Report 2 RNN and LSTM for Text Analysis

24011149X Song Zhengfei

1. Introduction

1.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) represent a specialized neural network architecture designed for the processing of sequential data. Their inherent cyclic structure enables the retention of state information from prior time steps, effectively capturing temporal dependencies within sequences. This capability renders RNNs particularly effective in various applications, including natural language processing, speech recognition, and time series prediction.

RNN networks can be represented as follows:

$$o_t = RNN(x_t, x_{t-1}, \ldots, x_{t-M})$$

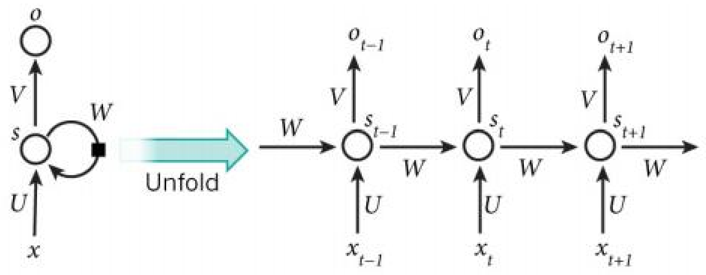


Figure1 structure of RNN and its unfolded framework[1]

x_t is the input at time t, and s_t is the hidden state, representing the memory of the network. The recurrent layer weights, including W, U and V, are shared across time-steps. o_t is the output at step t. And at time state t, RNN makes use of the former inputs and output o_t as the prediction of the next state input x_{t+1}.

RNNs perform well in some short-term memory tasks. However, when it comes to some long-term memory tasks, RNNs have limitations because they are weak in modeling long-term dependency. What's more, if the gap between relevant information is very large, they will be unable to learn the information.

1.2 Long Short-Term Memory (LSTM)

LSTM is a special type of RNN, which solves the problem that basic RNNs can't capture the long-term dependencies. It introduces a key structure named cell state, running through the entire chain. Information can be removed or added to the cell state depending on the input and hidden states.

Based on the previous hidden state and current input, LSTM firstly decides what information to remove. This step can be represented as $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$. Then, LSTM decides what information to add to the cell state by calculating i_t and $\widetilde{C}_t$.

[1] from EIE4122_6_DeepArch-2

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\widetilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Then, LSTM updates the cell state from C_{t-1} to C_t by calculating $f_t \odot C_{t-1}$ as things to forget and $i_t \odot \widetilde{C}_t$ as things to add.

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t$$

At last, LSTM decides the next hidden state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

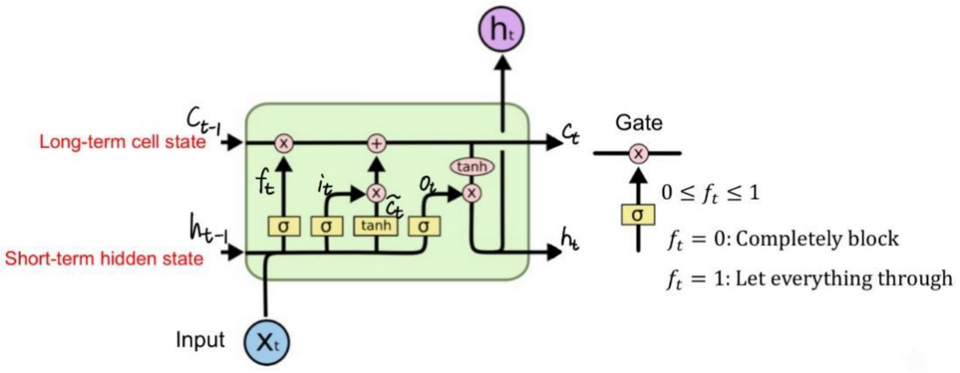


Figure2 structure of LSTM[2]

The bidirectional LSTM (Bi-LSTM) is a type of LSTM, which can model a sequence in both forward and backwards directions. It can utilize the information before and after the state to predict and is very useful for sequence-to-sequence tasks. Utilizing the property of Bi-LSTM, we can expand from predicting the words at the end of a sentence to predicting the words in the middle of a sentence.

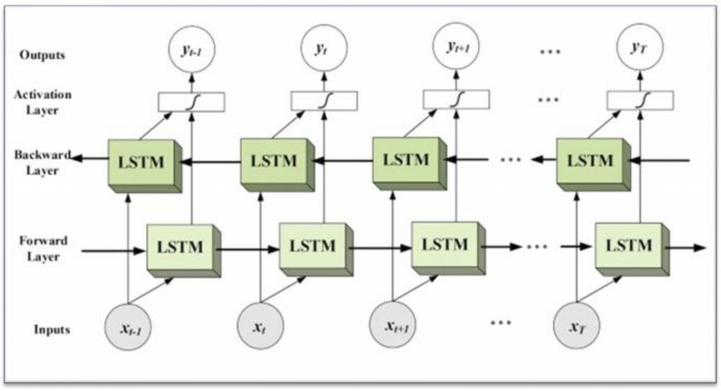


Figure3 structure of Bi-LSTM[3]

Multi-layer LSTMs extends the basic LSTM structure and enhances the model's expressiveness by stacking multiple LSTM layers. The output of each layer serves as the input of the next layer. This structure enables the model to learn more complex features. Compared with single-layer LSTM, multi-layer LSTMs generally provide better performance, especially on large and complex datasets.

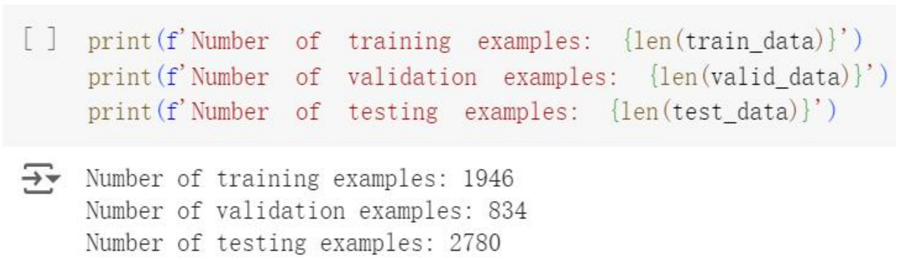
1.3 Experiment abstract

[2] [3] from EIE4122_6_DeepArch-2

In this lab, we carry out our experiments on Google Colab and use PyTorch to develop RNNs for sentiment classification. We first learn how to split datasets and get a deeper understanding of random seed, vocabulary, as well as embeddings through data preprocessing. We can also understand the architectures of RNNs and LSTMs, and the concepts of padding and packing sequences during the experiment.

2. RNN

We separate the original dataset of 5560 movie reviews into 1946 training examples, 834 validation examples, and 2780 testing examples.

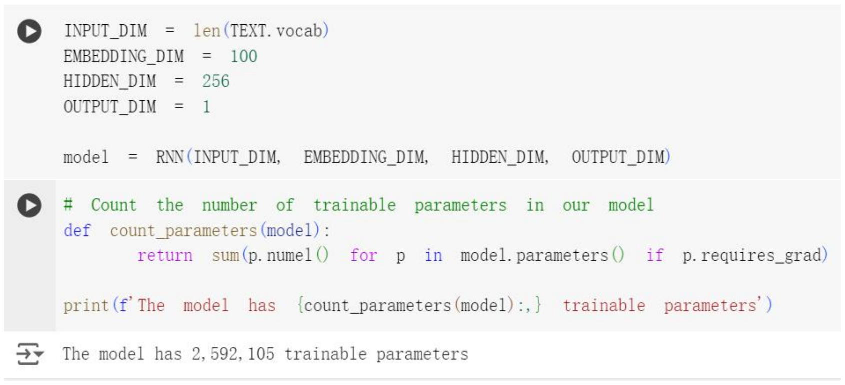


```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 1946
Number of validation examples: 834
Number of testing examples: 2780
```

Figure4 Split the dataset

First, train the basic RNN model for 10 epochs under initial settings, and we can get the results shown in Figure 7 and 8. The basic RNN can achieve 0.697 test loss and 49.89% test accuracy, which means it has a success rate of about half.

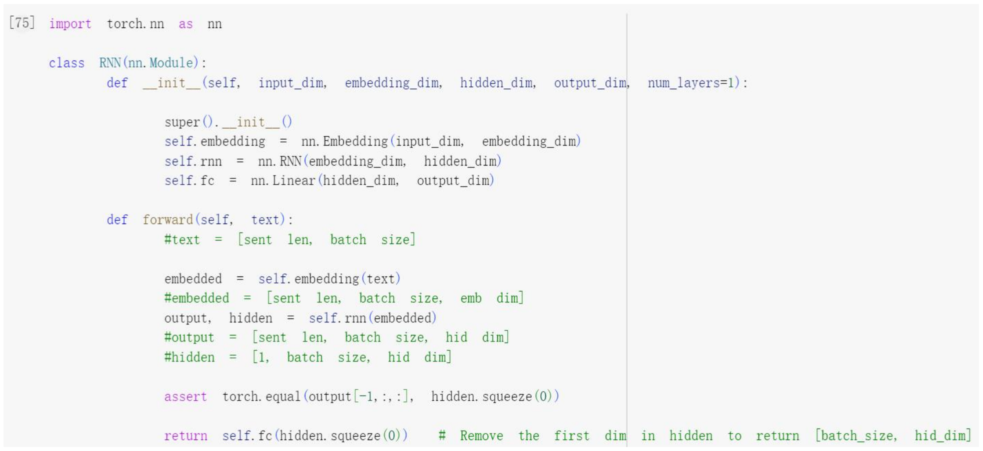


```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

```
# Count the number of trainable parameters in our model
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

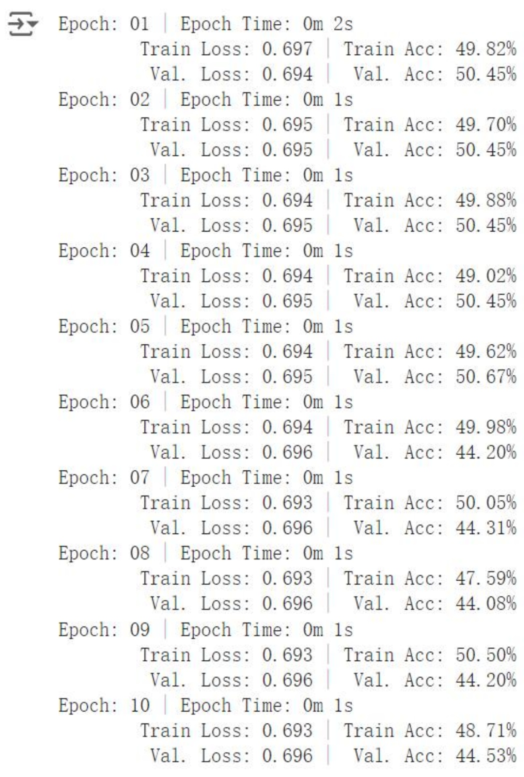
```
The model has 2,592,105 trainable parameters
```

Figure5 Initial settings



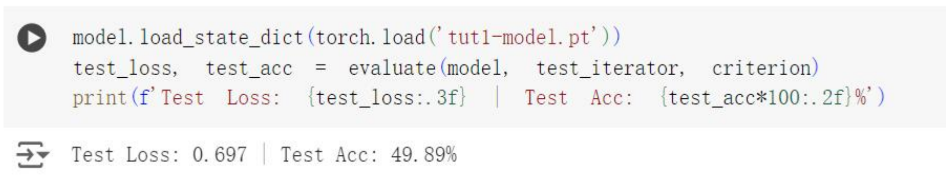
```
[75] import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim, num_layers=1):

        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]
        output, hidden = self.rnn(embedded)
        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))  # Remove the first dim in hidden to return [batch_size, hid_dim]
```

Figure6 Basic RNN structure



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.697 | Train Acc: 49.82%
         Val. Loss: 0.694 |  Val. Acc: 50.45%
Epoch: 02 | Epoch Time: 0m 1s
        Train Loss: 0.695 | Train Acc: 49.70%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.88%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.02%
         Val. Loss: 0.695 |  Val. Acc: 50.45%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.62%
         Val. Loss: 0.695 |  Val. Acc: 50.67%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.98%
         Val. Loss: 0.696 |  Val. Acc: 44.20%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.05%
         Val. Loss: 0.696 |  Val. Acc: 44.31%
Epoch: 08 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 47.59%
         Val. Loss: 0.696 |  Val. Acc: 44.08%
Epoch: 09 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.50%
         Val. Loss: 0.696 |  Val. Acc: 44.20%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 48.71%
         Val. Loss: 0.696 |  Val. Acc: 44.53%
```

Figure7 Training result of basic RNN



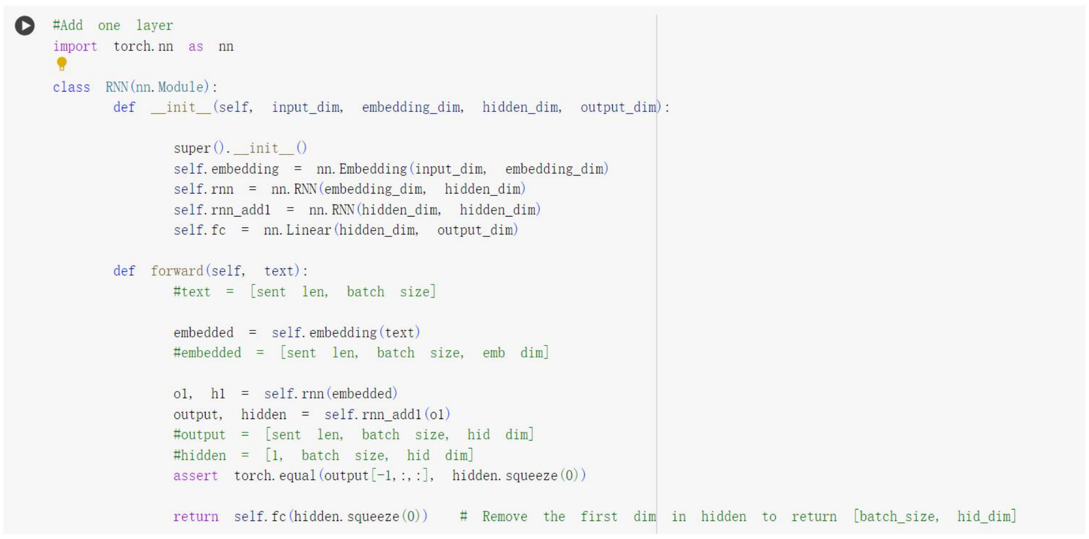
```
model.load_state_dict(torch.load('tut1-model.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.697 | Test Acc: 49.89%
```

Figure8 Testing result of basic RNN

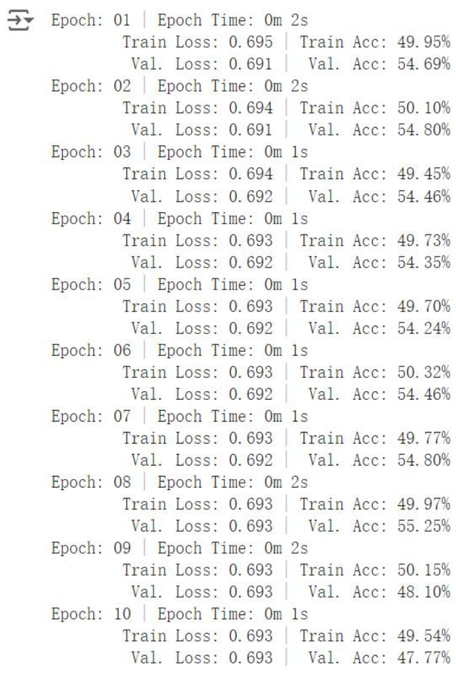
2.1 Increase the number of RNN layers

After adding one layer, I train again and get the results shown in Figure 10 and 11. The new test loss is 0.694, less than before, and the test accuracy is 49.69%. Adding one RNN layer does not help to improve the model performance on sentiment analysis.



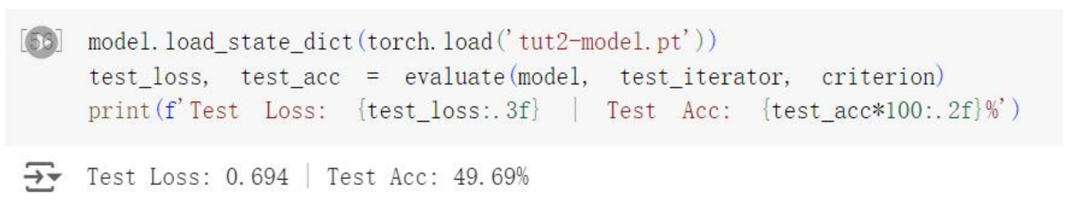
```
#Add one layer
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.rnn_add1 = nn.RNN(hidden_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]

        o1, h1 = self.rnn(embedded)
        output, hidden = self.rnn_add1(o1)
        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]
        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))   # Remove the first dim in hidden to return [batch_size, hid_dim]
```

Figure9 Adding one layer



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.695 | Train Acc: 49.95%
         Val. Loss: 0.691 |  Val. Acc: 54.69%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.694 | Train Acc: 50.10%
         Val. Loss: 0.691 |  Val. Acc: 54.80%
Epoch: 03 | Epoch Time: 0m 1s
        Train Loss: 0.694 | Train Acc: 49.45%
         Val. Loss: 0.692 |  Val. Acc: 54.46%
Epoch: 04 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.73%
         Val. Loss: 0.692 |  Val. Acc: 54.35%
Epoch: 05 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.70%
         Val. Loss: 0.692 |  Val. Acc: 54.24%
Epoch: 06 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 50.32%
         Val. Loss: 0.692 |  Val. Acc: 54.46%
Epoch: 07 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.77%
         Val. Loss: 0.692 |  Val. Acc: 54.80%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.97%
         Val. Loss: 0.693 |  Val. Acc: 55.25%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.15%
         Val. Loss: 0.693 |  Val. Acc: 48.10%
Epoch: 10 | Epoch Time: 0m 1s
        Train Loss: 0.693 | Train Acc: 49.54%
         Val. Loss: 0.693 |  Val. Acc: 47.77%
```

Figure10 Training result after adding one RNN layer

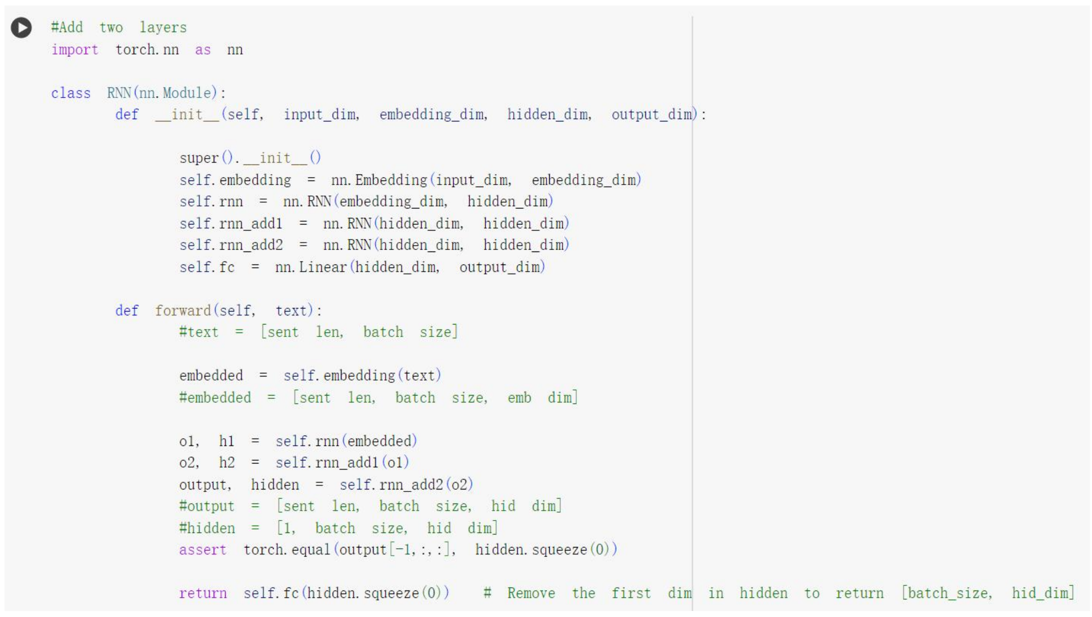


```
[56] model.load_state_dict(torch.load('tut2-model.pt'))
     test_loss, test_acc = evaluate(model, test_iterator, criterion)
     print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.694 | Test Acc: 49.69%
```

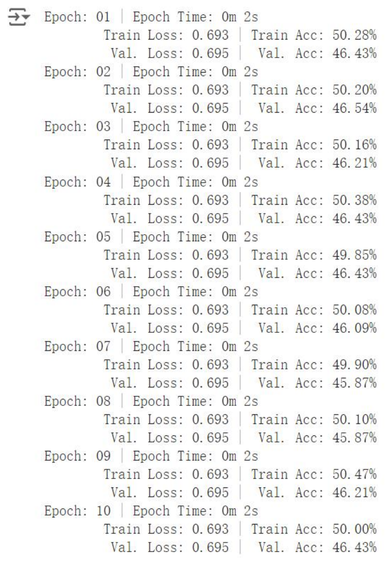
Figure11 Testing result after adding one RNN layer

I also try to add 2 RNN layers. This time, the model performs better than before, with testing loss of 0.693 and testing accuracy of 50.01%.



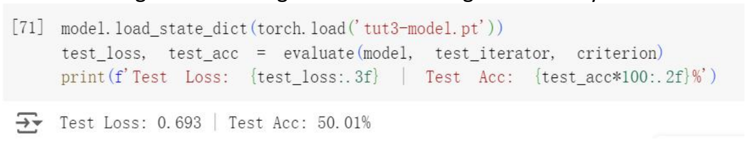
```
#Add two layers
import torch.nn as nn

class RNN(nn.Module):
        def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

                super().__init__()
                self.embedding = nn.Embedding(input_dim, embedding_dim)
                self.rnn = nn.RNN(embedding_dim, hidden_dim)
                self.rnn_add1 = nn.RNN(hidden_dim, hidden_dim)
                self.rnn_add2 = nn.RNN(hidden_dim, hidden_dim)
                self.fc = nn.Linear(hidden_dim, output_dim)

        def forward(self, text):
                #text = [sent len, batch size]

                embedded = self.embedding(text)
                #embedded = [sent len, batch size, emb dim]

                o1, h1 = self.rnn(embedded)
                o2, h2 = self.rnn_add1(o1)
                output, hidden = self.rnn_add2(o2)
                #output = [sent len, batch size, hid dim]
                #hidden = [1, batch size, hid dim]
                assert torch.equal(output[-1,:,:], hidden.squeeze(0))

                return self.fc(hidden.squeeze(0))  # Remove the first dim in hidden to return [batch_size, hid_dim]
```

Figure12 Adding two layers



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.28%
         Val. Loss: 0.695 |  Val. Acc: 46.43%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.20%
         Val. Loss: 0.695 |  Val. Acc: 46.54%
Epoch: 03 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.16%
         Val. Loss: 0.695 |  Val. Acc: 46.21%
Epoch: 04 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.38%
         Val. Loss: 0.695 |  Val. Acc: 46.43%
Epoch: 05 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.85%
         Val. Loss: 0.695 |  Val. Acc: 46.43%
Epoch: 06 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.08%
         Val. Loss: 0.695 |  Val. Acc: 46.09%
Epoch: 07 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.90%
         Val. Loss: 0.695 |  Val. Acc: 45.87%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.10%
         Val. Loss: 0.695 |  Val. Acc: 45.87%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.47%
         Val. Loss: 0.695 |  Val. Acc: 46.21%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.00%
         Val. Loss: 0.695 |  Val. Acc: 46.43%
```

Figure13 Training result after adding two RNN layers



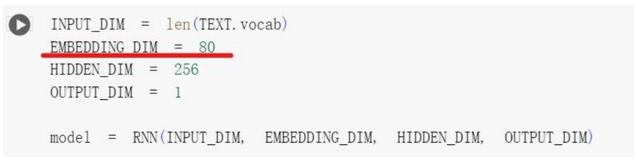
```
[71] model.load_state_dict(torch.load('tut3-model.pt'))
     test_loss, test_acc = evaluate(model, test_iterator, criterion)
     print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.693 | Test Acc: 50.01%
```

Figure14 Testing result after adding two RNN layers

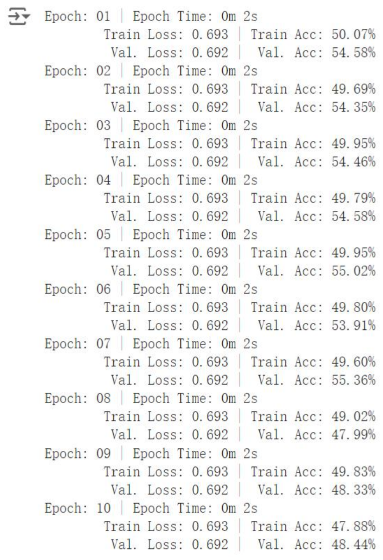
2.2 Change embedding dimension

Embedding dimension also influences the model’s performance. It specifies the dimensions of the continuous vector to which the discrete input values are mapped. Increasing the embedding dimension will give the model more information to calculate and represent, improving the model’s representation ability while increasing the computational parameters. However, large embedding dimension may make the model overfit, which will make its performance worse.



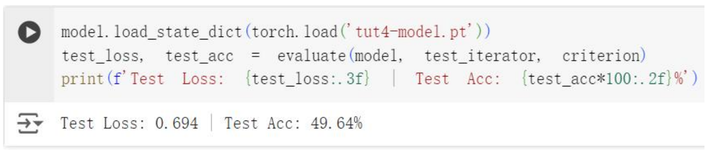
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 80
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Figure15 Change EMBEDDING_DIM to 80



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.07%
         Val. Loss: 0.692 |  Val. Acc: 54.58%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.69%
         Val. Loss: 0.692 |  Val. Acc: 54.35%
Epoch: 03 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.95%
         Val. Loss: 0.692 |  Val. Acc: 54.46%
Epoch: 04 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.79%
         Val. Loss: 0.692 |  Val. Acc: 54.58%
Epoch: 05 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.95%
         Val. Loss: 0.692 |  Val. Acc: 55.02%
Epoch: 06 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.80%
         Val. Loss: 0.692 |  Val. Acc: 53.91%
Epoch: 07 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.60%
         Val. Loss: 0.692 |  Val. Acc: 55.36%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.02%
         Val. Loss: 0.692 |  Val. Acc: 47.99%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.83%
         Val. Loss: 0.692 |  Val. Acc: 48.33%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 47.88%
         Val. Loss: 0.692 |  Val. Acc: 48.44%
```

Figure16 Training result with embedding dimension of 80

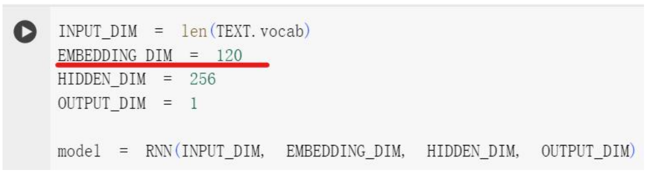


```
model.load_state_dict(torch.load('tut4-model.pt'))
test_loss,  test_acc  =  evaluate(model,  test_iterator,  criterion)
print(f'Test  Loss:  {test_loss:.3f}  |  Test  Acc:  {test_acc*100:.2f}%')
```

```
Test Loss: 0.694 | Test Acc: 49.64%
```

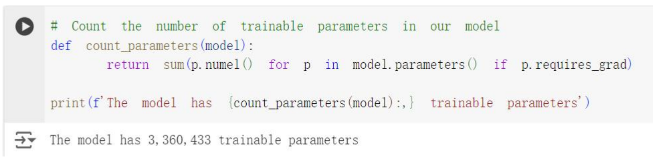
Figure17 Testing result with embedding dimension of 80

I change the embedding dimension from 100 to 80, 120, 140, 160 based on the model structure after adding two RNN layers, to improve its performance. When the embedding dimension decreases to 80, both its trainable parameters and its testing accuracy decrease, as shown in Figure 15 to 17. When the embedding dimension increases to 120, its trainable parameters become larger, with the testing accuracy increasing to 50.29%, as shown in Figure 18 to 21.



```
INPUT_DIM  =  len(TEXT.vocab)
EMBEDDING_DIM  =  120
HIDDEN_DIM  =  256
OUTPUT_DIM  =  1

model  =  RNN(INPUT_DIM,  EMBEDDING_DIM,  HIDDEN_DIM,  OUTPUT_DIM)
```

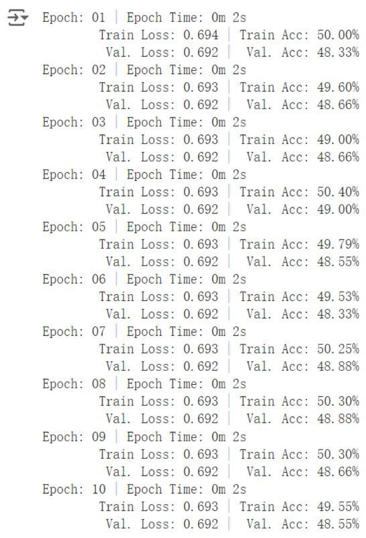
Figure18 Change EMBEDDING_DIM to 120



```
# Count  the  number  of  trainable  parameters  in  our  model
def  count_parameters(model):
      return  sum(p.numel()  for  p  in  model.parameters()  if  p.requires_grad)

print(f'The  model  has  {count_parameters(model):,}  trainable  parameters')
```

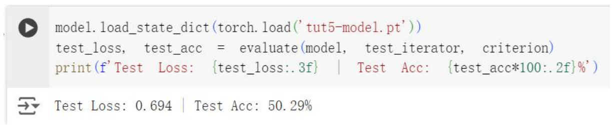
```
The model has 3,360,433 trainable parameters
```

Figure19 Trainable parameters with embedding dimension of 120



```
Epoch: 01 | Epoch Time: 0m 2s
	Train Loss: 0.694 | Train Acc: 50.00%
	 Val. Loss: 0.692 |  Val. Acc: 48.33%
Epoch: 02 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.60%
	 Val. Loss: 0.692 |  Val. Acc: 48.66%
Epoch: 03 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.00%
	 Val. Loss: 0.692 |  Val. Acc: 48.66%
Epoch: 04 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.40%
	 Val. Loss: 0.692 |  Val. Acc: 49.00%
Epoch: 05 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.79%
	 Val. Loss: 0.692 |  Val. Acc: 48.55%
Epoch: 06 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.53%
	 Val. Loss: 0.692 |  Val. Acc: 48.33%
Epoch: 07 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.25%
	 Val. Loss: 0.692 |  Val. Acc: 48.88%
Epoch: 08 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.30%
	 Val. Loss: 0.692 |  Val. Acc: 48.88%
Epoch: 09 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.30%
	 Val. Loss: 0.692 |  Val. Acc: 48.66%
Epoch: 10 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.55%
	 Val. Loss: 0.692 |  Val. Acc: 48.55%
```

Figure20 Training result with embedding dimension of 120

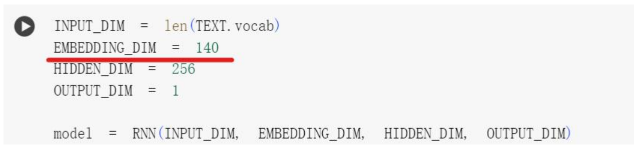


```
model.load_state_dict(torch.load('tut5-model.pt'))
test_loss,  test_acc  =  evaluate(model,  test_iterator,  criterion)
print(f'Test  Loss:  {test_loss:.3f}  |  Test  Acc:  {test_acc*100:.2f}%')
```

```
Test Loss: 0.694 | Test Acc: 50.29%
```

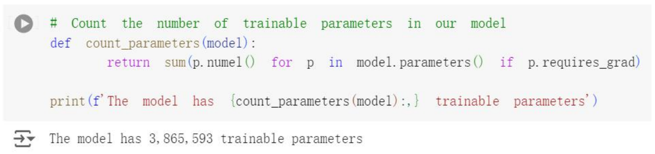
Figure21 Testing result with embedding dimension of 120

It still helps improve model's performance when embedding dimension is increased to 140, with testing accuracy of 51.35%. When I change embedding dimension to 160, the model's testing accuracy becomes 50.03%. It is very likely that overfitting has occurred, resulting in a decrease in model performance.



```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 140
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

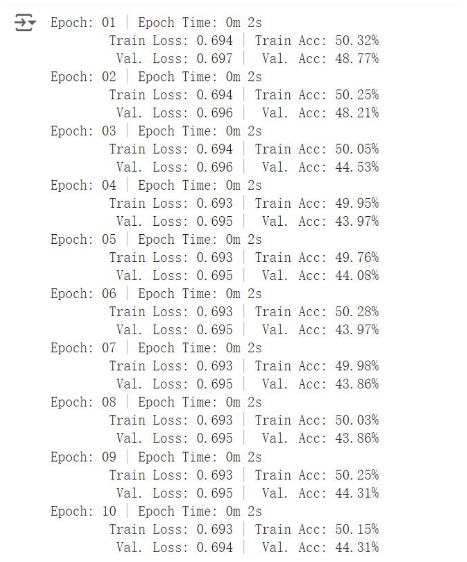
Figure22 Change EMBEDDING_DIM to 140



```
# Count the number of trainable parameters in our model
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

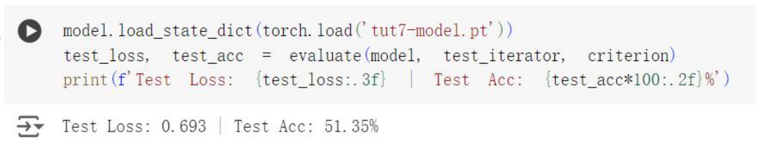
```
The model has 3,865,593 trainable parameters
```

Figure23 Trainable parameters with embedding dimension of 140



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.694 | Train Acc: 50.32%
         Val. Loss: 0.697 |  Val. Acc: 48.77%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.694 | Train Acc: 50.25%
         Val. Loss: 0.696 |  Val. Acc: 48.21%
Epoch: 03 | Epoch Time: 0m 2s
        Train Loss: 0.694 | Train Acc: 50.05%
         Val. Loss: 0.696 |  Val. Acc: 44.53%
Epoch: 04 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.95%
         Val. Loss: 0.695 |  Val. Acc: 43.97%
Epoch: 05 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.76%
         Val. Loss: 0.695 |  Val. Acc: 44.08%
Epoch: 06 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.28%
         Val. Loss: 0.695 |  Val. Acc: 43.97%
Epoch: 07 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 49.98%
         Val. Loss: 0.695 |  Val. Acc: 43.86%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.03%
         Val. Loss: 0.695 |  Val. Acc: 43.86%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.25%
         Val. Loss: 0.695 |  Val. Acc: 44.31%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.15%
         Val. Loss: 0.694 |  Val. Acc: 44.31%
```

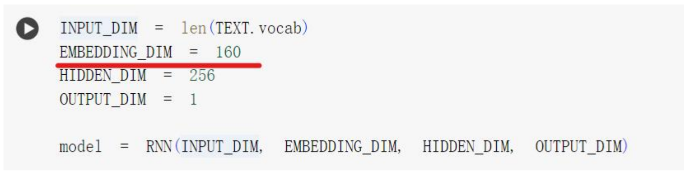
Figure24 Training result with embedding dimension of 140



```
model.load_state_dict(torch.load('tut7-model.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

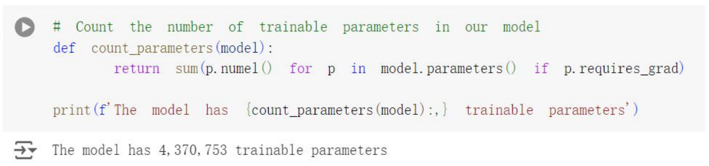
```
Test Loss: 0.693 | Test Acc: 51.35%
```

Figure25 Testing result with embedding dimension of 140



```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 160
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

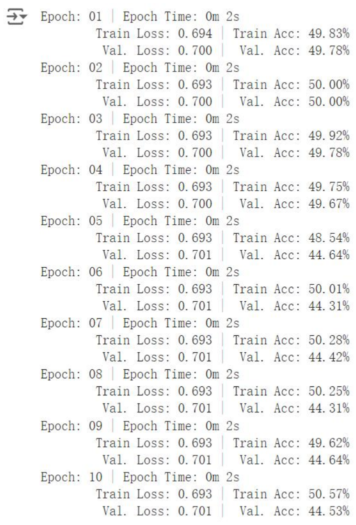
Figure26 Change EMBEDDING_DIM to 160



```
# Count the number of trainable parameters in our model
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

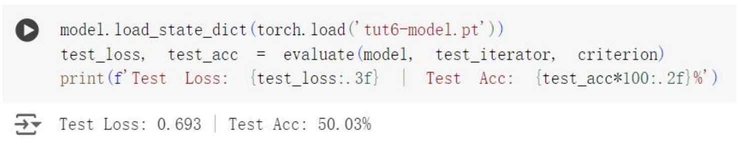
```
The model has 4,370,753 trainable parameters
```

Figure27 Trainable parameters with embedding dimension of 160



```
Epoch: 01 | Epoch Time: 0m 2s
	Train Loss: 0.694 | Train Acc: 49.83%
	 Val. Loss: 0.700 |  Val. Acc: 49.78%
Epoch: 02 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.00%
	 Val. Loss: 0.700 |  Val. Acc: 50.00%
Epoch: 03 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.92%
	 Val. Loss: 0.700 |  Val. Acc: 49.78%
Epoch: 04 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.75%
	 Val. Loss: 0.700 |  Val. Acc: 49.67%
Epoch: 05 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 48.54%
	 Val. Loss: 0.701 |  Val. Acc: 44.64%
Epoch: 06 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.01%
	 Val. Loss: 0.701 |  Val. Acc: 44.31%
Epoch: 07 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.28%
	 Val. Loss: 0.701 |  Val. Acc: 44.42%
Epoch: 08 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.25%
	 Val. Loss: 0.701 |  Val. Acc: 44.31%
Epoch: 09 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 49.62%
	 Val. Loss: 0.701 |  Val. Acc: 44.64%
Epoch: 10 | Epoch Time: 0m 2s
	Train Loss: 0.693 | Train Acc: 50.57%
	 Val. Loss: 0.701 |  Val. Acc: 44.53%
```

Figure28 Training result with embedding dimension of 160



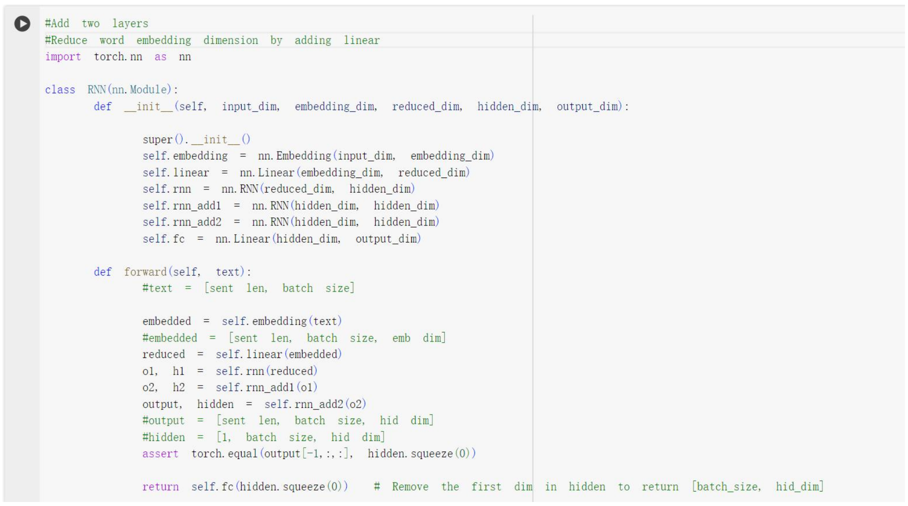
```
model.load_state_dict(torch.load('tut6-model.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.693 | Test Acc: 50.03%
```

Figure29 Testing result with embedding dimension of 160

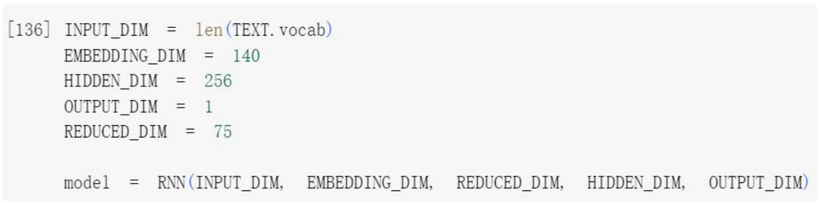
2.3 Reduce the Dimension of the Word Embeddings

Reducing dimension of embeddings after the word vector is generated, can also help remove redundant information to improve efficiency and performance of the model. Based on the model structure with two additional RNN layers and embedding dimension of 140, I add a linear processing to reduce the dimension of word embedding and improve the testing accuracy to 51.17%.



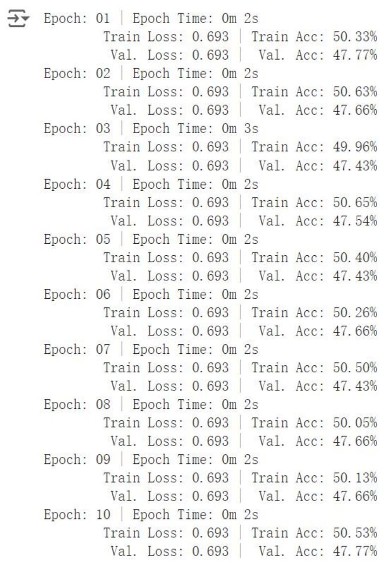
```
#Add two layers
#Reduce word embedding dimension by adding linear
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, reduced_dim, hidden_dim, output_dim):

        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.linear = nn.Linear(embedding_dim, reduced_dim)
        self.rnn = nn.RNN(reduced_dim, hidden_dim)
        self.rnn_add1 = nn.RNN(hidden_dim, hidden_dim)
        self.rnn_add2 = nn.RNN(hidden_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]
        reduced = self.linear(embedded)
        o1, h1 = self.rnn(reduced)
        o2, h2 = self.rnn_add1(o1)
        output, hidden = self.rnn_add2(o2)
        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]
        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))  # Remove the first dim in hidden to return [batch_size, hid_dim]
```

Figure30 Add linear processing to reduce word embedding dimension



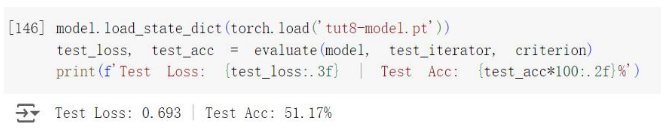
```
[136] INPUT_DIM = len(TEXT.vocab)
      EMBEDDING_DIM = 140
      HIDDEN_DIM = 256
      OUTPUT_DIM = 1
      REDUCED_DIM = 75

      model = RNN(INPUT_DIM, EMBEDDING_DIM, REDUCED_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Figure31 Dimension settings



```
Epoch: 01 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.33%
         Val. Loss: 0.693 |  Val. Acc: 47.77%
Epoch: 02 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.63%
         Val. Loss: 0.693 |  Val. Acc: 47.66%
Epoch: 03 | Epoch Time: 0m 3s
        Train Loss: 0.693 | Train Acc: 49.96%
         Val. Loss: 0.693 |  Val. Acc: 47.43%
Epoch: 04 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.65%
         Val. Loss: 0.693 |  Val. Acc: 47.54%
Epoch: 05 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.40%
         Val. Loss: 0.693 |  Val. Acc: 47.43%
Epoch: 06 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.26%
         Val. Loss: 0.693 |  Val. Acc: 47.66%
Epoch: 07 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.50%
         Val. Loss: 0.693 |  Val. Acc: 47.43%
Epoch: 08 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.05%
         Val. Loss: 0.693 |  Val. Acc: 47.66%
Epoch: 09 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.13%
         Val. Loss: 0.693 |  Val. Acc: 47.66%
Epoch: 10 | Epoch Time: 0m 2s
        Train Loss: 0.693 | Train Acc: 50.53%
         Val. Loss: 0.693 |  Val. Acc: 47.77%
```

Figure32 Training result

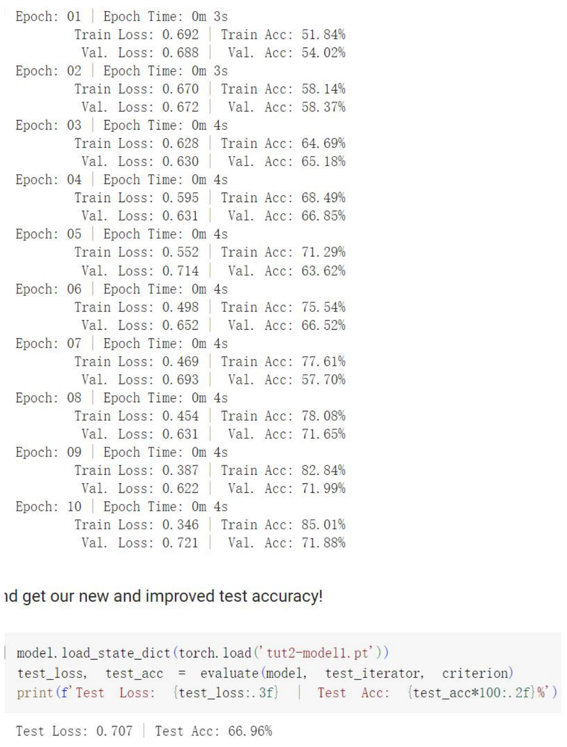


```
[146] model.load_state_dict(torch.load('tut8-model.pt'))
      test_loss, test_acc = evaluate(model, test_iterator, criterion)
      print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')

Test Loss: 0.693 | Test Acc: 51.17%
```

Figure33 Testing result

3. LSTM

First, train the LSTM model for 10 epochs under initial settings. It achieves testing loss of 0.707 and testing accuracy of 66.96%. When it is applied to analyzed the reviews, 'This film is bad.' and 'This film is good.', its result makes sense, with about 0.21 (less than half) for a negative review and 0.63 (more than half) for a positive review.

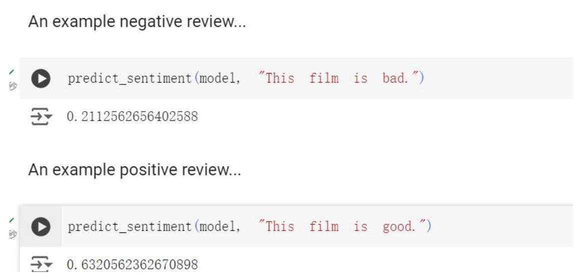


```
Epoch: 01 | Epoch Time: 0m 3s
        Train Loss: 0.692 | Train Acc: 51.84%
         Val. Loss: 0.688 |  Val. Acc: 54.02%
Epoch: 02 | Epoch Time: 0m 3s
        Train Loss: 0.670 | Train Acc: 58.14%
         Val. Loss: 0.672 |  Val. Acc: 58.37%
Epoch: 03 | Epoch Time: 0m 4s
        Train Loss: 0.628 | Train Acc: 64.69%
         Val. Loss: 0.630 |  Val. Acc: 65.18%
Epoch: 04 | Epoch Time: 0m 4s
        Train Loss: 0.595 | Train Acc: 68.49%
         Val. Loss: 0.631 |  Val. Acc: 66.85%
Epoch: 05 | Epoch Time: 0m 4s
        Train Loss: 0.552 | Train Acc: 71.29%
         Val. Loss: 0.714 |  Val. Acc: 63.62%
Epoch: 06 | Epoch Time: 0m 4s
        Train Loss: 0.498 | Train Acc: 75.54%
         Val. Loss: 0.652 |  Val. Acc: 66.52%
Epoch: 07 | Epoch Time: 0m 4s
        Train Loss: 0.469 | Train Acc: 77.61%
         Val. Loss: 0.693 |  Val. Acc: 57.70%
Epoch: 08 | Epoch Time: 0m 4s
        Train Loss: 0.454 | Train Acc: 78.08%
         Val. Loss: 0.631 |  Val. Acc: 71.65%
Epoch: 09 | Epoch Time: 0m 4s
        Train Loss: 0.387 | Train Acc: 82.84%
         Val. Loss: 0.622 |  Val. Acc: 71.99%
Epoch: 10 | Epoch Time: 0m 4s
        Train Loss: 0.346 | Train Acc: 85.01%
         Val. Loss: 0.721 |  Val. Acc: 71.88%
```

ıd get our new and improved test accuracy!

```
model.load_state_dict(torch.load('tut2-model1.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')

Test Loss: 0.707 | Test Acc: 66.96%
```

Figure34 Training and testing result of LSTM under initial setting



An example negative review...

```
predict_sentiment(model, "This film is bad.")
```

```
0.2112562656402588
```

An example positive review...

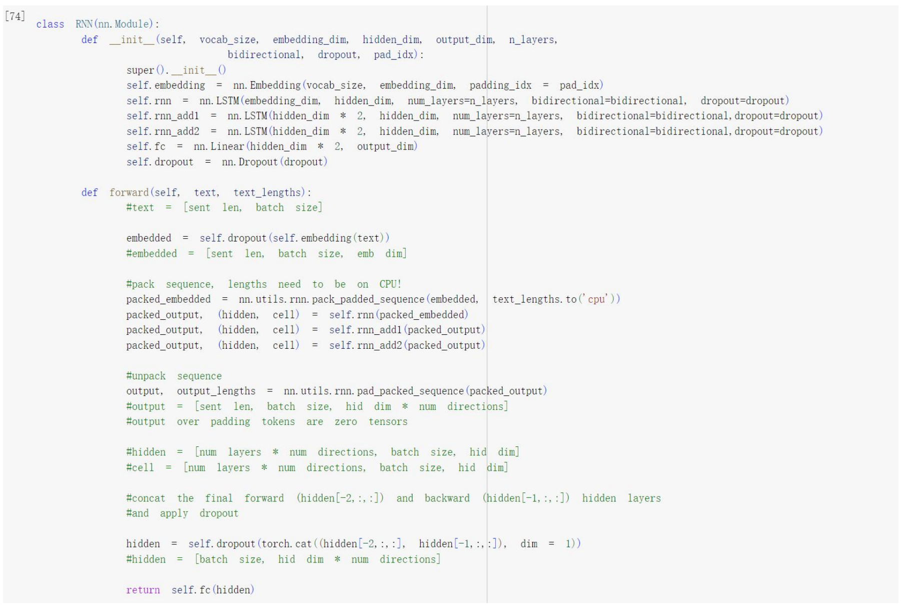
```
predict_sentiment(model, "This film is good.")
```

```
0.6320562362670898
```

Figure35 Example of sentiment analysis

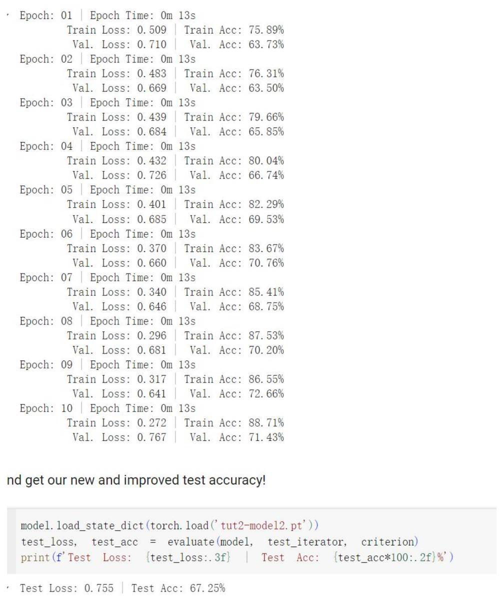
3.1 Increase the number of LSTM layers

Then, I increase the layer numbers of LSTM and train it for 10 epochs. Testing accuracy of LSTM is increased to 67.25%.



```
[74]
class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=dropout)
        self.rnn_add1 = nn.LSTM(hidden_dim * 2, hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=dropout)
        self.rnn_add2 = nn.LSTM(hidden_dim * 2, hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.dropout(self.embedding(text))
        #embedded = [sent len, batch size, emb dim]

        #pack sequence, lengths need to be on CPU!
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        packed_output, (hidden, cell) = self.rnn_add1(packed_output)
        packed_output, (hidden, cell) = self.rnn_add2(packed_output)

        #unpack sequence
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
        #output = [sent len, batch size, hid dim * num directions]
        #output over padding tokens are zero tensors

        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]

        #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
        #and apply dropout

        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        #hidden = [batch size, hid dim * num directions]

        return self.fc(hidden)
```

Figure36 Increase two layers



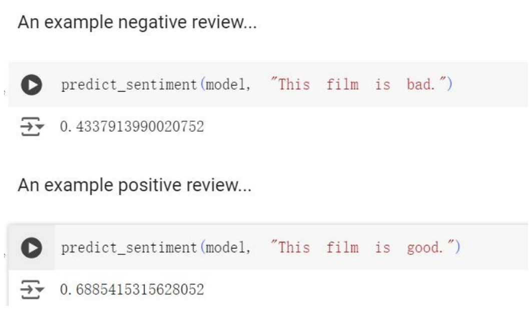
```
Epoch: 01 | Epoch Time: 0m 13s
	Train Loss: 0.509 | Train Acc: 75.89%
	 Val. Loss: 0.710 |  Val. Acc: 63.73%
Epoch: 02 | Epoch Time: 0m 13s
	Train Loss: 0.483 | Train Acc: 76.31%
	 Val. Loss: 0.669 |  Val. Acc: 63.50%
Epoch: 03 | Epoch Time: 0m 13s
	Train Loss: 0.439 | Train Acc: 79.66%
	 Val. Loss: 0.684 |  Val. Acc: 65.85%
Epoch: 04 | Epoch Time: 0m 13s
	Train Loss: 0.432 | Train Acc: 80.04%
	 Val. Loss: 0.726 |  Val. Acc: 66.74%
Epoch: 05 | Epoch Time: 0m 13s
	Train Loss: 0.401 | Train Acc: 82.29%
	 Val. Loss: 0.685 |  Val. Acc: 69.53%
Epoch: 06 | Epoch Time: 0m 13s
	Train Loss: 0.370 | Train Acc: 83.67%
	 Val. Loss: 0.660 |  Val. Acc: 70.76%
Epoch: 07 | Epoch Time: 0m 13s
	Train Loss: 0.340 | Train Acc: 85.41%
	 Val. Loss: 0.646 |  Val. Acc: 68.75%
Epoch: 08 | Epoch Time: 0m 13s
	Train Loss: 0.296 | Train Acc: 87.53%
	 Val. Loss: 0.681 |  Val. Acc: 70.20%
Epoch: 09 | Epoch Time: 0m 13s
	Train Loss: 0.317 | Train Acc: 86.55%
	 Val. Loss: 0.641 |  Val. Acc: 72.66%
Epoch: 10 | Epoch Time: 0m 13s
	Train Loss: 0.272 | Train Acc: 88.71%
	 Val. Loss: 0.767 |  Val. Acc: 71.43%
```

nd get our new and improved test accuracy!

```
model.load_state_dict(torch.load('tut2-model2.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.755 | Test Acc: 67.25%
```

Figure37 Training and testing result after increasing two layers



An example negative review...

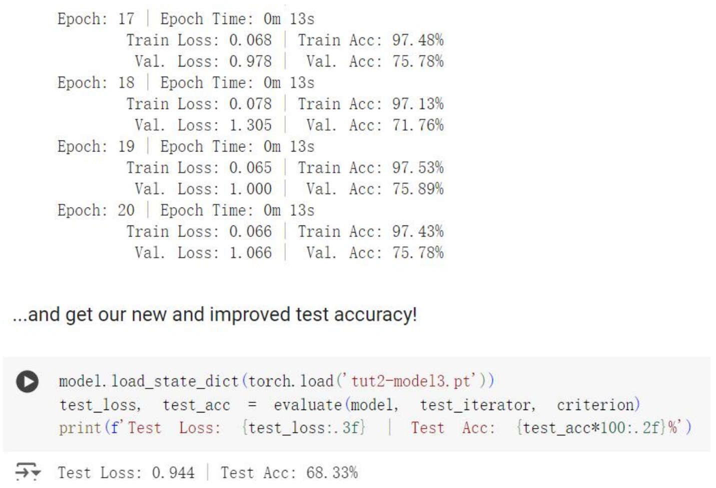
```
predict_sentiment(model, "This film is bad.")
0.4337913990020752
```

An example positive review...

```
predict_sentiment(model, "This film is good.")
0.6885415315628052
```

Figure38 Application example after increasing two layers

However, the application example indicates that the model’s capacity still need to be improved, so I train the LSTM after adding two layers for 20 epochs. This improves testing accuracy to 68.33%. And the LSTM performs very good on the negative example, with the result very close to zero.

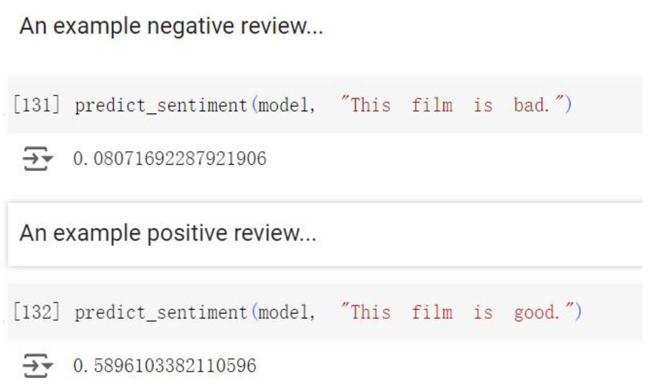


```
Epoch: 17 | Epoch Time: 0m 13s
        Train Loss: 0.068 | Train Acc: 97.48%
         Val. Loss: 0.978 |  Val. Acc: 75.78%
Epoch: 18 | Epoch Time: 0m 13s
        Train Loss: 0.078 | Train Acc: 97.13%
         Val. Loss: 1.305 |  Val. Acc: 71.76%
Epoch: 19 | Epoch Time: 0m 13s
        Train Loss: 0.065 | Train Acc: 97.53%
         Val. Loss: 1.000 |  Val. Acc: 75.89%
Epoch: 20 | Epoch Time: 0m 13s
        Train Loss: 0.066 | Train Acc: 97.43%
         Val. Loss: 1.066 |  Val. Acc: 75.78%
```

...and get our new and improved test accuracy!

```
model.load_state_dict(torch.load('tut2-model3.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')

Test Loss: 0.944 | Test Acc: 68.33%
```

Figure39 Training and testing result



An example negative review...

```
[131] predict_sentiment(model, "This film is bad.")
0.08071692287921906
```

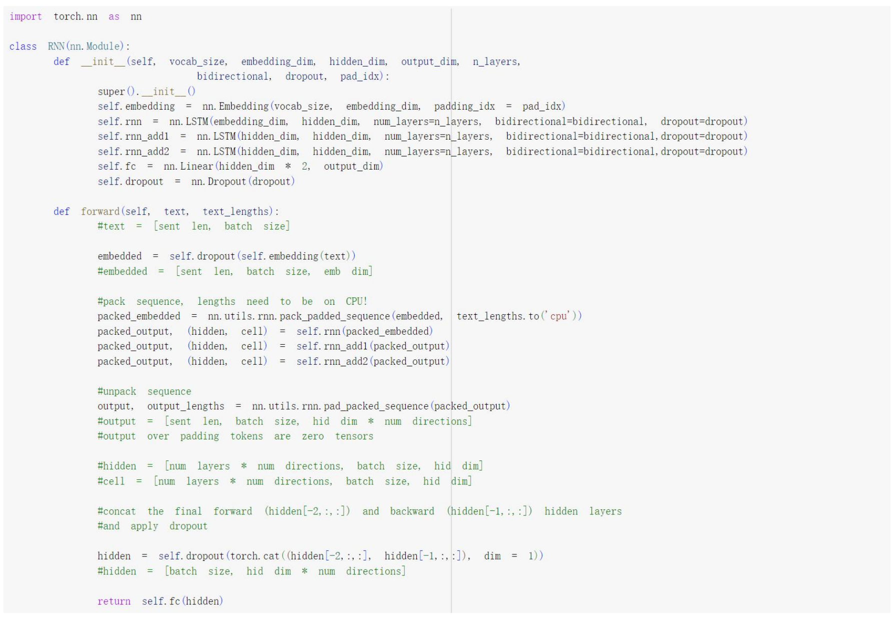
An example positive review...

```
[132] predict_sentiment(model, "This film is good.")
0.5896103382110596
```

Figure40 Application example

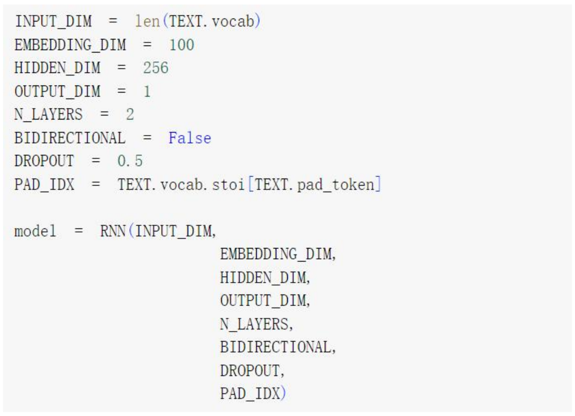
3.2 Use a uni-directional LSTM

Bidirectional LSTM is able to model sequences in both forward and backward directions, making use of the information before and after the state. Changing the bidirectional LSTM to a uni-directional LSTM will theoretically weaken the performance of the model.



```python
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers, bidirectional=bidirectional, dropout=dropout)
        self.rnn_add1 = nn.LSTM(hidden_dim, hidden_dim, num_layers=n_layers, bidirectional=bidirectional,dropout=dropout)
        self.rnn_add2 = nn.LSTM(hidden_dim, hidden_dim, num_layers=n_layers, bidirectional=bidirectional,dropout=dropout)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.dropout(self.embedding(text))
        #embedded = [sent len, batch size, emb dim]

        #pack sequence, lengths need to be on CPU!
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths.to('cpu'))
        packed_output, (hidden, cell) = self.rnn(packed_embedded)
        packed_output, (hidden, cell) = self.rnn_add1(packed_output)
        packed_output, (hidden, cell) = self.rnn_add2(packed_output)

        #unpack sequence
        output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
        #output = [sent len, batch size, hid dim * num directions]
        #output over padding tokens are zero tensors

        #hidden = [num layers * num directions, batch size, hid dim]
        #cell = [num layers * num directions, batch size, hid dim]

        #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
        #and apply dropout

        hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        #hidden = [batch size, hid dim * num directions]

        return self.fc(hidden)
```

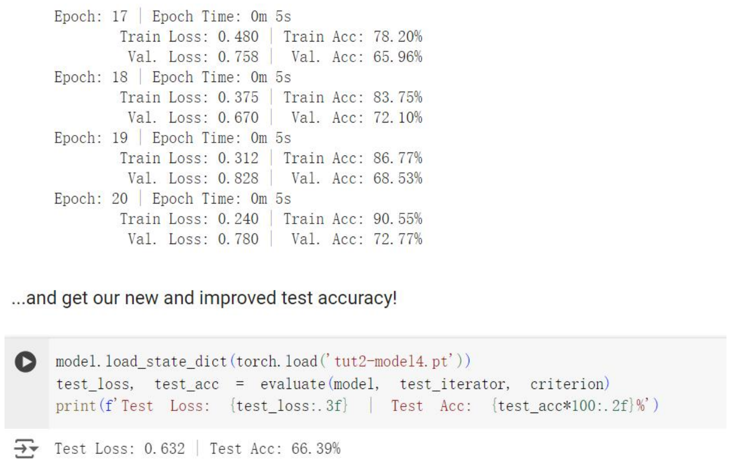
Figure41 Use uni-directional LSTM



```python
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = False
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure42 Set BIDIRECTIONAL to FALSE

The testing result decreases to 66.39%, which has proven the theory. We are more aware of the role of bidirectional LSTM.



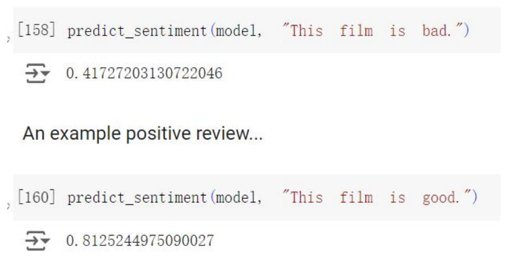
```
Epoch: 17 | Epoch Time: 0m 5s
        Train Loss: 0.480 | Train Acc: 78.20%
         Val. Loss: 0.758 |  Val. Acc: 65.96%
Epoch: 18 | Epoch Time: 0m 5s
        Train Loss: 0.375 | Train Acc: 83.75%
         Val. Loss: 0.670 |  Val. Acc: 72.10%
Epoch: 19 | Epoch Time: 0m 5s
        Train Loss: 0.312 | Train Acc: 86.77%
         Val. Loss: 0.828 |  Val. Acc: 68.53%
Epoch: 20 | Epoch Time: 0m 5s
        Train Loss: 0.240 | Train Acc: 90.55%
         Val. Loss: 0.780 |  Val. Acc: 72.77%
```

...and get our new and improved test accuracy!

```python
model.load_state_dict(torch.load('tut2-model4.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.632 | Test Acc: 66.39%
```

Figure43 Training and testing result of uni-directional LSTM



```
[158] predict_sentiment(model, "This film is bad.")
0.41727203130722046
```

An example positive review...

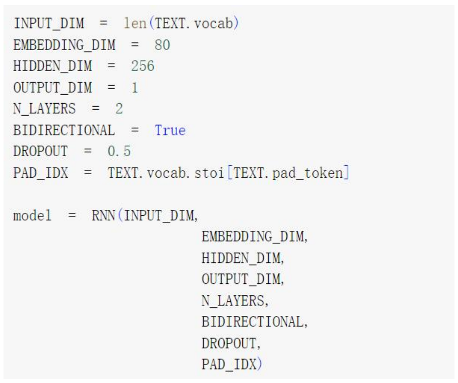
```
[160] predict_sentiment(model, "This film is good.")
0.8125244975090027
```

Figure44 Application example of uni-directional LSTM

3.3 Change the embedding dimension

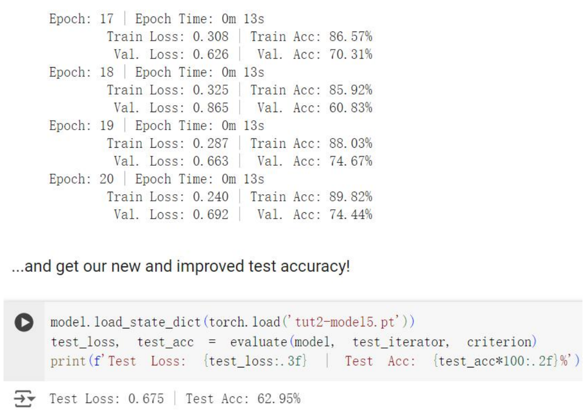
Then, I change the embedding dimension. Similar to RNNs, LSTM's performance becomes worse when I decrease embedding dimension, and LSTM performs better when the embedding dimension is larger.

When the embedding dimension is decreased to 80, the model's trainable parameters become less and its testing accuracy becomes 62.95%.



```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 80
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure45 Change EMBEDDING_DIM to 80

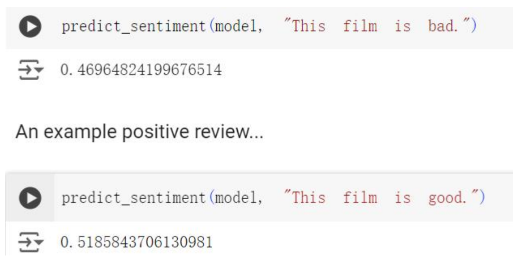


```
Epoch: 17 | Epoch Time: 0m 13s
	Train Loss: 0.308 | Train Acc: 86.57%
	 Val. Loss: 0.626 |  Val. Acc: 70.31%
Epoch: 18 | Epoch Time: 0m 13s
	Train Loss: 0.325 | Train Acc: 85.92%
	 Val. Loss: 0.865 |  Val. Acc: 60.83%
Epoch: 19 | Epoch Time: 0m 13s
	Train Loss: 0.287 | Train Acc: 88.03%
	 Val. Loss: 0.663 |  Val. Acc: 74.67%
Epoch: 20 | Epoch Time: 0m 13s
	Train Loss: 0.240 | Train Acc: 89.82%
	 Val. Loss: 0.692 |  Val. Acc: 74.44%
```

...and get our new and improved test accuracy!

```
model.load_state_dict(torch.load('tut2-model5.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')

Test Loss: 0.675 | Test Acc: 62.95%
```

Figure46 Training and testing result with embedding dimension of 80



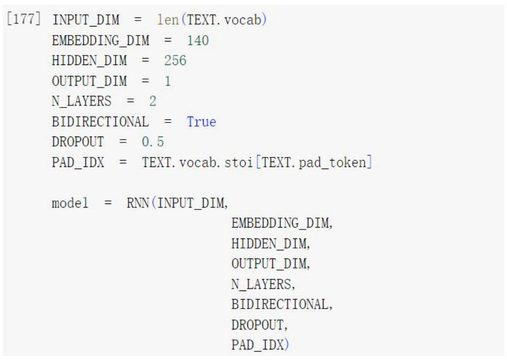
```
predict_sentiment(model, "This film is bad.")
0.46964824199676514
```

An example positive review...

```
predict_sentiment(model, "This film is good.")
0.5185843706130981
```

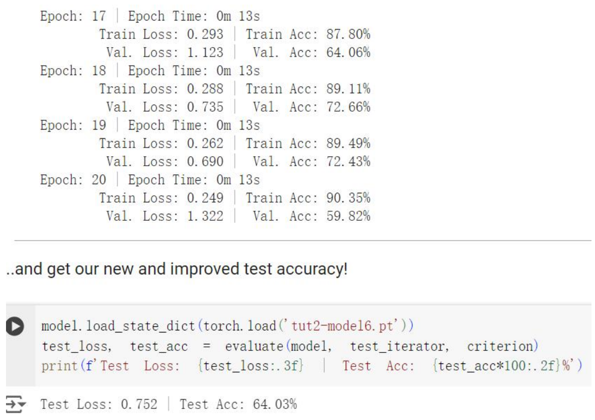
Figure47 Application example with embedding dimension of 80

When the embedding dimension is increased to 140, the trainable parameters become larger and its testing acuuracy achieves 64.03%.



```
[177] INPUT_DIM = len(TEXT.vocab)
      EMBEDDING_DIM = 140
      HIDDEN_DIM = 256
      OUTPUT_DIM = 1
      N_LAYERS = 2
      BIDIRECTIONAL = True
      DROPOUT = 0.5
      PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

      model = RNN(INPUT_DIM,
                  EMBEDDING_DIM,
                  HIDDEN_DIM,
                  OUTPUT_DIM,
                  N_LAYERS,
                  BIDIRECTIONAL,
                  DROPOUT,
                  PAD_IDX)
```

Figure48 Change EMBEDDING_DIM to 140



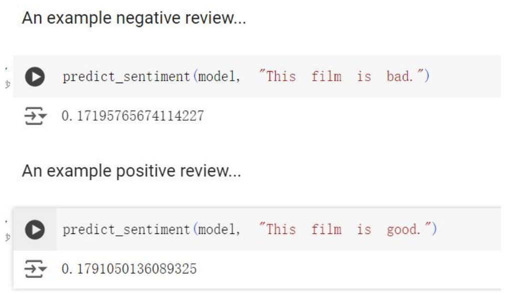
```
Epoch: 17 | Epoch Time: 0m 13s
        Train Loss: 0.293 | Train Acc: 87.80%
         Val. Loss: 1.123 |  Val. Acc: 64.06%
Epoch: 18 | Epoch Time: 0m 13s
        Train Loss: 0.288 | Train Acc: 89.11%
         Val. Loss: 0.735 |  Val. Acc: 72.66%
Epoch: 19 | Epoch Time: 0m 13s
        Train Loss: 0.262 | Train Acc: 89.49%
         Val. Loss: 0.690 |  Val. Acc: 72.43%
Epoch: 20 | Epoch Time: 0m 13s
        Train Loss: 0.249 | Train Acc: 90.35%
         Val. Loss: 1.322 |  Val. Acc: 59.82%
```

..and get our new and improved test accuracy!

```
model.load_state_dict(torch.load('tut2-model6.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.752 | Test Acc: 64.03%
```

Figure49 Training and testing result with embedding dimension of 140



An example negative review...

```
predict_sentiment(model, "This film is bad.")
```

```
0.17195765674114227
```

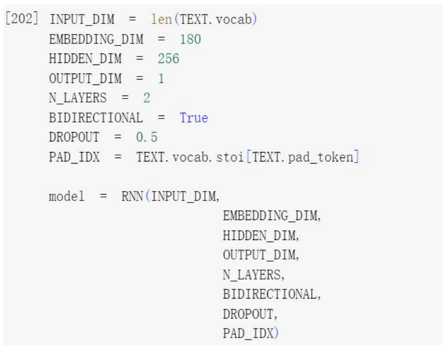
An example positive review...

```
predict_sentiment(model, "This film is good.")
```

```
0.1791050136089325
```

Figure50 Application example with embedding dimension of 140

However, the application result about positive review doesn't seem reasonable. I add the embedding dimension to 180, gaining the model more capacity to compute and represent information.

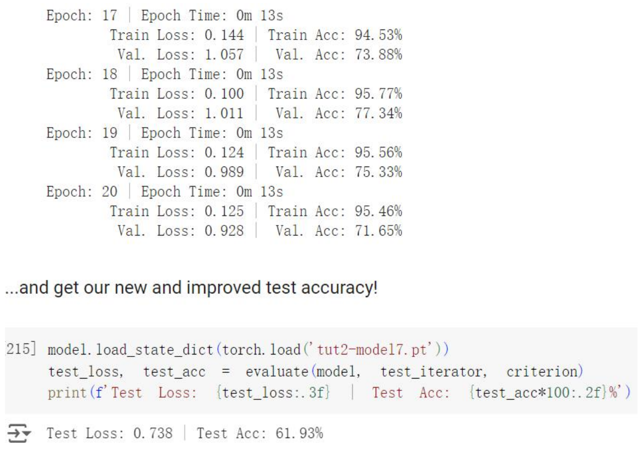


```
[202] INPUT_DIM = len(TEXT.vocab)
      EMBEDDING_DIM = 180
      HIDDEN_DIM = 256
      OUTPUT_DIM = 1
      N_LAYERS = 2
      BIDIRECTIONAL = True
      DROPOUT = 0.5
      PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

      model = RNN(INPUT_DIM,
                  EMBEDDING_DIM,
                  HIDDEN_DIM,
                  OUTPUT_DIM,
                  N_LAYERS,
                  BIDIRECTIONAL,
                  DROPOUT,
                  PAD_IDX)
```

Figure51 Change EMBEDDING_DIM to 180

When the embedding dimension is 180, the testing accuracy decreases to 61.39%, which indicates the embedding dimension may be too much and overfitting occurs.

And the application example seems more reasonable, with a result of 0.17 on a negative review and a result of 0.59 on a positive review. It should be mentioned that this is just an application that allows us to more intuitively see the task of film review sentiment analysis. It has a certain degree of randomness and cannot represent the full capabilities of the model.

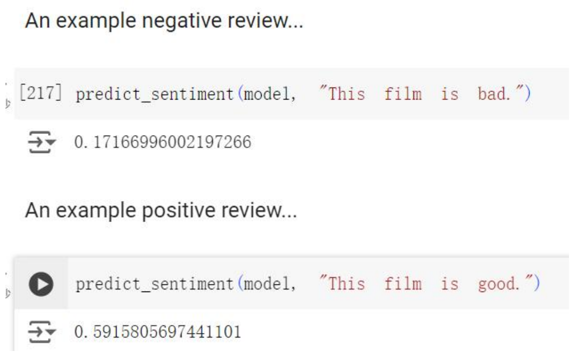


```
Epoch: 17 | Epoch Time: 0m 13s
        Train Loss: 0.144 | Train Acc: 94.53%
         Val. Loss: 1.057 |  Val. Acc: 73.88%
Epoch: 18 | Epoch Time: 0m 13s
        Train Loss: 0.100 | Train Acc: 95.77%
         Val. Loss: 1.011 |  Val. Acc: 77.34%
Epoch: 19 | Epoch Time: 0m 13s
        Train Loss: 0.124 | Train Acc: 95.56%
         Val. Loss: 0.989 |  Val. Acc: 75.33%
Epoch: 20 | Epoch Time: 0m 13s
        Train Loss: 0.125 | Train Acc: 95.46%
         Val. Loss: 0.928 |  Val. Acc: 71.65%
```

...and get our new and improved test accuracy!

```
215] model.load_state_dict(torch.load('tut2-model7.pt'))
     test_loss, test_acc = evaluate(model, test_iterator, criterion)
     print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')

Test Loss: 0.738 | Test Acc: 61.93%
```

Figure52 Training and testing result with embedding dimension of 180



An example negative review...

```
[217] predict_sentiment(model, "This film is bad.")

0.17166996002197266
```

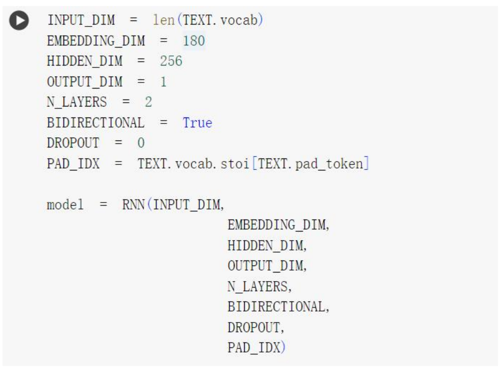
An example positive review...

```
predict_sentiment(model, "This film is good.")

0.5915805697441101
```

Figure53 Application example with embedding dimension of 180

3.4 Remove dropout

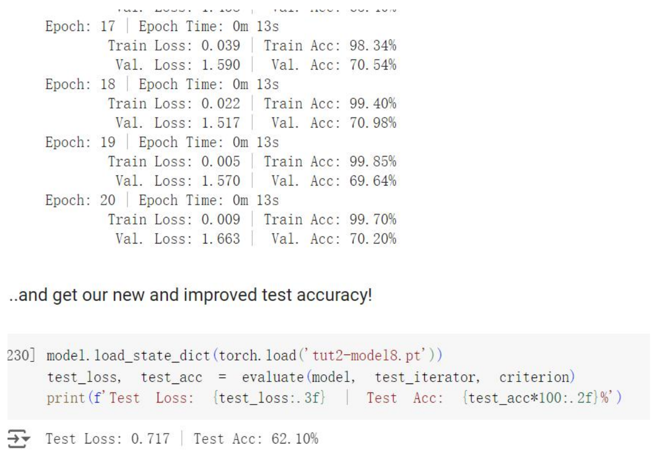
Dropout processing can prevent overfitting and improve the robustness of the model by randomly discarding some neurons. Removing dropout can be realized by changing the setting DROPOUT from 0.5 to zero.



```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 180
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = RNN(INPUT_DIM,
            EMBEDDING_DIM,
            HIDDEN_DIM,
            OUTPUT_DIM,
            N_LAYERS,
            BIDIRECTIONAL,
            DROPOUT,
            PAD_IDX)
```

Figure54 Change DROPOUT to 0

I train the LSTM model after removing dropout for 20 epochs, and it achieves testing accuracy of 62.10%.



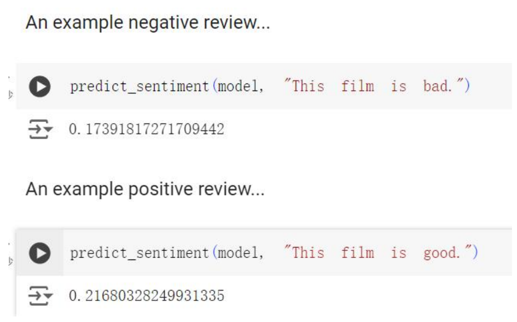
```
Epoch: 17 | Epoch Time: 0m 13s
        Train Loss: 0.039 | Train Acc: 98.34%
         Val. Loss: 1.590 |  Val. Acc: 70.54%
Epoch: 18 | Epoch Time: 0m 13s
        Train Loss: 0.022 | Train Acc: 99.40%
         Val. Loss: 1.517 |  Val. Acc: 70.98%
Epoch: 19 | Epoch Time: 0m 13s
        Train Loss: 0.005 | Train Acc: 99.85%
         Val. Loss: 1.570 |  Val. Acc: 69.64%
Epoch: 20 | Epoch Time: 0m 13s
        Train Loss: 0.009 | Train Acc: 99.70%
         Val. Loss: 1.663 |  Val. Acc: 70.20%
```

..and get our new and improved test accuracy!

```
230] model.load_state_dict(torch.load('tut2-model8.pt'))
     test_loss, test_acc = evaluate(model, test_iterator, criterion)
     print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.717 | Test Acc: 62.10%
```

Figure55 Training and testing result after removing dropout



An example negative review...

```
predict_sentiment(model, "This film is bad.")
```

```
0.17391817271709442
```

An example positive review...

```
predict_sentiment(model, "This film is good.")
```

```
0.21680328249931335
```

Figure56 Application example after removing dropout

4. Conclusion

In this lab, we focus on recurrent neural networks to experiment on the task of film review sentiment analysis. I haven't been exposed to labs on recurrent networks as well as text processing before this course, so this is interesting to me.

We conduct experiments on basic RNNS and LSTMs. The performance of basic RNN is just OK, with an accuracy of about half, and LSTMs perform better than basic RNNs. LSTMs has more complex structure as well as its key ingredient cell state. LSTMs can seperate the long-term memory and short-term memory, and use forget cells and memory cells to throw away things and add new things, making better use of the give information. Besides, LSTMs have larger trainable parameters, which may result in larger need of computational resource.

Through changing layers and settings to compare performance under different conditions, I have had a deeper understanding of the function of recurent models, and the architecture of RNNs, LSTMs, bidirectional-LSTMs, and multi-layer LSTMs.