NLP Mid-term Project: Chinese segmentaion model

18340146 Computer Science and Technology 宋渝杰

Contents of research:

Use BiLSTM+CRF word segmentation model to train and test Chinese word segmentation on the SIGHAN Microsoft Research data set

Plans of research/Design of experimental procedures:

I personally think that "research plan" and "experimental procedure design" are equivalent to **"experimental process".**

Since I learned neural networks in the "Artificial Intelligence" course, the main process of my research in this course project is relatively clear:

1. Establish BiLSTM+CRF word segmentation model (that is, Recurrent Neural Network model)
2. Use the training set `msr_training.uft8` to train the neural network (the details will be explained below)
3. Perform a verification every time iterative training (use the validation set `msr_test.utf8` and `msr_test_gold.utf8`), and calculate its F1 indicators
4. When the training iteration reaches a certain number of times, the training ends and the final verification test is performed, and the word segmentation results and F1 indicators are output

Statement of Principle:

The following will mainly explain the construction and operation principle of the BiLSTM+CRF word segmentation model, then the training and testing process of the model, and finally the generation of word segmentation results and the calculation of the F1 indicators score for this result.

Neural Networks/Neurons:

Neural network is a distributed parallel information processing algorithm mathematical model, divided into input layer, hidden layer, and output layer.The input layer has multiple neurons, which have the same dimension as the input data feature, and receive data input and output directly; the hidden layer has multiple neurons, which save some features learned by the neural network; the output layer also has multiple neurons, and same dimension as the output data label. Each neuron receives input signals from all neurons in the previous layer. These signals are transmitted through weighted connections. The total input value received by the neuron plus the offset is processed by an "activation function". Finally, the output of the neuron is generated, and the output of the output layer is the output of the neural network.

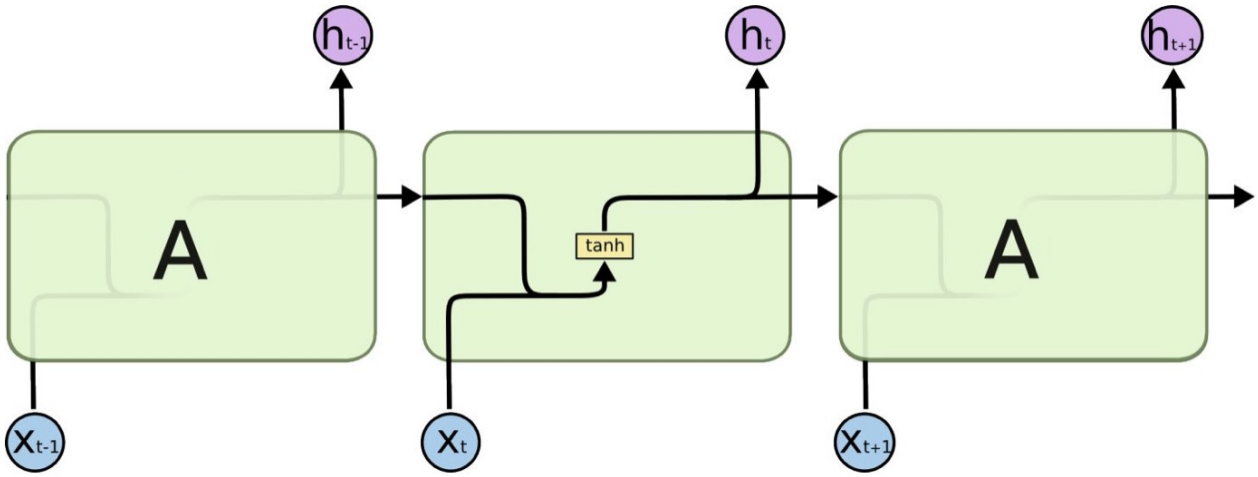
Recurrent Neural Network (RNN) :

RNN is a special kind of neural network (essentially a kind of mathematical model). Like ordinary neural networks, it is divided into input layer, hidden layer, and output layer. But the hidden layer is special, it will accept the last output of the neuron h_{t-1} as the second input and the original input x_t, which together affect the current output(h_t) of the neuron.

Standard RNN:

A standard RNN model consists of an input neuron, a hidden neuron, and an output neuron. However, due to its cyclic characteristics, it can be expanded into multiple input, hidden, and output neurons, and the number of expansions is the same as the dimension of the input data (in this experiment, it can be understood as the length of a sentence/the number of words).

The processing of hidden neurons is similar to the fully connected layer: the input x and the input h_{t-1} are used to form a fully connected layer $\mathrm{W}_{xh}\mathrm{x}_t$ and $\mathrm{W}_{hh}\mathrm{h}_{t-1}$, and then calculate the sum of the above two results with the two offsets b_x and b_h respectively, finally the output h_t is obtained through the activation function (usually tanh).



$$\mathrm{h}_t = \tanh(\mathrm{W}_{xh}\mathrm{x}_t + \mathrm{b}_x + \mathrm{W}_{hh}\mathrm{h}_{t-1} + \mathrm{b}_h)$$

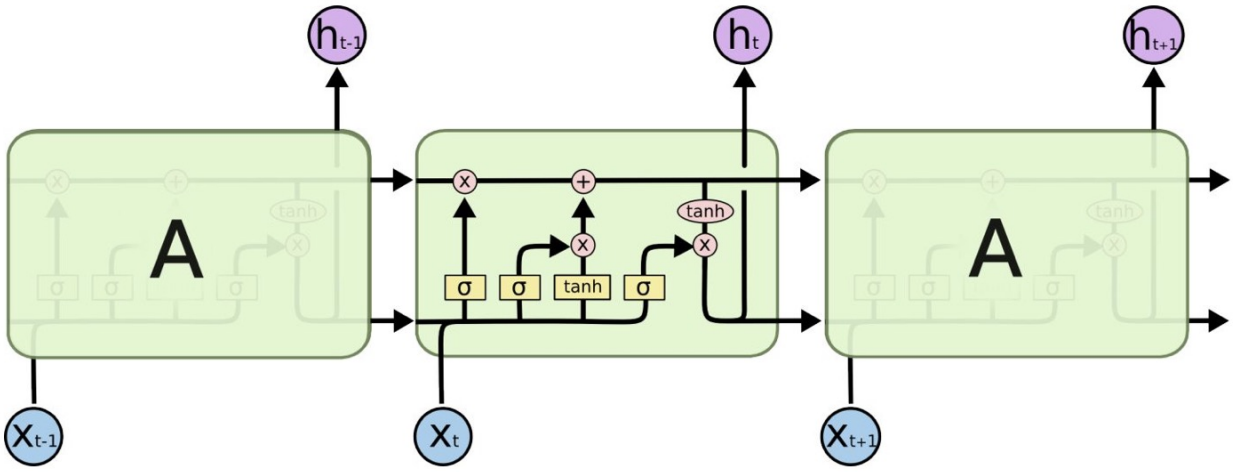
The processing of the output neuron is also similar to the fully connected layer: the input h_t is made into a fully connected layer to form $\mathrm{W}_{hz}\mathrm{h}_t$, and then sum it with the offset b_z, finally use the activation function (generally not to be used) to get the output z_t, which is the non-normalized probability of each label of the word.

$$\mathrm{z}_t = \mathrm{W}_{hz}\mathrm{h}_t + \mathrm{b}_z$$

Long Short-Term Memory Network (LSTM):

In the standard RNN, this repeated structural module has only a very simple tanh layer, and this simple level of RNN can only be inferred based on the previous relatively recent information, but if you need to use relatively distant information, the standard RNN is very Difficult to achieve results.

Long Short-Term Memory Network (LSTM) is a special RNN network to solve the long-term dependency problem. Compared with standard RNN, its hidden layer has four network layers:



Cell state: The standard RNN not only transmits h_{t-1} cyclically, but also transmits a "cell state" C_{t-1} to save the historical information of the sequence. Cell state usually changes very slowly, so it can also save long-distance information.

Forgotten Gate: The first layer of LSTM decides to discard certain information about the cell state. It outputs a vector f_t between 0~1 by looking at the information of h_{t-1} and x_{t-1}. The value of 0~1 in this vector indicates which information in the cell state C_{t-1} needs to be retained or discarded. 0 means not to retain, and 1 means to retain all.

After the vector is obtained, it is **multiplied** by the cell state, which is to realize the forgetting operation of information.

$$
\begin{aligned}
f_t &= \operatorname{sigmoid}(\mathrm{W}_f[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_f) \\
C'_{t-1} &= C_{t-1} \cdot f_t
\end{aligned}
$$

Input gate: The second and third layers of LSTM decide to add some new information to the cell state. It outputs a vector i_t between 0 and 1 and new candidate cell information $\tilde{C}_t$ by looking at the information of h_{t-1} and x_{t-1} , and then the value of 0~1 in the vector indicates which information in the new candidate cell information $\tilde{C}_t$ needs to be added and how much it needs to be added.

After getting the amount of added information, it is **summed** with the cell state, which is to realize the input operation of information:

$$
\begin{aligned}
i_t &= \operatorname{sigmoid}(\mathrm{W}_i[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_i) \\
\tilde{C}_t &= \tanh(\mathrm{W}_c[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_c) \\
C_t &= C'_{t-1} + i_t * \tilde{C}_t
\end{aligned}
$$

After the forget gate and input gate work, the cell state has also been **updated**.

Output gate: The fourth layer of LSTM, output h_t. It outputs a vector O_t between 0 and 1 by looking at the information of h_{t-1} and x_{t-1}, then the 0~1 value in the vector indicates which information in the new cell state C_t needs to be output and how much needs to be output.

$$
\begin{aligned}
O_t &= \operatorname{sigmoid}(\mathrm{W}_o[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_o) \\
\mathrm{h}_t &= O_t * \tanh(C_t)
\end{aligned}
$$

So in general, the steps of a neuron in LSTM can be summarized into four steps: **forget, input, update, and output.**

Two-way LSTM:

For this experiment, the judgment of the word segmentation label is not only related to the previous text, but also related to the following text. Therefore, passing the word sequence backward through the LSTM will have a better recognition effect. The two-way LSTM passes the sentence forward and backward through the LSTM in order to capture the two-way feature information.

Conditional Random Field (CRF):

The CRF layer can be understood as a transition probability matrix A: $A_{i,j}$ represents the transition probability from tag_i to tag_j.

The CRF layer can add some constraints to the last predicted label (set the transition probability between two labels) to ensure that the predicted label is legal. In the process of training data, these constraints can be automatically learned through the CRF layer, or can be obtained through artificial settings.

The working process of the CRF layer during **training**:

1. For a certain output label sequence y corresponding to the input sequence X, define the score as:

$$s(X, y) = \sum_{i=0}^{n} A_{y_{i+1}, y_i} + \sum_{i=1}^{n} P_{x_i, y_i}$$

   Among them, P_{x_i, y_i} represents the non-normalized probability of word x_i mapped to label y_i.

2. Using the Softmax function, get the probability value of each tag sequence (including the impossible), and output it:

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

The working process of the CRF layer during **verification** (Viterbi decoding process):

1. Calculate the start symbol first -> Probability of various tags for the first word (ie score)
2. Calculate a label from the previous word -> The score of a tag for this word:
    1. The score is obtained by summing "a tag of the previous word", "the transition probability", and "the emission score" of a tag of this word
    2. Calculate all the paths that "all tags of the previous word -> a certain label of this word", select the one with the highest score and record it as the optimal path of "start symbol -> a certain label of this word" (that is, the idea of dynamic programming pruning)
3. After calculating all the tag of the last word -> ending symbol, you can select the path with the highest score, that is, the start symbol -> ending symbol with the highest score, and then return that path

Training:

The training details are as follows:

1. Read a batch of sentences from the training set, convert them into word vectors (features) for each word, and generate corresponding word segmentation tag sequences (tags)
2. Pass the word vector list of the sentence forward through the cyclic neural network BiLSTM+Linear layer to obtain the emission probability (the non-normalized probability of each word segmentation label/score)

3. After obtaining the transmission probability, pass it forward through the CRF layer to obtain the probability of the output tag sequence y corresponding to the input sequence X : $p(y|X)$
4. In the training process, maximize the likelihood function $p(y|X)$. In order to speed up the calculation, the log likelihood function is used here:

$$\log(p(y|X)) = \log(\frac{e^{s(X,y)}}{\sum_{y' \in Y_X} e^{s(X,y')}}) = S(X,y) - \log(\sum_{y' \in Y_X} e^{S(X,y')})$$

5. Define the loss function as a negative log likelihood function $-\log(p(y|X))$, and use **stochastic gradient descent** method to update the parameters of the network:

$$\text{loss} = \log(\sum_{y' \in Y_X} e^{S(X,y')}) - S(X,y)$$

Verification:

There are some differences between verification and training. The specific steps are as follows:

1. Read a line of sentence from the verification set `msr_test.utf8`, convert it into a word vector (feature) for each word, and read the corresponding word segmentation tag sequence (tag) from `msr_test_gold.utf8`.
2. Pass the word vector list of the sentence forward through the cyclic neural network BiLSTM+Linear layer to obtain the emission probability (the non-normalized probability of each word segmentation label/score).
3. After obtaining the transmission probability, pass it through the CRF layer and use the Viterbi decoding process to obtain the tag sequence with the highest score.
4. Compare the optimal sequence with the real sequence, calculate its F1 indicators and output it.

Calculate F1 indicators:

The calculation formula of F1 indicator is as follows:

$$F1 = \frac{2 * p * r}{p + r}$$

p = (number of word pairs) / (total number of words predicted for word segmentation results)

r = (number of word pairs) / (total number of correct word segmentation results)

For the calculation of "total number of times", we can accumulate the sum of 'S' and 'B' (or 'E') labels.

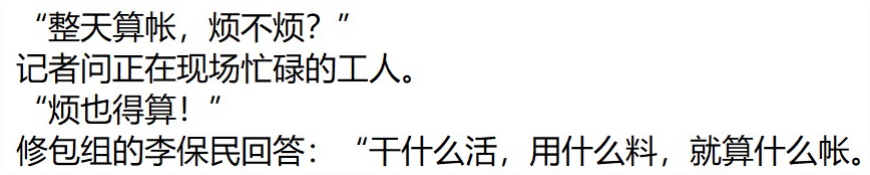
For the judgment of "pairing", my approach is: if the label sequence output by the verification set matches the corresponding position of the real sequence (at the same time as 'S' or the corresponding segment is at the same time as 'BM...ME'), it is judged as pairing correct, and the rest are judged to be wrong.

Noise of validation set:

During the experiment, the following problems were found in the verification set:

Some sentences in the msr_test.utf8 file after word segmentation, the quotation marks ran to the end of the previous line at the corresponding position in the msr_test_gold.utf8 file.

For example, in `msr_test.utf8`:



But in `msr_test_gold.utf8`:

```
去年 以来 ， 这个 工段 各个 班组 的 日 核算 从未 间断 过 ， 经济效益 与日俱增 。 “
整天 算 帐 ， 烦 不 烦 ？ ”
记者 问 正在 现场 忙碌 的 工人 。 “
烦 也 得 算 ！ ”
修 包 组 的 李保民 回答 ： “ 干什么 活 ， 用 什么 料 ， 就 算 什么 帐 。
```

Considering that these noises cause the length of the corresponding lines to be different, so in the calculation of the F1 indicator of the code, the problem of the length of the corresponding lines needs to be considered. My approach is to take the minimum length $\min(\text{len}(l_1), \text{len}(l_2))$.

Explanation of core code:

The explanation of the code in this article is mainly in the form of **comment**, and basically all key codes will be commented.

Import of training set, validation set data, and validation set result files:

```python
def read(root): # training set data + label
    f = codecs.open(root, 'r', encoding = 'utf-8')
    data = []
    for s in f.readlines():
        s = s.strip('\n').split()
        l, l1 = [], []
        for x in s: # 0-3 instead SBME
            if len(x) == 1: l1.append(0) # 'S'
            else:
                l1.append(1) # 'B'
                for i in range(len(x)-2): l1.append(2) # "M"
                l1.append(3) # 'E'
        l.append(list(''.join(s))) # data
        l.append(l1) # label
        data.append(l)
    return data

def test_data(root): # validation set data
    f = codecs.open(root, 'r', encoding = 'utf-8')
    data = []
    for s in f.readlines():
        data.append(list(s.strip('\n').strip('\r'))) # data
    return data

def test_tags(root): # validation set label
    f = codecs.open(root, 'r', encoding = 'utf-8')
    data = []
    for s in f.readlines():
        s, l = s.strip('\n').split(), []
        for x in s:
            if len(x) == 1: l.append(0) # 'S'
            else:
                l.append(1) # 'B'
                for i in range(len(x)-2): l.append(2) # 'M'
```

```
                l.append(3) # 'E'
        data.append(l) # label
    return data

train = read('msr_training.utf8') # training set
test1 = test_data('msr_test.utf8') # validation set data
test2 = test_tags('msr_test_gold.utf8') # validation set label
```

Establishment of BiLSTM-CRF model:

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim # word vector dimension
        self.hidden_dim = hidden_dim # hidden layer dimensions
        self.vocab_size = vocab_size # number of word types
        self.tag_to_ix = tag_to_ix # tag dictionary
        self.tagset_size = len(tag_to_ix) # number of label types
        self.word_embeds = nn.Embedding(vocab_size, embedding_dim) # word embedding
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2, num_layers = 1, bidirectional = True, batch_first = True) # BiLSTM layer
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size) # the fully connected layer outputs the predicted score of each label
        self.transitions = nn.Parameter(torch.randn(self.tagset_size, self.tagset_size)) # transition matrix (random initialization)
        # these two statements enforce such a constraint, we will not transfer it to the start mark, nor will it be transferred to the stop mark
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000
        self.hidden = self.init_hidden()

    def init_hidden(self): # random initialization of hidden layers
        return (torch.randn(2, 1, self.hidden_dim // 2).to(device),
                torch.randn(2, 1, self.hidden_dim // 2).to(device))
```

Forward propagation function of BiLSTM:

```
    def forward_LSTM_parallel(self, sentence): # BiLSTM forward propagation (training)
        self.hidden = self.init_hidden() # initialize the hidden layer when the first word is passed in
        embeds = self.word_embeds(sentence) # word embedding
        lstm_out, self.hidden = self.lstm(embeds) # through the BiLSTM layer
        lstm_feats = self.hidden2tag(lstm_out) # output the launch score through the fully connected layer
        return lstm_feats

        def forward_LSTM(self, sentence): # BiLSTM BiLSTM forward propagation(verification)
        self.hidden = self.init_hidden() # initialize the hidden layer when the first word is passed in
        embeds = self.word_embeds(sentence).unsqueeze(dim=0)  # word embedding
        lstm_out, self.hidden = self.lstm(embeds)  # through the BiLSTM layer
        lstm_feats = self.hidden2tag(lstm_out.squeeze()) # output the launch score through the fully connected layer
        if len(sentence) == 1: lstm_feats = lstm_feats.unsqueeze(0) # when the sentence length is 1, add one dimension
```

```
        return lstm_feats
```

The part that calculates the error during CRF training:

```
    def forward_CRF_parallel(self, feats): # the logsumexp part of the loss function (training)
        init_alphas = torch.full([feats.shape[0], self.tagset_size], -10000.)
        init_alphas[:, self.tag_to_ix[START_TAG]] = 0. # start symbol

        forward_var_list = []
        forward_var_list.append(init_alphas)
        for feat_index in range(feats.shape[1]): # sentence length
            gamar_r_l = torch.stack([forward_var_list[feat_index]] * feats.shape[2]).transpose(0, 1) # previous
            t_r1_k = torch.unsqueeze(feats[:, feat_index, :], 1).transpose(1, 2) # emission score
            aa = gamar_r_l.to(device) + t_r1_k.to(device) + torch.unsqueeze(self.transitions, 0) # previous+emission score+transition probability
            forward_var_list.append(torch.logsumexp(aa, dim = 2)) # first calculate the logsumexp of the word w_i
        terminal_var = forward_var_list[-1] + self.transitions[self.tag_to_ix[STOP_TAG]].repeat([feats.shape[0], 1]) # finally, only the forward var of the last word is added to the probability of transferring stop_tag
        alpha = torch.logsumexp(terminal_var, dim = 1)
        return alpha

    def score_parallel(self, feats, tags): # The S(X,y) part of the loss function (training)
        score = torch.zeros(tags.shape[0]).to(device)
        tags = torch.cat([torch.full([tags.shape[0],1],self.tag_to_ix[START_TAG]).long().to(device), tags], dim = 1)
        for i in range(feats.shape[1]):
            feat = feats[:,i,:]
            score = score + self.transitions[tags[:,i + 1], tags[:,i]] + feat[range(feat.shape[0]),tags[:,i + 1]] # transition probability + non-normalized emission probability
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[:,-1]] # add the end label
        return score

    def neg_log_likelihood_parallel(self, sentences, tags): # train and return the error (training)
        feats = self.forward_LSTM_parallel(sentences) # the output after the LSTM+Linear matrix is used as the input of CRF
        forward_score = self.forward_CRF_parallel(feats)
        gold_score = self.score_parallel(feats, tags)
        return torch.sum(forward_score - gold_score) # loss function: back propagation according to this difference
```

Output the optimal tag sequence during CRF verification (Viterbi decoding):

```
    def viterbi_decode(self, feats): # viterbi decoding, output path value (verification)
        backpointers = []
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0 # start tag

```

```
        forward_var_list = []
        forward_var_list.append(init_vvars)

        for feat_index in range(feats.shape[0]):
            gamar_r_l = torch.stack([forward_var_list[feat_index]] * feats.shape[1])
            gamar_r_l = torch.squeeze(gamar_r_l)
            next_tag_var = gamar_r_l.to(device) + self.transitions # previous step score + transition probability
            viterbivars_t, bptrs_t = torch.max(next_tag_var, dim = 1) # returns the maximum value and its subscript

            t_r1_k = torch.unsqueeze(feats[feat_index], 0) # unnormalized launch probability/launch score
            forward_var_new = torch.unsqueeze(viterbivars_t, 0) + t_r1_k # add launch score

            forward_var_list.append(forward_var_new)
            backpointers.append(bptrs_t.tolist()) # save part of the shortest path

        terminal_var = forward_var_list[-1] + self.transitions[self.tag_to_ix[STOP_TAG]] # add the probability of transition to the end symbol
        best_tag_id = torch.argmax(terminal_var).tolist()

        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers): # go from back to front and find an optimal path
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        start = best_path.pop() # discard start symbol
        best_path.reverse() # backward -> forward
        return best_path
```

Calculate F1 indicators:

```
def cou(l): # return word count
    return l.count(0)+l.count(1)

def correct(l1, l2): # return the correct number of words
    num = 0
    for i in range(min(len(l1),len(l2))):
        if l1[i] == l2[i] and l1[i] == 0: num += 1 # 'S'
        elif l1[i] == l2[i] and l1[i] == 1: # 'B'
            j = i
            while 1:
                j += 1
                if l1[j] == l2[j] and l1[j] == 3: # 'E'
                    num += 1
                    break
                elif l1[j] == l2[j] and l1[j] == 2: continue # 'M'
                else: break
    return num

# call main:
p, r = correct(l, test2[i])/cou(l), correct(l, test2[i])/cou(test2[i]) # two indicators
if not p+r == 0: ans += (2*p*r)/(p+r) # F1 (prevent division by 0)
```

Model instantiation and gradient descent optimizer:

```
model = BiLSTM_CRF(len(word_to_ix), tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM).to(device)
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9, weight_decay = 5e-4)
# optimization method: SGDM
```

Training:

```
for epoch in progressive:
    model.train() # training mode
    for i in range(len(train)//BATCH):
        sentence_in_pad, targets_pad = pre_batch(train[i*BATCH:(i+1)*BATCH], word_to_ix, tag_to_ix) # prepare a batch of data
        loss = model.neg_log_likelihood_parallel(sentence_in_pad.to(device), targets_pad.to(device)) # Backpropagation does not necessarily need to use forward(), and there is no need to define loss = nn.MSError(), etc., directly score1-score2 can backpropagate
        # BPNN General steps
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Verification:

```
    model.eval() # verification mode
    ans = 0.
    for i in range(len(test1)):
        sentence = pre(test1[i], word_to_ix).to(device)
        l = model(sentence)
        p, r = correct(l, test2[i])/cou(l), correct(l, test2[i])/cou(test2[i]) # two indicators
        if not p+r == 0: ans += (2*p*r)/(p+r) # F1 (prevent division by 0)
    ans = round(ans/len(test1), 3) # calculate the average
    print(ans) # output F1 indicator
```

Generate word segmentation results:

```
# generate word segmentation results
model = torch.load('model.pkl').to(device) # GPU

data = []
for i in range(len(test1)):
    sentence = pre(test1[i], word_to_ix).to(device)
    l, s = model(sentence), '' # get the word segmentation symbol sequence through the model
    for j in range(len(l)):
        if l[j] == 0 or l[j] == 3: s = s + test1[i][j] + ' ' # add a space after S'and'E'
        else: s = s + test1[i][j]
    data.append(s)

def text2(filename, data): # save word segmentation results
    file = open(filename,'w')
    for i in data: file.write(i+'\n')
    file.close()

text2('result.txt', data)
```

Calculate the F1 indicators of the word segmentation result:

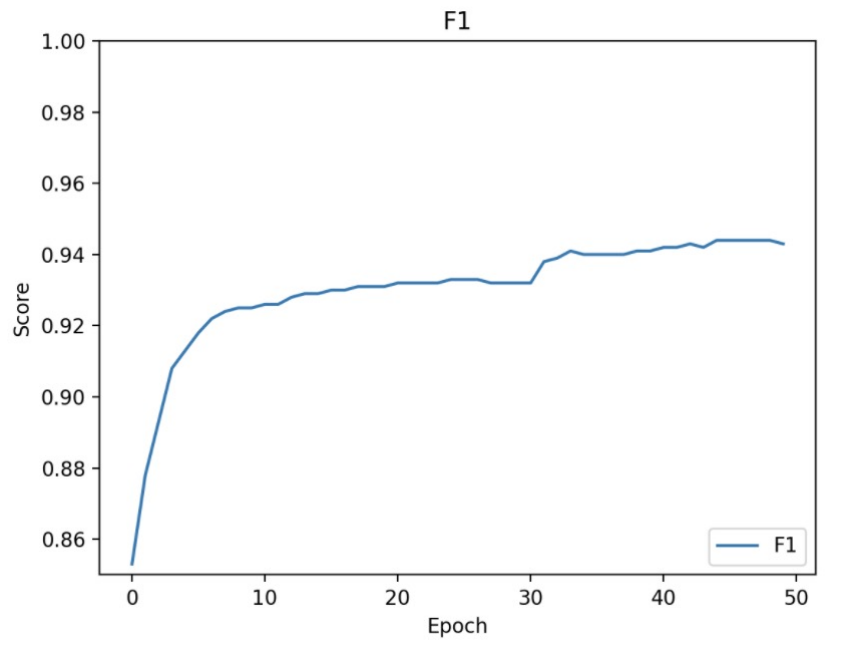
```
# calculate F1 indicators
test1 = test_tags('result.txt')
test2 = test_tags('msr_test_gold.utf8')
ans = 0.
for i in range(len(test1)):
    p, r = correct(test1[i], test2[i])/cou(test1[i]), correct(test1[i], test2[i])/cou(test2[i]) # two indicators
    if not p+r == 0: ans += (2*p*r)/(p+r) # F1 (prevent division by 0)
ans = round(ans/len(test1), 3)
print('F1 Score: ', ans)
```

Results of experimental:

Run the experimental source code (integrate the three parts of the training model, generate the word segmentation result, and calculate the F1 indicator of the generated word segmentation result into the code train.py, and the code for separately calculating the F1 indicator generating the word segmentation result is score.py). Train 50 iterations (about 2 hours) under GPU (Testa V100), configure the super parameter combination and the final F1 indicator score as follows:

Super parameter	Value
EMBEDDING_DIM	256
HIDDEN_DIM	256
BATCH	256
optimizer	SGDM
Learning rate	0.01（before 30 times）/0.001（after 20 times）
iterations	50

The verification indicator score during the training process is as follows:



The final F1 indicator is **0.944**.

```
[25]: run score.py
      F1 Score:  0.944
```

Optimization part:

After implementing the basic part, I adopted the following optimization plan:

Innovation in model structure and method:

Batch processing:

The standard recurrent neural network reads in one sequence at a time and outputs the result of one sequence, which is very slow for training with a large amount of data (the training set is iterated once more than an hour). Therefore, I adopted the method of data batch processing. Each time a batch of data is passed in, the average value of the error of a batch of data is output, and the parameters are updated by back propagation with this error.

It should be noted that the recurrent neural network requires the data to have the same dimension and length (that is, each sentence has the same length) during data batch processing. Therefore, in each batch of data, except for the longest sentence, I added invalid tags to the sentences, making these sentences the same length as the longest sentence:

```
PAD_TAG = "<PAD>"
BATCH = 256
def pre_batch(data ,word_to_ix, tag_to_ix): # batch data
    seqs, tags = [i[0] for i in data], [i[1] for i in data]
    max_len = max([len(seq) for seq in seqs]) # calculate the longest value in a batch of data
    seqs_pad, tags_pad = [], []
    for seq,tag in zip(seqs, tags):
        seq_pad = seq + ['<PAD>'] * (max_len-len(seq)) # tag other sentences invalid
        tag_pad = tag + ['<PAD>'] * (max_len-len(tag))
        seqs_pad.append(seq_pad)
        tags_pad.append(tag_pad)
    idxs_pad = torch.tensor([[word_to_ix[w] for w in seq] for seq in seqs_pad], dtype = torch.long)
    tags_pad = torch.tensor([[tag_to_ix[t] for t in tag] for tag in tags_pad], dtype = torch.long)
    return idxs_pad, tags_pad
```

Therefore, in each training, a batch of data is passed in for training:

```
   for i in range(len(train)//BATCH):
        sentence_in_pad, targets_pad = prepare_sequence_batch(train[i*BATCH:(i+1)*BATCH], word_to_ix, tag_to_ix) # prepare a batch of data
        loss = model.neg_log_likelihood_parallel(sentence_in_pad.to(device), targets_pad.to(device)) # Backpropagation does not necessarily need to use forward(), and there is no need to define loss = nn.MSError(), etc., directly score1-score2 can backpropagate
        # BPNN General steps
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

After adopting batch optimization, each training iteration time is reduced to about 2 minutes, which has a very good acceleration effect.

Decrease the learning rate in stages:

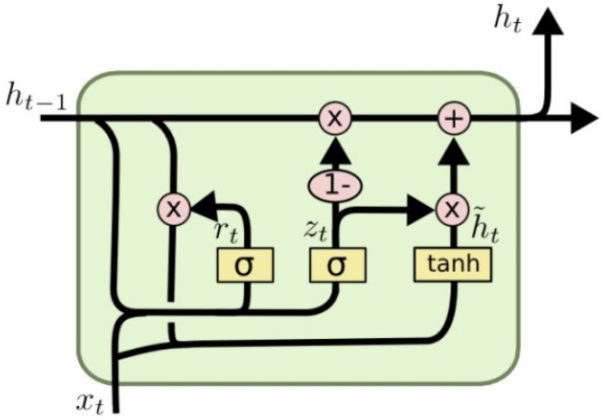
When a certain learning rate becomes stable in the verification indicator score, the learning rate is divided by 10 to continue learning. During the test, it was found that there was a certain improvement in indicator scores, so this experiment was optimized using this scheme, and simplified to: the learning rate of the first 30 times is 0.01, the learning rate of 30-50 times is 0.001, and a total of 50 iterations are updated.

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9, weight_decay = 5e-4)
# optimization method: SGDM
if epoch == 30: optimizer = optim.SGD(model.parameters(), lr = 0.001, momentum = 0.9,
weight_decay = 5e-4) # Optimization method: SGDM
```

It can be found from the F1 indicator score graph during the training process that there is a significant indicator score improvement after 30 iterations.

Gated Recurrent Unit (GRU):

GRU is a variant of LSTM. It combines the forget gate and input gate of LSTM into a single **update gate**, combining the cell states C_{t-1} and h_{t-1}, although the principles and results are not much different from LSTM, **the parameters of the model are reduced by about a quarter, and the training is faster**, so it has gradually begun to be an optimized substitute for LSTM.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Among them, σ is the activation function $\mathrm{sigmoid}$, formulas 1～3 omit the expression of offset.

Since the principle of GRU and LSTM is similar (but the gate operation is different), this article will not repeat its principle part.

```
        self.lstm = nn.GRU(embedding_dim, hidden_dim // 2, num_layers = 1, bidirectional =
True, batch_first = True) # BiGRU
```

However, in this experiment, it was found that the effect of BiGRU+CRF was not as good as that of BiLSTM+CRF (the highest value was only about 0.93), so the BiLSTM+CRF model was maintained in the end.

Thoughts of experiment:

In this experiment, the main task is to "build a BiLSTM+CRF word segmentation model, then train and test Chinese word segmentation on a data set". Since I have basically learned BiLSTM and the training and testing process of neural networks in the "artificial intelligence" course, the main difficulty for me in this experiment is the learning and application of the CRF layer (including the training process and the principles of verification/Viterbi decoding algorithm).

In the course of the experiment, after I learned about the working principle of CRF in the process of building the BiLSTM+CRF model, it did not take much time to build it. Instead, the process of training and adjusting the hyperparameter combination took a lot of time: when batch processing is not added, the training iteration is nearly an hour at a time. If you want to iterate training 50 times, it will take five days; after joining the batch, the batch size is 256, and it still takes two hours for 50 iterations. If you reduce the batch size or increase the number of iterations, the training time will cost more.

For the super parameter combination of the experimental results, it is the common super parameter combination that I used to do RNN experiments according to the "artificial intelligence" course, and in the process of multiple adjustments of parameters (modified to GRU, batch size to 512, and optimization method to Adam, etc.), are inferior to the original hyper-parameter combination (reflected in the first 10 iterations of the verified F1 indicators are significantly behind the initial hyper-parameter combination), so there is no need to run the iteration 50 times, I finally directly take the super parameter combination in the experimental results.