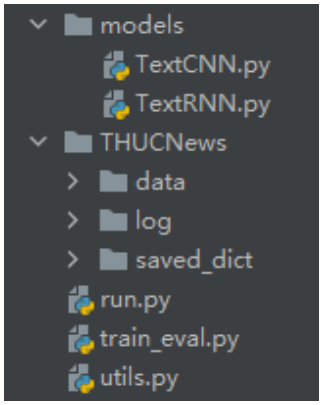
卷积神经网络实现THUCNews新闻文本分类

代码结构

整体代码结构如下图所示：



点击run.py文件，直接运行。可以手动调节参数以及更换模型

1数据集

本文采用的数据集属于清华NLP组提供的THUCNews新闻文本分类数据集的一个子集（原始的数据集大约74万篇文档，训练起来需要花较长的时间）。数据集请自行到THUCTC：一个高效的中文文本分类工具包下载，请遵循数据提供方的开源协议。

下载的数据放入THUCNews/data目录中。本次训练使用了其中的10个分类，每个分类6500条，总共65000条新闻数据。

类别如下：

> 体育, 财经, 房产, 家居, 教育, 科技, 时尚, 时政, 游戏, 娱乐

数据集划分如下：

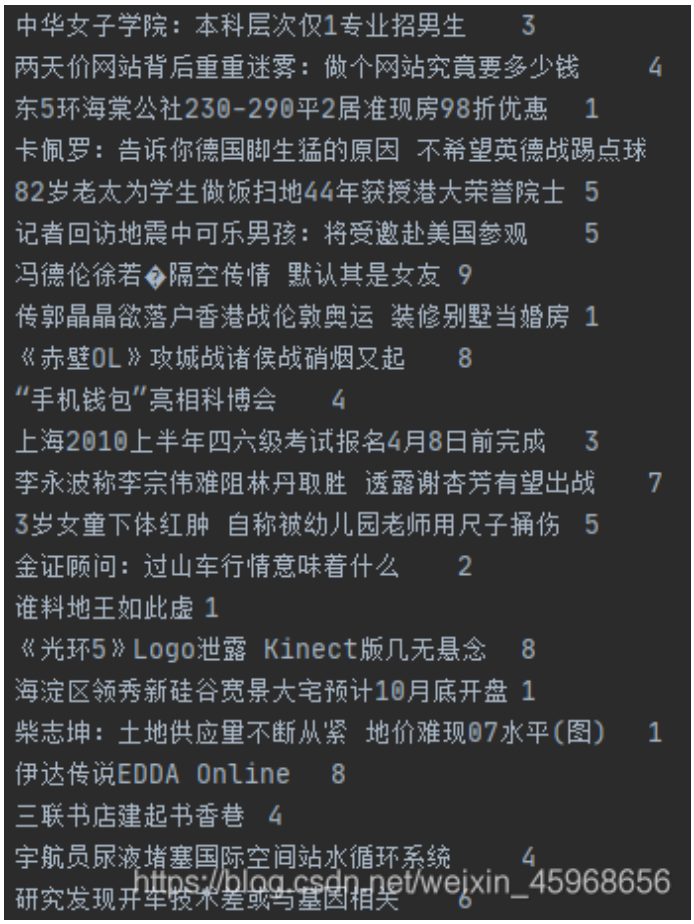
- 训练集：5000*10
- 验证集：500*10
- 测试集：1000*10

从原始数据集生成子集的过程请参看helper下的两个脚本。其中copy_data.sh用于从每个分类拷贝6500个文件，cnews_group.py用于将多个文件整合到一个文件中。执行该文件后，得到三个数据文件：

- train.txt: 训练集(50000条)

- dev.txt: 验证集(5000条)
- test.txt: 测试集(10000条)

测试集示例：



2预处理

调用加载数据的函数返回预处理的数据

Python

```python
def build_vocab(file_path, tokenizer, max_size, min_freq):
    vocab_dic = {}
    with open(file_path, 'r', encoding='UTF-8') as f:
        for line in tqdm(f):
            lin = line.strip()
            if not lin:
                continue
            content = lin.split('\t')[0]
            for word in tokenizer(content):
                vocab_dic[word] = vocab_dic.get(word, 0) + 1
```

```
        vocab_list = sorted([_ for _ in vocab_dic.items() if _[1] >= min_freq], key=lambda x: x[1], reverse=True)[:max_size]
        vocab_dic = {word_count[0]: idx for idx, word_count in enumerate(vocab_list)}
        vocab_dic.update({UNK: len(vocab_dic), PAD: len(vocab_dic) + 1})
    return vocab_dic
def build_dataset(config, ues_word):
    if ues_word:
        tokenizer = lambda x: x.split(' ')  # 以空格隔开，word-level
    else:
        tokenizer = lambda x: [y for y in x]  # char-level
    if os.path.exists(config.vocab_path):
        vocab = pkl.load(open(config.vocab_path, 'rb'))
    else:
        vocab = build_vocab(config.train_path, tokenizer=tokenizer, max_size=MAX_VOCAB_SIZE, min_freq=1)
        pkl.dump(vocab, open(config.vocab_path, 'wb'))
    print(f"Vocab size: {len(vocab)}")

    def load_dataset(path, pad_size=32):
        contents = []
        with open(path, 'r', encoding='UTF-8') as f:
            for line in tqdm(f):
                lin = line.strip()
                if not lin:
                    continue
                content, label = lin.split('\t')
                words_line = []
                token = tokenizer(content)
                seq_len = len(token)
                if pad_size:
                    if len(token) < pad_size:
                        token.extend([vocab.get(PAD)] * (pad_size - len(token)))
                    else:
                        token = token[:pad_size]
                        seq_len = pad_size
                # word to id
                for word in token:
                    words_line.append(vocab.get(word, vocab.get(UNK)))
                contents.append((words_line, int(label), seq_len))
        return contents  # [([...], 0), ([...], 1), ...]
    train = load_dataset(config.train_path, config.pad_size)
    dev = load_dataset(config.dev_path, config.pad_size)
    test = load_dataset(config.test_path, config.pad_size)
    return vocab, train, dev, test
```

以上代码：

1. 获取分词方式（单字或者单词，这里使用单字）
2. 获取字典类型的词汇表（key=字，value=索引）
3. 获取三个数据集分词之后的索引列表（padding之后长度固定为max_size）

然后将数据封装到迭代器中

Python

```python
class DatasetIterater(object):
    def __init__(self, batches, batch_size, device):
        self.batch_size = batch_size
        self.batches = batches
        self.n_batches = len(batches) // batch_size
        self.residue = False  # 记录batch数量是否为整数
        if len(batches) % self.n_batches != 0:
            self.residue = True
        self.index = 0
        self.device = device
    def _to_tensor(self, datas):
        x = torch.LongTensor([_[0] for _ in datas]).to(self.device)
        y = torch.LongTensor([_[1] for _ in datas]).to(self.device)、
        # pad前的长度(超过pad_size的设为pad_size)
        seq_len = torch.LongTensor([_[2] for _ in datas]).to(self.device)
        return (x, seq_len), y
    def __next__(self):
        if self.residue and self.index == self.n_batches:
            batches = self.batches[self.index * self.batch_size:
len(self.batches)]
            self.index += 1
            batches = self._to_tensor(batches)
            return batches
        elif self.index > self.n_batches:
            self.index = 0
            raise StopIteration
        else:
            batches = self.batches[self.index * self.batch_size: (self.index +
1) * self.batch_size]
            self.index += 1
            batches = self._to_tensor(batches)
            return batches
    def __iter__(self):
        return self

    def __len__(self):
        if self.residue:
            return self.n_batches + 1
        else:
            return self.n_batches
```

定义DatasetIterater类，并传入需要封装的数据以及需要的batch尺寸或长度。在该类中会对数据进行张量转换。关键是重写__next__()、**iter**()、**len**()三个方法。

next（）返回每个batch的张量数据

iter()迭代

len()返回根据数据总样本与batch尺寸计算出来的batch个数

预处理之后的数据只需要通过for循环就可以一次获取一个batch的张量数据

3定义CNN模型

首先将网络模型参数设置封装成类：

Python

```python
class configure:
    def __init__(self):
        self.dropout = 0.5                                              # 随机失活
        self.require_improvement = 1000                                 # 若超过1000batch效果还没提升，则提前结束训练
        self.num_classes = len(self.class_list)                         # 类别数
        self.n_vocab = 0                                                # 词表大小，在运行时赋值
        self.num_epochs = 20                                            # epoch数
        self.batch_size = 128                                           # mini-batch大小
        self.pad_size = 32                                              # 每句话处理成的长度(短填长切)
        self.learning_rate = 1e-3                                       # 学习率
        self.embed = self.embedding_pretrained.size(1)\
            if self.embedding_pretrained is not None else 300           # 字向量维度
        self.filter_sizes = (2, 3, 4)                                   # 卷积核尺寸
        self.num_filters = 256                                          # 卷积核数量(channels数)
```

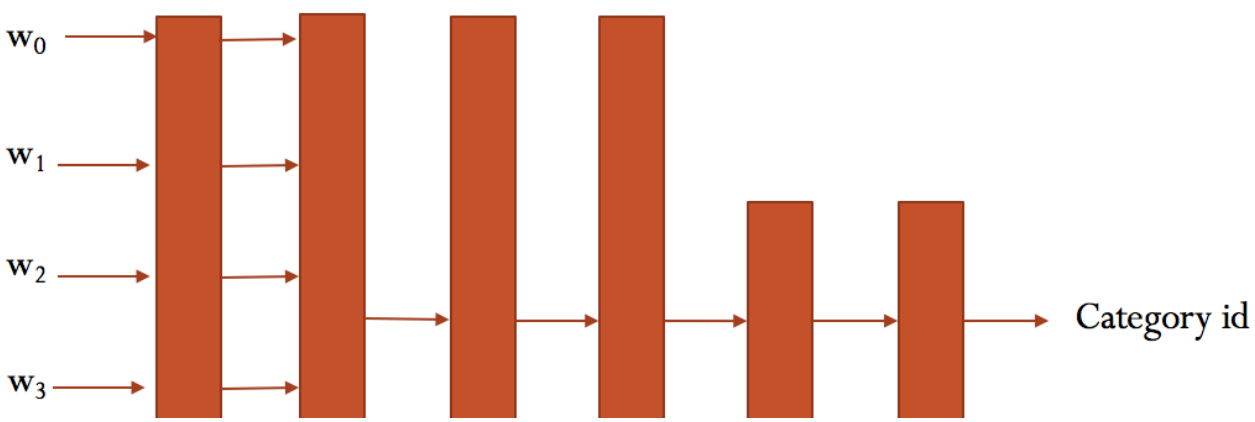
然后定义模型：

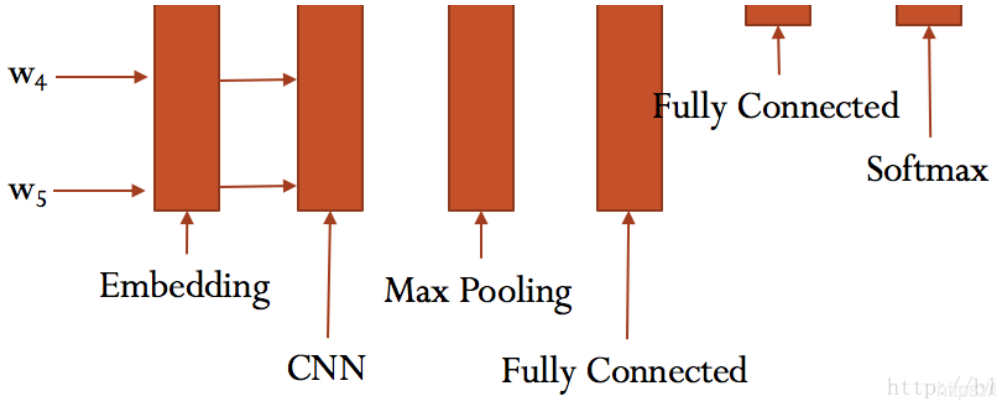
Python

```python
class Model(nn.Module):
    def __init__(self, config):
        super(Model, self).__init__()
        if config.embedding_pretrained is not None:
            self.embedding = nn.Embedding.from_pretrained(config.embedding_pretrained, freeze=False)
        else:
            self.embedding = nn.Embedding(config.n_vocab, config.embed, padding_idx=config.n_vocab - 1)
        self.convs = nn.ModuleList(
            [nn.Conv2d(1, config.num_filters, (k, config.embed)) for k in config.filter_sizes])
        self.dropout = nn.Dropout(config.dropout)
        self.fc = nn.Linear(config.num_filters * len(config.filter_sizes), config.num_classes)

    def conv_and_pool(self, x, conv):
        x = F.relu(conv(x)).squeeze(3)
        x = F.max_pool1d(x, x.size(2)).squeeze(2)
        return x
    def forward(self, x):
        #print (x[0].shape)
        out = self.embedding(x[0])
        out = out.unsqueeze(1)
        out = torch.cat([self.conv_and_pool(out, conv) for conv in self.convs], 1)
        out = self.dropout(out)
        out = self.fc(out)
        return out
```

该模型数据流向图：





整体模型打印如下：

Python

```
<bound method Module.parameters of Model(
  (embedding): Embedding(4762, 300)
  (convs): ModuleList(
    (0): Conv2d(1, 256, kernel_size=(2, 300), stride=(1, 1))
    (1): Conv2d(1, 256, kernel_size=(3, 300), stride=(1, 1))
    (2): Conv2d(1, 256, kernel_size=(4, 300), stride=(1, 1))
  )
  (dropout): Dropout(p=0.5)
  (fc): Linear(in_features=768, out_features=10, bias=True)
)>
```

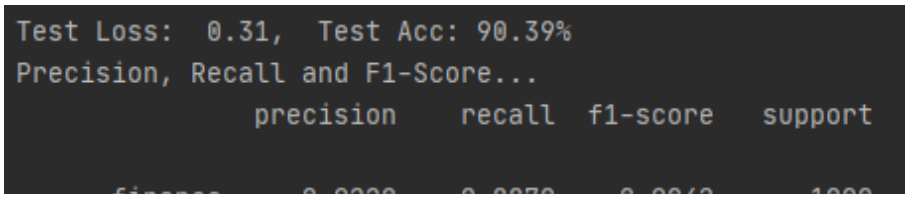
4训练与验证及测试：

Plain Text

```
Epoch [1/20]
Iter:      0,  Train Loss:   2.3,  Train Acc: 12.50%,  Val Loss:   2.7,  Val
```

```
Acc: 10.00%,  Time: 0:00:04 *
Iter:    100,  Train Loss:  0.75,  Train Acc: 70.31%,  Val Loss:  0.69,  Val Acc: 78.74%,  Time: 0:00:40 *
Iter:    200,  Train Loss:  0.69,  Train Acc: 76.56%,  Val Loss:  0.55,  Val Acc: 83.48%,  Time: 0:01:18 *
Iter:    300,  Train Loss:  0.47,  Train Acc: 82.81%,  Val Loss:  0.49,  Val Acc: 84.66%,  Time: 0:01:54 *
Iter:    400,  Train Loss:  0.73,  Train Acc: 78.12%,  Val Loss:  0.47,  Val Acc: 85.48%,  Time: 0:02:31 *
Iter:    500,  Train Loss:  0.39,  Train Acc: 87.50%,  Val Loss:  0.44,  Val Acc: 86.33%,  Time: 0:03:08 *
Iter:    600,  Train Loss:  0.49,  Train Acc: 84.38%,  Val Loss:  0.43,  Val Acc: 86.58%,  Time: 0:03:45 *
Iter:    700,  Train Loss:   0.5,  Train Acc: 83.59%,  Val Loss:  0.41,  Val Acc: 87.10%,  Time: 0:04:23 *
Iter:    800,  Train Loss:  0.47,  Train Acc: 84.38%,  Val Loss:  0.39,  Val Acc: 87.79%,  Time: 0:05:00 *
Iter:    900,  Train Loss:  0.43,  Train Acc: 86.72%,  Val Loss:  0.38,  Val Acc: 88.16%,  Time: 0:05:37 *
Iter:   1000,  Train Loss:  0.35,  Train Acc: 87.50%,  Val Loss:  0.39,  Val Acc: 87.94%,  Time: 0:06:14
Iter:   1100,  Train Loss:  0.42,  Train Acc: 89.84%,  Val Loss:  0.38,  Val Acc: 88.47%,  Time: 0:06:50 *
Iter:   1200,  Train Loss:  0.35,  Train Acc: 86.72%,  Val Loss:  0.36,  Val Acc: 88.99%,  Time: 0:07:27 *
Iter:   1300,  Train Loss:  0.44,  Train Acc: 88.28%,  Val Loss:  0.37,  Val Acc: 88.73%,  Time: 0:08:04
Iter:   1400,  Train Loss:  0.48,  Train Acc: 85.94%,  Val Loss:  0.36,  Val Acc: 88.92%,  Time: 0:08:41 *
Epoch [2/20]
Iter:   1500,  Train Loss:  0.39,  Train Acc: 90.62%,  Val Loss:  0.35,  Val Acc: 89.31%,  Time: 0:09:18 *
Iter:   1600,  Train Loss:  0.31,  Train Acc: 86.72%,  Val Loss:  0.35,  Val Acc: 89.06%,  Time: 0:09:54
Iter:   1700,  Train Loss:  0.34,  Train Acc: 92.19%,  Val Loss:  0.35,  Val Acc: 89.41%,  Time: 0:10:31 *
Iter:   1800,  Train Loss:  0.29,  Train Acc: 92.97%,  Val Loss:  0.37,  Val Acc: 88.60%,  Time: 0:11:08
Iter:   1900,  Train Loss:  0.38,  Train Acc: 89.06%,  Val Loss:  0.35,  Val Acc: 89.43%,  Time: 0:11:45 *
Iter:   2000,  Train Loss:  0.32,  Train Acc: 88.28%,  Val Loss:  0.34,  Val Acc: 89.41%,  Time: 0:12:22 *
Iter:   2100,  Train Loss:  0.32,  Train Acc: 89.06%,  Val Loss:  0.35,  Val Acc: 89.37%,  Time: 0:12:58
Iter:   2200,  Train Loss:  0.22,  Train Acc: 90.62%,  Val Loss:  0.34,  Val Acc: 89.44%,  Time: 0:13:35 *
Iter:   2300,  Train Loss:  0.39,  Train Acc: 91.41%,  Val Loss:  0.34,  Val Acc: 89.62%,  Time: 0:14:12 *
```

```
Iter:   2400,  Train Loss:  0.28,  Train Acc: 93.75%,  Val Loss:  0.34,  Val Acc: 89.54%,  Time: 0:14:49
Iter:   2500,  Train Loss:  0.21,  Train Acc: 92.97%,  Val Loss:  0.33,  Val Acc: 90.02%,  Time: 0:15:26 *
Iter:   2600,  Train Loss:  0.34,  Train Acc: 89.06%,  Val Loss:  0.33,  Val Acc: 89.90%,  Time: 0:16:03
Iter:   2700,  Train Loss:  0.26,  Train Acc: 91.41%,  Val Loss:  0.33,  Val Acc: 89.76%,  Time: 0:16:39
Iter:   2800,  Train Loss:  0.42,  Train Acc: 85.94%,  Val Loss:  0.34,  Val Acc: 89.52%,  Time: 0:17:16
Epoch [3/20]
Iter:   2900,  Train Loss:  0.34,  Train Acc: 89.84%,  Val Loss:  0.33,  Val Acc: 89.99%,  Time: 0:17:53 *
Iter:   3000,  Train Loss:  0.27,  Train Acc: 91.41%,  Val Loss:  0.33,  Val Acc: 89.70%,  Time: 0:18:29
Iter:   3100,  Train Loss:   0.3,  Train Acc: 89.06%,  Val Loss:  0.34,  Val Acc: 89.83%,  Time: 0:19:06
Iter:   3200,  Train Loss:   0.4,  Train Acc: 90.62%,  Val Loss:  0.33,  Val Acc: 90.00%,  Time: 0:19:43
Iter:   3300,  Train Loss:  0.37,  Train Acc: 89.84%,  Val Loss:  0.33,  Val Acc: 90.12%,  Time: 0:20:20 *
Iter:   3400,  Train Loss:  0.32,  Train Acc: 89.06%,  Val Loss:  0.33,  Val Acc: 90.07%,  Time: 0:20:57
Iter:   3500,  Train Loss:  0.19,  Train Acc: 92.97%,  Val Loss:  0.33,  Val Acc: 89.78%,  Time: 0:21:35
Iter:   3600,  Train Loss:  0.14,  Train Acc: 95.31%,  Val Loss:  0.33,  Val Acc: 89.74%,  Time: 0:22:12
Iter:   3700,  Train Loss:  0.29,  Train Acc: 89.84%,  Val Loss:  0.33,  Val Acc: 89.74%,  Time: 0:22:49
Iter:   3800,  Train Loss:  0.28,  Train Acc: 88.28%,  Val Loss:  0.33,  Val Acc: 90.11%,  Time: 0:23:25
Iter:   3900,  Train Loss:  0.32,  Train Acc: 87.50%,  Val Loss:  0.34,  Val Acc: 89.73%,  Time: 0:24:02
Iter:   4000,  Train Loss:  0.28,  Train Acc: 89.84%,  Val Loss:  0.33,  Val Acc: 89.97%,  Time: 0:24:39
Iter:   4100,  Train Loss:  0.26,  Train Acc: 90.62%,  Val Loss:  0.33,  Val Acc: 90.25%,  Time: 0:25:16
Iter:   4200,  Train Loss:  0.35,  Train Acc: 87.50%,  Val Loss:  0.33,  Val Acc: 90.04%,  Time: 0:25:53
```



```
      finance     0.9220    0.8870    0.9042       1000
       realty     0.9035    0.9360    0.9194       1000
       stocks     0.8483    0.8610    0.8546       1000
    education     0.9586    0.9490    0.9538       1000
      science     0.8381    0.8750    0.8562       1000
      society     0.8885    0.9080    0.8981       1000
     politics     0.9206    0.8460    0.8817       1000
       sports     0.9322    0.9620    0.9469       1000
         game     0.9154    0.9090    0.9122       1000
entertainment     0.9179    0.9060    0.9119       1000

     accuracy                         0.9039      10000
    macro avg     0.9045    0.9039    0.9039      10000
 weighted avg     0.9045    0.9039    0.9039      10000
```



```
Confusion Matrix...
[[887  19  53   0  12  10   7   6   2   4]
 [  9 936  14   2   4  13   5   3   3  11]
 [ 40  28 861   2  33   4  23   4   4   1]
 [  0   3   2 949   8  13   6   4   2  13]
 [  3   9  32   3 875  13  15   5  32  13]
 [  5  19   3  16  13 908  15   3   6  12]
 [ 10   9  39   9  25  45 846   6   2   9]
 [  1   2   3   1   6   7   1 962   3  14]
 [  4   4   7   4  52   4   1  11 909   4]
 [  3   7   1   4  16   5   0  28  30 900]]
```


在测试集中的正确率达到90.39%，precision、recall、f1-scores都达到了90%以上。

从混淆矩阵也可以看出分类效果非常优秀。