1 Introduction and Acknowledgements

This report discusses an experiment seeking to find the best settings for hyperparameters for Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) Network.

Models are built using data[1] provided by Kaggle user @arunava. The data is comprised of 50,000 movie reviews labelled good or bad. The challenge is to build a model which accurately predicts whether a review is positive or negative.

As a preprocessing step, the words are first converted into vectors using Glove's[2] pretrained word-vectors in the Wikipedia plus Gigaword 5 set.

Code for loading the data (*load_data.py*) was thankfully provided along with the dataset.

The code for loading the GloVE pretrained word embeddings and creating the embedding matrix (*preprocess_data.py*) is based on `https://github.com/keras-team/keras/blob/master/examples/pretrained_word_embeddings.py`.

The code for using an embedding layer follows the example from Keras' documentation `https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/`.

For both Vanilla RNN and LSTM, I systematically test different models by varying the state dimension, the batch size and the number of epochs. Then, I further refine the LSTM model by trying different learning rates and adding dropout.

2 Vanilla RNN

For this first test (*experiment_rnn.py*), the hyperparameters are a combination of:

```
NUM_EPOCHS = [3, 6, 10]
BATCH_SIZE = [128, 256]
STATE_DIM = [20, 50, 100, 200, 500]
```

See Appendix 1 for a table containing all results. The best performance appears to have been achieved with a state dimension of 100, batch size of 256 and 10 epochs.

State Dim	Batch	Epochs	Params	Train Error	Train Acc	Test Error	Test Acc	Runtime
20	128	6	4432708	0.6172	0.6644	0.6113	0.6813	428.7
50	128	6	4437298	0.5930	0.6791	0.5829	0.7024	621.3
100	128	6	4448948	0.5827	0.6849	0.5933	0.6706	1053.0
20	128	6	4432708	0.6078	0.6732	0.5907	0.6996	347.2
100	256	10	4448948	0.5593	0.7100	0.5678	0.7086	1489.2

Table 1: RNN models with best test performance

However, even in the best case we have only around 70% accuracy on the train/test set. Suspecting that perhaps 10 epochs were insufficient, I ran a second experiment with 15 epochs but it did not make much difference:

State Dim	Batch	Epochs	Params	Train Error	Train Acc	Test Error	Test Acc	Runtime
20	128	15	8864258	0.5764	0.6956	0.5526	0.7269	1765.4
50	128	15	8870348	0.5509	0.7152	0.5772	0.6901	17726.1
100	128	15	8884498	0.4885	0.7600	0.5716	0.7140	1819.4
200	128	15	8927798	0.3586	0.8366	0.7509	0.6379	1822.1
500	128	15	9177698	0.6932	0.5030	0.6932	0.5000	1833.0

Table 2: RNN models with 15 epochs

This second RNN experiment was performed using tensorflow-gpu on a computer with a GTX 1070 graphics card, and yet the training time is still excruciatingly slow.

At best, I obtained 70% error on train and test. With settings (200, 128, 15) the model reaches 83% on the train set, but fares worse on the test set and so is probably overfitting. Perhaps these medicocre results can be attributed to the vanishing gradient problem, since the LSTM models have significantly better performance.

3 LSTM

As in the previous section, this first experiment (*experiment_lstm.py*) trains LSTM models (embedding layer, lstm layer, dense layer) using different LSTM state dimensions, batch sizes and number of epochs. I chose to take combinations from:

```
NUM_EPOCHS = [1,2,3,4,5]
BATCH_SIZE = [64, 128, 256]
STATE_DIM = [20, 50, 100, 200]
```

This experiment was much too slow for my laptop's Intel i5 CPU, so I used a machine equipped with a Geforce GTX 1060 to run this series of tests. This experiment used a basic LSTM layer because the optimized version was causing an error I couldn't solve at the time (more in later experiments).

Because of the long running time, I decided to run each test for a smaller number of epochs. Also, this experiment crashed whenever the neuron parameter (the state dimension) for the LSTM layer was set to 500, so unfortunately that data is not available. I suspect that this crash was caused by exceeding the available memory on the graphics card. The table below lists the best models (see Appendix II for full table):

State Dim	Batch	Epochs	Params	Train Err	Train Acc	Test Err	Test Acc	Runtime
20	64	4	8869654	0.3828	0.8309	0.3606	0.8403	1881.4
100	64	4	8945494	0.3585	0.8431	0.3458	0.8526	1931.5
100	64	5	8945494	0.3280	0.8570	0.3401	0.8511	2324.2
100	128	5	8945494	0.3563	0.8419	0.3701	0.8336	1277.7
200	64	3	9112294	0.3710	0.8348	0.3453	0.8462	1598.0
200	64	4	9112294	0.3424	0.8504	0.3430	0.8455	1984.0
200	64	5	9112294	0.3141	0.8632	0.3223	0.8604	2371.3
200	256	5	9112294	0.4072	0.8131	0.3728	0.8339	848.0

Table 3: LSTM models with best test performance

The best performance (test accuracy 86%) in this run was achieved with settings (200, 64, 5).

The next table shows the effect of the batch size on training time and accurac:

State Dim	Batch	Epochs	Params	Train Err	Train Acc	Test Err	Test Acc	Runtime
200	64	3	9112294	0.3710	0.8348	0.3453	0.8462	1598.0
200	128	3	9112294	0.4757	0.7735	0.4411	0.7933	936.5
200	256	3	9112294	0.5647	0.7134	0.5496	0.7383	672.8
200	64	4	9112294	0.3424	0.8504	0.3430	0.8455	1984.0
200	128	4	9112294	0.3804	0.8278	0.3738	0.8363	1111.9
200	256	4	9112294	0.4792	0.7759	0.4170	0.8102	755.4
200	64	5	9112294	0.3141	0.8632	0.3223	0.8604	2371.3
200	128	5	9112294	0.3556	0.8444	0.4414	0.7852	1282.4
200	256	5	9112294	0.4072	0.8131	0.3728	0.8339	848.0

Table 4: LSTM varying batch size

I suspect tha using smaller batches does not take full advantage of the GPU's parallel processing power.

For the next experiment, I fixed the settings for the state dimension, batch size and number of epochs and tried a few different learning rates. I also resolved the problem with CuDNNLSTM: it turns out that CuDNNLSTM does not support the mask_zero feature of the embedding layer, which I had used for the other models. After removing this option, CuDNNLSTM works fine – and very quickly! Even after increasing the number of epochs to 25, the models still train faster than with simple LSTM. The table below shows the result of three different learning rate:

Learning Rate	State Dim	Batch	Epochs	Train Err	Train Acc	Test Err	Test Acc	Runtime
0.01	200	256	25	0.0520	0.9823	0.8744	0.83904	438.85
0.001	200	256	25	0.0731	0.9727	0.6378	0.83764	431.33
0.0001	200	256	25	0.4401	0.7958	0.4857	0.76332	433.65

Table 5: LSTM learning rate

Here, we finally see the model fit the training data with high accuracy. The test accuracy, however, is about the same as in the previous experiment. Perhaps early stopping has provided some benefit to the generalization power of the model.

Next, in an attempt to correct what seemed like overfitting, I trained models with previous hyperparameters fixed, but with a varying dropout rate. The results of this last experiment are listed below:

Dropout Rate	State Dim	Batch	Epochs	Train Err	Train Acc	Test Err	Test Acc	Runtime
0.1	200	256	9115694	0.05835	0.9799	0.6569	0.84508	529.00
0.2	200	256	9115694	0.06864	0.9766	0.7829	0.83916	524.99
0.3	200	256	9115694	0.08160	0.9718	0.7396	0.84604	524.07

Table 6: LSTM dropout rate

Unfortunately, adding dropout did not make a significant improvement on the test accuracy. The models still hover around 84% accuracy.

Optimizer	Glove Dim	State Dim	Batch	Epochs	Train Err	Train Acc	Test Err	Test Acc	Runtime
AdaDelta	100	200	256	30	0.2589	0.8908	0.3563	0.8476	507.73
RMSProp	200	200	256	30	0.02588	0.9913	1.1705	0.7935	608.46
RMSProp	200	200	256	22	0.05586	0.9795	0.8244	0.8125	452.37

Table 7: LSTM with different optimizers, higher dimension GloVe (word vectors)

4 Conclusion

In retrospect, it would have been much better to evaluate a model after each epoch, rather than running the each test again from scratch. It only occurred to me that this might be possible after the fact, and indeed it is possible to set a callback function to trigger after each training epoch.

5 Appendix 1 – RNN Experiment Results

State Dim	Batch	Epochs	Params	Train Error	Train Acc	Test Error	Test Acc	Runtime
20	128	3	4432708	0.6377	0.6432	0.6271	0.6621	249.6
50	128	3	4437298	0.6172	0.6618	0.6812	0.6125	360.6
100	128	3	4448948	0.6219	0.6533	0.6363	0.6226	615.5
200	128	3	4487248	0.6080	0.6589	0.5973	0.6767	1110.8
500	128	3	4722148	0.6932	0.4968	0.6931	0.5000	3458.8
20	128	3	4432708	0.6607	0.5970	0.6545	0.6018	209.8
50	256	3	4437298	0.6330	0.6451	0.6287	0.6466	318.6
100	256	3	4448948	0.6267	0.6444	0.6517	0.6188	562.8
200	256	3	4487248	0.6222	0.6452	0.6249	0.6444	1023.2
500	256	3	4722148	0.7037	0.5335	0.6821	0.5578	3257.6
20	128	6	4432708	0.6172	0.6644	0.6113	0.6813	428.7
50	128	6	4437298	0.5930	0.6791	0.5829	0.7024	621.3
100	128	6	4448948	0.5827	0.6849	0.5933	0.6706	1053.0
200	128	6	4487248	0.5720	0.6917	0.6346	0.6212	1970.6
500	128	6	4722148	0.6932	0.4983	0.6931	0.5000	6293.4
20	128	6	4432708	0.6078	0.6732	0.5907	0.6996	347.2
50	256	6	4437298	0.5954	0.6836	0.6174	0.6714	530.9
100	256	6	4448948	0.6165	0.6559	0.6364	0.6234	949.3
200	256	6	4487248	0.5674	0.7001	0.6098	0.6650	1810.3
500	256	6	4722148	0.6932	0.4970	0.6931	0.5000	5876.7
20	128	10	4432708	0.5847	0.6939	0.6314	0.6183	655.6
50	128	10	4437298	0.5760	0.6930	0.5980	0.6623	937.0
100	128	10	4448948	0.5585	0.7088	0.5889	0.6806	1643.7
200	128	10	4487248	0.5340	0.7234	0.6241	0.6687	3121.1
500	128	10	4722148	0.6935	0.4978	0.6937	0.5000	10129.6
20	128	10	4432708	0.6150	0.6628	0.6382	0.6440	540.0
50	256	10	4437298	0.5909	0.6888	0.6094	0.6631	822.2
100	256	10	4448948	0.5593	0.7100	0.5678	0.7086	1489.2
200	256	10	4487248	0.5196	0.7346	0.6190	0.6676	2878.4
500	256	10	4722148	0.6852	0.5666	0.6735	0.5828	9603.2

Table 8: Simple RNN varying state dimension, batch size and epochs

6 Appendix II – LSTM Experiment Results

State Dim	Batch	Epochs	Params	Train Err	Train Acc	Test Err	Test Acc	Runtime
20	64	1	8869654	0.6192	0.6558	0.8290	0.5847	798.7
50	64	1	8892094	0.5870	0.6826	0.4733	0.7769	802.2
100	64	1	8945494	0.6071	0.6668	0.4961	0.7730	806.5
200	64	1	9112294	0.6191	0.6570	0.4727	0.7749	813.5
20	128	1	8869654	0.6237	0.6548	0.5538	0.7205	578.2
50	128	1	8892094	0.6316	0.6414	0.9550	0.5036	589.8
100	128	1	8945494	0.6373	0.6334	0.5487	0.7325	595.5
200	128	1	9112294	0.6517	0.6222	0.5976	0.6752	597.9
20	256	1	8869654	0.6513	0.6128	0.5756	0.7138	498.8
50	256	1	8892094	0.6506	0.6180	0.6040	0.6816	506.0
100	256	1	8945494	0.6600	0.6046	0.5901	0.6954	507.3
200	256	1	9112294	0.6681	0.5987	0.5610	0.7182	509.6
20	64	2	8869654	0.5193	0.7458	0.4804	0.7666	1099.1
50	64	2	8892094	0.4329	0.8018	0.4262	0.8078	1125.5
100	64	2	8945494	0.4716	0.7806	0.4042	0.8178	1146.5
200	64	2	9112294	0.4284	0.8040	0.3814	0.8322	1147.2
20	128	2	8869654	0.5115	0.7545	0.4500	0.7930	693.4
50	128	2	8892094	0.5261	0.7428	0.5670	0.7187	717.6
100	128	2	8945494	0.5273	0.7426	0.4495	0.7928	678.8
200	128	2	9112294	0.5343	0.7387	0.5074	0.7414	712.9
20	256	2	8869654	0.5829	0.7016	0.6261	0.6500	549.1
50	256	2	8892094	0.5863	0.6945	0.5395	0.7333	551.3
100	256	2	8945494	0.5847	0.6980	0.6367	0.6522	549.9
200	256	2	9112294	0.6114	0.6739	0.6162	0.6643	553.7
20	64	3	8869654	0.4026	0.8214	0.4664	0.7816	1443.2
50	64	3	8892094	0.3885	0.8262	0.3716	0.8338	1492.5
100	64	3	8945494	0.3906	0.8255	0.3959	0.8362	1556.7
200	64	3	9112294	0.3710	0.8348	0.3453	0.8462	1598.0
20	128	3	8869654	0.4817	0.7718	0.4391	0.7979	890.1
50	128	3	8892094	0.4576	0.7886	0.4110	0.8129	924.7
100	128	3	8945494	0.4391	0.7930	0.3953	0.8219	933.9
200	128	3	9112294	0.4757	0.7735	0.4411	0.7933	936.5
20	256	3	8869654	0.5108	0.7552	0.4715	0.7844	645.6
50	256	3	8892094	0.5102	0.7524	0.5132	0.7446	664.2
100	256	3	8945494	0.5388	0.7308	0.5355	0.7227	658.5
200	256	3	9112294	0.5647	0.7134	0.5496	0.7383	672.8
20	64	4	8869654	0.3828	0.8309	0.3606	0.8403	1881.4
50	64	4	8892094	0.3652	0.8385	0.3851	0.8301	1933.1
100	64	4	8945494	0.3585	0.8431	0.3458	0.8526	1931.5
200	64	4	9112294	0.3424	0.8504	0.3430	0.8455	1984.0
20	128	4	8869654	0.4245	0.8067	0.3984	0.8221	1046.5
50	128	4	8892094	0.3983	0.8195	0.3760	0.8295	1092.7
100	128	4	8945494	0.3803	0.8280	0.4075	0.8090	1108.2
200	128	4	9112294	0.3804	0.8278	0.3738	0.8363	1111.9
20	256	4	8869654	0.5211	0.7468	0.5073	0.7542	731.4
50	256	4	8892094	0.4826	0.7708	0.4381	0.7986	747.2
100	256	4	8945494	0.4579	0.7845	0.4220	0.8015	739.4
200	256	4	9112294	0.4792	0.7759	0.4170	0.8102	755.4
20	64	5	8869654	0.3749	0.8322	0.3586	0.8392	2306.5
50	64	5	8892094	0.3366	0.8530	0.3599	0.8379	2306.6

State Dim	Batch	Epochs	Params	Train Err	Train Acc	Test Err	Test Acc	Runtime
100	64	5	8945494	0.3280	0.8570	0.3401	0.8511	2324.2
200	64	5	9112294	0.3141	0.8632	0.3223	0.8604	2371.3
20	128	5	8869654	0.3850	0.8297	0.3714	0.8360	1197.9
50	128	5	8892094	0.3575	0.8420	0.4097	0.8065	1252.3
100	128	5	8945494	0.3563	0.8419	0.3701	0.8336	1277.7
200	128	5	9112294	0.3556	0.8444	0.4414	0.7852	1282.4
20	256	5	8869654	0.4816	0.7738	0.4939	0.7679	803.4
50	256	5	8892094	0.4353	0.7988	0.4232	0.8092	827.1
100	256	5	8945494	0.4146	0.8092	0.4735	0.7713	832.0
200	256	5	9112294	0.4072	0.8131	0.3728	0.8339	848.0

Table 9: LSTM varying state dimension, batch size and epochs

References

[1] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[2] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162