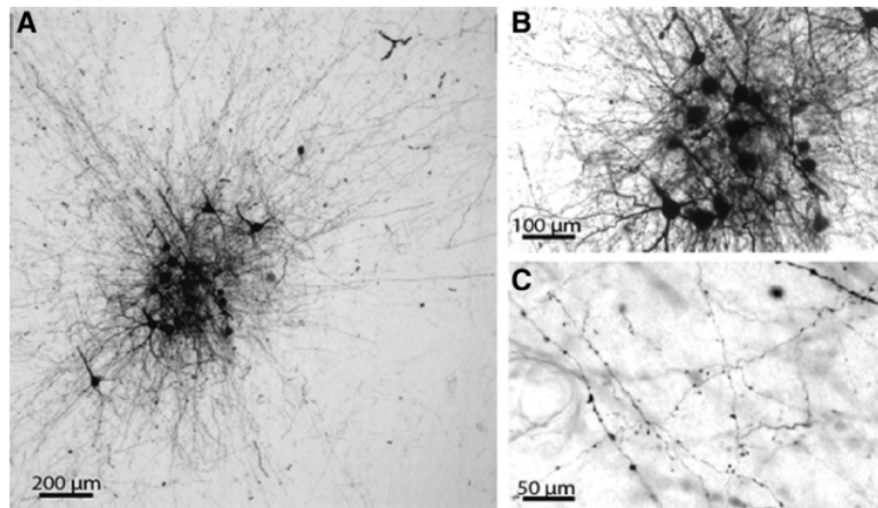


# STATS302 Final: Recurrent Neural Network

Haitong Lin, Zhaozhi (Claire) Li

## Introduction to RNN

A Recurrent Neural Network (RNN) is a special kind of neural network whose neurons send feedback signals to one another. The major motivations of RNN models are sequential processing and the modeling of neuronal connectivity. In reality, human brains do not look like regular neural networks. Instead, we have neurons that connect in a dense web called a recurrent network. With these motivations, RNNs are designed for processing sequential data. This concept now has a lot of possibilities and many variations, but to better understand its historical background, we need to briefly introduce the Hopfield Networks and the LSTM Networks.



Recurrent networks (in neuroscience)

RNN was based on psychologist David Rumelhart's work in 1986, and Hopfield Networks is the first formulation of a “recurrent-like” neural network. It was proposed by John Hopfield in 1982 and is a very general form of a neural network built in the context of theoretical neuroscience. Its attempt is to understand the mechanism underlying associative memory, which is still an important field of study in neuroscience to this day.

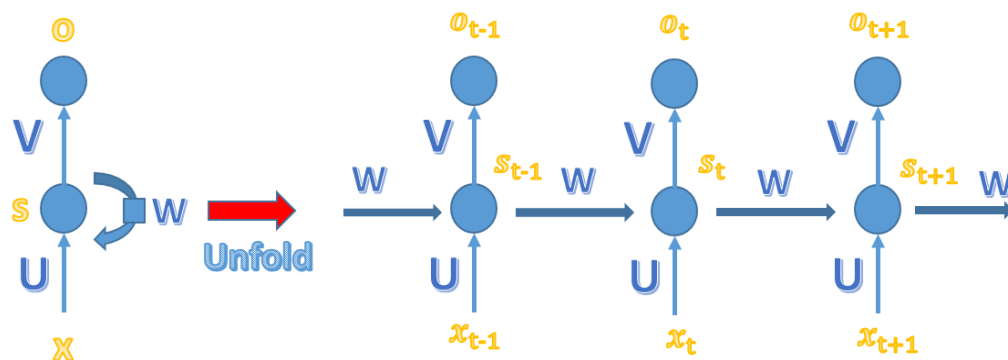
The Long Short-term Memory (LSTM) Networks were invented by Hochreiter and Schmidhuber in 1997. LSTM is an RNN architecture that has feedback connections and can not only process single data points, but also the entire data sequence. For instance, LSTM can process images (data points), but also videos (data sequences). However, even though LSTM was first proposed in 1997,

it wasn't until 2007 when computational resources were powerful enough to train LSTM networks. Since then, they have begun to revolutionize the field.

LSTM broke records in the field of speech recognition, machine translation, language modeling and multilingual language processing, et cetera. Firstly, RNN outperformed all traditional models in certain speech recognition applications. Then in 2009, RNNs set the record for hand-writing recognition. Chinese company Baidu used RNN to break the Switchboard Hub5'00 speech recognition dataset benchmark without using any traditional speech processing methods in 2014. Later on, in 2015, Google's speech recognition received a 47% jump-up improvement using an LSTM network. Together with convolutional neural networks (CNN), they have also significantly improved image captioning.

## The RNN Model

In addition to the composition of CNN, an input layer, a hidden layer, and an output layer, the RNN introduced a weighted matrix  $W$  to the model. Such design allows RNN to apply the impact of previous input vectors in the sequence when processing new inputs. That is to say, the output of  $s_t$  does not only depend on the input at current time\_step but also the hidden layer value of the previous time step,  $s_{t-1}$  -- Weighted matrix  $W$  uses  $s_{t-1}$  as weights on the current input.



RNN Illustration

## Coefficient Understanding

$x^{(1)}, x^{(2)}, \dots, x^{(\tau)}$ : Sequence of vectors input, denote  $x^{(t)}$ , where  $1 \leq t \leq \tau$ , as input at t.  
 $s^{(1)}, s^{(2)}, \dots, s^{(\tau)}$ : Sequence of hidden layer vectors, denote  $s^{(t)}$ , where  $1 \leq t \leq \tau$ , as input at t.  
 $h^{(t)}$  Activity of the neuron inside the RNN at time t  
 $o^{(t)}$  Output at time t  
 $y^{(t)}$  The target output at time t  
 $b_i$  Internal bias  
 $U_{ij}$  Input weights, connecting input j to RNN neuron i. (from previous hidden layer to current hidden layer )  
 $W_{ij}$  Internal weights, connecting RNN neuron j to RNN neuron i and  $b_i$  on RNN neuron i. (from hidden layer to output layer)  
 $V_{ij}$  Weighted matrix, of the output layer. (from the input layer to hidden layer)  
 $L(o^{(t)}, y^{(t)})$  Loss function

## Mathematical Calculation

RNN is mainly based on the two functions below,

$$\begin{aligned}
 o_t &= g(V s_t) \\
 s_t &= f(Ux_t + W s_{t-1})
 \end{aligned}$$

Where g and f are both activation functions.

Substituting  $s_t$  in  $o_t$ , we get below derivation.

$$\begin{aligned}
 o_t &= g(V s_t) \\
 &= V f(Ux_t + W s_{t-1}) \\
 &= V f(Ux_t + W f(Ux_{t-1} + W s_{t-2})) \\
 &= V f(Ux_t + W f(Ux_{t-1} + W f(Ux_{t-2} + W s_{t-3}))) \\
 &= V f(Ux_t + W f(Ux_{t-1} + W f(Ux_{t-2} + W f(Ux_{t-3} + \dots))))
 \end{aligned}$$

The result involves  $x_t, x_{t-1}, x_{t-2}, x_{t-3}, \dots$ , which explains how input at the previous time step also impacts current output.

## The Training of RNN

### Loss Function

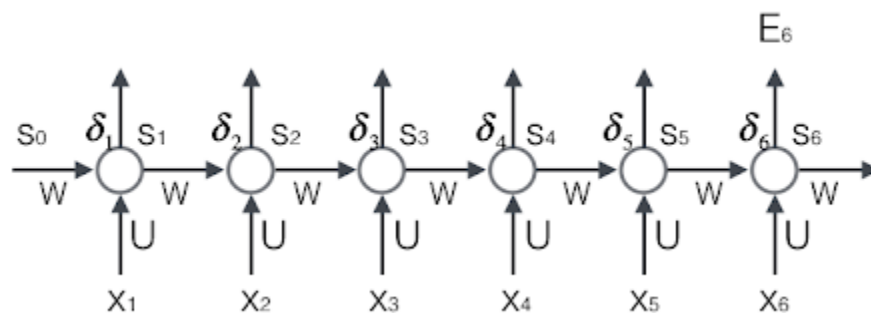
The loss function for RNN is decided by each specific application. In other words, we do not have a very general type of “loss function for RNN”, or we can say that the rules for choosing a loss function for RNN are generally the same for choosing that for a regular neural network. For example, when the output layer is softmax, we will choose cross-entropy as the loss function, and if we only have two output classes (like what we have in the application section below), we will use the binary cross-entropy as our loss function.

## Back-propagation Through Time

RNN is trained with back-propagation through time algorithm. Back-propagation directly works on a recurrent layer of RNN, updating the weight matrix. Similar to regular back-propagation, BPTT contains three key steps:

1. Forward propagation, calculating the output of each neuron
2. Backpropagation, calculating the error of output of each neuron.
3. Calculating the gradient of each weight

And finally, an updated weight matrix is obtained using stochastic gradient descent.



Back-propagation Illustration

### 1. Forward Propagation

In forward propagation, the output is computed in time order, from left to right.

$$s_t = f(Ux_t + W s_{t-1})$$

More explicitly, assuming shape of  $U$  is  $n \times m$ , shape of  $W$  is  $n \times n$ , function can be expressed in the matrix equation below.

$$\begin{bmatrix} s_1^t \\ s_2^t \\ \vdots \\ s_n^t \end{bmatrix} = f \left( \begin{bmatrix} u_{11} u_{12} \dots u_{1m} \\ u_{21} u_{22} \dots u_{2m} \\ \vdots \\ u_{n1} u_{n2} \dots u_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} w_{11} w_{12} \dots w_{1n} \\ w_{21} w_{22} \dots w_{2n} \\ \vdots \\ w_{n1} w_{n2} \dots w_{nn} \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \right)$$

Where  $s_i^t$  is the  $i$ -th element of vector  $s$  at  $t$ ;  $u_{ij}$  is the weight of  $i$ -th neuron of input layer at  $t$  on the  $j$ -th neuron of recurrent layer at  $t$ ; and  $w_{ij}$  is the weight of the  $i$ -th neuron of the recurrent layer at  $t-1$  on the  $j$ -th neuron of the recurrent layer at  $t$ .

## 2. Back-propagation

Next, we will introduce backpropagation to calculate the error of output in each time step. Backpropagation takes  $\delta_t^l$ , error value at  $t$  time step,  $l$  layer, and propagate along with two directions--propagate from the input layer to hidden layer, relating to  $U$ , and propagate from hidden layer to output layer, relating to  $W$ .

First, calculating the error value of  $W$ . Define  $net_t = U_{x_t} + W_{s_{t-1}}$ , and then  $s_{t-1} = f(net_{t-1})$ , where  $net_t$  is the weighted input of neuron at  $t$ . The calculation is propagated backward with the recurrent function below, by applying the chain rule:

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial net_{t-1}}$$

Both  $\frac{\partial net_t}{\partial net_{t-1}}$  and  $\frac{\partial net_t^l}{\partial net_{t-1}^{l-1}}$  can be unfolded into a Jacobian matrix. Combining two partial derivations together, we obtain the recurrent function in the form below. Such matrix describes the rule for BPTT to propagate through time, with which we could calculate the error value  $\delta_k$  of any time  $k$ . This is the algorithm for propagating the error. Where  $diag()$  denotes a diagonal matrix built with  $a$ .

$$\begin{aligned}\frac{\partial \text{net}_t}{\partial \text{net}_{t-1}} &= \frac{\partial \text{net}_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial \text{net}_{t-1}} \\ &= W \text{diag}[f'(\text{net}_{t-1})]\end{aligned}$$

$$\begin{aligned}\delta_k^T &= \frac{\partial E}{\partial \text{net}_k} \\ &= \frac{\partial E}{\partial \text{net}_t} \frac{\partial \text{net}_t}{\partial \text{net}_k} \\ &= \frac{\partial E}{\partial \text{net}_t} \frac{\partial \text{net}_t}{\partial \text{net}_{t-1}} \frac{\partial \text{net}_{t-1}}{\partial \text{net}_{t-2}} \cdots \frac{\partial \text{net}_{k+1}}{\partial \text{net}_k} \\ &= W \text{diag}[f'(\text{net}_{t-1})] W \text{diag}[f'(\text{net}_{t-2})] \cdots W \text{diag}[f'(\text{net}_k)] \delta_t^l \\ &= \delta_t^T \prod_{i=k}^{t-1} W \text{diag}[f'(\text{net}_i)]\end{aligned}$$

For calculating error propagation of U in the recurrent layer, the relationship of weighted input between two consecutive layers is defined below, where  $\text{net}_t^l$  is the weighted input of recurrent layer l;  $a_t^{l-1}$  is the output of layer l-1 neuron; and  $f^{l-1}$  is the activation function of layer l-1.

$$\begin{aligned}\text{net}_t^l &= U a_t^{l-1} + W_{s_{t-1}} \\ a_t^{l-1} &= f^{l-1}(\text{net}_t^{l-1})\end{aligned}$$

The calculation of the algorithm is similar to it in the fully connected layer. The error function is the rule of propagating error value to the previous layer.

$$\begin{aligned}\frac{\partial \text{net}_t^l}{\partial \text{net}_t^{l-1}} &= \frac{\partial \text{net}_t^l}{\partial a_t^{l-1}} \frac{\partial a_t^{l-1}}{\partial \text{net}_t^{l-1}} \\ &= U \text{diag}[f'^{l-1}(\text{net}_t^{l-1})] \\ (\delta_t^{l-1})^T &= \frac{\partial E}{\partial \text{net}_t^{l-1}} \\ &= \frac{\partial E}{\partial \text{net}_t^l} \frac{\partial \text{net}_t^l}{\partial \text{net}_t^{l-1}} \\ &= (\delta_t^l)^T U \text{diag}[f'^{l-1}(\text{net}_t^{l-1})]\end{aligned}$$

### 3. Gradient of Weight

The final step of BPTT is to calculate the gradient of each weight matrix. We will again, first calculate the gradient of error function over matrix W. The recursive function is achieved by applying the chain rule. The gradient of each time step is given below:

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}} &= \frac{\partial E}{\partial net_j^t} \frac{\partial net_j^t}{\partial w_{ji}} \\ &= \delta_j^t s_i^{t-1}\end{aligned}$$

The overall gradient of W is the sum of the gradient of all time

$$\nabla_W E = \sum_{i=1}^t \nabla_{W_i} E$$

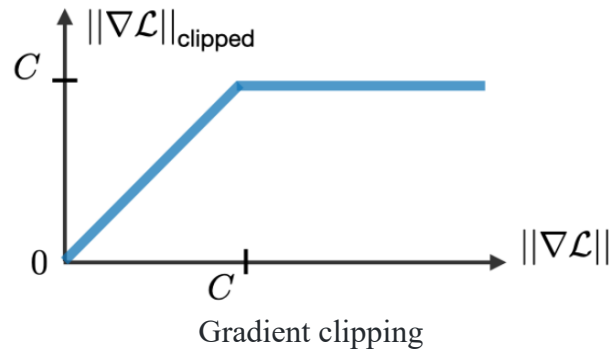
Similarly, we could obtain the gradient of U as below

$$\nabla_U E = \sum_{i=1}^t \nabla_{U_i} E$$

### Exploding and Vanishing Gradient

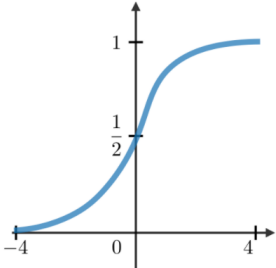
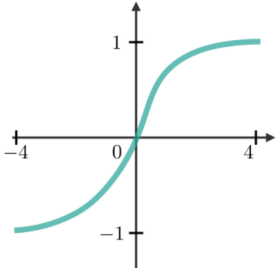
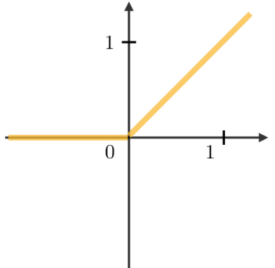
The biggest problems in training RNN are vanishing gradient and exploding gradient. The reason behind these is that the multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers. In other words, this happens because the functions computed by RNNs are highly nonlinear in general. Therefore, they tend to have derivatives with either very large or very small magnitudes.

When the gradient is exponentially increasing, we call it an exploding gradient. And to rectify this, we usually employ a technique called gradient clipping. It is a simple yet effective method achieved by capping a maximum value.



When the gradient is exponentially decreasing, we call it a vanishing gradient problem. Compared to exploding gradient problems, it is relatively harder to capture. To deal with vanishing gradients,

we have these common ways: initialize weight values such that the neurons do not easily go to the extremes. We can also use relu function as the activation function for the network instead of the sigmoid function and the tanh function.

| Sigmoid   | Tanh  | RELU  |
|---|---|---|
| $g(z) = \frac{1}{1 + e^{-z}}$   | $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$  | $g(z) = \max(0, z)$   |
|  |  |  |

Different activation functions

However, the most popular method to deal with this is using different RNN structures to avoid vanishing gradients. Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU.

## Application: Simple RNN Implementation

### Learning Task

We will implement RNN on an IMDB movie review dataset, and our learning task is to do sentiment analysis with the movie reviews. Our dataset is retrieved from <https://ai.stanford.edu/~amaas/data/sentiment/>, and we used a slightly shortened version to implement our model.

### Model Implementation



```
In [1]: 1 from tensorflow.keras.preprocessing import text_dataset_from_directory
2 from tensorflow.strings import regex_replace
3 from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
4 from tensorflow.keras.models import Sequential
5 from tensorflow.keras import Input
6 from tensorflow.keras.layers import Dense, LSTM, Embedding, Dropout
```

```
In [2]: 1 def prepareData(dir):
2     data = text_dataset_from_directory(dir)
3     return data.map(
4         lambda text, label: (regex_replace(text, '<br />', ' '), label),
5     )
6
7 train_data = prepareData('./train')
8 test_data = prepareData('./test')
9
10 for text_batch, label_batch in train_data.take(1):
11     print(text_batch.numpy()[0])
12     print(label_batch.numpy()[0]) # 0 = negative, 1 = positive
```

Found 25000 files belonging to 2 classes.

Found 25000 files belonging to 2 classes.

b"they have sex with melons in Asia. okay. first, i doubted that, but after seeing the wayward cloud, i changed my mind and was finally convinced that they have sex with watermelons, with people dead or alive. no safe sex of course. the (terrifyingly ugly) leading man shoots it all into the lady's mouth after he did the dead lady. never heard of HIV? guess not. the rest of this movie is mainly boring, but also incredibly revolting. as a matter of fact, in parts it got so disgusting i couldn't take my virgin eyes off. sex with dead people! how gross is that? and what's the message behind it all? we need water, we need melons, we need to be dead to have sex? sorry, but this stinks!"

0

Firstly, we will implement the relevant Keras packages and prepare the data. These packages have made it very convenient for us to read in our dataset. To see an example data, from the output in the above, there is a very aggressive review and so the number given is 0, which indicates very negative sentiment.

After these preparations, we are ready to build our model.

```
In [3]: 1 model = Sequential()
2
3 # 1. INPUT
4 model.add(Input(shape=(1,), dtype="string"))
```

We will use the Sequential class in Keras and initialize our first layer for input. We will define the input layer to take in 1 string input. Next, we will figure out what to do with this string.

```
In [4]: 1 # TEXT VECTORIZATION
2 max_tokens = 1000
3 max_len = 100
4 vectorize_layer = TextVectorization(max_tokens=max_tokens,output_mode="int",output_sequence_length=max_len,)
5
6 train_texts = train_data.map(lambda text, label: text)
7 vectorize_layer.adapt(train_texts)
8
9 model.add(vectorize_layer)
```

The second layer that we are adding is the text vectorization class. This layer processes the input string and turns it into a sequence of max\_len integers, each of which maps to a certain token. To initialize the layer we will need to call adapt(), which fits the TextVectorization layer to our text dataset. And this is when the max\_tokens, most common words (i.e. the vocabulary) are selected. And then we can add this layer to our model.

```

In [5]: 1 # EMBEDDING
        2 model.add(Embedding(max_tokens + 1, 128))

In [6]: 1 # RECURRENT LAYER
        2 model.add(LSTM(64))

In [7]: 1 # DENSE HIDDEN LAYER
        2 model.add(Dense(64, activation="relu"))

In [8]: 1 # OUTPUT
        2 model.add(Dense(1, activation="sigmoid"))

```

The code above shows the remaining layers we add to this model. The third layer we are adding is the embedding layer, which turns each integer (representing a token) from the previous layer into an embedding. Note that we're using `max_tokens + 1` here since there's an out-of-vocabulary (OOV) token that gets added to the vocab.

The fourth layer is the recurrent layer that finally makes our model an RNN. We will be using an LSTM layer here (which is a popular choice for this kind of problem). And to finish off our model, we will add a standard fully-connected (dense) layer and an output layer with a sigmoid function. We use a sigmoid as activation here because it outputs a number between 0 and 1, which is perfect for our problem - 0 represents a negative review, and 1 represents a positive one.

## Training and Testing Results

```

In [9]: 1 # Compile and train the model
        2 model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
        3 model.fit(train_data, epochs=10)
        4
        5 model.save_weights('rnn')
        6 model.load_weights('rnn')

Epoch 1/10
782/782 [=====] - 67s 86ms/step - loss: 0.5367 - accuracy: 0.7238
Epoch 2/10
782/782 [=====] - 75s 96ms/step - loss: 0.4432 - accuracy: 0.7950
Epoch 3/10
782/782 [=====] - 73s 94ms/step - loss: 0.4070 - accuracy: 0.8150
Epoch 4/10
782/782 [=====] - 67s 86ms/step - loss: 0.3814 - accuracy: 0.8306
Epoch 5/10
782/782 [=====] - 62s 79ms/step - loss: 0.3596 - accuracy: 0.8402
Epoch 6/10
782/782 [=====] - 94s 120ms/step - loss: 0.3383 - accuracy: 0.8506
Epoch 7/10
782/782 [=====] - 69s 89ms/step - loss: 0.3220 - accuracy: 0.8583
Epoch 8/10
782/782 [=====] - 78s 99ms/step - loss: 0.3087 - accuracy: 0.8652
Epoch 9/10
782/782 [=====] - 77s 99ms/step - loss: 0.2957 - accuracy: 0.8730
Epoch 10/10
782/782 [=====] - 75s 96ms/step - loss: 0.2904 - accuracy: 0.8760

Out[9]: <tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x644660450>

```

Next, we will compile and train our model. We have chosen binary cross-entropy here since our output only has two labels: positive and negative. The loss is evaluated over the final output, which indicates the true sentiment over the sentence. As is shown in our results, we have achieved an 87%

accuracy within 10 epochs. There is still large room for improvements, but for a simple RNN showcase, we think the accuracy is already pretty satisfying. Below is a summary of our model.

```
In [11]: 1 model.summary()
```

Model: "sequential"

| Layer (type)                 | Output Shape     | Param # |
|------------------------------|------------------|---------|
| text_vectorization (TextVect | (None, 100)      | 0       |
| embedding (Embedding)        | (None, 100, 128) | 128128  |
| lstm (LSTM)                  | (None, 64)       | 49408   |
| dense (Dense)                | (None, 64)       | 4160    |
| dense_1 (Dense)              | (None, 1)        | 65      |

=====  
Total params: 181,761  
Trainable params: 181,761  
Non-trainable params: 0  
=====

Finally, we will use our test data to test out our model. The model accuracy with our test data is 0.7858. And we inserted a very positive review and the model outputs a value of over 0.98, which is very reasonable. Then we inserted a very negative comment and the model gives us a value that is slightly over 0.04, which is also very reasonable since the sentiment of this review is negative.

```
In [35]: 1 model.evaluate(test_data)
```

782/782 [=====] - ETA: 0s - loss: 0.5206 - accuracy: 0.78 - 14s 17ms/step - loss: 0.5209 - accuracy: 0.7858

Out[35]: [0.5209300518035889, 0.7858399748802185]

```
In [36]: 1 print(model.predict([
2     "i loved it! highly recommend it to everyone looking for a great movie to watch.",
3 ]))
4 print(model.predict([
5     "this was awful and stupid! i hated it so much, it was just the absolute worst.",
6 ]))
```

[[0.986112]]  
[[0.04109952]]

```
In [37]: 1 print(model.predict([
2     "this movie is awful and i love it.",
3 ]))
4 print(model.predict([
5     "this movie is awful but i love it.",
6 ]))
```

[[0.15412727]]  
[[0.45177472]]

One interesting discovery is that when we input a relatively neutral comment like “this is awful and I like it”, the model outputs a value of 0.15, which indicates a pretty negative attitude. However, if we enter “this is awful but I like it”, the value will be 0.45, which is a big difference and it tells us that this model is able to detect minor sentiment changes. With that said, we believe that our simple RNN model produces relatively satisfying results.

## References

[https://en.wikipedia.org/wiki/Recurrent\\_neural\\_network](https://en.wikipedia.org/wiki/Recurrent_neural_network)

<http://abatanasov.com/Files/Deep%20Learning%201.pdf>

[https://en.wikipedia.org/wiki/Long\\_short-term\\_memory](https://en.wikipedia.org/wiki/Long_short-term_memory)

<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>

[http://www.scholarpedia.org/article/Recurrent\\_neural\\_networks](http://www.scholarpedia.org/article/Recurrent_neural_networks)

<https://zybuluo.com/hanbingtao/note/541458>

<https://ai.stanford.edu/~amaas/data/sentiment/>

<https://victorzhou.com/blog/keras-rnn-tutorial/#the-full-code>

<https://www.theverge.com/2020/12/14/22173803/gmail-youtube-google-assistant-docs-down-outage>