NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# CZ4042 – Neural Networks Project 2 Report


**Prepared by:**

Jordan Lam Yu Cheng JLAM014@e.ntu.edu.sg

Nathanael Raj nathanae001@e.ntu.edu.sg


*Note: Reports for Part A and Part B have been organised separately.*

Part 1

## Introduction

We utilised tiny images from the cifer-10 datasets. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. However, we are using a small subset of these for our training and test sets. The images are RGB with size 32 x 32.

This dataset contain classes such as airplane, automobile, bird, cat, dog etc.

## Methods

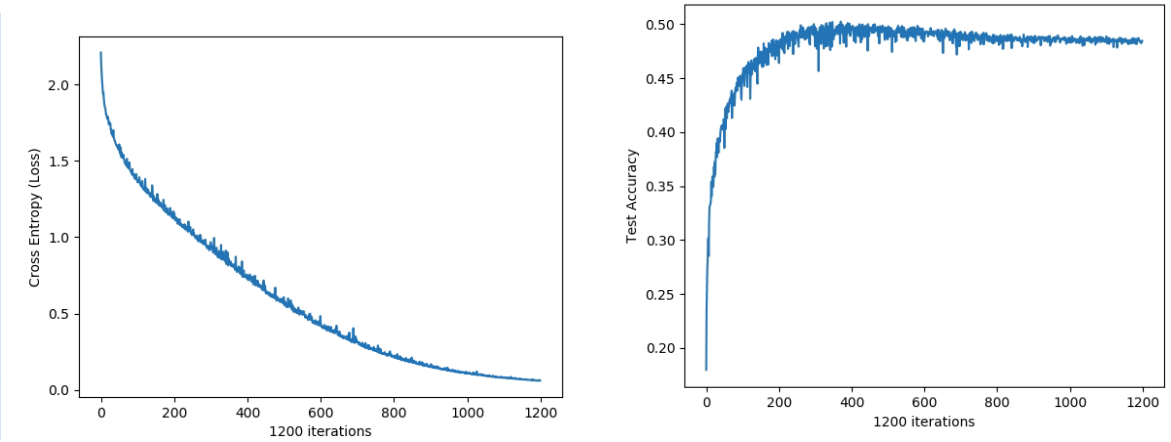
For this project, we implement a 2-layer convolutional neural network with pooling layers that feed into a fully connected layer. The final output is made using a SoftMax layer of size 10.

Here, we examine the effects of different learning algorithms like basic gradient descent, momentum, RMSProp and Adam optimizer as well as looking at the effects of dropouts.

## Experiment and Results

Here, we feed the inputs to the various methods above and compare the results using parameters like test accuracy, time to convergence and overall shape of the test and train curves. We also will utilize grid search to determine the optimal number of neurons in each convolutional layer.

1a)



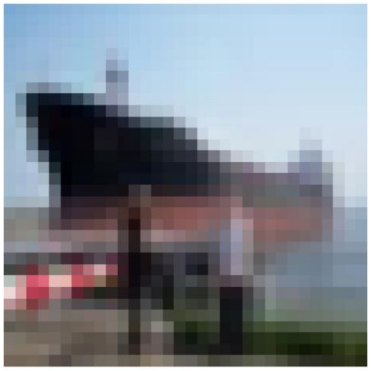

Left: Training cost (loss) vs epochs

Right: Test Accuracy vs epochs

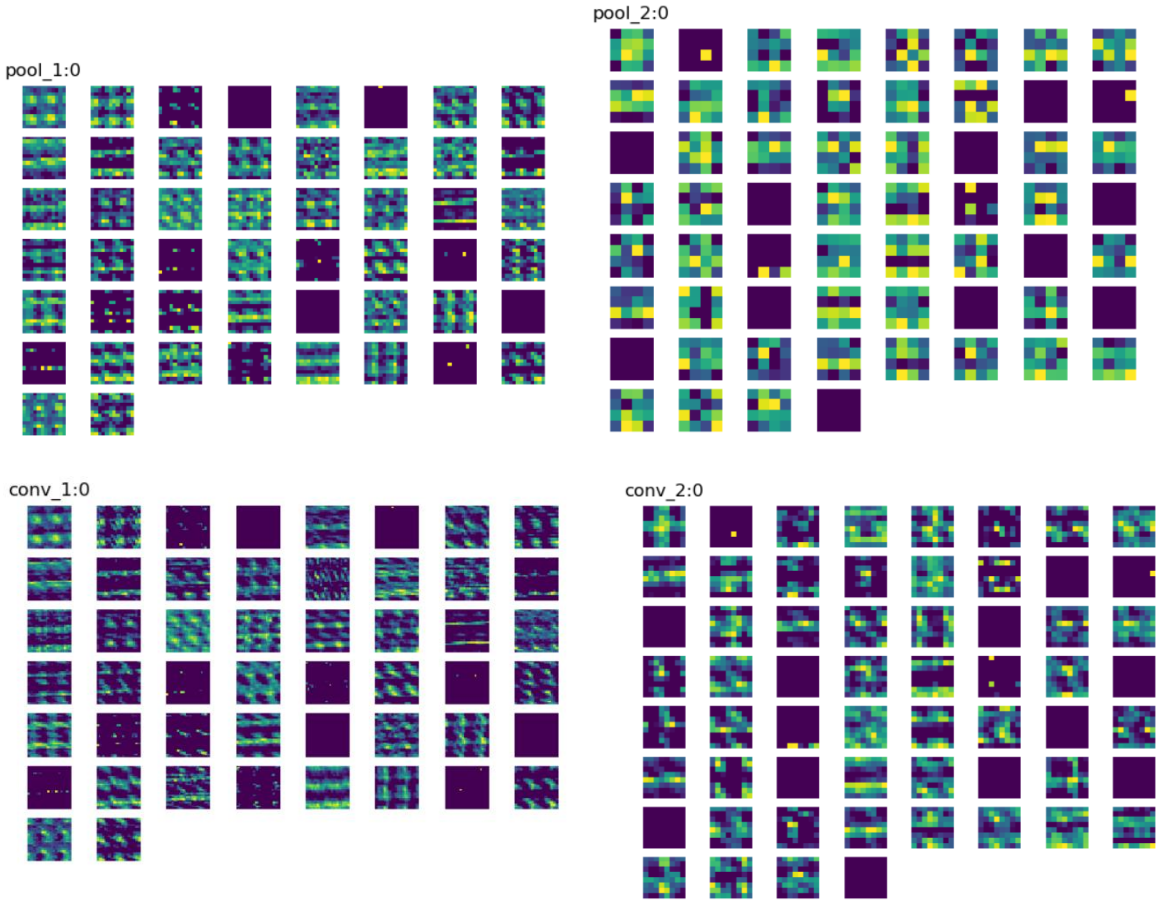
Even at 1200 epochs convergence has not occurred but test error clearly shows a downward trend showing we have over trained and slight overfitting is occurring.

1b)

Random Picture Selected

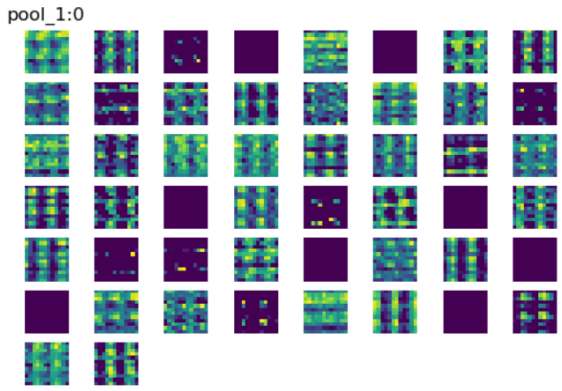


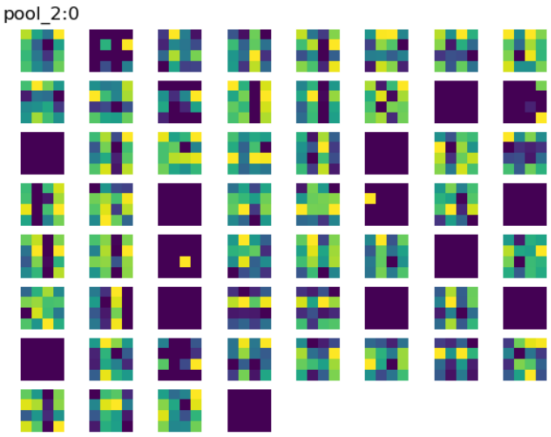
Feature Map activations for selected picture

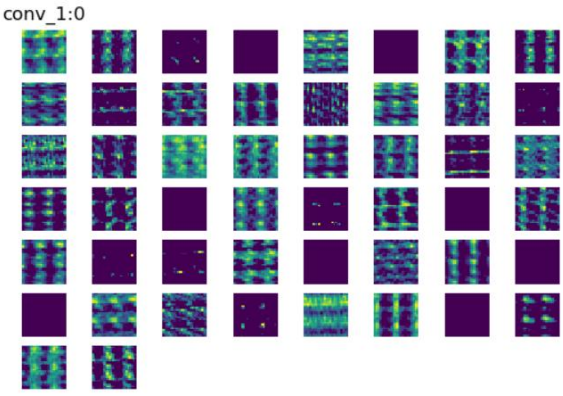


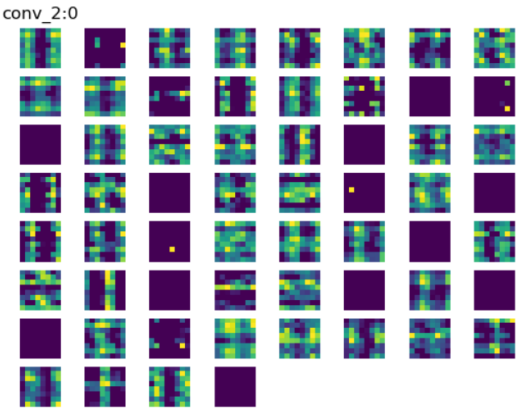
Random Picture Selected

Feature Map activations for selected features.



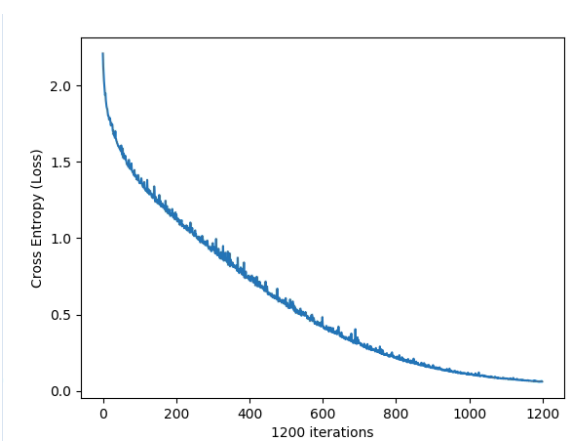



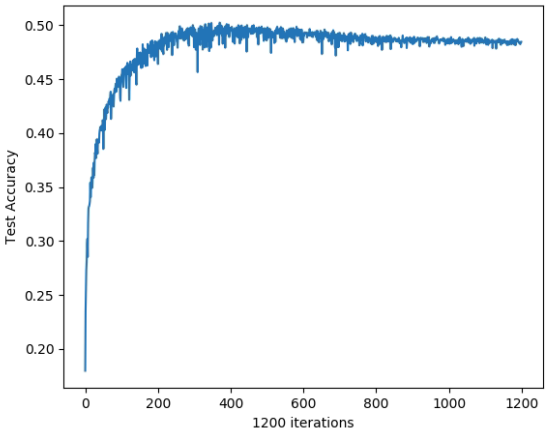



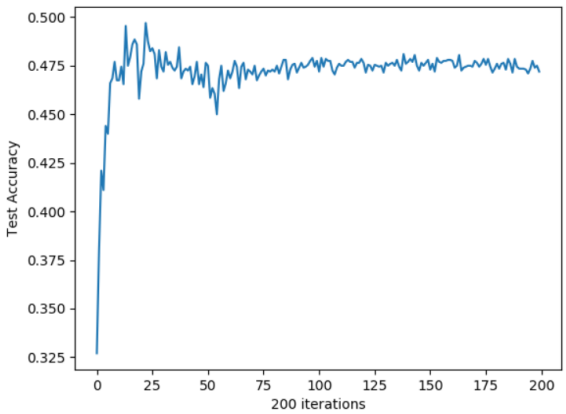
2)

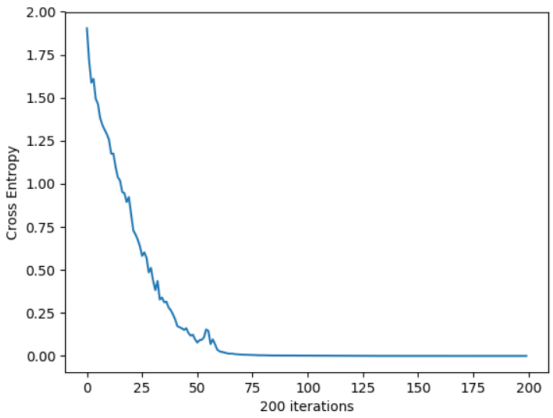
To find a good hyperparameter, we will use the search space [30,40,50,60,70,80]. However, convergence is slow for the normal gradient descent optimizer with convergence not met even after 1200 epochs. A proper and fine search would probably take time in an order of days.




We thus should explore other methods of improving the learning times. Using other faster optimizers will greatly improve the search time. For this purpose, we will utilize the adam optimizer which shows a much faster convergence rate.




Using Adam Optimizer, convergence occurs much earlier at about 75 epochs, about an order of magnitude increase in speed and this will make a finer search more possible.

The best test accuracies for each point in the search space are:

[[0.459  0.4715 0.453  0.4625 0.466  0.4575]

 [0.4555 0.461  0.469  0.4565 0.4555 0.4555]

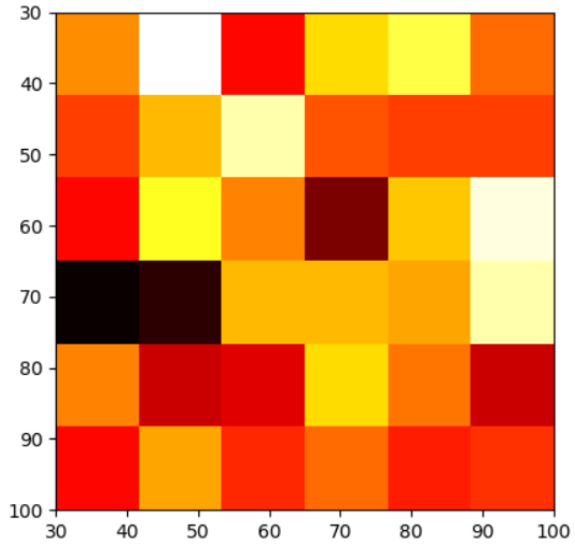
 [0.453  0.465  0.4585 0.447  0.4615 0.4705]

 [0.442  0.4435 0.461  0.461  0.46   0.469 ]

[0.4585 0.4505 0.4515 0.4625 0.458  0.4505]

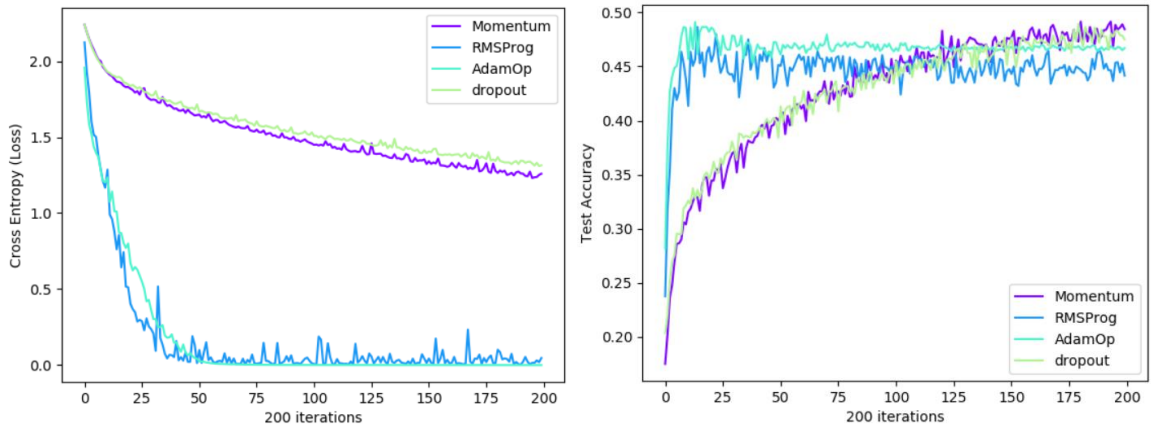
[0.453  0.46   0.4545 0.4575 0.454  0.455 ]]

With entry at (0,0) corresponding to 30 feature maps in both convolutional layer.



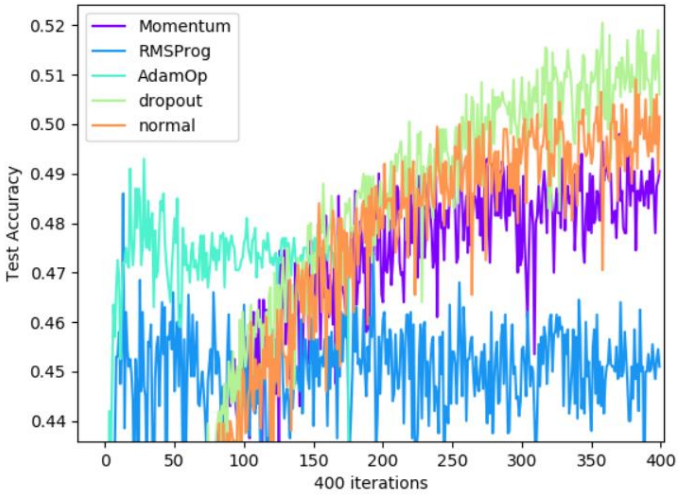

To represent this, we can use a heat map. Plotting the Search space above (white max, black min) shows which areas have the highest test accuracy. The highest point would be 30 feature maps for the first layer and 40 feature maps for the 2nd layer.

3)




From here we see that adamop and rmsprop converges much faster, however, Momentum and dropout have not converged.

Running a longer experiment with 400 epochs below:




Zooming in to the top layer, we see that dropout does in face perform better than the rest after a longer training period.

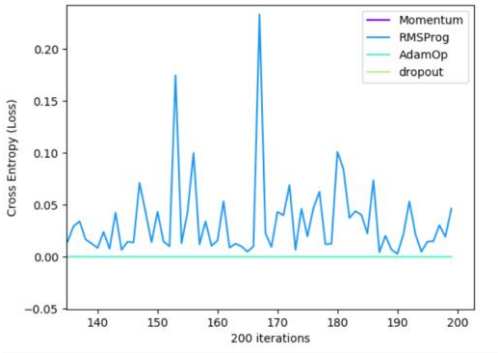
4)

**Speed**

From here, the most obvious observation would be the speed in which RMSProp and AdamOp converges is much faster than compared to normal optimization with dropout and Momentum.

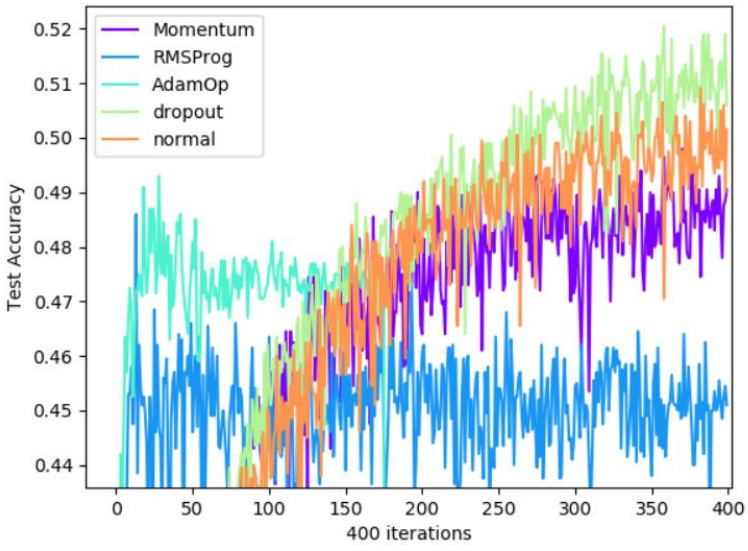
RmsProp and AdamOp both converges at around 50 epochs while even at 200 epochs Dropout and Momentum have yet to converge.

**Stability**




Even at convergence, we see that RMSProp does have some fluctuations while AdamOp remains stable at convergence.

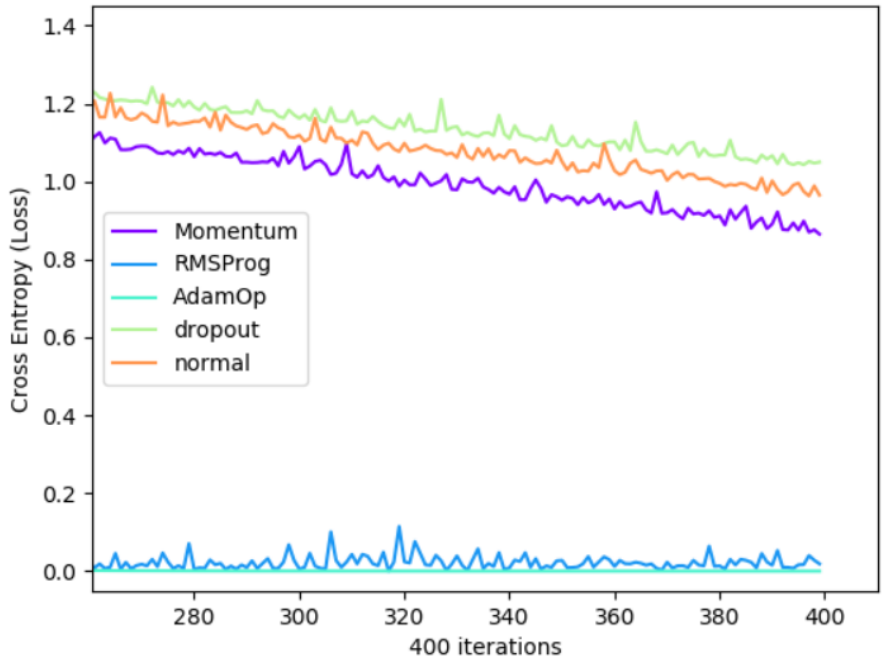
**Accuracy**




Running part 3 again for a longer period and adding in the model from part 1 (shown as “normal”) shows that after a long training period, dropout performed the best while the faster optimizers performed worse.

Dropout eventually had the highest accuracy of 52% at epoch 357. In this case, dropouts had helped to reduce overfitting and forced the model to better generalise the data leading to a better accuracy after a long training period.

Looking at the loss:




We can see the effect of dropout as a form of regularization that stops the loss function from falling as far as the other models. Although it had the highest loss, after a long period of training it had the best accuracy.

# Neural Networks Project 1 Part B

## Introduction

In this project, we aim to classify a dataset derived from Wikipedia entries. The dataset consist of the first paragraphs collected from Wikipedia page entries and the corresponding labels about their category. The labels are made up of 15 possible types of entries, for example, people, companies, schools, etc.

We trained the dataset using Convolutional (CNN) and Recurrent (RNN) Neural Networks using different processing and neural network techniques to compare the effect of each on the accuracy of our classifier.

## Methods

We implemented the both types of neural networks using the TensorFlow framework. For both types of networks, we experimented with using both word and character inputs to the network. For CNNs, we used a two layer convolution and pooling followed by a dense SoftMax layer and experimented with using dropouts on the dense layer. For RNNs, we used a single layer of 20 GRUs and a dense SoftMax layer and experimented with using dropouts on both layers. Subsequently we experimented with using different types of cells in the recurrent layer, namely vanilla RNN cells and LSTM cells, we also experimented with a two layer GRU and with gradient clipping.

## Experiment and Results

In this section, I will give an outline of the implementation of the basic CNN and RNN set ups followed by addressing the questions given.

**Experiment set up**

I will outline the process of both CNN and RNN architectures with an example using word inputs. The set ups which used character inputs differed only in the way the text was processed and is not the focus of this report

```
import numpy as np
import pandas
import tensorflow as tf
import csv

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

MAX_DOCUMENT_LENGTH = 100
HIDDEN_SIZE = 20
MAX_LABEL = 15
EMBEDDING_SIZE = 20
N_FILTERS = 10
FILTER_SHAPE1 = [20, 20]
FILTER_SHAPE2 = [20, 1]
POOLING_WINDOW = 4
POOLING_STRIDE = 2

batch_size = 128
no_epochs = 100
lr = 0.01

tf.logging.set_verbosity(tf.logging.ERROR)
seed = 10
tf.set_random_seed(seed)
```

The parameters for the CNN and RNNs are first initialised. As you can see, the filter shapes for each convolution layer is defined here. Also, pooling window and strides are defined. The embedding size is also defined. Embedding size determines the size of the vector used to encode each word in the paragraph. Hidden size is used in the RNN and it determines the number of GRU cells used in the hidden layer.

```
def data_read_words():
    x_train, y_train, x_test, y_test = [], [], [], []

    with open('train_medium.csv', encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            x_train.append(row[2])
            y_train.append(int(row[0]))

    with open("test_medium.csv", encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            x_test.append(row[2])
            y_test.append(int(row[0]))

    x_train = pandas.Series(x_train)
    y_train = pandas.Series(y_train)
    x_test = pandas.Series(x_test)
    y_test = pandas.Series(y_test)
    y_train = y_train.values
    y_test = y_test.values

    vocab_processor = tf.contrib.learn.preprocessing.VocabularyProcessor(
      MAX_DOCUMENT_LENGTH)

    x_transform_train = vocab_processor.fit_transform(x_train)
    x_transform_test = vocab_processor.transform(x_test)
    x_train = np.array(list(x_transform_train))
    x_test = np.array(list(x_transform_test))

    no_words = len(vocab_processor.vocabulary_)
    print('Total words: %d' % no_words)

    return x_train, y_train, x_test, y_test, no_words
```

This section shows how the dataset is transformed into a suitable format for input into the neural network. After reading the csv files and converting the data into a pandas dataframe, we make use of TensorFlow's Vocabulary Processor which assigns a id to each word and then redefines the dataset in terms of the ids, instead of words.

```python
def word_cnn_model(x, p):
    word_vectors = tf.contrib.layers.embed_sequence(
      x, vocab_size=n_words, embed_dim=EMBEDDING_SIZE)
    input_layer = tf.reshape(
      word_vectors, [-1, MAX_DOCUMENT_LENGTH, EMBEDDING_SIZE, 1])
    with tf.variable_scope('CNN_Layer1'):
        conv1 = tf.layers.conv2d(
            input_layer,
            filters=N_FILTERS,
            kernel_size=FILTER_SHAPE1,
            padding='VALID',
            activation=tf.nn.relu)
        pool1 = tf.layers.max_pooling2d(
            conv1,
            pool_size=POOLING_WINDOW,
            strides=POOLING_STRIDE,
            padding='SAME')
        conv2 = tf.layers.conv2d(
            pool1,
            filters=N_FILTERS,
            kernel_size=FILTER_SHAPE2,
            padding='VALID',
            activation=tf.nn.relu)
        pool2 = tf.layers.max_pooling2d(
            conv2,
            pool_size=POOLING_WINDOW,
            strides=POOLING_STRIDE,
            padding='SAME')

    pool2 = tf.squeeze(tf.reduce_max(pool2, 1), squeeze_dims=[1])

    logits = tf.layers.dense(pool2, MAX_LABEL, activation=None)
    return logits, word_vectors
```

Next, the CNN architecture is set up using the earlier defined parameters. The inputs are first passed through and embedding of size 20 followed by two convolution and pooling layers. It is finally passed to a dense layer which generates the logits to be passed on.

```python
def rnn_model(x):

    word_vectors = tf.contrib.layers.embed_sequence(
      x, vocab_size=n_words, embed_dim=EMBEDDING_SIZE)

    word_list = tf.unstack(word_vectors, axis=1)

    cell = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    _, encoding = tf.nn.static_rnn(cell, word_list, dtype=tf.float32)

    logits = tf.layers.dense(encoding, MAX_LABEL, activation=None)

        # TODO: make sure that dropout is not applied on testing
    return logits, word_list
```

For the RNN architecture, the inputs are similarly passed to an embedding layer of size 20 followed by a recurrent layer made up of 20 GRU cells.

```python
def main():
    global n_words
    #x_train is a list of ids corresponding to each word in the paragraph.
    x_train, y_train, x_test, y_test, n_words = data_read_words()
    # print('y_train:', y_train.shape)
    # Create the model


    loss = {}
    test_accs = {}
    #word_vectors is the vector representation of each id
    p = 0

    x = tf.placeholder(tf.int64, [None, MAX_DOCUMENT_LENGTH])
    y_ = tf.placeholder(tf.int64)

    logits, word_list = rnn_model(x)
    entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=tf.one_hot(y_, MAX_LABEL), logits=logits))
    train_op = tf.train.AdamOptimizer(lr).minimize(entropy)

    correct_prediction = tf.cast(tf.equal(tf.argmax(logits, 1), y_ ), tf.float32)
    accuracy = tf.reduce_mean(correct_prediction)

    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
```

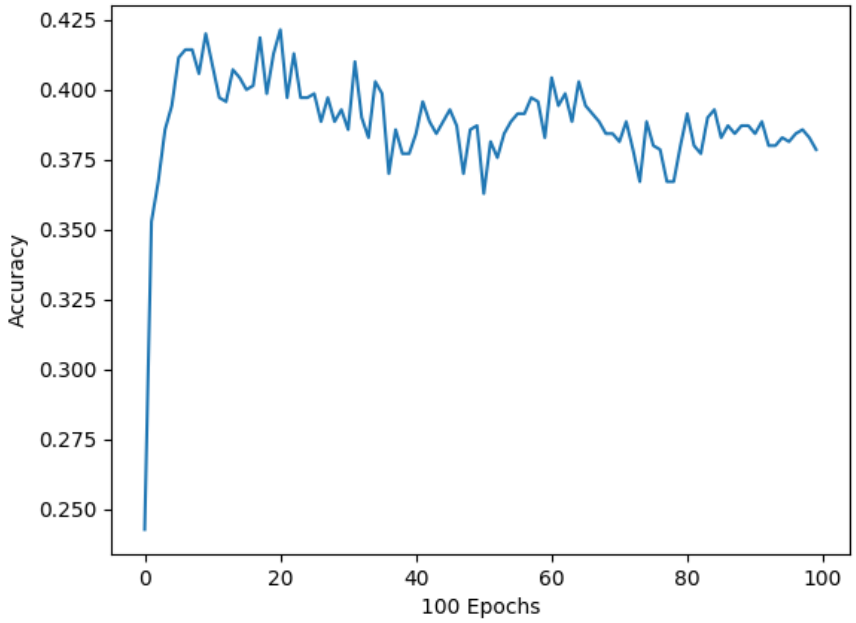
```python
# training
  loss[p] = []
  test_accs[p] = []
  idx = np.arange(x_train.shape[0])
  NUM_INPUT = x_train.shape[0]
  repetition_in_one_epoch = int(NUM_INPUT / batch_size)
  for e in range(no_epochs):
      np.random.shuffle(idx)
      x_train, y_train = x_train[idx], y_train[idx]
      start = -1 * batch_size
      end = 0
      for k in range(repetition_in_one_epoch):
          start += batch_size
          end += batch_size
          if end > NUM_INPUT:
              end = NUM_INPUT
          word_list_, _, loss_  = sess.run([word_list, train_op, entropy], {x: x_train[start:end], y_: y_train[start:end]})
      loss[p].append(loss_)
      acc = sess.run([accuracy], {x: x_test, y_: y_test })
      test_accs[p].append(acc[0])
      if e%10 == 0:
          print('epoch: %d, entropy: %g'%(e, loss[p][e]), 'accuracy:', test_accs[p][e])
```

Finally, the data is trained using mini batch gradient descent of batch size 128.

**Results**

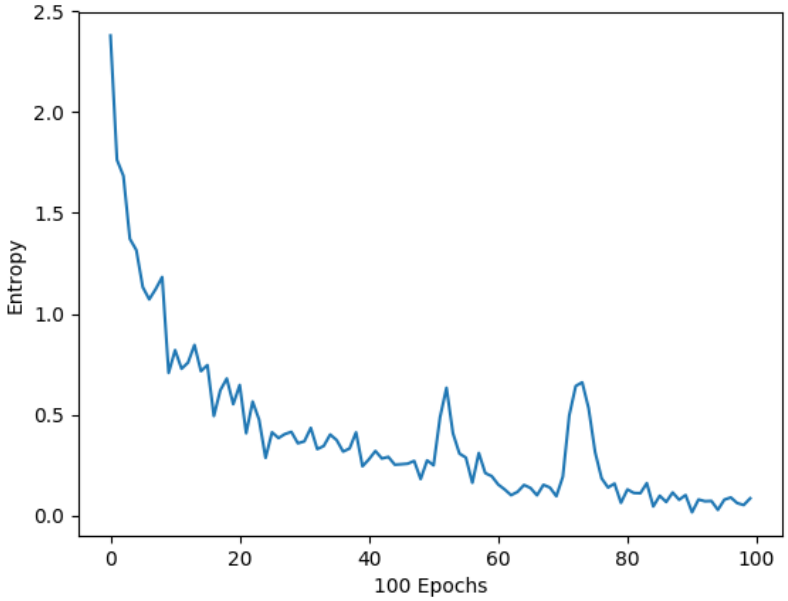
**Question 1**

Accuracy of Character CNN classifier against Epochs




Best accuracy: 42.3%

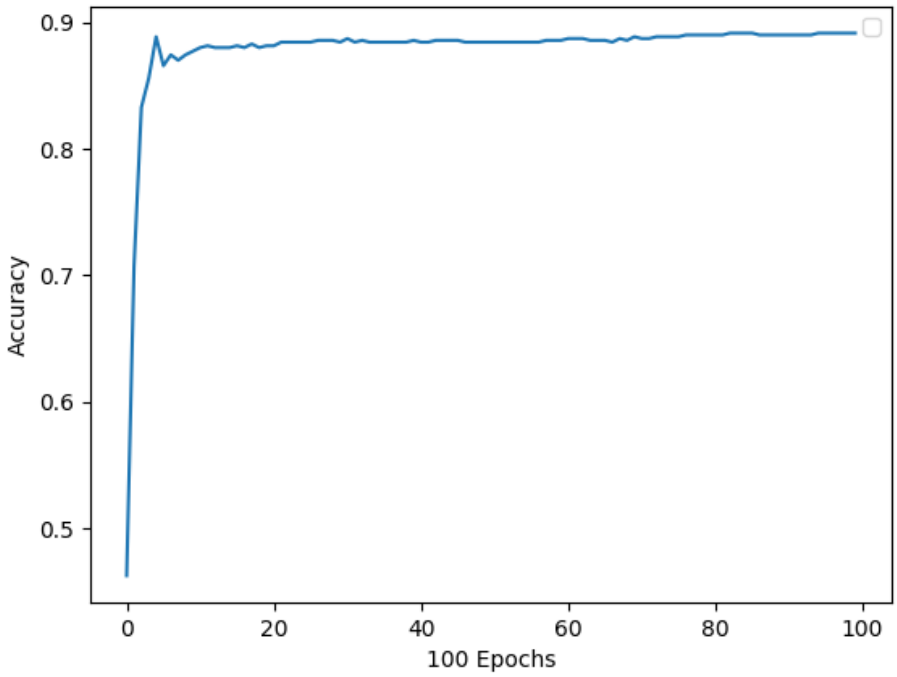
Entropy of Character CNN Classifier Against Epochs




Duration: 24.0s

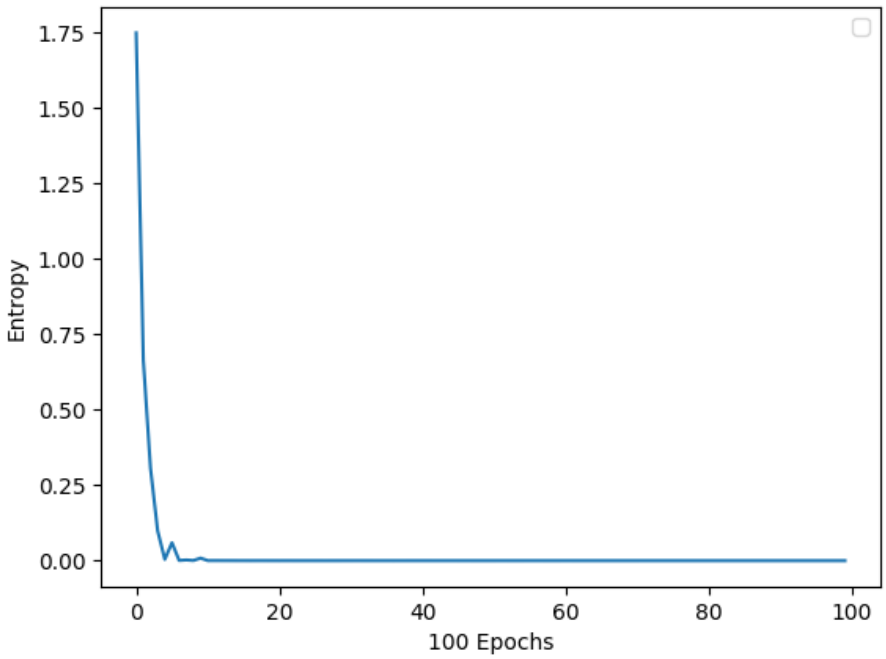
**Question 2**

Accuracy of Word CNN classifier against Epochs




Best accuracy: 89.1%

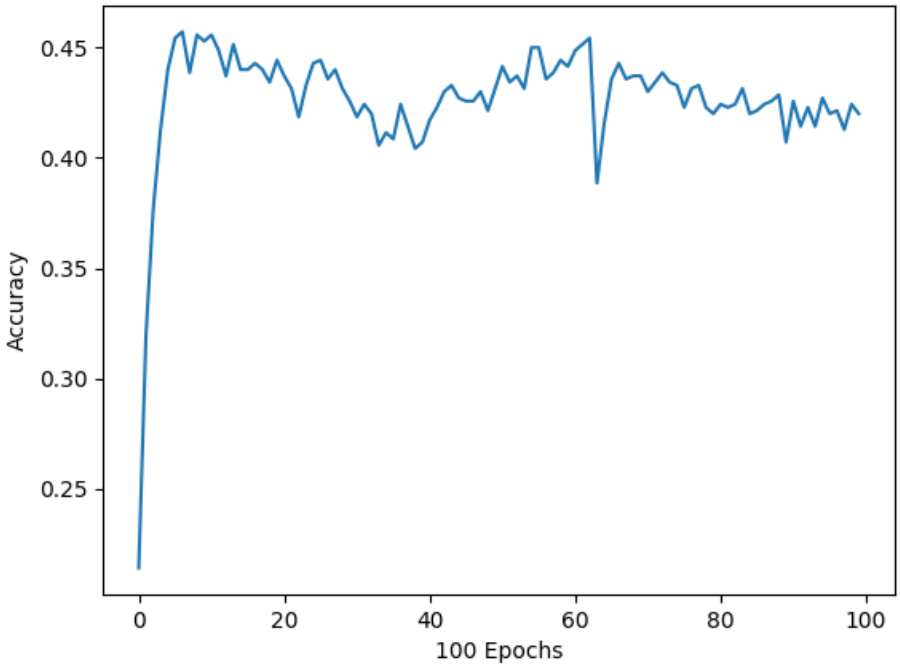
Entropy of Word CNN Classifier Against Epochs




Duration: 13.4s

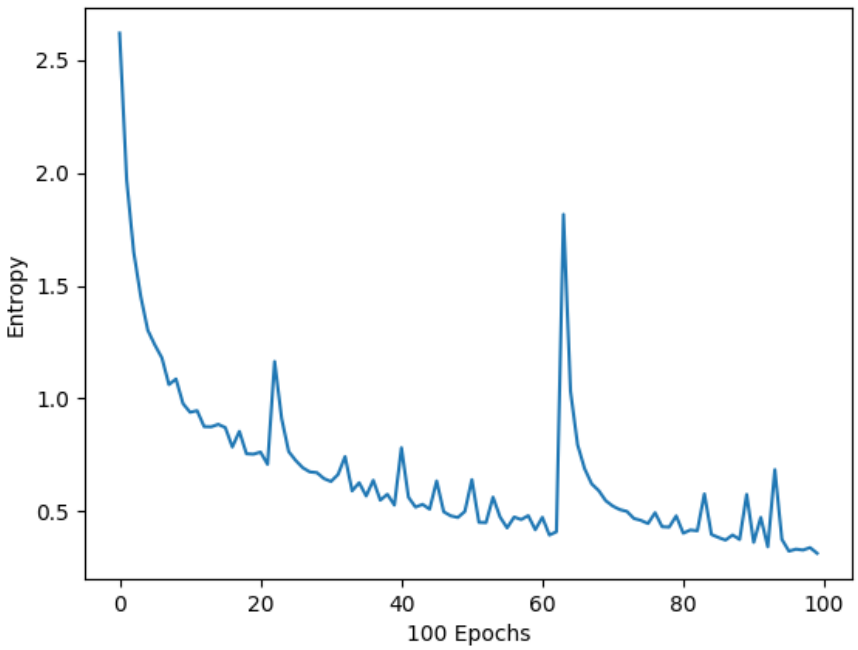
**Question 3**

Accuracy of Character RNN classifier against Epochs




Best accuracy: 45.8%

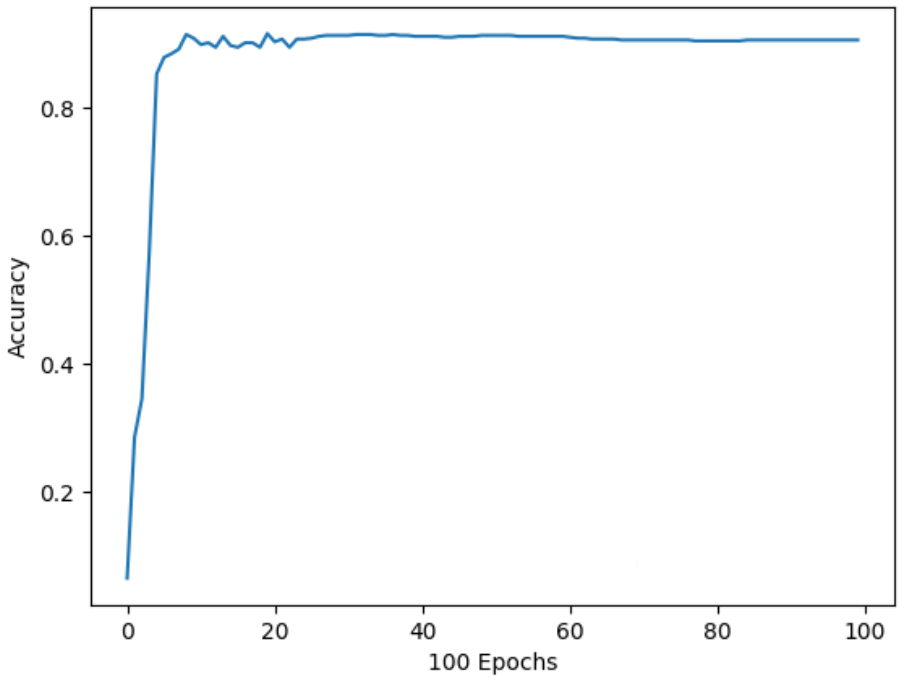
Entropy of Character RNN Classifier Against Epochs




Duration: 13.4s

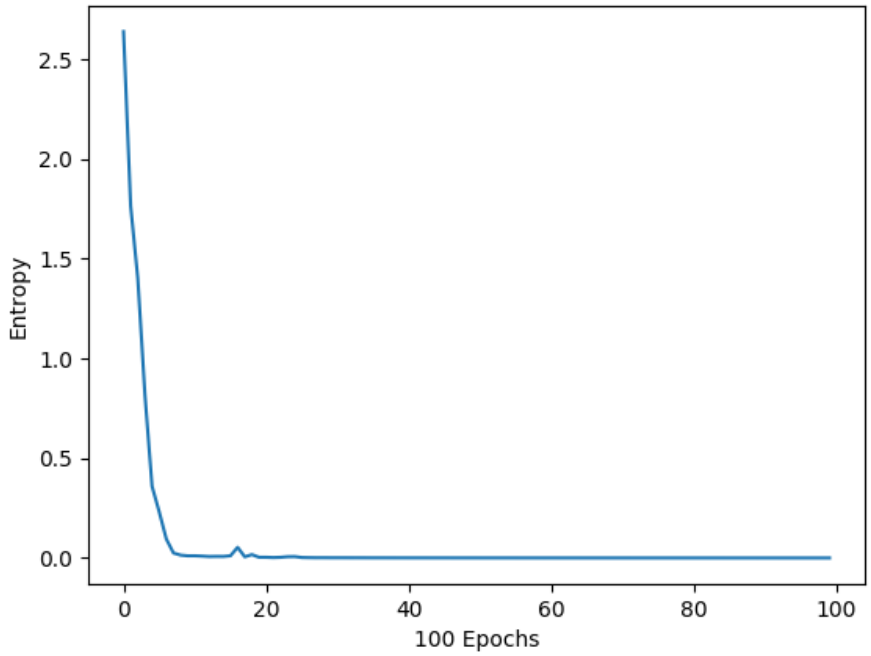
**Question 4**

Accuracy of Word RNN classifier against Epochs




Best accuracy: 90.6%

Entropy of Word RNN Classifier Against Epochs




Duration: 197.2s

**Question 5**

Comparing across (1) to (4)

| Model | Character CNN | Word CNN | Character RNN | Word RNN |
|---|---|---|---|---|
| **Accuracy (%)** | 42.3 | 89.0 | 45.8 | 90.6 |
| **Duration (s)** | 24.0 | 13.4 | 321.1 | 197.2 |

Word RNN classifier had the best accuracy at 90.6, while the word CNN had the fastest run time at 13.4 seconds.

In general, CNN ran much faster than RNN classifiers while RNN classifiers performed much better in terms of accuracy.

Overall, word CNN had a notable general performance since it ran fastest but only performed slightly worse than word RNN with a 1.6% lower accuracy.

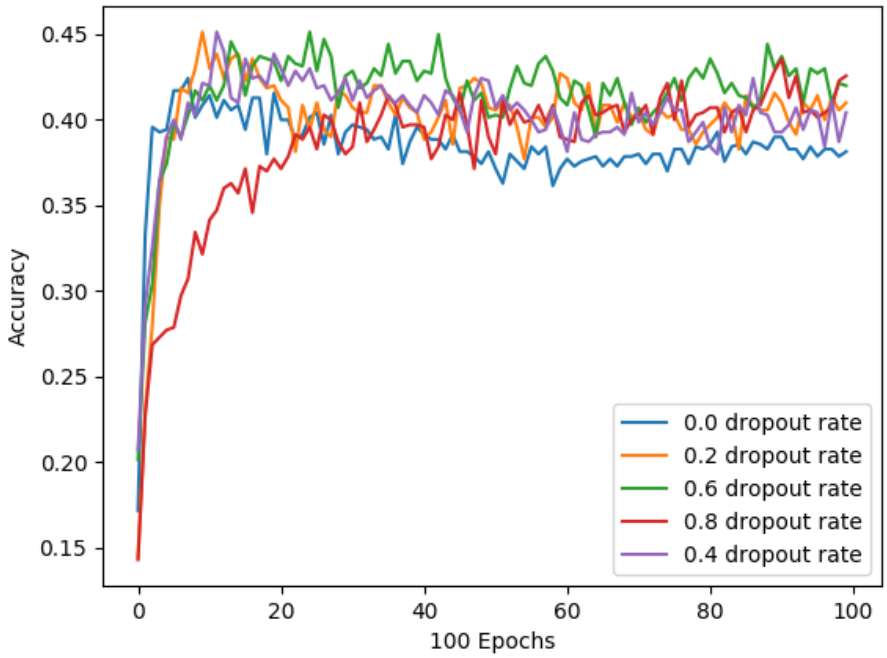
Experimenting with dropout

*In the diagrams to follow, dropout rate is defined as the probability that neurons are dropped out from each layer. E.g. 0.1 dropout means 10% of the neurons will be dropped out*

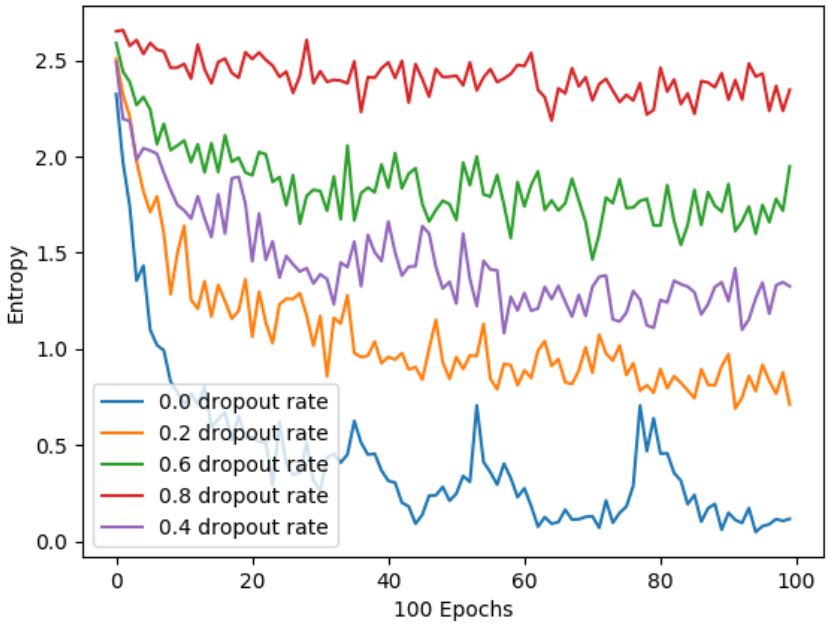
For CNN classifiers, dropout was applied at the final dense layer, while for the RNN classifier, dropout was applied at both recurrent layer and dense layer.

i. Character CNN classifier

Accuracy against Epochs



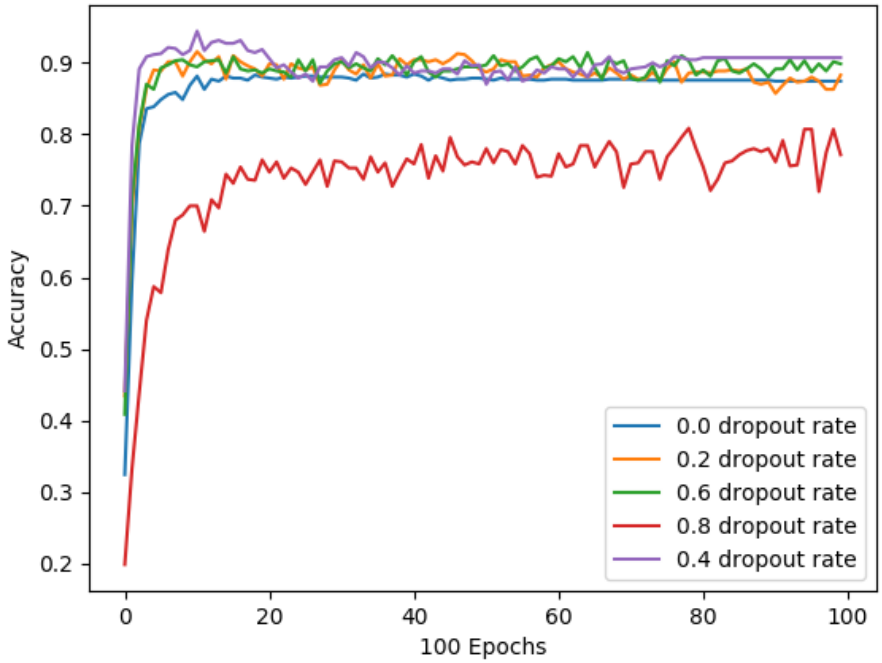

Entropy against Epochs



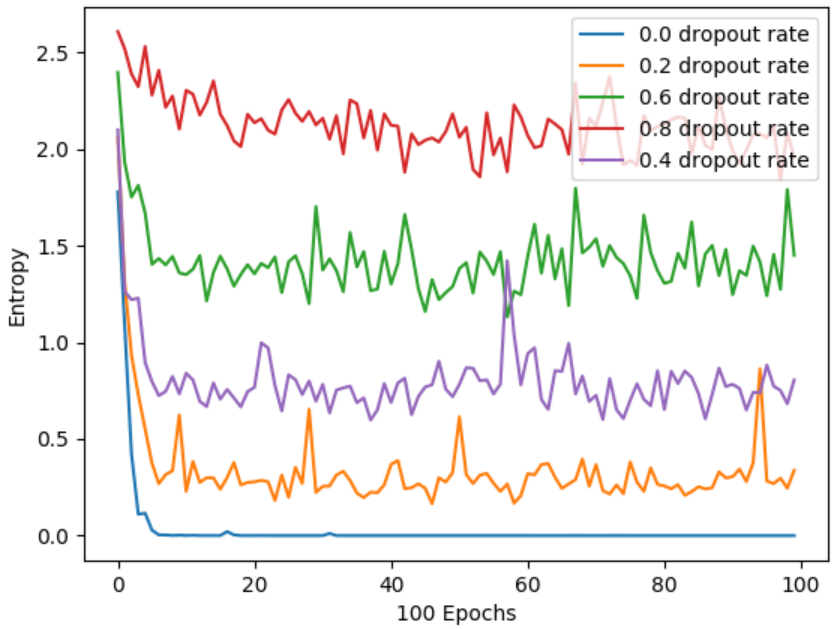

The best accuracy was achieved by using a dropout rate of 0.6 at around 40 epochs with 45.1% accuracy. Overall, using dropouts improved the performance of the character CNN classifier compared to not using any dropout.

ii. Word CNN classifier

Accuracy against Epochs



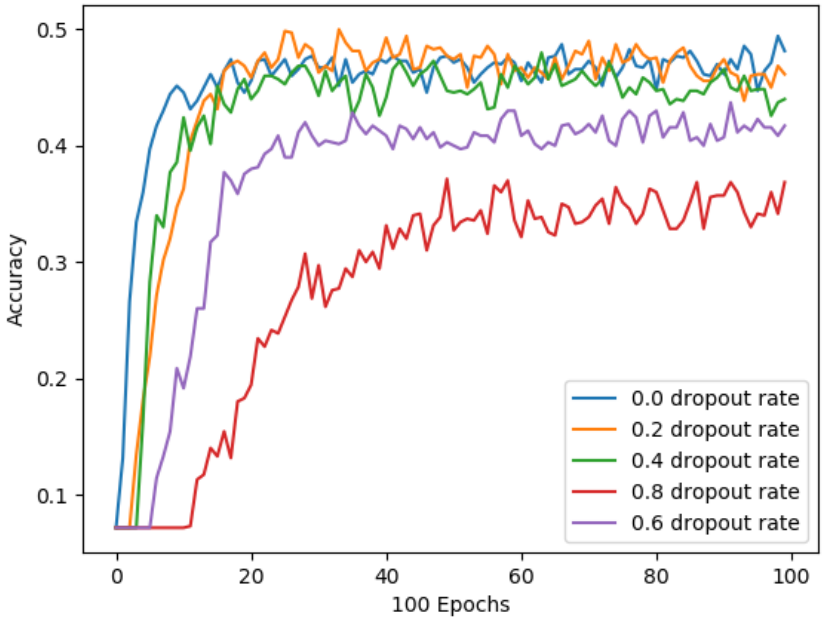

Entropy against Epochs



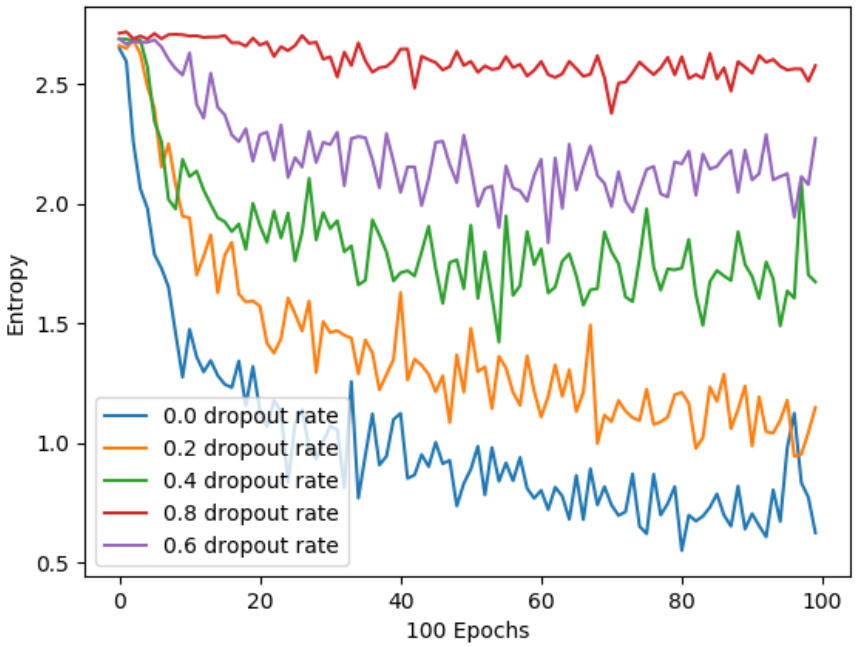

Using dropout at a rate of 0.4 gave the best accuracy at 91%. All the dropout rates performed better than when no dropout was used, other than a rate of 0.8 which performed much poorer, achieving 76.2% at best.

iii. Character RNN classifier

Accuracy against Epochs



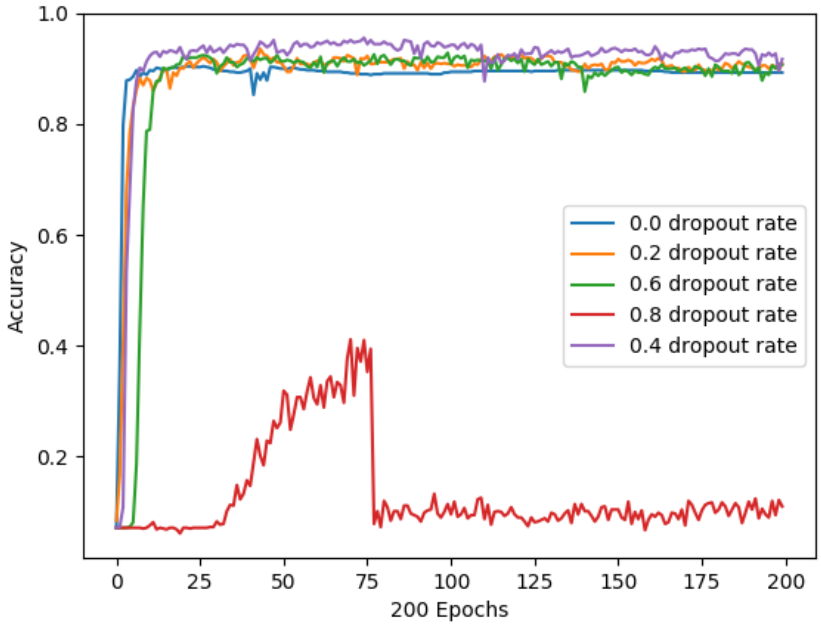

Entropy against Epochs



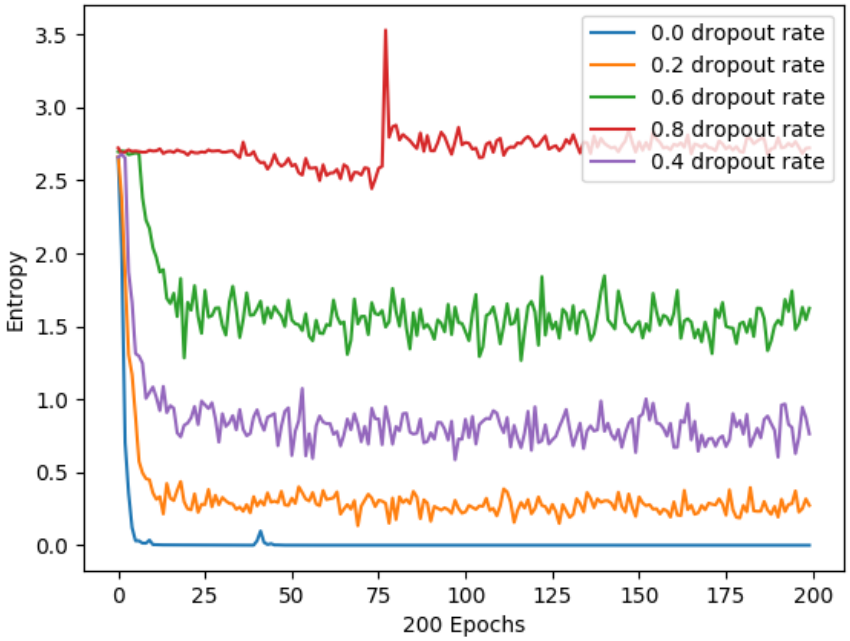

Although using no dropouts achieved the highest accuracy at 49.1%, similar levels were also achieved with 0.2 dropout rate at epoch 33.

iv. Word RNN classifier

Accuracy against Epochs




Entropy against Epochs




Using dropout at a rate of 0.4 gave the best accuracy at 93.3%. Although there was similar performance for all dropout rates. All the dropout rates performed better than with using no dropout, other than 0.8 which performed very badly

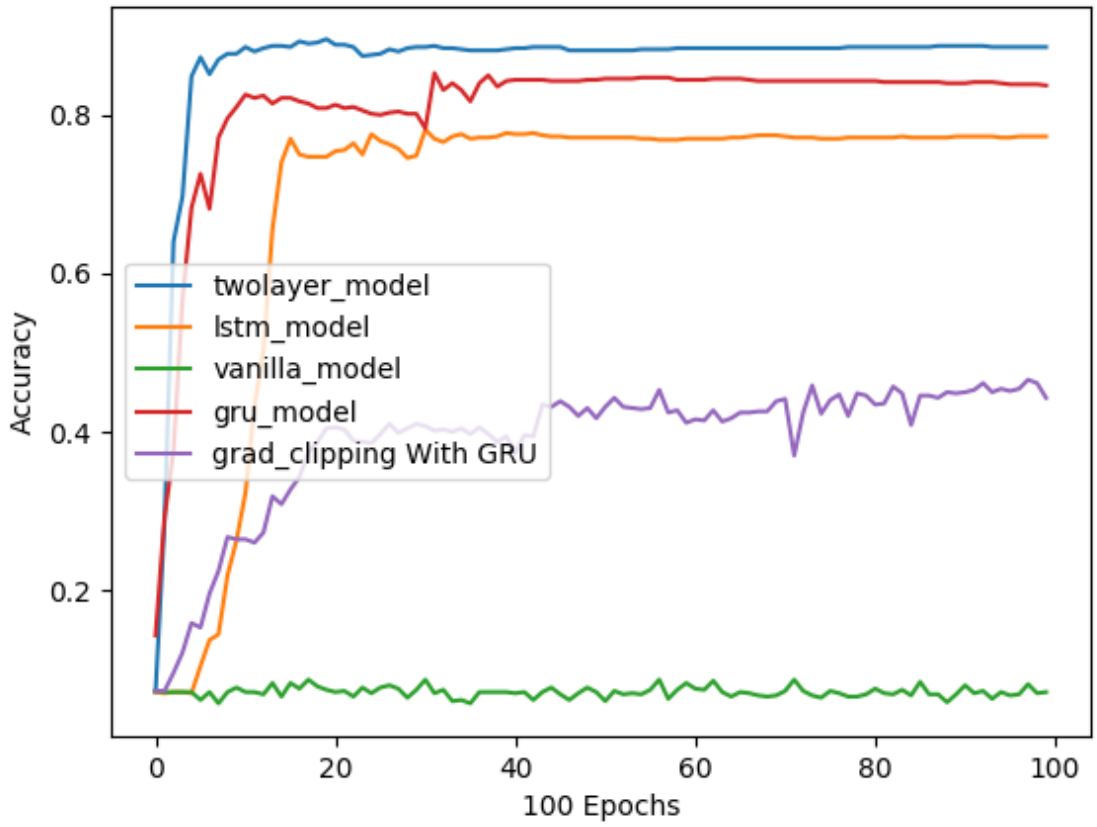
**Question 6**

*For this question I have organised the plots according to the technique used to process the data, i.e., using char inputs and word inputs. In each plot, I have included every permutation of techniques mentioned in 6a, 6b and 6c, namely:*

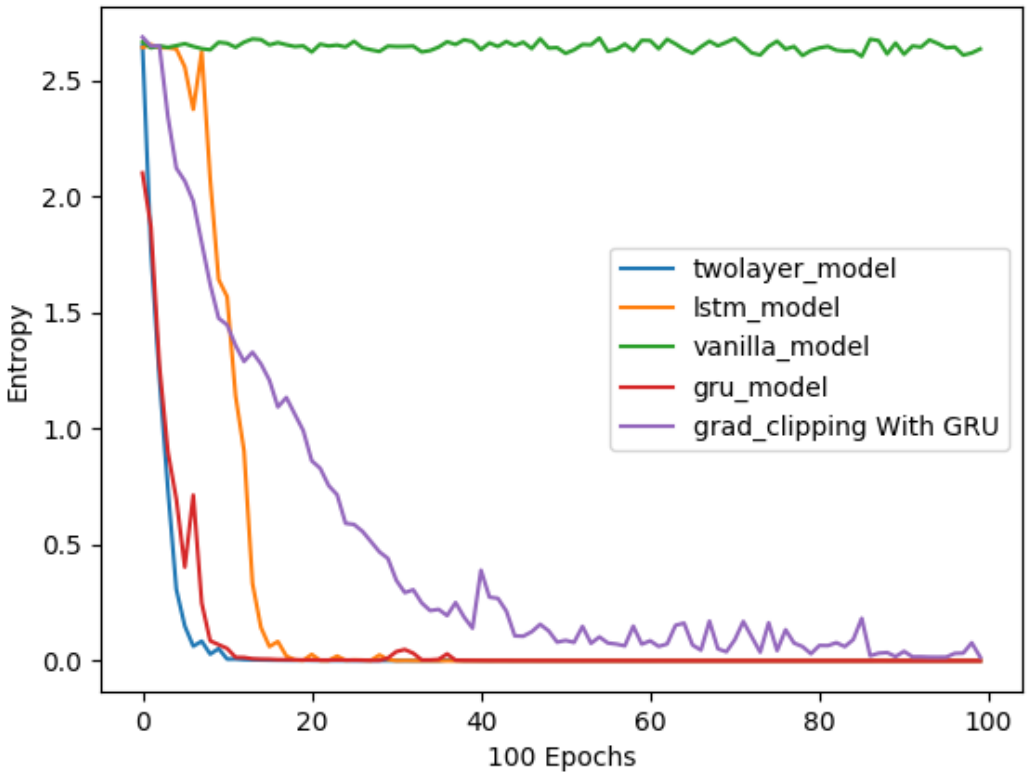
- *Two layer RNN using GRU cells, labelled twolayer_model*
- *One layer RNN using LSTM cells, labelled lstm_model*
- *One layer RNN using Vanilla RNN cells, labelled vanilla_model*
- *One layer RNN using GRU cells, labelled gru_model*
- *One layer RNN using GRU cells using gradient clipping, labelled gru_model*

**Using word inputs to the network:**

Accuracy against Epochs



Entropy against Epochs




Overall, the two layer model performed the best and it achieved an accuracy of 88.7%. The second best performer was the one layer GRU model at 84% accuracy. This was followed by the one layer LSTM model at 77% accuracy. Next, gradient clipping a one layer GRU model gave an accuracy of 45%. The one layer vanilla RNN cell model failed at classifying this dataset with an accuracy of 8%, which is hardly better than the expected accuracy of a random model. This tells us that vanilla RNN cells are not able to 'remember' words that came much earlier well enough to predict the category.

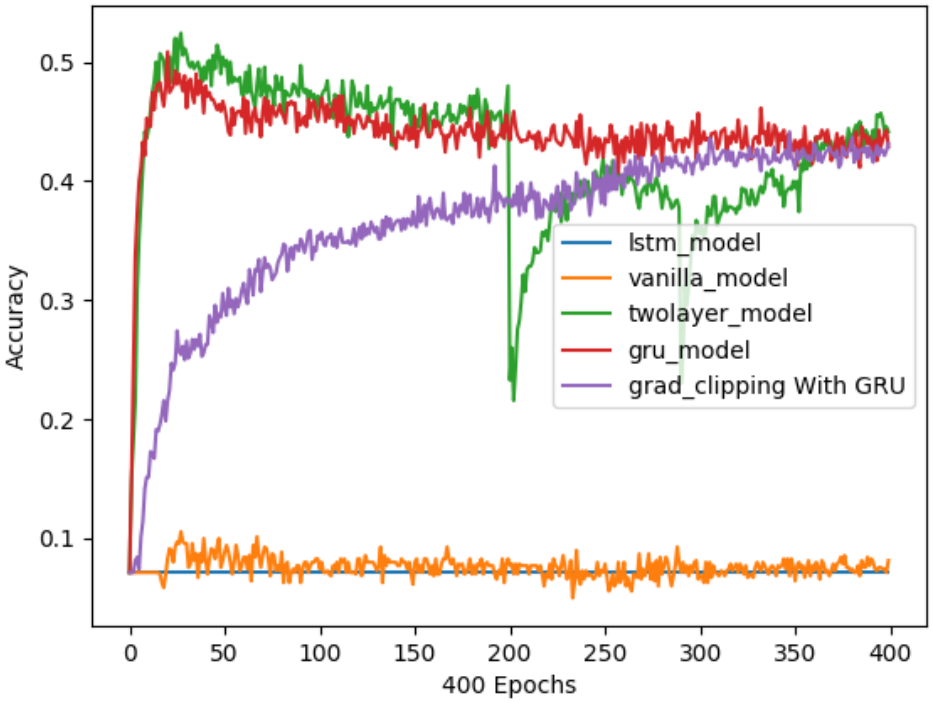
The top performance of the two layer GRU model tells us that the additional layer to the one layer GRU provided the needed complexity to the model to achieve better performance.

We also learn that GRU cells are better suited to this problem compared to LSTM cells.

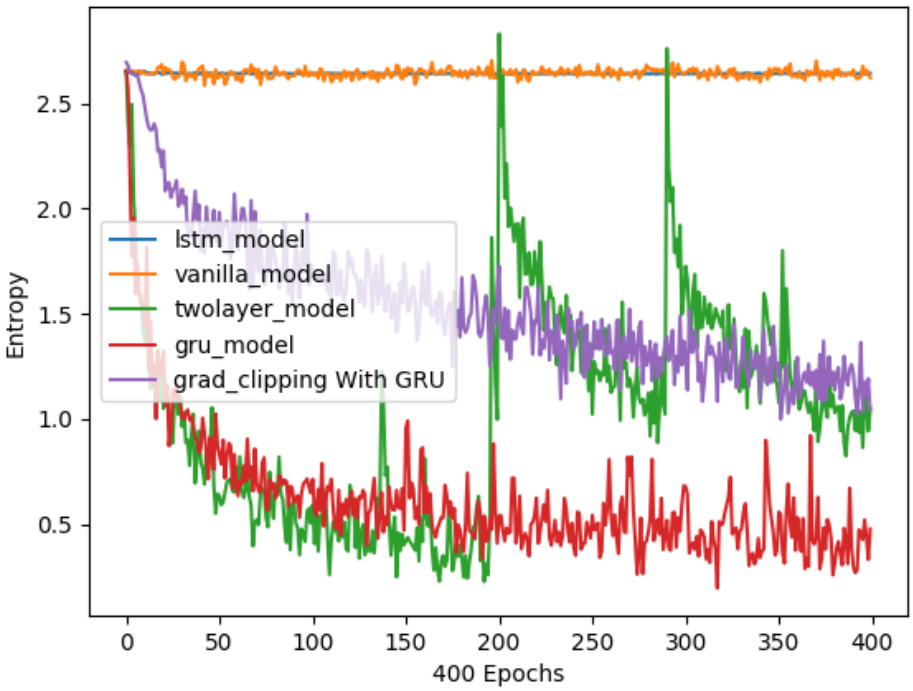
Finally, gradient clipping did not improve the accuracy at all and instead it caused a much poorer performance at nearly half the accuracy compared to using normal gradient descent.

**Using char inputs to the network:**

Accuracy against epochs




Entropy against epochs



This experiment was ran to 400 epochs instead of the usual 100 epochs to allow the gradient clipping line to converge. We can see that past epoch 50, the single layer and two layer models began to overfit and accuracy decreased.

Overall, the two layer model performed the best and the best accuracy it achieved was 52.1%(epoch 35), this is despite it decreasing drastically in accuracy at epochs 200 and 280. A possible explanation for this behaviour is the Gradient Explosion Problem. Having been trained past 100 epochs, gradients grow large, causing it to overshoot and pass the minima during backpropagation. This resulted in the increases in entropy and decrease in accuracy observed.

The second best performer was the one layer GRU model at 50.5% accuracy(epoch 25). Next, gradient clipping a one layer GRU model gave an accuracy of 44.8%. The one layer vanilla RNN cell model and LSTM failed at classifying this dataset with very low accuracies. The vanilla RNN gave its best accuracy of 10.2% (epoch 25), while the LSTM gave a constant accuracy of around 7.1%. This tells us that vanilla RNN cells are not able to 'remember' characters that came much earlier well enough to predict the category. The LSTM was not able to reduce entropy at all, indicating that this cell is not suited for this task.

The top performance of the two layer GRU model tells us that the additional layer to the one layer GRU provided the needed complexity to the model to achieve better performance.

Finally, at epoch 400, gradient clipping had a similar accuracy to the single layer GRU model of about 43%. Using gradient clipping slowed down the rate of convergence and it appears that gradient clipping still has potential to increase in accuracy. However due to time constraints, the model was not run for even longer epochs.

## Conclusion

In conclusion, we learnt that using dropouts at around rates of 0.4 to 0.6 can improve the accuracy of neural networks in classify the Wikipedia entries. We also learnt that RNNs perform better than CNNs in this task. Additionally, we discovered at increasing the number of hidden RNN layers can improve accuracy.