国科大

自然语言处理

文本分类系统说明文档

姓名：张笃振

学号：2019180014628019

2019 年 12 月 20 日

2019

UCAS

目录

1 小组成员信息及分工

表 1: 小组成员信息及分工

姓名	学号	分工	备注
张笃振	201918014628019	调研 DPCNN 模型并实现；编写文档；整合各个模型，组织项目结构，训练模型	组长
彭聪	2019E8020261037	调研 FastText 模型并实现；网页 Demo 的搭建	组员
曹萌	201928013229017	调研 TextCNN 模型并实现；调研 RCNN 模型并实现	组员
库鑫	201928013229055	调研 Attention 机制；调研并实现多层双向 LSTM with Attention 模型；	组员
冯春波	201928002829001	数据预处理；调研多层双向 LSTM 模型并实现	组员

2 任务定义

在过去的几十年中，文本分类问题在许多实际应用中得到了广泛的研究和解决。文本分类技术在信息检索、信息过滤、推荐系统、情感分析以及推荐系统等多个领域都有广泛的应用。文本分类任务是指根据已经定义好的类别标签对现有的一段文本进行标注的任务。一般来说，文本数据集包含一系列长短不一的文本片段，如 $D = \{X_1, X_2, ..., X_N\}$, 其中 X_i 代表一个数据点或文本片段，每个 X_i 包含 s 个句子，每个句子又包含 w_s 个单词，每个单词又可以包含 l_w 个字符。每个数据点 X_i 都用一组 k 个不同离散值索引中的类别标签进行标记。

3 方法描述及实现细节

我们的文本分类系统实现了一些基于深度学习的分类算法，主要是一些基于 CNN 和 RNN 的模型，包括 Fast-Text、TextCNN、DPCNN、RCNN、多层双向 LSTM/GRU 以及多层双向 LSTM/GRU with Attention 这六个模型。接下来，将逐一介绍上述模型的原理和实现细节。

3.1 FastText

1. 分类原理

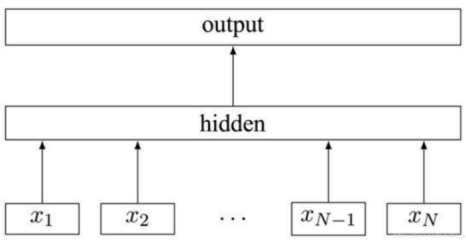

图 1: FastText 分类原理图示

输入样本是一系列整数索引 $(x_1, ..., x_N)$, 对应词典中相应的词，通过 embedding 得到每个词对应的词向量。对样本（文本）中每个词对应的词向量求平均，再通过一个全连接层进行分类即可。

2. 实现细节

```
class FastText(BasicModule): #继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.Module

    def __init__(self, vocab_size,opt): #opt是config类的实例 里面包括所有模型超参数的配置
        super(FastText, self).__init__()


        # 嵌入层
        self.embedding = nn.Embedding(vocab_size, opt.embed_size) #词嵌入矩阵 每一行代表词典中一个词对应的词向量;
        # 词嵌入矩阵可以随机初始化连同分类任务一起训练, 也可以用预训练词向量初始化 (冻结或微调)

        self.content_fc = nn.Sequential( #可以使用多个全连接层或batchnorm、dropout等 可以把这些模块用Sequential包装成一个大模块
        nn.Linear(opt.embed_size, opt.linear_hidden_size),
        nn.BatchNorm1d(opt.linear_hidden_size),
        nn.ReLU(inplace=True),
        #可以再加一个隐层
        # nn.Linear(opt.linear_hidden_size,opt.linear_hidden_size),
        # nn.BatchNorm1d(opt.linear_hidden_size),
        # nn.ReLU(inplace=True),
        #输出层
        nn.Linear(opt.linear_hidden_size, opt.classes)
        )


    def forward(self, inputs):
        #inputs(batch_size,seq_len)
        embeddings = self.embedding(inputs) # (batch_size, seq_len, embed_size)

        #对seq_len维取平均
        content = torch.mean(embeddings,dim=1) #(batch_size,1,embed_size)

        out = self.content_fc(content.squeeze(1)) #先压缩seq_len维 (batch_size,embed_size) 然后作为全连接层的输入
        #输出 (batch_size,classes)

        return out
```

3.2 TextCNN

1. 分类原理

TextCNN 可以从两个角度来解读，既可以把它看作但输入通道的 2 维卷积也可以把它看作多输入通道的 1 维卷积（其中词嵌入维度为通道维），二者其实是等价的。

如果把它看作一个单输入通道的 2 维卷积的话，它的分类流程就如图2所示。

1）把输入文本中的词转换为其对应的词向量，那么每个输入文本就可以表示为一个 n*d 的矩阵 (n 是输入文本包含的词数，d 为词向量的维数)。

2）对输入矩阵进行卷积操作。可以使用不同大小的卷积核，每种类型的卷积核可以有多个。假设卷积

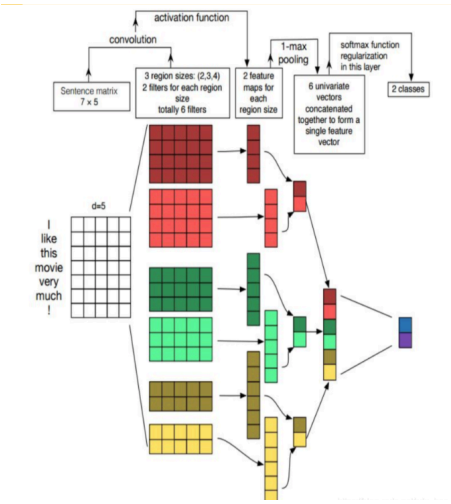

图 2: TextCNN 分类原理图示

核的大小是 (f,d),f 可以是不同的取值 (如 f=2，3，4), 而 d 是固定的，是词向量的维度，并且假设总共使用了 k 个卷积核，步长为 1。经过卷积操作后我们会得到 k 个向量，每个向量的长度是 n-f+1. 我们使用不同大小的卷积核，从输入文本中提取丰富的特征，这和 n-gram 特征有点相似 (f=2，3，4 分别对应于 2-gram，3-gram-4-gram)。

3）对卷积操作的输出进行全局 max-pooling 操作。作用于 k 个长度为 n-f+1 的向量上，每个向量整体取最大值，得到 k 个标量。

4）把 k 个标量拼接起来，组成一个向量表示最后提取的特征。他的长度是固定的，取决于我们所使用的不同大小的卷积核的总数（k）。

5)最后在接一个全联接层作为输出层,如果是 2 分类的话使用 sigmoid 激活函数,多分类则使用 softmax 激活函数，得到模型的输出。

2. 实现细节

```
#自定义时序（全局）最大池化层
class GlobalMaxPool1d(nn.Module):
def __init__(self):
super(GlobalMaxPool1d, self).__init__()
def forward(self, x):
# x (batch_size, channel, seq_len)
return F.max_pool1d(x, kernel_size=x.shape[2]) #  (batch_size, channel, 1)


# 多输入通道的一维卷积和单输入通道的2维卷积等价
# 这里按多输入通道的一维卷积来做 也可以用单输入通道的2维卷积来做
class TextCNN(BasicModule): #继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.Module

    def __init__(self, vocab_size, opt):#opt是config类的实例 里面包括所有模型超参数的配置

        super(TextCNN, self).__init__()


        # 嵌入层
```

```
        self.embedding = nn.Embedding(vocab_size,opt.embed_size)#词嵌入矩阵 每一行代表词典中一个词对应的词向量;
        # 词嵌入矩阵可以随机初始化连同分类任务一起训练，也可以用预训练词向量初始化（冻结或微调）

        # 创建多个一维卷积层
        self.convs = nn.ModuleList()
        for c, k in zip(opt.num_channels, opt.kernel_sizes): #num_channels定义了每种卷积核的个数 kernel_sizes定义了每种卷积核的大小
        self.convs.append(nn.Conv1d(in_channels=opt.embed_size,
        out_channels=c,
        kernel_size=k))
        #定义dropout层
        self.dropout = nn.Dropout(opt.drop_prop)
        #定义输出层
        self.fc = nn.Linear(sum(opt.num_channels), opt.classes)
        # 时序最大池化层没有权重，所以可以共用一个实例
        self.pool = GlobalMaxPool1d()


    def forward(self, inputs):
        # inputs(batch_size,seq_len)
        embeddings = self.embedding(inputs) # (batch_size, seq_len, embed_size)

        # 根据conv1d的输入要求 把通道维提前(这里的通道维是词向量维度)
        # (batch_size,channel/embed_size,seq_len)
        embeddings = embeddings.permute(0, 2, 1)
        # 对于每个一维卷积层，会得到一个(batch_size,num_channel(卷积核的个数),seq_len-kernel_size+1) 大小的tensor
        # 在时序最大池化后会得到一个形状为(batch_size, num_channel, 1)的 tensor
        # 使用squeeze去掉最后一维 并在通道维上连结 得到(batch_size,sum(num_channels))大小的tensor
        encoding = torch.cat([self.pool(F.relu(conv(embeddings))).squeeze(-1) for conv in self.convs], dim=1)

        # 应用丢弃法后使用全连接层得到输出 (batch_size,classes)
        outputs = self.fc(self.dropout(encoding))
        return outputs
```

3.3 RCNN

1. 分类原理

图3中中间是输入文本中每个单词的嵌入表示，左右使用双向 RNN 分别学习当前词 w_i 的左上下文表示 $cl(w_i)$ 和右上下文表示 $cr(w_i)$, 与当前词 w_i 本身的词向量连接，构成后续卷积层的输入 x_i。具体如下：

$$cl(w_i) = f(W^{(l)}cl(w_{i-1}) + W^{(sl)}e(w_{i-1})) \tag{1}$$

$$cl(w_i) = f(W^{(r)}cr(w_{i-1}) + W^{(sr)}e(w_{i-1})) \tag{2}$$

$$x_i = [cl(w_i); e(w_i); cr(w_i)] \tag{3}$$

与 TextCNN 比较类似，都是把文本表示为一个嵌入矩阵，再进行卷积操作。不同的是 TextCNN 中的文本嵌入矩阵每一行只是文本中一个词的向量表示，而在 RCNN 中，文本嵌入矩阵的每一行是当前词的词向量以及上下文嵌入表示的拼接。

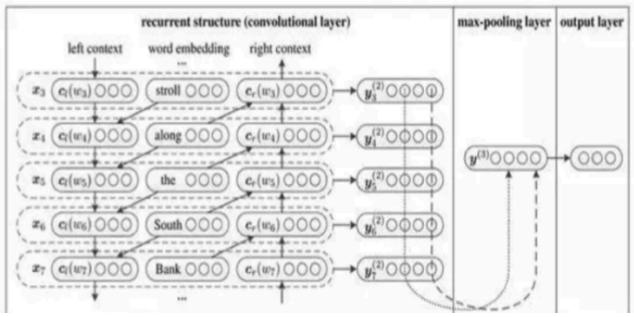

图 3: RCNN 分类原理图示

例如：图3中的 along 这一个单词，中间是他的词向量，左边利用一个 RNN 得到 along 之前上文所有单词的嵌入表示，右边同样利用一个反向的 RNN 得到 along 之后所有单词的嵌入表示。三者共同构成 along 的嵌入表示，文本中的其他词同理。最后文本被表示为上图左边的一个嵌入矩阵。

然后 x_i 将作为当前词 w_i 的嵌入表示，输入到卷积核大小为 1*d(d 为 x_i 的长度) 激活函数为 tanh 的卷积层，得到 w_i 潜在的语义向量：

$$y_i^{(2)} = tanh(W^{(2)}x_i + b^{(2)}) \tag{4}$$

在 TextCNN 中我们曾经设置了多个卷积核 f*d,RCNN 中将卷积核大小设置为 1*d 的原因是中已经包含了左右上下文的信息，无需再使用窗口大于 1 的卷积核进行特征提取。需要说明的是，实践中依然可以同时使用多个不同大小的卷积核，如 [1,2,3], 可能会取得更好的实践效果，一种解释是窗口大于 1 的卷积核强化了左右最近的上下文信息。此外实践中一般使用更复杂的 RNN 来捕捉的上下文信息，如 LSTM 和 GRU 等。

在经过卷积层以后，获得了文本中所有词的语义表示 $y_i^{(2)}$, 然后经过一个 max-pooling 层和 softmax 层 (输出层使用 softmax 激活函数) 进行分类：

$$y^{(3)} = \max_{i=1}^{n} y_i^{(2)} \tag{5}$$

$$y^{(4)} = W^{(4)}y^{(3)} + b^{(4)} \tag{6}$$

$$p_i = \frac{exp(y_i^{(4)})}{\sum_{k=1}^{n} exp(y_k^{(4)})} \tag{7}$$

2. 实现细节

```
class RCNN(BasicModule):#继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.Module

def __init__(self,vocab_size,opt):#opt是config类的实例 里面包括所有模型超参数的配置

    super(RCNN, self).__init__()
    # 嵌入层
    self.embedding = nn.Embedding(vocab_size, opt.embed_size)#词嵌入矩阵 每一行代表词典中一个词对应
        的词向量;
    # 词嵌入矩阵可以随机初始化连同分类任务一起训练，也可以用预训练词向量初始化（冻结或微调）
```

```
9
10          #双向lstm 由于RCNN中双向lstm一般只有一层 所以opt.drop_prop_rcnn=0.0(丢弃率)
11          self.lstm = nn.LSTM(opt.embed_size,opt.recurrent_hidden_size,num_layers=opt.num_layers_rcnn,
12          bidirectional=True,batch_first=True,dropout=opt.drop_prop_rcnn)
13
14          #全连接层 维度转换 卷积操作可以用全连接层代替
15          self.linear = nn.Linear(2*opt.recurrent_hidden_size+opt.embed_size,opt.recurrent_hidden_size)
16
17          #池化层
18          self.max_pool = nn.MaxPool1d(opt.max_len)
19
20          #全连接层分类
21          self.content_fc = nn.Sequential(
22          nn.Linear(opt.recurrent_hidden_size, opt.linear_hidden_size),
23          nn.BatchNorm1d(opt.linear_hidden_size),
24          nn.ReLU(inplace=True),
25          nn.Dropout(opt.drop_prop),
26          # 可以再加一个隐层
27          # nn.Linear(opt.linear_hidden_size,opt.linear_hidden_size),
28          # nn.BatchNorm1d(opt.linear_hidden_size),
29          # nn.ReLU(inplace=True),
30          # 输出层
31          nn.Linear(opt.linear_hidden_size, opt.classes)
32          )
33
34      def forward(self, inputs):
35          #inputs(batch_size,seq_len)
36          # 由于batch_first = True 所以inputs不用转换维度
37          embeddings = self.embedding(inputs)  # (batch_size, seq_len, embed_size)
38
39          outputs,_ = self.lstm(embeddings) #(batch_size, seq_len, recurrent_hidden_size*2)
40
41          #将前后向隐藏状态和embedding拼接
42          outputs = torch.cat((outputs[:,:,:outputs.size(2)//2],embeddings,outputs[:,:,outputs.size(2)
                //2:]),dim=2) #(batch_size, seq_len, embed_size+recurrent_hidden_size*2)
43
44          #做维度转换
45          outputs = self.linear(outputs) #(batch_size, seq_len, recurrent_hidden_size)
46
47          #沿seq_len维做最大池化(全局池化)
48          #先调整维度 交换recurrent_hidden_size维和seq_len维
49          #即把recurrent_hidden_size作为通道维 符合一维池化的输入
50          outputs = outputs.permute(0,2,1)  #(batch_size,recurrent_hidden_size,seq_len)
51          outputs = self.max_pool(outputs).squeeze(2)   #(batch_size,recurrent_hidden_size)
52
53          #通过全连接层 分类
54          outputs = self.content_fc(outputs) #(batch_size,classes)
55
56          return outputs
```

3.4 多层双向 LSTM

1. 分类原理

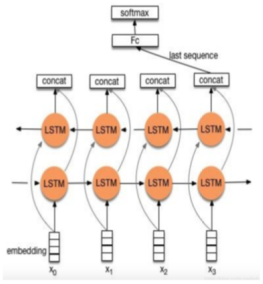

图 4: 多层双向 LSTM 分类原理图示

对于输入文本序列，在 LSTM 的每个时间步输入序列中一个单词的嵌入表示，计算当前时间步的隐藏状态，用于当前时间步的输出以及传递给下一个时间步和下一个单词的词向量一起作为 LSTM 单元输入，然后再计算下一个时间步的 LSTM 隐藏状态，以此重复... 直到处理完输入文本序列中的每一个单词，如果输入文本序列的长度为 n，那么就要经历 n 个时间步。

一般取前向和反向 LSTM 在最后一个时间步的隐藏状态，进行拼接，再接一个全连接层进行分类。由于 LSTM 训练比较困难，层数一般不超过两层。

2. 实现细节

```
class MulBiLSTM(BasicModule):#继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.Module
    def __init__(self, vocab_size,opt):#opt是config类的实例 里面包括所有模型超参数的配置
        super(MulBiLSTM, self).__init__()

        #嵌入层
        self.embedding = nn.Embedding(vocab_size, opt.embed_size)#词嵌入矩阵 每一行代表词典中一个词对应的词向量;
        # 词嵌入矩阵可以随机初始化连同分类任务一起训练，也可以用预训练词向量初始化（冻结或微调）

        # bidirectional设为True即得到双向循环神经网络
        self.encoder = nn.LSTM(input_size=opt.embed_size,
        hidden_size=opt.recurrent_hidden_size,
        num_layers=opt.num_layers,
        bidirectional=True,
        dropout=opt.drop_prop
        )
        self.fc = nn.Linear(4 * opt.recurrent_hidden_size, opt.classes)  # 初始时间步和最终时间步的隐藏状态作为全连接层输入

    def forward(self, inputs):
        # inputs的形状是(批量大小，词数)，因为上述定义的LSTM没有设置参数batch_first=True(默认False),所以需要将序列长度(seq_len)作为第一维，所以将输入转置后再提取词特征
        embeddings = self.embedding(inputs.permute(1,0)) # (seq_len, batch_size,embed_size)

        # rnn.LSTM只传入输入embeddings（第一层的输入），因此只返回最后一层的隐藏层在各时间步的隐藏状态。
        # outputs形状是(seq_len, batch_size, 2 * recurrent_hidden_size)
        outputs, _ = self.encoder(embeddings)  # output, (h, c)
        # 连结初始时间步和最终时间步的隐藏状态作为全连接层输入。它的形状为
        # (batch_size, 4 * recurrent_hidden_size)。
        encoding = torch.cat((outputs[0], outputs[-1]), -1)
        outs = self.fc(encoding)
```

```
29          #(batch_size, classes)
30          return outs
```

3.5 多层双向 LSTM with Attention

1. 分类原理

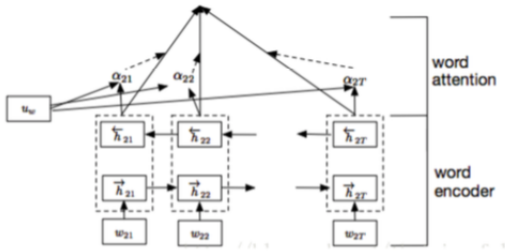

图 5: 多层双向 LSTM with Attention 分类原理图示

多层双向 LSTM with Attention 对多层双向 LSTM 模型做了一些改进，不再单纯地只利用最后时刻的隐藏状态（两个方向最后时刻隐藏状态拼接）进行分类，而是考虑每个时间步的隐藏状态，对每个时间步的隐藏状态（两个方向拼接）进行加权求和，然后对加权求和结果进行分类。其分类流程如图5所示。

Attention 机制的原理：

$$v_t = tanh(W_w h_t + b_w) \tag{8}$$

$$\alpha_t = \frac{exp(u_t^T u_w)}{\sum_t (exp(u_t^T u_w))} \tag{9}$$

$$s = \sum_t \alpha_t h_t \tag{10}$$

其中，W_w, b_w, u_w 都是需要自己设置的权重参数 (随模型训练)，u_w 可以认为是 Attention 机制中的 Query,u_t 可以认为是 Attention 机制中的 Key，h_t 是第 t 个时间步的隐藏状态 (两个方向拼接)，同时也是 Attention 机制中的 Value，α_t 为每个时间步隐藏状态对应的权重，s 是加权求和后得到的特征向量，用于最后接全连接层进行分类。

2. 实现细节

```
1   class MulBiLSTM_Atten(BasicModule):#继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.
        Module
2
3       def __init__(self,vocab_size,opt):#opt是config类的实例 里面包括所有模型超参数的配置
4
5           super(MulBiLSTM_Atten, self).__init__()
6           # 嵌入层
7           self.embedding = nn.Embedding(vocab_size, opt.embed_size)#词嵌入矩阵 每一行代表词典中一个词对应
                的词向量;
8           # 词嵌入矩阵可以随机初始化连同分类任务一起训练，也可以用预训练词向量初始化（冻结或微调）
9
10          #多层双向LSTM 默认seq_len作为第一维 也可以通过batch_first=True 设置batch_size 为第一维
```

```
        self.lstm = nn.LSTM(opt.embed_size,opt.recurrent_hidden_size, opt.num_layers,
        bidirectional=True, batch_first=True, dropout=opt.drop_prop)

        self.tanh1 = nn.Tanh()
        self.u = nn.Parameter(torch.Tensor(opt.recurrent_hidden_size * 2, opt.recurrent_hidden_size * 2)
            )
        #定义一个参数（变量） 作为Attention的Query
        self.w = nn.Parameter(torch.Tensor(opt.recurrent_hidden_size*2))

        #均匀分布 初始化
        nn.init.uniform_(self.w, -0.1, 0.1)
        nn.init.uniform_(self.u, -0.1, 0.1)

        #正态分布 初始化
        #nn.init.normal_(self.w, mean=0, std=0.01)
        self.tanh2 = nn.Tanh()
        #最后的全连接层
        self.content_fc = nn.Sequential(
        nn.Linear(opt.recurrent_hidden_size*2, opt.linear_hidden_size),
        nn.BatchNorm1d(opt.linear_hidden_size),
        nn.ReLU(inplace=True),
        nn.Dropout(opt.drop_prop),
        # 可以再加一个隐层
        # nn.Linear(opt.linear_hidden_size,opt.linear_hidden_size),
        # nn.BatchNorm1d(opt.linear_hidden_size),
        # nn.ReLU(inplace=True),
        # 输出层
        nn.Linear(opt.linear_hidden_size, opt.classes)
        )

    def forward(self, inputs):
        #由于batch_first = True 所有inputs不用转换维度
        embeddings = self.embedding(inputs)  # (batch_size, seq_len, embed_size)

        outputs,_ = self.lstm(embeddings)   #(batch_size,seq_len,recurrent_hidden_size*2)

        #M = self.tanh1(outputs) #(batch_size,seq_len,recurrent_hidden_size*2)
        M = torch.tanh(torch.matmul(outputs, self.u)) #也可以先做一个线性变换 再通过激活函数 作为Key

        #M (batch_size,seq_len,recurrent_hidden_size*2) self.w (recurrent_hidden_size*2,)
        #torch.matmul(M, self.w) (batch_size,seq_len) w作为Query与各个隐藏状态（Key） 做内积
        #再对第一维seq_len做softmax 转换为概率分布 (batch_size,seq_len) 得到权重
        alpha = F.softmax(torch.matmul(M, self.w), dim=1).unsqueeze(-1)  # (batch_size,seq_len,1)

        #对各个隐藏状态和权重 对应相乘
        out = alpha * outputs #(batch_size,seq_len,recurrent_hidden_size*2)

        #对乘积求和 out为加权求和得到的特征向量
        out = torch.sum(out,dim=1) #(batch_size,recurrent_hidden_size*2)

        #out = F.relu(out)

        out = self.content_fc(out)  #(batch_size,classes)

        return out
```

3.6 DPCNN

1. 分类原理

ACL2017 年中，腾讯 AI-lab 提出了 Deep Pyramid Convolutional Neural Networks for Text Categorization(DPCNN)。论文中提出了一种基于 word-level 级别的网络-DPCNN，由于之前介绍的 TextCNN 不能通过卷积获得文本的长距离依赖关系，而论文中 DPCNN 通过不断加深网络，可以抽取长距离的文本依赖关系。实验证明在不增加太多计算成本的情况下，增加网络深度就可以获得最佳的准确率。

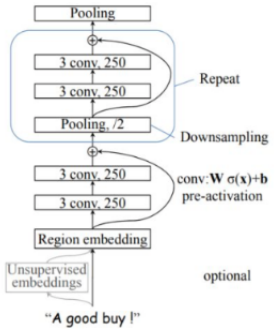

图 6: DPCNN 分类原理图示

1）等长卷积

首先交代一下卷积的的一个基本概念。一般常用的卷积有以下三类：

假设输入的序列长度为 n，卷积核大小为 m，步长 (stride) 为 s, 输入序列两端各填补 p 个零 (zero padding), 那么该卷积层的输出序列为 (n-m+2p)/s+1。

a. 窄卷积：步长 s=1, 两端不补零，即 p=0，卷积后输出长度为 n-m+1。

b. 宽卷积：步长 s=1, 两端补零 p=m-1，卷积后输出长度 n+m-1。

c. 等长卷积: 步长 s=1, 两端补零 p=(m-1)/2，卷积后输出长度为 n。（不改变序列长度）

2）池化

那么 DPCNN 是如何捕捉长距离依赖的呢？——Downsampling with the number of feature maps fixed。

作者选择了适当的两层等长卷积来提高词位 embedding 的表示的丰富性。然后接下来就开始 Downsampling (池化)。再每一个卷积块 (两层的等长卷积) 后，使用一个 size=3 和 stride=2 进行 maxpooling 进行池化。序列的长度就被压缩成了原来的一半。其能够感知到的文本片段就比之前长了一倍。

DPCNN 中重复使用多个这种模块，每个模块包含两层等长卷积（不改变序列长度）和一个池化下采样（序列长度减半）。

例如之前是只能感知 3 个词位长度的信息，经过 1/2 池化层后就能感知 6 个词位长度的信息啦，这时把 1/2 池化层和 size=3 的卷积层组合起来如图7所示。

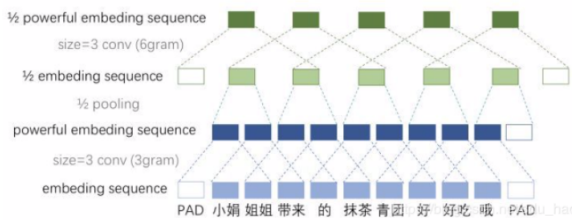

图 7: 感受野

3）固定 feature map（filters/卷积核）的数量

为什么要固定 feature maps 的数量呢? 许多模型每当执行池化操作时，增加 feature maps 的数量（即序列减半，输出通道数加倍），导致总计算复杂度是深度的函数。与此相反，作者对 feature map 的数量进行了修正，他们实验发现增加 feature map 的数量只会大大增加计算时间，而没有提高精度。

固定了 feature map 的数量，每当使用一个 size=3 和 stride=2 进行 max pooling 进行池化时，每个卷积层的计算时间减半 (数据大小减半/序列长度减半)，从而形成一个金字塔。

剩下的我们就只需要重复的进行等长卷积 + 等长卷积 + 使用一个 size=3 和 stride=2 进行 maxpooling 进行池化就可以啦，DPCNN 就可以捕捉文本的长距离依赖啦！还有一个问题，网络加深时，需要解决梯度消失问题。

4）shortcut connections with pre-activation

a. 初始化 CNN 的时，往往各层权重都初始化为很小的值，这导致了最开始的网络中，后续几乎每层的输入都是接近 0，这时的网络输出没有意义；

b. 小权重阻碍了梯度的传播，使得网络的初始训练阶段往往要迭代好久才能启动；

c. 就算网络启动完成，由于深度网络中仿射矩阵（每两层间的连接边）近似连乘，训练过程中网络也非常容易发生梯度爆炸或弥散问题。

当然，上述这几点问题本质就是梯度弥散问题。那么如何解决深度 CNN 网络的梯度弥散问题呢？ ResNet 中提出的 shortcut-connection/ skip-connection/ residual-connection（残差连接）就是一种非常简单、合理、有效的解决方案。

类似地，为了使深度网络的训练成为可能，作者为了恒等映射，所以使用加法进行 shortcut connections，即 z+f(z), 其中 f 用的是两层的等长卷积 (z 和 f(z) 的形状相同，可以直接相加)。这样就可以极大的缓解了梯度消失问题。

另外，作者也使用了 pre-activation。直观上，这种“线性”简化了深度网络的训练，类似于 LSTM 中 constant error carousels 的作用。而且实验证明 pre-activation 优于 post-activation（即把激活函数放在卷积操作前面，卷积层的顺序为 BatchNorm-ReLU-Conv）。

5）region embedding

同时 DPCNN 的底层保持了跟 TextCNN 一样的结构，这里作者将 TextCNN 的包含多尺寸卷积滤波器的卷积层的卷积结果称之为 Region embedding，意思就是对一个文本区域/片段（比如 3gram，卷积核大小为 3）进行一组卷积操作后生成的 embedding。

2. 实现细节

```
class ResnetBlock(nn.Module):
    def __init__(self, channel_size):
        super(ResnetBlock, self).__init__()
```

```
4
5           self.channel_size = channel_size
6           #将序列长度减半
7           self.maxpool = nn.Sequential(
8           nn.ConstantPad1d(padding=(0, 1), value=0), #在每个通道上(一维) 一边填充0个0(不填充) 另一边填1个0
9           nn.MaxPool1d(kernel_size=3, stride=2) #序列长度减半 height = (height-kernel_size+padding+stride)
                //stride=height // 2
10          )
11          self.conv = nn.Sequential(  #等长卷积 不改变height
12          nn.BatchNorm1d(num_features=self.channel_size),
13          nn.ReLU(),
14          nn.Conv1d(self.channel_size, self.channel_size, kernel_size=3, padding=1),
15
16          nn.BatchNorm1d(num_features=self.channel_size),
17          nn.ReLU(),
18          nn.Conv1d(self.channel_size, self.channel_size, kernel_size=3, padding=1),
19          )
20
21      def forward(self, x): #(batch_size,channel_size,seq_len)
22          x_shortcut = self.maxpool(x) # (batch_size,channel_size,seq_len//2)
23          x = self.conv(x_shortcut)#(batch_size,channel_size,seq_len//2)
24          x = x + x_shortcut#(batch_size,channel_size,seq_len//2) shortcut 残差连结
25
26      return x
27
28  class DPCNN(BasicModule):#继承自BasicModule 其中封装了保存加载模型的接口,BasicModule继承自nn.Module
29
30      def __init__(self,vocab_size,opt): #opt是config类的实例 里面包括所有模型超参数的配置
31
32          super(DPCNN, self).__init__()
33          # 嵌入层
34          self.embedding = nn.Embedding(vocab_size, opt.embed_size)#词嵌入矩阵 每一行代表词典中一个词对应
                的词向量;
35          # 词嵌入矩阵可以随机初始化连同分类任务一起训练, 也可以用预训练词向量初始化 (冻结或微调)
36
37          # region embedding
38          self.region_embedding = nn.Sequential(
39          nn.Conv1d(opt.embed_size, opt.channel_size, kernel_size=3, padding=1), #same卷积 不改变height/序
                列长度
40          nn.BatchNorm1d(num_features=opt.channel_size),
41          nn.ReLU(),
42          nn.Dropout(opt.drop_prop_dpcnn)
43          )
44
45          #卷积块 same卷积 不改变height
46          self.conv_block = nn.Sequential(
47          nn.BatchNorm1d(num_features=opt.channel_size),
48          nn.ReLU(),
49          nn.Conv1d(opt.channel_size, opt.channel_size, kernel_size=3, padding=1),
50          nn.BatchNorm1d(num_features=opt.channel_size),
51          nn.ReLU(),
52          nn.Conv1d(opt.channel_size, opt.channel_size, kernel_size=3, padding=1),
53          )
54
55          self.seq_len = opt.max_len #序列最大长度
```

```
        resnet_block_list = []  #存储多个残差块

        while (self.seq_len > 2): #每经过一个残差块 序列长度减半 只要长度>2 就不停地加残差块
        resnet_block_list.append(ResnetBlock(opt.channel_size))
        self.seq_len = self.seq_len // 2

        #将残差块 构成残差层 作为一个子模块
        self.resnet_layer = nn.Sequential(*resnet_block_list)

        #输出层 分类
        self.linear_out = nn.Linear(self.seq_len * opt.channel_size, opt.classes)

    def forward(self, inputs):

        embeddings = self.embedding(inputs) #(batch_size,max_len,embed_size)

        x = embeddings.permute(0, 2, 1) #(batch_size,embed_size,max_len) 交换维度 作为1维卷积的输入
            embed_size 作为通道维
        x = self.region_embedding(x) #(batch_size,channel_size,max_len)

        x = self.conv_block(x) #(batch_size,channel_size,max_len)

        x = self.resnet_layer(x) #经过多个残差块 每次长度减半 (batch_size,channel_size,self.seq_len)

        x = x.permute(0, 2, 1)  #(batch_size,self.seq_len,channel_size)

        x = x.contiguous().view(x.size(0), -1) #(batch_size,self.seq_len*channel_size) 拉伸为向量
        out = self.linear_out(x)      #(batch_size,classes)
        return out
```

4 数据集及预处理

4.1 数据集简介

THUCNews 是根据新浪新闻 RSS 订阅频道 2005 2011 年间的历史数据筛选过滤生成，包含 74 万篇新闻文档（2.19 GB），均为 UTF-8 纯文本格式。我们在原始新浪新闻分类体系的基础上，重新整合划分出 14 个候选分类类别：财经、彩票、房产、股票、家居、教育、科技、社会、时尚、时政、体育、星座、游戏、娱乐。(完整数据集压缩包下载)

4.2 数据预处理

在进行特征提取之前，需要对原始文本数据进行预处理，这对于特征提取来说至关重要，一个好的预处理过程会显著的提高特征提取的质量以及分类算法的性能。文本预处理一般包括以下步骤:

（1）分词: 首先，需要把文本切分成单词或短语。对于英文文本，可以直接按照空格进行切分（此时句末的标点不会单独切分出来）或使用一些英文分词工具如 nltk 中的分词工具; 对于中文文本，可以使用分词工具（如 jieba 等）进行切分。（文本分类算法有基于词和基于字符两种处理方式，一般来说基于词的文本分类算法效果更好，本专栏介绍的文本分类算法都是基于词的处理方式）。

(2)去停止词: 所谓停止词,就是在文本中大量出现但对分类并没有太多作用的词。如英文里的'a','an','the','above','afte
中文里的' 的',' 这',' 那',... 在这一步，把文本中的停止词过滤掉。(英文文本去停止词，可以直接使用 nltk
中封装的函数，中文文本可以自行下载停用词表（我们使用的是哈工大停用词表），来进行过滤)。

（3）小写化：这一步主要针对英文文本，大多数情况下需要把英文文本统一转换为小写形式。

（4）噪声移除：去除文本中的特殊符号，如特殊标点等。这些特殊符号对于人类理解文本可能很重要，但对于分类算法并没有太大意义（可以和停止词一并去除)。

（5）拼写检查： 数据集可能来自一些社交媒体，存在一些拼写错误，可以尝试对拼写问题进行纠正。

（6）俚语和缩写：可以尝试把缩写还原为完整形式或将一些较口语化的表示转换为书面语。

（7）词干提取和词型还原: 这一步主要针对英文文本，即将单词转换为最基本的形式。如 studying → study、apples → apple 等。

（8）词频统计与过滤: 对文本训练集进行上述处理后，统计剩余单词的词频，并过滤低频词（可以设置一个阈值（如，5)，保留词频大于阈值的单词；或者基于词频进行排序，取前 k 个词。前者词典的大小不确定，后者词典大小是确定的，就是 k（或 k+2 包括位置符号 unk 和填充符号 pad))。对剩余单词，构建词典。

4.3 实现细节

1. 读取数据

```
def read_cnews():
    #opt.data_root为数据集解压后 文件夹所在路径  定义在config.py中
    data = []
    #所有的主题标签
    labels = [label for label in os.listdir(opt.data_root) if label != '.DS_Store']
    print(labels)
    #标签到整数索引的映射
    labels2index = dict(zip(labels, list(range(len(labels)))))
    #整数索引到标签的映射
    index2labels = dict(zip(list(range(len(labels))),labels))
    print(labels2index)
    print(index2labels)

    #存储整数索引到标签的映射 以便预测时使用
    with open('index2labels.json','w') as f:
        json.dump(index2labels,f)
    #存储类别标签 打印分类报告会用到
    with open('labels.json','w') as f:
        json.dump(labels,f)


    for label in labels:
        folder_name = os.path.join(opt.data_root,label)
        datasub = []    #存储某一类的数据 [[string,index],...]
        for file in tqdm(os.listdir(folder_name)):
            with open(os.path.join(folder_name, file), 'rb') as f:
                #去除文本中空白符
                review = f.read().decode('utf-8').replace('\n', '').replace('\r','').replace('\t','')
                datasub.append([review,labels2index[label]])
        data.append(datasub) #存储所有类的数据[[[string,index],...],[[string,index],...],...]
```

```
31
32      return data
```

2. 切分训练集、验证集和测试集

```
1  def split_data(data):
2      #切分数据集 为训练集、验证集和测试集
3      train_data = []
4      val_data = []
5      test_data = []
6
7      #对每一类数据进行打乱
8      #设置验证集和测试集中每一类样本数都为200（样本均衡）
9      for data1 in data: #遍历每一类数据
10         np.random.shuffle(data1) #打乱
11         val_data += data1[:200]
12         test_data += data1[200:400]
13         train_data += data1[400:]
14
15     np.random.shuffle(train_data) #打乱训练集 测试机和验证集不用打乱
16
17     print(len(train_data))
18     print(len(val_data))
19     print(len(test_data))
20
21     return train_data,val_data,test_data
```

3. 基于训练集构建词典

```
1  #读取停用词
2  def stopwords(fileroot):
3      #fileroot为下载的停用词表所在的路径
4      with open(fileroot,'r') as f:
5          stopword = [line.strip() for line in f]
6      print(stopword[:5])
7      return stopword
8
9  #分词 去除停用词
10 def get_tokenized(data,stopword):
11     """
12     data: list of [string, label]
13     """
14     def tokenizer(text):
15         return [tok for tok in jieba.lcut(text) if tok not in stopword]
16     return [tokenizer(review) for review, _ in data]
17
18 #构建词典
19 def get_vocab(data,stopword):
20     tokenized_data = get_tokenized(data,stopword) #分词、去除停用词
21     counter = collections.Counter([tk for st in tokenized_data for tk in st]) #统计词频
22     return Vocab.Vocab(counter, min_freq=5,specials=['<pad>','<unk>']) #保留词频大于5的词 <pad>对应填充
           项（词典中第0个词） <unk>对应低频词和停止词等未知词（词典中第1个词）
```

4. 基于词典对训练集、验证集以及测试集进行处理

```
1  def preprocess_imdb(data, vocab,stopword):
2      #将训练集、验证集、测试集中单词转换为词典中对应的索引
```

```
    max_l = 500  # 将每条新闻通过截断或者补0，使得长度变成500（所有数据统一成一个长度，方便用矩阵并行计算（其实也可以每个 batch填充成一个长度，batch间可以不一样））

    def pad(x):
        return x[:max_l] if len(x) > max_l else x + [0] * (max_l - len(x))

    tokenized_data = get_tokenized(data,stopword) #分词、去停止词
    features = torch.tensor([pad([vocab.stoi[word] for word in words]) for words in tokenized_data]) #把单词转换为词典中对应的索引，并填充成固定长度，封装为tensor
    labels = torch.tensor([score for _, score in data])
    return features, labels
```

5. 保存一些中间结果

```
with open('word2index.json','w') as f: #保存词到索引的映射，预测和后续加载预训练词向量时会用到
    json.dump(vocab.stoi,f)

with open('vocabsize.json','w') as f:
#保存词典的大小（因为我们基于词频阈值过滤低频词，词典大小不确定，需要保存，后续模型中会用到）
    json.dump(len(vocab),f)

#保存预处理好的训练集、验证集和测试集 以便后续训练时使用
torch.save(X_train,'X_train.pt')
torch.save(y_train, 'y_train.pt')
torch.save(X_val, 'X_val.pt')
torch.save(y_val, 'y_val.pt')
torch.save(X_test, 'X_test.pt')
torch.save(y_test, 'y_test.pt')
```

5 项目组织结构

在学习某个深度学习框架时，掌握其基本知识和接口固然重要，但如何合理组织代码，使得代码具有良好的可读性和可扩展性也必不可少。做深度学习实验或项目时，为了得到最优的模型结果，中间往往需要很多次的尝试和修改（也就是所谓地调参）。从事大多数深度学习研究时，程序都需要实现以下几个功能：

1）模型定义

2）数据处理和加载

3）训练模型（Train&Validate）

4）训练过程的可视化或相关指标的计算

5）测试/预测（Test/Inference）

另外程序还应该满足以下几个要求：

1）模型需具有高度可配置性，便于修改参数、修改模型，反复实验

2）代码应具有良好的组织结构，使人一目了然

3）代码应具有良好的说明，使其他人能够理解

接下来我将应用这些内容，并结合实际的例子，来讲解如何合理组织我们的文本分类项目。

5.1 文件组织结构

首先来看程序文件的组织结构：

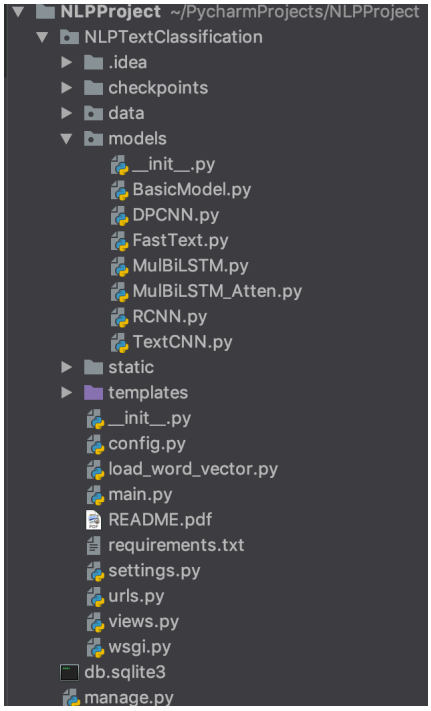

图 8: 程序文件的组织结构

其中：

1）checkpoints/：用于保存训练好的模型，可使程序在异常退出后仍能重新载入模型，恢复训练

2）data/：数据相关操作，包括数据预处理、dataset 实现等

3）models/：模型定义，可以有多个模型，例如上面的 FastText、TextCNN 等，一个模型对应一个文件

4）config.py：配置文件，所有可配置的变量都集中在此，并提供默认值

5）main.py：主文件，训练和预测程序的入口，可通过不同的命令来指定不同的操作和参数

6）load_word_vector.py: 定义加载预训练词向量的函数

7）.idea/、static/、templates/、settings.py、urls.py、views.py、wsgi.py、db.sqlite3、manage.py: 网页 Demo 运行支撑文件。

8）requirements.txt：程序依赖的第三方库

9）README.pdf：项目说明文档

5.2 数据预处理和加载

数据的相关预处理函数主要保存在 data/dataset.py 中。关于数据加载的相关操作，其基本原理就是使用 Dataset 进行数据集的封装，再使用 Dataloader 实现数据并行加载。

具体的预处理过程和实现细节在第三部分中已经详细介绍了。

使用时，我们可通过 dataloader 加载数据：

```
#读取之前预处理过程 保存的处理好的训练集、验证集和测试集
X_train = torch.load('./data/X_train.pt')
```

```
y_train = torch.load('./data/y_train.pt')
X_val = torch.load('./data/X_val.pt')
y_val = torch.load('./data/y_val.pt')
X_test = torch.load('./data/X_test.pt')
y_test = torch.load('./data/y_test.pt')

#封装成DataSet
trainset = Data.TensorDataset(X_train,y_train)
valset = Data.TensorDataset(X_val,y_val)
testset = Data.TensorDataset(X_test,y_test)

#使用DataLoader并行加载数据
train_iter = Data.DataLoader(trainset,opt.batch_size,shuffle=True,num_workers=opt.num_workers)
val_iter = Data.DataLoader(valset,opt.batch_size)
test_iter = Data.DataLoader(testset,opt.batch_size)
```

加载预训练词向量：

$load_word_vector.py$ 定义了加载预训练词向量的函数：

```
def read_word_vector(path): #path为 下载的预训练词向量 解压后的文件所在的路径
    #读取预训练词向量
    with open(path, 'r') as f:
        words = set()  # 定义一个words集合
        word_to_vec_map = {}  # 定义词到向量的映射字典
        for line in f:  #跳过文件的第一行
            break

        for line in f:  # 遍历f中的每一行
            line = line.strip().split()  # 去掉首尾空格，每一行以空格切分  返回一个列表  第一项为单词 其
                余为单词的嵌入表示
            curr_word = line[0]  # 取出单词
            words.add(curr_word)  # 加到集合/词典中
            # 定义词到其嵌入表示的映射字典
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)

    return words, word_to_vec_map

def load_pretrained_embedding(word2index, word2vector):
    #word2index是构建的词典（单词到索引的映射），word2vector是预训练词向量（单词到词向量的映射）

    embed = torch.zeros(len(word2index), opt.embed_size) # 初始化词嵌入矩阵为0
    oov_count = 0 # 找不到预训练词向量的词典中单词的个数

    for word, index in word2index.items(): #遍历词典中的每个单词 及其在词典中的索引
        try: #如果单词有对应的预训练词向量 则用预训练词向量对词嵌入矩阵的对应行进行赋值
            embed[index, :] = torch.from_numpy(word2vector[word])
        except KeyError:
            oov_count += 1

    if oov_count > 0:
        print("There are %d oov words."%oov_count)
    return embed #返回词嵌入矩阵
```

在主程序 main.py 的 train 函数中调用：

```
  #加载预训练词向量
```

```
if opt.use_pretrained_word_vector:
    words,word2vec = read_word_vector(opt.word_vector_path) #opt.word_vector_path为下载的预训练词向量 解压后的文件所在的路径
    print("预训练词向量读取完毕！")
    #读取之前预处理过程保存的词典（词到索引的映射）
    with open('./data/word2index.json') as f:
        word2index = json.load(f)

    model.embedding.weight.data.copy_(load_pretrained_embedding(word2index, word2vec)) #使用加载完预训练词向量的词嵌入矩阵 对embdding层的词嵌入矩阵赋值
    print("预训练词向量加载完毕！")
    if opt.frozen: #冻结还是finetuning
        model.embedding.weight.requires_grad = False
```

5.3 模型定义

各个模型的定义主要保存在 models/目录下，其中 BasicModule 是对 nn.Module 的简易封装，提供快速加载（可以处理 GPU 训练、CPU 加载的情况）和保存模型（提供多 GPU 训练时的模型保存方法）的接口，其他模型都继承自 BasicModule。

```

class BasicModule(nn.Module):
    '''
    封装了nn.Module，主要提供save和load两个方法
    '''
    def __init__(self,opt=None):
        super(BasicModule,self).__init__()
        self.model_name = str(type(self)) # 模型的默认名字

    def load(self, path):
        '''
        加载模型
        可指定路径
        '''
        self.load_state_dict(torch.load(path))

    def load_map(self, path,device): #如果在GPU上训练 在CPU上加载 可以调用这个函数
        '''
        加载模型
        可指定路径
        '''
        self.load_state_dict(torch.load(path,map_location=device))

    def save(self, name=None):
        '''
        保存模型，默认使用“模型名字_best”作为文件名，
        '''
        if name is None:
            prefix = 'checkpoints/' + self.model_name.split('.')[-2] + '_best.pth'
            #name = time.strftime(prefix + '%m%d_%H:%M:%S.pth')
        torch.save(self.state_dict(), prefix) #只保存模型的参数
        return name

```

```
    def save_multiGPU(self, name=None):  #如果使用多GPU训练，保存模型时，可以调用这个函数。
        '''
        保存模型，默认使用“模型名字_best”作为文件名，
        '''
        if name is None:
            prefix = 'checkpoints/' + self.model_name.split('.')[-2] + '_best.pth'
            # name = time.strftime(prefix + '%m%d_%H:%M:%S.pth')
        torch.save(self.module.state_dict(), prefix)  # 只保存模型的参数
        return name
```

在实际使用中，直接调用 model.save() 及 model.load(opt.load_path) 即可，我们已经对保存和加载做了封装。

其它自定义模型一般继承 BasicModule，然后实现自己的模型。其中 TextCNN.py 实现了 TextCNN，FastText.py 实现了 FastText 等。在 models/__init__.py 中，代码如下：

```
#本项目可选择的模型：
from .FastText import FastText
from .TextCNN import TextCNN
from .MulBiLSTM import MulBiLSTM
from .MulBiLSTM_Atten import MulBiLSTM_Atten
from .RCNN import RCNN
from .DPCNN import DPCNN
```

这样在主函数中就可以写成：

```
from models import TextCNN
```

或：

```
import models
model = models.TextCNN()
```

或：

```
import models
model = getattr(models, 'TextCNN')()
```

其中最后一种写法最为关键，这意味着我们可以通过字符串直接指定使用的模型，而不必使用判断语句，也不必在每次新增加模型后都修改代码。新增模型后只需要在 models/__init__.py 中加上

```
from .new_module import NewModule
```

各个模型的原理和实现细节在第二部分中已经详细介绍过了。

5.4 配置文件

在模型定义、数据处理和训练等过程都有很多变量，这些变量应提供默认值，并统一放置在配置文件中，这样在后期调试、修改代码或迁移程序时会比较方便，在这里我们将所有可配置项放在 config.py 中。

```
class DefaultConfig(object):

    model = 'FastText'  # 使用的模型，名字必须与models/__init__.py中的名字一致

    load_model_path = None  # 加载预训练的模型的路径，为None代表不加载

    batch_size = 256  # batch size
```

```
    num_workers = 4  # 加载数据使用的线程数

    #下载数据集 解压缩后得到的文件夹所在的路径
    data_root = '/Users/apple/Downloads/THUCNews-1'


    max_epoch = 20
    lr = 0.01  # initial learning rate
    weight_decay = 1e-4  # 损失函数 正则化
    embed_size = 100 #词嵌入维度
    drop_prop = 0.5 #丢弃率
    classes = 14  #分类类别数
    max_len = 500 #序列最大长度

    #学习率衰减相关超参数
    use_lrdecay = True #是否使用学习率衰减
    lr_decay = 0.95  # 衰减率
    n_epoch = 1  #每隔n_epoch个epoch衰减一次 lr = lr * lr_decay


    #TextCNN相关的超参数
    kernel_sizes = [3,4,5] #一维卷积核的大小
    num_channels = [100,100,100] #一维卷积核的数量

    #FastText相关的超参数
    linear_hidden_size =512  #隐层单元数

    #MulBiLSTM/MulBiLSTM_Atten相关超参数
    recurrent_hidden_size = 128 #循环层 单元数
    num_layers = 2      #循环层 层数

    #RCNN相关超参数
    num_layers_rcnn = 1 #循环层 层数
    drop_prop_rcnn = 0.0 #1个循环层设置为0 丢弃率

    #DPCNN相关超参数
    channel_size = 250
    drop_prop_dpcnn = 0.2

    #梯度裁剪相关超参数
    use_rnn = False
    norm_type = 1
    max_norm = 5

    #预训练词向量相关超参数
    use_pretrained_word_vector = False
    word_vector_path = '/Users/apple/Downloads/sgns.sogou.word'
    frozen = False

    #待分类文本
    text="众所周知，一支球队想要夺冠，超级巨星必不可少，不过得到超级巨星并不简单，方式无非两种，一是自己
        培养，这种方式适用于所有球队，二是交易，这种方式基本只适用于大市场球队——事实就是，30支球队之
        间并非完全公平，超级巨星依然更愿意前往大城市。"

    #预测时是否对文本进行填充或截断
```

```
    predict_pad = False
```

可配置的参数主要包括：

1）训练参数（学习率、训练 epoch 等）

2）各个模型相关的参数

这样我们在程序中就可以这样使用：

```
import models
from config import DefaultConfig

opt = DefaultConfig()
lr = opt.lr
model = getattr(models, opt.model)
```

这些都只是默认参数 (如果后续不在命令行指定的话，就是用默认参数)，在这里还提供了更新函数（根据命令行中指定的参数进行更新），根据字典更新配置参数。

```
def parse(self, kwargs):
    '''
    根据字典kwargs 更新 默认的config参数
    '''
    # 更新配置参数
    for k, v in kwargs.items():
        if not hasattr(self, k):
            # 警告还是报错，取决个人喜好
            warnings.warn("Warning: opt has not attribut %s" % k)
        setattr(self, k, v)

    # 打印配置信息
    print('user config:')
    for k, v in self.__class__.__dict__.items(): #python3 中iteritems()已经废除了
        if not k.startswith('__'):
            print(k, getattr(self, k))
```

这样我们在实际使用时，并不需要每次都修改 config.py（默认配置），只需要通过命令行传入所需参数，覆盖默认配置即可。

```
opt = DefaultConfig()
new_config = {'lr':0.1,'use_gpu':False}
opt.parse(new_config)
opt.lr == 0.1
```

5.5 main.py

在讲解主程序 main.py 之前，我们先来看看 2017 年 3 月谷歌开源的一个命令行工具 fire ，通过 pip install fire 即可安装。下面来看看 fire 的基础用法，假设 example.py 文件内容如下：

```
import fire
def add(x, y):
    return x + y

def mul(**kwargs):
    a = kwargs['a']
    b = kwargs['b']
```

```
    return a * b

if __name__ == '__main__':
    fire.Fire()
```

那么我们可以使用：

```
python example.py add 1 2 # 执行add(1, 2)
python example.py mul --a=1 --b=2 # 执行mul(a=1, b=2),kwargs={'a':1, 'b':2}
python example.py add --x=1 --y=2 # 执行add(x=1, y=2)
```

可见，只要在程序中运行 fire.Fire()，即可使用命令行参数 python file <function> [args,] –kwargs,。

在主程序 main.py 中，主要包含四个函数，其中三个需要命令行执行，main.py 的代码组织结构如下：

```
def train(**kwargs):
    '''
    训练
    '''
    pass

def evaluate_accuracy(data_iter, net,flag=False,labels=None):
    '''
    计算模型在验证集/测试集上的准确率等信息，用以辅助训练
    '''
    pass

def predict(**kwargs):
    '''
    对新样本进行预测
    '''
    pass

def help():
    '''
    打印帮助的信息
    '''
    print('help')

if __name__=='__main__':
    import fire
    fire.Fire()
```

1. 训练

训练的主要步骤如下：

1）定义网络

2）定义数据

3）定义损失函数和优化器

4）计算重要指标

5）开始训练

6）训练网络

7）计算在验证集上的指标

训练函数的代码如下：

```python
def train(**kwargs):

    # 根据命令行参数更新配置 否则使用默认配置
    opt.parse(kwargs)

    # step1: 数据
    #词典大小
    with open('./data/vocabsize.json') as f:
        vocab_size = json.load(f)
    print("词典大小:",vocab_size)
    #标签
    with open('./data/labels.json') as f:
        labels = json.load(f)

    #读取之前预处理过程 保存的处理好的训练集、验证集和测试集
    X_train = torch.load('./data/X_train.pt')
    y_train = torch.load('./data/y_train.pt')
    X_val = torch.load('./data/X_val.pt')
    y_val = torch.load('./data/y_val.pt')
    X_test = torch.load('./data/X_test.pt')
    y_test = torch.load('./data/y_test.pt')

    #封装成DataSet
    trainset = Data.TensorDataset(X_train,y_train)
    valset = Data.TensorDataset(X_val,y_val)
    testset = Data.TensorDataset(X_test,y_test)

    #使用DataLoader并行加载数据
    train_iter = Data.DataLoader(trainset,opt.batch_size,shuffle=True,num_workers=opt.num_workers)
    val_iter = Data.DataLoader(valset,opt.batch_size)
    test_iter = Data.DataLoader(testset,opt.batch_size)

    # step2: 模型
    model = getattr(models, opt.model)(vocab_size,opt)
    if opt.load_model_path:
        model.load(opt.load_model_path)

    #加载预训练词向量
    if opt.use_pretrained_word_vector:
        words,word2vec = read_word_vector(opt.word_vector_path) #opt.word_vector_path为下载的预训练词向量 解压后的文件所在的路径
        print("预训练词向量读取完毕! ")
        #读取之前预处理过程保存的词典（词到索引的映射）
        with open('./data/word2index.json') as f:
            word2index = json.load(f)

        model.embedding.weight.data.copy_(load_pretrained_embedding(word2index, word2vec)) #使用加载完预训练词向量的词嵌入矩阵 对embdding层的词嵌入矩阵赋值
        print("预训练词向量加载完毕! ")
        if opt.frozen: #冻结还是finetuning
            model.embedding.weight.requires_grad = False


    print("使用设备: ",device)
    if torch.cuda.device_count() > 1: #使用多GPU进行训练
```

```
54          print("Let's use", torch.cuda.device_count(), "GPUs!")
55          model = torch.nn.DataParallel(model)
56
57      model.to(device)
58
59      # step3: 目标函数和优化器
60      criterion = torch.nn.CrossEntropyLoss()
61      optimizer = torch.optim.Adam(model.parameters(),
62      lr = opt.lr,
63      weight_decay = opt.weight_decay)
64      scheduler = lr_scheduler.StepLR(optimizer,opt.n_epoch,opt.lr_decay)
65      # 训练
66      batch_count = 0
67      best_f1_val = 0.0
68
69
70      for epoch in range(opt.max_epoch):
71          train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
72          if opt.use_lrdecay:
73              scheduler.step()
74          for X, y in train_iter:
75              X = X.to(device)
76              y = y.to(device)
77              y_hat = model(X)
78              loss = criterion(y_hat, y)
79              optimizer.zero_grad()
80              loss.backward()
81              if opt.use_rnn: #梯度裁剪
82                  nn.utils.clip_grad_norm_(model.parameters(), max_norm=opt.max_norm, norm_type=opt.
                        norm_type)
83              optimizer.step()
84              train_l_sum += loss.cpu().item()
85              train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
86              n += y.shape[0]
87              batch_count += 1
88
89
90          #一个epoch后在验证集上做一次验证
91          val_f1,val_acc = evaluate_accuracy(val_iter, model)
92          if val_f1 > best_f1_val:
93              best_f1_val = val_f1
94              # 保存在验证集上weighted average f1最高的参数（最好的参数）
95              if torch.cuda.device_count() > 1: #多GPU训练时保存参数
96                  print("Saving on ", torch.cuda.device_count(), "GPUs!")
97                  model.save_multiGPU()
98              else:
99                  print("Saving on one GPU!")#单GPU训练时保存参数
100                 model.save()
101             #使用当前最好的参数，在测试集上再跑一遍
102             best_f1_test,best_acc_test = evaluate_accuracy(test_iter,model,True,labels)
103
104         print('epoch %d, lr %.6f,loss %.4f, train acc %.3f, val acc %.3f,val weighted f1 %.3f, val
                best_weighted f1 %.3f,test best_acc %.3f,test best_weighted f1 %.3f,time %.1f sec'
105         % (epoch + 1, optimizer.state_dict()['param_groups'][0]['lr'],train_l_sum / batch_count,
                train_acc_sum / n, val_acc,val_f1, best_f1_val,best_acc_test,best_f1_test,time.time() -
```

```
            start))
```

2. 验证/测试

验证相对来说比较简单，但要注意需将模型置于验证模式 (model.eval())，验证完成后还需要将其置回为训练模式 (model.train())，这两句代码会影响 BatchNorm 和 Dropout 等层的运行模式。

多分类我们使用 weighed average f1-score 作为评估指标，主要使用 sklearn 中的指标计算函数。

代码如下：

```
def evaluate_accuracy(data_iter, net,flag=False,labels=None):
    #计算模型在验证集上的相关指标 多分类我们使用 weighed average f1-score

    acc_sum, n = 0.0, 0
    net.eval()  # 评估模式, 这会关闭dropout
    y_pred_total = []
    y_total = []
    with torch.no_grad():
        for X, y in data_iter:
            #acc_sum += (net(X.to(device)).argmax(dim=1) == y.to(device)).float().sum().cpu().item()
            #n += y.shape[0]
            y_pred = net(X.to(device)).argmax(dim=1).cpu().numpy()
            y_pred_total.append(y_pred)
            y_total.append(y.numpy())

    y_pred = np.concatenate(y_pred_total)
    y_label = np.concatenate(y_total)
    weighted_f1 = f1_score(y_label,y_pred,average='weighted') #weighed average f1-score

    accuracy = accuracy_score(y_label,y_pred) #准确率
    if flag: #当在测试集上验证时 flag设置为True  额外打印分类报告和混淆矩阵
        print(classification_report(y_label,y_pred,digits=4,target_names = labels))
        cm = confusion_matrix(y_label,y_pred)
        print(cm)
    net.train()  # 改回训练模式

    return weighted_f1,accuracy
```

3. 预测

对于新的输入文本，我们加载训练好的模型进行预测，输出类别标签：

```
def predict(**kwargs):
    # 根据命令行参数更新配置 否则使用默认配置
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("使用设备:", device)
    opt.parse(kwargs)
    text = opt.text #待分类文本

    # 词典大小
    with open('./data/vocabsize.json') as f:
        vocab_size = json.load(f)
    print(vocab_size)

    #创建指定的模型对象
    model = getattr(models, opt.model)(vocab_size, opt)

```

```
    #加载训练好的模型参数
    if device.type=='cpu': #GPU训练 CPU预测 加载参数时需要对参数进行映射
        model.load_map('./checkpoints/'+opt.model+'_best.pth',device)
    else:
        model.load('./checkpoints/' + opt.model + '_best.pth')

    #加载之前预处理过程 保存的词到索引的映射字典
    with open('./data/word2index.json') as f:
        word2index = json.load(f)

    #device = list(model.parameters())[0].device
    if opt.predict_pad:             #预测时对文本进行填充（若文本长度<opt.max_len）或截断（若文本长度>
        opt.max_len)
        sentence = [word2index.get(word, 1) for word in jieba.lcut(text)]
        sentence = sentence[:opt.max_len] if len(sentence) > opt.max_len else sentence + [0] * (opt.
            max_len - len(sentence))
        sentence = torch.tensor(sentence,device=device)
    else:
        sentence = torch.tensor([word2index.get(word,1) for word in jieba.lcut(text)],device=device)
    print(sentence)
    #预测
    with torch.no_grad():
        model.eval()
        label = torch.argmax(model(sentence.view((1,-1))),dim=1)

    # 加载之前预处理过程 保存的索引到类别标签的映射字典
    with open('./data/index2labels.json') as f:
        index2labels = json.load(f)
    #输出新文本的类别标签
    print(index2labels[str(label.item())])
```

4. 帮助函数

为了方便他人使用，程序中还应当提供一个帮助函数，用于说明函数是如何使用。程序的命令行接口中有众多参数，如果手动用字符串表示不仅复杂，而且后期修改 config 文件时，还需要修改对应的帮助信息，十分不便。这里使用了 Python 标准库中的 inspect 方法，可以自动获取 config 的源代码。help 的代码如下：

```
def help():

    '''
    打印帮助的信息:  python file.py help
    '''

    print('''
    usage : python {0} <function> [--args=value,]
    <function> := train | test | help
    example:
    python {0} train --model='TextCNN' --lr=0.01
    python {0} test --text='xxxxx'
    python {0} help
    avai    able
    args: '''.format(__file__))

    from inspect import getsource
```

```
    source = (getsource(opt.__class__))
    print(source)
```

6 项目使用方式

1. 下载 THUCnews 数据集 (完整数据集压缩包下载)，解压缩，并修改 config.py:

```
#下载数据集 解压缩后得到的文件夹所在的路径
data_root = '/Users/apple/Downloads/THUCNews-1'
```

2. 下载预训练词向量（项目所使用预训练词向量），解压缩，并修改 config.py:

```
 #预训练词向量相关超参数
word_vector_path = '/Users/apple/Downloads/sgns.sogou.word' #下载的预训练词向量 解压后的文件所在的路径
```

3. 进入 data 目录下，运行 dataset.py，对数据进行预处理并生成必要的中间文件（数据集非常大，此过程需要 3-4 小时）:

```
python dataset.py
```

4. 之后便可以训练模型（在 main.py 所在的目录下运行）:

```
可以在命令行指定新的超参数覆盖默认超参数配置 不然将使用config.py中的默认超参数
# 训练模型 单GPU
CUDA_VISIBLE_DEVICES=5 nohup python -u main.py train
--model='TextCNN'
--lr=0.01
--batch-size=256
--max-epoch = 20 >zdz.log 2>&1 &

# 训练模型 多GPU
CUDA_VISIBLE_DEVICES=0,1,2,5 nohup python -u main.py train
--model='DPCNN'
--drop_prop_dpcnn=0.2
--batch-size=256
--max-epoch = 10 >zdz.log 2>&1 &

# 打印帮助信息
python main.py help
```

7 实验结果与分析

7.1 实验结果

训练各个模型的超参数均采用默认超参数，具体的配置在 config.py 中。各个模型在测试集上的 weight f1-score 值如表2所示:

表 2: 实验结果

模型	测试集 weighted f1-score	备注
FastText	92.6%	Baseline
TextCNN	93.8%	
RCNN	93.0%	
多层双向 LSTM	94.4%	
多层双向 LSTM with Attention	94.7%	
DPCNN	95.4%	

7.2 实验分析

- FastText 模型是我们的 Baseline，在测试集上的 weighted f1-score 为 92.6%
- DPCNN 模型在测试集上取得了最好的性能，其 weighted f1-score 为 95.4%
- Attention 机制在分类问题上的效果不是很明显，多层双向 LSTM 和多层双向 LSTM with Attention 的性能几乎差不多
- 比较意外地是 RCNN 模型 (结合 RNN 与 CNN) 并没有取得预期的性能，相比 FastText 提升不大，可能模型细节还需要细究
- TextCNN 是” 性价比” 最高的模型，模型比较简单，训练很容易，但效果非常不错。
- 上述实验结果均基于默认的超参数配置，调参可能会取得更好的性能
- 基于 RNN 的模型相对基于 CNN 的模型更难训练（相同配置下，训练时间更长），但效果并没有显著优势。基于 RNN 的模型训练一定轮数后可能会出现梯度爆炸，注意使用梯度剪切技巧
- 深层网络 (DPCNN) 相对于浅层网络 (TextCNN) 更难训练，但效果提升比较明显。深层网络要注意缓解梯度消失现象（比如，使用残差连结）。

8 预测与网页 Demo

8.1 通过命令行执行预测

模型训练完成后，便可以对新闻文本执行预测过程，预测其对应的主题标签（在 main.py 所在的目录下运行）：

```
# 通过命令行运行预测过程
python main.py predict
    --model='RCNN'  #指定预测所使用的model
    --text='众所周知，一支球队想要夺冠，超级巨星必不可少，不过得到超级巨星并不简单，方式无非两种，一是自
        己培养，这种方式适用于所有球队，二是交易。'  #待分类文本
```

8.2 通过网页 Demo 执行预测

1）进入项目目录，运行 manage.py（在 manage.py 所在的目录下运行）：

```
python manage.py runserver
```

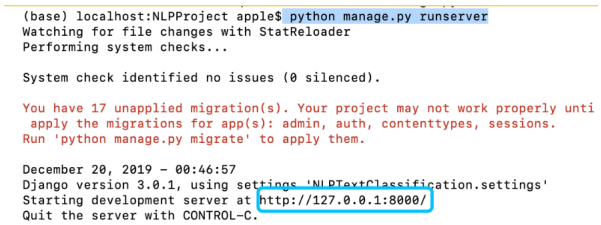

图 9: 运行网页 Demo

2）在浏览器打开网址：

```
http://127.0.0.1:8000/index  #不要忘了index
```

3）界面如下所示：

图 10: 网页 Demo 界面

把待分类的文本粘贴到上图中的红框中，在蓝框的下拉列表中选择模型，点击上传分析后，在下方的绿框中便可显示分类结果。

4）支持的新闻类别有 14 个：财经、彩票、房产、股票、家居、教育、科技、社会、时尚、时政、体育、星座、游戏、娱乐。

以下是一些示例（注意：我们的系统是对篇章进行分类，最长不要超过 500 个词。）：

a. 体育："众所周知，一支球队想要夺冠，超级巨星必不可少，不过得到超级巨星并不简单，方式无非两种，一是自己培养，这种方式适用于所有球队，二是交易，这种方式基本只适用于大市场球队——事实就是，30 支球队之间并非完全公平，超级巨星依然更愿意前往大城市。"

b. 体育："德国球星萨内告知拜仁慕尼黑，他希望在一月份转会加盟，但据英国媒体报道，曼城方面对他的要价高达 1 亿英镑。尽管萨内自从 8 月份以来就一直伤停，但拜仁仍对他感兴趣。曼城方面也清楚萨内想走，但他们的立场是，买家出价合适才会放人，而瓜迪奥拉对他的要价是 1 亿英镑，这笔钱将用来买进新球员补充阵容。"

c. 娱乐:“有网友晒出了范冰冰现身好莱坞华裔导演温子仁新作《恶毒》的杀青晚宴现场的照片。照片中范冰冰戴着暗绿色帽子，穿着黑色皮衣，留着大波浪长发，五官精致笑容甜美，与众主创合照站 C 位，很有排面。”

d. 娱乐:“北京时间 12 月 20 日消息，据香港媒体报道，电影《急先锋》日前在北京举行发布会，导演唐季礼联同成龙、杨洋、艾伦、朱正廷、母其弥雅（MIYA）等主角齐齐亮相，并分享台前幕后的故事。其中导演唐季礼特别提到，在拍摄过程中快艇不慎被石头掀翻，一下将成龙扣在水下，把自己吓哭了！今年成龙大哥又有新作，与老拍档唐季礼导演连手打造新片《急先锋》，并且找来一班新血组成中国版“复仇者”，在大年初一与观众贺岁，让影迷万分期待，《急先锋》日前在北京举行发布会。”

e. 游戏:“尽管新英雄厄斐琉斯还没有正式登陆各大服务器，但拳头已经通过邮件向国外网友发送了另一名新英雄的神秘技能卡片，卡片描绘出其他的英雄被某个技能打中的效果，但是这个打击效果的来源却是未知的。之前拳头曾在英雄制作大纲写道:“在厄斐琉斯之后的英雄会是一位来自艾欧尼亚的斗士。这名英雄在打斗中茁壮成长，在受到过对方强烈的击打后，他（她）会狂笑，并且将所有受到的挑衅全部释放到对方的脸上。如果你喜欢用拳头说话，喜欢致命搏击，或者喜欢在激烈的战斗中把对手的头打得粉碎，他（她）可能是你的本命英雄。”这是铁拳要进入联盟了吗”

9 所需环境

```
numpy    >=1.16.2
json    >=2.0.9
jieba    >=0.39
torch    >=1.1.0
torchtext    >=0.4.0
sklearn    >=0.20.3
django    >=3.0.1
fire    >=0.2.1
```

10 总结

1）本项目并没有解决训练集样本类别分布不均衡的问题，之后会考虑解决这个问题。

2）尽管各个模型在测试集上的准确率都在 90% 以上，但在预测时，准确率会有一定程度的下降，原因在于真实 (预测) 数据的分布和训练集数据的分布不同。

3）本项目主要实现了一些基于 CNN、RNN（Attention）的文本分类模型，并没有实现一些基于预训练语言模型（如 Bert、XLNet 等）的分类模型，之后会逐步完善。