中山大学计算机院本科生实验报告

(2020 秋季学期)

课程名称：自然语言处理　任课老师：权小军

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.11.22

1 Research Content

The Purpose of this experiment is to use BiLSTM+CRF word segmentation model to train and test Chinese word segmentation on the SIGHAN Microsoft Research data set.

2 Research Plan

2.1 Experimental Methods

2.1.1 BiLSTM

Long Short-Term Memory Network (LSTM) is a variant of recurrent neural network, which can effectively solve the problem of gradient explosion or disappearance of simple recurrent neural network.

The LSTM network introduces a new internal state $c_t \in R^D$, specifically for linear cyclic information transmission, and at the same time outputs information to the external state of the hidden layer $h_t \in \mathbb{R}^D$. The internal state c_t is calculated by the following formula:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t,$$

$$h_t = o_t \odot \tanh(c_t),$$

where $f_t \in [0,1]^D$、$i_t \in [0,1]^D$ 和 $o_t \in [0,1]^D$ are three gates to control the path of information transmission; Vector element product $\odot$; Memory unit at the last moment c_{t-1}; Candidate state obtained by nonlinear function: $\tilde{c}_t \in \mathbb{R}^D$

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c).$$

The LSTM network introduces a gating mechanism to control the path of information transmission. The three gates are input gate i_t, forget gate f_t and output gate o_t. The functions of these three doors are:

(1) The forget gate f_t controls the internal state of the last moment c_{t-1} how much information needs to be forgotten.

(2) The input gate controls how much information about the current candidate state $\tilde{c}_t$ needs to be saved.

(3) The output gate o_t controls how much information of the internal state c_t at the current moment needs to be output to the external state h_t.

The calculation of the three gates is:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i), \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f), \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o), \end{aligned}$$

The structural unit of LSTM is shown in Figure 1, and the calculation process is: 1) Use the external state h_{t-1} at the previous moment and the input x_t at the current moment to calculate three gates and the candidate state $\tilde{c}_t$; 2) Combine the forget gate f_t and the input gate i_t to update the memory unit c_t; 3) Combined with the output gate o_t, the information of the internal state is transferred to the external state h_t.

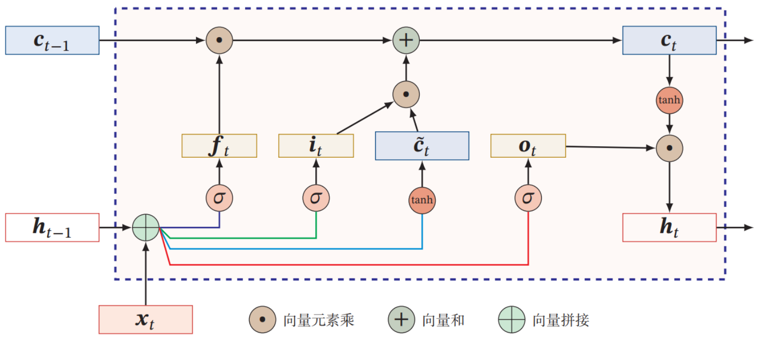

Figure 1: The structural unit of LSTM

2.1.2 BiLSTM+CRF

The model structure used in the experiment is **BiLSTM+CRF**, and the BiLSTM-CRF model mainly includes two parts: the BILSTM layer and the CRF loss layer, as shown in Figure 2. This network can efficiently use past input features via a LSTM layer and sentence level tag information via a CRF layer. A CRF layer is represented by lines which connect consecutive output layers. A CRF layer has a state transition matrix as parameters. With such a layer, we can efficiently use past and future tags to predict the current tag,which is similar to the use of past and future input features via a bidirectional LSTM network.[1]

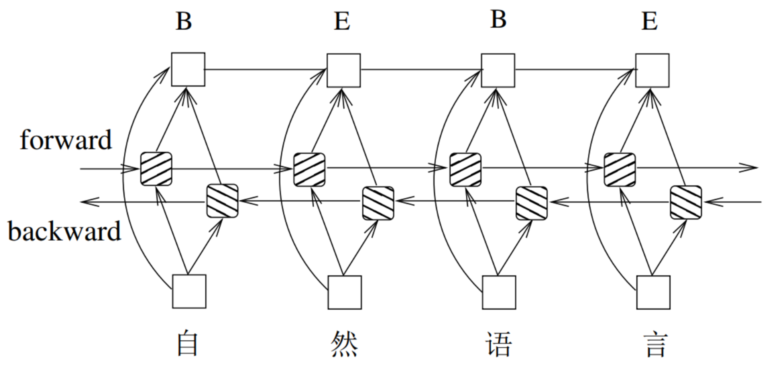

Figure 2: BiLSTM+CRF

For an input sentence, the sentence passes through the embedding layer to map each vocabulary or character into a word vector or character vector, and then passes it to the BiLSTM layer to obtain the forward and backward vectors of the sentence, and then combine the forward and backward vectors as the hidden state vector of the current word or character. The hidden state vector is used as the input of the CRF layer, and the tag sequence of the input sentence obtained by the model is obtained through the CRF layer. In the training process, the gradient descent method is used to update the parameters of the neural network to obtain the most consistent label sequence.

2.2 Model and Training procedure

The experiment transforms this task into a sequence tagging task, tagging each word in each sentence. The meaning of specific labels is shown in table 1.

Label	Meaning
B	The beginning character of Chinese word
M	The middle character of Chinese word
E	The ending character of Chinese word
S	Individual Chinese word
<START>	Start of sentence
<END>	End of sentence
<PAD>	The character to pad sentence

Table 1: The meaning of specific labels

Take the sentence as input, and record a Chinese sentence (character sequence) containing n characters as

$$x = (x_1, x_2, \cdots, x_n),$$

where x_i represents the id of the i Chinese character of the sentence in the dictionary, and then the word vector of each character can be obtained.

The first layer of the model is the embedding layer, which uses the pre-trained embededs matrix to map each Chinese character x_i in the sentence into a word vector $\mathbf{x}_i \in \mathbb{R}^d$, where d is the dimension of the word vector.

The second layer of the model is the two-way LSTM layer, which is used to extract the feature vector of the sentence, and the word vector sequence of each Chinese character of the sentence $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n)$ is used as the input of each time step of the BiLSTM network. Then concatenate the hidden state sequence $(h_{1f}, h_{2f}, \cdots, h_{nf})$ output by the forward LSTM and the hidden state of the reverse LSTM $(h_{1b}, h_{2b}, \cdots, h_{nb})$ into $h_t = [h_{tf}; h_{tb}]$ to obtain the complete hidden state sequence $(h_1, h_2, \cdots, h_n)$.

After setting the dropout, the second layer connects a linear layer to map the hidden state vector to a k-dimensional vector, where k is the number of tags in the annotation set, so as to obtain automatically extracted sentence features and get the score matrix $P = (p_1, p_2, \cdots, p_n) \in \mathbb{R}^{n \times k}$. Each dimension p_{ij} of $p_i \in \mathbb{R}^k$ can be regarded as a scoring system in which the word x_i is classified into the jth label. If P is subjected to $Softmax$, the result is an independent k class classification, without considering constraints.

If the CRF layer is not added, the one with the highest score can be selected as the label of the word. Although the correct label of each word in the sentence is obtained according to the score for optimization, it is not guaranteed that the label satisfies the constraint relationship between the labels. For example, the S label can be after the E label, but not after the B label. And the BS sequence may appear during the training process, which is obviously wrong. Therefore, it is necessary to increase the CRF layer to learn the constraints. The CRF layer can add some constraints to the final predicted label sequence to ensure that the predicted sequence is legal. In the training process of training data, these constraints can be automatically learned through the CRF layer.

The third layer of the model is the CRF layer, which is used to mark the sequence of sentences.The parameter of the CRF layer is a matrix T of $k \times k$. T_{ij} which represents the transfer score from the ith tag to the jth tag. If the tag sequence of the sentence is $y = (y_1, y_2, \cdots, y_n)$, then the model will score the tag y of the sentence x as

$$score(x, y) = \sum_{i=1}^{n} P_{i,y_i} + \sum_{i=1}^{n+1} T_{y_{i-1},y_i}.$$

The score is obtained by two parts, one part is determined by the output p_i obtained by the first two layers, and the other part is determined by the CRF transition matrix T. Then you can use Softmax to get the normalized probability:

$$P(y|x) = \frac{e^{score(x,y)}}{\sum_{y'} e^{score(x,y')}}.$$

The model is trained by minimizing the negative log-likelihood function. For a training sample (x, y^x), the loss function is

$$Loss = -log(P(y^x|x)) = -(score(x, y^x) - log(\sum_{y'} e^{score(x,y')})).$$

The model uses the ViterBi algorithm of dynamic programming to solve the optimal path during the prediction process:

$$y* = arg\,\max_{y'} score(x, y').$$

3 Critical Code

This experiment is implemented using the pytorch deep learning architecture, and the critical code is divided into three parts: data processing, BiLSTM and CRF.

3.1 Data Processing

In the experiment, each example is represented by a tuple

$$(sentence : [word_1, word_2, \cdots, word_n], tags : [tag_1, tag_2, \cdots, tag_n]),$$

where the sentence list is the Chinese charater sequence and tags saves the corresponding tag sequence. In the experiment, the GetSentenceTag function is used to obtain the tag sequence according to the result of each sentence segmentation. The implementation code is as follows:

```python
def GetSentenceTag(sentence_set, word_set):
    sentence_tag = ['S' for i in range(len(sentence_set))]
    index = 0
    for term in word_set:
        # 将每个分词拆分成字
        term_split = [one for one in term]
        word_length  = len(term_split)
        # 只有一个字则标注为 ‘S’
        if (word_length == 1):
            sentence_tag[index] = 'S'
        # 多个字则按‘BME’顺序标注
        else:
            sentence_tag[index] = 'B'
            index +=1
            word_length-=1
            while(word_length>1):
                sentence_tag[index] = 'M'
                word_length-=1
                index +=1
            sentence_tag[index] = 'E'
        index += 1
    return sentence_tag
```

In order to train multiple data at the same time, the length of the sentence needs to be processed. In the experiment, the sentence length is made the same by truncating or filling the word '<PAD>'. The implementation code is as follows:

```
def TruncAndPad(sentence_set, sentence_tag, max_len):
    # 句子长度大于设定长度则截断
    if (len(sentence_set) >= max_len):
        return sentence_set[0: max_len], sentence_tag[0:max_len]
    # 句子长度小于设定长度则在句子后增加“<PAD>”
    else:
        sentence_set.extend(["<PAD>"] * (max_len - len(sentence_set)))
        sentence_tag.extend(["<PAD>"] * (max_len - len(sentence_tag)))
    return sentence_set, sentence_tag
```

According to the above two functions to achieve data processing, LoadData function reads the data file and saves it to the tuple list. The implementation code is as follows:

```
def LoadData(file, load_from_csv=False, max_len=200):
    # 返回的数据元组列表
    data = []
    sentence_tag_list = []
    sentence_split_list = []
```

```python
    # 将训练数据保存在 csv 文件中
    if load_from_csv:
        with open('sentence_tag_list.csv', 'r', newline='',
        ↪   encoding='utf-8') as fp:
            sentence_tag_list = [i for i in csv.reader(fp)]
        with open('sentence_split_list.csv', 'r', newline='',
        ↪   encoding='utf-8') as fp:
            sentence_split_list = [i for i in csv.reader(fp)]
    else:
        with open(file, 'r', encoding='utf-8') as fp:
            sentences = fp.readlines()
        for i in range(len(sentences)):
            remove_chars = '[ \\n]+'
            # 获得字列表
            sentence_split = [one for one in re.sub(remove_chars, "",
            ↪   sentences[i])]
            sentence_split_list.append(sentence_split)
            # 获得词列表
            remove_chars = '[\\n]+'
            word_set = re.sub(remove_chars, "", sentences[i]).split()
            # 根据字列表与词列表得到该句子的标签序列
            sentence_tag = GetSentenceTag(sentence_split, word_set)
            sentence_tag_list.append(sentence_tag)
        # 保存读取的数据 方便下次训练
        with open('sentence_tag_list.csv', 'w', newline='',
        ↪   encoding='utf-8') as fp:
            write = csv.writer(fp)
            write.writerows(sentence_tag_list)
        with open('sentence_split_list.csv', 'w', newline='',
        ↪   encoding='utf-8') as fp:
            write = csv.writer(fp)
            write.writerows(sentence_split_list)
    # 按元组形式保存训练数据
    for i in range(len(sentence_split_list)):
        data.append(TruncAndPad(sentence_split_list[i],
        ↪   sentence_tag_list[i], max_len))
    return data
```

In order to train the samples in batches, torch.utils.data is used to process the data. In the file 'MyDataSet.py', implementation defines the data class used. The implementation code is as follows:

```python
class MyDataSet(Dataset):
    def __init__(self, data_from, word_to_ix, tag_to_ix):
        self.sentence, self.tag = Word2TensorBatch(data_from, word_to_ix,
        ↪ tag_to_ix)

    def __getitem__(self, item):
        return self.sentence[item], self.tag[item]

    def __len__(self):
        return len(self.sentence)
```

3.2 BiLSTM

In the file 'BiLSTM_CRF.py', implementation uses the lstm model in pytorch to build the BiLSTM model we need, and initializes the transition matrix that CRF needs to use in the class, and adds it to the training parameters. The class initialization code is as follows:

```python
    def __init__(self, input_size, hidden_size, num_layers, drop_rate,
    ↪ vocab_size, tag_to_ix, word_weight=None):
        super(LSTM, self).__init__()
        self.lstm = nn.LSTM(
            input_size=input_size,  # 输出向量大小 也为字向量大小
            hidden_size=hidden_size // 2,  # 隐状态输出向量大小 双向则为 1/2
            num_layers=num_layers,  # 层数
            bidirectional=True,  # 双向
            batch_first=True)  # 是否 batch
        # self.word_embeds = nn.Embedding(vocab_size, input_size)
        ↪ # 采用随机初始化的词向量 并做训练
        self.word_embeds = nn.Embedding.from_pretrained(word_weight)  # 采
        ↪ 用训练的词向量
        self.tag_to_ix = tag_to_ix
        self.tag_size = len(tag_to_ix)
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.hidden2tag = nn.Linear(hidden_size, self.tag_size)  # 线性层从
        ↪  隐状态向量到标签得分向量
        self.transitions = nn.Parameter(torch.randn(self.tag_size,
        ↪  self.tag_size))  # CRF 的转移矩阵表示从列序号对应标签转换到行序号
        ↪  对应标签
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000  # 任意标签
        ↪  不能转移到 start 标签
        self.transitions.data[:, tag_to_ix[END_TAG]] = -10000  # end 标签不
        ↪  能转移到任意标签
        self.hidden = self.HiddenInit()  # 隐藏层初始化
```

3.3 CRF

According to the loss function $-log(P(y^x|x))$, we need to calculate $log(\sum_{y'} e^{score(x,y')})$ and the score of each possible path y' of x. Using forward algorithm, we can calculate

$$log(\sum e^{log(\sum e^{score(x_i,y_i)})+T_{i,i+1}+P_{i+1,y_{i+1}}}) = log(\sum\sum e^{score(x_i,y_i)+T_{i,i+1}+P_{i+1,y_{i+1}}})$$

by the formula

$$log(\sum e^{log(\sum e^x)+y}) = log(\sum\sum e^{x+y}).$$

Then for the $logsumexp$ of the path score of the word x_{i+1}, we can get it by calculating the $logsumexp$ of the path score of the word x_i. The implementation code is as follows:

```
    def ForwardAlg(self, feats):
        if use_gpu:
            init_alphas = torch.full([feats.shape[0], self.tag_size],
            ↪  -10000.).cuda()
        else:
            init_alphas = torch.full([feats.shape[0], self.tag_size],
            ↪  -10000.)
        # 开始标签的转换得分为 0
        init_alphas[:, self.tag_to_ix[START_TAG]] = 0.
        # 输入的每个句子都进行前向算法
        forward_var_list = []
        forward_var_list.append(init_alphas)
        # 每个句子从句首开始迭代
        for feat_index in range(feats.shape[1]):
            # 迭代到某一词的 logsumexp
            tag_score_now = torch.stack([forward_var_list[feat_index]] *
            ↪  feats.shape[2]).transpose(0, 1)
            feats_batch = torch.unsqueeze(feats[:, feat_index, :],
            ↪  1).transpose(1, 2)
            # 新词的所有转移路径
            next_tag_score = tag_score_now + feats_batch +
            ↪  torch.unsqueeze(self.transitions, 0)
            forward_var_list.append(torch.logsumexp(next_tag_score, dim=2))
        # 加上 end 标签的得分
        terminal_var = forward_var_list[-1] +
        ↪  self.transitions[self.tag_to_ix[END_TAG]].repeat([feats.shape[0],
        ↪  1])
        # 每个句子进行 logsumexp 得到最终结果
        alpha = torch.logsumexp(terminal_var, dim=1)
        return alpha
```

For the score of a given sequence of tags, we can simply use the above formula for iterative calculation. The implementation code is as follows:

```
    # 给定序列 tags 的得分
    def Score(self, feats, tags):
        if use_gpu:
            score = torch.zeros(tags.shape[0]).cuda()
        else:
            score = torch.zeros(tags.shape[0])
        if use_gpu:
            tags = torch.cat([torch.full([tags.shape[0], 1],
            ↪  self.tag_to_ix[START_TAG]).long().cuda(), tags], dim=1)
        else:
            tags = torch.cat([torch.full([tags.shape[0], 1],
            ↪  self.tag_to_ix[START_TAG]).long(), tags], dim=1)
        for i in range(feats.shape[1]):
            # 第 i 个词得到的 feats 二维张量
            feat = feats[:, i, :]
            score = score + \
                    self.transitions[tags[:, i + 1], tags[:, i]] + feat[
                        range(feat.shape[0]), tags[:, i + 1]]
        score = score + self.transitions[self.tag_to_ix[END_TAG], tags[:,
        ↪   -1]]
        return score
```

According to the function implemented above, the loss function required for training can be obtained. The implementation code is as follows:

```
    def LossFuction(self, sentences, tags):
        feats = self.GetFeatsBatch(sentences)
        forward_score = self.ForwardAlg(feats)
        gold_score = self.Score(feats, tags)
        # 所有输出句子的误差和作为结果
        return torch.sum(forward_score - gold_score)
```

The above training model can be trained using pytorch's optimizer optim.Adam. The detailed training code is in the source file 'main.py'.

4 Experimental Results and Analysis

4.1 Hyperparameter Adjustment

The model parameters used in the experiment are shown in Table 2.

Table 2: Model parameters

Parameter	**Size**
Word vector dimension (input vector dimension)	100&300
BiLSTM layer output vector dimension	200
BiLSTM layer number	2
Drop Out Rate	0.6
Sentence length	30
Weight decay rate	1e-4

I adjust the training learning rate and set other hyperparameters to the same value. The change of Loss with the increase of training steps is shown in Figure 3. The curve with learning rate of 0.0001 and the curve with learning rate of 0.00005 in the figure have a better effect. In order to improve the convergence speed of the model, I initially chose 0.0001 as the learning rate of the model.

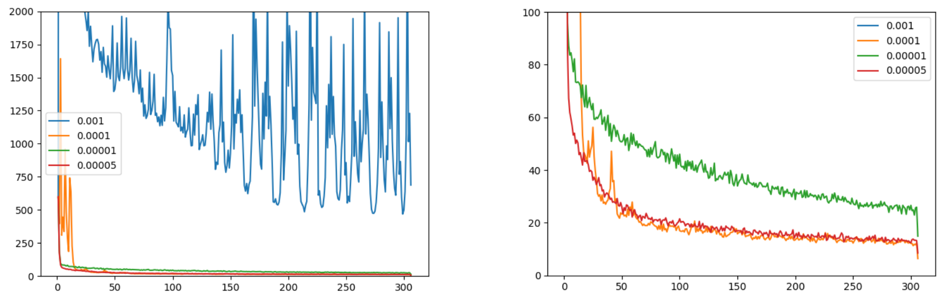

Figure 3: The changes of loss with different learning rate

I adjust the batchsize and set other hyperparameters to the same value. The change of Loss with the increase of training steps is shown in Figure 4. The curve with batchsize of 256 in the figure has a better effect. In order to improve the convergence speed of the model, I initially chose 256 as the batchsize of the model.

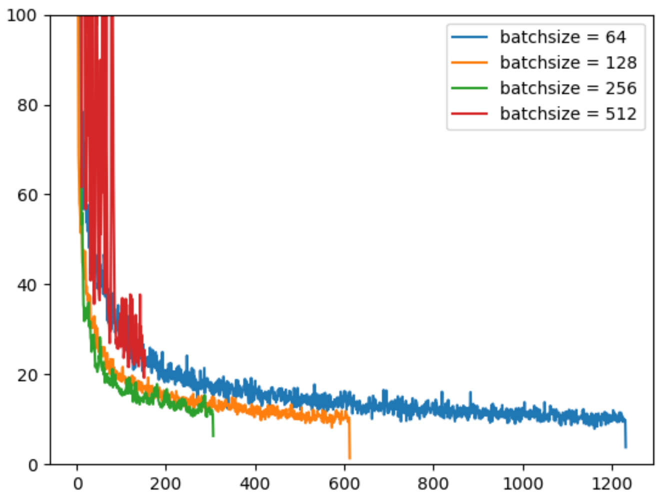

Figure 4: The changes of loss with different batch size

In th experiment, data in file ”msr_training.utf8” is divided randomly into training set and validation set. The proportion of the training set is 0.9. With the above hyperparameters, the change of loss with the increase of training steps is shown in Figure 5. At the same time, the score of F1 in validation set is shown in Figure 6.

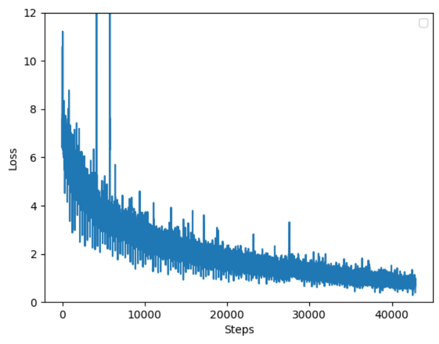

Figure 5: The changes of loss

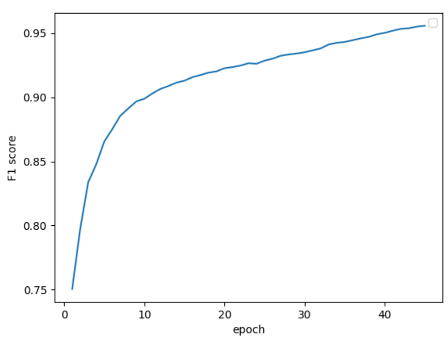

Figure 6: The score of F1 in validation set

After 150 epochs training, the loss of model drops below 1 and the score of F1 in validation set stabilizes at 0.95.

4.2 Word Vector Adjustment

There are two ways to obtain the word vector of Chinese characters in this experiment: 1) Use the Word2Vec module to train the sentences in the training set and test set to obtain the word vector; 2) Use the word vector that has been trained.

In the experiment, I used the Word2Vec tool to train sentences in the training set and test set to obtain word vectors. Since there are small corpus for training and many sentences in training set are rubbish, the word vectors cannot fully represent the meaning of words. However, because our test set is also bad, such word vectors can achieve better results. The F1 score of the final model on the test set using the above method is shown in Table 3.

Table 3: The F1 score of the final model using Word2Vec

Word vector dimension = 100	0.936
Word vector dimension = 300	0.926

In order to further improve the F1 score, I choose the open source trained word vector on github. The selected corpora are Baidu Encyclopedia and Mixed-large. In the experiment, I embedded the downloaded 300-dimensional word vector into the embedding layer of the model. The F1 score of the final model on the test set using the above method is shown in Table 4.

Table 4: The F1 score of the final model using trained word vector

Baidu Encyclopedia	0.922
Mixed-large	0.931

Due to the model structure, the final improvement effect of the model is limited. The F1 score of the finally trained model on the test set is above 0.9. This fully shows that the BiLSTM+CRF model is effective in Chinese word segmentation. The final word segmentation results of the models trained in different ways in the test set can be seen in the saved txt file.

References

[1] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015.