Project 2: 基于 Bi-LSTM-CRF 的序列标注

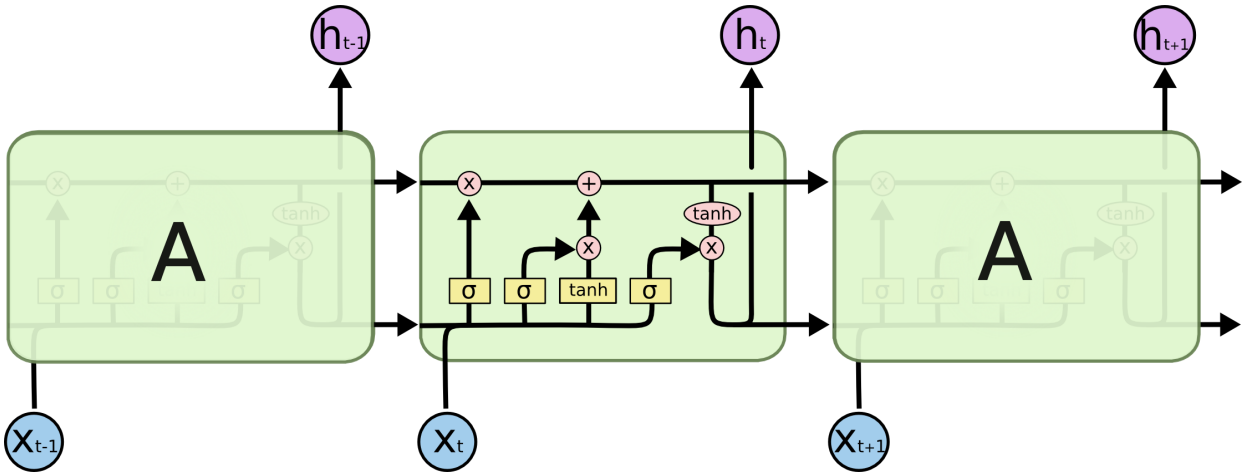
21307140069 田沐钊

Part 1: Bi-LSTM 与 CRF 的原理

1.1 对 LSTM 的原理分析

在RNN中，为了更好地处理序列任务，我们将上一步的输出并入当前步的输入，从而建立起上下文之间的联系，使最后的输出包含上下文关系。但是由于多层激活函数的嵌套，在反向传播时可能会出现梯度消失或梯度爆炸的情况，从而使前文信息丢失。于是，我们提出了LSTM（长短时记忆）。

在LSTM，我们引入了一组门控机制来维护长期记忆（或者说，细胞状态，Cell State），用长期记忆来储存前文信息，影响我们的输出。



为了维护长期记忆的及时有效更新，我们构造了如下门结构（以上图中从左到右的顺序依次介绍）：

1、遗忘门（forget gate）：使用sigmoid函数来决定前一时刻的细胞状态中有哪些内容需要被保留或遗忘。其结构如下：

```
#f_t权重初始化，在init函数中使用
self.U_f = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
self.V_f = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
self.b_f = nn.Parameter(torch.Tensor(hidden_sz))

# -----------手动分隔符-----------

#f_t门的激活运算，在forward函数中使用
f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.V_f + self.b_f)
```

然后将长时记忆与短时记忆f_t进行点乘。完成对细胞状态的一步更新。

2、输入门（input gate）：输入门由两个部分组成。分别是由sigmoid函数导出的输入门的激活值和由双曲正切激活函数导出的细胞候选值（Cell Candidate），细胞候选值在-1到1之间，具有较强的信息表达能力，而激活值在0到1之间，用于控制候选信息信息的保留程度，起到一个加权作用，对更关键的信息予以保留。这种结构可以更灵活有效地利用输入信息对细胞状态进行更新。

其具体结构如下：

```
#i_t权重初始化，在init函数中使用
self.U_i = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
self.V_i = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
self.b_i = nn.Parameter(torch.Tensor(hidden_sz))

# -----------手动分隔符-----------

#i_t门的激活运算，在forward函数中使用
i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.V_i + self.b_i)

# -----------手动分隔符-----------

#c_tilde_t权重初始化，在init函数中使用
self.U_c = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
self.V_c = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
self.b_c = nn.Parameter(torch.Tensor(hidden_sz))

# -----------手动分隔符-----------

#c_tilde_t门的激活运算，在forward函数中使用
c_tilde_t = torch.tanh(x_t @ self.U_c + h_t @ self.V_c + self.b_c)

#在forward函数中使用
c_t = f_t * c_t + i_t * c_tilde_t
```

3、输出门（output gate）：

```
#o_t权重初始化，在init函数中使用
self.U_o = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
self.V_o = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
self.b_o = nn.Parameter(torch.Tensor(hidden_sz))

# -----------手动分隔符-----------

#c_tilde_t门的激活运算，在forward函数中使用
o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.V_o + self.b_o)

#在forward函数中使用
h_t = o_t * torch.tanh(c_t)
```

注意，不要被长时记忆的“遗忘”、“更新”之类的操作字眼蒙蔽，而误以为其是某种复杂的、离散可枚举的具体数据类型，实际上其只是一种从前文数据总结出的一种“信息”，本身也是一个无甚玄妙之处的数据，本质上是让前文的数据加入当前输出的运算中来。

并且，由于长时记忆在更新过程中只涉及矩阵的乘法和加减运算，所以在反向传播时一般不会遇到梯度消失的情况（但无法避免梯度爆炸），所以能在当前步的输出中有效地纳入前文信息。

1.2 Bi-LSTM 与 LSTM 的关系

Bi-LSTM（双向长短期记忆网络）是LSTM（长短期记忆网络）的一种扩展形式。LSTM是一种递归神经网络（RNN）的变种，用于处理序列数据，具有记忆单元和门控机制，可以有效地捕捉和处理长期依赖关系。

LSTM通过使用门控单元来控制信息的流动和遗忘，从而解决了传统RNN在处理长序列时的梯度消失和梯度爆炸问题。它的门控机制包括输入门、遗忘门和输出门，使得网络可以选择性地接收、遗忘和输出信息。

Bi-LSTM则在LSTM的基础上引入了双向性。传统的LSTM只考虑了当前时间步之前的上下文信息，而Bi-LSTM在每个时间步同时考虑了过去和未来的上下文信息。它由两个独立的LSTM组成，一个按正序处理输入序列，另一个按反序处理输入序列。两个LSTM的隐藏状态可以通过连接或其他方式进行融合，以提取更全面的上下文特征。

Bi-LSTM在许多序列建模任务中表现出色，特别是对于需要全局上下文信息的任务。它能够捕捉序列中的前后依赖关系，并利用这些信息进行更准确的预测和建模。然而，相比于普通的LSTM，Bi-LSTM的计算成本更高，因为它需要同时处理两个方向的序列。

1.3 CRF

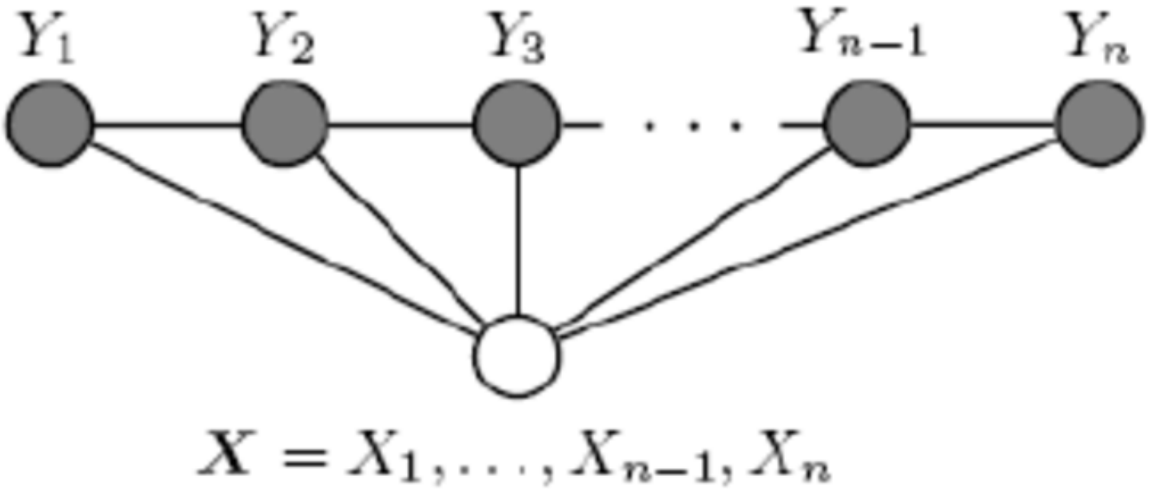
对于CRF，我们给出准确的数学语言描述：

设X与Y是随机变量，$P(Y|X)$是给定X时Y的条件概率分布，若随机变量Y构成的是一个马尔科夫随机场，则称条件概率分布$P(Y|X)$是条件随机场。

注意在CRF的定义中，我们并没有要求X和Y有相同的结构。而实现中，我们一般都假设X和Y有相同的结构，即:

$$X=(X_1,X_2,\dots X_n),\ \ Y=(Y_1,Y_2,\dots Y_n)$$

我们一般考虑如下图所示的结构：X和Y有相同的结构的CRF就构成了线性链条件随机场。



设$X=(X_1,X_2,\dots X_n),\ \ Y=(Y_1,Y_2,\dots Y_n)$均为线性链表示的随机变量序列，在给定随机变量序列$X$的情况下，随机变量$Y$的条件概率分布$P(Y|X)$构成条件随机场，即满足马尔科夫性：

$$P(Y_i|X,Y_1,Y_2,\dots Y_n)=P(Y_i|X,Y_{i-1},Y_{i+1})$$

则称$P(Y|X)$为线性链条件随机场。

那我们如何将其转化为可以学习的机器学习模型呢？这是通过特征函数和其权重系数来定义的。什么是特征函数呢？

在linear-CRF中，特征函数分为两类，第一类是定义在Y节点上的节点特征函数，这类特征函数只和当前节点有关，记为：

$$s_l(y_i, x, i), \quad l = 1, 2, \dots L$$

其中L是定义在该节点的节点特征函数的总个数，i是当前节点在序列的位置。

第二类是定义在Y上下文的局部特征函数，这类特征函数只和当前节点和上一个节点有关，记为：

$$t_k(y_{i-1}, y_i, x, i), \quad k = 1, 2, \dots K$$

其中K是定义在该节点的局部特征函数的总个数，i是当前节点在序列的位置。之所以只有上下文相关的局部特征函数，没有不相邻节点之间的特征函数，是因为我们的linear-CRF满足马尔科夫性。

无论是节点特征函数还是局部特征函数，它们的取值只能是0或者1。即满足特征条件或者不满足特征条件。同时，我们可以为每个特征函数赋予一个权值，用以表达我们对这个特征函数的信任度。假设t_k的权重系数是λ_k,s_l的权重系数是μ_l,则linear-CRF由我们所有的$t_k, \lambda_k, s_l, \mu_l$共同决定。

此时我们得到了linear-CRF的参数化形式如下：

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

其中，$Z(x)$为规范化因子：$Z(x) = \sum_y exp\Big(\sum_{i,k} \lambda_k t_k(y_{i-1}, y_i, x, i) + \sum_{i,l} \mu_l s_l(y_i, x, i)\Big)$

回到特征函数本身，每个特征函数定义了一个linear-CRF的规则，则其系数定义了这个规则的可信度。所有的规则和其可信度一起构成了我们的linear-CRF的最终的条件概率分布。

假设我们在某一节点我们有K_1个局部特征函数和K_2个节点特征函数，总共有$K = K_1 + K_2$个特征函数。我们用一个特征函数$f_k(y_{i-1}, y_i, x, i)$来统一表示如下:

$$f_k(y_{i-1}, y_i, x, i) = \begin{cases} t_k(y_{i-1}, y_i, x, i) & k = 1, 2, \dots K_1 \\ s_l(y_i, x, i) & k = K_1 + l, \ l = 1, 2 \dots, K_2 \end{cases}$$

对$f_k(y_{i-1}, y_i, x, i)$在各个序列位置求和得到：$f_k(y, x) = \sum_{i=1}^{n} f_k(y_{i-1}, y_i, x, i)$

同时我们也统一$f_k(y_{i-1}, y_i, x, i)$对应的权重系数w_k如下：

$$w_k = \begin{cases} \lambda_k & k = 1, 2, \dots K_1 \\ \mu_l & k = K_1 + l, \ l = 1, 2 \dots, K_2 \end{cases}$$

这样，我们的linear-CRF的参数化形式简化为：$P(y|x) = \frac{1}{Z(x)} exp \sum_{k=1}^{K} w_k f_k(y, x)$

其中，$Z(x)$为规范化因子：$Z(x) = \sum_y exp \sum_{k=1}^{K} w_k f_k(y, x)$

如果将上两式中的w_k与f_k的用向量表示，即:

$$w = (w_1, w_2, \dots w_K)^T \quad F(y, x) = (f_1(y, x), f_2(y, x), \dots f_K(y, x))^T$$

则linear-CRF的参数化形式简化为内积形式如下：

$$P_w(y|x) = \frac{exp(w \cdot F(y,x))}{Z_w(x)} = \frac{exp(w \cdot F(y,x))}{\sum\limits_y exp(w \cdot F(y,x))}$$

将统一后的linear-CRF公式加以整理，我们还可以将linear-CRF的参数化形式写成矩阵形式。为此我们定义一个$m \times m$的矩阵M，m为y所有可能的状态的取值个数。M定义如下：

$$M_i(x) = \Big[M_i(y_{i-1}, y_i|x)\Big] = \Big[exp(W_i(y_{i-1}, y_i|x))\Big] = \Big[exp(\sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, x, i))\Big]$$

我们引入起点和终点标记$y_0 = start, y_{n+1} = stop$, 这样，标记序列$y$的规范化概率可以通过$n+1$个矩阵元素的乘积得到，即：

$$P_w(y|x) = \frac{1}{Z_w(x)} \prod_{i=1}^{n+1} M_i(y_{i-1}, y_i|x)$$

其中$Z_w(x)$为规范化因子。

要计算条件概率$P(y_i|x)$和$P(y_{i-1}，y_i|x)$，我们可以使用使用前向后向算法来完成。首先我们来看前向概率的计算。

我们定义$\alpha_i(y_i|x)$表示序列位置i的标记是y_i时，在位置i之前的部分标记序列的非规范化概率。之所以是非规范化概率是因为我们不想加入一个不影响结果计算的规范化因子$Z(x)$在分母里面。

$$M_i(y_{i-1}, y_i|x) = exp(\sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, x, i))$$

这个式子定义了在给定y_{i-1}时，从y_{i-1}转移到y_i的非规范化概率。

这样，我们很容易得到序列位置$i+1$的标记是y_{i+1}时，在位置$i+1$之前的部分标记序列的非规范化概率$\alpha_{i+1}(y_{i+1}|x)$的递推公式：$\alpha_{i+1}(y_{i+1}|x) = \alpha_i(y_i|x)[M_{i+1}(y_{i+1}, y_i|x)] \;\; i = 1, 2, \ldots, n+1$

在起点处，我们定义：$\alpha_0(y_0|x) = \begin{cases} 1 & y_0 = start \\ 0 & else \end{cases}$

假设我们可能的标记总数是m, 则y_i的取值就有m个，我们用$\alpha_i(x)$表示这m个值组成的前向向量如下：$\alpha_i(x) = (\alpha_i(y_i = 1|x), \alpha_i(y_i = 2|x), \ldots \alpha_i(y_i = m|x))^T$

同时用矩阵$M_i(x)$表示由$M_i(y_{i-1}, y_i|x)$形成的$m \times m$阶矩阵：$M_i(x) = \Big[M_i(y_{i-1}, y_i|x)\Big]$

这样递推公式可以用矩阵乘积表示：$\alpha_{i+1}^T(x) = \alpha_i^T(x) M_{i+1}(x)$

同样的。我们定义$\beta_i(y_i|x)$表示序列位置i的标记是y_i时，在位置i之后的从$i+1$到n的部分标记序列的非规范化概率。

这样，我们很容易得到序列位置$i+1$的标记是y_{i+1}时，在位置i之后的部分标记序列的非规范化概率$\beta_i(y_i|x)$的递推公式：$\beta_i(y_i|x) = [M_{i+1}(y_i, y_{i+1}|x)]\beta_{i+1}(y_{i+1}|x)$

在终点处，我们定义：$\beta_{n+1}(y_{n+1}|x) = \begin{cases} 1 & y_{n+1} = stop \\ 0 & else \end{cases}$

如果用向量表示，则有：$\beta_i(x) = M_{i+1}(x)\beta_{i+1}(x)$

由于规范化因子$Z(x)$的表达式是：$Z(x) = \sum_{c=1}^{m} \alpha_n(y_c|x) = \sum_{c=1}^{m} \beta_1(y_c|x)$

也可以用向量来表示$Z(x)$:$Z(x) = \alpha_n^T(x) \cdot \mathbf{1} = \mathbf{1}^T \cdot \beta_1(x)$

其中，$\mathbf{1}$是m维全1向量。

在使用梯度下降法求解模型参数之前，我们需要定义我们的优化函数，一般极大化条件分布$P_w(y|x)$的对数似然函数如下：$L(w) = log\prod_{x,y} P_w(y|x)^{\overline{P}(x,y)} = \sum_{x,y} \overline{P}(x,y) log P_w(y|x)$

其中$\overline{P}(x,y)$为经验分布，可以从先验知识和训练集样本中得到,这点和最大熵模型类似。为了使用梯度下降法，我们现在极小化$f(w) = -L(P_w)$如下：

$$\begin{aligned} f(w) &= -\sum_{x,y} \overline{P}(x,y) log P_w(y|x) \\ &= \sum_{x,y} \overline{P}(x,y) log Z_w(x) - \sum_{x,y} \overline{P}(x,y) \sum_{k=1}^{K} w_k f_k(x,y) \\ &= \sum_{x} \overline{P}(x) log Z_w(x) - \sum_{x,y} \overline{P}(x,y) \sum_{k=1}^{K} w_k f_k(x,y) \\ &= \sum_{x} \overline{P}(x) log \sum_{y} exp \sum_{k=1}^{K} w_k f_k(x,y) - \sum_{x,y} \overline{P}(x,y) \sum_{k=1}^{K} w_k f_k(x,y) \end{aligned}$$

对w求导可以得到：$\frac{\partial f(w)}{\partial w} = \sum_{x,y} \overline{P}(x) P_w(y|x) f(x,y) - \sum_{x,y} \overline{P}(x,y) f(x,y)$

那么，给定条件随机场的条件概率$P(y|x)$和一个观测序列x，怎么求出满足$P(y|x)$最大的序列y呢。

我们使用维特比算法，其本身是一个动态规划算法，利用了两个局部状态和对应的递推公式，从局部递推到整体，进而得解。对于具体不同的问题，仅仅是这两个局部状态的定义和对应的递推公式不同而已。

对于我们linear-CRF中的维特比算法，我们的第一个局部状态定义为$\delta_i(l)$,表示在位置i标记l各个可能取值(1,2...m)对应的非规范化概率的最大值。之所以用非规范化概率是，规范化因子$Z(x)$不影响最大值的比较。根据$\delta_i(l)$的定义，我们递推在位置$i+1$标记l的表达式为：

$$\delta_{i+1}(l) = \max_{1 \le j \le m} \{\delta_i(j) + \sum_{k=1}^{K} w_k f_k(y_i = j, y_{i+1} = l, x, i)\} \ , l = 1, 2, \ldots m$$

和HMM的维特比算法类似，我们需要用另一个局部状态$\Psi_{i+1}(l)$来记录使$\delta_{i+1}(l)$达到最大的位置i的标记取值,这个值用来最终回溯最优解，$\Psi_{i+1}(l)$的递推表达式为：

$$\Psi_{i+1}(l) = arg \max_{1 \le j \le m} \{\delta_i(j) + \sum_{k=1}^{K} w_k f_k(y_i = j, y_{i+1} = l, x, i)\} \ , l = 1, 2, \ldots m$$

现在我们总结下 linear-CRF模型维特比算法流程：

输入：模型的K个特征函数，和对应的K个权重。观测序列$x = (x_1, x_2, \ldots x_n)$,可能的标记个数$m$

输出：最优标记序列$y^* = (y_1^*, y_2^*, \ldots y_n^*)$

1. 初始化：$\delta_1(l) = \sum_{k=1}^{K} w_k f_k(y_0 = start, y_1 = l, x, i)\} \ , l = 1, 2, \ldots m$
   $\Psi_1(l) = start \ , l = 1, 2, \ldots m$

2. 对于$i=1,2...n-1$,进行递推:

$$\delta_{i+1}(l)=\max_{1\le j\le m}\{\delta_i(j)+\sum_{k=1}^{K}w_kf_k(y_i=j,y_{i+1}=l,x,i)\}\ ,l=1,2,...m$$

$$\Psi_{i+1}(l)=arg\ \max_{1\le j\le m}\{\delta_i(j)+\sum_{k=1}^{K}w_kf_k(y_i=j,y_{i+1}=l,x,i)\}\ ,l=1,2,...m$$

3. 终止: $y_n^*=arg\ \max_{1\le j\le m}\delta_n(j)$

4)回溯: $y_i^*=\Psi_{i+1}(y_{i+1}^*)\ ,i=n-1,n-2,...1$

最终得到最优标记序列$y^*=(y_1^*,y_2^*,...y_n^*)$

Part 2: 代码分析

以下是模型架构的部分代码:

```
class BiLSTMCRF(nn.Module):
    def __init__(self, tag_size, vocab_size, dropout_rate=0.5, embed_size=256,
hidden_size=256):
        # 使用给定的参数初始化 BiLSTMCRF 模型
        super(BiLSTMCRF, self).__init__()
        # 存储参数
        self.dropout_rate = dropout_rate
        self.embed_size = embed_size
        self.hidden_size = hidden_size
        self.tag_size = tag_size
        self.vocab_size = vocab_size

        # 定义词嵌入层，将输入词转换为dense向量
        self.embedding = nn.Embedding(vocab_size + 1, embed_size)
        # 定义dropout层，以防止过拟合
        self.dropout = nn.Dropout(dropout_rate)
        # 定义双向LSTM层，以模型序列数据
        self.lstm = nn.LSTM(input_size=embed_size, hidden_size=hidden_size,
bidirectional=True)
        # 定义线性层，以将LSTM输出转换为发射分数
        self.hidden2emit_score = nn.Linear(hidden_size * 2, tag_size)
        # 定义CRF层的转移矩阵
        self.transition = nn.Parameter(torch.randn(tag_size, tag_size))

    def forward(self, sentences, tags, sen_lengths):
        # 创建掩码，以指示填充token
        mask = (sentences != self.vocab_size - 1).to(self.device)
        # 将输入句子嵌入
        sentences = self.embedding(sentences.transpose(0, 1))
        # 使用BiLSTM层计算发射分数
        emit_score = self.bilstm(sentences, sen_lengths)
        # 使用CRF层计算损失
        loss = self.crf(tags, mask, emit_score)
        return loss

    def bilstm(self, sentences, sent_lengths):
```

```
        # 将填充句子打包成单个张量
        padded_sentences = pack_padded_sequence(sentences, sent_lengths)
        # 使用BiLSTM层计算隐藏状态
        hidden_states, _ = self.lstm(padded_sentences)
        # 将填充隐藏状态unpack
        hidden_states, _ = pad_packed_sequence(hidden_states, batch_first=True)
        # 计算发射分数
        emit_score = self.hidden2emit_score(hidden_states)
        # 将dropout应用于发射分数
        emit_score = self.dropout(emit_score)
        return emit_score

    def crf(self, tags, mask, emit_score):
        # 计算给定标签的真实分数
        batch_size, sentence_len = tags.shape
        real_score = torch.gather(emit_score, dim=2,
index=tags.unsqueeze(dim=2)).squeeze(dim=2)
        real_score[:, 1:] += self.transition[tags[:, :-1], tags[:, 1:]]
        real_score = (real_score * mask.type(torch.float)).sum(dim=1)

        # 使用CRF算法计算前向分数
        temp = torch.unsqueeze(emit_score[:, 0], dim=1)
        for i in range(1, sentence_len):
            num_unfinished = mask[:, i].sum()
            temp_unfinished = temp[: num_unfinished]
            emit_plus_transition = emit_score[: num_unfinished, i].unsqueeze(dim=1)
+ self.transition
            forward_score = temp_unfinished.transpose(1, 2) + emit_plus_transition
            max_score = forward_score.max(dim=1)[0].unsqueeze(dim=1)
            forward_score -= max_score
            temp_unfinished = max_score + torch.logsumexp(forward_score,
dim=1).unsqueeze(dim=1)
            temp = torch.cat((temp_unfinished, temp[num_unfinished:]), dim=0)
        temp = temp.squeeze(dim=1)
        max_temp = temp.max(dim=-1)[0]
        logsumexp_all_scores = max_temp + torch.logsumexp(temp -
max_temp.unsqueeze(dim=1), dim=1)

        # 计算对数似然和损失
        log_likelihood = real_score - logsumexp_all_scores
        loss = -log_likelihood
        return loss

    def predict(self, sentences, sen_lengths):
        # 预测给定句子的标签
        batch_size = sentences.shape[0]
        mask = (sentences != self.vocab_size - 1)
        sentences = sentences.transpose(0, 1)
        sentences = self.embedding(sentences)
        emit_score = self.bilstm(sentences, sen_lengths)

        tags = [[[i] for i in range(self.tag_size)]] * batch_size
```

```
        temp = torch.unsqueeze(emit_score[:, 0], dim=1)
        for i in range(1, sen_lengths[0]):
            num_unfinished = mask[:, i].sum()
            temp_unfinished = temp[: num_unfinished]
            emit_plus_transition = self.transition + emit_score[: num_unfinished,
i].unsqueeze(dim=1)
            new_temp_unfinished = temp_unfinished.transpose(1, 2) +
emit_plus_transition
            temp_unfinished, max_idx = torch.max(new_temp_unfinished, dim=1)
            max_idx = max_idx.tolist()
            tags[: num_unfinished] = [[tags[b][k] + [j] for j, k in
enumerate(max_idx[b])] for b in range(num_unfinished)]
            temp = torch.cat((torch.unsqueeze(temp_unfinished, dim=1),
temp[num_unfinished:]), dim=0)
        temp = temp.squeeze(dim=1)

        _, max_idx = torch.max(temp, dim=1)
        max_idx = max_idx.tolist()
        tags = [tags[batch_idx][tag_idx] for batch_idx, tag_idx in
enumerate(max_idx)]
        return tags

    def save(self, filepath):
        # 将模型参数保存到文件
        params = {
            'tag_size': self.tag_size,
            'vocab_size': self.vocab_size,
            'args': dict(dropout_rate=self.dropout_rate, embed_size=self.embed_size,
hidden_size=self.hidden_size),
            'state_dict': self.state_dict()
        }
        torch.save(params, filepath)
```

其中重点解释 crf 部分的计算原理：

CRF 层在这段代码中计算损失的步骤如下：

1. **计算真实分数**：真实分数是给定输入句子下的真实标签的分数。这是通过使用 `emit_score` 和 `tags` 张量计算的。`emit_score` 是 BiLSTM 层的输出，而 `tags` 张量包含每个句子的真实标签。真实分数是通过将 `emit_score` 和标签之间的转移分数相加计算的。
2. **计算前向分数**：前向分数是所有可能标签序列的分数。这是通过使用 `emit_score` 和转移分数计算的。前向分数是使用动态规划方法计算的，其中每个标签的分数是在前一个标签的基础上计算的。
3. **计算似然**：似然是真实分数和前向分数之间的差异。这是通过计算 `log_likelihood = real_score - logsumexp_all_scores` 得到的。
4. **计算损失**：损失是似然的负值，计算为 `loss = -log_likelihood`。

Part 3：实验步骤

3.1 数据的预处理

我们定义数据集的类如下：

```
class NER_dataset():
    def __init__(self, datapath):
        super().__init__()
        self.sentences = []
        self.word_mapping = None
        self.tag_mapping = None

        current_sentence_words = []
        current_sentence_tags = []
        # 从文件路径中读取数据
        with open(datapath, 'r', encoding='utf-8') as file:
            reader = csv.DictReader(file)
            for row in reader:
                word = row['word']
                sentence_boundary = word.endswith('。') or word.endswith('? ') or
word.endswith('! ')

                current_sentence_words.append(word)
                current_sentence_tags.append(row['expected'])

                if sentence_boundary:
                    self.sentences.append((current_sentence_words,
current_sentence_tags))
                    current_sentence_words = []
                    current_sentence_tags = []

    def get_tag_mapping(self, tag_mapping):
        self.tag_mapping = tag_mapping

    def get_word_mapping(self, word_record):
        self.word_mapping = word_record

    def __getitem__(self, idx):
        sentence_words, sentence_tags = self.sentences[idx]
        if self.word_mapping is not None:
            sentence_words = [self.word_mapping[word] if word in self.word_mapping
else 0 for word in sentence_words]
        if self.tag_mapping is not None:
            sentence_tags = [self.tag_mapping[tag] for tag in sentence_tags]
        return sentence_words, sentence_tags

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

其中 `tag_mapping` 和 `word_mapping` 的定义如下：

```
class tag_mapping():
```

```
    def __init__(self):
        self.encode_mapping = {
            'O': 0,
            'S-GPE': 1,
            'S-PER': 2,
            'B-ORG': 3,
            'E-ORG': 4,
            'S-ORG': 5,
            'M-ORG': 6,
            'S-LOC': 7,
            'E-GPE': 8,
            'B-GPE': 9,
            'B-LOC': 10,
            'E-LOC': 11,
            'M-LOC': 12,
            'M-GPE': 13,
            'B-PER': 14,
            'E-PER': 15,
            'M-PER': 16,
            '<PAD>': 17
        }
        self.num_tag = len(self.encode_mapping)
        self.decode_mapping = {value: key for key, value in
self.encode_mapping.items()}

    def encode(self, tags):
        return [self.encode_mapping[tag] for tag in tags]

    def decode(self, codes):
        return [self.decode_mapping[code] for code in codes]

    def __len__(self):
        return self.num_tag

class word_mapping():
    def __init__(self, dataset):
        self.encode_mapping = {}
        self.num_word = 1

        # 从数据集中读取词汇表
        for words, _ in dataset:
            for word in words:
                if word not in self.encode_mapping:
                    self.encode_mapping[word] = self.num_word
                    self.num_word += 1
        self.encode_mapping["<PAD>"] = self.num_word
        self.encode_mapping["<unk>"] = 0
        self.num_word += 1
        self.decode_mapping = {val: key for key, val in self.encode_mapping.items()}

    def __len__(self):
        return self.num_word
```

```
    def encode(self, words):
        return [self.encode_mapping[word] if word in self.encode_mapping else 0 for
word in words]

    def decode(self, codes):
        return [self.decode_mapping[code] if code in self.decode_mapping else
'<unk>' for code in codes]
```

这里使用 `word_mapping` 将文本域的字符串映射到了整数域的索引，方便模型的进一步嵌入处理。

下面是数据预处理的代码：

```
# 数据准备与预处理
train_data_path = 'data/train.csv'
valid_data_path = 'data/dev.csv'

train_data = NER_dataset(train_data_path)
valid_data = NER_dataset(valid_data_path)

TagMapping = tag_mapping()
WordMapping = word_mapping(train_data)

train_data.get_tag_mapping(TagMapping.encode_mapping)
train_data.get_word_mapping(WordMapping.encode_mapping)
valid_data.get_tag_mapping(TagMapping.encode_mapping)
valid_data.get_word_mapping(WordMapping.encode_mapping)

tag_size = TagMapping.num_tag
vocab_size = WordMapping.num_word
```

3.2 模型的初始化与训练

这里我们并没有使用pytorch给定的dataloader，而是用自己实现的 batch_iter 对象实现了batch的构建。主要目的是方便对不同句长的序列进行padding，并对输出结果进行去padding操作。

其中 batch_iter 和 pad 的代码实现如下：

```
def batch_iter(data, batch_size=32, shuffle=True):
    # 计算数据点的总数
    data_size = len(data)
    # 创建表示数据点的索引列表
    indices = list(range(data_size))
    # 如果 shuffle 为 True，则打乱索引顺序
    if shuffle:
        random.shuffle(indices)
    # 计算批次的数量
    batch_num = (data_size + batch_size - 1) // batch_size
    # 遍历每个批次
    for i in range(batch_num):
        # 获取当前批次的索引
```

```
        batch_indices = indices[i * batch_size: (i + 1) * batch_size]
        # 使用索引获取当前批次的数据点
        batch = [data[idx] for idx in batch_indices]
        # 根据每个数据点中第一个元素的长度，按降序对批次进行排序
        batch = sorted(batch, key=lambda x: len(x[0]), reverse=True)
        # 将句子和标签从批次中分离出来
        sentences = [x[0] for x in batch]
        tags = [x[1] for x in batch]
        # 使用 yield 关键字将句子和标签作为一个批次返回
        yield sentences, tags


def pad(data, padded_token, device):
    lengths = [len(sent) for sent in data]
    max_len = lengths[0]
    padded_data = []
    for s in data:
        padded_data.append(s + [padded_token] * (max_len - len(s)))
    return torch.tensor(padded_data, device=device), lengths
```

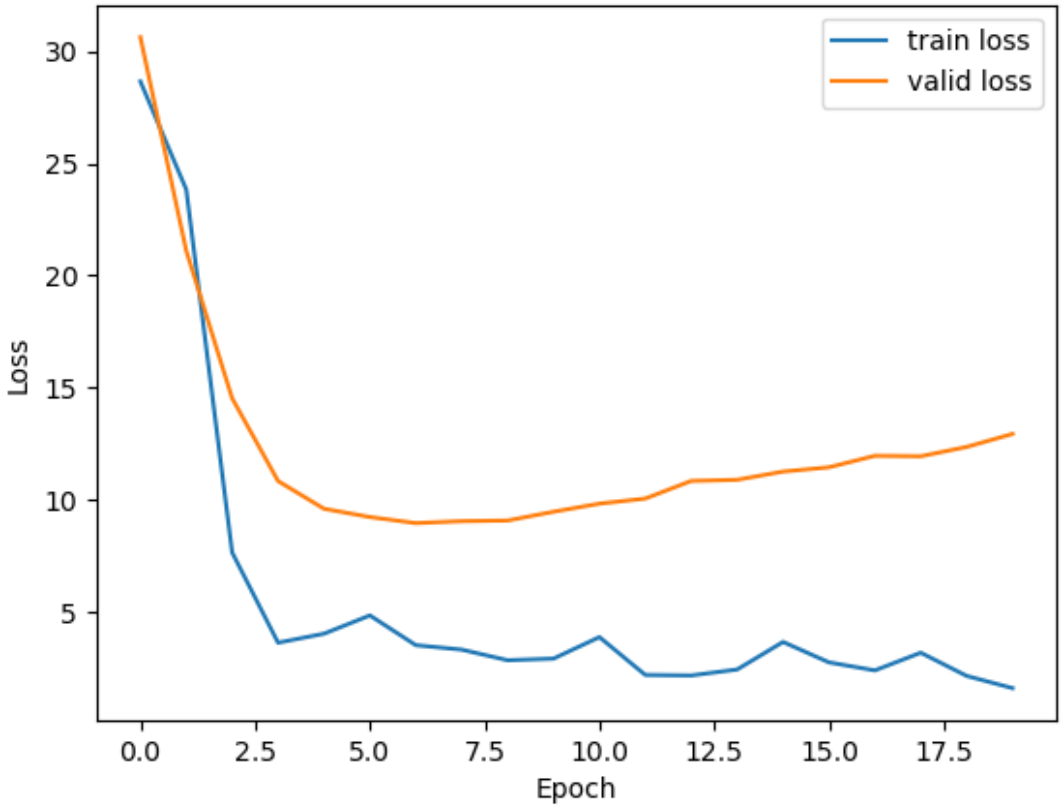
模型的构建和训练代码如下：

```
dropout = 0.5
embed_size = 256
hidden_size = 256
batch_size = 32
max_epoch = 0
lr = 0.001
clip_max_norm = 5.0

model_save_path = './model/model_1.pth'

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 对模型进行初始化
model = BiLSTM_CRF.BiLSTMCRF(tag_size, vocab_size, dropout, embed_size,
hidden_size).to(device)
for name, param in model.named_parameters():
    if 'weight' in name:
        nn.init.normal_(param.data, 0, 0.01)
    else:
        nn.init.constant_(param.data, 0)

optimizer = torch.optim.Adam(model.parameters(), lr=lr)

train_losses = []
valid_losses = []
print('start training...')
for epoch in range(max_epoch):
    num_iter = 0
    for sentences, tags in tqdm(batch_iter(train_data, batch_size=batch_size)):
```

```
        num_iter += 1
        sentences, sent_lengths = pad(sentences, vocab_size - 1, device)
        tags, _ = pad(tags, tag_size - 1, device)

        optimizer.zero_grad()
        batch_loss = model(sentences, tags, sent_lengths)  # shape: (b,)
        loss = batch_loss.mean()
        loss.backward()
        # 对梯度使用clip，防止参数过大
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=clip_max_norm)
        optimizer.step()

        if num_iter % 100 == 0:
            print('Epoch: %d, Iter: %d, Loss: %.4f' % (epoch, num_iter,
loss.item()))

    train_losses.append(loss.item())
    valid_losses.append(compute_valid_loss(model, valid_data, batch_size, device,
vocab_size, tag_size))
    # 每两个 epoch 保存一次模型
    if epoch % 2 == 0:
        model.save(model_save_path + f'_{epoch}')


model.save(model_save_path)
plot_losses(train_losses, valid_losses, 'losses_1.png')
```

这里每次迭代都使用了 `clip_grad_norm_`，用于对梯度进行裁剪。它的功能是限制梯度的范数不超过指定的阈值，以防止梯度爆炸的问题。该函数接受一个阈值参数 `max_norm`，它表示梯度的最大范数。函数会计算所有模型参数的梯度的总体范数，并检查是否超过了 `max_norm`。如果超过了阈值，则会对梯度进行缩放，使其范数不超过 `max_norm`，从而确保梯度的稳定性。

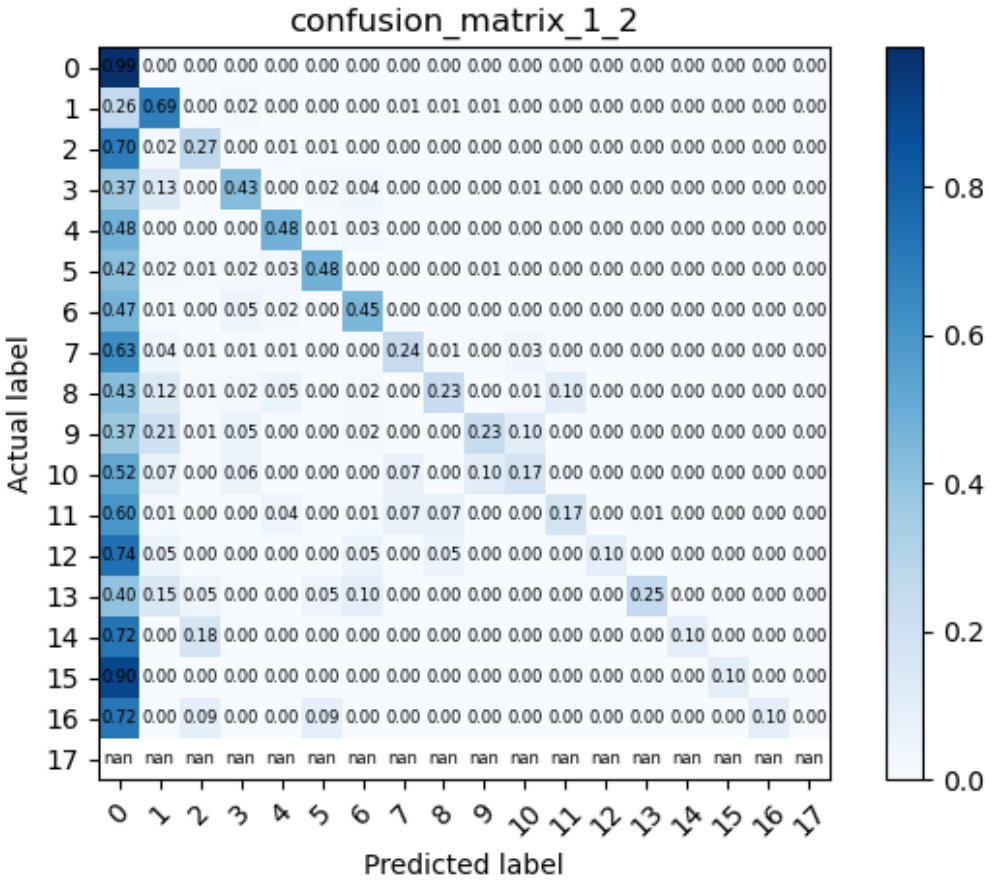
训练中在训练集和验证集上的 loss 曲线图如下：



可以看到模型在训练集上的 loss 函数在稳步下降并趋于收敛，但在验证集上的 loss 在第5个epoch后就开始上升，看似是发生了过拟合，但实际情况并非如此，具体见训练结果分析。

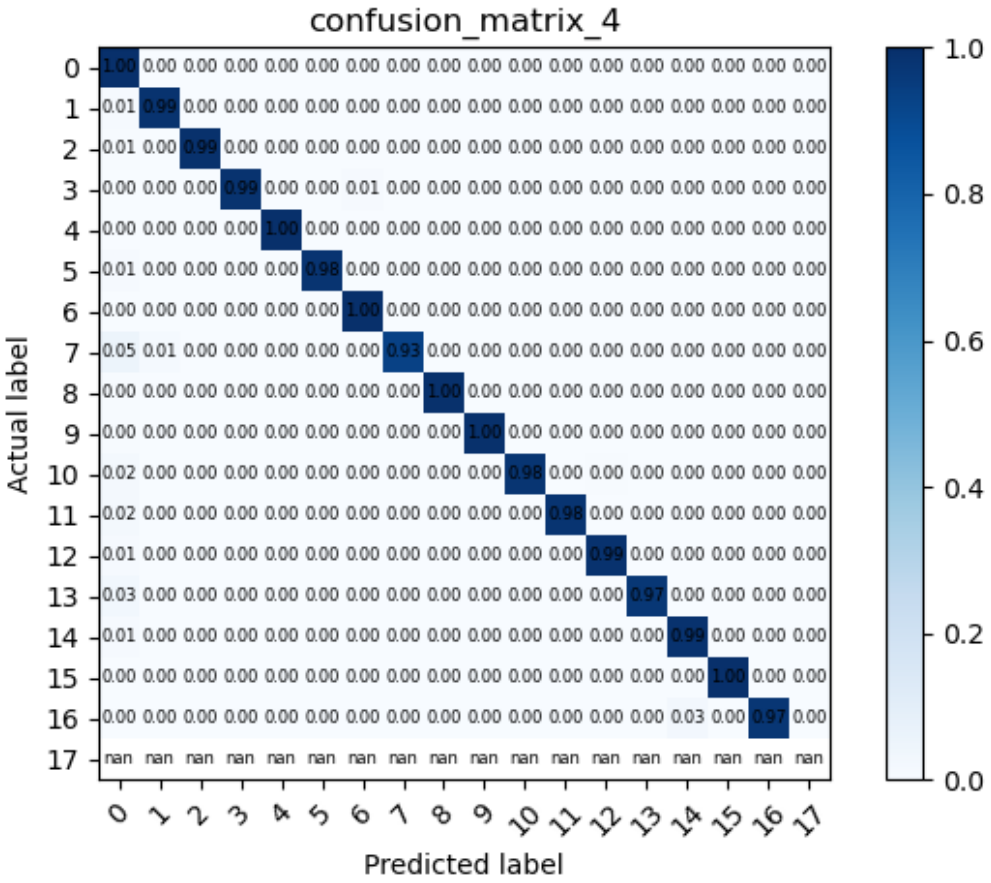
3.3 训练结果分析

下面我们用训练了20个epoch的模型在验证集上进行预测，并绘制了预测的混淆矩阵如下：



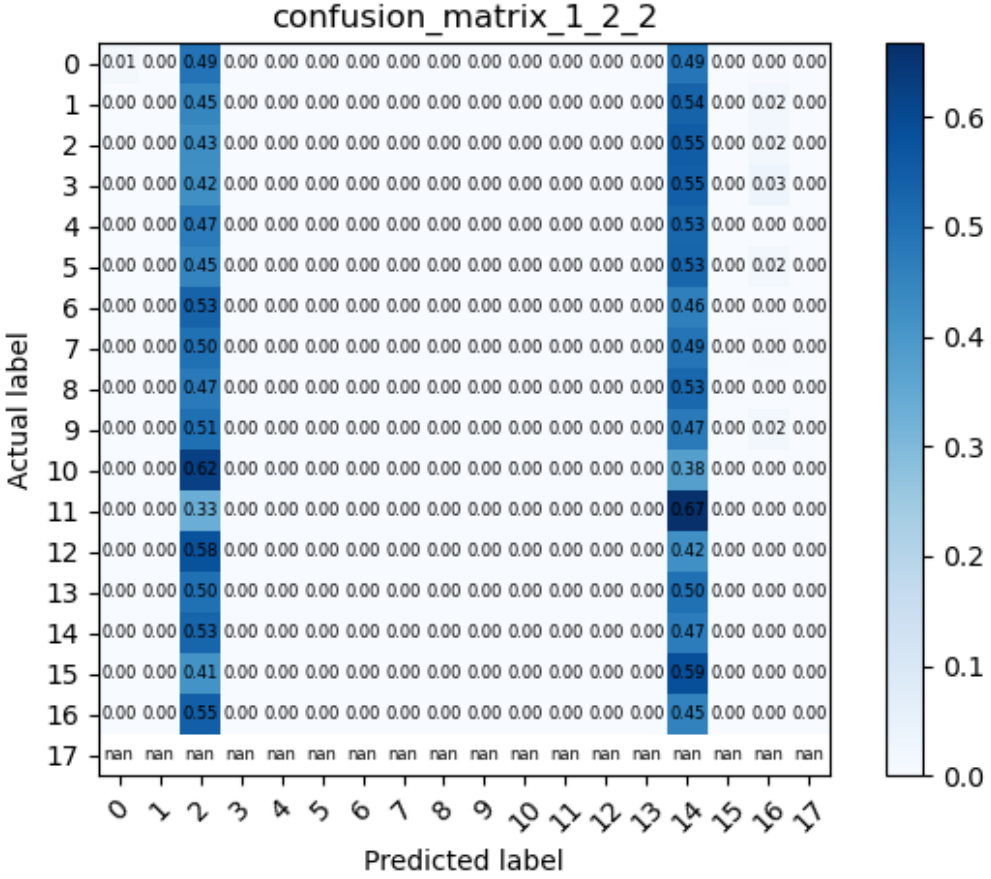
其中编号17对应的标签是用于填充的，故结果均为nan。可以看到其在预测时均有一定概率预测出准确的标签，但在许多标签上会过度倾向于误预测出标签`0`，整体准确率并不算很好。

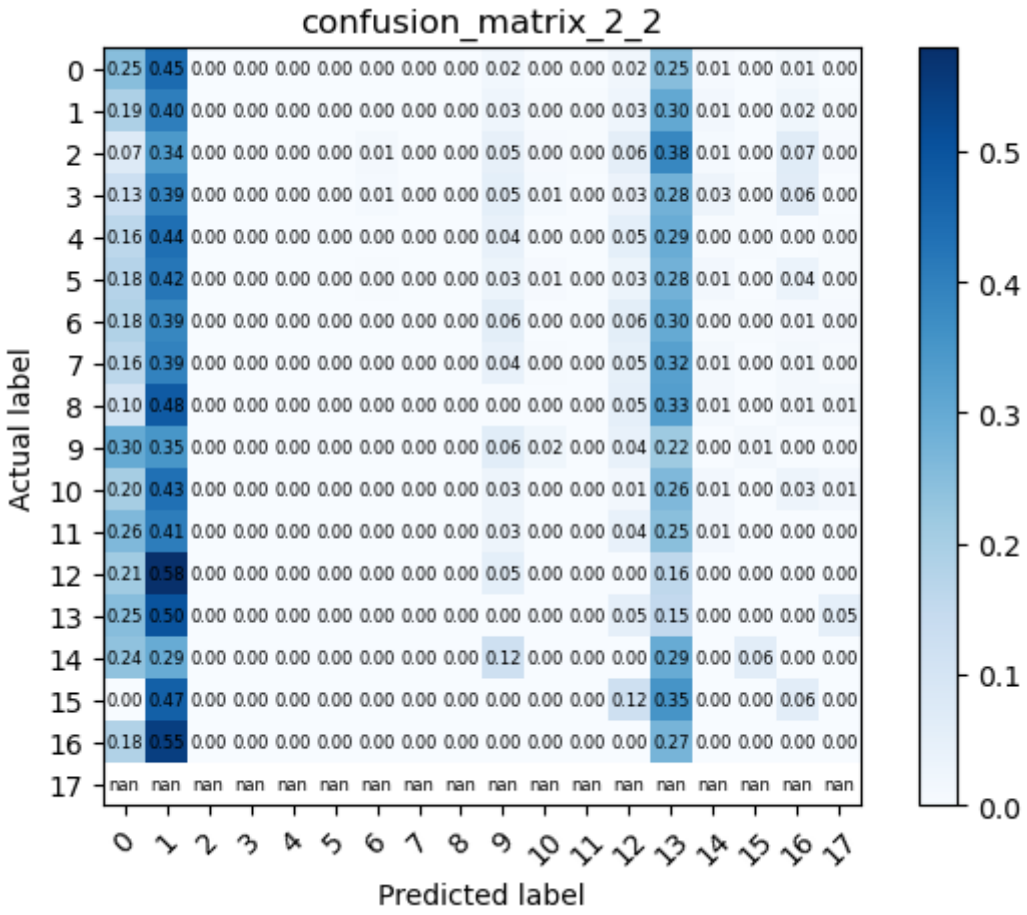
我们怀疑是模型没有训练充分，或者模型本身的架构或性能存在关键漏洞。但通过让模型在训练集上进行预测，我们绘制的混淆矩阵如下：

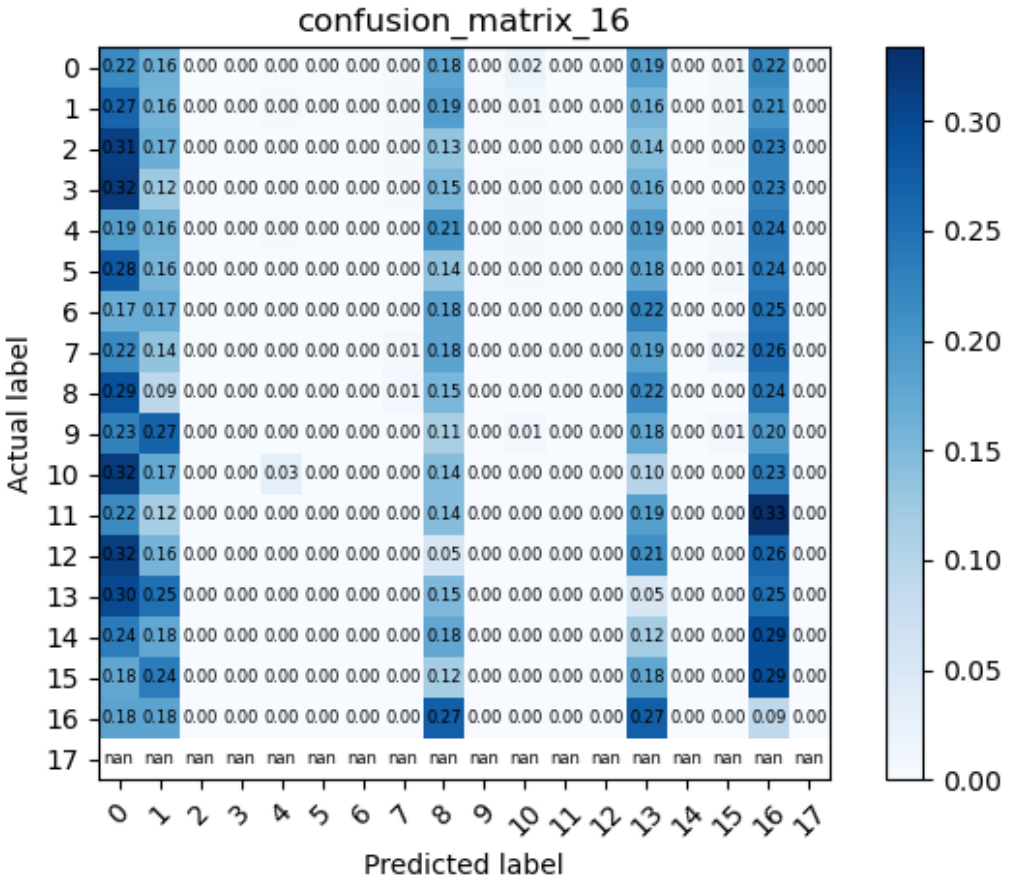


可见模型在训练集上的表现已经相当完美，可以认为其在该训练集上的训练效果已经达到了一定上限。

故我们怀疑是模型在训练过程中发生了过拟合。我们取第第6、10、15个epoch时的模型在验证集上进行预测，得到的预测混淆矩阵如下：


confusion_matrix_1_2_2
Actual label
Predicted label
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0.0 0.1 0.2 0.3 0.4 0.5 0.6


confusion_matrix_2_2
Actual label
Predicted label
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0.0 0.1 0.2 0.3 0.4 0.5



可以看到在模型训练初期，模型更倾向于把所有词语预测为某几个固定的标签，而随着进一步的训练，预测的结果才开始逐渐分散并趋于合理。

而对于为什么第5个epoch之后模型在验证集上的loss函数值反而有所提升，我们认为是因为把所有词语都预测为某几个固定的标签，从loss计算角度得到的数值反而是更低的，但并不真正合理。

综上，我们认为这就是该模型在给定数据集上的训练效果上限。

至于为何模型在训练集和验证集上的差距如此之大，我们认为是数据量过少，或者数据本身的质量不足，导致模型没有学习到更充足的信息。

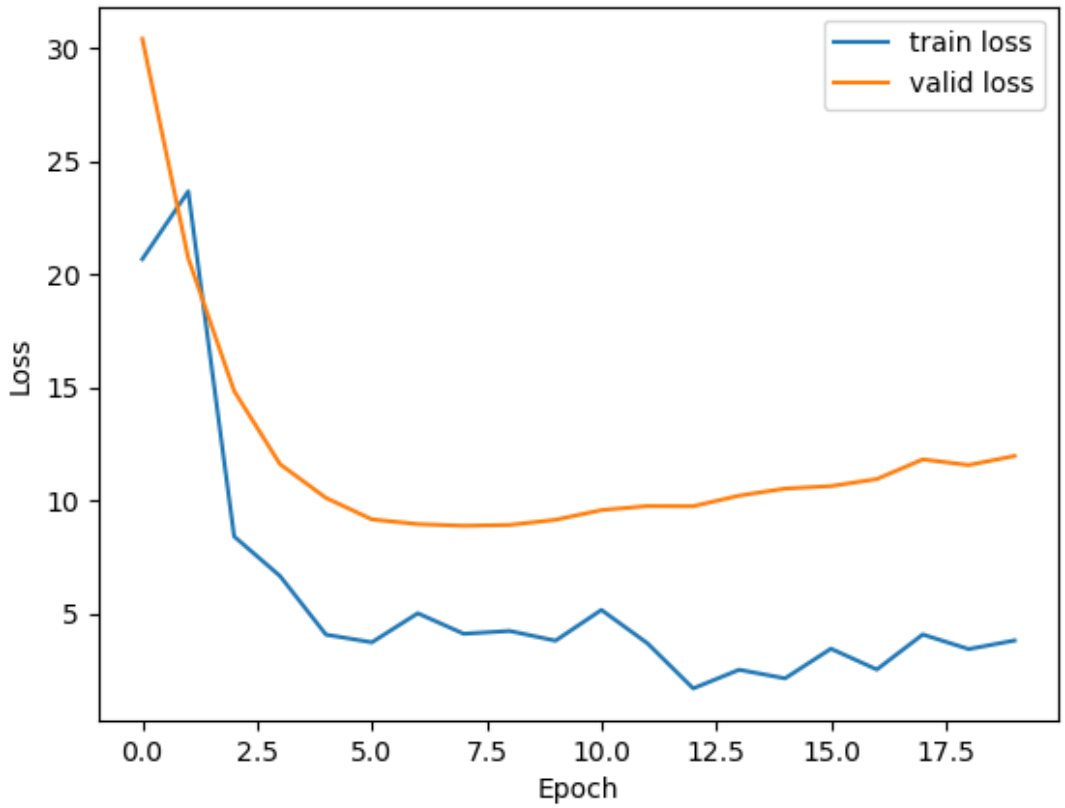
通过对数据的分析，我们发现，其中`o`标签对应的词占到了96%以上，显著多于其他标签，并且其中很多词语的标注是有误的，很多本身具有实义的词被标为了`o`。具有这样特征的数据显然不利于模型的训练学习。

我们考虑对数据进行数据增强或更进一步的预处理，但无奈在不借助带有强先验知识的工具（如预训练好的BERT词嵌入，同义词近义词映射表等）的情况下对文本数据进行处理或增强困难过大，故在本项目中选择不擅自使用额外的工具。

Part 4：探索模型结构对训练效果的影响

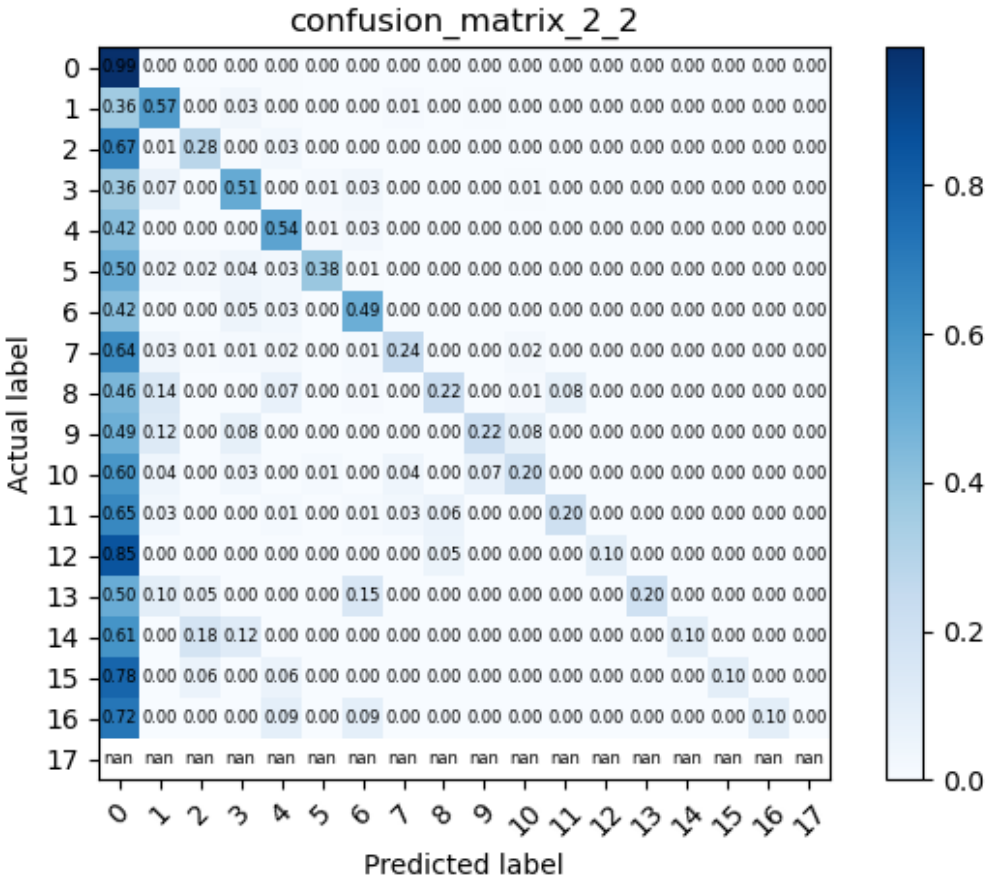
4.1 embed_size 对训练效果的影响

如 Part 3 所示，我们的默认模型的embed_size 为256，这里我们保持其他参数不变，仅将词嵌入维度改为100，对新的模型进行训练。其训练过程中在训练集和验证集上的 loss 曲线图如下：



可见模型在训练集上的 loss 函数在稳步下降并趋于收敛，但在验证集上的 loss 在第5个epoch后就开始上升，与默认模型的训练过程没有显著不同。

下面是该模型在验证集上进行预测的混淆矩阵：

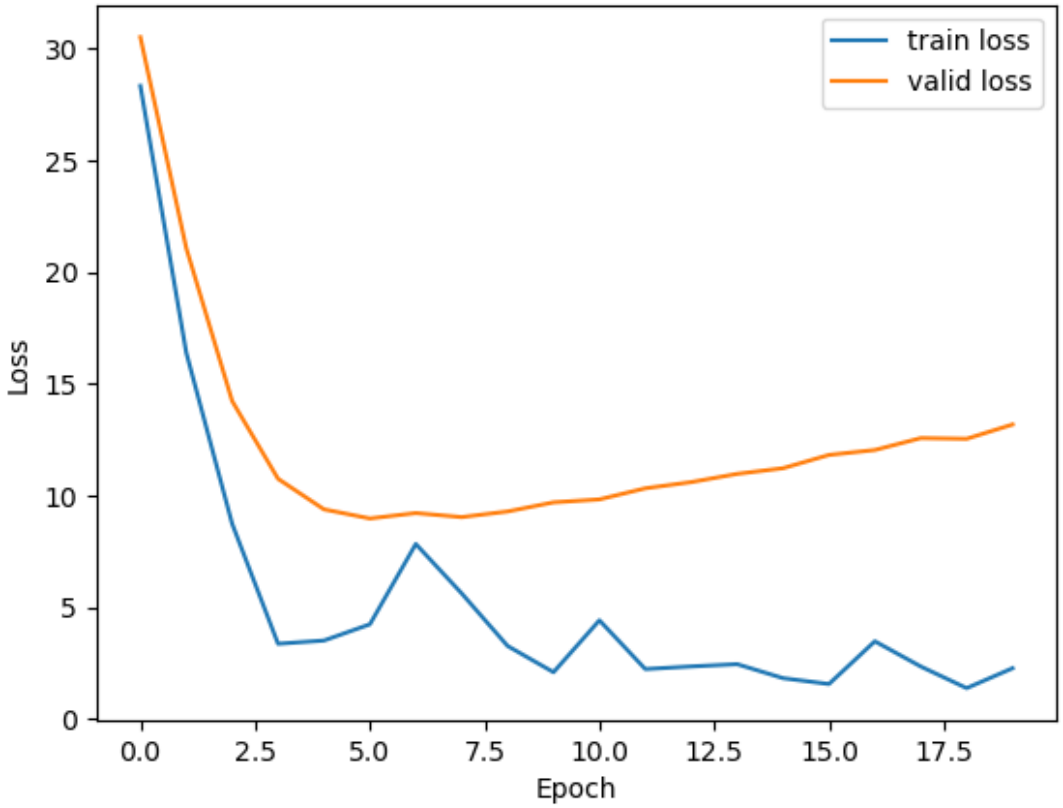


可见其在许多标签上的预测准确率都明显低于默认模型。

可见嵌入维度降低在一定程度上使模型的训练效果发生了下降。我们分析降低 embed_size 会减少减少了模型对输入数据的表示能力。较低的 embed_size 可能无法捕捉到输入数据中的丰富语义和特征信息，导致模型性能下降。

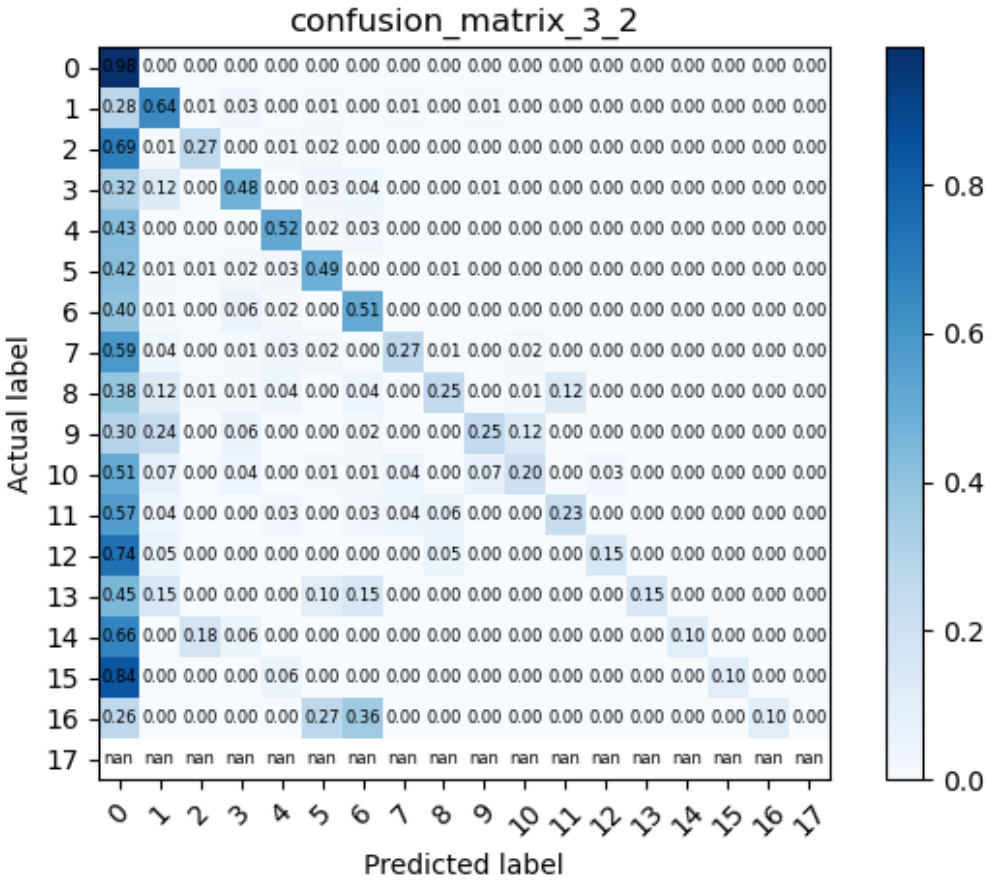
4.2 hidden_size 对训练效果的影响

如 Part 3 所示，我们的默认模型的hidden_size 为256，这里我们保持其他参数不变，仅将词嵌入维度改为512，对新的模型进行训练。其训练过程中在训练集和验证集上的 loss 曲线图如下：



可见模型在训练集上的 loss 函数在稳步下降并趋于收敛，但在验证集上的 loss 在第5个epoch后就开始上升，与默认模型的训练过程没有显著不同。

下面是该模型在验证集上进行预测的混淆矩阵：



可见其在许多标签上的预测准确率都略高于默认模型，但并没有显著不同。

我们分析，提高 hidden_size 会增加模型中隐藏状态的维度，从而增加模型的容量和表示能力。较大的 hidden_size 可能更有能力捕捉输入序列中的复杂依赖关系和语义信息，有助于提升模型性能。并且较大的 hidden_size 可能能够更好地学习和表示输入序列中的长期依赖关系和上下文信息。这对于处理长序列或需要考虑较长上下文的任务尤为重要。但是对于该任务的训练数据质量和数据规模，进一步提升模型容量并不能从本质上改善模型的训练效果。故提高 hidden_size 并不总是会模型的训练效果产生显著的正面影响。