NLP Generation using LSTMs

Student ID: 230161210

Student name: Qing Zhao

1. Long Short Term Memory networks Review

Long Short Term Memory networks (LSTMs) are a special kind of RNNs, capable of learning long-term dependencies. They are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer. The repeating module in LSTMs has a different structure:the cell state and three gates (forget gate, input gate, and output gate).The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged. Gates are a way to optionally let information through. Forget gate decides what information to discard from the cell state of the previous time step. Input Gate: Determines what new information to store in the cell state. Output Gate (o_t): Controls what information from the cell state (c_t) is used to influence the output of the LSTM unit. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.A value close to one indicates that the information should be retained, while a value close to zero indicates it can be forgotten.

2. Problems Formulation

In this experiment, I will use the open source Shakespeare Sonnets Dataset as the corpus to train a LSTM model to generate texts. The Shakespeare Sonnets Dataset contains 2159 lines of text extracted from Shakespeare's sonnets. Figure 1 demonstrates examples of sonnets.

```
corpus=data.lower().split('\n')
corpus[:10]
```

```
['from fairest creatures we desire increase,',
 "that thereby beauty's rose might never die,",
 'but as the riper should by time decease,',
 'his tender heir might bear his memory:',
 'but thou, contracted to thine own bright eyes,',
 "feed'st thy light'st flame with self-substantial fuel,",
 'making a famine where abundance lies,',
 'thyself thy foe, to thy sweet self too cruel.',
 "thou that art now the world's fresh ornament",
 'and only herald to the gaudy spring,']
```

Figure 1: examples of The Shakespeare Sonnets

3. Data Preprocessing

3.1 Build the word vocabulary

First, to convert the text into a digital representation, I used tokenizer to tokenize the corpus. It is a small corpus so that we can expect the generation ability of the model we are going to build is limited due to the available words. But to explore the LSTMs and its application in text generation, it is enough for our experiment. The following is the first word-index mapping and the total number of words is 3211.

word index dictionary: {'and': 1, 'the': 2, 'to': 3, 'of': 4, 'my': 5,}

total words: 3211

Figure 2: examples of word index dictionary.

3.2 prepare training dataset

After tokenizing the corpus, I converted the text into sequences. So we can see that the sentence in sonnet1 displayed above was converted to the digital sequences [34, 417, 877, 166, 213, 517]. Since the feature of the LSTM model is to use the preview sequences to predict the next word, I need to keep on dealing with the sequence into n_gram sequences. Then padding these sequences so that all sequences would have the same length which is the request of the input layer in any neural network. Figure 3 and figure 4 demonstrate results after these two steps.

```
[[34, 417],
 [34, 417, 877],
 [34, 417, 877, 166],
 [34, 417, 877, 166, 213],
 [34, 417, 877, 166, 213, 517]]
```

Figure 3 : n_gram sequences

```
[[  0,   0,   0,   0,  34, 417],
 [  0,   0,   0,  34, 417, 877],
 [  0,   0,  34, 417, 877, 166],
 [  0,  34, 417, 877, 166, 213],
 [ 34, 417, 877, 166, 213, 517]]
```

Figure 4: padding to maintain the same length

3.3 split training data and targets

For each sequence after padding, the last token is the target we hope the model would generate given the preview token sequence. Now I splitted the dataset into training data and target data in order to train the LSTM model. Until now, the data preparation step for a text generation task is completed, gathering 15462 training samples and corresponding targets.

4. Building a LSTM model for NLP Generation

About the LSTM model, I set the 100-dimensions vector to embed, in order to cluster words with similar semantics. Embedding is a common method for NLP tasks and usually serves as the first layer in the neural network. To better learn the relationship of words, I employed binary direction LSTM with 150 units each, which definitely increased the training time but it was worth trying.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_5 (Embedding)     (None, 15, 100)           269000

 bidirectional_7 (Bidirecti  (None, 300)               301200
 onal)

 dense_3 (Dense)             (None, 2690)              809690

=================================================================
Total params: 1379890 (5.26 MB)
Trainable params: 1379890 (5.26 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5:the architecture of the LSTM model

Adopting 100 epochs, the loss and accuracy tended to be stable, with the figure of 0.53 and 0.85 respectively.

The configuration of the model displayed as follows.

```
cell_units=150
embedding_dim=100
lr=0.01
epochs=100
batch_size=128
input_length=max_sequence_len-1

model=create_model(total_words,embedding_dim,input_length,cell_units)
#model.summary()

model.compile(loss='categorical_crossentropy',optimizer=Adam(learning_rate=lr),metrics=['accuracy'])

model.fit(X_train,y_train,epochs=epochs,batch_size=batch_size)
```

Figure 6: The configuration of the model

5. Texts Generation

Finally, I tried to employ this model to generate the following 30 words given a seed text as “he told me”. The model produced “i was so love and the heart in a heart and as the purse declining i was to i bow bow was bow bow as be bow bow bow.”

Analyzing this generation of texts, I discovered that the last part of the texts occurred with substantial repetition. This mainly caused by the limited training corpus which only contained 3211 words, additionally the longest sentences in the Shakespeare Sonnets Dataset contains 11 words, so the model was trained with the input size quite smaller than it was used in the generation task, which lead to its poor performance in the real application.

```
def generate_text(seed_text, next_words, model, max_sequence_len):

  for _ in range(next_words):
      token_list = tokenizer.texts_to_sequences([seed_text])[0]
      padded_token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding='pre')

  # predict the next word
      probabilities = model.predict(padded_token_list, verbose=0)
      predicted = np.argmax(probabilities, axis=-1)[0]

      if predicted != 0:
        output_word = tokenizer.index_word[predicted]
        seed_text += " " + output_word
      else:
        break

  return seed_text
```

```
print(generate_text("He told me", 30, model, max_sequence_len))
```

```
He told me i was be so love and the heart in a heart and as the purse declining i was to i bow bow was bow bow was be bow bow bow
```

Figure 7: generation texts example 1

Using “the world” as seed text to generate the following 8 texts, we got the result “ the world will be thy widow and still weep to ”, which had less repetition.

```
print(generate_text("the world", 8, model, max_sequence_len)
```

```
the world will be thy widow and still weep to
```

Figure 8: generation texts example2

References

1. course video: MIT S191: Recurrent Neural Networks
2. course video: Natural Language Processing in TensorFlow
3. textbook: Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron Courville
4. colah’ log: https://colah.github.io/posts/2015-08-Understanding-LSTMs/