Independent Project

2018 Spring – 6910A

A talkative agent

Supervisor: Prof. David ROSSITER

CAO chen

Contents

Abstract

In this project, I used Tensorflow framework to build a recurrent neural network model through python3 to train the chatbot. After a series of tuning and several trainings with GPU, I could achieve the basic communication between the program and the user.

1. Introduction

Nowadays, computing power is greatly enhanced. In particular, the Nvidia graphics card can support the deep learning related framework: Tensorflow. Individuals can also partially implement AI training. Therefore, I try to build a dialogue system to complete the interaction between the program and the user. I will use python3 to complete the text processing, model building, word vector embedding, Tensorflow training and prediction. Because of the large amount of machine calculations and the data loaded, it can't be completed without processing, so I will also use some techniques to approximate the processing.

2. Design

In terms of this project, I intend to use the Recurrent Neural Network to train the chatbot as well as do the predictions to interact with users. The reason why I choose RNN is that RNN could involve more mapping in the hidden layer, represented by the recurrent according to the time step, and reflect the relationships between the current word with previous states. However, the problem of gradient explosion as well as low efficiency would always occur because calculating the current gradient need to include all the previous gradient by chain rules. Thus, I choose the Long Short-Term Memory(LSTM) as basic cells for my designed Neural Networks.

2.1LSTM

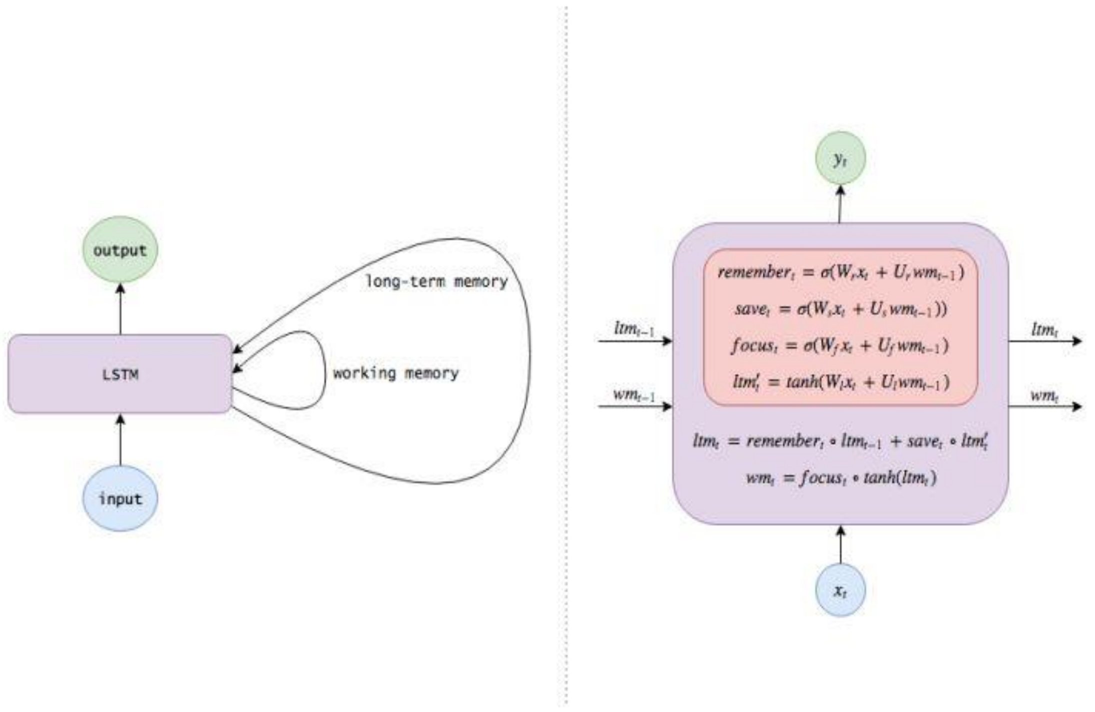

output
LSTM
input
long-term memory
working memory
y_t
$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$
$save_t = \sigma(W_s x_t + U_s wm_{t-1}))$
$focus_t = \sigma(W_f x_t + U_f wm_{t-1})$
$ltm'_t = tanh(W_l x_t + U_l wm_{t-1})$
ltm_{t-1}
wm_{t-1}
ltm_t
wm_t
$ltm_t = remember_t \circ ltm_{t-1} + save_t \circ ltm'_t$
$wm_t = focus_t \circ tanh(ltm_t)$
x_t

Figure 1. Demonstration of LSTM

There are four components in LSTM, namely, *Input gate, Forget gate, Output gate and New Memory cell*.

The structure of LSTM is just like Figure 1, it can learn long-term dependencies, which means that this structure can memory the useful information happened previously in a short or long time ago and forget useless information for the current state. For the dialogue system, there are about 400 thousand of conversations prepared for training in the corpus. Therefore, it is not appropriate to throw so many sentences in the fully connected RNN and compute the probability of current word for the given words.

2.2 Seq2seq

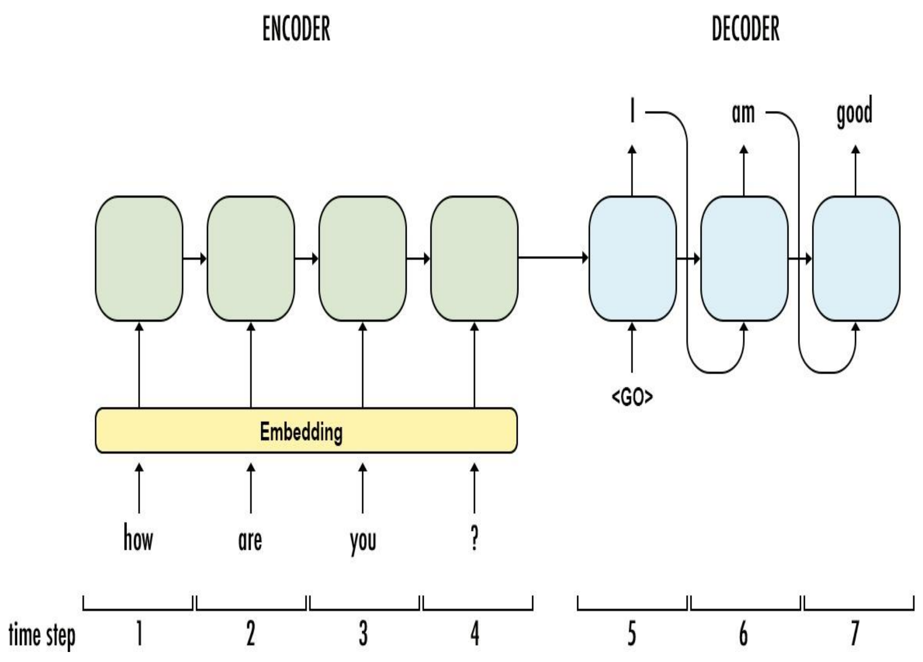


Figure 2. Structure of Seq2seq

The main feature of Seq2seq model is including the encoder and decoder, which means that the questions or input sentences are mapped into an intermediate matrix representing the main features of the data and then decoded to the corresponding answer. Seq2seq is appropriate for this chatting task because it can handle the flexible length of sequences compared with traditional RNN structures. In general, every input word and previous states could obtain a probability of different word in the dictionary and return the max one in a timestamp of RNN. Thus, if the size of output sequence is same as the input sequence, which is obvious not good for dialogue system.

To achieve the Seq2seq model, at least two basic LSTM RNN cell are needed. One of the cell is used as encoder and the other is used as decoder.

2.3Loss Function

The loss function of the network is the Softmax function for probability of P(y|x). X is the given words and Y is the predicted word..

2.4Optimize

Adam optimization method involved with quadratic gradient correction is used because it has the ability of avoiding the local and convergence fast.

2.5Prediction

As shown in Figure 3, during the test, the decoder outputs the current word based on the weights of trained model and the word from the previous state.

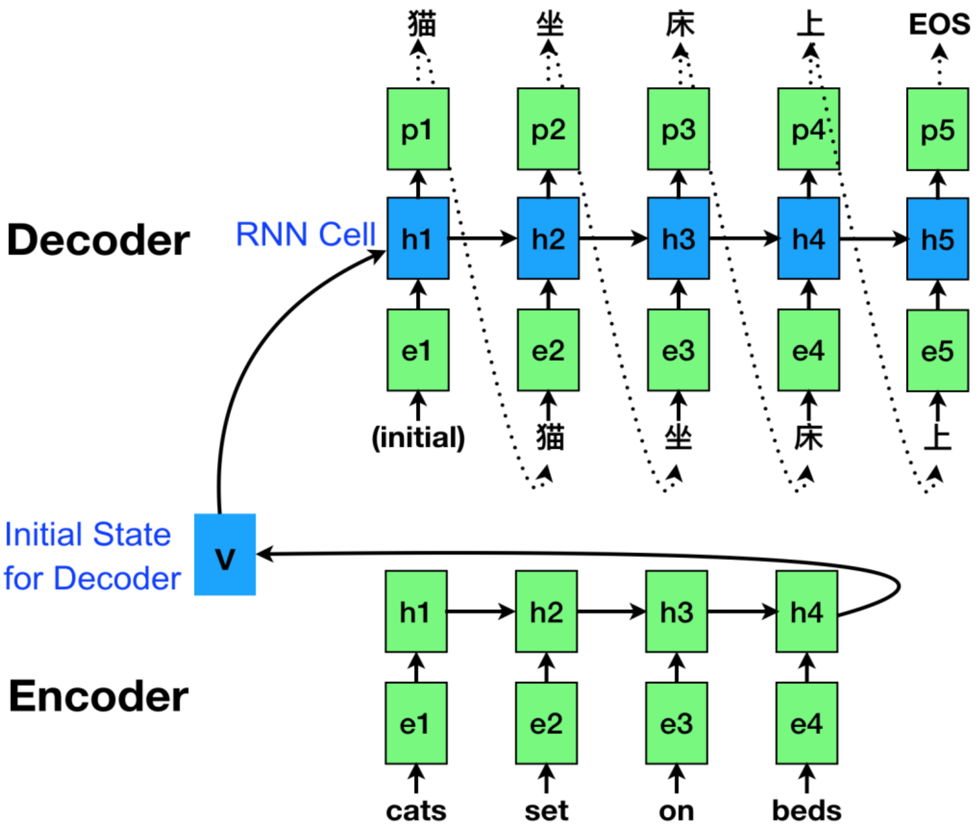


Figure 3. How decoder predicts

3. Implementation

3.1Text Processing

- **Corpus**

Use the cornel corpus including plenty of movie scripts involved with kinds of dialogues.

- **Build the dictionaries**

To create the corresponding input and output sequence that the model could handle, two dictionaries are needed. One is called word2id, the other is called id2word.

- **Handle some low frequency words**

Some uncommon words would affect the quality of interactive because it is wired to say some unfamiliar words in daily oral scenarios. And the “nltk” package of Python could help us with its encapsulated functions:

- **Batching processing**

It is not possible to throw all the train data into the Neural Network once time for our general machine, and what we need to do is getting samples from the processed conversations and converting these samples to sequence batches with the help of word2id dictionary just built. The batches involve input sequence, output sequence, target sequence and weights. The size of every batch is set according to the length od the encoder and decoder.

- **Create the Feed_Dict**

After creating those batches, I could use these batches to feed the part of placeholders called “encoderInputs”, “decoderOutputs” I have just set in the session prepared for the training or predictions.

3.2Tesorflow

➢ Training

- **Construct session**

After building the inference model and setting the loss function as well as the optimizer, I could start to train models. First, the object of session should be created and open.

- **Enable device**

I choose to train the model with GPU of NVIDIA because it has more calculation units.

The default GPU is enabled after executing this following code:

with tf.device('/gpu:0')

- **Manage the model and parameters**

The session object could be saved after completing every epoch with encapsulated function as:

saver.save(sess, model_name)

And the session could be reloaded to continue to train or be used for test with encapsulated function:

saver.restore(sess, modelName)

- **Sampled_softmax**

The probability of next word given existing words should be calculated with Softmax function as $P(y=j|\mathbf{x}) = \frac{e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_k}}$. The numerator represents the current word and the denominator represents the sum of all the vocabularies in the dictionary of Word2id just built. However, it is too much slow to include all the vocabularies to calculate the probability of one word. So, approximation method is needed.

One encapsulated function of approximation is *tf.nn.sampled_softmax_loss()*, which could return a vector of loss prepared for minimized. This function would approximate the probability by negative sampling according to the positive one.

- **Projection**

To achieve the *sampled_softmax* approximation, the original dimension of vocabulary size should also be projected into the size of approximated that could fit the dimension for weights as well as bias.

- **Dropout**

Dropout mechanism should also be introduced to prevent the underfitting. It could drop some units of the RNN structures according to the specific probability.

- **Embeding**

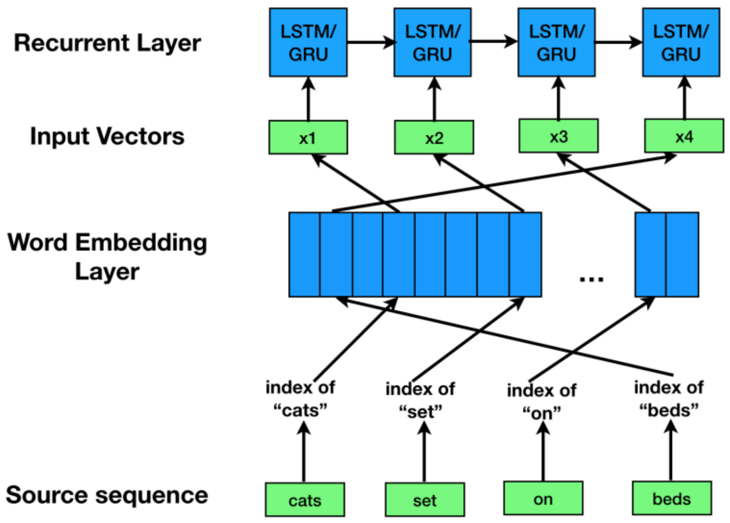
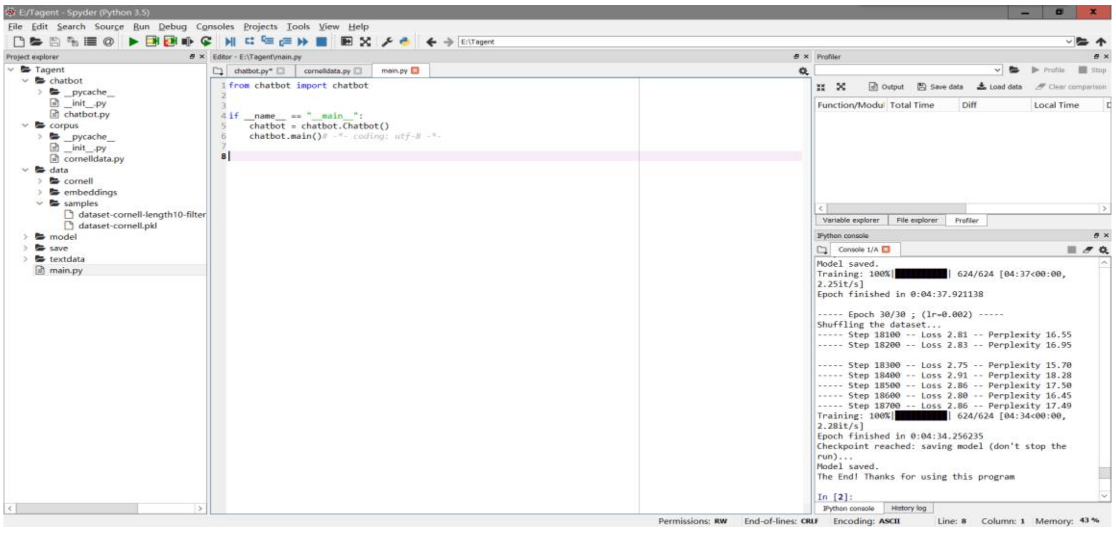


Figure 4. How word vector works in LSTM

One necessary component of RNN training for NLP is word vectors, equivalently the embedding matrix. Although the dictionary word2id been built for every word has been built, it is not appropriate to represent words only by their IDs because they are independent with each other. And the embedding matrix could map words into another space to show their relationships.

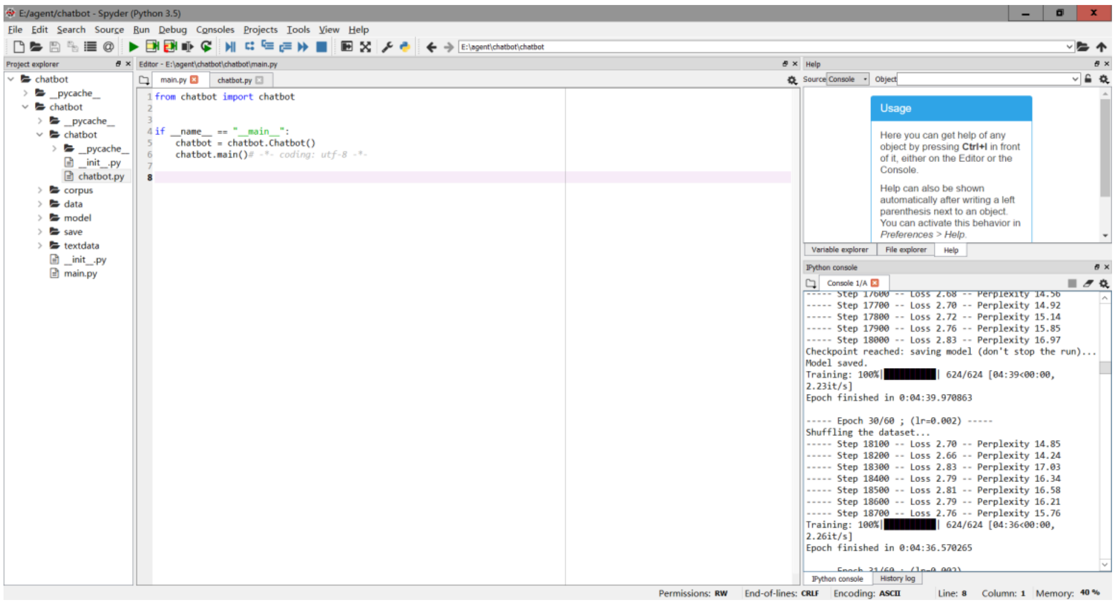
- **Training process**



```python
from chatbot import chatbot


if __name__ == "__main__":
    chatbot = chatbot.Chatbot()
    chatbot.main()# -*- coding: utf-8 -*-

```

```
Model saved.
Training: 100%|██████████| 624/624 [04:37<00:00,
2.25it/s]
Epoch finished in 0:04:37.921138

----- Epoch 30/30 ; (lr=0.002) -----
Shuffling the dataset...
----- Step 18100 -- Loss 2.81 -- Perplexity 16.55
----- Step 18200 -- Loss 2.83 -- Perplexity 16.95

----- Step 18300 -- Loss 2.75 -- Perplexity 15.70
----- Step 18400 -- Loss 2.91 -- Perplexity 18.28
----- Step 18500 -- Loss 2.86 -- Perplexity 17.50
----- Step 18600 -- Loss 2.80 -- Perplexity 16.45
----- Step 18700 -- Loss 2.86 -- Perplexity 17.49
Training: 100%|██████████| 624/624 [04:34<00:00,
2.28it/s]
Epoch finished in 0:04:34.256235
Checkpoint reached: saving model (don't stop the
run)...
Model saved.
The End! Thanks for using this program

In [2]:
```

Figure 5. Initial model

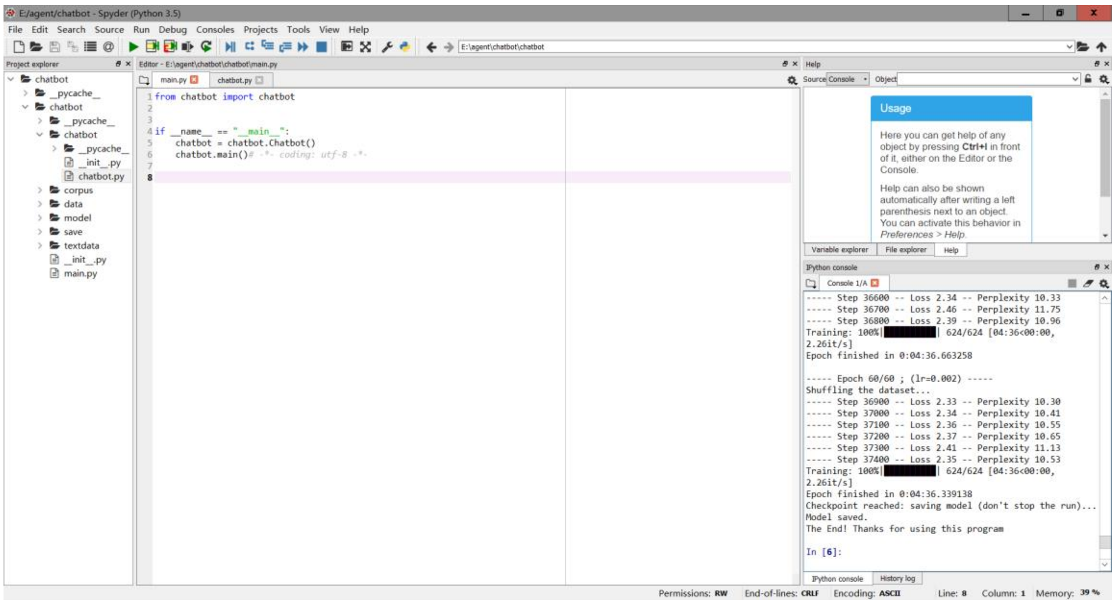

E:/agent/chatbot - Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Project explorer
Editor - E:\agent\chatbot\chatbot\main.py
Help
Usage
Variable explorer
File explorer
IPython console
Console 1/A
History log

```python
from chatbot import chatbot


if __name__ == "__main__":
    chatbot = chatbot.Chatbot()
    chatbot.main()# -*- coding: utf-8 -*-
```

```
----- Step 17600 -- Loss 2.68 -- Perplexity 14.56
----- Step 17700 -- Loss 2.70 -- Perplexity 14.92
----- Step 17800 -- Loss 2.72 -- Perplexity 15.14
----- Step 17900 -- Loss 2.76 -- Perplexity 15.85
----- Step 18000 -- Loss 2.83 -- Perplexity 16.97
Checkpoint reached: saving model (don't stop the run)...
Model saved.
Training: 100%|██████████| 624/624 [04:39<00:00,
2.23it/s]
Epoch finished in 0:04:39.970863

----- Epoch 30/60 ; (lr=0.002) -----
Shuffling the dataset...
----- Step 18100 -- Loss 2.70 -- Perplexity 14.85
----- Step 18200 -- Loss 2.66 -- Perplexity 14.24
----- Step 18300 -- Loss 2.83 -- Perplexity 17.03
----- Step 18400 -- Loss 2.79 -- Perplexity 16.34
----- Step 18500 -- Loss 2.81 -- Perplexity 16.58
----- Step 18600 -- Loss 2.79 -- Perplexity 16.21
----- Step 18700 -- Loss 2.76 -- Perplexity 15.76
Training: 100%|██████████| 624/624 [04:36<00:00,
2.26it/s]
Epoch finished in 0:04:36.570265

----- Epoch 31/60 ; (lr=0.002)
```

```
Permissions: RW   End-of-lines: CRLF   Encoding: ASCII   Line: 8   Column: 1   Memory: 40 %
```

Figure 6. changed Model 1


E:/agent/chatbot - Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Project explorer
Editor - E:\agent\chatbot\chatbot\main.py
Help
Usage
Variable explorer
File explorer
IPython console
Console 1/A
History log

```python
from chatbot import chatbot


if __name__ == "__main__":
    chatbot = chatbot.Chatbot()
    chatbot.main()# -*- coding: utf-8 -*-
```

```
----- Step 36600 -- Loss 2.34 -- Perplexity 10.33
----- Step 36700 -- Loss 2.46 -- Perplexity 11.75
----- Step 36800 -- Loss 2.39 -- Perplexity 10.96
Training: 100%|██████████| 624/624 [04:36<00:00,
2.26it/s]
Epoch finished in 0:04:36.663258

----- Epoch 60/60 ; (lr=0.002) -----
Shuffling the dataset...
----- Step 36900 -- Loss 2.33 -- Perplexity 10.30
----- Step 37000 -- Loss 2.34 -- Perplexity 10.41
----- Step 37100 -- Loss 2.36 -- Perplexity 10.55
----- Step 37200 -- Loss 2.37 -- Perplexity 10.65
----- Step 37300 -- Loss 2.41 -- Perplexity 11.13
----- Step 37400 -- Loss 2.35 -- Perplexity 10.53
Training: 100%|██████████| 624/624 [04:36<00:00,
2.26it/s]
Epoch finished in 0:04:36.339138
Checkpoint reached: saving model (don't stop the run)...
Model saved.
The End! Thanks for using this program

In [6]:
```

```
Permissions: RW   End-of-lines: CRLF   Encoding: ASCII   Line: 8   Column: 1   Memory: 39 %
```

Figure 7. changed model 2

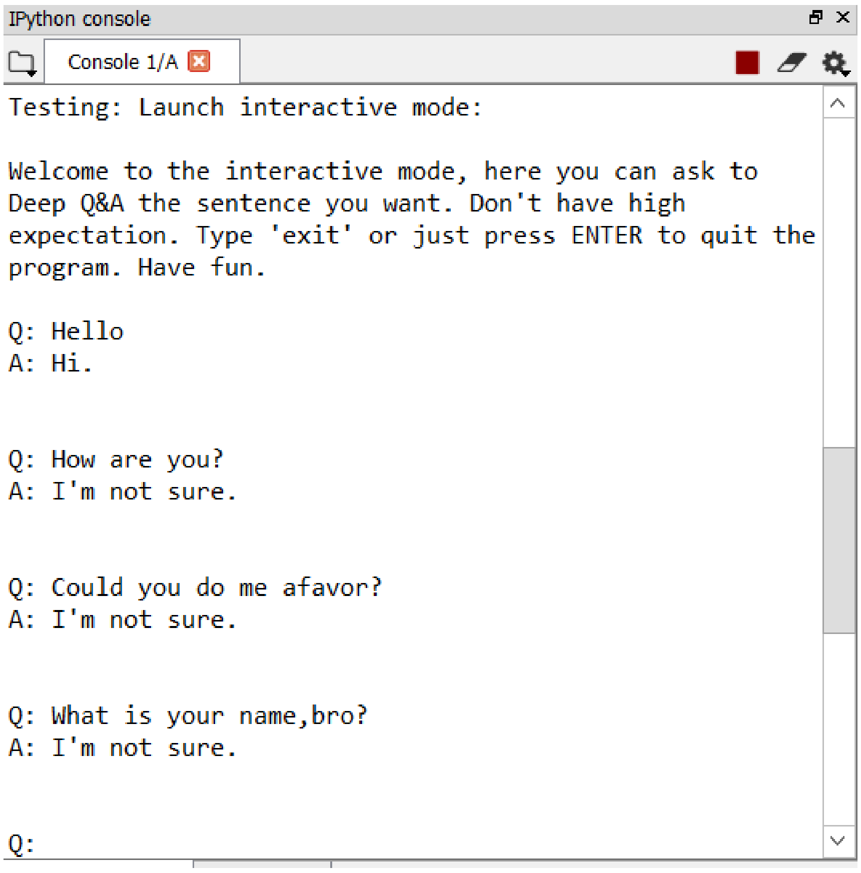
One problem when training is the loss value too big equal to 2.8. Can be seen from the training process, the loss value is initially more than 5 and decreased fast during the former epochs. However, the value fluctuates at 3 and finally ends up at 2.86.

The initial model in figure 5 is set by the empirical knowledge that looks not good, and so I try to adjust the model from two aspects. The one is increasing the rate of dropping the original NN units by adjusting the parameter of DropoutWrapper, the value of keep, from 0.9 to 0.8, as shown in figure 6. And it comes to underfitting and bad performance if adjusting the value to 0.7 or less .

On the other hand, I also increase the number of epoch one more time (from 30 to 60) just like the figure 7. As seen in the figure, the final loss value has decreased significantly to 2.32 even though it has fluctuated at 2.8 for a long time.

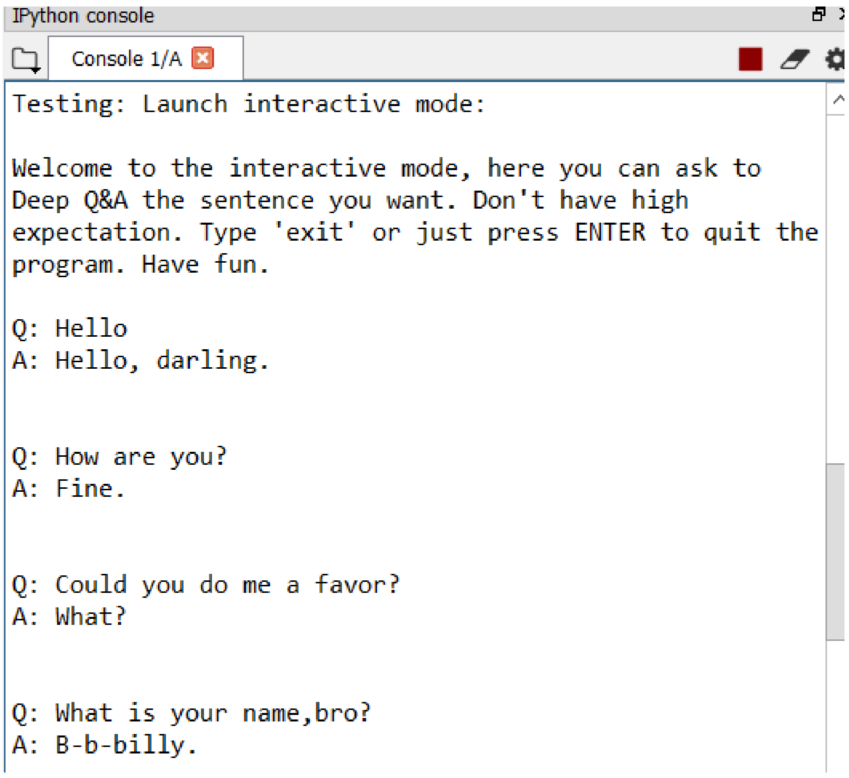
➢ Interactive

- **Q&A**



```
Testing: Launch interactive mode:

Welcome to the interactive mode, here you can ask to
Deep Q&A the sentence you want. Don't have high
expectation. Type 'exit' or just press ENTER to quit the
program. Have fun.

Q: Hello
A: Hi.


Q: How are you?
A: I'm not sure.


Q: Could you do me afavor?
A: I'm not sure.


Q: What is your name,bro?
A: I'm not sure.


Q:
```

Figure 8. Result before optimization



```
Testing: Launch interactive mode:

Welcome to the interactive mode, here you can ask to
Deep Q&A the sentence you want. Don't have high
expectation. Type 'exit' or just press ENTER to quit the
program. Have fun.

Q: Hello
A: Hello, darling.


Q: How are you?
A: Fine.


Q: Could you do me a favor?
A: What?


Q: What is your name,bro?
A: B-b-billy.
```

Figure 9. Result after optimization

As can been seen from the above two pictures, the prediction results in figure 8 without model optimization are bad and give some no sense answers. While the agent could give better answers for the same questions after optimization, which looks much better.

- **Analysis**

Due to the limited dimensions word vectors, lack of a large amount of artificially annotated text data, especially the computational power of the personal computer, this agent cannot perform very accurate Q&A. My current consideration is that the main reason of this problem is caused by The RNN model based on the probabilistic of current word given previous word. It means that even asking the original sentence of the train set to the agent may not get the correct answer, because partial words of the question may appear in the former text separately and corresponding answers are different, which affects the probability a lot.

4. Further work

Although I could not possess the powerful resource to make a marvelous smart agent, but I think a function may be helpful to make the agent more intelligent: if the answer is found to be inaccurate, the same question and the answer can be input at certain times, then loaded into the training set, and the new training set is input into the model in the next training, just like teaching children to speak. This function would significantly increase the probability of a correct match for a specific question.

5. Conclusion

This project achieves the purpose of function to interact with users by a series of work from initial text processing to the final model adjustment. And it also proves that individuals could achieve this kind of training complex models with existing electronical devices. Although during the training process, it would come into some awkward situations that the calculation is too large for normal process units, we could still find approximation ways to achieve the similar outcomes.