

基于TextCNN的文本分类任务

汇报人：程 征

内容概要

- 词表映射
- 词向量层
- 数据处理
- 模型构建
- 模型训练

词表映射

无论是使用深度学习，还是传统的机器学习方法处理自然语言，首先都需要将输入的语言符号，通常为标记（Token）映射为大于等于0、小于词表大小的整数，该整数也被称作一个标记的索引值或下标。本实验将通过以下词表实现标记与索引之间的相互映射。完整的代码如下：



```
from collections import defaultdict #当字典里的key不存在但被查找时，返回的不是keyError而是一个默认值

class Vocab:
    def __init__(self, tokens=None):
        self.idx_to_token = list() #词表
        self.token_to_idx = dict() #词表及对应单词位置

    if tokens is not None:
        if "<unk>" not in tokens:
            tokens = tokens + ["<unk>"]
        for token in tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1 #标记每个单词的位置
        self.unk = self.token_to_idx['<unk>'] #开始符号的位置
```

词表映射

```
@classmethod
#不需要实例化, 直接类名.方法名()来调用 不需要self参数, 但第一个参数需要是表示自身类的cls参数,
#因为持有cls参数, 可以来调用类的属性, 类的方法, 实例化对象等
def build(cls, text, min_freq=1, reserved_tokens=None):
    token_freqs = defaultdict(int)
    for sentence in text:
        for token in sentence:
            token_freqs[token] += 1
    uniq_tokens = ["<unk>"] + (reserved_tokens if reserved_tokens else [])
    uniq_tokens += [token for token, freq in token_freqs.items() \
                    if freq >= min_freq and token != "<unk>"]
    return cls(uniq_tokens)

def __len__(self):
    #返回词表的大小, 即词表中有多少个互不相同的标记
    return len(self.idx_to_token)
def __getitem__(self, token):
    #查找输入标记对应的索引值, 如果该标记不存在, 则返回标记<unk>的索引值 (0)
    return self.token_to_idx.get(token, self.unk)
def convert_tokens_to_ids(self, tokens):
    #查找一系列输入标记对应的索引值
    return [self[token] for token in tokens]
def convert_ids_to_tokens(self, indices):
    #查找一系列索引值对应的标记
    return [self.idx_to_token[index] for index in indices]
```

词向量层

在使用深度学习进行自然语言处理时，将一个词（或者标记）转换成一个低维、稠密、连续的词向量（也称Embedding）是一种基本的词表示方法，通过torch.nn包提供的Embedding层即可实现该功能。创建Embedding对象时，需要提供两个参数，分别是num_embeddings，即词表的大小；以及embedding_dim，即Embedding向量的维度。调用该对象实现的功能是将输入的整数张量中的每个整数（通过词表映射功能获得标记对应的整数）映射为相应维度（embedding_dim）的张量。如下面的例子所示：



```
# 创建一个词表大小为8, Embedding向量维度为3的Embedding对象
embedding = nn.Embedding(num_embeddings=8, embedding_dim=3)

inputs = torch.tensor([[0,1,2,1],[4,6,6,7]], dtype=torch.long)
inputs.shape
```

```
[10]: torch.Size([2, 4])
```

```
outputs = embedding(inputs)
outputs
outputs.shape
```

```
tensor([[[[-2.0448,  0.3243, -0.0095],
          [ 0.6108,  0.5181,  0.4062],
          [-0.1027, -1.2090, -0.5231],
          [ 0.6108,  0.5181,  0.4062]],

        [[ 0.0365,  1.0829, -0.4578],
          [-1.1569, -0.1114, -0.2731],
          [-1.1569, -0.1114, -0.2731],
          [ 1.5142,  1.8929, -1.7223]]], grad_fn=<EmbeddingBackward0>)
torch.Size([2, 4, 3])
```

数据处理——数据获取与预处理



```
def data_access(filepath):  
    """数据获取与预处理"""  
    raw_iter = pd.read_csv(filepath)  
    # raw_iter['label'].value_counts()  
    data = []  
    for raw in raw_iter.values:  
        label, s = raw[-1], raw[1] # 标签和文本  
        s = re.sub(r'http://.*?(\s|$)', '<URL>+' + '\\1', s)  
        s = re.sub(r'@.*?(\s|$)', '<@ID>+' + '\\1', s)  
        s = re.sub('\u200B', '', s)  
        s = HanziConv.toSimplified(s.strip())  
        data.append((s, label))  
    return data  
  
filepath = '../input/fakenews/fake_news.csv'  
data = data_access(filepath)  
data[:3]
```

[13]: [('#上海瑞金医院持刀伤人者被警方开枪制服#有什么事可以好好商量，为啥要做过激事情？持刀伤人是犯法的，自己的问题不但解决不了，还把自己搭进去，幸好警察处置及时，否则定要出大乱子！吃一堑长一智吧！<URL> <URL>',
0),
(('清晨日出🌄2022年7月10日 星期日 农历六月十二日1、高考补录今日起开始报名：计划招录5182人，其中3800余个计划招录应届毕业生。
2、上海瑞金医院一男子持刀伤人致4人受伤，民警开枪将其击伤制服。3、因虚假账户等问题，马斯克正式宣布终止收购推特，推特：法庭见。#
新闻[超话]##早安##早安,新的一天##新闻看点#',
0),
(('我不会去蹭，上海瑞金医院流量，也不会去蹭，小日本JP安培流量！.....一切一切，都有因果报应轮回！.....你吃烧烤被打了，，那为什么不打别人，为什么砍你？你说我是陌生人不认识对方，（说明你命里有一劫，前世作恶多端）...为什么枪击安倍，.....反正都是死，不如多开机枪扫射，为什么没有？说明现场人都是命硬的人，前世今生修来的福？自己品品吧！.....别一天涨停板涨停板的，，烦死人了',
0)]

数据处理 —— 数据集划分



```
torch.manual_seed(1234)
# 数据集划分 确定训练集、测试集大小，以8: 2划分训练集和测试集
num_train = int(len(data) * 0.8)
num_test = len(data) - num_train
train_data, test_data = random_split(data, [num_train, num_test])
```

+ Code

+ Markdown

[15]:

```
def sample_count(data_total, data_splited):
    """观察划分后的训练集和测试集中正负样本的比例"""
    pos = 0
    for index in data_splited.indices:
        s, l = data_total[index]
        if int(l) == 1:
            pos += 1
    neg = len(data_splited) - pos
    return pos, neg

train_pos, train_neg = sample_count(data, train_data)
test_pos, test_neg = sample_count(data, test_data)
print(f'训练集中正样本为{train_pos}条，负样本为{train_neg}条。')
print(f'测试集中正样本为{test_pos}条，负样本为{test_neg}条。')
```

训练集中正样本为570条，负样本为754条。
测试集中正样本为150条，负样本为182条。

数据处理 —— 词表映射



```
# 根据训练集进行词表映射
train_sentences = [s for s,l in train_data]
vocab = Vocab.build(train_sentences)
vocab
```

[19]: <__main__.Vocab at 0x7f1bb128f950>



```
# 词典属性
vocab.idx_to_token # ['<unk>', '希', '望', '大', '家', ...]
vocab.token_to_idx # {'<unk>': 0, '希': 1, '望': 2, '大': 3, '家': 4, ...}
# 词典方法
vocab.convert_tokens_to_ids(['希', '望'])
vocab.convert_ids_to_tokens([1,2])
```

[26]: [1, 2]

[26]: ['希', '望']



```
# 词表映射
train_data = [(vocab.convert_tokens_to_ids(list(sentence)), label) for sentence,label in train_data]
test_data = [(vocab.convert_tokens_to_ids(list(sentence)), label) for sentence,label in test_data]
```


数据处理 —— 批次内数据整理



```
class FakeNewsDataset(Dataset):
    def __init__(self, dataset):
        self.dataset = dataset
        self.lens = len(dataset)
    def __getitem__(self, index):
        sen, label = self.dataset[index]
        return sen, label
    def __len__(self):
        return len(self.dataset)

def collate_fn_textcnn(examples):
    """ 对一个批次内的数据进行处理 """
    inputs = [torch.tensor(ex[0]) for ex in examples]
    trargets = torch.tensor([ex[1] for ex in examples], dtype=torch.long)
    # 对批次内的样本进行补齐, 使其具有相同长度
    inputs = pad_sequence(inputs, batch_first=True)
    return inputs, trargets
```

补充知识：pad_sequence 批次内序列补齐



```
import torch
from torch.nn.utils.rnn import pad_sequence

a = torch.tensor([1,2,3])
b = torch.tensor([1,2,3,4])
c = torch.tensor([1,2,3,4,5])
l = [a,b,c]
```

[4]: [tensor([1, 2, 3]), tensor([1, 2, 3, 4]), tensor([1, 2, 3, 4, 5])]

```
pad_sequence(l)
```

```
tensor([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3],
        [0, 4, 4],
        [0, 0, 5]])
```

```
pad_sequence(l, batch_first=True)
```

```
tensor([[1, 2, 3, 0, 0],
        [1, 2, 3, 4, 0],
        [1, 2, 3, 4, 5]])
```

补充知识：permute 交换tensor的维度



```
# .permute  
t = torch.rand(2,3,4)  
t  
t.shape  
t.permute(2,0,1).shape
```

```
[11]: tensor([[[[0.3440, 0.1259, 0.5299, 0.0142],  
               [0.2827, 0.3576, 0.8928, 0.8003],  
               [0.6631, 0.0635, 0.3075, 0.8089]],  
              [[0.1151, 0.9474, 0.6773, 0.7316],  
               [0.8888, 0.3237, 0.7087, 0.9710],  
               [0.0575, 0.0704, 0.9196, 0.4396]]]])  
[11]: torch.Size([2, 3, 4])  
[11]: torch.Size([4, 2, 3])
```

上图中变量t共三个维度，分别为0,1,2，使用permute方法进行维度交换时，仅需指定对应维度的顺序即可。

模型构建



```
class TextCNN(nn.Module):
    ...

    vocab_size: 词表大小
    embedding_dim: 经过embedding转换后词向量的维度
    filter_size: 卷积核的大小
    num_filter: 卷积核的个数
    num_class: 类别数
    ...

    def __init__(self, vocab_size, embedding_dim, filter_size, num_filter, num_class):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # padding=1 表示在卷积操作之前, 将序列的前后各补充1个输入, 这里没有找到详细的解释, 考虑是为了让卷积核充分学习序列的信息
        self.conv1d = nn.Conv1d(embedding_dim, num_filter, filter_size, padding=1)
        self.activate = F.relu
        self.linear = nn.Linear(num_filter, num_class)

    def forward(self, inputs):
        embedding = self.embedding(inputs) # Embedding层

        convolution = self.activate(self.conv1d(embedding.permute(0, 2, 1))) # 卷积层

        pooling = F.max_pool1d(convolution, kernel_size=convolution.shape[2]) # 池化层聚合

        probs = self.linear(pooling.squeeze(dim=2)) # 全连接层
        return probs

# 模型实例化
model = TextCNN(vocab_size=len(vocab), embedding_dim=128, filter_size=3, num_filter=100, num_class=2)
```





模型训练


▷


```
for epoch in range(config['num_epochs']):
    total_loss = 0
    total_acc = 0
    for batch in tqdm(train_loader, desc=f'Training Epoch {epoch+1}'):
        # 将数据加载至GPU
        inputs, targets = [x.to(config['device']) for x in batch]
        # 将特征带入到模型
        probs = model(inputs)
        # 计算损失
        loss = criterion(probs, targets)
        optimizer.zero_grad() # 梯度清零
        loss.backward() # 反向传播
        optimizer.step() # 更新参数
        acc = (probs.argmax(dim=1) == targets).sum().item() # item()用于在只包含一个元素的tensor中提取值
        total_acc += acc # 最终得到整个epoch的准确率
        total_loss += loss.item() # 最终得到整个epoch的损失

# 打印的是整个epoch上的样本损失的平均值以及准确率
print(f'\tTrain Loss:{total_loss/len(train_loader):.4f}\tTrain Accuracy:{total_acc/len(train_dataset):.4f}')
```

Training Epoch 1: 100%  42/42 [00:00<00:00, 139.91it/s]
Train Loss:0.5302 Train Accuracy:0.7719

Training Epoch 2: 100%  42/42 [00:00<00:00, 156.02it/s]
Train Loss:0.2985 Train Accuracy:0.9215

Training Epoch 3: 100%  42/42 [00:00<00:00, 156.43it/s]
Train Loss:0.1714 Train Accuracy:0.9675

Training Epoch 4: 100%  42/42 [00:00<00:00, 154.95it/s]

详细训练方法与手写数字识别案例中训练方法基本一致，这里不在赘述。