一、数据集-conll2003

> 官网：https://www.clips.uantwerpen.be/conll2003/ner/
>
> 数据集介绍：[Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition]()

1.基本介绍

- 类别

  `We will concentrate on four types of named entities: persons, locations, organizations and names of miscellaneous entities that do not belong to the previous three groups.`

  四个类别：persons, locations, organizations ,miscellaneous entities

- 样例：使用BIO标注法，B-来标记实体的开始部分，I-来标记实体的其它部分，O表示该字或词不组成命名实体

```
    U.N.         NNP  I-NP  I-ORG
    official     NN   I-NP  O
    Ekeus        NNP  I-NP  I-PER
    heads        VBZ  I-VP  O
    for          IN   I-PP  O
    Baghdad      NNP  I-NP  I-LOC
    .            .    O     O
```

  四列分别是单词，词性，语法块，实体标签，在NER任务中，只关心第一列和第四列。实体类别标注采用BIO标注法

  所以标签总共有9类：

```
label2dict = { 'B-PER': 0,
              'I-PER': 1,
              'B-LOC': 2,
              'I-LOC': 3,
              'B-ORG': 4,
              'I-ORG': 5,
              'B-MISC': 6,
              'I-MISC': 7,
              'O': 8}
```

- 数据处理

  数据集中的每一行如样例所示，一行表示一个词的信息，每句话以'.'结尾且使用空行分割，如1-12行的单词组成一句话

```
SOCCER	O
-	O
JAPAN	B-LOC
GET	O
LUCKY	O
WIN	O
,	O
CHINA	B-PER
IN	O
SURPRISE	O
DEFEAT	O
.	O

Nadim	B-PER
Ladki	I-PER

AL-AIN	B-LOC
,	O
United	B-LOC
```

获取每句话的tokens和tags数组，如：

```
[(
    "John lives in New York and works for the European Union".split(),
    "B-PER O O B-LOC I-LOC O O O O B-ORG I-ORG".split()
), (
    ...
)]
```

2.评价指标

- 精度(precision)、召回率(recall)、 $F_1 = \frac{2*precision*recall}{precison+recall}$
- 实体边界和实体类型都要匹配正确

  如：New York 的预测值要为(B-LOC, I-LOC)两者错一个就算错

二、BERT

- 使用BERT后接一个全连接层输出分类结果，之前F1_score值的计算方式是一一对比，是0.5，实际的F1要比这个更低。
  - 遇到的一些问题：一个batchpad的时候输出的label怎么pad，pad成了一个专门的标签，计算loss的时候也加进去。
  - 有的seq超过512了，是直接截断还是另起一段，使用的是后者
  - CrossEntropyLoss: LogSoftmax + NLLloss，当在模型中使用softmax，然后计算nllloss就是负值
- 使用bert的时候输出Bert每一个layer的输出
  - 比如layer是12层，hiddenstate是一个长度为12的列表
  - 每个元素是一个元组：Tuple of `torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.做attention的时候是乘以一个矩阵然后训练，还是直接做attention?
- 最后发现[Bert有专门针对NER的模型]，没做尝试，但记录如下

```python
# 引入模型
from transformers import BertForTokenClassification
# 创建模型
model = BertForTokenClassification.from_pretrained(bert_model_dir, num_labels=self.opt.tag_nums)
out = model(batch_data, token_type_ids=None, attention_mask=batch_masks, labels=labels)

# 示例
```

```
from transformers import BertTokenizer, BertForTokenClassification
import torch
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForTokenClassification.from_pretrained('bert-base-uncased')
inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
labels = torch.tensor([1] * inputs["input_ids"].size(1)).unsqueeze(0)  # Batch size 1
outputs = model(**inputs, labels=labels)
loss, scores = outputs[:2]
```

参数解释：

输入：

- input_ids：训练集，torch.LongTensor类型，shape是[batch_size, sequence_length]
- token_type_ids：可选项，当训练集是两句话时才有的。
- attention_mask：可选项，当使用mask才有，可参考原论文。
- labels：数据标签，torch.LongTensor类型，shape是[batch_size]

输出：

- 如果labels不是None（训练时）：输出的是loss,scores(batch_size, sequence_length, config.num_labels)
- 如果labels是None（评价时）：只输出scores

三、LSTM+CRF

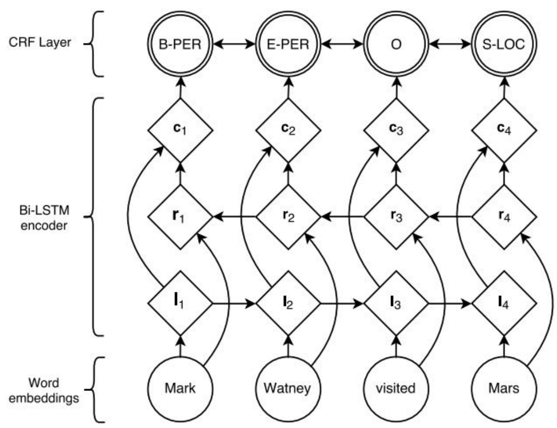


Figure 1: Main architecture of the network. Word embeddings are given to a bidirectional LSTM. $\mathbf{l}_i$ represents the word i and its left context, $\mathbf{r}_i$ represents the word i and its right context. Concatenating these two vectors yields a representation of the word i in its context, $\mathbf{c}_i$.

> CRF的实现参考pytorch官方文档
>
> 推荐大佬博客, 对CRF的解释写的很详细
>
> 推荐两者搭配观看

为什么在Bi-LSTM后加一层CRF呢?
虽然BiLSTM学习到了上下文的信息，但是输出相互之间并没有影响，它只是在每一步挑选一个最大概率值的label输出，最后的标注是各个序列位置标注的拼接，这样只是获得的局部最优解而没有考虑到全局，会导致所获得的标注出现不合规则的情况，而CRF能够从训练集中学到一些约束，比如不可能出现 "O I-"，因为实体名称必须是B-开头等

1.CRF Layer

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：

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

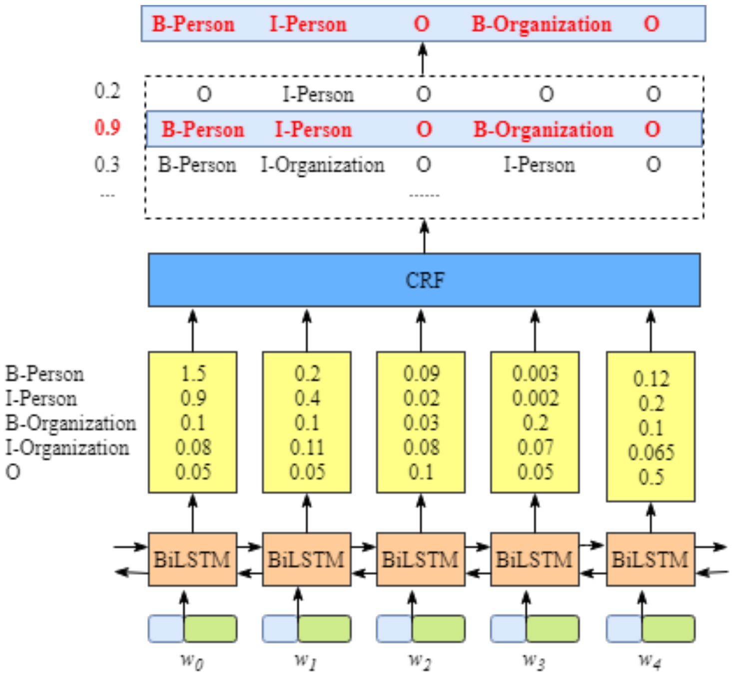
包括状态特征和转移特征，所以标签的score分成两部分，Emisson Score和Transition Score，用于计算loss

则对于输入序列X，预测其标签序列为y的得分如下，$A_{i,j}$表示状态i转移到状态j的score，P_{i,y_i}表示index为i的word，tag为y_i的score

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

1.1 Emission Score

Emission Score用Bi-LSTM的输出[seq_len, tag_size]来表示，例如w_0的tag为B-Person的score为1.5



1.2 Transition Score

定义一个状态转移矩阵T，大小为[tagset_size, tagset_size]，T_{y_i,y_j}表示状态y_i转移到状态y_j的score，这个矩阵就是CRF要学习到的参数.

一般添加两个TAG `START` 和 `END` 用来标志句子的开始和结尾，T示例如下：

	START	B-Person	I-Person	B-Organization	I-Organization	O	END
START	0	0.8	0.007	0.7	0.0008	0.9	0.08
B-Person	0	0.6	0.9	0.2	0.0006	0.6	0.009
I-Person	-1	0.5	0.53	0.55	0.0003	0.85	0.008
B-Organization	0.9	0.5	0.0003	0.25	0.8	0.77	0.006
I-Organization	-0.9	0.45	0.007	0.7	0.65	0.76	0.2
O	0	0.65	0.0007	0.7	0.0008	0.9	0.08
END	0	0	0	0	0	0	0

分析其约束，可得到：

1. 句子中的第一个单词的标记应该是以“B-“ 或者 “O”开头, 并不会是 “I-”形式的标记。（“START” 到 “I-Person or I-Organization” 的转移值非常的小。）
2. 在“B-label1 I-label2 I-label3 I-…”这样形式的标注序列中， label1, label2, label3 … 应该是同种实体的标签。比如，“B-Person I-Person” 是合理有效的标注序列，而 “B-Person I-Organization” 则不是。(“B-Organization” to “I-Person” 转移值为0.0003)
3. 标签序列“O I-label” 是 非法的.实体标签的首个标签应该是“B-” ，而非“I-”(“START” to “I-Person”)

   ……

2.Loss Function

- Loss Funtion 由真实路径得分和所有可能的路径得分组成，真实路径的得分应该是所有可能转移路径中分数最高的
- 假定每一个可能的路径有一个分数值P_i , 那么对于所有 N 条可能的路径的总分数值为 $P_{total} = P_1 + P_2 + \ldots + P_N = e^{S_1} + e^{S_2} + \ldots + e^{S_N}$
- $LossFunction = \frac{P_{RealPath}}{P_1+P_2+\ldots+P_N}$
  随着训练时参数值的不断更新，LossFunction的值应该越来越大，即真实路径的分数值占比应越来越高
  为了便于计算，改写loss，$LogLossFunction = log\frac{P_{RealPath}}{P_1+P_2+\ldots+P_N}$，训练模型时，通常是最小化损失函数，取负可得到

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

- **问题**
  - 如何定义一个路径的得分?
  - 如何计算所有可能路径的总得分?
  - 计算总得分需要列出所有的可能路径吗?

2.1 真实路径得分

这一部分比较好计算，假设真实路径为：“START B-Person I-Person O B-Organization O END”，则操作如下：

- 假设该句子有5个单词组成：w_1, w_2, w_3, w_4, w_5；

- 再额外加两个单词w_0, w_6分别表示该句子的开头和结果；
- S_i由两部分计算得到：$S_{realpath} = EmissionScore + TransitionScore$

发射得分 $EmissionScore = x_{0,START} + x_{1,B-Person} + x_{2,I-Person} + x_{3,O} + x_{4,B-Organization} + x_{5,O} + x_{6,END}$

- $x_{index,label}$,是第index个词被标记为label的得分
- $x_{1,B-Person}$, $x_{2,I-Person}$, $x_{3,O}$, $x_{4,B-Organization}$, $x_{5,O}$都是从BiLSTM的输出得到的
- 对于$x_{0,START}$和$x_{6,END}$,我们可以将他们设为0

转移得分

$TransitionScore = t_{START->B-Person} + t_{B-Person->I-Person} + t_{I-Person->O} + t_{O->B-Organization} + t_{B-Organization->O} + t_{O->END}$

- $t_{label1->label2}$是从label1到label2的转移得分
- 转移得分来自CRF层

```
def _score_sentence(self, feats, tags):  # gives a score of a provided tag squence 根据真实标签计算的
score
    score = torch.zeros(1, device=self.device)
    tags = torch.cat([torch.tensor([self.tag_to_idx['START']], dtype=torch.long,
device=self.device), tags])
    for i, feat in enumerate(feats):
        score += self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]

    score += self.transitions[self.tag_to_idx['STOP'], tags[-1]]
    return score
```

2.2 所有路径的总得分

所有路径的总得分：$P_{total} = P_1 + P_2 + \ldots + P_N = e^{S_1} + e^{S_2} + \ldots + e^{S_N}$，根据lossFunction，计算
$log(e^{S_1} + e^{S_2} + \ldots + e^{S_N})$

最简单的一种方法是：枚举所有可能的路径，然后计算总得分，非常低效

上式是一个累加的过程，其思想和动态规划类似，例如要计算w0→w1→w2的得分，需先计算w0所有路径的总得分，然后计算w0→w1的总得分，再利用上一个得分计算 w0→w1→w2的得分，即为我们所需要的最终得分，在这个过程中要定义两个变量obs和previous，obs表示当前word的信息，previous表示先前步骤的结果，一个简单的示例如下：

基于一个长度为3的句子训练模型：$seq = [w_0, w_1, w_2], LabelSet = l_1, l_2$EmissionScore和TransitionScore分别用x和t来表示, $X : [3, 2], T : [2, 2]$

- w_0

  $obs = [x_{01}, x_{02}]$

  $previous = None$

  该语句中仅有一个单词w_0，我们没有之前的词的结果，所以 previous 为 None。此外，我们也只能获取到第一个单词，其信息 obs=$[x_{01}, x_{02}]$，即发射得分。 那么 w0所有可能路径的总得分即为： $TotalScore(w_0) = log(e^{x_{01}} + e^{x_{02}})$ # 代表w0的两条路径

- $w_0 -> w_1$

  $obs = [x_{11}, x_{12}]$

  $previous = [x_{01}, x_{02}]$

  - 首先将previous扩展为：$previous = \begin{pmatrix} x_{01} & x_{01} \\ x_{02} & x_{02} \end{pmatrix}$
  - 将obs扩展为：$obs = \begin{pmatrix} x_{11} & x_{12} \\ x_{11} & x_{12} \end{pmatrix}$
  - 将 previous obs和转移得分进行相加：$scores = \begin{pmatrix} x_{01} & x_{01} \\ x_{02} & x_{02} \end{pmatrix} + \begin{pmatrix} x_{11} & x_{12} \\ x_{11} & x_{12} \end{pmatrix} + \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix}$

    $scores = \begin{pmatrix} x_{01} + x_{11} + t_{11} & x_{01} + x_{12} + t_{12} \\ x_{02} + x_{11} + t_{21} & x_{02} + x_{12} + t_{22} \end{pmatrix}$

- 更新 previous： $previous = [log(e^{x_{01}+x_{11}+t_{11}} + e^{x_{02}+x_{11}+t_{21}}), log(e^{x_{01}+x_{12}+t_{12}} + e^{x_{02}+x_{12}+t_{22}})]$ # 两个元素分别表示 w_1 为 l_1 和 l_2
- 迭代完成，计算totalscore

$$
\begin{aligned}
&TotalScore(w_0 \to w_1) \\
&= \log(e^{previous[0]} + e^{previous[1]}) \\
&= \log(e^{\log(e^{x_{01}+x_{11}+t_{11}}+e^{x_{02}+x_{11}+t_{21}})} + e^{\log(e^{x_{01}+x_{12}+t_{12}}+e^{x_{02}+x_{12}+t_{22}})}) \\
&= \log(e^{x_{01}+x_{11}+t_{11}} + e^{x_{02}+x_{11}+t_{21}} + e^{x_{01}+x_{12}+t_{12}} + e^{x_{02}+x_{12}+t_{22}})
\end{aligned}
$$

正是我们要计算的 $log(e^{S_1} + e^{S_2} + \ldots + e^{S_N})$ 分别对应 w_0 到 w_1 的四条可能路径的分数

- $w_0 -> w_1 -> w_2$ 同上

```python
# 找出概率最大的路径的分数，使用的是动态规划的思想
def _forward_arg(self, feats):
    init_alphas = torch.full((1, self.tagset_size), -10000., device=self.device)  # [1, tagset_size]

    # 初始的时候start_tag = 0  START到任何tag的值都为0，表示开始传播
    init_alphas[0][self.tag_to_idx['START']] = 0.

    # 赋值给变量方便后向传播，forward_var是之前步骤的score
    forward_var = init_alphas

    # 开始迭代
    for feat in feats:  # feat:[tagset_size] 每个word可能的label，对seq的每个word进行遍历
        alpha_t = []  # The forward tensors at this timestep
        for next_tag in range(self.tagset_size):  # 这一轮迭代：所有其他标签到这个词的概率
            # 状态特征函数得分,feat是emission matrix
            # [1, tagset_size]  表示next_tag为label[i]的emission score
            emit_score = feat[next_tag].view(1, -1).expand(1, self.tagset_size)

            # 状态转移函数得分,其他状态转移到状态next_tag的得分
            # [1, tagset_size]  trans_score[0,i]表示第i个tag转移到next_tag的score
            trans_score = self.transitions[next_tag].view(1, -1)

            # [1, tagset_size] next_tag_var[0,i]表示第i个tag到next_tag的整条路径的分数
            next_tag_var = forward_var + trans_score + emit_score

            # 到next_tag的最好路径的score，for执行完之后是一个长为tagsize的数组
            # 其实这里取得的是最大值，动态规划的思想，不影响最后结果
            alpha_t.append(log_sum_exp(next_tag_var).view(1))

        # [1, tagset_size] forward——var[0][i]当前word到tag[i]的最好的得分
        forward_var = torch.cat(alpha_t).view(1, -1)


    terminal_var = forward_var + self.transitions[self.tag_to_idx['STOP']]
    alpha = log_sum_exp(terminal_var)
    return alpha
```

```python
# 计算loss
def neg_log_likelihood(self, sentence, tags):
    feats = self._get_lstm_features(sentence)  # [seq_len, tag_size]
    forward_score = self._forward_arg(feats)
    gold_score = self._score_sentence(feats, tags)  # 根据两者之间的差值进行反向传播
    return forward_score - gold_score
```

3.Inference

预测的时候使用viterbi算法，同李航《统计学习方法》，与2类似，更新previous不同，改成到每个label的score最大的那个路径
$previous = [max(scores[00], scores[10]), max(scores[01], scores[11])]$

还需要两个额外的数组,α_0, α_1,分别用于存放最大分数，即最大分数对应的路径

```python
def _viterbi_decode(self, feats):  # feats:[seq_len, tagset_size]
    backpointers = []

    # 初始化
    init_vvars = torch.full((1, self.tagset_size), -10000, device=self.device)
    init_vvars[0][self.tag_to_idx['START']] = 0

    # 步骤i的forward_var保留步骤i-1的viterbi变量
    forward_var = init_vvars
    for feat in feats:  # feat:[1, tagset size] word的每一个可能的label
        bptrs_t = []  # holds the breakpointers for this step,即当前到所有tag的最大值索引
        viterbivars_t = []  # holds the viterbi variables for this step, 当前word到所有tag的最大分数

        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)  # 选最大
            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_idx['STOP']]
    best_tag_id = argmax(terminal_var)
    path_score = terminal_var[0][best_tag_id]  # 开始记录分数

    # Follow the back pointers to decode the best path  根据\delta找最大路径
    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
    start = best_path.pop()
    assert start == self.tag_to_idx['START']
    best_path.reverse()

    return path_score, best_path
```

四、问题记录

- 关于device的问题

  使用model.to('cusa')只能把model的init中的self的属性、函数放到cuda上，其他函数则不能，需要手工放上去

  一些需要计算的中间变量也需要认为放到cuda上

- 一个因为device无故停止的问题

```
self.hidden = self.init_hidden()  # 在init中定义的hidden
def init_hidden(self):
     return (torch.randn(2, 1, self.hidden_dim // 2, device=self.device), torch.randn(2, 1,
self.hidden_dim // 2, device=self.device))  # 定义lstm的hidden状态

lstm_out, self.hidden = self.lstm(embeds, self.hidden)  # 使用LSTM的时候
```

  一开始init_hidden的返回值没有设置device，因为init_hidden不是init中定义的函数，所以其返回值没有定义在cuda上

  colab用gpu跑的时候，在lstm这里就卡住了，而且没报任何错误

- 至今尚未解决的错误

  使用LSTM+CRF，loss.backward（）时错误

```
Traceback (most recent call last):
  File "main.py", line 22, in <module>
    trainer.train()
  File "/content/gdrive/My Drive/NER/bilstm_crf/train.py", line 48, in train
    train_loss, train_precison, train_recall, train_F1 = self.train_epoch()
  File "/content/gdrive/My Drive/NER/bilstm_crf/train.py", line 88, in train_epoch
    loss.backward()
  File "/usr/local/lib/python3.6/dist-packages/torch/tensor.py", line 198, in backward
    torch.autograd.backward(self, gradient, retain_graph, create_graph)
  File "/usr/local/lib/python3.6/dist-packages/torch/autograd/__init__.py", line 100, in backward
    allow_unreachable=True)  # allow_unreachable flag
RuntimeError: Function SubBackward0 returned an invalid gradient at index 0 - expected type TensorOptions(dtype=f
frame #0: c10::Error::Error(c10::SourceLocation, std::string const&) + 0x46 (0x7f9519ae5536 in /usr/local/lib/pyt
frame #1: <unknown function> + 0x2d83d24 (0x7f955477ad24 in /usr/local/lib/python3.6/dist-packages/torch/lib/libt
frame #2: torch::autograd::Engine::evaluate_function(std::shared_ptr<torch::autograd::GraphTask>&, torch::autogra
frame #3: torch::autograd::Engine::thread_main(std::shared_ptr<torch::autograd::GraphTask> const&, bool) + 0x3d2
frame #4: torch::autograd::Engine::thread_init(int) + 0x39 (0x7f9554776e59 in /usr/local/lib/python3.6/dist-packa
frame #5: torch::autograd::python::PythonEngine::thread_init(int) + 0x38 (0x7f95610be488 in /usr/local/lib/python
frame #6: <unknown function> + 0xbd6df (0x7f956363b6df in /usr/lib/x86_64-linux-gnu/libstdc++.so.6)
frame #7: <unknown function> + 0x76db (0x7f956471d6db in /lib/x86_64-linux-gnu/libpthread.so.0)
frame #8: clone + 0x3f (0x7f9564a56a3f in /lib/x86_64-linux-gnu/libc.so.6)
```

  查阅资料，是loss和需要回传的模型参数没在一个device上，(但明明在一个上面啊。。)

  然后就放弃了。。

- 一个小问题：LSTM内的dropout，只有num_layer>1时dropout才有效，因为是在层与层之间加dropout所以也好理解，最后一层没有dropout