基于 BERT 和 TextCNN 的新闻舆情分类对比

谷 玥 (231017000021) 徐 鑫 (231017000190) 杨 思 源 (23101700017) 邢 小 斌 (231017000178)

项目背景及目标

基于新闻舆情数据，分析文本内容，判断新闻所属类别。

数据准备及代码实现

数据准备及处理

测试数据集选取 20 万条中文新闻标题，文本长度在 50 字符以内，新闻共有 10 个分类（财经、房产、股票、教育、科技、社会、时政、体育、游戏、娱乐），平均每类 2 万条。
训练数据集选用搜狗新闻，对新闻分类编码，加工预训练数据为[["新闻标题"],["新闻分类编码"]]的格式。

样例数据：

```
体验 2D 巅峰  倚天屠龙记十大创新概览	8
60 年铁树开花形状似玉米芯(组图)	5
同步 A 股首秀：港股缩量回调	2
中青宝 sg 现场抓拍  兔子舞热辣表演	8
锌价难续去年辉煌	0
2 岁男童爬窗台不慎 7 楼坠下获救(图)	5
布拉特：放球员一条生路吧  FIFA 能消化俱乐部的攻击7
金科西府  名墅天成	1
状元心经：考前一周重点是回顾和整理	3
发改委治理涉企收费每年为企业减负超百亿	6
```

数据集划分

训练集|18 万
验证集|1 万
测试集|1 万

模型构建及结果展示

采用 bert 和 TextCNN 两种方法对训练模型。

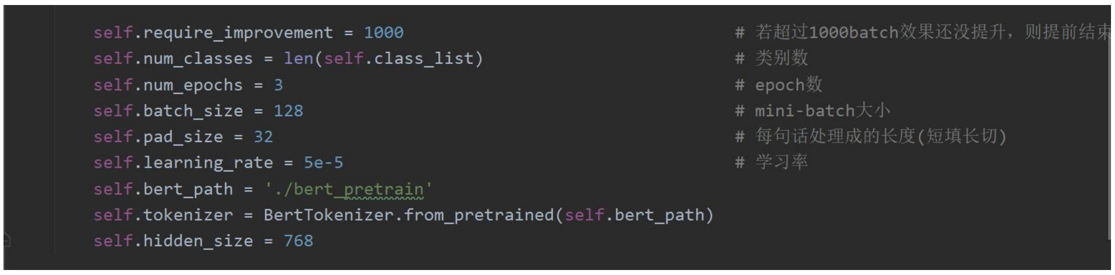
BERT 模型

采用 bert 预训练模型进行训练和预测。

bert 参数:

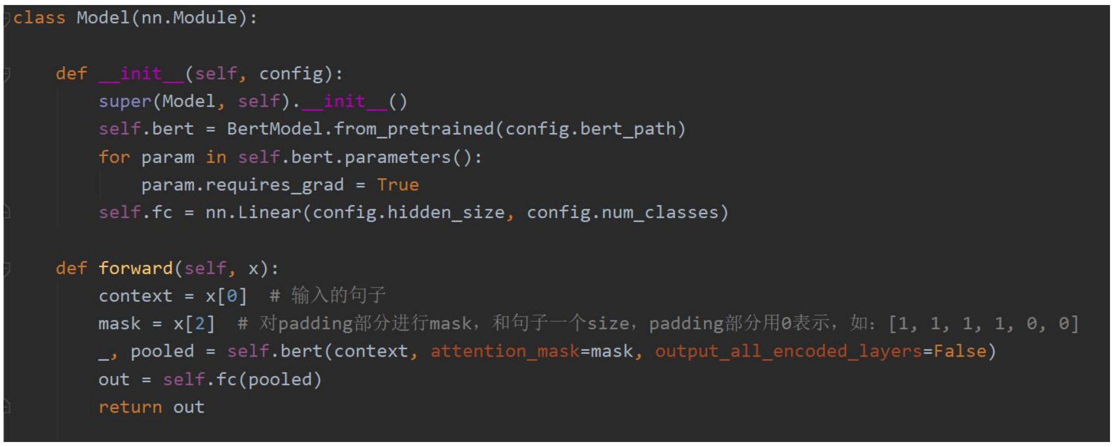
```
{
  "attention_probs_dropout_prob": 0.1,
  "directionality": "bidi",
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pooler_fc_size": 768,
  "pooler_num_attention_heads": 12,
  "pooler_num_fc_layers": 3,
  "pooler_size_per_head": 128,
  "pooler_type": "first_token_transform",
  "type_vocab_size": 2,
  "vocab_size": 21128
}
```

模型超参数设置



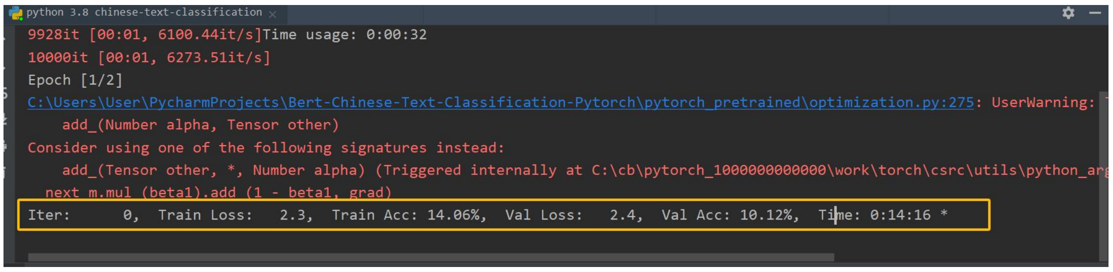
```
self.require_improvement = 1000                                  # 若超过1000batch效果还没提升，则提前结束
self.num_classes = len(self.class_list)                          # 类别数
self.num_epochs = 3                                              # epoch数
self.batch_size = 128                                            # mini-batch大小
self.pad_size = 32                                               # 每句话处理成的长度(短填长切)
self.learning_rate = 5e-5                                        # 学习率
self.bert_path = './bert_pretrain'
self.tokenizer = BertTokenizer.from_pretrained(self.bert_path)
self.hidden_size = 768
```

模型



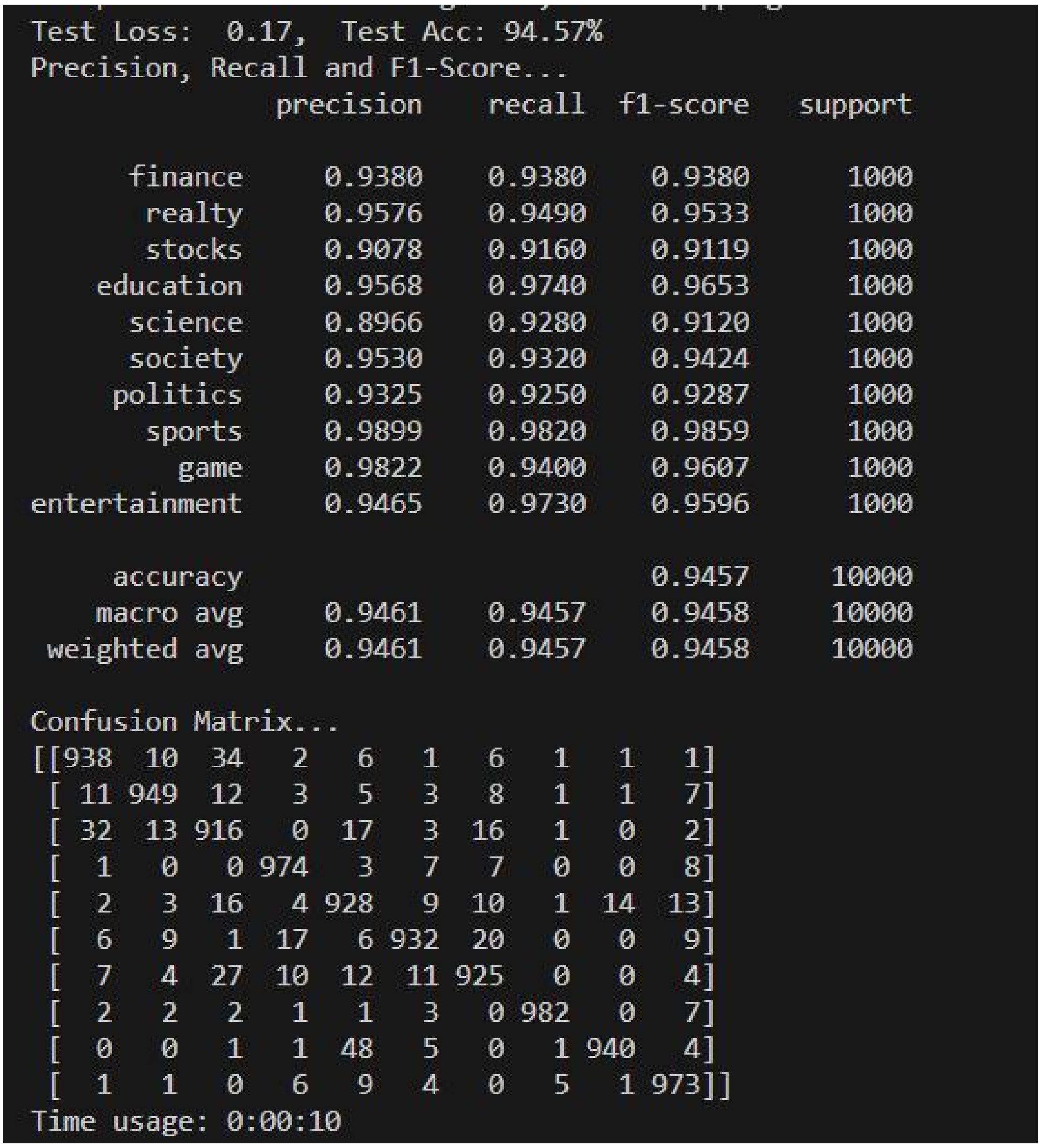
```
class Model(nn.Module):

    def __init__(self, config):
        super(Model, self).__init__()
        self.bert = BertModel.from_pretrained(config.bert_path)
        for param in self.bert.parameters():
            param.requires_grad = True
        self.fc = nn.Linear(config.hidden_size, config.num_classes)

    def forward(self, x):
        context = x[0]  # 输入的句子
        mask = x[2]  # 对padding部分进行mask，和句子一个size，padding部分用0表示，如：[1, 1, 1, 1, 0, 0]
        _, pooled = self.bert(context, attention_mask=mask, output_all_encoded_layers=False)
        out = self.fc(pooled)
        return out
```

训练：



```
9928it [00:01, 6100.44it/s]Time usage: 0:00:32
10000it [00:01, 6273.51it/s]
Epoch [1/2]
C:\Users\User\PycharmProjects\Bert-Chinese-Text-Classification-Pytorch\pytorch_pretrained\optimization.py:275: UserWarning:
    add_(Number alpha, Tensor other)
Consider using one of the following signatures instead:
    add_(Tensor other, *, Number alpha) (Triggered internally at C:\cb\pytorch_1000000000000\work\torch\csrc\utils\python_ar
  next_m.mul_(beta1).add_(1 - beta1, grad)
Iter:      0,  Train Loss:    2.3,  Train Acc: 14.06%,  Val Loss:    2.4,  Val Acc: 10.12%,  Time: 0:14:16 *
```

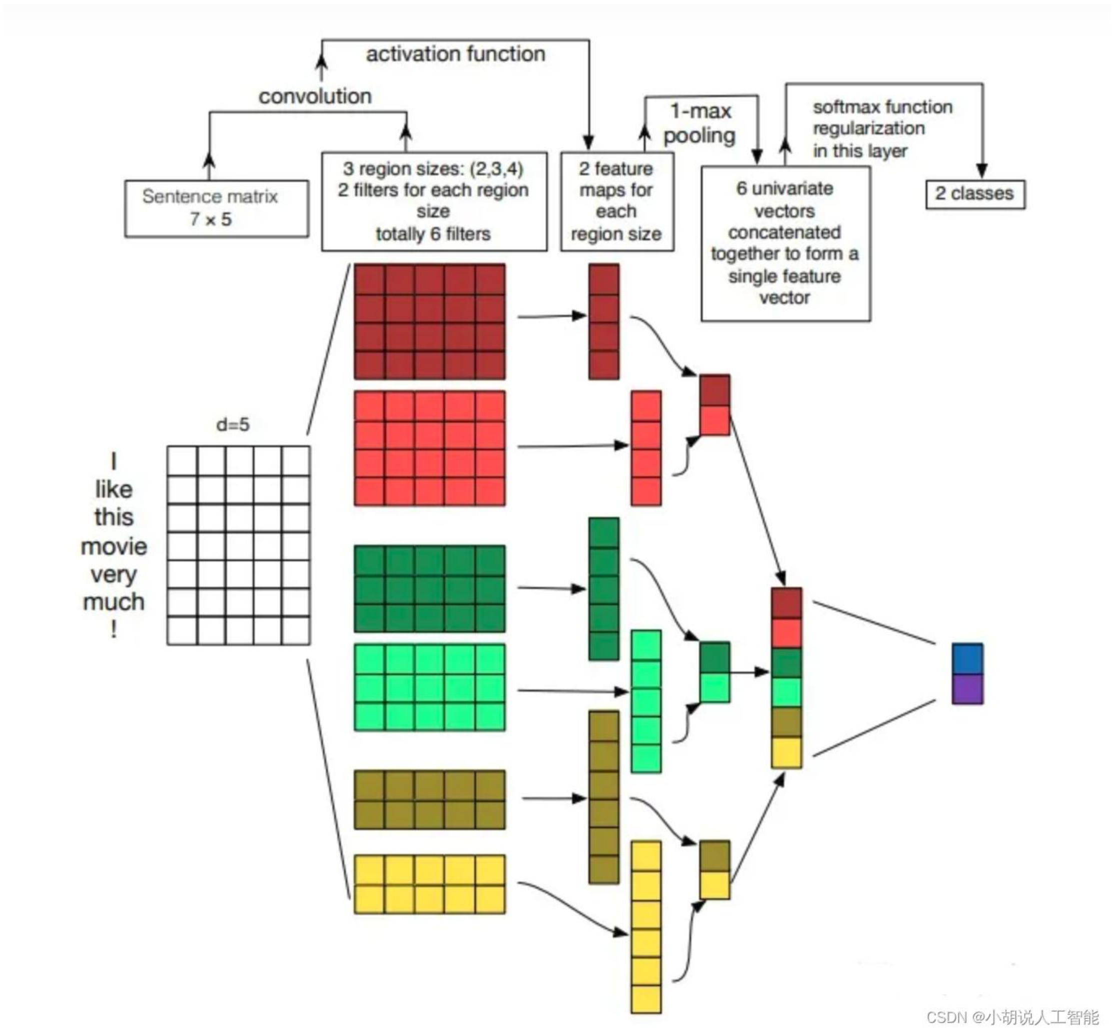
结果：准确率在 95%左右。



```
Test Loss:  0.17,  Test Acc: 94.57%
Precision, Recall and F1-Score...
               precision    recall  f1-score   support

      finance     0.9380    0.9380    0.9380      1000
       realty     0.9576    0.9490    0.9533      1000
       stocks     0.9078    0.9160    0.9119      1000
    education     0.9568    0.9740    0.9653      1000
      science     0.8966    0.9280    0.9120      1000
      society     0.9530    0.9320    0.9424      1000
     politics     0.9325    0.9250    0.9287      1000
       sports     0.9899    0.9820    0.9859      1000
         game     0.9822    0.9400    0.9607      1000
entertainment     0.9465    0.9730    0.9596      1000

     accuracy                         0.9457     10000
    macro avg     0.9461    0.9457    0.9458     10000
 weighted avg     0.9461    0.9457    0.9458     10000

Confusion Matrix...
[[938  10  34   2   6   1   6   1   1   1]
 [ 11 949  12   3   5   3   8   1   1   7]
 [ 32  13 916   0  17   3  16   1   0   2]
 [  1   0   0 974   3   7   7   0   0   8]
 [  2   3  16   4 928   9  10   1  14  13]
 [  6   9   1  17   6 932  20   0   0   9]
 [  7   4  27  10  12  11 925   0   0   4]
 [  2   2   2   1   1   3   0 982   0   7]
 [  0   0   1   1  48   5   0   1 940   4]
 [  1   1   0   6   9   4   0   5   1 973]]
Time usage: 0:00:10
```

TextCNN 模型

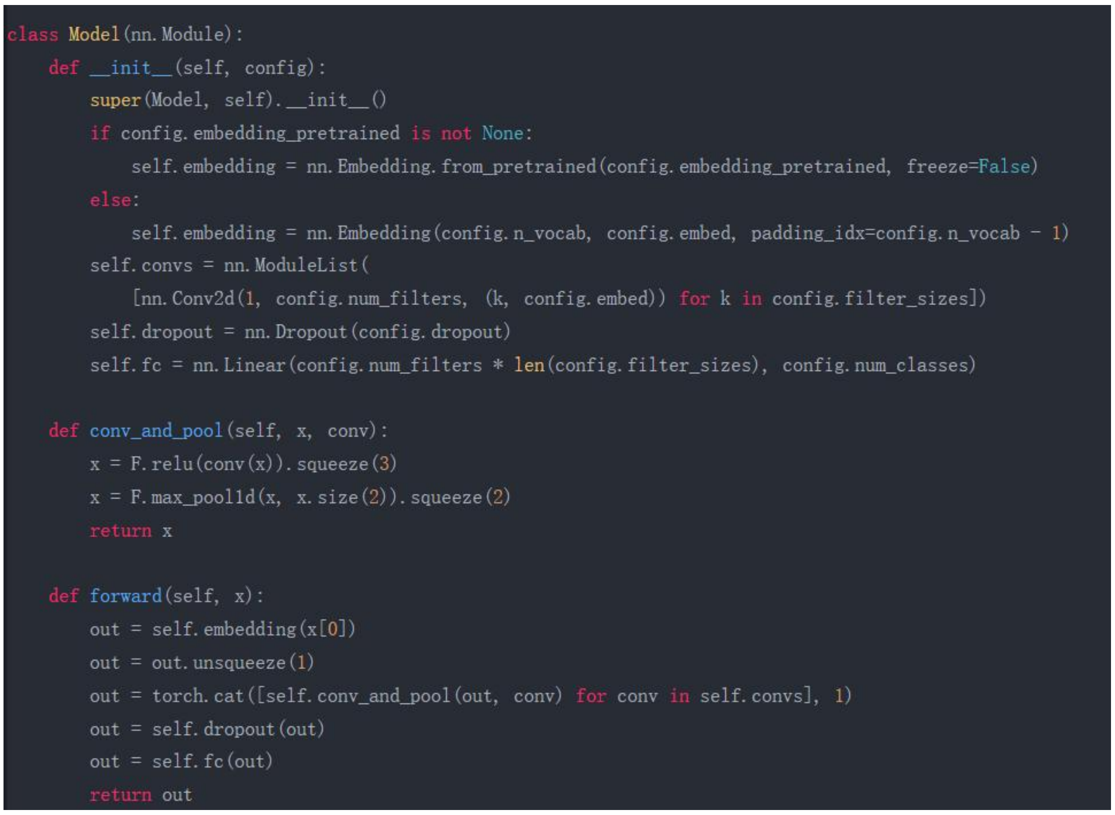
模型原理展示：


activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I like this movie very much !
CSDN @小胡说人工智能

模型初始化参数定义：

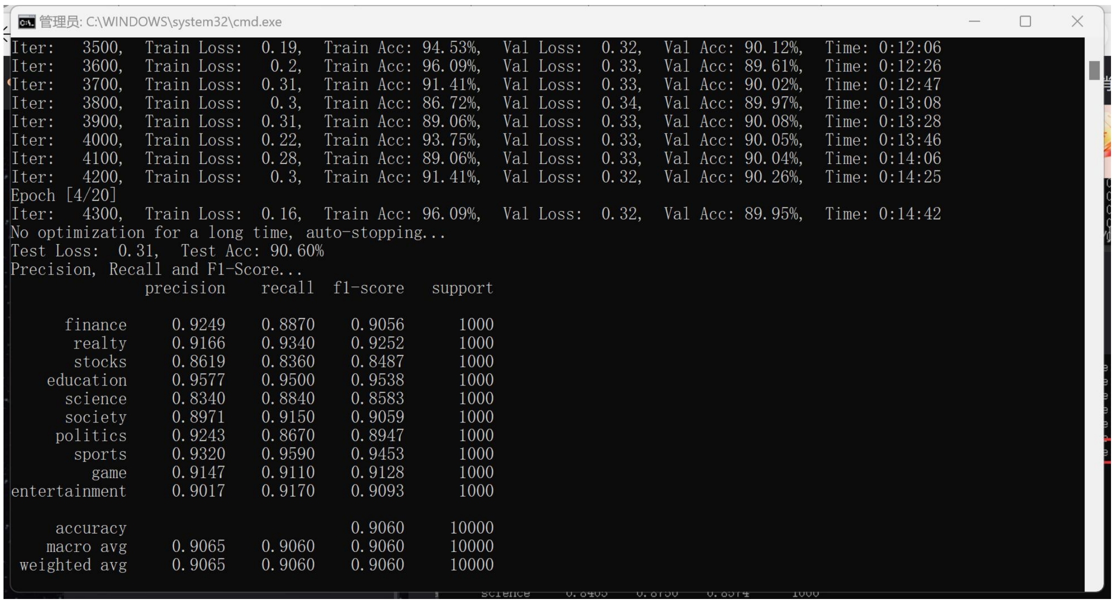
```python
class Config(object):

    """配置参数"""
    def __init__(self, dataset, embedding):
        self.model_name = 'TextCNN'
        self.train_path = dataset + '/data/train.txt'                                # 训练集
        self.dev_path= dataset + '/data/dev.txt'                                   # 验证集
        self.test_path = dataset + '/data/test.txt'                                  # 测试集
        self.class_list = [x.strip() for x in open(dataset + '/data/class.txt', encoding='utf-8').readlines()]              # 类别名单
        self.vocab_path=dataset+'/data/vocab.pkl'                                   # 词表
        self.save_path = dataset + '/saved_dict/' + self.model_name + '.ckpt'  # 模型训练结果
        self.log_path = dataset + '/log/' + self.model_name
        self.embedding_pretrained = torch.tensor(
            np.load(dataset + '/data/' + embedding)["embeddings"].astype('float32'))\
            if embedding != 'random' else None                                       # 预训练词向量
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')   # 设备

        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)
```

经过 embedding 层、三个卷积层、dropout 层、全连接层，模型构建如下



```
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):
        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
```

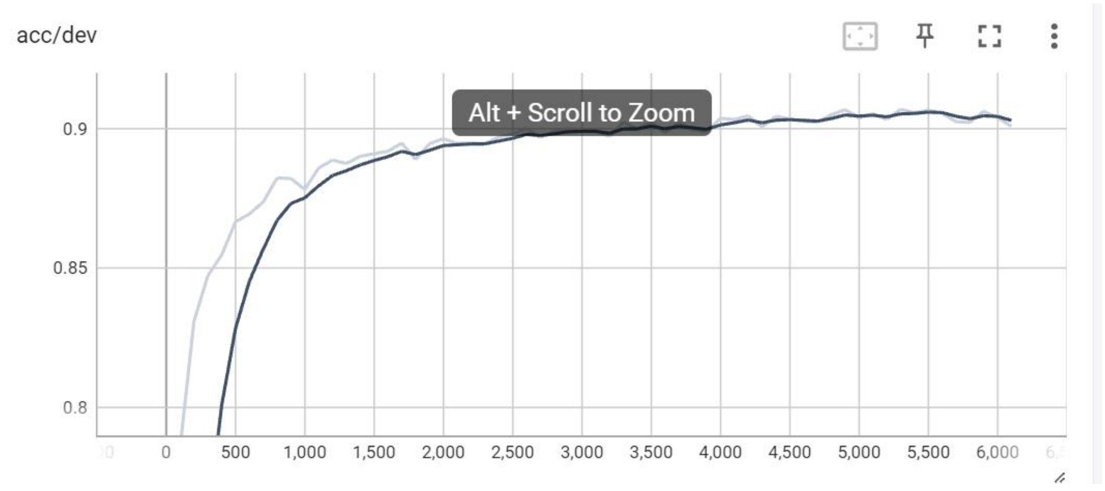
最终输出新闻的具体分类，分类准确率在 90%左右。

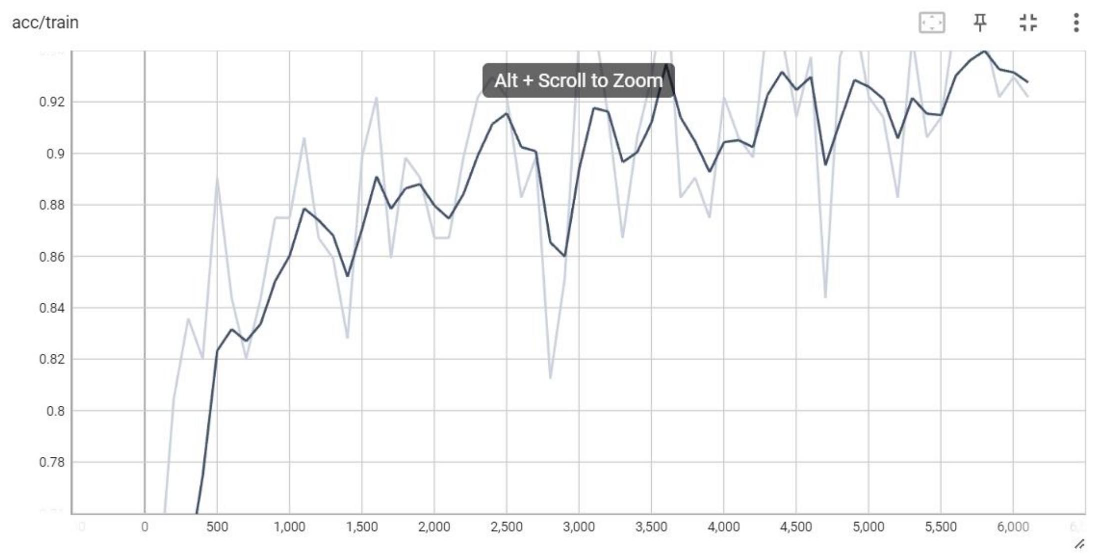


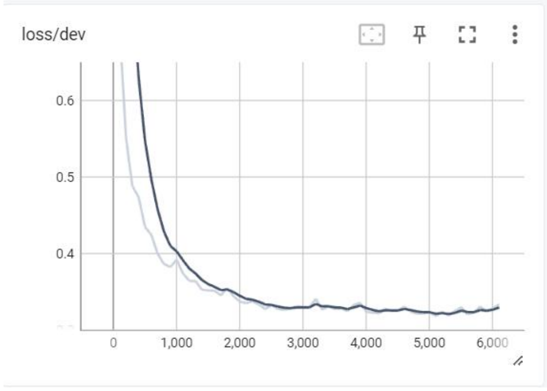
管理员: C:\WINDOWS\system32\cmd.exe

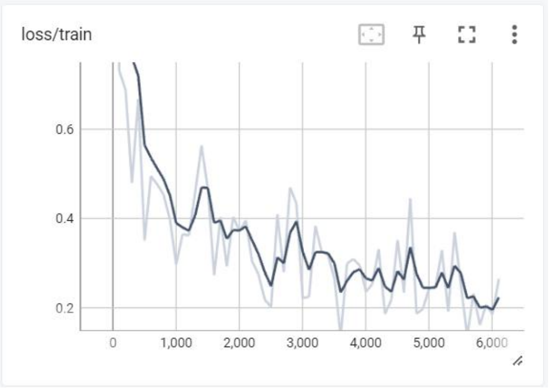
```
Iter:   3500,  Train Loss:  0.19,  Train Acc: 94.53%,  Val Loss:  0.32,  Val Acc: 90.12%,  Time: 0:12:06
Iter:   3600,  Train Loss:   0.2,  Train Acc: 96.09%,  Val Loss:  0.33,  Val Acc: 89.61%,  Time: 0:12:26
Iter:   3700,  Train Loss:  0.31,  Train Acc: 91.41%,  Val Loss:  0.33,  Val Acc: 90.02%,  Time: 0:12:47
Iter:   3800,  Train Loss:   0.3,  Train Acc: 86.72%,  Val Loss:  0.34,  Val Acc: 89.97%,  Time: 0:13:08
Iter:   3900,  Train Loss:  0.31,  Train Acc: 89.06%,  Val Loss:  0.33,  Val Acc: 90.08%,  Time: 0:13:28
Iter:   4000,  Train Loss:  0.22,  Train Acc: 93.75%,  Val Loss:  0.33,  Val Acc: 90.05%,  Time: 0:13:46
Iter:   4100,  Train Loss:  0.28,  Train Acc: 89.06%,  Val Loss:  0.33,  Val Acc: 90.04%,  Time: 0:14:06
Iter:   4200,  Train Loss:   0.3,  Train Acc: 91.41%,  Val Loss:  0.32,  Val Acc: 90.26%,  Time: 0:14:25
Epoch [4/20]
Iter:   4300,  Train Loss:  0.16,  Train Acc: 96.09%,  Val Loss:  0.32,  Val Acc: 89.95%,  Time: 0:14:42
No optimization for a long time, auto-stopping...
Test Loss:  0.31,  Test Acc: 90.60%
Precision, Recall and F1-Score...
               precision    recall  f1-score   support

      finance     0.9249    0.8870    0.9056      1000
       realty     0.9166    0.9340    0.9252      1000
       stocks     0.8619    0.8360    0.8487      1000
    education     0.9577    0.9500    0.9538      1000
      science     0.8340    0.8840    0.8583      1000
      society     0.8971    0.9150    0.9059      1000
     politics     0.9243    0.8670    0.8947      1000
       sports     0.9320    0.9590    0.9453      1000
         game     0.9147    0.9110    0.9128      1000
entertainment     0.9017    0.9170    0.9093      1000

     accuracy                         0.9060     10000
    macro avg     0.9065    0.9060    0.9060     10000
 weighted avg     0.9065    0.9060    0.9060     10000
```

训练效果图：


acc/dev
Alt + Scroll to Zoom
0.9
0.85
0.8
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000


acc/train
Alt + Scroll to Zoom
0.92
0.9
0.88
0.86
0.84
0.82
0.8
0.78
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
5,000
5,500
6,000


loss/dev
0.6
0.5
0.4
0
1,000
2,000
3,000
4,000
5,000
6,000


loss/train
0.6
0.4
0.2
0
1,000
2,000
3,000
4,000
5,000
6,000