案例四：基于LSTM模型的相似文本生成

1. 案例目的

• 掌握自然语言处理的基本方法；

• 理解LSTM的原理；

• 掌握基于LSTM模型的相似文本生成。

2. 案例内容

本案例以《爱丽丝梦游仙境》原著小说为训练数据，建立模型，让模型有能力根据给出的文本生成和原小说相似的内容。

3. 案例知识点

• LSTM模型

• 文本数据向量化

• 相似文本生成

4. 案例时长

共4学时，具体安排如下：

• 处理文本数据（1学时）；

• 构建并训练LSTM模型（2学时）；

• 生成相似文本（1学时）。

5. 案例实验环境

**•操作系统**：

1）Linux Ubuntu 16.04 64位操作系统

**•软件环境：**

1）Python 3.7

2）Tensorflow 2.3.0

3）Keras 2.4.3

**•开发环境与工具：**

1）Spyder 4

6. 案例分析

本案例主要分为以下3部分：

1）数据是文本类型，需要先对文本数据做文本向量化处理；

2）构建LSTM模型，训练模型，保存模型，验证模型是否训练完毕；

3）调用预训练模型继续写小说。

7. 案例实验过程

基于LSTM模型继续写小说，可分为以下3个步骤：

1、处理文本数据；

2、构建并训练LSTM模型；

3、生成相似文本。

7.1准备工作

在桌面上新建一个目录，名字叫similar-text。本案例中所有的数据集和代码都放在该目录下。

7.1.1 打开Spyder软件

在桌面上点击Spyder图标打开Spyder，并把工作目录切换至similar-text目录下。

7.1.2 获取数据

在similar-text目录下新建一个目录，命名为data。该目录用于存放数据集。

本案例需要1个数据集，是爱丽丝梦游仙境的小说内容（wonderland.txt）。

点击右侧菜单栏中的【资源下载】按钮，复制提供的URL前缀，如https://xxx.eec-cn.com/。

将URL前缀和数据集地址拼接到一起，如https://xxx.eec-cn.com/dataSet/systemLib/45fe1f9b9c3d41fb8b5639f684a238fd.zip，然后解压缩。

解压后有2个文件，一个是数据集文件（wonderland.txt），另一个是预训练模型文件（model-improvement-30-1.3634.h5）。

将数据集文件（wonderland.txt）复制到刚刚创建的similar-text/data目录下。

7.1.3 新建代码文件

确保Spyder的工作目录为similar-text。使用Spyder新建4个Python文件，文件名和作用如下：

process\_data\_temp.py: 处理文本数据，将文本向量化。这是个临时文件。

process\_data.py: 由于处理后的文本数据在训练模型和测试模型时都被用到，所以把处理文本数据的代码整理成函数的形式，减少重复代码。

train\_novel.py: 构建LSTM模型，训练模型，保存模型，验证模型是否收敛。

test\_novel.py: 调用预训练模型生成相似文本。

7.1.4 新建模型目录

在similar-text目录下新建一个目录，命名为output。该目录用于存储训练好的模型。

模型本应该同学们自己训练，可是考虑到训练时间过长，所以建议训练一会儿就终止程序，然后使用预训练的模型做预测。

将【7.1.2 获取数据】小节中下载下来的model-improvement-30-1.3634.h5文件复制到output目录下。

至此，整个案例的目录结构如图1所示。

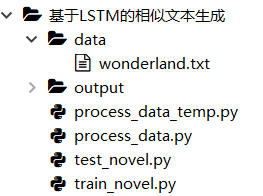


图1 案例的目录结构

7.1.5 了解数据集

数据集是《爱丽丝梦游仙境》这篇小说的文本，文本示例如图2所示。



图2 小说文本

7.1.6 案例实验思路

生成相似文本本质上是分类问题。输入1个序列，输出1个类别。输出的类别可以是1个汉字，可以是1个英文单词，也可以是1个英文字符。

7.2处理文本数据

首先要对文本数据做处理，处理成LSTM需要的格式。

为了方便同学们理解，看到每一步的执行结果，本部分代码暂时不以函数的形式呈现。等代码完成后，再改造成函数的形式。

7.2.1 定义全局常量

向process\_data\_temp.py中添加代码。

全局常量是序列的长度。代码中中序列长度是100，表示根据前100个字母预测下一个字母。【7.2.5】小节会详细解释。

全局变量是小说的完整路径。

【代码7-2-1】process\_data\_temp.py

# -\*- coding: utf-8 -\*-

"""

小说文本处理

"""

# 导入库

import numpy as np

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

# 定义全局变量

text\_file = "data/wonderland.txt"

执行代码块。

从现在开始，建议切换到Spyder的Variable explorer面板，查看变量信息。

7.2.2 读取小说内容

【代码7-2-2】process\_data\_temp.py

# 读取小说，并转换为小写

raw\_text = open(text\_file).read() # 读取

raw\_text = raw\_text.lower() # 转换为小写字母

执行代码块。

此时，raw\_text变量存储着小说的内容。双击变量名字可以查看完整的变量值。

7.2.3 文本向量化

由于神经网络模型不识别字符，只识别数字，所以首先应把文本数据转换为数字。

在自然语言处理中，文本向量化是个复杂且重要的概念。这里只做简单的讨论。

将小说文本转换为数字，有2种思路：

1）得到小说中的字符集（26个英文字母+标点符号），然后每个字符与1个数字相对应；

2）得到小说中的单词集，然后每个单词和1个数字相对应；

第2种思路看起来更合理一些，但是单词集比字符集大几个数量级。上面提到过，生成相似文本本质上是分类问题，即：单词集有几千个单词，输出就有几千个分类，神经网络的输出层就有几千个神经元，这会导致神经网络异常复杂。

所以本案例采用第1种思路。

等到同学们学习了《自然语言处理》课程的文本向量化知识点，就可以尝试采用第2种思路。

得到字符->数字的对应关系，得到数字->字符的对应关系，代码如下：

【代码7-2-3】process\_data\_temp.py

# 得到 字符->数字 和 数字->字符的对应关系

chars = sorted(list(set(raw\_text)))

char\_to\_int = dict((c, i) for i, c in enumerate(chars))

int\_to\_char = dict((i, c) for i, c in enumerate(chars))

执行代码块。

这3个变量的信息如图3所示。

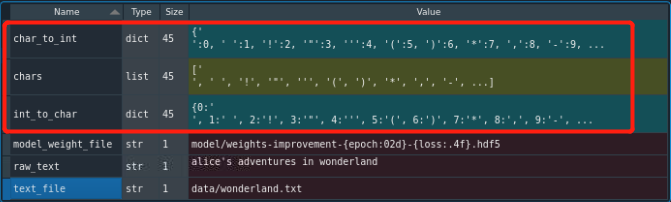


图3 3个变量的信息

由上图可见，字符集中共有45个字符。

双击int\_to\_char ，查看该变量的值，效果如图4所示。

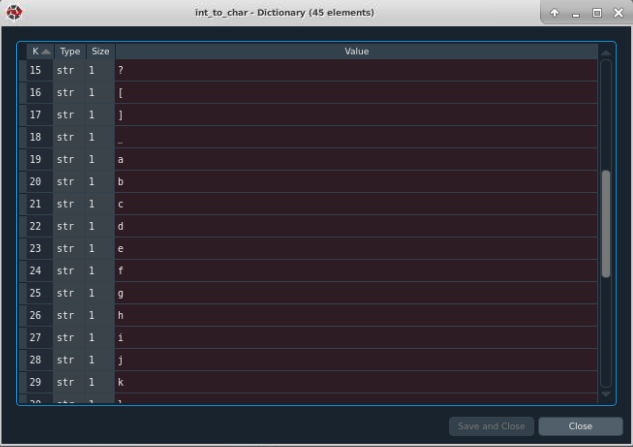


图4 数字与字符的对应关系

由上图可见，15代替了?，19代替了a。于是,bed这个单词就可以用20 23 22代替。

文本向量化完成。

7.2.4 统计小说信息

【代码7-2-4】process\_data\_temp.py

# 统计小说信息

n\_chars = len(raw\_text)

n\_vocab = len(chars)

print("小说的总字符个数是: ", n\_chars)

print("小说的字符种类是: ", n\_vocab)

执行代码块。

小说的总字符个数是144407。

小说的字符种类是45。

7.2.5 构造自变量和因变量

在全局常量中，定义了SEQ\_LENGTH的值是100，即使用100个字符预测1个字符。

为简单起见，假设小说的内容是【i-have-a-dream】，假设SEQ\_LENGTH的值是5。那么自变量和因变量的值如图5所示。

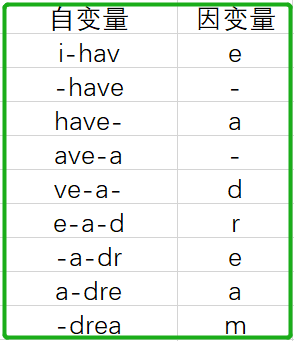


图5 自变量和因变量的值

由上图可见，“i-have-a-dream”有14个字符，SEQ\_LENGTH的值是5，自变量和因变量的记录数是14-5=9条。

为爱丽丝梦游仙境小说生成自变量和因变量，代码如下：

【代码7-2-5】process\_data\_temp.py

# 构造自变量和因变量

dataX\_temp = []

dataY\_temp = []

for i in range(0, n\_chars - SEQ\_LENGTH):

seq\_in = raw\_text[i:i + SEQ\_LENGTH]

seq\_out = raw\_text[i + SEQ\_LENGTH]

dataX\_temp.append(seq\_in)

dataY\_temp.append(seq\_out)

执行代码块。

自变量和因变量有144307条记录。

上面统计过，小说的总字符数是144407，SEQ\_LENGTH的值是100。144407-100=144307。

双击dataX\_temp查看自变量的值，如图6所示。

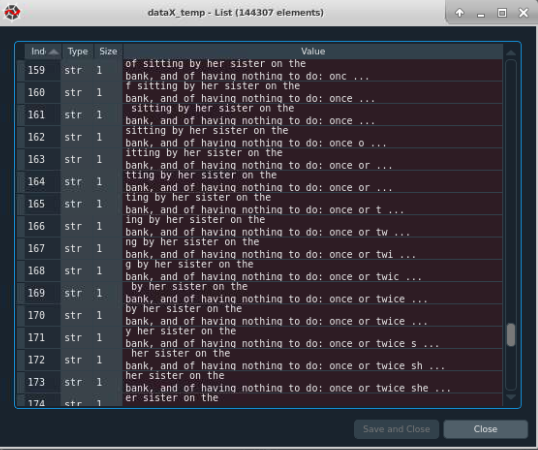


图6 自变量的值

7.2.6 自变量和因变量向量化

对自变量和因变量向量化，代码如下：

【代码7-2-6】process\_data\_temp.py

#对自变量与因变量做文本向量化

dataX = []

dataY = []

for i in dataX\_temp:

dataX.append([char\_to\_int[char] for char in i])

for i in dataY\_temp:

dataY.append(char\_to\_int[i])

执行代码块。

文本向量化后自变量的值如图7所示。

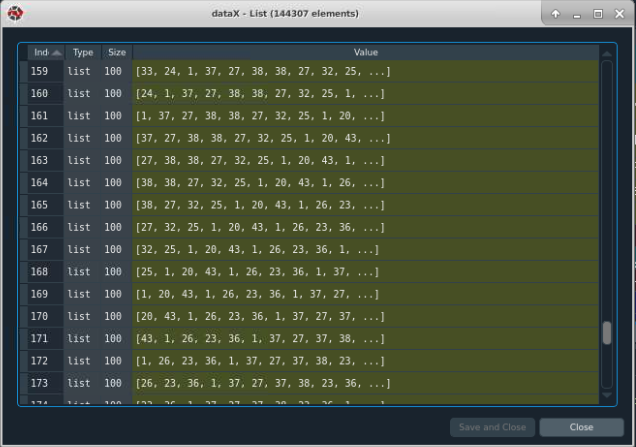


图7 文本向量化后自变量的值

7.2.7 自变量和因变量转换为NumPy类型

神经网络参数需要的类型是NumPy类型。代码如下：

【代码7-2-7】process\_data\_temp.py

# 将dataX和dataY转换为NumPy类型

dataX = np.array(dataX)

dataY = np.array(dataY)

执行代码块。

自变量和因变量的信息如图8所示。

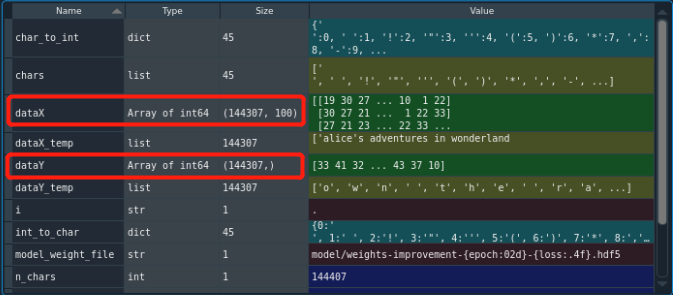


图8 自变量和因变量的信息

由上图可见，dataX有100列，因为SEQ\_LENGTH的值是100。

7.2.8 转换成函数形式

由于处理后的文本数据在训练模型和测试模型时都被用到，所以把处理文本数据的代码整理成函数的形式，减少重复代码。

改动2个地方：

1. 把全局变量和全局常量去掉，作为函数的参数

2. 返回重要的变量值

把转换后的代码添加到process\_data.py中。转换后代码如下：

【代码7-2-8】process\_data.py

# -\*- coding: utf-8 -\*-

"""

小说文本处理

"""

# 导入库

import numpy as np

def process\_novel\_data(text\_file, SEQ\_LENGTH):

# 读取小说，并转换为小写

raw\_text = open(text\_file).read() # 读取

raw\_text = raw\_text.lower() # 转换为小写字母

# 得到 字符->数字 和 数字->字符的对应关系

chars = sorted(list(set(raw\_text)))

char\_to\_int = dict((c, i) for i, c in enumerate(chars))

int\_to\_char = dict((i, c) for i, c in enumerate(chars))

# 统计小说信息

n\_chars = len(raw\_text)

n\_vocab = len(chars)

print("小说的总字符个数是: ", n\_chars)

print("小说的字符种类是: ", n\_vocab)

# 构造自变量和因变量

dataX\_temp = []

dataY\_temp = []

for i in range(0, n\_chars - SEQ\_LENGTH):

seq\_in = raw\_text[i:i + SEQ\_LENGTH]

seq\_out = raw\_text[i + SEQ\_LENGTH]

dataX\_temp.append(seq\_in)

dataY\_temp.append(seq\_out)

# 使用词袋模型对自变量与因变量做文本向量化

dataX = []

dataY = []

for i in dataX\_temp:

dataX.append([char\_to\_int[char] for char in i])

for i in dataY\_temp:

dataY.append(char\_to\_int[i])

# 将dataX和dataY转换为NumPy类型

dataX = np.array(dataX)

dataY = np.array(dataY)

return dataX, dataY, n\_vocab, int\_to\_char

代码不需要执行，后面会调用到。

数据处理完毕。

7.3构建并训练LSTM模型

构建LSTM模型，训练模型，保存模型，验证模型是否训练完毕。

向train\_movel.py中添加代码。

7.3.1 导入库

【代码7-3-1】train\_novel.py

# -\*- coding: utf-8 -\*-

"""

构建模型、训练模型、保存模型

"""

# 导入库

import numpy as np

from keras.utils import np\_utils

import matplotlib.pyplot as plt

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import LSTM

from keras.callbacks import ModelCheckpoint

from process\_data import process\_novel\_data

执行代码块。

7.3.2 定义全局变量和全局常量

全局常量是模型超参数。由于超参数的值决定了模型的性能，所以把超参数放在一起方便调试。

全局变量是小说文件路径和模型文件保存路径。把全局变量放在一起增加代码可读性。

【代码7-3-2】train\_novel.py

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

EPOCHS = 30

BATCH\_SIZE = 128

# 定义全局变量

text\_file = "data/wonderland.txt"

model\_file = 'output/model-improvement-{epoch:02d}-{loss:.4f}.h5'

执行代码块。

7.3.3 得到处理后的数据

调用文本处理函数，得到处理后的数据。

【代码7-3-3】train\_novel.py

# 得到处理后的数据

dataX, dataY, n\_vocab, int\_to\_char = process\_novel\_data(text\_file, SEQ\_LENGTH)

执行代码块。

7.3.4 自变量转换为3维数组

将自变量转换为3维数组，格式是[samples, time steps, features]。

features的意思是，预测器的数量。在本案例中它的值是1。

《爱丽丝梦游仙境》的作者是刘易斯·卡罗尔。1981年，他又写了《爱丽丝镜中奇遇记》，这本小说是《爱丽丝梦游仙境》的续篇。如果我们同时用这2篇小说预测下一个字符，那么features的值是2。

【代码7-3-4】train\_novel.py

# 将自变量转换为3维 [samples, time steps, features]

X = np.reshape(dataX, (dataX.shape[0], dataX.shape[1], 1))

执行代码块。

此时X变量存储着自变量的值。X变量的形状为(144307, 100, 1)。dataX的形状为(144307, 100)。自变量转换为3维数组，由(144307, 100)转换为(144307, 100, 1)。

7.4.5 数据标准化

神经网络模型需要传入标准化的数据。数据标准化处理使得数据的值在[0, 1]之间，代码如下：

【代码7-3-5】train\_novel.py

# 标准化

X = X / float(n\_vocab)

执行代码块。

7.3.6 因变量独热编码

字符集有45个字符，神经网络的输出层需要有45个分类，所以因变量需要45列。

双击dataY变量查看因变量的值。转换前的因变量如图9所示。

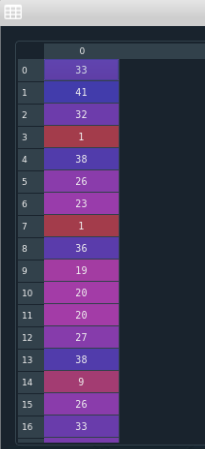


图9 转换前的因变量

将因变量转换为45列的代码如下：

【代码7-3-6】train\_novel.py

# 独热编码

y = np\_utils.to\_categorical(dataY)

执行代码块。

转换后因变量的值存储在y变量中。转换后的因变量如图10所示。

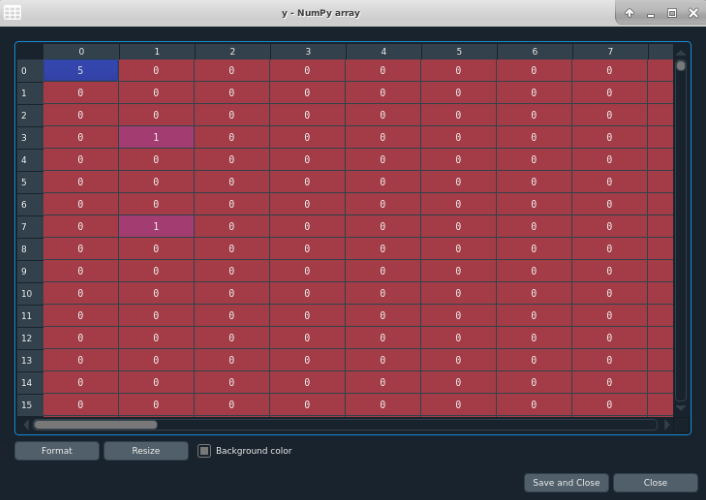


图10 转换后的因变量

文本数据处理完毕。接下来训练LSTM模型。

7.3.7 初始化LSTM模型

【代码7-3-7】train\_novel.py

# 初始化LSTM模型

model = Sequential()

执行代码块。

7.3.8 构建输入层+第1层LSTM

输入层的参数是(timesteps, features)；第1层LSTM层定义256个神经元；return\_sequences表明后面还有LSTM层；激活函数默认是tanh；引入Dropout机制防止过拟合。

【代码7-3-8】train\_novel.py

# 构建输入层+第1层LSTM

model.add(LSTM(256, input\_shape=(X.shape[1], X.shape[2]), return\_sequences=True))

model.add(Dropout(0.2)) # 防止过拟合

执行代码块。

7.3.9 构建第2层LSTM层

第2层LSTM层定义256个神经元；return\_sequences=False表明后面没有LSTM层（return\_sequences的默认值是False）；激活函数默认是tanh；引入Dropout机制防止过拟合。

【代码7-3-9】train\_novel.py

# 构建第2层LSTM层

model.add(LSTM(256))

model.add(Dropout(0.2))

执行代码块。

7.3.10 构建输出层

前面提到，生成相似文本本质上是分类问题，所以输出层的神经元个数是字符集的个数（本案例有45个）；激活函数使用softmax。

【代码7-3-10】train\_novel.py

# 构建输出层

model.add(Dense(y.shape[1], activation='softmax'))

执行代码块。

7.3.11 编译模型

前面提到，生成相似文本本质上是分类问题，所以loss使用多元交叉熵；优化算法使用adam。

【代码7-3-11】train\_novel.py

# 编译模型

model.compile(loss='categorical\_crossentropy', optimizer='adam')

执行代码块。

7.3.12 定义checkpoint

由于模型比较复杂，所以定义checkpoint，使得在训练过程中只要模型性能有所提高便保存模型。这样做的好处是：一旦断电或机器重启，无需从头开始重新训练，可以加载已经训练好的模型继续训练。

在训练过程中模型会被不断保存到output目录下。

【代码7-3-12】train\_novel.py

# 定义checkpoint

checkpoint = ModelCheckpoint(model\_file, monitor='loss', verbose=1, save\_best\_only=True,

save\_weights\_only=False, mode='min')

callbacks\_list = [checkpoint]

执行代码块。

7.3.13 训练模型

最后训练模型。

在4核CPU环境下模型训练1个epoch大约需要16分钟。30个epoch大约需要8个小时。

考虑到同学们宝贵的实验时间，可以手动终止程序。

模型没有训练完也没有关系，本案例提供了预训练模型，【7.4 生成相似文本】小节的运行不受影响。

【代码7-3-13】train\_novel.py

# 训练模型

H = model.fit(X, y, epochs=EPOCHS, batch\_size=BATCH\_SIZE, callbacks=callbacks\_list)

执行代码块。

如果手动终止了程序，请直接跳到【7.4 生成相似文本】小节。

模型训练过程如图10所示。

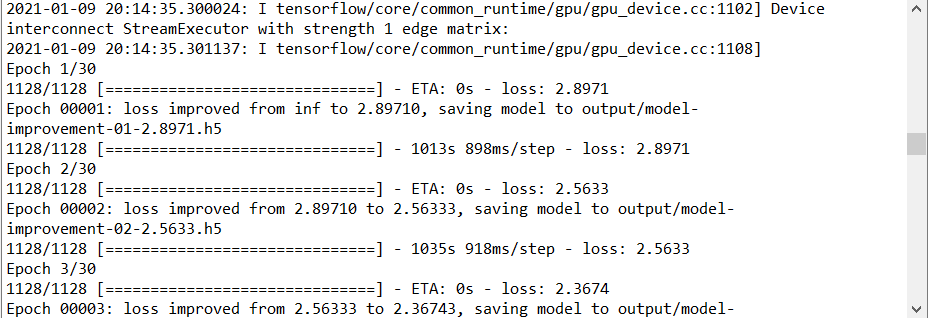


图10 模型训练过程

注意：由于模型初始会设置随即权重，所以同学们的结果不会和本案例一模一样。

由上图可知，checkpoint也被打印出来了，即:只要模型性能有所提升，模型便会保存下来。比如，第2个epoch训练结束后，loss值从2.89710降低到了2.56333，所以模型会被保存下来。Checkpoint信息如图11所示。

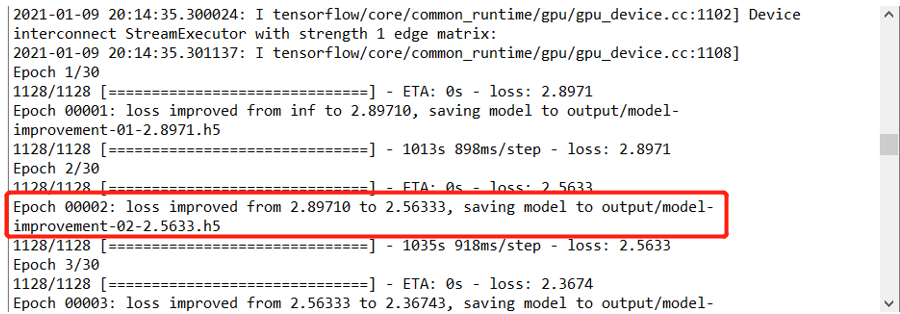


图11 checkpoint信息

模型训练结束后，output目录下会保存多个模型权重文件，如图12所示。

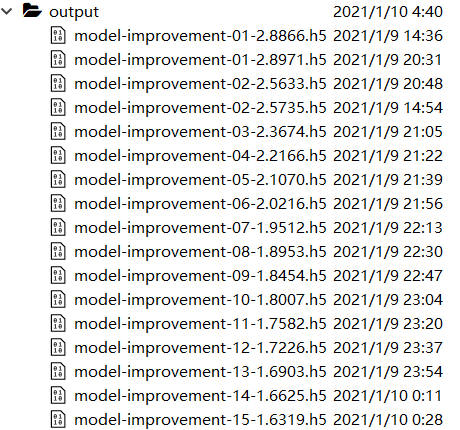


图12 output目录下的多个模型权重文件

7.3.14 验证模型是否收敛

模型训练30个epoch是否足够呢？如果loss曲线收敛了，说明30个epoch是足够的。如果loss曲线没有收敛，说明还需要继续训练。

如果模型没有训练完毕，执行此段代码会报错。

画出loss曲线的代码如下：

【代码7-3-14】train\_novel.py

# 画出loss曲线

plt.figure()

plt.plot(np.arange(0, EPOCHS), H.history['loss'], label = 'train\_loss')

plt.title('Training loss')

plt.xlabel('Epoch #')

plt.ylabel('Loss')

plt.legend(loc = 'lower left')

plt.show()

执行代码块。

Loss曲线如图13所示。

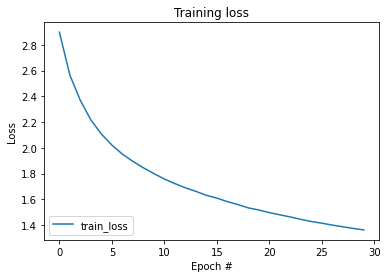


图13 Loss曲线

由上图可知，loss是逐渐下降的，说明模型性能每经过1个epoch都有所提升。Loss曲线并没有趋于平稳，所以30个epoch是不够的。

考虑到同学们宝贵的实验时间，就不再继续训练了。在时间充足的情况下，可以再训练几十个epoch。

模型训练到此结束。接下来使用训练好的模型生成相似文本。

模型没有训练完也没有关系，本案例提供了预训练模型。

7.4生成相似文本

调用预训练模型继续写小说。

向test\_novel.py中添加代码。

由于这是一个新的需要执行的.py文件，建议打开一个新的console，或者重启kernel。

7.4.1 导入库

【代码7-4-1】test\_novel.py

# -\*- coding: utf-8 -\*-

"""

调用模型生成小说

"""

# 导入库

import numpy as np

import sys

from keras.models import load\_model

from process\_data import process\_novel\_data

执行代码块。

7.4.2 定义全局变量和全局常量

全局常量是序列的长度，必须和训练模型时的序列长度一致。

全局变量是小说路径和预训练模型的路径。

【代码7-4-2】test\_novel.py

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

# 定义全局变量

text\_file = "data/wonderland.txt"

model\_file = 'output/model-improvement-30-1.3634.h5'

执行代码块。

7.4.3 得到处理后的数据

调用文本处理函数，得到处理后的数据。

【代码7-4-3】test\_novel.py

# 得到处理后的数据

dataX, dataY, n\_vocab, int\_to\_char = process\_novel\_data(text\_file, SEQ\_LENGTH)

执行代码块。

7.4.4 加载预训练模型

加载预训练模型，准备用它生成小说。

【代码7-4-4】test\_novel.py

# 加载预训练模型

model = load\_model(model\_file)

执行代码块。

7.4.5 随机从dataX抽取1行

【代码7-4-5】test\_novel.py

# 随机从dataX中取1行（有100列）

start = np.random.randint(0, len(dataX)-1)

pattern = dataX[start]

执行代码块。

start和pattern变量的信息如图14所示。

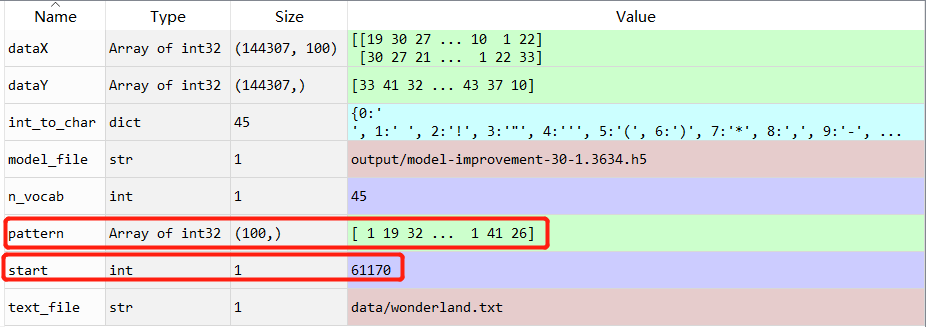


图14 start和pattern变量的信息

7.4.6 随机抽取1段文字

该段文字有100个字符。代码如下：

【代码7-4-6】test\_novel.py

print("从小说中随机抽取的1段序列为:")

print(''.join([int\_to\_char[value] for value in pattern]))

执行代码块。

随机抽取的一段文字如图15所示。

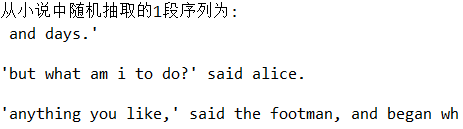


图15 随机抽取的一段文字

在原文中，这段文字后面的内容如图16所示。

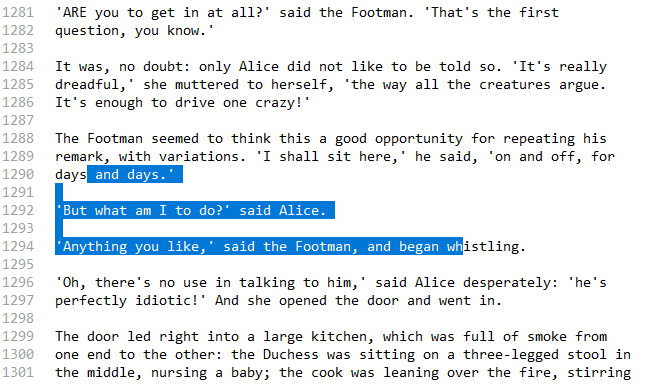


图16 随机抽取的文字后面的内容

7.4.7 生成相似文本

生成相似的100个字符。

调整similar\_text\_n\_chars变量的值改变生成的相似文本的字符数。

代码如下：

【代码7-4-7】test\_novel.py

# 生成相似文本

print("模型生成的序列为:")

similar\_text\_n\_chars = 100 # 相似文本的字符数

for i in range(similar\_text\_n\_chars):

x = np.reshape(pattern, (1, len(pattern), 1))# 将pattern转换为3维 [samples, time steps, features]

x = x / float(n\_vocab) # 数据标准化

prediction = model.predict(x, verbose=0) # 得到预测向量

index = np.argmax(prediction) # 得到预测的类别（共45个类别）

result = int\_to\_char[index]# 将预测的数字转换为字符

sys.stdout.write(result) # 输出预测值

pattern = np.append(pattern, index) # 在pattern后面添加新预测的字符

pattern = pattern[1:len(pattern)] # 把pattern第1个字符去掉，因为用不到了

执行代码块。

生成的相似文本如图17所示。



图17 生成的相似文本

结论：LSTM模型有能力写出相似文本。

在本案例中结果不太准确，是因为模型训练的epoch数量不够，模型未收敛。同学们如果有时间请继续训练，直至模型收敛，然后查看生成的相似文本效果。

如果同学们感兴趣，可以尝试更高级的文本向量化技术提高模型性能。请查阅资料了解此知识点。

8. 案例代码

【案例代码】process\_data\_temp.py

# -\*- coding: utf-8 -\*-

"""

小说文本处理

"""

# 导入库

import numpy as np

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

# 定义全局变量

text\_file = "data/wonderland.txt"

# 读取小说，并转换为小写

raw\_text = open(text\_file).read() # 读取

raw\_text = raw\_text.lower() # 转换为小写字母

# 得到 字符->数字 和 数字->字符的对应关系

chars = sorted(list(set(raw\_text)))

char\_to\_int = dict((c, i) for i, c in enumerate(chars))

int\_to\_char = dict((i, c) for i, c in enumerate(chars))

# 统计小说信息

n\_chars = len(raw\_text)

n\_vocab = len(chars)

print("小说的总字符个数是: ", n\_chars)

print("小说的字符种类是: ", n\_vocab)

# 构造自变量和因变量

dataX\_temp = []

dataY\_temp = []

for i in range(0, n\_chars - SEQ\_LENGTH):

seq\_in = raw\_text[i:i + SEQ\_LENGTH]

seq\_out = raw\_text[i + SEQ\_LENGTH]

dataX\_temp.append(seq\_in)

dataY\_temp.append(seq\_out)

# 使用词袋模型对自变量与因变量做文本向量化

dataX = []

dataY = []

for i in dataX\_temp:

dataX.append([char\_to\_int[char] for char in i])

for i in dataY\_temp:

dataY.append(char\_to\_int[i])

# 将dataX和dataY转换为NumPy类型

dataX = np.array(dataX)

dataY = np.array(dataY)

【案例代码】process\_data.py

# -\*- coding: utf-8 -\*-

"""

小说文本处理

"""

# 导入库

import numpy as np

def process\_novel\_data(text\_file, SEQ\_LENGTH):

# 读取小说，并转换为小写

raw\_text = open(text\_file).read() # 读取

raw\_text = raw\_text.lower() # 转换为小写字母

# 得到 字符->数字 和 数字->字符的对应关系

chars = sorted(list(set(raw\_text)))

char\_to\_int = dict((c, i) for i, c in enumerate(chars))

int\_to\_char = dict((i, c) for i, c in enumerate(chars))

# 统计小说信息

n\_chars = len(raw\_text)

n\_vocab = len(chars)

print("小说的总字符个数是: ", n\_chars)

print("小说的字符种类是: ", n\_vocab)

# 构造自变量和因变量

dataX\_temp = []

dataY\_temp = []

for i in range(0, n\_chars - SEQ\_LENGTH):

seq\_in = raw\_text[i:i + SEQ\_LENGTH]

seq\_out = raw\_text[i + SEQ\_LENGTH]

dataX\_temp.append(seq\_in)

dataY\_temp.append(seq\_out)

# 使用词袋模型对自变量与因变量做文本向量化

dataX = []

dataY = []

for i in dataX\_temp:

dataX.append([char\_to\_int[char] for char in i])

for i in dataY\_temp:

dataY.append(char\_to\_int[i])

# 将dataX和dataY转换为NumPy类型

dataX = np.array(dataX)

dataY = np.array(dataY)

return dataX, dataY, n\_vocab, int\_to\_char

【案例代码】train\_novel.py

# -\*- coding: utf-8 -\*-

"""

构建模型、训练模型、保存模型

"""

# 导入库

import numpy as np

from keras.utils import np\_utils

import matplotlib.pyplot as plt

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import LSTM

from keras.callbacks import ModelCheckpoint

from process\_data import process\_novel\_data

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

EPOCHS = 30

BATCH\_SIZE = 128

# 定义全局变量

text\_file = "data/wonderland.txt"

model\_file = 'output/model-improvement-{epoch:02d}-{loss:.4f}.h5'

# 得到处理后的数据

dataX, dataY, n\_vocab, int\_to\_char = process\_novel\_data(text\_file, SEQ\_LENGTH)

# 将自变量转换为3维 [samples, time steps, features]

X = np.reshape(dataX, (dataX.shape[0], dataX.shape[1], 1))

# 标准化

X = X / float(n\_vocab)

# 独热编码

y = np\_utils.to\_categorical(dataY)

# 初始化LSTM模型

model = Sequential()

# 构建输入层+第1层LSTM

model.add(LSTM(256, input\_shape=(X.shape[1], X.shape[2]), return\_sequences=True))

model.add(Dropout(0.2)) # 防止过拟合

# 构建第2层LSTM层

model.add(LSTM(256))

model.add(Dropout(0.2))

# 构建输出层

model.add(Dense(y.shape[1], activation='softmax'))

# 编译模型

model.compile(loss='categorical\_crossentropy', optimizer='adam')

# 定义checkpoint

checkpoint = ModelCheckpoint(model\_file, monitor='loss', verbose=1, save\_best\_only=True,

save\_weights\_only=False, mode='min')

callbacks\_list = [checkpoint]

# 训练模型

H = model.fit(X, y, epochs=EPOCHS, batch\_size=BATCH\_SIZE, callbacks=callbacks\_list)

# 画出loss曲线

plt.figure()

plt.plot(np.arange(0, EPOCHS), H.history['loss'], label = 'train\_loss')

plt.title('Training loss')

plt.xlabel('Epoch #')

plt.ylabel('Loss')

plt.legend(loc = 'lower left')

plt.show()

【案例代码】test\_novel.py

# -\*- coding: utf-8 -\*-

"""

调用模型生成小说

"""

# 导入库

import numpy as np

import sys

from keras.models import load\_model

from process\_data import process\_novel\_data

# 定义全局常量

SEQ\_LENGTH = 100 # 序列长度是100

# 定义全局变量

text\_file = "data/wonderland.txt"

model\_file = 'output/model-improvement-30-1.3634.h5'

# 得到处理后的数据

dataX, dataY, n\_vocab, int\_to\_char = process\_novel\_data(text\_file, SEQ\_LENGTH)

# 加载预训练模型

model = load\_model(model\_file)

# 随机从dataX中取1行（有100列）

start = np.random.randint(0, len(dataX)-1)

pattern = dataX[start]

print("从小说中随机抽取的1段序列为:")

print(''.join([int\_to\_char[value] for value in pattern]))

# 生成相似文本

print("模型生成的序列为:")

similar\_text\_n\_chars = 100 # 相似文本的字符数

for i in range(similar\_text\_n\_chars):

x = np.reshape(pattern, (1, len(pattern), 1))# 将pattern转换为3维 [samples, time steps, features]

x = x / float(n\_vocab) # 数据标准化

prediction = model.predict(x, verbose=0) # 得到预测向量

index = np.argmax(prediction) # 得到预测的类别（共45个类别）

result = int\_to\_char[index]# 将预测的数字转换为字符

sys.stdout.write(result) # 输出预测值

pattern = np.append(pattern, index) # 在pattern后面添加新预测的字符

pattern = pattern[1:len(pattern)] # 把pattern第1个字符去掉，因为用不到了