第5章　视频分类

互联网上图像和视频的规模日益庞大，据统计，Youtube 网站每分钟就有数百小时的视频产生，这使得研究人员急切需要研究视频分类相关算法来帮助人们更加容易地找到感兴趣的视频。这些视频分类算法可以自动分析视频所包含的语义信息，理解其内容，对视频进行自动标注、分类和描述，达到与人媲美的准确率。大规模视频分类是继图像分类问题后的又一个急需解决的关键问题。

视频分类是指给定一个视频片段，对其中包含的内容进行分类。类别通常是动作（如做蛋糕）、场景（如海滩）、物体（如桌子）等。其中又以视频动作分类最为热门，毕竟动作本身就包含“动”态的因素，不是“静”态的图像所能描述的，因此也是最体现视频分类功底的。视频分类的主要目的是理解视频中包含的内容，确定视频对应的几个关键主题。视频分类不仅是要理解视频中的每一帧图像，更重要的是要理解多帧之间包含的更深层次的语义信息。视频分类的研究内容主要包括多标签的通用视频分类和人类行为识别等，如图 5-0-1 所示。与之密切相关的是，视频描述生成（Video Captioning）试图基于视频分类的标签，形成完整的自然语句，为视频生成包含最多动态信息的描述说明。

图 5-0-1　视频分类示意图

在深度学习方法广泛应用之前，大多数的视频分类方法采用基于人工设计的特征和典型的机器学习方法研究行为识别和事件检测。

传统的视频分类研究专注于采用对局部时空区域的运动信息和表观（Appearance）信息编码的方式获取视频描述符，然后利用词袋模型（Bag of Words）等方式生成视频编码，最后利用视频编码来训练分类器（如 SVM），区分视频类别。视频的描述符依赖人工设计的特征，如使用运动信息获取局部时空特征的梯度直方图（Histogram of Oriented Gradients，HOG）；使用不同类型轨迹的光流直方图（Histogram of Optical Flow，HOF）和运动边界直方图（Motion Boundary Histogram，MBH）；使用词袋模型或 Fisher 向量方法来生成视频编码等。当前，基于轨迹的方法（尤其是 DT 和 IDT）是最高水平的人工设计特征算法的基础。许多研究者正在尝试改进 IDT，如通过增加字典的大小和融合多种编码方法，通过开发

子采样方法生成 DT 特征的字典，在许多行为数据集上获得了不错的性能。

然而，随着深度神经网络的兴起，特别是 CNN、LSTM、GRU 等在视频分类中的成功应用，其分类性能逐渐超越了基于 DT 和 IDT 的传统方法，使得这些传统方法逐渐淡出了人们的视野。深度网络为解决大规模视频分类问题提供了新的思路和方法。近年来得益于深度学习研究的巨大进展，特别是卷积神经网络（Convolutional Neural Networks，CNN）作为一种理解图像内容的有效模型，在图像识别、分割、检测和检索等方面取得了不错的研究成果。CNN 在静态图像识别问题中取得了空前的成功，国内外研究者也开始研究将 CNN 等深度网络应用到视频和行为分类任务中。除此之外，近几年 Transformer 在视觉领域也表现出不错的效果，在计算机视觉的各个领域大放异彩，这当然也包括视频分类领域。

基于 TSN 模型的视频分类

5.1 实践一：基于 TSN 模型的视频分类

本节将通过实现 TSN 网络在 HMDB51 数据集上实现视频分类。

Temporal Segment Network（TSN）是视频分类领域经典的基于 2D-CNN 的解决方案。该方法主要解决视频的长时间行为判断问题。通过稀疏采样视频帧的方式代替稠密采样，既能捕获视频全局信息，也能去除冗余，减少计算量。稀疏采样并提取特征后，将每帧特征融合得到视频的整体特征，并用于分类。TSN 的整体过程如下：

① 将输入视频划分成 K 个片段，每个片段随机取一帧；

② 使用两个卷积网络分别提取空间和时序特征（RGB 图像和光流图像，可以只采用 RGB 分支）；

③ 通过片段共识函数，分别融合两个分支不同片段结果；

④ 两类共识函数的结果融合。

下面我们实现采用 ResNet-50 骨干网络的单路 TSN 网络。

步骤 1：了解 TSN 项目整体结构

整个 TSN 项目如图 5-1-1 所示，configs 文件夹中存储着网络的配置文件；model 文件夹中是网络结构搭建的部分；reader 文件用于数据集的定义和数据的读取；avi2jpg.py 用于将 hmdb51 数据中的视频文件逐帧处理为 jpg 文件并保存在以视频名称命名的文件夹下；jpg2pkl.py 和 data_list_gener.py 用于将同一视频对应的 jpg 文件转换成 pkl 文件中，并划分数据集生成用于训练、验证和测试；train.py 和 infer.py 分别用于 TSN 的训练和测试（如图 5-1-1 所示）。

如图 5-1-2 所示，Configs 文件下的 tsn.txt 存储整个项目需要用到的超参数。

MODEL 部分主要包括数据加载的格式（jpg\pkl）、分类的数目、每个视频片段被划分成几份（seg_num）、每份中抽取几帧用于训练测试（seg_len）、图像归一化时所用的均值和方差（image_mean、image_std）等。TRAIN、VALID、TEST、INFER 则是网络在进行训练、验证、测试、预测等阶段时需要设定的图像尺寸、读取图像的线程、批次大小，以及训练轮数等。

```
|--configs                                  # 配置
|--model                                    # 模型
|--reader                                   # 读取数据
|--data                                     # 数据
|--data_list_gener.py                       # 生成train、test、eval
|--infer.py                                 # 模型推断
|--avi2jpg.py                               # 视频变成帧，保存为jpg
|--train.py                                 # 训练脚本
|--utils.py                                 # 通用工具
|--jpg2pkl.py                               # jpg变成pkl
|--config.py                                # 读取配置并生成
```

图 5-1-1　TSN 项目结构

```
[MODEL]
name = "TSN"
format = "pkl"
num_classes = 10
seg_num = 3
seglen = 1
image_mean = [0.485, 0.456, 0.406]
image_std = [0.229, 0.224, 0.225]
num_layers = 50

[TRAIN]
epoch = 45
short_size = 240
target_size = 224
num_reader_threads = 1
buf_size = 1024
batch_size = 10
use_gpu = True
num_gpus = 1
filelist = "./data/hmdb_data_demo/train.list"
learning_rate = 0.01
learning_rate_decay = 0.1
l2_weight_decay = 1e-4
momentum = 0.9
total_videos = 80

[VALID]
short_size = 240
target_size = 224
num_reader_threads = 1
buf_size = 1024
batch_size = 2
filelist = "./data/hmdb_data_demo/val.list"

[TEST]
seg_num = 7
short_size = 240
target_size = 224
num_reader_threads = 1
buf_size = 1024
batch_size = 10
filelist = "./data/hmdb_data_demo/test.list"

[INFER]
short_size = 240
target_size = 224
num_reader_threads = 1
buf_size = 1024
batch_size = 1
filelist = "./data/hmdb_data_demo/test.list"
```

图 5-1-2　TSN 配置文件

步骤 2：认识 HMDB51 数据集

1. 数据集概览

数据集采用 HMDB51 数据集，HMDB51 数据集于 2011 年由 Brown university 发布，该数据集视频多数来源于电影，还有一部分来自公共数据库以及 YouTube 等网络视频库。数据集包含 6849 段样本，分为 51 类，每类至少包含 101 段样本（如图 5-1-3 所示）。

图 5-1-3　数据集类别

HMDB51 所包含的动作主要分为五类。

① 一般面部动作：微笑，大笑，咀嚼，交谈。

② 面部操作与对象操作：吸烟，吃，喝。

③ 一般的身体动作：侧手翻，拍手，爬，爬楼梯，跳，落在地板上，反手翻转、倒立、跳、拉、推、跑，坐下来，坐起来，翻跟头，站起来，转身，走，跛。

④ 与对象交互动作：梳头，抓，抽出宝剑，运球、打高尔夫、打东西，踢球，挑，倒、推东西，骑自行车，骑马，射球，射弓、枪，摆棒球棍，舞剑锻炼，扔。

⑤ 人体动作：击剑，拥抱，踢某人，亲吻，拳打，握手，剑战。

2. 数据集下载

HMDB51 数据集可在其官网下载：https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/#Downloads。

步骤 3：视频数据处理与加载

1. 数据预处理

TSN 网络以一个视频片段的多张视频帧作为输入，因此数据处理的第一步要将视频片段提取成一张张视频帧并存储下来。在该部分遍历访问所有视频片段，对于每一个视频片

段通过 Opencv 的 VideoCapture 类进行解析，将每一张图像存储到以视频名字命名的文件下，并记录对应的 label。

```
import os
import numpy as np
import cv2
    for each_video in videos:
        cap = cv2.VideoCapture(each_video)
        frame_count = 1
        success = True
        while success:
            success, frame = cap.read()
            # print('read a new frame:', success)
            params = []
            params.append(1)
            if success:
                cv2.imwrite(each_video_save_full_path + each_video_name + "_%d.jpg" %
frame_count, frame, params)
            frame_count += 1
        cap.release()
np.save('label_dir.npy', label_dir)
```

将视频处理抽取为视频帧后，通过下面的代码，将同一视频对应的 jpg 文件以及标签保存在以视频命名的 pkl 文件中，对于每一类抽取该类视频总数的 80％为训练集，10％为验证集，10％为测试集，同时，分别生成对应训练、验证和测试的 txt 列表。

```
from multiprocessing import Pool
label_dic = np.load('label_dir.npy', allow_pickle=True).item()
for key in label_dic:
    each_mulu = key + '_jpg'
    print(each_mulu, key)
    label_dir = os.path.join(source_dir, each_mulu)
    label_mulu = os.listdir(label_dir)
    tag = 1
    for each_label_mulu in label_mulu:
        image_file = os.listdir(os.path.join(label_dir, each_label_mulu))
        image_file.sort()
        image_name = image_file[0][:-6]
        image_num = len(image_file)
        frame = []
        vid = image_name
        for i in range(image_num):
            image_path = os.path.join(os.path.join(label_dir, each_label_mulu), image_name +
'_' + str(i+1) + '.jpg')
            frame.append(image_path)
        output_pkl = vid + '.pkl'
        if tag < 9:
            output_pkl = os.path.join(target_train_dir, output_pkl)
        elif tag == 9:
            output_pkl = os.path.join(target_test_dir, output_pkl)
        elif tag == 10:
            output_pkl = os.path.join(target_val_dir, output_pkl)
        tag += 1
```

```
        f = open(output_pkl, 'wb')
        pickle.dump((vid, label_dic[key], frame), f, -1)
        f.close()
```

2. 数据集类定义

定义 HMDB51Dataset 类，用于构建训练、验证和测试过程中的数据读取器。

init()函数有 name、mode、cfg 三个输入参数，其中 name 表示模型的名字，mode 决定用于训练还是测试，cfg 则是 TSN 配置文件的路径。通过读取 cfg 配置文件来初始化输入网络的图像大小、视频划分片段、每个片段抽取的图像数目、归一化的均值等超参数。

```
def __init__(self, name, mode, cfg):
    '''初始化函数'''
    self.cfg = cfg                                          # 相关配置
    self.mode = mode                                        # 用于训练还是测试
    self.name = name                                        # 模型名字
    self.format = cfg.MODEL.format                          # 数据格式
    self.num_classes = self.get_config_from_sec('model', 'num_classes')   # 数据集的类别数
    self.seg_num = self.get_config_from_sec('model', 'seg_num')
    self.seglen = self.get_config_from_sec('model', 'seglen')

    self.seg_num = self.get_config_from_sec(mode, 'seg_num', self.seg_num)
    self.short_size = self.get_config_from_sec(mode, 'short_size')
    self.target_size = self.get_config_from_sec(mode, 'target_size')
    self.num_reader_threads = self.get_config_from_sec(mode, 'num_reader_threads')
                                                                    # 读取数据的线程数
    self.buf_size = self.get_config_from_sec(mode, 'buf_size')
    self.enable_ce = self.get_config_from_sec(mode, 'enable_ce')
    self.img_mean = np.array(cfg.MODEL.image_mean).reshape([3, 1, 1]).astype(np.float32)
                                                                    # 图像均值
    self.img_std = np.array(cfg.MODEL.image_std).reshape([3, 1, 1]).astype(np.float32)
                                                                    # 图像方差
    # set batch size and file list
    self.batch_size = cfg[mode.upper()]['batch_size']               # 数据批大小
    self.filelist = cfg[mode.upper()]['filelist']                   # 数据列表
    if self.enable_ce:
        random.seed(0)
        np.random.seed(0)
    self.samples = open(self.filelist, 'r').readlines()
    if self.mode == 'train':
        np.random.shuffle(self.samples)
```

decode_pickle()函数通过索引 idx，在事先处理好的 pickle 文件中加载用于训练、验证的视频图像列表和对应的标注或用于测试的图像列表，同时会对图像列表中的图像通过 imgs_transform()函数进行一系列的图像操作。

```
def decode_pickle(self,idx):
    sample = self.samples[idx].strip()
    pickle_path = sample
    try:
        if python_ver < (3, 0):
            data_loaded = pickle.load(open(pickle_path, 'rb'))          #读取 PKL 文件
        else:
```

```
            data_loaded = pickle.load(open(pickle_path, 'rb'), encoding='bytes') vid,
label, frames = data_loaded
            if len(frames) < 1:
                logger.error('{} frame length {} less than 1.'.format( pickle_path, len(frames)))
                return None, None
        except:
            logger.info('Error when loading {}'.format(pickle_path))
            return None, None
        if self.mode == 'train' or self.mode == 'valid' or self.mode == 'test':
            ret_label = label
        elif self.mode == 'infer':
            ret_label = vid
        imgs = video_loader(frames, self.seg_num, self.seglen, self.mode)      #读取视频图片
        return self.imgs_transform(imgs, ret_label)                #对视频图片列表进行处理并返回
```

imgs_transform()函数的主要功能是对图像列表中的图像进行一系列的图像处理，包括对训练图像进行随机裁剪、水平翻转以及对测试数据进行中心裁剪，对输入网络的图像进行归一化和尺寸统一等。

```
def imgs_transform(self, imgs, label):
    imgs = group_scale(imgs, self.short_size)
    if self.mode == 'train':
        #训练数据加载时进行随机裁剪和水平翻转
        if self.name == "TSN":
            imgs = group_multi_scale_crop(imgs, self.short_size)
        imgs = group_random_crop(imgs, self.target_size)
        imgs = group_random_flip(imgs)
    else:
        #测试数据加载时进行中心裁剪
        imgs = group_center_crop(imgs, self.target_size)
    np_imgs = (np.array(imgs[0]).astype('float32').transpose((2, 0, 1))).reshape(1, 3,
self.target_size, self.target_size) / 255
    for i in range(len(imgs) - 1):
        img = (np.array(imgs[i + 1]).astype('float32').transpose((2, 0, 1))).reshape(1, 3,
self.target_size, self.target_size) / 255
        np_imgs = np.concatenate((np_imgs, img))
    imgs = np_imgs
    imgs -= self.img_mean
    imgs /= self.img_std
    imgs = np.reshape(imgs,(self.seg_num, self.seglen * 3, self.target_size, self.target_size))
    return imgs, label
```

getitem()函数在迭代的过程中，通过调用 decode_pickle 返回视频图像和对应的标签，如果是测试阶段，标签会返回为空。

```
def __getitem__(self,idx):
    '''根据给定索引读取数据'''
    if self.format == 'pkl':
        #如果数据格式是 pkl,则使用 decode_pickle()函数进行读取
        imgs, label = self.decode_pickle(idx)
    elif self.format == 'mp4':
        #如果数据格式是 mp4,则使用 decode_mp4()函数进行读取
        imgs, label = self.decode_mp4(idx)
```

```
        else:
            raise "Not implemented format {}".format(self.format)
        return imgs, label
```

步骤 4：搭建 TSN 网络

TSN 以 Resnet 作为特征提取网络，同时提取一个视频的多帧图像特征。因此，TSN 网络的搭建过程主要是搭建 Resnet 的过程，但是在网络的输入和输出上需要进行针对性的调整。

ConvBNLayer 同前面的几个实验一样，继承了 Paddle.nn.Layer，用于构建卷积＋BN 的结构，这个结构是接下来搭建 TSN 网络的基础结构。ConvBNLayer 包含两部分，首先是通过 Paddle.nn.Conv2D 构建一个卷积层，紧接着通过 Paddle.nn.BatchNorm2D 实现批归一化。在调用 ConvBNLayer 的过程中通过 num_channels 等参数确定卷积核的大小、数目、步长以及激活函数等。

```
class ConvBNLayer(Layer):
    '''构建卷积 + BN 层的结合，在网络中这个组合比较常用'''
    def __init__(self,
                 name_scope,
                 num_channels,
                 num_filters,
                 filter_size,
                 stride = 1,
                 groups = 1,
                 act = None):
        super(ConvBNLayer, self).__init__(name_scope)
        self._conv = Conv2D(
            in_channels = num_channels,
            out_channels = num_filters,
            kernel_size = filter_size,
            stride = stride,
            padding = (filter_size - 1) // 2,
            groups = groups,
            bias_attr = False
        )
        self._batch_norm = BatchNorm2D(num_filters, act = act)
    def forward(self, inputs):
        '''网络前向传播过程'''
        y = self._conv(inputs)
        y = self._batch_norm(y)
        return y
```

BottleneckBlock 与 2.4 节相似，是用于构建 Resnet 网络的残差模块。在 BottleneckBlock 中，输入的特征依次进入一个 1×1、3×3 和 1×1 的卷积，并根据选择的模式进行①或者②。

① 与原始输入特征相加；

② 进行一次 1×1 卷积。

```
class BottleneckBlock(Layer):
    def __init__(self,
                 name_scope,
```

```
                 num_channels,
                 num_filters,
                 stride,
                 shortcut = True):
        super(BottleneckBlock, self).__init__(name_scope)
        self.conv0 = ConvBNLayer(
            self.full_name(),
            num_channels = num_channels,
            num_filters = num_filters,
            filter_size = 1,
            act = 'relu')
        self.conv1 = ConvBNLayer(
            self.full_name(),
            num_channels = num_filters,
            num_filters = num_filters,
            filter_size = 3,
            stride = stride,
            act = 'relu')
        self.conv2 = ConvBNLayer(
            self.full_name(),
            num_channels = num_filters,
            num_filters = num_filters * 4,
            filter_size = 1,
            act = None)
        if not shortcut:
            self.short = ConvBNLayer(
                self.full_name(),
                num_channels = num_channels,
                num_filters = num_filters * 4,
                filter_size = 1,
                stride = stride)
        self.shortcut = shortcut
        self._num_channels_out = num_filters * 4
    def forward(self, inputs):
        '''网络前向传播过程'''
        y = self.conv0(inputs)
        conv1 = self.conv1(y)
        conv2 = self.conv2(conv1)
        if self.shortcut:
            short = inputs
        else:
            short = self.short(inputs)
        y = paddle.add(x = short, y = conv2)
        layer_helper = paddle.incubate.LayerHelper(self.full_name(), act = 'relu')
        return layer_helper.append_activation(y)
```

解下来构建整个 TSN 特征提取部分，首先通过一个大小为 7×7、步长为 2 的卷积，紧接着通过根据输入的深度要求，循环地调用 BottleneckBlock 搭建 50、101 或 152 层特征提取网络。

```
class TSNResNet(Layer):
    def __init__(self, name_scope, layers = 50, class_dim = 102, seg_num = 10, weight_devay = None):
        super(TSNResNet, self).__init__(name_scope)
```

```
        self.layers = layers
        self.seg_num = seg_num
        supported_layers = [50, 101, 152]
        depth = [3, 4, 6, 3]
        num_filters = [64, 128, 256, 512]
        self.conv = ConvBNLayer(self.full_name(),
num_channels = 3,num_filters = 64,filter_size = 7,stride = 2,act = 'relu')
        self.pool2d_max = MaxPool2D(kernel_size = 3,stride = 2,padding = 1)
        self.bottleneck_block_list = []
        num_channels = 64
        for block in range(len(depth)):
            shortcut = False
            for i in range(depth[block]):
                bottleneck_block = self.add_sublayer(
                    'bb_%d_%d' % (block, i),
                    BottleneckBlock(
                        self.full_name(),
                        num_channels = num_channels,
                        num_filters = num_filters[block],
                        stride = 2 if i == 0 and block != 0 else 1,
                        shortcut = shortcut))
                num_channels = bottleneck_block._num_channels_out
                self.bottleneck_block_list.append(bottleneck_block)
                shortcut = True
        self.pool2d_avg = AvgPool2D(kernel_size = 7)
        import math
        stdv = 1.0 / math.sqrt(2048 * 1.0)
        self.out = Linear(
            in_features = num_channels,
            out_features = class_dim,
        )
        self.softmax = Softmax()
        self.metric = paddle.metric.Accuracy()
```

TSN 要同时提取一个视频中多个帧的特征，多帧图像会叠加在一起作为网络的输入，但是最后我们需要得到的是每帧图像的特征。因此，对于输入网络的多帧图像，会首先经过 reshape 操作进行融合，之后通过 TSN 网络的各层网络提取特征，再通过 reshape 操作将特征划分为不同帧图像对应的特征。

```
def forward(self, inputs, label = None):
    '''网络前向传播过程'''
    out = paddle.reshape(inputs, [-1, inputs.shape[2], inputs.shape[3], inputs.shape[4]])
    y = self.conv(out)
    y = self.pool2d_max(y)
    for bottleneck_block in self.bottleneck_block_list:
        y = bottleneck_block(y)
    y = self.pool2d_avg(y)
    y = paddle.reshape(x = y, shape = [-1, self.seg_num, y.shape[1]])
    y = paddle.mean(y, axis = 1)
    out = self.out(y)
    y = self.softmax(out)
    if label is not None:
        acc = paddle.mean(self.metric.compute(pred = y, label = label))
```

```
            return out, acc
        else:
            return y
```

步骤 5：训练 TSN 网络

定义 train 类：首先通过 pddle.set_device 设置使用 CPU 还是 GPU 进行训练，然后根据配置及文件中的内容，通过 TSNResNet 类实例化用于训练的网络 train_model。

```
def train(args):
    # 设置在 GPU 上训练还是在 CPU 上训练
    place = "gpu" if args.use_gpu else "cpu"
    paddle.set_device(place)
    # 进行训练参数配置
    config = parse_config(args.config)
    train_config = merge_configs(config, 'train', vars(args))
    print_configs(train_config, 'Train')
    # 创建训练网络
    train_model = TSN1.TSNResNet('TSN', train_config['MODEL']['num_layers'],
                                 train_config['MODEL']['num_classes'],
                                 train_config['MODEL']['seg_num'], 0.00002)
```

通过 paddle.optimizer.Momentum 定义优化器，并加载预训练模型或之前训练的模型参数。

```
# 创建网络优化器
opt = paddle.optimizer.Momentum(0.001, 0.9, parameters = train_model.parameters())
if args.pretrain:
    # 加载上一次训练的模型，继续训练
    state_dict = paddle.load(args.save_dir + '/tsn_model.pdparams')
    train_model.set_state_dict(state_dict)

if not os.path.exists(args.save_dir):
    os.makedirs(args.save_dir)
```

通过 HMBD51Dataset 类和 paddle.io.DataLoader 创建训练数据读取器，并通过 paddle.nn.CrossEntropyLoss 实现交叉熵损失。

```
# 创建训练数据读取器
train_reader = HMBD51Dataset(args.model_name.upper(), 'train', train_config)
traindataloder = paddle.io.DataLoader(train_reader,
                    batch_size = train_config.TRAIN.batch_size,
                    num_workers = 0, collate_fn = train_reader.collate_fn)

    epochs = args.epoch or train_model.epoch_num()
    # 定义损失函数计算方式
    ce_loss = paddle.nn.CrossEntropyLoss()
```

整个数据集训练 epochs 次，对于数据读取器每次返回的图像和标注输入网络，并计算输出和标注之间的交叉熵损失。通过 backward()进行反向传播，学习网络的参数。每次反向转播后，通过 opt.clear_grad()清空梯度，并在训练的过程中输出网络的损失、精度。

```
for i in range(epochs):
```

```
    for batch_id, data in enumerate(traindataloder):
        img = data[0].astype('float32')
        label = data[1].astype('int64')
        label.stop_gradient = True
        # 进行网络前向传播
        out, acc = train_model(img, label)
        # 计算损失
        loss = ce_loss(out, label)
        avg_loss = loss
        avg_loss.backward()                                    # 进行反向传播
        opt.step()
        opt.clear_grad()
        if batch_id % 10 == 0:
            # 进行模型保存
            logger.info("Loss at epoch {} step {}: {}, acc: {}".format(i, batch_id, avg_loss.
numpy(), acc.numpy()))
            print("Loss at epoch {} step {}: {}, acc: {}".format(i, batch_id, avg_loss.numpy(),
acc.numpy()))
            paddle.save(train_model.state_dict(), args.save_dir + '/tsn_model.pdparams')
    logger.info("Final loss: {}".format(avg_loss.numpy()))
    print("Final loss: {}".format(avg_loss.numpy()))
```

网络的训练过程如图 5-1-4 所示。

```
Loss at epoch 0 step 0: [4.054985], acc: [0.2]
Loss at epoch 0 step 1: [3.5705905], acc: [0.2]
Loss at epoch 0 step 2: [4.8680696], acc: [0.1]
Loss at epoch 0 step 3: [3.4408088], acc: [0.4]
Loss at epoch 0 step 4: [2.1347373], acc: [0.5]
Loss at epoch 0 step 5: [1.670853], acc: [0.3]
Loss at epoch 0 step 6: [2.4477277], acc: [0.1]
Loss at epoch 0 step 7: [2.1186583], acc: [0.1]
Final loss: [2.1186583]
Loss at epoch 1 step 0: [2.5484908], acc: [0.3]
Loss at epoch 1 step 1: [2.3663619], acc: [0.2]
Loss at epoch 1 step 2: [3.6884594], acc: [0.1]
Loss at epoch 1 step 3: [1.8204875], acc: [0.1]
```

图 5-1-4　训练过程

步骤 6：视频预测

模型预测部分整体与训练部分相似，首先读取配置文件并创建网络，然后加载训练后的参数，并通过 HMDB51Dataset 创建预测数据读取器。

```
def infer(args):
    #进行推理参数配置
    config = parse_config(args.config)
    infer_config = merge_configs(config, 'infer', vars(args))
    print_configs(infer_config, "Infer")
    # 创建网络
    infer_model = TSN1.TSNResNet('TSN', infer_config['MODEL']['num_layers'],
                                 infer_config['MODEL']['num_classes'],
                                 infer_config['MODEL']['seg_num'], 0.00002)
```

```
label_dic = np.load('label_dir.npy', allow_pickle=True).item()
label_dic = {v: k for k, v in label_dic.items()}
infer_reader = HMDB51Dataset(args.model_name.upper(), 'infer', infer_config)
#如果没有权重文件,则停止
if args.weights:
    weights = args.weights
else:
    print("model path must be specified")
    exit()
#加载训练好的模型
state_dict = paddle.load(weights)
infer_model.set_state_dict(state_dict)
infer_model.eval()
```

与训练过程不同的是,在预测过程中数据读取器只返回图像,同时网络也直接输出预测的结果,不再需要计算损失和反向传播梯度。

```
acc_list = []
for batch_id, data in enumerate(infer_reader):
    img = data[0].astype('float32')
    img = paddle.to_tensor(img[np.newaxis, :])
    y_data = data[1]
    out = infer_model(img).numpy()[0]                    #进行网络前向传播,预测结果
    label_id = np.where(out == np.max(out))
    print("实际标签{}, 预测结果{}".format(y_data, label_dic[label_id[0][0]]))
```

至此,我们就完成了 TSN 网络的搭建、训练和预测过程。

5.2　实践二:基于 ECO 模型的视频分类

基于 ECO 模型的视频分类

在本节,我们将通过实现 ECO 网络在 UCF101 数据集上实现视频分类。

Efficient Convolutional Network for Online Video Understanding(ECO)是视频分类领域经典的基于 2D-CNN 和 3D-CNN 融合的解决方案。该方案主要解决视频的长时间行为判断问题,通过稀疏采样视频帧的方式代替稠密采样,既能捕获视频全局信息,也能去除冗余,减少计算量。最终将 2D-CNN 和 3D-CNN 的特征融合得到视频的整体特征,并进行视频分类。本代码实现的模型为基于单路 RGB 图像的 ECO-full 网络结构,2D-CNN 部分采用修改后的 Inception 结构,3D-CNN 部分采用裁剪后的 3DResNet18 结构。ECO 的模型结构如图 5-2-1 所示。

步骤 1:认识 ECO 项目结构

整个 ECO 项目如图 5-2-2 所示,configs 文件夹中存储着网络的配置文件,config.py 用于加载 configs 文件中存储的配置文件;model 文件夹中是网络结构搭建的部分,分为两个部分,分别是 3D 卷积部分和 ECO 整体网络结构;best_model 文件下存储训练过程中最优的网络参数;result 下存储网络训练过程中的数据;reader.py 用于数据集的定义和数据的读取;avi2jpg.py 用于将 ucf101 数据中的视频文件逐帧处理为 jpg 文件并保存在以视频名称命名的文件夹下;jpg2pkl.py 和 data_list_gener.py 用于将同一视频对应的 jpg 文件转换

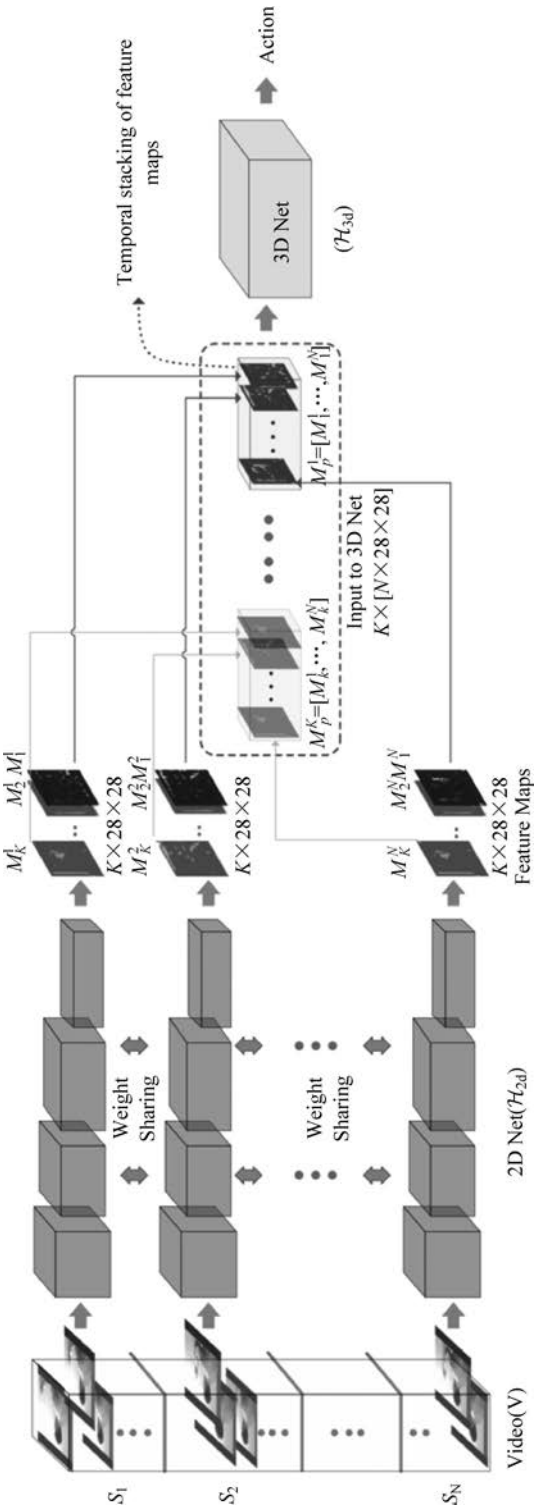


图 5-2-1　ECO 网络结构

成 pkl 文件中，并划分数据集生成训练、验证和测试集；train. py 和 infer. py 分别用于 TSN 的训练和测试。

```
|--configs                    # 配置
|--model                      # 模型
|--best_model                 # 训练好的模型
|--result                     # 训练过程中的数据
|--reader                     # 读取数据
|--data                       # 数据
|--data_list_gener.py         # 生成train、test、eval
|--test.py                    # 模型测试
|--avi2jpg.py                 # 视频变成帧，保存为jpg，图片质量95
|--train.py                   # 训练脚本
|--utils.py                   # 通用工具
|--jpg2pkl.py                 # jpg变成pkl
|--config.py                  # 读取配置并生成
```

图 5-2-2　ECO 项目结构

步骤 2：认识 UCF101 数据集

1. 数据集概览

UCF101 是一个现实视频的动作识别数据集，收集自 YouTube，提供了来自 101 个动作类别的 13320 个视频。UCF101 是 UCF50 数据集的扩展。

UCF101 在动作方面提供了较大的多样性，并且在摄像机运动、对象外观和姿态、对象规模、视点、杂乱的背景、照明条件等方面有很大的变化。101 个动作类别中的视频被分成 25 组，每组中每一个动作会包含 4～7 个视频。同一组的视频可能有一些共同的特点，比如相似的背景，相似的观点等。

UCF101 解压后就是分类数据集的标准目录格式，二级目录名为人类活动类别也就是视频的标签，二级目录下就是对应的视频数据。每个短视频时长不等(零到十几秒都有)，分辨率为 320×240，帧率不固定，一般为 25 帧或 29 帧，一个视频中只包含一类动作行为。

预处理时，需要将 UCF101 中的视频数据逐帧分解为图像。相同的活动下，有不同的视频是截取自同一个长视频的片段，即视频中的人物和背景等特征基本相似。因此为了避免此类视频被分别划分到 train 和 test 集合引起训练效果不合实际而精度过高，UCF 提供了标准的 train 和 test 集合检索文件，有三种数据集划分方案。

2. 数据集下载

UCF101 数据集可通过以下链接下载：https://www. crcv. ucf. edu/data/UCF101/UCF101. rar。

步骤 3：视频预处理与加载

ECO 跟 TSN 网络在数据加载部分一样，需要将一个视频片段的多张视频帧作为输入，数据加载部分不再赘述，详见 5.1 节数据加载部分。

步骤 4：搭建 ECO 网络

ECO 的网络结构分为 2DNet、3DNet 和 2DNets 三个部分，如图 5-2-3 所示，其中 2D

Net 用于从视频帧中提取特征，3DNet 和 2DNets 用于融合各个视频帧的特征。

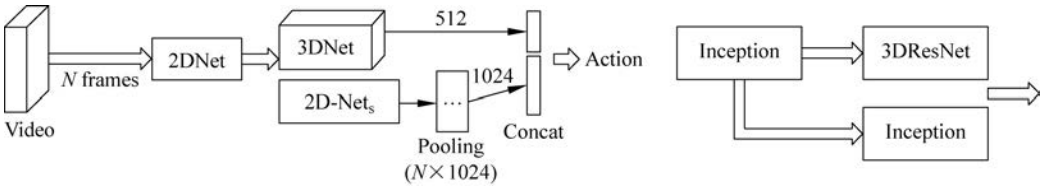


图 5-2-3　ECO 网络结构简图

2DNet 和 2DNets 采用的是 BN-Inception 架构，其中 2DNet 采用的 BN-Inception 架构的第一部分，即 Inception(3a)、Inception(3b)、Inception(3c)；2Dnets 采用则是 Inception(4a)层到最后的池化层。3D Net 的网络结构如图 5-2-4 所示，为 3D-Resnet18。

layer name	output size	2D-Net(H_{2D})	layer name	output size	3D-Net(H_{3D})
conv1_x	112×112	[2D conv 7×7 64]	conv3_x	28×28×N	[3D conv 3×3×3 128 3D conv 3×3×3 128] ×2
pool1	56×56	[max pool 3×3]	conv4_x	14×14×[N/2]	[3D conv 3×3×3 256 3D conv 3×3×3 256] ×2
conv2_x	56×56	[2D conv 3×3 192]	conv5_x	7×7×[N/4]	[3D conv 3×3×3 512 3D conv 3×3×3 512] ×2
pool2	28×28	[max pool 3×3]		1×1×1	pooling, "#c"-d fc, softmax
inception(3a)	28×28	[−256]	—	—	—
inception(3b)	28×28	[−320]	—	—	—
inception(3c)	28×28	[−96]	—	—	—

图 5-2-4　3D-Resnet18 网络

在本次实验中应用的 API 接口如下。

```
paddle.nn.Conv3D(in_channels,
                 out_channels,
                 kernel_size,
                 stride = 1,
                 padding = 0,
                 dilation = 1,
                 groups = 1,
                 padding_mode = 'zeros',
                 weight_attr = None,
                 bias_attr = None,
                 data_format = 'NCDHW'):
```

该 OP 是三维卷积层（convolution3D layer），根据输入、卷积核、步长（stride）、填充（padding）、空洞大小（dilations）一组参数计算得到输出特征层大小。输入和输出是 NCDHW 或 NDHWC 格式，其中 N 是批尺寸，C 是通道数，D 是特征层深度，H 是特征层高度，W 是特征层宽度。三维卷积(Convlution3D)和二维卷积(Convlution2D)相似，但多了一维深度信息(depth)。如果 bias_attr 不为 False，卷积计算会添加偏置项。

- **in_channels**(int)：输入图像的通道数。
- **out_channels**(int)：由卷积操作产生的输出的通道数。

- **kernel_size**（int|list|tuple）：卷积核大小。可以为单个整数或包含三个整数的元组或列表，分别表示卷积核的深度、高和宽。如果为单个整数，表示卷积核的深度、高和宽都等于该整数。
- **stride**（int|list|tuple，可选）：步长大小。可以为单个整数或包含三个整数的元组或列表，分别表示卷积沿着深度、高和宽的步长。如果为单个整数，表示沿着高和宽的步长都等于该整数。默认值：1。
- **padding**（int|list|tuple|str，可选）：填充大小。如果它是一个字符串，可以是"VALID"或者"SAME"，表示填充算法，计算细节可参考上述 padding = "SAME" 或 padding = "VALID" 时的计算公式。如果它是一个元组或列表，它可以有 3 种格式。①包含 5 个二元组：当 data_format 为"NCDHW"时为 [[0,0]，[0,0]，[padding_depth_front，padding_depth_back]，[padding_height_top，padding_height_bottom]，[padding_width_left，padding_width_right]]，当 data_format 为"NDHWC"时为[[0,0]，[padding_depth_front，padding_depth_back]，[padding_height_top，padding_height_bottom]，[padding_width_left，padding_width_right]，[0,0]]。②包含 6 个整数值：[padding_depth_front，padding_depth_back，padding_height_top，padding_height_bottom，padding_width_left，padding_width_right]。③包含 3 个整数值：[padding_depth，padding_height，padding_width]，此时 padding_depth_front = padding_depth_back = padding_depth，padding_height_top = padding_height_bottom = padding_height，padding_width_left = padding_width_right = padding_width。若为一个整数，padding_depth = padding_height= padding_width = padding。默认值：0。
- **dilation**（int|list|tuple，可选）：空洞大小。可以为单个整数或包含三个整数的元组或列表，分别表示卷积核中的元素沿着深度、高和宽的空洞。如果为单个整数，表示深度、高和宽的空洞都等于该整数。默认值：1。
- **groups**（int，可选）：三维卷积层的组数。根据 Alex Krizhevsky 的深度卷积神经网络（CNN）论文中的成组卷积。当 group=n 时，输入和卷积核分别根据通道数量平均分为 n 组，第一组卷积核和第一组输入进行卷积计算，第二组卷积核和第二组输入进行卷积计算……第 n 组卷积核和第 n 组输入进行卷积计算。默认值：1。
- **padding_mode**（str，可选）：填充模式。包括 'zeros'，'reflect'，'replicate' 或者 'circular'。默认值：'zeros'。
- **weight_attr**（ParamAttr，可选）：指定权重参数属性的对象。默认值为 None，表示使用默认的权重参数属性。
- **bias_attr**（ParamAttr|bool，可选）：指定偏置参数属性的对象。若 bias_attr 为 bool 类型，只支持为 False，表示没有偏置参数。默认值为 None，表示使用默认的偏置参数属性。
- **data_format**（str，可选）：指定输入的数据格式，输出的数据格式将与输入保持一致，可以是"NCDHW"和"NDHWC"。N 是批尺寸，C 是通道数，D 是特征深度，H 是特征高度，W 是特征宽度。默认值："NCDHW"。

```
paddle.nn.BatchNorm3D(num_features,
                      momentum = 0.9,
                      epsilon = 1e - 05,
                      weight_attr = None,
                      bias_attr = None,
                      data_format = 'NCDHW',
                      name = None):
```

该接口用于构建 BatchNorm3D 类的一个可调用对象。可以处理 4D 的 Tensor，实现了批归一化层(Batch Normalization Layer)的功能，可用作卷积和全连接操作的批归一化函数，根据当前批次数据按通道计算的均值和方差进行归一化。

- num_features(int)：指明输入 Tensor 的通道数量。
- epsilon(float，可选)：为了数值稳定加在分母上的值。默认值：1×10^{-5}。
- momentum(float，可选)：此值用于计算 moving_mean 和 moving_var。默认值：0.9。
- weight_attr(ParamAttr|bool，可选)：指定权重参数属性的对象。如果为 False，则表示每个通道的伸缩固定为 1，不可改变。默认值为 None，表示使用默认的权重参数属性。
- bias_attr(ParamAttr，可选)：指定偏置参数属性的对象。如果为 False，则表示每一个通道的偏移固定为 0，不可改变。默认值为 None，表示使用默认的偏置参数属性。
- data_format(string，可选)：指定输入数据格式，数据格式可以为"NCDHW"。默认值为"NCDHW"。
- name(string，可选)：BatchNorm 的名称，默认值为 None。

```
paddle.nn.AdaptiveAvgPool3D(output_size,
                            data_format = 'NCDHW',
                            name = None):
```

该算子根据输入 x，output_size 等参数对一个输入 Tensor 计算 3D 的自适应平均池化。输入和输出都是 5-D Tensor，默认是以“NCDHW”格式表示的，其中 N 是 batch size，C 是通道数，D 是特征图长度，H 是输入特征的高度，W 是输入特征的宽度。

- output_size(int|list|tuple)：算子输出特征图的尺寸，如果其是 list 或 turple 类型的数值，必须包含三个元素，即 D，H 和 W。D，H 和 W 既可以是 int 类型值，也可以是 None，None 表示与输入特征尺寸相同。
- data_format(str，可选)：输入和输出的数据格式，可以是"NCDHW"和"NDHWC"。N 是批尺寸，C 是通道数，D 是特征长度，H 是特征高度，W 是特征宽度。默认值为"NCDHW"。
- name(str，可选)：操作的名称(可选，默认值为 None)。

3DNet：3DNet 采用的是 3D-Resnet18 的网络结构，首先搭建 3D-Resnet18 的基础结构 ConvBNLayer_3d。

ConvBNLayer_3d 与前面章节的 ConvBNLayer 相似，都是继承 paddle.nn.Layer，对于输入的特征先后经过 paddle.nn.Conv3D 和 paddle.nn.BatchNorm3D 进行 3D 卷积和 3D 的 BatchNorm。

```
class ConvBNLayer_3d(nn.Layer):
    def __init__(self,
                 name_scope,
                 num_channels,
                 num_filters,
                 filter_size,
                 stride = 1,
                 groups = 1,
                 act = None):
        super(ConvBNLayer_3d, self).__init__(name_scope)
        self._conv = Conv3D(
            in_channels = num_channels,
            out_channels = num_filters,
            kernel_size = filter_size,
            stride = stride,
            padding = (filter_size - 1) // 2,
            groups = groups,
            bias_attr = False)
        self._batch_norm = BatchNorm3D (num_filters, act = act)
    def forward(self, inputs):
        y = self._conv(inputs)
        y = self._batch_norm(y)
        return y
```

BottleneckBlock_3d 用于构建 3D 残差块。与 2D 的残差块相似，将输入的特征顺序经过两次 Conv3D＋BN 后，与原始输入的特征相加，构成跳跃链接的残差结构。其中，根据输入的 shortcut 参数，选择直接与原始输入特征相加还是原始输入特征经过卷积后再相加。

```
class BottleneckBlock_3d(nn.Layer):
    def __init__(self,name_scope,num_channels, num_filters,stride,
shortcut = True):
        super(BottleneckBlock_3d, self).__init__(name_scope)
        self.conv0 = ConvBNLayer_3d(self.full_name(),
num_channels = num_channels,num_filters = num_filters,filter_size = 3,act = 'relu')
        self.conv1 = ConvBNLayer_3d(self.full_name(),
            num_channels = num_filters,num_filters = num_filters,filter_size = 3,
            stride = stride,act = 'relu')
        if not shortcut:
            self.short = ConvBNLayer_3d(self.full_name(),
            num_channels = num_channels,num_filters = num_filters,filter_size = 3,
stride = stride)
        self.shortcut = shortcut
        self._num_channels_out = num_filters
    def forward(self, inputs):
        y = self.conv0(inputs)
        conv1 = self.conv1(y)
        if self.shortcut:
            short = inputs
        else:
            short = self.short(inputs)
        y = paddle.add(x = short, y = conv1)
        layer_helper = paddle.incubate.LayerHelper(self.full_name(), act = 'relu')
        return layer_helper.append_activation(y)
```

ResNet3D类用于构建3D-Resnet18网络。根据3D-Resnet18的网络结构，先后经过3组（每组两层卷积）卷积核数目分别为128、256、512的3D残差块。最后通过paddle.nn.AdaptiveAvgPool3D实现3D的平均池化，得到最后的特征。

```
class ResNet3D(nn.Layer):
    def __init__(self, name_scope, channels, modality="RGB"):
        super(ResNet3D, self).__init__(name_scope)
        self.modality = modality
        self.channels = channels
        self.pool3d = nn.AdaptiveAvgPool3D(output_size=1)
        depth_3d = [2, 2, 2] # part of 3dresnet18
        num_filters_3d = [128, 256, 512]
        self.bottleneck_block_list_3d = []
        num_channels_3d = self.channels
        for block in range(len(depth_3d)):
            shortcut = False
            for i in range(depth_3d[block]):
                bottleneck_block = self.add_sublayer(
                    'bb_%d_%d' % (block, i),
                    BottleneckBlock_3d(
                        self.full_name(),
                        num_channels=num_channels_3d,
                        num_filters=num_filters_3d[block],
                        stride=2 if i == 0 and block != 0 else 1,
                        shortcut=shortcut))
                num_channels_3d = bottleneck_block._num_channels_out
                self.bottleneck_block_list_3d.append(bottleneck_block)
                shortcut = True
    def forward(self, inputs, label=None):
        y = inputs
        for bottleneck_block in self.bottleneck_block_list_3d:
            y = bottleneck_block(y)
        y = self.pool3d(y)
        return y
```

构建完3DNet后，接下里要构建2DNet和2DNets，继而构建整个ECO网络结构。ConvBNLayer构建一个卷积+BN的操作作为搭建ECO的2D网络的基础模块，具体详见5.1节卷积+BN部分。

LinConPoo类是实现BN-Inception网络结构的基础结构。LinConPoo类根据输入的列表内容，依次根据列表中的网络层搭建网络结构。

```
class LinConPoo(Layer):
    def __init__(self, sequence_list):
        '''
          实际上该类是用于'ConvBNLayer', `Conv2D`, `AvgPool2D`, `MaxPool2D`, `Linear`的排列组合
        super(LinConPoo, self).__init__()
        self.__sequence_list = copy.deepcopy(sequence_list)
        self._layers_squence = Sequential()
        self._layers_list = []
        LAYLIST = [ConvBNLayer, Conv2D, Linear, AvgPool2D, MaxPool2D]
        for i, layer_arg in enumerate(self.__sequence_list):
            if isinstance(layer_arg, dict):
```

```
                layer_class = layer_arg.pop('type')
                # 实例化该层对象
                layer_obj = layer_class(**layer_arg)
            elif isinstance(layer_arg, list):
                layer_class = layer_arg.pop(0)
                # 实例化该层对象
                layer_obj = layer_class(*layer_arg)
            else:
                raise ValueError("sequence_list 中, 每一个元素必须是列表或字典")
            # 指定该层的名字
            layer_name = layer_class.__name__ + str(i)
            # 将每一层添加到 `self._layers_list` 中
            self._layers_list.append((layer_name, layer_obj))
            self._layers_squence.add_sublayer(*(layer_name, layer_obj))
        self._layers_squence = Sequential(*self._layers_list)
    def forward(self, inputs, show_shape=False):
        return self._layers_squence(inputs)
```

接下来我们要构建整个 BN-Inception 网络的各个模块，如图 5-2-5 所示，BN-Inception 由 Inception(3a)、Inception(3b)、Inception(3c)、Inception(4a)、Inception(4b)、Inception(4c)、Inception(4d)、Inception(4e)、Inception(5a)、Inception(5b)多个模块组成，可以看到除了 Inception(3c)、Inception(4c)、Inception(5a)、Inception(5b)之外，其他层之间只是通道数目上的差异，而 Inception(3c)类、Inception(4c)类、Inception(5a)类、Inception(5b)类则采用了不同池化方式和步长。因此在这部分我们需要分别构建 Inception 类、Inception(3c)类、Inception(4c)类、Inception(5a)类、Inception(5b)类。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	double #3×3 reduce	double #3×3	Pool+proj
convolution*	7×7/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	1		64	192			
max pool	3×3/2	28×28×192	0						
inception(3a)		28×28×256	3	64	64	64	64	96	avg+32
inception(3b)		28×28×320	3	64	64	96	64	96	avg+64
inception(3c)	stride 2	28×28×576	3	0	128	160	64	96	max+pass through
inception(4a)		14×14×576	3	224	64	96	96	128	avg+128
inception(4b)		14×14×576	3	192	96	128	96	128	avg+128
inception(4c)		14×14×576	3	160	128	160	128	160	avg+128
inception(4d)		14×14×576	3	96	128	192	160	192	avg+128
inception(4e)	stride 2	14×14×1024	3	0	128	192	192	256	max+pass through
inception(5a)		7×7×1024	3	352	192	320	160	224	avg+128
inception(5b)		7×7×1024	3	352	192	320	192	224	max+128
avg pool	7×7/1	1×1×1024	0						

图 5-2-5　BN-Inception 网络结构

Inception 类用于构建 BN-Inception 网络的绝大多数模块。其结构如图 5-2-6 所示，输入的特征并行地进行 1×1 卷积＋3×3 卷积＋3×3 卷积、1×1＋3×3 卷积、池化＋1×1 卷

积和1×1卷积四个支路，并将4个支路的特征融合。

其中num_channels表示传入特征通道数；ch1x1表示1×1卷积操作的输出通道数；ch3x3reduced表示3×3卷积之前的1×1卷积的通道数；ch3x3表示3×3卷积操作的输出通道数；doublech3x3reduce表示两个3×3卷积叠加之前的1×1卷积的通道数；doublech3x3_1表示第一个3×3卷积操作的输出通道数；doublech3x3_2表示第二个3×3卷积操作的输出通道数；pool_proj表示池化操作之后1×1卷积的通道数。

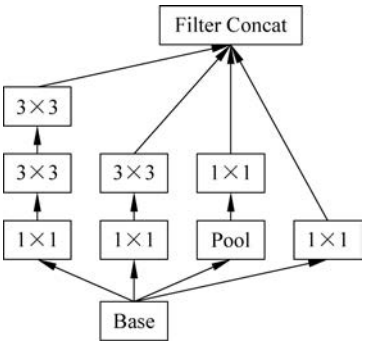


图5-2-6 Inception类结构示意

```
class Inception(nn.Layer):
    def __init__(self, num_channels, ch1x1, ch3x3reduced, ch3x3, doublech3x3reduced,
doublech3x3_1, doublech3x3_2,pool_proj):
        super(Inception, self).__init__()
        branch1_list = [
            {'type': ConvBNLayer, 'num_channels': num_channels, 'num_filters': ch1x1, 'filter_
size': 1, 'stride': 1,
             'padding': 0, 'act': 'relu'}]
        self.branch1 = LinConPoo(branch1_list)
        branch2_list = [
            {'type': ConvBNLayer, 'num_channels': num_channels, 'num_filters': ch3x3reduced,
'filter_size': 1,'stride': 1, 'padding': 0, 'act': 'relu'},
            {'type': ConvBNLayer, 'num_channels': ch3x3reduced, 'num_filters': ch3x3, 'filter_
size': 3, 'stride': 1,'padding': 1, 'act': 'relu'},]
        self.branch2 = LinConPoo(branch2_list)
        branch3_list = [
            {'type': ConvBNLayer, 'num_channels': num_channels, 'num_filters': doublech3x3reduced, 'filter_size': 1, 'stride': 1, 'padding': 0, 'act': 'relu'},
            {'type': ConvBNLayer, 'num_channels': doublech3x3reduced, 'num_filters': doublech3x3_1, 'filter_size': 3, 'stride': 1, 'padding': 1, 'act': 'relu'},
            {'type': ConvBNLayer, 'num_channels': doublech3x3_1, 'num_filters': doublech3x3_
2, 'filter_size': 3,
             'stride': 1, 'padding': 1, 'act': 'relu'},]
        self.branch3 = LinConPoo(branch3_list)
        branch4_list = [
            {'type': AvgPool2D, 'kernel_size': 3, 'stride': 1, 'padding': 1},
            {'type': ConvBNLayer, 'num_channels': num_channels, 'num_filters': pool_proj,
'filter_size': 1, 'stride': 1,'padding': 0, 'act': 'relu'},]
        self.branch4 = LinConPoo(branch4_list)
    def forward(self, inputs):
        branch1 = self.branch1(inputs)
        branch2 = self.branch2(inputs)
        branch3 = self.branch3(inputs)
        branch4 = self.branch4(inputs)
        outputs = paddle.concat([branch1, branch2, branch3, branch4], axis=1)
        return outputs
```

Inception(3c)类、Inception(4c)类、Inception(5a)类、Inception(5b)类与Inception类整体上比较相似。通过多个LinConPoo实例实现，仅在结构上有一些差异，在本部分不展开描

述，详细可参照 Inception 类。

接下来搭建整个的 ECO 网络。这部分主要通过之前定义好的 Resnet3D、Inception 的各个模块类，实现搭建 CEO 的整体网络结构的搭建。

首先是 BN-Inception 网络的部分，在 init() 函数中，我们通过 Inception 类，实例化 inception_3a、inception_3b、inception_4a、inception_4b、inception_4c、inception_4d 的结构；通过 Inception3c 类、Inception4e 类、Inception5a 类、Inception5b 类分别实现 inception_3c、inception_4e、inception_5a、inception_5b 的实例化。

```
class GoogLeNet (nn.Layer):
    def __init__(self, class_dim = 101, seg_num = 12, seglen = 1, modality = "RGB", weight_
devay = None):
        self.seg_num = seg_num
        self.seglen = seglen
        self.modality = modality
        self.channels = 3 * self.seglen if self.modality == "RGB" else 2 * self.seglen
        super(GoogLeNet, self).__init__()
        part1_list = [
            {'type': ConvBNLayer, 'num_channels': self.channels, 'num_filters': 64, 'filter_
size': 7, 'stride': 2,
             'padding': 3, 'act': 'relu'},
            {'type': MaxPool2D, 'kernel_size': 3, 'stride': 2, 'padding': 1},]
        part2_list = [
            {'type': ConvBNLayer, 'num_channels': 64, 'num_filters': 64, 'filter_size': 1,
'stride': 1, 'padding': 0,'act': 'relu'},
            {'type': ConvBNLayer, 'num_channels': 64, 'num_filters': 192, 'filter_size': 3,
'stride': 1, 'padding': 1,'act': 'relu'},
            {'type': MaxPool2D, 'kernel_size': 3, 'stride': 2, 'padding': 1},]
        self.googLeNet_part1 = Sequential(
            ('part1', LinConPoo(part1_list)),
            ('part2', LinConPoo(part2_list)),
            ('inception_3a', Inception(192, 64, 64, 64, 64, 96, 96, 32)), ('inception_3b',
Inception(256, 64, 64, 96, 64, 96, 96, 64)), )
        self.before3d = Sequential(
            ('Inception3c', Inception3c(320, 128, 160, 64, 96, 96))
        )
        self.googLeNet_part2 = Sequential(
            ('inception_4a', Inception(576, 224, 64, 96, 96, 128, 128, 128)),
            ('inception_4b', Inception(576, 192, 96, 128, 96, 128, 128, 128)),
            ('inception_4c', Inception(576, 160, 128, 160, 128, 160, 160, 128)),
            ('inception_4d', Inception(608, 96, 128, 192, 160, 192, 192, 128)),
        )
        self.googLeNet_part3 = Sequential(
            ('inception_4e', Inception4e(608, 128, 192, 192, 256, 256, 608)),
            ('inception_5a', Inception5a(1056, 352, 192, 320, 160, 224, 224, 128)),
            ('inception_5b', Inception5b(1024, 352, 192, 320, 192, 224, 224, 128)),
            ('AvgPool1', AdaptiveAvgPool2D(1)), # [2,1024,1,1]
        )
```

然后，通过 Res3D 类实现 3DResNet 网络的实例化，并生成用于分类的全连接层。

```
self.res3d = Res3D.ResNet3D('resnet', modality = 'RGB', channels = 96)
                                                    # channel 数与 2D 网络输出 channel 数一致
```

```
self.dropout1 = nn.Dropout(p = 0.5)
self.softmax = nn.Softmax()
self.out = nn.Linear(in_features = 1536, out_features = class_dim,
weight_attr = paddle.framework.ParamAttr(initializer = paddle.nn.initializer.XavierNormal())
self.dropout2 = nn.Dropout(p = 0.6)
self.out_3d = []
```

在 forward()函数中，构建前向传播的过程。对于输入的多帧图像，与 TSN 一样，首先通过 reshape 融合在一起，再依次通过 inception_3a、inception_3b、inception_3c 提取特征。得到的特征分别输入到 ResNet3D 和 BN-Inception 剩余的结构中去。最后将两部分得到的特征进行融合，输出属于每个类的概率。

```
def forward(self, inputs, label = None):
    inputs = paddle.reshape(inputs, [ - 1, inputs.shape[2], inputs.shape[3], inputs.shape[4]])
    googLeNet_part1 = self.googLeNet_part1(inputs)
    googleNet_b3d, before3d = self.before3d(googLeNet_part1)
    if len(self.out_3d) == self.seg_num:
        self.out_3d[:self.seg_num - 1] = self.out_3d[1:]
        self.out_3d[self.seg_num - 1] = before3d
        for input_old in self.out_3d[:self.seg_num - 1]:
            input_old.stop_gradient = True
    else:
        while len(self.out_3d) < self.seg_num:
            self.out_3d.append(before3d)
    y_out_3d = self.out_3d[0]
    for i in range(len(self.out_3d) - 1):
        y_out_3d = paddle.concat([y_out_3d, self.out_3d[i + 1]], axis = 0)
          y_out_3d = paddle.reshape(y_out_3d, [ - 1, self.seg_num, y_out_3d.shape[1], y_out_3d.
shape[2], y_out_3d.shape[3]])
          y_out_3d = paddle.reshape(y_out_3d, [y_out_3d.shape[0], y_out_3d.shape[2], y_out
_3d.shape[1], y_out_3d.shape[3], y_out_3d.shape[4]])
    out_final_3d = self.res3d(y_out_3d)
    out_final_3d = paddle.reshape(out_final_3d, [ - 1, out_final_3d.shape[1]])
    out_final_3d = self.dropout1(out_final_3d)
    out_final_3d = paddle.reshape(out_final_3d, [ - 1, self.seg_num, out_final_3d.shape[1]])
    out_final_3d = paddle.mean(out_final_3d, axis = 1)
    googLeNet_part2 = self.googLeNet_part2(googleNet_b3d)
    googLeNet_part3 = self.googLeNet_part3(googLeNet_part2)
    googLeNet_part3 = self.dropout2(googLeNet_part3)
    out_final_2d = paddle.reshape(googLeNet_part3, [ - 1, googLeNet_part3.shape[1]])
    out_final_2d = paddle.reshape(out_final_2d, [ - 1, self.seg_num, out_final_2d.shape[1]])
    out_final_2d = paddle.mean(out_final_2d, axis = 1)
    out_final = paddle.concat([out_final_2d, out_final_3d], axis = 1)
    out_final = self.out(out_final)
    if label is not None:
        acc = paddle.metric.Accuracy().compute(out_final, label)
        return out_final, acc
    else:
        return out_final
```

步骤 5：训练 ECO 网络

接下来实现 ECO 网络的训练过程。首先，依次加载配置文件，实例化网络，实例化优化

器，创建训练数据读取器，创建验证数据读取器，创建优化器，定义损失函数。然后，通过traindataloder 类，加载视频图像和对应的标注送入 ECO 网络，计算精度和损失。在进行反向转播和优化器优化后就完成了一次的迭代训练。

```
def train(args):
    paddle.set_device('gpu')                                          #使用 gpu 进行训练
    config = parse_config(args.config)                                #读取输入的参数
    train_config = merge_configs(config, 'train', vars(args))         #将输入的参数与配置文
    train_model = ECO.GoogLeNet(train_config['MODEL']['num_classes'],
                      train_config['MODEL']['seg_num'],
                      train_config['MODEL']['seglen'], 'RGB', 0.00002)
    opt = paddle.optimizer.Momentum(0.005, 0.9, parameters=train_model.parameters())
    train_reader = KineticsReader(args.model_name.upper(), 'train', train_config)
    traindataloder = paddle.io.DataLoader(train_reader, batch_size=train_config['TRAIN']
['batch_size'],
                num_workers=0, drop_last=True, collate_fn=train_reader.collate_fn,batch_
sampler=None)
    epochs = args.epoch or train_model.epoch_num()
    #定义损失函数计算方式
    CrossEntropyLoss=nn.CrossEntropyLoss(reduction='mean')
    for i in range(epochs):
        train_model.train()
        for batch_id, data in enumerate(traindataloder):
            img = data[0].astype('float32')
            label = data[1].astype('int64')
            label.stop_gradient = True
            out, acc = train_model(img, label)                        #前向传播得到结果
            if out is not None:
                avg_loss = CrossEntropyLoss(out, label)               #计算损失值
                avg_loss.backward()                                   #反向传播
                opt.step()
                opt.clear_grad()
                if batch_id % 200 == 0:
                    #每迭代 200 次，保存一次模型
                    paddle.save(train_model.state_dict(), args.save_dir + '/ucf_model_v2/
gen_b2a.pdparams')
```

步骤 6：视频预测

模型预测的部分与训练过程相似，但不再需要损失计算和梯度反向传播。

```
def eval(args):
    config = parse_config(args.config)                                #读取输入的参数
    val_config = merge_configs(config, 'test', vars(args))            #将输入的参数与配置文件中
                                                                        的参数进行合并
    paddle.set_device('gpu')                                          #使用 gpu 进行预测
    val_model = ECO.GoogLeNet(val_config['MODEL']['num_classes'],
                    val_config['MODEL']['seg_num'],
                    val_config['MODEL']['seglen'], 'RGB')             #创建测试网络
    label_dic = np.load('label_dir.npy', allow_pickle=True).item()
    label_dic = {v: k for k, v in label_dic.items()}
    val_reader = KineticsReader(args.model_name.upper(), 'test', val_config)
```

```
    valdataloder = paddle.io.DataLoader(val_reader, batch_size = val_config['TEST']['batch_size'],
            num_workers = 0, collate_fn = val_reader.collate_fn, batch_sampler = None)
    weights = args.weights
    model = paddle.load(weights)
    val_model.set_state_dict(model)
    val_model.eval()
    acc_list = []
    for batch_id, data in enumerate(valdataloder):
        dy_x_data = data[0].astype('float32')
        y_data = data[1].astype('int64')
        img = paddle.to_tensor(dy_x_data)
        label = paddle.to_tensor(y_data)
        label.stop_gradient = True
        out, acc = val_model(img, label)                    #进行前向传播,预测结果
        acc_list.append(acc.numpy()[0])
print("测试集准确率为:{}".format(np.mean(acc_list)))
```

至此，我们就完成了 ECO 网络的搭建、训练和预测过程，你学会了吗？

基于 TimeSformer 模型的视频分类

5.3 实践三：基于 TimeSformer 模型的视频分类

在前面的实践中，我们通过 Transformer 结构实现了图像分类、目标检测和图像分类。这不禁使我们思考，视频是由一系列的图像帧组成的，那么 Transformer 是否在视频分类领域也能有所应用呢？在本次实践中，我们将实现 Transformer 在视频分类领域的经典算法 TimeSformer，从而在 UCF101 数据集上进行视频分类。

TimeSformer 是 Facebook AI 于 2021 年提出的无卷积视频分类方法，将标准的 Transformer 体系结构适应于视频分类。视频任务与图像不同，不仅包含空间信息，还包含时间信息。TimeSformer 针对这一特性，对一系列的帧级图像块进行时空特征提取，从而适配视频任务。TimeSformer 在多个行为识别基准测试中达到了 SOTA 效果，其中包括 TimeSformer-L，以更短的训练用时（Kinetics-400 数据集训练用时 39 小时）在 Kinetics-400 上达到了 80.7%的准确率，超过了经典的基于 CNN 的视频分类模型 TSN、TSM 及 Slowfast 方法。而且，与 3D 卷积网络相比，TimeSformer 的模型训练速度更快，也拥有更高的测试效率。

步骤 1：UCF101 数据集预处理与加载

1. 数据集概览

在本次实践中，我们依旧使用视频分类 UCF101 的数据集。数据集格式如图 5-3-1 所示，UCF101 目录下存放着用每一个类名命名的文件夹，每个文件夹下为对应该类别的视频片段。

ucf101_train_video.txt 和 ucf101_val_videos.txt 两个文档分别存储用于训练和验证的视频，其部分内容如图 5-3-2 所示，每行为一个样本数据，分别记录视频的路径和视频所属类别，中间用空格隔开。

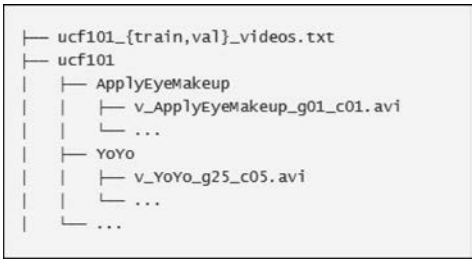


图 5-3-1　UCF101 数据集结构

```
1   ucf101/ApplyEyeMakeup/v_ApplyEyeMakeup_g01_c01 0
2   ucf101/ApplyEyeMakeup/v_ApplyEyeMakeup_g01_c02 0
3   ucf101/ApplyEyeMakeup/v_ApplyEyeMakeup_g01_c03 0
4   ucf101/ApplyEyeMakeup/v_ApplyEyeMakeup_g01_c04 0
```

图 5-3-2　训练、验证文档示例

2. 数据处理与加载

在进行数据处理的过程中，我们需要对视频进行解帧、抽帧，将视频转换成一系列的图像。在这里我们通过 VideoDecoder 类来实现解帧和抽帧的过程（该部分代码主要为基础的视频解帧处理，就不展开展示了）。

```
class VideoDecoder(object):
  """
  Decode mp4 file to frames.
  Args:
    filepath: the file path of mp4 file
  """
  def __init__(self,
           backend = 'pyav',
           mode = 'train',
           sampling_rate = 32,
           num_seg = 8,
           num_clips = 1,
           target_fps = 30):
    ......
  def __call__(self, results):
    """
    Perform mp4 decode operations.
    return:
      List where each item is a numpy array after decoder.
    """
    ......
```

Sampler 类以 VideoDecoder 类解帧后的图像为输入，对视频进行分段，并在每段中抽取指定数目的视频帧（该部分代码主要为基础的数据操作，就不展开展示了）。

```
class Sampler(object):
  def __init__(self,
           num_seg,
           seg_len,
           valid_mode = False,
           select_left = False,
           dense_sample = False,
           linspace_sample = False):
    ......

  def __call__(self, results):
```

```
    """
    Args:
      frames_len: length of frames.
    return:
      sampling id.
    """
    ......
```

将视频处理成图像之后，要对每个图象进行预处理的操作。其中包括：通过Normalization类实现图像归一化；通过Image2Array类将图像由PIL.Image格式转化为numpy array格式；通过JitterScale类将图像的短边随机resize到min_size至max_size之间的某一数值，长边等比例缩放；通过RandomCrop和UniformCrop对图像进行不同方式的裁剪；通过RandomFlip对图像进行随机翻转。

定义完各种用于数据处理的类后，我们通过Compose类来实现由视频到处理后图像序列的转换。

```
class Compose(object):
  def __init__(self, train_mode=False):
    self.pipelines = []
    if train_mode:
      self.pipelines.append(VideoDecoder(mode='train'))
      self.pipelines.append(Sampler(num_seg=8, seg_len=1, valid_mode=False, linspace_
sample=True))
    else:
      self.pipelines.append(VideoDecoder(mode='test'))
      self.pipelines.append(Sampler(num_seg=8, seg_len=1, valid_mode=True, linspace_
sample=True))
    self.pipelines.append(Normalization(mean=[0.45, 0.45, 0.45], std=[0.225, 0.225,
0.225], tensor_shape=[1, 1, 1, 3]))
    self.pipelines.append(Image2Array(data_format='cthw'))
    if train_mode:
      self.pipelines.append(JitterScale(min_size=256, max_size=320))
      self.pipelines.append(RandomCrop(target_size=224))
      self.pipelines.append(RandomFlip())
    else:
      self.pipelines.append(JitterScale(min_size=224, max_size=224))
      self.pipelines.append(UniformCrop(target_size=224))
```

接下来我们继承paddle的Dataset来构建一个数据读取器VideoDataset类，在迭代的过程中通过getitem()函数调用prepare_train()和prepare_test()函数来加载训练、验证和测试过程中所需要的数据。需要注意的是，这里我们返回的每个数据是从一个视频片段中抽取的多帧图像。

```
class VideoDataset(paddle.io.Dataset):
  def __init__(self, file_path, pipeline, num_retries=5, suffix='', test_mode=False):
    self.file_path = file_path
    self.pipeline = pipeline
    self.num_retries = num_retries
    self.suffix = suffix
    self.info = self.load_file()
    self.test_mode = test_mode
```

```
    super(VideoDataset, self).__init__()
  def load_file(self):
    """Load index file to get video information."""
    ......
  def prepare_train(self, idx):
    """TRAIN & VALID. Prepare the data for training/valid given the index."""
    for ir in range(self.num_retries):
      results = copy.deepcopy(self.info[idx])
      results = self.pipeline(results)
    return results['imgs'], np.array([results['labels']])
  def prepare_test(self, idx):
    """TEST. Prepare the data for test given the index."""
    for ir in range(self.num_retries):
      results = copy.deepcopy(self.info[idx])
      results = self.pipeline(results)
    return results['imgs'], np.array([results['labels']])
  def __getitem__(self, idx):
    """ Get the sample for either training or testing given index"""
    if self.test_mode:
      return self.prepare_test(idx)
    else:
      return self.prepare_train(idx)
```

步骤 2：TimeSformer 模型搭建

TimeSformer 的模型包括三部分内容：主干网络 VIT，TimeSformer 模型的头部预测部分（包括输出层设置和使用的损失函数等）以及将主干网络和头部进行封装的 RecognizerTransformer。接下来我们将分别完成这三部分的实现。

（1）VIT。

在构建 VIT 的过程中，我们要依次实现 MLP、Attention、PatchEmbed 和 Block 类，并在最后通过 VisionTransformer 类来实现整个 VIT 的结构。其中，MLP、Attention 是 Transformer 的基础结构（我们在 2.5 节中已经实现了，在这里就不再重复）。

PatchEmbed 类用于对输入的图像进行 Embedding。因为输入的是视频的图像序列，所以在进行转换前要先对输入数据的维度进行调整，将 Batch 和时间序列合并在一起（在 5.1 节和 5.2 节中我们也采用了一样的操作），由[B，T，C，H，W]变换成[BT，C，H，W]。

```
class PatchEmbed(nn.Layer):
  def __init__(self,
          img_size=224,
          patch_size=16,
          in_channels=3,
          embed_dim=768):
    super().__init__()
    img_size = to_2tuple(img_size)
    patch_size = to_2tuple(patch_size)
    num_patches = (img_size[1] // patch_size[1]) * (img_size[0] //
                                patch_size[0])
    self.img_size = img_size
    self.patch_size = patch_size
```

```
    self.num_patches = num_patches
    self.proj = nn.Conv2D(in_channels,
                          embed_dim,
                          kernel_size=patch_size,
                          stride=patch_size)

  def forward(self, x):
    B, C, T, H, W = x.shape
    x = x.transpose((0, 2, 1, 3, 4))
    x = x.reshape([B * T if B > 0 else -1, C, H, W])
    x = self.proj(x)
    W = x.shape[-1]
    x = x.flatten(2).transpose((0, 2, 1))
    return x, T, W # [BT', nH'nW', embed_dim], T', nW'
```

Block 类是 TimeSformer 模型的核心部分。我们通过 Block 类来实现分开的时空注意力机制（divided space-time attention），因此在 init()函数中，我们要分别实例化时间注意力和空间注意力。

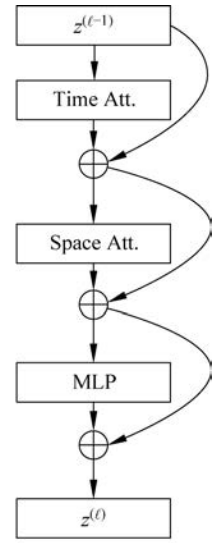
```
class Block(nn.Layer):
  def __init__(self,
               dim,
               num_heads,
               mlp_ratio=4.0,
               qkv_bias=False,
               qk_scale=None,
               drop=0.0,
               attn_drop=0.0,
               drop_path=0.1,
               act_layer=nn.GELU,
               norm_layer='nn.LayerNorm',
               epsilon=1e-5,
               attention_type='divided_space_time'):
    super().__init__()
    self.attention_type = attention_type
    self.norm1 = eval(norm_layer)(dim, epsilon=epsilon)
    self.attn = Attention(dim,
                num_heads=num_heads,
                qkv_bias=qkv_bias,
                qk_scale=qk_scale,
                attn_drop=attn_drop,
                proj_drop=drop)
    # Temporal Attention Parameters
    if self.attention_type == 'divided_space_time':
      self.temporal_norm1 = eval(norm_layer)(dim, epsilon=epsilon)
      self.temporal_attn = Attention(dim,
                           num_heads=num_heads,
                           qkv_bias=qkv_bias,
                           qk_scale=qk_scale,
                           attn_drop=attn_drop,
                           proj_drop=drop)
      self.temporal_fc = nn.Linear(dim, dim)
    self.drop_path = DropPath(drop_path) if drop_path > 0. else Identity()
```

```
        self.norm2 = eval(norm_layer)(dim, epsilon=epsilon)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim,
                       hidden_features=mlp_hidden_dim,
                       act_layer=act_layer,
                       drop=drop)
```

如图 5-3-3 所示，在前向传播的过程中，对于输入的特征先后进行时间注意力、空间注意力和 MLP。其中，在时间注意力中，图像块仅和其余帧对应位置提取出的图像块进行 attention；在空间注意力中，图像块仅和同一帧提取出的图像块进行 attention。

```
    def forward(self, x, B, T, W):
        num_spatial_tokens = (x.shape[1] - 1) // T # nHnW
        H = num_spatial_tokens // W # nH
        ########## Temporal ##########
        xt = x[:, 1:, :] # [B, nHnW*T, embed_dim]
        _b, _h, _w, _t, _m = B, H, W, T, xt.shape[-1]
        xt = xt.reshape([_b * _h * _w if _b>0 else -1, _t, _m])
        res_temporal = self.drop_path( self.temporal_attn(self.temporal_norm1(xt)))
        _b, _h, _w, _t, _m = B, H, W, T, res_temporal.shape[-1]
        res_temporal = res_temporal.reshape([_b, _h * _w * _t, _m])
        res_temporal = self.temporal_fc(res_temporal)
        xt = x[:, 1:, :] + res_temporal
        ########## Spatial ##########
        init_cls_token = x[:, 0, :].unsqueeze(1)
        cls_token = init_cls_token.tile((1, T, 1))
        _b, _t, _m = cls_token.shape
        cls_token = cls_token.reshape([_b * _t, _m]).unsqueeze(1)
        xs = xt
        _b, _h, _w, _t, _m = B, H, W, T, xs.shape[-1]
        xs = xs.reshape([_b, _h, _w, _t, _m]).transpose(
          (0, 3, 1, 2, 4)).reshape([_b * _t if _b>0 else -1, _h * _w, _m])
        xs = paddle.concat((cls_token, xs), axis=1)
        res_spatial = self.drop_path(self.attn(self.norm1(xs)))
        cls_token = res_spatial[:, 0, :]
        _b, _t, _m = B, T, cls_token.shape[-1]
        cls_token = cls_token.reshape([_b, _t, _m])
        cls_token = paddle.mean(cls_token, axis=1, keepdim=True)
        res_spatial = res_spatial[:, 1:, :]
        _b, _t, _h, _w, _m = B, T, H, W, res_spatial.shape[-1]
        res_spatial = res_spatial.reshape([_b, _t, _h, _w, _m]).transpose(
          (0, 2, 3, 1, 4)).reshape([_b, _h * _w * _t, _m])
        res = res_spatial
        x = xt
        x = paddle.concat((init_cls_token, x), axis=1) + paddle.concat(
          (cls_token, res), axis=1)
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x
```

完成所需的各个模块后，接下来通过 VisionTransformer 类来实现 VIT 结构的搭建。这部分与 2.5 节中相似，不同之处在于使用了带有时间 attention 的 block。因此，需要添加与之对应的 Time Embeddings 部分。



Divided Space-Time Attention(T+S)

图 5-3-3　分离的空间注意力和时间注意力

```
class VisionTransformer(nn.Layer):
  """ Vision Transformer with support for patch input
def forward_features(self, x):
......
    # Time Embeddings
    if self.attention_type != 'space_only':
      cls_tokens = x[:B, 0, :].unsqueeze(1) if B > 0 else x.split(T)[0].index_select(paddle.
to_tensor([0]), axis = 1)
      x = x[:, 1:] # [BT, nHnW, embed_dim]
      _bt, _n, _m = x.shape
      _b = B
      _t = _bt // _b if _b != -1 else T
      x = x.reshape([_b, _t, _n, _m]).transpose(
        (0, 2, 1, 3)).reshape([_b * _n if _b > 0 else -1, _t, _m]) # [B * nHnW, T', embed_dim]
      time_interp = (T != self.time_embed.shape[1])
      if time_interp: # T' != T
        time_embed = self.time_embed.transpose((0, 2, 1)).unsqueeze(0)
        new_time_embed = F.interpolate(time_embed,
                        size = (T, x.shape[-1]),
                        mode = 'nearest').squeeze(0)
        x = x + new_time_embed
      else:
        x = x + self.time_embed
      x = self.time_drop(x) # [B * nHnW, T', embed_dim]
      _bn, _t, _m = x.shape
      _b = B
      x = x.reshape([_b, _n * _t, _m] if _n > 0 else [_b, W * W * T, _m])
      x = paddle.concat((cls_tokens, x), axis = 1) # [B, 1 + nHnW * T', embed_dim]
    for blk in self.blocks:
      x = blk(x, B, T, W)
```

```
    x = self.norm(x)
    return x[:, 0] # [B, 1, embed_dim]
```

(2) TimeSformer 模型的头部预测部分。

接下来,通过 TimeSformerHead 来实现 TimeSformer 用于预测的分类层和损失函数。其中,分类层采用的是对应 VIT 输入维度和分类数目的全连接层,损失函数采用的则是分类任务中常见的交叉熵损失。

```
class TimeSformerHead(nn.Layer):
  """TimeSformerHead Head."""
  def __init__(self,
          num_classes,
          in_channels,
          std=0.02,
          ls_eps=0.):

    super().__init__()
    self.std = std
    self.num_classes = num_classes
    self.in_channels = in_channels
    self.fc = Linear(self.in_channels, self.num_classes)
    self.loss_func = paddle.nn.CrossEntropyLoss()
    self.ls_eps = ls_eps

  def forward(self, x):
    score = self.fc(x)
    return score

  def loss(self, scores, labels, valid_mode=False, **kwargs):
    if len(labels) == 1: #commonly case
      labels = labels[0]
      losses = dict()
      if self.ls_eps != 0. and not valid_mode: # label_smooth
        loss = self.label_smooth_loss(scores, labels, **kwargs)
      else:
        loss = self.loss_func(scores, labels, **kwargs)
      top1, top5 = self.get_acc(scores, labels, valid_mode)
      losses['top1'] = top1
      losses['top5'] = top5
      losses['loss'] =
      return losses
    else:
      raise NotImplemented
```

(3) Recognizer Transformer。

定义了 VIT 和 TimeSformer 的预测头部网络后,通过 RecognizerTransformer 类来实现整个 TimeSformer 的网络结构。在__init__()函数中传入 backbone 和 head。当进行前向传播的过程时候,输入的图像序列先通过 backbone 提取特征,再将提取的特征输入 head 就实现了最终的分类。

```
class RecognizerTransformer(nn.Layer):
  """Transformer's recognizer model framework."""
```

```
def __init__(self, backbone = None, head = None):
  super().__init__()
  self.backbone = backbone
  self.backbone.init_weights()
  self.head = head
  self.head.init_weights()
def forward_net(self, imgs):
  if self.backbone != None:
    feature = self.backbone(imgs)
  else:
    feature = imgs
  if self.head != None:
    cls_score = self.head(feature)
  else:
    cls_score = None
  return cls_score
```

步骤 3：模型训练与验证

(1) 准备工作。

在进行模型的训练和验证前，我们需要先实例化在训练和验证过程中所需的数据读取器、网络结构、优化器(训练过程)。在实例化网络结构的过程中，我们首先实例化用于提取特征的 VIT 和用于最后预测结果的 head，然后将 VIT 和 head 作为输入实例化我们的 TimeSformer 网络。

```
timesformer = VisionTransformer(pretrained = pretrained,
                    img_size = img_size,
                    patch_size = patch_size,
                    in_channels = in_channels_backbone,
                    embed_dim = embed_dim,
                    depth = depth,
                    num_heads = num_heads,
                    mlp_ratio = mlp_ratio,
                    qkv_bias = qkv_bias,
                    epsilon = epsilon,
                    seg_num = seg_num,
                    attention_type = attention_type
                    )
head = TimeSformerHead(num_classes = num_classes,
             in_channels = in_channels_head,
             std = std
             )
model = RecognizerTransformer(backbone = timesformer, head = head)
```

数据集的加载则需要通过我们定义的 VideoDataset 类、用于分布式批采样的 paddle.io.DistributedBatchSampler 以及我们常用的用于返回数据的迭代器的 paddle.io.DataLoader 来分别实现训练集和验证集的加载(代码部分仅展示训练集的加载)。

```
train_pipeline = Compose(train_mode = True)
train_dataset = VideoDataset(file_path = train_file_path, pipeline = train_pipeline, suffix =
suffix)
```

```
train_sampler = paddle.io.DistributedBatchSampler(
  train_dataset,
  batch_size = batch_size,
  shuffle = train_shuffle,
  drop_last = True
)
train_loader = paddle.io.DataLoader(
  train_dataset,
  num_workers = num_workers,
  batch_sampler = train_sampler,
  places = paddle.set_device('gpu'),
  return_list = True
)
```

优化器采用的是 Momentum，其中学习率不再设定为固定的值，而是采用的逐步衰减的策略（paddle.optimizer.lr.MultiStepDecay 实现）。

```
lr = paddle.optimizer.lr.MultiStepDecay(learning_rate = learning_rate, milestones =
milestones, gamma = gamma)
optimizer = paddle.optimizer.Momentum(
  learning_rate = lr,
  momentum = momentum,
  parameters = model.parameters(),
  weight_decay = paddle.regularizer.L2Decay(0.0001),
  use_nesterov = True
)
```

（2）模型训练。

完成准备工作之后就可以开始模型的训练了。训练过程由两层循环构成。第一层控制全部数据的训练次数，第二层则完成每次 batch 的训练，依次进行前向传播、反向传播、优化参数和情况梯度。值得注意的是，考虑视频和模型占用的显存较高，这里添加了梯度累加的模式，可以在显存不足的情况下增大 batchsize。

```
for epoch in range(0, epochs):
    model.train()
    record_list = build_record(framework)
    tic = time.time()
    for i, data in enumerate(train_loader):
      record_list['reader_time'].update(time.time() - tic)
      # 4.1 forward
      outputs = model.train_step(data)
      # 4.2 backward
      if use_gradient_accumulation and i == 0:
        optimizer.clear_grad()
      avg_loss = outputs['loss']
      avg_loss.backward()
      # 4.3 minimize
      if use_gradient_accumulation:
        if (i + 1) % num_iters == 0:
          for p in model.parameters():
            p.grad.set_value(p.grad / num_iters)
```

```
            optimizer.step()
            optimizer.clear_grad()
        else:
            optimizer.step()
            optimizer.clear_grad()
    # learning rate epoch step
    lr.step()
```

（3）模型验证。

在模型验证之前，我们先定义用于计算精度的 CenterCropMetric 类。通过 CenterCropMetric 类，可以计算验证过程中的 TOP1 和 TOP5 的精度。

```
class CenterCropMetric(object):
    def __init__(self, data_size, batch_size, log_interval=1):
        super().__init__()
        self.data_size = data_size
        self.batch_size = batch_size
        self.log_interval = log_interval
        self.top1 = []
        self.top5 = []
    def update(self, batch_id, data, outputs):
        """update metrics during each iter"""
        labels = data[1]
        top1 = paddle.metric.accuracy(input=outputs, label=labels, k=1)
        top5 = paddle.metric.accuracy(input=outputs, label=labels, k=5)

        self.top1.append(top1.numpy())
        self.top5.append(top5.numpy())
        if batch_id % self.log_interval == 0:
            print("[TEST] Processing batch {}/{} ...".format(
                batch_id,
                self.data_size // self.batch_size))
    def accumulate(self):
        """accumulate metrics when finished all iters."""
        print('[TEST] finished, avg_acc1= {}, avg_acc5= {} '.format(
            np.mean(np.array(self.top1)), np.mean(np.array(self.top5))))
```

完成 CenterCropMetric 类后，就可以开始验证过程了。验证的过程相对而言比较简单，开启模型的验证模式，加载训练好的参数。在每次迭代的过程中，将数据输入网络，并将得到的输出和标注通过 CenterCropMetric 计算 TOP1 和 TOP5 的精度。

```
model.eval()
state_dicts = load(weights)
model.set_state_dict(state_dicts)
data_size = len(valid_loader)
metric = CenterCropMetric(data_size=data_size, batch_size=val_batch_size)
for batch_id, data in enumerate(valid_loader):
    outputs = model.test_step(data)
    metric.update(batch_id, data, outputs)
metric.accumulate()
```

步骤 4：模型预测

在模型预测的过程要对每一个视频预测其所归属的类别。对于网络输出的结果，要通

过 softmax 转化为每个类别的置信度，置信度最高的类别就是网络预测的结果。

```
model.eval()
state_dicts = paddle.load(model_file)
model.set_state_dict(state_dicts)
for batch_id, data in enumerate(test_loader):
  _, labels = data
  outputs = model.test_step(data)
  scores = F.softmax(outputs)
  class_id = paddle.argmax(scores, axis = -1)
  pred = class_id.numpy()[0]
  label = labels.numpy()[0][0]
```

至此，我们就实现了 TimeSformer，快去动手试一试吧！