华南理工大学

South China University of Technology

# The Experiment Report of *Machine Learning*

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

**SUBJECT:** SOFTWARE ENGINEERING


*Author:*
Xiujun Zhou
Jieqiong Liu

*Supervisor:*
Mingkui Tan

*Student ID:*
201830664237
201830661465

*Grade:*
Undergraduate


November 28, 2020

# Chinese-English Translation Machine Based on Sequence to Sequence Network

***Abstract*—In this experiment we used Sequence-to-Sequence network[1] and attention mechanism[2] to realize a simple Chinese-English translation machine. We trained the network and used the model to understand the natural language processing.**

## I. Introduction

SEQUENCE- to-Sequence network is a circulating neural network, including encoder and decoder. It is an important model in natural language processing and can be used in machine translation, conversational systems and automatic abstracts. To understand the network and learn from it, we started this experiment. Here are some descriptions of the experiment.

Motivation of this Experiment includes:

- To understand natural language processing
- To have a better understanding of the classic Sequence-to-Sequence machine translation model
- Master the application of attention mechanism in machine translation model.
- Cultivate engineering capabilities when Building machine translation models and verifying model performance on simple and small-scale datasets.
- Understand the application of Transformer in machine translation tasks.

## II. METHODS AND THEORY

### A. *Seq-to-Seq Model*

A Recurrent Neural Network (RNN) is a network that operates on a sequence and uses its own output as next input for subsequent steps. A Sequence to Sequence network (seq2seq network) or Encoder-Decoder network, is a model consisting of two RNNs called the encoder and decoder.

The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence. Unlike sequence prediction with a single RNN, where every input corresponds to an output, the seq2seq model frees us from sequence length and order, which makes it ideal for translation task.

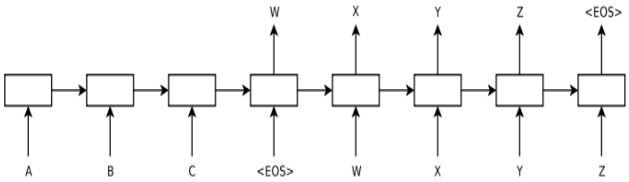


Fig. 1: Input and output of a Seq2seq model

Assume that we have a sequence of inputs $(x_1, ..., x_T)$, a RNN outputs a sequence $(y_1, ..., y_T)$ by iterating the following equation:

$$h_t = sigm(W^{hx}x_t + W^{hh}_{t-1})$$

$$y_t = W^{yh}h_t$$

In the formula, $W$ represents the weight matrix for corresponding parameters. The basic model of RNN is shown in the figure. The input of each neuron includes: the hidden layer state $h$ of the previous neuron (for memory) and the current input $X$ (for current information). Once the neuron has received the input, it calculates the new hidden state $h$ and the output $y$, and then passes them on to the next neuron. Because of the existence of hidden state $h$, RNN has the ability to 'memorize' things in the past.

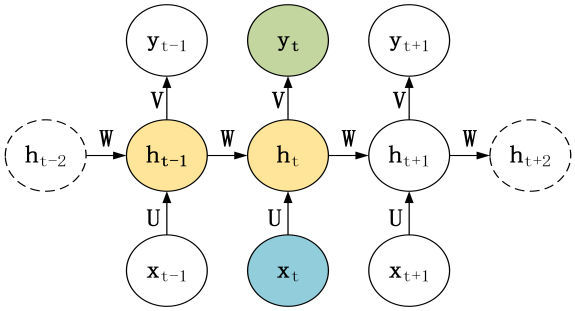


Fig. 2: Structure of RNN

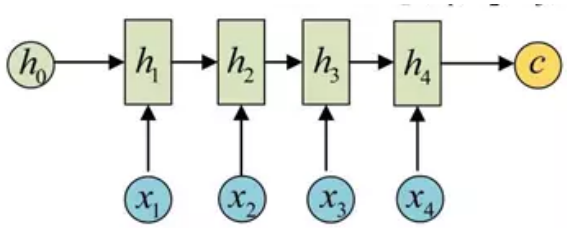


Fig. 3: Structure of RNN

However, a simple RNN cannot deal with problems whose input and output sequences have different lengths with complicated and non-monotonic relationships, Consider the sentence "Je ne suis pas le chat noir" to "I am not the black cat". Most of the words in the input sentence have a direct translation in the output sentence, but are in slightly different orders, e.g. "chat noir" and "black cat". In the "ne/pas" construction there is also one more word in the input sentence. It would be difficult to produce a correct translation directly from the sequence of input words.

In order to solve this problem, the encoder in a Sqe2Seq Model creates a single vector which, in the ideal case, encodes the "meaning" of the input sequence into a single vector. This vector represents a single point in some $N$ dimensional space of sentences.

The architecture can be divided into 2 main parts: the encoder and the decoder. The training objective is to maximize the log probability of a correct translation $T$ of the given source sentence $S$

$$\frac{1}{|S|}\Sigma_{(T,S)\in S} logp(T|S)$$

Once we completed the training process, the network produce translation by finding the most likely translation according to RNN.

$$\hat{T} = \arg\min_{T} p(T|S)$$

*1) **Encoder**:*

The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word. The encoder compresses an input sequence into a vector of specified length, which can be regarded as the semantics of the sequence. This process is called encoding.

*2) **Decoder**:*

The decoder is responsible for generating specified sequences from semantic vectors, a process also known as decoding. In the simplest seq2seq decoder it uses only last output of the encoder. This last output is sometimes called the context vector as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder. At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string SOS token, and the first hidden state is the context vector (the encoder's last hidden state).

Seq2seq model feeds the encoder with a reversed data sequence. Additionally, according to some papers[4], the encoder can also be implemented in a bidirectional way. There are 2 encoders in the network proposed, a forward RNN and a backward RNN. The forward RNN $\vec{f}$ reads the sequence in its original way $(x_1, x_2, ..., x_{T_x})$, calculates a sequence of forward hidden states $(\vec{h_1}, \vec{h_2}, ..., \vec{h_{T_x}})$. In a similar way, the backward RNN $\overleftarrow{f}$ reads a reverse sequence $(x_{T_x}, ..., x_2, x_1)$, resulting in a sequence of backward hidden states $(\vec{h_{T_x}}, ..., \vec{h_2}T_x, \vec{h_1})$.

### B. Attention Mechanism

In this experiment, we embedded the decoder with an attention mechanism. Attention mechanism allows the decoder network to 'focus' on a certain part of the encoder's outputs for every step of the decoder's own outputs.

At first, we calculate a set of attention weights. These will be multiplied by the encoder output vectors to create a weighted combination. The result (called attn_applied in the code) should contain information about the specific part of the input sequence, and thus helping the decoder choose the right output words.

Calculating the attention weights is done with another feed-forward layer, using the decoder's input and hidden state as inputs. Here we assume the hidden states is $(h_1, h_2, ..., h_T)$. Suppose that the hidden state of current state is $s_{t-1}$, we can calculate the correlation between each input position $j$ and current output position $i$ by following formula:

$$e_{ij} = a(s_{t-1}, h_j)$$

namely

$$\vec{e_t} = (a(s_{t-1}, h_j), ..., a(s_{t-1}, h_r))$$

And $a$ represents a kind of operator, for example, dot-multiply $\vec{e_t} = s_{t-1}\vec{h}$, weighted dot-multiply $\vec{e_t} = s_{t-1}\vec{W}\vec{h}$ and sum $\vec{e_t} = \vec{v}tanh(W_1\vec{h} + W_2\vec{s_{t-1}})$. We apply Softmax operation on $\vec{e_t}$ to normalize it and then get the distribution of attention:

$$\vec{\alpha_t} = softmax(\vec{e_t})$$

And

$$\alpha_{ij} = \frac{exp(e_{ij})}{\Sigma_{k=1}^{T} exp(e_{tk})}$$

This is a critical step in attention mechanism. With this attention weight attached, the position that is more important to current output position will be offered larger weight in the prediciton.

After that, we cam get the output context vector and next hidden state:

$$\vec{c_t} = \Sigma_{j=1}^{T} \alpha_{ij} h_j$$

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

The output on this position is:

$$p(y_t|y_1, ..., y_{t-1}, \vec{x}) = g(y_{i-1}, s_i, c_i)$$

Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to.

Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few. Following figure shows the principles of the attention decoder model.

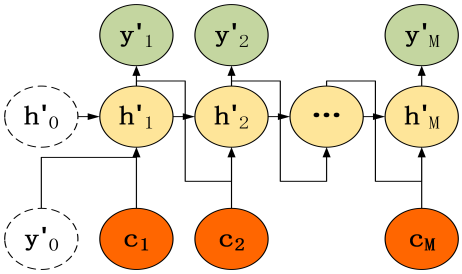


Fig. 4: Attention mechanism

We are freed from the limitation that encoder can only output one single vector. With attention mechanism, the model has the ability to focus on the important part of next input word and improve the performance of the model as a result. Another advantage is that with attention mechanism, we can easily get the change of attention weight matrix, thus have a better understanding of Seq2seq model.

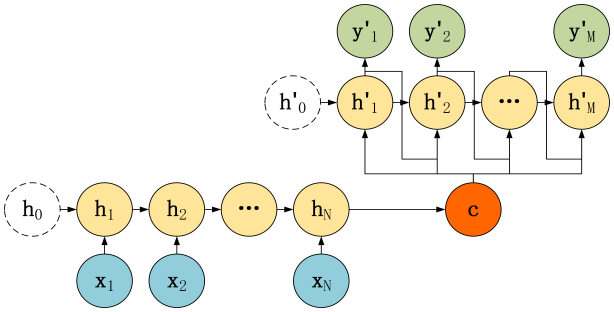


Fig. 5: Overall architecture

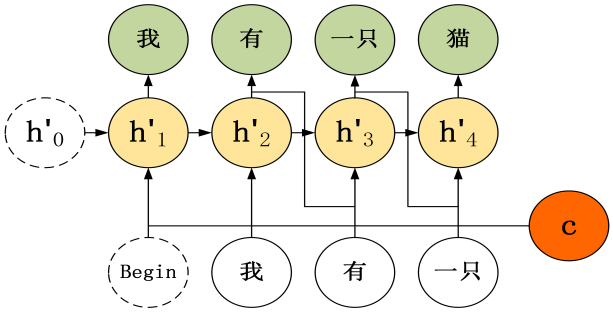


Fig. 7: With teacher forcing

### C. *Teacher Forcing*

There are 2 different training modes in RNN:

- free-running mode
- teacher-forcing mode

'Free-running' means the usual way we train our network with the output from last step as input to next step. 'Teacher forcing' is the concept of using the real target outputs as each next input, instead of using the decoder's guess as the next input. Using teacher forcing causes it to converge faster but when the trained network is exploited, it may exhibit instability.

The input of the Decoder model neuron includes the output of the previous neuron. If the output of the previous neuron is wrong, the output of the next neuron can easily be wrong, causing the error to be passed on forever. Teacher Forcing can alleviate the above problems to a certain extent. When training Seq2Seq model, every decoder neuron does not necessarily use the output of the previous neuron, but adopts the correct sequence as input in a certain proportion.

You can observe outputs of teacher-forced networks that read with coherent grammar but wander far from the correct translation - intuitively it has learned to represent the output grammar and can 'pick up' the meaning once the teacher tells it the first few words, but it has not properly learned how to create the sentence from the translation in the first place.

Because of the freedom PyTorch's autograd gives us, we can easily choose to use teacher forcing or not with a simple if statement. Turn the parameter named teacher_forcing_ratio_up to use more of it. Following 2 figures show the difference of applying or not applying teacher forcing. In conclusion, teaching forcing is a technique for faster training.

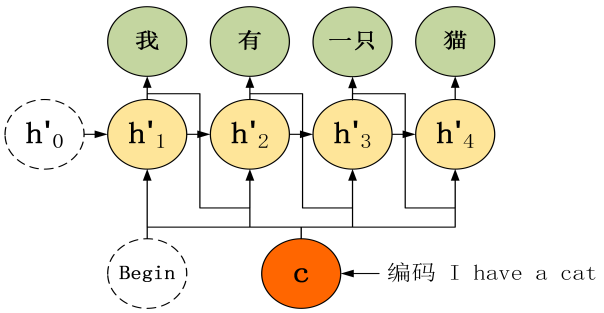


Fig. 6: Without teacher forcing

## III. Experiments

### A. *Dataset and Environment*

We use Chinese and English translation dataset called cmn-eng on the website. There are a total of 23,610 translation data pairs, and each pair of translation data is on the same line: English on the left, Chinese in the middle, and other attributes information on the right. The separator is
t.

The environment for the experiment is as follows:

- Pytorch
- python3 including sklearn, numpy, jupyter, matplotlib

### B. *Coding and processing*

Before training, we should preprocess training data.To train, for each pair we need an input tensor (indexes of the words in the input sentence) and target tensor (indexes of the words in the target sentence). Moreover, we append the EOS token to both sequences while creating these vectors.

Thus we used three methods, namely indexesFromSentence, tensorFromSentence and tensorsFromPair to complete this task.

Listing 1: **3 methods.py**

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in
        sentence.split(' ')]

def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.
        long, device=device).view(-1, 1)

def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang,
        pair[0])
    target_tensor = tensorFromSentence(output_lang
        , pair[1])
    return (input_tensor, target_tensor)
```

To train we run the input sentence through the encoder, and keep track of every output and the latest hidden state. Then the decoder is given the SOS token as its first input, and the last hidden state of the encoder as its first hidden state.

We refer to a Pytorch tutorial to get the sample codes of the machine translation model based on the attention mechanism. The Pytorch tutorial is written by Sean Robertson. The methods are as follows: In this experiment we will be teaching a neural network to translate from Chinese to English, which is achieved by the simple but powerful idea of the sequence to sequence network.

Seq-to-seq network includes two recurrent neural networks work together to transform one sequence to another, which employees an encoder-decoder architecture.

An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence. To improve upon this model, we'll use an attention mechanism, which lets the decoder learn to focus over a specific range of the input sequence.

Listing 2: **Encoder and Decoder.py**

```
class EncoderRNN(nn.Module):
    def forward(self, input, hidden):
        embedded = self.embedding(
            input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size,
            device=device)

class AttnDecoderRNN(nn.Module):
    def forward(self,
        input, hidden, encoder_outputs):
        embedded = self.embedding(
            input).view(1, 1, -1)
        embedded = self.dropout(embedded)
        attn_weights = F.softmax(self.attn(torch.
            cat((embedded[0], hidden[0]), 1)), dim
            =1)
        attn_applied = torch.bmm(attn_weights.
            unsqueeze(0),encoder_outputs.
            unsqueeze(0))
        output = torch.cat((embedded[0],
            attn_applied[0]), 1)
        output = self.attn_combine(output).
            unsqueeze(0)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = F.log_softmax(self.out(output[0])
            , dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size,
            device=device)
```

After loading the dataset and preprocessing, we split words from the training sentences and construct a comparison table of Chinese and English words, which appear in the code in the form of pairs. We need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we use a helper class called Lang which has word to index (word2index) and index to word (index2word) dictionaries, as well as a count of each word word2count to use to later replace rare words. Because the files are all in Unicode, to simplify we turn Unicode characters to ASCII in lowercase, and trimmed most punctuation. Thus, we define two methods called unicodeToAscii(s) and normalizeString(s).

Listing 3: **data preprocessing.py**

```
pairs = [[normalizeString(s) for s in l.split('\t')
    ][:2]] for l in lines]
# split the input words in pairs
def normalizeString(s):
    s = s.lower().strip()
    if ' ' not in s:
        s = list(s)
        s = ' '.join(s)
    s = unicodeToAscii(s)
    s = re.sub(r"([.!?])", r" \1", s)
    return s
```

To read the data file we split the file into lines, and then split lines into pairs. Then we use a method called readLangs to achieve it, in the method, we only extract top-2 split for each line when constructing the pairs. The reverse parameter can be set to achieve a reserved translation.

Listing 4: **readLangs.py**

```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")
    # Read the file and split into lines
    file_path = "./data/eng-cmn.txt"
    with open(file_path, encoding='utf-8') as
        file:
        lines = file.readlines()
    # Split every line into pairs and
        normalize
    pairs = [[normalizeString(s) for s in l.split(
        '\t')[:2]] for l in lines]
    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(
            reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)
    return input_lang, output_lang, pairs
```

Since there are a lot of example sentences, to minimize training time, we trim the dataset to only relatively short and simple sentences.

Here the maximum length is 10 words (that includes ending punctuation) and we're filtering to sentences that translate to the form 'I am' or 'He is' etc. There are two methods called filterPair and filterPairs defined. And there is a method called prepareData to prepare the data in a full process.

### C. *Parameters and Optimizers*

We set the value of some parameters. Hidden_size is 256, n_iters is 75000, print_every is 5000, plot_every is 100, learning_rate is 0.01. We use SGD to optimize encoder and decoder, and we use function NLLLoss() to get the loss.

Listing 5: **parameters.py**

```
encoder_optimizer = optim.SGD(encoder.parameters()
    , lr = learning_rate )
decoder_optimizer = optim.SGD(decoder.parameters()
    , lr = learning_rate )
training_pairs = [tensorsFromPair(random.choice(
    pairs ))
for i in range( n_iters )]
    criterion = nn.NLLLoss()
```

### D. *Training and Evaluation*

Then we call train many times and occasionally print the progress (percentage of examples, time so far, estimated time) and average loss. The whole training process can be expressed like this:

- Start a timer;
- Initialize optimizers and criterion;
- Create set of training pairs; Start empty losses array for plotting.

After training, we plotted the loss curve with matplotlib, Whose values come from loss values plot_losses saved while training.

Evaluation is mostly the same as training, but there are no targets so we simply feed the decoder's predictions back to itself for each step. Every time it predicts a word we append it to the output string, and if it predicts the EOS token we stop there. We also store the decoder's attention outputs for display later. We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements when we called the function evaluateRandomly.

We also have a visualizing Attention. A useful property of the attention mechanism is its highly interpretable outputs. Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step. We also run plt.matshow (attentions) to represent attention output displayed as a matrix, with the columns being input steps and rows being output steps. For a better visualization effects we do the extra work of adding axes and labels, using the methods showAttention() and evaluateAndShowAttention().

Listing 6: **Data visualization.py**

```
def showAttention(input_sentence, output_words,
    attentions ):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='
        bone')
    fig.colorbar(cax)
    # Set up axes
    ax.set_xticklabels(['']+ input_sentence.split
        (' ')+
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels(['']+ output_words)
    # Show label at every tick
    ax.xaxis.set_major_locator(ticker.
        MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.
        MultipleLocator(1))
    plt.show()

def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words,
        attentions )
```

Following are the results of the training process.

*1) Qualified Results:*

```
4m 49s (- 67m 29s) (5000 6%) 2.2736
9m 45s (- 63m 24s) (10000 13%) 0.5236
14m 53s (- 59m 35s) (15000 20%) 0.0795
19m 57s (- 54m 53s) (20000 26%) 0.0480
25m 0s (- 50m 1s) (25000 33%) 0.0402
30m 4s (- 45m 6s) (30000 40%) 0.0383
35m 12s (- 40m 14s) (35000 46%) 0.0376
40m 18s (- 35m 16s) (40000 53%) 0.0357
45m 25s (- 30m 17s) (45000 60%) 0.0341
50m 37s (- 25m 18s) (50000 66%) 0.0329
55m 48s (- 20m 17s) (55000 73%) 0.0309
61m 1s (- 15m 15s) (60000 80%) 0.0292
66m 11s (- 10m 11s) (65000 86%) 0.0294
71m 17s (- 5m 5s) (70000 93%) 0.0355
76m 28s (- 0m 0s) (75000 100%) 0.0309

<Figure size 432x288 with 0 Axes>
```

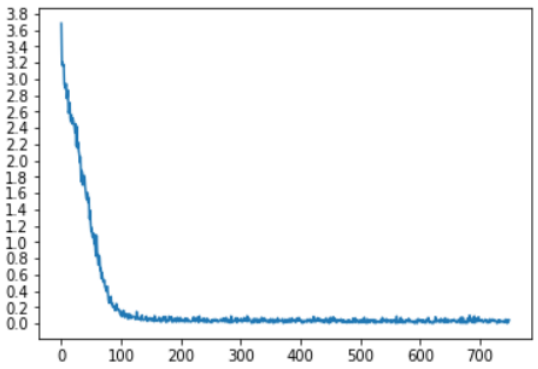


Fig. 8: Loss curve

We use evaluateRandomly() to test it and use sentence_bleu to get the score. We just choose 100 samples randomly. In the first image, for every group, the first line is the input, the second line is the ground truth, the third line is the output.

```
> 他 开 始 掉 头 发 了
= he is beginning to lose his hair
< he is beginning to lose his hair <EOS>
```

Fig. 9: Output results

```
evaluateRandomly(encoder1, attn_decoder1)
executed in 2.18s, finished 20:56:04 2020-11-24
= i am coming
< i am coming <EOS>

> 她 在 节 食
= she is on a diet
< she is dieting a diet <EOS>

> 她 跟 我 生 气 了
= she is mad at me
< she is mad at me <EOS>

> 我 在 下 一 站 下 车
= i am getting off at the next station
< i am getting off at the next station <EOS>

> 他 會 彈 吉 他
= he is able to play the guitar
< he is able to play the guitar <EOS>

The bleu_score is  0.9366917348932489
```

Fig. 10: Output results

```
evaluateRandomly(encoder1, attn_decoder1)
executed in 2.23s, finished 21:15:02 2020-11-24
= i am interested in english
< i am interested in english <EOS>

> 他 怕 得 到 肺 癌
= he is afraid of getting lung cancer
< he is afraid of getting lung cancer <EOS>

> 他 是 個 好 人
= he is a good fellow
< he is a good fellow <EOS>

> 我 会 开 车
= i am able to drive a car
< i am able to drive a car <EOS>

> 她 很 可 爱
= she is very pretty
< she is very pretty <EOS>

The bleu_score is  0.9859460355750136
```

Fig. 11: Output results

*2) Visualization:*

The image shows the weight value corresponding to every word in the input and every word in output.

```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "他 很 少 心 情 很 好")
plt.matshow(attentions.numpy())
plt.show()
executed in 474ms, finished 20:02:12 2020-11-24
```

Fig. 12: Test code

In similar ways, we tested the model with following input.

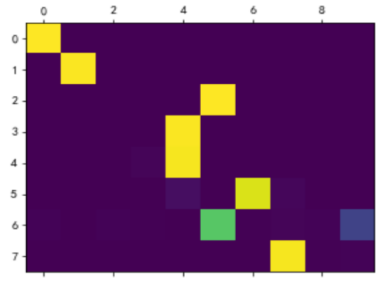


Fig. 13: Result visualization for test code

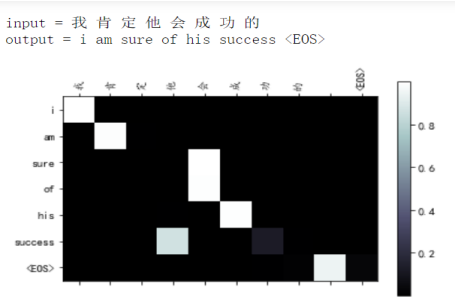


Fig. 14: Result visualization

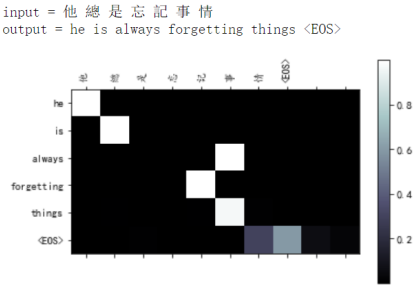


Fig. 15: Result visualization

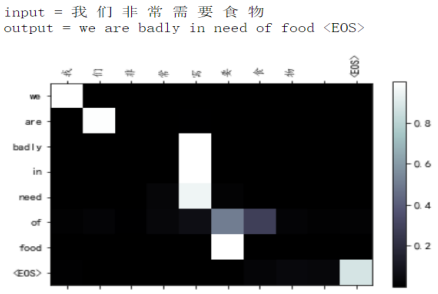


Fig. 16: Result visualization

However,the result is not alway good. When the RNN encounters with some texts or text patterns that haven't appear before, it's output is unsatisfying. For example, when we input the text "you are just saying" or "you just play" in Chinese, its outputs are similar, which is not our expected result.

The reason why this situation happens, from our opinion, has something to do with the teacher forcing mode. Although teacher forcing mode makes RNN easier to train and has a better score on the training set, it may leads to an unnecessary

```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "你 只 是 說 說")
print(output_words)
plt.matshow(attentions.numpy())
plt.show()
```

```
['you', 'are', 'a', 'coward', '<EOS>']
```

Fig. 17: Correct output

```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "你 只 是 玩")
print(output_words)
plt.matshow(attentions.numpy())
plt.show()
```

```
['you', 'are', 'so', '<EOS>']
```

Fig. 18: Incorrect output

dependence on the training set. As a result, when the network encounters with something it haven't learnt, its ability to learn new things is extremely limited. That is to say, if testing data and training data are from two totally different field, the performance will get worse.

According to the information we found on the Internet, there are some optimizations for this dilemma.

- Beam Search
  When we predict the discrete output like words, a common way is to search the probability of each word in the wordlist, and thus generate multiple candidate sequences. This approach is usually applied to machine translation problem to improve the output sequence. This search is a kind of heuristic search, which can reduce the performance difference between training stage and testing stage.
- Curriculum Learning
  If the model is to predict real-value instead of discrete value, the Beam Search cannot be used. Thus, we can apply Curriculum Learning, which is originated from teacher forcing. This method forces the model to deal with its mistakes gradually in training process. We apply a probability $p$ to choose whether to use the output of groud truth $y(t)$ or the output at last time $h(t)$ as an input to current input. Additionally, this probability changes during the process, which is called scheduled sampling. The model will reduce the possibility to use help from ground truth gradually.

## IV. Conclusion

From the results above, we find that the model is convergent before the 200th iteration and the precision is high but unstable.

A simple and short sentence as input can contribute to the good result. The large iteration times is another reason. Moreover, we also find that attention mechanism has a great effect on the performance.

The whole attention and attention mechanism can been regarded as an equivalent to adding a layer of 'packaging' on the Seq2Seq structure, and internally calculate the Attention vector through function score(), so as to add additional information to Decoder RNN to improve performance. In fact, for various tasks such as machine translation, speech recognition, natural language processing (NLP) and text recognition (OCR), attention mechanism is of a significant help.

We also have knowledge of this model's drawbacks and methods to improve it.For example, we know that in a general Seq2Seq model, Encoder always encodes all the information of source sentences into a fixed length context vector c, which stays unchanged during the Decoder decoding process. This characteristic have many drawbacks. For long sentences, it is difficult to express their meaning completely with a vector c of fixed length. RNN suffers from gradient disappearance for long sequence due to its highly recursive structure. As a result, the vector obtained by including only the last neuron's output is not ideal.

When we read a text, we pay attention to the current sentence. However, unlike human attention, the network is unable to focus its attention on specific parts. As a result, when equipped with an attention decoder, the network gets a better performance.

Moreover, we know more about some training strategies of RNN, for example, the teaching force and its improvement, Beam Search and Curriculum Learning. After the experiment, we understand natural language processing and the classic Sequence-to-Sequence machine translation model. We master the application of attention mechanism in machine translation model. We also built a machine translation models, verified model performance on simple and small-scale datasets, thus cultivate our engineering capabilities.

The Seq2Seq model allows us to use input and output sequences of varying lengths for a wide range of scenarios such as machine translation, conversational systems, and reading comprehension.

It can be optimized by Teacher Forcing, Attention Mechanism, Beam Search and other methods. There is a lot more for us to explore about the Seq2Seq model.

## V. References

[1] Sutskever I , Vinyals O , Le Q V . Sequence to Sequence Learning with Neural Networks[J]. Advances in neural information processing systems, 2014.

[2] Sean Robertson. NLP FROM SCRATCH: TRANSLATION WITH A SEQUENCE TO SEQUENCE NETWORK AND ATTENTION.

[3] Cho, K., Merrienboer, B.V., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. ArXiv, abs/1406.1078.

[4] Bahdanau D , Cho K , Bengio Y . Neural Machine Translation by Jointly Learning to Align and Translate[J]. Computer ence, 2014.