Chinese Word Segmentation - Report

Yutong Zou

18340238

1 Introduction

Chinese word segmentation (CWS) has been a fundamental work for NLP tasks in terms of dealing with Chinese words. Word segmentation is usually the first step for many other downstream tasks such as machine translation and sentiment analysis. Therefore, it is meaningful to study CWS.

For most cases, CWS is accomplished by sequence tagging. Previous work has explored many sequence tagging models, including Hidden Markov Models (HMM), Long Short-Term Memory Neural Networks (LSTM) (Hochreiter and Schmidhuber, 1997), Conditional Random Fields (CRF) (Lafferty et al., 2001) and bidirectional LSTM networks with a CRF layer (BiLSTM-CRF) (Huang et al., 2018). The last model proved to be very efficient in terms of CWS tasks.

In this report, I mainly introduce my understanding and experiments on a given CWS task using BiLSTM-CRF model. The dataset for this task is from MSR SIGHAN 2005 bake-off task (Emerson, 2005). In the following sections of this report, I would mainly focus on network architectures, data preprocessing steps, key parts of code implementation and experiment analysis.

2 Model

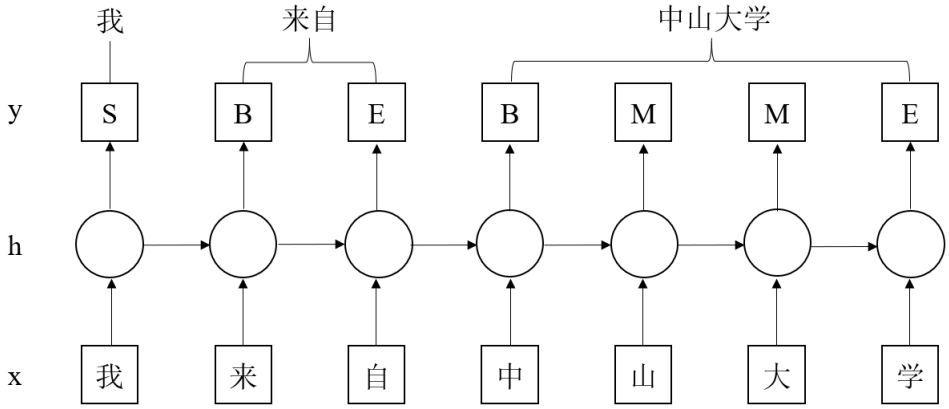
In this sections, I mainly describe the model architectures and theories of its components.

2.1 Background

In this report, the final model is composed of a BiLSTM network with a CRF layer. I would first introduce biLSTM and CRF respectively.

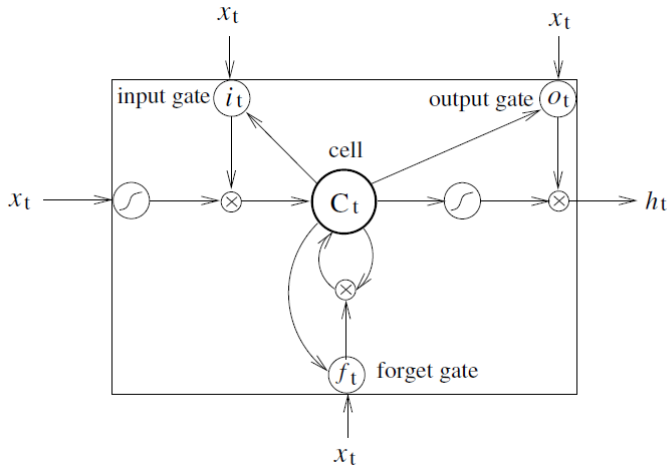
2.1.1 LSTM Networks

The long short-term memory neural network (Hochreiter and Schmidhuber, 1997) addresses the problem of learning long-term dependencies. In this case, namely sequence tagging, an LSTM network can learn the context meaning of a sentence, and its hidden layers can be used to predict tags for each Chinese character. Each character can be tagged as *seperate (S)*, *begin (B)*, *middle (M)* or *end (E)*. Characters with *S* tags can be single-character Chinese words. Two characters with *BE* tag can compose a two-character Chinese word. More than three characters that begin with *B* tag and end with *E* tag, with mutiple *M* tags in the middle can compose an n-gram Chinese word. The following figure shows how LSTM network works for Chinese word sequence tagging, with input layer x, hidden layer h and output layer y.



Suppose the input Chinese sentence is composed of l Chinese characters $\boldsymbol{S} = (c_1, c_2, \dots c_l)$. Adopting pre-trained word embeddings (the details of which would be explained in section 3.1) to each word, we obtain embedding representations for each word $\boldsymbol{X} = (\boldsymbol{e_1}, \boldsymbol{e_2}, \cdots, \boldsymbol{e_l})$.

The following figure illustrates a single LSTM memory cell (Graves et al., 2005).



The LSTM transition equations of a single cell are as follows,

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{Ci}C_{t-1} + b_i) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{Cf}C_{t-1} + b_f) \\
C_t &= f_t C_{t-1} + i_t \tanh\left(W_{xC}x_t + W_{hC}h_{t-1} + b_C\right), \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{Co}C_t + b_o) \\
h_t &= o_t \tanh(C_t)
\end{aligned}
$$

where σ is the logistic sigmoid function, and i, f, o and C are the input gate, forget gate, output gate and cell vectors respectively.

2.2.2 Bidirectional LSTM Networks

While forward LSTM network can learn past features of a given sequence, going backward can learn future features of it. BiLSTM combines both forward and backward states, the output representation of which is the concatenation of forward and backward LSTM hidden states.

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}].$$

2.3.3 CRF

Conditional random field (Lafferty et al., 2001) models the probability distribution of a given sequence. Instead of indicidual positions, CRF focuses on sentence level information. A CRF network has two vital features: emission scores and transition scores. In the case of sequence tagging, emission scores indicate the probabilities of tags given a known word, and transition scores indicate the probabilities of the next tag given current tags. For

an input sequence $\boldsymbol{X} = (x_1, x_2, \cdots, x_l)$ and a sequence of predictions $\boldsymbol{y} = (y_1, y_2, \cdots, y_l)$ of length l. Consider the emission scores matrix E of size $l \times k$, where k is the classes of tags. We define the CRF scores to be

$$s(\boldsymbol{X}, \boldsymbol{y}) = \sum_{i=0}^{l} T_{y_i, y_{i+1}} + \sum_{i=1}^{l} E_{i, y_i},$$

where T is the transition scores matrix such that $T_{i,j}$ represents the score of a transition from the tag i to tag j. y_0 and y_l are the *start-of-sentence* and *end-of-sentence* tags of a sentence. T is therefore a square matrix of size $k + 2$.

Therefore, we apply a softmax to the scores to represent a likelihood for the sequence y:

$$p(\boldsymbol{y}|\boldsymbol{X}) = \frac{e^{s(\boldsymbol{X}, \boldsymbol{y})}}{\sum_{\hat{\boldsymbol{y}} \in \boldsymbol{Y_X}} e^{s(\boldsymbol{X}, \hat{\boldsymbol{y}})}}.$$

We need to maximize the probability for the prediction sequence. Thus, the objective function we need to optimize is the negative log-likelihood:

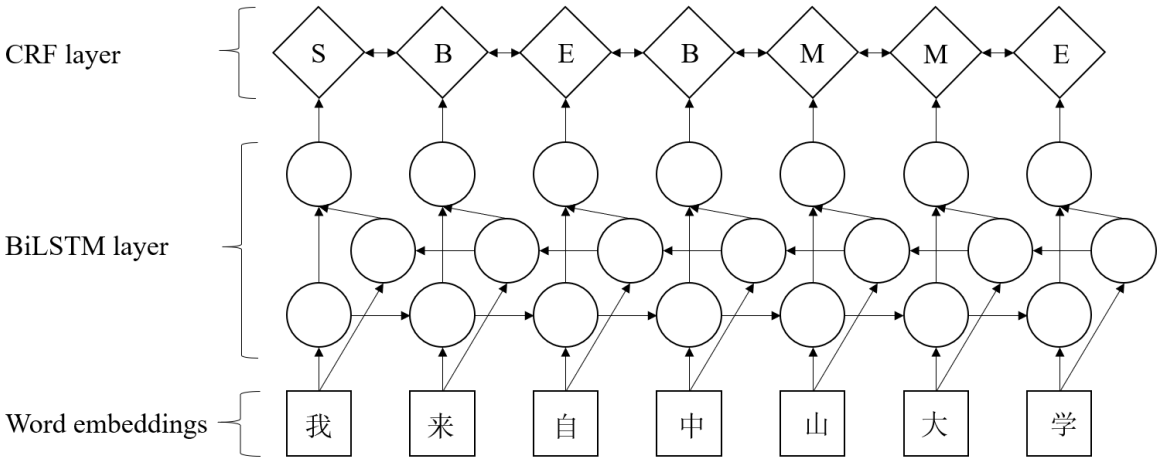
$$\begin{aligned} J &= log(p(\boldsymbol{y}|\boldsymbol{X})) \\ &= s(\boldsymbol{X}, \boldsymbol{y}) - log(\sum_{\hat{\boldsymbol{y}} \in \boldsymbol{Y_X}} e^{s(\boldsymbol{X}, \hat{\boldsymbol{y}})}), \end{aligned}$$

where $\boldsymbol{Y_X}$ represents all possible tag sequences for a sentence $\boldsymbol{X}$.

2.2 BiLSTM-CRF

BiLSTM can learn context features and long-distance information of a given sentence, but it cannot guarantee the validity of the output tag sequence. For example, instead of *BE* or *BME*, which starts with *B* and ends with *E*, the output of an BiLSTM network could be *EB* or *ME*. CRF can restrain the order of a tag sequence, but it cannot learn context information. We need to find a way to both guarantee the validity of tag sequene and learn its context features. Thus, we introduce BiLSTM-CRF, which adds an additional CRF layer at the top of a BiLSTM network. Consequently, we use the hidden layers of BiLSTM network to represent emission scores of the CRF layer.

The overall network architecture of my work is illustrated in the following figure:



3 Experiment Procedure

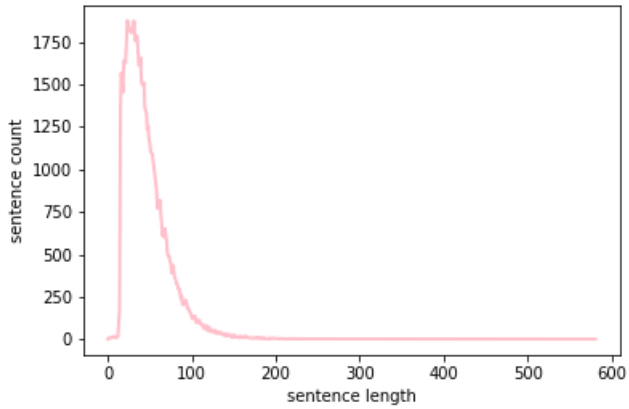
3.1 Word Embedding

Instead of complext pre-trained models such as ELMo or BERT, simple pre-trained word vectors are applied in this experiment. This is because most pre-trained models are based on English corpus. There are few Chinese-based pre-trained models indeed, but the scale of their corpus is rather small and the corpus resources are not abundant enough. Thus I chose the Pre-Trained Chinese Word Vectors (Li et al., 2018) for this experiment.

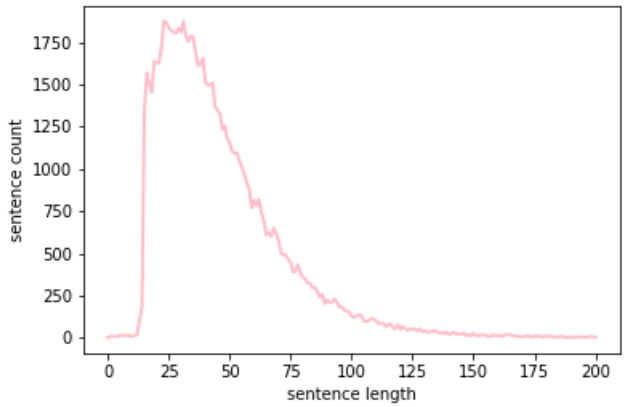
3.2 Data Preprocessing

3.2.1 Data Cleaning

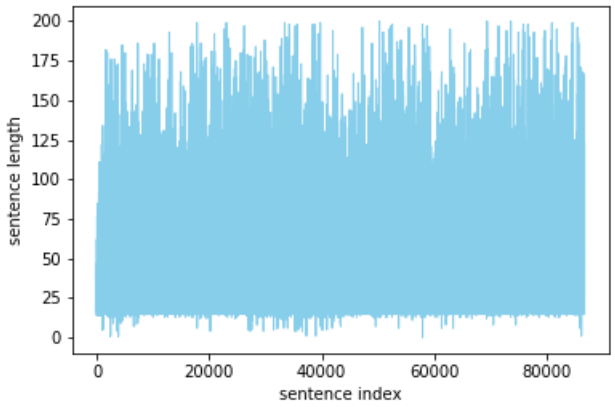
Observing the distribution of sentence length as shown in the following figure,



notice that the sentence lengths can be very large (more than 200), yet the number of long sentences is small. Therefore, I decided to discard sentences longer than 200. After discard, the distribution of sentence length is as follows:



Moreover, as shown in the following figure, we notice that the sentence length distribution by index is extremely irregular.



Obviously, this experiment requires the input sequence length to be equal by padding zero or random vectors in each batch. Even after adopting some measures to change padding size according to each batch, using original datasets withou sorting could make padding size very large. This could evntually let the model learn many noisy features. Thus, I decided to sort the datasets according to sentence length in advance.

The code of data cleaning is as follows:

```
def data_cleaning(df):
    df['sentence'] = df['original'].apply(lambda x: x.replace(' ', ''))
    df['words'] = df['original'].apply(lambda x: [a for a in x.split(' ') if a != '' ])
    df['length'] = df['sentence'].apply(lambda x: len(x))
    # discard long sentences
    df = df.drop(df[df['length'] > pad_size].index)
    # discard short sentences
    df = df.drop(df[df['length'] <= 5].index)
    # sort sentences by lengths
    df = df.sort_values(by = 'length')
    df = df.drop('length', axis = 1)
    df = df.reset_index(drop = True)
    return df
```

3.2.2 Dataloader

To train data in the PyTorch framework, we need to load the data into a Dataloader. Part implementation of loading word embeddings and Dataloader is shown in the following code blocks.

```
vocab_model = KeyedVectors.load_word2vec_format('sgns.context.word-character.char1-1.bz2',
binary = False, encoding = "utf-8", unicode_errors = "ignore", limit = None)
pad_size = 200
tag2idx = {'S': 0, 'B': 1, 'M': 2, 'E': 3}
idx2tag = {0: 'S', 1: 'B', 2: 'M', 3: 'E'}
PAD_IDX = 0 # index in word embeddings for padding words
UNK_IDX = 1 # index in word embeddings for unkown words
# randomize word vectors for padding words and unknown words
pad_vec = np.random.normal(size = 300).reshape(1, -1)
unk_vec = np.random.normal(size = 300).reshape(1, -1)

word_embeddings = np.concatenate((pad_vec, unk_vec, vocab_model.vectors), axis = 0)
vocab_size = len(word_embeddings)
```

```
class MSRSEG(Dataset):
    def __init__(self, df):
        self.df = df # df is the original data loaded as pandas.DataFrame
        self.df = data_cleaning(self.df) # data cleaning
```

```
        self.df['tags'] = self.df['words'].apply(tag)
        self.df['sentence_idx'] = self.df['sentence'].apply(id_and_pad_sentence)

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        sentence_idx = self.df['sentence_idx'].iloc[idx]
        tags = self.df['tags'].iloc[idx]
        return torch.LongTensor(np.array(sentence_idx)), torch.LongTensor(np.array(tags))

'''
each batch has different padding size,
in case there are too many paddings for very short sentences
'''
def collate_fn(batch):
    # sort sentences by lengths in each batch
    batch.sort(key = lambda x: len(x[0]), reverse = True)
    sentence = [item[0] for item in batch]
    tags = [item[1] for item in batch]
    length_list = [len(x[0]) for x in batch]
    # pad sentences and their corresponding tags
    padded_sentence = rnn_utils.pad_sequence(sentence, batch_first=True,
padding_value=PAD_IDX)
    padded_tags = rnn_utils.pad_sequence(tags, batch_first=True, padding_value=tag2idx['S'])
    return padded_sentence, padded_tags, length_list
```

```
train= pd.read_csv('msr_training.utf8', header = None) # read train data
train.columns = ['original']
test = pd.read_csv('msr_test.utf8', header = None) # read test data
test.columns = ['sentence']
test_gold = pd.read_csv('msr_test_gold.utf8', header = None)
test.insert(0, 'original', test_gold)

train_dataset = MSRSEG(train)
train_loader = Data.DataLoader(dataset=train_dataset,
                               batch_size=batch_size,
                               collate_fn=collate_fn,
                               shuffle=False)
test_dataset = MSRSEG(test)
test_loader = Data.DataLoader(dataset=test_dataset,
                              batch_size=batch_size,
                              collate_fn=collate_fn,
                              shuffle=False)
```

3.3 Model Constructing

The BiLSTM-CRF model is implemented under the PyTorch framework.

```
class biLSTM_CNN_CRF(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, tag_to_idx, dropout_rate):
        super(biLSTM_CNN_CRF, self).__init__()
        # class initiation
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.hidden_size = hidden_size
        self.num_classes = num_classes
        self.tag_to_idx = tag_to_idx
        self.tagset_size= len(tag_to_idx)
        # load word embeddings
        self.embedding = nn.Embedding(self.vocab_size, embedding_dim=self.embedding_dim,
padding_idx=PAD_IDX)
        self.embedding.from_pretrained(torch.from_numpy(word_embeddings), freeze=True)

        self.biLSTM = self.biLSTM = nn.LSTM(input_size=self.embedding_dim,
hidden_size=self.hidden_size, batch_first=True,bidirectional=True)
```

```
        # a linear layer for tag classification
        self.linear = nn.Linear(self.hidden_size*2, self.tagset_size)
        # CRF layer is implemented using the torchcrf library
        self.crf = CRF(self.tagset_size, batch_first=True)

        self.init_linears()
        self.dropout = nn.Dropout(dropout_rate) # add some dropout to prevent overfitting

    # use xavier initiation to speed up convergence
    def init_linears(self):
        nn.init.xavier_uniform_(self.linear.weight)
        nn.init.zeros_(self.linear.bias)

    def forward(self, sentence, tags, length_list):
        embeds = self.embedding(sentence)
        '''
        the input of BiLSTM network is packed sequence
        this allows each batch to have different padding size
        '''
        packed_embeds = rnn_utils.pack_padded_sequence(input=embeds,
                                                       lengths=length_list,
                                                       batch_first=True,
                                                       enforce_sorted=True)
        lstm_output, _ = self.biLSTM(packed_embeds)
        # unpack the packed sequence
        lstm_output, _ = rnn_utils.pad_packed_sequence(lstm_output, batch_first=True)
        lstm_output = self.dropout(lstm_output)
        lstm_output = self.linear(lstm_output)

        emissions = lstm_output # use BiLSTM output as emission scores for CRF layer
        loss = -self.crf(emissions, tags)
        tag_seq_list = self.crf.decode(emissions) # viterbi decoder

        return tag_seq_list, loss
```

3.4 Evaluation

Usually, the performance of CWS models is evaluated by F1-score instead of accuracy. In this experiment, the F1-score is calculated as follows:

$$\begin{aligned}\text{precision} &= \frac{\text{number of correct predicted word segmentations}}{\text{number of all predicted word segmentations}}\\ \text{recall} &= \frac{\text{number of correct predicted word segmentations}}{\text{number of all standard word segmentations}}\\ \text{F1-score} &= \frac{2\cdot\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}.\end{aligned}$$

To compute F1-score, we need to convert tag seqence into segmentation sequence. Instead of converting into words directly, I used a set of tuples to represent segmentation result, where the first element of the tuple is the beginning index of a word while the second element is the end index.

```
def to_words_list(L):
    words_list = []
    start = 0
    while start < len(L):
        end = start + 1
        if idx2tag[L[start]] == 'S':
            words_list.append((start, start))
            start += 1
        elif idx2tag[L[start]] == 'B':
            while end < len(L) and idx2tag[L[end]] == 'M':
                end += 1
            words_list.append((start, end))
            start = end + 1
        else:
            start += 1
```

```
    return words_list

def evaluate(output, tags, length_list):
    tags = tags.cpu().numpy().tolist()
    num_same_words = 0
    num_output_words = 0
    num_tags_words = 0
    predict_words = []
    for i, item in enumerate(length_list):
        output[i] = output[i][:item]
        tags[i] = tags[i][:item]

        predict_words.append(to_words_list(output[i]))
        # a set of tuples
        output_words_list = set(predict_words[-1])
        tags_words_list = set(to_words_list(tags[i]))
        # number of same words is the size of the intersection of two sets
        num_same_words += len(output_words_list & tags_words_list)
        num_output_words += len(output_words_list)
        num_tags_words += len(tags_words_list)

    precision = 0 if num_output_words == 0 else num_same_words / num_output_words
    recall = 0 if num_tags_words == 0 else num_same_words / num_tags_words
    F1 = 2 * precision * recall / (precision + recall + 1e-6)
    return F1, precision, recall, predict_words
```

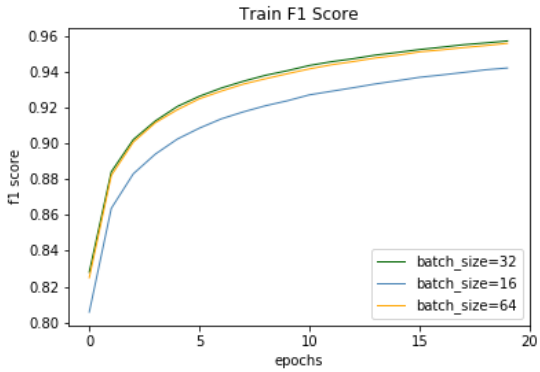
4 Result Analysis

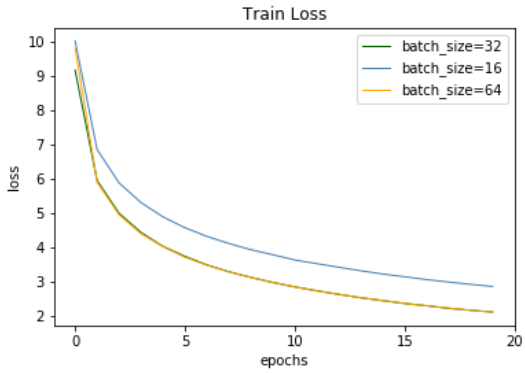
In this experiment, the BiLSTM-CRF model was applied to the dataset from MSR SIGHAN 2005 bake-off task (Emerson, 2005). The dataset has more than 80K sentences in train set and more than 3K sentences in test set, which are pretty sufficient to measure the performance of the model.

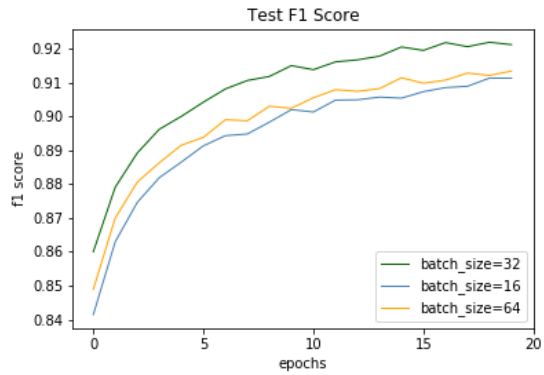
4.1 Training Configuration

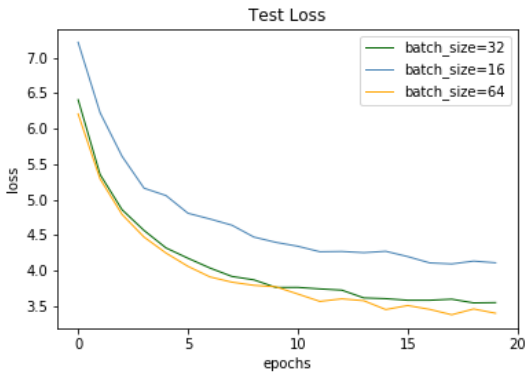
There are multiple hyperparameters that would affect the performance of the model, suche as batch size, dropout rate, learning rate and optimizer. In this section, I would demonstrate the result based on different combination of hyperparameters in this section.

Consider different batch size, using stochastic gradient descent (SGD) with a learning rate of 0.001, dropout rate of 0.5 and weight decay of 1e-4. The F1-score and loss curves in 20 epochs on train and test sets are shown in the following figures.



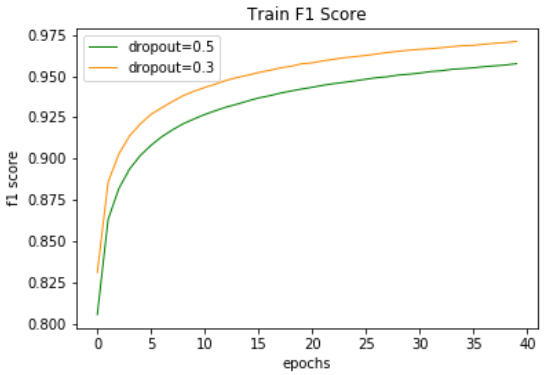


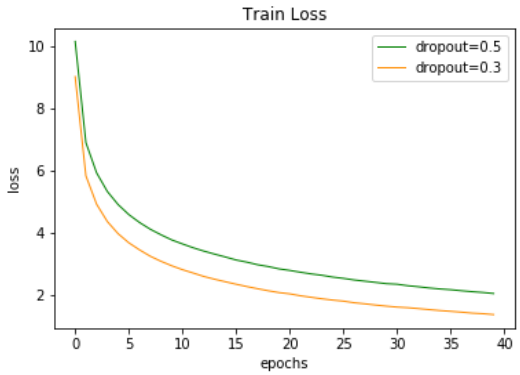


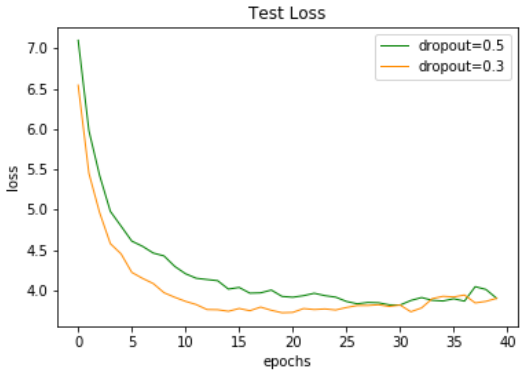


The above figures indicate that batch size of 32 yields the best result.

Consider different dropout rate, using SGD with a learning rate of 0.001 and weight decay of 1e-4, batch size set to 32. The F1-score loss curves in 40 epochs on train and test sets are shown in the following figures.

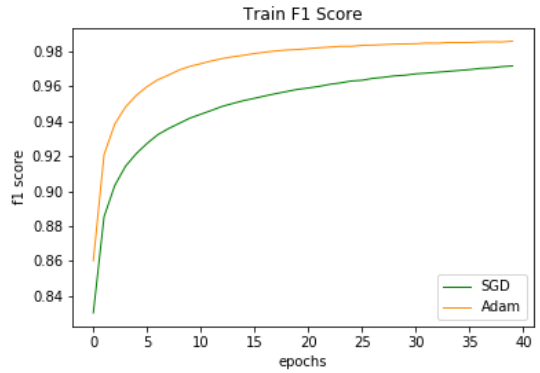


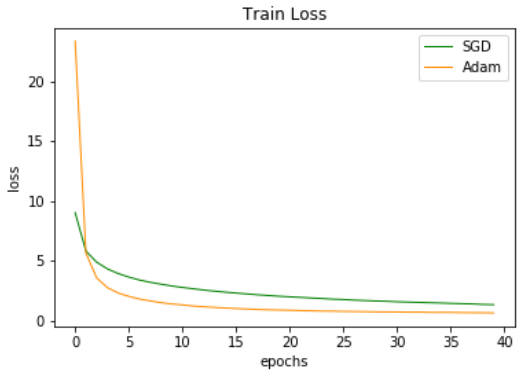


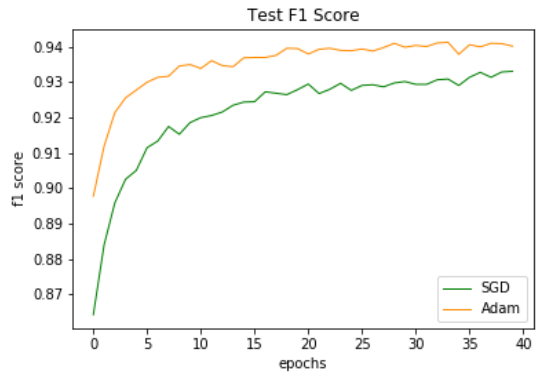


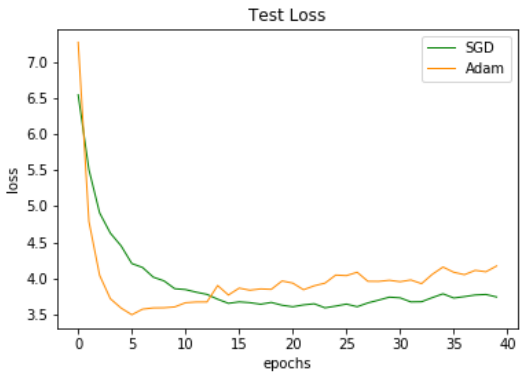
The above figures indicate that dropout rate of 0.3 yields a better result.

Consider different optimizers. Besides SGD, Adam (Kingma and Ba, 2014) is also a popular opimizer. With a learning rate of 0.001, dropout of 0.3, the F1-score and loss curves in 40 epochs on train and test sets are shown in the following figures.









The above figures indicate that Adam optimizer achives better performance in terms of F1-scores on bothe train and test set. Also, using Adam optimizer makes loss converges much faster than SGD does. However, Adam optimizer appears to yield slight overfitting on test set.

I have also set the learning rate to 0.01 and 0.0001, but the result is far worse than when the learning rate is 0.001. Therefore, curves of the result are omitted here.

The overall comparison is shown in the following table.

	Batch size			Dropout rate		Learning rate			Optimizer	
Values	16	32	64	0.3	0.5	0.01	0.001	0.0001	SGD	Adam
Train F1	94.21%	**97.12%**	95.59%	**97.11%**	96.89%	92.64%	**97.12%**	95.39%	97.16%	**98.58%**
Test F1	91.13%	**92.93%**	91.34%	**93.31%**	92.62%	89.76%	**92.93%**	90.08%	93.42%	**94.23%**

4.2 Error Analysis

In this part, we take a closer look at certain segmentation result on test set, where the best F1-score reaches 94.23%, and compare the predicted segmentation with the standard segmentation and analyze the errors in the result.

There are several error types. The most wrong predictions are wrongly seperating a long word, due to part of the word . For example, in the sentence '他/来到/中国/，/成为/第一个/访/华/的/大/船主/。', the model makes correct segmentation for all other characters except for the last three characters 大船主. Instead of the right segmentation 大/船主 which stands for 'famous' and 'captain' respectively, the predicted result is 大船/主 that stands for 'huge ship' and 'master' respectively. Combining different words in to a whole word is another common error. Consider the sentence '负责/开发区/工业/园/的/策划/规划', the prediction combines the first four words into one word '负责开发区工业园'. In the sentence '两/点/之间/直线/最/短/，/但/更/多/的/人/是/走/曲线/的/。', the predicted result is '两点' instead of '两/点' and '走曲线' instead of '走/曲线'. The meaning of the prediced segmentations also makes sense in Chinese, but they indeed can be devided furthur.

5 Conclusion

This experiment demonstrates that bidirectional long short-term memory network combined with a conditional random field layer can achieve high F1 scores on Chinese word segmentation tasks. For this particular task, an idea combination of hyperparameters is as follows,

Batch size	Optimizer	Learning rate	Dropout rate
32	Adam	0.001	0.3

whose F1-score can top to 94.23% on test set. Althougth there are some wrong segmentations, these errors are acceptable and actually do not affect the semantic meaning.

All in all, BiLSTM-CRF network is an ideal tool for Chinese word segmentation tasks.

References

[Hochreiter and Schmidhuber. 1997] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation, 9(8):1735–1780, 1997.

[Lafferty et al. 2001] J. Lafferty, A. McCallum, and F. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. Proceedings of ICML.

[Huang et al. 2018] Zhiheng Huang, Wei Xu, and Kai Yu. 2018. Bidirectional LSTM-CRF Models for Sequence Tagging. arxiv.

[Graves et al. 2005] A. Graves and J. Schmidhuber. 2005. Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures. Neural Networks.

[Li et al. 2018] Shen Li, Zhe Zhao, Renfen Hu, Wensi Li, Tao Liu, Xiaoyong Du. 2018. Analogical Reasoning on Chinese Morphological and Semantic Relations. Proceedings of ACL.

[Kingma and Ba. 2014] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.