Tools of Intelligent Systems Summer 2020 Assignment #3

Hamza Mahdi
Electrical and Computer Engineering
hmahdi@uwaterloo.ca

Abstract

This report addresses questions 1 - 3 of the third homework in ECE 657. Tables, figures or code related to each question are presented along with some discussion. All networks were trained using the Keras API with TensorFlow backend.

Index Terms

Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Long Sort Term Memory (LSTM), Gated Recurrent Unit (GRU), Word Embeddings, Natural Language Processing (NLP)

I. Question 1

In this question, the CIFAR10 dataset is utilized.The dataset consists of RGB images belonging to 10 unique labels.

In terms of preprocessing, the dataset was mostly unaltered with with exception of normalizing data to the range [0,1]. Additionally, images were converted to grayscale for the MLP network in order to reduce the number of inputs. The reason behind doing little pre-processing is because the images are already small in terms of pixels and the availability of GPUs through google Colab made it easy to process the 32x32 pixel images relatively fast. If this was not the case, I would have likely used PCA to reduce the dimensionality of the dataset. Additionally, no filtering was needed as the NN weights will act to extract features from images.

For the output layer, I used the softmax activation function since this is a multi-class problem with mutually exclusive classes. The softmax works well in this case because it outputs a normalized probability distribution as opposed to the sigmoid function where the provavilities do not necessarily sum up to 1 [1]. Eq. 1 shows the softmax activation function where z is a vector of distribution scores.

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{1}$$

To complement the softmax activation function, cross-entropy is used as a loss function. Cross entropy is useful because it utilizes log probabilities as shown in the formula below [1]:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{2}$$

Where L is the cross entropy loss. Specifically, I used categorical cross-entropy in the keras library which takes into account the multi-class nature of this problem.

A. MLP

In this subsection, I present plots for training and validation accuracies/losses. Discussion will be presented in later subsections.

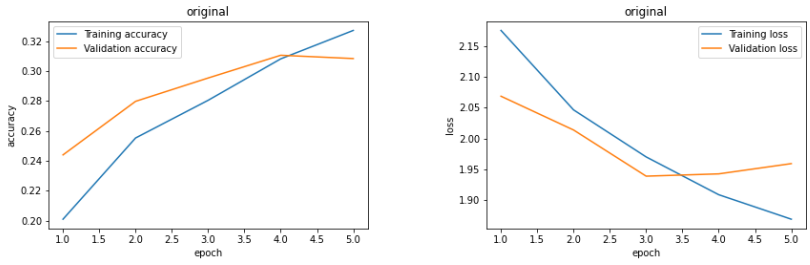


Fig. 1: Training and Validation Accuracies/Losses Over Training Epochs for the MLP Network Requested in Question 1 (No Changes to Architecture)

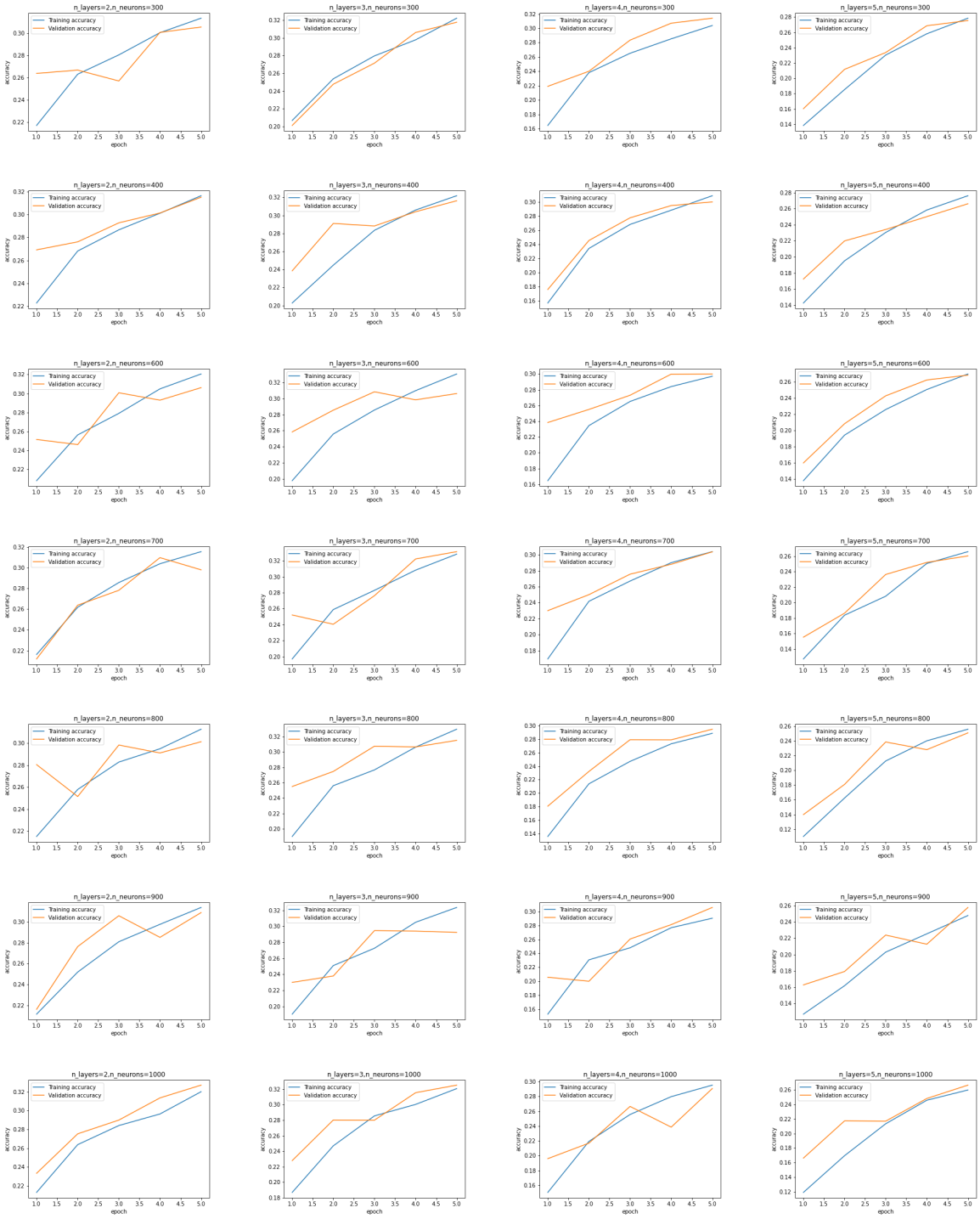


Fig. 2: Training and Validation Accuracy for MLP with Hyperparameter Tuning

Note on varying the number of nodes and layers: As requested I changed the number of layers and the number of neurons per layer in the MLP and the results are shown in Figs. 2 and 3. It is important to note that we cant conclude anything about the benefits/drawbacks of a particular architecture without performing cross-validation as results may vary from something as simple as the randomly initiated weights. That being said, I observed a trend of the accuracy being lower and loss being higher for the MLP network with 5 layers regardless of the number of nodes involved. This could be attributed to an onset of vanishing or exploding gradient problem (was not observed in this case but if more layers were used it might have been

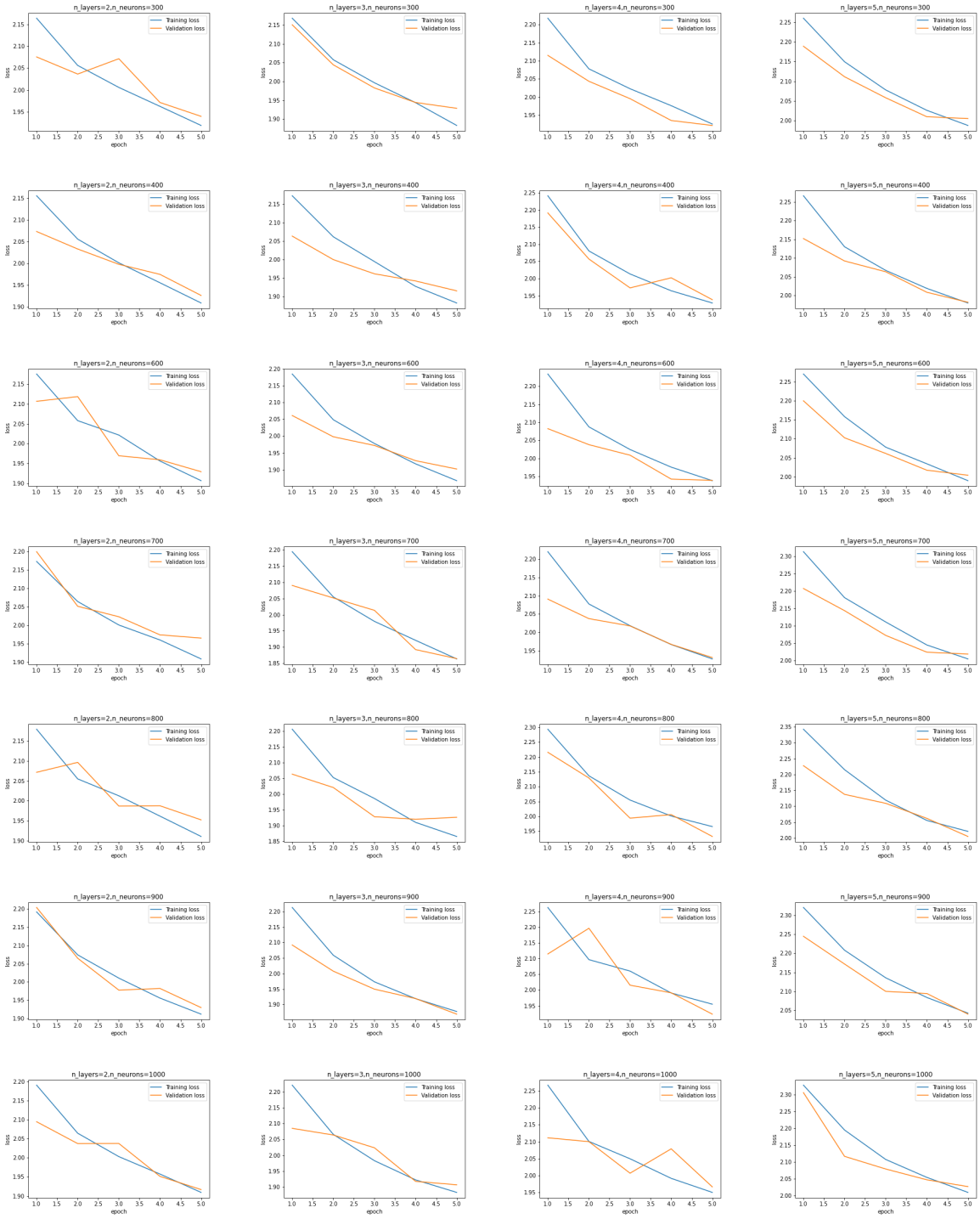


Fig. 3: Training and Validation Loss for MLP with Hyperparameter Tuning

observed). As for varying the number of nodes, it did not seem to have an impact on accuracy or loss. I expected the network to overfit when the number of nodes was increased but was not able to support my hypothesis.

B. MLP vs. CNN

As seen in Table I, the testing accuracy is relatively low and even worse than random in the MLP case. This can be attributed to the exceptionally low number of samples compared to the dimensionality of the data. The CNNs perform much better than MLP which can be attributed to their ability to deal with "spacial" data and extract features present locally in an image. On

TABLE I: Training and Testing Accuracy of the Three Models from Question 1

Network	Train Accuracy	Test Accuracy
MLP	0.3272	0.3050
CNN1	0.9307	0.5480
CNN2	0.5705	0.5466

the other hand, MLP does not preserve "spacial" interaction as well as CNN which is why it results in much lower accuracies.

C. CNN1 vs. CNN2

By examining Figs. 4 and 5, we can see that CNN2 performs much better in terms of avoiding overfitting. This can be attributed to the dropout layer which ensures that the network is not memorizing the training dataset in each epoch. Additionally, CNN1 takes more than double the time it takes CNN2 to complete an epoch (5 seconds for CNN1 vs 2 seconds for CNN2). This can be attributed to the max pooling layers which drastically reduce the dimensionality of data and allows for the network to train much faster.

Even though CNN2 has less trainable parameters than CNN1, it performs just as well on the validation and testing sets which is a big advantage especially when considering that CNN1 has more than 25 million trainable parameters while CNN2 has a little less than 1.5 million trainable parameters.

If the networks are trained for more epochs, CNN1 will reach a training accuracy of 100% fairly quickly and the performance of the model on the validation and testing sets might deteriorate. CNN2 will be more robust to overfitting and might result in better accuracy and lower training/validation losses.

In terms of improvements, I would not use CNN1 and instead improve on CNN2. Adding more convolutional layers might help separate high level and low level features in the image and could result in better accuracy as richer features are extracted from the image. State of the art networks such as LeNet and resnet utilize much deeper convolutional architecture with some layer skips. While the use of such netowrks could certainly help, it is very important to use more data than what was used in this exercise. Each image comes with 32x32x3 "features" which requires many training samples to achieve a practical accuracy and avoid "the curse of dimensionality".

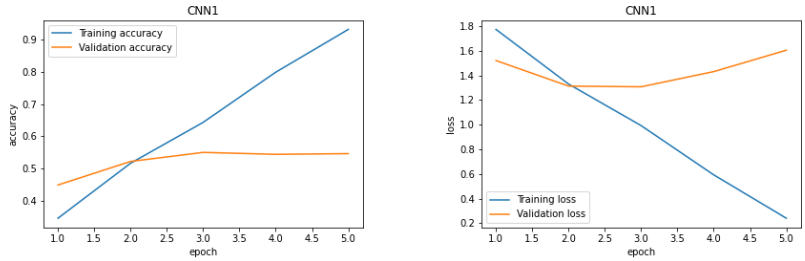


Fig. 4: Training and Validation Accuracies/Losses Over Training Epochs for the CNN1 Network Requested in Question 1

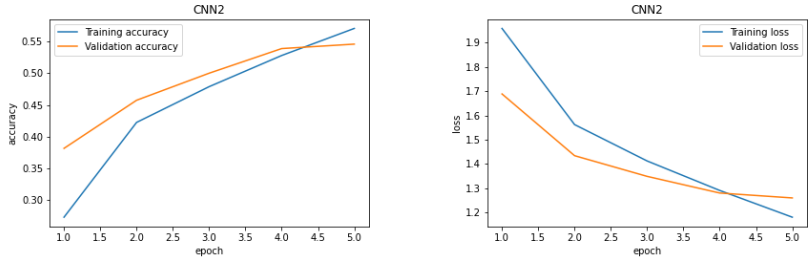


Fig. 5: Training and Validation Accuracies/Losses Over Training Epochs for the CNN2 Network Requested in Question 1

II. Question 2

A. Dataset Creation

The dataset was created by grouping sets of three days with the opening price from the next day. It is important to note that the original csv was ordered newest to oldest, and to create the data properly, the data had to be ordered by date. Once the data was in the correct order, creating the dataset was as easy as using two nested for loops. Data was then randomly split and saved to two csv files as requested in the question.

B. Preprocessing

When data is loaded into the script, a simple, but essential, preprocessing step is used; data is standardized so that each feature has a mean of zero and a standard deviation of 1. This mitigates any effects of different variables having a different impact on the network training, namely, the volume variable (its values are much higher than the other variables concerned with the stock price). Additionally, the labels are normalized to produce a normalized loss when training the network. This results in a more interpretable loss which ultimately is important for properly training the network.

C. Network Design

Since the number of days and which features to use were already given in the question, it made designing the network simpler. Additionally, since this question is concerned with RNNs, only variations of RNNs were explored, namely, traditional simple RNNs, LSTMs and GRUs. Additionally, the number of reccurent units, the nodes in the fully connected layers following the recurrent layers and the number of layers was varied to obtain the best combination for the design. This was done as I had no previous experience or intuition regarding RNNs.

To understand the difference between traditional RNNs, LSTMs and GRUs, I consulted the literature. Chung et al. empirically evaluated the different types of RNNs and found that gated networks such as LSTMs and GRUs have comparable performance, and work better than traditional RNNs [2]. They stated that the most important difference between traditional RNNs and those with gated units is the additive nature of the units in LSTM and GRU. On the other hand, traditional RNNs with activation units such as tanh, replaces (overrides) old values in units with a new one that depends on the current input and the previous state [2]. GRU and LSTM differ in terms of complexity of gating where LSTM has a more complicated structure [3].

A total of 60 different options were explored as shown in Tables II and III and in Figs. 6, 7, 8. I noticed the best results at 128 hidden nodes in the fully connected layer. However, given that the input dimensionality is small, I decided to go with a GRU RNN layer with 64 units and 32 hidden nodes in the fully connected layer.

TABLE II: Final Normalized Training Loss for Hyperparameter Tuning of the RNN Network

# RNN units	8					16					32					64				
# Hidden Nodes	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
Simple RNN	6.58E-04	3.38E-04	2.48E-04	2.02E-04	2.46E-04	3.80E-04	2.99E-04	1.70E-04	1.60E-04	1.43E-04	2.57E-04	2.44E-04	1.65E-04	1.28E-04	1.25E-04	1.21E-04	1.53E-04	1.22E-04	8.64E-05	9.41E-05
GRU	2.07E-04	1.97E-04	1.87E-04	1.20E-04	1.47E-04	2.04E-04	1.84E-04	1.81E-04	1.44E-04	1.39E-04	1.90E-04	1.34E-04	1.23E-04	1.55E-04	1.02E-04	1.20E-04	1.34E-04	1.43E-04	1.16E-04	1.10E-04
LSTM	1.80E-04	2.52E-04	2.39E-04	1.59E-04	1.56E-04	1.98E-04	1.55E-04	1.48E-04	1.53E-04	1.45E-04	1.64E-04	1.26E-04	1.44E-04	1.26E-04	1.28E-04	1.31E-04	1.28E-04	1.29E-04	1.29E-04	1.23E-04

TABLE III: Final Normalized Validation Loss for Hyperparameter Tuning of the RNN Network

# RNN units	8					16					32					64				
# Hidden Nodes	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
Simple RNN	4.79E-04	3.04E-04	6.30E-04	2.79E-04	3.39E-04	5.42E-04	3.79E-04	4.76E-04	3.63E-04	2.33E-04	7.21E-04	6.45E-04	5.28E-04	3.87E-04	2.64E-04	4.01E-04	3.21E-04	3.38E-04	2.47E-04	1.62E-04
GRU	4.04E-04	1.67E-04	2.56E-04	2.11E-04	1.98E-04	2.10E-04	1.64E-04	1.71E-04	1.60E-04	1.52E-04	1.62E-04	1.41E-04	1.88E-04	1.58E-04	1.42E-04	1.63E-04	1.72E-04	1.38E-04	1.51E-04	1.35E-04
LSTM	3.60E-04	3.30E-04	2.47E-04	4.43E-04	1.81E-04	2.48E-04	2.01E-04	2.28E-04	2.00E-04	1.72E-04	2.40E-04	1.92E-04	1.83E-04	1.67E-04	1.90E-04	2.20E-04	1.86E-04	1.60E-04	1.74E-04	1.69E-04

D. Architecture of Final Network:

The $model.summary()$ function was used to display the network architecture shown in Fig. 9. While the figure shows most of the required information, it does not include the number of epochs (93), the batch size (256), the loss function (MSE) and the optimizer (adam). It's worth noting that the batch size was selected to make use of the exceptionally powerful GPUs available on google colab. It made the training noticeably faster.

E. Output of Training Loops

See Fig. 10

F. Output from Testing

The final testing loss (normalized) is: **0.000204**. Please see Fig. 11

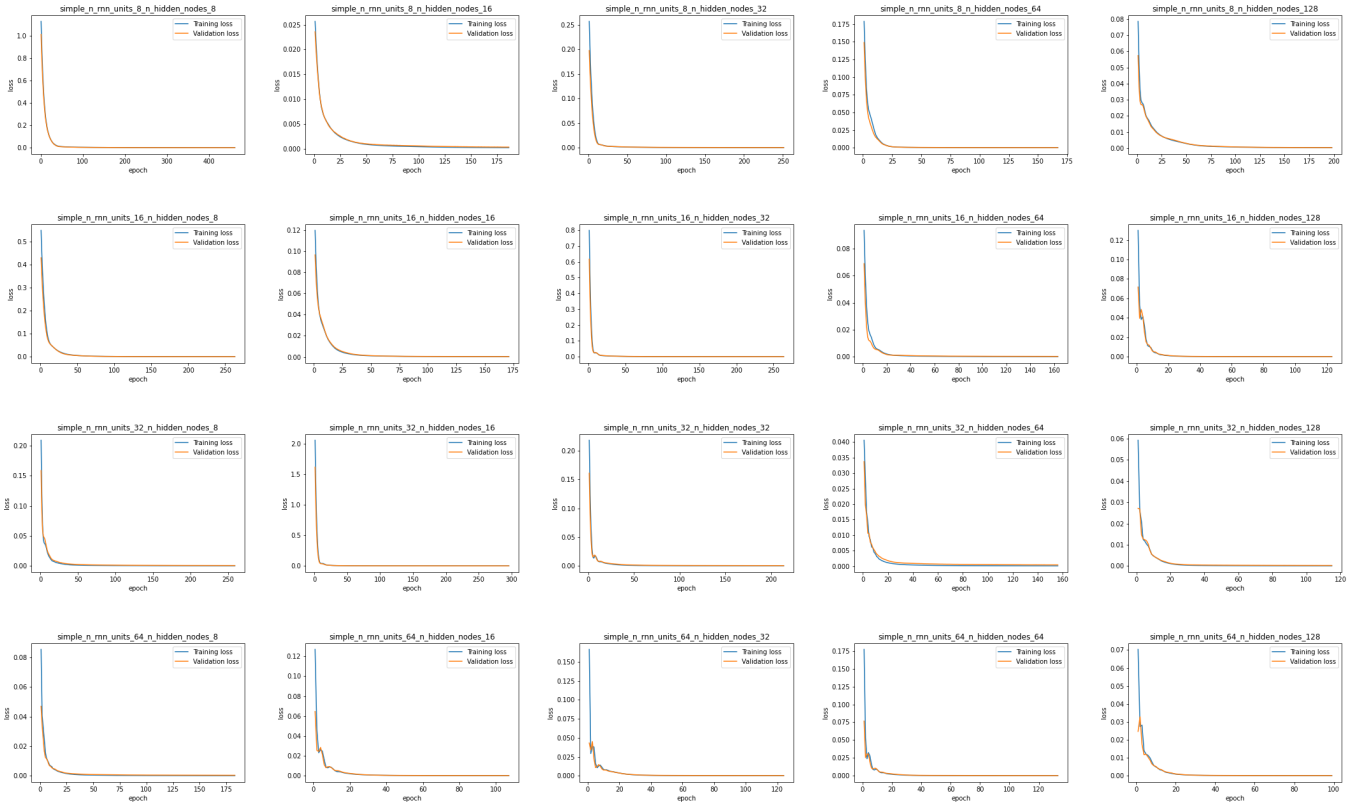


Fig. 6: Training and Validation Loss for SimpleRNN with Hyperparameter Tuning

G. Result of Using More Days for Features

I did not observe any significant difference when increasing the days to 6, 12 or 30. However, running this without cross validation would not give any reliable results to begin with. I did observe that the network took longer to converge with more days. This could be due to the fact that only a few days are required to predict the next day's opening price, and the network was simply learning to ignore the older days when using more days for features.

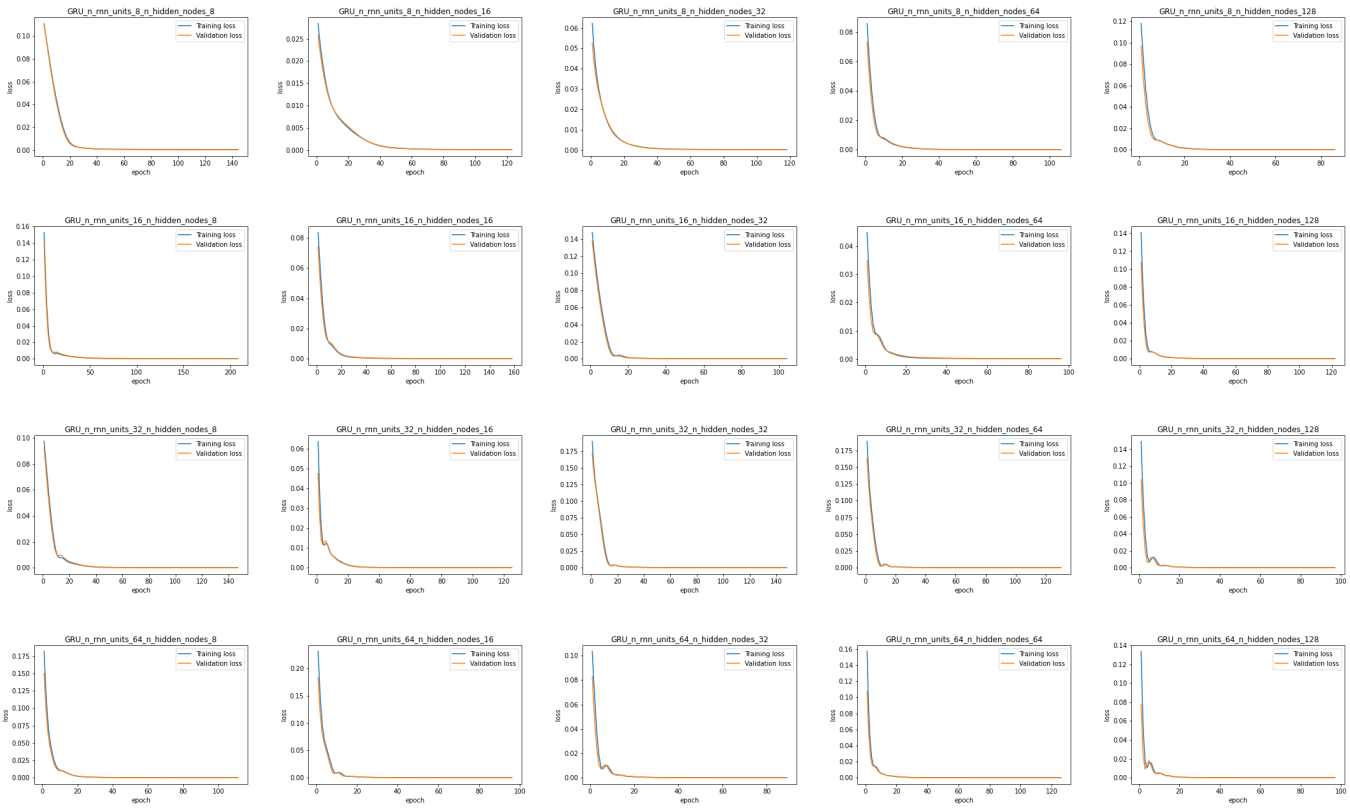


Fig. 7: Training and Validation Loss for GRU with Hyperparameter Tuning

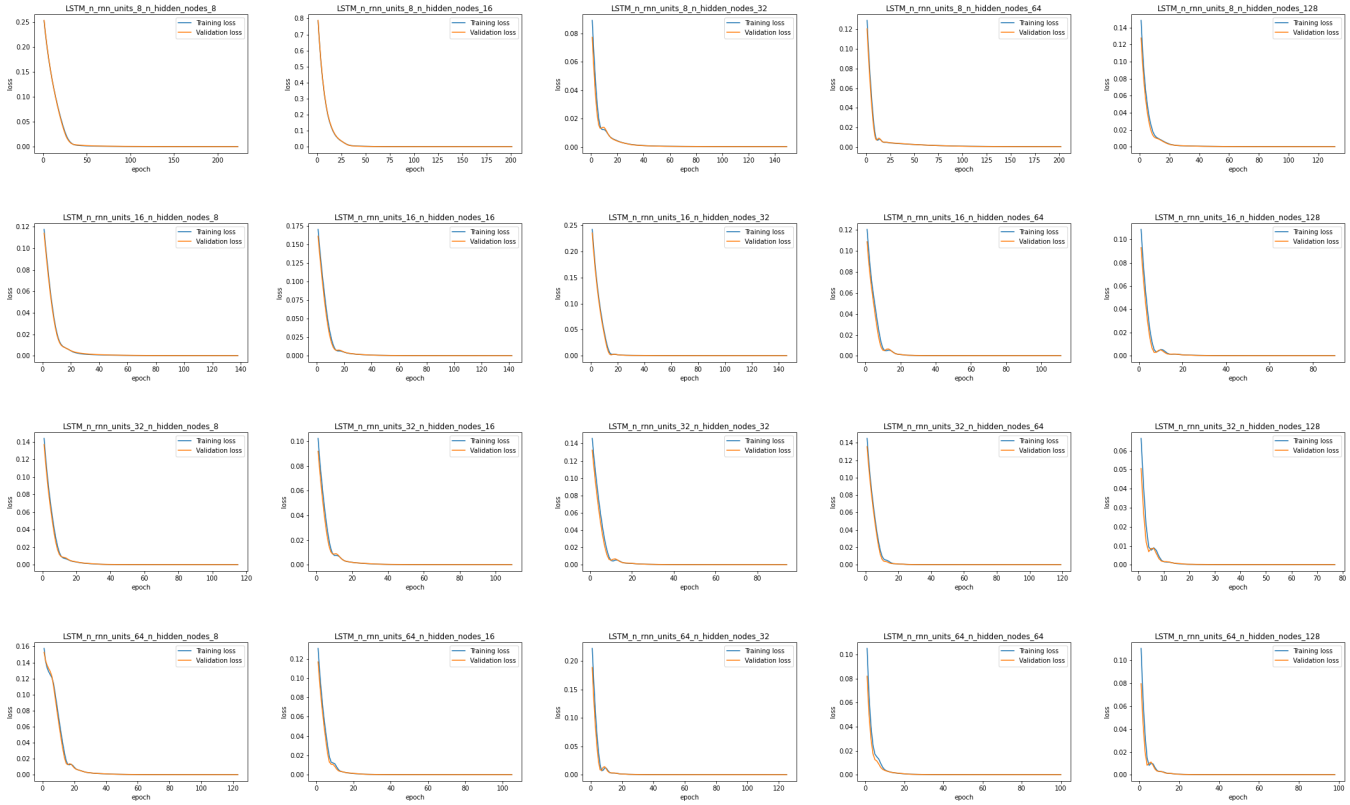


Fig. 8: Training and Validation Loss for LSTM with Hyperparameter Tuning

Layer (type)	Output Shape	Param #
gru_2 (GRU)	(None, 64)	13248
dense_7 (Dense)	(None, 32)	2080
dense_8 (Dense)	(None, 1)	33

Total params: 15,361
Trainable params: 15,361
Non-trainable params: 0

Fig. 9: Architecture of Final Network

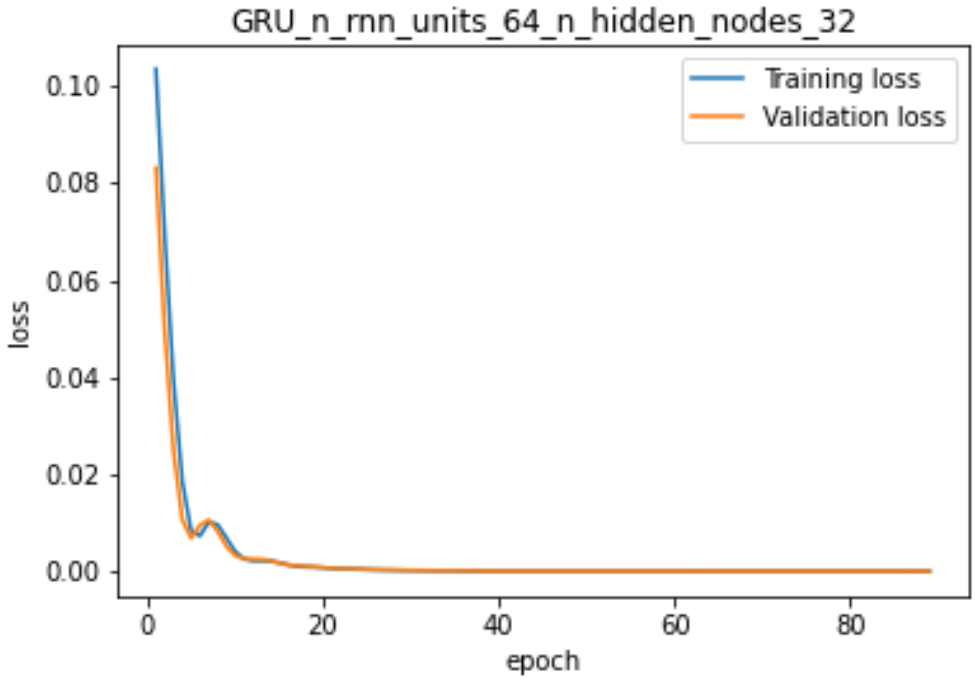


Fig. 10: Output of the Training Loop for the Final Network

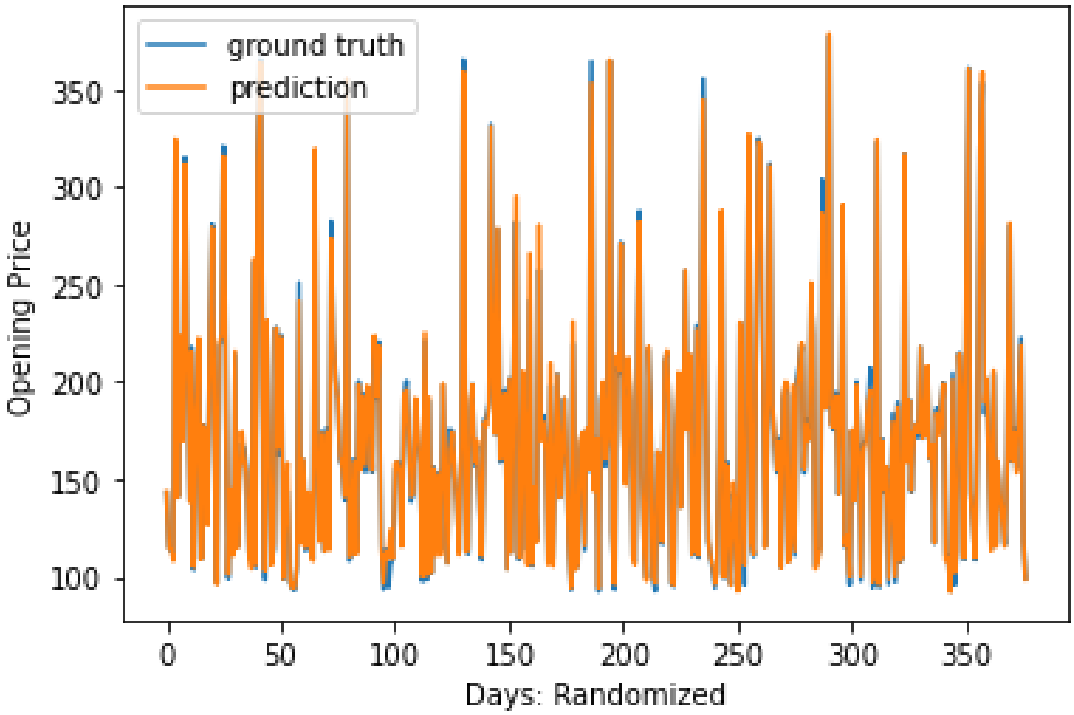


Fig. 11: Network Prediction for the Testing Set

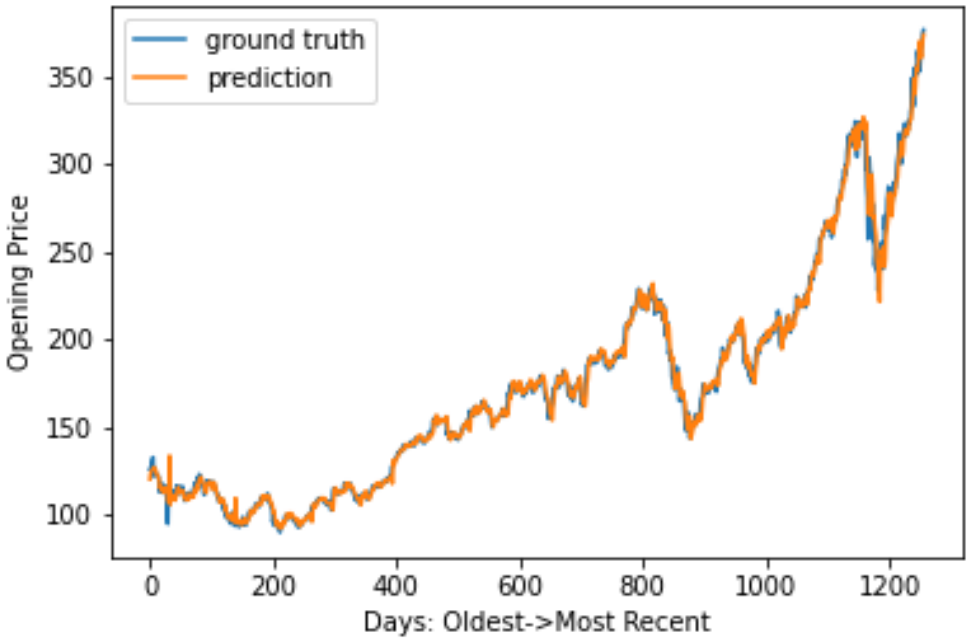


Fig. 12: Network Prediction for the Entire Dataset

III. Question 3

Considering that there are 50000 reviews, it was difficult to anticipate what "issues" might be present. For example, a reviewer could include a url in their review, introducing unwanted and meaningless text that could throw the prediction off. A problem that was apparent in all reviews was HTML artifacts such as $< br >$ which is present as a result of mining the IMDB website. Some other issues might be present such as vague wording and unique expressions.

Given that I am new to NLP, I consulted some online resources regarding what to remove/modify as a preprocessing step. A general consensus seemed that letters should be made into lower case, punctutaion should be removed, HTML tags and URLs should be discarded and stopwords can give better performance if removed [4]. Another source suggested that stopwords might carry some meaning such as the case of negation [5]. The argument was that if you have a negation such as "the product is not good", it would turn into "product good". This was a valid argument so I ended up comparing the classification accuracy in both cases (with and without stopwords), and I observed no changes in the final accuracy or loss. Thus, decided to keep stopwords as they could carry some meaning.

Additionally, I tokenized the words into numpy arrays using the keras tokenizer class. I passed a vocabulary size of 10000 to the tokenizer as suggested in the lecture. That being said I tried to increase the vocabulary size but that resulted in a longer tokenization process and no improvements in performance, so I kept it at 10000. Next, I padded the text to have a uniform size for all inputs. In the padding process, sentences that are too long, are cut down to the maximum length set by the user and sentences that are too short, are padded by zeros so that they are at the maximum length. After the padding process, preprocessing was complete, and the data was ready to be fed into the network.

Note: Given the pre-processing steps mentioned above, the embedding layer will have a size of (10000,300). Given that this is a large tensor, I had to reduce my batch size to 32 to run on my local computer. On colab, the network trained even with the larger batch size.

In terms of design, I tried network architectures that examine relationships between sequences of data. As this is not a property of MLP, I avoided that specific architecture. Two architectures standout: RNN and CNN; both architectures examine relationships between sequences but in slightly different mechanisms. RNN examines the "temporal" relationships between sequences, and even allow to examine past and future with relation to the present feature (bi-directional RNN) [6]. CNNs, explore the "spatial" relationships between data by performing convolution on sequences of data (determine by the kernel size). After deciding what types of architecture to explore, I decided what specific options to choose for each architecture. I tried SimpleRNN, LSTM and GRU for the RNN architecture. As for CNN, I tried 1D convolution as the input sequence is one dimensional. In retrospect, it might be possible to use both 1D and 2D convolution by perhaps reshaping the output of the embedding layer (the layer we are trying to learn).

I explored 5 models and based on their performance, decided on the final architecture. The performance for each model is shown in Fig. 14. Additionally, the "design" of each model is shown in the left most column of Fig. 14. All models performed similarly, however, I opted for the 1D convolution (model 5) as it seemed to perform better when repeating the training.

In terms of final training and testing accuracies, I obtained 94.49% and 88.63% respectively after 2 epochs. If the network trains for more epochs, the training accuracy eventually converges to 99% while the testing accuracy goes down. I originally had an automatic early stopping criteria that would save the model with the lowest validation loss. However, that always ended with a relatively low training accuracy even though it performed well on the validation set. To achieve a balance between training and validation accuracies, I manually set the epochs to 2 which resulted in a repeatable and acceptable training/validation losses.

Comment on the output: it seems the model was able to successfully learn the weights of the embedding layer as shown in Fig. 13,which is a TSNE visualization of the embedding layer. It is clear that data is separated into two clear groups with some "irrelevant" words in between. This visualization was obtained by uploading the weights to the Tensorflow Embeddings Projector.

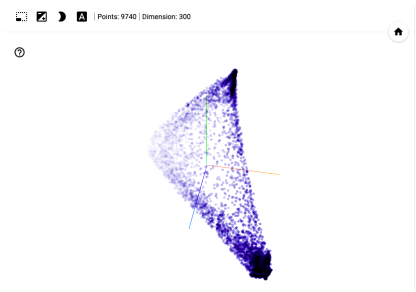
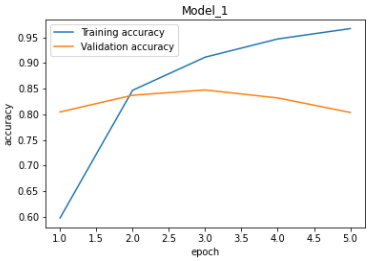
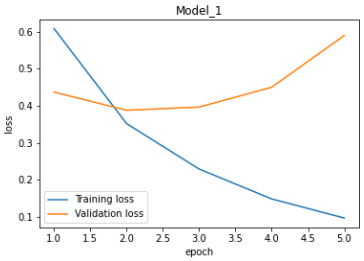


Fig. 13: TSNE Representation of the Learned Word Embeddings. Two Distinct Groups Can Be Seen

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 300)	3000000
bidirectional (Bidirectional	(None, 16)	19776
dense (Dense)	(None, 64)	1088
dense_1 (Dense)	(None, 1)	65

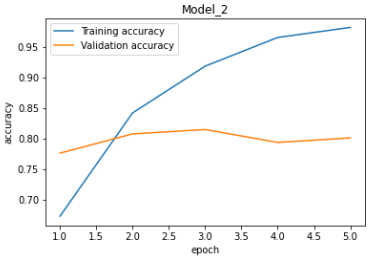
Total params: 3,020,929
Trainable params: 3,020,929
Non-trainable params: 0

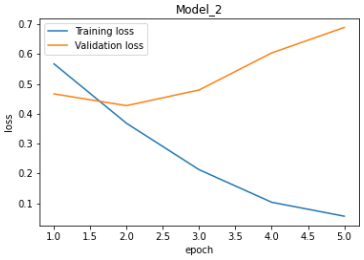




Layer (type)	Output Shape	Param #
embedding_8 (Embedding)	(None, None, 300)	3000000
bidirectional_1 (Bidirection	(None, 16)	4944
dense_6 (Dense)	(None, 64)	1088
dense_7 (Dense)	(None, 1)	65

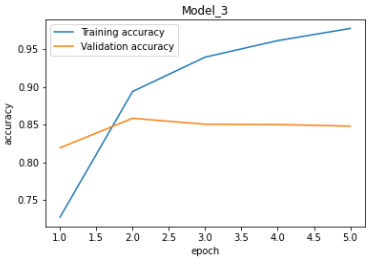
Total params: 3,006,097
Trainable params: 3,006,097
Non-trainable params: 0

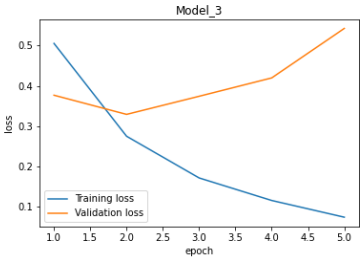




Layer (type)	Output Shape	Param #
embedding_10 (Embedding)	(None, None, 300)	3000000
bidirectional_4 (Bidirection	(None, None, 128)	186880
bidirectional_5 (Bidirection	(None, 64)	41216
dense_10 (Dense)	(None, 64)	4160
dropout_1 (Dropout)	(None, 64)	0
dense_11 (Dense)	(None, 1)	65

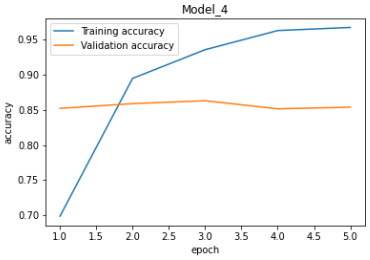
Total params: 3,232,321
Trainable params: 3,232,321
Non-trainable params: 0

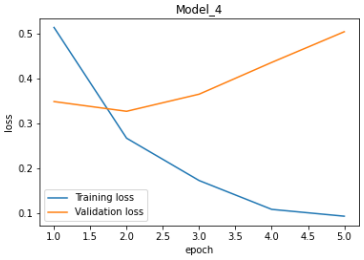




Layer (type)	Output Shape	Param #
embedding_11 (Embedding)	(None, None, 300)	3000000
bidirectional_6 (Bidirection	(None, None, 128)	140544
bidirectional_7 (Bidirection	(None, 64)	31104
dense_12 (Dense)	(None, 64)	4160
dropout_2 (Dropout)	(None, 64)	0
dense_13 (Dense)	(None, 1)	65

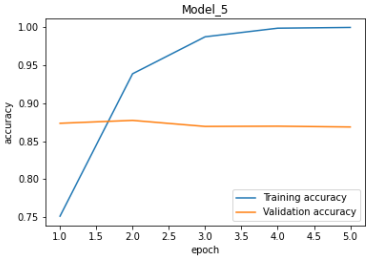
Total params: 3,175,873
Trainable params: 3,175,873
Non-trainable params: 0





Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 300, 300)	3000000
conv1d_1 (Conv1D)	(None, 300, 32)	28832
flatten_1 (Flatten)	(None, 9600)	0
dense_1 (Dense)	(None, 250)	2400250
dense_2 (Dense)	(None, 1)	251

Total params: 5,429,333
Trainable params: 5,429,333
Non-trainable params: 0



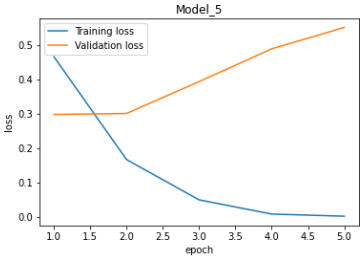


Fig. 14: Design, Training and Validation Loss for The 5 Models Explored in this Problem.
Note: Though not visible in the figures, models 1 and 3 use LSTM, model 2 uses SimpleRNN and model 4 uses GRU

REFERENCES

[1] (). Cs231n convolutional neural networks for visual recognition, Stanford University, [Online]. Available: https://cs231n.github.io/linear-classify/#softmax.

[2] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[3] D. Britz. (). Recurrent neural network tutorial, part 4 – implementing a gru/lstm rnn with python and theano, [Online]. Available: http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/.

[4] S. Kumar. (). Getting started with text preprocessing, [Online]. Available: https://www.kaggle.com/sudalairajkumar/getting-started-with-text-preprocessing.

[5] G. Singh. (). Why you should avoid removing stopwords, [Online]. Available: https://towardsdatascience.com/why-you-should-avoid-removing-stopwords-aa7a353d2a52.

[6] Tensorflow. (). Tf.keras.layers.bidirectional, [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional.