自然语言处理

期中大作业

姓名：郭俊楠　学号：17341045　时间：2019 年 11 月 20 日

1　爬取新闻数据

本项目中需要预先爬取科技新闻数据，所以我选定了新浪科技频道滚动新闻作为爬取对象。如图 1、2 所示，新浪滚动科技新闻的页面比较规范，使用翻页设计，新闻链接在代码中的分布很有规律，有利于我们爬取新闻。具体实现上，我主要使用了 selenium 库来模拟浏览器行为，这样做比较不容易被服务器发现爬虫行为；另外还使用了 BeautifulSoup 解析页面内容和获取新闻标题及相应的链接。

图 1: 新浪科技频道滚动新闻页面

在获取新闻链接后，便可以直接通过链接来请求页面内容。如图 3 所示，在源码中正文的起始和结束都比较有特点，特别是“新增众测推广文案”等字样的出现。经过观察，几乎所有近期的新闻页面源码都有类似的特点，于是我们可以通过正则表达式筛选出正文部分，并得到

图 2: 源码中的新闻标题和链接部分

包含正文的那部分源码。但是，此时，我们还需要进一步过滤掉 html 标签的数据噪音；最后，如图 4 所示，我们便可以得到较为纯净的新闻正文数据了。

图 3: 源码中的正文部分

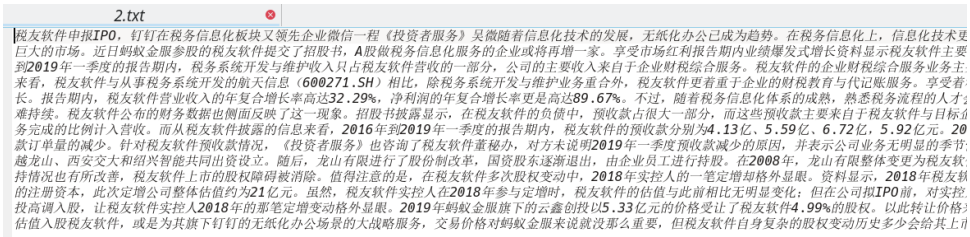
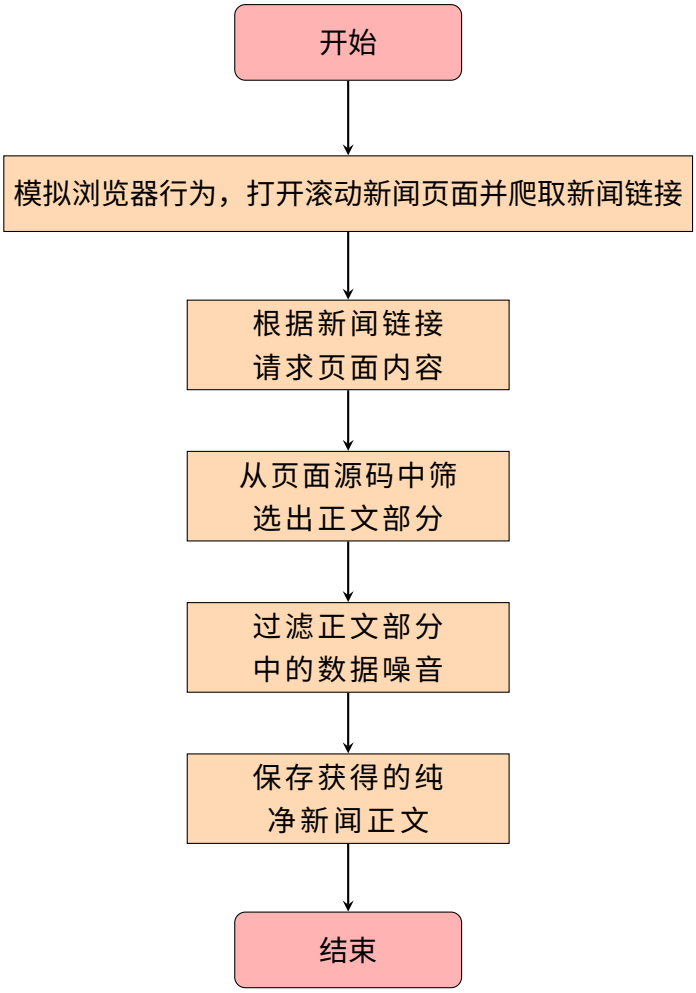


图 4: 爬取得到的新闻正文示例

新闻爬取的总流程图如下所示。

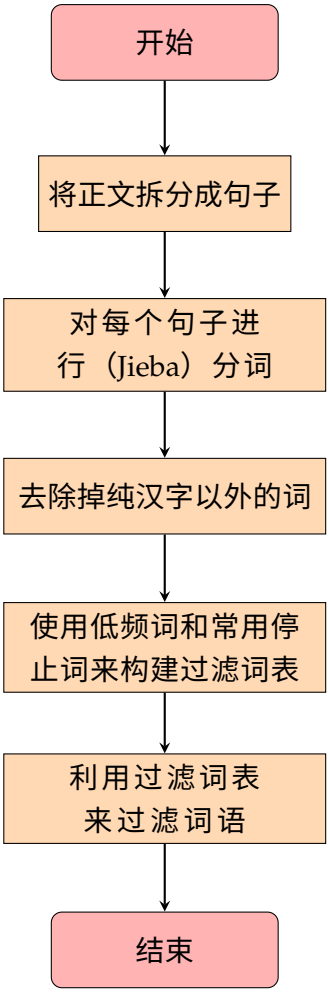


2 数据预处理

在进行数据预处理前，我们先要明确要解决什么任务、建立什么样的模型以及需要什么样的数据。

简单地说，在给定缺词句子的情况下，我们需要建立语言模型来预测所缺少的那个词是什么。这里我们选择的语言模型是 3-gram 模型和 LSTM 模型。很显然，我们处理的是中文句子，所以不妨将所猜测的词限定在中文词的范围内，去除掉符号、数字和英文等，只保留中文，这样不仅使得模型的更加细化，减小训练压力，还可以让我们的数据更加地纯净。此外，我们还可以考虑去除低频词，因为低频词的语料数据量要远小于其他，在训练模型的时候可能更多的是作为噪声而起作用。而通过检查可以发现，原始的语料库中总词数为 619027，其中常用停止词有 129788 个，如果去除停止词的话，总词数大约下降 $1/4$，语料库中的信息密度会增大，且（一般认为）不会对模型最终的性能造成太大的影响。

综上所述，数据预处理的流程如下。



最后，我们得到的语料如图 5 所示（与图 4 对比），显然预处理后我们还是能大致猜出句子原意的，这说明预处理前后信息的损失不大。而除了对训练集进行上述的处理外，我们也要对测试集进行相同的处理，并保存。同时为了方便处理，我们还可以对处理过后的数据进行统计，得出一个词典并储存，以便之后训练模型时可以直接使用。

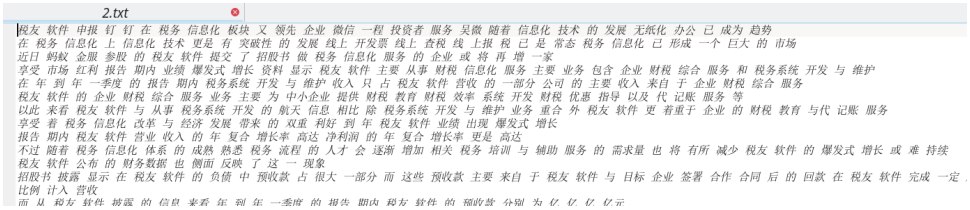


图 5: 数据预处理后的图 4 正文

3 模型实现与训练

3.1 3-gram

3.1.1 模型建立与训练

这里我们要先建立一个 3-gram 模型。

注意到前面我们对数据进行了预处理，并保存为一个个文本文件，每个文件包含若干个以

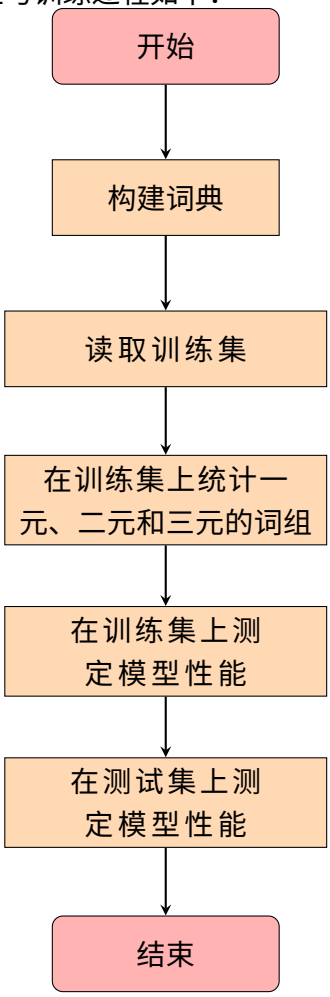
词语列表形式储存的句子。同时，我们还有一个词典文件，经过简单的处理就可以直接使用。值得注意的是，训练语料中所有的词都包含在词典里，而测试集语料却不一定，所以我们为未知的词预留了序号 0，以防之后的测试过程出错。

在建立模型前，我们还要考虑好数据平滑问题，即：测试集中有我们训练过程中不曾见过的词，应该怎么处理。这里我们采用回退策略，即一开始使用 3-gram，如果出现平滑问题，则转为使用 2-gram，要是问题还存在，就使用 1-gram。在具体实现上，我们更多考虑的是相关词是否包含在词典里，而至于词组是否在训练集中出现倒显得无关紧要。回退策略描述如下：

Algorithm 1 Backoff Strategy

procedure BACKOFF(w_1, w_2)
  if w_1, w_2 in $word\_dict$ **then**
    choose $\max_{\hat{w}}(ternary\_count[w_1 w_2 \hat{w}])$
  else if w_2 in $word\_dict$ **then**
    choose $\max_{\hat{w}}(binary\_count[w_2 \hat{w}])$
  else
    choose $\max_{\hat{w}}(unary\_count[\hat{w}])$

为了配合回退策略，我们需要分别对一元、二元、和三元的词组进行统计。于是，模型的建立与训练过程如下：



在使用比较大的数据集测定模型性能之前，我们不妨先用一个小例子测试一下模型是否有效。如图 6 所示，我们选定训练集的中“美国”、“东部”和“时间”这个词组，并将“美国”和

“东部”输入到模型。结果如图 7 所示，模型正确预测出了“时间”这个词。

图 6: 3-gram 模型测试样例

```
predict_word = num2word[predict_number]
print(predict_word)
```

```
时间
```

图 7: 3-gram 模型样例测试结果

3.1.2 模型评价

下面我们展示模型在训练集和测试集上的评价结果。

结果如图 8 所示，模型在训练集上的表现比较好，准确率能达到 0.83 左右，这是符合我们预期的，因为 3-gram 是基于统计的语言模型，在经过有效的数据预处理和模型的一些细节处理后，能在生成该模型的数据集上达到一个较好的表现结果是很正常的；而模型在测试集上的表现则不尽如意，准确率只有 0.02，这有可能是因为目前的模型效果太差，也有可能是因为测试集中要猜测的词原本就不在词典内，即超出模型的预测范围并导致预测错误是一个必然的结果。于是，我们对测试集进行检查，结果如图 9 所示。可以发现，理论上模型有可能猜对的只有 36 个，换句话说，模型在测试集上的准确率也可以说是 $\frac{2}{36} = 0.06$。于是，我们可以认为前面提到的两个原因都是存在的：测试集上真正可以用来测试的只有 $\frac{1}{3}$，而在测试集中删去了不可能预测正确的那部分文本，模型的表现仍然不佳。

```
# meowjolan @ meowjolan in ~/Documents/NLP/中期大作业 [9:45:55] C:1
$ python -u "/home/meowjolan/Documents/NLP/中期大作业/N-gram.py"
On training set:
Total: 967269, Correct: 799516, Accuracy: 0.8265704783260913
On test set:
Total: 100, Correct: 2, Accuracy: 0.02
```

图 8: 3-gram 模型评价结果

3.2 单向 LSTM

3.2.1 模型与训练过程设计

这里我们分析一下要建立的 LSTM 模型。

```
words = [s_list[i][ans_pos[i]] for i in range(len(s_list))]
result = [w in word2num for w in words]
print('All: {}, In dict: {}'.format(len(result), sum(result)))
```

```
All: 100, In dict: 36
```

图 9: 测试集的检查结果

我们需要一个能根据上下文预测词语的语言模型，不妨先考虑仅根据上文来预测，所以我们首先使用单向的 LSTM 模型；同时，为了最后能计算出词语概率，我们可以在 LSTM 层之后多加一两层全连接层，并在输出层使用 $Softmax$ 进行激活；而 LSTM 层接收的是词向量序列，所以我们也需要在 LSTM 层前加上词嵌入层。于是，我们最终建立的模型如图 10 所示。

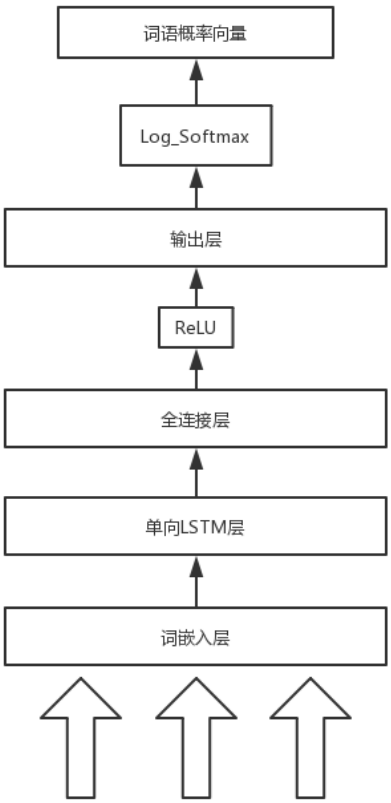


图 10: 单向 LSTM 模型

在训练过程中，当我们接收一个句子时，我们不会丢弃 LSTM 层各个单元的输出，因为那些都是有用的训练语料。这样，模型最后输出的将是整个句子中各个位置的词语概率向量，而通过选取概率最大的词作为我们的预测结果，我们便可以得到一个预测得出的句子；将该句子与原句子进行对比并计算损失，之后进行反向传播，这大致就是我们的一个训练示例。

最后，我们要观察模型在测试集上的表现。这里值得注意的是，我们不再需要预测得出的某个句子，而是其中某个位置的词。

3.2.2 进行初步尝试

为了检测模型实现是否正确，我们可以用一些超参数进行测试。我们选定的超参数如图 11 所示，而相应的损失曲线、训练结果和测试结果如图 12、图 13 和图 14 所示。

```
# 定义超参数
# 一批数据包含多少个句子
batch_size = 400
# lstm层节点数
lstm_dim = 192
# 全连接层节点数
h1_dim = 256
h2_dim = 256
# 学习率
lr = 1e-3
# 训练次数
epoch_num = 6000
# 丢弃比例
dropout = 0.2
# lstm层数
num_lstm_layers = 2
# 词向量维度
embedding_dim = 300
```

图 11: 单向 LSTM 超参数 1

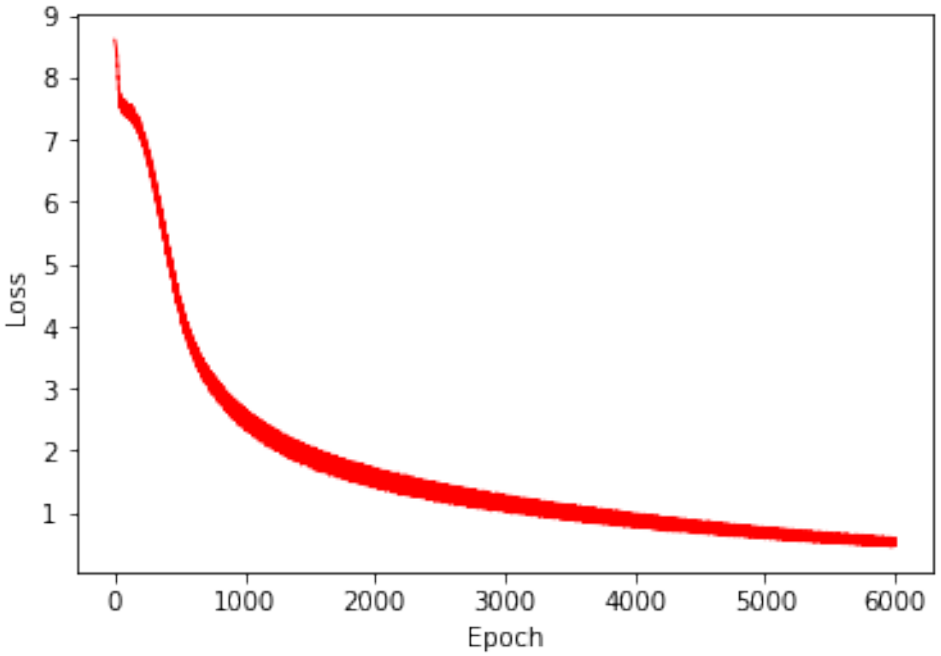


图 12: 单向 LSTM 模型损失曲线 1

可以看见，模型大约在第 5000-6000 次训练的时候已经差不多收敛；训练集上的准确率达到了 0.86 左右，比之前的 3-gram 有了一些提升；测试集上的准确率仍为 0，这方面比之前的 3-gram 要差。

```
print("Total: {}, Correct: {}, Accuracy: {}".format(total_count, correct_count, int(correct_count)/int(total_count)))
```

```
Total: 364435, Correct: 314431, Accuracy: 0.8627903467010578
```

图 13: 单向 LSTM 模型训练结果 1

```
correct_count += torch.sum(predictions == targets)

print("Total: {}, Correct: {}, Accuracy: {}".format(len(predictions), correct_count, int(correct_count)/len(predictions)))
```

```
Total: 100, Correct: 0, Accuracy: 0.0
```

图 14: 单向 LSTM 模型测试结果 1

3.2.3 最终结果

很显然，我们的模型还有提升空间，于是我试验了多组超参数，其中表现较好的一组如图 15 所示，相应的损失曲线、训练结果和测试结果如图 16、图 17 和图 18 所示。

```
[ ]  # 定义超参数
     # 一批数据包含多少个句子
     batch_size = 400
     # lstm层节点数
     lstm_dim = 256
     # 全连接层节点数
     h1_dim = 256
     h2_dim = 512
     # 学习率
     lr = 1e-3
     # 训练次数
     epoch_num = 6000
     # 丢弃比例
     dropout = 0.2
     # lstm层数
     num_lstm_layers = 2
     # 词向量维度
     embedding_dim = 300
```

图 15: 单向 LSTM 超参数 2

可以看到，我们增加了 LSTM 层的单元数和输出层的节点数，而结果则是损失曲线在第 4000-5000 次训练时大致收敛，比初次的尝试收敛更快；在训练集上的准确率为 0.96，比初次尝试有了很大的提升；而测试集上的结果与初次尝试一样，准确率仍为 0.

3.3 双向 LSTM

3.3.1 模型与训练过程设计

前面我们实现的模型中，不论是 3-gram 还是单向 LSTM，都只是依靠上文信息来预测，而忽略了下文信息。所以我们可以考虑双向 LSTM 模型，将下文信息也进行有效利用。

为了增强对比性，我们沿用单向 LSTM 模型最后的超参数，并在原有的模型上增加了与原来的单向 LSTM 层并行的另一个单向 LSTM 层，以便处理反向文本，模型组成如图 19 所示。值得注意的是，两个 LSTM 层的输出并非简单的拼接，而是需要先经过一定的变换。简单来说，两个 LSTM 层拼接后，序列位置 i 的输出一半为正向 LSTM 层中单元 $i-1$ 的输出，另一半为反向 LSTM 层中单元 $i+1$ 的输出（假设反向 LSTM 层中单元顺序也是反向的）。这样就导致模

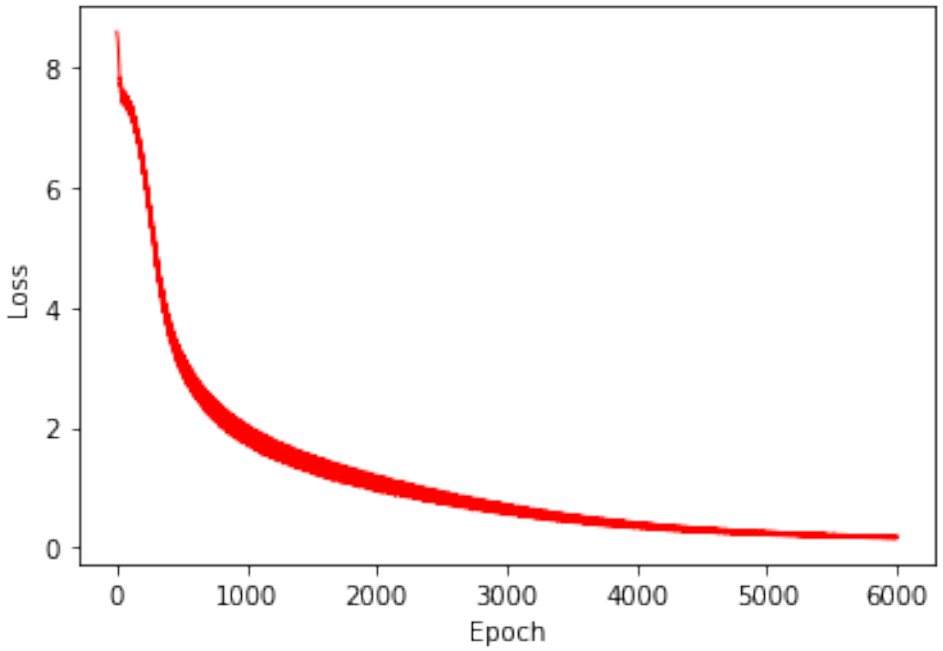


图 16: 单向 LSTM 模型损失曲线 2

```
print("Total: {}, Correct: {}, Accuracy: {}".format(total_count, correct_count, int(correct_count)/int(total_count)))
```

```
Total: 364435, Correct: 349311, Accuracy: 0.9585001440586113
```

图 17: 单向 LSTM 模型训练结果 2

型输出的序列中位置 i 部分与输入序列中位置 i 是无关的，即模型输出的序列中位置 i 部分用来预测输入序列中位置 i 的词语。

运行代码，训练模型后，损失曲线如图 20 所示。可以看到，双向 LSTM 模型在单向 LSTM 之上有了很大的提升。原本的单向 LSTM 大约在第 5000-6000 次训练时收敛，而现在双向 LSTM 模型在第 2000 次训练时已经大致收敛，且两者的收敛损失值相差不大。

对双向 LSTM 模型进行评价，结果如图 21、图 22 所示。在训练集上的准确率为 0.93 左右，与单向 LSTM 模型相差不大；在测试集上的准确率为 0, 与单向 LSTM 模型相差不大。

综上所述，虽然目前双向 LSTM 模型与单向 LSTM 模型的评价表现相差不大，但双向 LSTM 模型收敛得很快，其收敛速度几乎是单向 LSTM 模型的三倍。

4 模型测试

前面我们对每个模型都在测试集上进行了总体的测试，但却一直保持低准确率。为了更深入地了解模型在测试集上的真实效果，我们不妨找个测试样例来观察。如图 23、图 24、图 25 所示，我们选定了一个问题，并观察 3-gram 模型和双向 LSTM 模型给出的结果。可以看见，标准答案填入问题后，整个句子显得很完整。

```
print("Total: {}, Correct: {}, Accuracy: {}".format(len(predictions), correct_count, int(correct_count)/len(predictions)))
```

```
Total: 100, Correct: 0, Accuracy: 0.0
```

图 18: 单向 LSTM 模型测试结果 2

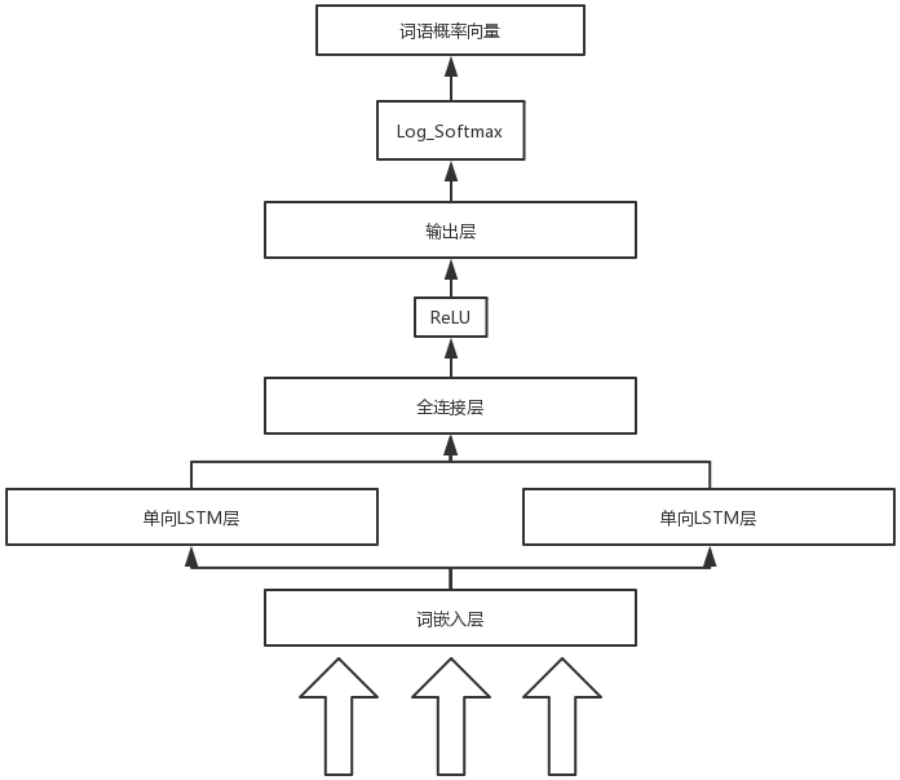


图 19: 双向 LSTM 模型

3-gram 模型结果填入句子后，句子意义不明，但若只看 3-gram 模型结果与前几个词的搭配效果，即“但大多数是”，却能发现是有意义的词组。这一点是因为 3-gram 至多考虑前三个词，而句子的其他词都无法考虑，所以填词后整个句子有问题是正常的。而要想比较客观地评价 3-gram 模型，应该用词组进行测试，因为使用句子的话，本质上还是在用词组进行测试，只不过句子的其他成分对答案进行了很强的约束，这很容易造成准确率很低的表象。

双向 LSTM 模型结果填入句子后，发现整个句子的意思还是能让人大概明白的，但是有一点的违和感。换句话说，将双向 LSTM 模型结果填入句子可以说是合适的，但没有标准答案那么契合。在进行总体测试时，双向 LSTM 模型的准确率为 0，但经过样例分析后我们却可以发现，模型的表现并没有看上去那么糟糕，这也说明了目前的测试方法还是略有不妥的。

5 总结与思考

5.1 pytorch 的 BiLSTM

前面我们提到了双向 LSTM 模型，如果直接使用 torch 相关的库来实现双向 LSTM 层，会在拆分正、反向输出上遇到麻烦，因为无法确定哪一部分是正向、哪一部分是反向，而在官方文档上也没有相关的说明。于是，这里我们使用了两个独立并列的单向 LSTM 层来模拟双向 LSTM 层。值得的注意的是，一个单向 LSTM 层接收正向序列，另一个单向 LSTM 层则是接收反向序列。

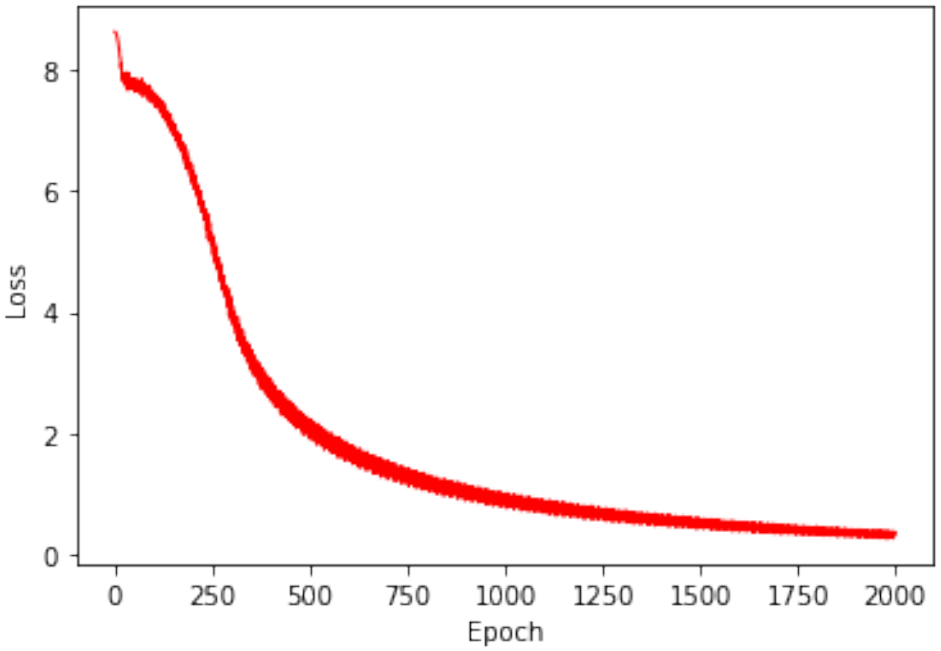


图 20: 双向 LSTM 模型损失曲线

```
print("Total: {}, Correct: {}, Accuracy: {}".format(total_count, correct_count, int(correct_count)/int(total_count)))
```

```
Total: 364435, Correct: 338818, Accuracy: 0.9297076296184504
```

图 21: 双向 LSTM 模型训练结果

5.2 词频统计与内存限制

我们使用了 3-gram 模型，需要对三元词组进行统计，但如果我们单纯使用数组的方式来储存统计结果，会发现完全不可行。这里的词典中词的个数为 6000 以上，如果以纯数组来保存三元统计结果，假设每个元素为 1 个字节，那么最后需要的空间为 216GB 以上，这明显不在我们的承受范围之内。

如果使用字典的方式来储存，即将一个三元词组映射到相应频数，就会需要大量的插入操作，会在统计过程中浪费很多时间。

最后，我才用了字典加数组的方式来综合两者的优缺点，即将一个二元词组（三元词组的前两个词）映射为一个数组，该数组表示第三个位置可能出现的词以及相应的频数。采用这种储存方式，一般是不会占用超过 1GB 空间的，而且速度也在我们的接受范围之内。

从这里，我们也可以认识到：在解决问题的过程中，一些细节是不可忽略的。如果我们随便使用一个结构来储存结果，却占用了 4GB 的空间，那么显然不会报错，但这种空间资源的占用让人难以接受，而此时其表现形式仅仅为代码运行过程变慢，可能让人难以察觉原因，只是百思不得其解。注重细节的设计，能让我们少走很多弯路。

```
print("Total: {}, Correct: {}, Accuracy: {}".format(len(predictions), correct_count, int(correct_count)/len(predictions)))
```

```
Total: 100, Correct: 0, Accuracy: 0.0
```

图 22: 双向 LSTM 模型测试结果

图 23: 测试样例

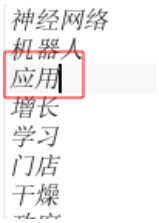


图 24: 测试样例-标准答案

图 25: 测试样例-3-gram 模型结果

图 26: 测试样例-双向 LSTM 模型结果

5.3 测试集的改进

前面在进行测试集测试的时候，我们便发现了不好的现象：模型给出的预测结果并不是百分百与标准答案一致的，但却可能在句子中替代标准答案并使句子保持大致的原意，而我们仍将这种预测结果判断为错误。考虑到中文中存在大量的近义词和同义词，这种模型评价方式显然不够客观准确，因为一般在面对词语预测问题时我们希望得到一个完整无错而又有意义的句子，只要模型的预测结果能达到这样的目的，我们就不应该判断它为完全错误。

解决上述问题的一个方法是使用一些损失函数来代替准确率作为评价标准，但这虽然解决了预测结果的“非正即负”问题，却没有解决标准答案的单一性问题。

这里我们可以考虑以下两个方案：

- 增加测试集中标准答案的个数。

  这样，即使使用准确率作为评价标准，在面对预测结果合理而不准确的情况时，我们仍能将其判断为正确。

- 增强训练集中的对比语料，同时模型给出的预测结果为概率最大的前 k 个。

  这样做可以使得模型给出候选结果而不是单一的预测结果，既符合我们对词语预测模型的期望，又可以在一定程度上解决上述问题。

5.4 LSTM 模型的改进方向

前面我们使用了单向 LSTM 模型和双向 LSTM 模型，并在训练集上取得了不错的效果。要想进行改进模型，我认为可以进行以下尝试：

- 使用其他超参数，使模型收敛得更快。

- 增加各层的节点（单元）数，使模型拟合得更好
- 使用预训练过或自己构建的词向量模型来代替原本简单的词嵌入层，如 Glove 和 word2vec 等。
- 使用更大规模的数据集进行训练。