1 改进后的 TextCNN

1.1 解题思路

TextCNN 模型在进行整句的情感倾向性分类时具有显著性的优点，其模型简单，能够有效提取文本特征，可以比较高效的识别出句子情感。

在进行面向 Aspect 的情感倾向性分类时，问题由分类问题变为了序列标注问题，需要采取一些手段将其转化为正常的多分类问题。对数据集进行一定的处理，让输入变为句子 + 实体单词的组合，并使句子能够参杂对应要分析情感的实体单词的有效信息，使其能被 CNN 网络认出。

TextCNN 结构

1. 词嵌入层：对于输入的句子，其经过 Embedding 之后为一个

$$[setence\_len, embedding\_dim]$$

大小的矩阵。

2. 卷积层：与在计算机视觉（CV）中常规 CNN 的卷积核不同。在 CV 中，卷积核往往都是正方形的，比如 3*3 的卷积核，然后卷积核在整张 image 上沿高和宽按步长移动进行卷积操作。而在 NLP 中卷积核的宽和句子矩阵的宽相同，该宽度即为词向量大小，且卷积核只会在高度方向移动。因为对句子矩阵进行横向的卷积是没有什么意义的，窗口横向滑动都只会获得同一个单词的词向量的一部分。这样的卷积操作结束后，对于每一个卷积核，其得到了一个长度为

$$[sentence\_len - filter\_window\_size + 1]$$

的列向量。

3. 一维最大池化层：由于句子长度和卷积核宽度各有所不同，最后输出的向量长度也会不同，利用一个一维的最大池化层，取每一个卷积结果的最大值并拼凑出新的向量，这个向量的长度是可以确定的，为卷积核的数量。

4. dropout 层与全连接层：因为我们要执行分类任务，最后接上一个全连接层，并用 softmax 激活函数输出每个类别的概率。

以输入句子 “wait for the video and don’t rent it”、卷积核数量为 4 的情况为例，如下图所示，词向量组成的矩阵在四个不同的卷积核作用下得到四个长度不尽相同的输出向量，分别经过最大池化层处理后拼接成一个长度为 4 的向量，经过全连接层连接到长度为 2 的输出层，从而进行二分类判断。全连接层可以有 dropout 处理。

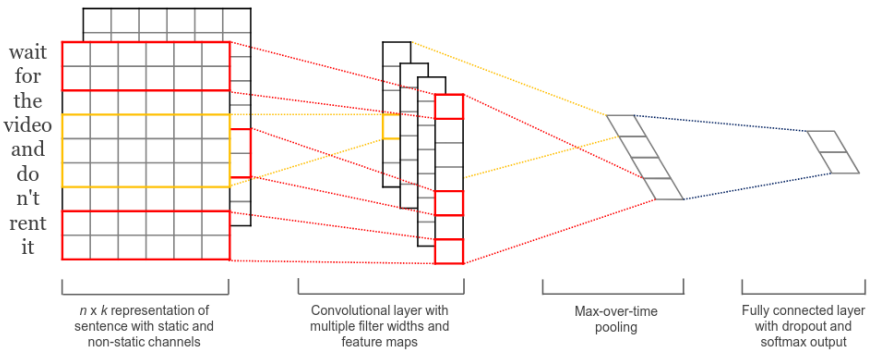

图 1: TextCNN 结构图

问题转化

上述的 TextCNN 结构只能处理给定句子分析整个句子的情感分类问题，要处理面向 aspect 的情感倾向性分类问题，需要将之转化为常规的句子情感分类问题，因此我需要尝试将实体信息融入输入当中，与数据集中的格式不太相同，每个实体均要对应一个输入，这个输入是对实体所在句子做出一些变动得到的结果。

这里我想到了四种方案：

1. 复制实体词：对于一个句子和其中的实体词，将实体词在原位置复制多份，从而让 CNN 可以有效的识别到这一信息，比如句子“Even as Trump was hit by bullets, he was still shouting justice”，其中的一个实体词为“Trump”，将句子改为“Even as Trump Trump Trump Trump was hit by bullets, he was still shouting justice”后作为输入。这样“Trump”所对应的词向量会在句子矩阵中占有比较大的分量。

2. 为实体词前后添加标签：对于句子“Even as Trump was hit by bullets, he was still shouting justice”，将句子改为“Even as <attention> Trump <attention> was hit by bullets, he was still shouting justice”后作为输入。当构建词向量的层参与训练时，词“<attention>”会被逐渐赋予比较大的分量，从而达到突出实体词的作用。

3. 将实体词额外放在句子指定位置：上述句子被改为“Trump : Even as Trump was hit by bullets, he was still shouting justice”，CNN 天生带有一部分类似于注意力机制的效果，这样可以尝试让 CNN 注意到实体词。

4. 为实体词的词向量额外加上一个特殊的词向量：在将上述句子转化为词向量构成的矩阵后，对实体词“Trump”所对应的词向量所在位置额外加上一个词“<attention>”所对应的词向量，当构建词向量的层参与训练时，词“<attention>”会被逐渐赋予比较大的分量，从而达到突出实体词的作用。

4 种方案均尝试过后，我选择了方案 4。方案一与方案四收敛后均可以达到比较好的分类效果，而方案四具有更快的收敛速度。对于方案二，其效果没有想象中的好，原因可能是 CNN 过于注重周围词对“<attention>”的影响，反而疏忽了实体词。方案三效果很差，原因可能是虽然 CNN 具有一定的局部注意力，但是缺少全局注意力，最后仍然演变成了整句情感分类。

1.2 关键代码分析

虽然 TextCNN 在在预训练好词向量的情况下才能发挥出较好的水准，但是那是对于整句分类而言，因为我们要解决面向 aspect 的句子情感分类，因此我打算自己训练词嵌入层，从而让 CNN 尽可能注意到实体信息。

- **1. 数据预处理**

首先对数据集进行处理，数据集中每一行分别包括了句子中的一个词、它的实体标签（O、B-ASP、I-ASP 三种）、它的情感标签（0、1、2 分别代表消极、中立、积极）。首先将它们三个按句子提炼出来，并调用 load_data 方法对每个实体词（实体标签非‘O’）构建一个由（句子、实体位置、实体的情感标签）这样的三元组组成的数据。

```
class AtepcDataset(Dataset):
    def __init__(self, data_path):
        with  open(data_path, 'r', encoding='utf-8') as f:
            setences = []
            aspects = []
            labels = []

            setence = []
            aspect = []
            label = []
            for line in f:
```

```
            if line == '\n':
                setences.append(setence)
                aspects.append(aspect)
                labels.append(label)
                setence = []
                aspect = []
                label = []
                continue
            word = line.split(' ')
            assert  len(word) == 3
            setence.append(word[0])
            vocab.add(word[0])
            aspect.append(word[1])
            label.append( int(word[2]))

        self.data = self.load_data(setences, aspects, labels)

    def load_data(self, setences, aspects, labels):
        data = []
        for sentence, aspect, label in  zip(setences, aspects, labels):
            for index, asp in  enumerate(aspect):
                if asp == 'O':
                    continue
                if label[index] == -1:
                    continue
                data.append((sentence, index, label[index]))
        return data

    def __len__(self):
        return  len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

在读取数据集之后，我们得到了一个单词表 vocab，并将其按索引值得到一个字典，这样每个词与一个唯一的数字标签对应起来，从而能够通过词嵌入层构建词向量。

在 DataLoader 中，将同一批次的所有句子扩充到相同长度（采用一个特殊的扩充词 “<pad>” 进行扩充），并将每个句子按字典转化为一个一维张量并同批次拼接。

```
MIN_PADDING_LENGTH = 16
def collate_fn(batch):
    sentence_lengths = [ len(sentence) for sentence, _, _ in batch]
    max_sentence_length =  max( max(sentence_lengths), MIN_PADDING_LENGTH)
    sentence_tensor = torch.zeros(( len(batch), max_sentence_length), dtype=torch. long)
    aspect_tensor = torch.zeros(( len(batch)), dtype=torch. long)
    label_tensor = torch.zeros(( len(batch)), dtype=torch. long)

    for i, (sentence, index, label) in  enumerate(batch):
        sentence_tensor[i, :
            len(sentence)] = torch.tensor([vocab_index[word] for word in sentence], dtype=torch. long)
        sentence_tensor[i,  len(sentence):] = torch.tensor([vocab_index[PAD]] * (max_sentence_length -
```

```
            len(sentence)), dtype=torch. long)
        aspect_tensor[i] = index
        label_tensor[i] = label

    return sentence_tensor, aspect_tensor, label_tensor

BATCH_SIZE = 128

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=collate_fn)
```

- **2. TextCNN**

卷积层中采用多个不同宽度的卷积核，卷积核宽度越大，提取到目标词周围其他词信息的范围越大，同时也更容易受到无关因素影响。

每个大小卷积核设置多个输出通道（这里通道的实际作用其实是同样大小的多个不同的卷积核，并非常规意义上 CNN 中的通道的作用，理论上只是多个相同大小的卷积核，实际输出通道数应该均为 1）来试图做到提取不同因素所产生的影响。

```
filter_sizes = [3, 5, 7, 9]
out_channels = [128, 64, 32, 16]
assert  len(filter_sizes) ==  len(out_channels)
EMBEDDING_DIM = 256

class TextCNN(nn.Module):
    def __init__(self, vocab_size):
         super(TextCNN, self).__init__()
        # 词嵌入层
        self.embedding = nn.Embedding(vocab_size + 1, EMBEDDING_DIM)
        # 卷积层
        self.convs = nn.ModuleList([nn.Conv2d(1, c, (k, EMBEDDING_DIM)) for c, k in
            zip(out_channels, filter_sizes)])
        # 随机失活层
        self.dropout = nn.Dropout(0.5)
        # 全连接层
        self.fc = nn.Linear( sum(out_channels), 3)

    def forward(self, x, aspect):
        # 构建词向量
        x = self.embedding(x)  # (batch_size, seq_len, embedding_dim)
        # 为实体词的词向量加上词 “<attention>” 所对应的词向量
        atte_embedding = self.embedding(ATTE_TERSOR)  # (1, embedding_dim)
        atte_embedding = atte_embedding.squeeze(0)  # (embedding_dim)
        for batch, aspect_index in  enumerate(aspect):
            x[batch, aspect_index, :] += atte_embedding
        x = x.unsqueeze(1)  # (batch_size, 1, seq_len, embedding_dim)
        # 多核多通道卷积与池化
        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs]  # [(batch_size, out_channel, seq_len -
            filter_size + 1)] * len(filter_sizes)
        x = [F.max_pool1d(filt_result, filt_result.shape[2]).squeeze(2) for filt_result in x] # [(
            batch_size, out_channel)] * len(filter_sizes)
```

```
        # 不同核不同通道结果拼接
        x = torch.cat(x, 1)  # (batch_size, len(filter_sizes) * out_channel)
        # 获得最终类别概率
        x = self.dropout(x) # (batch_size, len(filter_sizes) * out_channel)
        x = self.fc(x)  # (batch_size, 3)
        x = F.softmax(x, dim=1)  # (batch_size, 3)
        return x
```

• 3. 训练与推断

```
net = TextCNN( len(vocab)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

def train(epoch):
    net.train()
    for i, (sentence, aspect, label) in  enumerate(train_loader):
        optimizer.zero_grad()
        sentence = sentence.to(device)
        aspect = aspect.to(device)
        output = net(sentence, aspect)
        output = output.to(torch.device('cpu'))
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        if i % 10 == 0:
            print('Epoch: {}/{}, Step: {}/{}, Loss: {:.4f}'. format(epoch+1, EPOCHS, i+1,
                len(train_loader), loss.item()))
            wandb.log({'train_loss': loss.item(), 'epoch': epoch+1})

def evaluate(epoch):
    net. eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for sentence, aspect, label in test_loader:
            sentence = sentence.to(device)
            aspect = aspect.to(device)
            output = net(sentence, aspect)
            output = output.to('cpu')
            _, predicted = torch. max(output.data, 1)
            total += label.size(0)
            correct += (predicted == label). sum().item()
    accuracy = 100 * correct / total
    print('Accuracy of the network on the test set: {:.4f} %'. format(accuracy))
    wandb.log({'test_accuracy': accuracy, 'epoch': epoch+1})
    return accuracy

for epoch in  range(EPOCHS):
    train(epoch)
    accuracy = evaluate(epoch)
```

```
    if accuracy > max_accuracy:
        max_accuracy = accuracy
save_model('models/final.pth')
print('Max accuracy:', max_accuracy)
```

1.3 实验验证与实验结果

设置词嵌入维数为 256，宽度为 3、5、7、9 的卷积核各 128、64、32、16 个，dropout 比率为 0.5，batch 大小 32，迭代 20 次。

输出为 3 分类，分别有消极、中立、积极三种。

在各个测试集上得到的结果如下，由于任务为三分类，只计算准确率：

Dataset	CNN Acc
camera	88.52%
car	86.43%
laptop	60.53%
notebook	83.94%
phone	87.88%
restaurant	71.29%
twitter	64.08%
mixed	78.24%

表 1: CNN 在不同数据集上的准确率

可以看到虽然效果比在整句上的情感分类要差一些，但是效果依旧不错，证明 TextCNN 也可以胜任面向 Aspect 的情感分类任务。部分测试集上效果较差，分析原因应该是由于是现训练的词嵌入层，对于一部分数据集没有很好的构建出合适的词向量，导致训练结果较差。若采用预训练好的 word2vec 等来构建词向量，这种问题应该可以避免，并且准确率应该会较大幅度提升。

1.4 过程中遇到的问题

1. 在为实体词的词向量增加另一个词向量时，我忘记了我采用了 minibatch 机制，导致同一个 batch 内的句子矩阵会在一个 batch 内实体词出现的全部位置均加上特殊词向量，这会导致同一个 batch 之间的数据相互影响，导致收敛速度大幅度下降，收敛后的准确率较低一些（但是对比起来，收敛后竟然就只小了差不多 2%）。

2. 由于矩阵运算所需要的条件，同一 batch 内所有句子需要扩充到同一长度，我最初采用了 0 扩充，但是实际上这会与单词表中第一个单词冲突，导致出现了该单词的句子不能准确预测，后来我为扩充专门准备了一个词“<pad>”，其两边加上符号以防止与某个词冲突，改正后那些原本出问题的句子也可以正常预测了。