# Assignment 2

Maverick Ho, Alice Liang, Richard Shang

February 2020

## Part 1

1. Our goal is to build a RNN-based text classifier for movie reviews. We begin by splitting the IMBD review database in half, one for training and the other for testing. We then further split the training data into a training set and a development set, with a 60:40 ratio. The training set is further split into batches of size 32 and a vocabulary is built from the training set. The batches are then encoded as well. We then finally proceed to build the model. For the first part the model is a simple single layer RNN. Our model consists of 3 parts, an embedding layer, a single SimpleRNN layer, and a Dense layer. We picked the parameters, drop out rate, activation function and epochs. We tested each individually by keeping the rest of the parameters constant throughout the test.

2. For all of our tests, the following parameters were constant: a batch size of 32, and utilized the ADAM optimization. First we take a look at which drop out rate works best.

| Tests | Activation Function | Dropout | Epochs | Train Acc | Dev Acc |
|---|---|---|---|---|---|
| 1 | tanH | .1 | 5 | .8589 | .5665 |
| 2 | tanH | .15 | 5 | .7592 | .6476 |
| 3 | tanH | .25 | 5 | .8551 | .6153 |
| 3 | tanH | .3 | 5 | .8816 | .5152 |

We test for best activation function works best. ReLU is the winner.

| Tests | Activation Function | Dropout | Epochs | Train Acc | Dev Acc |
|---|---|---|---|---|---|
| 1 | Sigmoid | .15 | 5 | .8589 | .5665 |
| 2 | tanH | .15 | 5 | .7592 | .6476 |
| 3 | reLU | .15 | 5 | .9296 | .6973 |
| 4 | exponential | .15 | 5 | .5007 | .4990 |
| 5 | softmax | .15 | 5 | ..6733 | .5331 |

We then test for what number of epochs is the best.

| Tests | Activation Function | Dropout | Epochs | Train Acc | Dev Acc |
|---|---|---|---|---|---|
| 1 | reLU | .15 | 5 | .9296 | .6973 |
| 2 | reLU | .15 | 10 | .9827 | .7279 |
| 3 | reLU | .15 | 15 | .9891 | .6801 |

So the best parameters are dropout at .15, epochs at 10, and activation function reLU. We then run it on the test set for our final accuracy.

| Test | Activation Function | Dropout | Epochs | Test Acc |
|---|---|---|---|---|
| 5 | relu | .15 | 10 | .6770 |

# Part 2

1. In part 2, we changed our simple RNN layer into a GRU layer with the same hyper-parameters : units = 128 , activation = "relu" , dropout = .15 , epochs = 10. Using the same batch sizes for train,development, and test from part 1, we got the following accuracies.

Train and Development

| Epochs | Accuracy | Val_Accuracy |
|---|---|---|
| 8 | 0.9869 | 0.7461 |
| 9 | 0.9915 | 0.7475 |
| 10 | 0.9953 | 0.7477 |

Test

| Loss | Accuracy |
|---|---|
| 1.8911 | 0.7273 |

```
[130] embed_size = 128
      model = keras.models.Sequential([
          keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                                 mask_zero=True, # not shown in the book
                                 input_shape=[None]),
          keras.layers.GRU(128,activation ='relu', dropout = .15),
          keras.layers.Dense(1, activation="sigmoid")
      ])
      model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
      history = model.fit(train_set, validation_data=dev_set, epochs=10)
```

```
Epoch 1/10
469/469 [==============================] - 50s 106ms/step - loss: 0.6198 - accuracy: 0.6256 - val_loss: 0.4965 - val_accuracy: 0.7624
Epoch 2/10
469/469 [==============================] - 48s 103ms/step - loss: 0.3839 - accuracy: 0.8300 - val_loss: 0.5303 - val_accuracy: 0.7608
Epoch 3/10
469/469 [==============================] - 49s 104ms/step - loss: 0.2568 - accuracy: 0.8970 - val_loss: 0.6398 - val_accuracy: 0.7515
Epoch 4/10
469/469 [==============================] - 49s 105ms/step - loss: 0.1916 - accuracy: 0.9269 - val_loss: 0.9313 - val_accuracy: 0.7185
Epoch 5/10
469/469 [==============================] - 49s 105ms/step - loss: 0.1530 - accuracy: 0.9430 - val_loss: 0.9231 - val_accuracy: 0.7219
Epoch 6/10
469/469 [==============================] - 49s 104ms/step - loss: 0.1061 - accuracy: 0.9606 - val_loss: 1.0578 - val_accuracy: 0.7483
Epoch 7/10
469/469 [==============================] - 49s 105ms/step - loss: 0.0693 - accuracy: 0.9764 - val_loss: 1.2937 - val_accuracy: 0.7469
Epoch 8/10
469/469 [==============================] - 50s 106ms/step - loss: 0.0398 - accuracy: 0.9869 - val_loss: 1.6181 - val_accuracy: 0.7461
Epoch 9/10
469/469 [==============================] - 49s 105ms/step - loss: 0.0262 - accuracy: 0.9915 - val_loss: 1.6655 - val_accuracy: 0.7475
Epoch 10/10
469/469 [==============================] - 50s 106ms/step - loss: 0.0165 - accuracy: 0.9953 - val_loss: 1.7630 - val_accuracy: 0.7477
```

```
[131] history.model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, None, 128)         1408000
_________________________________________________________________
gru_9 (GRU)                  (None, 128)               99072
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 129
=================================================================
Total params: 1,507,201
Trainable params: 1,507,201
Non-trainable params: 0
_________________________________________________________________
```

```
[132] test_set = test_data.batch(512).map(preprocess)
      test_set = test_set.map(encode_words).prefetch(1)
```

embedds the test data, this is how well it performs?

```
loss,accuracy = model.evaluate(test_set,steps=10)
```

```
10/10 [==============================] - 1s 80ms/step - loss: 1.8911 - accuracy: 0.7273
```

Or using manual masking: This basically ignores the padding tokens

2. Changing the batch size from 32 for training and development to 64 and keeping batch size 512 for testing, we get the following accuracies.

Train and Development

| Epochs | Accuracy | Val_Accuracy |
|---|---|---|
| 8 | 0.9704 | 0.7395 |
| 9 | 0.9812 | 0.7440 |
| 10 | 0.9887 | 0.7416 |

Test

| Loss | Accuracy |
|---|---|
| 1.4861 | 0.7209 |

```
[142] embed_size = 128
      model = keras.models.Sequential([
          keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                                 mask_zero=True, # not shown in the book
                                 input_shape=[None]),
          keras.layers.GRU(128,activation ='relu', dropout = .15),
          keras.layers.Dense(1, activation="sigmoid")
      ])
      model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
      history = model.fit(train_set, validation_data=dev_set, epochs=10)
```

```
Epoch 1/10
235/235 [==============================] - 28s 117ms/step - loss: 0.6577 - accuracy: 0.5878 - val_loss: 0.5415 - val_accuracy: 0.7340
Epoch 2/10
235/235 [==============================] - 27s 114ms/step - loss: 0.4216 - accuracy: 0.8049 - val_loss: 0.5922 - val_accuracy: 0.7619
Epoch 3/10
235/235 [==============================] - 26s 112ms/step - loss: 0.2944 - accuracy: 0.8806 - val_loss: 0.6849 - val_accuracy: 0.7667
Epoch 4/10
235/235 [==============================] - 27s 116ms/step - loss: 0.2233 - accuracy: 0.9123 - val_loss: 0.6629 - val_accuracy: 0.7516
Epoch 5/10
235/235 [==============================] - 26s 113ms/step - loss: 0.1880 - accuracy: 0.9271 - val_loss: 0.7222 - val_accuracy: 0.7528
Epoch 6/10
235/235 [==============================] - 27s 116ms/step - loss: 0.1510 - accuracy: 0.9417 - val_loss: 0.8844 - val_accuracy: 0.7338
Epoch 7/10
235/235 [==============================] - 26s 112ms/step - loss: 0.1193 - accuracy: 0.9546 - val_loss: 0.8862 - val_accuracy: 0.7380
Epoch 8/10
235/235 [==============================] - 27s 115ms/step - loss: 0.0809 - accuracy: 0.9704 - val_loss: 1.1190 - val_accuracy: 0.7395
Epoch 9/10
235/235 [==============================] - 27s 114ms/step - loss: 0.0532 - accuracy: 0.9812 - val_loss: 1.1963 - val_accuracy: 0.7440
Epoch 10/10
235/235 [==============================] - 27s 114ms/step - loss: 0.0350 - accuracy: 0.9887 - val_loss: 1.3593 - val_accuracy: 0.7416
```

```
[143] history.model.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_10 (Embedding)     (None, None, 128)         1408000
_________________________________________________________________
gru_10 (GRU)                 (None, 128)               99072
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 129
=================================================================
Total params: 1,507,201
Trainable params: 1,507,201
Non-trainable params: 0
_________________________________________________________________
```

```
[144] test_set = test_data.batch(512).map(preprocess)
      test_set = test_set.map(encode_words).prefetch(1)
```

embedds the test data, this is how well it performs?

```
loss,accuracy = model.evaluate(test_set,steps=10)
```

```
10/10 [==============================] - 1s 84ms/step - loss: 1.4861 - accuracy: 0.7209
```

Changing the batch size from 32 for training and development to 16 and keeping batch size 512 for testing, we get the following accuracies.

Train and Development

| Epochs | Accuracy | Val_Accuracy |
|---|---|---|
| 8 | 0.9898 | 0.7494 |
| 9 | 0.9950 | 0.7508 |
| 10 | 0.9971 | 0.7497 |

Test

| Loss | Accuracy |
|---|---|
| 2.3921 | 0.7258 |

When comparing the accuracies between a simple RNN and a GRU, holding all hyper-parameters constant, a GRU has a higher accuracy value than a simple RNN, specifically by 0.0503. However, when doubling the batch sizes within a GRU and holding everything else constant, having a changed batch size to 64 (from 32) for both training and development decreases the test accuracy by 0.0064. A half in size for batch size within a GRU holding everything else constant results in a decrease of .0015. This makes sense because a smaller batch size results in a higher accuracy. However, too small of a batch size might also decrease accuracy.

By the data given from above, our model does hold true for an increase in accuracy from changing from a simple RNN to a GRU.

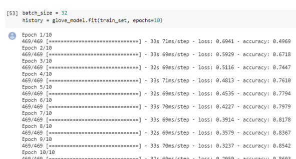
# Part 3

1. The pre-trained word embedding that we use is the GloVe embedding. GloVe stands for Global Vectors for word representation, working under

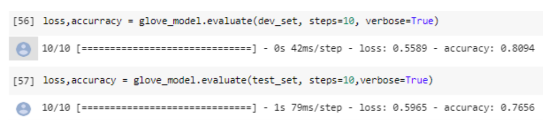
the observation that ratios of word-to-word occurrence probabilities have a higher probability to occur with certain words. For example, if the word sunny appears, it is more likely that the word day might come after it than lets say, night. We use a pre-trained word vectors provided by the team at Stanford that trained word embedding on 6 billion tokens from Wikipedia and Gigaword, located in the file glove6B. The embed size for each word vector is 100.

Citation: Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.

2. We used the same parameters based on part 2 to train the GloVe model. We used an input size of 200, dropout rate of 0.14, relu for the activation function, and 5 epochs. We trained the train data and the development data with a batch size of 32, and test data used a batch size of 512. The



(a) Training set Accuracy




(b) Dev and Test Accuracy

accuracy has improved significantly.

3. Antonyms usually have similar embeddings because a word's antonym have similar structure in relation to other words. For example, an antonym of great could be the word terrible. When used in sentences, "Wow it is a great day!" can just as easily be replaced with "Wow it is a terrible day!". Depending on the context of that phrase, a model that predicts words would probably weigh great and terrible to have similar probability, even though they have vastly different meanings.

4. If we use a word that would definitely trigger a negative or positive label, the antonyms would evoke a similar response. Words like "loved" or "awesome" would elicit a higher probability of a positive score, while words like "hated" or "terrible" would probably assign a worse score. And judging from the predictions on the sentences

negative_sentence = "I absolutely hated this movie"
positive_sentence = "I absolutely loved this movie"

```
b'I absolutely loved this movie'
Positive results: [[0.98872954]]
b'I absolutely hated this movie'
Negative results: [[0.05390472]]
```

It is clear the antonyms hated and loved heavily swayed the GloVe model, and this would hold true for words whose semantic meaning is usually determined by that one word.. However, when calculating the cosine similarity between both sentences' embeddings, we got a similar score of

```
cos_sim = dot(pos_arr.flatten(),neg_arr.flatten())/(norm(pos_arr.flatten())*norm(neg_arr.flatten()))

print(cos_sim)
```

```
0.80763806264761
```

Similar antonyms like terrible and great had similar results. They had high similar cosine-similarity .98, and accurate predictions.

However, words that rely on the context of the sentence have high cosine probability scores, such as "nothing" and "everything", yet low prediction scores.

```
b'This was everything like I hoped for'
Positive results: [[0.63972294]]
b'This was nothing like I hoped for'
Negative results: [[0.5567835]]
0.9988582900252582
```

(a) Antonyms: everything and nothing

```
b'This movie made me want to live'
Positive results: [[0.7590981]]
b'This movie made me want to die'
Negative results: [[0.7463346]]
0.9632550825984528
```

(b) Antonyms: live and die

These types of words could be used in various positive or negative sentences, and had very high cosine similarities, yet the model was unsure of a prediction.