基于CNN-LSTM的汉语词类自动标注系统

姓名：李奇、高云鹏

学号：2019302863、2019302859

学院：计算机学院、软件学院

日期：2021.11.14

一、 背景与方法调研

1.1 课题背景

词性标注是自然语言处理中的一个重要问题，广泛应用于机器翻译、文本识别、语音识别、信息检索等相关领域。词性标注是确定给定句子中每个单词的语法类别，确定其词性并对其进行标注，并通过上下文为每个单词指定正确的候选词性，从而提供词汇分析的过程，自然语言分析中的语法分析和语义提供了支持。

国外对英语词性自动标注的研究起步很早，已经进入了一个快速发展的阶段。不同的改进方法各具特点，大致可以分为以下几个阶段：

（1）20世纪50年代末，出现了最早的词性赋值（也称为词性标注）算法TDAP使用14条规则进行词性消歧，使用的词性标签序列是所有后续算法的原型。

（2）20世纪60年代初，在TDAP之后，提出了一种由字典、形态分析器和上下文歧义消解器组成的“计算语法编码器”。在运行时，一个单词通过字典和形态分析器生成一组候选词性，并使用包含1500条上下文规则的规则集消除歧义。

（3）20世纪70年代初，基于规则的标注方法迅速兴起，其中最典型的是TAGGIT系统。与CGC相比，TAGGIT扩展了字典的规模，增加了标记集的数量。使用TAGGIT标记棕色语料库，准确率可以达到77%。

（4）从20世纪70年代末到80年代初，统计标记方法在经验主义方法的影响下开始变得活跃起来。

（5）20世纪90年代，基于统计的方法开始在词性标注阶段发挥主导作用。研究人员在标注系统中明确使用HMM，并将其与EM算法相结合。近年来，各种统计和机器学习方

法被应用于词性标注。比如有学者提出使用决策树来估计标签的概率等。

本课题所关注的汉语词性标注是汉语信息处理技术中的一个基本课题。一方面，它的研究成果可以直接集成到信息抽取、信息检索、机器翻译等许多实际应用系统中；另一方面，汉语自动词性标注也是汉语块识别器、汉语语法分析器和汉语语义分析器的一个重要的前端处理工具。因此，研究和实现汉语词性标注具有重要的理论意义和实用价值。

与国外词性标注研究现状相比，国内对汉语词性标注的研究起步较晚，加上训练语料库规模的限制，以及汉语本身的复杂性，增加了汉语词性标注研究的难度。20 世纪 80 年代以来，中文信息处理技术的研究越来越受到重视。汉语知识库和语法规则的出现为词性标注提供了基础信息。到目前为止，已经出现了各种各样的汉语标注方法，如基于完全隐马尔可夫模型的汉语词性标注研究，若要在更广范围内提高词性标注的实用性，仍需进一步深入研究。

1.2 相关可行算法

隐马尔科夫模型(Hidden Markov Model,简称 HMM)

条件随机场(Conditional Random Fields,简称 CRF)

自动编码器(AutoEncoder,简称 AE)

卷积神经网络(Convolutional Neural Network,简称 CNN)

长短期记忆模型(Long-Short Term Memory,简称 LSTM)

二、 方法或算法间比较与分析

2.1 算法归纳

词性标注的方法归纳起来，可以分为 4 类：

1)基于规则的方法。2008 年，有学者提出了基于规则优先级的词性标注方法。基于规则的方法简单，易于实现，但手工构造规则是一项非常艰难的任务。

2)基于统计的方法。该方法客观性强，准确性较高，但需要处理兼类词和未登录词的问题。常见的方法有：隐马尔科夫模型、最大熵、条件随机场等. 曾经有学者将隐马尔科夫模型应用在 50 万词的英文语料库上，词性标注准确率达到了 95%。

3)基于规则和统计的方法。充分利用两种方法的各自优势，比单一使用一种的准确性要高，但该类方法依赖于建立的规则或人工特征的选取，同时也与任务领域的资源有

很大的相关性，一旦领域变化，标注效果就会受较大影响。

4)基于深度学习的方法。通过对数据多层建模获得数据的特征和分布式表示，避免繁琐的人工特征抽取，具有良好的泛化能力。深度学习常见的模型主要有多层感知器、自动编码器、卷积神经网络和长短期记忆模型等.这些模型中 CNN 和 LSTM 应用较为广泛。CNN 是目前效果最好的深度学习模型之一，利用滑动窗口，可以很好的解决词的组合特征及一定程度上的依赖问题，表达句子之间的关系较为自然，由于参数共享，因此它的计算量较小，计算速度较快。LSTM 隶属于循环反馈神经网络，它可以将文本中某一序列元素与某时刻模型的输入对应起来，利用隐层单元的记忆模块，保存长间隔信息，对序列元素逐一标注。

2.2 算法比较

2.2.1 HMM:

1) 优点

对转移概率和表现概率直接建模，统计共现概率。

2) 缺点

隐马模型一个最大的缺点就是由于其输出独立性假设，导致其不能考虑上下文的特征，限制了特征的选择，而最大熵隐马模型则解决了这一问题，可以任意的选择特征，但由于其在每一节点都要进行归一化，所以只能找到局部的最优值，同时也带来了标记偏见的问题（label bias)，即凡是训练语料中未出现的情况全都忽略掉。

3) 难点

评估问题、解码问题、学习问题。

4) 可行性

关键点：显示层的结点状态是可以直接观测获取的；作为隐藏层的马尔可夫链的各结点状态不直接可见，需要用可见层的状态去推测。

应用场景：在一定程度上相关，但又不是绝对相关的两种事情进行建模分析：假如知道儿子每天的活动：打球，看书，访友，每天只有一种状态，推测在该天儿子当地的天气情况，因此显示层就是儿子每天的日常活动，隐藏层就是每天的天气，是一个马尔可夫的时间状态链的模型。

显示层与隐藏层的状态不是绝对的映射关系：天气隐藏状态只是一种可能的状态，不是绝对靠谱的事件发生序列。也就是晴天可能打球，但打球不一定就是晴天。

2.2.2 CRF:

1) 优点

与 HMM 比较。CRF 没有 HMM 那样严格的独立性假设条件，因而可以容纳任意的上下文信息。特征设计灵活（与 ME 一样）MEMM 比较。由于 CRF 计算全局最优输出节点的条件概率，它还克服了最大熵马尔可夫模型标记偏置（Label-bias）的缺点。与 ME 比较。CRF 是在给定需要标记的观察序列的条件下，计算整个标记序列的联合概率分布，而不是在给定当前状态条件下，定义下一个状态的状态分布。

2) 缺点

训练代价大、复杂度高。

3) 难点

a.对一个可能的 path 的分数怎么算？即如何定义 score 函数。

b.归一化项 Z 的复杂度是指数递增的，如何计算？

4) 可行性

CRF 的应用场景十分广泛：与深度学习结合，产生了 BiLSTM-CRF、BiLSTM-CNN-CRF 等模型，在中文分词、命名实体识别、词性标注也取得不错的效果。与 Attention 机制结合，又发展成了 Transformer-CRF、BERT-BiLSTM-CRF 等模型，使中文分词、命名实体识别、词性标注效果又有显著提高。

2.2.3 AE:

1) 优点

泛化性强，无监督不需要数据标注。

2) 缺点

针对异常识别场景，训练数据需要为正常数据。

3) 难点

在无监督的情况下，我们没有异常样本用来学习，而算法的基本上假设是异常点服从不同的分布。根据正常数据训练出来的 Autoencoder，能够将正常样本重建还原，但是却无法将异于正常分布的数据点较好地还原，导致还原误差较大。

4) 可行性

自编码器是前馈神经网络的一种，最开始主要用于数据的降维以及特征的抽取，随着技术的不断发展，现在也被用于生成模型中，可用来生成图片等。

2.2.4 CNN:

1) 优点

共享卷积核，处理高维数据无压力；

可以自动进行特征提取 卷积层可以提取特征，卷积层中的卷积核（滤波器）真正发挥作用，通过卷积提取需要的特征，详细解释参考牛人博文 CNN 入门讲解：卷积层是如何提取特征的？

2）缺点

当网络层次太深时，采用 BP 传播修改参数会使靠近输入层的参数改动较慢；

采用梯度下降算法很容易使训练结果收敛于局部最小值而非全局最小值；

池化层会丢失大量有价值信息，忽略局部与整体之间关联性；

由于特征提取的封装，为网络性能的改进罩了一层黑盒。

3）难点

CNN 的方法也训练起来也比较简单，现阶段最后实验的效果也是最好。但有一些窗口大小上选取的经验问题，对文本长程依赖上的问题也并不是很好解决。

4）可行性

CNN 是目前效果最好的深度学习模型之一，利用滑动窗口，可以很好的解决词的组合特征及一定程度上的依赖问题，表达句子之间的关系较为自然，由于参数共享，因此它的计算量较小，计算速度较快。

2.2.5 LSTM:

1）优点

LSTM 是 RNN 的一个优秀的变种模型，继承了大部分 RNN 模型的特性，同时解决了梯度反传过程由于逐步缩减而产生的梯度消失问题。具体到语言处理任务中，LSTM 非常适合用于处理与时间序列高度相关的问题，例如机器翻译、对话生成、编码\解码等。

2）缺点

RNN 的梯度问题在 LSTM 及其变种里面得到了一定程度的解决，但还是不够。它可以处理 100 个量级的序列，而对于 1000 个量级，或者更长的序列则依然会显得很棘手。

计算费时。每一个 LSTM 的 cell 里面都意味着有 4 个全连接层(MLP),如果 LSTM 的时间跨度很大，并且网络又很深，这个计算量会很大，很耗时。

3）难点

LSTM 一共有三个门，输入门、遗忘门、输出门，i,f,o 分别为三个门的程度参数，g 是对输入的常规 RNN 操作。公式里可以看到 LSTM 的输出有两个，细胞状态 c’和隐状

态 h’，c’是经输入、遗忘门的产物，也就是当前 cell 本身的内容，经过输出门得到 h’，就是想输出什么内容给下一单元。

难点在于：

a. 根据任务需求，结合数据，确定网络结构。

b. 确定训练集、验证集和测试集，并尽可能的确保它们来自相同的分布。

c. 确定单一的评估算法的指标。

d. 对数据进行归一化/标准化处理。

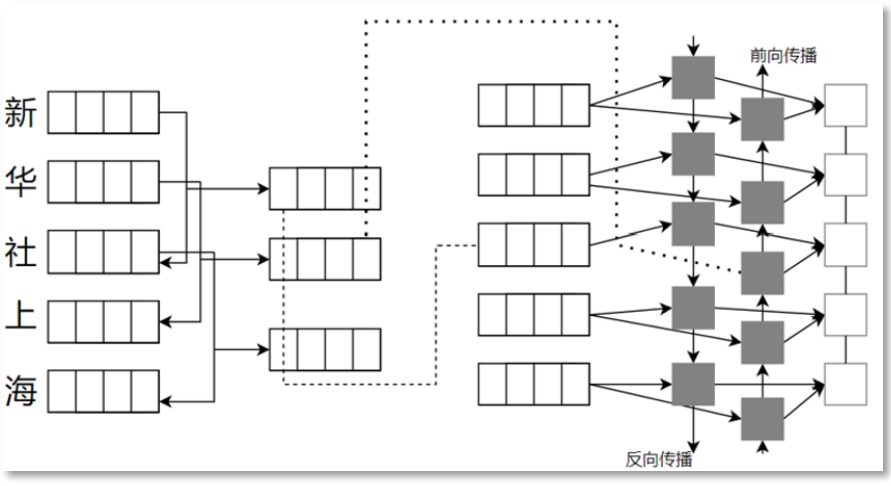
4) 可行性

LSTM 的时序性非常适合标注这种序列任务。它隶属于循环反馈神经网络，它可以将文本中某一序列元素与某时刻模型的输入对应起来，利用隐层单元的记忆模块，保存长间隔信息，对序列元素逐一标注。

三、拟使用算法

3.1 模型结构

本模型主要结构如下，分为三个层次：第一层通过使用 Word Embedding 将中文树库中的文本词语和字符级别转换为词向量，第二层为 CNN 层，利用滑动窗口，将第一层产生的词向量输入到 CNN 层，计算前后词对当前词的影响，生成词语表示特征；第三层为双向 LSTM 层，将 CNN 生成的词向量和字符级表示特征依次输入到双向 LSTM 各节点，通过一层前向传播和反向传播，预测最终生成的词性标注标签。



3.2 词向量处理WordEmbedding

Embedding 使用了 Word2Vec 方法，可以理解为一种对单词 onehot 向量的一种降维处理，通过一种映射关系将一个 n 维的 onehot 向量转化为一个 m 维的空间实数向量（可以理解为原来坐标轴上的点被压缩嵌入到一个更加紧凑的空间内），由于 onehot 向量在矩阵乘法的特殊性，我们得到的表示映射关系的 n*m 的矩阵中的每 k行，其实就表示语料库中的第 k 个单词。

采用这种空间压缩降维的处理方式对语料库中的词进行训练，主要有两种方式：

- skip-gram 模型：一种隐层为 1 的全连接神经网络，且隐层没有激活函数，输出层采用 softmax 分类器输出概率。输入为一个单词，输出为每个单词是输入单词的上下文的概率，真实值为输入单词的上下文中的某个单词。它主要通过滑动窗口控制，它代表着我们从当前输入单词的一侧(左边或右边)选取词的数量。假如我们有一个句子“The dog barked at the mailman”，我们选取“dog”作为输入单词，那么我们最终获得窗口中的词（包括输入单词在内）就是['The', 'dog', 'barked', 'at']。另一个参数是 numskips， 它代表着我们从整个窗口中选取多少个不同的词作为 output word， 当skipwindow=2、num_skips=2 时，我们将会得到两组 (input word, output word) 形式的训练数据，即 ('dog', 'barked')， ('dog', 'the')。
- CBOW 模型：原理与 skip-gram 类似，但是输入为上下文信息，输出为信息中的中心词。

3.3 卷积网络层

令 vi 为第 i 个词的词向量，vi 的维度为 d 维。当句子词语数为 L，卷积神经网络的滑动窗口大小为 k 时，落入第 j 个(j<=L-1)滑动窗口的词向量依次为 vj，vj-1，...，vj+l-1，可以将他们表达为窗口向量：

$$X_j = [v_j, v_{j+1}, \ldots, v_{j+k-1}]$$

对于每个窗口向量，用卷积核 W 进行卷积运算得到当前窗口特征：

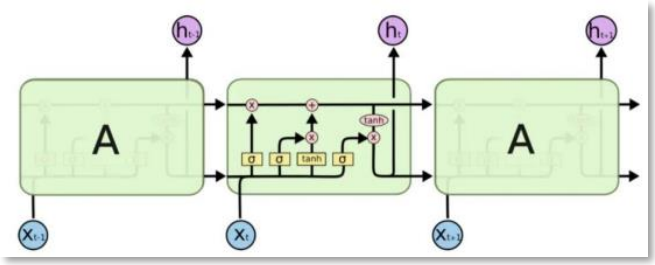
$$Y_j = f(X_j * W + b)$$

使用 Sigmod 激活函数加速收敛，再用最大池化操作来最大化词语特征表示。

3.4 长短期记忆网络模型处理层

3.4.1 LSTM

长期短期记忆网络(LSTM)是一种特殊的 RNN，能够学习长期依赖关系。它是由 Hochriter & Schmidhuber(1997)提出的，其结构如下图所示：



标准 LSTM 模型是一种特殊的 RNN 类型，在每一个重复的模块中有四个特殊的结构，以一种特殊的方式进行交互。在图中，每一条黑线传输着一整个向量，粉色的圈代表一种 pointwise 操作(将定义域上的每一点的函数值分别进行运算)，诸如向量的和，而黄色的矩阵就是学习到的神经网络层。

LSTM 有通过精心设计的称作为“门”的结构来去除或者增加信息到 LSTM 单元状态的能力。门是一种让信息选择式通过的方法。他们包含一个 sigmoid 神经网络层和一个 pointwise 乘法操作。Sigmoid 层输出 0 到 1 之间的数值，描述每个部分有多少量可以通过。0 代表“不许任何量通过”，1 就指“允许任意量通过”。LSTM拥有三个门，来保护和控制 LSTM 单元状态。每个 LSTM 单元的实现都有下面几个公式构成：

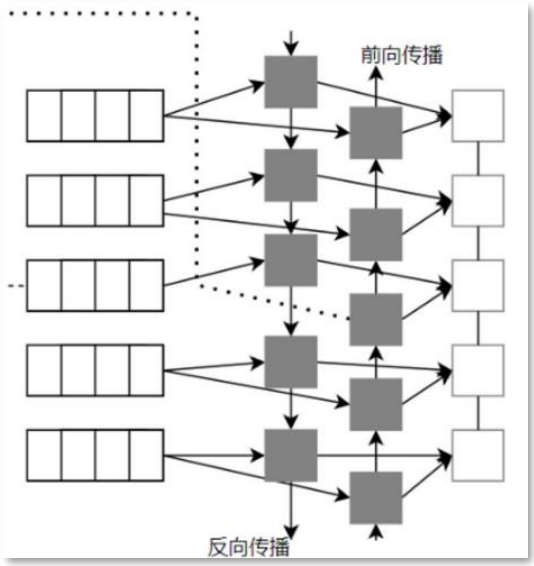
$$\begin{gathered} i_t = \sigma(W_{x_i} x_t + W_{h_i} h_{t-1} + W_{ci} c_{t-1} + b_i) \\ f_t = \sigma(W_{x_f} x_t + W_{h_f} h_{t-1} + W_{cf} c_{t-1} + b_f) \\ c_t = f_t c_{t-1} + i_t tanh(W_{x_c} x_t + W_{h_c} h_{t-1} + b_c) \\ o_t = \sigma(W_{x_o} x_t + W_{h_o} h_{t-1} + W_{co} c_{t-1} + b_o) \\ h_t = o_t tanh(c_t) \end{gathered}$$

其中 σ 是 logistic sigmoid 函数，i、f、o 和 c 是输入门、遗忘门、输出门和单元向量，所有这些向量的大小与隐藏向量 h 相同。权重矩阵下标的含义顾名

思义。例如，Whi 是隐藏的输入门矩阵，Wxo 是输入——输出门矩阵等。从单元到门向量（例如 Wci）的权重矩阵是对角的，因此每个门向量中的元素 m 仅接收来自单元向量的元素 m 的输入。

在双向 LSTM 的输入中，我们应用字符级别和词级别融合的算法，将两者做拼接，一并送入输入门中，以达到更佳的特征提取效果。

3.4.2 双向 LSTM



LSTM 是一种前向传播算法，因此对词性标注任务而言，需要综合反向的 LSTM进行学习，即双向 LSTM。此方法可以高效地表示出这个单词在上下文中的含义，在标准 Bi-LSTM 中，输出通过接入 Softmax 输出层预测节点间的分类标签。

双向 LSTM 网络结构如上图所示，一个 LSTM 网络计算前向的隐特征（从下到上），另一个 LSTM 网络计算后向的隐特征（从上到下）。我们把这两个 LSTM 输出的结果拼接，就得到双向 LSTM 网络，用于特征提取。

3.4.3 算法实现难度

首先，在词向量嵌入过程中，我们使用 gensim 工具包的 word2vec 训练，使用简单速度快，效果比 Google 的 word2vec 效果好。

其次，我们采用的 CNN 的一大优势在于快，非常快，卷积是计算机图形学的核心部分，已经可以通过 GPU 硬件级实现。与 n-grams 等方法相比，CNN 在表征方面也更加高效。随着词汇量越来越大，计算任何多于 3-grams 的东西都会

非常昂贵。而且 NLP 任务中由于语句天然存在前后依赖关系，使用 CNN 的参数共享特点能获得一定的特征表达。

LSTM 的输入参数很难和前面公式里的那些联系起来，不便于理解和灵活使用。

3.5 对现有算法的删改

对 CNN，我们仅对字符级别的词向量进行 Encoder，对 LSTM，我们将 CNN 卷积运算中获取的字符集向量和词级别词嵌入向量做拼接，一并送入输入门中，以达到更佳的特征提取效果。

四、主要步骤介绍

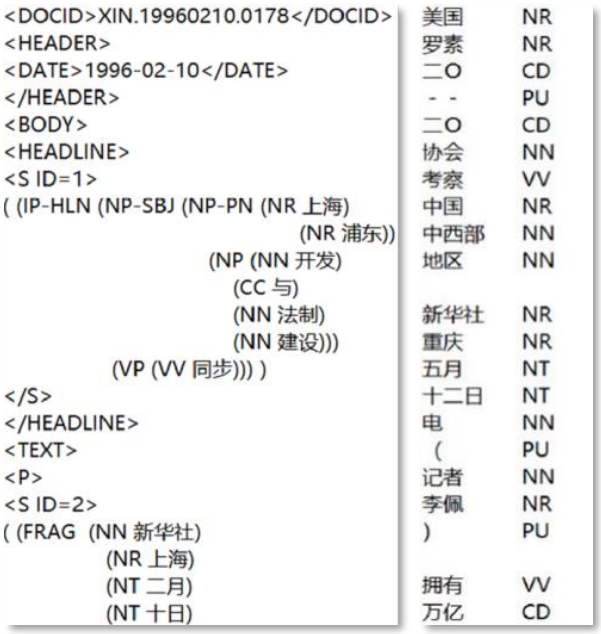
4.1 数据集预处理

本项目采用的是 LDC 整合的中文树库（Chinese TreeBank8.0）结构如下：

我们提取 bracketed 中的数据，文件的文本结构是树状分支：

我们先删去括号等无关字符，再提取文本中的 word 和 pos，分别做成词级别、字符级别和词性的字典，并按 9:1:1 的比例划分训练集、测试集、验证集。

最后得到的数据如下图所示：



```
<DOCID>XIN.19960210.0178</DOCID>
<HEADER>
<DATE>1996-02-10</DATE>
</HEADER>
<BODY>
<HEADLINE>
<S ID=1>
( (IP-HLN (NP-SBJ (NP-PN (NR 上海)
                         (NR 浦东))
                  (NP (NN 开发)
                      (CC 与)
                      (NN 法制)
                      (NN 建设)))
          (VP (VV 同步))) )
</S>
</HEADLINE>
<TEXT>
<P>
<S ID=2>
( (FRAG  (NN 新华社)
         (NR 上海)
         (NT 二月)
         (NT 十日)
```

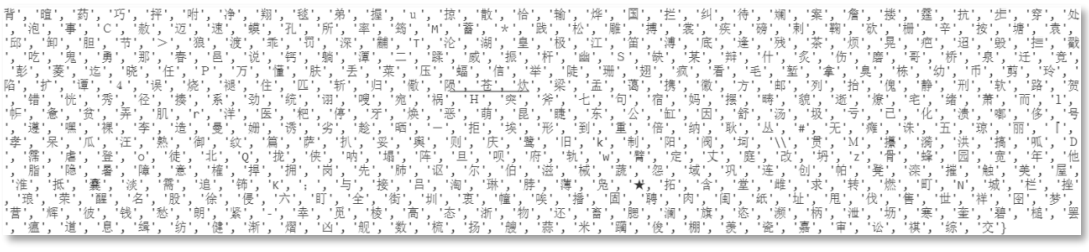
```
美国    NR
罗素    NR
二O     CD
- -     PU
二O     CD
协会    NN
考察    VV
中国    NR
中西部  NN
地区    NN

新华社  NR
重庆    NR
五月    NT
十二日  NT
电      NN
(       PU
记者    NN
李佩    NR
)       PU

拥有    VV
万亿    CD
```

4.2 词向量训练

CreateVocabs(corpuspath)函数：

- 我们将所有数据整合，再将其词、词性分开存储，提取词中的字符单独存储，将词、字符整理成一个以\t 作为分隔符的字典，放入 Word2Vec 中训练，得到 word2vec.txt 和 char_vectors.txt 文件供训练数据使用。
- 运用 word2vecgenism.py 得到的词、字向量矩阵，每一行的格式：第一个为词／字，其余为对应的矩阵，每一行只有一个词／字，在 getembedding_weights 函数中，进行索引的构建，并用用词向量填充矩阵，在进行归一化，得到最终的词向量词典和字符向量词典。

示例词、字符、词性原始词典如下图所示：



下图是处理好的词向量词典：

```
[[-3.135056   -3.0745358   2.4060922  ...   1.6213574   1.0928906
  -0.6082333 ]
 [ 0.2977171   1.5184691   1.3729707  ... -0.6817758   1.610216
   0.8173386 ]
 [ 0.3269037  -1.4773941   1.6598862  ... -0.00549742 -0.9386338
   1.0845954 ]
 ...
 [ 1.2047886  -0.15368676 -0.63711435 ...  0.66213757 -0.8047817
   0.55358016]
 [ 0.6445963   0.44589233 -1.2325156  ...  0.17130695  0.6607188
   0.41398725]
 [-0.46651903 -1.2087593   0.26545104 ... -1.0447538   0.8768938
  -1.7287456 ]]
```

4.3 CNN

利用上一步已经训练好的字符向量输入到 CNN 中进行特征提取，步骤如下：

1)将字符 Embedding 的向量归一化。

2)设置的窗口大小分别为 2*2，3*3，4*4 ，步长为 1 的卷积核。

3)设置网络中的全连接层，长度为 winsize * charhidden_size。

4)进行 dropout，drop_rate 设为 0.3。

5)使用掩码处理 padding 过的不定长序列卷积结果：

a.根据实际序列长度计算卷积输出的长度。

b.在 step 1 的基础上生成 mask。

c.mask 乘以卷积输出的结果。

4.4 双向 LSTM

1)将 2 维的时间序列多变量数据转化为 3 维，input 和 output 大小相等，选用 dense，隐藏层用 lstm 层。

2)将字符级别和词级别的特征矩阵，两者做拼接，一并送入输入门中。

3)一个 LSTM 网络计算前向的隐特征，另一个 LSTM 网络计算后向的隐特征。

4)训练 LSTM 网络，将两个 LSTM 进行结果拼接。

4.5 可能存在的难点：

1)数据处理方面存在困难，word2Vec 的训练格式和训练参数，词典的收录，未登录词录入处理和词、字符与 id 的互相转换都是数据处理中遇到的问题。

解决方法：学习 word2Vec 训练格式，是否归一化处理。

2)由于批次处理的问题，却要求整个训练集是规整的，这就要求每个句子的长度要保持一致。处理不定长句子也是一个问题。

解决方法：首先，对整篇文章的词进行排序编码，给出每个词的编码表示。然后将每个句子转换为词编码向量，再使用 pytorch 中的函数处理为同一长度，方便处理。

3)调整 LSTM 超参数，模型调优将是训练时主要花费时间的问题。

解决方法：a) 根据任务需求，结合数据，确定网络结构 b) 确定训练集、验证集和测试集，并尽可能的确保它们来自相同的分布，并且训练集与测试集的划分通常是9: 1，然后在训练集中在进行验证集的划分，验证集的划分可以是交叉验证，也可以是固定比例。c) 确定单一的评估算法的指标。d) 对数据进行 归一化／标准化处理。

4.6 项目分工

4.6.1 前期论文调研

李奇：查找相关学术论文、分析文章所解决的问题和可借鉴性、相关算法流程分析。

高云鹏：将筛选的论文整合、得出主要思路、分析可行性。

4.6.2 代码编写与调试：

李奇：改进项目代码并调试、代码中注释的书写。

高云鹏：撰写项目代码、验证思路可行性。

4.6.3 撰写项目报告：

李奇：背景与方法调研、算法间的比较分析、项目总结评估及未来发展展望。

高云鹏：拟使用算法、主要步骤介绍、代码实现与解释性标注。

五、代码实现与解释性标注

模型已部署到网站，运行app.py(或运行predict.bat)，之后打开浏览器进入http://127.0.0.1/5000即可。

5.1 模型源代码

代码详细见附件zip文件。这里列出**主体代码函数的定义及解释**，代码中均标有详细注释：

5.1.1 train_easy.py：

```
"""
计算精确度
"""
def calc_acc(pred, target):
    """
    Args:
        pred (batch_size, seq_len): pred tag result, in numpy format
        target (batch_size, seq_len): target tag result, in numpy format

    Returns:
        与预测相等的标签个数
    """

"""
计算误差
"""
def calc_loss(pred, target):
    """
        Args:
        pred (batch_size, seq_len): pred tag result, in numpy format
        target (batch_size, seq_len): target tag result, in numpy format

    Returns:
```

```
        NLLLoss 误差值
    """


"""
在 test.tsv 上评估模型
"""
def evaluate(test_data, tagger, vocab, char_vocab, config):
    """
        input:
        test_data (batch_size, seq_len): 需要预测的数据
        tagger : 已经训练好的权重
        vocab : 中文词词典
        char_vocab : 字符词典
        config : 自定义参数
    """


"""
训练模型
"""
def train(train_data, test_data, dev_data, vocab, config, wd_embed_weights,
char_embed_weights):
    """
    Args:
        train_data (batch_size): 训练集，格式：每行为[词 词性]格式
        test_data (batch_size): 测试集，格式：[词 词性]格式
        dev_data (batch_size): 开发集，格式：[词 词性]格式
        vocab : 中文词词典
        config : 自定义参数
        wd_embed_weights (嵌入词向量)：格式：每行第一个为词，接长度 300 的向量
        char_embed_weights (嵌入字符向量)：格式：每行第一个为字符，接长度 300 的
向量
    """
```

5.1.2 TaggerModel_easy.py:

```
class POSTagger(nn.Module):
   def __init__(self, vocab, config, embedding_weights=None):
      super(POSTagger, self).__init__()

      self.config = config
      self.bidirectional = True
      self.embedding_size = embedding_weights.shape[1]
      self.embedding =
nn.Embedding.from_pretrained(torch.from_numpy(embedding_weights))
```

```
        self.rnn_encoder = RNNEncoder(
            input_size=self.embedding_size,
            hidden_size=self.config.hidden_size,
            num_layers=self.config.nb_layers,
            dropout=self.config.drop_rate,
            bidirectional=self.bidirectional,
            batch_first=True
        )

        self.hidden2pos = nn.Linear(config.hidden_size, vocab.pos_size)
        # self.hidden2pos = nn.Linear(num_directions * config.hidden_size,
vocab.pos_size)
        self.embed_dropout = nn.Dropout(config.drop_embed_rate)
        self.linear_dropout = nn.Dropout(config.drop_rate)

    def forward(self, inputs, mask):
        # batch_size = inputs.size(0)
        # print(inputs.shape)  # (batch_size, seq_len)

        # print('data is in cuda: ', inputs.device, mask.device)

        embed = self.embedding(inputs)  # (batch_size, seq_len, embed_size)
        # print(embed.shape)

        if self.training:  # 预测时要置为 False
            embed = self.embed_dropout(embed)

        rnn_out, hidden = self.rnn_encoder(embed, mask)  # (batch_size,
seq_len, hidden_size)
        # print(rnn_out.shape)

        if self.bidirectional:
            rnn_out = rnn_out[:, :, :self.config.hidden_size] + rnn_out[:, :,
self.config.hidden_size:]

        if self.training:
            rnn_out = self.linear_dropout(rnn_out)

        pos_space = self.hidden2pos(rnn_out)  # (batch_size, seq_len,
pos_size)
        # print(pos_space.shape)
        pos_space = pos_space.reshape(-1, pos_space.size(-1))  # (batch_size *
seq_len, pos_size)
```

```
        pos_score = F.log_softmax(pos_space, dim=1)  # (batch_size * seq_len,
pos_size)
        # print(pos_score.shape)

        return pos_score
```

5.1.3 rnn_easy.py

```
'''
LSTMCell
输入: input, (h_0, c_0)
    input (seq_len, batch, input_size): 包含输入序列特征的 Tensor。也可以是
packed variable
    h_0 (batch, hidden_size): 保存着 batch 中每个元素的初始化隐状态的 Tensor
    c_0 (batch, hidden_size): 保存着 batch 中每个元素的初始化细胞状态的 Tensor

输出: h_1, c_1
    h_1 (batch, hidden_size): 下一个时刻的隐状态。
    c_1 (batch, hidden_size): 下一个时刻的细胞状态。


LSTM
输入: input, (h_0, c_0)
    input (seq_len, batch, input_size): 包含输入序列特征的 Tensor。也可以是
packed variable ，详见
[pack_padded_sequence](#torch.nn.utils.rnn.pack_padded_sequence(input,
lengths, batch_first=False[source])
    h_0 (num_layers * num_directions, batch, hidden_size):保存着 batch 中每个元
素的初始化隐状态的 Tensor
    c_0 (num_layers * num_directions, batch, hidden_size): 保存着 batch 中每个
元素的初始化细胞状态的 Tensor

输出: output, (h_n, c_n)
    output (seq_len, batch, hidden_size * num_directions): 保存 RNN 最后一层的
输出的 Tensor。 如果输入是 torch.nn.utils.rnn.PackedSequence，那么输出也是
torch.nn.utils.rnn.PackedSequence。
    h_n (num_layers * num_directions, batch, hidden_size): Tensor，保存着 RNN
最后一个时间步的隐状态。
    c_n (num_layers * num_directions, batch, hidden_size): Tensor，保存着 RNN
最后一个时间步的细胞状态。
'''
class RNNEncoder(nn.Module):
    def __init__(self, input_size=0, hidden_size=0, num_layers=1,
batch_first=False, bidirectional=False, dropout=0.0, rnn_type='lstm'):
```

```
        super(RNNEncoder, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.batch_first = batch_first
        self.bidirectional = bidirectional
        self.dropout = dropout
        self.num_directions = 2 if self.bidirectional else 1

        self._rnn_types = ['RNN', 'LSTM', 'GRU']
        self.rnn_type = rnn_type.upper()
        assert self.rnn_type in self._rnn_types
        # 获取 torch.nn 对象中相应的的构造函数
        self._rnn_cell = getattr(nn, self.rnn_type+'Cell')  # getattr 获取对象
的属性或者方法

        # ModuleList 是 Module 的子类，当在 Module 中使用它的时候，就能自动识别为子
module
        # 当添加 nn.ModuleList 作为 nn.Module 对象的一个成员时（即当我们添加模块到我
们的网络时），
        # 所有 nn.ModuleList 内部的 nn.Module 的 parameter 也被添加作为我们的网络的
parameter
        self.fw_cells, self.bw_cells = nn.ModuleList(), nn.ModuleList()
        for layer_i in range(self.num_layers):
            layer_input_size = self.input_size if layer_i == 0 else
self.num_directions * self.hidden_size
            self.fw_cells.append(self._rnn_cell(input_size=layer_input_size,
hidden_size=self.hidden_size))
            if self.bidirectional:
                self.bw_cells.append(self._rnn_cell(input_size=layer_input_size,
hidden_size=self.hidden_size))

        # self.cell = nn.LSTMCell(
        #  input_size=self.input_size,   # 输入的特征维度
        #  hidden_size=self.hidden_size  # 隐层的维度
        # )
```

5.1.4 Vocab.py

```
class POSVocab(object):
```

```
    def __init__(self, words_set, pos_set):
        self.UNK = 0

        self._wd2idx = None
        self._idx2wd = None

        self._pos2idx = {pos: idx for idx, pos in enumerate(pos_set)}
        self._idx2pos = {idx: pos for pos, idx in self._pos2idx.items()}
        print('词性数量：', len(self._pos2idx))

    def get_embedding_weights(self, embed_path):
        # 保存每个词的词向量
        wd2vec_tab = {}
        vector_size = 0
        with open(embed_path, 'r', encoding='utf-8', errors='ignore') as fin:
            for line in fin:
                tokens = line.split()
                vector_size = len(tokens) - 1
                wd2vec_tab[tokens[0]] = list(map(lambda x: float(x),
tokens[1:]))

        self._wd2idx = {wd: idx + 1 for idx, wd in
enumerate(wd2vec_tab.keys())}  # 词索引字典 {词: 索引}，索引从 1 开始计数
        self._wd2idx['<unk>'] = self.UNK
        self._idx2wd = {idx: wd for wd, idx in self._wd2idx.items()}

        vocab_size = len(self._wd2idx)  # 词典大小(索引数字的个数)
        embedding_weights = np.zeros((vocab_size, vector_size),
dtype='float32')  # vocab_size * EMBEDDING_SIZE 的 0 矩阵
        for idx, wd in self._idx2wd.items():  # 从索引为 1 的词语开始，用词向量填
充矩阵
            if idx != self.UNK:
                embedding_weights[idx] = wd2vec_tab[wd]
                embedding_weights[self.UNK] += wd2vec_tab[wd]

        # 对于 OOV 的词赋值为 0 或 随机初始化 或 赋其他词向量的均值
        # embedding_weights[self.UNK, :] = np.random.uniform(-0.25, 0.25,
config.embedding_size)
        # embedding_weights[self.UNK] = np.random.uniform(-0.25, 0.25,
config.embedding_size)
        embedding_weights[self.UNK] = embedding_weights[self.UNK] / vocab_size
        embedding_weights = embedding_weights / np.std(embedding_weights)  #
归一化
        return embedding_weights
```

5.1.5 predict.py

```
def load_model(model_path):
    """
    Args:
        model_path : 模型放置路径
    Returns:
    模型文件
    """

def load_vocab(vocab_path):
    """
    加载词典
    Args:
        vocab_path  词典路径

    Returns:
        词典文件(pkl)
    """

# 预测数据不含标签，标签值置成 -1
def predict(pred_data, tagger, vocab, char_vocab):
    """
    预测模型
    Args:
        pred_data (batch_size): 测试数据（需要将输入的句子将每个词用空格隔开）
        vocab : 中文词词典
        config : 自定义参数

    Returns:
        预测词性在词典中的索引
```

5.1.6 dataloader.py

```
# 一个 Instance 对应一行记录
class Instance(object):
    """
    将word 和 pos 封装成一个实体
    Args:
      object:
    return:
      Instance(word, pos)
    """
    def __init__(self, words, pos):
        self.words = words  # 保存词序列
        self.pos = pos      # 保存词序列对应的词性序列
```

```
    def __str__(self):
        return ' '.join([wd+'_'+p for wd, p in zip(self.words, self.pos)])


# 加载数据集，数据封装成 Instance 实体
def load_data(corpus_path):    #一个 word，一个 pos 的格式
    """
    Args:
        corpus_path: 文件路径

    Returns:
        Instance(word, pos)
    """

# 获取 batch 数据
def get_batch(data, batch_size, shuffle=True):
    """
    Args:
        data : (数据集)
        batch_size : 批处理个数
        shuffle : 是否打乱数据集

    Yields:
        batch_data: 一个batch 的数据集
    """

def create_vocabs(corpus_path):
    """

    Args:
        corpus_path :  词典路径

    Returns:
        CharVocab(char_set) : 生成 CharVocab 实体
        PosVocab(pos_set) : 生成 PosVocab 实体
    """

def pred_data_variable(insts, vocab, char_vocab):
    """
    Args:
        insts : Instance(word, pos)
          vocab : 中文词词典
          char_vocab : 字符词典
```

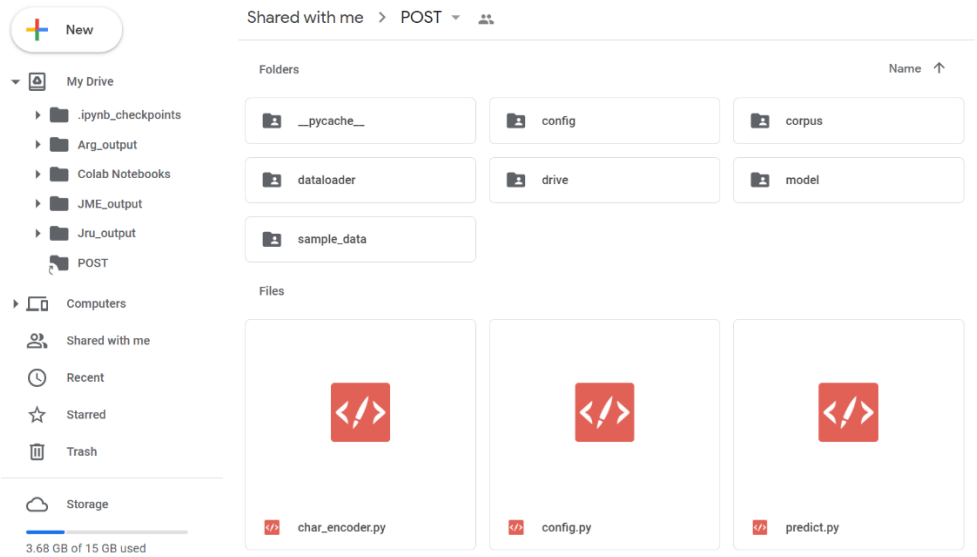
```
    Returns:
        wds_ids : 每个词对应的id索引词典
        char_ids : 每个字符所对应的d字符词典
        seq_lens : 序列长度
    """

def batch_variable_mask_easy(batch_data, vocab, char_vocab):
    """

    Args:
        batch_data : 批处理数据
          vocab : 中文词词典
          char_vocab : 字符词典
    Returns:
        wds_idxs : 每个词对应的id索引词典
        char_idxs : 每个字符所对应的id字符词典
        pos_idxs : 每个POS所对应的id词性词典
        seq_lens : 序列长度
    """
```

5.2 环境配置与安装步骤

5.2.1 环境配置：python 3.7+torch 1.9.0+cuda 11.1

详情见README.md文件。

可使用Google Colab：



5.2.2 参数配置，格式类似于windows INI文件：

```
[Data]
data_dir = corpus
train_data_path = %(data_dir)s/train.ctb60.pos
test_data_path = %(data_dir)s/test.ctb60.pos
dev_data_path = %(data_dir)s/dev.ctb60.pos
embedding_path = %(data_dir)s/word2vec.txt

[Save]
model_dir = model
load_vocab_path = %(model_dir)s/vocab.pkl
save_vocab_path = %(model_dir)s/vocab.pkl
load_model_path = %(model_dir)s/pos_model.pkl
save_model_path = %(model_dir)s/pos_model.pkl

[Optimizer]
learning_rate = 1e-3
weight_decay = 1e-7

[Network]
epochs = 15
nb_layers = 1
# 最大序列长度
max_len = 100
hidden_size = 128
batch_size = 64
embedding_size = 200
drop_rate = 0.3
drop_embed_rate = 0.3
```

六、项目总结评估及未来发展展望

6.1 代码评估及性能分析

一开始在训练过程中，loss在训练到5个epoch之后就不降反升，训练集开发集准确率也在87%左右不变，经过分析，我们认为可能是数据集的处理问题。模型有可能出现过拟合，未登录词(oov)量过大，shuffle导致失去了句子之间的联系，POS标签数量太多导致难以拟合，再就是LSTM模型本身的问题，亦或者epoch的量不够等（每一部分具体些）。

我们依次采取的步骤如下：审视模型(CharEncoding_Bi-Lstm),减少数据量和标签，改变激活函数，增加epoch，最后我们采用了官网中文维基百科的与训练词向量代替自己用Gensim训练的词向量,降低了学习率和权重衰减,最终的训练结果如下：

```
GPU可用： True
CuDNN： True
GPUs： 1
training......
Data
data_dir : corpus
train_data_path : corpus/train.tsv
test_data_path : corpus/test.tsv
dev_data_path : corpus/dev.tsv
word_embedding_path : corpus/wd_embed.txt
char_embedding_path : corpus/char_vectors.txt
Save
model_dir : model
load_vocab_path : model/vocab.pkl
save_vocab_path : model/vocab.pkl
load_char_vocab_path : model/char_vocab.pkl
save_char_vocab_path : model/char_vocab.pkl
load_model_path : model/pos_model.pkl
save_model_path : model/pos_model.pkl
```

```
Optimizer
learning_rate : 2e-4
weight_decay : 1e-6
Network
epochs : 20
nb_layers : 1
max_len : 100
hidden_size : 128
char_hidden_size : 64
batch_size : 64
drop_rate : 0.3
drop_embed_rate : 0.3
train data size: 1302161
test data size: 161407
dev data size: 156994
词性数量： 41
355793 4935
 -- Epoch 1
time: 3.941 min
train: |loss: 9480.291558  acc: 0.839470|
```

```
acc: [0.8394699272977766]
loss: [9480.291557900608]
dev: |loss: 808.476160  acc: 0.874530|
 -- Epoch 2
time: 3.897 min
train: |loss: 7083.880707  acc: 0.867137|
acc: [0.8394699272977766, 0.8671370130114479]
loss: [9480.291557900608, 7083.880707189441]
dev: |loss: 778.775558  acc: 0.878677|
 -- Epoch 3
time: 3.818 min
train: |loss: 6750.648207  acc: 0.871683|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965]
dev: |loss: 761.343842  acc: 0.878811|
 -- Epoch 4
time: 3.810 min
train: |loss: 6575.415804  acc: 0.873818|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142]
```

```
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566]
dev: |loss: 757.905278  acc: 0.880059|
 -- Epoch 5
time: 3.819 min
train: |loss: 6464.547073  acc: 0.875379|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653]
dev: |loss: 759.038817  acc: 0.879110|
```

```
 -- Epoch 6
time: 3.832 min
train: |loss: 6383.471538  acc: 0.876273|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589]
dev: |loss: 749.875280  acc: 0.880320|
 -- Epoch 7
time: 3.818 min
train: |loss: 6323.248430  acc: 0.877129|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369]
dev: |loss: 752.654122  acc: 0.880027|
 -- Epoch 8
time: 3.838 min
train: |loss: 6281.996727  acc: 0.877551|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322]
dev: |loss: 749.717782  acc: 0.880384|
```

```
 -- Epoch 9
time: 3.877 min
train: |loss: 6232.934989  acc: 0.878308|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304]
dev: |loss: 750.243538  acc: 0.880537|
 -- Epoch 10
time: 3.846 min
train: |loss: 6212.100043  acc: 0.878594|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253]
dev: |loss: 743.810502  acc: 0.881537|
 -- Epoch 11
time: 3.865 min
train: |loss: 6179.469014  acc: 0.878974|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025]
```

```
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698]
dev: |loss: 745.794564  acc: 0.880868|
 -- Epoch 12
time: 3.872 min
train: |loss: 6161.289114  acc: 0.879288|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744]
```

```
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919]
dev: |loss: 751.689678  acc: 0.879639|
 -- Epoch 13
```

```
time: 3.853 min
train: |loss: 6132.548744  acc: 0.879752|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317]
dev: |loss: 745.261077  acc: 0.874568|
 -- Epoch 14
```

```
time: 3.863 min
train: |loss: 6123.343005  acc: 0.880040|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275]
dev: |loss: 745.419034  acc: 0.874174|
 -- Epoch 15
time: 3.839 min
train: |loss: 6100.634132  acc: 0.880070|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911]
```

```
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045]
dev: |loss: 744.899476  acc: 0.880881|
 -- Epoch 16
time: 3.843 min
train: |loss: 6088.188899  acc: 0.880349|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045, 6088.188898734748]
dev: |loss: 747.012381  acc: 0.880562|
 -- Epoch 17
time: 3.861 min
train: |loss: 6070.771781  acc: 0.880778|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948, 0.8807781833429199]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045, 6088.188898734748, 6070.771780680865]
dev: |loss: 743.211407  acc: 0.881390|
 -- Epoch 18
time: 3.857 min
```

```
train: |loss: 6058.501422  acc: 0.880740|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948, 0.8807781833429199, 0.8807397856332665]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045, 6088.188898734748, 6070.771780680865, 6058.501422278583]
dev: |loss: 749.479646  acc: 0.880811|
 -- Epoch 19
time: 3.848 min
```

```
train: |loss: 6050.862069  acc: 0.880816|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948, 0.8807781833429199, 0.8807397856332665,
0.8808158130983803]
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045, 6088.188898734748, 6070.771780680865, 6058.501422278583,
6050.8620690479875]
dev: |loss: 747.075400  acc: 0.879849|
 -- Epoch 20
time: 3.869 min
train: |loss: 6037.442985  acc: 0.881050|
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948, 0.8807781833429199, 0.8807397856332665,
0.8808158130983803, 0.8810500391272661]
```

```
loss: [9480.291557900608, 7083.880707189441, 6750.6482074297965,
6575.415804095566, 6464.547073304653, 6383.471538070589, 6323.248429529369,
6281.996726846322, 6232.934988707304, 6212.100043155253, 6179.469014331698,
6161.289114072919, 6132.548743650317, 6123.3430047780275,
6100.6341323927045, 6088.188898734748, 6070.771780680865, 6058.501422278583,
6050.8620690479875, 6037.4429854899645]
dev: |loss: 747.090974  acc: 0.881422|
```

```
acc: [0.8394699272977766, 0.8671370130114479, 0.8716825338802191,
0.873818214491142, 0.8753794653656499, 0.8762725960921883,
0.8771288650174595, 0.877551239823647, 0.8783076747038193,
0.8785941216178338, 0.8789742589434025, 0.8792875842541744,
0.8797521965409807, 0.8800401793633813, 0.880070129576911,
0.8803488969489948, 0.8807781833429199, 0.8807397856332665,
0.8808158130983803, 0.8810500391272661] loss: [9480.291557900608,
7083.880707189441, 6750.6482074297965, 6575.415804095566, 6464.547073304653,
6383.471538070589, 6323.248429529369, 6281.996726846322, 6232.934988707304,
6212.100043155253, 6179.469014331698, 6161.289114072919, 6132.548743650317,
6123.3430047780275, 6100.6341323927045, 6088.188898734748,
6070.771780680865, 6058.501422278583, 6050.8620690479875,
6037.4429854899645]
```

```
val_acc: [0.8745302368243373, 0.8786768921105265, 0.8788106551842746,
0.8800591105392562, 0.879110029682663, 0.8803202670165738,
0.8800272621883639, 0.8803839637183587, 0.8805368358026422,
0.8815368740206632, 0.880868058651923, 0.8796387123074767,
0.8745684548454081, 0.8741735352943425, 0.8808807979922799,
0.8805623144833561, 0.8813903716065582, 0.8808107316203166,
0.8798489114233665, 0.8814222199574506] val_loss: [808.4761595521122,
778.7755575925112, 761.343841612339, 757.9052779376507, 759.0388173833489,
749.875279635191, 752.6541217640042, 749.7177824452519, 750.2435381636024,
743.810501960601, 745.7945639999671, 751.6896777227521, 745.2610769298626,
745.4190336614847, 744.8994759181514, 747.0123808626086, 743.2114069260424,
749.4796459190547, 747.0753995081614, 747.0909741222858]
test: |loss: 769.125106  acc: 0.882496|
```

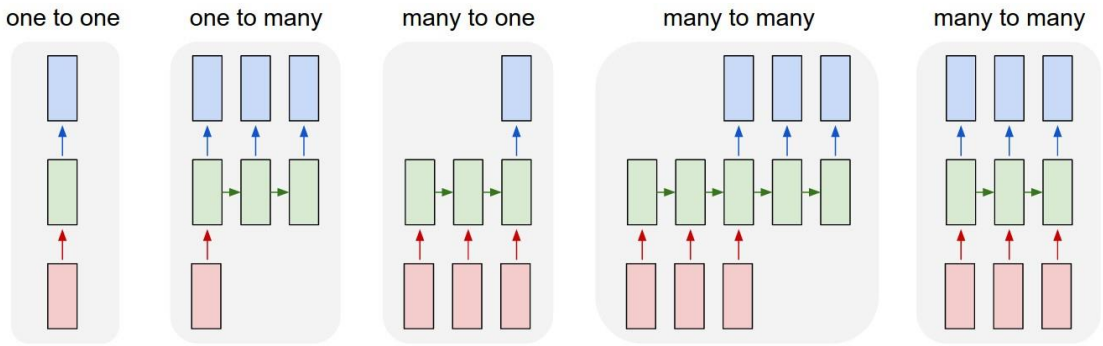
最终的acc提升到88%以上，是目前能达到的最好效果。

6.2 开发过程经验总结

6.2.1 训练开始前要做什么

1）根据任务需求，结合数据，确定网络结构。例如对于RNN而言，你的数据是变长

还是非变长；输入输出对应关系是many2one还是many2many等等。



2）确定训练集、验证集和测试集，并尽可能的确保它们来自相同的分布，并且训练集与测试集的划分通常是7：3，然后在训练集中在进行验证集的划分，验证集的划分可以是交叉验证，也可以是固定比例。一旦确定了数据集的划分，就能够专注于提高算法的性能。如果能够保证三者来自相同的分布，对于后续的问题定位也会有着极大的意义。例如，某个模型在训练集上效果很好，但是在测试集上的结果并不如意，如果它们来自相同的分布，那么就可以肯定：模型在训练集上过拟合了（overfitting），那么对应的解决办法就是获取更多的训练集。但是如果训练集和测试集来自不同的分布，那么造成上述结果的原因可能会是多种的：(i).在训练集上过拟合；(ii).测试集数据比训练集数据更难区分，这时，有必要去进行模型结构，算法方面的修改；(iii).测试集并不一定更难区分，只是与训练集的分布差异比较大，那么此时如果我们去想方设法提高训练集上的性能，这些工作都将是白费努力。

6.2.2 开始调参前要做什么

1）首先不使用Dropout以及正则化项，使用一个较小的数据集（从原始数据集中取出一小部分），让你的网络去训练拟合这个数据集，看看能否做到损失为0 / 准确率为1 （前提是这个小数据集不能只包含一类样本）。

2）在一轮epoch中，打印出输入、输出，检测数据的正确性（例如图像数据确保size，其他数据检查是否输入为0，以及检查是否每个batch都是相同的值，检查特征与标签是否对应）

3）去除正则化项，观察初始的loss值，并对loss进行预估。例如，一个二分类问题，使用softmax分类器，那么当样本属于两个类的概率都为0.5的时候，此时的loss = -ln(0.5) = 0.69，后续当网络的loss不再变化时，看看那时候的loss与这个值的关系。如果最终不

再变化的loss值等于这个值，那么也就是说网络完全不收敛。

4）可视化训练过程，在每一轮epoch训练完成后，计算验证集上的loss与准确率（你的评价指标），并记录下每一轮epoch后训练集与验证集的loss与评价指标。如果是图像数据，可以进行每一层的可视化。

5）如果可以的话，在开始训练之前，尝试用经典的数据集（网上公开数据集，经常在深度学习的网络中使用的数据集）先进行训练，因为这些经典数据集都有参考标准（baseline），而且没有数据方面的问题（如噪声、不平衡、随机性过大导致难以学习的问题等等，尤其是在你自己设计了一个新的网络结构时。

6.2.3 如何进行调参

1）在确保了数据与网络的正确性之后，使用默认的超参数设置，观察loss的变化，初步定下各个超参数的范围，再进行调参。对于每个超参数，我们在每次的调整时，只去调整一个参数，然后观察loss变化，千万不要在一次改变多个超参数的值去观察loss。

2）对于loss的变化情况，主要有以下几种可能性：上升、下降、不变，对应的数据集有train与val（validation），那么进行组合有如下的可能：

train loss 不断下降，val loss 不断下降——网络仍在学习；

train loss 不断下降，val loss 不断上升——网络过拟合；

train loss 不断下降，val loss 趋于不变——网络欠拟合；

train loss 趋于不变，val loss 趋于不变——网络陷入瓶颈；

train loss 不断上升，val loss 不断上升——网络结构问题；

train loss 不断上升，val loss 不断下降——数据集有问题；

其余的情况，也是归于网络结构问题与数据集问题中。

3）当loss趋于不变时观察此时的loss值与1-3中计算的loss值是否相同，如果相同，那么应该是在梯度计算中出现了nan或者inf导致oftmax输出为0。此时可以采取的方式是减小初始化权重、降低学习率。同时评估采用的loss是否合理。

6.2.4 如何解决遇到的问题

1)当网络过拟合时,可以采用的方式是正则化(regularization)与丢弃法(dropout)以及BN层（batch normalization），正则化中包括L1正则化与L2正则化，在LSTM中采用L2

正则化。另外在使用dropout与BN层时，需要主要注意训练集和测试集上的设置方式不同，例如在训练集上dropout设置为0.5，在验证集和测试集上dropout要去除。

2）当网络欠拟合时，可以采用的方式是：去除 / 降低 正则化；增加网络深度（层数）；增加神经元个数；增加训练集的数据量。

3）设置early stopping，根据验证集上的性能去评估何时应该提早停止。

4）对于LSTM，可使用softsign（而非softmax）激活函数替代tanh（更快且更不容易出现饱和（约0梯度））

5）尝试使用不同优化算法，合适的优化器可以是网络训练的更快，RMSProp、AdaGrad或momentum（Nesterovs）通常都是较好的选择。

6）使用梯度裁剪（gradient clipping），归一化梯度后将梯度限制在5或者15。

7）学习率（learning rate）是一个相当重要的超参数，对于学习率可以尝试使用余弦退火或者衰减学习率等方法。

8）可以进行网络的融合（网络快照）或者不同模型之间的融合。

6.3 项目领域未来展望

中文词性标注经过几十年的发展已经取得了长足的进步，但很多根本性的问题仍未能有效地解决，其难点主要在于，相对于英文或其他外文，中文没有屈折变化，也没有附加词缀，所以一个词如果要活用，根本不需要改变其形态。也就是说，中文实际上要靠语序来判断句子成分，而不像屈折语和黏着语那可以靠词缀，难度自然就上去了；其次在中文里一词多词性很常见。统计发现，一词多词性的概率高达22.5%。而且越常用的词，多词性现象越严重。比如“研究”既可以是名词（“基础性研究”），也可以是动词（“研究计算机科学”）；再者就是中文的词性划分标准不统一。词类划分粒度和标记符号等目前还没有一个广泛认可的统一的标准。比如LDC标注语料中，将汉语一级词性划分为33类，而北京大学语料库则将其划分为26类；还有词类划分标准和标记符号的不统一，以及分词规范的含糊，都给词性标注带来了很大的困难；之后就是未登录词问题，未登录词不能通过查找字典的方式获取词性，和分词一样，其词性也是一个比较大的课题。

未来Chinese POSTag任务的发展前景是被看好的，词性标注是数据标注的一种，而

数据标注与审核是整个人工智能行业的基础，数据是人工智能的燃料，如果没有数据人工智能就没有办法开展运算，所以数据标注师被称作“人工智能背后的人”，也有人说“有多少人工就有多少智能”。在当前的大背景下标注任务有着十分丰富的应用场景和广阔的发展前景，比如监督学习下的深度学习算法训练十分依赖人工标注数据，近年来人工智能行业不断优化算法增加深度神经网络层级，利用大量的数据集训练提高算法精准性，ImageNet开源的1400多万张训练图片和1000余种分类在其中起到重要作用，为了继续提高精准度，保持算法优越性，市场中产生了大量的标注数据需求。人工智能是趋势，数据标注是基础，人工智能的发展必然会带给相关行业磅礴的生机。

6.4 相关文献及资料整理

6.4.1 参考博客

人人都能看懂的LSTM

LSTM超参数调试注意事项

神经网络为什么要归一化

LSTM+RNN等

6.4.2 跟踪中文词性标注乃至中文nlp其他各个领域的最新进展与动态，可访问网站中文自然语言处理 Chinese NLP。

6.4.3 以下是整理的近几年涌现出的表现出色的模型、准确率及来源文章，以及部分公开的源代码地址：

Model	Accuracy	Paper / Source	Code
Meta BiLSTM (Bohnet et al., 2018)	97.96	Morphosyntactic Tagging with a Meta-BiLSTM Model over Context Sensitive Token Encodings	Official
Flair embeddings (Akbik et al., 2018)	97.85	Contextual String Embeddings for Sequence Labeling	Flair framework

Char Bi-LSTM (Ling et al., 2015)	97.78	Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation	
Adversarial Bi-LSTM (Yasunaga et al., 2018)	97.59	Robust Multilingual Part-of-Speech Tagging via Adversarial Training	
BiLSTM-CRF + IntNet (Xin et al., 2018)	97.58	Learning Better Internal Structure of Words for Sequence Labeling	
Yang et al. (2017)	97.55	Transfer Learning for Sequence Tagging with Hierarchical Recurrent Networks	
Ma and Hovy (2016)	97.55	End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF	
LM-LSTM-CRF (Liu et al., 2018)	97.53	Empowering Character-aware Sequence Labeling with Task-Aware Neural Language Model	
NCRF++ (Yang and Zhang, 2018)	97.49	NCRF++: An Open-source Neural Sequence Labeling Toolkit	NCRF++
Feed Forward (Vaswani et a. 2016)	97.4	Supertagging with LSTMs	
Bi-LSTM (Ling et al., 2017)	97.36	Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation	
Bi-LSTM (Plank et al., 2016)	97.22	Multilingual Part-of-Speech Tagging with Bidirectional Long Short-Term Memory Models and Auxiliary Loss	

ACE + fine-tune (Wang et al., 2020)	93.4	Automated Concatenation of Embeddings for Structured Prediction	Official
PretRand (Meftah et al., 2019)	91.46	Joint Learning of Pre-Trained and Random Units for Domain Adaptation in Part-of-Speech Tagging	
FastText + CNN + CRF	90.53	Twitter word embeddings (Godin et al. 2019 (Chapter 3))	

6.5 课程总结

八周时间匆匆而过，校园也再一次经历了银装素裹，起初怀着好奇之心踏入这间教室，最终带着知识和感悟走出课堂。感谢卢老师半个学期以来的辛勤付出，让我们了解了自然语言处理技术框架，初步建立起了知识体系，掌握了中文分词、词性标注、句法分析、语义分析、语音识别、语音合成等自然语言处理技术原理，掌握了文本分类、文本检索和信息提取、文本排重、文本摘要、文本主题分析、文本情感分析等自然语言处理应用，熟悉了智能问答、机器翻译的深度学习应用，懂得了应用系统开发的基本技能，让我们受益匪浅。

但要想真正踏入自然语言处理领域的大门，仅凭一门课的学习是远远不够的，需要继续由兴趣驱动着我们，在今后继续投入大量时间，所谓日拱一卒，功不唐捐，玉汝于成。

本报告完成于十一月中旬，其间虽投入了不小精力，但由于水平有限，难免有诸多的小问题。不尽之处，还望批评指正！