声音事件检测实践报告

519030910383 张哲昊

1. 摘要

本次实践项目中完成了基础部分：理解代码逻辑，了解弱监督情况下进行时间轴预测的难点，以及 baseline 的原理，同时实现了 CRNN 模型。然后完成了高阶要求：修改模型参数，深度，优化模型结果，调研音频中的数据增强方式同时进行应用。

2. 代码逻辑以及模型理解

声音事件检测任务是从给定的音频文件中判断出不同的声音在哪个时间段发生。具体来说，需要模型输出每一帧音频有哪些类别的声音。其难点在于，这是一个序列数据的多标签任务，即每一帧对应的标签可能不止一个，同时前后帧会对当前帧结果产生影响。

我们首先需要从音频文件中得到数据特征，而在 SED 任务中，我们选择的是 log mel energies。该特征能够有效地体现音频在各个时间段地能量信息。该特征的维度为 [时间步长，频率 bin 的数量]，通过 CRNN 模型之后得到维度为 [时间步长，类别总数] 的帧级别的概率。

CRNN 的模型架构如下图所示：首先将输入的特征进行 1 维的批归一化，之后经过 3 个卷积模块，每个卷积模块包括卷积层，2 维批归一化以及激活函数和最大池化层。三个卷积层的目的是提取 log mel energies 中的一些深层特征，然后将卷积得到的所有通道的 feature map 拼接起来通过 GRU 层。GRU 层的目的是利用循环神经网络来建模序列模型。由于输入的特征是时序特征，每一帧之间是存在较强的关系。通过双向的循环神经网络，可以使得模型更好地利用前后帧的信息。经过 GRU 之后再对得到的特征进行插值（也可以在过 GRU 之前做，或在最大池化操作时不改变时间维度），然后通过全连接层和 simoid 函数得到每一帧对应到每个类别的概率。然后将帧级别的概率通过 linear_softmax_pooling 进行合并与弱标签进行 Binary cross entropy 损失函数的计算。选择该损失函数的主要原因是，该任务是多分类任务，标签可能不止一个，因此不能选用常规的交叉熵损失函数。

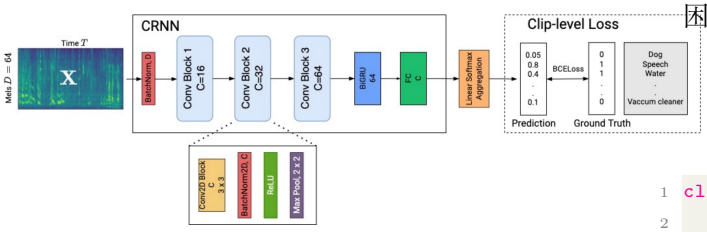

图 1: *crnn* 的架构图

3. 弱监督下的难点与 baseline 原理

由于弱监督情况下，只提供了每一段音频对应的标签，而无法得到帧级别的标签，故无法通过损失函数的梯度回传使得模型在帧级别有很好的表现。总而言之，在监督学习的范式之下，这种方式的监督信号粒度不够细，导致模型在 event-base 的测试表现不够理想。同时声音事件检测数据本身的其他问题（如声音时长不一，离接收器远近不一，多声道的互相影响，噪音）也会使得任务更加困难。baseline 原理已经在2中介绍。

4. CRNN 的实现与实验结果

```
class conv_block(nn.Module):
    def __init__(self, channel_in, channel_out,
    kernel_size=3, padding=1):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(channel_in,
                      channel_out,
                      kernel_size=kernel_size,
                      padding=padding,
                      bias=False),
            nn.BatchNorm2d(channel_out),
            nn.ReLU(),
```

```
            nn.MaxPool2d(kernel_size=(2,2)),
            )
    def forward(self, x):
        return self.block(x)

class Crnn(nn.Module):
    def __init__(self, num_freq, class_num):
        super().__init__()
        self.bn_1d = nn.BatchNorm1d(num_freq)
        self.conv_1 = conv_block(1,16)
        self.conv_2 = conv_block(16,32)
        self.conv_3 = conv_block(32,64)
        self.conv_4 = conv_block(64,128)
        self.conv_all = nn.Sequential(self.conv_1,
    self.conv_2,self.conv_3)
        self.gru = nn.GRU(512,64,num_layers=1,
    bidirectional=True,batch_first=True)
        self.linear = nn.Linear(128, class_num)

    def detection(self, x):
        #[32, 501, 64]
        t = x.shape[1]
        x = x.transpose(1, 2).contiguous()
        x = self.bn_1d(x)
        x = x.unsqueeze(1)
        x = self.conv_all(x)
        x = x.transpose(1, 3).contiguous()
        x = x.flatten(-2)
        x,_ = self.gru(x)
        x = self.linear(x)
        x = x.transpose(1, 2).contiguous()
        x = F.interpolate(x,size = (t))
        x = x.transpose(1, 2).contiguous()
        frame_wise_prob = torch.sigmoid(x).clamp(1
    e-7, 1.)
```

表 1: *baseline* 的实验效果

	f_measure	precision	recall
event base	0.116033	0.107227	0.139698
segment base	0.588161	0.626255	0.563423
tagging base	0.652802	0.743304	0.591568

5. 模型调参与发现

本次实践任务使用 wandb 库进行调参。该库能够高效清晰地 grid search （或贝叶斯优化）所指定地参数类型，只需要提前在配置文件中设定好需要超参数范围，通过 sweep 功能一次运行即可完成所有参数在给定范围内地搜索。运行结束之后能够在网页端快速清晰地查看实验结果，并可以自动汇总。本次调整地主要参数包括：循环神经网络隐层数量，Dropout，循环神经网络地层数，卷积核的大小。由于实验无法完全复现，每组配置均使用 3 个不同的随机数种子进行实验。配置文件实例如下：

```
program: run.py
method: grid
metric:
  name: event_f1
  goal: maximize
parameters:
  seed:
    values: [ 40,41,42]
  width:
    values: [ 256,512,1024]
  dropout:
    values: [ 0.3,0.4,0.5]
  num_layers:
    values: [ 1,2]
```

5.1. 其他循环神经网络替代 GRU

本次实验尝试将 GRU 更换为 LSTM。LSTM 对记忆单元有更加精细的设计，使得其能够建模更长的序列，通过实验发现，在其他参数保持一致（最优参数设定）之下 LSTM 的实验效果较好。

5.2. 循环神经网络隐层数量

增加 basline 中 gru 的隐层数量（64），发现当隐层数量增加时效果会有一定提升，但增加太多之后提升并不明显。

5.3. 循环神经网络层数

尝试增加循环神经网络的层数，通过实验结果发现，当层数增加到 2 时实验效果会有一定的提高，但是继续提高循环神经网络的层数反而会使得其性能下降，可能原因为产生了过拟合。

5.4. Dropout

本次实验在每个卷积层之后加入了 dropout 层随机失活部分神经元从而达到防止过拟合的效果。通过调参发现 dropout=0.5 时效果最好。

5.5. 插值的位置

本次实验考虑将插值放在循环神经网络之前和之后两种模型。实验过程中发现，当插值放在循环神经网络之前时，由于特征的时间维度得到恢复，导致程序运行时间明显增加，同时性能上并没有得到提升。

5.6. 卷积核的大小

在调参过程当中发现当卷积核大小为 7，padding 大小为 3 时效果较好。

5.7. 调参的最优结果

调参之后的最优结果见表2，超参数配置为 dropout = 0.5，LSTM 隐层数量为 256，层数为 1。

表 2: 调参的最优结果

	f_measure	precision	recall
event base	0.212563	0.212507	0.239237
segment base	0.568723	0.613917	0.546164
tagging base	0.61841	0.692085	0.566297

6. 数据增强方法的调研与应用

本次项目中调研到以下几种音频的数据增强的方法：加入噪声，时域压缩和拉伸，时移，片段交换拼接，音调改变等。本次项目尝试了如下几种方法进行实验。由于训练数据均使用弱标签，故以上方法均无需改动标签。

6.1. 加入噪声

考虑对每一帧的特征加入高斯噪声，在 dataset.py 中修改 train_dataset 函数中的 feat 变量。对应代码如下：

```
if self.augment  == 'noi':
    #对每一帧加高斯噪声
    for i in range(feat.size(0)):
        feat[i] += random.gauss(0,0.01)
```

加入噪声后的实验结果如表3所示：

表 3: 加入噪声后的实验结果

	f_measure	precision	recall
event base	0.196178	0.203557	0.224655
segment base	0.562495	0.620875	0.545191
tagging base	0.619964	0.701859	0.571115

6.2. 片段拼接

本次实验尝试交换音频片段开始和最后的随机长度的片段，代码如下：

```
elif self.augment  == 'mix':
    #交换音频最开始的一部分和最后一部分
    rand_num = random.randint(2,6)
    time_step = feat.size(0)
    seg_length = int(time_step/rand_num)
    temp_1 = deepcopy(feat[0:seg_length])
    temp_2 = deepcopy(feat[-seg_length:])
    feat[0:seg_length] = temp_2
    feat[-seg_length:] = temp_1
```

实验结果如下：

表 4: 片段拼接后的实验结果

	f_measure	precision	recall
event base	0.18653	0.194694	0.204755
segment base	0.556809	0.62076	0.521727
tagging base	0.616286	0.703134	0.562228

6.3. 音调改变

对于每段音频，随机整体升高或降低频率分布，代码如下：

```
elif self.augment  == 'pitch':
    #所有帧的频率整体向上或向下平移
    rand_num = random.randint(2,10)
    for i in range(feat.size(0)):
        #rand_num为奇数时频率整体升高
        if rand_num%2 == 0:
            feat[i,rand_num:] = deepcopy(feat[i
    ,0:(feat.size(1)-rand_num)])
            feat[i,0:rand_num] = 0
        #rand_num为偶数时频率整体下降
        else:
            feat[i,0:(feat.size(1)-rand_num)] =
    deepcopy(feat[i,rand_num:] )
            feat[i,-rand_num] = 0
```

实验结果如下：

表 5: 音调改变后的实验结果

	f_measure	precision	recall
event base	0.206688	0.205769	0.23785
segment base	0.608128	0.659909	0.582797
tagging base	0.669049	0.735164	0.623377

7. 总结

本次实践项目加深我对声音事件监测任务的了解，提高了模型构建，调参，音频数据增强相关的能力。