



# 基于深度学习的文本分类任务

[文本分类任务的实现思路](#)

[文本预处理](#)

[文本分词](#)

[Jieba中文分词](#)

[简介](#)

[安装](#)

[核心功能](#)

[文本分词器SentencePiece](#)

[安装](#)

[模型训练](#)

[2. 转换值转变模型导入数据](#)

[3. 模型结构搭建](#)

[文本结构化转换](#)

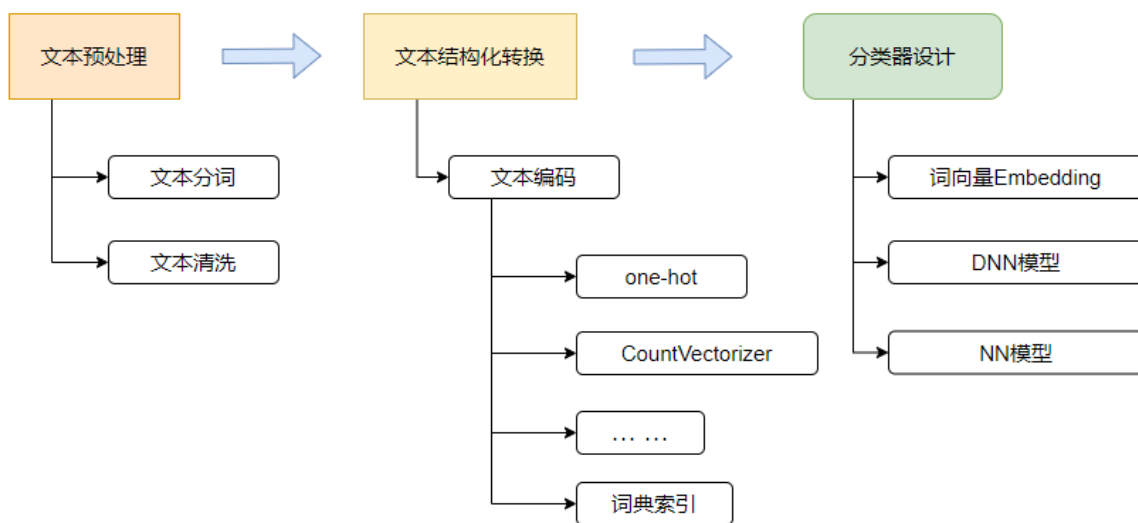
[语料转换为训练样本](#)

[词向量Embedding](#)

[TextRNN](#)

## 文本分类任务的实现思路

文本分类就是一个将文本分配到预定义类别的一个过程。



### 整体流程包括：

- 文本语料的获取和预处理
- 文本语料的结构化编码
- 分类模型设计及实现

其中的文本语料结构化编码是重点，具体实现包括

文本语料 → 通过语料创建词典 → 语料转换为训练样本 → 词向量Embedding

## 文本预处理

模型训练的样本，通常会以句（**sentence**）或段落（**paragraph**）的方式呈现。

所以通常会先进行文本分词，在自然语言处理过程中，称为标记（**tokenize**）。

### 文本分词

英文语料进行分词，相对比较简单。可以直接通过字符串的 `split()` 函数来直接拆分使用空格分隔开的单词。

当然，这种直接的方式也存在着一些问题。例如：文本中的标点符号抽取，连字符的拼接等，也需要注意。

中文语料的话，通常会使用分词工具。例如：jieba、sentence piece

## Jieba中文分词

### 简介

jieba是一款强大的Python中文分词组件，采用MIT授权协议，兼容Python 2/3。它支持精确、全、搜索引擎、paddle四种分词模式，还能处理繁体中文，支持自定义词典，应用广泛。

### 安装

```
pip install jieba
```

低版本升级

```
pip install jieba --upgrade
```

### 核心功能

#### 1. 分词

- `jieba.cut()` 接受字符串，返回可迭代生成器。

参数：

- `cut_all` - 控制是否全模式
- `HMM` - 是否用HMM模型
- `use_paddle` - 是否用paddle模式
- `jieba.cut_for_search` 用于搜索引擎分词。
- `jieba.lcut` 和 `jieba.lcut_for_search` 直接返回python列表。

代码示例：

```

import jieba
strs = ["我来到北京清华大学", "乒乓球拍卖完了", "中国科学技术大学"]
for str in strs:
    seg_list = jieba.cut(str, use_paddle=True)
    print("Paddle Mode: " + '/'.join(list(seg_list)))

seg_list = jieba.cut("我来到北京清华大学", cut_all=True)
print("Full Mode: " + "/ ".join(seg_list))
seg_list = jieba.cut("我来到北京清华大学", cut_all=False)
print("Default Mode: " + "/ ".join(seg_list))
seg_list = jieba.cut("他来到了网易杭研大厦")
print(", ".join(seg_list))

seg_list = jieba.cut_for_search("小明硕士毕业于中国科学院计算所，后在日本京都大学深造")
print(", ".join(seg_list))

```

## 输出：

```

Building prefix dict from the default dictionary ...
Loading model from cache /tmp/jieba.cache
Loading model cost 1.280 seconds.
Prefix dict has been built successfully.
Paddle Mode: 我/来到/北京/清华大学
Paddle Mode: 乒乓球/拍卖/完了
Paddle Mode: 中国/科学技术/大学
Full Mode: 我/ 来到/ 北京/ 清华/ 清华大学/ 华大/ 大学
Default Mode: 我/ 来到/ 北京/ 清华大学
他, 来到, 了, 网易, 杭研, 大厦
小明, 硕士, 毕业, 于, 中国, 科学, 学院, 科学院, 中国科学院, 计算, 计算所, , , 后, 在, 日本, 京都, 大学, 日本京都大学, 深造

```

## 2. 添加自定义词典

- 使用 `jieba.load_userdict(file_name)` 加载UTF-8编码的外部词典文件。

词典每行格式为：

词语 词频(可省略) 词性(可省略)

- 使用 `add_word(word, freq=None, tag=None)` 和 `del_word(word)` 可以动态修改词典，
- 使用 `suggest_freq(segment, tune=True)` 可以调节词频。

## 3. 关键词提取

- 基于TF-IDF算法：** `jieba.analyse.extract_tags(sentence, topK=20, withWeight=False, allowPOS=())`，可指定返回关键词数量、是否返回权重、词性筛选条件。还能切换IDF和停止词自定义语料库路径。
- 基于TextRank算法：** `jieba.analyse.textrank(sentence, topK=20, withWeight=False, allowPOS=('ns', 'n', 'vn', 'v'))`，默认过滤词性。

4. **词性标注**: `jieba.posseg.POSTokenizer(tokenizer=None)` 新建分词器, `jieba.posseg.dt` 为默认词性标注分词器。支持paddle模式下词性标注, 用 `pseg.cut` 方法。

代码示例:

```
import jieba
import jieba.posseg as pseg

words = pseg.cut("我爱北京天安门")
for word, flag in words:
    print('%s %s' % (word, flag))
```

输出:

```
Building prefix dict from the default dictionary ...
Loading model from cache /tmp/jieba.cache
Loading model cost 0.600 seconds.
Prefix dict has been built successfully.
我 r
爱 v
北京 ns
天安门 ns
```

5. **并行分词**: 基于 `multiprocessing` 模块, 仅支持默认分词器。 `jieba.enable_parallel(4)` 开启 (参数为进程数), 可以通过调用 `jieba.disable_parallel()` 方法关闭。

6. **Tokenize (返回词语位置)**: 有默认和搜索模式

代码示例:

```
result = jieba.tokenize(u'永和服装饰品有限公司')
for tk in result:
    print("word %s\t\t start: %d \t\t end:%d" % (tk[0],tk[1],tk[2]))

result = jieba.tokenize(u'永和服装饰品有限公司', mode='search')
for tk in result:
    print("word %s\t\t start: %d \t\t end:%d" % (tk[0],tk[1],tk[2]))
```

输出:

```
Building prefix dict from the default dictionary ...
Loading model from cache /tmp/jieba.cache
Loading model cost 0.555 seconds.
Prefix dict has been built successfully.
word 永和\t\t start: 0 \t\t end:2
word 服装\t\t start: 2 \t\t end:4
word 饰品\t\t start: 4 \t\t end:6
word 有限公司\t\t start: 6 \t\t end:10
word 永和\t\t start: 0 \t\t end:2
word 服装\t\t start: 2 \t\t end:4
word 饰品\t\t start: 4 \t\t end:6
word 有限\t\t start: 6 \t\t end:8
word 公司\t\t start: 8 \t\t end:10
word 有限公司\t\t start: 6 \t\t end:10
```

- 命令行使用

`python -m jieba [options] filename`，可指定输入文件，有多种可选参数，如指定分隔符、启用词性标注、使用自定义词典等。若未指定文件名，则从标准输入读取。

- 其他操作

1. **延迟加载与词典设置**：Jieba延迟加载，可手动 `jieba.initialize()` 初始化。还能用 `jieba.set_dictionary('data/dict.txt.big')` 指定主词典路径。
2. **更换词典**：可下载其他词典（如 `dict.txt.small` 内存小，`dict.txt.big` 对繁体支持好）覆盖 `jieba/dict.txt`，或用 `jieba.set_dictionary` 指定。

## 文本分词器SentencePiece

SentencePiece 是一种无监督的文本分词器和去分词器，主要用于基于神经网络的文本生成系统，其中词汇量在神经模型训练之前预先确定。SentencePiece 实现了子词单元 **subword units**（例如，字节对编码 byte-pair-encoding(BPE))和一元语言模型 **unigram language model**），并扩展了从原始句子的直接训练方式。SentencePiece 允许我们制作一个纯粹的端到端系统，不依赖于特定于语言的预处理/后处理。

<https://github.com/google/sentencepiece>

### 需要预定义token的数量

通常情况下，基于神经网络的机器翻译模型使用的都是固定词汇表。不像绝大多数假设无限词汇量的无监督分词算法，SentencePiece训练分词模型是要确保最终词汇量是固定的，例如 8k、16k 或 32k。

请注意，SentencePiece 指定了训练的最终词汇量大小，这与使用合并操作次数的 subword-nmt 不同。合并操作的次数是 BPE 特定的参数，不适用于其他分割

算法，包括 unigram、word 和 character。

## 通过原始语料训练

以前的分词实现需要输入的语句进行预标记处理。这种要求是训练确保模型有效性所必需的。但这会使预处理变得复杂，因为我们必须提前运行语言相关的分词器。SentencePiece 就是一个可以满足快速从原始句子进行训练的模型。这对于训练中文和日文的分词器和分词解码器都很有用，因为单词之间不存在明确的空格。

## 空格被视为基本符号

自然语言处理的第一步是文本标记化。例如，标准的英语分词器会分割文本“Hello world.”。分为以下三个token。

| [Hello][World][.]

一种观察结果是原始输入和标记化序列之间是**不可逆的**。例如，“World”和“.”之间没有空格的信息。从标记化序列中删除，因为例如下面的情况：

```
Tokenize("World.") == Tokenize("World .")
```

SentencePiece 将输入文本视为 Unicode 字符序列。空格也作为普通符号处理。为了显式地将空格作为基本标记处理，SentencePiece 首先使用元符号“\_”（U+2581）对空格进行转义，如下所示。

| Hello\_\_World.

然后，将这段文本分割成小块，例如：

| [Hello] [\_\_Wor] [ld] [.]

由于空格保留在分段文本中，因此我们可以对文本进行解码，而不会产生任何歧义。

```
detokenized = ''.join(pieces).replace('_', ' ')
```

此功能可以在不依赖特定语言资源的情况下执行去标记化。请注意，在使用标准分词器拆分句子时，我们无法应用相同的无损转换，因为它们将空格视为特殊符号。标记化序列不保留恢复原始句子所需的信息。

- (en) Hello world. → [Hello] [World] [.] (A space between Hello and World)

- (ja) こんにちは世界。 → [こんにちは] [世界] [。] (No space between こんにちは and 世界)

### 子词正则化和 BPE-dropout

子词正则化和 BPE-dropout是简单的正则化方法，它们通过动态子词采样实际上增加了训练数据，这有助于提高 NMT 模型的准确性和鲁棒性。

#### byte-pair-encoding(BPE)

BPE首先把一个完整的句子分割为单个的字符，频率最高的相连字符对合并以后加入到词表中，直到达到目标词表大小。对测试句子采用相同的subword分割方式。BPE分割的优势是它可以较好的平衡词表大小和需要用于句子编码的token数量。

<https://arxiv.org/pdf/1710.02187.pdf>

为了启用于子词正则化，您希望将 SentencePiece 库 (C++/Python) 集成到 NMT 系统中，为每个参数更新采样一个分词，这与标准的离线数据准备不同。

#### Subword Regularization 使用多个子词候选改进神经网络翻译模型

1. 根据训练语料设置一个合理的seed词表。
2. 重复下列步骤直到达到目标词表的大小。
  - a. 修改词表，用EM算法来优化 $p(x)$
  - b. 对于每一个subword  $x_i$ ，计算 $loss_i$ (困惑度)。这里 $loss_i$ 表示当subword  $x_i$ 被移出当前词表的时候，似然函数减小的可能性。
  - c. 根据 $loss_i$  对符号进行排序，然后保留前a%的subword。

<https://arxiv.org/pdf/1804.10959.pdf>



在Python 库的示例中。会发现 'New York' 在每个 SampleEncode (C++) 上的分段方式不同，或者使用 enable\_sampling=True (Python) 进行编码。采样参数的细节可以在sentencepiece\_processor.h 中找到。

```
import sentencepiece as spm

s = spm.SentencePieceProcessor(model_file='spm.model')
for n in range(5):
    s.encode('New York', out_type=str, enable_sampling=True, alpha=0.1, nk
```

```
['_', 'N', 'e', 'w', '__York']
['_', 'New', '__York']
['_', 'New', '__Y', 'o', 'r', 'k']
['_', 'New', '__York']
['_', 'New', '__York']
```

## 安装

```
pip install sentencepiece
```

其它安装方法：

<https://github.com/google/sentencepiece/blob/master/python/README.md>

## 模型训练

```
import sentencepiece as spm

spm.SentencePieceTrainer.train(
    input=files, model_prefix='words_technology', vocab_size=16000)
```

方法参数：

- `input` : 每行一句的原始语料库文件。无需预处理。默认情况下，SentencePiece 使用 Unicode NFKC 规范化输入。还可以传递逗号分隔的文件列表。

- `model_prefix`: 输出的模型名前缀。算法最后会生成<model\_name>.model和<model\_name>.vocab两个文件。
- `vocab_size`: 词表大小，类似于8000,16000,32000等值
- `character_coverage`: 模型覆盖的字符数量，良好的默认值：对于具有丰富字符集的语言（如日语或中文）使用0.9995；对于其它小字符集的语言使用1.0。
- `model_type`: 模型算法类型。从 unigram（默认）、bpe、char、word 中进行选择。使用word类型时，输入句子必须预先标记。

更完整的参数列表请参阅下面的链接：

<https://github.com/google/sentencepiece/blob/master/doc/options.md>

## 分词

```
import sentencepiece as spm

sp = spm.SentencePieceProcessor(model_file=model_path)
out = sp.encode(
    '所谓“低代码”或“零代码”，指的是不编写或少编写代码，就能完成开发任务。这既有助于扩大用户规模，获得更大的市场，也有助于程序员减轻工作负荷，避免重复劳动。',
)
sp.id_to_piece(out)
```

```
['_', '所谓', '"', '低', '代码', '"', '或', '"', '零', '代码', '"',
 '指', '的是', '不', '编写', '或', '少', '编写', '代码', ',', '就能',
 '完成', '开发', '任务', '。', '这', '既', '有助于', '扩大', '用户规模',
 ',', '获得', '更大的', '市场', ',', '也', '有助于', '程序', '员',
 '减轻', '工作', '负', '荷', ',', '避免', '重复', '劳动', '。']
```

模型训练后还生成了一个 `<model_name>.vocab` 的文件，里面包含了指定大小的词典。

- 文字处理基本单元（token） 字符或词汇
- 不重复字符构建字典
  - 当模型推理过程中，遇到字典中没有包含token时，出现key index错误。Out Of Value（OOV问题）解决思路：通过特殊token，统一替代未知token
- 通过字典映射，文本转换token index

## 2. 转换值转变模型导入数据

- 数据还需要从文本到数值转换，使用Dataset封装上面处理逻辑

## 3. 模型结构搭建

- token\_index → embedding → rnn → linear → logits
- embedding shape:[batch, token\_len, features] 符合 RNN输入要求

## 文本结构化转换

分词之后通常会选择下一步做的就是构建词典(词汇表 vocabulary)。它包含语料中所有不重复的词汇的集合。

词典 (**vocabulary**) 本质上就是一个 `token ↔ index` 的一个dict。

```
class Vocabulary:

    def __init__(self, vocab):
        self.vocab = vocab

    @classmethod
    def from_documents(cls, documents):
        # 字典构建 (字符为token、词汇为token)
        no_repeat_tokens = set()
        for cmt in documents:
            no_repeat_tokens.update(list(cmt)) # token list
        # set转换为list, 第0个位置添加统一特殊token
        tokens = ['PAD', 'UNK'] + list(no_repeat_tokens)

        vocab = { tk:i for i, tk in enumerate(tokens)}

        return cls(vocab)
```

我们会通过数据集中所有词的列表来构建一个**不重复的词汇表对象**。在这个对象里，每个词汇都会被分配一个唯一的索引值**token\_index**。

## 关于Token

在NLP的语料处理分词环节，因语言不同，往往也会产生出不同的结果。以东亚语言为例(中、日、韩)：我们既可以把文章拆分为“词”，又可以把文章拆分为“字”。这种结果导致我们在描述分词时往往会让人产生歧义：到底是“词”还是“字”？

token可以解决这个问题(或者说描述)问题，不论我们拆分的是什么，每个被拆分出的内容，统一以**token**指代。

**词汇表中保存的是不重复的token，token\_index也就是给每个token分配的唯一索引值。**

代码中的 **PAD** 和 **UNK** 是在词表中添加的特殊符号。这里要解释一下NLP中的**OOV**概念。

## 关于OOV (Out Of Value)

OOV是指模型在测试或预测中遇到的新词。这些新词在模型已构建的词汇表中并不存在。因此，会导致模型方法无法识别并处理这些OOV新词。

例如我们的词表是基于标准的汉语词典词汇。当遇到类似“爷青回”，“口嗨”，“布吉岛”等新词时，词表里面没有这样的词，也就无法把新词映射为对应的索引。

遇到新词 OOV，模型就真的无能为力了么？其实方法还是有的，我们可以给未来的新词设置1个或多个**特殊符号**，这些token会插入在词表的顶端（索引为0的位置）。在

构建词表时，使用 `UNK` 来表示 *unknown*。当遇到词表中不存在的 token 时，可以使用 `UNK` 作为该 token 的默认索引。

这样，OOV 的问题就有了解决方案！

## 语料转换为训练样本

得到了词汇表，我们就把文本语料集中的每个样本，统一转换为 token\_index 的集合。

```
class CommentDataset:

    def __init__(self, comments, labels, vocab):
        self.comments = comments
        self.labels = labels

        self.vocab = vocab

    def __getitem__(self, index):
        # dict.get(key, default_value) 当key不存在, 返回default_value
        # token_index = [self.vocab.get(tk, self.vocab['UNK']) for tk in list(self.comments[index]) if tk != ' ']
        token_index = []
        for tk in list(self.comments[index]):
            if tk != ' ':
                tk_idx = self.vocab.get(tk, 0)
                token_index.append(tk_idx)

        # token_index生成固定长度tensor
        index_tensor = torch.zeros(size=(125,), dtype=torch.int)
        for i in range(len(token_index)):
            index_tensor[i] = token_index[i]

        return index_tensor, torch.tensor(self.labels[index]) # index索引对应记录

    def __len__(self):
        return len(self.labels) # 数据集包含元素数量
```

输出：

```
token_index length:106
token_index head:[565, 31, 117, 200, 258, 283, 17, 14, 35, 262]
token_index length:17
token_index head:[9, 112, 683, 126, 1630, 104, 921, 249, 22, 9]
token_index length:18
token_index head:[324, 131, 29, 468, 447, 279, 13, 133, 24, 10]
token_index length:387
token_index head:[17, 265, 261, 276, 49, 17, 74, 74, 20, 109]
token_index length:131
```

```
token_index head:[88, 174, 156, 287, 34, 571, 575, 24, 196, 534]
.....
```

观察可以发现，转换成功了！由于文本长度的不确定性，我们发现有的文本长度只有18，而有的文本长度达到了387。

这些token\_index可以直接导入模型进行训练么？不可以。因为样本的特征信息太少了。

通过词频矩阵转TF/IDF矩阵？可以，但不是很有效。尤其是**与时间序列相关的RNN模型，TF/IDF不能提供token之间的前后关联信息。**

NLP最常用的做法是：每个token映射一个对应的向量。把词汇表中所有token的向量组成矩阵，就称之为**词嵌入(词向量矩阵) Embedding**。

有了这个指导思路，下面要解决的问题有两个✌️：

1. 要解决不同长度文本的对齐。这样才可以在指定batch\_size后，批量的把数据导入模型进行训练。
2. 把token\_index映射到Embedding

先解决第1个问题，下面的小节解决第2个。

回忆下 `torch.utils.data.DataLoader` 类，它可以帮助我们封装 `Dataset`，给模型训练提供指定batch\_size大小的训练集。

`DataLoader` 类有个 `collate_fn` 的参数，可以指定一个自定义方法。在数据样本传递给模型训练前进行预处理。我们可以把自定义的方法赋值给 `collate_fn` 参数，在方法内部让每个批次的数据文本长度对齐。

```

from torch.nn.utils.rnn import pad_sequence

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 把word转为word_index
text_pipeline = lambda x: vocab(tokenizer(x))
# 把label自然索引转换为以0起始的索引
label_pipeline = lambda x: int(x) - 1


def collate_batch(batch):
    label_list, text_list = [], []
    for (_label, _text) in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
        text_list.append(processed_text)
    label_list = torch.tensor(label_list, dtype=torch.int64)
    text_list = pad_sequence(text_list, batch_first=True, padding_value=vocab['PAD'])
    return label_list.to(device), text_list.to(device)

# 定制DataLoader, 每batch_size的样本数据都会经collate_batch方法处理后再传给模型
dataloader = DataLoader(train, batch_size=8, shuffle=False, collate_fn=collate_batch)

```

collate\_batch方法的内部，是通过使用 `torch.nn.utils.rnn.pad_sequence` 方法来实现文本对齐的。这个方法可以统计 `text_list` 中最长的list子元素长度，然后把在其它list子元素末尾填充0，统一 `text_list` 中所有子元素的长度。

1	13	28	9		
6	11	22	3	9	5
11	9	4			

1	13	28	9	0	0
6	11	22	3	9	5
11	9	4	0	0	0

## 词向量Embedding

`torch.nn.Embedding` 类就是一个专门为词向量而设计的一个Model。它内置参数，可以和其它Model组合进行训练。

对于上面 `pad_sequence` 方法在填充的0值，Embedding也可以通过`padding_idx`参数来指定，这样对应的0值填充的也是一个全0向量且不参与梯度更新。

```
from torch.nn import Embedding
emb = Embedding(vocab_size, hidden_size, padding_idx=0)
```

那么为什么padding\_idx取值是0呢？因为它和vocab词表中的索引对应。还记得前面在构建词表时使用的 `["PAD","UNK"]` 么，我们通过调用词表的查询方法，就可以发现，这两个特殊的token恰好就在词表的第0和第1个索引位置上🤔

```
print(vocab['PAD'])
print(vocab['UNK'])
```

输出:

0

1

在RNN为主导的网络模型中，token所映射的Embedding也参与模型训练和优化。训练后得到的词向量类似于word2vec训练后的效果。

```
class SomeModel(nn.Module):
    def __init__(self, vocab_size, emb_hidden_size, rnn_hidden_size, num_layers, num_class):
        super(SomeModel, self).__init__()
        # Embedding模型
        self.embedding = nn.Embedding(vocab_size, emb_hidden_size, padding_idx=0)
        # LSTM模型
        self.rnn = nn.LSTM(
            input_size = emb_hidden_size,
            hidden_size = rnn_hidden_size,
            num_layers = num_layers,
            batch_first = True,
        )
        self.out = nn.Linear(rnn_hidden_size, num_class)

    def forward(self, x):
        out = self.embedding(x)
        r_out, (c_n, h_n) = self.rnn(out)
        out = self.out(r_out[:,-1,:])
        return out
```



通过DataLoader实例，就可以带入批次样本进行计算了

```
model = SomeModel(len(vocab), 128, 128, 1, 5)
model.to(device)
for label, text in dataloader:
    text.to(device)
    out = model(text)
    print(out)
    break
```

输出：

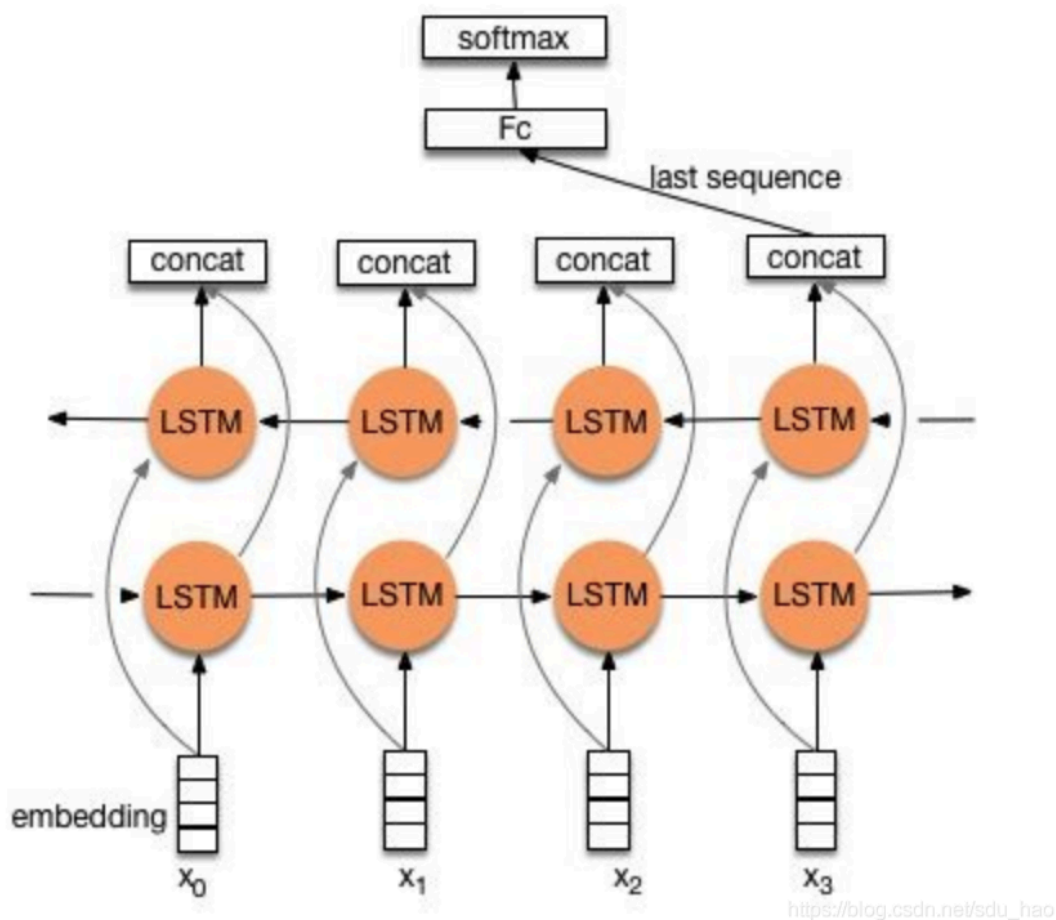
```
tensor([[ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201],
        [ -0.0423, -0.1396, -0.1168,  0.0249,  0.1375],
        [ -0.0289,  0.0494, -0.0421,  0.0109,  0.0201]], device='cuda:0',
grad_fn=<AddmmBackward>)
```

## TextRNN

TextRNN本质是：词向量+RNN+神经网络的文本分类预测模型。

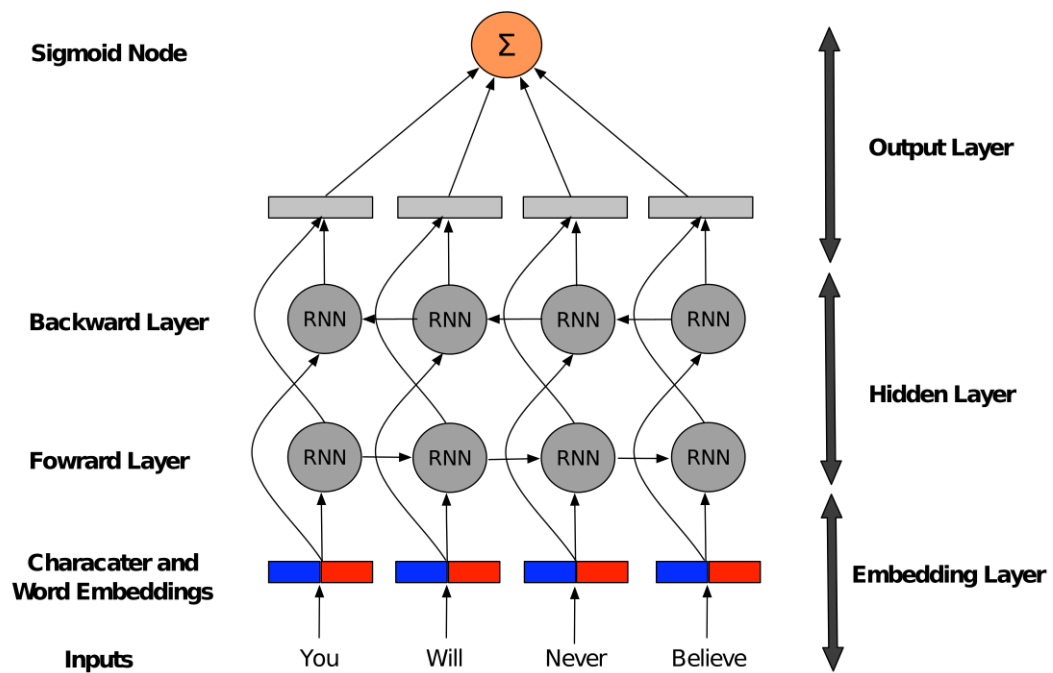
根据结构，可以分为两种：

- RNN最后一层的输出作为神经网络的输入，从而导入分类结果：



一般取前向/反向LSTM在最后一个时间步长上隐藏状态，然后进行拼接，在经过一个softmax层(输出层使用softmax激活函数)进行一个多分类。

- RNN所有的输出，拼接后进行分类预测



取前向/反向LSTM在每一个时间步长上的隐藏状态，对每一个时间步长上的两个隐藏状态进行拼接，然后对所有时间步长上拼接后的隐藏状态取均值，再经过一个softmax层(输出层使用softmax激活函数)进行一个多分类(2分类的话可以使用sigmoid激活函数)。

完整的模型代码参考：

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from tqdm import tqdm
import random
from torch.utils.tensorboard import SummaryWriter

from comment_classifier_model import CommentsClassifier
from comment_dataset import CommentDataset, Vocabulary

class SummaryWrapper:

    def __init__(self) -> None:
        self.writer = SummaryWriter()
        self.train_loss_cnt = 0

    def train_loss(self, func):
        def wrapper(*args):
            # 调用模型训练函数
            result = func(*args)
            self.writer.add_scalar('train_loss', result, self.train_loss_cnt)
            self.train_loss_cnt += 1
            return result
        return wrapper

sw = SummaryWrapper()

def train_test_split(X, y, split_rate=0.2):
    # 数据拆分流程
    # 1. 拆分比率
    # 2. 样本随机性
    # 3. 构建拆分索引
    # 4. 借助slice拆分
    split_size = int(len(X) * (1 - split_rate))

    split_index = list(range(len(X)))
    random.shuffle(split_index)
    x_train = [X[i] for i in split_index[:split_size]]
    y_train = [y[i] for i in split_index[:split_size]]

    x_test = [X[i] for i in split_index[split_size:]]
    y_test = [y[i] for i in split_index[split_size:]]

    return (x_train, y_train), (x_test, y_test)

def train(model, train_dl, criterion, optimizer):
    model.train()
    tpbar = tqdm(train_dl)
    for tokens, labels in tpbar:
        loss = train_step(model, tokens, labels, criterion)
        loss.backward()
        optimizer.step()
        model.zero_grad()
        tpbar.set_description(f'epoch:{epoch+1} train_loss:{loss.item():.4f}')

@sw.train_loss
def train_step(model, tokens, labels, criterion):
    tokens, labels = tokens.to(device), labels.to(device)
    logits = model(tokens)
    loss = criterion(logits, labels)
    return loss

if __name__ == '__main__':

    # tensorboard跟踪记录实现
    # 1. SummaryWriter全局对象
    # 2. 跟踪相关
    # title: train_loss, val_acc
    # value: loss, acc
    # indexer: train_cnt, acc_cnt

    # hyperparameter 超参数
    BATCH_SIZE=32
    EPOCHS=10
    EMBEDDING_SIZE=200
    RNN_HIDDEN_SIZE=100
    LEARN_RATE=1e-3
    NUM_LABELS=2

    device = torch.device('cuda' if torch.cuda.is_available() else ('mps' if torch.mps.is_available() else 'cpu'))

    # 数据准备
    import pickle
    with open('comments.bin', 'rb') as f:
        comments, labels = pickle.load(f)

    vocab = Vocabulary.from_documents(comments)

    # 数据拆分
    (x_train, y_train), (x_test, y_test) = train_test_split(comments, labels)

    # 自定义Dataset处理文本数据转换
    train_ds = CommentDataset(comments, y_train, vocab.vocab)
    train_dl = DataLoader(train_ds, batch_size=10, shuffle=True)

    # 模型构建
    model = CommentsClassifier(
        vocab_size=len(train_ds.vocab),
        emb_size=EMBEDDING_SIZE,
        rnn_hidden_size=RNN_HIDDEN_SIZE,
        num_labels=NUM_LABELS
    )
    model.to(device)

    # loss function、optimizer
    optimizer = optim.Adam(model.parameters(), lr=LEARN_RATE)
    criterion = nn.CrossEntropyLoss()

    # 训练
    for epoch in range(EPOCHS):
        train(model, train_dl, criterion, optimizer)
        # model.eval()

    torch.save(
        {'model_state': model.state_dict(),
         'model_vocab': vocab, 'model_objs.bin'})

```

