VII. Lab Seven - 31298516

Sequence-to-sequence learning(Seq2Seq) is a machine learning technology aims to train models for converting sequences from one domain to sequences in another domain which is quite different from the previous area of computer vision.

This is widely used in the area of natural language processing(NLP) for machine translation or for free-from question answering. And the encoder and decoder, recurrent neural network(RNN) and long-short term memory network(LSTM) are the universal foundation of the researches in this area.

In this lab, we implemented a fundamental Seq2Seq model for a translation task that convert Morse code to English letters.

A. Complete and Train a Sequence-to-Sequence Model

The following code snippet is how I implement the feedforward function, under the lab guidance which says the forward method should return the hidden state and the cell state of the LSTM as its output.

```
def forward(self, src_batch):
    # src [sent len, batch size]
    embedded = self.embedding(src_batch)
        ↪  # [sent len, batch size, emb
        ↪ dim]
    outputs, hidden = self.rnn(embedded)
        ↪  # [sent len, batch size,
        ↪ hidden dim]
    # outputs -> [sent len, batch size,
        ↪ hidden dim * n directions]
    # hidden -> [n layers * n directions
        ↪ , batch size, hidden dim]
    return outputs, hidden
```

After the forward function being finished, we can train the model and plot the loss curve which is shown in 'Figure 1'.

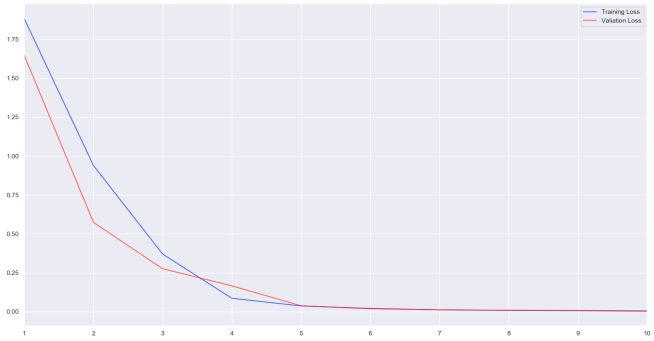

Figure 1: Curve of the loss function of our Matrix Factorisation implementation while epochs=1000 and lr=0.005.

B. Now Use It!

By sending codes through the decoder, we can get:

```
code1 = '.--.␣...␣.--.␣.-.␣/-....␣.␣/..-.␣
    ↪ ---.-..␣.-..␣---.--..␣-.␣--.'
print('The␣first␣code␣can␣be␣converted␣as:␣'
    ↪ ,decode(code1))
>>> The first code can be converted as:
    ↪ psprbefllzng
code2 = '.--....␣-.--/..␣...␣/-....␣.␣
    ↪ /---.-.␣-..␣.␣.-.␣/---..-.␣/-␣....␣.␣
    ↪ /---..--.--.␣..--/.-.␣.␣...-.␣.-.␣...
    ↪ ␣.␣-..'
print('The␣second␣code:␣',decode(code2))
>>> The second code: hlsbepderfthepfefrsed
code3 = '.--....␣.--/..␣...␣/-....␣.␣/.--.␣
    ↪ ---..␣-.␣-/---..-.␣/-.␣.--.-.␣....␣.␣
    ↪ .-.␣/..-.␣---.-.␣-.-.␣..␣-.␣--.'
print('The␣third␣code:␣',decode(code3))
>>> The third code: hlsbepzncnpherfpcing
```

Besides, attention is introduced into this model; hence the input is reversed. Therefore there is [::-1] at the end of the given code snippet, to reverse the output again. It is quite useful on solving the gradient disappears while the sentence is overly long. As to the 'teacher forcing', which is efficient for quickly training RNN models that use the ground truth from a previous time step as input. It allows trained models better to handle open loop applications of this type of network. It also performs well on addressing slow convergence and instability while training these types of recurrent networks.

C. Sequence Lengths

If the model works with longer chunks, it may output a sequence that is not a word in the target vocabulary list so it will be difficult to interpret it accurately. Also, if the model works with longer chunks, it may not be able to know where they should be inserted spaces.

Apart from this, people have been attaching more and more importance to the attention in the original Seq2Seq model. It helps us focus on the specific part of the sentence, and can also help us capture the long term dependency. However, if we feed the overly long chunks to the model, it may perform negatively on obtaining the long term dependency.

I reckon that while the training set and test set are similar in length, the model performs well, and our chunks are relatively short. If the training set and test set are very different; the model will work with longer chunks, and the performance of the model will be negatively influenced.

As to the dataset,