自然语言处理 语言模型实验报告

> 姓名：TRY
>
> 学号：
>
> 专业：计算机科学与技术

> 时间：2020/12/26

一、研究内容

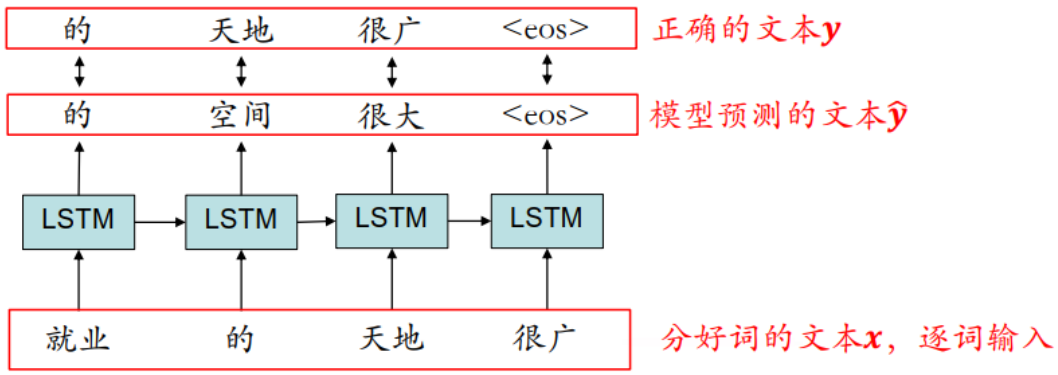
本次任务是使用$LSTM$来实现**语言模型**，并在SIGHAN Microsoft Research数据集（经过上次实验的“中文分词模型”处理）上进行中文分词的**训练和测试**。

语言模型是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。我们可以把一段自然语言文本看做一段离散的时间序列。假设一段长度为T的文本中的词依次为$w_1, w_2, \ldots, w_T$，那么在离散的时间序列中，$w_t(1 \leq t \leq T)$可看做在时间步 t 的输出或者标签。给定一个长度为T的词的序列$w_1, w_2, \ldots, w_T$，语言模型将计算该序列的概率：

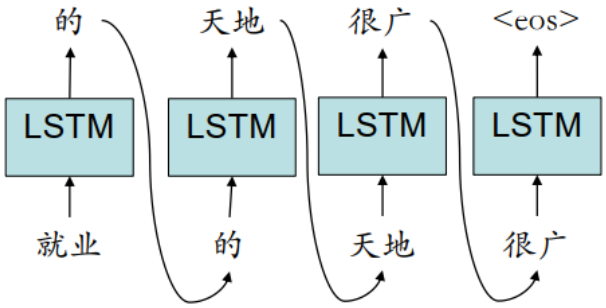
$$P(w_1, w_2, \ldots, w_T)$$

语言模型可用于提升语音识别和机器翻译的性能。例如在语言识别中，给定一段“厨房食油用完了”的语音，有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率，我们就可以根据相同读音的语言输出“厨房里食油用完了”的文本序列。 而在机器翻译中，如果对英文 “you go first” 逐词翻译成中文的话，可能得到“你先走” “你走先”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于其他排列方式的文本序列的概率，我们就可以把 “you go first” 翻译成 “你先走”。

而在本次实验中，**语言模型**实际上就是**文本生成**问题。在**数据预处理**中，利用上个实验中得到的中文分词模型，对数据集中的训练、测试集中的文本重新分词；在**训练**过程中，训练$LSTM$模型对训练集进行文本生成，即逐词输入已分好词的训练集文本 x 到$LSTM$模型中进行对应的下一个词的预测，得到预测文本 $\hat{y}$，然后再与正确的文本 y 进行逐词交叉熵的计算，得到`loss`，更新参数。



在**预测**过程中，同样逐句进行预测，即将测试集中每一句话的第一个词 x_0 输入到训练得到的 $LSTM$ 模型中，得到预测的词 y_0，然后使用这个时刻的 y_0 作为下一时刻模型的输入 x_1，不断迭代此过程，即使用上一时刻模型输出的结果 y_{t-1} 作为当前时刻的输入 x_t，直到输出结果为结束符号 $< EOS >$，则此文本生成结束。最终输出文本生成样例。



二、研究方案

语言模型（Language Model，LM）的解决方案有许多种，如传统语言模型（以n-gram为代表）和神经网络语言模型。而后者又分为两类：前馈神经网络模型（FFLM）和循环神经网络模型（RNNLM）。而在本次实验中，就使用了RNNLM作为解决方法，应用了经典的$LSTM$以解决语言模型中的文本生成问题，并通过深度学习框架$keras$来实现。

2.1 背景：传统语言模型

传统的离散模型主要是统计语言模型，比如bigram或者n-gram语言模型，是对n个连续的单词出现概率进行建模。其基本假设是单词的分布服从n阶马尔可夫链。通过对连续n个单词出现频率进行计数并平滑来估计单词出现的概率。但由于是离散模型，因此有**稀疏性**和**泛化能力低**的缺点。

- 稀疏性：n-gram模型只能对文本中出现的单词进行建模，当新文本中出现意义相近但没在训练文本中出现的单词时，传统离散模型无法正确计算这些单词的概率，错误地赋予它们0概率的预测值。这是非常不符合语言规律的事情。 为了解决这个问题，传统方法是引入一些平滑或者back-off的技巧，但整体上效果并不好。

  > 例如：在汽车新闻中，“SUV”和“吉普”是同义词，可以交替出现，但若整个训练集中没有出现过“吉普”这个单词，则在传统模型中“吉普”这个词的出现概率会接近于0。

- 泛化能力低：离散模型依赖于固定单词组合，需要完全的模式匹配，否则也无法正确输出单词组出现的概率。

  > 例如：假设新闻中一段话是“作为翻山越岭常用的SUV”，这句话和“作为越野用途的吉普”本身意思相近，一个好的语言模型是应该能够识别出这两句话无论从语法还是语义上都是非常近似，应该有近似的概率分布。但离散模型无法达到这个要求，即体现出泛化能力不足。

2.2 神经网络语言模型

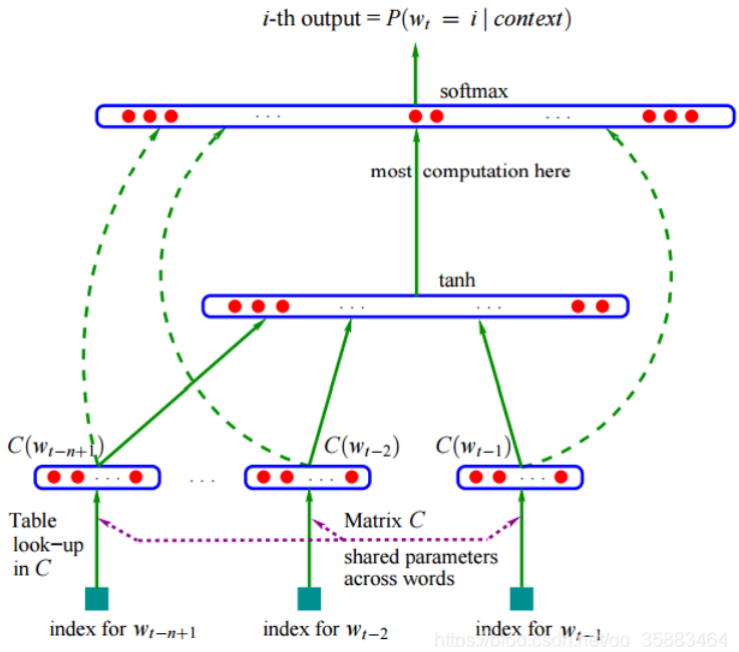
传统语言模型的上述内在缺陷使人们开始把目光转向神经网络模型，期望深度学习技术能自动化学习代表语法和语义的特征，解决稀疏性问题，并提高泛化能力。

主要有两类神经网络语言模型：

- **前馈神经网络模型（FFLM）**：解决稀疏性问题；
- **循环神经网络模型（RNNLM）**：解决泛化能力，尤其是对长上下文信息的处理。

2.2.1 前馈神经网络模型（FFLM）

前馈神经网络模型由三层全连接神经网络模型构成（嵌入层、全连接层、输出层），以估计给定$n-1$个上文的情况下，第 n 个单词出现的概率。其架构如下图所示：



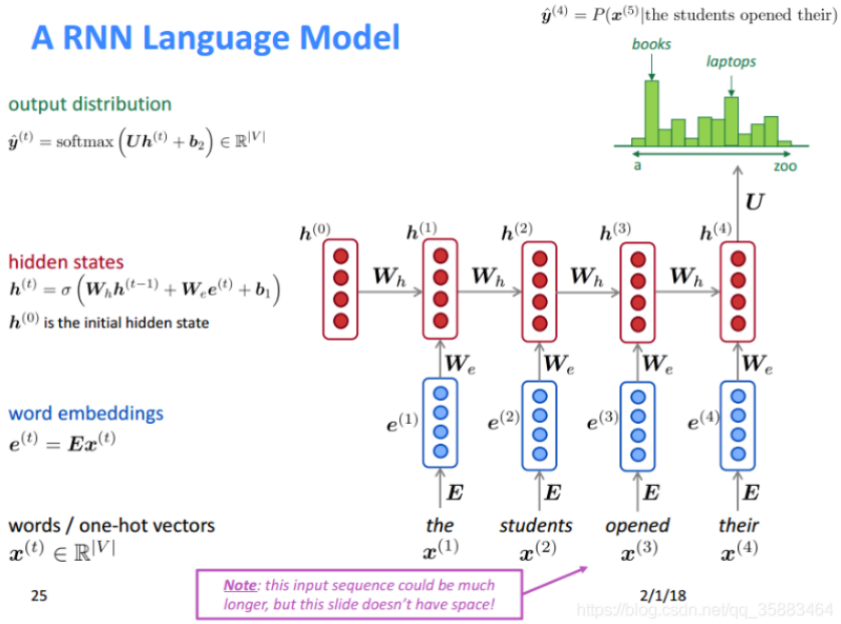
通过使用词向量的映射，FFLM能解决稀疏性的问题。一些在训练集中没有遇到过的单词由于其与上下文同时出现的关系，在词向量的空间中会与相类似的单词处于相近的位置，从而降低出现接近于0的条件概率的问题。该模型在实际应用过程中表现出了一定的泛化能力，但没有明确地对超出观察窗口的上下文信息进行处理。

2.2.2 循环神经网络模型（RNNLM）

RNNLM就是为了解决上述固定窗口问题而出现的。FFLM假设每个输入都是独立的，但这不合理。经常一起出现的单词以后一起出现的概率也会更高，并且当前应该出现的词通常是由前面一段文字决定的，利用这个相关性能提高模型的预测能力。RNNLM就是利用文字的上下文序列关系建模。

> 例如："我最近要去美国出差，想顺便买点东西，因此需要兑换_ "。 对于在 "__"中需要填写的内容，RNNLM能回溯到前两个分句的内容，形成对 “买”，“兑换”等上下文的记忆，推测出是“美元”。

在本次实验中，**LSTM模型**就是RNNLM的变种，其网络示意图如下：有输入层、词嵌入层、隐藏层（LSTM）、输出层等。



1. **输入层：**比如句子 x 有 n 个词。首先将 n 个词做one-hot得到稀疏向量，再通过由训练集和测试集所有词语构建的`vocab`词典，得到 n 个词的 d 维稀疏词向量。并将句子的长度设置为统一长度`maxlen`（通过padding实现“多删少补”）。
2. **词嵌入：**直接通过查找**预训练好的中文词向量集**来得到每个词的 D 维稠密词向量。因此，经过词嵌入层后，每一个词都是由word embedding来构成的向量。
3. **LSTM层：**迭代训练语言模型，得到 $maxlen$ 个 D 维向量，并通过全连接层。
4. **输出层：**通过`softmax`归一化操作，得到最终的输出结果，即为下一位置的各词语的预测概率。

三、核心代码讲解

以下讲解各部分的核心代码，详细见代码注释。

3.1 数据预处理

- **读文件`readfile()`函数和`get_word()`函数：**实现从utf8文件中读取内容，并获得整个数据集的词列表`word`。
  - 其中，调用了`get_word`函数，获得每行的word，并且**去除了句子中的标点符号。**

> 解释：由于标点符号其实只起间隔作用，例如“，”后可以跟很多词，这样不利于模型的预测，所以删除了所有的标点符号。

```
def read_file(file):
    word, content = [], []
    maxlen = 0

    for i in range(len(file)):
        line = file.loc[i,0]   # 用loc来访问dataframe
        line = line.strip('\n') #去掉换行符
        line = line.strip(' ')  #去掉开头和结尾的空格

        word_list = get_word(line)         #获得字列表：去掉标点，（不添加<EOS>结束符

        maxlen = max(maxlen, len(word_list))
        word.extend(word_list)             #每一个单元是1个词，且不加<EOS>符号
        content.append(word_list)          # 每一个单元是一行里面的各个词（分好）
    return word, content, maxlen  # word是单列表，content是双层列表

# 将句子转换成词序列
def get_word(sentence):
    word_list = []
    sentence = re.sub("[+\.\!\/_,$%^*(+\“\”\‘\’]+|[+——！，。？、；：《》【】~@#￥%……&*（）]", "",sentence)  # 去掉所有除空格外的标点符号
    sentence = sentence.split() #去掉空格
    return sentence
```

- **加工数据函数** `process_data()`，主要涉及对`word`序列进行padding操作。首先，构建`vocab2idx`字典，形成词语到序号的映射；并对整个数据集的`word_list`中每一个句子的`word`进行序号的映射，根据最大长度`MAXLEN`进行padding，得到`x`。同理，构造当前位置的下一位置词语的序号列表，并进行padding操作，得到`y`。最后，再对`y`进行归一化操作。
  - padding调用`pad_sequences()`函数进行实现，具体操作为：大于`MAXLEN`的进行截断，小于`MAXLEN`的进行padding。且padding是默认的left_padding，因此这里需要设置`padding`和`truncating`为`post`，即表示从句子的后面进行补零或截断。
  - `word`列表的默认padding值为0，表示`<PAD>`。
  - 对`y`的归一化操作具体调用`to_categorical()`函数实现，用于训练时计算交叉熵。
  - 在构建下一位置词语列表`y`时，需要判断当前位置是不是句子的最后一个词语，如果是则为`1`，表示下一位置为`<EOS>`。（`1`在vocab中表示`<EOS>`）

```
# process data: padding
def process_data(word_list, vocab, MAXLEN):
    # vocab to idx dictionary:
    vocab2idx = {word: idx for idx, word in enumerate(vocab)}
    # x: get every idx of every word, map to idx in vocab, set to <EOS>
if not in vocab(<EOS> not included in vocab)
    x = [[vocab2idx.get(word, 1) for word in s] for s in word_list]

    # y: get next word idx
    y = []
    for i in range(len(word_list)):
        temp = []
        for j in range(len(word_list[i])):
            if j == len(word_list[i]) - 1:
                temp.append(1)  # 1 means <EOS>
            else:
                temp.append(x[i][j+1])
        y.append(temp)

    # padding of x, default is 0(symbolizes <PAD>). padding
includes:over->cutoff, less->padding. default: left_padding,needs
changing
    x = pad_sequences(x, maxlen=MAXLEN, value=0, padding='post',
truncating='post')
    # padding of y, default is 0. right padding
    y = pad_sequences(y, maxlen=MAXLEN, value=0, padding='post',
truncating='post')
    # one-hot of y
    y = to_categorical(y, len(vocab))

    return x, y
```

- **加载数据`load_data()`函数：**调用`read_file()`函数，得到训练集和测试集的词语列表，且词语列表有单列表形式的`train_word`，也有双层列表形式的`train_content`。然后构建`train_word`和`test_word`词语除重后组成的词表`vocab`。
  - 其中，`train_word`是用来和`test_word`一起形成整个数据集的词表（用来除重使用的）。
  - 并且得到词表`vocab`之后，需要加上两个特殊词`<PAD>`和`<EOS>`，形成完整的词表。
  - 在处理训练集和测试集的时候，得到它们分别的最大句子长度`MAXLEN`，但此时没有选取这个长度作为模型的统一长度。

    - 因为在语言模型中，没有要求像中文分词模型一样输出整个句子的分词结果，因此可以统一做截断来训练和预测。如选取`MAXLEN=50`，可以加快模型训练速度。
  - **注意：**`<EOS>`只在构建下一位置词语列表时有用，在构建当前位置词语列表时不需要加入`<EOS>`标志。

```
def load_data():
    train_word, train_content, _ = read_file(train_set)
    test_word, test_content, maxlen = read_file(test_set)

    vocab = list(set(train_word + test_word))   # 合并，构成大词表
    special_chars = ['<PAD>', '<EOS>']   #特殊词表示：PAD表示padding，EOS表示句子结尾
    vocab = special_chars + vocab

    # save initial config data
    with open(SAVE_PATH, 'wb') as f:
        pickle.dump((vocab), f)

    # process data: padding
    print('maxlen is %d' % maxlen)
    return train_content, test_content, vocab, maxlen
```

3.2 词嵌入

- **读取预训练的词向量集：**这里，调用了genism库中的`KeyedVectors.load_word2vec_format`函数对词向量集进行读取。

```
word2vec_model_path = 'sgns.wiki.word.bz2'  #词向量位置
word2vec_model = KeyedVectors.load_word2vec_format(word2vec_model_path, binary=False, unicode_errors='ignore')
```

- **构造整个词表的大词向量矩阵：**这里需要构建一个**词对词向量**的大矩阵，用于后面的embedding层。且构建的顺序就是每一个字在vocab中的顺序。
  - 如果这个字不在词向量列表中，则将其赋值为全0.

```
def make_embeddings_matrix(word2vec_model, vocab):
    char2vec_dict = {}    # 字对词向量
    vocab2idx = {char: idx for idx, char in enumerate(vocab)}
    for char, vector in zip(word2vec_model.vocab, word2vec_model.vectors):
        char2vec_dict[char] = vector
    embeddings_matrix = np.zeros((len(vocab), EMBED_DIM))# form huge matrix
    for i in tqdm(range(2, len(vocab))):
        char = vocab[i]
        if char in char2vec_dict.keys():    # 如果char在词向量列表中，更新权重；否则，赋值为全0（默认）
            char_vector = char2vec_dict[char]
            embeddings_matrix[i] = char_vector
    return embeddings_matrix
```

3.3 构建$LSTM$模型

- **构建$LSTM$模型**：调用`keras.models`和`keras.layers`中的各个函数进行实现。其中包括的层有：Input输入层，Embedding嵌入层（包含加载预训练词向量的操作），LSTM层，TimeDistributed全连接层，激活层（`softmax`归一化）。并调用了`model.summary()`函数和`model.compile()`函数，前者输出model的各项参数信息；后者compile模型，参数可指定目标函数类型，如`adam`，`RMSprop`等等，`loss`为交叉熵。
  - **注意：**这里不需要添加Dropout层！因为本模型本来就不会过拟合，若添加了Dropout层，效果会更差！

```
train_content, test_content, vocab, maxlen = load_data()
# change maxlen
maxlen = 50
embeddings_matrix = make_embeddings_matrix(word2vec_model, vocab)
# input layer
inputs = Input(shape=(maxlen, ), dtype='int32')
# embedding layer: map the word to it's weights(with embedding-matrix)
x = Embedding(len(vocab), EMBED_DIM, weights=[embeddings_matrix],
input_length=maxlen, trainable=True)(inputs)
# LSTM layer
x = LSTM(RNN_UNITS, input_shape=(maxlen, EMBED_DIM),
return_sequences=True)(x)
# 一维展开，全连接
x = TimeDistributed(Dense(len(vocab)))(x)
# 激活函数：softmax
outputs = Activation('softmax')(x)
# model
model = Model(inputs=inputs, outputs=outputs)
# print arguments of each layer
model.summary()
# target_function: includes optimizer, function_type, metrics
RMSPROP = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None,
decay=0.0)
model.compile(optimizer='RMSprop', loss='categorical_crossentropy',
metrics=['accuracy'])
```

3.4 训练和测试

- **训练函数**：调用`fit`函数是实现训练。并且，由于此次训练所需内存非常大，因此需要分段进行训练，即200个样本进行一次训练（即一次只申请200个训练样本的内存），本质相同。

```
# train
def train(start1, end1, epochs1, batch_size1):
    model.load_weights('model.h5')
    maxlen = 50   # maxlen取50，直接截断
    start = start1
    EPOCHS = epochs1
    TRAIN_BATCH = 200
    while start < len(train_content):
        print(start)
        if start == end1:
```

```
            break
        if start+TRAIN_BATCH <= len(train_content):
            train_x, train_y = process_data(train_content[start:
start+TRAIN_BATCH], vocab, maxlen)
        else:
            train_x, train_y = process_data(train_content[start: ],
vocab, maxlen)

        model.fit(train_x, train_y, batch_size=batch_size1,
epochs=EPOCHS, verbose=2, validation_split=0.1)
        start += TRAIN_BATCH
    model.save_weights('model.h5')
```

- **测试函数**：调用predict函数实现预测，先输入第一个词，再将当前时刻的输出作为下一时刻的输入放到模型中。具体来说：对预测出来的各词语的概率取最大值，最大值对应的下标即为预测的下一位置的词语`index`。且预测出来的`index`需要放到对应的位置`j+1`来构建出数组`temp`（其余位置为0），并作为下一时刻的输入放到模型中。

```
def build_test_data(word_idx, maxlen, index):
    result = [0] * maxlen
    result[index] = word_idx
    result = np.array(result)
    return result

def test1(test_num1):
    model.load_weights('model.h5')
    i = 0
    j = 0
    TEST_NUM = test_num1
    for i in range(TEST_NUM):
        sentence = [test_content[i][0]]
        word_idx = vocab2idx.get(test_content[i][0])
        test_x = build_test_data(word_idx, maxlen, 0)
        for j in range(0, 49):
            temp = []
            temp.append(test_x)
            temp = np.array(temp)
            next_word = model.predict(temp, batch_size=1)  # 输入得是
numpy数组，不能是list
            index = np.argmax(next_word[0][j])
            if index == 1 or index == 0:   # means predict <EOS>
                print(i,j, index)
                break
            sentence.append(vocab[index])
            test_x = build_test_data(index, maxlen, j+1)   # position:
j+1
        print(sentence)
```

四、实验步骤设计

本次实验使用$keras$深度学习框架进行编写，具体步骤如下：

1. 安装$keras$ 2.3.1版本和$tensorflow$ 2.2版本，并安装$gensim$库。

- $keras$和$tensorflow$版本一定要对应且不能过高！否则会出现很多奇奇怪怪的报错。
- $gensim$用于读取预训练好的词向量。

2. 定义超参量`RNN_UNITS`，`BATCH_SIZE`，`EMBED_DIM`，`EPOCHS`，`MAXLEN`等。
3. 利用$pandas$库，读取训练集和测试集的utf8文件。
4. 利用$gensim$库，读取预训练好的词向量文件。
5. 调用`load_data()`函数，读取训练集和测试集的词列表，并进行padding处理和one-hot处理，实现“数据预处理”。
6. 根据预训练的词向量，构建大的词向量矩阵，下标对应`vocab`的顺序，实现“词嵌入”。
7. 构建$LSTM$的model，对训练集进行迭代训练。
8. 利用上步得到的model，对测试集进行预测，输出结果。

五、实验结果

在本次实验中，一开始我在LSTM的模型搭建中添加了Masking（屏蔽层）和Dropout（正则化，防止过拟合）两个层，经过训练集EPOCHS=50次的迭代之后，发现在测试集的预测效果依旧不好。效果如下：（会常预测出相同的词）

```
0 5 0
['扬帆', '香港', '作为', '三', '所', '作']
1 7 0
['希腊', '的', '同时', '必须', '作为', '三', '所', '作']
2 6 0
['海运', '工作', '必须', '作为', '三', '所', '作']
3 5 0
['另外', '他', '作为', '三', '所', '作']
4 7 0
['多年来', '的', '同时', '必须', '作为', '三', '所', '作']
5 3 0
['十几年', '来', '所', '作']
6 7 0
['瓦西里斯', '的', '同时', '必须', '作为', '三', '所', '作']
7 3 0
['他', '说', '这是', '近']
```

而在删除了这两个层的操作后，发现效果明显变好，收敛速度明显加快，并行粒度也可相应增大(batch_size可以取更大的值，从4变到32)。

因此，本次实验我也没有进行过多的调参，只对优化器Optimizer进行了调参，有Adam和RMSprop。

5.1 RMSprop

以下预测结果使用如下参数：

参数	取值
BATCH_SIZE	32
Optimizer	RMSprop
RNN_UNITS	300
EMBED_SIZE	300
Epoch	50

输出样例如下：

```
57 4 0
['我们', '将', '其', '自己', '的']
58 8 0
['天灾人祸', '决定', '我', '为', '新', '将', '其', '自己', '的']
59 7 0
['父母', '的', '常委', '近', '国有', '及', '世界', '的']
60 1 1
['一个', '重要']
61 1 1
['一个', '重要']
62 5 0
['不少', '地方', '将', '其', '自己', '的']
63 4 0
['我们', '将', '其', '自己', '的']
64 3 0
['意志', '是', '世界', '的']
```

- 样例1：

```
输入：海运
输出：海运 的 常委 近 国有 及 世界 的
```

- 样例2：

```
输入：两
输出：两 院 生产 世界 的
```

- 样例3：

```
输入：天灾人祸
输出：天灾人祸 决定 我 为 新 将 其 自己 的
```

- 样例4：

```
输入：要
输出：要 想 附 图片 1 张
```

5.2 Adam

以下预测结果使用如下参数：

参数	取值
BATCH_SIZE	32
Optimizer	Adam
RNN_UNITS	300
EMBED_SIZE	300
Epoch	25

输出样例如下：

```
0 2 0
['扬帆', '人', '中']
1 1 0
['希腊', '的']
2 9 0
['海运', '不是', '一件', '每个', '也', '也', '可以', '认识', '是', '和']
3 5 0
['另外', '健身', '出于', '满意', '是', '和']
4 5 0
['多年来', '个人', '卓越', '不能', '违背', '和']
5 1 0
['十几年', '前']
```

- 样例1：

```
输入：多年来
输出：多年来 个人 卓越 不能 违背 和
```

- 样例2：

```
输入：尤其是
输出：尤其是 处在 受到 热情 也 也 可以 认识 是 和
```

- 样例3：

```
输入：振奋
输出：振奋 精神 在于 热情 也 也 可以 认识 是 和
```

- 样例4：

```
输入：根据
输出：根据 中央 也 必须 在 不
```

5.4 最佳参数

从模型的loss和预测结果可以看出，Adam的效果要优于RMSprop的效果。

因此，本次实验使用$keras$框架时的最佳参数为：

参数	最佳取值
BATCH_SIZE	32
Optimizer	Adam
BiRNN_UNITS	300
EMBED_SIZE	300
Epoch	25

结果保存为文件：`result.txt`。

六、加分项

在本次实验中，加分项为：使用预训练词向量，模型超参数组合。