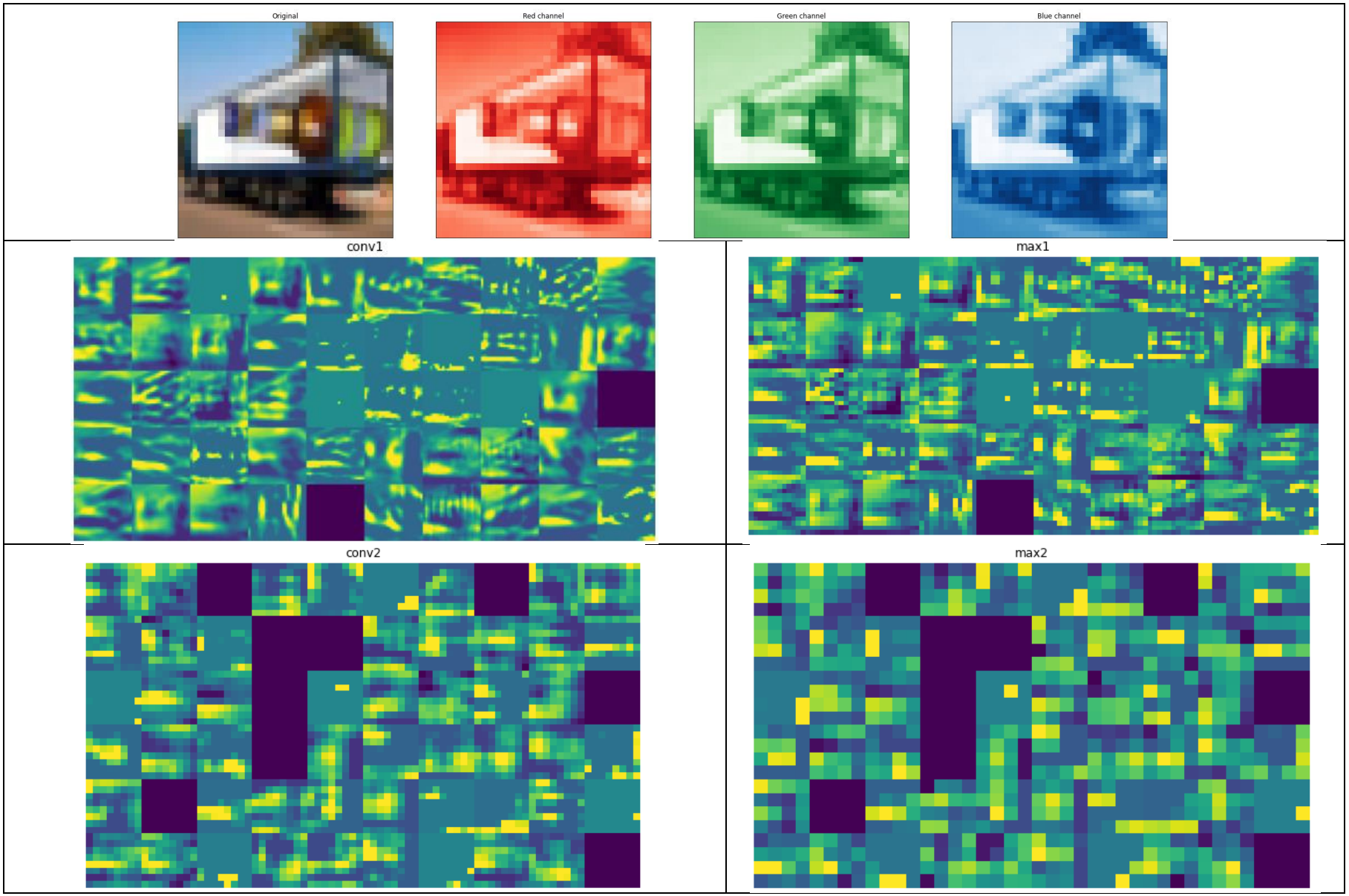

Conv1


Pool1

x_test image 0



x_test image 1



Question 2 – Hyperparameter sweep

Experiments

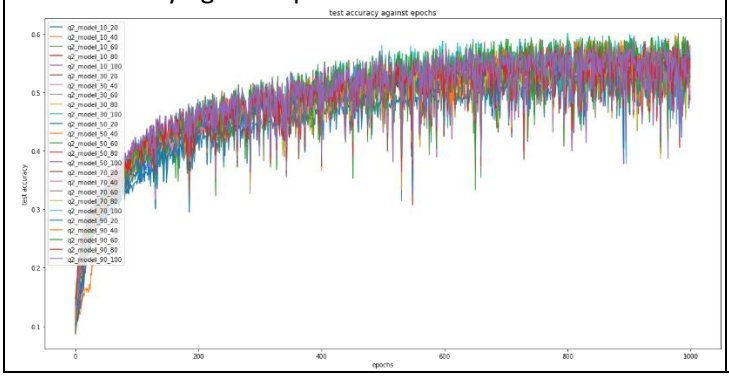
Question 1 involved us using 50 channels for C1, S1 and 60 channels for C2, S2. Now, we will do a hyperparameter sweep over the following: C1, S1 – 10, 30, 50, 70, 90 and C2, S2 – 20, 40, 60, 80 ,100. We will use test accuracy to determine the optimal combination.

Notebook title: 2a/2a2.ipynb

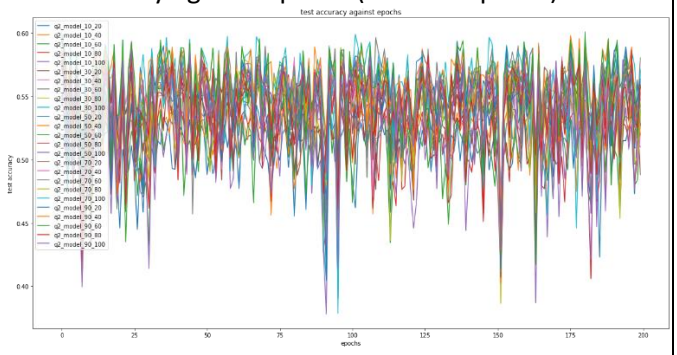
Results

Test accuracy (all 25 combinations)

Test accuracy against epochs



Test accuracy against epochs (last 200 epochs)



Given the noise in the test accuracy graphs, we cannot visually inspect and determine the best model. We shall take the average test accuracy over the last 200 epochs (after model convergence) and the model with the highest average test accuracy over the last 200 epochs will be the best model.

```
Format: 'q2_model_{channel 1}_{channel 2}': average test accuracy over last 200 epochs

{'q2_model_10_100': 0.5092549993097782,
 'q2_model_10_20': 0.5072299998998642,
 'q2_model_10_40': 0.5165350003540516,
 'q2_model_10_60': 0.5079275012016297,
 'q2_model_10_80': 0.5130325011909008,
 'q2_model_30_100': 0.5356725010275841,
 'q2_model_30_20': 0.5303774985671044,
 'q2_model_30_40': 0.5441125003993511,
 'q2_model_30_60': 0.5437750002741814,
 'q2_model_30_80': 0.5301324997842312,
 'q2_model_50_100': 0.5432974998652935,
 'q2_model_50_20': 0.5372199986875057,
 'q2_model_50_40': 0.5496200008690357,
 'q2_model_50_60': 0.5447699983417987,
 'q2_model_50_80': 0.5449249994754791,
 'q2_model_70_100': 0.5574675002694129,
 'q2_model_70_20': 0.5477649991214275,
 'q2_model_70_40': 0.548912497907877,
 'q2_model_70_60': 0.5547874991595745,
 'q2_model_70_80': 0.5480099998414516,
 'q2_model_90_100': 0.5468275010585785,
 'q2_model_90_20': 0.5526650014519692,
 'q2_model_90_40': 0.5589424978196621,    → best
 'q2_model_90_60': 0.5553849995136261,
 'q2_model_90_80': 0.5434250001609325}
```

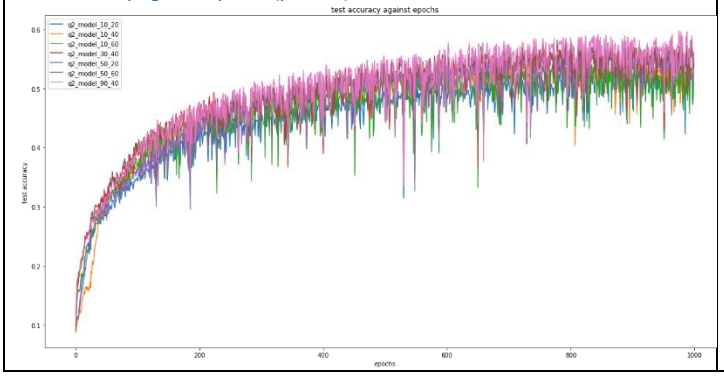
Test accuracy (pruned models where max(accuracy) – min(accuracy) < 0.15)

We further notice from the graphs of the last 200 epochs that certain models have very wild fluctuations. We can also attempt to prune off the models with the largest variances in performance and pick the top model from those that are more stable.

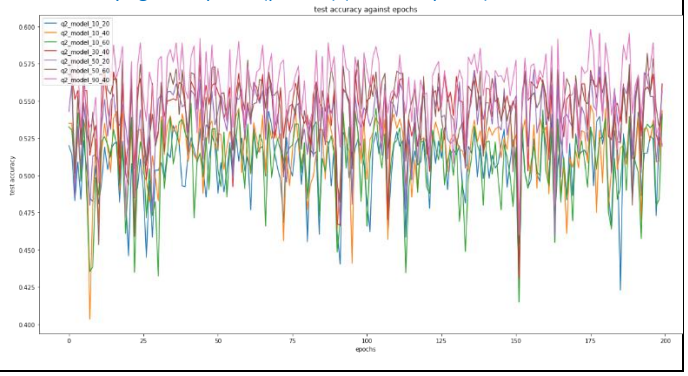
```
histories_json_pruned = {}

for model_name in histories_json.keys():
  test_acc = histories_json[model_name]['val_accuracy'][800:]
  if np.max(test_acc) - np.min(test_acc) < 0.15:
    histories_json_pruned[model_name] = histories_json[model_name]
```

Test accuracy against epochs (pruned)



Test accuracy against epochs (pruned) (last 200 epochs)



Format: 'q2_model_{channel 1}_{channel 2}': average test accuracy over last 200 epochs (for pruned models)

```
{'q2_model_10_20': 0.5072299998998642,
 'q2_model_10_40': 0.5165350003540516,
 'q2_model_10_60': 0.5079275012016297,
 'q2_model_30_40': 0.5441125003993511,
 'q2_model_50_20': 0.5372199986875057,
 'q2_model_50_60': 0.5447699983417987,
 'q2_model_90_40': 0.5589424978196621}    → best
```

Best model

We see that the model with 90 channels in C1, S1 and 40 channels in C2, S2 performed the best in both analyses. This fits right in with our hypothesis that the first convolutional layer should have more channels so that it can create more feature maps and pick up various characteristics of the image, while the second convolutional layer should aggregate the feature maps and pick up salient points that are relevant for the classification output.

Question 3 – Optimization

Experiments

With the optimal parameter set of channels {90, 40} found in question 2, we will now experiment with various optimizers. Optimizers will affect how our model learns – choosing a bad optimizer will cause our model to either learn slowly, poorly, or never converge. A good optimizer can allow us to converge quickly and accurately.

The optimization choices in question are:

1. Mini-Batch Gradient Descent with momentum γ = 0.1
2. RMSProp algorithm
3. Adam optimizer
4. Mini-Batch Gradient Descent with dropout, probability = 0.5 to the fully connected layers

For dropout, an amendment was made. In our model, there are only two fully connected (dense) layers. However, the second fully connected layer is the output nodes for classification and applying dropouts on this layer does not make any sense – we end up removing the prediction classes. Therefore, dropouts will be applied after the 'flatten' layer, and the first 'dense' layer. Therefore, we will end up with the model as shown on the right if we were to use dropout.

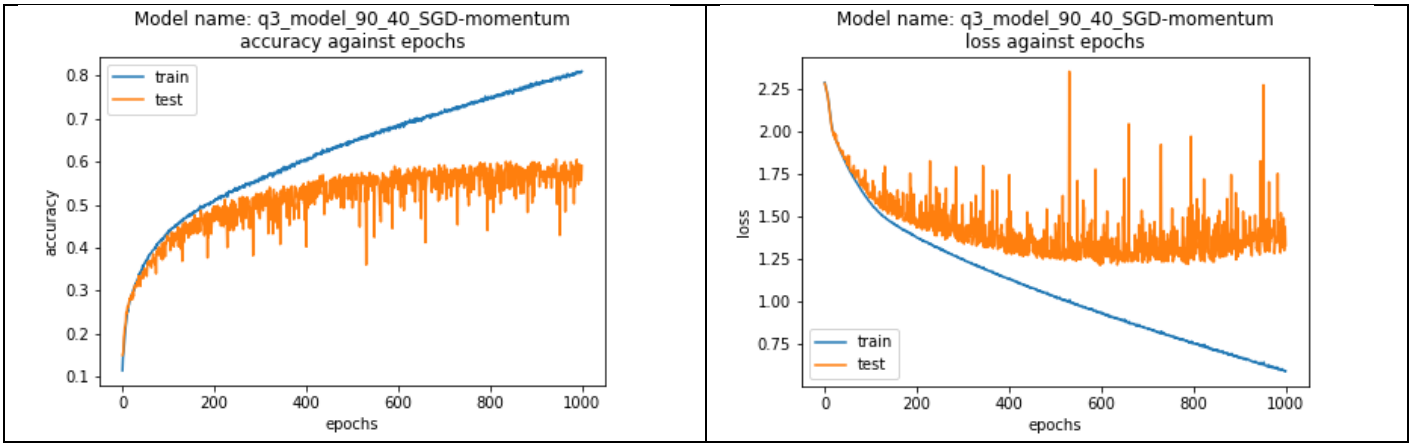
Although the 'flatten' layer is not technically a dense layer in keras, it can be treated as such since it is the one-dimensional vector representation of every synaptic output from the previous convolutional/maxpool operations.

Notebook title: 2a/2a3.ipynb

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1 (Conv2D)               (None, 24, 24, 90)        21960
_________________________________________________________________
max1 (MaxPooling2D)          (None, 12, 12, 90)        0
_________________________________________________________________
conv2 (Conv2D)               (None, 8, 8, 40)          90040
_________________________________________________________________
max2 (MaxPooling2D)          (None, 4, 4, 40)          0
_________________________________________________________________
flatten (Flatten)            (None, 640)               0
_________________________________________________________________
dropout1 (Dropout)           (None, 640)               0
_________________________________________________________________
dense1 (Dense)               (None, 300)               192300
_________________________________________________________________
dropout2 (Dropout)           (None, 300)               0
_________________________________________________________________
dense2 (Dense)               (None, 10)                3010
=================================================================
Total params: 307,310
Trainable params: 307,310
Non-trainable params: 0
_________________________________________________________________
```

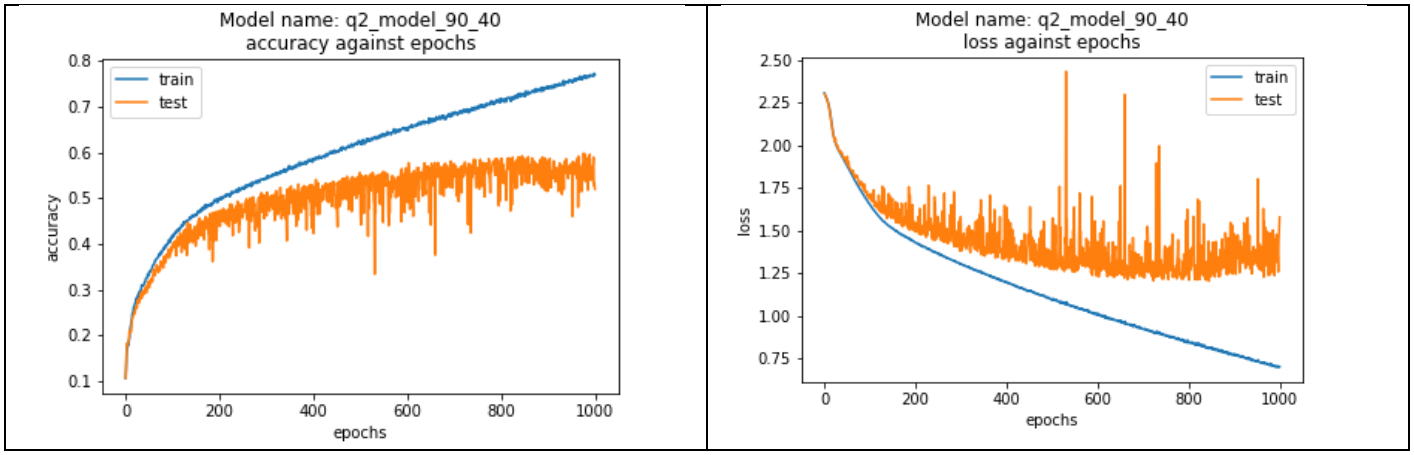
Results

Mini-Batch Gradient Descent with momentum γ = 0.1

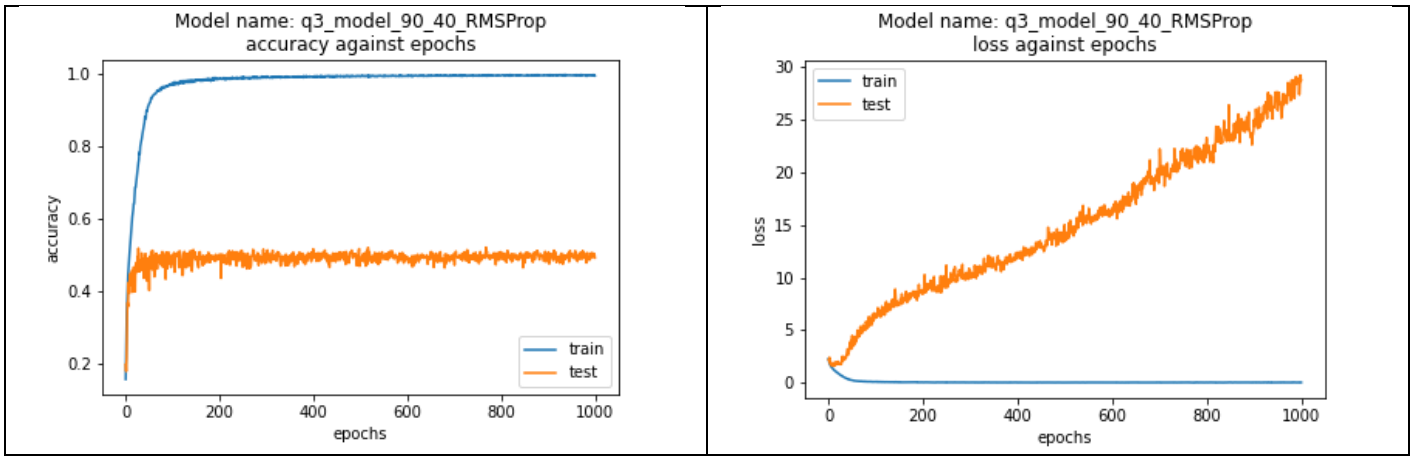


We have achieved convergence for this model with momentum. For accuracy, it was obtained around 600 epochs, while loss was stable around 500 epochs.

If we compare to the equivalent model without momentum (accuracy converged at 800 epochs, loss at 600), we can see that the convergence of the model was achieved much faster. This is likely due to the momentum term aiding the model in moving towards the minimum points.

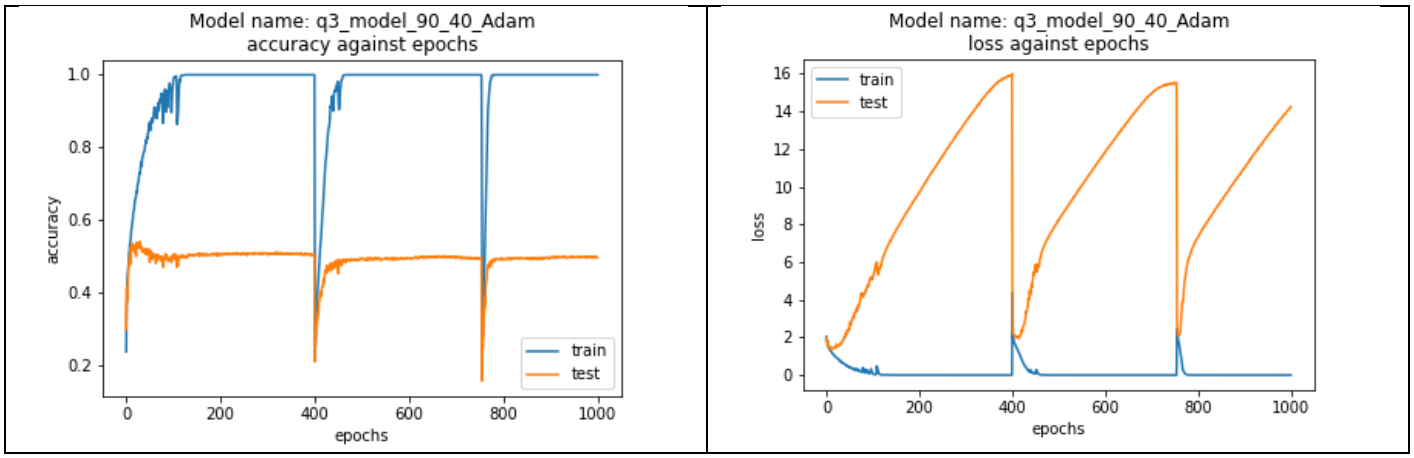


RMSProp algorithm



We have achieved convergence on the test set for accuracy, but not loss with RMSProp. The reason for this appears to be due to overfitting – the training loss for RMSProp is close to zero while accuracy is close to 1, which suggests that RMSProp is perfectly predicting the training set. This is a clear sign of overfitting and is reinforced by the fact that the test loss is climbing, since the model is unable to generalize to new data as it continues to overfit. While the test accuracy has not fallen, it will fall eventually as the probability of correct prediction falls as the loss climbs. Once it hits a tipping point (more epochs), the model will then fail to predict on test data.

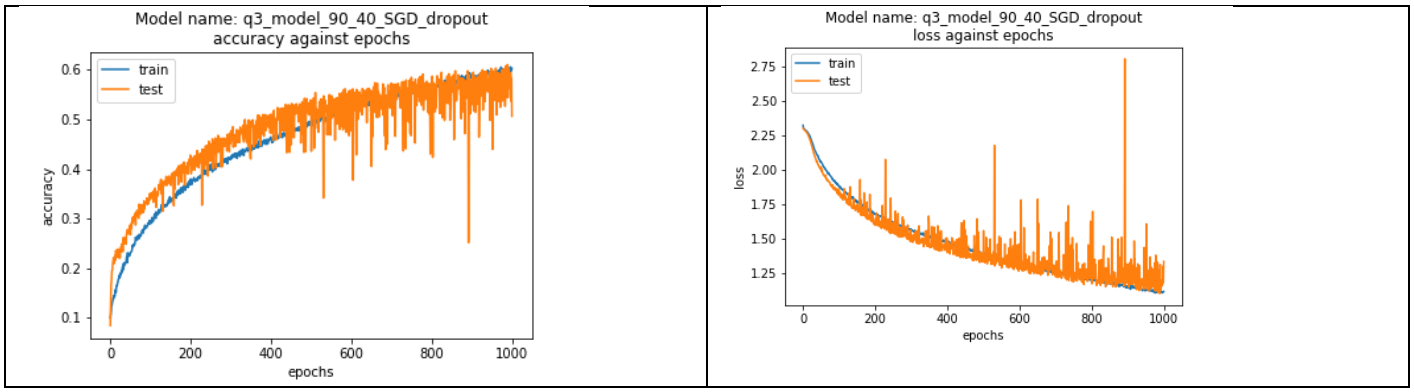
Adam optimizer



Adam optimizer performs cyclically. It is very interesting to note that the Adam optimizer is able to reach convergence extremely quickly for the test set with rather good accuracy and loss numbers, but once a few epochs passes the optimal point, the model will spike in loss till a peak value (around 16) and the accuracy will fall all the way down, only to recover shortly thereafter.

As discussed earlier under **Methods**, the Adam optimizer utilizes an exponential moving average of the gradient and squared gradients to adjust the learning rate dynamically. A possible reason for this sudden fall in performance, but very quick recovery thereafter is that the dynamically adjusted learning rate remained very high even though an average was taken, simply because the gradients were too steep. Therefore, divergent behavior occurs.

Mini-Batch Gradient Descent with dropout, probability = 0.5 to the fully connected layers



We observe quite clearly that the model is basically equally good at predicting both the test and the train data, although there is a significant variation in the performance of the model on the test data. This is directly attributable to the model having dropout layers included in its architecture. By randomly killing some neurons, we cause the model to lose some information and weights, and if these weights happen to be overfitted weights, the entire model benefits as a whole as we remove overfitting and the rest of the model needs to learn to generalize.

Question 4 – Comparison

Experiments

We are to compare the models from Question 1 to Question 3. Note that the model in Question is subsumed under the hyperparameter sweep in Question 2, so we only need to compare Question 2 to Question 3.

There will be several categories of comparison.

1. Optimal number of channels in the first convolutional / maxpool layer
    a. The options were [10, 30, 50, 70, 90]
2. Optimal number of channels in the second convolutional / maxpool layer
    a. The options were [20, 40, 60, 80, 100]
3. The optimal model in Question 2
4. Optimal model from Question 2, versus the four models in Question 3
    a. The difference is in the optimization

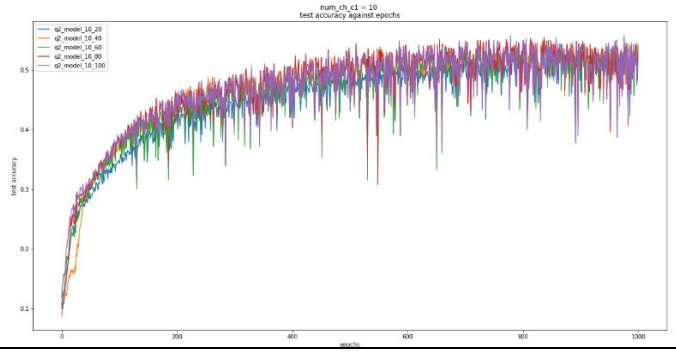
Notebook title: 2a/2a4.ipynb

Results

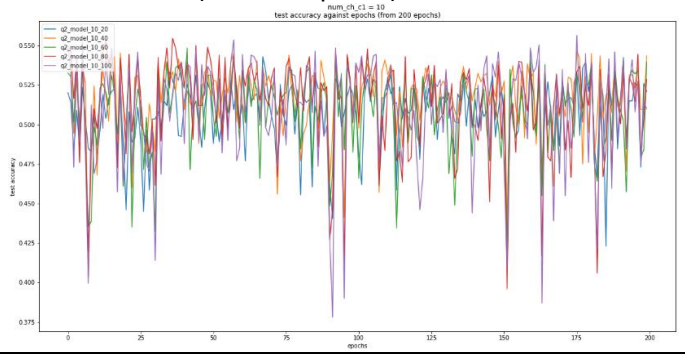
Optimal number of channels in the first and second convolutional / maxpool layer + optimal model in question 2

Below is a plot of the test accuracies for the five models that have num_ch_c1 = 10. We can see that there is a lot of variation in the test accuracies, therefore we shall take an average of the test accuracies over the last 200 epochs and tabulate them instead.

Test accuracies



Test accuracies (last 200 epochs)

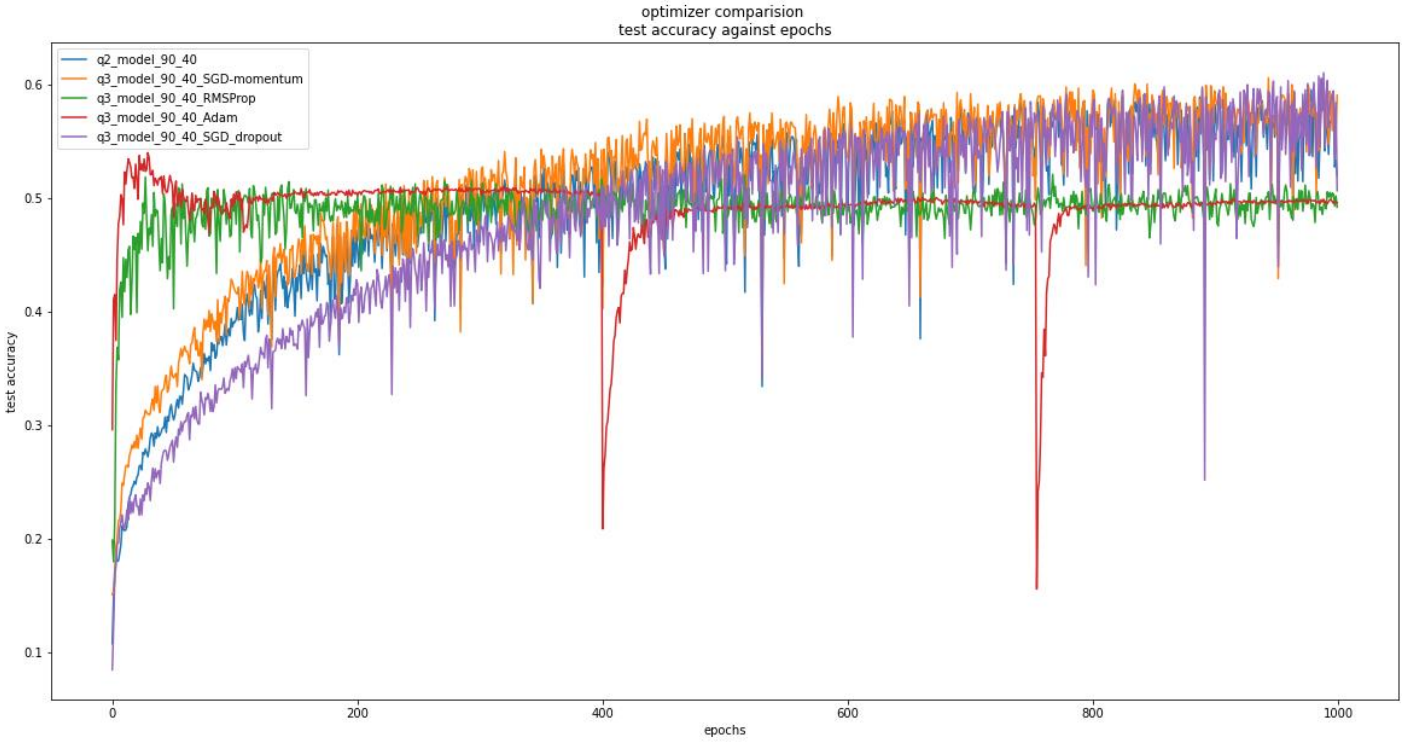


Ave Test Acc. (last 200 epochs)		num_ch_1					
		10	30	50	70	90	Row Ave
Num_ch_2	20	0.5072299998	0.5303774985	0.5372199986	0.5477649991	0.5526650014	0.5350514995
	40	0.5165350003	0.5441125003	0.5496200008	0.5489124979	0.5589424978	**0.5436244994**
	60	0.5079275012	0.5437750002	0.5447699983	0.5547874991	0.5553849995	0.5413289996
	80	0.5130325011	0.5301324997	0.5449249994	0.5480099998	0.5434250001	0.5359050000
	100	0.5092549993	0.5356725010	0.5432974998	0.5574675002	0.5468275010	0.5385040003
	Col Ave	0.510796000	0.5368140000	0.5439664994	0.5513884992	**0.5514490000**	

90 channels for the first layer appears to be optimal for the first set of layers, while 40 channels for the second layer is optimal. Unsurprisingly, our best model is the model with the 90-40 configuration.

The reason is likely because more first-layer features will capture more basic details and characteristics of the image. Too few and we capture too little detail to differentiate features of different image classes effectively. Conversely, fewer second-layer features will capture the salient features of each image class. Given that we have 10 image classes to predict, we should not have too many salient features, otherwise we run into problems of different classes having many of the salient features.

Optimal model from Question 2, versus the four models in Question 3



From the figure above, we observe the following:

1. RMSProp and Adam optimizer do not perform that well in the long run
2. RMSProp and Adam optimizer perform significantly better than all other models in small number of epochs

As explained in Question 3, the gradient and learning rate manipulation works against RMSProp and Adam optimizer in the long run. However, in the short term, they clearly work wonders as both models required less than 50 epochs to reach their peak test accuracy.

This is surprisingly short and significantly faster than the other three “top models”, which leads us to conclude that if training time/cost is of concern, RMSProp and Adam optimizer are amazing options as they allow for rapid model training and strong performance in a short period of time

3. Dropout has a lot of fluctuations among the models that perform well

This is to be expected. As some high-weightage nodes are randomly turned off, we can expect the model performance to deteriorate significantly before recovering

4. The best models perform very similarly, but the best on average (last 200 epochs) is Dropout

```
{'q2_model_90_40': 0.5589424978196621,
'q3_model_90_40_SGD-momentum': 0.5681024992465973,
'q3_model_90_40_RMSProp': 0.4952874992787838,
'q3_model_90_40_Adam': 0.4952874985337257,
'q3_model_90_40_SGD_dropout': 0.559667500257492,
'ave': 0.5354574990272523}
```

We can see that dropout, while simple, has allowed us to obtain impressive results. This may be due to the small training set which makes overfitting a problem. However, we can imagine that if we combine these optimization methods, we will obtain a much better test accuracy than what we have now, where we applied optimization methods independently.

Summary

The process of finetuning the hyperparameters of a model will eventually lead to us having a more robust model that can generalize to new and unseen test cases. We have discussed the reason why 90 and 40 channels might be the optimal channels to select for the two convolutional/maxpool layers and saw the benefits that different optimization methods might bring for our models.

Part B: Text Classification

Introduction

Part B is a text classification problem using the first paragraphs collected form Wikipedia entries, which are split into 15 categories. The end goal is to perform a multi-class classification and output a class label corresponding to the input paragraph.

There are four tasks given, namely

1. Implement a Character level CNN
2. Implement a Word level CNN
3. Implement a Character level RNN with GRU
4. Implement a Word level RNN with GRU
5. Experiment with dropouts on the above models and compare accuracies
6. Experiment with Word and Character RNN variations (LSTM, Vanilla, increase RNN layers, gradient clipping)

The primary goal of this problem is to discover the best model that can be used to classify texts.

Methods

There are several primary techniques that are common throughout the notebooks used for Part B. We shall address why these methods were necessary, and hence chosen.

Convolutional Neural Networks (CNN)

CNNs have already been described in **Part A Methods**. However, the difference here is the representation of the input data. Instead of having images, what we have are vectors that correspond to either characters or to words.

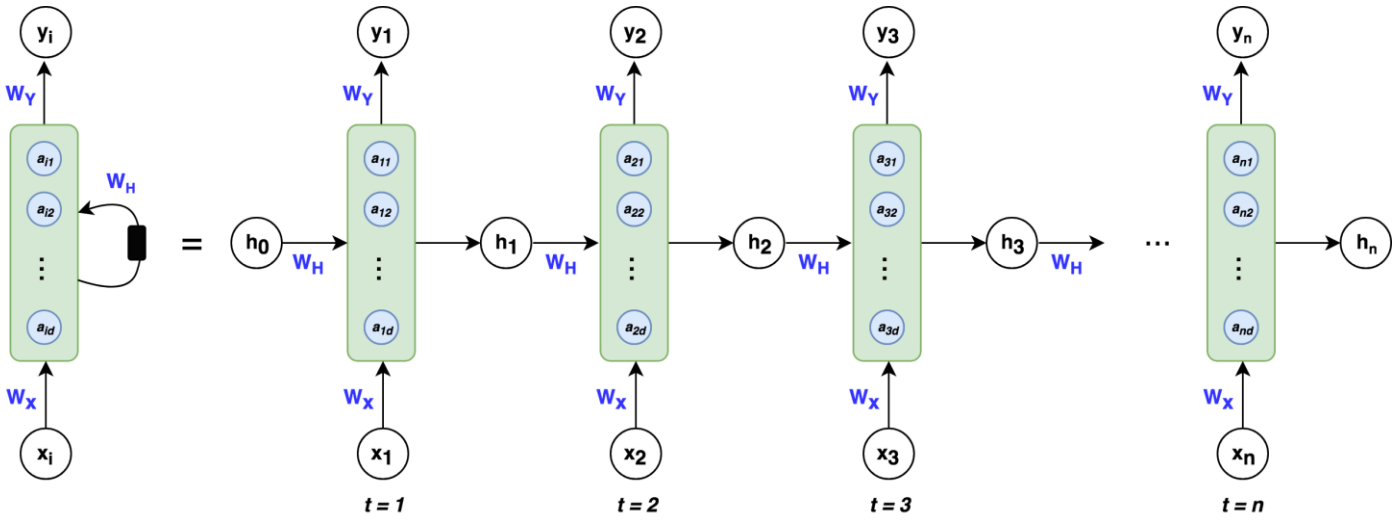
Recurrent Neural Networks (RNN)

RNNs are very powerful tools in natural language processing. One of the main drawbacks of CNNs is the loss of the temporal dimension when we process it with kernels. RNNs are very deep in the temporal dimension – each character/word is processed in the sequence that it appears in, sequentially. RNNs will utilize the hidden state output of the $t\text{-}1^{th}$ input and feed it forward to the t^{th} input, together with the hidden layer weights. The hidden layer weights are updated and passed onwards to the next hidden layer, therefore propagating information in the temporal dimension. This allows RNNs to "remember" information from the previous states, and given that we are working with language prediction, the next state (character/word) depends heavily on the state before it. This makes RNN a very useful tool in natural language processing.

RNN Types

There are several types of RNNs as well.

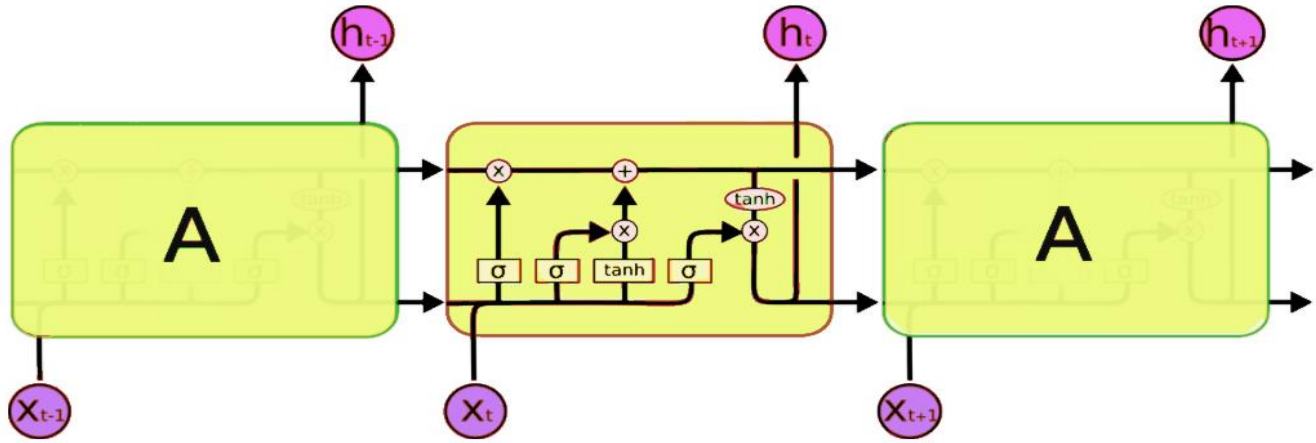
Vanilla RNN – this is the simplest implementation of RNN, which can be visualized as below. In this case, we simply rely on the hidden state weight to remember information about the sequence history in order to make a better prediction.



In Vanilla RNN, we encounter the gradient vanishing problem in long input sequences because the same weight is being multiplied many times in the back-propagation step together with the gradient. If the weight is small (with an eigenvalue < 1), it is proven that gradient vanishing will occur in exponential time. Conversely if the weight has an eigenvalue > 1, the gradient will explode.

Ultimately, this means that the hidden state can only carry recent information – information long ago in the temporal dimension will likely be lost. This affects the performance of the RNN model.

LSTM – Long Short Term Memory. This is a type of RNN that was created to resolve the gradient vanishing problem of Vanilla RNNs. There is a separate memory called "cell state" that carries long term information for the model.



The cell state is controlled through three gates

1. Forget gate – controls the amount of information to forget from the cell state
2. Input gate – controls the amount of information to put into the cell state, from the current hidden state
3. Output gate – controls the amount of information to retrieve from the cell state

Each of these gates are associated with their own weight matrix, and forms three more learnable parameters for the LSTM model. This will allow the model to learn when it should remember long term information, and what long term information should be kept. This sidesteps the vanishing gradient problem of Vanilla RNNs.

However, this comes at the cost of the three extra weight matrices to learn, plus the cell state to remember. This makes LSTM slow to train.

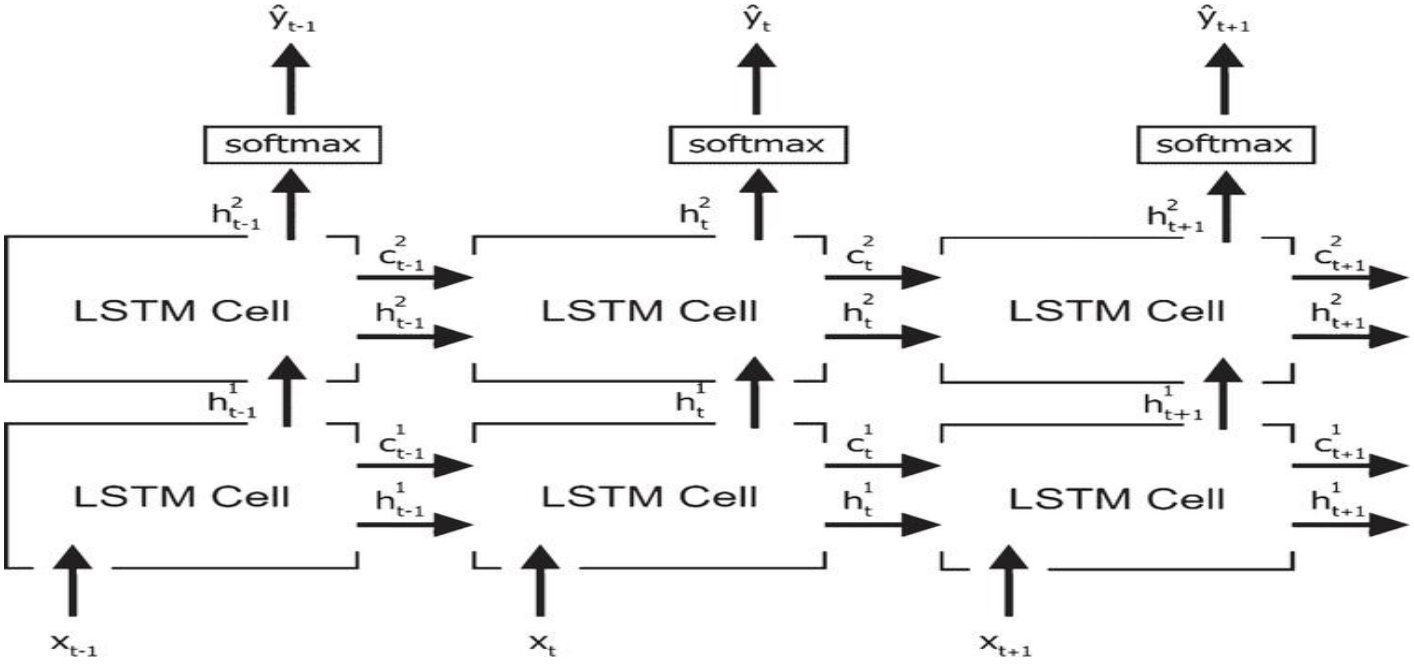
GRU – Gated Recurrent Unit. This is a simpler version of the LSTM. Instead of having three gates, it only has two. Also, it uses the hidden state to transfer information, thus it is cheaper and faster to train than LSTM.

1. Reset gate – controls the amount of information to forget
2. Update gate – decides what information to throw away and what information to add

We expect that the three RNNs will have performances in the following order: (best) LSTM, GRU, Vanilla.

Number of RNN layers

RNNs are naturally deep in the temporal dimension as the sequentially process the inputs to the model. Stacking RNNs on top of each other will make the RNN model deep in not only the temporal dimension but become a true deep neural network. This involves feeding the outputs of a RNN cell's hidden states as inputs into the next RNNs, thereby hoping to learn some hidden representation through a deeper model.



Gradient Clipping

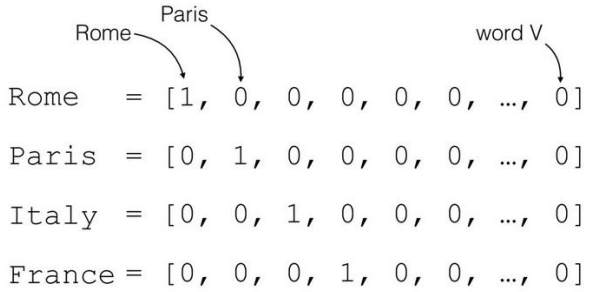
As mentioned, gradient explosions might occur with the same weight matrix being multiplied over and over in an RNN. Therefore, one way to mitigate the effects of a gradient explosion (which causes erratic training) will be to clip the gradient, i.e. set it to a threshold value if the gradient exceeds the threshold. This is expected to help if gradient explosion occurs. Otherwise it has no effect.

Character and Word Embedding

Characters are represented as one-hot vectors of the type [0 0 0…. 1 …0 0 0] within a vocabulary of characters. This is a sufficient representation as characters themselves do not carry much meaning. However, words have significantly more meaning to them, and representing them with a one-hot vector does not allow us to see the meaning of the word. For example, comparing the words “motel” and “hotel” with one-hot vectors will imply that they are not related, because they are orthogonal to each other.

Word embeddings, or word vectors, essentially represent the words as a N-dimensional vector. Therefore, similar words will then have similar word embeddings. This is a more complete representation of a word and allows us to have a representation of its context, meaning, relationships etc. with a dense vector versus a sparse vector.

One-hot



Word Embedding

It is expected that word embedding models will be more powerful than character embedding models because they are a more powerful representation of the input.

Saving Data

The history of each model generated by the model.fit( ) method will be saved. Logging these outputs is necessary, to analyze them afterwards without having to keep the notebook instance running constantly. It will allow us to easily plot any graph that we want without having to run the model and go through the model training every time.

We save the history of the model, which includes loss and accuracy information, as a .json file, which we can easily retrieve through helper functions that have been defined.

Epochs and Optimizers

I have followed the sample code provided, and used the following for my experiments:
250 epochs, SGD optimizer for CNN models ||| 100 epochs, Adam optimizer for RNN models

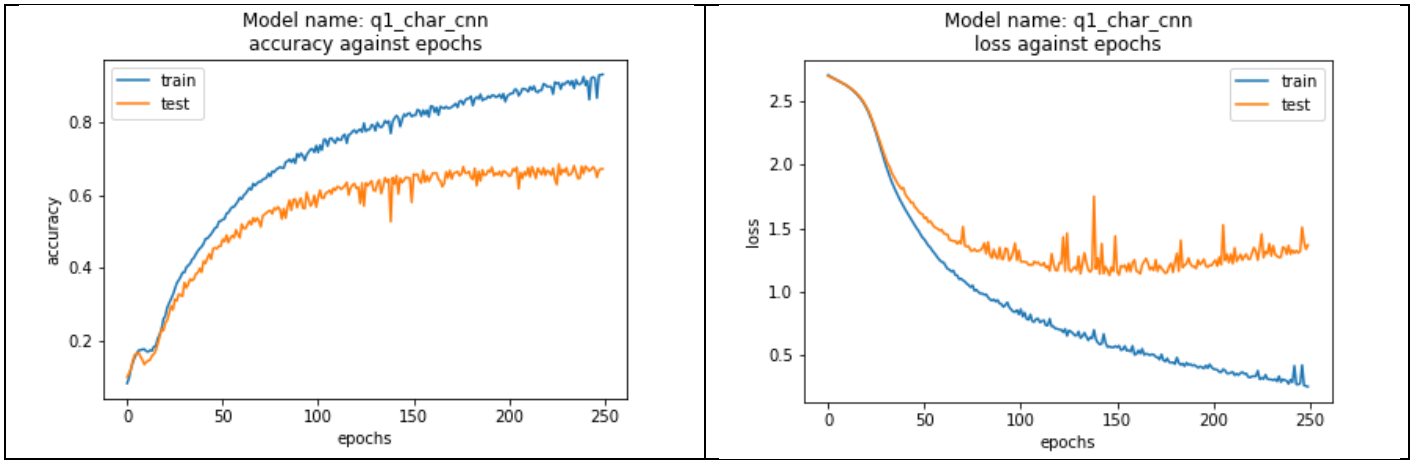
Question 1 – CharCNN

Experiments

We were given specifications for a Character CNN and asked to plot the entropy cost (loss) and accuracy.

Notebook title: 2b/2b1_char_cnn.ipynb

Results



We can see that the model has achieved convergence for the test set around 150 epochs.

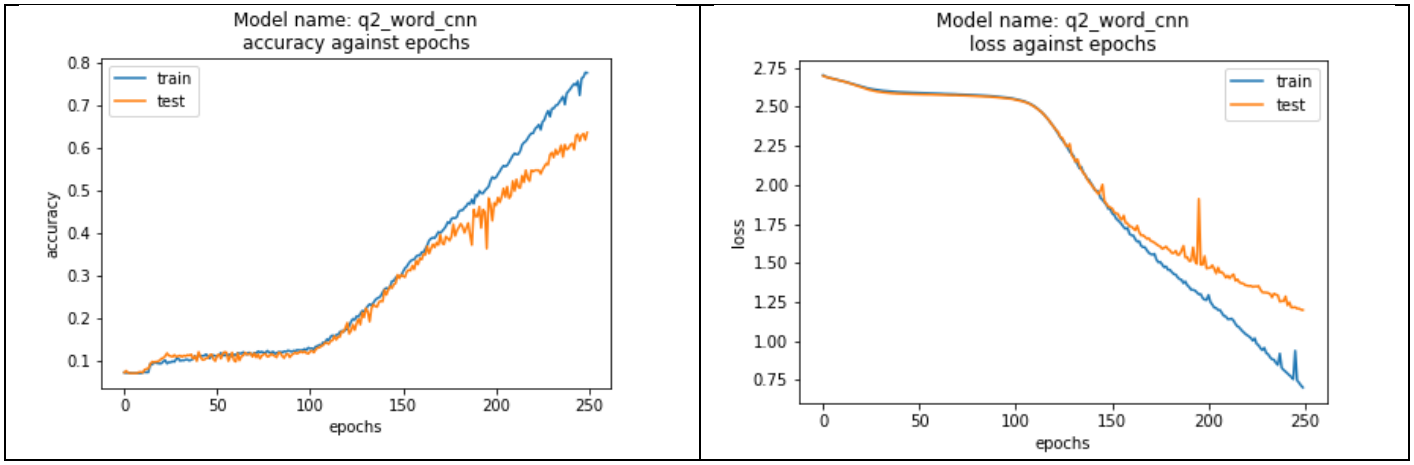
Question 2 – WordCNN

Experiments

We were given specifications for a Word CNN and asked to plot the entropy cost (loss) and accuracy.

Notebook title: 2b/2b2_word_cnn.ipynb

Results



Our WordCNN has not yet achieved convergence in 250 epochs. We can see both train/test loss still falling and accuracy still rising. This implies very strongly that word embeddings are a more powerful than character representations because they allow for more information to be captured. However, the assignment asked to not exceed 250 epochs hence we do not know the convergence point.

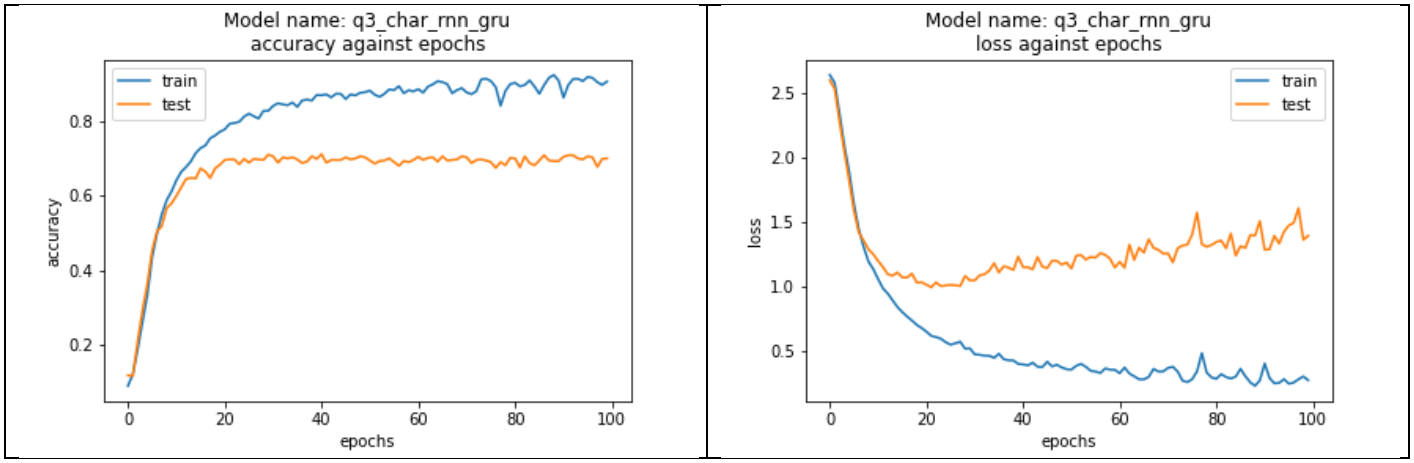
Question 3 – CharRNN GRU

Experiments

We were given specifications for a Character RNN GRU and asked to plot the entropy cost (loss) and accuracy.

Notebook title: 2b/2b3_char_rnn_gru.ipynb

Results



We can see that the CharRNN GRU model has achieved convergence for the test set around 20 epochs. The equivalent CharCNN model converged only around 150 epochs. This demonstrates that RNNs are much more suitable to use when working on NLP problems, likely because it captures the temporal aspect of the input sequence well.

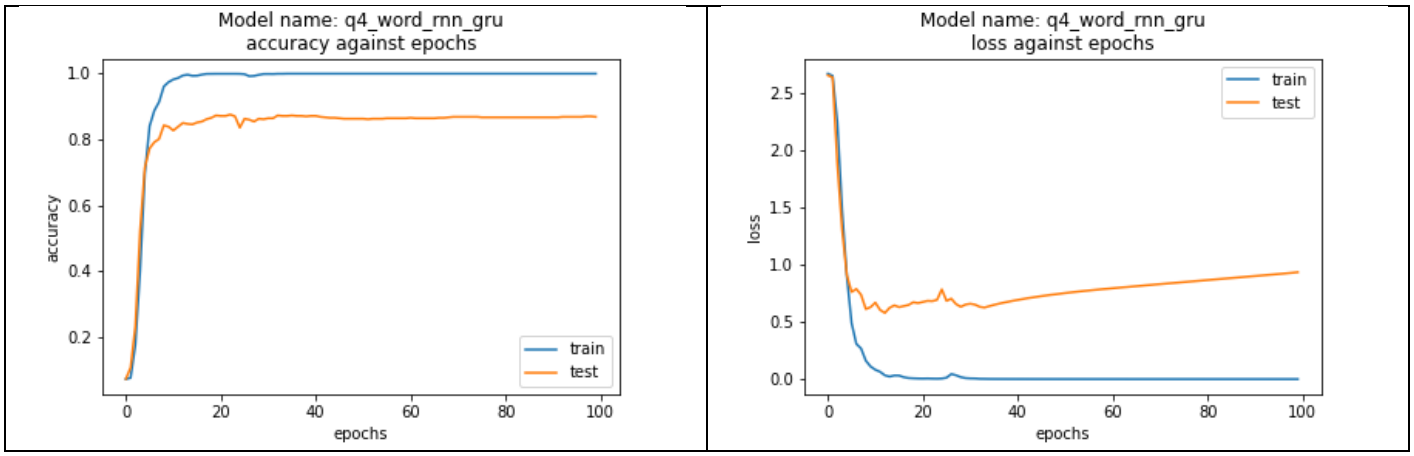
Question 4 – WordRNN GRU

Experiments

We were given specifications for a Word RNN GRU and asked to plot the entropy cost (loss) and accuracy.

Notebook title: 2b/2b4_word_rnn_gru.ipynb

Results



The WordRNN GRU (10 epochs) converged faster than the CharRNN GRU (20 epochs), implying word embeddings are still better than characters. Furthermore, it has achieved the best test accuracies so far out of the four models.

Question 5 – Comparison and Dropout

Experiments

We are to compare the models form question 1 to question 4. Subsequently, we are to implement dropout layers on those four models, and do a comparison again

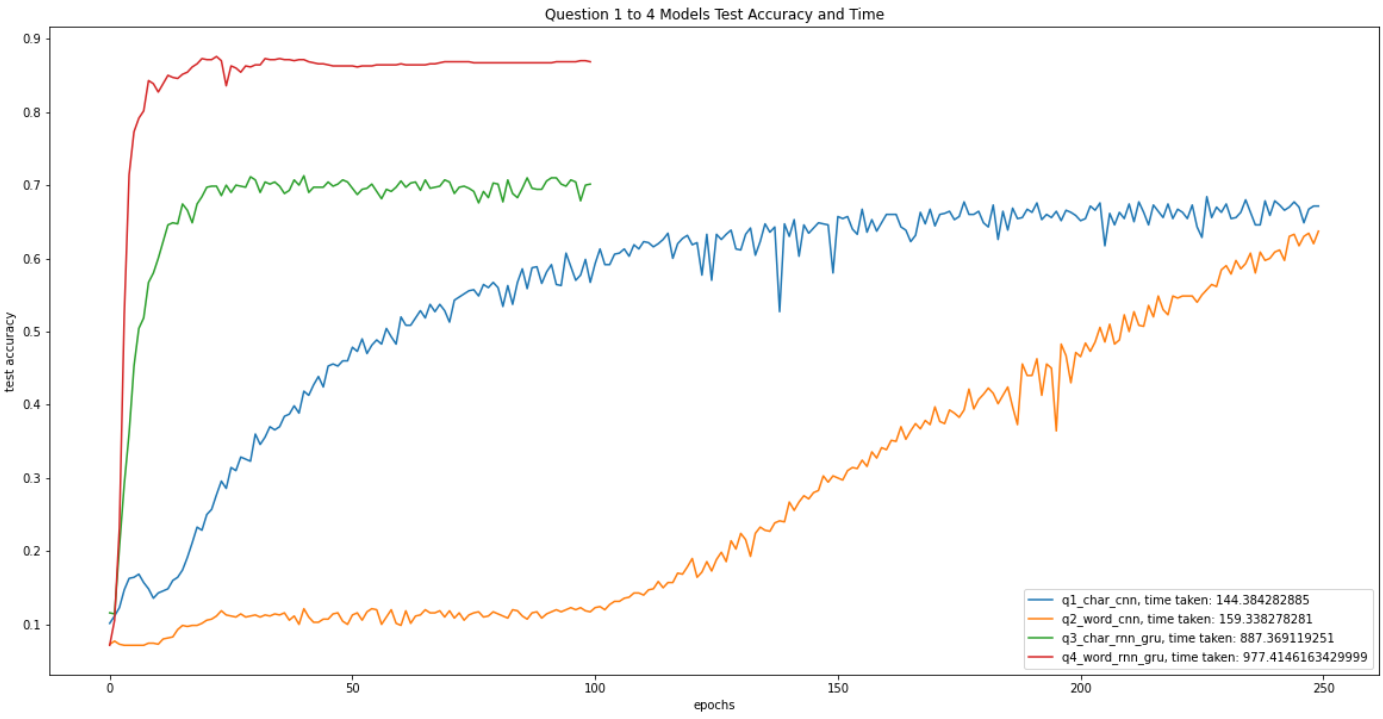
Notebook title:

2b/2b5.ipynb

2b5_1_char_cnn_dropout.ipynb	2b5_2_word_cnn_dropout.ipynb
2b5_3_char_rnn_gru_dropout.ipynb	2b5_4_word_rnn_gru_dropout.ipynb

Results

Comparing test accuracies, running time for question 1 to question 4



q1_char_cnn, time taken: 144.384282885
q2_word_cnn, time taken: 159.338278281
q3_char_rnn_gru, time taken: 887.369119251
q4_word_rnn_gru, time taken: 977.41461634429999

We observe the following:

1. RNN models (red, green) perform better than CNN models

This is evidenced by their higher accuracies achieved in smaller number of epochs. We have already mentioned some of the reasons why RNNs are more suited for natural language processing in both Methods, and Question 3 and

Question 4. This is mainly due to it capturing the history of the sequence prior to the current input, which is an important aspect for any form of language prediction as the current character/word depends heavily on the prior ones.

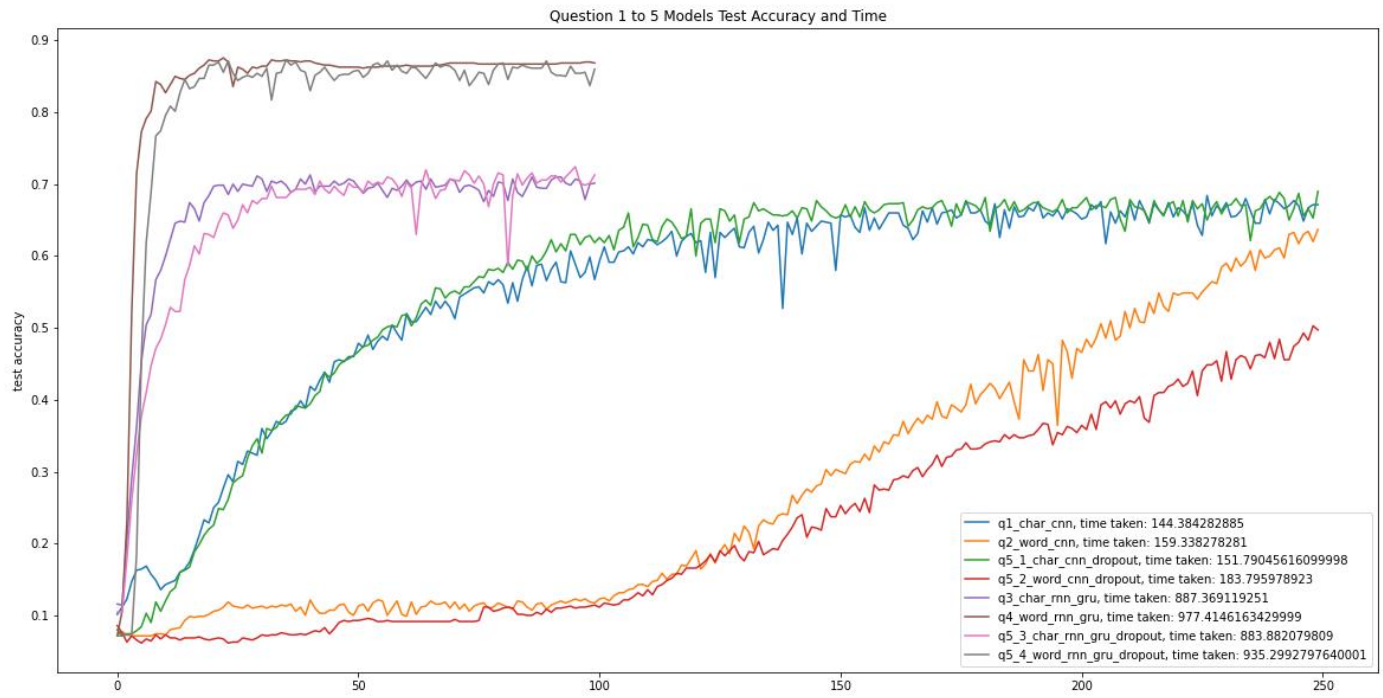
2. CNN models (around 150 seconds) take far less time than RNN models (around 940 seconds)

RNNs are strictly linear models that are non-parallelizable because the next input depends on the entire sequence of previous inputs. Therefore, there is no choice but to run the entire model in one sitting, and one processor. CNNs however can be easily parallelized because each kernel convolving over the input space can be processed in different parallel processors, this significantly shortening the time that we need.

3. WordCNN has not converged

250 epochs do not seem sufficient for WordCNN to converge.

Dropout versions, and comparisons



q1_char_cnn, time taken: 144.384282885
q5_1_char_cnn_dropout, time taken: 151.79.045616099998

q2_word_cnn, time taken: 159.338278281
q5_2_word_cnn_dropout, time taken: 183.795978923

q3_char_rnn_gru, time taken: 887.369119251
q5_3_char_rnn_gru_dropout, time taken: 883.882079809

q4_word_rnn_gru, time taken: 977.41461634429999
q5_4_word_rnn_gru_dropout, time taken: 935.2992797640001

We observe the following:

1. Dropouts do not affect the time needed very significantly. This is to be expected, turning off neurons takes very little time.
2. Dropouts affect performance, but not significantly except for word_cnn

char_cnn: blue vs green (dropout) → very similar performance. However, dropout accuracy fluctuates more

word_cnn: yellow vs red (dropout) → dropout significantly worsened the performance. This can be attributed to the fact that the model was not done training (accuracies have not converged yet). Therefore, instead of preventing overfitting, the dropout caused the model to lose information and therefore perform worse

char_rnn_gru: purple vs pink (dropout) → very similar performance. However, dropout accuracy fluctuates more

word_rnn_gru: Brown vs grey (dropout) → very similar performance. However, dropout accuracy fluctuates more

Dropout is not a regularization technique that will always work. We must have overfitting of the data for it to prove useful.

Question 6 – Modifying RNN

Experiments

We will modify the RNN and see what modifications will be helpful to create a more optimal model. We will modify the CharRNN GRU and WordRNN GRU in Question 3 and 4 respectively with the following:

1. Replace RNN GRU with
    a. Vanilla RNN
    b. LSTM RNN
2. Increase number of RNN GRU layers to 2
3. Add gradient clipping to RNN GRU with clipping threshold 2

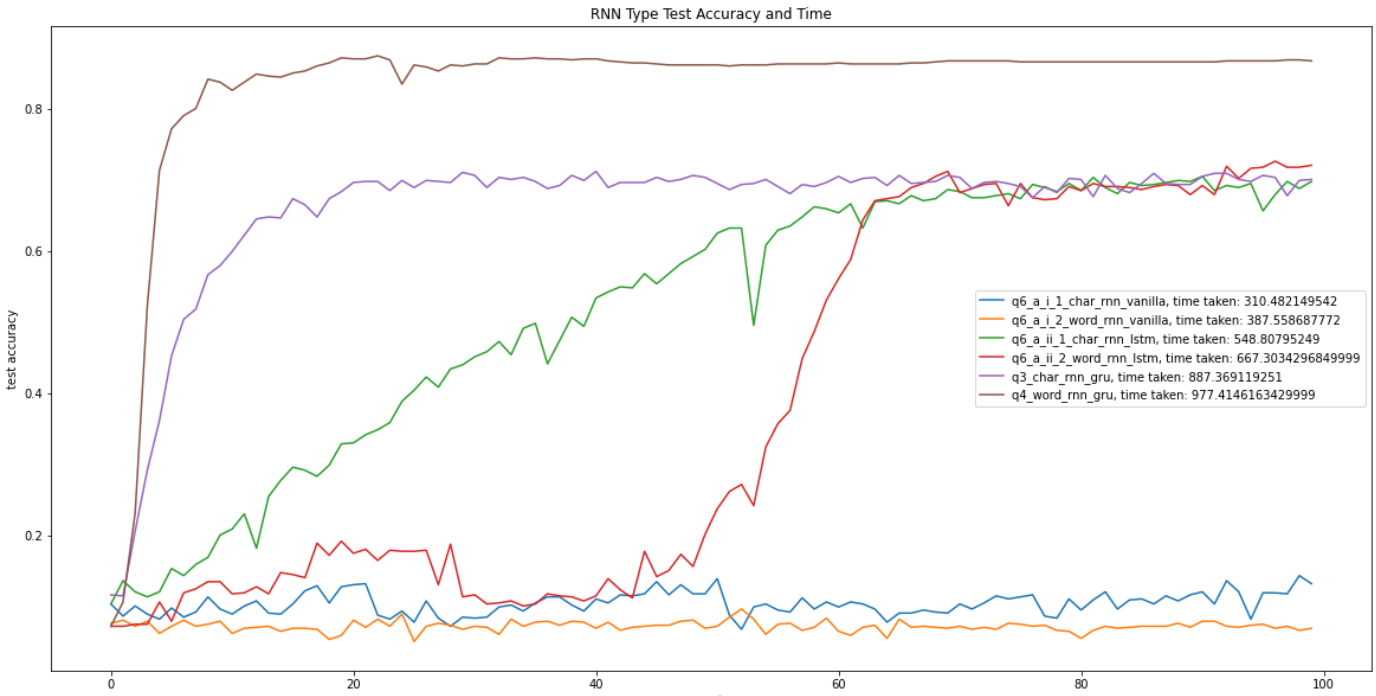
Notebook title:

2b/2b6.ipynb

2b6_a_i_1_char_rnn_vanilla.ipynb	2b6_a_i_2_word_rnn_vanilla.ipynb
2b6_a_ii_1_char_rnn_lstm.ipynb	2b6_a_ii_2_word_rnn_lstm.ipynb
2b6_b_1_char_rnn_gru_two_layer.ipynb	2b6_b_2_word_rnn_gru_two_layer.ipynb
2b6_c_1_char_rnn_gru_gradient_clipping .ipynb	2b6_c_2_word_rnn_gru_gradient_clipping.ipynb

Results

Replacing GRU layer with Vanilla RNN and LSTM layer



q3_char_rnn_gru, time taken: 887.369119251
q4_word_rnn_gru, time taken: 977.4146163429999
q6_a_i_1_char_rnn_vanilla, time taken: 310.482149542
q6_a_i_2_word_rnn_vanilla, time taken: 387.558687772
q6_a_ii_1_char_rnn_lstm, time taken: 548.80795249
q6_a_ii_2_word_rnn_lstm, time taken: 667.3034296849999

We observe the following:

1. WordRNN GRU is the best model
2. Vanilla RNNs in general performed the worst

This is as expected because no enhancements have been provided for Vanilla RNN. GRUs and LSTMs are both improvements on the Vanilla RNN, so it comes as no surprise.

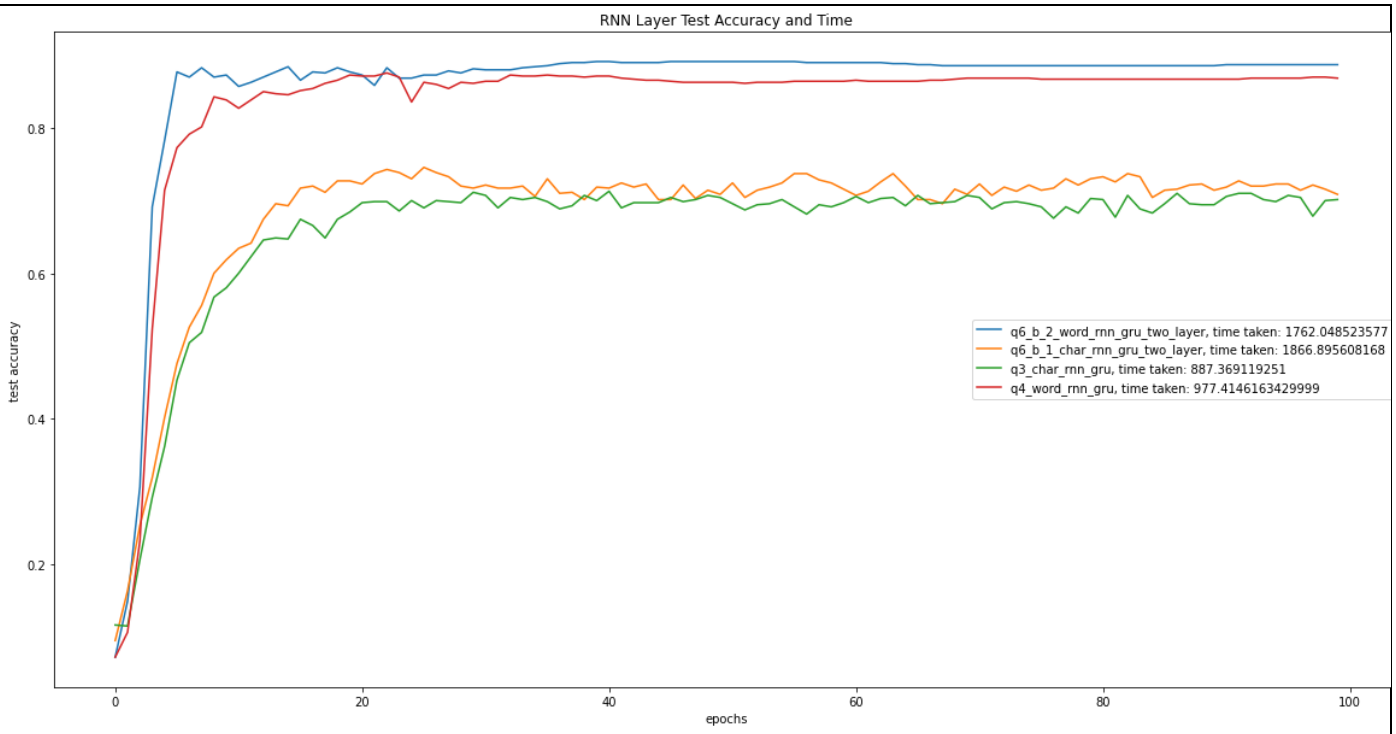
3. CharRNN GRU performed the same as CharRNN LSTM

Given that characters carry less information, it stands to reason that Char models will perform the same

4. GRU takes the longest time to run while Vanilla is the fastest

This is surprising because LSTMs are supposed to be slower than GRUs due to it having to remember more information by way of weights for three gates and remember a cell state too.

Increasing GRU layers to two



q3_char_rnn_gru, time taken: 887.369119251
q4_word_rnn_gru, time taken: 977.4146163429999
q6_b_1_char_rnn_gru_two_layer, time taken: 1866.895608168
q6_b_2_word_rnn_gru_two_layer, time taken: 1762.048523577

We observe the following:

1. Two layers will perform better than just having one layer

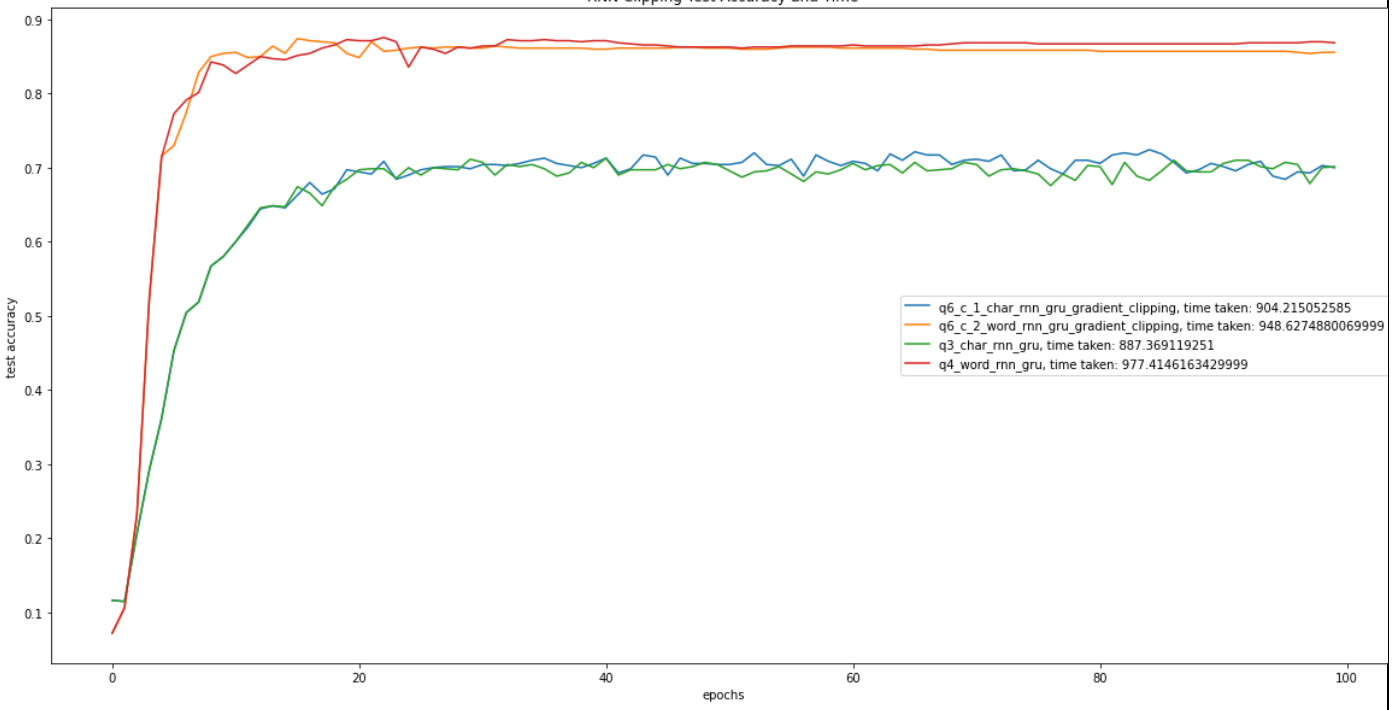
This is to be expected – stacking more layers might improve performance because the model can learn more information about the input data. However, the improvement does not appear to be all that significant, as evident from the graphs, the corresponding pairs of lines for two-layer and single-layer models are very close together.

2. Time taken by two layers is much longer – about double. This makes complete sense

This is as expected because no enhancements have been provided for Vanilla RNN. GRUs and LSTMs are both improvements on the Vanilla RNN, so it comes as no surprise.

We realize that simply increasing the number of layers in a model might not lead to a performance increase, because the model could already have learnt all the information it could with a single layer.

Adding gradient clipping



q3_char_rnn_gru, time taken: 887.369119251
q4_word_rnn_gru, time taken: 977.4146163429999
q6_c_1_char_rnn_gru_gradient_clipping, time taken: 904.215052585
q6_c_2_word_rnn_gru_gradient_clipping, time taken: 948.6274880069999

We observe the following:

1. Gradient clipping does almost nothing for both models

This implies that the gradients did not need to be clipped, which means gradient explosion did not occur. Therefore, the inclusion of gradient clipping did not affect the model in any way.

Summary

We have learnt that RNNs are much more powerful at natural language processing tasks as compared to CNNs, due to the nature of RNNs being deep in the temporal dimension. We have also observed that of the three types of RNNs, GRUs have emerged as the best model for our test dataset. However, the different types of RNNs all have their unique properties that can be exploited in different ways depending on the requirements.

<<< END >>>