

# 循环神经网络





# 目录

Contents



◆ 自然语言处理概述

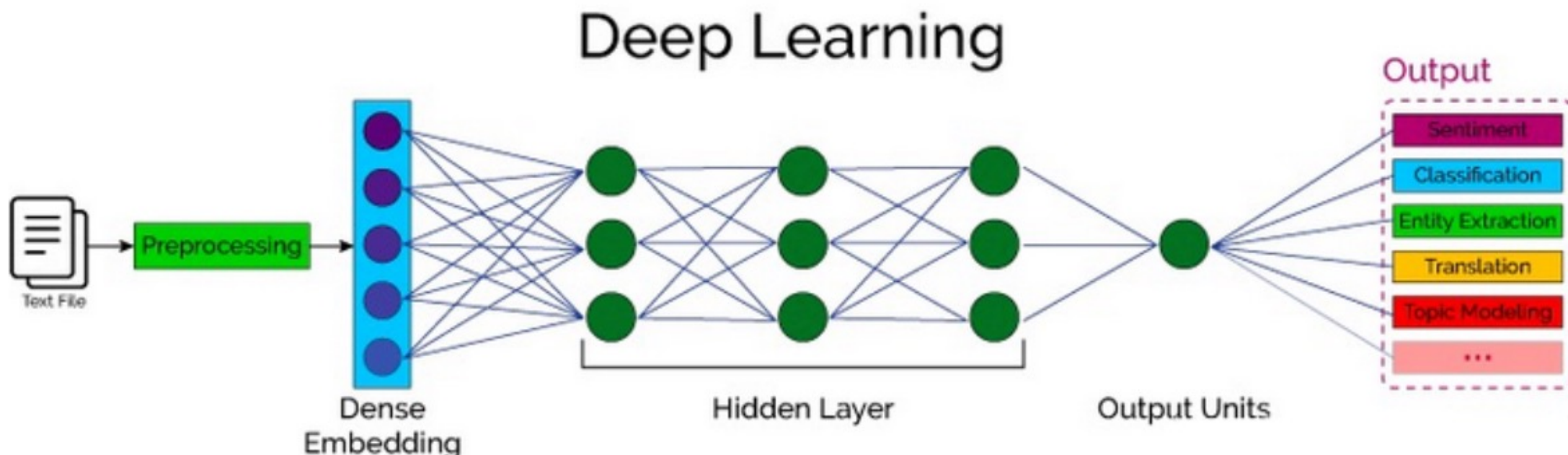
◆ 词嵌入层

◆ 循环网络RNN

◆ 文本生成案例

## 自然语言处理概述

自然语言处理（Nature language Processing, NLP）研究的主要是通过计算机算法来理解自然语言。对于自然语言来说，处理的数据主要就是人类的语言，例如：汉语、英语、法语等，该类型的数据不像我们前面接触过的结构化数据、或者图像数据可以很方便的进行数值化。





# 目录

Contents



- ◆ RNN概述
- ◆ 词嵌入层
- ◆ 循环网络层
- ◆ 文本生成案例

# 学习目标

Learning Objectives

1. 知道词嵌入概念
2. 掌握PyTorch词嵌入API

## 词嵌入层

词嵌入层的作用就是将文本转换为向量。

词嵌入层首先会根据输入的词的数目构建一个**词向量矩阵**，例如：我们有 100 个词，每个词希望转换成 128 维度的向量，那么构建的矩阵形状即为：100\*128，输入的每个词都对应了一个该矩阵中的一个向量。

<b>cat</b>	=>	1.2	-0.1	4.3	3.2
<b>mat</b>	=>	0.4	2.5	-0.9	0.5
<b>on</b>	=>	2.1	0.3	0.1	0.4
...					

## 词嵌入层

在 PyTorch 中，使用 `nn.Embedding` 词嵌入层来实现输入词的向量化。

```
nn.Embedding(num_embeddings=10, embedding_dim=4)
```

`nn.Embedding` 对象构建时，最主要有两个参数：

1. `num_embeddings` 表示**词的数量**
2. `embedding_dim` 表示**用多少维的向量来表示每个词**

## 词嵌入层

接下来，我们将会学习如何将词转换为词向量，其步骤如下：

1. 先将语料进行分词，构建词与索引的映射，我们可以把这个映射叫做词表，词表中每个词都对应了一个唯一的索引
2. 然后使用 `nn.Embedding` 构建词嵌入矩阵，词索引对应的向量即为该词对应的数值化后的向量表示。

例如，我们的文本数据为："北京冬奥的进度条已经过半，不少外国运动员在完成自己的比赛后踏上归途。"，



## 词嵌入层

接下来，我们就实现下刚才的需求：

```
import torch
import torch.nn as nn
import jieba

if __name__ == '__main__':
    # 0. 文本数据
    text = '北京冬奥的进度条已经过半，不少外国运动员在完成自己的比赛后踏上归途。'
    # 1. 文本分词
    words = jieba.lcut(text)
    print('文本分词:', words)
    # 2. 分词去重并保留原来的顺序获取所有的词语
    unique_words = list(set(words))
    print("去重后词的个数:\n", len(unique_words))
    # 3. 构建词嵌入层:num_embeddings: 表示词的总数量;embedding_dim: 表示词嵌入的维度
    embed = nn.Embedding(num_embeddings=len(unique_words), embedding_dim=4)
    print("词嵌入的结果: \n", embed)
    # 4. 词语的词向量表示
    for i, word in enumerate(unique_words):
        # 获得词嵌入向量
        word_vec = embed(torch.tensor(i))
        print('%3s\t' % word, word_vec)
```

## 词嵌入层

输出的结果是：

文本分词：['北京', '冬奥', '的', '进度条', '已经', '过半', ' ', ' ', '不少', '外国', '运动员']  
去重后词的个数：

18

词嵌入的结果：

Embedding(18, 4)

```
进度条  tensor([-0.3230,  0.6118,  1.1127, -1.8573], grad_fn=<EmbeddingBackward0>)
北京   tensor([ 0.4924,  1.9053,  0.5551, -0.4056], grad_fn=<EmbeddingBackward0>)
冬奥   tensor([-0.7237, -0.3153, -1.0946,  0.5241], grad_fn=<EmbeddingBackward0>)
后     tensor([ 1.7348, -0.3621, -0.1740,  0.6375], grad_fn=<EmbeddingBackward0>)
运动员 tensor([ 0.9090,  0.2903,  1.3590, -1.0113], grad_fn=<EmbeddingBackward0>)
归途   tensor([-0.8221,  0.4773,  1.0013, -0.1903], grad_fn=<EmbeddingBackward0>)
已经   tensor([ 0.2596,  1.5450,  1.7050, -0.3949], grad_fn=<EmbeddingBackward0>)
比赛   tensor([-0.9220, -1.0200, -0.3822,  0.5478], grad_fn=<EmbeddingBackward0>)
踏上   tensor([ 1.2970, -0.2169,  0.3552,  0.1369], grad_fn=<EmbeddingBackward0>)
,       tensor([ 0.3914, -0.4157,  1.1152,  0.0727], grad_fn=<EmbeddingBackward0>)
自己   tensor([-1.5722, -0.1995,  0.1097, -0.4549], grad_fn=<EmbeddingBackward0>)
完成   tensor([ 0.2163, -0.9119,  1.1095,  0.8218], grad_fn=<EmbeddingBackward0>)
过半   tensor([0.9948, 2.6069, 1.5194, 0.4751], grad_fn=<EmbeddingBackward0>)
外国   tensor([-0.0833,  0.6791,  0.0911, -0.8597], grad_fn=<EmbeddingBackward0>)
。     tensor([ 0.2335, -1.3990,  0.8407, -0.5460], grad_fn=<EmbeddingBackward0>)
不少   tensor([ 0.2476, -0.5952, -0.3386,  0.7944], grad_fn=<EmbeddingBackward0>)
的     tensor([ 0.6788,  0.6474, -1.2109,  0.0276], grad_fn=<EmbeddingBackward0>)
在     tensor([-1.9989,  0.4322, -0.5250,  0.2621], grad_fn=<EmbeddingBackward0>)
```



# 总结

## 1. 词嵌入层的作用

主要作用就是将输入的词映射为词向量，便于在网络模型中进行计算。

## 2. 词嵌入层的API

```
nn.Embedding(num_embeddings=10, embedding_dim=4)
```



# 目录

Contents

- ◆ RNN概述
- ◆ 词嵌入层
-  ◆ 循环网络层
- ◆ 文本生成案例



# 学习目标

Learning Objectives

1. 掌握RNN网络原理
2. 掌握PyTorch RNN API

## RNN网络原理

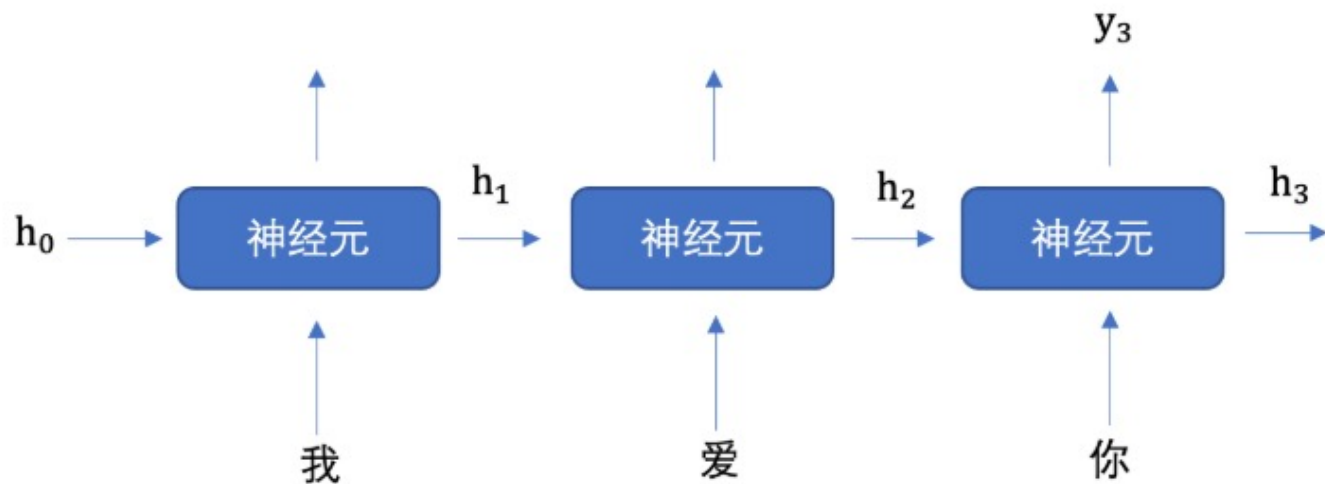
文本数据是具有序列特性的

例如："我爱你"，这串文本就是具有序列关系的，"爱"需要在"我"之后，"你"需要在"爱"之后，如果颠倒了顺序，那么可能就会表达不同的意思。

为了表示出数据的序列关系，需要使用循环神经网络(Recurrent Neural Networks, RNN) 来对数据进行建模，RNN 是一个作用于处理带有序列特点的样本数据。

## RNN网络原理

RNN 是如何计算过程是什么样的呢?

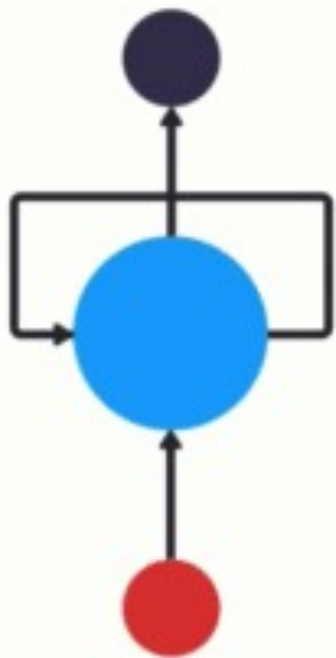


$h$  表示隐藏状态，  
每一次的输入都会包含两个值：上一个时间步的隐藏状态、当前状态的输入值，输出当前时间步的隐藏状态和当前时间步的预测结果。

## RNN网络原理

上一页PPT中画了 3 个神经元, 但是实际上只有一个神经元, "我爱你" 三个字是重复输入到同一个神经元中

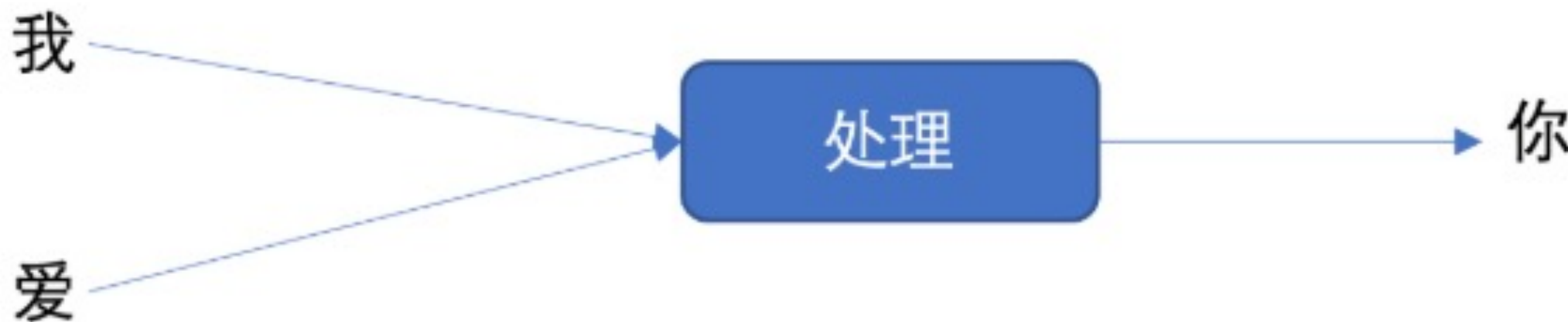
。





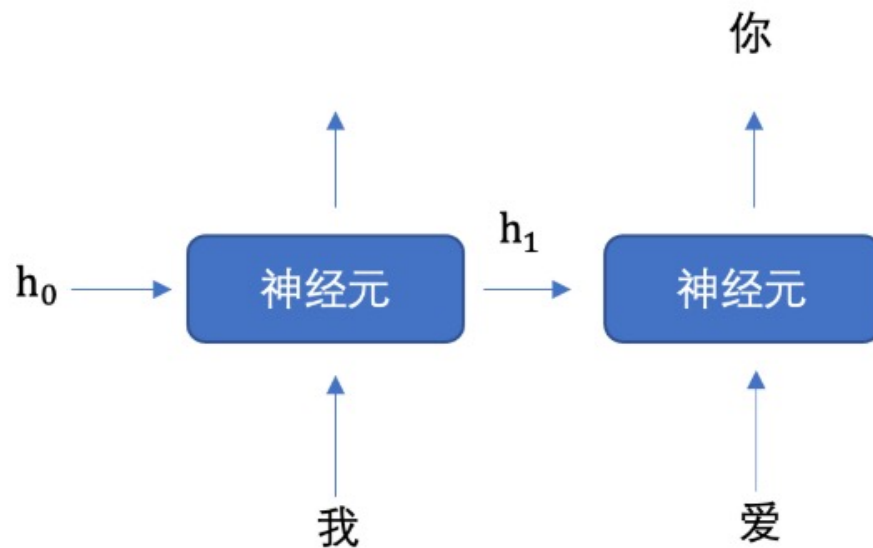
## RNN网络原理

我们举个例子来理解上图的工作过程，假设我们要实现文本生成，也就是输入 "我爱" 这两个字，来预测出 "你"，其如下图所示：



## RNN网络原理

将上图展开成不同时间步的形式，如下图所示：



首先初始化出第一个隐藏状态 $h_0$ ，一般都是全0的一个向量，然后将“我”进行词嵌入，转换为向量的表示形式，送入到第一个时间步，然后输出隐藏状态 $h_1$ ，然后将 $h_1$ 和“爱”输入到第二个时间步，得到隐藏状态 $h_2$ ，将 $h_2$ 送入到全连接网络，得到“你”的预测概率。

## RNN网络原理

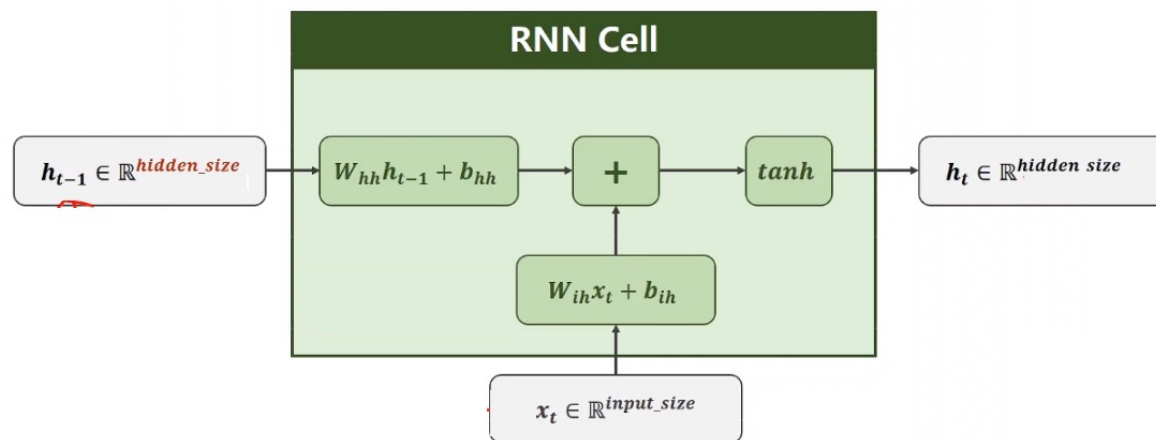
每个神经元内部是如何计算的呢？

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

上述公式中：

1.  $W_{ih}$  表示输入数据的权重
2.  $b_{ih}$  表示输入数据的偏置
3.  $W_{hh}$  表示输入隐藏状态的权重
4.  $b_{hh}$  表示输入隐藏状态的偏置

最后对输出的结果使用  $\tanh$  激活函数进行计算，得到该神经元的输出。



## PyTorch RNN层的使用

### ● API介绍

```
RNN = torch.nn.RNN(input_size, hidden_size, num_layer)
```

参数意义是：

1. input\_size: 输入数据的维度，一般设为词向量的维度；
2. hidden\_size: 隐藏层h的维数，也是当前层神经元的输出维度；
3. num\_layer: 隐藏层h的层数，默认为1.

将RNN实例化就可以将数据送入进行处理。

## PyTorch RNN层的使用

- 输入数据和输出结果

将RNN实例化就可以将数据送入其中进行处理，处理的方式如下所示：

```
output, hn = RNN(x, h0)
```

- 输入数据:输入主要包括词嵌入的x、初始的隐藏层h0

- x的表示形式为[seq\_len, batch, input\_size]，即[句子的长度，batch的大小，词向量的维度]
- h0的表示形式为[num\_layers, batch, hidden\_size]，即[隐藏层的层数，batch的大，隐藏层h的维数]

- 输出结果：主要包括输出结果output,最后一层的hn

- output的表示形式与输入x类似，为[seq\_len, batch, hidden\_size]，即[句子的长度，batch的大小，输出向量的维度]
- hn的表示形式与输入h0一样，为[num\_layers, batch, hidden\_size]，即[隐藏层的层数，batch的大，隐藏层h的维度]

## PyTorch RNN层的使用

```
import torch
import torch.nn as nn

# RNN层送入批量数据
def test():
    # 词向量维度 128, 隐藏向量维度 256
    rnn = nn.RNN(input_size=128, hidden_size=256)
    # 第一个数字: 表示句子长度, 也就是词语个数
    # 第二个数字: 批量个数, 也就是句子的个数
    # 第三个数字: 词向量维度
    inputs = torch.randn(5, 32, 128)
    hn = torch.zeros(1, 32, 256)
    # 获取输出结果
    output, hn = rnn(inputs, hn)
    print("输出向量的维度: \n", output.shape)
    print("隐含层输出的维度: \n", hn.shape)

if __name__ == '__main__':
    test()
```

输出结果:

输出向量的维度: torch.Size([5, 32, 256])

隐含层的输出结果: torch.Size([1, 32, 256])



# 总结

## 1、RNN网络原理

处理带有序列特点的样本数据

## 2、pyTorch RNN API

```
RNN = torch.nn.RNN(input_size, hidden_size, num_layer)
```



# 目录

Contents



- ◆ RNN概述
- ◆ 词嵌入层
- ◆ 循环网络层
- ◆ 文本生成案例



# 学习目标

Learning Objectives

1. 掌握文本生成模型构建流程

## 项目需求

文本生成任务是一种常见的自然语言处理任务，输入一个开始词能够预测出后面的词序列。本案例将会使用循环神经网络来实现周杰伦歌词生成任务。



## 项目实施

导入工具包：

```
import torch
import re
import jieba
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import time
```

## 数据集

我们收集了周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中的歌词，来训练神经网络模型，当模型训练好后，我们就可以用这个模型来创作歌词。数据集如下：

```
想要有直升机  
想要和你飞到宇宙去  
想要和你融化在一起  
融化在宇宙里  
我每天每天每天在想想想著你  
这样的甜蜜  
让我开始相信命运  
感谢地心引力  
让我碰到你  
漂亮的让我面红的可爱女人  
...
```

该数据集共有 5819 行文本。

## 获取数据集并构建词表

在进行自然语言处理任务之前，首要做的就是构建词表。

所谓的词表就是将数据进行分词，然后给每一个词分配一个唯一的编号，便于我们送入词嵌入层获取每个词的词向量。

1	who
2	when
3	where
4	what
5	why
6	whose
7	which

## 获取数据集并构建词表

接下来, 我们对周杰伦歌词的数据进行处理构建词表, 具体实现如下所示:

整体流程是:

- 获取文本数据
- 分词, 并进行去重
- 构建词表

## 构建词表

```
# 获取数据, 并进行分词, 构建词表
def build_vocab():
    # 数据集位置
    file_name = 'data/jaychou_lyrics.txt'
    # 分词结果存储位置
    unique_words = []
    all_words = []
    # 遍历数据集中的每一行文本
    for line in open(file_name, 'r'):
        # 使用jieba分词, 分割结果是一个列表
        words = jieba.lcut(line)
        # print(words)
        # 所有的分词结果存储到all_sentences, 其中包含重复的词组
        all_words.append(words)
        # 遍历分词结果, 去重后存储到unique_words
        for word in words:
            if word not in unique_words:
                unique_words.append(word)
    # 语料中词的数量
    word_count = len(unique_words)
```

## 构建词表

```
# 词到索引映射
word_to_index = {word: idx for idx, word in enumerate(unique_words)}
# 词表索引表示
corpus_idx = []
# 遍历每一行的分词结果
for words in all_words:
    temp = []
    # 获取每一行的词，并获取相应的索引
    for word in words:
        temp.append(word_to_index[word])
    # 在每行词之间添加空格隔开
    temp.append(word_to_index[' '])
    # 获取当前文档中每个词对应的索引
    corpus_idx.extend(temp)
return unique_words, word_to_index, word_count, corpus_idx
```



## 构建词表

```
if __name__ == "__main__":  
    ## 获取数据  
    unique_words, word_to_index, word_count, corpus_idx = build_vocab()  
    print("词的数量: \n", word_count)  
    print("去重后的词: \n", unique_words)  
    print("每个词的索引: \n", word_to_index)  
    print("当前文档中每个词对应的索引: \n", corpus_idx)
```

我们的词典主要包含了:

- 1.unique\_words: 存储了词到编号(编号是索引)的映射
- 2.index\_to\_word: 存储了编号到词的映射

词的数量:

5703

去重后的词:

['想要', '有', '直升机', ..., '做作', '天生', '甚至', '会怪', '\*', '充分', '太好', '楼',

每个词的索引:

{'想要': 0, '有': 1, '直升机': 2, ..., '做作': 5689, '天生': 5690, '甚至': 5691, '会怪':

当前文档中每个词对应的索引:

[0, 1, 2, 3, 40, 0, 4, 5, 6, 7, 8, 3, 40, 0, 4, 5, 9, 10, 11, 3, 40, 9, 10, 7, 12,

## 构建数据集对象

我们在训练的时候，为了便于读取语料，我们会构建一个 Dataset 对象，如下所示：

```
class LyricsDataset(torch.utils.data.Dataset):
    def __init__(self, corpus_idx, num_chars):
        # 文档数据中词的索引
        self.corpus_idx = corpus_idx
        # 每个句子中词的个数
        self.num_chars = num_chars
        # 词的数量
        self.word_count = len(self.corpus_idx)
        # 句子数量
        self.number = self.word_count // self.num_chars
    def __len__(self):
        # 返回句子数量
        return self.number
    def __getitem__(self, idx):
        # idx指词的索引，并将其修正索引值到文档的范围里面
        start = min(max(idx, 0), self.word_count - self.num_chars - 2)
        # 输入值
        x = self.corpus_idx[start: start + self.num_chars]
        # 网络预测结果（目标值）
        y = self.corpus_idx[start + 1: start + 1 + self.num_chars]
        # 返回结果
        return torch.tensor(x), torch.tensor(y)
```

## 构建数据集对象

我们在训练的时候，为了便于读取语料，我们会构建一个 Dataset 对象，如下所示：

```
if __name__ == "__main__":  
    # 数据获取实例化  
    dataset = LyricsDataset(corpus_idx, 5)  
    x, y = dataset.__getitem__(0)  
    print("网络输入值: ", x)  
    print("目标值: ", y)
```

输出结果为：

```
网络输入值:  tensor([ 0,  1,  2,  3, 40])  
目标值:  tensor([ 1,  2,  3, 40,  0])
```

## 构建网络模型

我们用于实现《歌词生成》的网络模型，主要包含了三个层:

1. 词嵌入层: 用于将语料转换为词向量
2. 循环网络层: 提取句子语义
3. 全连接层: 输出对词典中每个词的预测概率

## 构建网络模型

# 模型构建

```
class TextGenerator(nn.Module):
    def __init__(self, word_count):
        super(TextGenerator, self).__init__()
        # 初始化词嵌入层: 词向量的维度为128
        self.ebd = nn.Embedding(word_count, 128)
        # 循环网络层: 词向量维度 128, 隐藏向量维度 128, 网络层数1
        self.rnn = nn.RNN(128, 128, 1)
        # 输出层: 特征向量维度128与隐藏向量维度相同, 词表中词的个数
        self.out = nn.Linear(128, word_count)
    def forward(self, inputs, hidden):
        # 输出维度: (batch, seq_len, 词向量维度 128)
        embed = self.ebd(inputs)
        # 修改维度: (seq_len, batch, 词向量维度 128)
        output, hidden = self.rnn(embed.transpose(0, 1), hidden)
        # 输入维度: (seq_len*batch, 词向量维度) 输出维度: (seq_len*batch, 5073)
        output = self.out(output.reshape((-1, output.shape[-1])))
        # 网络输出结果
        return output, hidden
    def init_hidden(self, bs):
        # 隐藏层的初始化:[网络层数, batch, 隐藏层向量维度]
        return torch.zeros(1, bs=2, 128)
```

## 构建训练函数

前面的准备工作完成之后，我们就可以编写训练函数。训练函数主要负责编写数据迭代、送入网络、计算损失、反向传播、更新参数，其流程基本较为固定。

由于我们要实现文本生成，文本生成本质上，输入一串文本，预测下一个文本，也属于分类问题，所以，我们使用多分类交叉熵损失函数。优化方法我们学习过 SGB、AdaGrad、Adam 等，在这里我们选择学习率、梯度自适应的 Adam 算法作为我们的优化方法。

训练完成之后，我们使用 `torch.save` 方法将模型持久化存储。

## 构建训练函数

```
# 模型训练
def train():
    # 构建词典
    index_to_word, word_to_index, word_count, corpus_idx = build_vocab()
    # 数据集
    lyrics = LyricsDataset(corpus_idx, 32)
    # 初始化模型
    model = TextGenerator(word_count)
    # 损失函数
    criterion = nn.CrossEntropyLoss()
    # 优化方法
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    # 训练轮数
    epoch = 10
```

## 构建训练函数

```
for epoch_idx in range(epoch):
    # 数据加载器
    lyrics_dataloader = DataLoader(lyrics, shuffle=True, batch_size=2)
    # 训练时间
    start = time.time()
    iter_num = 0 # 迭代次数
    # 训练损失
    total_loss = 0.0
    # 遍历数据集
    for x, y in lyrics_dataloader:
        # 隐藏状态的初始化
        hidden = model.init_hidden()
        # 模型计算
        output, hidden = model(x, hidden)
        # 计算损失
        # y:[batch,seq_len]->[seq_len,batch]->[seq_len*batch]
        y = torch.transpose(y, 0, 1).contiguous().view(-1)
        loss = criterion(output, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        iter_num += 1 # 迭代次数加1
        total_loss += loss.item()
    # 打印训练信息
    print('epoch %3s loss: %.5f time %.2f' % (epoch_idx + 1, total_loss / iter_num, time.time() - start))
    # 模型存储
    torch.save(model.state_dict(), 'data/lyrics_model_%d.pth' % epoch)
```



## 调用模型训练函数

```
if __name__ == "__main__":  
    ## 获取数据  
    unique_words, word_to_index, word_count, corpus_idx = build_vocab()  
    # 数据获取实例化  
    dataset = LyricsDataset(corpus_idx, 5)  
    train()
```

输出结果为：

```
epoch    1 loss: 1.16320 time 13.63  
epoch    2 loss: 0.15565 time 14.40  
epoch    3 loss: 0.11856 time 15.30  
epoch    4 loss: 0.10874 time 14.10  
epoch    5 loss: 0.10612 time 18.70  
epoch    6 loss: 0.10446 time 17.40  
epoch    7 loss: 0.10211 time 16.28  
epoch    8 loss: 0.10188 time 16.24  
epoch    9 loss: 0.10124 time 15.49  
epoch   10 loss: 0.10058 time 16.29
```

## 构建预测函数

从磁盘加载训练好的模型，进行预测。预测函数，输入第一个指定的词，我们将该词输入网路，预测出下一个词，再将预测的出的词再次送入网络，预测出下一个词，以此类推，知道预测出我们指定长度的内容。

```
def predict(start_word, sentence_length):  
    # 构建词典  
    index_to_word, word_to_index, word_count, _ = build_vocab()  
    # 构建模型  
    model = TextGenerator(word_count)  
    # 加载参数  
    model.load_state_dict(torch.load('data/lyrics_model_10.pth'))  
    # 隐藏状态  
    hidden = model.init_hidden(bs=1)  
    # 将起始词转换为索引  
    word_idx = word_to_index[start_word]  
    # 产生的词的索引存放位置  
    generate_sentence = [word_idx]  
    # 遍历到句子长度，获取每一个词  
    for _ in range(sentence_length):  
        # 模型预测  
        output, hidden = model(torch.tensor([[word_idx]]), hidden)  
        # 获取预测结果  
        word_idx = torch.argmax(output)  
        generate_sentence.append(word_idx)  
    # 根据产生的索引获取对应的词，并进行打印  
    for idx in generate_sentence:  
        print(index_to_word[idx], end="")
```

## 构建预测函数

```
if __name__ == "__main__":  
    # 调用预测函数  
    predict('分手', 50)
```

输出结果:

分手的话像语言暴力  
我已无能为力再提起 决定中断熟悉  
然后在这里 不限日期  
然后将过去 慢慢温习  
让我爱上你 那场悲剧  
是你完美演出的一场戏



# 总结

构建了一个《歌词生成》的项目，该项目的实现流程如下：

1. 构建词汇表
2. 构建数据对象
3. 编写网络模型
4. 编写训练函数
5. 编写预测函数



传智教育旗下高端IT教育品牌