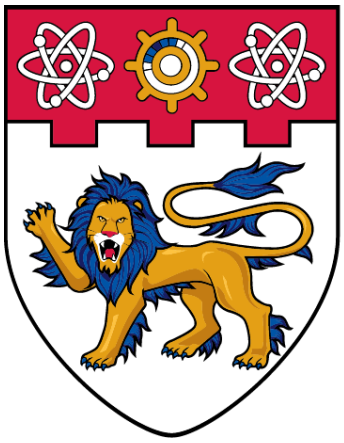


NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CZ4042 Neural Networks and Deep Learning

Individual Assignment 2

Zhang Yuhan

U1823060F

Table of Contents

Part A: Object Recognition

Introduction

This problem aims at creating an optimal convolutional neural network for recognizing objects from images. The CIFAR-10 dataset will be used for training and testing the models. The file **data_batch_1** contains 10,000 samples of data which will be used for training, while the file **test_batch_trim** contains the samples of data that will be used during testing. The convolutional neural network used in this part will contain the following layers:

- An Input layer of 32x32x3 dimensions
- A convolution layer with 50 channels, window size 9x9, VALID padding, and ReLU activation
- A max pooling layer with a pooling window of size 2x2, stride = 2, and VALID padding
- A convolution layer with 60 channels, window size 5x5, VALID padding, and ReLU activation
- A max pooling layer with a pooling window of size 2x2, stride = 2, and VALID padding
- A fully-connected layer of size 300 with no activation
- A fully-connected layer of size 10 with Softmax activation

All figures and codes provided in this report is present in the corresponding jupyter notebook files for each question.

Methods

It had been observed that there were problems with the starter code given in the project that caused the image data to be reshaped abnormally. This was fixed by using the reshape() and transpose() functions from the tensorflow library as shown below:

```
data = tf.reshape(data, [-1,3, 32, 32])
data = tf.transpose(data,(0,2,3,1))
```

Figure 1: Data pre-processing

In this project, convolutional neural networks (CNN) will be used to train the data after successful pre-processing. CNN is a powerful Deep Learning algorithm that has architectures similar to that of the connectivity patterns of neurons in the human brain. It consists of and an input and an output layer as well as one or more hidden layers in between. It is commonly

applied in the visual imagery and is able to efficiently capture the spatial and temporal dependencies within each dataset. An overview of CNN architecture is shown below:

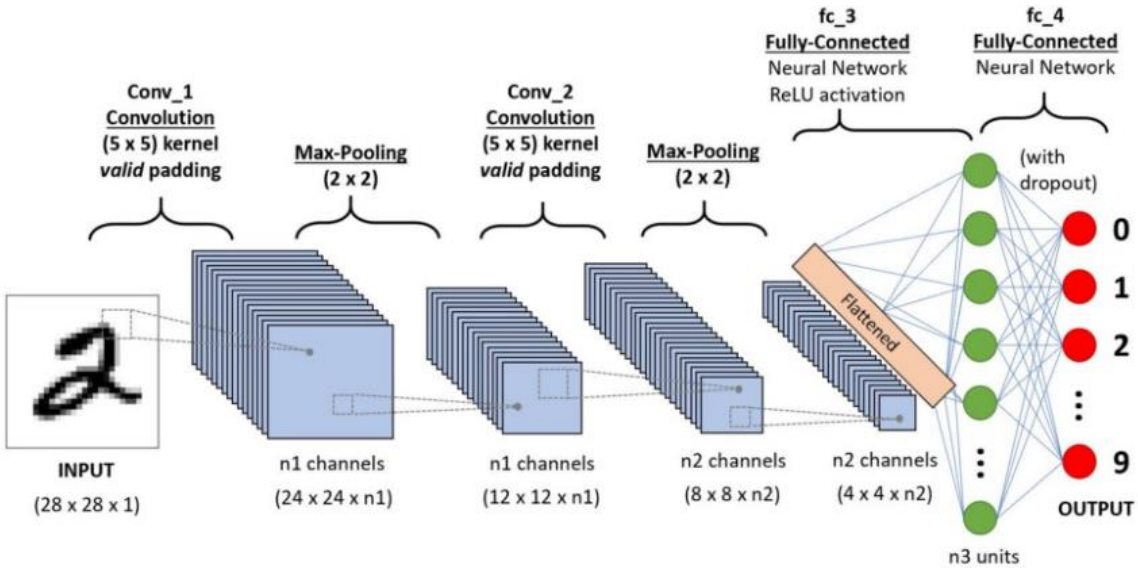


Figure 2: CNN architecture src: https://towardsdatascience.com/

At each layer, the results of applying the filter on the input image can be captured by feature maps. By analysing these maps, we can gain understanding of what features the CNN detects.

To improve the results of CNNs, a variety of optimizers may be used. Optimizers are algorithms that change the attributes of the CNN in order to reduce losses and increase accuracy. In later parts of this report, I will be analysing the effects of the following optimizers:

- Stochastic Gradient Descent (SGD)
- SGD with momentum
- RMSProp
- Adam

In order to prevent possible overfitting, dropouts may be introduced in the CNNs. Dropout is a regularization technique that randomly selects neurons to be 'dropped out' during training. The rate at which this happens will depend on a predetermined value that is set before training. A visual representation of this is shown below:

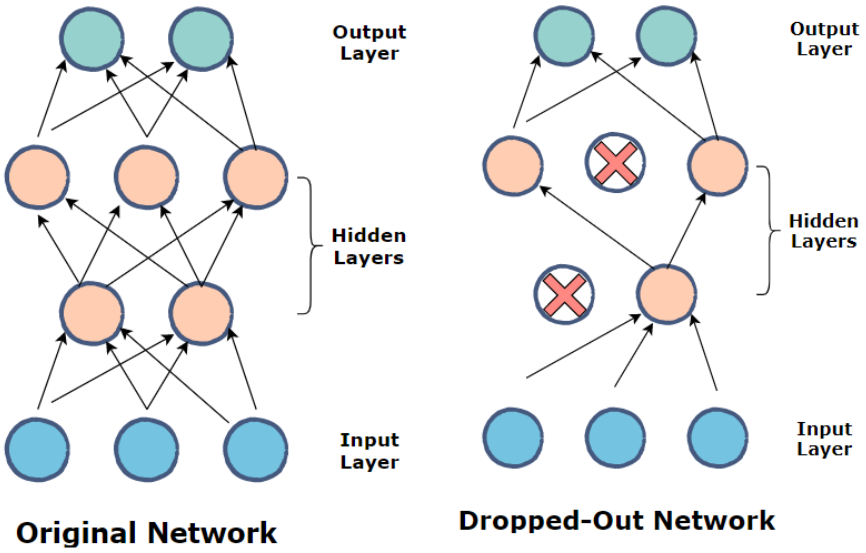


Figure 3: Dropouts in CNN src: https://www.educative.io

After a model is finished training, the history and model will be saved locally in the folders ‘./histories’ and ‘./models’ respectively. This allows us to reload the model information for future processes. The function used to save the model and model history is shown below:

```
def save_model(num_ch_c1,num_ch_c2,optimizer_,history,use_dropout):
    # Create folder to store models and results
    if not os.path.exists('./models'):
        os.mkdir('./models')
    if not os.path.exists('./results'):
        os.mkdir('./results')
    if not os.path.exists('./histories'):
        os.mkdir('./histories')

    # Save model
    if use_dropout:
        model.save(f'./models/QuestionA2_{num_ch_c1}_{num_ch_c2}_{optimizer_}_dropout')
        with open(f'histories/{num_ch_c1}_{num_ch_c2}_{optimizer_}_dropout', 'wb') as file_pi:
            pickle.dump(history.history, file_pi)
    else:
        model.save(f'./models/QuestionA2_{num_ch_c1}_{num_ch_c2}_{optimizer_}_no_dropout')
        with open(f'histories/{num_ch_c1}_{num_ch_c2}_{optimizer_}_nodropout', 'wb') as file_pi:
            pickle.dump(history.history, file_pi)
```

Figure 4: Code for saving model and model history.

Experiments and Results

Question 1:

Train the network using mini-batch gradient descent learning for 1000 epochs. Set the batch size to 128, and learning rate = 0.001.

a. Plot the (1) training cost, (2) test cost, (3) training accuracy, and (4) test accuracy against learning epochs. One plot for the costs and one plot for the accuracies.

b. For the first two test images, plot the feature maps at both convolution layers (C1 and C2) and pooling layers (S1 and S2) along with the test images. (In total one image and four feature maps)

   Note: A feature map with N channels can be viewed as N grayscale images. Do make sure that the pixel values are in the correct range when you plot them.

The hyperparameters used for this question is as follows:

```
num_ch_c1 = 50
num_ch_c2 = 60

epochs = 1000
batch_size = 128
learning_rate = 0.001
optimizer_ = 'SGD'
use_dropout = False
```

Part a)

- Plot the (1) training cost, (2) test cost, (3) training accuracy, and (4) test accuracy against learning epochs. One plot for the costs and one plot for the accuracies.

The data is loaded by the **load_data()** function as shown in figure _____. For this question, categorical cross entropy is used as the loss function and accuracy is used as the metric for measuring model performance. The model that is generated and trained using the hyperparameters above is as follows:

```
def make_model(num_ch_c1, num_ch_c2, use_dropout):

    model = tf.keras.Sequential()
    model.add(layers.Input(shape=(32,32,3)))
    model.add(layers.Conv2D(num_ch_c1, 9, padding='valid', activation='relu', input_shape=(None, None, 3)))
    model.add(layers.MaxPool2D(pool_size=(2, 2), strides=2, padding='valid'))
    model.add(layers.Conv2D(num_ch_c2, 5, padding='valid', activation='relu'))
    model.add(layers.MaxPool2D(pool_size=(2, 2), strides=2, padding='valid'))
    model.add(layers.Flatten())
    model.add(layers.Dense(300, activation=None))
    model.add(layers.Dense(10, use_bias=True, activation='softmax', input_shape=(300,)))
    return model
```

The accuracy and costs against the number of epochs for the training and testing data can be observed from the graphs below:

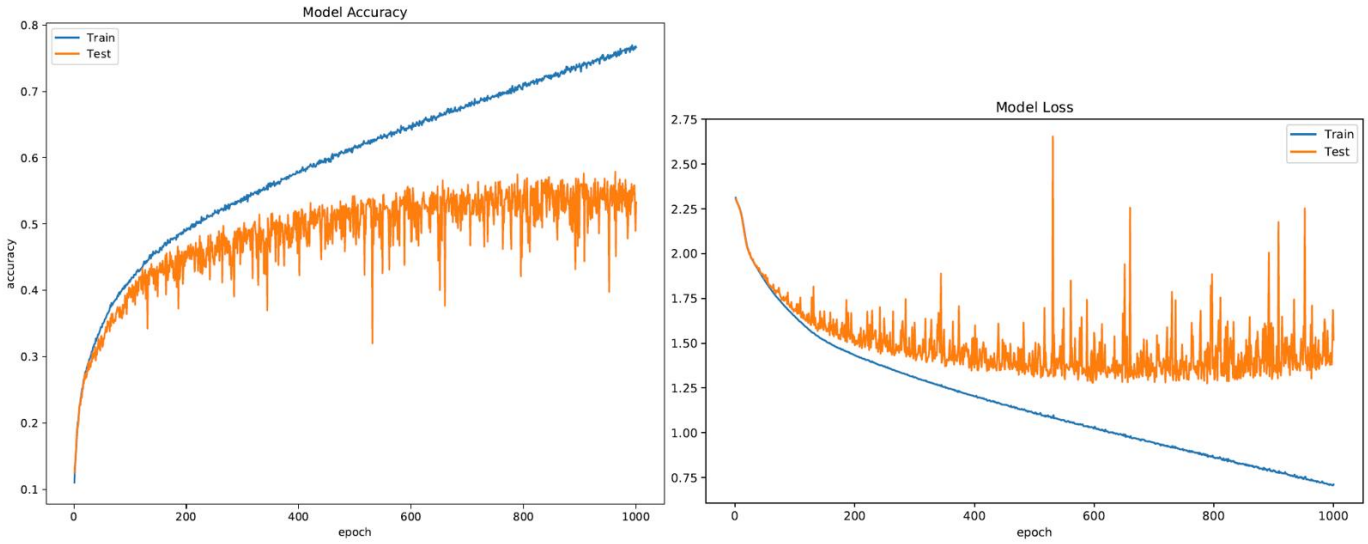


Figure 5: Question 1A accuracy and cost for training and testing data

Part b)

a. For the first two test images, plot the feature maps at both convolution layers (C1 and C2) and pooling layers (S1 and S2) along with the test images. (In total one image and four feature maps)

This means that for the first convolution and pooling layers, there will be 50 feature maps, and for the second convolution and pooling layers, there will be 60 feature maps. These maps are represented in the following figures.

The original test image 1 is shown below:

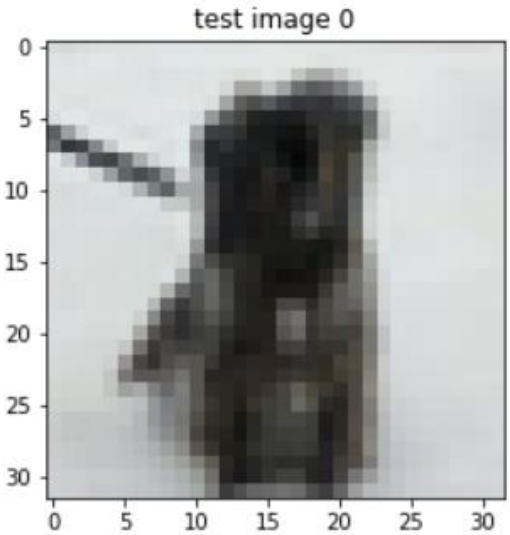


Figure 6: test image 1

The feature maps at convolution and pooling layers for test image 1 are as follows:

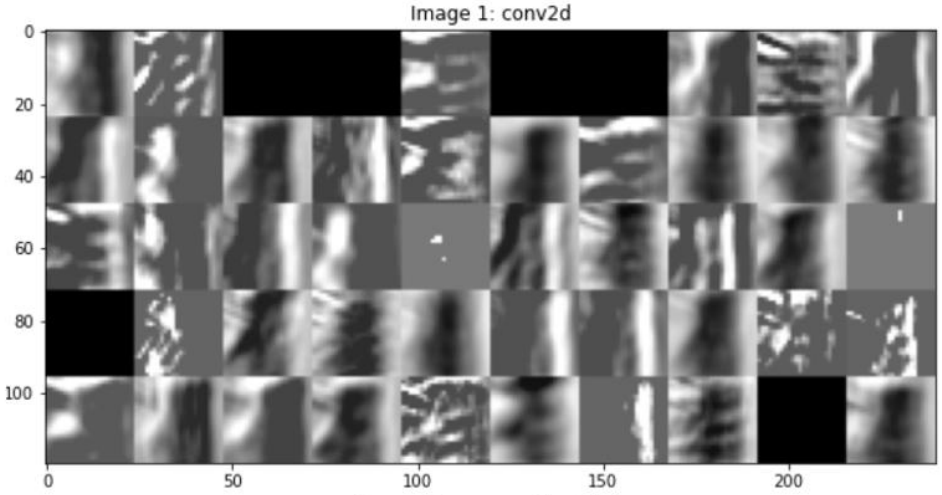


Image 1: conv2d
0
20
40
60
80
100
0
50
100
150
200

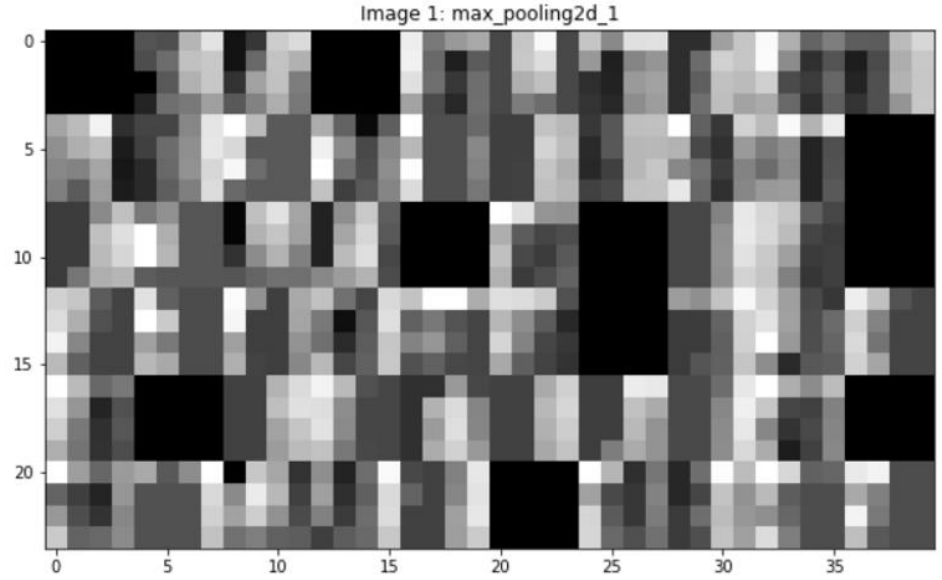


Image 1: max_pooling2d_1
0
5
10
15
20
0
5
10
15
20
25
30
35

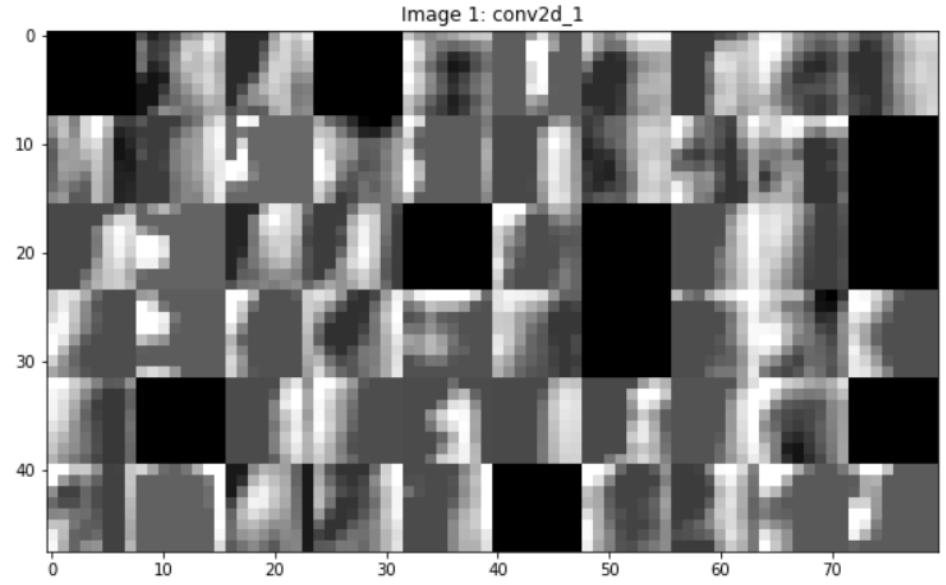


Image 1: conv2d_1
0
10
20
30
40
0
10
20
30
40
50
60
70

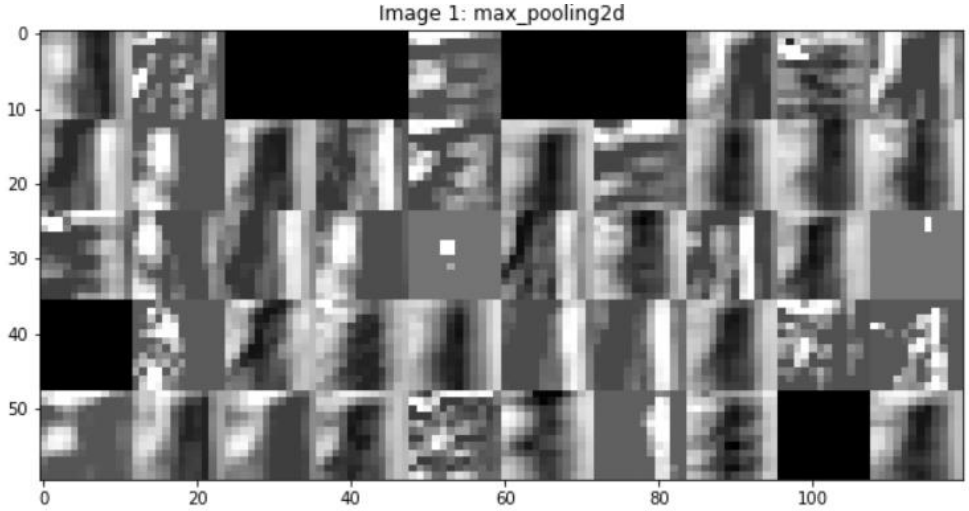


Figure 7: Feature maps of test image 1

The original test image 2 is shown below:

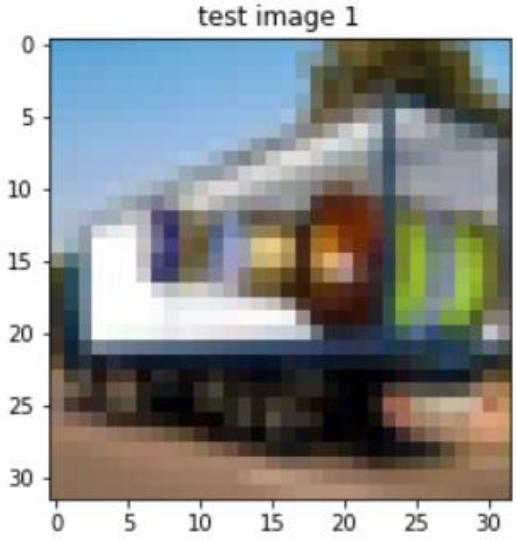
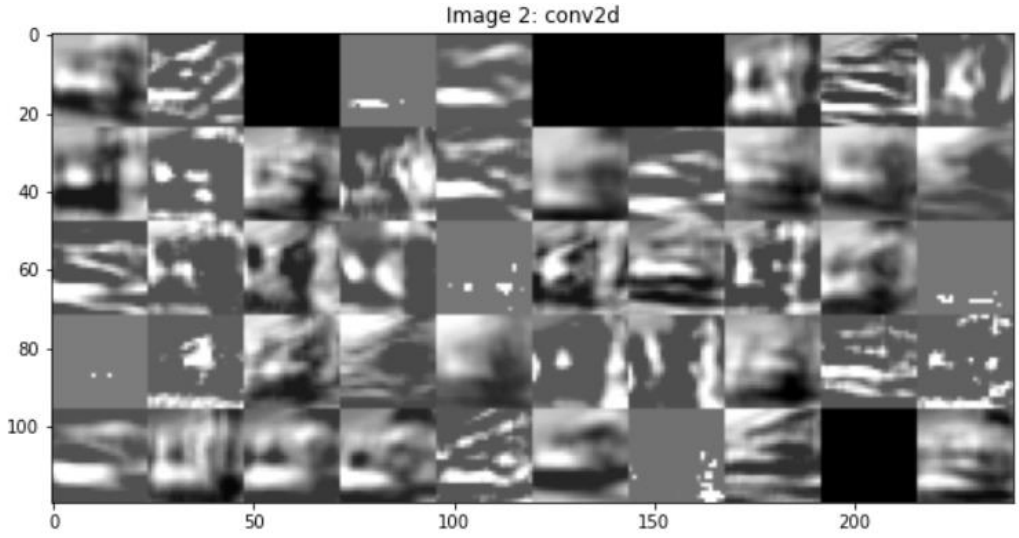


Figure 8: test image 2

The feature maps at convolution and pooling layers for test image 2 are as follows:



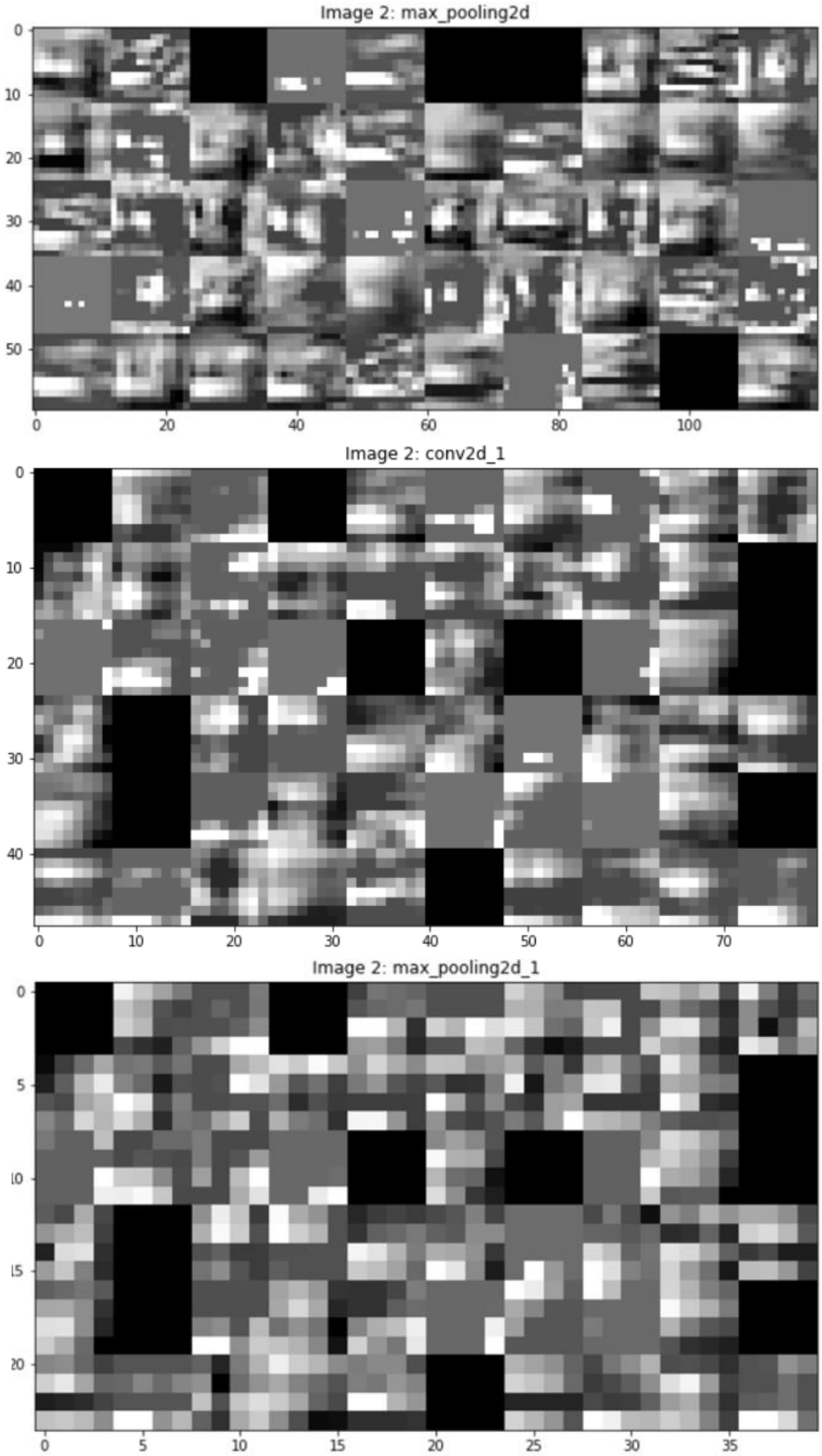


Figure 9: Feature maps of test image 2

Question 2:

- Use a grid search (C1ϵ {10, 30, 50, 70, 90}, C2 ϵ {20, 40, 60, 80, 100} , in total 25 combinations) to find the optimal combination of the numbers of channels at the convolution layers. Use the test accuracy to determine the optimal combination. Report all 25 accuracies.

For this question, a nested for loop is used to iterate through C1 and C2 for all 25 combinations of number of channels at the two convolution layers. Since it was observed from the previous question that the results begin to converge at around 500 epochs and the time taken to run all 100 epochs is significantly larger, 500 epochs will be used as a hyperparameter for the rest of the question. The results of test accuracies for the 25 combinations is shown below:

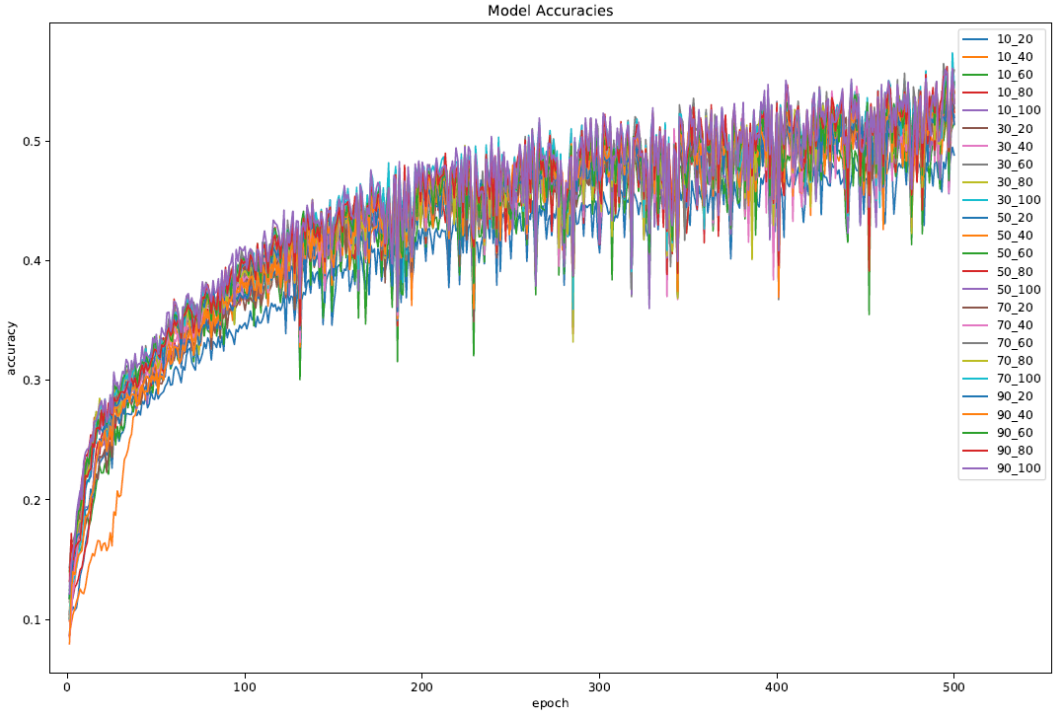


Figure 10: test accuracies for 25 channel combinations

In order to allow for closer inspection, the last 100 epochs of the above graph are shown below:

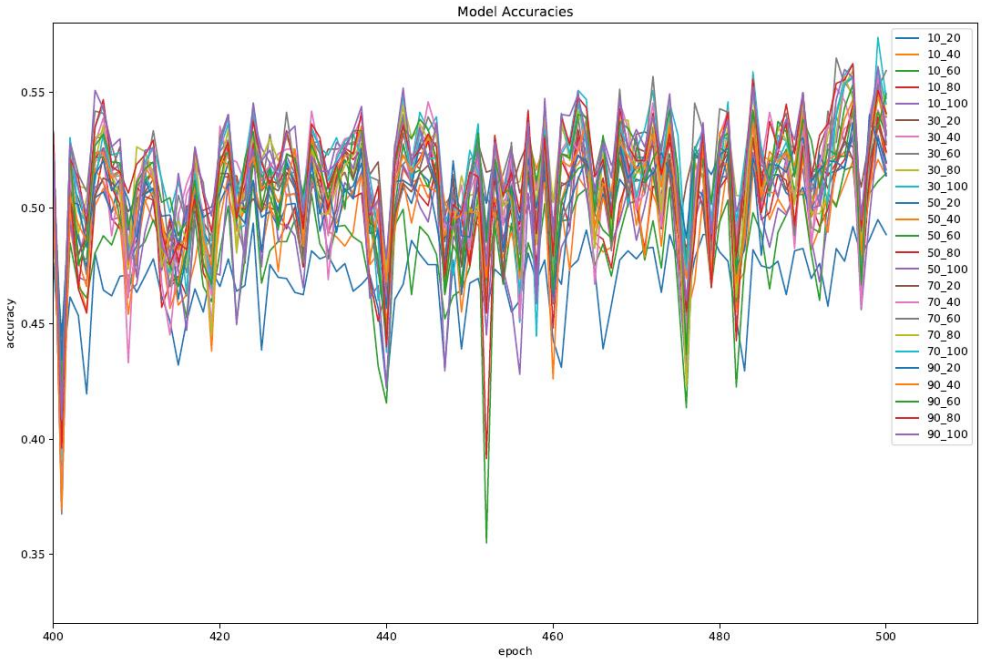


Figure 11: test accuracies for 25 channel combinations (last 100 epochs)

To find the most optimal combination, the average accuracy of the last 100 epochs was used. The results are shown in the table below:

Combination	**Accuracy (Average of last 100 epochs)**
10_20	0.4678699994087219
10_40	0.49419000089168547
10_60	0.480765001475811
10_80	0.4956400007009506
10_100	0.4897599998116493
30_20	0.5129200002551079
30_40	0.5062499994039535
30_60	0.5053050005435944
30_80	0.5116800019145011
30_100	0.5139799970388412
50_20	0.4999800005555153
50_40	0.49674000054597856
50_60	0.5035749992728233
50_80	0.505135001540184
50_100	0.51049999833107
70_20	0.5054099994897843

70_40	0.5129199987649917
70_60	***0.5190199983119964***
70_80	0.5115300014615058
70_100	0.5149500003457069
90_20	0.5066049998998642
90_40	0.5085800009965896
90_60	0.5123349997401238
90_80	0.5159800010919571
90_100	0.5181699967384339

Table 1: Accuracy of each combination of number of channels

The most optimal combination is found to be 70 channels for the first convolution layer and 60 channels for the second convolution layer. This combination will be used for the rest of the question.

Question 3:

Using the optimal combination found in part (2), train the network by:

a. adding the momentum term with momentum = 0.1
b. Using RMSProp algorithm for learning
c. Using Adam optimizer for learning
d. Adding dropout (probability=0.5) to the two fully connected layers. Plot the costs and accuracies against epochs (as in question 1(a)) for each case. Note that the sub-questions are independent. For instance, in (d), you do not need to modify the optimizer.

In order to tackle this question, the following function is used to facilitate running different optimizers:

```
def create_opt(optimizer_,lr):
    if optimizer_ == 'SGD':
        optimizer = keras.optimizers.SGD(learning_rate=lr)
    elif optimizer_ == 'SGD-momentum':  # Question 3(a)
        optimizer = keras.optimizers.SGD(learning_rate=lr, momentum = 0.1)
    elif optimizer_ == 'RMSProp':  # Question 3(b)
        optimizer = keras.optimizers.RMSprop(learning_rate=lr)
    elif optimizer_ == 'Adam':  # Question 3(c)
        optimizer = keras.optimizers.Adam(learning_rate=lr)
    else:
        raise NotImplementedError(f'You do not need to handle [{optimizer_}] in this project.')
    return optimizer
```

Part A)

The results of adding momentum = 0.1 to the optimal model is shown below:

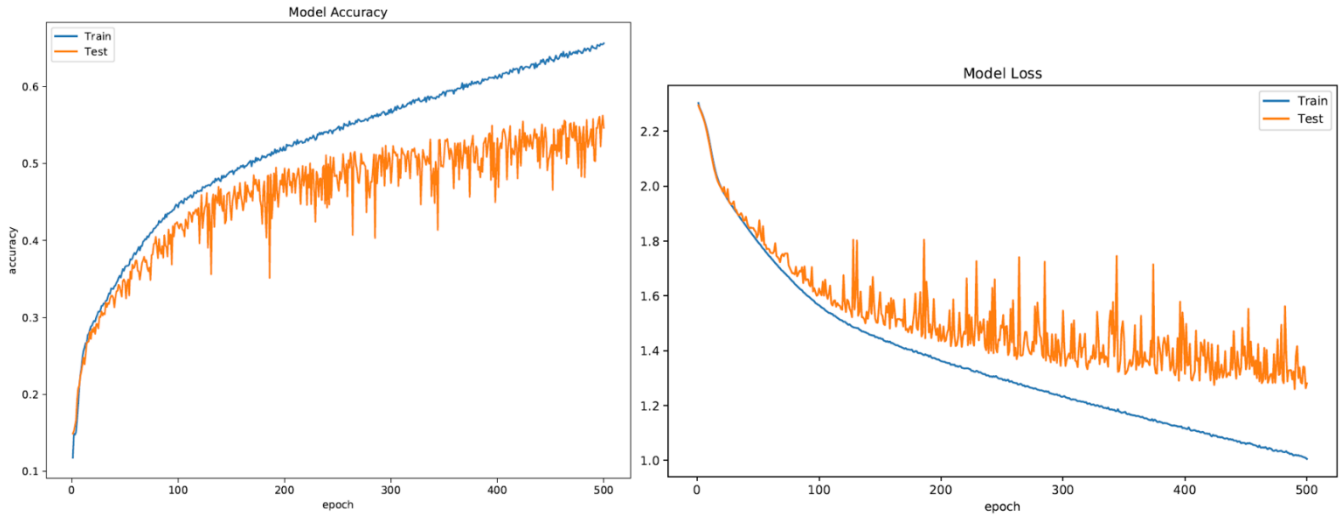


Figure 12: Performance of model with momentum = 0.1

Part B)

The results of using RMSProp instead of SGD for the optimizer is shown below:

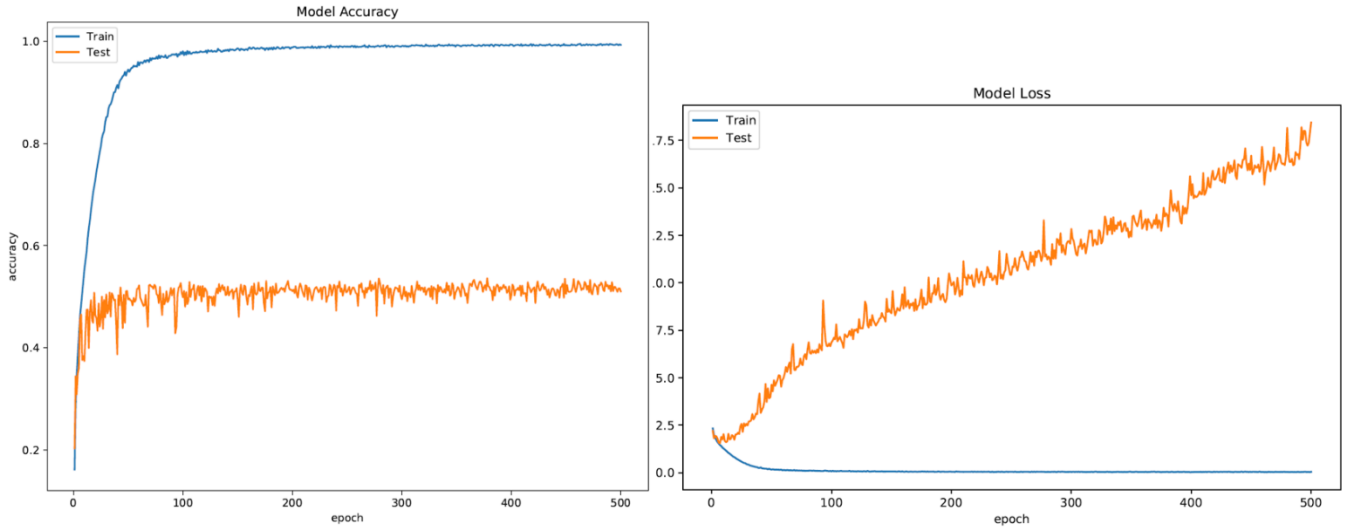


Figure 13: Performance of RMSProp model

Part C)

The results of using Adam optimizer instead of SGD is shown below:

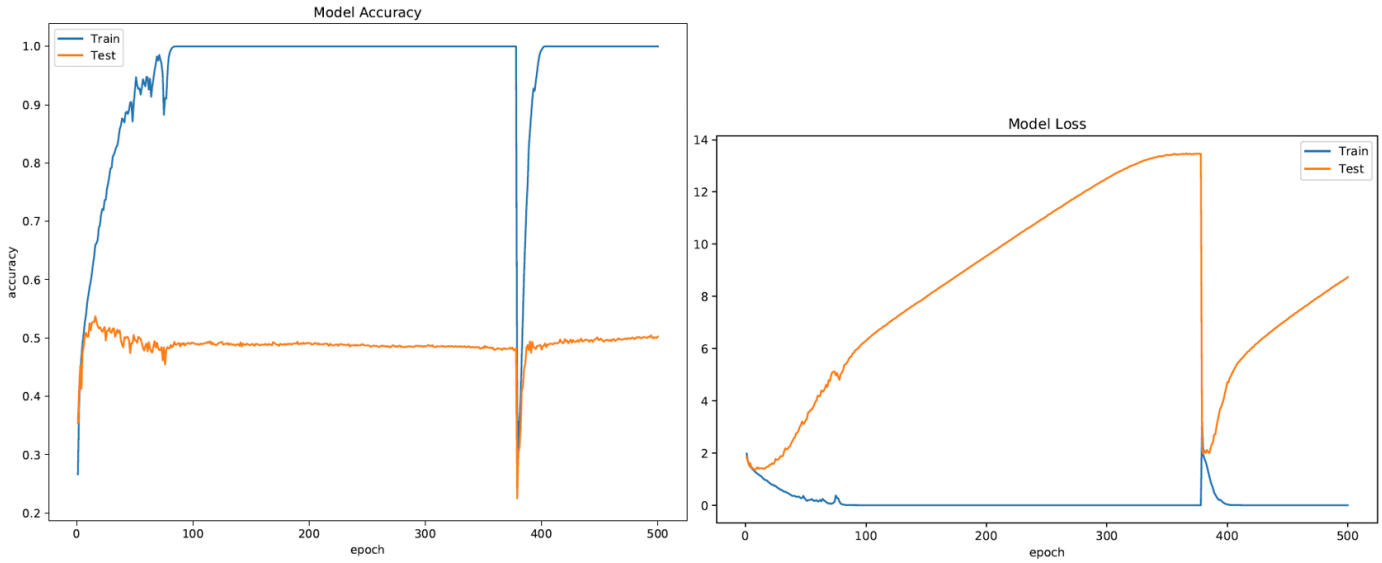


Figure 14: Performance of Adam model

Part D)

The results of introducing dropout with probability of 0.5 is shown below:

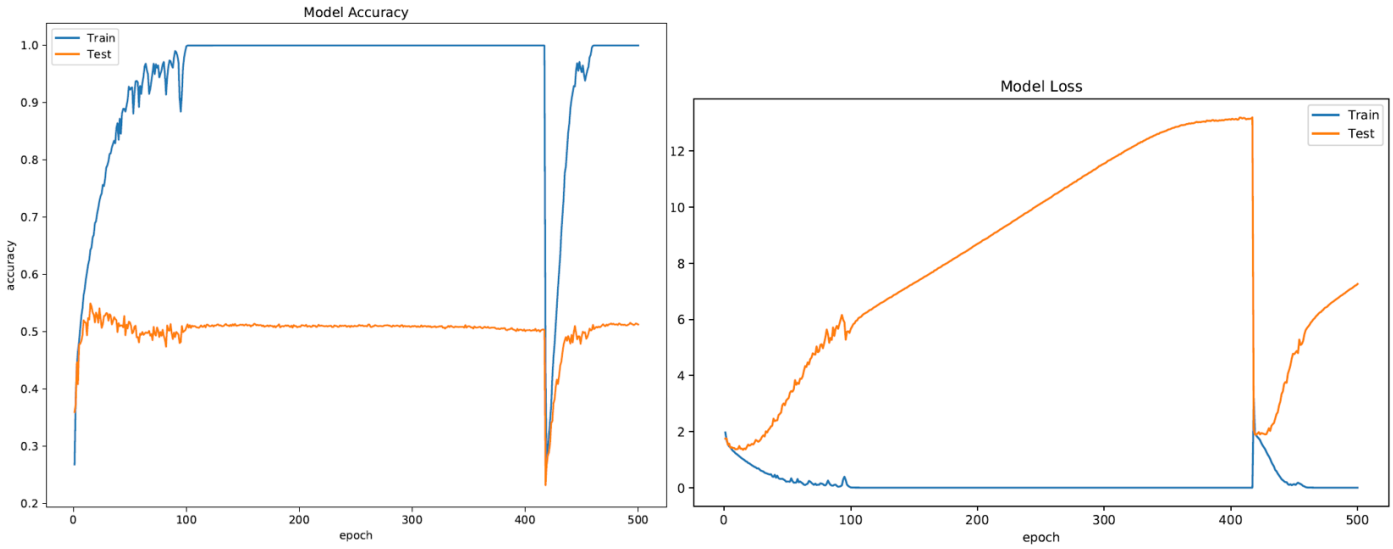


Figure 15: Performance of Adam model with dropout rate = 0.5

Question 4:

- Compare the accuracies of all the models from parts (1) - (3) and discuss their performances.

Adding the models in Question 3 into the graph of accuracies in Question 2, the follow graph was found for test accuracies:

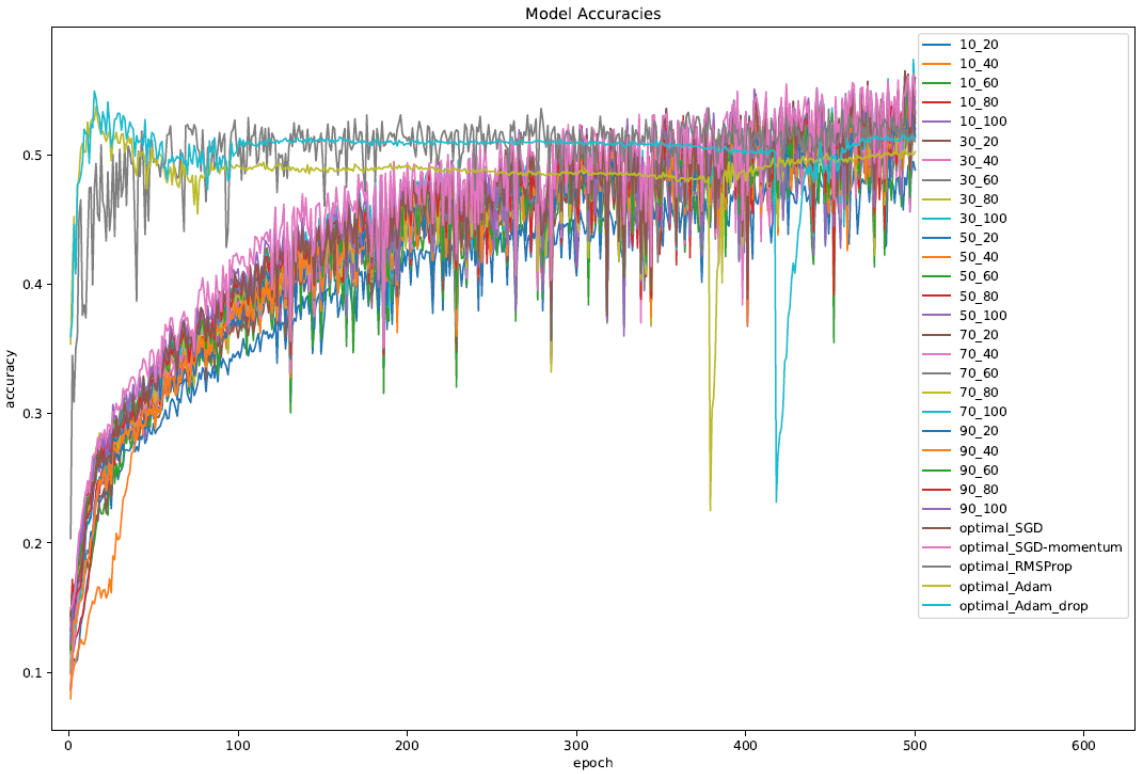


Figure 16: Results of test accuracies from questions 1-3

To find the best model from the candidates above, the average test accuracies of the final 100 epochs for models are calculated. However, since it has been observed from the models with Adam optimizer that there are brief, sharp drops in accuracy and losses regularly, only the final 50 epochs will be taken into account for the Adam models:

Changes made	**Accuracy (Average of last 100 epochs)**
SGD with momentum = 0.1	***0.527365002930164 3***
RMSProp	0.5185400012135506
Adam	0.4981699997186661
Adam with dropout = 0.5	0.5091800010204315
SGD	0.5189500007033349

Table 2: Accuracies of results from question 3

As can be observed from the table above, the most optimal model is the model with 70 channels in the first convolution layer, 60 channels in the second convolution layer, and using the SGD optimizer with a momentum of 0.1.

From Figure 16 and tables 1 and 2, we can infer that:

The number of channels for the first and second filters have an effect on the overall performance of the model. Generally speaking, increasing the number of channels will also improve performance, especially for the first convolution layer. This may be because the earlier filters are generally used to capture the basic details of the input image, thus having a larger first layer can result in and faster and more accurate training. However, the trade-off for this is the computational time since the amount of time needed to train models with larger number of channels for each layer is significantly higher than those with small number of channels.

The results also show that adding momentum to the SGD optimizer is effective in improving performance. Momentum is used for helping SGD accelerate in the correct direction and dampen oscillations around local optimum points. This may lead to faster convergence and better performance in general. This holds true for the case in this question as can be observed in the figure below, which is a comparison between the SGD model with and without momentum. The model with momentum converges slightly faster and is able to achieve a slightly better result than the model without momentum.

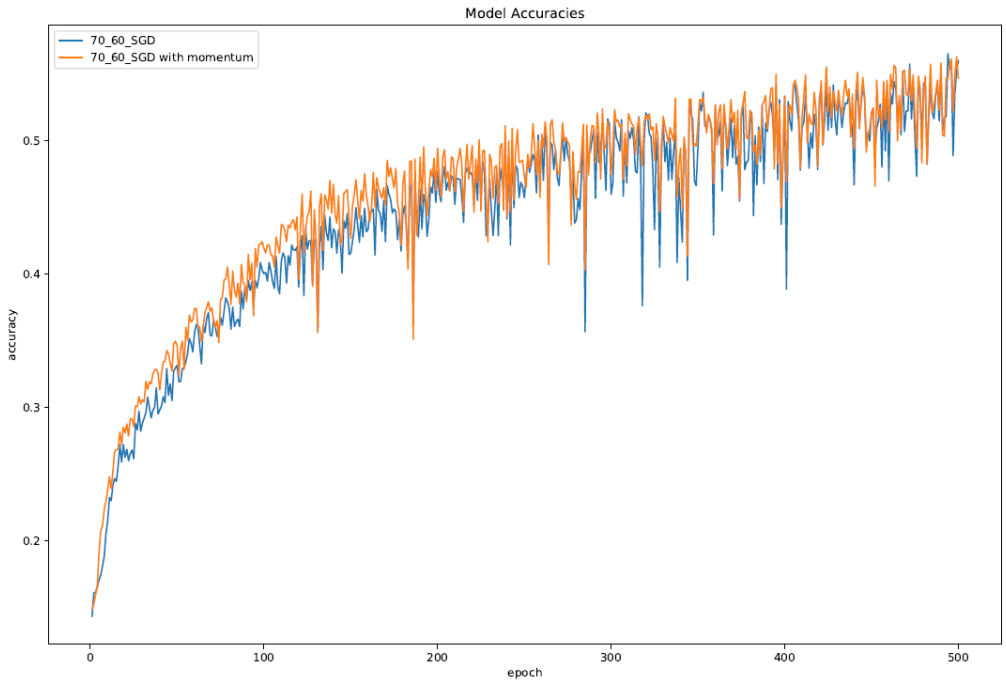


Figure 17: SGD vs SGD with momentum

It can also be observed that the models using the Adam optimizer and RMSProp optimizer is able to converge significantly faster than those using variations of the SGD optimizer. This may be due to the fact that the learning rate is not fixed for the former two optimizers, thus

allowing them to take bigger steps at the beginning epochs of training. However, given enough number of epochs (around 400-500 judging from Figure 16), the models using SGD tends to outperform models using Adam or RMSProp optimizers.

Furthermore, the Adam and RMSProp optimizers seem to have increasing losses from extremely early in training. This means that these two optimizers are able to achieve convergence extremely quickly, then begins to overfit.

From figure 14, 15 and table 2, we can observe that applying dropout to the Adam model improves the test accuracy considerably. This shows that the model with Adam optimizer is likely experiencing overfitting since dropouts have a chance of disabling high impact nodes. The reason for this may be because of the small data size used for training.

Conclusion

In this part of the assignment, convolutional neural networks were trained and analysed. The optimal combination of characteristics for the CNN with best test accuracy was found to be:

Model type	SGD
Momentum	0.1
Number of channels (First convolution layer)	70
Number of channels (Second convolution layer)	60

Table 1: Optimal hyperparameters

Depending on the number of epochs and the size of the training dataset, the Adam and RMSProp optimizers may achieve better results than that of SGD/SGD with momentum. However, for the dataset and parameters given in this question, Adam and RMSProp optimizers achieve convergence significantly faster than SGD optimizers but SGD optimizers are able to achieve a higher test accuracy at the end of 250 epochs. Thus, SGD with momentum is chosen to be the most optimal optimizer for this question.

Part B: Text Classification

Introduction

The aim of this assignment is to analyse the effectiveness of different neural networks (namely CNNs and RNNs) in processing text-based information and classifying them into 15 different categories. The results for both character and word classification will be analysed. The dataset used is gathered from the first paragraphs of Wikipage entries, split into 5600 entries for training data and 700 entries for testing data. In this assignment, the following models will be trained and analysed:

- CNN classifier with 2 convolutional layers and 2 max pooling layers (details will be given in later sections)
- RNN classifier using GRU layer
- RNN classifier using vanilla RNN layer
- RNN classifier using LSTM layer
- RNN classifier using 2 GRU layers
- RNN classifier with gradient clipping

All figures and codes provided in this report is present in the corresponding jupyter notebook files for each question.

Methods

Data is processed differently for character and word level classifiers.

For character level, one-hot encoding is used, where the data can be converted into a binary format for each unique entry. By doing this, neural networks would be able to process information more efficiently. An example of one hot encoding is as follows:

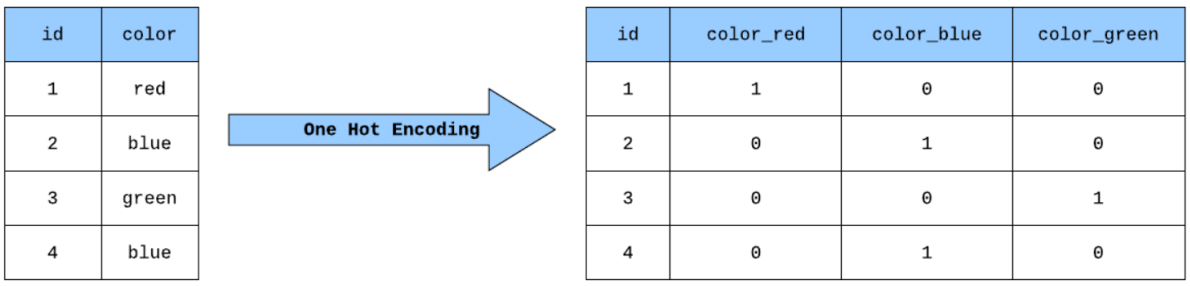


Figure 1: One-hot encoding

For word level processing, word embedding is used to represent words in a vector form. By doing this, words with similar meanings will have similar representations and a better context of the word can be captured. This is a very popular technique in deep learning which is capable of achieving good results. A representation of word embeddings is shown below:

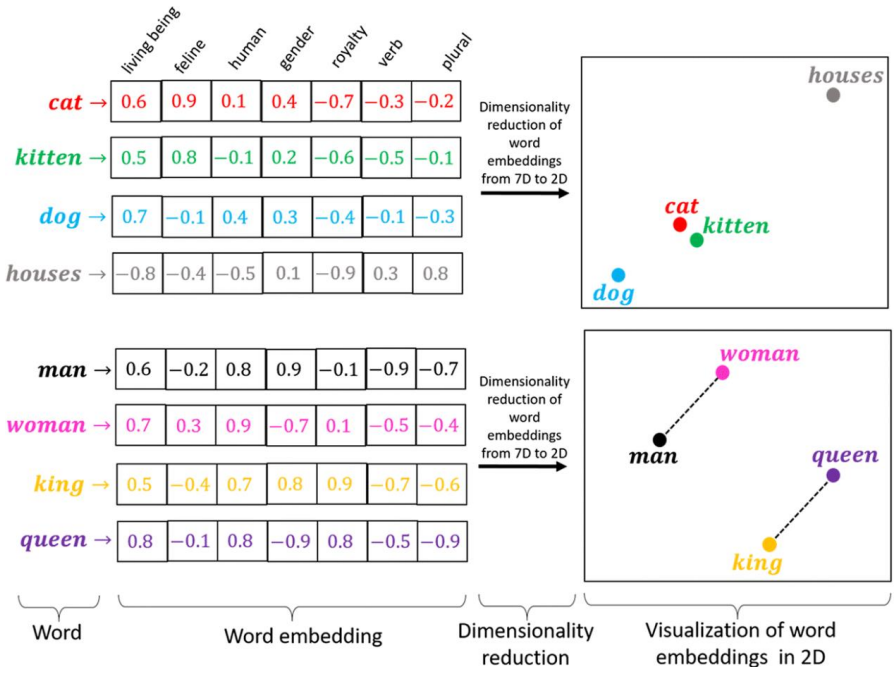


Figure 2: Representation of word embeddings

The implementations of data loading and pre-processing for character and word level processing is shown below:

```
def read_data_chars():
    x_train, y_train, x_test, y_test = [], [], [], []
    cop = re.compile("[^a-z^A-Z^0-9^,^.^' ']")
    with open('./data/train_medium.csv', encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            data = cop.sub("", row[1])
            x_train.append(data)
            y_train.append(int(row[0]))

    with open('./data/test_medium.csv', encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            data = cop.sub("", row[1])
            x_test.append(data)
            y_test.append(int(row[0]))
    vocab_size, char_to_ix = vocabulary(x_train+x_test)
    x_train = preprocess(x_train, char_to_ix, MAX_DOCUMENT_LENGTH)
    y_train = np.array(y_train)
    x_test = preprocess(x_test, char_to_ix, MAX_DOCUMENT_LENGTH)
    y_test = np.array(y_test)

    x_train = tf.constant(x_train, dtype=tf.int64)
    y_train = tf.constant(y_train, dtype=tf.int64)
    x_test = tf.constant(x_test, dtype=tf.int64)
    y_test = tf.constant(y_test, dtype=tf.int64)

    return x_train, y_train, x_test, y_test
```

```python
def preprocess(strings, char_to_ix, MAX_LENGTH):
    data_chars = [list(d.lower()) for _, d in enumerate(strings)]
    for i, d in enumerate(data_chars):
        if len(d)>MAX_LENGTH:
            d = d[:MAX_LENGTH]
        elif len(d) < MAX_LENGTH:
            d += [' '] * (MAX_LENGTH - len(d))

    data_ids = np.zeros([len(data_chars), MAX_LENGTH], dtype=np.int64)
    for i in range(len(data_chars)):
        for j in range(MAX_LENGTH):
            data_ids[i, j] = char_to_ix[data_chars[i][j]]
    return np.array(data_ids)
```

```python
def vocabulary(strings):
    chars = sorted(list(set(list(''.join(strings)))))
    char_to_ix = { ch:i for i,ch in enumerate(chars) }
    vocab_size = len(chars)
    return vocab_size, char_to_ix
```

Figure 3: Implementation of data processing for character level RNNs

```python
def read_data_chars():
    x_train, y_train, x_test, y_test = [], [], [], []
    cop = re.compile("[^a-z^A-Z^0-9^,^.^' ']")
    with open('./data/train_medium.csv', encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            data = cop.sub("", row[1])
            x_train.append(data)
            y_train.append(int(row[0]))

    with open('./data/test_medium.csv', encoding='utf-8') as filex:
        reader = csv.reader(filex)
        for row in reader:
            data = cop.sub("", row[1])
            x_test.append(data)
            y_test.append(int(row[0]))

    word_dict = build_word_dict(x_train+x_test)
    x_train = preprocess(x_train, word_dict, MAX_DOCUMENT_LENGTH)
    y_train = np.array(y_train)
    x_test = preprocess(x_test, word_dict, MAX_DOCUMENT_LENGTH)
    y_test = np.array(y_test)

    x_train = [x[:MAX_DOCUMENT_LENGTH] for x in x_train]
    x_test = [x[:MAX_DOCUMENT_LENGTH] for x in x_test]
    x_train = tf.constant(x_train, dtype=tf.int64)
    y_train = tf.constant(y_train, dtype=tf.int64)
    x_test = tf.constant(x_test, dtype=tf.int64)
    y_test = tf.constant(y_test, dtype=tf.int64)

    vocab_size = tf.get_static_value(tf.reduce_max(x_train))
    vocab_size = max(vocab_size, tf.get_static_value(tf.reduce_max(x_test))) + 1

    return x_train, y_train, x_test, y_test, vocab_size
```

```python
def build_word_dict(contents):
    words = list()
    for content in contents:
        for word in word_tokenize(clean_str(content)):
            words.append(word)

    word_counter = collections.Counter(words).most_common()
    word_dict = dict()
    word_dict["<pad>"] = 0
    word_dict["<unk>"] = 1
    word_dict["<eos>"] = 2
    for word, _ in word_counter:
        word_dict[word] = len(word_dict)
    return word_dict
```

```python
def preprocess(contents, word_dict, document_max_len):
    x = list(map(lambda d: word_tokenize(clean_str(d)), contents))
    x = list(map(lambda d: list(map(lambda w: word_dict.get(w, word_dict["<unk>"]), d)), x))
    x = list(map(lambda d: d + [word_dict["<eos>"]], x))
    x = list(map(lambda d: d[:document_max_len], x))
    x = list(map(lambda d: d + (document_max_len - len(d)) * [word_dict["<pad>"]], x))
    return x
```

Figure 4: Implementation of data processing for word level RNNs

For the models in this assignment, Convolutional Neural Networks (CNN) and Recurrent Neural Networks will be used.

As mentioned in Part A, CNNs are a class of deep neural networks that is commonly applied in imagery. Their architecture is inspired by the connectivity pattern of neurons in biology.

RNNs, on the other hand, differ from CNNs in its architecture. RNNs are designed to process and interpret temporal data and uses other data points in a sequence to improve its results. This means that RNNs are able to effectively learn from inputs that come in sequence, such as sentences or videos. Several RNNs will be analysed in this assignment, including Vanilla RNN, LSTM, and GRU.

Vanilla RNN, every prediction is dependent on the hidden state weight which is responsible for remembering information about the previous sequences. This means Vanilla RNN is susceptible to the gradient vanishing problem if the weight is small since it uses back-propagation. Due to this, important information may be lost if it is far away temporally.

Long Short Term Memory (LSTM) attempts to provide a solution to the gradient vanishing problem in Vanilla RNNs. It does this by incorporating cell states which may store long term information. There are three gates that are responsible how the information is discarded, stored, and retrieved. They are the forget gate, Input gate, and output gate respectively. The LSTM approach has improved the performances of RNNs significantly.

Gated Recurrent Unit (GRU) also attempts to solve the gradient vanishing problem, but does so in a simpler fashion than the LSTM mentioned above. GRUs only utilise two gates: the reset gate and update gate which determines what information should be passed to the output.

Experiments and Results

Question 1:

Design a Character CNN Classifier that receives character ids and classifies the input. The CNN has two convolution and pooling layers:

- A convolution layer C1 of 10 filters of window size 20x256, VALID padding, and ReLU neurons. A max pooling layer S1 with a pooling window of size 4x4, with stride = 2, and padding = 'SAME'.

- A convolution layer C2 of 10 filters of window size 20x1, VALID padding, and ReLU neurons. A max pooling layer S2 with a pooling window of size 4x4, with stride = 2 and padding = 'SAME'.

Plot the entropy cost on the training data and the accuracy on the testing data against training epochs.

The hyperparameters used for this question is as follows:

```
MAX_DOCUMENT_LENGTH = 100
N_FILTERS = 10
FILTER_SHAPE1 = [20, 256]
FILTER_SHAPE2 = [20, 1]
POOLING_WINDOW = 4
POOLING_STRIDE = 2
MAX_LABEL = 15

batch_size = 128
no_epochs = 250
lr = 0.01
# Dropout with probability of 0.5
use_dropout = True

seed = 10
tf.random.set_seed(seed)
```

The data is loaded by the **read_data_chars()** function and shuffled before training. For this question, categorical cross entropy is used as the loss function and accuracy is used as the metric for measuring model performance, the SGD optimizer is also used to train the model. The model that is generated and trained using the hyperparameters above is as follows:

```
tf.keras.backend.set_floatx('float32')
class CharCNN(tf.keras.Model):
    def __init__(self, vocab_size, drop_rate):
        super(CharCNN, self).__init__()
        self.vocab_size = vocab_size
        self.drop_rate = drop_rate

        # Weight variables and RNN cell
        self.conv1 = tf.keras.layers.Conv2D(N_FILTERS, FILTER_SHAPE1, padding='VALID', activation='relu', use_bias=True)
        self.pool1 = tf.keras.layers.MaxPool2D(POOLING_WINDOW, POOLING_STRIDE, padding='SAME')
        self.conv2 = tf.keras.layers.Conv2D(N_FILTERS, FILTER_SHAPE2, padding='VALID', activation='relu', use_bias=True)
        self.pool2 = tf.keras.layers.MaxPool2D(POOLING_WINDOW, POOLING_STRIDE, padding='SAME')
        self.flatten = tf.keras.layers.Flatten()
        self.dense = tf.keras.layers.Dense(MAX_LABEL, activation='softmax')

    def call(self, x, drop_rate):
        # forward
        x = tf.one_hot(x, 256)
        x = x[..., tf.newaxis]
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = tf.nn.dropout(x, drop_rate)
        logits = self.dense(x)
        return logits
```

The accuracy and costs of training and testing data against the number of epochs can be observed from the graph below:

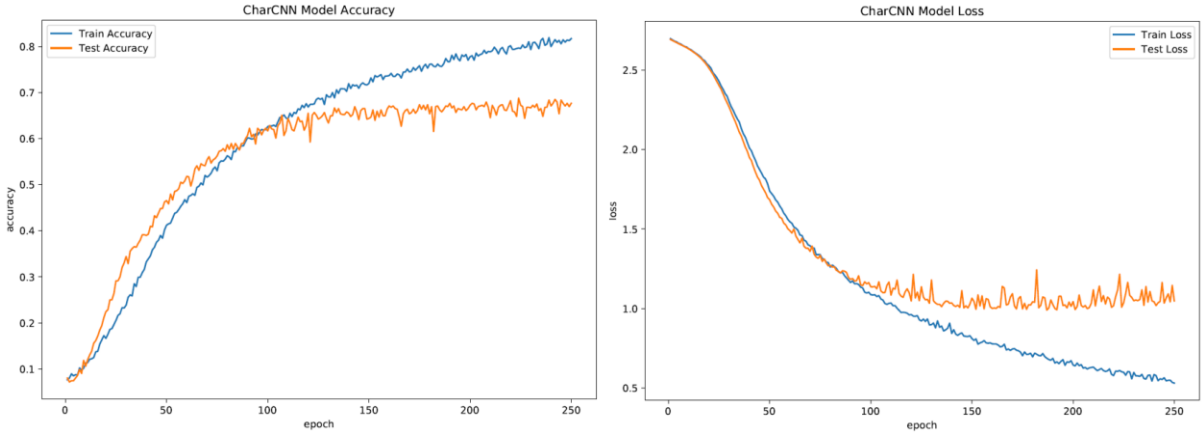


Figure 5: CharCNN model performance

Question 2:

Design a Word CNN Classifier that receives word ids and classifies the input. Pass the inputs through an embedding layer of size 20 before feeding to the CNN. The CNN has two convolution and pooling layers with the following characteristics:

- A convolution layer C1 of 10 filters of window size 20x20, VALID padding, and ReLU neurons. A max pooling layer S1 with a pooling window of size 4x4, with stride = 2 and padding = 'SAME'.
- A convolution layer C2 of 10 filters of window size 20x1, , VALID padding, and ReLU neurons. A max pooling layer S2 with a pooling window of size 4x4, with stride = 2 and padding = 'SAME'.

Plot the entropy cost on training data and the accuracy on testing data against training epochs.

The hyperparameters used for this question is as follows:

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

batch_size = 128
no_epochs = 250
lr = 0.01
use_dropout = True

seed = 10
tf.random.set_seed(seed)
```

The data is loaded by the **read_data_chars()** function and shuffled before training. For this question, categorical cross entropy is used as the loss function and accuracy is used as the metric for measuring model performance, the SGD optimizer is also used to train the model. The model that is generated and trained using the hyperparameters above is as follows:

```
tf.keras.backend.set_floatx('float32')
class WordCNN(tf.keras.Model):
    def __init__(self, vocab_size, drop_rate):
        super(WordCNN, self).__init__()
        self.vocab_size = vocab_size
        self.drop_rate = drop_rate

        # Weight variables and RNN cell
        self.embedding = tf.keras.layers.Embedding(vocab_size, EMBEDDING_SIZE, input_length=MAX_DOCUMENT_LENGTH)
        self.conv1 = tf.keras.layers.Conv2D(N_FILTERS, FILTER_SHAPE1, padding='VALID', activation='relu', use_bias=True)
        self.pool1 = tf.keras.layers.MaxPool2D(POOLING_WINDOW, POOLING_STRIDE, padding='SAME')
        self.conv2 = tf.keras.layers.Conv2D(N_FILTERS, FILTER_SHAPE2, padding='VALID', activation='relu', use_bias=True)
        self.pool2 = tf.keras.layers.MaxPool2D(POOLING_WINDOW, POOLING_STRIDE, padding='SAME')
        self.flatten = tf.keras.layers.Flatten()
        self.dense = tf.keras.layers.Dense(MAX_LABEL, activation='softmax')

    def call(self, x, drop_rate):
        # forward
        x = self.embedding(x)
        x = x[..., tf.newaxis]
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = tf.nn.dropout(x, drop_rate)
        logits = self.dense(x)
        return logits
```

The accuracy and costs of training and testing data against the number of epochs can be observed from the graph below:

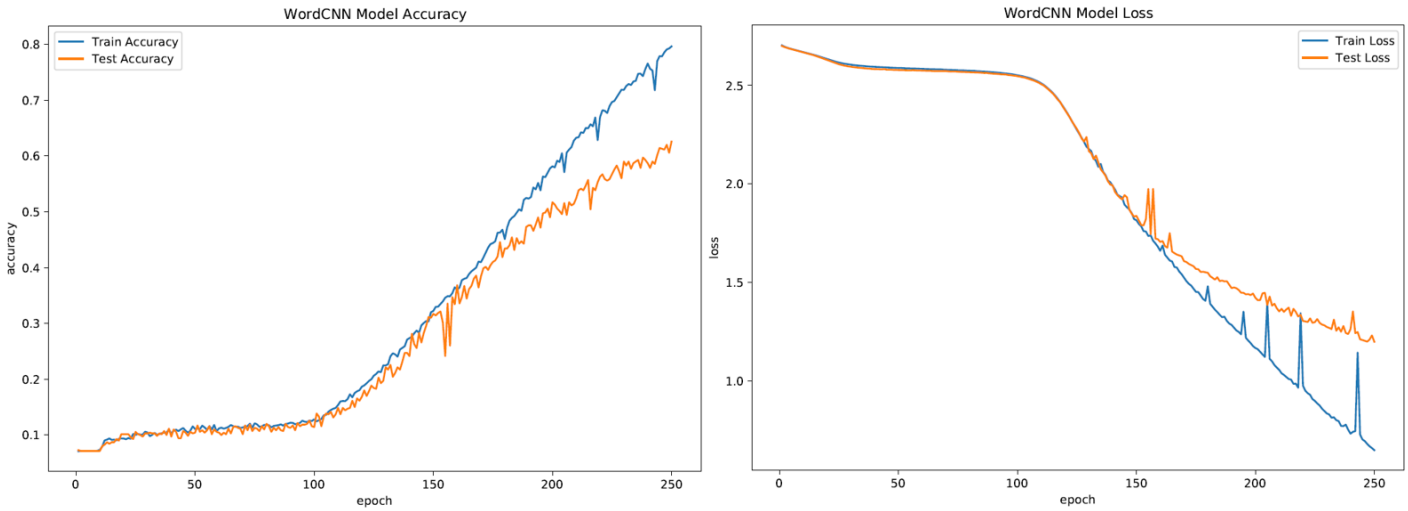


Figure 6: WordCNN model performance

Question 3:

Design a Character RNN Classifier that receives character ids and classify the input. The RNN is GRU layer and has a hidden-layer size of 20.

Plot the entropy cost on training data and the accuracy on testing data against training epochs.

For this question, categorical cross entropy is used as the loss function and accuracy and mean squared error is used as metrics for measuring model performance. The Adam optimizer is also used to train the model instead. The parameter HIDDEN_SIZE is set to 20, and all other hyperparameters remains the same as the ones in Question 1. The code for the model used in this question is shown below:

```
class CharRNN(Model):
    def __init__(self, vocab_size, hidden_dim=20):
        super(CharRNN, self).__init__()
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        # Weight variables and RNN cell
        self.rnn = layers.RNN(
            tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        self.dense = layers.Dense(MAX_LABEL, activation=None)
    def call(self, x, drop_rate):
        # forward logic
        x = tf.one_hot(x, 256)
        x = self.rnn(x)
        x = tf.nn.dropout(x, drop_rate)
        logits = self.dense(x)

        return logits
```

The accuracy and loss of training and testing data against the number of epochs can be observed from the graph below:

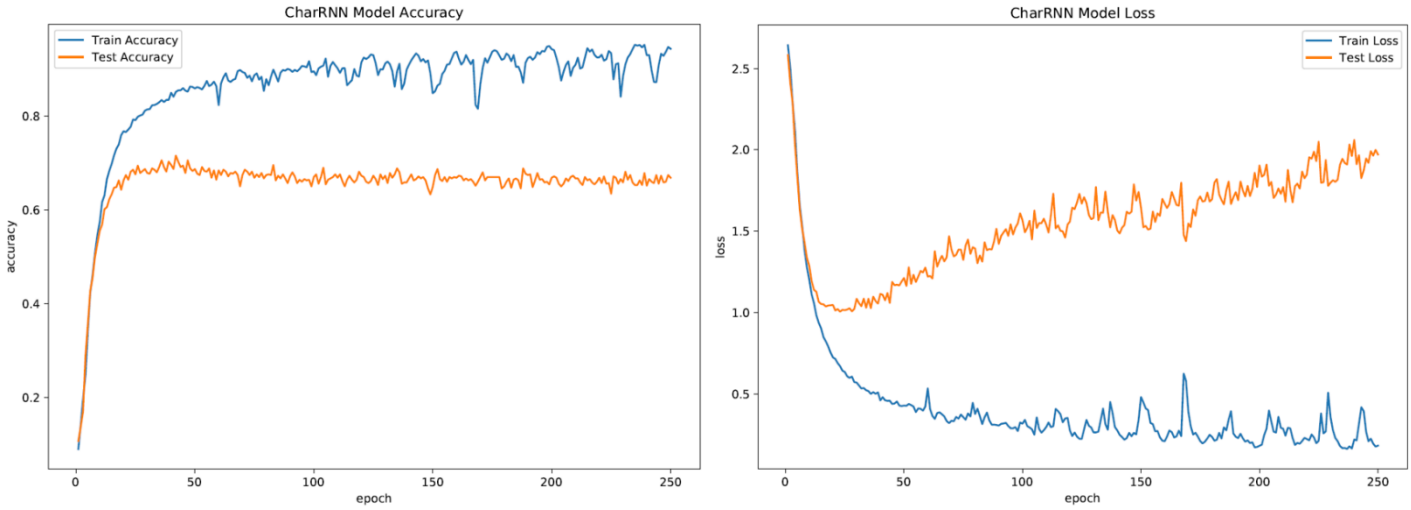


Figure 7: CharRNN GRU model performance

Question 4:

- Design a word RNN classifier that receives word ids and classify the input. The RNN is GRU layer and has a hidden-layer size of 20. Pass the inputs through an embedding layer of size 20 before feeding to the RNN.

- Plot the entropy on training data and the accuracy on testing data versus training epochs.

For this question, categorical cross entropy is used as the loss function and accuracy and mean squared error is used as metrics for measuring model performance. The Adam optimizer is used to train the model instead. The parameter HIDDEN_SIZE is set to 20, and all other hyperparameters remains the same as the ones in Question 2. The code for the model used in this question is shown below:

```
class WordRNN(Model):
    def __init__(self, vocab_size, hidden_dim=20):
        super(WordRNN, self).__init__()
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size

        # Weight variables and RNN cell
        self.embedding = layers.Embedding(vocab_size, EMBEDDING_SIZE, input_length=MAX_DOCUMENT_LENGTH)
        self.rnn = layers.RNN(
            tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        self.dense = layers.Dense(MAX_LABEL, activation=None)

    def call(self, x, drop_rate):
        # forward logic
        embedding = self.embedding(x)
        encoding = self.rnn(embedding)

        encoding = tf.nn.dropout(encoding, drop_rate)
        logits = self.dense(encoding)

        return logits
```

The accuracy and loss of training and testing data against the number of epochs can be observed from the graph below:

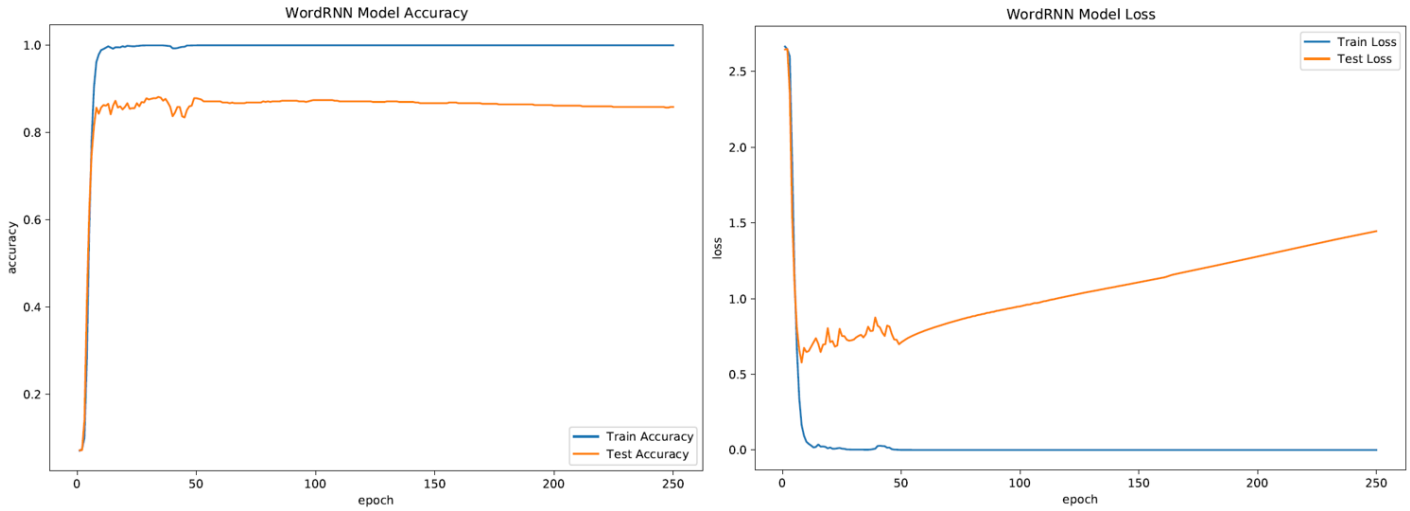


Figure 8: WordRNN GRU model performance

Question 5:

- Compare the test accuracies and the running times of the networks implemented in parts (1) – (4). Experiment with adding dropout to the layers of networks in parts (1) – (4), and report the test accuracies.
- Compare and comment on the accuracies of the networks with/without dropout.

The test accuracies of models in Questions 1-4 against the number of epochs and their time taken to train is shown below:

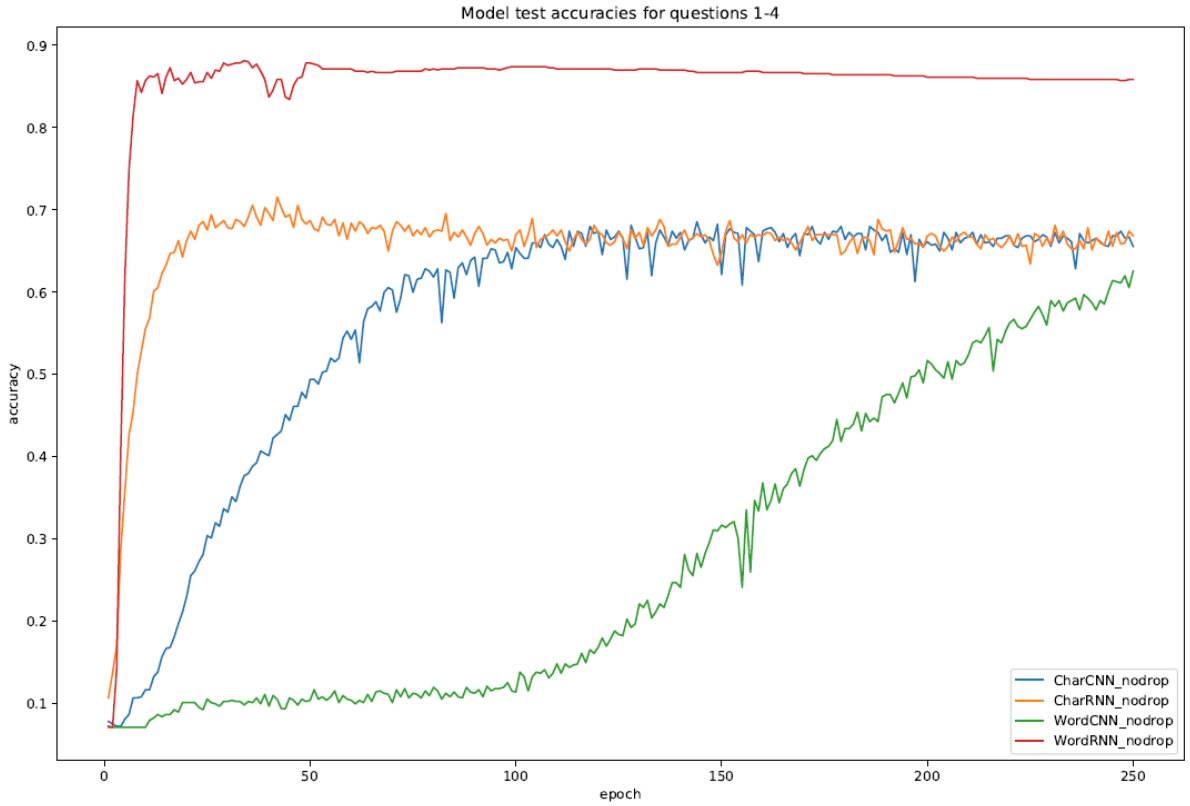


Figure 9: Test accuracies for models in Questions 1-4 (with no dropouts)

Model Name	**Time taken (seconds)**
CharCNN	442.877528945197
WordCNN	917.851375485221
CharRNN	2869.507857322693
WordRNN	3191.319160461426

Table 1: Time taken to train models in Questions 1-4 (with no dropouts)

As can be seen from Table 1, the CNN models for both character and word levels take notably less time than their RNN counterparts. However, as observed from Figure 9, the CharRNN model is able to converge at around 25 epochs and the WordRNN model is able to

converge at around 20 epochs. This is significantly better than the convergence point of both the CNN models. Thus, RNN models can be considered to have better performance in these cases. This may be due to the fact that CNNs are more tailored to classifying individual data presented in an imagelike format whereas RNNs are made to take into account the temporal information at every point. Thus, in the case of this question which requires language processing, the architecture of RNNs have an advantage over that of CNNs.

By changing the dropout from False to True and rerunning the code, dropout is applied to the models in previous questions as shown below:

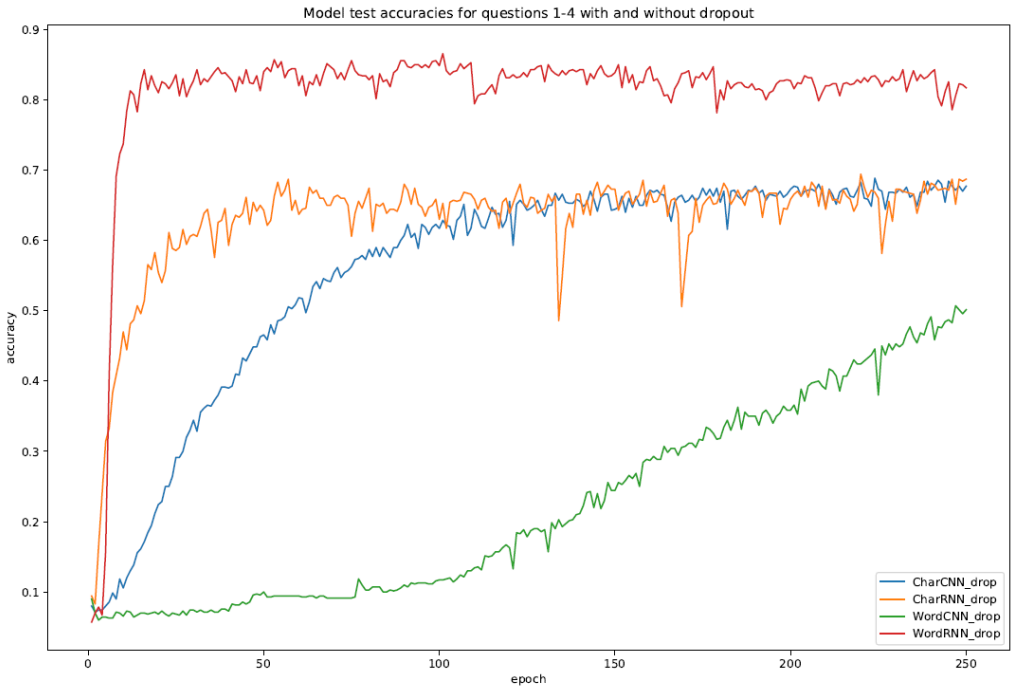


Figure 10: Test accuracies for models in Questions 1-4 (with dropouts)

By displaying the results of the models trained with dropout on the same graph as their non-dropout counterparts, we can analyse the similarities and differences between them:

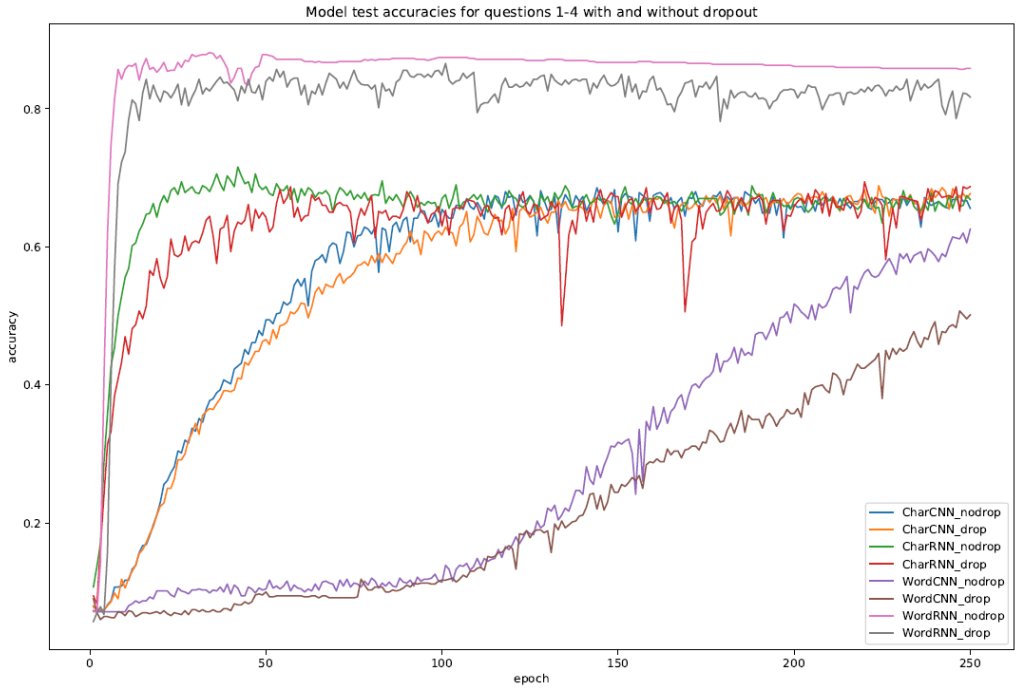


Figure 11: Test accuracies for models in Questions 1-5

The accuracies and time taken of the above models are shown below:

Model Name	**Test Accuracy**	**Time Taken**
CharCNN no dropout	0.6638857150077819	342.8775289451971
CharCNN with dropout	0.6652999997138977	403.5856545202113
WordCNN no dropout	0.4843142853677273	917.8513754852217
WordCNN with dropout	0.3771999987959862	1034.254899142719
CharRNN no dropout	0.6641999989748001	2869.507857322693
CharRNN with dropout	0.6578857159614563	3002.123514427462
WordRNN no dropout	0.8624571400880814	3191.319160461426
WordRNN with dropout	0.8231999975442886	3316.153774323003

Table 2: Time taken to train models in Questions 1-5

From Table 2 and Figure 11 above, the following can be inferred:

Adding dropouts to each model will slightly increase the computational time needed to train the models. This makes sense because adding dropouts means that there is one more function that has to be run at each epoch. However, this does not affect the total time taken significantly since turning off neurons is not computationally intensive.

Adding dropouts tend to hinder the overall performance of each model, especially for the WordCNN models. This may be due to the fact that dropout adds a level of randomness to training that may lead to a longer time for the model to achieve convergence.

Since none of the models benefitted from adding dropouts, it is reasonable to guess that none of the models have notable signs of overfitting.

Question 6:

- For RNN networks implemented in (3) and (4), perform the following experiments with the aim of improving performances, compare the accuracies and report your findings:
    - Replace the GRU layer with (i) a vanilla RNN layer and (ii) a LSTM layer.
    - Increase the number of RNN layers to 2 layers
    - Add gradient clipping to RNN training with clipping threshold = 2.

For this question, the hyperparameters remain the same as the ones in the previous questions. The implementations of the models are as shown below:

```
class CharRNN(Model):
    def __init__(self, vocab_size, hidden_dim=20):
        super(CharRNN, self).__init__()
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        # Weight variables and RNN cell
        if RNN_type == 'GRU2':
            self.rnn = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True, return_sequences=True)
            self.rnn2 = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'GRU':
            self.rnn = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'Vanilla':
            self.rnn = layers.RNN(
                tf.keras.layers.SimpleRNNCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'LSTM':
            self.rnn = layers.RNN(
                tf.keras.layers.LSTMCell(self.hidden_dim), unroll=True)
        self.dense = layers.Dense(MAX_LABEL, activation=None)
    def call(self, x, drop_rate):
        # forward logic
        x = tf.one_hot(x, 256)
        x = self.rnn(x)
        if RNN_type == 'GRU2':
            x = self.rnn2(x)
        x = tf.nn.dropout(x, drop_rate)
        logits = self.dense(x)

        return logits
```

```python
class WordRNN(Model):
    def __init__(self, vocab_size, hidden_dim=20):
        super(WordRNN, self).__init__()
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size

        # Weight variables and RNN cell
        self.embedding = layers.Embedding(vocab_size, EMBEDDING_SIZE, input_length=MAX_DOCUMENT_LENGTH)
        if RNN_type == 'GRU2':
            self.rnn = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True, return_sequences=True)
            self.rnn2 = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'GRU':
            self.rnn = layers.RNN(
                tf.keras.layers.GRUCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'Vanilla':
            self.rnn = layers.RNN(
                tf.keras.layers.SimpleRNNCell(self.hidden_dim), unroll=True)
        elif RNN_type == 'LSTM':
            self.rnn = layers.RNN(
                tf.keras.layers.LSTMCell(self.hidden_dim), unroll=True)
        self.dense = layers.Dense(MAX_LABEL, activation=None)

    def call(self, x, drop_rate):
        # forward logic
        embedding = self.embedding(x)
        encoding = self.rnn(embedding)
        if RNN_type == 'GRU2':
            encoding = self.rnn2(encoding)
        encoding = tf.nn.dropout(encoding, drop_rate)
        logits = self.dense(encoding)

        return logits
```

The accuracy and loss of training and testing data against the number of epochs for vanilla RNN layer can be observed from the graph below:

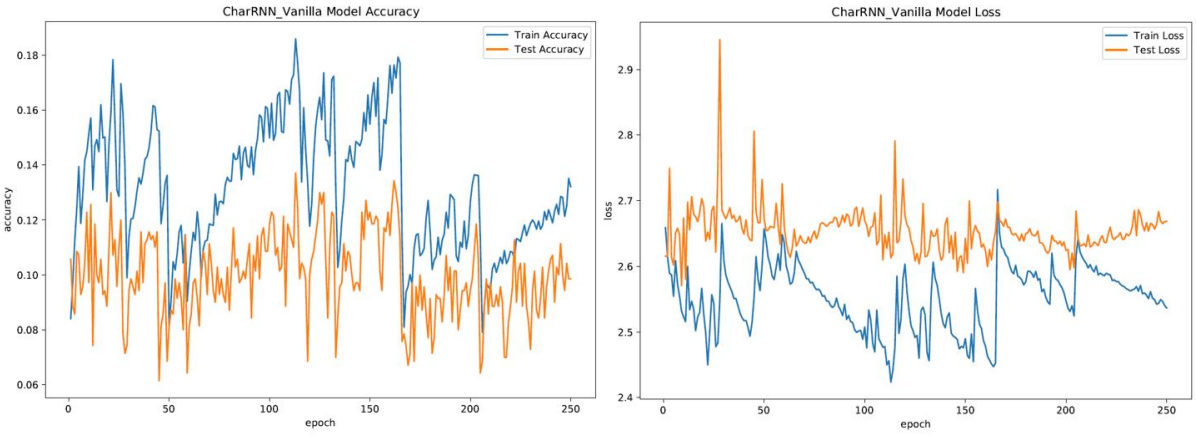


Figure 12: CharRNN Vanilla model performance

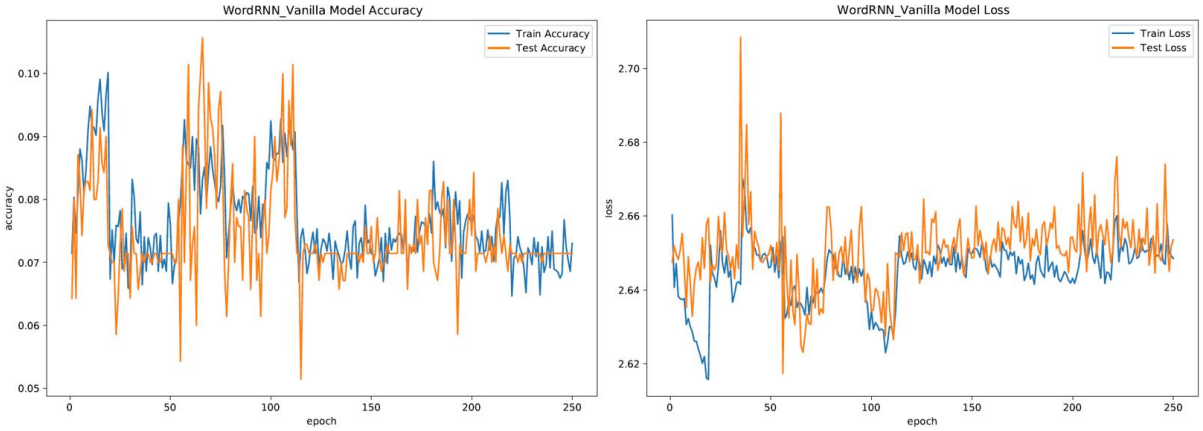


Figure 13: WordRNN Vanilla model performance

The accuracy and loss of training and testing data against the number of epochs for LSTM layer can be observed from the graph below:

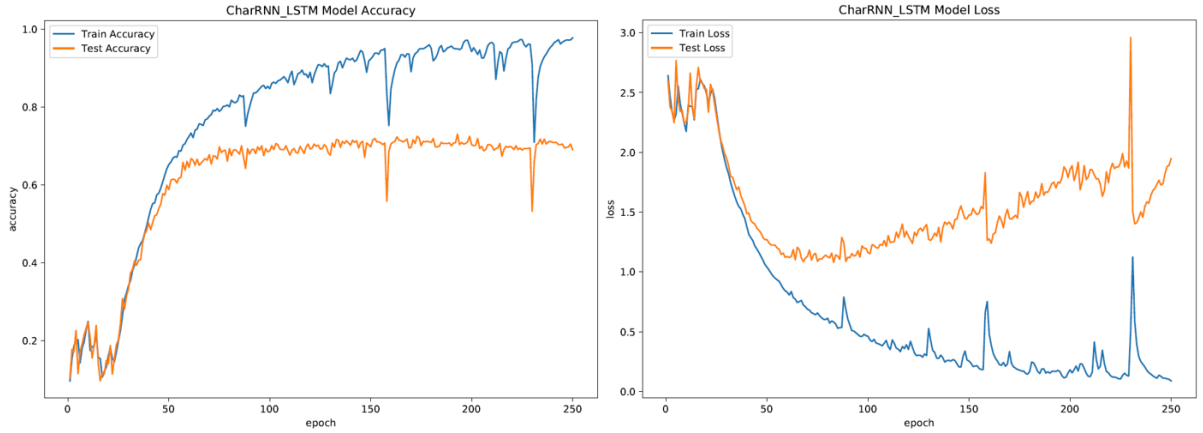


Figure 14: CharRNN LSTM model performance

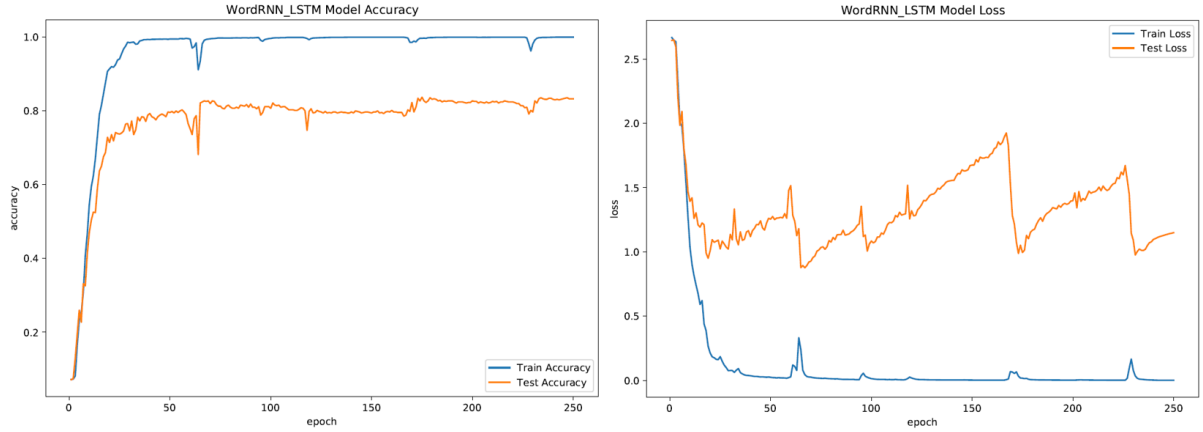


Figure 15: WordRNN LSTM model performance

The accuracy and loss of training and testing data against the number of epochs for 2-layer RNN model can be observed from the graph below:

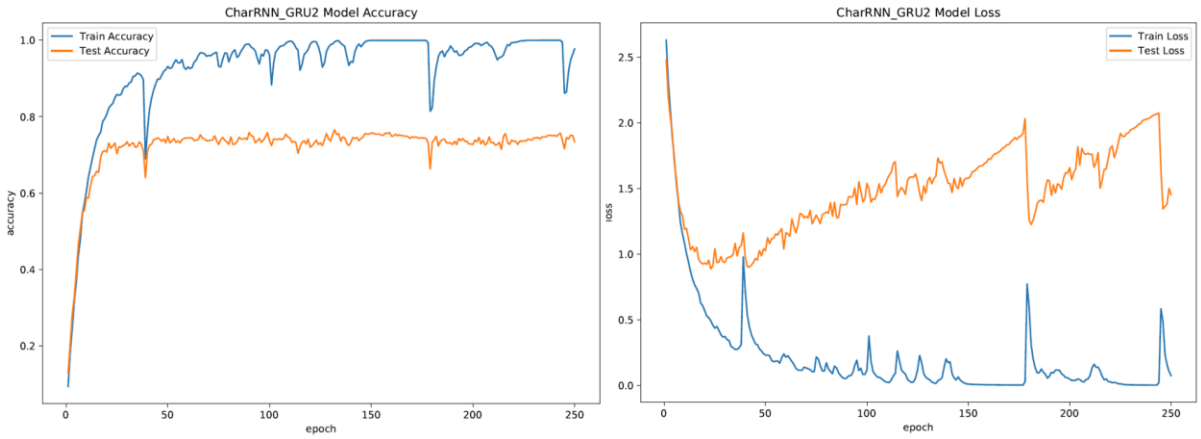


Figure 16: CharRNN 2 layer GRU model performance

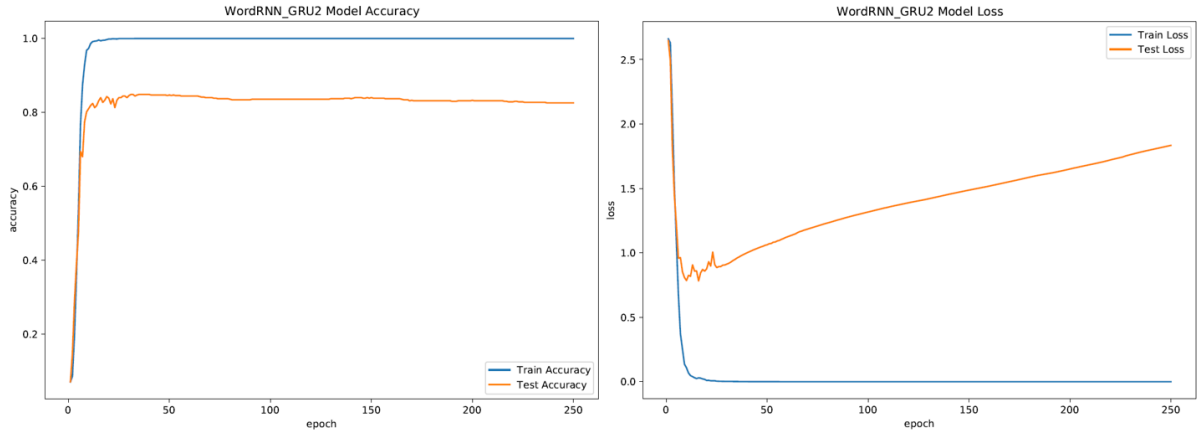


Figure 17: WordRNN 2 layer GRU model performance

The accuracy and loss of training and testing data against the number of epochs for RNN model with gradient clipping can be observed from the graph below:

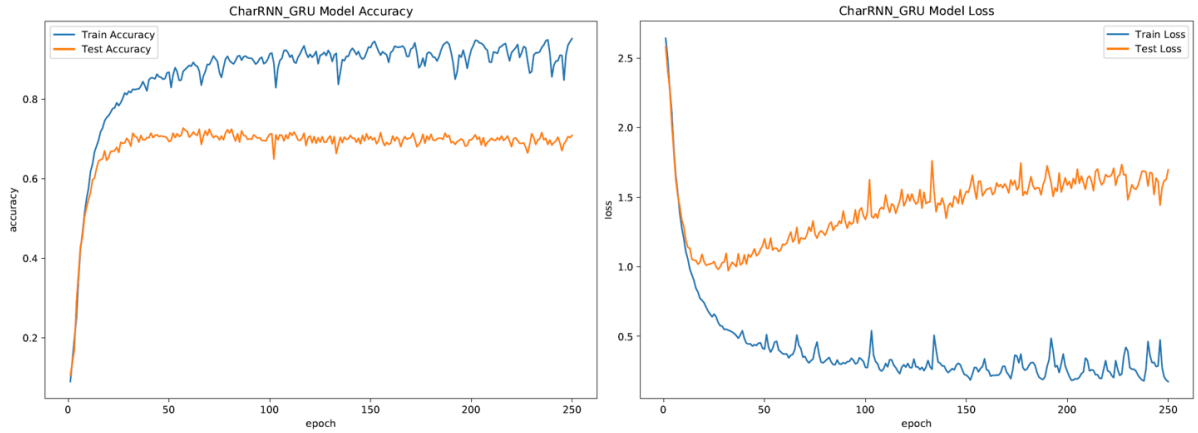


Figure 18: CharRNN GRU with clipping model performance

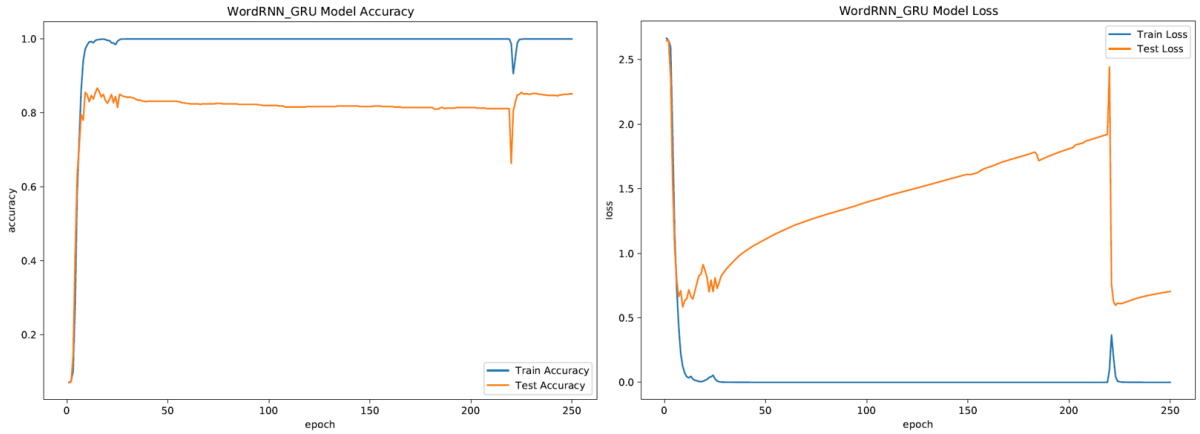


Figure 19: CharRNN GRU with clipping model performance

A comparison of test accuracies for all models in this question is shown below:

Figure 20: Test accuracies for all models in Q6

Model Name	**Accuracy (Last 100 epochs)**
CharRNN_GRU	0.6641999989748001
CharRNN_Vanilla	0.09538571417331695
CharRNN_GRU2	0.7404714238643646
CharRNN_LSTM	0.7002428567409515
CharRNN_GRU with clipping	0.6962285709381103
WordRNN_GRU	**0.8624571400880814**

WordRNN_Vanilla	0.07248571567237377
WordRNN_GRU2	0.83092857837677
WordRNN_LSTM	0.7334428542852401
WordRNN_GRU with clipping	0.8225571376085281

Table 3: Test accuracies for all models in Q6

From data gathered in Table 3 and Figures 12-20, the following is observed:

It is apparent that the Vanilla RNN models for both character and word processing performed the worst (Not even able to achieve 0.1 accuracy). This may be due to the gradient vanishing problem that is prevalent in language processing tasks. Since Vanilla RNN models have no modifications, it is expected that it will have the worst performance out of all the candidates in this question.

For the variations of CharRNN models, the two-layer GRU model performed the best, followed by LSTM, then gradient clipping and the original GRU. This shows that the dataset used for CharRNN models is sufficiently complex to benefit from increasing the number of layers. Gradient clipping does not seem to have much of an effect on the overall accuracy of the model, suggesting that the cases of gradient explosion is relatively small during training. It is not surprising that the LSTM model outperformed the GRU model since it uses more parameters to make predictions.

For the variations of WordRNN models, the original GRU model remains to be the best performing out of all the candidates. This means that all other modifications to the current model worsened the model's performance. This suggests that adding a second GRU layer or using a slightly more complex model (LSTM) will cause the model to overfit.

Conclusion

From questions 1 through 6, we have found that the original WordRNN with one GRU layer results in the highest test accuracy out of all the models analysed. This shows that the parameters used to create and train the original WordRNN model is already near the optimal parameters. Thus, changing any of them significantly may lead to overfitting or underfitting and hinder the performance of the model.

From the data obtained above, we can also observe the following:

- RNN models are able to converge much faster than CNN models, at the cost of higher computational time.
- CNN is more suitable for processing image data, and RNN is more suitable for processing sequential data.
- The Vanilla RNN model with no modifiers is not a very good model for processing textual data (significantly outperformed all models , including: GRU, LSTM, two layered GRU).
- Adding dropout to models have the potentially of significantly improving the model performance given that the model is overfitting the data, otherwise it may slightly hinder performance instead.