

Project2 HMM、CRF、 BiLSTM+CRF

计算机科学技术学院 周训哲 20307110315

2023 年 5 月 22 日

Part1: HMM 实现命名实体识别 (NER) 任务

一、HMM 简介

隐马尔可夫模型 (Hidden Markov Model, HMM) 是一种用于建立时序数据模型的统计模型, 可以应用于自然语言处理、语音识别、生物信息学、信号处理等多个领域。HMM 可以看做是一个双层的马尔可夫模型, 其中状态的转移是一个马尔可夫链, 而观测值的生成是一个以状态为条件概率的生成过程。

在 HMM 中, 状态是不可见的隐变量, 而观测值是可见的变量。我们只能观测到观测值序列, 而不知道状态序列。HMM 中的每个状态都有一个对应的概率分布, 表示在该状态下生成观测值的概率。我们可以通过观测值序列来估计模型的参数, 从而对未知序列进行预测或分类等任务。

具体实现过程中, 包含了三种概率, 分别是初始概率、转移概率、发射概率。其中, 初始概率主要负责统计所有状态的出现概率, 实现过程中只需要统计每一种状态的出现次数即可; 转移概率负责统计相邻两个状态之间的转移的概率, 实现过程中统计不同状态之间的转移的次数即可; 发射概率则负责统计状态到观测结果之间的概率关系, 则需要统计某种观测结果对应的状态并计算概率。

在统计相应的概率的出现次数之后需要计算概率, 此过程如果是简单的频率估计则需要防止数据下溢, 对数据需要进行归一化, 采用对数处理

方法。

在估计过程中需要使用 Viterbi 解码的方法进行状态估计。具体算法为：首先对初始状态利用初始概率与发射概率进行相乘，初步估计初始的状态，然后对之后的状态利用发射概率和转移概率计算上一状态转移到当前的状态的概率，并计算估计状态对应的发射概率，用于估计当前的状态。最后在反向过程中，利用估计的概率选取上一状态的最大概率对应的状态，估计对应的当前的状态，得到最后的估计结果。

$$v_{1,j} = P(O_1|s_j)P(s_j|s_{start}) \quad (1)$$

$$v_{t,j} = \max_i \{v_{t-1,i}P(s_j|s_i)P(O_t|s_j)\} \quad (2)$$

$$\delta_t(j) = \arg \max_i \{v_{t-1,i}P(s_j|s_i)P(O_t|s_j)\} \quad (3)$$

$$s_t = \delta_t(s_t) \quad (4)$$

二、代码架构

代码主要划分为 model 用于设计 hmm 模型，main 用于计算对应的状态估计结果，并输出评分，utils 用于设计相应的函数。model 中又划分为概率估计和 Viterbi 解码两个部分。

概率估计

```

1 class HMM:
2     def __init__(self, states, observations):
3         self.states = states
4         self.observations = observations
5         self.n_states = len(states)
6         self.n_observations = len(observations)
7         self.initial_prob = np.zeros(self.n_states)
8         self.transition_prob = np.zeros((self.n_states, self.n_states))
9         self.emission_prob = np.zeros((self.n_states, self.n_observations))

```

```

10         self.min = 1e-100
11
12     def fit(self, data):
13         # 计算初始概率
14         for i in range(len(data)):
15             state = data[i][1]
16             j = self.states.index(state)
17             self.initial_prob[j] += 1
18         # 计算状态转移概率
19         for i in range(len(data) - 1):
20             state_i = data[i][1]
21             state_j = data[i + 1][1]
22             j = self.states.index(state_i)
23             k = self.states.index(state_j)
24             self.transition_prob[j, k] += 1
25         # 计算发射概率
26         for i in range(len(data)):
27             state = data[i][1]
28             observation = data[i][0]
29             j = self.states.index(state)
30             k = self.observations.index(observation)
31             self.emission_prob[j, k] += 1
32
33         # 为矩阵中所有是 0 的元素赋值为一个接近 0 的数
34         self.initial_prob[self.initial_prob == 0] = self.min
35         # 防止数据下溢，对数据进行对数归一化
36         self.initial_prob = np.log(self.initial_prob) -
37             np.log(np.sum(self.initial_prob))
38         self.transition_prob[self.transition_prob == 0] = self.min
39         self.transition_prob = np.log(self.transition_prob) -
40             np.log(np.sum(self.transition_prob, axis=1, keepdims=True))

```

```

41         self.emission_prob[self.emission_prob == 0] = self.min
42         self.emission_prob = np.log(self.emission_prob) -
43         np.log(np.sum(self.emission_prob, axis=1, keepdims=True))

```

Viterbi 解码

```

1     def predict(self, observations):
2         # Viterbi 算法
3         n_steps = len(observations)
4         # 前向算法
5         delta = np.zeros((n_steps, self.n_states))
6         psi = np.zeros((n_steps, self.n_states), dtype=np.int)
7         # 计算初始时刻所有发射结果的概率
8         if observations[0] in self.observations:
9             delta[0] = self.initial_prob + self.emission_prob[:, self.observations[0]]
10        else:
11            delta[0] = np.log(self.min)
12        for t in range(1, n_steps):
13            for j in range(self.n_states):
14                # 当前是 t 时刻, temp 表示从上一时刻转移到 j 的概率, 即当
15                temp = delta[t - 1] + self.transition_prob[:, j]
16                psi[t, j] = np.argmax(temp)
17                # 当前是 j 状态, 发射观测结果的概率
18                if observations[t] in self.observations:
19                    delta[t, j] = temp[psi[t, j]] + self.emission_prob[j, self.observations[t]]
20                else:
21                    delta[t, j] = np.log(self.min)
22        # 后向算法
23        path = np.zeros(n_steps, dtype=np.int)
24        path[-1] = np.argmax(delta[-1])
25        for t in range(n_steps - 2, -1, -1):

```

```

26         path[t] = psi[t + 1, path[t + 1]]
27     return [self.states[i] for i in path]

```

三、实验结果

全部数据一起进行预测 首先进行的尝试是直接做除法计算各种概率，然后输入所有的数据一起进行预测，估计对应的状态。

micro avg	0.0359	0.2143	0.0615	8603
macro avg	0.1295	0.1252	0.0090	8603
weighted avg	0.1636	0.2143	0.0153	8603

图 1: 英文全部数据作为输入

micro avg	0.7089	0.0066	0.0132	8437
macro avg	0.2665	0.0061	0.0113	8437
weighted avg	0.6903	0.0066	0.0131	8437

图 2: 中文全部数据作为输入

实现结果非常不理想，原因是因为对于所有数据直接进行预测，会导致相邻的句子之间的估计会出现问题，同时没有使用 \log 估计和参数为 0 的时候的处理，会出现 nan 以及数据下溢的情况。

按一句话为单位进行批预测 然后按一句话为单位进行批预测，估计对应的状态。

micro avg	0.1542	0.5576	0.2415	8603
macro avg	0.7591	0.5011	0.5192	8603
weighted avg	0.6958	0.5576	0.4900	8603

图 3: 英文批数据作为输入

micro avg	0.8534	0.7253	0.7841	8437
macro avg	0.5026	0.4692	0.4760	8437
weighted avg	0.8587	0.7253	0.7855	8437

图 4: 中文批数据作为输入

实现结果仍然不理想，原因是没有使用 \log 估计和参数为 0 的时候的处理，仍然会出现 nan 以及数据下溢的情况。

使用对数对概率进行估计 由于数据集中部分数据出现次数差异太大，可以使用对数进行处理使得数据概率估计更加集中。

micro avg	0.5354	0.7735	0.6328	8603
macro avg	0.6971	0.7447	0.6890	8603
weighted avg	0.6668	0.7735	0.6789	8603

图 5: 英文数据对数概率估计

使用批训练的方法

micro avg	0.8483	0.8706	0.8593	8437
macro avg	0.5601	0.6521	0.5881	8437
weighted avg	0.8542	0.8706	0.8615	8437

图 6: 中文数据对数概率估计

micro avg	0.9060	0.6365	0.7477	8603
macro avg	0.8966	0.6433	0.7429	8603
weighted avg	0.9115	0.6365	0.7423	8603

图 7: 英文数据批训练

micro avg	0.8733	0.9029	0.8879	8437
macro avg	0.6118	0.7152	0.6484	8437
weighted avg	0.8786	0.9029	0.8898	8437

图 8: 中文数据批训练

实现结果基本符合预期。

四、对 HMM 模型的理解

HMM 模型作为最基础的序列预测模型，使用隐马模型对显性和隐性状态进行了估计，对于状态的转移估计也能较好的拟合实际情况，使用最基础的参数估计和预测的方法符合基础的统计学思想。是一个很好的学习人工智能的模型。

Part2: CRF 实现命名实体识别 (NER) 任务

一、CRF 简介

CRF 全称为条件随机场 (Conditional Random Field)，是一种用于序列标注 (Sequence Labeling) 问题的概率图模型。在自然语言处理、计算机视觉和生物信息学等领域得到了广泛应用。

CRF 是一种无向图模型，可以对输入序列和输出序列的联合分布建模。它通过定义一组特征函数，将输入序列和输出序列映射到实数域上，然后通过特征函数值的加权和进行归一化，得到输出序列的概率分布。

与隐马尔可夫模型 (HMM) 相比，CRF 不仅考虑了当前观察到的状态，还考虑了整个输入序列的上下文信息，因此在处理自然语言处理任务时效果更好。

CRF 在 HMM 得到的数据上还进行了一下操作，认为设定特征，并对特征和发射频率进行统计。对于训练数据中出现的在特征对应的数据上进行 +1，而对于 hmm 中出现的数据则进行 -1 操作，目的是为了训练模型使得预测结果更加符合实际情况。并进行对数似然运算，求得相应概率的最大似然。

$$L(\mathbf{w}) = \sum_{k=1}^K \log P(\mathbf{y}^{(k)} | \mathbf{x}^{(k)}) - \frac{\lambda}{2} \|\mathbf{w}\|^2 \quad (5)$$

然后利用之前计算的每一种特征的得分和发射概率进行组合得到分数再次进行维特比解码就可以得到新的预测结果了。

$$\mathbf{w}^{new} = \mathbf{w}^{old} + \eta \left(\sum_{k=1}^K \sum_{i=1}^n \sum_{j=1}^m f_j(y_i^{(k)}, y_{i-1}^{(k)}, \mathbf{x}^{(k)}, i) - \mathbb{E}_{P_{\mathbf{w}^{old}}} \left[\sum_{i=1}^n \sum_{j=1}^m f_j(y_i, y_{i-1}, \mathbf{x}, i) \right] - \lambda \mathbf{w}^{old} \right) \quad (6)$$

二、代码架构

代码主要有两个部分，其中一个部分是调用 python 已有的函数包，直接实现 crf 的命名实体识别任务，然后另一部分为手写实现，虽然结果完全不符合预期。

调用库实现的主体部分

```

1 language = 'Chinese'
2 train_path = language + '\\train.txt'
3 validation_path = language + '\\validation.txt'
4 train_states, train_observation, train_data = get_divided_data(train_path)
5 validation_states, validation_observation,
6     validation_data = get_divided_data(validation_path)
7
8 # 定义标签序列
9 train_states = [l + ['0'] * (len(s) - len(l))
10     for s, l in zip(train_observation, train_states)]
11 validation_states = [l + ['0'] * (len(s) - len(l))
12     for s, l in zip(validation_observation, validation_states)]
13
14 # 将训练数据和测试数据转换为特征向量序列
15 x_train = [sent2features(s) for s in train_observation]
16 y_train = train_states
17 x_test = [sent2features(s) for s in validation_observation]
18 y_test = validation_states
19

```

```

20 # 定义 CRF 模型
21 crf = sklearn_crfsuite.CRF(
22     algorithm='lbfgs',
23     c1=0.1,
24     c2=0.1,
25     max_iterations=100,
26     all_possible_transitions=False
27 )
28 # 训练模型
29 crf.fit(x_train, y_train)
30 # 预测标签序列
31 test_states = crf.predict(x_test)
32
33 with open(language + '\\result.txt', 'w', encoding='UTF-8') as f:
34     for observation, state in zip(validation_observation, test_states):
35         for o, s in zip(observation, state):
36             if o != '\n':
37                 f.write(str(o) + ' ' + str(s) + '\n')
38             else:
39                 f.write('\n')
40 f.close()
41
42 check(language, validation_path, language + "\\result.txt")

```

特征函数的实现

```

1 def feat1(sent, i):
2     # 0.9397
3     word = sent[i]
4     features = {
5         'bias': 1.0,

```

```
6         'word.lower()': word.lower(),
7         'word[-3:]': word[-3:],
8         'word[-2:]': word[-2:],
9         'word.isupper()': word.isupper(),
10        'word.istitle()': word.istitle(),
11        'word.isdigit()': word.isdigit(),
12    }
13    if i > 0:
14        # 前一个字
15        prev_word = sent[i - 1]
16        features.update({
17            '-1:word.lower()': prev_word.lower(),
18            '-1:word.istitle()': prev_word.istitle(),
19            '-1:word.isupper()': prev_word.isupper(),
20        })
21    else:
22        features['BOS'] = True
23    if i < len(sent) - 1:
24        # 后一个字
25        next_word = sent[i + 1]
26        features.update({
27            '+1:word.lower()': next_word.lower(),
28            '+1:word.istitle()': next_word.istitle(),
29            '+1:word.isupper()': next_word.isupper(),
30        })
31    else:
32        features['EOS'] = True
33    return features
```

三、实验结果

对于 CRF 而言，测试结果与特征的选取密切相关，在本次实验中主要使用了两种特征。

特征 1 第一种特征主要考虑的就只有前后两个的输出的观测值，对于中英文来说效果都挺好。

micro avg	0.9045	0.8671	0.8854	8603
macro avg	0.9032	0.8419	0.8704	8603
weighted avg	0.9045	0.8671	0.8849	8603

图 9: 英文数据特征 1

micro avg	0.9325	0.9469	0.9397	8437
macro avg	0.7174	0.7244	0.7208	8437
weighted avg	0.9325	0.9469	0.9396	8437

图 10: 中文数据特征 1

特征 2 由于采用的特征较为简单，之后改进使用更多的特征函数。

micro avg	0.6924	0.6628	0.6773	8603
macro avg	0.6791	0.6378	0.6559	8603
weighted avg	0.6914	0.6628	0.6758	8603

图 11: 英文数据特征 2

micro avg	0.9476	0.9563	0.9519	8437
macro avg	0.7188	0.7343	0.7254	8437
weighted avg	0.9481	0.9563	0.9520	8437

图 12: 中文数据特征 2

在增加特征之后，可以明显的发现英文的效果反而变得更差了，而中文的效果更好了，这大概率是因为英文语料库中对于特征的出现次数不同于中文，中文使用的是字的组合可以合成词语，而英文单词之间合成的是短语，很明显英文短语出现的次数会远小于中文词语的出现次数。

四、手写实现 CRF (bonus)

这个部分是所有实验中最耗时间也是完成效果最差的部分，自认为是没有完全理解 CRF 的工作原理。

具体实现

根据论文《Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms》统计相应的真实值与 hmm 预测值之间的预测误差和相应的发射概率误差。并且将不同特征得到的结果进行加和，但是由于时间和数学能力有限，暂时没有实现文章中所说的最大似然估计得到相应的特征函数。

对于最后的状态的估计也是采用和文章所说以及老师上课所说相同的方法——将特征函数对应的结果进行加和，然后顺序估计状态的最大可能的结果，最后进行维特比解码。

具体实现过程中对于 unigram 统计当前状态以及特征函数中的观测值的次数，如果真实值与 hmm 预测值不同则真实值对应的位置 +1，hmm 对应的位置-1。而计算特征函数的过程中，则是直接将统计结果作为特征函数值。这样做的原因，一方面是因为如果使用像 hmm 一样的方法，采用频率估计概率，则会出现负数概率，取对数之后会变成 nan，另一方面则是在后续的维特比解码过程中，由于其单位并非和概率相同，在和其他概率进行

计算的过程中则会导致量级不一。所以直接使用统计结果作为特征函数的结果，后续操作中只需将特征函数，即统计结果加和即可。

在维特比解码过程中使用的方法和老师上课讲的方法相同。就是对于每一个观测结果，都统计对应所有的特征函数的值的总和，取最高的值进行输出为当前的状态，并使用类似动态规划的方法将结果暂存，最后反向传播的过程中统计所有状态的结果并输出。

在尝试过程试过很多种方法的排列组合，比如尝试使用初始概率以及发射概率统计无法使用特征函数的边界值；使用对数将统计结果进行处理，防止数据下溢；特征函数的值与发射概率相乘表示当前的状态等。还有一些其他的统计和测试方法未经列出。总之，在自己手动根据论文实现 CRF 的过程中，不仅使用了大量的时间，最终的结果还不好，根本没有得到一个有效的输出。

最后需要说明的是，特征函数的选择就是助教给出的模板。最终测试结果不佳大概率是因为没有完全理解 CRF 的工作原理。

代码实现

在代码实现过程中尝试了多种方法，代码只列出结果最好的代码，虽然实现过程并非完全按照老师及论文中提及的方法，而是经过改进的方法，作者也不知道为什么这种方法的结果却是最好的，即便结果也不算好。

预测过程

```
1 def predict(self, observations):
2     # Viterbi 算法
3     n_steps = len(observations)
4     delta = np.zeros((n_steps, self.n_states))
5
6     # DP 算法，只跟当前状态有关
7     for t in range(n_steps):
8 for feature in self.features:
9     gram_size = feature.gram_size
```

```

10  if gram_size == 1:
11      off = feature.off[0]
12      for j in range(self.n_states):
13          if max(0, -off) <= t < min(n_steps - off, n_steps):
14              # 当前实际状态为 self.states.index(hmm_states[t])
15              # 计算如果当前状态为 j, 对应的 ngram 的值
16              if observations[t + off] in self.observations
17              and observations[t] in self.observations:
18                  delta[t, j] += feature.unigram_prob[j,
19                      self.observations.index(observations[t + off])] \
20                      * self.emission_prob[j, self.observations.index(observat
21              else:
22                  if observations[t] in self.observations:
23                      delta[t, j] += self.initial_prob[j] \
24                      * self.emission_prob[j, self.observations.index(observat
25  if gram_size == 2:
26      off1, off2 = feature.off[0], feature.off[1]
27      for j in range(self.n_states):
28          if max(0, -max(off1, off2)) <= t < min(n_steps - max(off1, of
29              # 当前实际状态为 self.states.index(hmm_states[t])
30              # 计算如果当前状态为 j, 对应的 ngram 的值
31              if observations[t + off1] in self.observations \
32              and observations[t + off2] in self.observations \
33              and observations[t] in self.observations:
34                  delta[t, j] += feature.bigram_prob[j,
35                      self.observations.index(observations[t + off1]),
36                      self.observations.index(observations[t + off2])] \
37                      * self.emission_prob[j, self.observations.index(
38              else:
39                  if observations[t] in self.observations:
40                      delta[t, j] += self.initial_prob[j] \

```

```

41         * self.emission_prob[j, self.observations.index(observat
42     # 更新当前状态
43     path = np.zeros(n_steps, dtype=np.int)
44     for t in range(n_steps):
45         path[t] = np.argmax(delta[t])
46     return [self.states[i] for i in path]

```

获取数据 在获取数据过程我自定义了一个 Feature 类用来存储不同的特征

```

1 def load_feature(template_file, n_states, n_observations):
2     with open(template_file, 'r') as f:
3         crf_template = f.readlines()
4     features = []
5     for line in crf_template:
6         line = line.strip()
7         if line.startswith('#'):
8             # 忽略注释行
9             continue
10        elif line.startswith('U'):
11            # 处理 Unigram 特征
12            parts = line.split(':')
13            expression = parts[1]
14            matches = re.findall(r'\-?\d+', expression)
15            feat = Feature(n_states, n_observations,
16                          int(matches[0]))
17            features.append(feat)
18        elif line.startswith('B'):
19            # 处理 Bigram 特征
20            parts = line.split(':')
21            expression = parts[1]

```



```

22             matches = re.findall(r'\-?\d+', expression)
23             feat = Feature(n_states, n_observations,
24                             int(matches[0]), int(matches[2]))
25             features.append(feat)
26         else:
27             # 忽略其他行
28             continue
29
30     return features
31
32
33 class Feature:
34     def __init__(self, n_states, n_observations, off1, off2=None):
35         self.unigram_prob = np.zeros((n_states, n_observations))
36         self.bigram_prob = np.zeros((n_states, n_observations,
37                                     n_observations), dtype='int8')
38         self.gram_size = 0
39         self.off = [0, 0]
40         self.off[0] = off1
41         self.gram_size += 1
42         if off2 is not None:
43             self.off[1] = off2
44             self.gram_size += 1

```

实验结果

可能是由于对于 CRF 的特征提取有误，导致最终的结果很差。同时，由于英文单词过多，在测试英文语料的过程中，使用自己构造的 Feature 类出现了 n-gram 矩阵的空间爆炸的问题，无法申请这么多的内存，即便改用其他的数据类型也无法处理，所以该实验中并未测试英文语料。

micro avg	0.8589	0.8009	0.8289	9890
macro avg	0.6109	0.5045	0.5163	9890
weighted avg	0.8611	0.8009	0.8207	9890

图 13: 手写 CRF 中文语料结果

五、对 CRF 模型的理解

CRF 模型算是对 HMM 模型的改进，相比于之前的模型，增加了非线性的结构，不仅可以关注相邻两个状态之间的关系，还可以长距离的关注更多信息，将语料库中词与词组固定搭配的信息以及句子开头结尾的信息引入，可以更加准确的预测状态。

Part3: BiLSTM+CRF 实现命名实体识别 (NER) 任务

一、模型简介

BiLSTM，全称为双向长短期记忆网络 (Bidirectional Long Short-Term Memory)，是一种常用于自然语言处理 (NLP) 和序列建模任务的神经网络模型。BiLSTM 是对传统的 LSTM 模型的扩展，旨在解决 LSTM 在处理序列数据时可能遗忘了之前和之后的上下文信息的问题。

LSTM (长短期记忆网络) 是一种递归神经网络 (RNN)，它具有记忆单元来捕捉和存储长期依赖关系。然而，传统的 LSTM 只能在一个方向 (从前到后或从后到前) 处理序列数据，这可能导致在某些任务中丢失关键的上下文信息。

LSTM

首先对于基础的 LSTM 来说，是对于 RNN (循环神经网络) 的一个改进。由于简单的 RNN 使用的只有前一个时刻的模型的值与当前时刻的输

入和输出，所以会出现长程依赖问题，而在 LSTM 中使用了长短程的记忆门控机制可以有效的处理这样的问题。

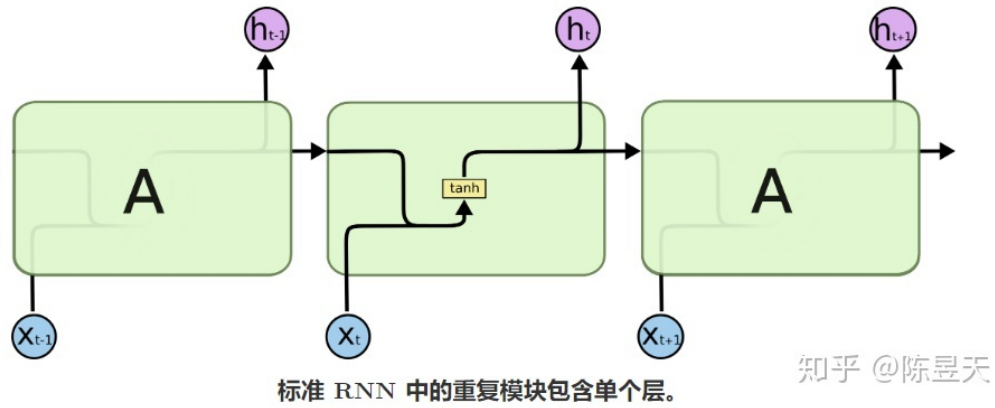


图 14: RNN 的结构示意图

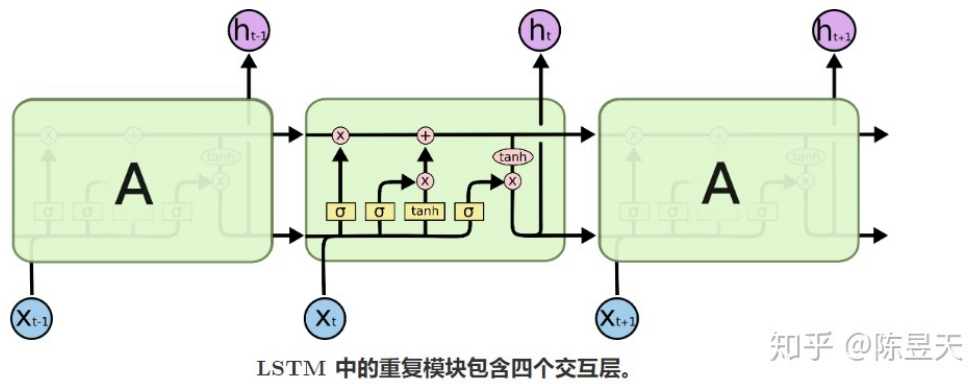


图 15: LSTM 的结构示意图

BiLSTM

BiLSTM 通过在同一层中同时运行两个 LSTM 网络，一个从前向后处理序列，另一个从后向前处理序列，以捕获完整的上下文信息。两个方向的

LSTM 网络分别称为前向 LSTM 和后向 LSTM。这样，BiLSTM 可以同时利用过去和未来的信息来预测当前的输出。

在 BiLSTM 中，前向 LSTM 和后向 LSTM 的隐藏状态通过拼接或相加的方式来传递给后续层或任务。这种结构允许模型在任何时刻都能够访问序列中的全局上下文，从而更好地捕捉序列中的模式和依赖关系。

BiLSTM 广泛应用于许多 NLP 任务，例如命名实体识别、情感分析、文本分类、机器翻译等。通过利用前向和后向信息，BiLSTM 能够更好地理解和表示输入序列中的语义和结构。

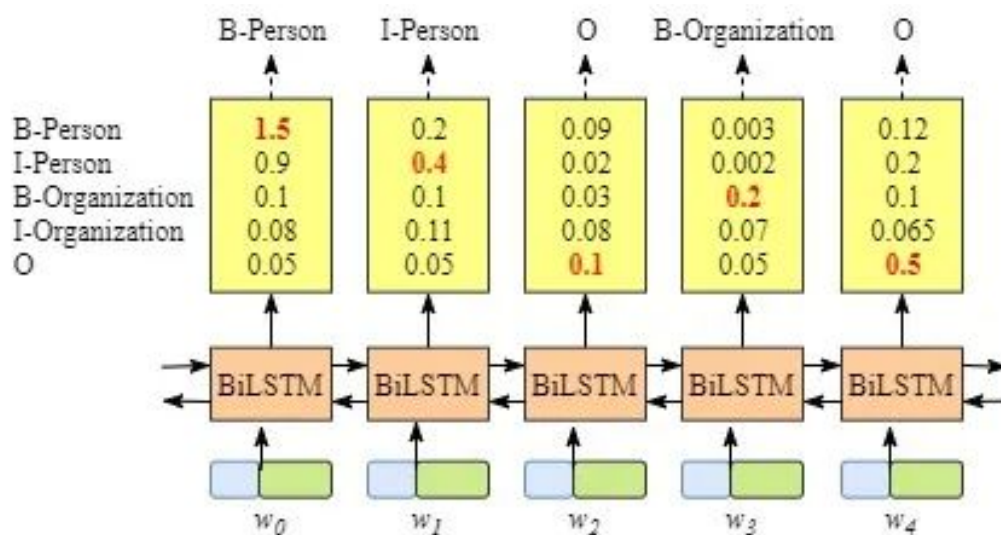


图 16: 单独使用 BiLSTM 进行命名实体识别

如果单独使用 BiLSTM 进行命名实体预测，其实就是简单的使用神经网络对语料的状态进行拟合，只不过在每一个 LSTM 模块中可以记忆前一个模块中的信息，有点类似与 HMM 的方法。而使用上 CRF 的特征识别和维特比解码效果会更加良好。

BiLSTM+CRF

当加入 CRF 层之后，可以对 BiLSTM 输出的发射概率加上转移概率进行修正，使得结果更加符合实际的语言的结构，比如 M 不可能出现在一个句子的开头等约束。

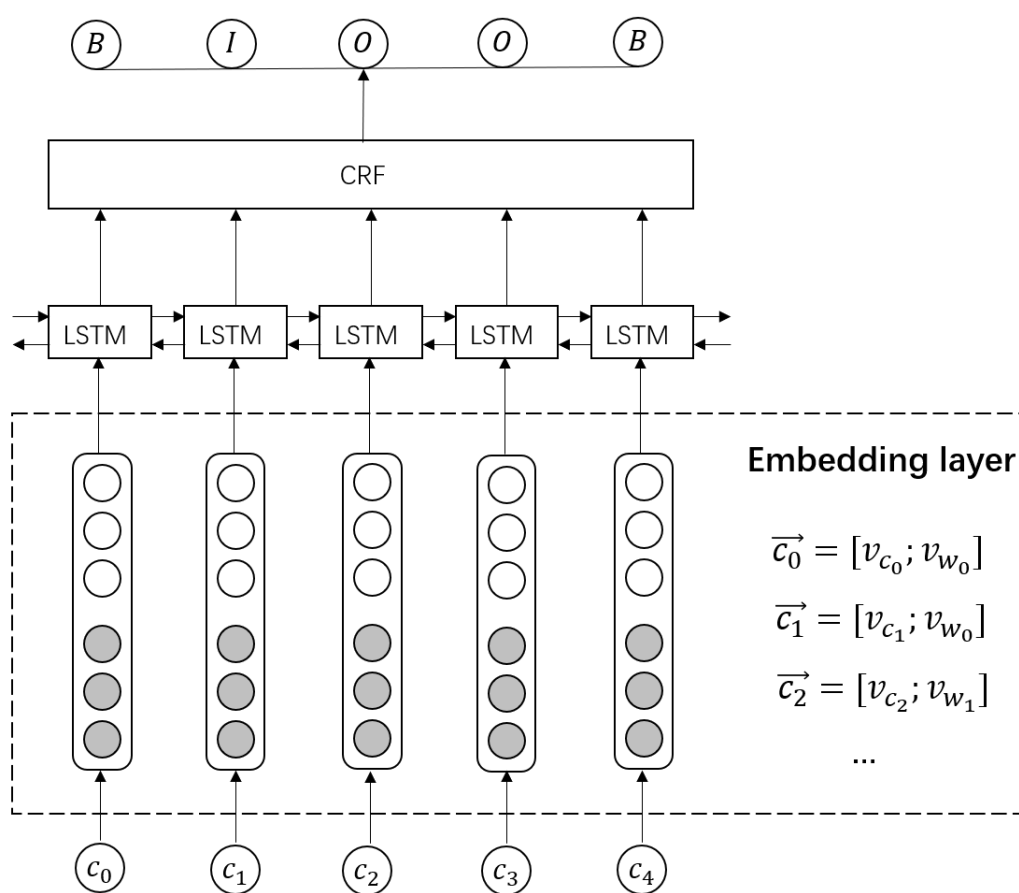


图 17: 使用 BiLSTM+CRF 进行命名实体识别，需要实现特征函数的处理

该模型的原理其实就是将 CRF 中特征函数的发射概率使用 EM 算法估计改成使用 BiLSTM 估计，而其他步骤基本与 CRF 无异。如果要实现一个好的模型，还是要实现特征函数的提取的特征工程。而在实现过程中，如果继续手写实现这一部分来放到 BiLSTM 中进行学习和识别，势必会是一个很大的工程。所以在实现这一部分的过程当中，只是简单的实现了对于中文字符和英文单词的概率计算，所以在实现这一过程中更准确的说法应该是 BiLSTM+HMM 的方法。并未实现 Embedding Layer 特征识别的过程。

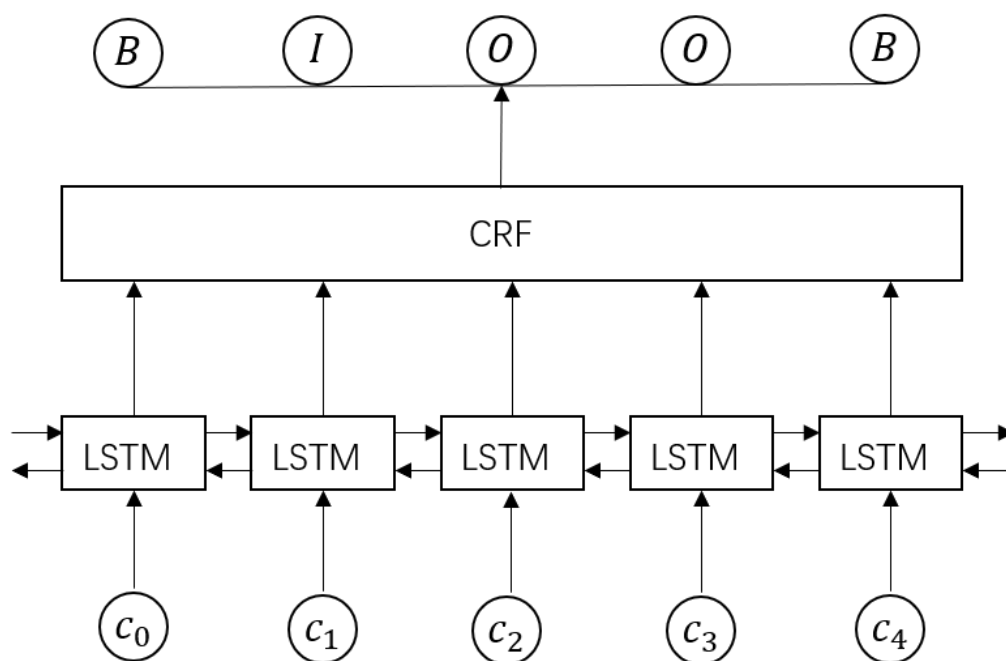


图 18: 使用 BiLSTM+HMM 进行命名实体识别，只是实现了根据 BiLSTM 计算的发射概率放在所谓的 CRF 层中计算对应的转移矩阵，最后输出对应的隐藏状态

二、代码架构

根据第一部分模型的介绍，在实现这一部分的过程中，由于 LSTM 对于特征函数的识别并不能像之前调用 crf 工具一样可以很方便的对特征函数的处理结果：**字典型**，进行很方便的处理，如果需要实现则会需要使用大量的时间。而在第二部分的过程中，我已经实现了手写 CRF 的过程，即便对于特征函数的估计部分并未进行详细的实现。但是总体来说，由于时间有限，在本部分只是将字向量输入，通过 BiLSTM 模块对字向量对应的发射矩阵进行估计，然后使用 CRF 模块对转移概率进行估计，最后使用维特比解码得到最后的输出状态。

代码主要划分为 model 用于设计模型，main 用于计算将输入的文字序列数据转换为 id，utils 用于设计相应的函数。model 中主要有使用 lstm 获取发射概率分数，获取序列标注的转移分数，维特比解码部分。

model

```

1 class BiLSTM_CRF(nn.Module):
2     def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
3         super(BiLSTM_CRF, self).__init__()
4         self.embedding_dim = embedding_dim
5         self.hidden_dim = hidden_dim
6         self.vocab_size = vocab_size
7         self.tag_to_ix = tag_to_ix
8         self.tagset_size = len(tag_to_ix)
9
10        self.word_embeds = nn.Embedding(vocab_size, embedding_dim).to(device)
11        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
12                             num_layers=1, bidirectional=True).to(device)
13        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size).to(device)
14        # 转移矩阵, transitions[i][j] 表示从 label_j 转移到 label_i 的概率
15        # ,虽然是随机生成的但是后面会迭代更新
16        self.transitions = nn.Parameter(torch.randn(

```

```

17         self.tagset_size, self.tagset_size)).to(device)
18         # 从任何标签转移到 START_TAG 不可能
19         self.transitions.data[tag_to_ix[START_TAG], :] = -10000
20         # 从 STOP_TAG 转移到任何标签不可能
21         self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000
22
23         self.hidden = self.init_hidden() # 随机初始化 LSTM 的输入 (h_0,
24
25     def init_hidden(self):
26         return (torch.randn(2, 1, self.hidden_dim // 2).to(device),
27                 torch.randn(2, 1, self.hidden_dim // 2).to(device))
28
29     def _forward_alg(self, feats):
30         """
31         输入：发射矩阵 (emission score)，实际上就是 LSTM 的输出
32         ——sentence 的每个 word 经 BiLSTM 后，对应于每个 label 的得分
33         输出：所有可能路径得分之和/归一化因子/配分函数/Z(x)
34         """
35         init_alphas = torch.full((1, self.tagset_size), -10000.).to(device)
36         init_alphas[0][self.tag_to_ix[START_TAG]] = 0.
37
38         # 包装到一个变量里面以便自动反向传播
39         forward_var = init_alphas
40         for feat in feats: # w_i
41             alphas_t = []
42             for next_tag in range(self.tagset_size): # tag_j
43                 # t 时刻 tag_i emission score (1 个) 的广播。需要
44                 previous_tags 转移到该 tag_i 的 transition scores
45                 emit_score = feat[next_tag].view(1, -1).expand(1,
46                 # t-1 时刻的 5 个 previous_tags 到该 tag_i 的 transition
47                 trans_score = self.transitions[next_tag].view(1,

```



```

48
49         next_tag_var = forward_var + trans_score + emit_
50         # 求和，实现  $w_{(t-1)}$  到  $w_t$  的推进
51         alphas_t.append(log_sum_exp(next_tag_var).view(1))
52         forward_var = torch.cat(alphas_t).view(1, -1).to(device)
53
54     # 最后将最后一个单词的 forward var 与转移 stop tag 的概率相加
55     terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
56     alpha = log_sum_exp(terminal_var)
57     return alpha
58
59     def _get_lstm_features(self, sentence):
60         """
61         输入: id 化的自然语言序列
62         输出: 序列中每个字符的 Emission Score
63         """
64         self.hidden = self.init_hidden() # (h_0, c_0)
65         embeds = self.word_embeddings(sentence).view(len(sentence), 1, -1)
66         lstm_out, self.hidden = self.lstm(embeds, self.hidden)
67         lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
68         lstm_feats = self.hidden2tag(lstm_out) # len(s)*5
69         return lstm_feats
70
71     def _score_sentence(self, feats, tags):
72         """
73         输入: feats——emission scores; tags——真实序列标注，以此确定转移
74         输出: 真实路径得分
75         """
76         score = torch.zeros(1).to(device)
77         # 将 START_TAG 的标签 3 拼接 tag 序列最前面
78         tags = torch.cat([torch.tensor([self.tag_to_ix[START_TAG]]),

```

```

79         dtype=torch.long), tags]).to(device)
80     for i, feat in enumerate(feats):
81         score = score + \
82             self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
83     score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
84     return score
85
86     def _viterbi_decode(self, feats):
87         # 预测序列的得分，维特比解码，输出得分与路径值
88         backpointers = []
89
90         init_vvars = torch.full((1, self.tagset_size), -10000.).to(device)
91         init_vvars[0][self.tag_to_ix[START_TAG]] = 0
92
93         forward_var = init_vvars
94         for feat in feats:
95             bptrs_t = []
96             viterbivars_t = []
97
98             for next_tag in range(self.tagset_size):
99                 # forward_var 保存的是之前的最优路径的值
100                 next_tag_var = forward_var + self.transitions[next_tag, :]
101                 best_tag_id = argmax(next_tag_var) # 返回最大值
102                 bptrs_t.append(best_tag_id)
103                 viterbivars_t.append(next_tag_var[0][best_tag_id])
104
105             forward_var = (torch.cat(viterbivars_t) + feat).view(1, self.tagset_size)
106             backpointers.append(bptrs_t) # bptrs_t 有 5 个元素
107
108         # 其他标签到 STOP_TAG 的转移概率
109         terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG], :]

```

```

110         best_tag_id = argmax(terminal_var)
111         path_score = terminal_var[0][best_tag_id]
112
113         best_path = [best_tag_id]
114         for bptrs_t in reversed(backpointers):
115             best_tag_id = bptrs_t[best_tag_id]
116             best_path.append(best_tag_id)
117
118         # 无需返回最开始的 START 位
119         start = best_path.pop()
120         assert start == self.tag_to_ix[START_TAG]
121         best_path.reverse() # 把从后向前的路径正过来
122         return path_score, best_path
123
124     def neg_log_likelihood(self, sentence, tags): # 损失函数
125         feats = self._get_lstm_features(sentence) # len(s)*5
126         forward_score = self._forward_alg(feats) # 规范化因子/配分函数
127         gold_score = self._score_sentence(feats, tags) # 正确路径得分
128         return forward_score - gold_score # Loss (已取反)
129
130     def forward(self, sentence):
131         """
132         解码过程，维特比解码选择最大概率的标注路径
133         """
134         lstm_feats = self._get_lstm_features(sentence)
135
136         score, tag_seq = self._viterbi_decode(lstm_feats)
137         return score, tag_seq

```

主函数的数据处理

```
1 word_to_ix = {}
2 tag_to_ix = {}
3 for sentence, tags in training_data:
4     for word in sentence:
5         if word not in word_to_ix:
6             word_to_ix[word] = len(word_to_ix)
7     for tag in tags:
8         if tag not in tag_to_ix:
9             tag_to_ix[tag] = len(tag_to_ix)
10 tag_to_ix[START_TAG] = len(tag_to_ix)
11 tag_to_ix[STOP_TAG] = len(tag_to_ix)
```

三、实验结果

本次测试由于 BiLSTM 的训练过程过于耗时，即便是使用 cuda 加速都无法很快的训练。所以本次实验主要针对训练的次数进行改进。在测试过程中，主要测试了训练维数为 1 维和 10 维的模型。效果都不如使用特征函数的 CRF。甚至效果不如简单的 HMM 的效果好。

中文数据

micro avg	0.6092	0.6003	0.6047	8437
macro avg	0.1260	0.1110	0.1095	8437
weighted avg	0.5634	0.6003	0.5672	8437

图 19: 中文数据训练 1 轮

micro avg	0.7967	0.7781	0.7873	8437
macro avg	0.3337	0.3189	0.3188	8437
weighted avg	0.7986	0.7781	0.7821	8437

图 20: 中文数据训练 10 轮

micro avg	0.9424	0.9442	0.9433	8437
macro avg	0.6951	0.7203	0.7056	8437
weighted avg	0.9425	0.9442	0.9432	8437

图 21: 中文数据训练 30 轮

英文数据

micro avg	0.2713	0.0642	0.1038	8603
macro avg	0.1600	0.0408	0.0609	8603
weighted avg	0.2281	0.0642	0.0944	8603

图 22: 英文数据训练 1 轮

micro avg	0.4600	0.3430	0.3930	8603
macro avg	0.3382	0.2527	0.2835	8603
weighted avg	0.4318	0.3430	0.3788	8603

图 23: 英文数据训练 10 轮

micro avg	0.9424	0.9442	0.9433	8437
macro avg	0.6951	0.7203	0.7056	8437
weighted avg	0.9425	0.9442	0.9432	8437

图 24: 英文数据训练 30 轮

四、对 BiLSTM+CRF 模型的理解

该模型的原理其实就是将 CRF 中特征函数的发射概率使用 EM 算法估计改成使用 BiLSTM 估计，而其他步骤基本与 CRF 无异。如果要想实现一个好的模型，还是需要实现特征函数的提取的特征工程。在实验过程中发现，使用 BiLSTM 估计的发射概率计算的效果甚至不如 HMM，更不用说没有实现的 CRF 的特征函数，猜测结果是由于数据量使用太少，对于少量数据，使用简单的特征工程反而更能很好的预测有限的数据，而对于更大规模的数据，使用神经网络可能可以估计更加复杂的语言模型。

Part4: Bonus 部分完成情况及任务总结

一、手写 CRF

该部分的实验总结已在 Part2: CRF 实现命名实体识别 (NER) 任务中进行阐述。

在助教的提示下，对手写 crf 进行了改进，对 bigram 中增加状态的变化统计，然后进行批训练的处理，最终结果如下

micro avg	0.7637	0.5200	0.6187	8112
macro avg	0.7123	0.5229	0.5895	8112
weighted avg	0.7542	0.5200	0.6013	8112

图 25: 英文数据

micro avg	0.8589	0.8009	0.8289	9890
macro avg	0.6109	0.5045	0.5163	9890
weighted avg	0.8611	0.8009	0.8207	9890

图 26: 中文数据

二、CRF 进行分词分词

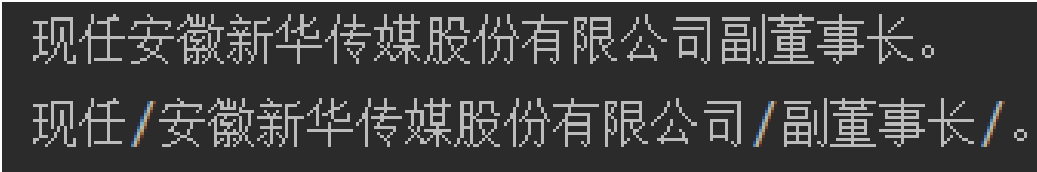
该部分的实现较为简单，即使通过模型对输入的语言数据的状态进行预测，然后观测状态中表示“开始状态”，“结束状态”，“单个字为一个意思的句子”的状态，对其进行分词即可。

分词示例结果如下：

谢卫东先生：1966年12月出生，汉族，硕士研究生、高级工程师、国家注册造价工程师。
 谢卫东/先生：1966年12月出生，/汉族/，/硕士研究生/、/高级工程师/、/国家注册造价工程师/。

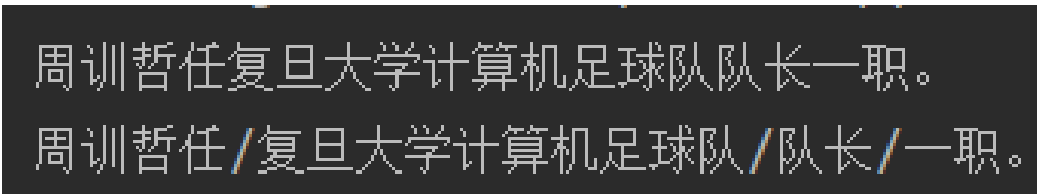
图 27: 采用 validation 中数据进行分词

值得注意的是，该分词结果对于不是语料库中的句子的分词效果并不好，原因也很好理解，就是对于不是语料库中的句子或词的状态并没有一个特定的标签进行标识，导致其无法识别。



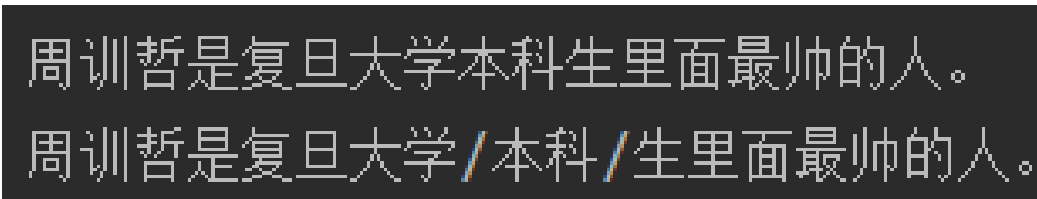
现任安徽新华传媒股份有限公司副董事长。
现任 / 安徽新华传媒股份有限公司 / 副董事长 / 。

图 28: 采用 validation 中数据进行分词



周训哲任复旦大学计算机足球队队长一职。
周训哲任 / 复旦大学计算机足球队 / 队长 / 一职。

图 29: 比较成功的示例



周训哲是复旦大学本科生里面最帅的人。
周训哲是复旦大学 / 本科 / 生里面最帅的人。

图 30: 相对不完善的示例

三、实验总结

本次 project 主要围绕命名实体识别进行任务的完成，在实验过程中，主要实现了 HMM, CRF 以及神经网络的方法完成命名实体识别的任务。其中，在 HMM 实验过程中，我体会到序列数据批处理和不是批处理的巨大差异，同时还体会到使用对数对差异巨大的数据进行归一化，防止数据下溢等操作的重要性；在 CRF 实验过程中，也是消耗时间最多的项目当中，我深刻理解了模型对于数据处理的方便性，可以使用机器学习框架直接可以简化很多复杂的数学计算和估计过程；在最后 BiLSTM-CRF 的实验过程中，我第一次尝试使用 LSTM 模型，学会了如何调用该模型。

总之，在本次实验过程中，我深刻体会到特征工程阶段对于数据处理的重要性，以及神经网络直接将数据喂进模型的简便，也深刻理解和领会了三种模型的特点和相似之处，对于以后人工智能序列数据的处理有着极大的帮助。