人工神经网络原理期末大作业

陈政培 17363011 智科1班

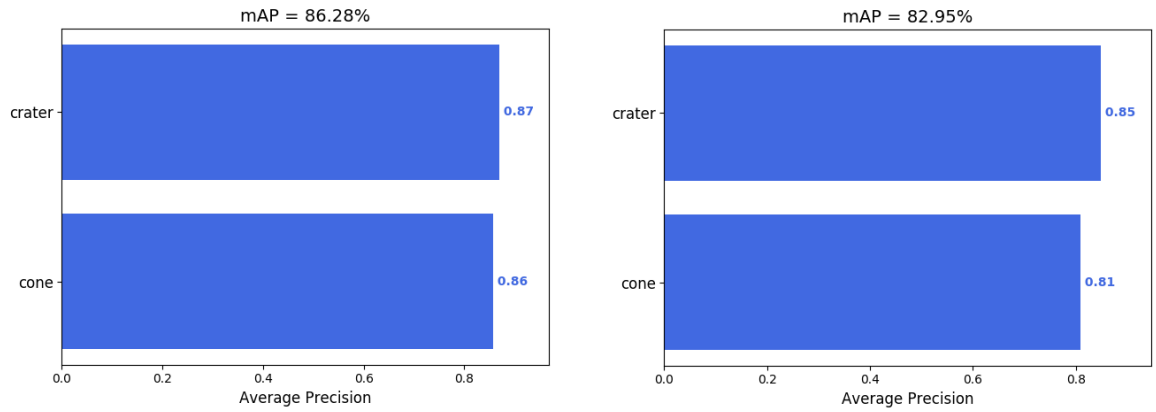
两种网络的优劣对比

Faster R-CNN中的ROIPooling部分很难弄，所以Faster R-CNN代码中不好设置batch_size的大小只能强制为1，所以训练速度非常慢。

从理论上，相比RCNN系列物体检测方法，YOLO识别物体位置精准性差；召回率低，在每个网格中预测两个bbox这种约束方式减少了对同一目标的多次检测(R-CNN使用的region proposal方式重叠较多)，从而减少了候选框的数量，相比R-CNN使用Selective Search产生2000个proposal，YOLO仅使用7x7x2个。加了 BN 层，扩大输入维度，使用了 Anchor，训练的时候数据增强，仿ResNet的Darknet-53，仿SqueezeNet的纵横交叉网络，进化到YOLOv3。

由于时间原因本次Faster R-CNN只训练了25代，如果能够训练50代mAP将更高。实验数据表明Faster R-CNN对小目标的检测效果还是最好（只用25代达到和YOLO持平），YOLO v3吸取了SSD的一些优点，比Faster R-CNN快、比SSD检测小目标准，效果中规中矩。

左侧为YOLO训练后mAP，右侧为Faster R-CNN训练后mAP（只训练25代，常规是50代，时间原因）



YOLOv3实现

YOLOv3就是把一张图片划分成不同的网络，每个网络点负责一个区域的预测，只要物体的中心点落在这个区域，这个物体就由这个网络点确定

环境

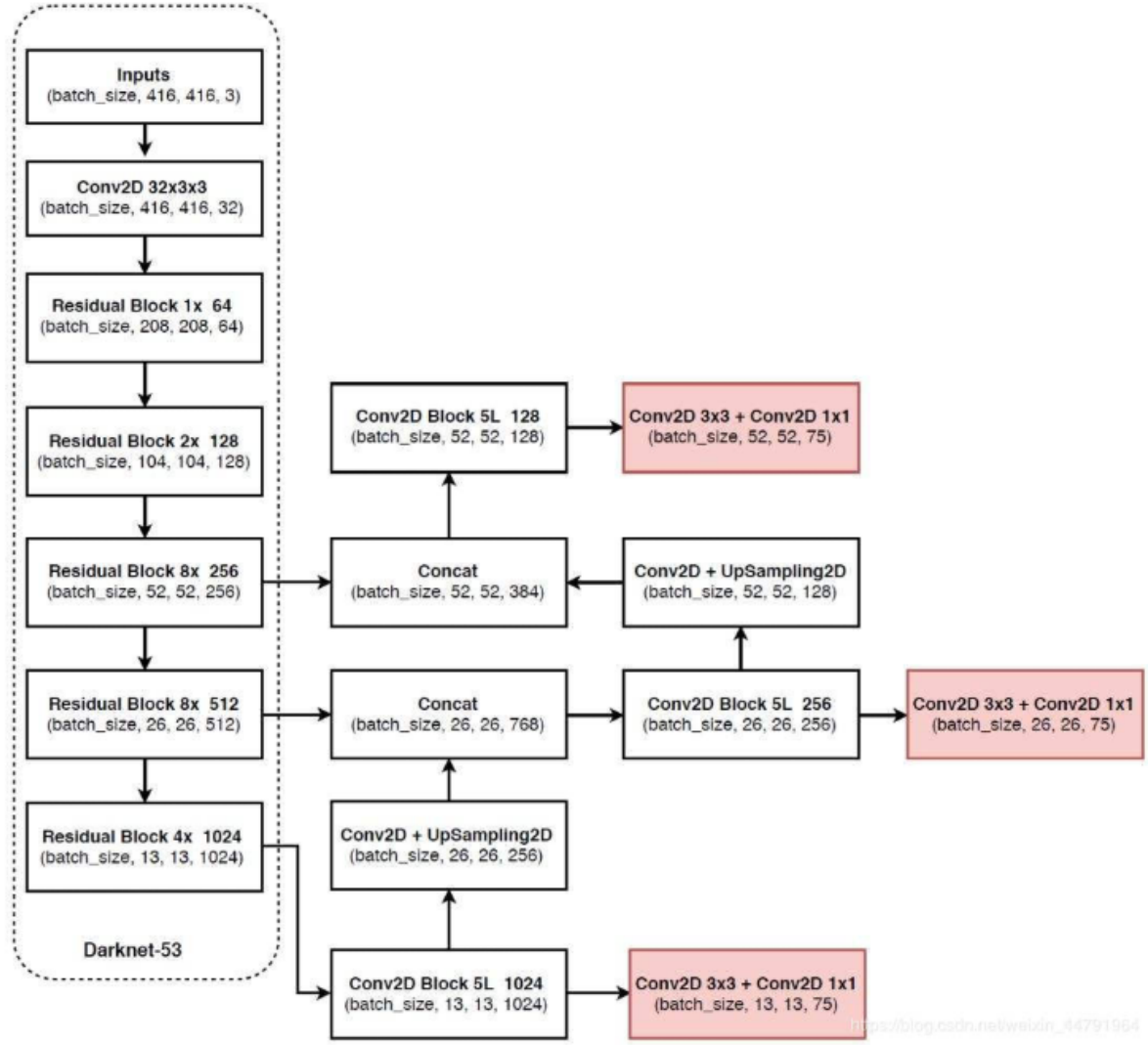
Python：3.7.5

numpy：1.17.4

tensorflow-gpu：1.13.2

keras：2.1.5

基本结构实现



特征提取网络darknet-53

特征提取网络使用的是darknet-53，将输入图像进行卷积后，再通过一系列残差网路的特征提取，提取完后将得到3个不同shape的特征值输出，分别是13×13，26×26，52×52特征层，三部分特征层再放入YOLO进行后续多尺度操作。darknet53的每一个卷积部分使用了特有的DarknetConv2D结构，每一次卷积的时候进行l2正则化，完成卷积后进行BatchNormalization标准化与LeakyReLU。普通的ReLU是将所有的负值都设为零，Leaky ReLU则是给所有负值赋予一个非零斜率

darknet-53部分参照网络上的模型，用keras库来实现。darknet-53卷积的过程不同于普通的卷积，它内部经过了正则化的改进，帮助提升性能

```
@wraps(Conv2D)
def DarknetConv2D(*args, **kwargs):
    darknet_conv_kwargs = {'kernel_regularizer': l2(5e-4)}  #实现正则化
    darknet_conv_kwargs['padding'] = 'valid' if kwargs.get('strides')==(2,2)
else 'same'
    darknet_conv_kwargs.update(kwargs)
    return Conv2D(*args, **darknet_conv_kwargs)
```

结合了darknet-53的特殊卷积块，包含3个部分分别是DarknetConv2D、标准化、LeakyReLU，就不需要在后面再加入标准化和激活函数

```
#   卷积块
#   DarknetConv2D + BatchNormalization + LeakyReLU
def DarknetConv2D_BN_Leaky(*args, **kwargs):
    no_bias_kwargs = {'use_bias': False}
    no_bias_kwargs.update(kwargs)
    return compose(
        DarknetConv2D(*args, **no_bias_kwargs),
        BatchNormalization(), #标准化
        LeakyReLU(alpha=0.1))
```

卷积层中调用了残差网络的输入调整了长和宽，然后进行两次卷积，先压缩卷积再进行扩张卷积，x、y为残差结构，通过add相加。

```
#   卷积块
#   DarknetConv2D + BatchNormalization + LeakyReLU
def resblock_body(x, num_filters, num_blocks):
    x = ZeroPadding2D(((1,0),(1,0)))(x)
    x = DarknetConv2D_BN_Leaky(num_filters, (3,3), strides=(2,2))(x)
    for i in range(num_blocks):
        y = DarknetConv2D_BN_Leaky(num_filters//2, (1,1))(x)
        y = DarknetConv2D_BN_Leaky(num_filters, (3,3))(y)
        x = Add()([x,y])
    return x
```

darknet53主体部分，其中resblock_body会调用特殊卷积块，x为输入特征层，64为输出通道数，1为重复次数。feat1、feat2、feat3分别用来记录需要传到YOLO处理网络中的特征层。

```
def darknet_body(x):
    x = DarknetConv2D_BN_Leaky(32, (3,3))(x) #调整通道数
    x = resblock_body(x, 64, 1) #x为输入特征层，64为输出通道数，1为重复次数
    x = resblock_body(x, 128, 2)
    x = resblock_body(x, 256, 8)
    feat1 = x #网络结构52，52，256
    x = resblock_body(x, 512, 8)
    feat2 = x #网络结构26，26，512
    x = resblock_body(x, 1024, 4)
    feat3 = x #网络结构13，13，1024
    return feat1,feat2,feat3
```

YOLO3特征处理网络

获得三个特征层的提取结果，每个图分为13x13、26x26、52x52的网格上3个预测框的位置。特征层上每个网络点负责一个区域的检测，特征层的预测结果对应着三个预测框的位置，所以需要先将其reshape一下，其结果为(N，13，13，3，21)，(N，26，26，3，21)，(N，52，52，3，21)。

每一层中13，13，3，21分别代表将图片分为13×13个格子，每个单元生成3个先验框，每个先验框有7个参数。前面4个参数是先验框的中心相对于单元中心点的偏移量（x、y偏移量，框的长和宽h、w），然后1个参数代表置信度，后面的2个参数是分类结果（crater和cone）。

通过上采样形成特征金字塔 `Spatial pyramid` 的结构：

1. 13x13的特征层在YOLO卷积完后也会通过上采样，和上层的网络进行堆叠，另外的部分还要进行3×3的卷积并调整通道数，然后输出13x13网络每个网格里3个先验框的参数
2. 26x26的特征层在堆叠后也进行卷积，卷积后进行上采样和再上一层的网络进行堆叠，另外部分进行卷积调整通道，输出参数

3. 52x52的特征层堆叠后进行卷积得到输出参数。

```
#   特征层->最后的输出
def yolo_body(inputs, num_anchors, num_classes):
    # 生成darknet53的主干模型
    feat1,feat2,feat3 = darknet_body(inputs) #13x13、26x26、52x52三个特征层
    darknet = Model(inputs, feat3)

    # 第一个特征层
    # y1=(batch_size,13,13,3,21)
    x, y1 = make_last_layers(darknet.output, 512, num_anchors*(num_classes+5))

    x = compose(
            DarknetConv2D_BN_Leaky(256, (1,1)),
            UpSampling2D(2))(x)
    x = Concatenate()([x,feat2])
    # 第二个特征层
    # y2=(batch_size,26,26,3,21)
    x, y2 = make_last_layers(x, 256, num_anchors*(num_classes+5))

    x = compose(
            DarknetConv2D_BN_Leaky(128, (1,1)),
            UpSampling2D(2))(x)
    x = Concatenate()([x,feat1])
    # 第三个特征层
    # y3=(batch_size,52,52,3,21)
    x, y3 = make_last_layers(x, 128, num_anchors*(num_classes+5))

    return Model(inputs, [y1,y2,y3])
```

YOLO输出参数部分的卷积网络，包括正则化的卷积和Leaky的激活函数。

```
#   特征层->最后的输出
def make_last_layers(x, num_filters, out_filters):
    # 五次卷积
    x = DarknetConv2D_BN_Leaky(num_filters, (1,1))(x)
    x = DarknetConv2D_BN_Leaky(num_filters*2, (3,3))(x)
    x = DarknetConv2D_BN_Leaky(num_filters, (1,1))(x)
    x = DarknetConv2D_BN_Leaky(num_filters*2, (3,3))(x)
    x = DarknetConv2D_BN_Leaky(num_filters, (1,1))(x)

    # 将最后的通道数调整为outfilter
    y = DarknetConv2D_BN_Leaky(num_filters*2, (3,3))(x)
    y = DarknetConv2D(out_filters, (1,1))(y)

    return x, y
```

yolo3的解码过程就是将每个网格点加上它对应的x和y，加完后的结果就是预测框的中心，然后再利用先验框和h、w结合计算出预测框的长和宽。这样就能得到整个预测框的位置了。但是调整的预测框位置还需要进行得分排序与非极大抑制筛选，其对于每一个类进行判别：1.取出每一类得分大于threshold的框和得分 2.利用框的位置和得分进行非极大抑制，非极大抑制可以通过IOU去除重合度高的框。

```
#   图片预测
def yolo_eval(yolo_outputs,
```

```
                anchors,
                num_classes,
                image_shape,
                max_boxes=20,
                score_threshold=.6,
                iou_threshold=.5):
    # 获得特征层的数量
    num_layers = len(yolo_outputs)
    anchor_mask = [[6,7,8], [3,4,5], [0,1,2]]

    ...

    return boxes_, scores_, classes_
```

一副图像如果被卷积多次，其长、宽被过度压缩的话，对于小目标的检测效果就会越来越差。小的分格网络会用来检测大目标，大的网络会用来检测小目标因为大网格丢失的信息更少。`yolo.py` 文件中则是解码主干模型并解码的代码，获取特征之后进行分类并绘制检测框。

```
class YOLO(object):
    def __init__(self, **kwargs): #初始化yolo
        self.__dict__.update(self._defaults)
        self.class_names = self._get_class() #所有类的名字
        self.anchors = self._get_anchors() #获得所有先验框 9x2
        self.sess = K.get_session()
        self.boxes, self.scores, self.classes = self.generate()
    def generate(self): #获得所有的分类
        ...
    def detect_image(self, image):  #检测图片
        ...
```

YOLO3解码的核心代码在于 `yolo3.py` 文件中的 `yolo_head` 函数，将网格画出来，把预测结果的值加上网格求出来框的中心点，利用先验框求框的宽和高。

```
#   将预测值的每个特征层调成真实值
def yolo_head(feats, anchors, num_classes, input_shape, calc_loss=False):
    num_anchors = len(anchors)
    # [1, 1, 1, num_anchors, 2]
    anchors_tensor = K.reshape(K.constant(anchors), [1, 1, 1, num_anchors, 2])

    # 获得x，y的网格
    # (13, 13, 1, 2)
    grid_shape = K.shape(feats)[1:3] # height, width
    grid_y = K.tile(K.reshape(K.arange(0, stop=grid_shape[0]), [-1, 1, 1, 1]),
        [1, grid_shape[1], 1, 1])
    grid_x = K.tile(K.reshape(K.arange(0, stop=grid_shape[1]), [1, -1, 1, 1]),
        [grid_shape[0], 1, 1, 1])
    grid = K.concatenate([grid_x, grid_y])
    grid = K.cast(grid, K.dtype(feats))

    # (batch_size,13,13,3,85)
    feats = K.reshape(feats, [-1, grid_shape[0], grid_shape[1], num_anchors,
num_classes + 5])

    # 将预测值调成真实值
    # box_xy对应框的中心点
    # box_wh对应框的宽和高
```

```
    box_xy = (K.sigmoid(feats[..., :2]) + grid) / K.cast(grid_shape[::-1],
K.dtype(feats))
    box_wh = K.exp(feats[..., 2:4]) * anchors_tensor / K.cast(input_shape[::-1],
K.dtype(feats))
    box_confidence = K.sigmoid(feats[..., 4:5])
    box_class_probs = K.sigmoid(feats[..., 5:])

    # 在计算loss的时候返回如下参数
    if calc_loss == True:
        return grid, feats, box_xy, box_wh
    return box_xy, box_wh, box_confidence, box_class_probs
```

评价参数

损失函数loss

其实际上计算的总的loss是三个loss的和，这三个loss分别是：

1. 实际存在的框，编码后的长宽与xy轴偏移量与预测值的差距。
2. 实际存在的框，预测结果中置信度的值与1对比；实际不存在的框，在上述步骤中，在第四步中得到其最大IOU的值与0对比。
3. 实际存在的框，种类预测结果与实际结果的对比。

准确率precision

TP（True Positives）意思就是被分为了正样本，而且分对了。
TN（True Negatives）意思就是被分为了负样本，而且分对了，
FP（False Positives）意思就是被分为了正样本，但是分错了（事实上这个样本是负样本）。
FN（False Negatives）意思就是被分为了负样本，但是分错了（事实上这个样本是这样本）。

$$Precision = \frac{TP}{TP + FP}$$

召回率Recall。

$$Recall = \frac{TP}{TP + FN}$$

训练数据集

VOC数据集文件夹包括 `Annotations` 用来存放标签、 `ImageSets` 设置训练集和测试集、 `JPEGImages` 存放图片

`voc_annotation.py` 文件中则是用来把测试集转换成YOLO训练需要的格式，将train.txt读取生成标签文件和图片的映射关系，每一行，前者为图片位置，后者738,375,800,432,0为目标的左上角右下角位置，0为目标类型的编号

```
./yolo3-keras-
master/VOCdevkit/VOC2007/JPEGImages/mudESP_025439_2210_RED03_05.jpg
738,375,800,432,0 613,527,718,600,0
```

`train.py` 文件就是调用训练的文件，读取需要的文件，并载入预训练好的权重 `yolo_weights.h5` ，通过迁移学习使用YOLO本身训练好的参数更适合用来目标分类，darknet-53会提取出特别适合目标检测的特征。

```
if __name__ == "__main__":
    # 标签的位置
    annotation_path = '2007_train.txt'
    # 获取classes和anchor的位置
    classes_path = 'model_data/voc_classes.txt'
    anchors_path = 'model_data/yolo_anchors.txt'
    weights_path = 'model_data/yolo_weights.h5'
```

YOLO3模型表现提升

冻结部分darknet层加快训练速度

因为使用预训练好的参数，可以在一开始的时候先 `freeze_layers` 冻结前面适合目标检测的参数，只训练最后部分的权重，可以更快的分类。

```
freeze_layers = 249
    for i in range(freeze_layers): model_body.layers[i].trainable = False
    print('Freeze the first {} layers of total {} layers.'.format(freeze_layers,
len(model_body.layers)))

    # 训练参数设置
    logging = TensorBoard(log_dir=log_dir)
    checkpoint = ModelCheckpoint(log_dir + 'ep{epoch:03d}-loss{loss:.3f}-
val_loss{val_loss:.3f}.h5',
        monitor='val_loss', save_weights_only=True, save_best_only=False,
period=2)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2,
verbose=1)
    early_stopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=6,
verbose=1) #防止过拟合

    # 0.1用于验证，0.9用于训练
    val_split = 0.1
    with open(annotation_path) as f:
        lines = f.readlines()
    np.random.seed(10101)
    np.random.shuffle(lines)
    np.random.seed(None)
    num_val = int(len(lines)*val_split)
    num_train = len(lines) - num_val

    # 调整非主干模型first
    if True:
        model.compile(optimizer=Adam(lr=1e-3), loss={
            'yolo_loss': lambda y_true, y_pred: y_pred})

        batch_size = 1
        print('Train on {} samples, val on {} samples, with batch size
{}.'.format(num_train, num_val, batch_size))
        model.fit_generator(data_generator(lines[:num_train], batch_size,
input_shape, anchors, num_classes),
                steps_per_epoch=max(1, num_train//batch_size),
                validation_data=data_generator(lines[num_train:], batch_size,
input_shape, anchors, num_classes),
                validation_steps=max(1, num_val//batch_size),
```

```
                epochs=50,
                initial_epoch=0,
                callbacks=[logging, checkpoint, reduce_lr, early_stopping])
        model.save_weights(log_dir + 'trained_weights_stage_1.h5')
```

然后再恢复冻结进行全面的训练，因为所有层开放后参数量非常的大需要把 `batch_size` 调小一点

```
    for i in range(freeze_layers): model_body.layers[i].trainable = True #解冻所有层

    # 解冻后训练
    if True:
        model.compile(optimizer=Adam(lr=1e-4), loss={
            'yolo_loss': lambda y_true, y_pred: y_pred})

        batch_size = 4
        print('Train on {} samples, val on {} samples, with batch size {}.'.format(num_train, num_val, batch_size))
        model.fit_generator(data_generator(lines[:num_train], batch_size, input_shape, anchors, num_classes),
                steps_per_epoch=max(1, num_train//batch_size),
                validation_data=data_generator(lines[num_train:], batch_size, input_shape, anchors, num_classes),
                validation_steps=max(1, num_val//batch_size),
                epochs=100,
                initial_epoch=50,
                callbacks=[logging, checkpoint, reduce_lr, early_stopping])
        model.save_weights(log_dir + 'last1.h5')
```

自适应学习率变化和过拟合停止

训练过程中的两个步骤：

1. `ReduceLROnPlateau` 可以实现两次loss不下降则下降学习率
2. `EarlyStopping` 则实现了loss一直不下降，说明网络过拟合了，就可以提前停止训练

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, verbose=1)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=6, verbose=1) #防止过拟合
```

还可以尝试更精细化调整学习率，学习率大加快学习效率但是可能导致不收敛，学习率小有助于模型细化提高精度，但收敛速度慢

使用阶层性学习率下降，使用ReduceLROnPlateau可以指定验证集的loss不下降后，突然下降学习率，变为原来的0.1倍

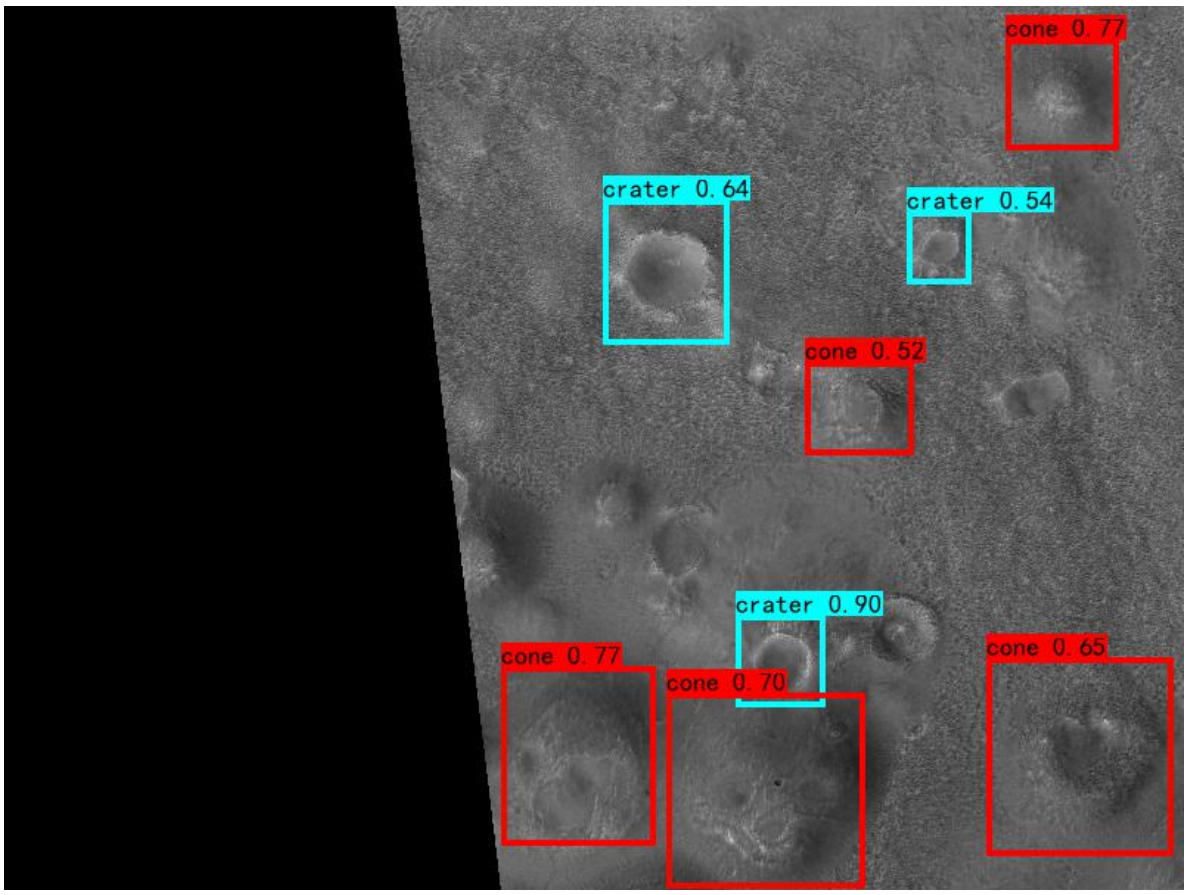
```
exponent_lr = ExponentDecayScheduler(learning_rate_base = learning_rate_base,
                                     global_epoch_init = init_epoch,
                                     decay_rate = 0.9,
                                     min_learn_rate = 1e-6
                                     )
```

YOLOv3实验结果

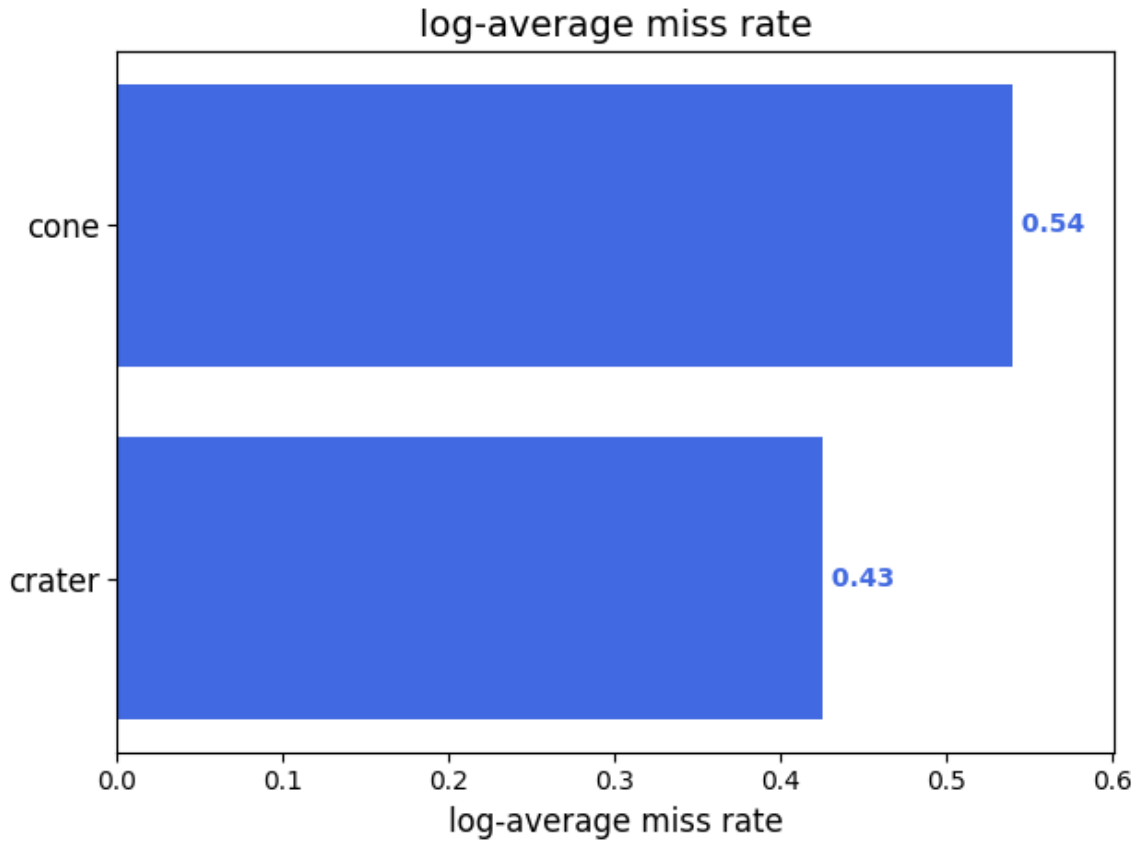
训练在yolo_weights.h5的参数基础上主干网络训练74次loss停止减小，停止网络学习过程，此时的loss是39.9012

```
49/49 [==============================] - 16s 326ms/step - loss: 43.4322 - val_loss: 38.9243
Epoch 71/100
49/49 [==============================] - 16s 321ms/step - loss: 43.5915 - val_loss: 51.8727

Epoch 00071: ReduceLROnPlateau reducing learning rate to 4.999999873689376e-05.
Epoch 72/100
49/49 [==============================] - 16s 319ms/step - loss: 41.3493 - val_loss: 42.4649
Epoch 73/100
49/49 [==============================] - 17s 342ms/step - loss: 40.1485 - val_loss: 48.8073

Epoch 00073: ReduceLROnPlateau reducing learning rate to 2.499999936844688e-05.
Epoch 74/100
49/49 [==============================] - 15s 311ms/step - loss: 39.9012 - val_loss: 40.6919
Epoch 00074: early stopping
```

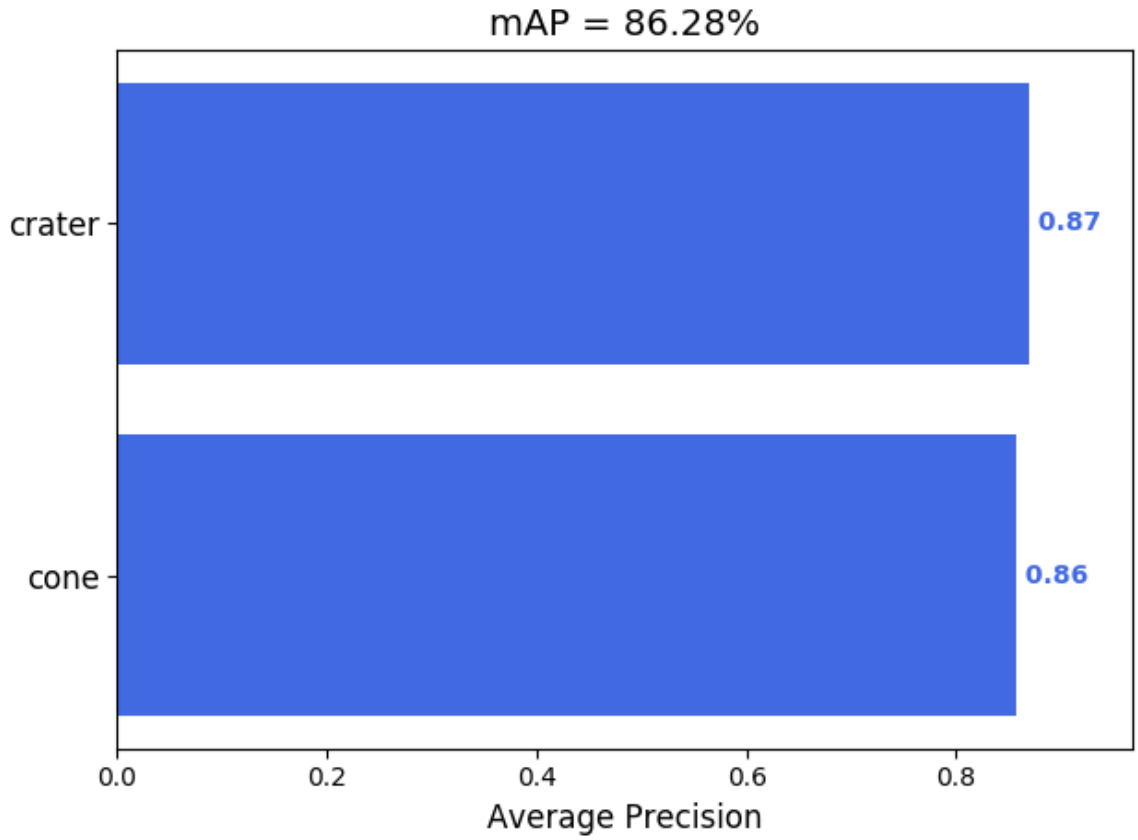
运行 `mudESP_040775_2235_RED03_13.jpg` 测试模型效果



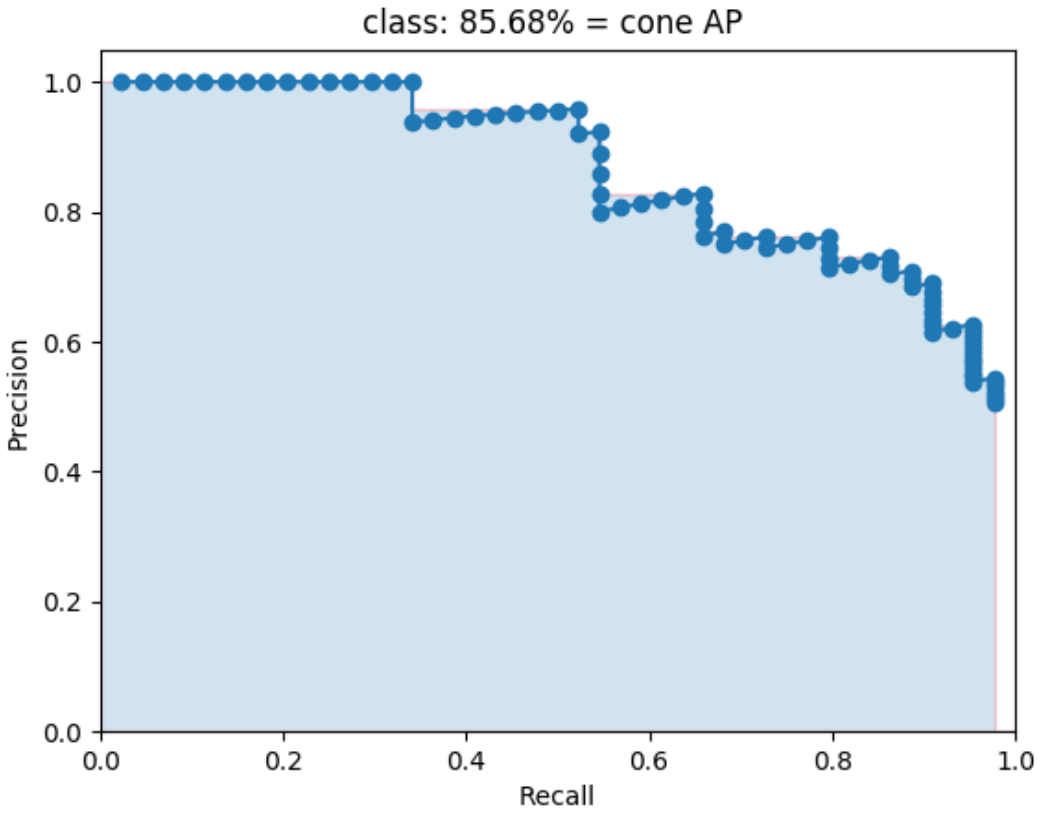
全测试集的识别精度Accuracy如图

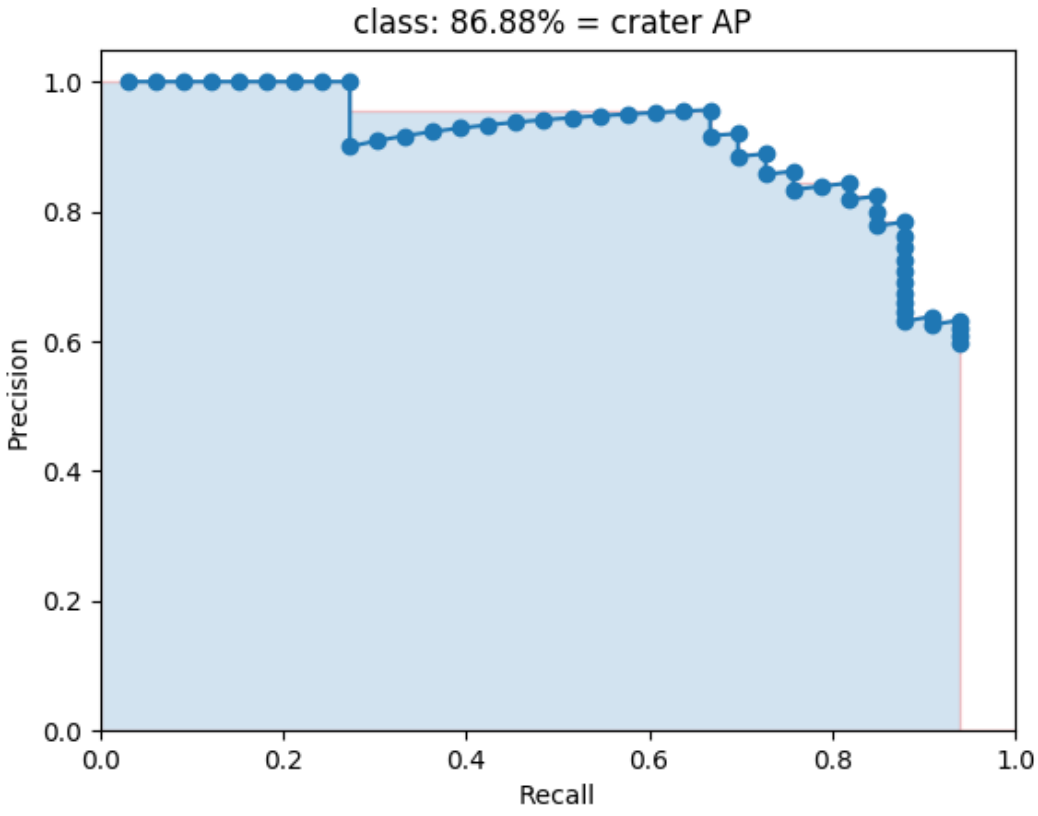


经过对测试集的测试，得到的mAP为86.28%，其中cone的AP为85.68%，crater的AP为86.88%



cone和crater两个类型的召回率Recall和准确率Precision分别绘制如图


class: 85.68% = cone AP
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0


class: 86.88% = crater AP
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0

Faster R-CNN实现

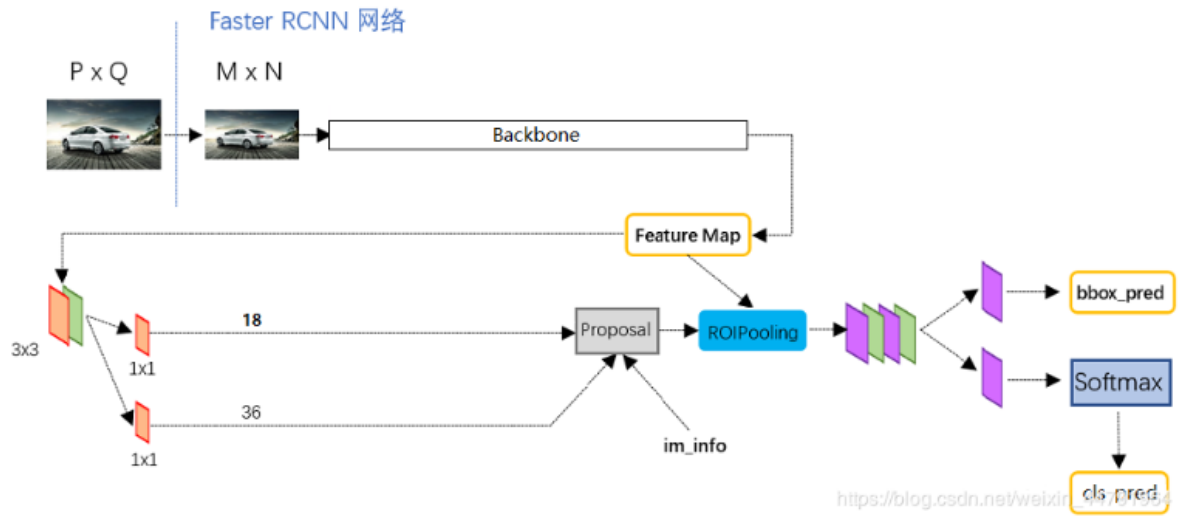
环境

Python：3.7.5

tensorflow-gpu：1.13.2

cupy：10.1

基本结构实现



主干特征提取网络

将resize过后的图像利用backbone进行特征提取，一般主干提取网络可以使用VGG或者Resnet。此处运用的resnet-50，当输入图经过resnet-50特征提取后，我们会获得一个共享特征层feature map，后续很多操作都会在此feature map上进行操作。feature map将图片分成38×38个网格，每个网格上有9个先验框。

ResNet50有两个基本的块，分别名为Conv Block和Identity Block，两者的最大差别在于残差边上是否有卷积和正则化。通过残差卷积，可以实现输出层维度的变化。其中Conv Block输入和输出的维度是不一样的，所以不能连续串联，它的作用是改变网络的维度；Identity Block输入维度和输出维度相同，可以串联，用于加深网络的。

`resneet50.py` 中当输入和输出维度变化的时候瓶颈结构Bottleneck代表Conv Block，维度相同时Bottleneck代表Identity Block。

```
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, stride=stride,
bias=False) # change
        self.bn1 = nn.BatchNorm2d(planes) #第一个卷积压缩通道数

        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, # change
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes) #利用3x3的卷积进行特征提取

        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
```

```
        self.bn3 = nn.BatchNorm2d(planes * 4) #1×1的卷积扩展通道数

        self.relu = nn.ReLU(inplace=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: #是否有残差卷积操作，区分Conv Block和
Identity Block
            residual = self.downsample(x)

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

        return out
```

resnet-50结构

```
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1000):
        self.inplanes = 64
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
                    bias=False) #对输入进来的图片进行卷积
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=0,
ceil_mode=True) # 最大池化
        #通过make_layer实现Conv Block和Identity Block
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AvgPool2d(7)
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

```
    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential( #定义残差边
                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
```

主干特征提取网络和分类模型的分割

```
def resnet50():
    model = ResNet(Bottleneck, [3, 4, 6, 3])
    # 获取特征提取部分
    features = list([model.conv1, model.bn1, model.relu, model.maxpool,
model.layer1, model.layer2, model.layer3])
    # 获取分类部分
    classifier = list([model.layer4, model.avgpool])
    features = nn.Sequential(*features)
    classifier = nn.Sequential(*classifier)
    return features,classifier
```

建议框网络

将输入的共享特征层进行3×3的卷积，然后对其结果进行两次1×1的卷积，一个通道数为18，一个通道数为36。两个1×1的卷积就是为了判断网格中是否真的包含物体，然后根据结果对先验框进行调整，调整成为建议框，其中18通道可以拆分为9×2的卷积用来判断先验框内部是否真实的包含物体，其中的2个参数中包含先验框是背景的概率和先验框是物体的概率，然后36通道拆分为9×4，其中的4个参数代表先验框的位置。

`frcnn.py` 中此段代码为收到resnet网络结果，构建RegionProposalNetwork类构建建议框网络

```
elif backbone == 'resnet50':
    self.extractor, self.classifier = resnet50()

    self.rpn = RegionProposalNetwork(
        1024, 512,
        ratios=ratios,
        anchor_scales=anchor_scales,
        feat_stride=self.feat_stride,
        mode = mode
    )
    self.head = Resnet50RoIHead(
        n_class=num_classes + 1,
        roi_size=14,
        spatial_scale=(1. / self.feat_stride),
        classifier=self.classifier
    )
```

RegionProposalNetwork类

```
super(RegionProposalNetwork, self).__init__()
    self.anchor_base = generate_anchor_base(anchor_scales=anchor_scales,
ratios=ratios)
    # 步长，压缩的倍数
    self.feat_stride = feat_stride
    self.proposal_layer = ProposalCreator(mode)
    # 每一个网格上默认先验框的数量
    n_anchor = self.anchor_base.shape[0]
    # 先进行一个3x3的卷积
    self.conv1 = nn.Conv2d(in_channels, mid_channels, 3, 1, 1)
    # 分类预测先验框内部是否包含物体
    self.score = nn.Conv2d(mid_channels, n_anchor * 2, 1, 1, 0)
    # 回归预测对先验框进行调整
    self.loc = nn.Conv2d(mid_channels, n_anchor * 4, 1, 1, 0)
    normal_init(self.conv1, 0, 0.01)
    normal_init(self.score, 0, 0.01)
    normal_init(self.loc, 0, 0.01)
```

ROIPooling层就会利用建议框对共享特征层进行截取，由于我们获得的建议框大小并不是一样大的，因此进行截取时我们获取到的局部特征层是不一样大的，然后ROIPooling就会对局部特征层进行分区域的池化，获得的所有局部特征层就一样大了。

```
def forward(self, x, img_size, scale=1.):
    n, _, hh, ww = x.shape
    # 对共享特征层进行一个3x3的卷积
    h = F.relu(self.conv1(x))
    # 回归预测
    rpn_locs = self.loc(h)
    rpn_locs = rpn_locs.permute(0, 2, 3, 1).contiguous().view(n, -1, 4)
    # 分类预测
    rpn_scores = self.score(h)
    rpn_scores = rpn_scores.permute(0, 2, 3, 1).contiguous().view(n, -1, 2)
    # 进行softmax
    rpn_softmax_scores = F.softmax(rpn_scores, dim=-1)
    rpn_fg_scores = rpn_softmax_scores[:, :, 1].contiguous()
```

```
    rpn_fg_scores = rpn_fg_scores.view(n, -1)
    rpn_scores = rpn_scores.view(n, -1, 2)
```

我们利用所有的局部特征层进行分类预测和回归预测，回归预测的结果会直接对建议框进行调整，并获得最终预测框，分类预测结果会判断建议框中是否真正包含物体和物体的种类，之后我们就可以判断预测框内物体种类。事实上在最后一步进行回归预测的时候，他们的结果都是去判断建议框是否真实包含物体，回归预测也会对其进行调整。

先验框

`anchors.py` 文件生成的基础先验框结构

```
PS D:\Study\Collection\Object_detection\FasterRCNN\pyt
ction/FasterRCNN/pytorch-faster-RCNN/utils/anchors.py
[[ -45.254833  -90.50967    45.254833   90.50967 ]
 [ -90.50967  -181.01933    90.50967   181.01933 ]
 [-181.01933  -362.03867   181.01933   362.03867 ]
 [ -64.        -64.         64.         64.      ]
 [-128.       -128.        128.        128.      ]
 [-256.       -256.        256.        256.      ]
 [ -90.50967   -45.254833   90.50967    45.254833]
 [-181.01933   -90.50967   181.01933    90.50967 ]
 [-362.03867  -181.01933   362.03867   181.01933 ]]
```

组合先验框和网格中心

```
def _enumerate_shifted_anchor(anchor_base, feat_stride, height, width):
    # 计算网格中心点
    shift_x = np.arange(0, width * feat_stride, feat_stride)
    shift_y = np.arange(0, height * feat_stride, feat_stride)
    shift_x, shift_y = np.meshgrid(shift_x, shift_y) #网格中心组合
    shift = np.stack((shift_x.ravel(),shift_y.ravel(),
                      shift_x.ravel(),shift_y.ravel(),), axis=1) #将网格中心和先验
框进行组合

    # 每个网格点上的9个先验框
    A = anchor_base.shape[0]
    K = shift.shape[0]
    anchor = anchor_base.reshape((1, A, 4)) + \
             shift.reshape((K, 1, 4))
    # 所有的先验框
    anchor = anchor.reshape((K * A, 4)).astype(np.float32)
    return anchor
```

利用先验框预测结果调整建议框

但建议框 proposal 也只是对图像中物体的粗略筛选，并非完全准确，后面还需要进一步调整。建议框会和共享特征层进行结合，传入都ROIPooling层中。以先验框的分数，来调整参数

```
def loc2bbox(src_bbox, loc):
    if src_bbox.shape[0] == 0:
        return np.zeros((0, 4), dtype=loc.dtype)
    #中心和宽高进行先验框位置确定
```

```
    src_bbox = src_bbox.astype(src_bbox.dtype, copy=False)
    src_width = src_bbox[:, 2] - src_bbox[:, 0]
    src_height = src_bbox[:, 3] - src_bbox[:, 1]
    src_ctr_x = src_bbox[:, 0] + 0.5 * src_width
    src_ctr_y = src_bbox[:, 1] + 0.5 * src_height
    #中心点位置和宽高进行调整
    dx = loc[:, 0::4]
    dy = loc[:, 1::4]
    dw = loc[:, 2::4]
    dh = loc[:, 3::4]
    #调整后的先验框
    ctr_x = dx * src_width[:, np.newaxis] + src_ctr_x[:, np.newaxis]
    ctr_y = dy * src_height[:, np.newaxis] + src_ctr_y[:, np.newaxis]
    w = np.exp(dw) * src_width[:, np.newaxis]
    h = np.exp(dh) * src_height[:, np.newaxis]
    #转变尚未筛选的先验框格式
    dst_bbox = np.zeros(loc.shape, dtype=loc.dtype)
    dst_bbox[:, 0::4] = ctr_x - 0.5 * w
    dst_bbox[:, 1::4] = ctr_y - 0.5 * h
    dst_bbox[:, 2::4] = ctr_x + 0.5 * w
    dst_bbox[:, 3::4] = ctr_y + 0.5 * h

    return dst_bbox
```

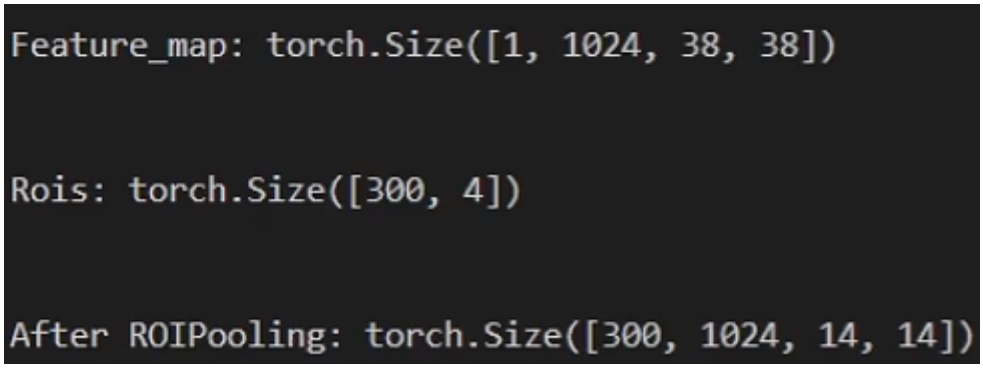
利用非极大抑制，不断去除置信度最高的框与其他框进行IOU计算，IOU过大则去掉其他框，防止一定区域框过多

```
# 取出成绩最好的一些建议框
order = score.ravel().argsort()[::-1]
if n_pre_nms > 0:
order = order[:n_pre_nms]
roi = roi[order, :]
roi = nms(roi,self.nms_thresh)
roi = torch.Tensor(roi)
roi = roi[:n_post_nms]
return roi
```

筛选出300个得分最高的建议框，然后对参数处理后将建议框放回去，后面会利用建议框对共享特征层进行截取，利用ROIPooling统一大小

ROIPooling层

ROIPooling层主要是利用C++语言编写的，直接采取网络提供的代码进行训练。文件名`roi_cupy.py`。Rois为建议框，第三行为ROIPooling后结果



利用ROIPooling层结果进行分类预测和回归预测

对resnet-50结果进行第五次压缩处理

```
def forward(self, x, rois, roi_indices):
        roi_indices = torch.Tensor(roi_indices).cuda().float()
        rois = torch.Tensor(rois).cuda().float()
        indices_and_rois = torch.cat([roi_indices[:, None], rois], dim=1)

        xy_indices_and_rois = indices_and_rois[:, [0, 1, 2, 3, 4]]
        indices_and_rois =  xy_indices_and_rois.contiguous()
        # 利用建议框对公用特征层进行截取
        pool = self.roi(x, indices_and_rois)
        fc7 = self.classifier(pool) #300，2048，1，1
        fc7 = fc7.view(fc7.size(0), -1) #300，2048
        roi_cls_locs = self.cls_loc(fc7) #分类预测
        roi_scores = self.score(fc7) #回归预测
        return roi_cls_locs, roi_scores
```

利用预测结果对建议框进行判断和调整

对建议框参数进行标准化后reshape，进行建议框维度的改变。decode过程

```
class DecodeBox():
    def __init__(self, std, mean, num_classes):
        self.std = std
        self.mean = mean
        self.num_classes = num_classes+1

    def forward(self, roi_cls_locs, roi_scores, rois, height, width,
score_thresh):

        rois = torch.Tensor(rois)
   #解码过程
        roi_cls_loc = (roi_cls_locs * self.std + self.mean)
        roi_cls_loc = roi_cls_loc.view([-1, self.num_classes, 4])
        roi = rois.view((-1, 1, 4)).expand_as(roi_cls_loc)
        cls_bbox = loc2bbox((roi.cpu().detach().numpy()).reshape((-1, 4)),
                            (roi_cls_loc.cpu().detach().numpy()).reshape((-1,
4)))
        cls_bbox = torch.Tensor(cls_bbox)
        cls_bbox = cls_bbox.view([-1, (self.num_classes), 4])
        # clip bounding box
        cls_bbox[..., 0] = (cls_bbox[..., 0]).clamp(min=0, max=width)
        cls_bbox[..., 2] = (cls_bbox[..., 2]).clamp(min=0, max=width)
        cls_bbox[..., 1] = (cls_bbox[..., 1]).clamp(min=0, max=height)
        cls_bbox[..., 3] = (cls_bbox[..., 3]).clamp(min=0, max=height)

        prob = F.softmax(torch.tensor(roi_scores), dim=1)
        raw_cls_bbox = cls_bbox.cpu().numpy()
        raw_prob = prob.cpu().numpy()
#对每个类进行IOU判断和非极大抑制的过程
        outputs = []
        for l in range(1, self.num_classes):
            cls_bbox_l = raw_cls_bbox[:, l, :]
            prob_l = raw_prob[:, l]
            mask = prob_l > score_thresh
            cls_bbox_l = cls_bbox_l[mask]
            prob_l = prob_l[mask]
```

```
            if len(prob_l) == 0:
                continue
            label = np.ones_like(prob_l)*(l-1)
            detections_class =
np.concatenate([cls_bbox_l,np.expand_dims(prob_l,axis=-1),np.expand_dims(label,a
xis=-1)],axis=-1)

            prob_l_index = np.argsort(prob_l)[::-1]
            detections_class = detections_class[prob_l_index]
            nms_out = nms(detections_class,0.3)
            if outputs==[]:
                outputs = nms_out
            else:
                outputs = np.concatenate([outputs,nms_out],axis=0)
        return outputs
```

Faster R-CNN模型表现提升

使用更好的特征网络

通过将VGG-16替换成ResNet-101，mAP从73.2%提高到76.4% 。在速度方面，ResNet比VGG16更慢，同时需要训练的次数也更多，（论文数据）vgg16 训练一轮耗时1.5s，ResNet版本一轮耗时2.0s，但是内存占用量也远远大于VGG16。

voc_weights_resnet.pth	2020/7/7 2:20	PTH 文件	111,488 KB
voc_weights_vgg.pth	2020/7/7 2:21	PTH 文件	535,470 KB

但是由于Faster R-CNN训练漫长，来不及再用vgg训练一次网络了，但是vgg部分的代码附在了代码文件中

Hierarchy Feature特征层次来加速训练速度

将多层次的卷积网络feature map接在一起。将conv1，conv3，conv5三层接在一起，形成一个Hyper Feature，以Hpyer Feature maps 代替原有的conv5_3，用于RPN和Fast-RCNN。由于CNN的本身特点，随着层数加深，特征变得越来越抽象和语义，但分辨率却随之下降。相当于可以获得更好的精度，而conv5_3代表的语义信息对分类有帮助，结合下来，相当于一个定位精度和分类精度的折中。

论文结论https://arxiv.org/pdf/1604.00600.pdf，由于时间问题未对此优化方案进行实现和测试

柔性非极大抑制Soft-NMS

```
while np.shape(detection)[0]>0:
    best_box.append(detection[0])
    if len(detection) == 1:
        break
    ious = iou(best_box[-1],detection[1:])
    detection[1:,4] = np.exp(-(ious * ious) / sigma)*detection[1:,4]
    detection = detection[1:]
    scores = detection[:,4]
    arg_sort = np.argsort(scores)[::-1]
    detection = detection[arg_sort]
```

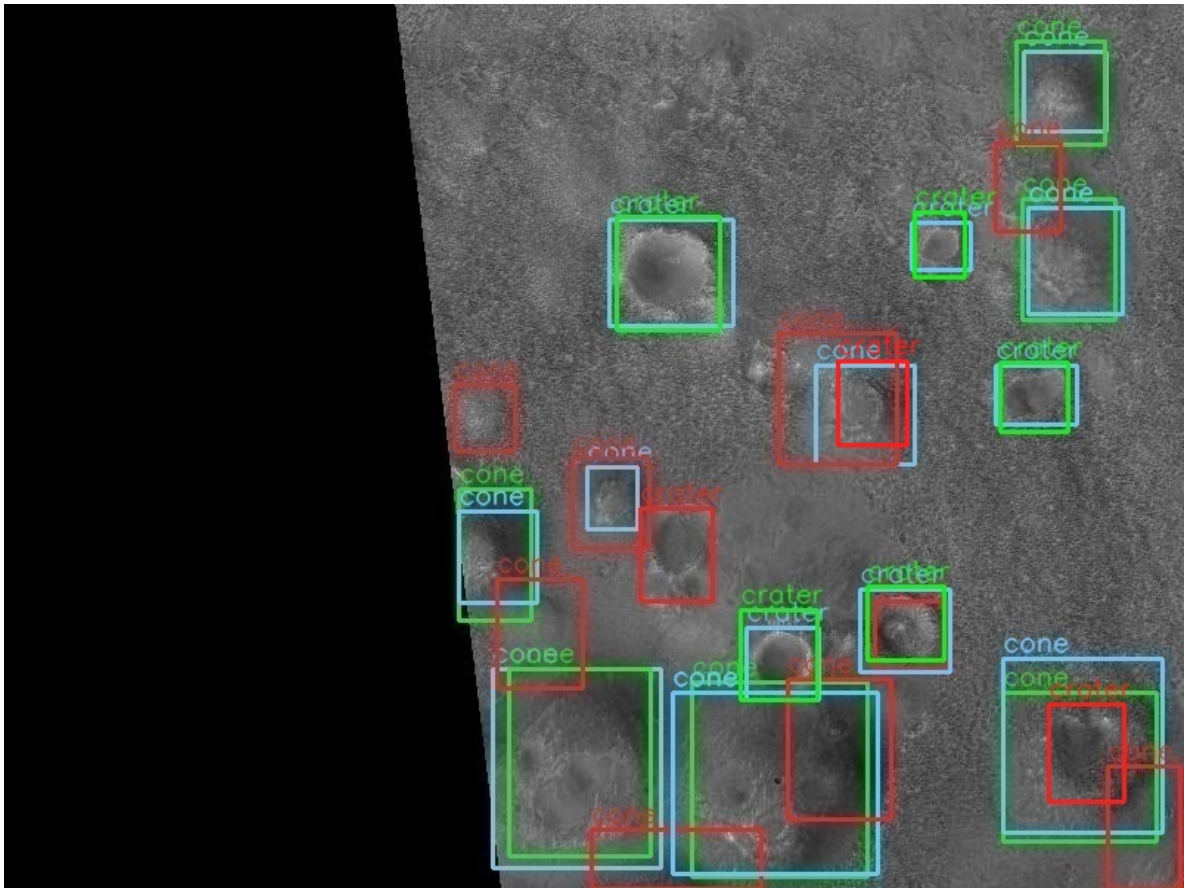
作为一种后处理方式，Soft-NMS相比之前代码使用的非极大抑制不同，Soft-NMS认为不应该直接只通过重合程度进行筛选，Soft-NMS认为在进行非极大抑制的时候要同时考虑得分和重合程度。对于NMS而言，其直接将与得分最大的框重合程度较高的其它预测剔除。而Soft-NMS则以一个权重的形式，将获得的IOU取高斯指数后乘上原得分，之后重新排序。继续循环。会将结果准确度提升几个百分点。

Faster R-CNN实验结果

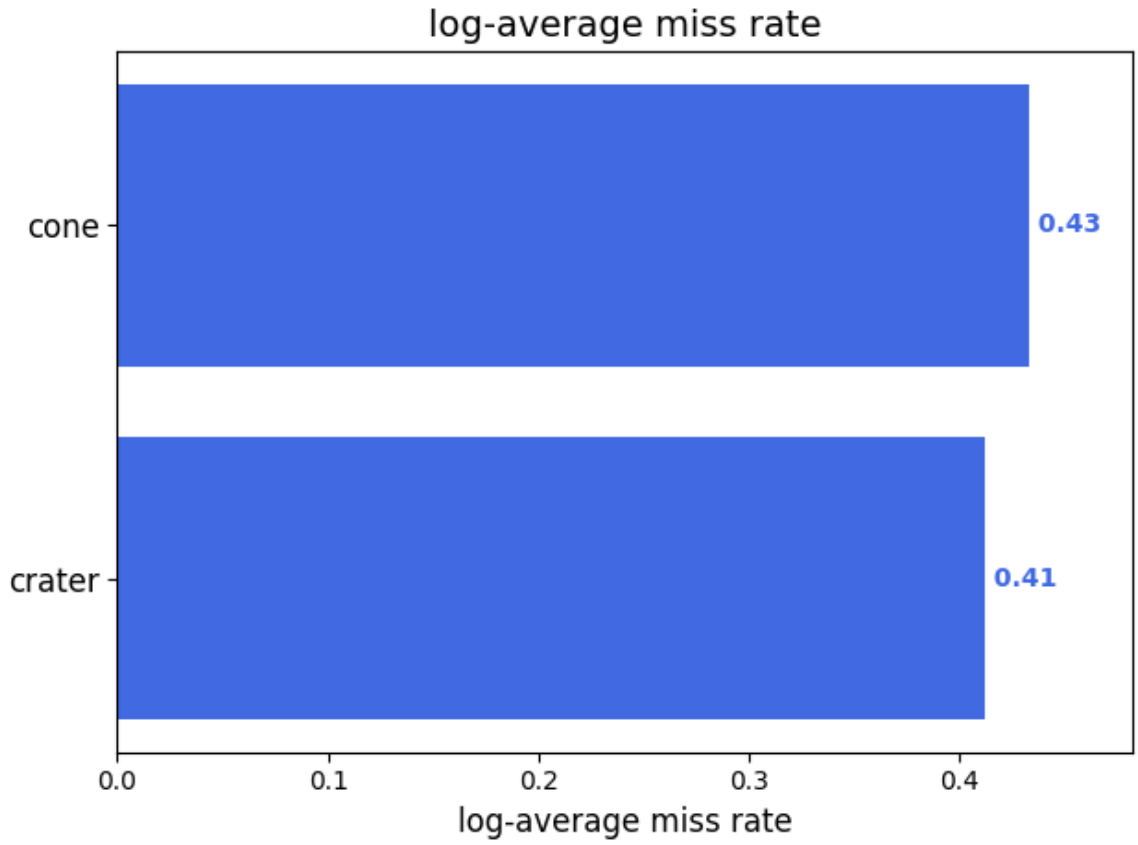
训练25个世代后网络总loss为0.6811

```
Epoch:25/25
iter:999/1000 || total_loss: 0.6811|| rpn_loc_loss: 0.1667 || rpn_cls_loss: 0.1419 || roi_loc_loss: 0.1843 || roi_cls_loss: 0.1882 || 0.6379s/step
Start Validation
Finish Validation

Epoch:25/25
Total Loss: 0.6804 || Val Loss: 0.8194
Saving state, iter: 25
```

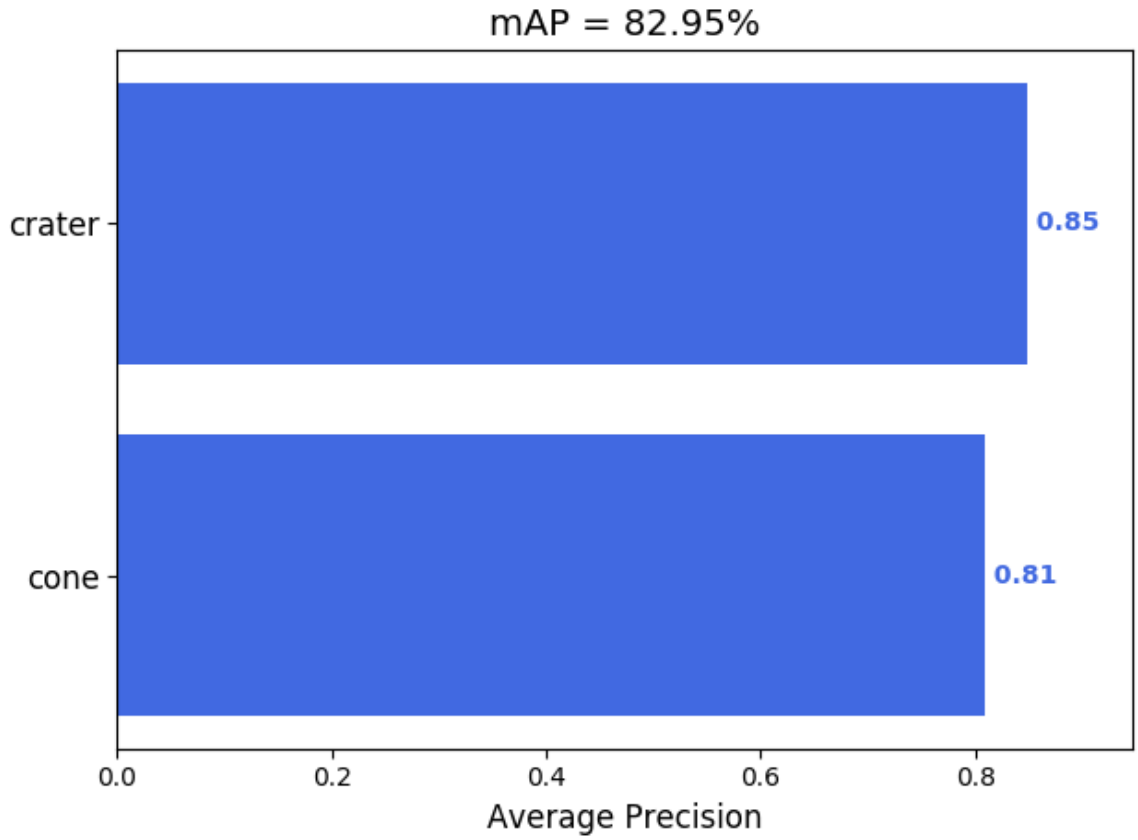
运行`mudESP_040775_2235_RED03_13.jpg`测试模型效果，可见比起xml中标记，R-CNN能够得到更多的cone和crater更加精确



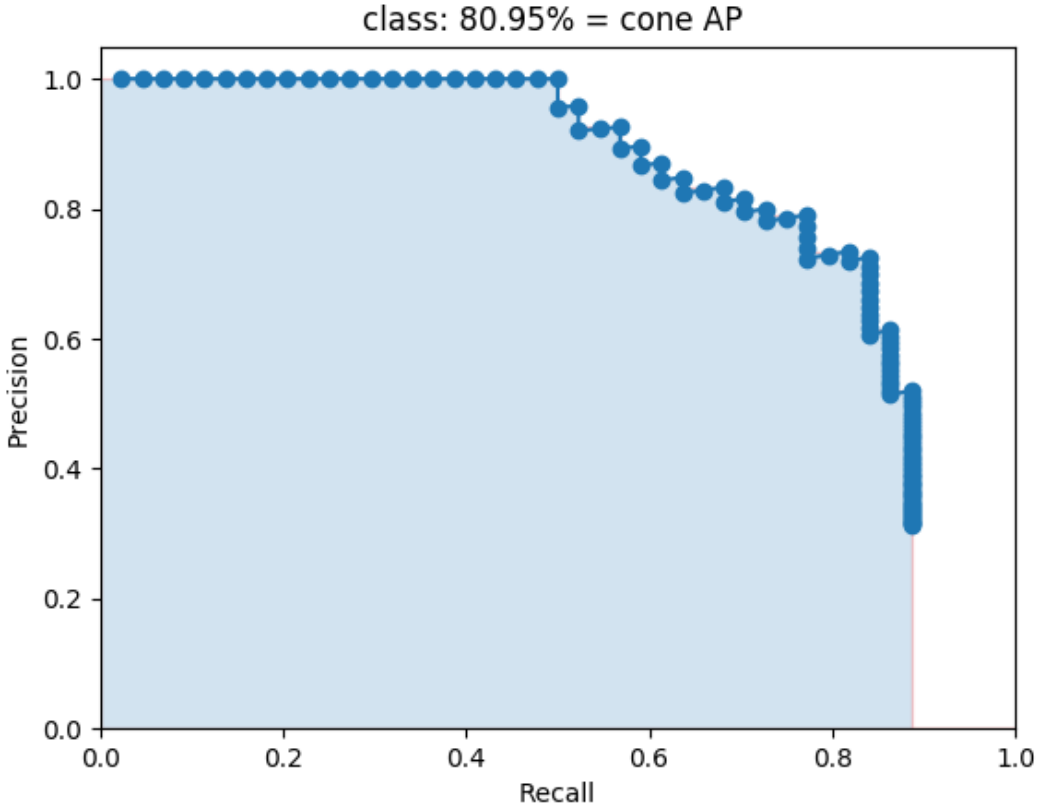
全测试集的识别精度Accuracy如图

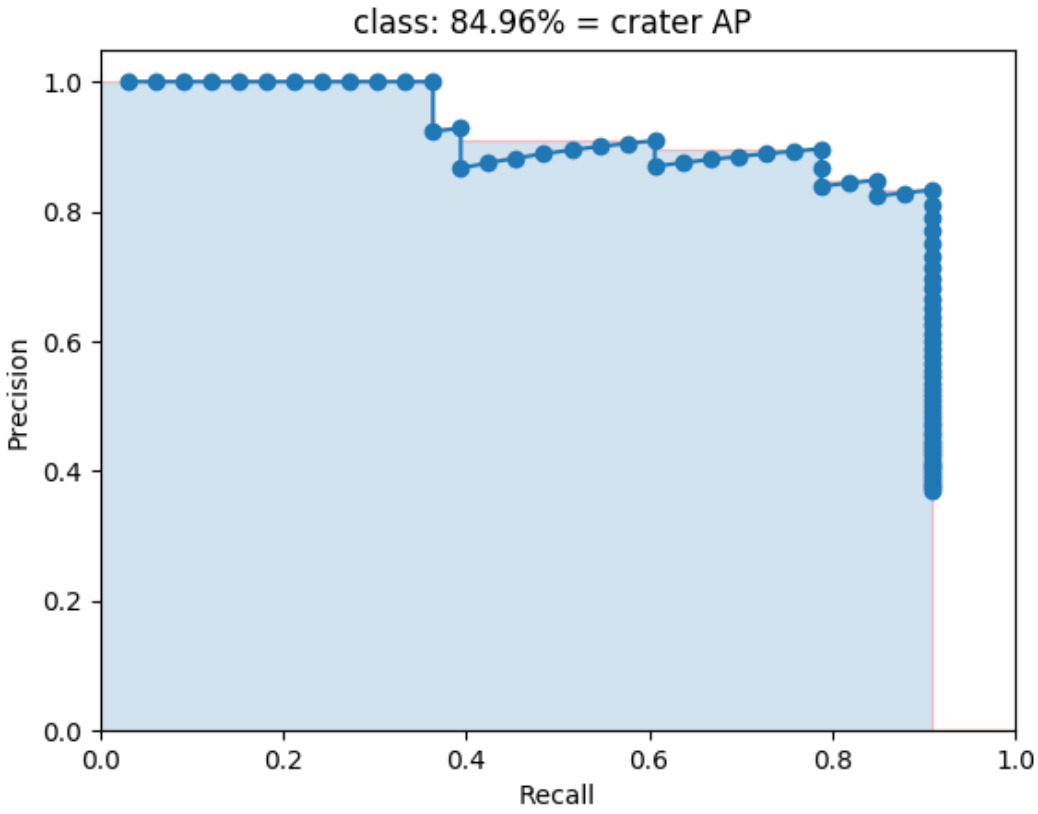


经过对测试集的测试，得到的mAP为82.95%，其中cone的AP为81%，crater的AP为85%



cone和crater两个类型的召回率Recall和准确率Precision分别绘制如图


class: 80.95% = cone AP
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0


class: 84.96% = crater AP
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0

参考文献

1. 睿智的目标检测11——Keras搭建yolo3目标检测平台 https://blog.csdn.net/weixin_44791964/article/details/103276106
2. 睿智的目标检测27——Pytorch搭建Faster R-CNN目标检测平台 https://blog.csdn.net/weixin_44791964/article/details/105739918
3. 神经网络学习小记录45——Keras常用学习率下降方式汇总 https://blog.csdn.net/weixin_44791964/article/details/105334098
4. 目标检测中precision、recall、AP、mAP的含义 https://blog.csdn.net/yangzzguang/article/details/80540375
5. python PIL 图像处理 https://www.jianshu.com/p/e8d058767dfa
6. Faster R-CNN 深入理解 && 改进方法汇总 https://blog.csdn.net/zchang81/article/details/73176497
7. 睿智的目标检测20——利用mAP计算目标检测精确度 https://blog.csdn.net/weixin_44791964/article/details/104695264