

**Project 2:**

Convolutional and Recurrent Neural Networks

CZ4042 Project report

*by Hans Albert Lianto (U1620116K) and Eko Edita Limanta (U1620574A)*

Nanyang Technological University, AY2019-2020 Semester 1

# Project 2A: Object Recognition

## Introduction­­

In this section, we are creating a convolutional neural network to predict the label from the first batch of CIFAR-10 dataset. There are 10,000 training samples and 2,000 test samples.

## 

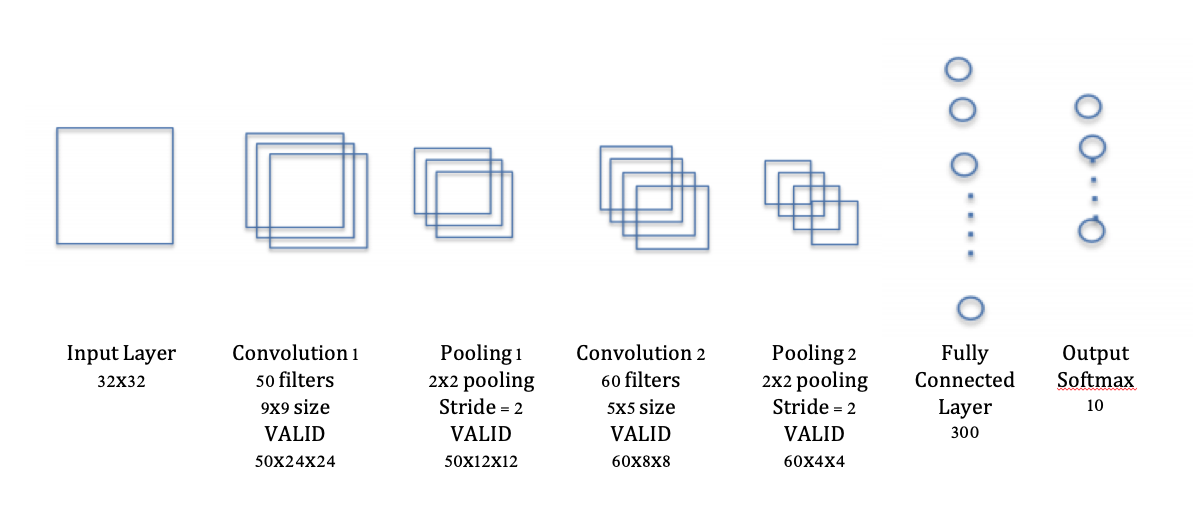
*A simple visualization of CIFAR-10 dataset*

## Methods

We design a convolutional neural networks with the following layers:

* An Input layer of 3x32x32 dimensions
* A convolution layer 𝐶1 of 50 filters of window size 9x9, VALID padding, and ReLU neurons.
* A max pooling layer 𝑆1 with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.
* A convolution layer 𝐶2 of 60 filters of window size 5x5, VALID padding, and ReLU neurons.
* • A max pooling layer 𝑆2 with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.
* A fully connected layer 𝐹3 of size 300.
* A softmax layer 𝐹4 of size 10.

Here is an illustration of the architecture, similar to the one in lecture notes.



This convolutional neural networks will be used across the experiments.

## Experiments and Results

### Pre-processing

In the experiments, we use different methods to train the convolutional neural networks and compare the results. We experiment with various hyperparameters such as learning rate and number of feature maps, and number of features. We also employ various training methods after discovering the optimal number of filters.

Before the experiment, we split the training dataset in the ratio 9:1 for training and validation respectively. We use this validation set to find optimal hyperparameters in the convolutional loop.

Furthermore, we also normalized the features in all of the dataset partitions (training, testing, validation) to prevent dominance of dimensions. After the training, we then use the test dataset to measure the accuracies of the models trained.

### Mini-batch Gradient Descent

We trained the network using mini-batch gradient descent learning with 128 batch size and learning rate 𝛼 = 0.001. The images were scaled using the formula:

We ran the model with 1000 epochs and measure the training cost and validation cost.

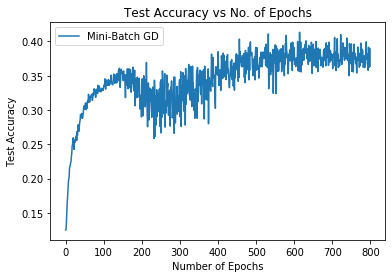
The result is as follows:



Here, we can observe that at around 100 epochs, the training cost and validation cost are starting to diverge. The rate of decrease of the validation cost is also decreasing as number of epochs go up. Above 800 epochs, the validation cost is almost constant or event increasing.

With this observation, we implemented early stopping. The idea is to prevent overfitting by stopping the training when validation loss starts to increase even when training loss is still decreasing. We adjusted the number of epochs to be 800. Number of epochs means the number of times the model has gone through the training data.

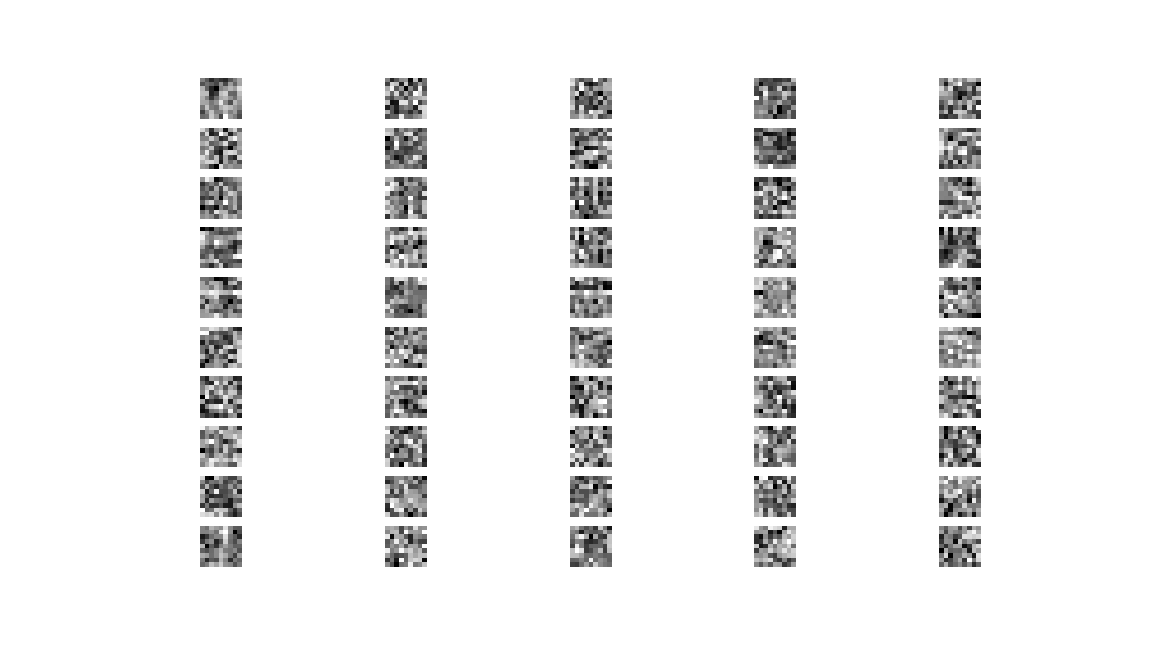
Using early stopping at 800, we have plotted the test accuracy vs epoch graph.



We can see that indeed, the test accuracy is stagnating and using early stopping mechanism, we help save training time.

Next, we plotted the feature maps at both convolution layers and pooling layers along with the test patterns.

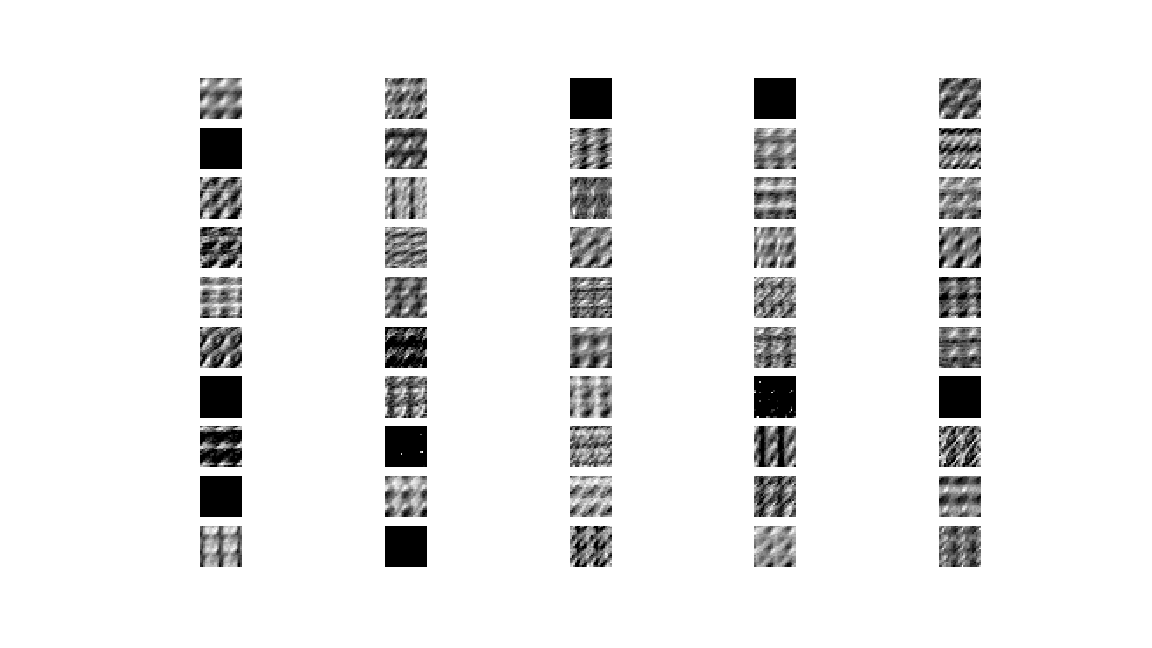
Weights learned at convolution layer 1 C1:



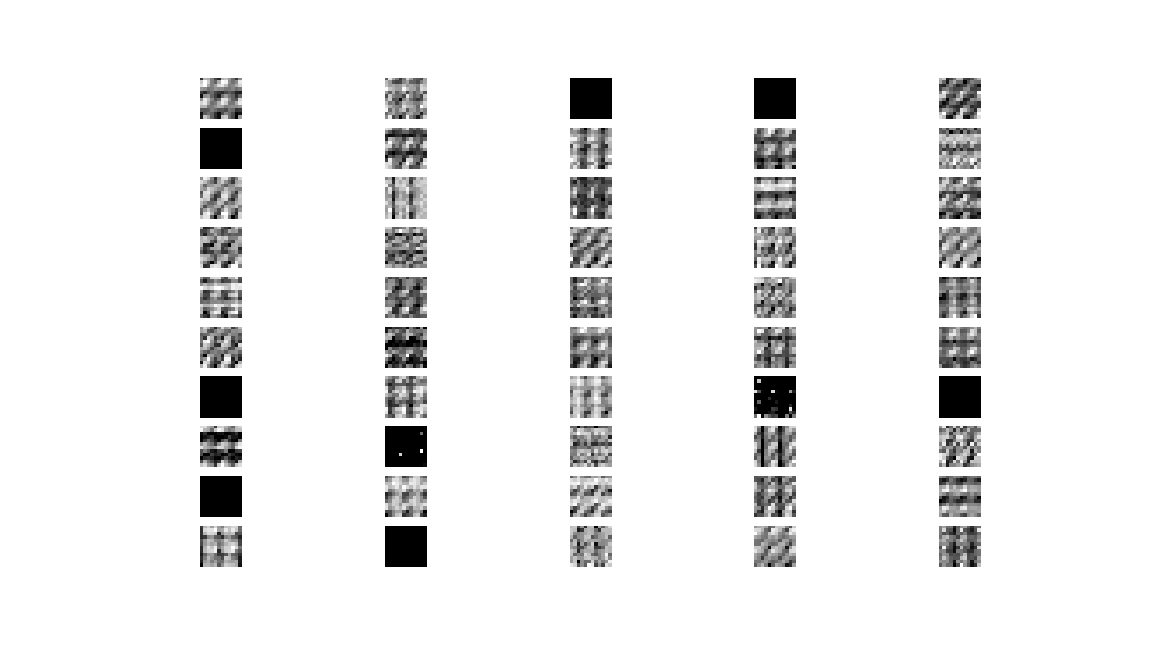
Test Pattern 1:

Project%202/Part%20A/Q1/figures/1b_1.png

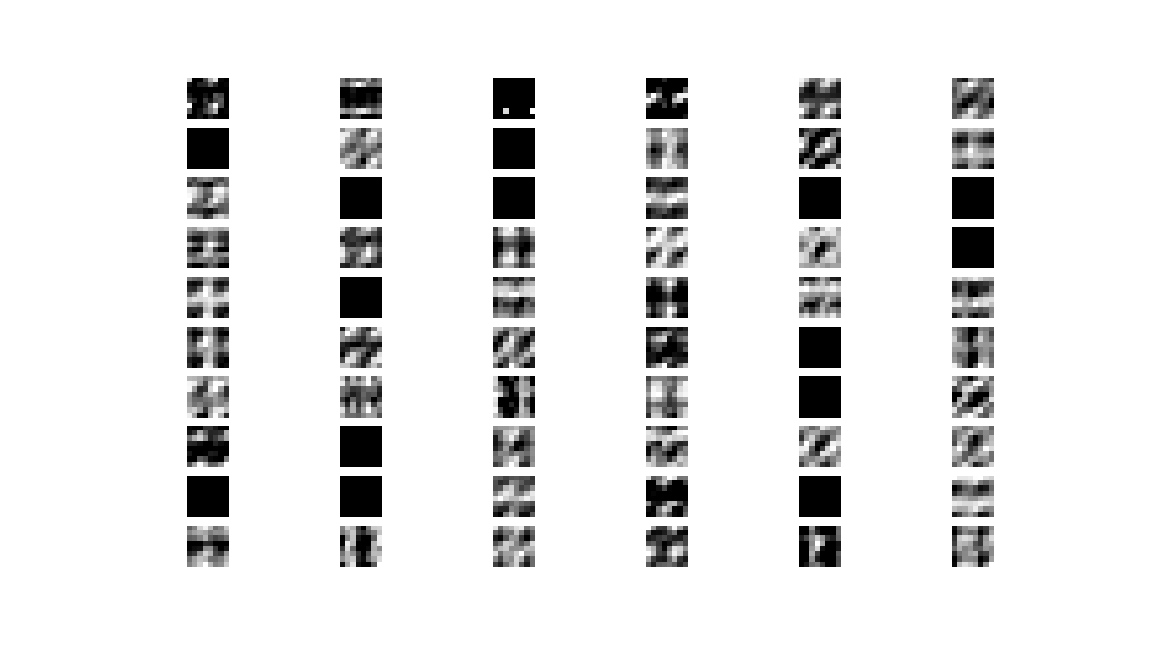
Feature map at convolution layer 1 C1:



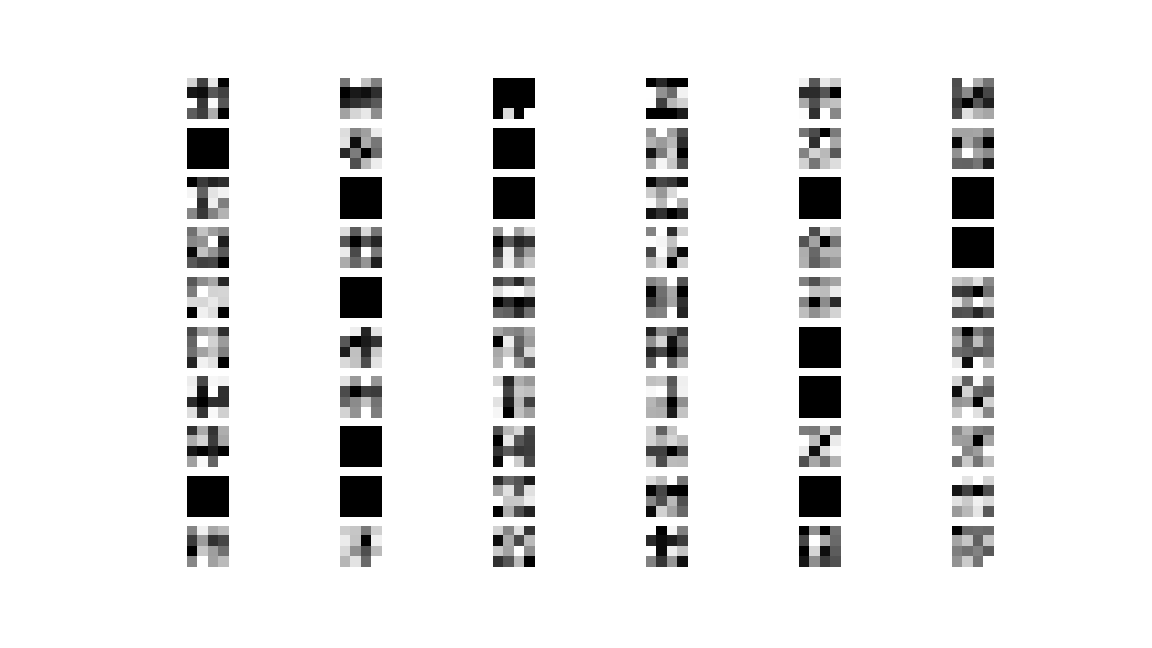
Pooling Layer S1:



Feature map at convolution layer 2 C2:



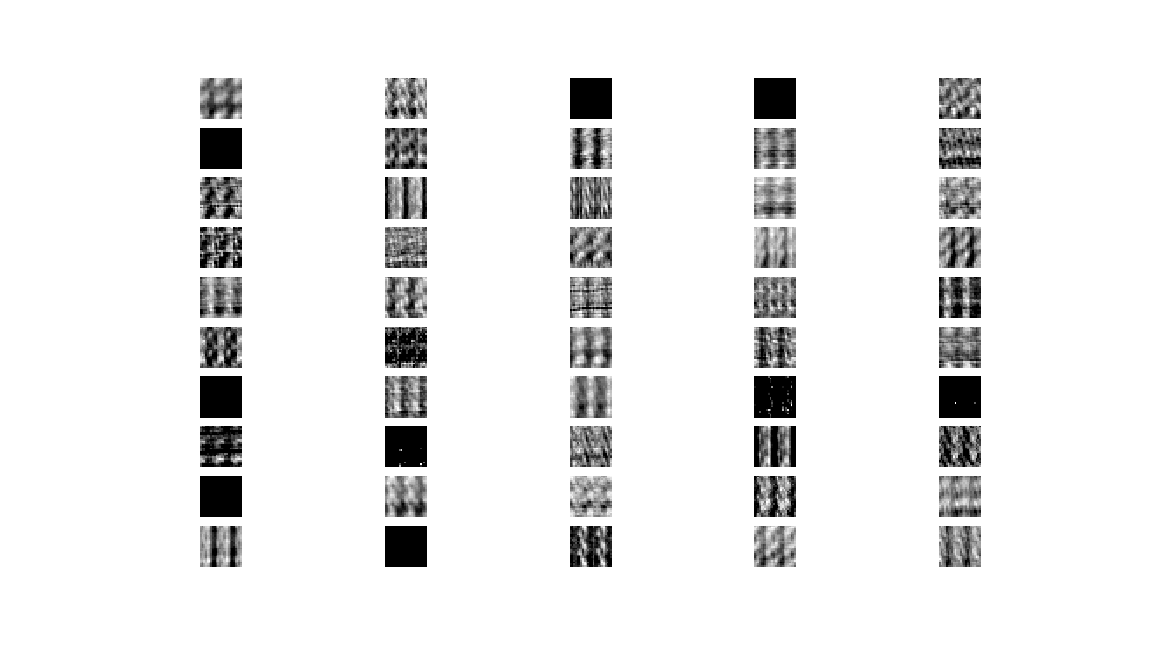
Pooling Layer S2:

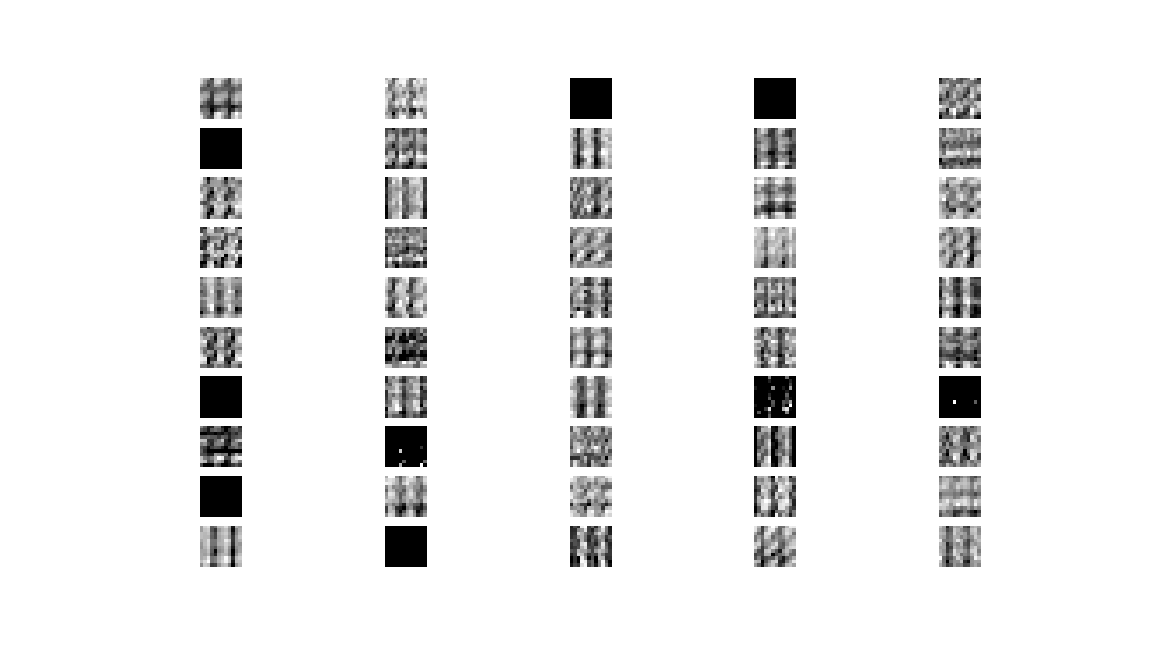


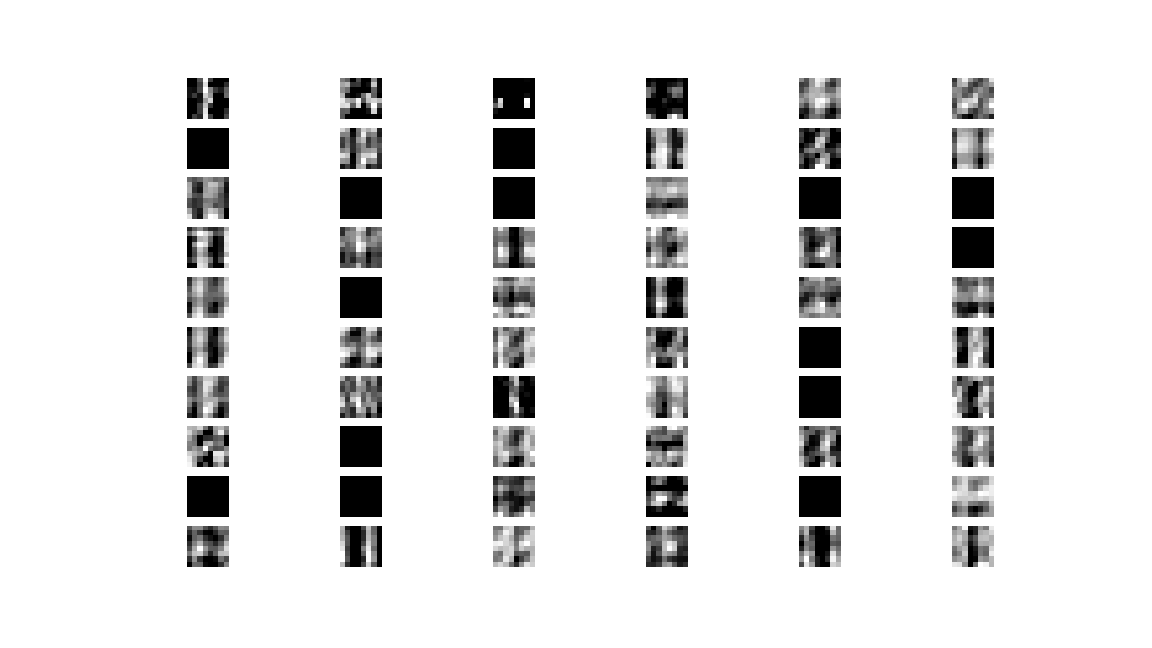
Using test pattern 2:

Project%202/Part%20A/Q1/figures/1b_2.png

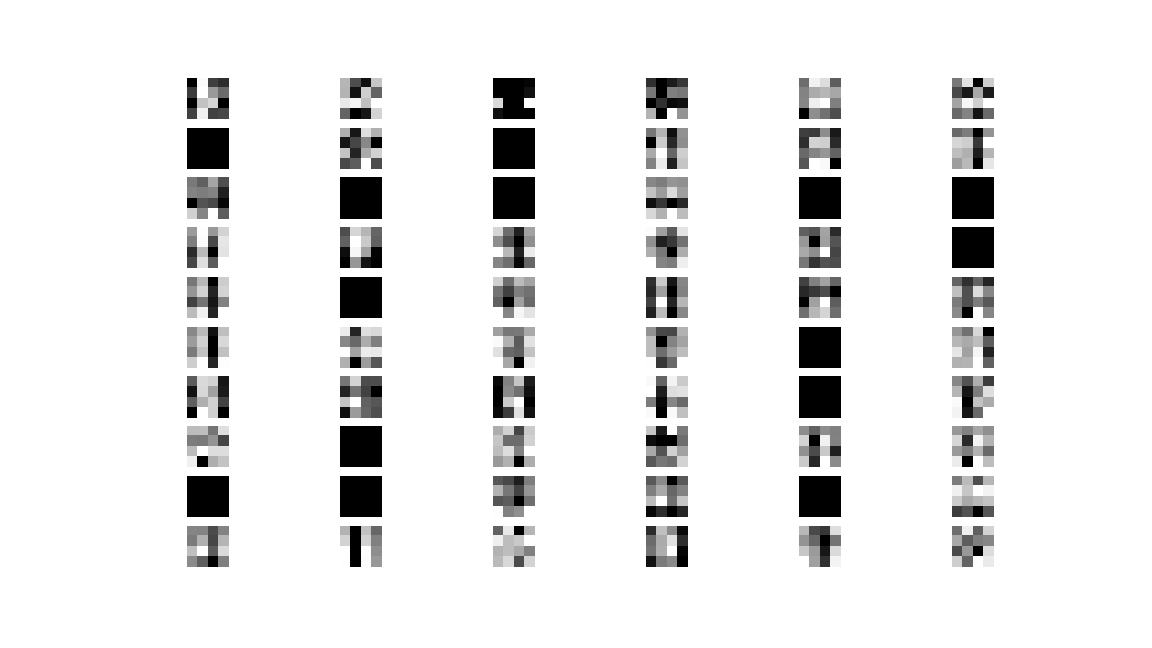
Feature map at convolution layer 1 C1:



Pooling Layer S1:

Feature map at convolution layer 2 C2

Pooling Layer S2:



From the feature maps, we can see that the weights learned at the first convolutional layer is not meaningful. This result is expected as we have only 40% accuracy at the end of the training. Even if we increase the number of epochs to 1000, the accuracy doesn’t improve significantly.

There are some possibilities for this observation:

* The dataset is too complex for the model. This may be because of the complex features across classes which exist in the CIFAR-10 dataset.
* We only use one batch of the CIFAR-10 dataset, hence less accurate model resulted.
* Dataset uses 3 color channel, which may increase complexity further.

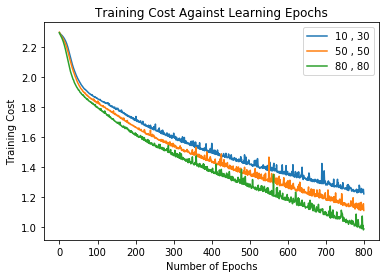
Next, we used a grid search to find the optimal numbers of feature maps at the convolutional layers. We used the test accuracy to determine the optimal numbers.

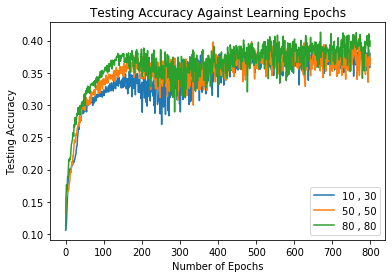
Here, we have used several combinations for grid search at both C1 and C2. This have been selected by choosing a few nearest neighbours after each round. Some hyperparameters we tried are as follows:

hyper\_parameter = [(10,30), (50,50), (80, 80)]

hyper\_parameter = [(80,80), (80,85), (80, 90), (85,90), (90,90)]

We start small and try several combinations around the best results. For the first round, we get the following results:

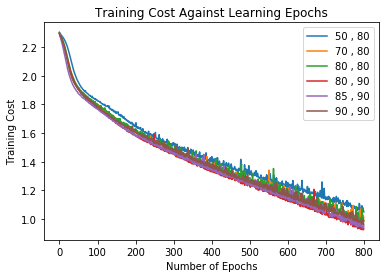
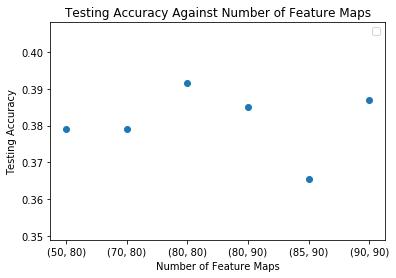
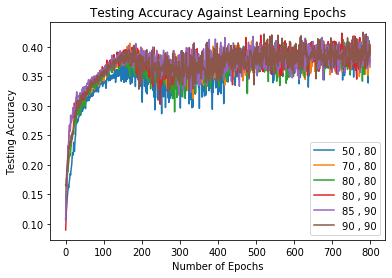




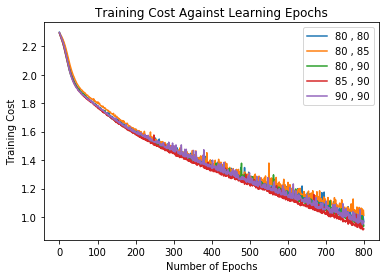


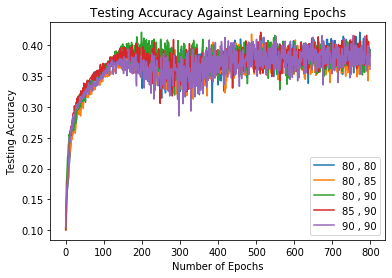
As we can see, the best testing accuracy was obtained with (80, 80) as hyperparameters. This result is expected as we have more filters, the training cost will be lower.

Next search is around (80, 80) as it gives the best testing accuracy. The results are as follows:



After the second search, we observed that the best number of feature maps is around 80-90. We then performed another search around this figure. The result is as following:



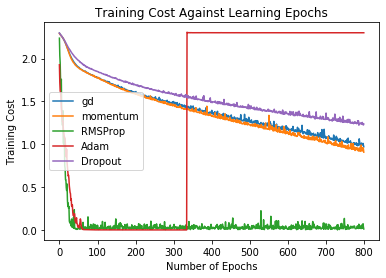


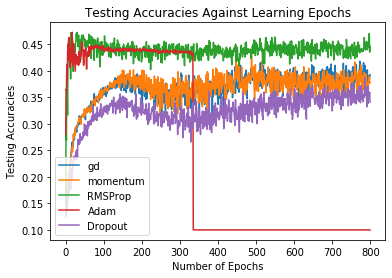
As we can see here, the best optimal number of feature maps is (80, 80).

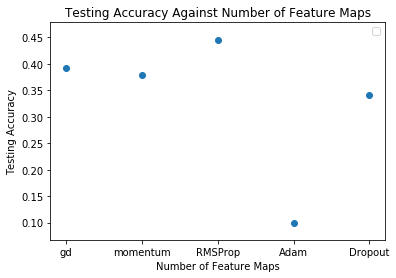
In the next part, we use the optimal number of filters found to train the network using different methods:

* Adding 0.1 as momentum term
* Using RMSProp algorithm for learning
* Using Adam optimizer for learning
* Adding dropout to the layers

Here are the results of the trainings:

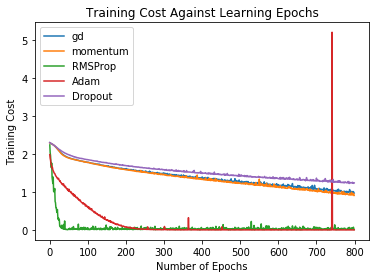


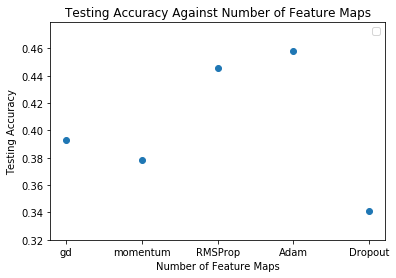
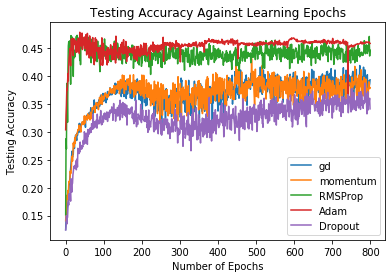




Based on the results, we can see that RMSProps method gives us the best performance in terms of training loss and testing accuracy. It also converges faster than other methods.

There is a strange observation in the Adam optimizer graph. It is possible that when the Adam optimizer is close to optimal, the moving average of squares of gradients can become extremely small. This will cause the reverse to be an extremely large number, causing spikes in the graph. One way to prevent this is to reduce the learning rate. We ran another round of training with Adam optimizer using a learning rate lower by a power of 10, keeping the other methods the same as before. The result is as follows:





As we can see, that the anomaly in result for Adam has disappeared. We concluded that it was indeed due to the cause previously discussed. This time, the Adam method gives better accuracy but this is expected as the learning rate is much lower.

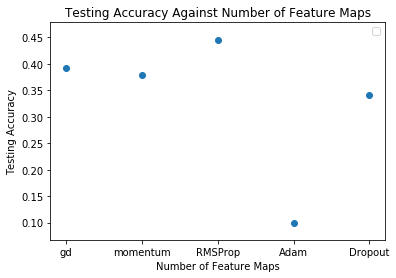
## After all the experiments, we compared the models from the three parts and discuss their performance.

In part (1), we trained the model using mini batch gradient descent. We could see from the result that the test accuracy of the final model is not very good at 40%. One of the factors causing this is clearly because the complexity of the CIFAR-10 dataset. The model is not complex enough to yield higher accuracy. There are many complex features across the CIFAR-10 dataset and this is further amplified by 3 color channels used.

In part (2), we tried to find the optimal number of feature maps. We achieved higher accuracy compared to part (1) as the number of filters are increased, thus increasing the complexity of model. But the improvement is not that significant. Also, due to the small training dataset, overfitting is very possible.

During the experiment, we tried using hyperparameters above (90, 90) but it gave us an out of memory error. We tried using Google Colab GPU and it still throws this error. We also started from small number of features for the initial layers. This is because we want to increase the number of layers in the later parts to get more complex structures. If we use a large number at the beginning, it will not be manageable in the later layers.

In part (3), we trained the model again using multiple optimization algorithm. Using these optimization, we obtained better results, especially using Adam optimizer and RMSProp for learning. Mini batch gradient descent does not work well with CIFAR-10 dataset as it tends to oscillate near the optimum, causing a slow convergence rate. Mini batch gradient descent also reduces the variance of individual patterns and achieves stable convergence, but at the expense of true minimum of the complex error profiles. This works against the CIFAR-10 dataset as it has very complex error profiles.



The best results are obtained using RMSProp given the optimal number of feature maps according to our experiments. This method uses exponentially decaying average to decay from extreme gradients. Comparatively, using Adam optimizer with lower learning rate gives better test accuracies but converges slower than RMSProp.

## Conclusion

We have to keep in mind that we are using a very complex dataset (CIFAR-10). We also only use the first batch of the dataset which results in insufficient training data. Moreover, the convolutional layer model we use is also not complex enough to learn this dataset.

From all our training, we can see that the results are not good (40%-50%) even after using optimization methods. In comparison, there are models which can get up to 80% accuracies on this same dataset. But the models are much more complex and they use the full CIFAR-10 dataset.

Here are some improvements we can do:

* Use the full CIFAR-10 dataset
* Increase the number of convolution and pooling layer
* Use cross validation techniques along with optimization algorithm such as Adam optimizer and RMSProps
* Use fully connected layer

# Project 2B: Text Classification

## Introduction

We are given a dataset containing Wikipage entries and labels (0-14) that indicate what category the Wikipage belongs to.

Information regarding the dataset is given below:

|  |  |
| --- | --- |
| **train\_medium.csv** | **test\_medium.csv** |
| **No of entries in dataset** = 5600 | **No of entries in dataset** = 700 |
| **No of possible output classes** = 15  Only the first 100 letters/words are fed into the model. | |

Visualization of a fragment *(10 tuples)* of the training dataset is shown below:

A screenshot of a computer

Description automatically generated

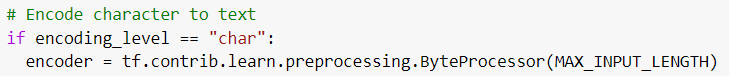
In this part-project, convolutional and recurrent neural network architectures will be used to train the model to classify the category of a Wikipage based on the Wikipage article title (input encoded at the character-level) and on the article’s first paragraph content (input encoded at the word-level). Different types of architectures and experiments will be performed on both the CNNs and RNNs to see how to optimize or what would improve each architecture’s test accuracy—these will involve using dropout, and in the RNN case, gradient clipping, multiple cell/neuron layers and the RNN cell type used (i.e. LSTM, GRU, Vanilla [BasicRNNCell])

## Methods

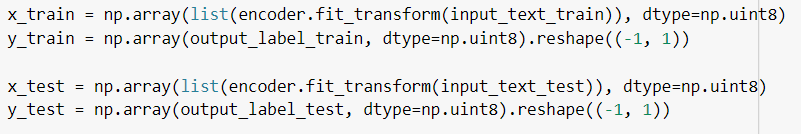
### Character-level Encoding for Character Classifiers

When **train\_medium.csv** and **test\_medium.csv** are read for the character classifiers used in this part-project, the second row of the input file (which contains the Wikipage article title) is processed as follows:

1. A **TensorFlow ByteProcessor** encoder is created that encodes the first 100 (MAX\_INPUT\_LENGTH = 100) characters in the second row.



1. The contents of the second row are encoded with this encoder to form an input tensor of size (INPUT\_DATASET\_SIZE [5600 for train and 700 for test], 100) to be fed into the CNN or RNN model.



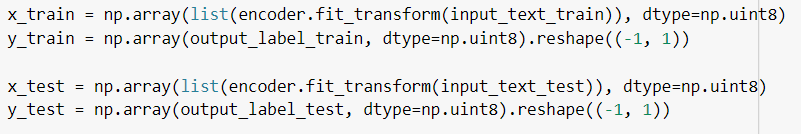
### Word-level Encoding for Word Classifiers

When **train\_medium.csv** and **test\_medium.csv** are read for the word classifiers used in this part-project, the third row of the input file (which contains the Wikipage article’s first paragraph text content) is processed as follows:

1. A **TensorFlow VocabularyProcessor** encoder is created that encodes the first 100 (MAX\_INPUT\_LENGTH = 100) words in the third row.

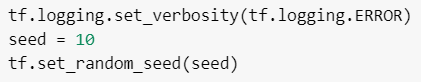


1. The contents of the third row are encoded with this encoder to form an input tensor of size (INPUT\_DATASET\_SIZE [5600 for train and 700 for test], 100) to be fed into the CNN or RNN model.



### Seed initialization for predictable pseudo-randomness

The seed for initializing weights and biases for the model are always the **same**; so, every time training is run, the initial weights and biases are the same. This causes initialization to be predictable so that the only factor causing training to run differently is solely in the change of hyperparameters. In addition, the seed for shuffling the dataset for mini-batch stochastic gradient descent is also kept the same at the beginning of each training. This causes the script to return consistent results every time it is run.



### 

### Training using Mini-batch Stochastic Gradient Descent

When training, mini-batch stochastic gradient descent will be used with a batch size of 128. The output of each model is passed through a final softmax layer, before its softmax cross-entropy is computed. This entropy will be the quantity TensorFlow’s optimizer will minimize. The learning rate for training used is 0.05, and the network is trained for 100 epochs. During each training, the test accuracy (1 - misclassification rate) and “train” entropy at the end of each epoch will be recorded. In addition, the “training” accuracy (1 - misclassification rate) (accuracy computed when the training dataset is fed into the feed\_dict) and the “test” entropy (entropy computed when the test dataset is fed into the feed\_dict) are also recorded at each epoch. The initial networks trained in Section 1 and 2 will have their trainings timed using **time.time()** and compared in Section 5.

### CNN Network Architectures Used

The table below outlines the CNN network architectures used, alongside the section where the network architecture is mentioned in the report:

|  |  |
| --- | --- |
| **Character-CNN**  **[Section No. 1]** | **Word-CNN**  **[Section No. 2]** |
| **Convolution Layer 1**  10 filters of window size 20 x 256  VALID padding  ReLU neurons  **Max Pooling Layer 1**  Pooling window with size 4 x 4  Stride = 2  SAME padding  **Convolution Layer 2**  10 filters of window size 20 x 1  VALID padding  ReLU neurons  **Max Pooling Layer 2**  Stride = 2  SAME padding | **Convolution Layer 1**  10 filters of window size 20 x 20  VALID padding  ReLU neurons  **Max Pooling Layer 1**  Pooling window with size 4 x 4  Stride = 2  SAME padding  **Convolution Layer 2**  10 filters of window size 20 x 1  VALID padding  ReLU neurons  **Max Pooling Layer 2**  Stride = 2  SAME padding |
| **Character-CNN with Dropout**  **[Section No. 5]** | **Word-CNN with Dropout**  **[Section No. 5]** |
| Same architecture as Character-CNN, but Dropout rates of 0.2 (0.2 chance of a neuron being dropped) applied to each conv-pool layer. | Same architecture as Word-CNN, but Dropout rates of 0.2 (0.2 chance of a neuron being dropped) applied to each conv-pool layer. |

### RNN Network Architectures Used

The table below outlines the RNN network architectures used, alongside the section where the network architecture is mentioned in the report:

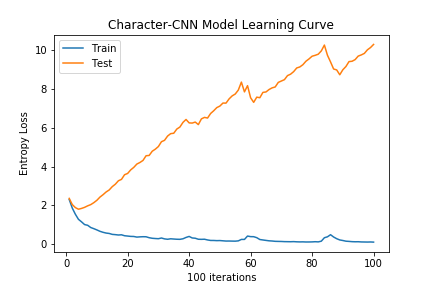
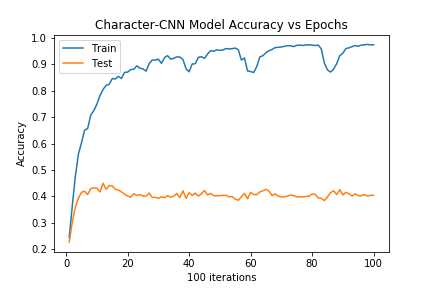
|  |  |
| --- | --- |
| **Character-RNN**  **[Section No. 3]** | **Word-RNN**  **[Section No. 4]** |
| GRU Layer  Hidden-layer size = 20 | Word vectors passed to **embedding** layer of size 20 before being fed to RNN.  GRU Layer  Hidden-layer size = 20 |
| **Character-RNN with Dropout**  **[Section No. 5]** | **Word-RNN with Dropout**  **[Section No. 5]** |
| Same architecture as Character-RNN, but a Dropout rate of 0.2 is applied to the RNN layer. | Same architecture as Word-RNN, but a Dropout rate of 0.2 is applied to the RNN layer. |
| **Character-RNN with Vanilla Cell**  **[Section No. 6a]** | **Word-RNN with Vanilla Cell**  **[Section No. 6a]** |
| Vanilla (BasicRNNCell) Cell Layer  Hidden-layer size = 20 | Word vectors passed to **embedding** layer of size 20 before being fed to RNN.  Vanilla (BasicRNNCell) Cell Layer  Hidden-layer size = 20 |
| **Character-RNN with LSTM Cell**  **[Section No. 6a]** | **Character-RNN with LSTM Cell**  **[Section No. 6a]** |
| LSTM Cell Layer  Hidden-layer size = 20 | Word vectors passed to **embedding** layer of size 20 before being fed to RNN.  LSTM Cell Layer  Hidden-layer size = 20 |

|  |  |
| --- | --- |
| **2-layer Character-RNN**  **[Section No. 6b]** | **2-layer Word-RNN**  **[Section No. 6b]** |
| GRU Layer  First hidden-layer size = 20  Second hidden-layer size = 20 | Word vectors passed to **embedding** layer of size 20 before being fed to RNN.  GRU Cell Layer  First hidden-layer size = 20  Second hidden-layer size = 20 |
| **Character-RNN with Gradient Clipping**  **[Section No. 6c]** | **Word-RNN with Gradient Clipping**  **[Section No. 6c]** |
| GRU Layer  Hidden-layer size = 20  Gradient clipping threshold = 2 | Word vectors passed to **embedding** layer of size 20 before being fed to RNN.  Gradient clipping threshold = 2 |

## Experiments and Results

### Character CNN Classifier

The regular Character-CNN classifier outlined above is trained. Below is the classifier’s entropy cost, a plot of the training and test accuracy vs epoch number, and the final entropy costs and accuracies of the model:

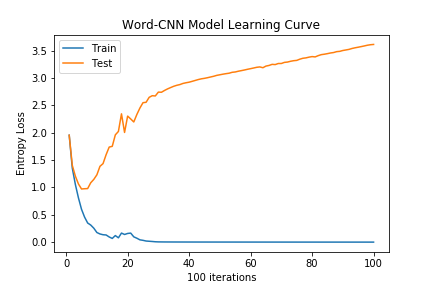
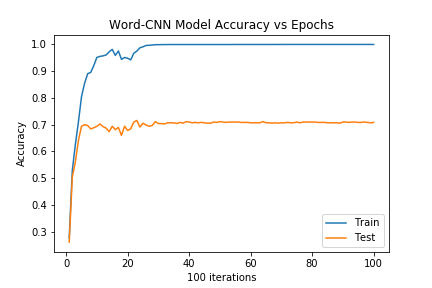
|  |  |  |
| --- | --- | --- |
|  | Dataset used | |
| Train | Test |
| Final accuracy | 0.974 | 0.404 |
| Final entropy loss | 0.0956 | 10.3 |

Notice that though the entropy on training data generally decreases and converges to nearly zero, the entropy on testing data increases after less than 10 epochs, indicating that the network has **overfit** the training set. This is further proven by the training accuracy being far higher than the testing accuracy in the accuracy vs epochs plot.

The regular character-CNN classifier may not be a good ‘fit’ network for the test dataset. Regularizing techniques such as dropout or finding other features to classify paragraph content to category may be needed instead.

### Word CNN Classifier

The regular Word-CNN classifier outlined above is trained. Below is the classifier’s entropy cost, a plot of the training and test accuracy vs epoch number, and the final entropy costs and accuracies of the model:

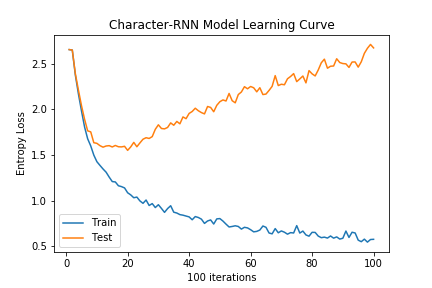
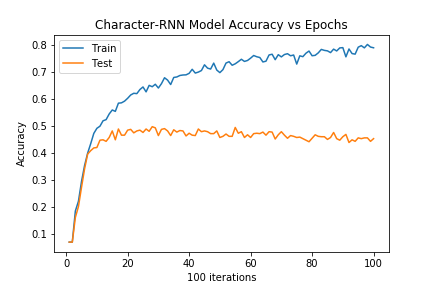
|  |  |  |
| --- | --- | --- |
|  | Dataset used | |
| Train | Test |
| Final accuracy | 0.999 | 0.709 |
| Final entropy loss | 4.77 x 10-4 | 3.61 |

Notice that though the entropy on training data generally decreases and converges to nearly zero, the entropy on testing data increases after less than 10 epochs, indicating that the network has **overfit** the training set. This is further proven by the training accuracy being far higher than the testing accuracy in the accuracy vs epochs plot.

The regular word-CNN classifier may also not be a good-enough ‘fit’ network for the test dataset. Regularizing techniques such as dropout or finding other features to classify paragraph content to category may be needed instead. In addition, a CNN is not appropriate to fit this dataset. This is because the dataset is sequential data and is temporal in nature. A convolutional neural network exploits features within local regions of the dataset, which is more appropriate for image data. A **recurrent** neural network is a more appropriate network to classify this data, since it exploits the temporal nature of the paragraph data (outputs of data at sequence T depends on previous sequences (i.e. 1, 2, …, T – 1)). This reasoning also applies to the Character-CNN classifier.

### Character RNN Classifier

The regular Character-CNN classifier outlined above is trained. Below is the classifier’s entropy cost, a plot of the training and test accuracy vs epoch number, and the final entropy costs and accuracies of the model:

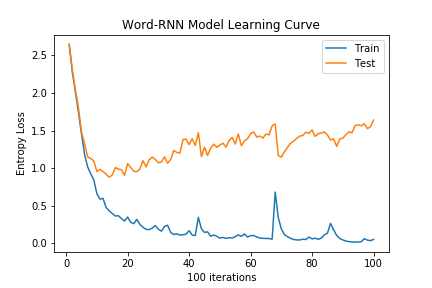
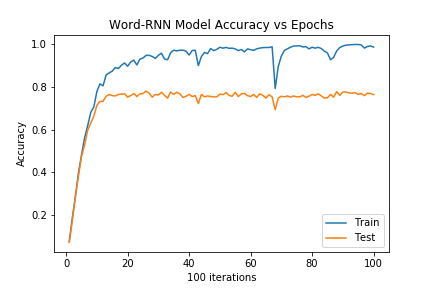
 

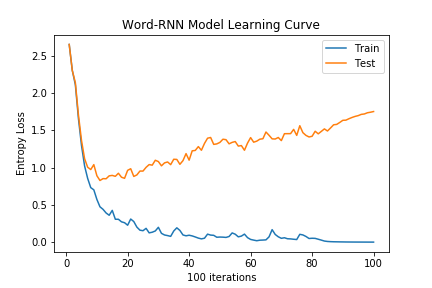
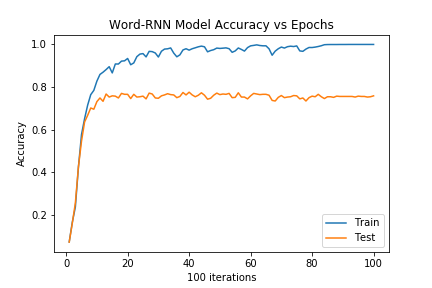
|  |  |  |
| --- | --- | --- |
|  | Dataset used | |
| Train | Test |
| Final accuracy | 0.791 | 0.454 |
| Final entropy loss | 0.578 | 2.67 |

Again, the network has **overfit** the training set, as the entropy loss on test data has diverged away from the minimum after nearly 20 epochs. The test accuracy of the model is also worse than the previous Word-CNN model, only reaching 0.454, which is about 0.05 higher than the Character-CNN model. This is a good indicator that the RNN model performs better than the CNN model in classifying paragraph title data or character/word data that is, by nature, temporal.

### Word RNN Classifier

The regular Word-CNN classifier outlined above is trained. Below is the classifier’s entropy cost, a plot of the training and test accuracy vs epoch number, and the final entropy costs and accuracies of the model:

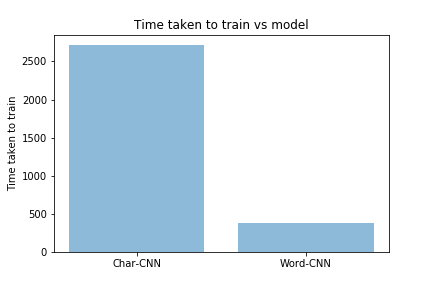
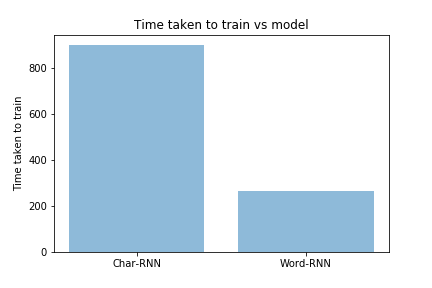
|  |  |  |
| --- | --- | --- |
|  | Dataset used | |
| Train | Test |
| Final accuracy | 1.0 | 0.759 |
| Final entropy loss | 0.00167 | 1.75 |

The entropy on test data still increases in the Word-RNN case, but it does not increase at an alarming rate, only increasing from slightly below 1 to 1.75. The Word-RNN also performs better than the previous three models, reaching a final test accuracy of 0.759. This shows that (1) encoding Wikipage paragraph data at the **word-level** is the best approach to pre-process the data before feeding it to the RNN and (2) an RNN network exploits the nature of the paragraph data much more effectively than a CNN network (for the reasons mentioned above in the Word-CNN classifier section (Section 2)).

### Comparing CNN and RNN Classifier Training Time and Adding Dropout to CNN and RNNs

#### Comparing Training Times

The training times for the 4 networks are visualized and mentioned below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Encoding Level | |
| Character | Word |
| Network | CNN | 2900.3 | 402.1 |
| RNN | 900.7 | 265.2 |

Time taken to train in seconds for different models

Some patterns arise from these observed data:

1. Word-level encoded classifiers trained faster than character-level encoded classifiers.
   1. The character-CNN uses a 20 x 256 filter while the word-CNN uses a 20 x 20 filter in its first layer. Hence the character-CNN requires to update more parameters during training and hence training time for the character-CNN is more than that of the word-CNN.
   2. The input fed into the character-RNN at sequence = t has dimensions [1, 256], while for the word-RNN it is [1, 20]. Hence there are more parameters to train and update for the character-RNN (from the input layer to the hidden GRU layer) than for the word-RNN during backpropagation through time in mini-batch stochastic gradient descent.
2. Fixing the encoding level, RNNs train in a shorter time than CNNs. This could be presumably because the CNN has two layers and has 10 filters in each layer to train and update. Because of the CNN’s higher ‘relative complexity’ in terms of the number of parameters to train than the RNN, longer training times are needed for this CNN than this CNN’s corresponding RNN. However, if the RNN has, in general, the same number of parameters to train as the CNN, RNN training time would significantly take a much longer time due to its sequential parameter updating scheme using backpropagation through time (BPTT) (i.e.: Character-RNN taking a longer time to train than Word-CNN)

#### Adding Dropout to CNNs

The following are the entropy and accuracy plots of the Character and Word CNN when a dropout rate of 0.2 is applied to each conv-pool layer:

#### 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Char-CNN w/ out Dropout | Char-CNN with Dropout | Word-CNN w/ out Dropout | Word-CNN with Dropout |
| Final Training Accuracy | 0.974 | 0.879 | 0.999 | 0.985 |
| Final Test Accuracy | 0.404 | 0.444 | 0.709 | 0.674 |
| Final Training Loss | 0.0956 | 0.398 | 4.77 x 10-4 | 0.0677 |
| Final Test Loss | 10.3 | 3.77 | 3.61 | 1.75 |

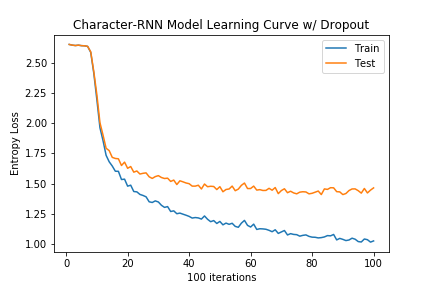
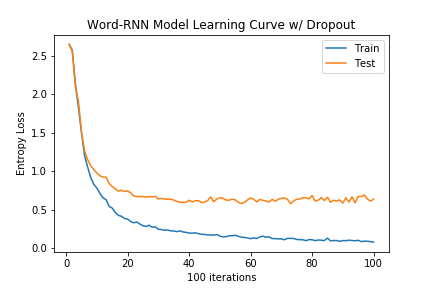
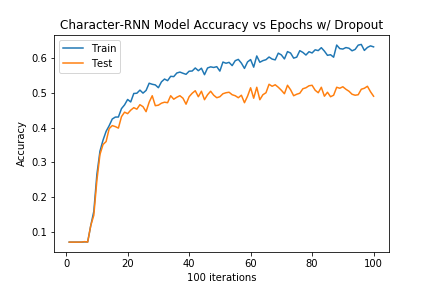
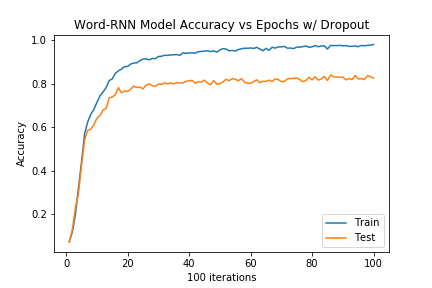
As seen above, both the Character and Word CNNs with dropout applied to them would in general have **less** training accuracy and have **greater** entropy loss with training data. However, the Character and Word CNNs with dropout would have **greater** training accuracies and **less** entropy loss with test data than the Character and Word CNNs without dropout.

This is because dropout is a regularization technique to prevent overfitting of training data by not relying too much on certain features (some neurons are ‘dropped’ to reduce dependencies on this neuron during training). Hence both networks with dropout can fit the test dataset much better than those without dropout.

The only **exception** to this trend is the test accuracy for the Word-CNN with Dropout, which is 0.674, lower than 0.709, which is the test accuracy for the Word-CNN without Dropout. This is presumably because we allowed the network to train for 100 epochs and overfitting still occurs for the Word-CNN. Had early stopping been done, the test accuracy for the Word-CNN might have been higher with dropout than without. Another possible reason for this may be because the Word-CNN is already a relatively good-fit model (of its kind, of CNN), so regularizing using dropout did not affect the model result by much and instead made classification error even higher.

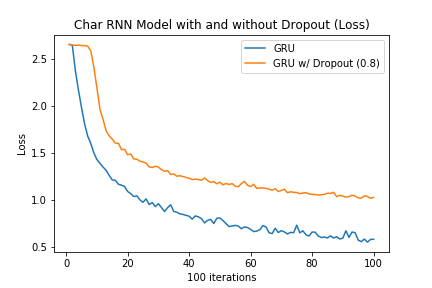
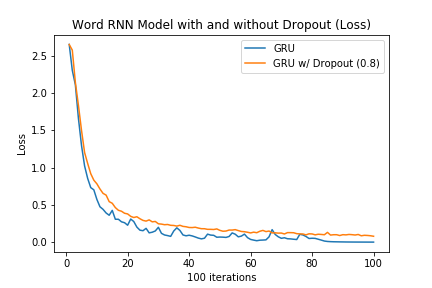
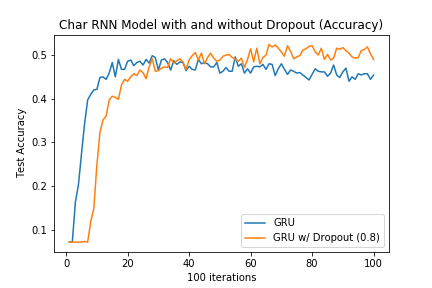
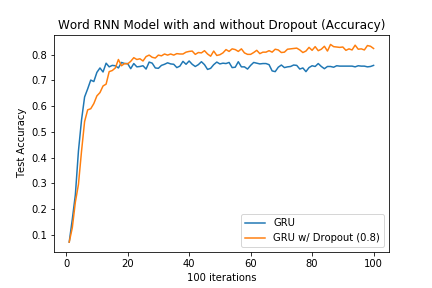
#### Adding Dropout to RNNs

The following are the entropy and accuracy plots of the Character and Word CNN when a dropout rate of 0.2 is applied to each conv-pool layer:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Char-RNN w/ out Dropout | Char-RNN with Dropout | Word-RNN w/ out Dropout | Word-RNN with Dropout |
| Final Training Accuracy | 0.791 | 0.632 | 1.0 | 0.980 |
| Final Test Accuracy | 0.454 | 0.490 | 0.759 | 0.824 |
| Final Training Loss | 0.578 | 1.02 | 0.00167 | 0.0801 |
| Final Test Loss | 2.67 | 1.46 | 1.75 | 0.640 |

Further figures were plotted in the next page to compare the RNN networks with and without dropout:

Again, from the figures above, both the Character and Word RNNs with dropout applied to them would in general have **less** training accuracy and have **greater** entropy loss with training data. However, these networks with dropout would have **greater** training accuracies and **less** entropy loss with test data than the Character and Word CNNs without dropout.

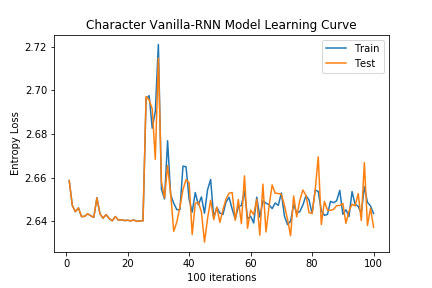
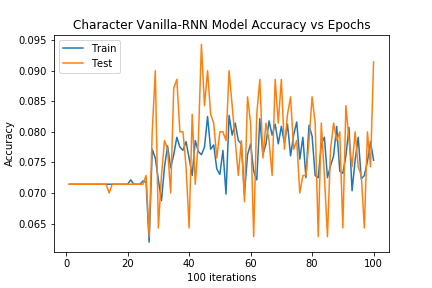
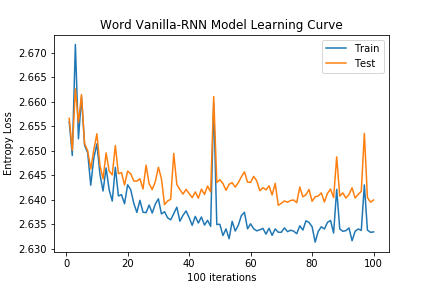
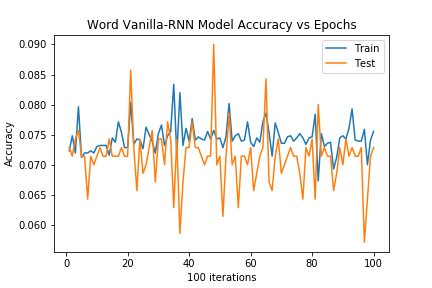
This is because dropout is a regularization technique to prevent overfitting of training data by not relying too much on certain features (some neurons are ‘dropped’ to reduce dependencies on this neuron during training). Hence both networks with dropout can fit the test dataset much better than those without dropout.

### Further Experiments with RNN Architectures

#### RNNs with Vanilla/LSTM cells vs GRU

##### Vanilla

The figures below show the results of replacing the Character and Word RNN cell type from GRU to Vanilla (BasicRNNCell from TensorFlow):

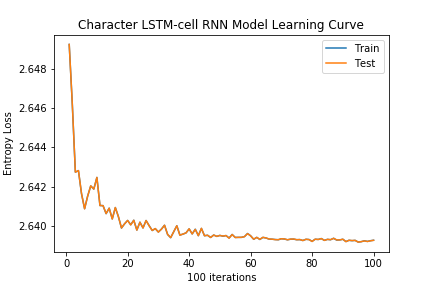
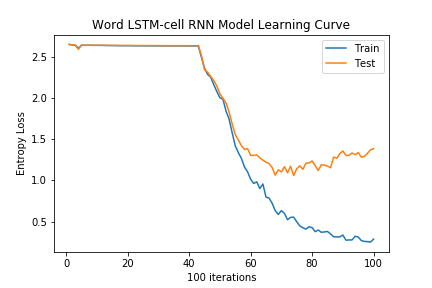
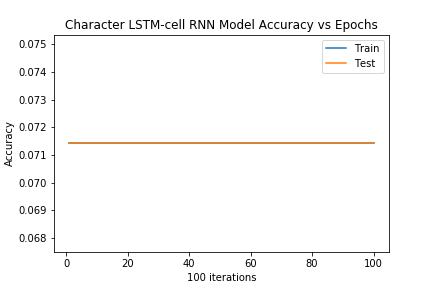
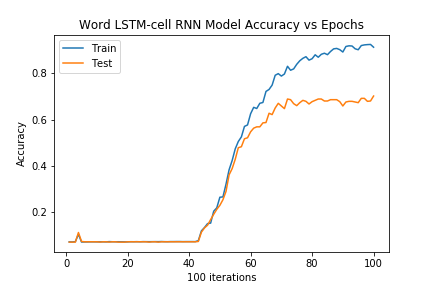
   

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Char-RNN w/ GRU | Char-RNN w/ Vanilla | Word-RNN w/ GRU | Word-RNN w/ Vanilla |
| Final Training Accuracy | 0.791 | 0.0754 | 1.0 | 0.0755 |
| Final Test Accuracy | 0.454 | 0.0914 | 0.759 | 0.0729 |
| Final Training Loss | 0.578 | 2.64 | 0.00167 | 2.63 |
| Final Test Loss | 2.67 | 2.64 | 1.75 | 2.64 |

For both character and word RNNs, when the cell type layer is replaced to Vanilla, training and test accuracy does not significantly increase from its initial value at the beginning of training. Likewise, entropy for train and test data also does not significantly decrease as training progresses. The RNN architectures for Vanilla do not train until convergence after 100 epochs, hence the final accuracies (respectively entropy losses) of Vanilla Char-RNN are far lower (respectively far higher) than that of GRU Char-RNN.

This may be because convergence takes far more than 100 epochs. This could also be because of the Vanishing Gradient problem, where the RNN gradients vanish (there is no gate ‘memory’ in Vanilla RNN cells) up to zero and learning does not proceed.

###### LSTM

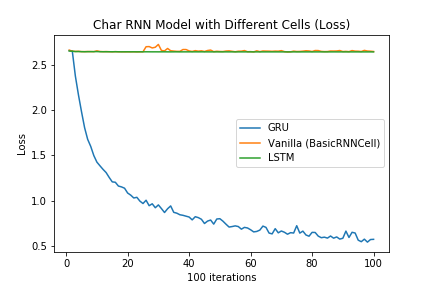
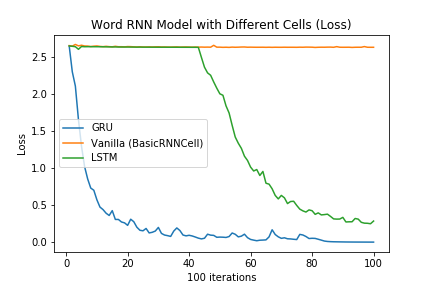
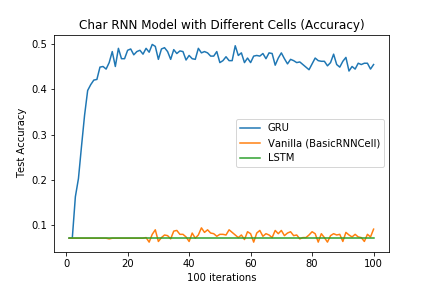
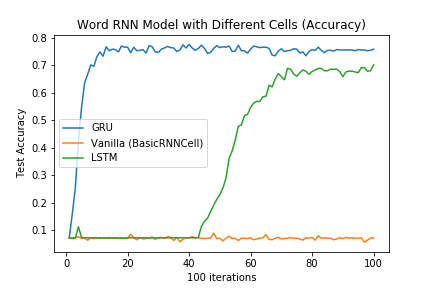
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Char-RNN w/ GRU | Char-RNN w/ LSTM | Word-RNN w/ GRU | Word-RNN w/ LSTM |
| Final Training Accuracy | 0.791 | 0.0714 | 1.0 | 0.912 |
| Final Test Accuracy | 0.454 | 0.0714 | 0.759 | 0.701 |
| Final Training Loss | 0.578 | 2.64 | 0.00167 | 0.286 |
| Final Test Loss | 2.67 | 2.64 | 1.75 | 1.38 |

For the character-RNN, when the cell type layer is replaced to LSTM, training and test accuracy also does not significantly increse from its initial value. Likewise, entropy for train and test data does not significantly decrease as training progresses. The reason why the character-RNNs do not ‘learn’ is the same with that of the Vanilla RNNs mentioned above.

For the word-RNN, when the cell type layer is replaced to LSTM, learning can still happen after slightly more than 40 iterations, hence the vanishing gradient problem is overcome. The final test accuracy for LSTM word-RNN is 0.701, which is lower than that of GRU word-RNN at 0.759. Hence the GRU word-RNN still performs better than the LSTM word-RNN.

##### Overall verdict

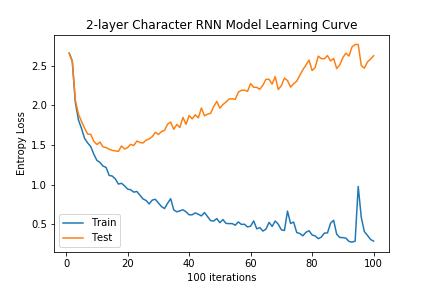
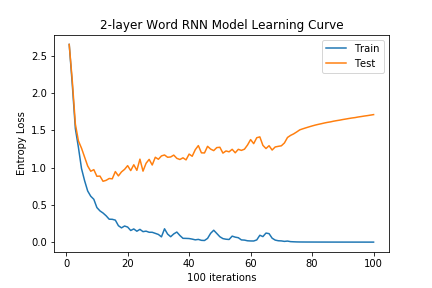
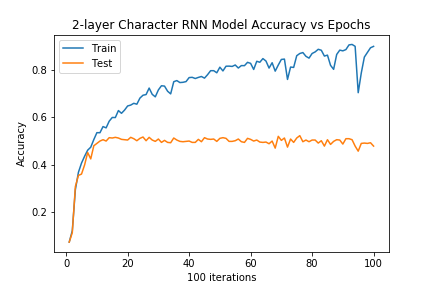
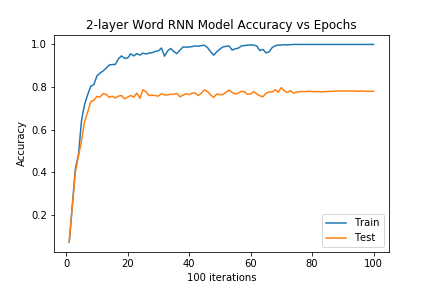
More figures comparing the cell type used for the RNN are plotted below:

According to these figures, it is obvious that using GRU as the cell layer type is the best choice for the RNN model architecture, both at the character and word-level.

#### 1-layer vs 2-layer RNNs

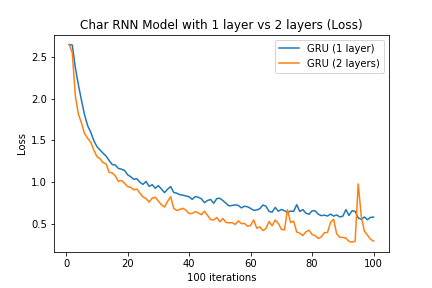
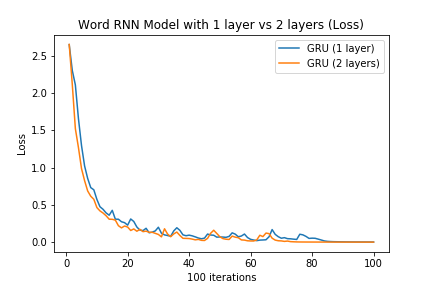
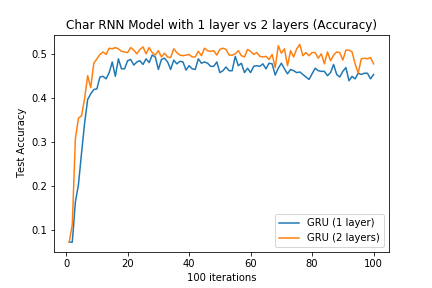
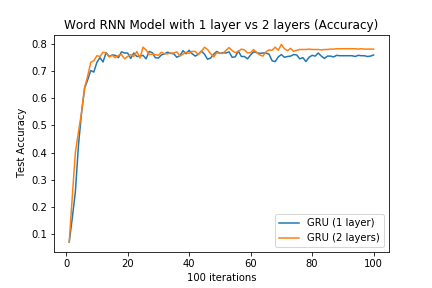
The figures below show the results of using a 2-layer RNNs as compared to a 1-layer RNN:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1-layer Char-RNN | 2-layer Char-RNN | 1-layer Word-RNN | 2-layer Word-RNN |
| Final Training Accuracy | 0.791 | 0.901 | 1.0 | 1.0 |
| Final Test Accuracy | 0.454 | 0.479 | 0.759 | 0.780 |
| Final Training Loss | 0.578 | 0.291 | 0.00167 | 0.000438 |
| Final Test Loss | 2.67 | 2.62 | 1.75 | 1.71 |

According to the figures above, for both Character and Word RNNs, if the number of GRU layers increase from 1 to 2, the final test accuracy (respectively training loss) slightly increases (respectively slightly decreases). The 2-layer character RNN has a final test accuracy of 0.479, higher than its 1-layer counterpart at 0.454. Likewise, the 2-layer character RNN has a final test accuracy of 0.780, higher than its 1-layer counterpart at 0.759.

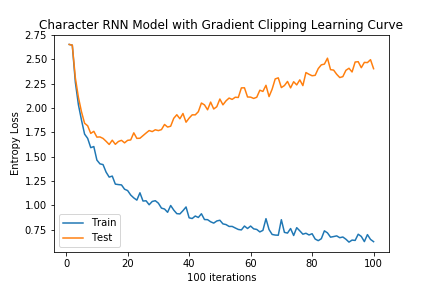
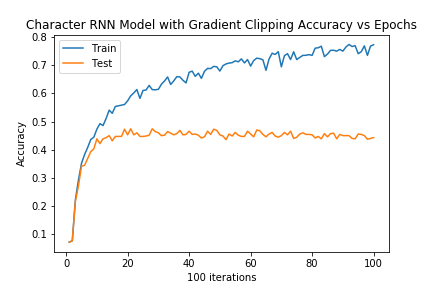
Hence the 2-layer RNNs are more accurate and fit the training dataset better than their 1-layer RNN counterparts. More figures to illustrate this are plotted in the next page:

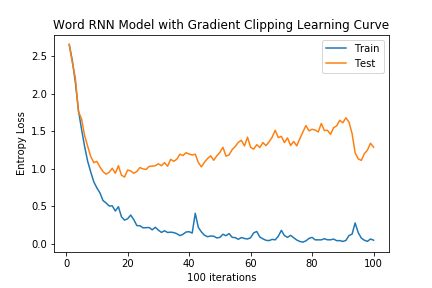
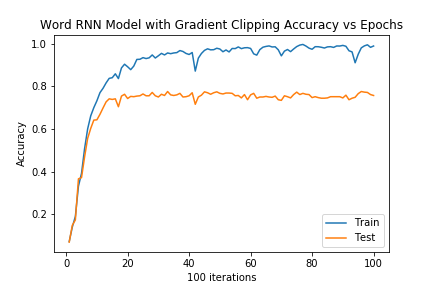
  

Hence 2-layer RNNs in this case are more preferred than their 1-layer RNN counterparts. This may be because the 2-layer RNNs can approximate more complex functions, and hence can fit the test dataset better as the network has not overfit yet. Albeit, the increase in test accuracy is not so significant, but its increase is regardless still there and can be seen in the figures.

#### RNNs with and without Gradient Clipping

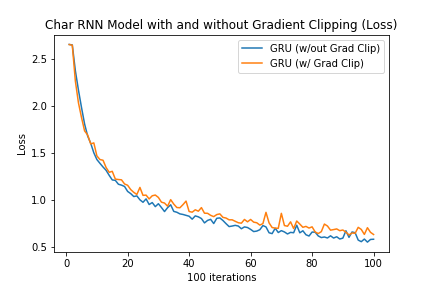
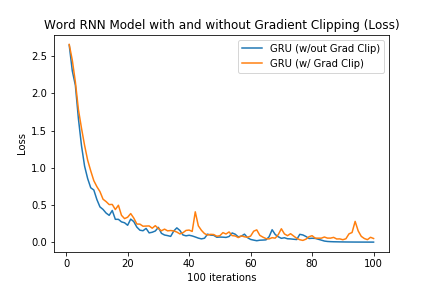
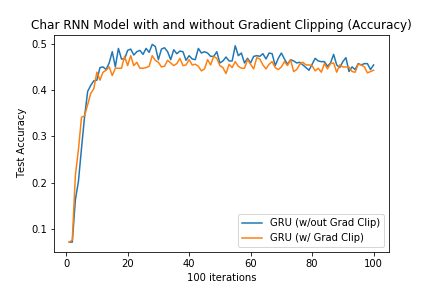
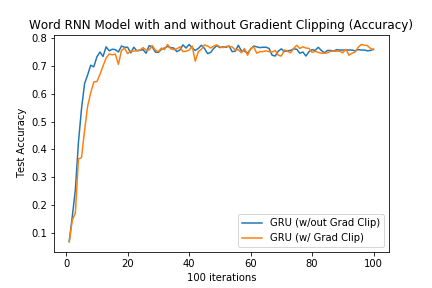
The figures below show the results of using gradient clipping with threshold 2.0 as compared to if gradient clipping isn’t applied:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Char-RNN w/ out Gradient Clipping | Char-RNN w/ Gradient Clipping | Word-RNN w/ out Gradient Clipping | Word-RNN w/ Gradient Clipping |
| Final Training Accuracy | 0.791 | 0.773 | 1.0 | 0.989 |
| Final Test Accuracy | 0.454 | 0.443 | 0.759 | 0.757 |
| Final Training Loss | 0.578 | 0.629 | 0.00167 | 0.0503 |
| Final Test Loss | 2.67 | 2.40 | 1.75 | 1.29 |

More figures are plotted on the next page to compare the character and word RNNs with and without gradient clipping:

According to these four figures above, gradient clipping with a threshold of 2 does not seem to have a significant effect on the ‘speed to convergence’ of the model and does not have a significant effect on the model test accuracy both at the character level and the word level.

This may be because the optimization space does not have any sudden peaks or valleys that trap training in local minima, so whether gradient clipping is applied or not does not affect the where final ‘optima’ is that is reached by the model.

Clipping gradients at lower norms will also not affect the model test accuracy and rate of convergence (these experiments were tried afterwards, but this data was not recorded nor plotted).

## Conclusions

In this part-project, we have learned how to utilize CNNs and RNNs to classify paragraph data and ‘categorize’ them, as well as the considerations (i.e.: RNN cell type used, encoding level for input, etc.) one should take to come up with a good model architecture to optimize model accuracy on test data.