



学院：数据科学与计算机学院  
学号：17341213

专业：计算机科学与技术  
姓名：郑康泽

科目：自然语言处理

## 中期大作业报告

### 一. 新闻数据爬取

#### 1. 网站

<http://tech.sina.com.cn/roll>

#### 2. 原因

首先展示该网站的显示内容：

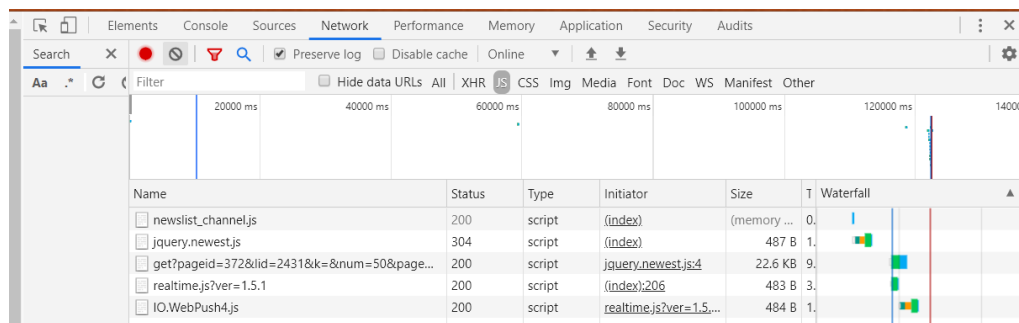


栏目	标题	时间
[全部]	Google危机：一场被揭露的商业化丑闻	11-23 15:10
[全部]	英媒：美团、拼多多等中国科技“独角兽”受市场追捧	11-23 14:46
[全部]	特斯拉皮卡，会大卖吗？	11-23 14:33
[全部]	“捧杯”一月后 李国庆投了一家新公司	11-23 13:42
[全部]	Facebook 版“微信支付”来了	11-23 13:39
[全部]	谷歌提升奖金计划额度 最高可获150万美元	11-23 13:00
[全部]	特斯拉股价因Cybertruck皮卡太丑而暴跌6%	11-23 12:43
[全部]	俞敏洪：把“信善勇让”放心 想不把企业做大都难	11-23 11:28
[全部]	谁在银河系黑洞附近吹“泡泡”？	11-23 11:08
[全部]	为什么有些人会“无酒自醉”？	11-23 11:05

该网页有 50 条新闻，并且都是科技新闻，所以我们只需要获取该网页中的 url 即可。然后继续下一页获取另外 50 条新闻的 url，这样 1000 条新闻一下子就爬到了。如果是从一个新闻的网页中获取另外一条新闻的 url 就麻烦了，因为这样写起来就想写深搜或者广搜一样，代码量也比较大。所以我就选取了这种好爬的滚动新闻。

#### 3. 方法

一开始我是找不到这些新闻的 url，即右键选择“检查网页源代码”后弹出的代码中，是找不到这些新闻的 url。原因就是这些 url 是动态获取的，不是静态的，所以这些 url 不可能写在实现静态页面的代码中。所以我们就要找到网页获取这些新闻的 url 的接口，打开 google 浏览器的开发者工具，刷新一下网页，可以在开发者工具看到以下内容：



Name	Status	Type	Initiator	Size	T	Waterfall
newslist_channel.js	200	script	(index)	(memory ...	0.	
jquery.newest.js	304	script	(index)	487 B	1.	
get?pageid=372&lid=2431&k=&num=50&page...	200	script	jquery.newest.js:4	22.6 KB	9.	
realtime.js?ver=1.5.1	200	script	(index):206	483 B	3.	
IO.WebPush4.js	200	script	realtime.js?ver=1.5...	484 B	1.	

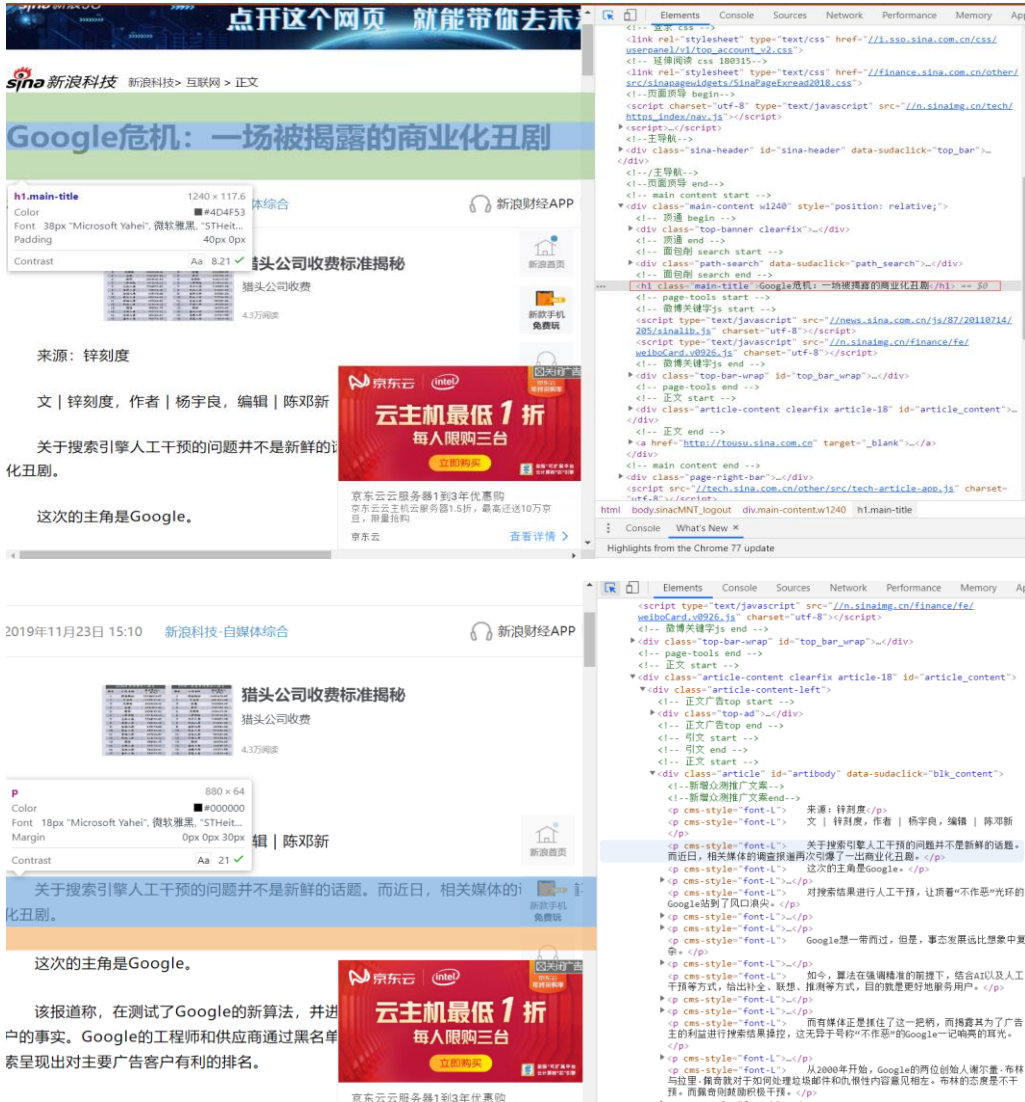
我们要找到的接口就在其中，看名字就觉得“get?pageid=372...”那个包就很有可能，打开



所以我们只需要一个 `for` 循环改变一下参数 `page` 的值，即可获得无数的 `url`。至于如何从这个类似 `json` 格式的字符串中获取 `url`，可以用正则表达式取出。



接下来我们再来看下如何筛选网页内容。新闻一般都是一个标题，然后接一篇正文，所以我们只需要找到这两个部分在 html 中的即可，同样是利用开发者工具，快速找到这两部分在代码中的位置，如下：



可见，标题一般在 h1 标签中，当然也有其他情况，遇到其他情况就不要标题了；正文一般在 div 标签且 class 属性为"article"中的 p 标签中，当然也会有特殊情况，比如 class 属性不为"article"的，这里可以选择不要这篇新闻，或者继续将特殊情况加入代码中。

#### 4. 代码

1) 从 api 接口提取出 50 条新闻的 url:



```
def get_url(i):
    # 向api发请求
    url = URL + str(i)
    response = requests.get(url, headers=HEADERS)
    response = response.text

    # 去转义字符
    response = response.replace('\\', '')

    # 正则提取url
    urls_1 = re.findall('https://tech\\.sina\\.com\\.cn/.+?html', response)
    urls_2 = re.findall('http://tech\\.sina\\.com\\.cn/.+?html', response)
    urls = urls_1 + urls_2

    # 去重
    urls = list(set(urls))
    copy_urls = copy.deepcopy(urls)
    for each in copy_urls:
        # 检查日期 在2019的新闻
        date = re.search('\\d{4}-\\d{2}-\\d{2}', each)
        if int(date[0][0:4]) < 2019:
            urls.remove(each)

    # 避免与之前的重复
    if os.path.exists(URL_RECORD):
        # print('in')
        for line in open(URL_RECORD, 'r'):
            if line.strip('\\n') in urls:
                urls.remove(line.strip('\\n'))
    for each in urls:
        print(each)
    print(len(urls))

    # 记录爬过的网页
    f = open(URL_RECORD, 'a')
    for url in urls:
        f.write(url)
        f.write('\\n')
    f.close()

    return urls
```

2) 从上面的函数获取新闻的 url，然后开始爬取数据：

```
def run(urls):
    global RECORD_NUM
    for url in urls:
        # 返回网页代码
        response = requests.get(url, headers=HEADERS)

        # 指定编码
        response.encoding = 'utf-8'
        soup = bf(response.text, 'lxml')

        # 找到包含标题的标签
        title = soup.find(attrs={'class': 'main-title'})
        if title is None:
            title = soup.find('h1')
        if title is None:
            break
```



```
# 找到包含正文的标签
contents = soup.find(attrs={'class': 'article'})
if contents is None:
    contents = soup.find(attrs={'class': 'blkContainerSblkCon BSHARE_POP'})
    # BSHARE_POP blkContainerSblkCon clearfix blkContainerSblkCon_14
if contents is None:
    break
contents = contents.find_all('p')
RECORD_NUM += 1

# 写入文件
f = open('data/'+str(RECORD_NUM)+'.txt', 'w', encoding='utf-8')
f.write(title.text.strip())
for content in contents:
    f.write('\n')
    f.write(content.text.strip())
f.close()
```

## 二. 爬取数据预处理

### 1. 预处理步骤

#### 1) 分句:

以”。？！；……”中任意一个符号分句，也可以以”，”分句，但是可能会出现句子太短的情况，所以就没有以”，”分句；

#### 2) 分词:

利用 jieba 分词;

#### 3) 去燥:

去燥包括两部分，第一是去掉非中文部分即去掉标点符号以及英文，第二部分是去掉停止词。去掉标点符号及英文可以用正则表达式以及 utf-8 搭配实现，去掉停止词需要下载停止词表，再进行筛选;

#### 4) 写文件:

在第 3) 步中，可以记录下来每句话保留的词语，然后将词语以及每句话写到文件中，方便下一步训练。

### 2. 代码

#### 1) 预处理函数:

```
def preprocess():
    for i in range(1, Num):
        # 读取原文
        lines = open(Path1 + FileName.format(i), 'r', encoding='utf-8').readlines()

        # 分词、去燥、写文件
        with open(Path2 + FileName.format(i), 'w', encoding='utf-8') as f:
            for line in lines:
                # 以；。？！……分句
                line = [line]
                for char in Split_Punctuation:
                    sentences = [] # 记录一行中的句子
                    for j in range(len(line)):
                        sentences += line[j].split(char)
                line = sentences
```



```
# 对句子先分词再去噪
for sentence in sentences:
    if sentence == '':
        continue
    # 分词
    sentence = jieba.lcut(sentence)
    # sentence = pynlpic.segment(sentence, pos_tagging=False)

    # 去除非中文
    for j in range(len(sentence)):
        sentence[j] = re.sub('[^\u4e00-\u9fa5]', '', sentence[j])
    while '' in sentence:
        sentence.remove('')

    # 去停止词
    sentence = [word for word in sentence if word not in Stop_Word]

    # 更新词表
    Word_Table.update(sentence)
    # 写文件
    if len(sentence):
        f.write(' '.join(sentence) + '\n')
print('Finish {d}...'.format(i))
```

2) 写词表函数:

```
def write_table():
    with open(Path2 + Table_Path, 'w', encoding='utf-8') as f:
        for i, word in enumerate(Word_Table):
            f.write(str(i) + ' ' + word + '\n')
```

## 三. N-Gram 模型

### 1. 原理

利用大规模语料库,采用统计方法来计算一个句子的概率。对于一个句子  $s = w_1 w_2 \dots w_m$ , 它的先验概率为:

$$p(s) = p(w_1) \times p(w_2|w_1) \times p(w_3|w_1 w_2) \times \dots \times p(w_m|w_1 \dots w_{m-1})$$

$$= \prod_{i=1}^m p(w_i|w_1 \dots w_{i-1})$$

但这个计算方法显然是有问题,如果句子的长度过长,那么计算量将非常大,所以就提出了 n-gram 模型。n-gram 模型将上式中  $p(w_i|w_1 \dots w_{i-1})$  改为  $p(w_i|w_{i-n} \dots w_{i-1})$ , 这样就减少了许多计算量。并且为了保证条件概率在  $i = 1$  时有意义,同时为了保证句子内所有字符串的概率和为 1,可以在句子首尾两端增加两个标志: <BOS> 和 <EOS>。

当  $n = 2$ ,  $p(s)$  可以分解为:

$$p(s) = \prod_{i=1}^{m+1} p(w_i|w_{i-1}), \text{ where } w_0 \text{ is } < \text{BOS} >, w_{m+1} \text{ is } < \text{EOS} >$$

对于  $n > 2$ ,  $p(s)$  可以分解为:





$$p(s) = \prod_{i=1}^{m+1} p(w_i | w_{i-n+1}^{i-1}), \text{ where } w_i^j \text{ denotes sequence } w_1 \dots w_j$$

那么 $p(w_i | w_{i-n+1}^{i-1})$ 如何获得呢？可以由最大参数似然估计求得：

$$p(w_i | w_{i-n+1}^{i-1}) = f(w_i | w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^i)}{c(w_{i-n+1}^{i-1})},$$

where  $c(w_i^j)$  denotes the number of appearances of  $w_i^j$  in the corpus

但随之而来有一个问题，就是分子分母可能为零的问题，可以利用加一法或者减值法解决，核心就是调整最大似然估计的概率值，使零概率增值，使非零概率下调，消除领概率。

## 2. 方法：

首先看一下预测数据的格式：

一直以来，有不少家长误认为，在开车时将孩子放在儿童安全[MASK]上是最为安全的方式。[MASK]为需要填写的单词，那么基于语料库，我们可以尝试将所有的单词代入其中，算出句子的概率，而这个概率就代表着这个词语填入这个句子的可能性。但是每次代入词语后，算句子的概率，这种方法是非常慢的。根据公式，其实就是定义一个大小为 $n$ 的窗口，然后在预处理后的句子上滑动，每次算出框住的概率，最后乘起来。那么对于这个测试数据来看，如果 $n = 2$ ，那么我们只需要算出 $p([\text{mask}]|\text{安全})$ 和 $p(\text{上}|\text{[mask]})$ 即可，因为这两个值是根据填入的词语而决定，其他窗口算出来的概率是一定的。显然采用这种计算方法，可以节省许多时间。对 $n > 2$ 也是同理，当然就要考虑到越界的问题了，并不是每次都要算 $n$ 个概率值的，这取决于[mask]的位置。

## 3. 代码

1) 统计 $c(w_{i-n+1}^i)$ 和 $c(w_{i-n+1}^{i-1})$ ：

```
def count():
    for i in range(1, Text_Num + 1):
        with open(FileNme.format(i), 'r', encoding='utf-8') as f:
            # 读取每个文件内容
            lines = f.readlines()
            for line in lines:
                # 添加<BOS>和<EOS>
                line = '<BOS> ' + line + ' <EOS>'

                # 以n-gram形式分割
                res = ngrams(line.split(), N)
```

```
for each in res:
    # 公式中的分子
    if each in Dict_for_Up.keys():
        Dict_for_Up[each] += 1
    else:
        Dict_for_Up[each] = 1

    # 公式中的分母
    if each[:N - 1] in Dict_for_Down.keys():
        Dict_for_Down[each[:N - 1]] += 1
    else:
        Dict_for_Down[each[:N - 1]] = 1

print('Finish {d}...'.format(i))
```



```
# 写入文件
with open(Dict_Up_Path, 'w', encoding='utf-8') as f:
    for key, value in Dict_for_Up.items():
        f.write(' '.join(key) + ' ' + str(value) + '\n')
print('Finish writing Dict_for_Up...')

with open(Dict_Down_Path, 'w', encoding='utf-8') as f:
    for key, value in Dict_for_Down.items():
        f.write(' '.join(key) + ' ' + str(value) + '\n')
print('Finish writing Dict_for_Down...')
```

## 2) 预测函数:

```
def predict():
    # 添加词语, 使jieba能够正常分词
    jieba.add_word(['MASK'])
    # 记录预测对的个数
    accuracy = 0
    # 记录预测的答案e
    prediction = []

    with open(Test_Path, 'r', encoding='utf-8') as f:
        lines = f.readlines()
        # 正确答案在AnswerR里的索引
        index = 0
        for line in lines:
            # 记录词语对应的概率
            word_prob = dict()
            # print(line)

            # 分词
            sentence = jieba.lcut(line)
            mask_pos = sentence.index('MASK')
```

```
# 去非中文、去停止词
for i in range(len(sentence)):
    # 跳过MASK
    if i == mask_pos:
        continue
    sentence[i] = re.sub('[^\u4e00-\u9fa5]', '', sentence[i])
while '' in sentence:
    sentence.remove('')
sentence = [_word for _word in sentence if _word not in Stop_Word]

# n-gram预测
sentence.insert(0, '<EOS>')
sentence.append('<EOS>')
mask_pos = sentence.index('MASK')
for word in Word_Table:
    prob = 1.0
    # 将词语代入到MASK
    sentence[mask_pos] = word
```

```
for i in range(N):
    # 保证窗口不会越界
    if mask_pos - N + i + 1 > -1 and mask_pos + i < len(sentence):
        up = tuple(sentence[mask_pos - N + i + 1: mask_pos + i + 1])
        down = tuple(sentence[mask_pos - N + i + 1: mask_pos + i])
        # 分子值
        if up in Dict_for_Up.keys():
            up_num = float(Dict_for_Up[up] + 1)
        else:
            up_num = 1.0
        # 分母值
        if down in Dict_for_Down.keys():
            down_num = float(Dict_for_Down[down] + Word_Num)
        else:
            down_num = float(Word_Num)
        prob *= up_num / down_num
    word_prob[word] = prob
```





```
# 排序、取概率值最高的作为答案
word_prob = sorted(word_prob.items(), key=lambda x: x[1], reverse=True)
prediction.append(word_prob[0][0])

# 预测正确
if Answer[index] == word_prob[0][0]:
    print('answer:', Answer[index])
    print()
    accuracy += 1

index += 1

# 将预测答案写入文件
prediction = '\n'.join(prediction)
with open('./test_data/n-gram_prediction.txt', 'w', encoding='utf8') as f:
    f.write(prediction)

print('accuracy: {:.2%}'.format(float(accuracy) / float(len(Answer))))
```

#### 4. 预测结果（基于 1000 篇新闻，一共有 4 万多词语）

只输出预测正确的词语及正确率：

```
answer: 机器人
answer: 发展
answer: 运营商
answer: 宣布
answer: 用户
answer: 汽车
answer: 价格
answer: 媒体
answer: 套餐
answer: 营收
answer: 国家
answer: 速度
accuracy: 12.00%
```

$n = 2$

```
answer: 座椅
answer: 机器人
answer: 人类
answer: 建设
answer: 发展
answer: 运营商
answer: 媒体
answer: 套餐
answer: 营收
answer: 产品
accuracy: 10.00%
```

$n = 3$

```
answer: 机器人
answer: 人类
answer: 建设
answer: 运营商
answer: 媒体
answer: 套餐
answer: 营收
answer: 产品
accuracy: 8.00%
```

$n = 4$

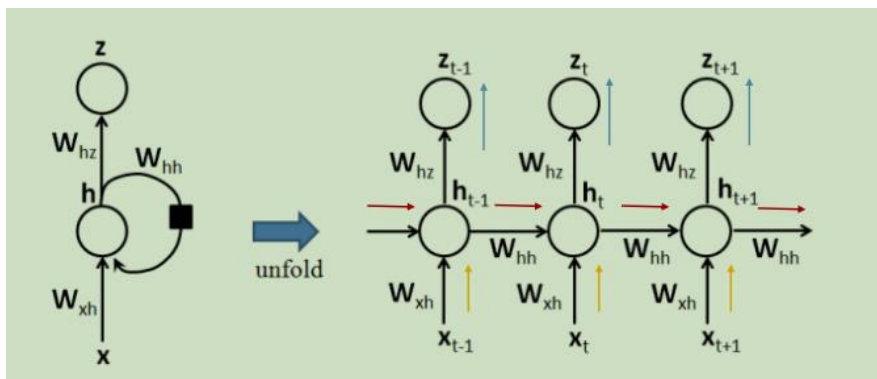
## 四. RNN 模型

### 1. 原理

1) RNN 基础介绍：



RNN 主要针对的是序列数据，通过神经网络在时序上的展开，找到样本之间的序列相关性。基本的 RNN 网络如图：



对于 RNN 中的其中一个神经元，它随着时序的展开，会结合之前的学习到的，以及新的输入，来更新自己的权重。神经元在时序上的迭代公式为(注意，在时序上共享权重)：

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad f \text{ is one kind of activation functions.}$$

$$z_t = \text{softmax}(W_{hz}h_t + b_z)$$

然后通过定义一个总损失函数，对权重进行求导，并且沿着梯度的反方向更新，这就是反向传播的过程，由于 RNN 是一个时序演化过程，所以对有些权重的求导会用到之前的输出，所以 RNN 的反向传播叫做 BPTT。

接下来我们写下求导的过程，还是利用上图，设在  $t$  时刻的损失函数与  $z_t$ （预测值）和  $y_t$ （真实值）有关，即  $loss = loss(z_t, y_t)$ 。那么对  $W_{hz}$  的求导为：

$$\frac{\partial loss}{\partial W_{hz}} = \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial W_{hz}}$$

对  $W_{hh}$  的求导为：

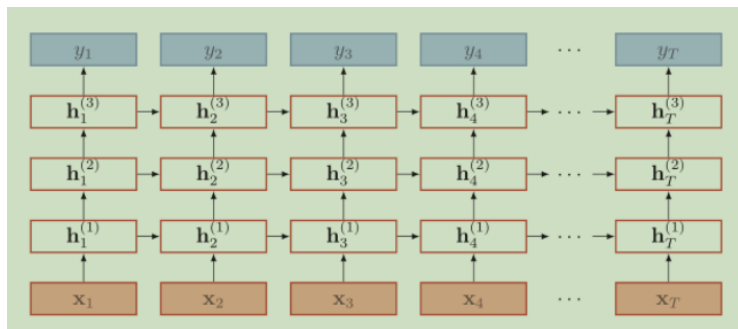
$$\frac{\partial loss}{\partial W_{hh}} = \sum_{i=0}^t \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial h_t} \left( \prod_{j=i+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_j}{\partial W_{hh}}$$

对  $W_{xh}$  的求导为：

$$\frac{\partial loss}{\partial W_{xh}} = \sum_{i=0}^t \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial h_t} \left( \prod_{j=i+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_j}{\partial W_{xh}}$$

## 2) 多种 RNN 模型：

A. 像全连接层一样，RNN 也可以多层堆叠，堆叠的 RNN 叫做(Stacked RNN). 模型如下：

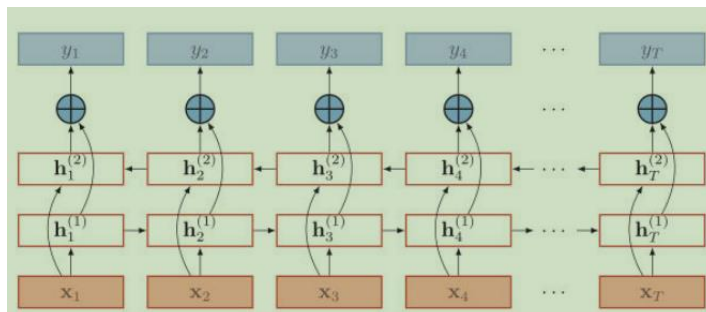




优点：加强网络的可表示性，即可以表示更多非线性关系；

缺点：层数过多则参数更难优化，并且容易出现梯度消失或梯度爆炸。

B. 双向 RNN(Bidirection RNN)，既然可以正方向训练，也可以反着训练：



优点：相比于单向的 RNN，双向的 RNN 显然可以发现更多的相关性。

3) 解决梯度消失或梯度爆炸的问题：

在 1) RNN 基础介绍中，我们写出了损失函数 $loss$ 对各个权重的求导公式，这里再搬过来：

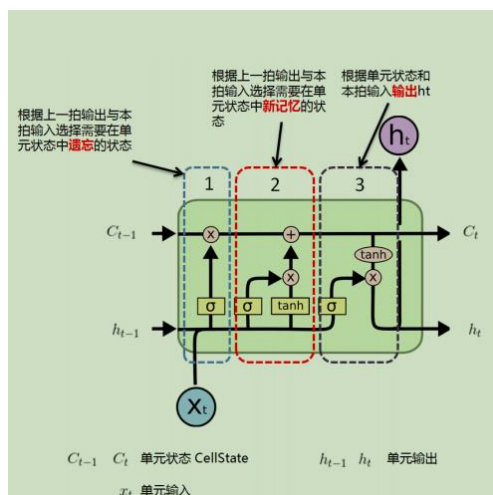
$$\frac{\partial loss}{\partial W_{hz}} = \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial W_{hz}}$$
$$\frac{\partial loss}{\partial W_{hh}} = \sum_{i=0}^t \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial h_t} \left( \prod_{j=i+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_j}{\partial W_{hh}}$$
$$\frac{\partial loss}{\partial W_{xh}} = \sum_{i=0}^t \frac{\partial loss}{\partial z_t} \frac{\partial z_t}{\partial h_t} \left( \prod_{j=i+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_j}{\partial W_{xh}}$$

$\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{t-1}} = \tanh' W$ 这个地方就容易出现个问题，如果 $\frac{\partial h_j}{\partial h_{j-1}}$ 过大，那 $\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{t-1}}$

就会很大，导致梯度爆炸，同理，如果 $\frac{\partial h_j}{\partial h_{j-1}}$ 过小，那 $\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{t-1}}$ 就会很小，导致梯

度消失。所以为了解决这个问题，就提出了两种改进的结构，不再是上面那张图的单个神经元那种结构，而是称作 LSTM 和 GRU 的结构，而 GRU 是 LSTM 的改进版，所以这里只介绍 LSTM。

LSTM 的结构如图：





图中 1、2、3 分别对应 LSTM 结构的遗忘门、输入门和输出门。

遗忘门根据上一个  $C_{t-1}$  选择当前单元状态的输入的每部分需要保留的程度，公式为：

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_{t1} = f_t * C_{t-1}$$

输入门先将输入通过两个激活函数，然后点乘获得当前产生的隐状态有多少需要保留并加入到下一个  $C_i$ ，公式为：

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_{t2} = i_t * \tilde{C}_t$$

此时可以更新细胞状态 (Cell State)：  $C_t = C_{t1} + C_{t2}$

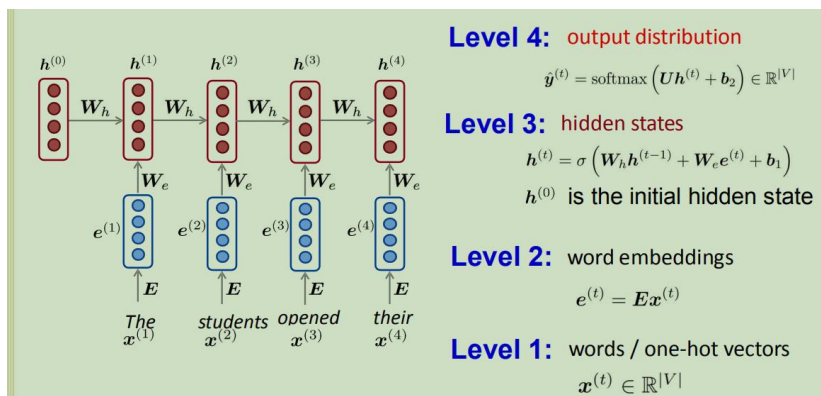
输出门将输出通过激活函数，再与新的细胞状态点乘得到当前单元状态的输出：

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

此时我们再来看下  $\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{t-1}}$ ，由于这里的  $\frac{\partial h_j}{\partial h_{t-1}} = \tanh' \sigma$ ，而这值不是 0 就是 1，

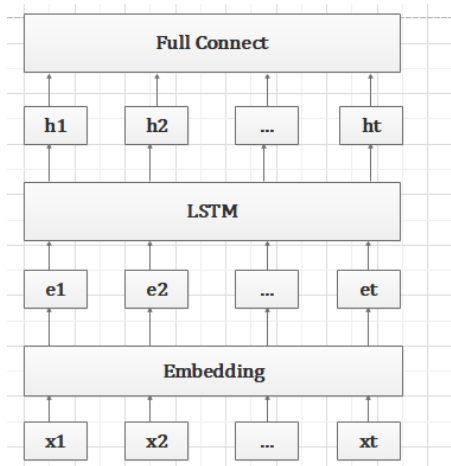
那么  $\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{t-1}}$  也是不是 0 就是 1，所以就解决了梯度消失或者梯度爆炸的问题。

## 2. 网络结构



- 1) 首先将句子中的每个词的 one-hot 表示  $x$  通过 Embedding 层，从高维降到低维，因为 one-hot 向量的长度是词表大小，而词表大小有几万，这个大小的向量是不可能直接进 RNN 的，内存是不允许的。而经过 Embedding 层可以降维到几十或者几百维，这可以自己尝试，看看哪种效果好；
- 2) 每个词经过 Embedding 层后降维成  $e$ ，然后进入 LSTM，至于 LSTM 的输出是几维，这也是自己试的。一开始我想让输出的维度直接是词表大小，这样之后容易算交叉熵，结果发现内存炸了，所以我还是设置 LSTM 的输出维度跟  $e$  的维度一样；
- 3) 由于 LSTM 输出的维度不是词表大小，又为了能够做交叉熵，所以对于 LSTM 每个节点的输出，都要经过一个全连接层，升维到词表大小，然后做交叉熵；

以下是网络结构图：



### 3. 代码

- 1) 建立词到索引以及索引到词之间的字典，方便之后的 Embedding 以及通过神经网络预测出来的向量找到词：

```
def build_vocab():
    # 建立词到id的映射
    word_to_id = {}
    id_to_word = {}

    # word -> id
    with open(Table_Path, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.split()
            word_to_id[line[1]] = int(line[0]) + 1
    word_to_id['Unknown'] = 0

    # id -> word
    for key, value in word_to_id.items():
        id_to_word[value] = key

    return word_to_id, id_to_word
```

- 2) 设计一个类，功能如下：

- A) 将训练文本中的每个句子转化为向量，并记录每个句子的实际长度，然后把它们补到一样的长度。假设句向量长度为 $l$ ，则将句向量前 $l-1$ 的向量作为输入，将后 $l-1$ 的向量作为输出；
- B) 实现批量读取；

```
class Train_Text2Vec:
    def __init__(self, train_file_name, train_text_num, num_word):
        """
        :param train_file_name: 训练文本的位置
        :param train_text_num: 训练文本的数量
        :param num_word: 一个句子实际长度，超过就截断，不足就padding
        """

        self.num_word = num_word          # 一个句子的最大长度
        self.input = []                   # 网络的输入
        self.output = []                  # 网络的输出
        self.length = []                  # 输入向量的实际长度
        self.mask = []                    # 长度跟输入一样，有实际输入的位置置一，padding的位置上置零
        self.start_read = 0               # 开始读取的位置
        self.train_num = 0                 # 训练样本即句子的数量

        self.transform(train_file_name, train_text_num)  # 将训练文本转成向量
        self.make_mask()                               # 制作self.mask
```



```
def transform(self, train_file_name, train_text_num):
    """
    :param train_file_name: 训练文本的位置
    :param train_text_num: 训练文本的数量
    """
    for i in range(1, train_text_num):
        with open(train_file_name.format(i), 'r', encoding='utf-8') as f:
            for line in f:
                line = line.split()
                sen2vec = [Word2Id[word] if word in Word2Id.keys() else 0 for word in line] # 句向量

                # 多删少添
                l = len(sen2vec)
                if l == 1: # 句子只有一个词，舍去
                    continue
                if l < self.num_word: # 少添
                    sen2vec += [0] * (self.num_word - l)
                    self.length.append(l - 1)
                else: # 多删
                    sen2vec = sen2vec[:self.num_word]
                    self.length.append(self.num_word - 1)
                self.input.append(sen2vec[: self.num_word - 1]) # 句子的前1-1个词为输入

                self.output.append(sen2vec[1: self.num_word]) # 句子的后1-1个词为输出

            self.train_num = len(self.input)

    def make_mask(self):
        for i in range(self.train_num):
            # 实际输入的位置置一，padding位置置零
            t = [1] * self.length[i] + [0] * (self.num_word - 1 - self.length[i])
            self.mask.append(t)

    def next_batch(self, batch_size):
        """
        :param batch_size: 批量读取大小
        :return: 批量数据
        """
        end_read = (self.start_read + batch_size) % self.train_num # 结束的位置
        if self.start_read > end_read:
            input_batch = self.input[self.start_read:] + self.input[: end_read]
            output_batch = self.output[self.start_read:] + self.output[: end_read]
            length_batch = self.length[self.start_read:] + self.length[: end_read]
            mask_batch = self.mask[self.start_read:] + self.mask[: end_read]
        else:
            input_batch = self.input[self.start_read: end_read]
            output_batch = self.output[self.start_read: end_read]
            length_batch = self.length[self.start_read: end_read]
            mask_batch = self.mask[self.start_read: end_read]

        self.start_read = end_read # 更新下次读取的位置

        return input_batch, output_batch, length_batch, mask_batch
```

3) 设计一个类，功能如下：

- A) 将测试文本分词、去燥，然后转化为向量，再做多删少补的操作，使得句向量的长度与训练时的长度一样；
- B) 返回测试数据；





```
class Test_Text2Vec:
    def __init__(self, test_file_name, num_word):
        """
        :param test_file_name: 测试文本位置
        :param num_word: 句子最大长度
        """
        self.input = [] # 网络的输出
        self.length = [] # 句向量实际长度
        self.num_word = num_word # 一个句子的最大长度
        self.test_num = 0 # 测试数据即句子的数量

        self.transform(test_file_name, num_word) # 将测试文本转成向量
```

```
    def transform(self, test_file_name, num_word):
        """
        :param test_file_name: 测试文本位置
        :param num_word: 句子最大长度
        """
        jieba.add_word('[MASK]')
        with open(test_file_name, 'r', encoding='utf-8') as f:
            for line in f:
                # 分词、定位MASK位置
                sentence = jieba.lcut(line)
                mask_pos = sentence.index('[MASK]')
                sentence = sentence[:mask_pos] # 截断

                # 去非中文、去停止词
                for i in range(mask_pos):
                    sentence[i] = re.sub('[^\u4e00-\u9fa5]', '', sentence[i])
                while '' in sentence:
                    sentence.remove('')
                sentence = [_word for _word in sentence if _word not in Stop_Word]

                sen2vec = [Word2Id[word] if word in Word2Id.keys() else 0 for word in sentence] # 句向量
                l = len(sen2vec)
                if l > num_word: # 太长，取MASK前num_word个词
```

```
                    sen2vec = sen2vec[-num_word:]
                    self.length.append(num_word)
                else: # 短 补零
                    sen2vec += [0] * (num_word - l)
                    self.length.append(l)

                self.input.append(sen2vec)

        self.test_num = len(self.length)

    def get_test_data(self):
        """
        :return: 测试数据
        """
        output = np.zeros([self.test_num, self.num_word]) # 只为了feed占位符，无意义
        mask = np.zeros([self.test_num, self.num_word]) # 只为了feed占位符，无意义
        return self.input, output, self.length, mask
```

#### 4) tensorflow 实现 RNN 模型:



```
class LSTM_Model:
    def __init__(self, embedding_size, nstep, batch_size):
        """
        :param embedding_size: embedding后词语的向量维度
        :param nstep: LSTM的时间节点个数
        :param batch_size: 每次喂入网络数据的数量
        """
        self.accuracy = 0 # 当前最高准确率
        self.vocab_size = len(Word2Id) # 词表大小
        self.embedding_size = embedding_size
        self.nstep = nstep
        self.nhidden = embedding_size
        self.batch_size = batch_size
        self.global_step = tf.Variable(0, trainable=False) # 训练了多少次了

        self.learning_rate = tf.compat.v1.train.exponential_decay(Learning_Rate_Base,
                                                                    self.global_step,
                                                                    100,
                                                                    Learning_Rate_Decay,
                                                                    staircase=True) # 指数衰减学习率

    self.x = tf.compat.v1.placeholder(shape=[self.batch_size, self.nstep], dtype=tf.int32) # 输入
    self.y = tf.compat.v1.placeholder(shape=[self.batch_size, self.nstep], dtype=tf.int32) # 输出
    self.length = tf.compat.v1.placeholder(shape=[self.batch_size], dtype=tf.int32) # 输入的实际长度
    self.mask = tf.compat.v1.placeholder(shape=[self.batch_size, self.nstep], dtype=tf.float32) # 权重矩阵
    self.keep_prob = tf.compat.v1.placeholder(dtype=tf.float32) # dropout

    # embedding_layer
    self.embedding_mat = tf.Variable(tf.compat.v1.random_uniform([self.vocab_size, self.embedding_size], -1, 1))
    self.inputs = tf.nn.embedding_lookup(self.embedding_mat, self.x)
    self.dinputs = tf.nn.dropout(self.inputs, keep_prob=self.keep_prob)

    # lstm_layer
    self.lstm_cell = rnn.BasicLSTMCell(self.nhidden, forget_bias=1.0, state_is_tuple=True)
    self.lstm = rnn.DropoutWrapper(self.lstm_cell, output_keep_prob=self.keep_prob)
    self.initial_state = self.lstm.zero_state(self.batch_size, dtype=tf.float32)
    self.output, _ = tf.nn.dynamic_rnn(self.lstm, self.dinputs, initial_state=self.initial_state, dtype=tf.float32,
                                       time_major=False) # sequence_length=self.length,

    self.reshape_output = tf.reshape(self.output, [self.batch_size*self.nstep, self.nhidden])

    # full_connect
    self.w = tf.Variable(tf.compat.v1.truncated_normal([self.nhidden, self.vocab_size]), dtype=tf.float32)
    self.b = tf.Variable(tf.zeros([self.vocab_size]))
    self.prob = tf.matmul(self.reshape_output, self.w) + self.b
    self.reshape_prob = tf.reshape(self.prob, [self.batch_size, self.nstep, self.vocab_size])

    # loss
    self.loss = tf.contrib.seq2seq.sequence_loss(
        self.reshape_prob,
        self.y,
        self.mask,
        average_across_timesteps=False,
        average_across_batch=True)
    self.mloss = tf.reduce_mean(self.loss)

    # train
    self.cost = tf.reduce_sum(self.loss)
    tvars = tf.trainable_variables()
    grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tvars), 5) # 梯度裁剪 防梯度爆炸
    self.train = tf.compat.v1.train.AdamOptimizer(self.learning_rate).apply_gradients(
        zip(grads, tvars), global_step=self.global_step)

def test(self, sess, x, y, l, m):
    """
    :param sess: 当前会话
    :param x: 测试数据的输入
    :param y: 仅为feed占位符, 无意义
    :param l: 输入的实际长度
    :param m: 仅为feed占位符, 无意义
    :return: 在测试集上准确率
    """
    accuracy = 0 # 本次预测的准确率
    predictions = [] # 本次预测的结果
    size = len(x) # 本次预测的数量
    output = sess.run(self.reshape_prob, feed_dict={self.x: x, self.y: y, self.length: l,
                                                    self.mask: m, self.keep_prob: 1.0}) # 获取预测结果

    for i in range(size):
        prediction = np.argmax(output[i, 1[i]-1, :]) # 最大可能的词语的id
        prediction = Id2Word[prediction] # 通过id获取词语
        predictions.append(prediction)
        # print(prediction, Answer[i])
        if prediction == Answer[i]: # 预测正确
            print(prediction)
            accuracy += 1
```



```
if accuracy > self.accuracy:          # 大于当前准确率
    self.accuracy = accuracy          # 更新最高准确率
    predictions = '\n'.join(predictions) # 更新预测
    with open('./test_data/rnn_prediction.txt', 'w', encoding='utf-8') as f:
        f.write(predictions)

return float(accuracy) / float(size)
```

5) 主函数:

```
if __name__ == '__main__':
    # 建立映射
    Word2Id, Id2Word = build_vocab()

    # 训练文本转换为向量
    train_data = Train_Text2Vec(Train_File_Name, Train_Text_Num + 1, nStep + 1)

    # 预处理测试文本并转化为向量
    test_data = Test_Text2Vec(Test_File_Name, nStep)

    # 获取预测数据
    test_x, test_y, test_l, test_m = test_data.get_test_data()

    # 加载停止词
    with open(StopWord_Path, 'r', encoding='utf-8') as f:
        for line in f:
            Stop_Word.append(line.strip())
```

```
# 加载答案
with open(Answer_Path, 'r', encoding='utf-8') as f:
    for line in f:
        Answer.append(line.split()[0])

# 声明模型
model = LSTM_Model(Embedding_Size, nStep, Batch_Size)
# 建立会话
with tf.compat.v1.Session() as sess:
    # 初始化
    sess.run(tf.compat.v1.global_variables_initializer())
    # 训练Train_Step次
    for i in range(Train_Step + 1):
        input_batch, output_batch, length_batch, mask_batch = train_data.next_batch(Batch_Size)
        _, _loss, step = sess.run([model.train, model.mloss, model.global_step],
                                   feed_dict={model.x: input_batch, model.y: output_batch,
                                               model.length: length_batch, model.mask: mask_batch,
                                               model.keep_prob: 1.0})
```

```
if i % 100 == 0:
    print('Step: {:d}, Loss {:.4f}'.format(step, _loss))
    print('Accuracy: {:.2%}'.format(model.test(sess, test_x, test_y, test_l, test_m)))
    print()
```

#### 4. 预测结果（基于 1000 篇新闻，一共有 4 万多词语）

只显示预测正确的词语及正确率（只展示最好的结果）:

Step: 1601, Loss 7.157462

手机  
机器人  
厂商  
成本  
宣布  
汽车  
时间  
媒体  
套餐

Accuracy: 9.00%



由于训练数据较少，loss 一直降不下去，所以准确率也一直在 9.0%附近波动。

## 五. 总结与思考

通过本次实验，我学习到了多方面的知识，例如爬虫、n-gram 语言模型及神经网络等。

一开始我的爬虫水平仅限于爬取网页的静态代码，完全不知道网页的动态部分怎么爬取，通过学习，我学会找到 api 接口来获取动态部分的信息，所以爬虫水平还是提高了一点。

至于数据预处理部分，由于时间关系，没有来得及比较，去不去停止词哪个效果更好，所以在数据预处理部分，并没有什么收获。

在写 n-gram 过程中，一开始我是将词表中每个词代入进去[MASK]，求整个句子的概率，最后通过比较，找到代入后句子概率最大的词语，将其选为预测结果，但是这个需要计算的时间太长了，所以为了缩短时间，通过观察计算公式，发现可以只计算[MASK]附近的概率即可获得整个句子的相对概率，具体操作可以看报告的对应该部分（三. RNN 模型）。速度提升也是肉眼可见的，本来半个小时都计算不完，现在只需要 2 分钟不到就可以计算完了。奇怪的是当  $n = 2$  时，正确率是大于  $n = 3$  和  $n = 4$  的，但正常来讲， $n = 4$  的正确率应该是大于  $n = 3$  的，这可能是数据太少的原因吧。

至于 RNN 神经网络模型，之前在 AI 的大作业已经学习了，所以本次搭建神经网络的工作也比较轻松，但调参永远都是做神经网络过程中最耗时的，果然，调了好多天，甚至都不能使得 loss 降到 1（loss 的定义是如 ppt 中说的每个节点输出与真实值的交叉熵），loss 一直都在 4、5、6 之间波动，准确率也一直都是 6.0%、7.0%、8.0%和 9.0%之间波动，暂且还是归咎于数据太少的原因吧。

一开始没做 n-gram 和 RNN 之前，我觉得 RNN 预测的准确率绝对会比 n-gram 预测的准确率高，结果出来确实大跌眼镜，2-gram 预测的准确率可以“高达”12.0%，而 RNN 预测的准确率最多只有 9.0%。事实证明，在小样本的前提下，数据驱动的神经网络确实不如模型驱动的 n-gram，所以在样本不足的前提下，模型建的好才是最重要的。

本次实验也让我见识到了自然语言处理的难处，确实需要建立在大数据的基础上，才能训练出差强人意的模型。同时大数据带来的问题是，需要大量的计算，需要足够的算力。

## 六. 参考资料

1. <https://www.cnblogs.com/bonelee/p/10475453.html>
2. <https://zhuanlan.zhihu.com/p/28749444>
3. <https://blog.csdn.net/coderTC/article/details/73864097>
4. <https://blog.csdn.net/luoxuexiong/article/details/90345143>
5. Tang Shancheng ; Bai Yunyue ; Ma Fuyu. A Semantic Text Similarity Model for Double Short Chinese Sequences. IEEE, 2018