CSI 5340 – Deep Learning and Reinforcement Learning

Assignment 3

Student Name: Nathaniel Bowness

Student Number: 7869283

Due Date: November 23rd, 2023

This assignment asks students to develop two text classification models using Vanilla RNN and LSTM on the IMDB Movie Review Dataset. This requires designing two text classification models and tuning each model's hyper-parameters to obtain the best classification results on the testing data set. The assignment aims to help students understand the concepts behind the two models and how to perform basic text classification and data preprocessing using lecture topics.

Data Preprocessing:

For this assignment, the IMDB Movie Review Dataset was used to create a text classification model to determine if a review is positive or negative. I downloaded the dataset from the given link to a working directory to access the dataset. Once downloaded, I read the training and test datasets into in-memory arrays. I also parsed the naming conventions of the files to create arrays for the training and test labels as well.

After loading in the training and test dataset (movie reviews and corresponding labels), I used an existing tokenizer available in the "torchtext" library called "basic_english" to split each movie review into a list of words/punctuation known as tokens. The average length of each review, when tokenized, can be seen in Table 1 below. The average length was 268 and 265 for the training and test reviews. After further inspection of the tokenized reviews, I believe the tokenization could have been improved to also improve the model's accuracy. More details are available in the last section of this report.

Table 1: *The average length of the movie review dataset once it was tokenized.*

Movie Review Dataset	**Average Tokenized Sentence Length**
Training	268
Test	265

To have the training and test reviews represented as numbers for easier use in the model, I converted each "token" currently represented as a string and mapped it to a number, i.e., a numbered token. The numbers were assigned in order of appearance across the whole dataset to keep it simple. With the reviews now being mapped to a sequence of numbers, it is much easier to interact with for model training. As a sequence of numbers, the arrays were easier to pad and assign a value of 0 to. The maximum length of the reviews, and thus the padding required, was changed as a hyperparameter and tested with a few different values.

I tried to use the longest review as the original length and pad all the other reviews to that maximum length. The issue I found was that the longest review had over 2000 tokens. So, the results were very poor when all other reviews were padded to that length. As most reviews were approximately 250 real tokens, the 1800 padded ones. Therefore, variable-length reviews were tested to try to find the optimal results.

Pre-trained embedding vectors were used from the Glove dataset to help find similarities between the words in the reviews for training. I loaded the pre-trained embedding vectors into memory. I created a matrix that maps each numbered token representing a tokenized word from the initial review to return the embedding vector for easy use in the model later. As an example of the preprocessing from end to end, let's give an example of 1 review that mentions "I was here.". The tokenized array would come out to ["I", "was", "here", "."]. When mapping that array to numbers, it was converted into the following representation [1, 2, 3, 4]. If another array in the set was "I found him here," its number representation would be [1, 5, 6, 3].

Using the embedding matrix, I could easily get the embedding vector for each numbered token and train the RNN and LSTM models.

Shared Model Setup and Model Settings:

Some model architecture and settings were maintained consistently through the training to allow for a better comparison between models. The Vanilla RNN and the LSTM contained the same number of layers and followed the same training process. Both were of the following format:

1. Embedding layer: Used to match the inputted test to an embedding dimension
2. RNN Layer (Vanilla or LSTM): Input Dimension → Embedding Dimension, Output → Emb. Dim.
3. Dense Linear Sigmoid Activation Layer: Input → Embedding Dimension, Output → 1

With those 3 layers, both models were trained to be fairly accurate on the test data. Throughout the testing, the hyper-parameters found in Table 2 below were kept constant. This was to help reduce the number of variables and required rounds of testing. But it was also based on intuition that the batch size, learning rate and number of epochs similarly affect how often and how many times the weights are modified during training. Therefore, I believe modifying any of these settings while trying to find the ideal conditions would affect the training similarly and potentially contradict one another as the largest learning factor likely needs the least number of epochs.

Table 2: *Hyper-parameters that were chosen and remained constant across test runs.*

Hyper-Parameter	Values
Learning Rate	0.001
Batch Size	64

Hyper-Parameter Tuning:

The following hyper-parameters shown in Table 3 below were tested with various combinations of the values shown to find the best results for each RNN and LSTM model. The testing was not exhaustive of every possible combination but tried to capture patterns that improved accuracy. The goal was to increase the accuracy of the models as much as possible just by modifying the parameters for each state dimension. The embedding dimension was varied to see the impact on accuracy. However, not every size of an embedding vector from the Glove dataset was tried. I did try out various embedding vector dimensions and found higher dimensions of embedding vectors resulted in better accuracy. Therefore, the smallest and largest embedding vectors seemed appropriate to contrast.

Table 3: *Hyper-parameter values modified and tested throughout the model training to achieve the best accuracy.*

Hyper-Parameter	Values
Number of Epochs	[10, 15, 25]
State Dimension	[20, 50, 100, 200, 500]
Maximum Review Length in Tokens	[170, 250, 350, 500]
Embedding Dimension	[50, 300]

Vanilla RNN:

Using the model described above and the various hyper-parameters – some constant, some not. I built Table 3 with numerous results obtained from testing. The highest accuracy run for each state dimension is highlighted in blue. Generally, for each state dimension, the best runs had the highest embedding dimension, 15 epochs and an average review length of either 350 or 500.

Table 4: *Accuracy results for the Vanilla RNN models with various combinations of the hyper-parameters mentioned in Table 3. The highest accuracy for each state dimension is highlighted in blue.*

State Dimension	Embedding Dimension	Number Epochs	Average Review Length	Accuracy
20	50	10	170	79.04
20	50	15	170	80.7
20	50	25	170	80.82
20	300	15	170	82.96
20	300	15	250	84.29
20	300	15	350	86.15
20	**300**	**15**	**500**	**86.72**
50	50	10	170	81.50
50	50	15	170	81.66
50	50	25	170	81.44
50	300	15	170	83.92
50	300	15	250	86.25
50	**300**	**15**	**350**	**87.61**
50	300	15	500	87.40
100	50	10	170	82.30
100	50	15	170	82.07
100	50	25	170	81.96
100	300	15	170	84.54
100	300	15	250	86.38
100	300	15	350	86.41
100	**300**	**15**	**500**	**88.30**
200	50	10	170	82.93
200	50	15	170	83.11
200	50	25	170	82.24
200	300	15	170	84.54
200	300	15	250	87.02
200	300	15	350	86.65
200	**300**	**15**	**500**	**87.72**
500	50	10	170	82.18
500	50	15	170	76.67
500	50	25	170	80.95
500	300	15	170	84.44
500	300	15	250	86.95
500	**300**	**15**	**350**	**87.75**
500	300	15	500	87.74

Once the models were trained past 15 epochs, the results overall seemed to decrease. In the table, we see that at 25 epochs, the higher state dimensions perform worse, but the lower state dimensions perform better, showing it's possible they need more training to get accurate weights. The table shows that the optimal tokenized review length is likely somewhere between 350 and 500 since the best results come almost equally from each section. A review length of 450, for example, may have been ideal.

The best and worst hyper-parameter combinations for Vanilla RNN model accuracy can be seen below in Figure 1. This figure shows the importance of trialling and testing different hyper-parameter combinations, as almost a 6% increase in accuracy was achieved in the model just by modifying those parameters. The training time difference between the best and worst model configurations was 70% more for the accurate model configuration, which is a trade-off.

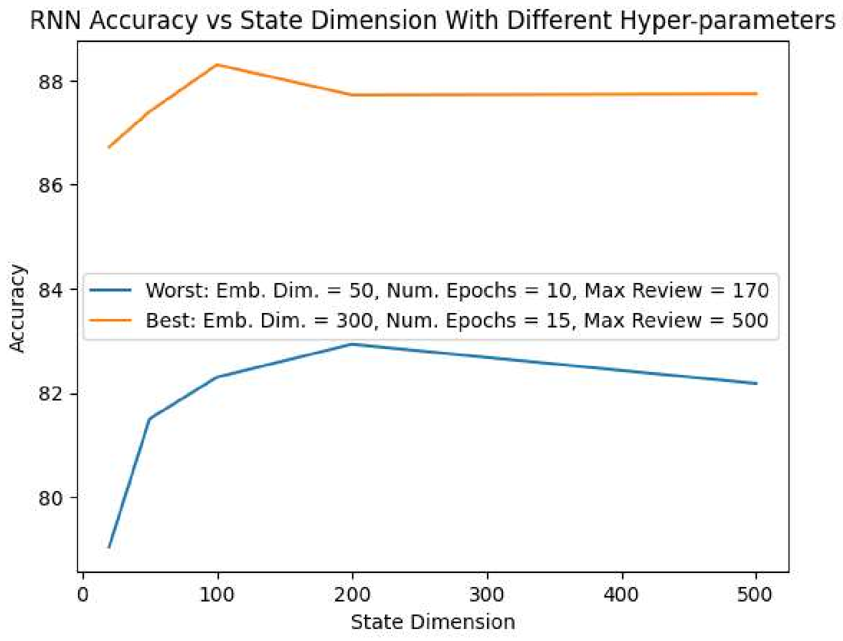

Figure 1: *Best and Worst hyper-parameters combinations for Vanilla RNN model accuracy.*

LSTM:

The Long Short-Term Memory (LSTM) is an RNN that is very good at capturing long-term dependencies. Using the model architecture described above and the various hyper-parameters – some constant, some not. I built Table 4 with numerous results obtained from testing. The highest accuracy run for each state dimension is highlighted in blue. Like the Vanilla RNN model, the best results were with an average tokenized review length of 350 to 500 tokens and 15 epochs. The LSTM did not have as much performance degradation at 25 epochs, but the accuracy was still likely due to overfitting.

Table 5: *Accuracy results for the LSTM model with various combinations of the hyper-parameters mentioned in Table 3. The highest accuracy for each state dimension is highlighted in blue.*

State Dimension	**Embedding Dimension**	**Number Epochs**	**Average Review Length**	**Accuracy**
20	50	10	170	77.62
20	50	15	170	83.00
20	50	25	170	82.83
20	300	15	170	83.36
20	300	15	250	87.10
20	300	15	350	88.10
20	**300**	**15**	**500**	**88.83**
50	50	10	170	80.70
50	50	15	170	81.38
50	50	25	170	83.84
50	300	15	170	85.37
50	300	15	250	87.64
50	**300**	**15**	**350**	**88.95**
50	300	15	500	88.82
100	50	10	170	81.44
100	50	15	170	83.99
100	50	25	170	83.36
100	300	15	170	85.98
100	300	15	250	87.61
100	300	15	350	89.14
100	**300**	**15**	**500**	**89.81**
200	50	10	170	82.04
200	50	15	170	83.76
200	50	25	170	83.36
200	300	15	170	85.82
200	300	15	250	87.94
200	300	15	350	89.26
200	**300**	**15**	**500**	**89.51**
500	50	10	170	81.89
500	50	15	170	84.30
500	50	25	170	82.38
500	300	15	170	86.73
500	300	15	250	88.52
500	300	15	350	89.67
500	**300**	**15**	**500**	**90.30**

Once again, it appears that the idea tokenized review length would be between 350 and 500 as those two parameters always result in the best accuracy. It seems like the performance overall was better for LSTM as well, which is expected.

The best and worst hyper-parameter combinations for LSTM model accuracy can be seen below in Figure 2. Even more than the Vanilla RNN, LSTM benefitted from finding the ideal hyperparameters. The difference between the best and worst combinations was over a 10% increase in accuracy for some state dimensions. However, the training time difference for the best LSTM configuration was 250% more than the worst LSTM configuration, so the additional training time was significant on this smaller dataset on the same hardware.

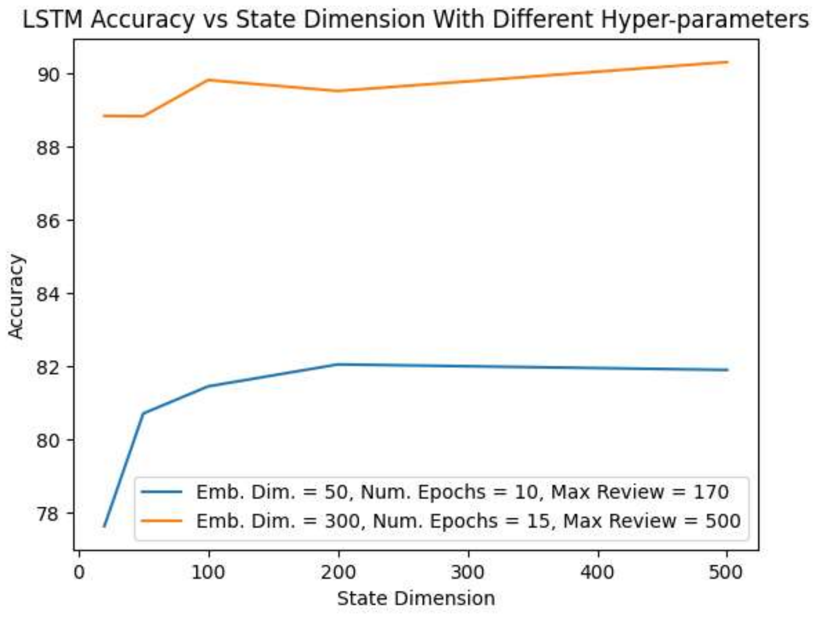

Figure 2: *Best and Worst hyper-parameters combinations for Vanilla RNN model accuracy.*

Comparison of Vanilla RNN and LSTM:

A comparison of Vanilla RNN and LSTM models can be seen in Figure 3 below. The figure shows the top accuracy of both models at each state dimension, which is also reported in Tables 4 and 5. The maximum accuracy for each state has almost all the same hyper-parameters shown in the list below:

- Embedding dimension → 300,
- Number of epochs →
- Learning Rate → 0.001
- Batch size → 64

The only hyper-parameter that differed between the results with a top accuracy was the tokenized review length of 350 and 500.

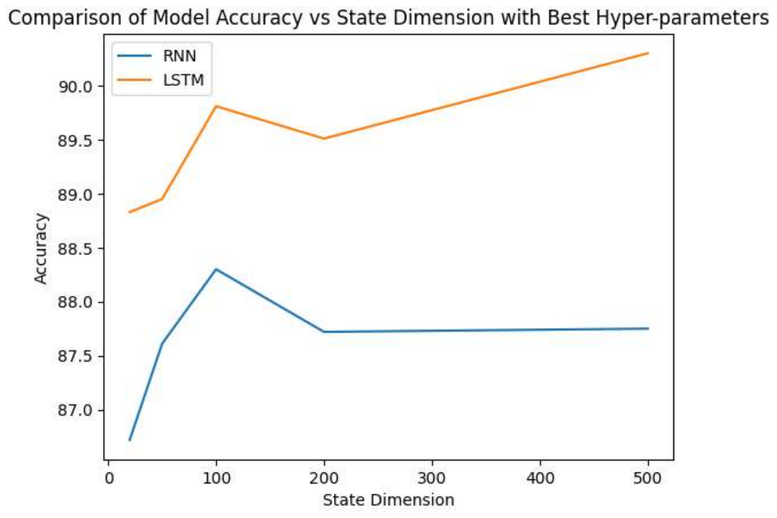

***Figure 3:** Accuracy versus state dimension for LSTM and Vanilla RNN models for text classification. The models shown use the hyperparameter settings that achieved the highest accuracy overall from the above sections.*

The figure shows that LSTM performs better overall than Vanilla RNN, as expected from the literature. From the graph, it's apparent that LSTM does much better at larger state dimensions. This is apparent as the models transition from a state dimension of 100 to 500. The LSTM model continues to increase in accuracy, whereas the Vanilla RNN accuracy slightly decreases. The Vanilla RNN's decrease in accuracy can be attributed to the "gradient vanishing problem" in Vanilla RNNs – that LSTMs were created to help fix.

I was surprised to see both models dip in accuracy overall from 100 to 200, but the LSTM accuracy did improve again, whereas the Vanilla RNN did not. It's possible the RNN accuracy was less because of some overfitting on the model. Seeing the LSTM model grow in accuracy as the state dimension increases was very interesting. I tried out state dimensions of 750 with a longer sequence of tokenized reviews (750) and saw the accuracy improve again, but it was not enough to make the additional training time worth it. Overall, this shows that the LSTM does correct the one flaw often attributed to RNNs in the gradient vanishing issue.

Overall, I think that the graph does a great job showing the pros and cons of the Vanilla RNN and LSTM. Vanilla RNNs have a flaw large state but otherwise perform similarly to LSTM with a less complex system. LSTM, with the additional complexity, performs better at the higher state dimensions.

Possible Improvements:

While not implemented, for this assignment, I think using a tokenization algorithm like “Spacy” would have benefited the training and accuracy of the models. When using Spacy, the average review length of tokens was considerably longer, and it did a better job of breaking up complex language. An overview of the average token length can be seen in Table 6 below.

Table 6: *The average length of the movie review dataset once it was tokenized.*

Movie Review Dataset	Average Basic Torch Tokenized Sentence Length	Average Spacy Tokenize Sentence Length
Training	268	356
Test	265	289

When inspecting that data, using Spacy resulted in a larger number of tokens because it splits up joined words common in the IMDB movie dataset. Like “check-in,” “far-fetched,” etc., having words like that split into [“check”, “-”, “in”] I’m sure would have helped the algorithm. Some of these hyphenated words did not exist when investigating the Glove embedding entries. Thus, embeddings couldn’t be found. But if they were split using Spacy, it would help the model get more information from the review.

Another improvement that I think may have been beneficial would be adding dropout to the model architecture so that the model accuracy does not converge as quickly. This may have resulted in needing more epochs, but it would hopefully give better results.

One last improvement I would have liked to try was finding the ideal tokenized sentence length for the model. The options for 350 and 500 seemed right between the perfect tokenized sentence length for the best accuracy.