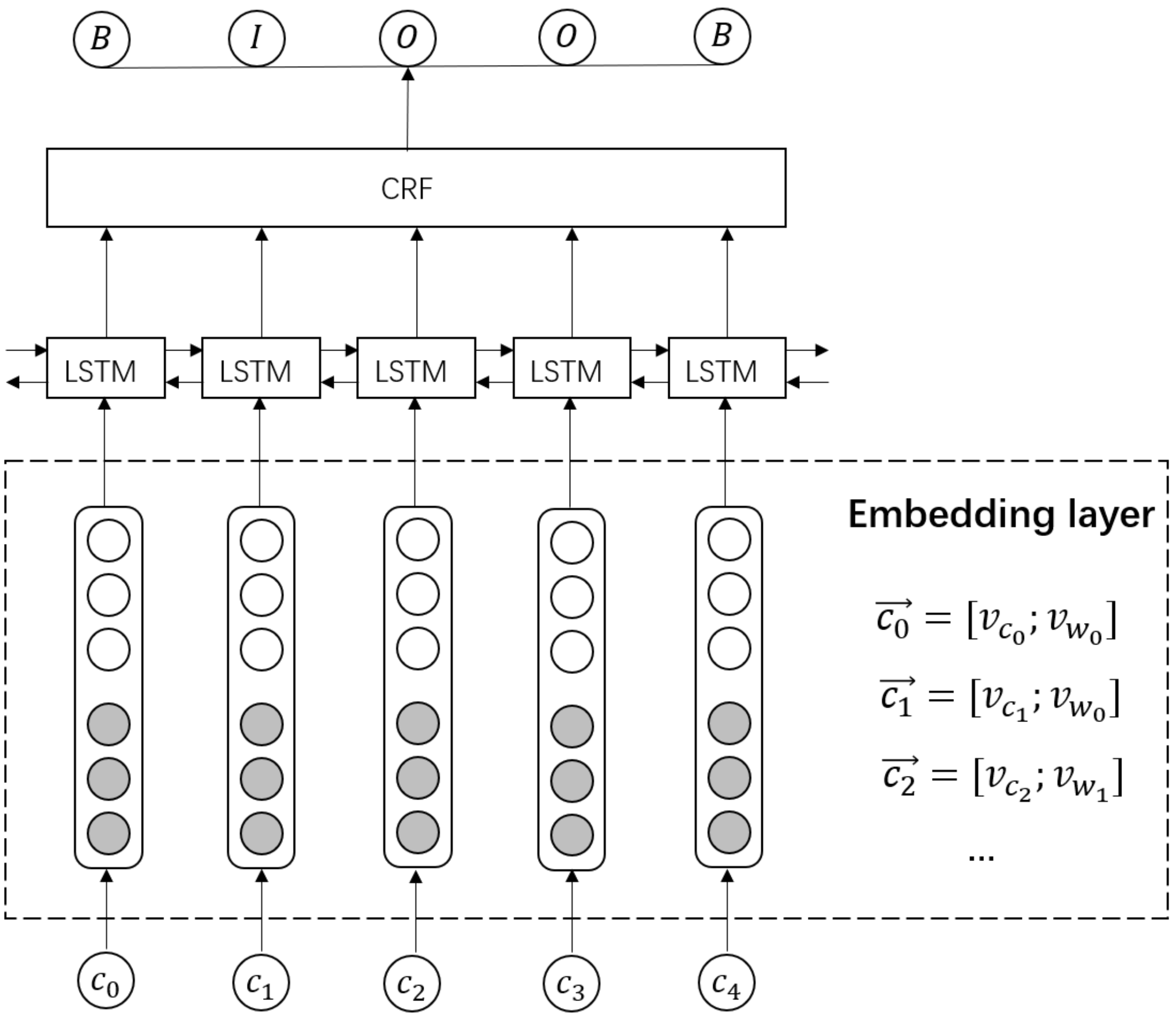
PJ2 Part3实验报告

BiLSTM+CRF

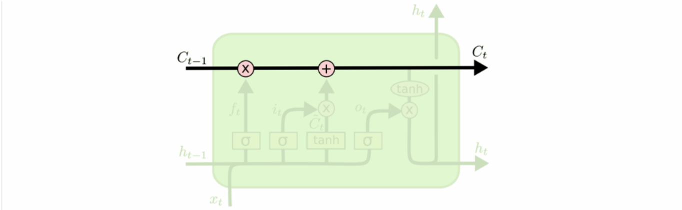
BiLSTM-CRF模型主体由双向长短时记忆网络（Bi-LSTM）和条件随机场（CRF）组成，模型输入是字符特征，输出是每个字符对应的预测标签。BiLSTM-CRF模型的主体框架如下，下面将对BiLSTM和 CRF 进行分述。

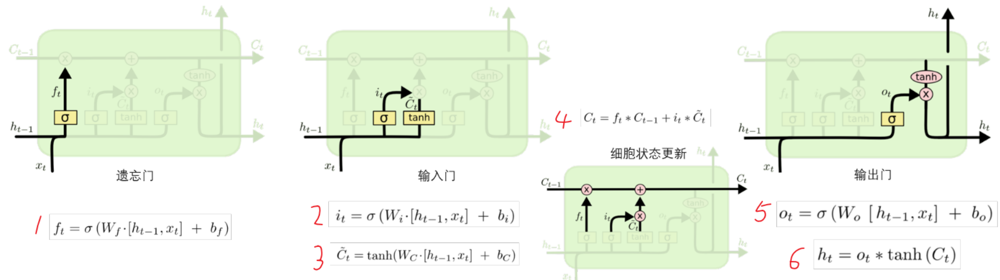


BiLSTM-CRF框架图

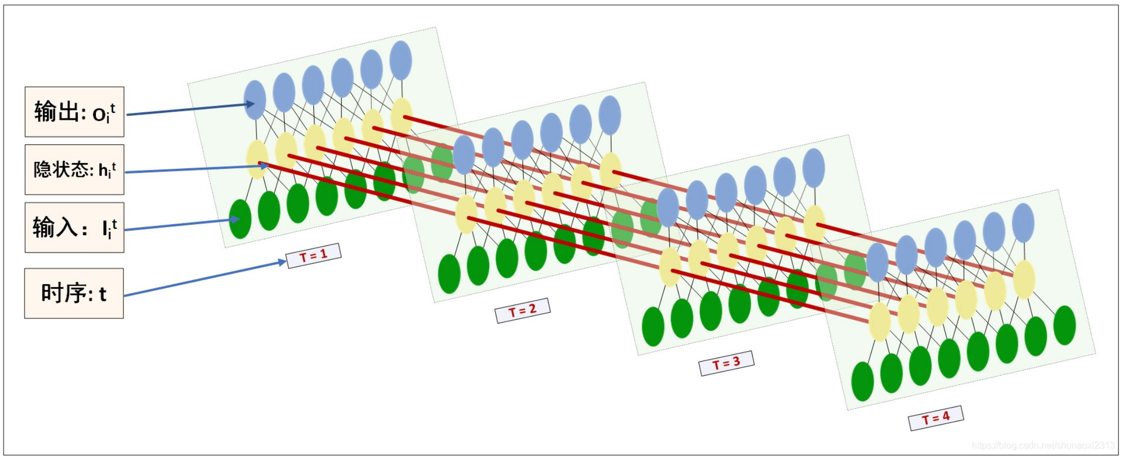
LSTM

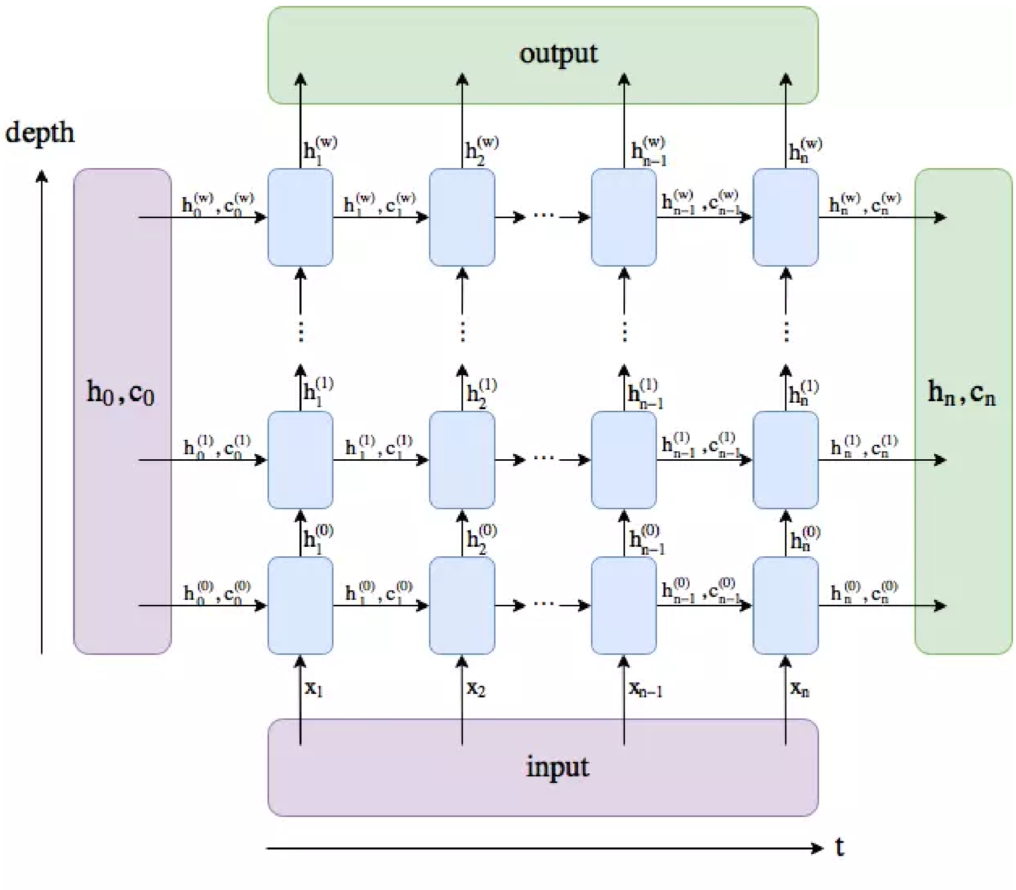
LSTM是一种特殊的循环神经网络，可以解决RNN的长期依赖问题，其关键就是细胞状态，见下图中贯穿单元结构上方的水平线。细胞状态在整个链上运行，只有一些少量的线性交互，从而保存长距离的信息流。具体而言，LSTM一共有三个门来维持和调整细胞状态，包括遗忘门，输入门，输出门。

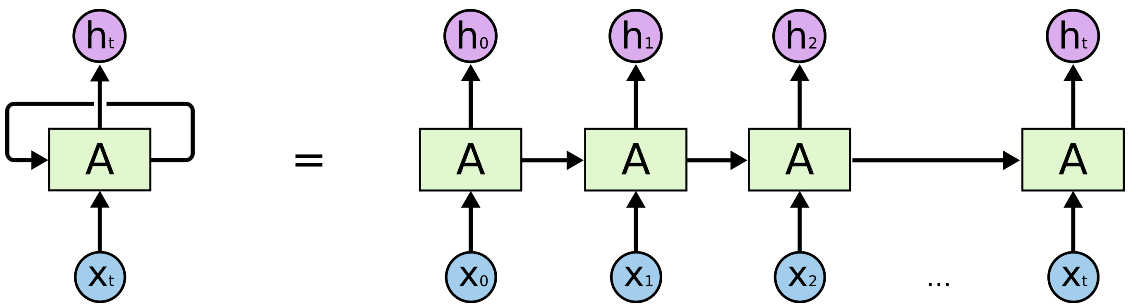




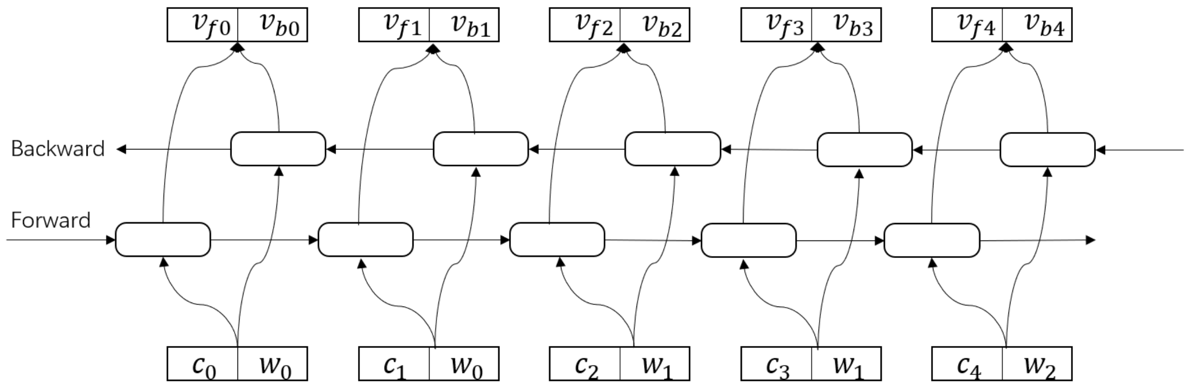
遗忘门接收h_{t-1}和x_t，通过公式1输出一个在 0 到 1 之间的数值f_t，该数值会作用于上一个细胞状态C_{t-1}，1 表示“完全保留”，0 表示“完全忘记”；输入门接收h_{t-1}和x_t，通过公式2输出一个在 0 到 1 之间的数值，已控制当前候选状态$\hat{C}_t$有多少信息需要保留，至于候选状态$\hat{C}_t$，则通过公式3由tanh层创建一个新的候选值向量，然后根据上一个细胞状态C_{t-1}和遗忘值f_t、新的细胞状态C_t和输入值i_t，由公式4更新细胞状态；输出门接收h_{t-1}和x_t，通过公式5输出一个在 0 到 1 之间的数值o_t，最后公式6决定了当前状态C_t有多少信息需要输出。

> 对于LSTM时间维度的理解：
>
> 一般来说，循环神经网络是MLP增加了一个时间维度，结合下图可视化LSTM辅助理解：
>
> 
>

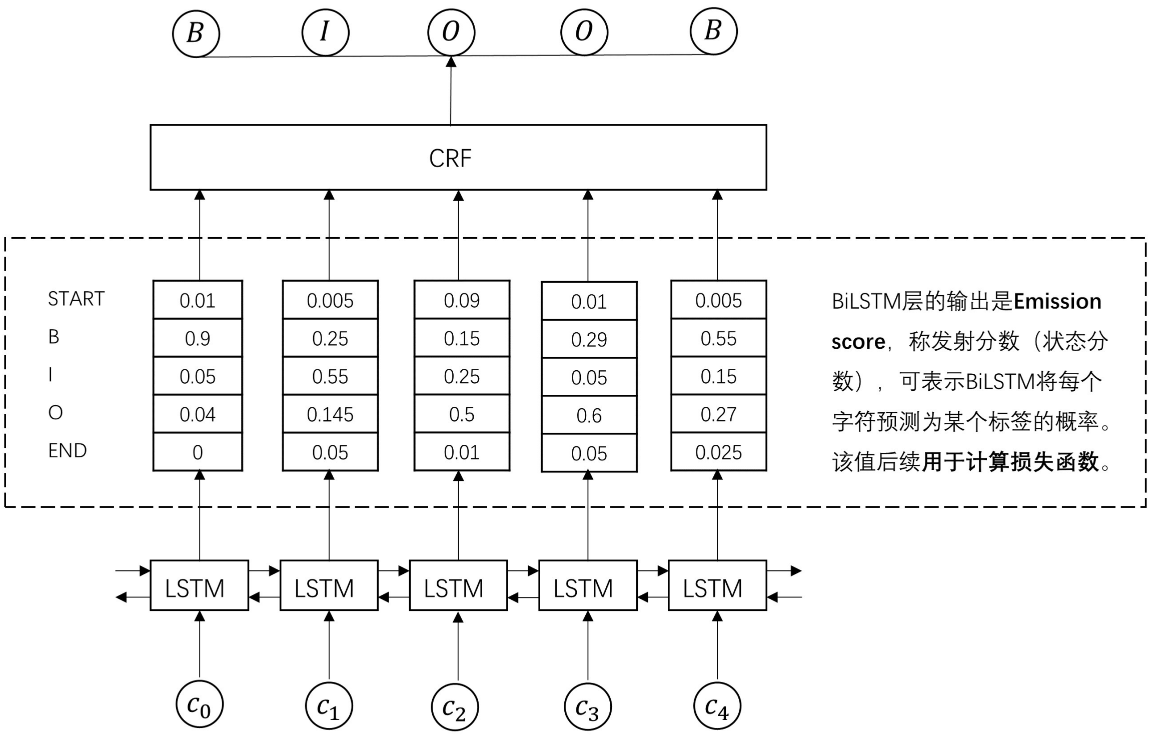


> LSTM的隐藏态参数不仅受到输入输出的影响，也会随着时间改变，这个改变就体现在上述的细胞中的三种门对之前知识的取舍。
>
> 而RNN是在时间上共享参数。也就是在每一层depth中只有一个LSTMcell，即上面第二张图中每一层只有一个LSTM单元,如下图：
>
> 
>

在BiLSTM-CRF中，一般使用一层的双向LSTM是足够的。因此，BiLSTM对输入embeddings的特征提取过程如下图：



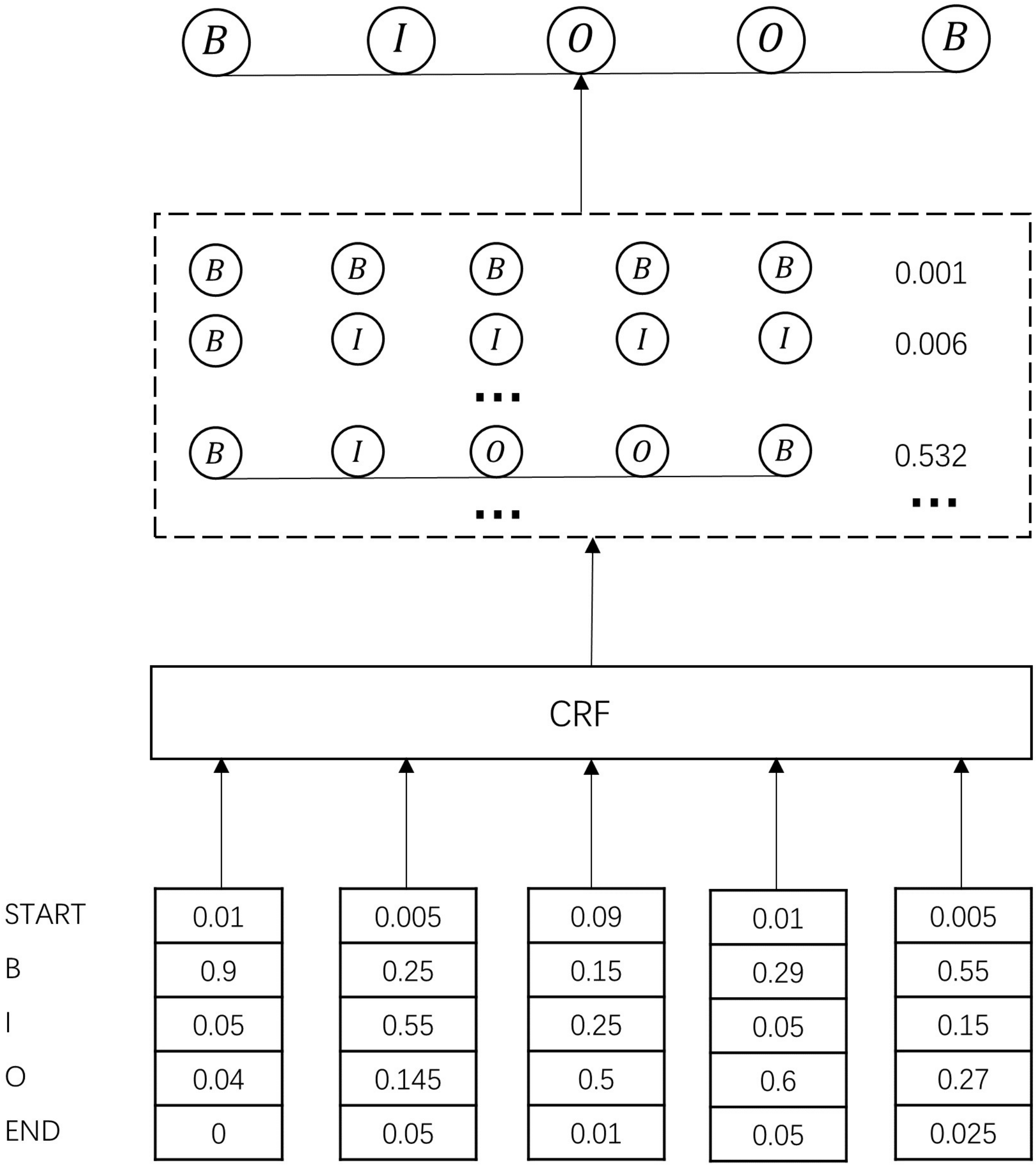
BiLSTM接收每个字符的embedding，并预测每个字符的对每个标注标签的概率。但是，我们也知道上图得到的拼接向量维度大小为[num_directions, hidden_size]。为将输入表示为字符对应各个类别的分数，需要在BiLSTM层加入一个全连接层，通过softmax将向量映射为一个5数值的分布概率。（也可不加softmax激活函数，直接使用全连接层映射为一个5数值分布即可，区别仅在于得到的emission score值大小及该维加和是否等于1。下图示表示使用了softmax，可见各列加和为1）



似乎我们通过BiLSTM已经找到每个单词对应的最大标签类别，但实际上，直接选择该步骤最大概率的标签类别得到的结果并不理想，原因在于，尽管LSTM能够通过双向的设置学习到观测序列之间的依赖，但softmax层的输出是相互独立的，输出相互之间并没有影响，只是在每一步挑选一个最大概率值的label输出，**这样的模型无法学习到输出的标注之间的转移依赖关系**（标签的概率转移矩阵）以及序列标注的约束条件，如句子的开头应该是“B”或“O”，而不是“I”等。为此，引入CRF层学习序列标注的约束条件，通过转移特征考虑输出label之间的顺序性，确保预测结果的有效性。

CRF

CRF层将BiLSTM的Emission_score作为输入，输出符合标注转移约束条件的、最大可能的预测标注序列(维特比解码)。如下，从这层意义上来说，BiLSTM-CRF模型本质上还是CRF模型，只不过是CRF部分的输入特征是BiLSTM部分的输出。



此处引用一下《统计学习方法》第11章对线性条件随机场的参数化形式定义：

定理 11.2（线性链条件随机场的参数化形式） 设 $P(Y|X)$ 为线性链条件随机场，则在随机变量 X 取值为 x 的条件下，随机变量 Y 取值为 y 的条件概率具有如下形式：

$$P(y|x)=\frac{1}{Z(x)}\exp\left(\sum_{i,k}\lambda_k t_k(y_{i-1},y_i,x,i)+\sum_{i,l}\mu_l s_l(y_i,x,i)\right) \tag{11.10}$$

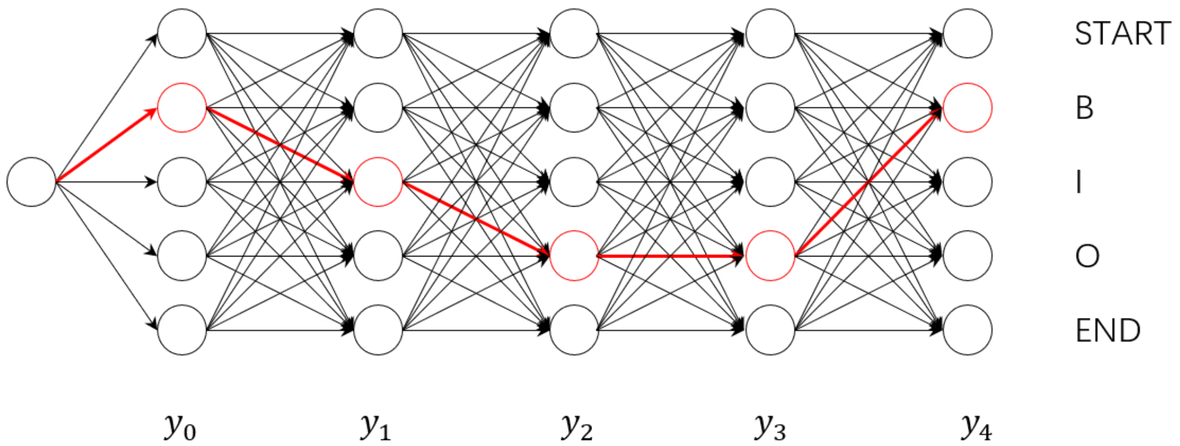
其中，

$$Z(x)=\sum_{y}\exp\left(\sum_{i,k}\lambda_k t_k(y_{i-1},y_i,x,i)+\sum_{i,l}\mu_l s_l(y_i,x,i)\right) \tag{11.11}$$

式中，t_k 和 s_l 是特征函数，λ_k 和 μ_l 是对应的权值。$Z(x)$ 是规范化因子，求和是在所有可能的输出序列上进行的。

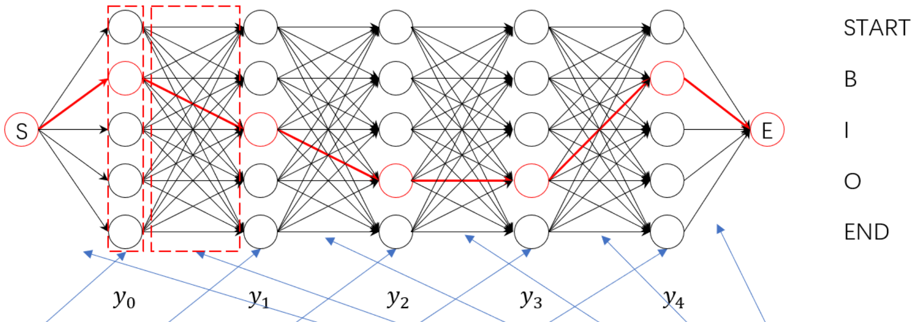
式 (11.10) 和式 (11.11) 是线性链条件随机场模型的基本形式，表示给定输入序列 x，对输出序列 y 预测的条件概率。式 (11.10) 和式 (11.11) 中，t_k 是定义在边上的特征函数，称为转移特征，依赖于当前和前一个位置；s_l 是定义在结点上的特征函数，称为状态特征，依赖于当前位置。t_k 和 s_l 都依赖于位置，是局部特征函数。通常，特征函数 t_k 和 s_l 取值为 1 或 0；当满足特征条件时取值为 1，否则为 0。条件随机场完全由特征函数 t_k，s_l 和对应的权值 λ_k，μ_l 确定。

理解红框中的规范化因子、转移特征、状态特征是理解BiLSTM-CRF模型中CRF的关键，以下面这张图进行说明：



在我们的例子中，输入x为c_0,c_1,c_2,c_3,c_4，理想输出y 为B,I,O,O,B，上图中红色线路。

- $Z(x)$，称规范化因子或配分函数。在公式（11.10）中，"$Z(x)$是规范化因子，求和是在所有可能的输出序列上进行的"。对应到我们的图中，其实就是图中所有可能的路径组合，由于输入序列长度为5，标注类型也为5，因此上图中共有5^5条不同路径。每条路径根据$\exp(*)$计算该路径的得分，加和得到$Z(x)$。
- s_t是节点上的状态特征，取决于当前节点；t_k是边上的转移特征，取决于当前和前一个节点。根据它们的定义，可以很自然的将它们与BiLSTM-CRF中的Emission Score和Transition Score匹配：Emission Score是由BiLSTM生成的、对当前字符标注的概率分布；Transition Score是加入CRF约束条件的、字符标注之间的概率转移矩阵。**从这个意义上讲，BiLSTM-CRF其实就是一个CRF模型，只不过用BiLSTM得到状态特征值s_t，用反向传播算法更新转移特征值t_k。**



0.01	0.005	0.09	0.01	0.005
0.9	0.25	0.15	0.29	0.55
0.05	0.55	0.25	0.05	0.15
0.04	0.145	0.5	0.6	0.27
0	0.05	0.01	0.05	0.025

BiLSTM的Emission Score

	STATE	B	I	O	END
STATE	0	0.45	0.02	0.45	0.08
B	0	0.1	0.4	0.3	0.2
I	0	0.1	0.3	0.4	0.2
O	0	0.25	0.05	0.3	0.4
END	0	0	0	0	0

序列标注转移矩阵

在模型训练过程中，模型损失函数定义如下：

$$\mathrm{P}\left(\bar{\mathrm{y}}\,|\mathrm{x}\right)=\frac{\exp(\mathrm{score}(\mathrm{x},\bar{\mathrm{y}}\,))}{\sum_{\mathrm{y}}\exp(\mathrm{score}(\mathrm{x},\mathrm{y}))}$$

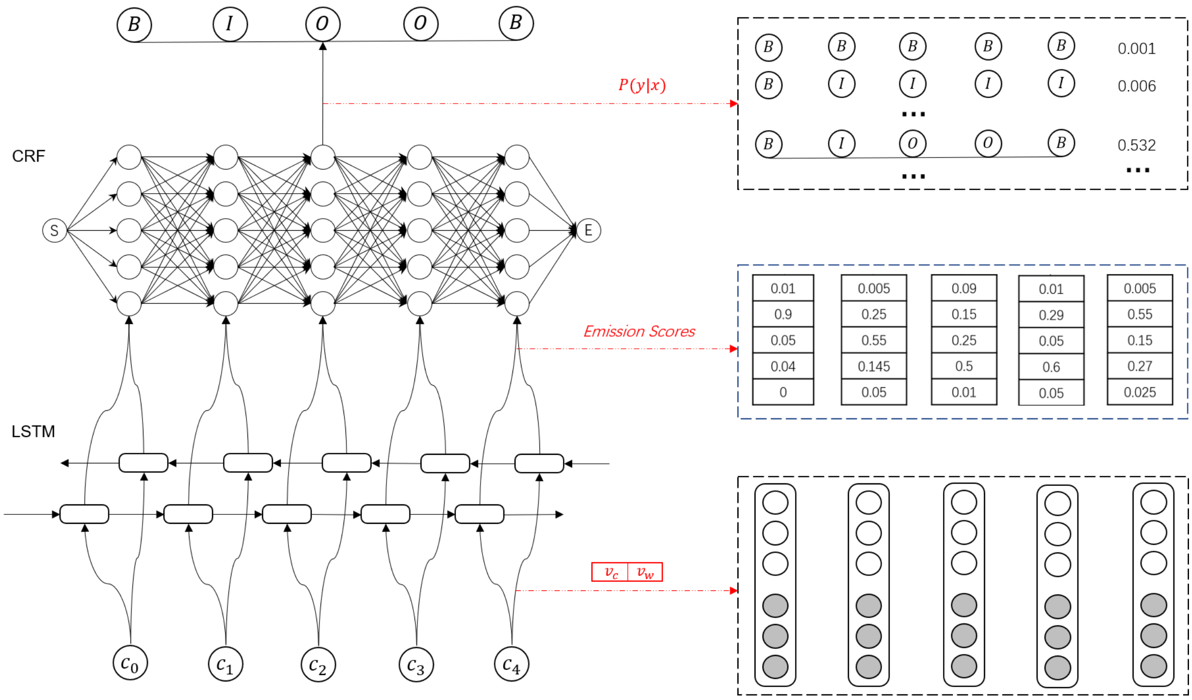
$$\mathrm{score}(\mathrm{x},\mathrm{y})=\sum_{\mathrm{i}=1}^{\mathrm{n}}\mathrm{P}_{\mathrm{i},\mathrm{y}_\mathrm{i}}+\sum_{\mathrm{i}=0}^{\mathrm{n}}\mathrm{A}_{\mathrm{y}_{\mathrm{i}-1},\mathrm{y}_\mathrm{i}}$$

其中，$\mathrm{P}_{\mathrm{i},\mathrm{y}_\mathrm{i}}$ **和$\mathrm{A}_{\mathrm{y}_{\mathrm{i}-1},\mathrm{y}_\mathrm{i}}$ 分别表示标注序列y中y_i 的Emission Score和Transition Score**，通过查找上图中的"BiLSTM的Emission Score"和"序列标注转移矩阵"可以得到每个字符位置的得分，整个序列相加得到$\mathrm{score}(\mathrm{x},\mathrm{y})$。

模型训练过程中最大化对数似然函数：

$$\log\mathrm{P}\ (\bar{\mathrm{y}}\ |\ \mathrm{x})=\mathrm{score}\,(\mathrm{x},\bar{\mathrm{y}}\,)-\log\left(\sum_{\mathrm{y}}\exp\left(\mathrm{score}\left(\mathrm{x},\mathrm{y}\right)\right)\right)$$

最终BiLSTM-CRF模型如下：



主要代码

中英文任务答题代码相同，只有部分细节不同，此处以中文任务代码为例展示。

标签列表：

```
label = ['O','B-NAME', 'M-NAME', 'E-NAME', 'S-NAME','B-CONT',
         'M-CONT', 'E-CONT', 'S-CONT','B-EDU', 'M-EDU', 'E-EDU',
         'S-EDU','B-TITLE', 'M-TITLE', 'E-TITLE', 'S-TITLE',
         'B-ORG', 'M-ORG', 'E-ORG', 'S-ORG','B-RACE', 'M-RACE',
         'E-RACE', 'S-RACE','B-PRO', 'M-PRO', 'E-PRO', 'S-PRO',
         'B-LOC', 'M-LOC', 'E-LOC', 'S-LOC']
label_E = ["O","B-PER","I-PER","B-ORG","I-ORG","B-LOC","I-LOC","B-MISC","I-
MISC"]
```

工具函数：

```
# 工具函数
def argmax(vec):
    # return the argmax as a python int
    _, idx = torch.max(vec, 1)
    return idx.item()

def prepare_sequence(seq, to_ix):
    idxs = [to_ix.get(w,len(to_ix)) for w in seq]
    return torch.tensor(idxs, dtype=torch.long)

# Compute log sum exp in a numerically stable way for the forward algorithm
def log_sum_exp(vec):
    max_score = vec[0, argmax(vec)]
    max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
    return max_score + \
        torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

BiLSTM_CRF类定义：

```python
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
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.tagset_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # Maps the output of the LSTM into tag space.
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # Matrix of transition parameters.  Entry i,j is the score of
        # transitioning *to* i *from* j.
        self.transitions = nn.Parameter(
            torch.randn(self.tagset_size, self.tagset_size))

        # These two statements enforce the constraint that we never transfer
        # to the start tag and we never transfer from the stop tag
        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):
        return (torch.randn(2, 1, self.hidden_dim // 2),
                torch.randn(2, 1, self.hidden_dim // 2))

    def _forward_alg(self, feats):
        # Do the forward algorithm to compute the partition function
        init_alphas = torch.full((1, self.tagset_size), -10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # Wrap in a variable so that we will get automatic backprop
        forward_var = init_alphas

        # Iterate through the sentence
        for feat in feats:
            alphas_t = []  # The forward tensors at this timestep
            for next_tag in range(self.tagset_size):
                # broadcast the emission score: it is the same regardless of
                # the previous tag
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.tagset_size)
                # the ith entry of trans_score is the score of transitioning to
                # next_tag from i
                trans_score = self.transitions[next_tag].view(1, -1)
                # The ith entry of next_tag_var is the value for the
                # edge (i -> next_tag) before we do log-sum-exp
                next_tag_var = forward_var + trans_score + emit_score
                # The forward variable for this tag is log-sum-exp of all the
```

```python
                # scores.
                alphas_t.append(log_sum_exp(next_tag_var).view(1))
            forward_var = torch.cat(alphas_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]],
dtype=torch.long), tags])
        for i, feat in enumerate(feats):
            score = score + \
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vvars
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.tagset_size):
                # next_tag_var[i] holds the viterbi variable for tag i at the
                # previous step, plus the score of transitioning
                # from tag i to next_tag.
                # We don't include the emission scores here because the max
                # does not depend on them (we add them in below)
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
                viterbivars_t.append(next_tag_var[0][best_tag_id].view(1))
            # Now add in the emission scores, and assign forward_var to the set
            # of viterbi variables we just computed
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            backpointers.append(bptrs_t)

        # Transition to STOP_TAG
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        best_tag_id = argmax(terminal_var)
        path_score = terminal_var[0][best_tag_id]
```

```
        # Follow the back pointers to decode the best path.
        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        # Pop off the start tag (we dont want to return that to the caller)
        start = best_path.pop()
        assert start == self.tag_to_ix[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, sentence, tags):
        feats = self._get_lstm_features(sentence)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, tags)
        return forward_score - gold_score

    def forward(self, sentence):  # dont confuse this with _forward_alg above.
        # Get the emission scores from the BiLSTM
        lstm_feats = self._get_lstm_features(sentence)

        # Find the best path, given the features.
        score, tag_seq = self._viterbi_decode(lstm_feats)
        return score, tag_seq
```

主函数（数据导入、模型训练、模型保存）：

```
if __name__=='__main__':
    START_TAG = "<START>"
    STOP_TAG = "<STOP>"
    EMBEDDING_DIM = 5
    HIDDEN_DIM = 4

    loader = load.DataLoad()    # 数据导入对象
    training_data = loader.load2('Chinese/train.txt')

    word_to_ix = {}
    for sentence, tags in training_data:
        for word in sentence:
            if word not in word_to_ix:
                word_to_ix[word] = len(word_to_ix)
    with open('word_to_ix.txt','w',encoding='utf-8') as f:
        json_str=json.dumps(word_to_ix)
        f.write(json_str)
    f.close()

    tag_to_ix = {}
    num = 0
    for tag in label:
        tag_to_ix[tag] = num
        num += 1

    tag_to_ix[START_TAG] = num
    num += 1
    tag_to_ix[STOP_TAG] = num
```

```
    model = BiLSTM_CRF(len(word_to_ix)+1, tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM)
    optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)

    # Make sure prepare_sequence from earlier in the LSTM section is loaded
    from tqdm import tqdm

    for epoch in range(15):
        print("EPOCH{}".format(epoch))
        training_data_ = tqdm(training_data)
        for sentence, tags in training_data_:
            # Step 1. Remember that Pytorch accumulates gradients.
            # We need to clear them out before each instance
            model.zero_grad()

            # Step 2. Get our inputs ready for the network, that is,
            # turn them into Tensors of word indices.
            sentence_in = prepare_sequence(sentence, word_to_ix)
            targets = torch.tensor([tag_to_ix[t] for t in tags],
dtype=torch.long)

            # Step 3. Run our forward pass.
            loss = model.neg_log_likelihood(sentence_in, targets)

            # Step 4. Compute the loss, gradients, and update the parameters by
            # calling optimizer.step()
            loss.backward()
            optimizer.step()

    import joblib
    joblib.dump(model, "BiLSTM_CRF_15.joblib")
```

实验结果

中文：f1-score达到0.9235

micro avg	0.9235	0.9235	0.9235	13882
macro avg	0.7205	0.7098	0.7066	13882
weighted avg	0.9255	0.9235	0.9176	13882

英文：f1-score达到0.8954

micro avg	0.8930	0.8978	0.8954	51016
macro avg	0.7635	0.4816	0.5739	51016
weighted avg	0.8829	0.8978	0.8803	51016