深度学习代码

2022年5月12日 21:40

1. IOU算法

```
class Rectangle(object):
    def __init__(self, x1, y1, x2, y2, *center):
        # 定义(左上角, 右下角)
        self.x_min = x1
        self.x_max = x2
        self.y_min = y1
        self.y_max = y2
        # 定义(x,y,w,h)
        self.center_x = center[0]
        self.center_y = center[1]
        self.width = center[2]
        self.heigh = center[3]

class Solution(object):
    def IOU(self, rec1, rec2):
        # 当为(x, y, w, h)时先转换为(x_min, x_max, y_min, y_max):
        rec1.x_min, rec1.x_max = rec1.center_x - rec1.width / 2, rec1.center_x + rec1.width / 2
        rec1.y_min, rec1.y_max = rec1.center_y - rec1.heigh / 2, rec1.center_y + rec1.heigh / 2
        rec2.x_min, rec2.x_max = rec2.center_x - rec2.width / 2, rec2.center_x + rec2.width / 2
        rec2.y_min, rec2.y_max = rec2.center_y - rec2.heigh/ 2, rec2.center_y + rec2.heigh / 2

        S_1 = (rec1.x_max-rec1.x_min) * (rec1.y_max-rec1.y_min)
        S_2 = (rec2.x_max-rec2.x_min) * (rec2.y_max-rec2.y_min)
        Ix_min = max(rec1.x_min, rec2.x_min)
        Ix_max = min(rec1.x_max, rec2.x_max)
        Iy_min = max(rec1.y_min, rec2.y_min)
        Iy_max = min(rec1.y_max, rec2.y_max)
        if Iy_max > Iy_min and Ix_max > Ix_min:
            Inter = (Iy_max-Iy_min) * (Ix_max-Ix_min)
            res = Inter / (S_1 + S_2 - Inter)
            return res
        else:
            return 0

if __name__ == '__main__':
    rec1 = Rectangle(100, 100, 200, 200, 150, 150, 100, 100)
    rec2 = Rectangle(120, 120, 220, 220, 170, 170, 100, 100)
    rel = Solution().IOU(rec1, rec2)
    print(rel)
```

2. NMS

```
import numpy as np

def NMS(dets, thresh):
    # 左上角,右下角坐标
    x_min = dets[:, 0]
    y_min = dets[:, 1]
    x_max = dets[:, 2]
    y_max = dets[:, 3]
    # confidence
    score = dets[:, 4]
    # 每个box的面积
    areas = (x_max - x_min + 1) * (y_max - y_min + 1)
    # confidence排序
    # 输出score中元素从大到小排列后对应的index(索引)
    order = score.argsort()[::-1]
    # 保存保留下来的bbox
    keep = []
    while order.size > 0:
        i = order[0]    # 最大confidence对应的index
        keep.append(i)    # 保存最大confidence的bbox的索引
        # 计算当前bbox与剩余bbox的交并比
        # 交集
        Ix_min = np.maximum(x_min[i], x_min[order[1:]])
        Ix_max = np.minimum(x_max[i], x_max[order[1:]])
        Iy_min = np.maximum(y_min[i], y_min[order[1:]])
        Iy_max = np.maximum(y_max[i], y_max[order[1:]])
        # 计算相交区域的W和H,如果无相交区域, 则为0
        H = np.maximum(0.0, Iy_max - Iy_min + 1)
        W = np.maximum(0.0, Ix_max - Ix_min + 1)
        Inter = W * H
        # IOU
        iou = Inter/(areas[i] + areas[order[1:]] - Inter)
        # 保留交集小于阈值的bbox的index
        idx = np.where(iou <= thresh)[0]
        order = order[idx + 1]
    return keep

if __name__ == '__main__':
    dets = np.array([
        [204, 102, 358, 250, 0.5],
        [257, 118, 380, 250, 0.7],
        [280, 135, 400, 250, 0.6],
        [255, 118, 360, 235, 0.7]
    ])
    thresh = 0.7
    rel = NMS(dets, thresh)
    print(rel)
```

3. 手写卷积

```
import numpy as np

# 给特征加padding
def padd(feature, num):
    w, h = feature.shape
    new_w = w + 2 * num
    new_h = h + 2 * num
    new_wh = (new_w, new_h)
    new_feature = np.zeros(new_wh)
    # 中间位置填充为原始特征的值
    new_feature[num:num+w, num:num+h] = feature[:, :]
```

```
    return new_feature
# 卷积操作
def conv(feature, conv_mask, padding, step):
    w, h = feature.shape
    conv_x, conv_y = conv_mask.shape
    target_w = (w - conv_x + 2 * padding) / step + 1
    target_h = (h - conv_y + 2 * padding) / step + 1
    target_wh = (int(target_w), int(target_h))
    target_feature = np.zeros(target_wh)
    # padding补零
    if padding != 0:
        feature = padd(feature, padding)
        w, h = feature.shape
    # 卷积核尺寸比特征尺寸大
    if conv_x > w or conv_y > h:
        return
    for i in range(0, w-conv_x+1, step):
        for j in range(0, h-conv_y+1, step):
            if i+conv_x > w and j+conv_y > h:
                break
            temp = feature[i:i+conv_x, j:j+conv_y]
            conv_rel = (temp * conv_mask).sum()
            target_feature[i//step][j//step] = conv_rel
    return target_feature

if __name__ == '__main__':
    lis_arry = np.array([
        [1, 2, 3, 4, 5, 6],
        [4, 5, 6, 7, 8, 9],
        [7, 8, 9, 10, 11, 12]
    ])
    conv_mask = np.array([
        [1, 1],
        [1, 1]
    ])
    padding = 1
    step = 2
    rel = conv(lis_arry, conv_mask, padding, step)
    print(rel)
```

4. 手写池化

```
import numpy as np

def Pooling(feature, pooling_kernel, step):
    w, h = feature.shape
    p_w, p_h = pooling_kernel.shape
    t_w = (w - p_w) // step + 1
    t_h = (h - p_h) // step + 1
    target_wh = (t_w, t_h)
    target_feature = np.zeros(target_wh)
    for i in range(0, w-p_w+1, step):
        for j in range(0, h-p_h+1, step):
            if i + p_w > w and j + p_h > h:
                break
            # 最大池化
            # max_val = np.max(feature[i:i+p_w, j:j+p_h])
            # target_feature[i // step][j // step] = max_val
            # 平均池化
            avg_val = np.sum(feature[i:i+p_w, j:j+p_h]) // (p_h*p_w)
            target_feature[i // step][j // step] = avg_val
    return target_feature

if __name__ == '__main__':
    lis_arry = np.array([
        [1, 2, 3, 4, 5, 6],
        [4, 5, 6, 7, 8, 9],
        [7, 8, 9, 10, 11, 12]
    ])
    pool_mask = np.array([
        [1, 1],
        [1, 1]
    ])
    step = 2
    rel = Pooling(lis_arry, pool_mask, step)
    print(rel)
```

5. ResNet

```
import torch
import torch.nn as nn
import math
import torch.utils.model_zoo as model_zoo

# 常见3*3卷积结构
def conv3(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride, padding=1, bias=False)
# ****
# 残差网络的basicblock: 3*3 + 3*3
class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, inplanes, outplanes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        # conv1
        self.conv1 = conv3(inplanes, outplanes, stride)
        self.bn1 = nn.BatchNorm2d(outplanes)
        self.relu = nn.ReLU(True)
        # conv2
        self.conv2 = conv3(outplanes, outplanes)
        self.bn2 = nn.BatchNorm2d(outplanes)
     # 下采样
     self.downsample = downsample
     self.stride = stride
  def forward(self, x):
     residual = x
     out = self.conv1(x)
     out = self.bn1(out)
     out = self.relu(out)

     out = self.conv2(out)
     out = self.bn2(out)

    if self.downsample is not None:
```

```
        residual = self.downsample(x)

      out = out + residual
      out = self.relu(out)
      return out

# ***
# Bottleneck: 1*1 + 3*3 + 1*1
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, outplanes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        # conv1 : 1*1
        self.conv1 = nn.Conv2d(inplanes, outplanes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(outplanes)
        # conv2: 3*3
        self.conv2 = nn.Conv2d(outplanes, outplanes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(outplanes)
        # conv3: 1*1
        self.conv3 = nn.Conv2d(outplanes, outplanes*4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(outplanes*4)
        self.relu = nn.ReLU(True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
      residual = x
      out = self.conv1(x)
      out = self.bn1(out)
      out = self.relu(out)
      out = self.conv2(out)
      out = self.bn2(out)
      out = self.relu(out)
      out = self.conv3(out)
      out = self.bn3(out)
      if self.downsample is not None:
        residual = self.downsample(x)
      out = out + residual
      out = self.relu(out)
      return out

# 不同深度ResNet中block堆叠个数
# ResNet18: ResNet(BasicBlock[2,2,2,2])
# ResNet34: ResNet(BasicBlock[3,4,6,3])
# Resnet50: ResNet(Bottleneck[3,4,6,3])
# ResNet101:ResNet(Bottleneck[3,4,23,3])
# ResNet152:ResNet(Bottleneck[3,8,36,3])

# ResNet18
def ResNet18(pretrained=False):
    model = ResNet(BasicBlock, [2, 2, 2, 2])
    if pretrained:
        model.load_state_dic(model_zoo.load_url('***'))
    return model

def ResNet50(pretrained=False):
    model = ResNet(Bottleneck, [3, 4, 6, 3])
    if pretrained:
        model.load_state_dict(model_zoo.load_url('***'))

class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1000):
        super(ResNet, self).__init__()
        self.inplanes = 64
        # conv1: 7*7
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(True)
        # max_pooling
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        # layer1:
        self.layer1 = self._make_layer(block, 64, layers[0])
        # layer2:
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        # layer3:
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        # layer4:
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        # avg_pooling
        self.avgpool = nn.AvgPool2d(7)
        self.fc = nn.Linear(512*block.expansion, num_classes)

  def _make_layer(self, block, planes, blocks, stride=1):
      downsample = None
      if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
          nn.Conv2d(self.inplanes, planes*block.expansion, kernel_size=1, stride=stride, bias=False),
          nn.BatchNorm2d(planes*block.expansion),
        )
       layers = []
       layers.append(block(self.inplanes, planes, stride, downsample))
       self.inplanes = planes * block.expansion
       for i in range(1, blocks):
          layers.append(block(self.inplanes, planes))
       return nn.Sequential(*layers)

  def forward(self, x):
      x = self.conv1(x)
      x = self.bn1(x)
      x = self.relu(x)
      x = self.maxpool(x)

      x = self.layer1(x)
      x = self.layer2(x)
      x = self.layer3(x)
      x = self.layer4(x)

      x = self.avgpool(x)
      x = x.view(x.size(0), -1)
      x = self.fc(x)
      return x
```

6. VGG16
   - `import torch`

```
import torch.nn as nn

cfg = {
    # VGG11
    'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    # VGG13
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    # VGG16
    'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
     # VGG19
     'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M']
}

class VGG16(nn.Module):
    def __init__(self, features, num_classes=3, init_weight=True):
        super(VGG16, self).__init__()
        self.features = features
        # 构造序列器
        self.classifier = nn.Sequential(
            # FC1
            nn.Linear(7*7*512, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            # FC2
            nn.Linear(4906, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            # FC3
            nn.Linear(4096, num_classes)
        )

      if init_weight:
       self._initialize_weights()

    def forward(self, x):
       x = self.features(x)
       # reshape为(B, C*W*H)
       x = x.view(x.size(0), -1)
       # 分类
        x = self.classifier(x)
        return x

  # 初始化权重
   def _initialize_weights(self):
     for m in self.modules():
       if isinstance(m, nn.Conv2d):
          nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
          if m.bias is not None:
             nn.init.constant_(m.bias, 0)
       elif isinstance(m, nn.BatchNorm2d):
          nn.init.constant_(m.weight, 1)
          nn.init.constant_(m.bias, 0)
       elif isinstance(m, nn.Linear):
          nn.init.normal_(m.weight, 0, 0.01)
          nn.init.constant_(m.bias, 0)

# 卷积层实现
def make_layers(cfg, batch_normal=False):
    layers = []
    in_channels = 3
    for v in cfg:
        # 池化层
        if v == 'M':
            layers += nn.MaxPool2d(kernel_size=2, stride=2)
        # 卷积层
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_normal:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(True)]
            else:
                layers += [conv2d, nn.ReLU(True)]
            in_channels = v
  return nn.Sequential(*layers)

def vgg16(**kwargs):
  model = VGG16(make_layers(cfg['D'], batch_normal=False), **kwargs)
  return model
```

7. PyTorch搭建网络及训练

```
import torch
import torchvision
from torch import nn
# 导入记好了，2维卷积，2维最大池化，展成1维，全连接层，构建网络结构辅助工具
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter

# 加载数据
# 参数：下载保存路径、train=训练集(True)或者测试集(False)、download=在线(True) 或者 本地(False)、数据类型转换
test_data = torchvision.datasets.CIFAR10("./dataset",
                                         train=False,
                                         download=True,
                                         transform=torchvision.transforms.ToTensor())
# # 格式打包
# # 参数：数据、1组几个、下一轮是否打乱、进程个数、最后一组是否凑成一组
test_loader = DataLoader(dataset=test_data, batch_size=4, shuffle=True, num_workers=0, drop_last=True)


class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.model1 = Sequential(
            # 输入，输出，卷积核、补几圈零
            Conv2d(3, 32, (5, 5), padding=2),
            # 池化核
            MaxPool2d(2),
            Conv2d(32, 32, (5, 5), padding=2),
```

```
            MaxPool2d(2),
            Conv2d(32, 64, (5, 5), padding=2),
            MaxPool2d(2),
            Flatten(),
            Linear(1024, 64),
            Linear(64, 10))

    def forward(self, x):
        x = self.model1(x)
        return x


if __name__ == '__main__':
    # #测试
    # tudui = Tudui()
    # # 验证网络 须知输入图像，设定全1矩阵测试
    # input = torch.ones((64, 3, 32, 32))
    # output = tudui(input)
    # print(output.shape)
    # # 绘制网络结构图
    # writer = SummaryWriter("log")
    # # 参数：网络结构对象、输入图像矩阵
    # writer.add_graph(tudui, input)
    # writer.close()

    # 模型加载
    tudui = Tudui()
    # 损失函数
    loss = nn.CrossEntropyLoss()
    # 优化器
    optim = torch.optim.Adam(tudui.parameters(), lr=0.0001)
    for epoch in range(20):
        # 每一轮损失
        running_loss = 0.0
        for data in test_loader:
            # 加载数据
            imgs, targets = data
            # 将数据放入网络
            outputs = tudui(imgs)

            # 损失函数：网络输出(预测)、标签
            result_loss = loss(outputs, targets)

            # 优化器 梯度清零
            optim.zero_grad()
            # 反向传播
            result_loss.backward()
            # 调用优化器
            optim.step()

            # 累计损失
            running_loss += result_loss
            # \r{} 可不换行直接显示，加上[]是为了好看一点
            print("\r[{}]".format(result_loss), end="")
        print(running_loss)
```

8. matplotlib显示图像
   - 显示图片
     - ```
       import cv2
       from matplotlib import pyplot as plt
       img = cv2.imread('cat.jpg')

       b, g, r = cv2.split(img)


       plt.subplot(2, 2, 1)
       plt.title('origin')
       plt.imshow(img[:,:,::-1])

       plt.subplot(2, 2, 2)
       plt.title('blue channel')
       plt.imshow(b, cmap='Blues')

       plt.subplot(2, 2, 3)
       plt.title('green channel')
       plt.imshow(g, cmap='Greens')

       plt.subplot(2, 2, 4)
       plt.title('red channel')
       plt.imshow(r, cmap='Reds')


       plt.show()
       ```
   - 目标检测框
     - ```
       import d2lzh as d2l
       from PIL import Image
       import torch

       d2l.set_figsize()
       img = Image.open('catdog.jpg')
       d2l.plt.imshow(img)
       d2l.plt.show()

       # bbox是bounding box的缩写
       # [左上角x,左上角y,右下角x,右下角y]
       dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 114, 655, 493]
       box = torch.tensor((dog_bbox, cat_bbox))
       print(box)


       def box_corner_to_center(boxes):
           """从（左上，右下）转换到（中间，宽度，高度）"""
       ```

```
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes


def box_center_to_corner(boxes):
    """从（中间，宽度，高度）转换到（左上，右下）"""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes


def bbox_to_rect(bbox, color):
    # 将边界框(左上x,左上y,右下x,右下y)格式转换成matplotlib格式:
    # ((左上x,左上y),宽,高)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2] - bbox[0], height=bbox[3] - bbox[1],
        fill=False, edgecolor=color, linewidth=2)


print(box_center_to_corner(box_corner_to_center(box)) == box)
'''
tensor([[True, True, True, True],
        [True, True, True, True]])
'''
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'))
d2l.plt.axis('off')
fig = d2l.plt.savefig('catdogbox.jpg')
```

9. CV2图像处理
   - 读取图像
     - ```
       import cv2

       # 读取一张400x600分辨率的图像
       color_img = cv2.imread('test_400x600.jpg')
       print(color_img.shape)

       # 直接读取单通道
       gray_img = cv2.imread('test_400x600.jpg', cv2.IMREAD_GRAYSCALE)
       print(gray_img.shape)

       # 把单通道图片保存后，再读取，仍然是3通道，相当于把单通道值复制到3个通道保存
       cv2.imwrite('test_grayscale.jpg', gray_img)
       reload_grayscale = cv2.imread('test_grayscale.jpg')
       print(reload_grayscale.shape)

       # cv2.IMWRITE_JPEG_QUALITY指定jpg质量，范围0到100，默认95，越高画质越好，文件越大
       cv2.imwrite('test_imwrite.jpg', color_img, (cv2.IMWRITE_JPEG_QUALITY, 80))

       # cv2.IMWRITE_PNG_COMPRESSION指定png质量，范围0到9，默认3，越高文件越小，画质越差
       cv2.imwrite('test_imwrite.png', color_img, (cv2.IMWRITE_PNG_COMPRESSION, 5))
       ```
   - 图像缩放，裁剪、补边
     - ```
       import cv2

       # 读取一张四川大录古藏寨的照片
       img = cv2.imread('tiger_tibet_village.jpg')

       # 缩放成200x200的方形图像
       img_200x200 = cv2.resize(img, (200, 200))

       # 不直接指定缩放后大小，通过fx和fy指定缩放比例，0.5则长宽都为原来一半
       # 等效于img_200x300 = cv2.resize(img, (300, 200))，注意指定大小的格式是(宽度,高度)
       # 插值方法默认是cv2.INTER_LINEAR，这里指定为最近邻插值
       img_200x300 = cv2.resize(img, (0, 0), fx=0.5, fy=0.5,
                                interpolation=cv2.INTER_NEAREST)

       # 在上张图片的基础上，上下各贴50像素的黑边，生成300x300的图像
       img_300x300 = cv2.copyMakeBorder(img, 50, 50, 0, 0,
                                        cv2.BORDER_CONSTANT,
                                        value=(0, 0, 0))

       # 对照片中树的部分进行剪裁
       patch_tree = img[20:150, -180:-50]

       cv2.imwrite('cropped_tree.jpg', patch_tree)
       cv2.imwrite('resized_200x200.jpg', img_200x200)
       cv2.imwrite('resized_200x300.jpg', img_200x300)
       cv2.imwrite('bordered_300x300.jpg', img_300x300)
       ```
10. IHS转换
    - ```
      def IHS(data_ms, data_pan):
          """
          基于IHS变换融合算法
          输入：np.ndArray格式的三维数组
          返回：可绘出图像的utf-8格式的三维数组
          """
          # RGB->IHS正变换矩阵
          A = [[1./3., 1./3., 1./3.], [-np.sqrt(2)/6., -np.sqrt(2)/6., 2*np.sqrt(2)/6], [1./np.sqrt(2), -1./np.sqrt(2), 0.]]
          A = np.matrix(A)

          band, w, h = data_ms.shape
          band_, w_, h_ = data_pan.shape
          pixels_ms = w * h
          pixels_pan = w_ * h_
      ```
      ```

```
    data_ms = data_ms.reshape(3, pixels_ms)
    # print('data_ms:', data_ms.shape)
    data_pan = data_pan.reshape(3, pixels_pan)
    # print('data_pan:', data_pan.shape)


    pan_i = np.dot(A, np.matrix(data_pan))  # Pan正变换  # [3, w*h]
    pan_i = np.array(pan_i)
    pan_i = pan_i.reshape(band_, w_, h_)
    pan_i = pan_i[0, :, :].reshape(1, w_, h_)


    ms_ihs = np.dot(A, np.matrix(data_ms))  # MS正变换  # [3, w*h]
    ms_ihs = np.array(ms_ihs)
    ms_ihs = ms_ihs.reshape(band_, w, h)
    ms_ihs = ms_ihs[1:, :, :]
    return  pan_i, ms_ihs
```

11. 特征图可视化

```
# heat 为某层的特征图，自己手动获取
def visual(heat):
    for i in range(len(heat)):
        heat1 = heat[i][0]
        heat1 = heat1.data.cpu().numpy()              # 将tensor格式的feature map转为numpy格式
        # print(heat1.shape)
        # heat = np.squeeze(heat, 0)                  # 0维为batch维度，由于是单张图片，所以batch=1，将这一维度删除
        heat1 = heat1[253,:]      # 切片获取某几个通道的特征图
        heatmap = np.maximum(heat1, 0)         # heatmap与0比较
        # heatmap = np.mean(heatmap, axis=0)  # 多通道时，取均值
        # heatmap /= np.max(heatmap)          # 正则化到 [0,1] 区间，为后续转为uint8格式图做准备
        # plt.matshow(heatmap)                                  # 可以通过 plt.matshow 显示热力图
        # plt.show()

        # 用cv2加载原始图像
        img = cv2.imread('/media/newamax/94d146aa-e21d-4f2d-ae9d-1f54448708202/lina/EAT_AugNWPU/VOCNWPU_ori/VOC2012/JPEGImages/311.jpg')
        img = cv2.resize(img, (400, 400))
        heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))  # 特征图的大小调整为与原始图像相同
        heatmap = np.uint8(255 * heatmap) # 将特征图转换为uint8格式
        heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)  # 将特征图转为伪彩色图
        heat_img = cv2.addWeighted(img, 0.5, heatmap, 0.5, 0)      # 将伪彩色图与原始图片融合
        #heat_img = heatmap * 0.5 + img                    # 也可以用这种方式融合
        cv2.imwrite('/media/newamax/94d146aa-e21d-4f2d-ae9d-1f54448708202/lina/EAT_AugNWPU/Visual_Feautures/311_'+str(i)+'.jpg', heat_img) # 将图像保存
```

12. 上采样

```
import torch.nn.functional as F
def Upsample(x, y):
    """
    input:
        x:[B,C,NW,NH]
        y:[B,C,W,H]
    """
    _, _, h1, w1 = x.size()
    result = F.upsample(y, size=(h1, w1), mode='bilinear')
    return result
```

13. Softmax实现

```
def Softmax(output):
    # output:[B,N]
    y = np.exp(output) / np.sum(np.exp(output), axis=1)
    return y
```

14. 交叉熵损失函数

```
def Entropy_loss(output, label):
    # output:[B,N]
    # label:[B,N]
    B, N = output.shape
    # 如果label只有一维
    if label.ndim == 1:
        output = output.reshape(1, B*N)
        label = label.reshape(1, B*N)
    loss = -np.sum(label*np.log(output)) / B
    return loss
```

15. 二维BN实现（B，N) 你不不不

```
import numpy as np
def BN_forward(x, gamma, beta, eps=1e-5):
    # 计算均值
    mean_val = np.mean(x, axis=0)   # 每一列
    # 计算方差
    std_val = np.var(x, axis=0)
    # 归一化
    temp = (x - mean_val) / np.sqrt(std_val + eps)
    # 缩放平移
    out = gamma * temp + beta
    return out
```

16. 四维BN实现（B，C，H，W）

- BN在整个Batchsize上进行，在B，H，W上进行标准化，与通道数无关，执行完有C个均值，C个方差

```
def BN(X, gamma=3, betta=4, eps=1e-7):
    # B: [B, C, H, W]
    B, C, H, W = X.shape
    m = np.mean(X, axis=(0, 2, 3)).reshape(1, C, 1, 1)
    v = np.var(X, axis=(0, 2, 3)).reshape(1, C, 1, 1)
    out = (X-m)/np.sqrt(v+eps)
    out = gamma*out + betta
    return out
```

17. LR实现（B，C，H，W）

- LN是针对每个样本的所有特征进行标准化，在C，H，W上进行归一化，与Batchsize无关，执行完有B个均值，B个方差

```
def LR(X, eps=1e-7):
    B, C, H, W = X.shape
    m = np.mean(X, axis=(1, 2, 3)).reshape(B, 1, 1, 1)
    v = np.var(X, axis=(1, 2, 3)).reshape(B, 1, 1, 1)
    out = (X - m) / np.sqrt(v+eps)
    return out
```

18. IN（B，C，H，W）

- IN是针对单个样本进行标准化，在H，W上进行归一化，与Batch和Layer都无关，执行完有B，C个均值，B，C个方差
- ```
  def IN(X, eps=1e-7):
      B, C, H, W = X.shape
      m = np.mean(X, axis=(2, 3)).reshape(B, C, 1, 1)
      v = np.var(X, axis=(2, 3)).reshape(B, C, 1, 1)
      out = (X - m) / np.sqrt(v + eps)
      return out
  ```

19. GN（B，C，H，W）
    - GN将C分为多个group，B,C,H,W转换为B*G,C/G,H,W然后对每个组进行归一化，执行完有B,G个均值，B,G个方差
    - ```
      defGN(X,eps=1e-7,group=5):
      B,C,H,W=X.shape
      X=X.reshape(B,group,-1)
      m=np.mean(X,axis=-1).reshape(B,group,1)
      v=np.var(X,axis=-1).reshape(B,group,1)
      out=(X-m)/np.sqrt(v+eps)
      out=out.reshape(B,C,H,W)
      Return out
      ```