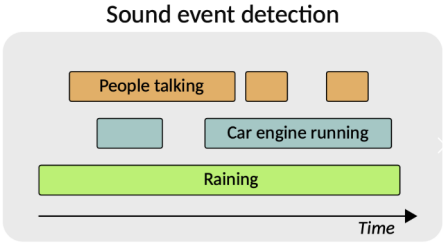
声音事件检测

521030910381-姚博

1. 引入

声音事件检测，即通过识别在各种场景下录制的不同声源的音频来判断音频中出现的声源或发生的事件，要求识别出一个音频中声音事件的种类，以及检测出声音事件发生和结束的时间。声音事件检测的数据可以分为三类：一个是无标签数据，一个是弱标签数据，即只有声音事件的分类，没有时间信息，最后是强标签数据，既有声音事件的分类，也有声音事件的起止时间。对于强标签任务，可以相当于重心在分类上面。本次实践中我们的数据集中一共包含十个类别：Speech，Dog,Cat,Alarm_bell_ringing, Dishes,Frying,Blender,Running_water, Vacuum_cleaner,Electric_shaver_toothbrush.

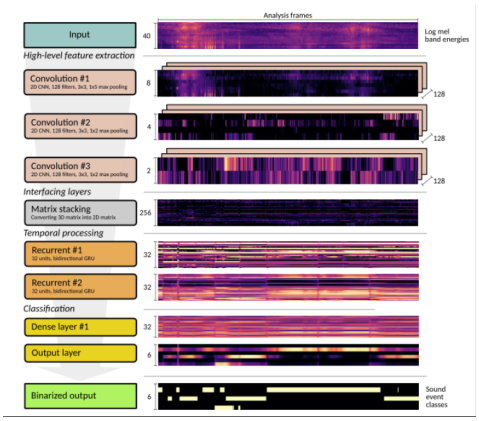


2. 模型架构与训练

数据准备：我们利用所给的脚本对数据集音频（包括训练集与测试集）进行处理，得到对应的 features,labels 等信息。CRNN 模型: CRNN 模型是一种结合了卷积神经网络（CNN）和循环神经网络（RNN）的深度学习模型，用于处理时序数据。能够同时捕获时频特征和时序信息。通用的 baseline 模型与传统机器学习任务的步骤类似，baseline 采用三层卷积模块的 CRNN 架构，即对输入的三维张量（三个维度分别代表 batch size、时间步长和频数）进行多次卷积和 maxpooling 操作，并且通过 BiGRU 层后，用全连接层将其改变为 batch size、时间步长和声音事件类别数对应的三维张量，以分数表示声音时间出现的概率。我们采用 BCEloss 作为损失函数：

$$\text{BCELoss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\hat{y}_i) + (1-y_i)\cdot \log(1-\hat{y}_i)]$$

2.1. baseline CRNN



基本步骤是：对输入三维张量数据（三个维度分别代表 batch size、时间步长和频数）进行 unsqueeze 操作变换为四维，首先通过一个 BatchNorm2d 层做 batch normalization 操作，接下来分别进入三个卷积模块，继续进行 batch normalization、relu 和 max pooling 操作得到模块的输出，三个卷积模块的输入维度和输出维度分别为 1->16,16->32,32->64。接下来将卷积模块的输出 reshape 为三维张量后通过一个单层的 BiGRU 层，最后用全连接层将其第三个维度对应到到声音事件的总类别数（10），最后从网络中输出，后续还需要通过线性的 softmax 回归层（在这里由于使用 BCELoss 即 sigmoid 层），得到预测的该段音频出现每一个声音事件的概率.

2.2. 代码实现

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

```
    def __init__(self, num_freq, num_class):
        ##############################
        # YOUR IMPLEMENTATION
        # Args:
        #     num_freq: int, mel frequency bins
        #     num_class: int, the number of output classes
        ##############################
        super(Crnnplus, self).__init__()
        self.inputdim = num_freq
        self.outputdim = num_class


        self.bn0 = nn.BatchNorm2d(64)
        self.bn1 = nn.BatchNorm2d(16)
        self.bn2 = nn.BatchNorm2d(32)
        self.bn3 = nn.BatchNorm2d(64)


        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 16,
                               kernel_size=(3, 3), stride=(1, 1),
                               padding = (1, 1), bias=False)

        self.conv2 = nn.Conv2d(in_channels = 16, out_channels = 32,
                               kernel_size=(3, 3), stride=(1, 1),
                               padding = (1, 1), bias=False)

        self.conv3 = nn.Conv2d(in_channels = 32, out_channels = 64,
                               kernel_size=(3, 3), stride=(1, 1),
                               padding = (1, 1), bias=False)

        self.bigru = nn.GRU(input_size = 64 * 8, hidden_size = 128, num_layers = 1, batch_first = True, bidirectional = True)
        self.fc = nn.Linear(256, num_class, bias=True)

        self.relu = nn.ReLU()

        self.init_bn(self.bn0)
        self.init_bn(self.bn1)
        self.init_bn(self.bn2)
        self.init_bn(self.bn3)
        self.init_layer(self.fc)
        self.init_gru(self.bigru)
    def detection(self, x):
        ##############################
        # YOUR IMPLEMENTATION
        # Args:
        #     x: [batch_size, time_steps, num_freq]
        # Return:
        #     frame_prob: [batch_size, time_steps, num_class]
        ##############################
        batch_size, time_steps, num_freq = x.shape[0], x.shape[1], x.shape[2]
        x = x.unsqueeze(1)
        x = x.transpose(1, 3)
        x = self.bn0(x)
        x = x.transpose(1, 3)

        x = self.relu(self.bn1(self.conv1(x)))
        x = nn.functional.max_pool2d(x, kernel_size = (1, 2), stride = (1, 2))

        x = self.relu(self.bn2(self.conv2(x)))
        x = nn.functional.max_pool2d(x, kernel_size = (1, 2), stride = (1, 2))

        x = self.relu(self.bn3(self.conv3(x)))
        x = nn.functional.max_pool2d(x, kernel_size = (1, 2), stride = (1, 2))

        x = x.transpose(1, 2)
        x = x.reshape(batch_size, time_steps, -1)
        x, _ = self.bigru(x)
        x = torch.sigmoid(self.fc(x))

        return x

    def forward(self, x):
        frame_prob = self.detection(x)  # (batch_size, time_steps, class_num)
        clip_prob = linear_softmax_pooling(frame_prob)
        # (batch_size, class_num)
        '''(samples_num, feature_maps)'''
        return {
            'clip_prob': clip_prob,
            'frame_prob': frame_prob
        }
```

我们这里只展示init_ 方法与 detection 函数解释模型。

2.3. 实验结果

	f_measure	precision	recall
event_based	0.125194	0.100751	0.199621
segment_based	0.600531	0.678131	0.552422
tagging_based	0.651628	0.720691	0.612963

表 1: *CRNN_plus*

平均 MAP：0.69904

3. 模型探究与分析

我在想会不会更多的卷积层能够提高模型的分辨能力，或者是较少的卷积层已经能够很好的完成任务，因此我采用了不同层数的 CRNN, 以下是不同模型结构的表现：

只使用两层卷积：平均 MAP：0.66815

	f_measure	precision	recall
event_based	0.0898606	0.0701401	0.143831
segment_based	0.56817	0.659592	0.508428
tagging_based	0.622605	0.717372	0.557563

表 2: *CRNN*

增加一层卷积，即加一层 128 维模块：平均

	f_measure	precision	recall
event_based	0.0802556	0.0626079	0.143595
segment_based	0.574471	0.640967	0.539497
tagging_based	0.604498	0.722956	0.556427

表 3: *CRNN1*

MAP:0.69291

增加到 256 维：平均 MAP：0.70232

	f_measure	precision	recall
event_based	0.105807	0.0938937	0.158871
segment_based	0.58634	0.720328	0.515347
tagging_based	0.635826	0.723961	0.586488

表 4: *CRNN2*

若是再添加到 512 维，事实上，从这里已经感觉到模型性能下降了，因此以此做一个趋势验证：平均 MAP:0.70634

我们更加关心 F-measure, 并且注意到 recall，baseline 模型即三层应该是最好的效果，证明过于复杂反而会适得其反。

4. 数据增强

通常我们可以通过 time stretching (时间拉伸) pitch shifting (音高变换) dynamic range compression (动态范围拉伸) sub-frame time shifting (子帧时间偏移) block mixing (块混合) mixup (混音) 等方法进行音频信号的增强。

	f_measure	precision	recall
event_based	0.0938441	0.0757102	0.151806
segment_based	0.600982	0.675321	0.556482
tagging_based	0.650016	0.731558	0.599795

表 5: *CRNN3*

5. 实验总结

这次任务让我学习从数据预处理、特征提取到进行训练、测试的全过程，了解声音事件检测任务的大体流程，以及经典的 CRNN 模型的设计思路与方法，通过调整模型的各级参数以及网络模型结构，让我对模型结构对模型性能的影响有了进一步的认识。感谢老师与学长的帮助!