Lab2说明文档

By 黄子豪 18302010034

对模型的理解

HMM

隐马尔科夫模型（Hidden Markov Model）经常被用在时间序列（例如一段时间内的声音信号，本次lab我们用它来处理中文的分词问题）的建模与分析。

它有**三个要素**

- 可见随机变量：用来描述一个变化的可观测的量，在本次lab中对应于每个字。
- 隐含的状态变量：一个假设的存在，每个时间点都对应一个状态量，在本次lab中对应于每个字的标签
- 变量间的关系：用概率的方法描述以下三个关系或变量：初始状态量，当前的隐含状态量与下一个隐含状态量间关系，当前的隐含状态量与可见随机量间关系。在代码中分别是初始概率矩阵、转移矩阵和发射矩阵。

马尔科夫性假设及其局限性

- 假设了当前可见随机变量只与当前隐藏状态有关；
- 假设了当前状态只与上一状态有关；

这一假设有一定的合理性，但是其实也有很大的局限性，因为对于中文分词的问题来讲，邻近的字往往有很大的关系，邻近的状态即标签也有很大的关系，而不仅仅是只需要考虑上一个状态的影响，这一点在CRF模型中通过特征模板的定义来解决。

模型原理及应用模型的步骤

首先需要从给出的语料集中统计出hmm的三个参量：初始矩阵、转移矩阵和发射矩阵，这样就得到了一个可应用的hmm模型了。接下来对于给定的字符串序列，只需要使用维特比算法，先前向计算所有可能情况的概率，然后后向取概率最大的那条路径，即最大似然估计。这样得到的隐藏状态就是我们预测的分词标签。

CRF

基本原理及训练步骤

CRF模型弥补了hmm中马尔科夫性假设的不足，定义了很多的特征模板，并且也考虑了可见状态之间的转换（字序列的关系），由此通过不同的特征模板的定义可以生成很多的特征函数，这样就能考虑到前后几个状态和字之间的转换关系，例如$S_{-1}S_0C_0$可以考虑到前一状态、当前状态和当前字的联合概率。

接下来就是在训练集上训练得到恰当的模板频率，对于预测输出错误的特征函数，给它们的频率减一，对于预测正确的特征函数给他们的频率加一。在训练得到了不同特征函数的频率之后，就可以通过维特比解码找出最大概率的隐状态路径了。

优点

- 提供了一个简单的方式将概率模型的结构可视化。
- 通过观察图形，可以更深刻的认识模型的性质，包括条件独立性。
- 高级模型的推断和学习过程中的复杂计算可以利用图计算来表达，图隐式的承载了背后的数学表达式。

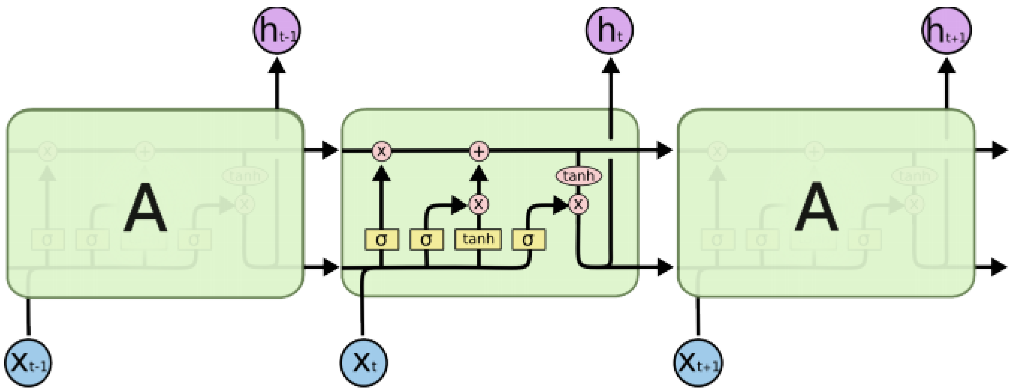
缺点

- 仍然只能考虑到不超过相邻五项的影响。
- 需要人为定义恰当的模板，预测的精确度很大程度上受数据集和模板影响。

BiLSTM + CRF

LSTM基本原理和为什么要加上CRF

BiLSTM通过增加记忆单元和激活函数，实现了能够记忆以前所有出现过的单元的功能，解决长序列训练过程中的梯度消失和梯度爆炸问题，并且能够考虑到当前输入字的影响，具体计算模型见下图：

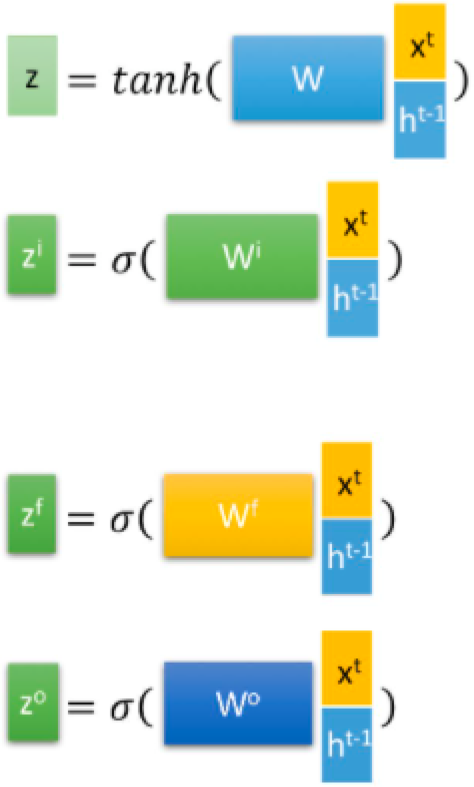


训练的过程即是得到恰当的权重矩阵的模型。

但是这样得到的BiLSTM网络仍然有一个缺点：BiLSTM 可以预测出每一个字属于不同标签的概率，然后使用 Softmax 得到概率最大的标签，作为该位置的预测值。这样在预测的时候会忽略了标签之间的关联性，例如 BiLSTM 可能把第一个字预测成 s，把第二个字预测成 e。但是实际上在分词时 s 后面是不会出现 e 的，因此 BiLSTM 没有考虑标签间联系。因此 BiLSTM+CRF 在 BiLSTM 的输出层加上一个 CRF，使得模型可以考虑类标之间的相关性，标签之间的相关性就是 CRF 中的转移矩阵，表示从一个状态转移到另一个状态的概率。这样子就得到一个准确率更高、更有效的网络。

训练步骤

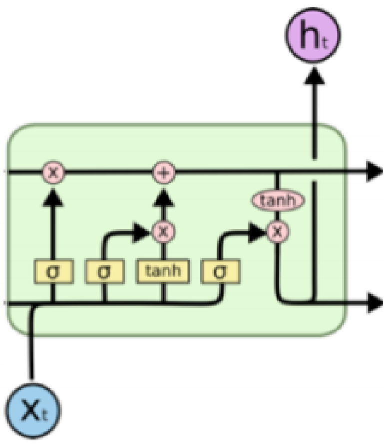
首先首先使用LSTM的当前输入 x^t 和上一个状态传递下来的 h^{t-1} 拼接训练得到四个状态。



其中，z^f，z^i，z^o 是由拼接向量乘以权重矩阵之后，再通过一个 $sigmoid$ 激活函数转换成0到1之间的数值，来作为一种门控状态。而 z 则是将结果通过一个 $tanh$ 激活函数将转换成-1到1之间的值（这里使用 $tanh$ 是因为这里是将其做为输入数据，而不是门控信号）。

这是这四个门在LSTM中的使用：

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$



- $x_t \in R^d$: input vector to the LSTM unit
- $f_t \in R^h$: forget gate's activation vector
- $i_t \in R^h$: input gate's activation vector
- $o_t \in R^h$: output gate's activation vector
- $h_t \in R^h$: output vector of the LSTM unit
- $c_t \in R^h$: cell state vector
- $W \in R^{h\times d}$, $U \in R^{h\times h}$ and $b \in R^h$: weight matrices and bias vector
- σ_g: sigmoid function.
- σ_c: hyperbolic tangent function.
- σ_h: hyperbolic tangent function or, $\sigma_h(x) = x$.

主要可以解释为三个阶段：

1. 忘记阶段。这个阶段主要是对上一个节点传进来的输入进行**选择性**忘记。简单来说就是会 "忘记不重要的，记住重要的"。具体来说是通过计算得到的 z^f （f表示forget）来作为忘记门控，来控制上一个状态的 c^{t-1} 哪些需要留哪些需要忘。

2. 选择记忆阶段。这个阶段将这个阶段的输入有选择性地进行“记忆”。主要是会对输入 x^t 进行选择记忆。哪些重要则着重记录下来，哪些不重要，则少记一些。当前的输入内容由前面计算得到的 z 表示。而选择的门控信号则是由 z^i （i代表information）来进行控制。
3. 输出阶段。这个阶段将决定哪些将会被当成当前状态的输出。主要是通过 z^o 来进行控制的。并且还对上一阶段得到的 c^o 进行了放缩（通过一个tanh激活函数进行变化）。

优点

1. 可以拟合很久远的时间序列的影响。lstm提供了长距离的依赖建模，序列标注问题比较简单就得到了还可以的效果。
2. crf构造字符间的特征对应关系，而lstm提供了词向量的泛化。

缺点

1. 对外部语料集依赖很大（这一点后续试验对比和分析会解释说明）。
2. 参数过多，训练和测试时间过长。

实验对比和分析

这个部分对比在不同的训练集和测试集上的预测效果，对于训练我采用了dataset1和dataset2，对于预测我分别采用了example（助教给出的测试）、dataset1的20%、dataset2的20%进行测试。表格中dataset1训练，dataset1测试和dataset2训练，dataset2测试没有实际意义，故并不列出。

HMM

	dataset1训练	dataset2训练
example测试	79.3	78.4
dataset1测试		80.8
dataset2测试	81.2	

总体来看，无论是用dataset1还是dataset2来训练和测试并没有显著差异，但是他们都比助教给出的测试集性能要好，分析原因可能是助教的测试集样本过少，不具备广泛的代表能力，当训练集扩张到dataset1或者dataset2的20%时正确率有了显著的提升。

CRF

	dataset1训练	dataset2训练
example测试	84.2	83.0
dataset1测试		85.2
dataset2测试	85.8	

CRF的测试结果仍然表明更多样本的测试集能够提高预测的精确度，另外使用dataset1训练的模型预测能力略高于使用dataset2预测的模型，经过对比两个数据集我发现dataset1的数据比dataset2要多，前者大约是后者的两倍，所以猜测这样的现象应该是由于更多的数据能更多的fit庞大的参数量，训练出更好的模型。

BiLSTM+CRF

	dataset1训练	dataset2训练
example测试	84.1	84.3
dataset1测试		93.4
dataset2测试	89.9	

在BiLSTM+CRF模型上的训练，进一步证明了这一点：更多样本的测试集能够减小预测误差，提高预测的精确度。同时我发现此时dataset2训练的模型表现要明显优于dataset1训练的模型，我仔细观察了两个语料集，猜测是一由于测试集的语句分布更加接近于dataset2中的分布，同时相比较而言dataset2中有更多未出现的字，dataset1中有更多少出现的字，因此dataset2的准确率更高。

具体代码实现

HMM

统计参数矩阵，再使用维特比解码：

```
class HmmModel:
    state_list = ['B', 'I', 'E', 'S']
    line_num = -1
    #INPUT_DATA = "../../dataset1/train.utf8"

    def __init__(self, input_data):
        self.trans_p = {}  # 转移概率矩阵
        self.emit_p = {}  # 发射概率矩阵
        self.Count_dic = {}
        self.initial_p = {}  # 初始状态分布
        self.INPUT_DATA = input_data

        self.train()

    def init(self):  # 初始化字典
        for state in self.state_list:
            self.trans_p[state] = {}
            for state1 in self.state_list:
                self.trans_p[state][state1] = 0.0
        for state in self.state_list:
            self.initial_p[state] = 0.0
            self.emit_p[state] = {}
            self.Count_dic[state] = 0

    # 输出模型的三个参数：初始概率+转移概率+发射概率
    def output(self):
        for key in self.initial_p:  # 状态的初始概率
            self.initial_p[key] = self.initial_p[key] * 1.0 / self.line_num*1000

        for key in self.trans_p:  # 状态转移概率
            for key1 in self.trans_p[key]:
                self.trans_p[key][key1] = self.trans_p[key][key1] /
self.Count_dic[key]*100

        for key in self.emit_p:  # 发射概率(状态->词语的条件概率)
            for word in self.emit_p[key]:
```

```
                self.emit_p[key][word] = self.emit_p[key][word] /
self.Count_dic[key]*100

    def train(self):
        self.init()
        ifp = open(self.INPUT_DATA, encoding="utf8")
        word_list = []
        line_state = []
        for line in ifp:
            line = line.strip()
            if not line:
                self.line_num += 1
                for i in range(len(line_state)):
                    if i == 0:
                        self.initial_p[line_state[0]] += 1  # initial_p记录句子第
一个字的状态，用于计算初始状态概率
                        self.Count_dic[line_state[0]] += 1  # 记录每一个状态的出现次
数
                    else:
                        self.trans_p[line_state[i - 1]][line_state[i]] += 1  #
用于计算转移概率
                        self.Count_dic[line_state[i]] += 1
                        if not word_list[i] in self.emit_p[line_state[i]]:
                            self.emit_p[line_state[i]][word_list[i]] = 1.0
                        else:
                            self.emit_p[line_state[i]][word_list[i]] += 1  # 用于
计算发射概率
                word_list = []
                line_state = []
                continue
            else:
                word_list.append(line[0])
                line_state.append(line[2])
        self.output()
        ifp.close()

    def decode(self, sequence):
        """
        Decode the given sequence.
        """
        sequence_length = len(sequence)

        delta = {}
        for state in self.state_list:
            if not sequence[0] in self.emit_p[state]:
                delta[state] = self.initial_p[state] / self.Count_dic[state]
            else:
                delta[state] = self.initial_p[state] * self.emit_p[state]
[sequence[0]]

        pre = []
        for index in range(1, sequence_length):
            # if sequence[index] == "\n":
            #     continue
            delta_bar = {}
            pre_state = {}
            for state_to in self.state_list:
                max_prob = 0
```

```
                max_state = None
                for state_from in self.state_list:
                    prob = delta[state_from] * self.trans_p[state_from]
[state_to]
                    if prob >= max_prob:
                        max_prob = prob
                        max_state = state_from
                if not sequence[index] in self.emit_p[state_to]:
                    self.emit_p[state_to][sequence[index]] = 1.0 /
self.Count_dic[state_to]
                delta_bar[state_to] = max_prob * self.emit_p[state_to]
[sequence[index]]
                pre_state[state_to] = max_state
            delta = delta_bar
            pre.append(pre_state)

        max_state = None
        max_prob = 0
        for state in self.state_list:
            if delta[state] >= max_prob:
                max_prob = delta[state]
                max_state = state

        if max_state is None:
            return []

        result = [max_state]
        for index in range(sequence_length - 1, 0, -1):
            max_state = pre[index - 1][max_state]
            result.insert(0, max_state)
        return ''.join(result)
```

CRF

定义特征模板，由特征模板生成特征函数，预训练得到特征函数初始频率，对于预测错误的序列加减对应的特征函数频率，最后再使用维特比解码：

```
from wordseg import Simple_Func
import pickle


# 特征模板定义
tempCS0 = [[-1], [0]]
tempC0S0 = [[0], [0]]
tempC1S0 = [[1], [0]]
tempCC0S0 = [[-1, 0], [0]]
tempC0C1S0 = [[0, 1], [0]]
tempCC1S0 = [[-1, 1], [0]]

tempCSS0 = [[-1], [-1, 0]]
tempC0SS0 = [[0], [-1, 0]]
tempC1SS0 = [[1], [-1, 0]]
tempCC0SS0 = [[-1, 0], [-1, 0]]
tempC0C1SS0 = [[0, 1], [-1, 0]]
tempCC1SS0 = [[-1, 1], [-1, 0]]
temp_list = []
temp_list.append(tempCS0)
```

```
temp_list.append(tempC0S0)
temp_list.append(tempC1S0)
temp_list.append(tempCC0S0)
temp_list.append(tempC0C1S0)
temp_list.append(tempCC1S0)

temp_list.append(tempCSS0)
temp_list.append(tempC0SS0)
temp_list.append(tempC1SS0)
temp_list.append(tempCC0SS0)
temp_list.append(tempC0C1SS0)
temp_list.append(tempCC1SS0)

freq_dict_list = []
for i in range(len(temp_list)):
    freq_dict = {}
    freq_dict_list.append(freq_dict)

fi = open("../../dataset2/train.utf8", "r", 1, encoding='utf-8')
all_lines = fi.readlines()
all_char = ""
all_state = ""
for line in all_lines:
    line = line.strip()
    if line != "":
        all_char = all_char + line[0]
        all_state = all_state + line[2]
fi.close()

for i in range(len(all_char)):
    for j in range(len(temp_list)):
        key_char = ""
        for k in range(len(temp_list[j][0])):
            index = i + temp_list[j][0][k]
            if 0 <= index < len(all_char):
                key_char += all_char[index]
            else:
                key_char += "NIL"
        key_state = ""
        for k in range(len(temp_list[j][1])):
            index = i + temp_list[j][1][k]
            if 0 <= index < len(all_char):
                key_state += all_state[index]
            else:
                key_state += "NIL"
        key = key_char + key_state
        freq_dict_list[j][key] = freq_dict_list[j].get(key, 0) + 1



count = 1
while True:
    print("第" + str(count) + "轮")
    count += 1
    hit = len(all_state)
    # result1 = viterbi(all_char, "BEIS")
    result1 = Simple_Func.viterbi(all_char, "BEIS", temp_list, freq_dict_list)
    for i in range(len(all_state)):
```

```
        if result1[i] != all_state[i]:
            hit -= 1
            for j in range(len(temp_list)):
                key_char = ""
                for k in range(len(temp_list[j][0])):
                    index = i + temp_list[j][0][k]
                    if 0 <= index < len(all_char):
                        key_char += all_char[index]
                    else:
                        key_char += "NIL"
                key_state = ""
                for k in range(len(temp_list[j][1])):
                    index = i + temp_list[j][1][k]
                    if 0 <= index < len(all_char):
                        key_state += all_state[index]
                    else:
                        key_state += "NIL"
                key1 = key_char + key_state
                freq_dict_list[j][key1] = freq_dict_list[j].get(key1, 0) + 1

                char1 = key_char
                state1 = result1[i]
                if len(temp_list[j][1]) > 1:
                    if i - 1 < 0:
                        state1 = "NIL" + result1[i]
                    else:
                        state1 = result1[i - 1] + result1[i]
                key1 = char1 + state1
                freq_dict_list[j][key1] = freq_dict_list[j].get(key1, 0) - 1
    print(hit / len(all_state))

    obs_test = Simple_Func.get_content_list("../example_dataset/input.utf8")
    gold_test = Simple_Func.get_content_list("../example_dataset/gold.utf8")
    result_test = Simple_Func.viterbi(obs_test, "BEIS", temp_list,
freq_dict_list)
    hit = 0
    for i in range(len(result_test)):
        # print(obs_test[i] + " " + gold_test[i] + " " + result_test[i])
        if result_test[i] == gold_test[i]:
            hit += 1
    print(hit / len(result_test))

    with open("../crf_model/model" + str(count) + ".pkl", 'wb') as outp:  # 保存
        pickle.dump(temp_list, outp)
        pickle.dump(freq_dict_list, outp)
```

BiLSTM+CRF

```
import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim

torch.manual_seed(1)
START_TAG = "<START>"
STOP_TAG = "<STOP>"
```

```
def argmax(vec):
    '''
    计算一维vec最大值的坐标
    '''
    # return the argmax as a python int
    _, idx = torch.max(vec, 1)
    return idx.item()


def log_sum_exp(vec):
    '''
    计算vec的 log(sum(exp(xi)))=a+log(sum(exp(xi-a)))
    '''
    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)))

class Model(nn.Module):

    def __init__(self, vocab_size, tag2id, embedding_dim, hidden_dim):
        super(Model, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag2id = tag2id
        self.tagset_size = len(tag2id)

        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 = trans[i][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[tag2id[START_TAG], :] = -10000
        self.transitions.data[:, tag2id[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):
        '''
        :param feats: LSTM+hidden2tag的输出
        :return: 所有tag路径的score和
        forward_var：之前词的score和
        '''
        # Do the forward algorithm to compute the partition function
```

```python
        init_alphas = torch.full((1, self.tagset_size), -10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag2id[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:   #every word
            alphas_t = []  # The forward tensors at this timestep
            for next_tag in range(self.tagset_size): #every word's tag
                # 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
                # 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.tag2id[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 当前句子的tag路径score
        score = torch.zeros(1)
        tags = torch.cat([torch.tensor([self.tag2id[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.tag2id[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = [] #路径保存

        # Initialize the viterbi variables in log space
        init_vvars = torch.full((1, self.tagset_size), -10000.)
        init_vvars[0][self.tag2id[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = init_vvars
        for feat in feats: # for every word
```

```python
            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): #for every possible tag of
the word
                # 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.tag2id[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.tag2id[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def forward(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 test(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
```