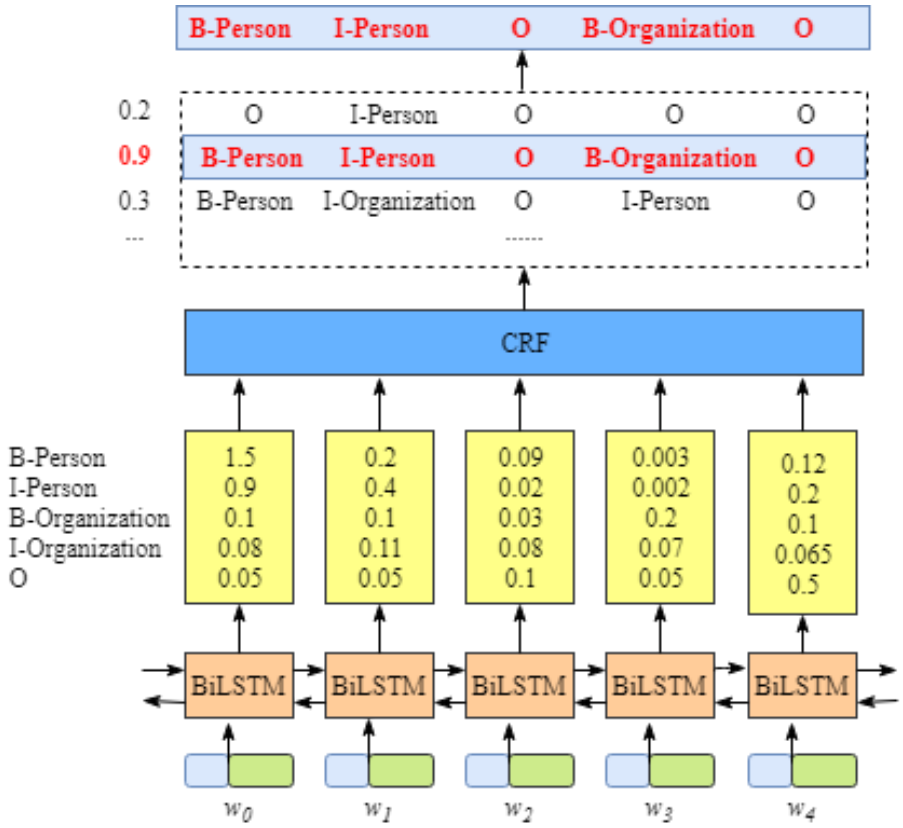
pj2 part3 BiLSTM+CRF模型

实验原理



该模型主要可以分为CRF层和BiLSTM层。虽然不需要知道BiLSTM层的细节，但是为了更容易的理解CRF层，我们需要知道BiLSTM层输出的意义是什么。上图说明BiLSTM层的输出是每个标签的分数，这些分数将作为CRF层的输入。然后，将BiLSTM层预测的所有分数输入CRF层。在CRF层中，选择预测得分最高的标签序列作为最佳答案。

在CRF层的损失函数中，我们有两种类型的参数：emission分数来自BiLSTM层，transition得分矩阵（为了使transition评分矩阵更健壮，我们将添加另外两个标签，<s>和</s>）存储了所有标签之间的所有得分。

$$LossFunction = \frac{P_{RealPath}}{P_1+P_2+\ldots+P_N}$$

上式为该模型中CRF层的损失函数。在训练过程中，CRF损失函数只需要两个分数：真实路径的分数和所有可能路径的总分数。所有可能路径的分数中，真实路径分数所占的比例会逐渐增加。

a. 真实路径分数：e^Si，其中Si 由两部分组成：Si=EmissionScore+TransitionScore

b. 所有可能路径分数：可以将损失函数变形为

$$\begin{aligned}
&LogLossFunction = -\log \frac{P_{RealPath}}{P_1+P_2+\ldots+P_N} \\
&= -\log \frac{e^{S_{RealPath}}}{e^{S_1}+e^{S_2}+\ldots+e^{S_N}} \\
&= -(\log(e^{S_{RealPath}}) - \log(e^{S_1}+e^{S_2}+\ldots+e^{S_N})) \\
&= -(S_{RealPath} - \log(e^{S_1}+e^{S_2}+\ldots+e^{S_N})) \\
&= -(\sum_{i=1}^{N} x_{iy_i} + \sum_{i=1}^{N-1} t_{y_i y_{i+1}} - \log(e^{S_1}+e^{S_2}+\ldots+e^{S_N}))
\end{aligned}$$

现在的问题转化为求红色方框中表达式的值。与viterbi解码类似，此次求解也利用动态规划的想法，利　　用两个向量进行存储和迭代：previous维数为标签tag的个数，每一维存储的是上一个位置所有的以某个tag为结尾的路径总分数的log之后的值；obs：维数为标签tag的个数，每一维存储的是当前位置所对应的为某个tag的emission score。

每一次迭代，Score=previous+obs+EmissionScore，然后更新previous值。

Code

class CRF

Plain Text | 复制代码

```
class CRF():
    def __init__(self, tags_dict):
        self.tags_dict = tags_dict
        self.tags_size = len(self.tags_dict)
        self.START_TAG = "<s>"
        self.STOP_TAG = "</s>"

        self.trans = nn.Parameter(torch.randn(self.tags_size, self.tags_size)).to(device)
        self.trans.data[self.tags_dict[self.START_TAG], :] = -10000
        self.trans.data[:, self.tags_dict[self.STOP_TAG]] = -10000

    def _forward_alg(self, feats, seq_len):
        init_alphas = torch.full((self.tags_size,), -10000.).to(device)
        init_alphas[self.tags_dict[self.START_TAG]] = 0.

        log_prob = torch.zeros(feats.shape[0], feats.shape[1] + 1, feats.shape[2], dtype=torch.float32).to(device)
        log_prob[:, 0, :] = init_alphas # reset start of each sentence to init_alphas

        trans = self.trans.unsqueeze(0).repeat(feats.shape[0], 1, 1)
        for seq_i in range(feats.shape[1]):
            emit = feats[:, seq_i, :]
            raw_prob = (log_prob[:, seq_i, :].unsqueeze(1).repeat(1, feats.shape[2], 1)  # (batch_size, tagset_size, tagset_size)
                        + trans + emit.unsqueeze(2).repeat(1, 1, feats.shape[2]))
            cloned = log_prob.clone()
            cloned[:, seq_i + 1, :] = log_sum_exp(raw_prob)
            log_prob = cloned

        log_prob = log_prob[range(feats.shape[0]), seq_len, :]
        log_prob += self.trans[self.tags_dict[self.STOP_TAG]].unsqueeze(0).repeat(feats.shape[0], 1)
        return log_sum_exp(log_prob)

    def _score_sentence(self, feats, tags, seq_len):
        score = torch.zeros(feats.shape[0]).to(device)
        start_tag = torch.tensor([self.tags_dict[self.START_TAG]]).unsqueeze(0).repeat(feats.shape[0], 1).to(device)
        tags = torch.cat([start_tag, tags], dim=1)
        for batch_i in range(feats.shape[0]):
            score[batch_i] = torch.sum(self.trans[tags[batch_i, 1:seq_len[batch_i] + 1], tags[batch_i, :seq_len[batch_i]]]) \
```

```
                + torch.sum(feats[batch_i, range(seq_len[batch_i]), tags[batch_i][1:seq_len[batch_i] + 1]]) \
                + self.trans[self.tags_dict[self.STOP_TAG], tags[batch_i][seq_len[batch_i]]]
        return score

    def _viterbi_decode(self, feats):
        states = []
        log_prob = torch.full((1, self.tags_size), -99999.).to(device)
        log_prob[0][self.tags_dict[self.START_TAG]] = 0

        for feat in feats:
            previous = []  # holds the backpointers for this step
            obs = []  # holds the viterbi variables for this step

            for next_tag in range(self.tags_size):
                next_prob = log_prob + self.trans[next_tag]
                best_tag_id = argmax(next_prob)
                previous.append(best_tag_id)
                obs.append(next_prob[0][best_tag_id].view(1))
            log_prob = (torch.cat(obs) + feat).view(1, -1)
            states.append(previous)

        log_prob += self.trans[self.tags_dict[self.STOP_TAG]]
        best_tag_id = argmax(log_prob)
        path_score = log_prob[0][best_tag_id]

        best_path = [best_tag_id]
        for state in reversed(states):
            best_tag_id = state[best_tag_id]
            best_path.append(best_tag_id)
        start = best_path.pop()
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, feats, tags, seq_len):
        forward_score = self._forward_alg(feats, seq_len)
        gt_score = self._score_sentence(feats, tags, seq_len)
        return torch.mean(forward_score - gt_score)
```

上述代码是CRF层的类定义，在此主要解释前溯函数_forward_alg和打分函数_score_sentence。

1. 前溯函数：

先初始化一个大小为 (batch_size, seq_len+1, tagset_size) 的 tensor，用来存储每个状态的概率，其中第一维是 batch 的大小，第二维是序列长度 + 1，第三维是标签数。将第一个位置（也就是起始位

置）的概率设置为初始状态的概率。将转移矩阵 self.trans 扩展为大小为 (batch_size, tagset_size, tagset_size) 的三维 tensor，其中第一维是 batch 的大小，后面两维是标签数。

计算在当前位置的所有状态的概率，并更新到 log_prob 中。

提取出每个句子的最终状态的概率，并将其加上 STOP_TAG 到每个标签的转移概率，再用 log_sum_exp() 函数计算对数和的最大值，最终得到该句子的概率。

2. 打分函数：

计算句子得分：将起始标签添加到 tags 序列的开头；用 self.trans 和 feats 分别计算转移概率和发射概率，然后求和；加上 STOP_TAG 到最后一个标签的转移概率。最后返回当前批次所有句子的得分。

class BiLSTM

Plain Text | 复制代码

```
class BiLSTM_CRF(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, words_dict, tags_dict):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.words_size = len(words_dict)
        self.tags_size = len(tags_dict)
        self.state = 'train'

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

        self.hidden2tag = nn.Linear(hidden_dim, self.tags_size, bias=True)
        self.crf = CRF(tags_dict)
        self.dropout = nn.Dropout(p=0.5, inplace=True)
        self.layer_norm = nn.LayerNorm(self.hidden_dim)

    def _get_lstm_features(self, sent, seq_len):
        embeds = self.word_embeds(sent)
        self.dropout(embeds)

        seq_len_cpu = seq_len.to("cpu")
        packed = torch.nn.utils.rnn.pack_padded_sequence(embeds, seq_len_cpu, batch_first=True, enforce_sorted=False)
        lstm_out, _ = self.lstm(packed)
        seq_unpacked, _ = torch.nn.utils.rnn.pad_packed_sequence(lstm_out, batch_first=True)

        seqence_output = self.layer_norm(seq_unpacked)
        lstm_feats = self.hidden2tag(seqence_output)
        return lstm_feats

    def forward(self, sent, seq_len, tags=''):
        feats = self._get_lstm_features(sent, seq_len)
        if self.state == 'train':
            loss = self.crf.neg_log_likelihood(feats, tags, seq_len)
            return loss
        elif self.state == 'eval':
            all_tag = []
            for i, feat in enumerate(feats):
                all_tag.append(self.crf._viterbi_decode(feat[:seq_len[i]])[1])
            return all_tag
        else:
```

```
42                 return self.crf._viterbi_decode(feats[0])[1]
43
```

上述代码是整个Bi-LSTM+CRF的类定义。

构造函数中进行了模型参数的初始化和定义。首先创建一个嵌入层（word_embeds），用于将输入的单词转换为词向量。接着建立一个LSTM层（lstm），该层包括两个LSTM（双向LSTM）层，其中隐层大小为hidden_dim/2，这样可以在前向和后向两个方向上捕获文本序列中的上下文信息。该层接受的输入是词向量，输出是隐层状态和LSTM单元状态。然后是一个全连接层（hidden2tag），它将隐层状态转换为标记预测。最后是一个CRF层（crf），它将模型输出映射为标记序列，并利用CRF算法进行解码。

_get_lstm_features方法用于计算LSTM层的输出。在该方法中，首先将输入的句子（也就是单词的索引）传递给嵌入层，得到词向量，然后应用dropout技术以防止过拟合。在最后一步中，利用pytorch提供的pack_padded_sequence和pad_packed_sequence函数，将LSTM的序列长度进行压缩和还原操作，以便根据序列的实际长度计算损失。然后将LSTM的输出传递到全连接层（hidden2tag），该层将LSTM的隐层状态转换为标记。

forward方法在模型的训练和评估期间使用。它接受三个参数：句子（sent）、每个句子的长度（seq_len）和可能的标记（tags）。使用_get_lstm_features方法计算出LSTM层的输出。如果状态为训练，则计算CRF层的损失（使用负对数似然）。如果状态为评估，则对于每个句子，使用CRF层的viterbi解码方法解码标记序列，并返回预测的所有标记。如果状态不是训练或评估，则使用CRF层的viterbi解码方法返回最可能的标记序列。

实验结果

在中文validation上的精度：0.9481

```
/Project2/BiLstm+CRF/NER/train.py
 -------< Evaluating >-------

micro_avg f1-score: 0.0053
Epoch 5
 -------< Evaluating >-------

micro_avg f1-score: 0.9481
```

在英文validation上的精度：0.8954

```
              precision    recall  f1-score   support

           O     0.9065    0.9879    0.9455     42759
       B-PER     0.6988    0.5239    0.5988      1842
       I-PER     0.7851    0.5731    0.6625      1307
       B-ORG     0.7900    0.3870    0.5195      1341
       I-ORG     0.7461    0.3209    0.4488       751
       B-LOC     0.8760    0.4192    0.5670      1837
       I-LOC     0.6948    0.4163    0.5207       257
      B-MISC     0.6106    0.2245    0.3283       922

   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

micro_avg f1-score: 0.8954
```

实验思考

与其他的模型相比，LSTM有如下几个优点：

1. 对序列上下文信息的建模：LSTM能够有效地捕获序列中的上下文信息，特别是对于长期依赖信息的捕获。因此，LSTM可以将之前的单词信息融合到后面的单词中，从而更好地进行序列标注。同时，CRF层可以利用前后标记之间的依赖建模整个序列的标注，从而增强了模型的表达能力。
2. 鲁棒性较强：LSTM能够有效地避免标注序列中存在的一些不规则情况（如缺失标记、噪声标记等），并能够在标注不完整的情况下保持较好的性能。
3. 模型的可解释性较好：CRF层的建模方式清晰明确，显式地建模了标记之间的依赖关系。这种显式地建模方式使得模型的输出更容易解释和理解。

参考链接

https://www.cnblogs.com/zjuhaohaoxuexi/p/15257605.html

CRF层部分参考了班上戴敖博韬的代码