Ryan McCrory
Professor Flanegan
Natural Language Processing
January 20, 2020

Assignment 2

Part 1:

1.1 Once I familiarized myself with the starter code, I modified it such that it split the IMDB data into a training set (60% of the data), a validation set (40% of the data), and a test set (100% of the data). I then preprocessed each of the data sets. Then I changed the model from using GRU layers, as it was in the starter code, to using a single SimpleRNN layer. Then I added the validation set to the history = model.fit() line, and began to play around with several hyperparameters. The accuracy of the models with several hyperparameter settings are shown in the tables below. Note that each table below uses the best (highlighted) hyperparameter of each of the tables before it, such that the number of epochs table uses the best optimizer, the training batch size table uses the best number of epochs and the best optimizer, and so on.

Optimizer

	Training	**Validation**
Optimizer = “adam”	0.5101	0.4990
Optimizer = “sgd”	0.5054	0.4996
Optimizer = “rmsprop”	0.5079	0.4990

Training batch size

	Training	**Validation**
batch = 32	0.5717	0.5106
batch = 128	0.5931	0.5219
batch = 512	0.5858	0.5036

Number of Epochs

	Training	**Validation**
Epochs = 1	0.5054	0.4996
Epochs = 3	0.5276	0.4996
Epochs = 5	0.5708	0.5040

Embed size

	Training	**Validation**
Embed size = 64	0.6043	0.5108
Embed size = 128	0.5931	0.5219
Embed size = 256	0.5973	0.5183

Dropout rate

	Training	Validation
0	0.5931	0.5219
0.2	0.5633	0.5151
0.4	0.5527	0.5020

1.2 As shown above, the best performing model had an accuracy of 0.5931 on the training set, and 0.5219 on the validation set. This was with the following hyperparameters:

Optimizer = “sgd”, Number of Epochs = 5, Training batch size = 128
Embed size = 128, Dropout rate = 0

Applying these hyperparameters to the test set results in an accuracy of 0.523. This is not an extremely impressive accuracy due to the simplicity of this model, vanishing gradients, and the model forgetting too much during each iteration, but I am sure the GRU model in part 2 will perform better.

Part 2:

2.1 Because the SimpleRNN model had low accuracy on the test set of only 0.523, I am going to use a gated recurrent unit (GRU) model instead. The results of this model using the same experimental procedure as in part 1 are shown in the tables below

Optimizer

	Training	Validation
Optimizer = “adam”	0.9625	0.7592
Optimizer = “sgd”	0.5043	0.4989
Optimizer = “rmsprop”	0.9378	0.7698

Training batch size

	Training	Validation
batch = 32	0.8761	0.7734
batch = 128	0.8755	0.7765
batch = 512	0.8770	0.7768

Number of Epochs

	Training	Validation
Epochs = 1	0.6851	0.7310
Epochs = 3	0.8761	0.7734
Epochs = 5	0.9378	0.7698

Embed size

	Training	Validation
Embed size = 64	0.8639	0.7712
Embed size = 128	0.8755	0.7768
Embed size = 256	0.8877	0.7761

Dropout rate

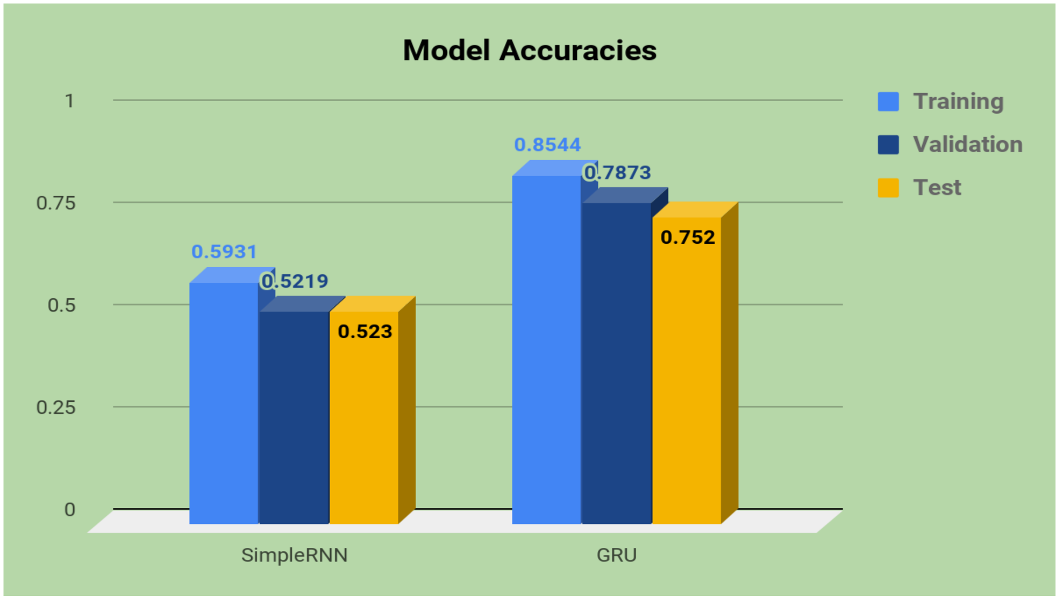
	Training	**Validation**
0	0.8755	0.7768
0.2	0.8683	0.7764
0.4	0.8544	0.7873

As shown above, the best performing model had an accuracy of 0.8544 on the training set, and 0.7873 on the validation set. This was with the following hyperparameters:

Optimizer = “rmsprop”, Number of Epochs = 3, Training batch size = 512
Embed size = 128, Dropout rate = 0.4

Applying these hyperparameters to the test set results in an accuracy of 0.752.

2.2 We would expect the GRU model to have better accuracy than the simple RNN model on longer sequences. This is because the simple RNN model suffers from vanishing gradients during learning, which makes it hard to propagate error into the distant past. Additionally, state tends to change a lot on each iteration, meaning the model “forgets” too much. We would not expect this to be the case for the GRU model because the update and reset gates allow the model to remember important data and forget less important data. This assumption does hold for my models, as shown below. The GRU model performed better than the Simple RNN model in every data set.



Part 3:

3.1 The embeddings I used in the previous sections were embedding layers which were trained alongside the model whereas the embeddings I used in this section were pretrained from glove (https://nlp.stanford.edu/projects/glove). Using an embedding layer can be beneficial because you can train it with respect to the problem you are trying to solve, whereas using a pretrained embedding may not be good for a given problem, although it does simplify the learning of the model.

3.2 Below are the charts showing the hyperparameters I tested while optimizing my model using the same experimental procedure as in the previous sections.

Optimizer

	Training	Validation
Optimizer = “adam”	0.9542	0.7396
Optimizer = “sgd”	0.5313	0.5056
Optimizer = “rmsprop”	0.8858	0.7608

Training batch size

	Training	Validation
Batch = 8	0.8480	0.7654
Batch = 16	0.8431	0.7658
batch = 32	0.8514	0.7648

Number of Epochs

	Training	Validation
Epochs = 3	0.8015	0.7561
Epochs = 4	0.8514	0.7648
Epochs = 5	0.8858	0.7608

Dropout Rate

	Training	Validation
0	0.8431	0.7658
0.2	0.8111	0.7673
0.4	0.7629	0.7622

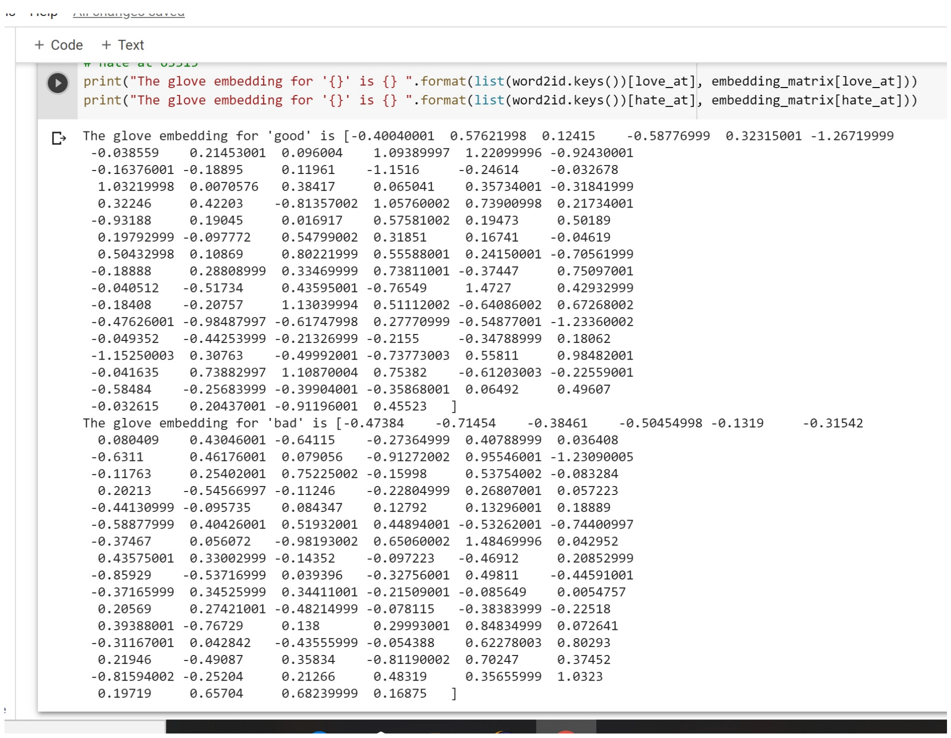
As shown above, the best performing model had an accuracy of 0.8111 on the training set, and 0.7673 on the validation set. This was with the following hyperparameters:

Optimizer = “rmsprop”, Number of Epochs = 4, Training batch size = 16
Dropout rate = 0.2

Applying these hyperparameters to the test set results in an accuracy of 0.746. Performance does not change much using pre trained embeddings. My GRU model in which I trained the embedding had an accuracy of 0.752 on the test set, while the GRU model with pre-trained embeddings had an accuracy of 0.746. So, the model in which I trained the embeddings performed 0.006 better than the pre-trained model, which is not a significant difference. Perhaps a different pre-trained embedding would have made more of a difference for this specific task, as the two I used turned out to be quite similar.

3.3 Antonyms may often have similar word embeddings because the models are trained on large amounts of data where antonyms are often used in similar contexts. For example, take the word “good” in the sentence “I can’t believe I have a good grade in NLP”. If there was another sentence in the training data in which “good” was replaced with “bad”, the context of the sentence would be unchanged, although its meaning would be the opposite. This causes antonyms to have similar word embeddings.

3.4 Below are the embedding matrices for the words “good” and “bad”, which are antonyms.



```
print("The glove embedding for '{}' is {} ".format(list(word2id.keys())[love_at], embedding_matrix[love_at]))
print("The glove embedding for '{}' is {} ".format(list(word2id.keys())[hate_at], embedding_matrix[hate_at]))
```

```
The glove embedding for 'good' is [-0.40040001  0.57621998  0.12415    -0.58776999  0.32315001 -1.26719999
 -0.038559    0.21453001  0.096004    1.09389997  1.22099996 -0.92430001
 -0.16376001 -0.18895     0.11961    -1.1516     -0.24614    -0.032678
  1.03219998  0.0070576   0.38417     0.065041    0.35734001 -0.31841999
  0.32246     0.42203    -0.81357002  1.05760002  0.73900998  0.21734001
 -0.93188     0.19045     0.016917    0.57581002  0.19473     0.50189
  0.19792999 -0.097772    0.54799002  0.31851     0.16741    -0.04619
  0.50432998  0.10869     0.80221999  0.55588001  0.24150001 -0.70561999
 -0.18888     0.28808999  0.33469999  0.73811001 -0.37447     0.75097001
 -0.040512   -0.51734     0.43595001 -0.76549     1.4727      0.42932999
 -0.18408    -0.20757     1.13039994  0.51112002 -0.64086002  0.67268002
 -0.47626001 -0.98487997 -0.61747998  0.27770999 -0.54877001 -1.23360002
 -0.049352   -0.44253999 -0.21326999 -0.2155     -0.34788999  0.18062
 -1.15250003  0.30763    -0.49992001 -0.73773003  0.55811     0.98482001
 -0.041635    0.73882997  1.10870004  0.75382    -0.61203003 -0.22559001
 -0.58484    -0.25683999 -0.39904001 -0.35868001  0.06492     0.49607
 -0.032615    0.20437001 -0.91196001  0.45523   ]
The glove embedding for 'bad' is [-0.47384    -0.71454    -0.38461    -0.50454998 -0.1319     -0.31542
  0.080409    0.43046001 -0.64115    -0.27364999  0.40788999  0.036408
 -0.6311      0.46176001  0.079056   -0.91272002  0.95546001 -1.23090005
 -0.11763     0.25402001  0.75225002 -0.15998     0.53754002 -0.083284
  0.20213    -0.54566997 -0.11246    -0.22804999  0.26807001  0.057223
 -0.44130999 -0.095735    0.084347    0.12792     0.13296001  0.18889
 -0.58877999  0.40426001  0.51932001  0.44894001 -0.53262001 -0.74400997
 -0.37467     0.056072   -0.98193002  0.65060002  1.48469996  0.042952
  0.43575001  0.33002999 -0.14352    -0.097223   -0.46912     0.20852999
 -0.85929    -0.53716999  0.039396   -0.32756001  0.49811    -0.44591001
 -0.37165999  0.34525999  0.34411001 -0.21509001 -0.085649    0.0054757
  0.20569     0.27421001 -0.48214999 -0.078115   -0.38383999 -0.22518
  0.39388001 -0.76729     0.138       0.29993001  0.84834999  0.072641
 -0.31167001  0.042842   -0.43555999 -0.054388    0.62278003  0.80293
  0.21946    -0.49087     0.35834    -0.81190002  0.70247     0.37452
 -0.81594002 -0.25204     0.21266     0.48319     0.35655999  1.0323
  0.19719     0.65704     0.68239999  0.16875   ]
```

The embeddings are fairly similar, but not to a great extent. The first entry in the embedding matrix for “good” is -0.40040001 and the first entry in the embedding matrix for “bad” is -0.47384, which are similar. There are several of these similarities throughout the matrices, although not in every position.

Note: I could not get the code to work to test whether or not substituting these antonyms in a sentence changed the sentiment prediction.