西北大学

智能信息系统综合实践
实验报告

题　　目：　　目标检测

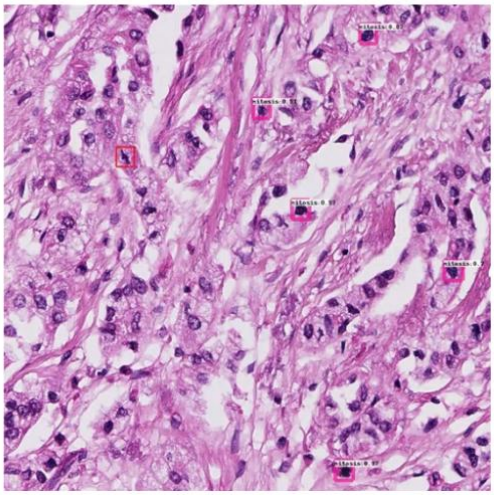
年　　级：　　2020

专　　业：　　软件工程

姓　　名：　　朱泽昊

一、题目（原题目）

有丝分裂细胞检测



提供数据集
训练集：313张图像
测试集：80张图像
标注格式：.xml，可再提供xml转为txt/csv的代码
（这三种格式几乎支持开源你的目标检测程序）

代码：
提供Faster RCNN代码（包含数据预处理、训练、测试、指标计算），tensorflow

目标：
请在测试集中达到更好的结果
AP，recall

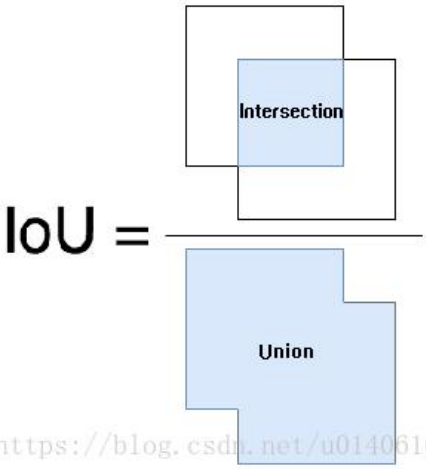
二、解题步骤（思路+代码）

目标检测： 目标检测（Object Detection）是一种在图像或视频中自动检测和定位多个目标的任务。与图像分类任务不同，目标检测不仅需要判断图像中是否存在某个特定物体，还需要确定物体的位置和大小。

目标检测任务可以分为两个主要步骤：候选框生成和目标分类与位置回归。候选框生成指的是在图像中生成多个可能包含目标的候选框，这些候选框通常是由一些启发式算法（如滑动窗口、区域提议等）生成的。目标分类与位置回归是指对于每个候选框，通过深度神经网络进行分类和位置回归，以确定该候选框是否包含目标，并准确地定位目标。

在目标检测中，深度学习的方法主要有两种：基于区域的卷积神经网络（R-CNN）和单阶段检测器。R-CNN包括三个主要部分：候选框生成、卷积神经网络特征提取和目标分类与位置回归。这种方法虽然准确率高，但速度较慢。为了解决这个问题，单阶段检测器（如 YOLO 和 SSD）被提出。这些方法不需要候选框生成步骤，直接在图像上进行密集的分类和回归，速度快但准确率相对较低。

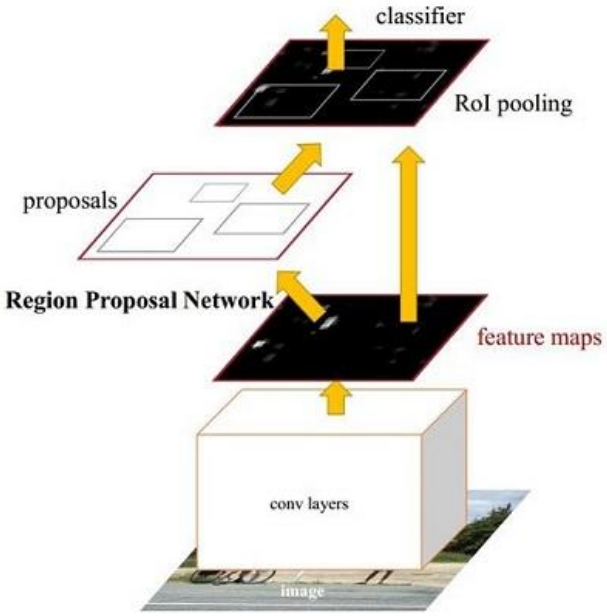
目标检测任务的评价指标包括精度、召回率、准确率等等。其中，精度指的是检测框与真实框之间的 IoU（Intersection over Union）值，召回率指的是成功检测到的目标数与总目标数之间的比例，而准确率指的是成功检测到的目标数与总检测框数之间的比例。



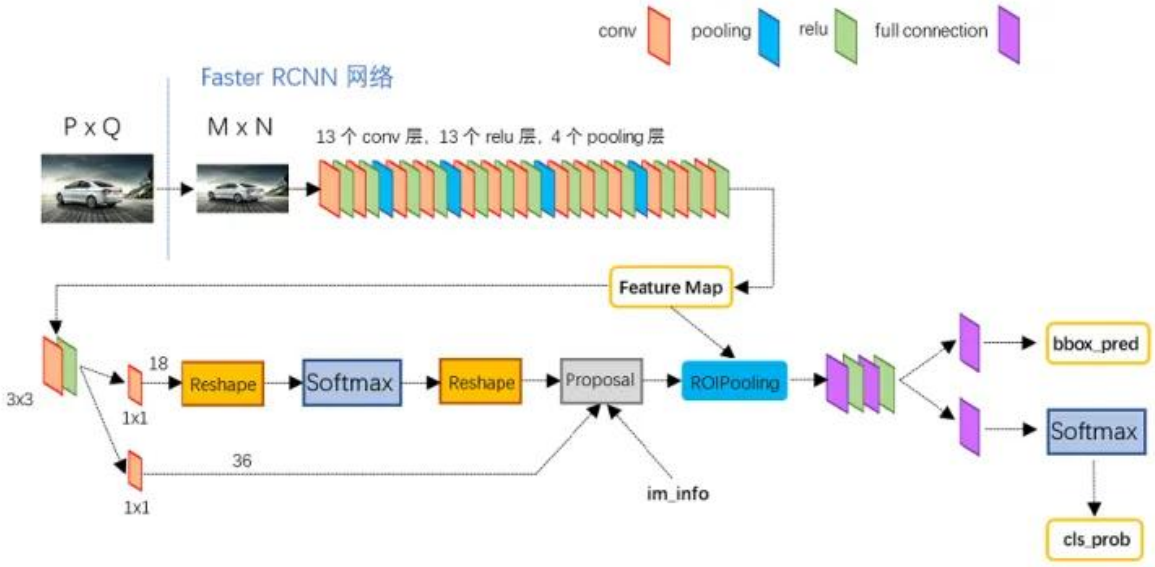
在本实验中我选择使用 Faster R-CNN，SSD，retinanet 模型对本数据集进行目标检测，并且画出 map 曲线以及展示一张图片上面不同模型的效果。本实验以使用 resnet50 作为 backbone，并且使用预训练模型为例子，讲述实验流程。

Faster-rcnn： Faster R-CNN 由两个主要部分组成：提取特征的卷积神经网络（CNN）和用于预测目标边界框的区域提议网络（Region Proposal Network，RPN）。

在 Faster R-CNN 中，RPN 网络首先对输入图像进行卷积特征提取，然后在每个位置生成一组锚点框，并在这些锚点框上预测目标的边界框和目标类别。在训练过程中，RPN 网络通过计算目标边界框和锚点框之间的 IoU（Intersection over Union）来确定正负样本，并使用基于 IoU 的损失函数进行优化。在测试阶段，RPN 网络会生成一些候选框，然后送入 RoI Pooling 层中进行特征提取，并通过分类器和回归器预测每个框的目标类别和边界框。



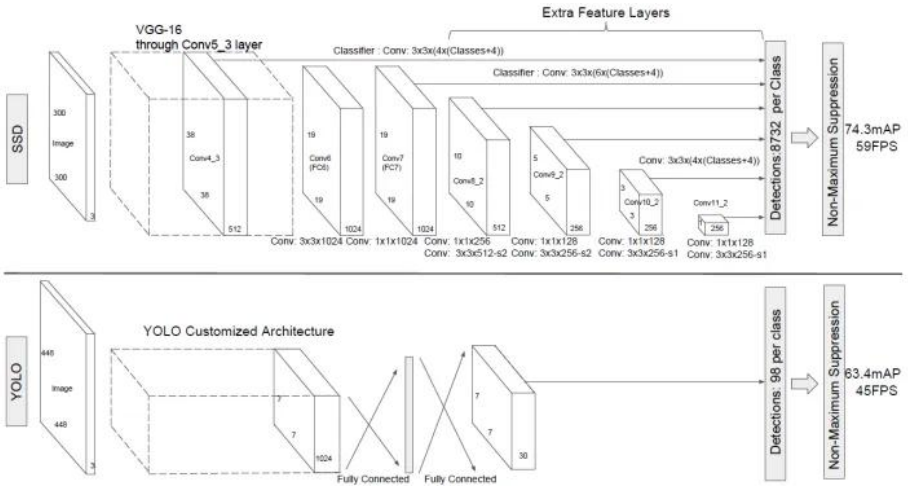
Faster-rcnn 网络基本结构



Faster-rcnn 网络详细结构

SSD：（Single Shot MultiBox Detector）是一种单阶段目标检测算法。与传统的两阶段目标检测算法（如 Faster R-CNN）不同，SSD 在一张图像上只需要进行一次前向传播，即可同时进行目标检测和定位，因此检测速度很快。

SSD 主要由两个部分组成：特征提取网络和多尺度特征图上的预测网络。特征提取网络通常采用预训练好的深度卷积神经网络，如 VGG16 或 ResNet 等，用于提取输入图像的卷积特征。预测网络则对特征图上的每个位置进行目标检测和定位，并输出每个检测框的位置和类别信息。



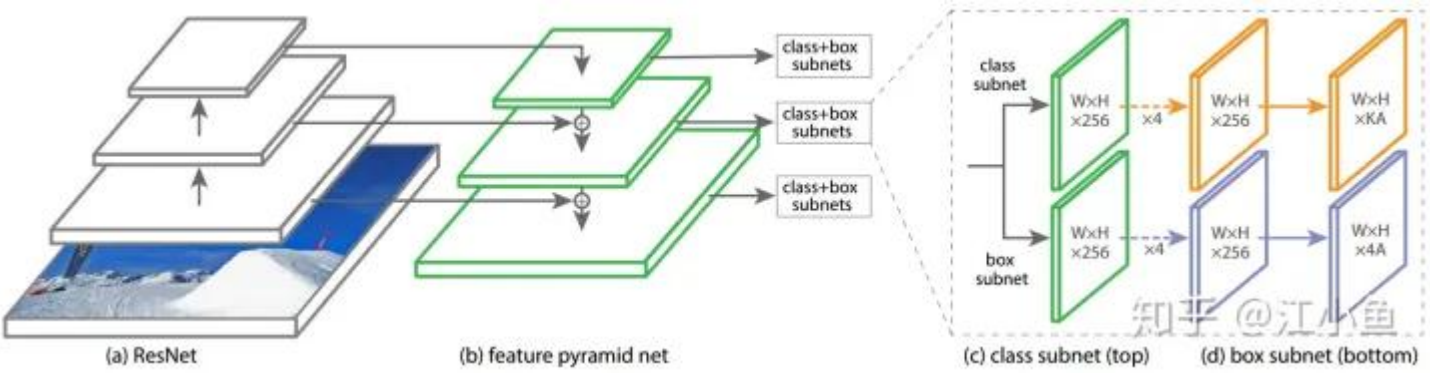
SSD 中的预测网络在特征图的不同尺度上进行预测，以检测不同大小的目标。为了实现这个目标，SSD 在网络中插入了多个卷积层和池化层，用于提取多个不同尺度的特征图，并在每个特征图上预测检测框。

RetinaNet：其设计目的是解决早期目标检测算法存在的缺点，如物体小目标检测精度低、较大目标检测速度慢等问题。

RetinaNet 采用了一种新颖的基于特征金字塔网络（Feature Pyramid Network，FPN）的方法，即通过建立高层次特征和低层次特征之间的连接，从而实现对各种尺度的目标进行检测。此外，RetinaNet 还引入了一种新的损失函数，称为 Focal Loss，可有效缓解类别不平衡问题，从而提高小目标的检测精度。

与传统的目标检测算法相比，RetinaNet 具有以下优点：

1. 高准确率：在物体小目标检测方面表现突出，准确率显著提高。
2. 高效性：在不降低准确率的情况下，检测速度比以往算法更快。
3. 高可扩展性：可以在多种计算机视觉任务中应用。

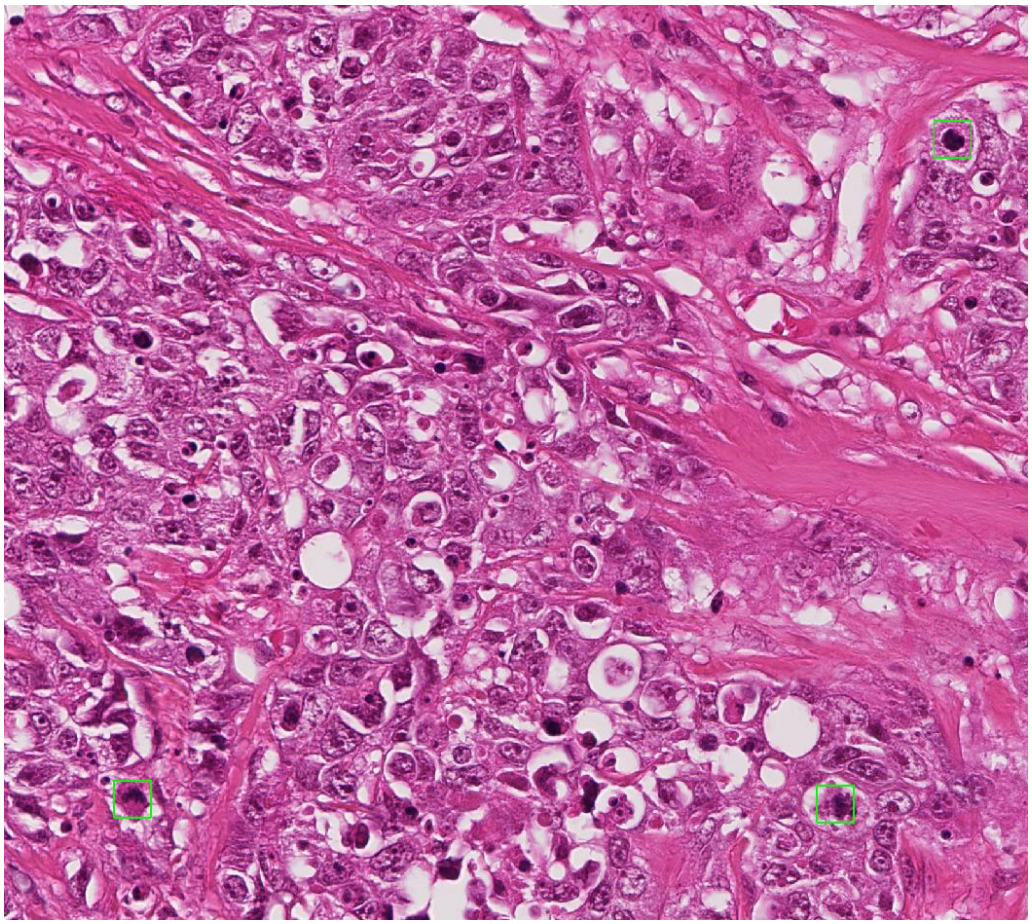


解题步骤：

（1）首先是将数据处理，目标数据集的标注是 Pascalvoc 格式的数据，所以要先将拿到的数据集转换成通用的 pascalvoc 格式的数据。所以要将 train，test 文件夹下的所有图片的信息分别读取到两个 txt 文件之中，文件保存两个文件夹下数据的名字，但是省去了它们的后缀名。之后将所有的标注文件以及所有的图片混合在一起。**（在实验之中有一个很大的问题，该标注不能直接使用通用的读取方式进行读取，因为该标注信息有 xml 的版本信息和编码信息，要先把前面的这些信息处理掉才能进行读取）**


```
├── Annotations                所有的图像标注信息(XML文件)
├── ImageSets
│   ├── Action                 人的行为动作图像信息
│   ├── Layout                 人的各个部位图像信息
│   ├── Main                   目标检测分类图像信息
│   │   ├── train.txt          训练集 5717
│   │   ├── val.txt            验证集 5823
│   │   └── trainval.txt       训练集+验证集 11540
│   └── Segmentation           目标分割图像信息
├── JPEGImages                 所有图像文件
├── SegmentationClass          图像分割png图（基于类别）
└── SegmentationObject         图像分割png图（基于目标）
```

（2）先将数据集中随机选取一张图片进行标注框显示，并且之后所有算法都使用这张图片进行预测。



（3）图片预处理：首先需要对输入的图像进行预处理，以便后续算法能够更好地对其进行处理。预处理包括将图像转换为合适的格式（如 RGB 或灰度），进行归一化处理，以及进行图像增强等操作。

（4）候选框生成：目标检测算法通常采用候选框（也称为区域提议）的方式来检测目标。候选框生成算法的目的是在图像中生成一组可能包含目标的矩形框。常见的候选框生成算法包括 Selective Search、Region Proposal Network（RPN）等。

（5）特征提取：对于每个候选框，需要提取其特征以便后续分类器进行分类。常见的特征提取算法包括卷积神经网络（CNN）和基于特征金字塔的算法等。

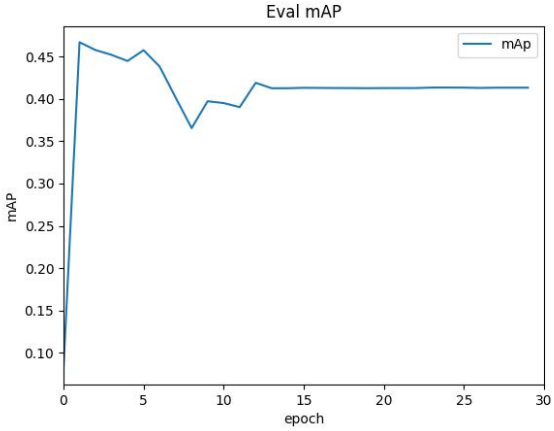
（6）目标分类：在提取出候选框的特征后，需要使用分类器对其进行分类，以确定其中是否包含目标物体。常用的分类器包括支持向量机（SVM）、逻辑回归（LR）、神经网络等。

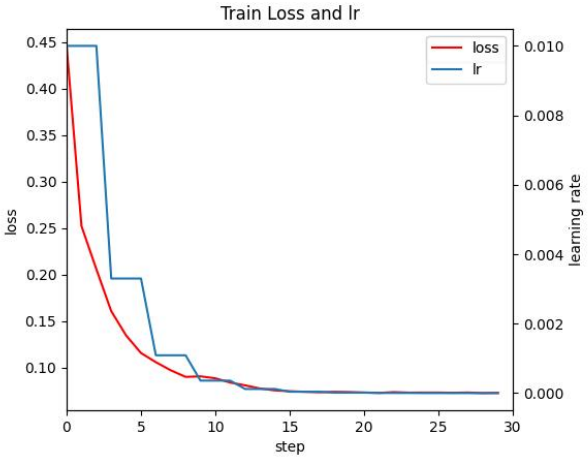
（7）边框回归：在确定了目标物体的存在后，需要进一步调整候选框的位置和大小，以更好地匹配目标物体的实际位置。边框回归算法的目的就是调整候选框的位置和大小，使其更好地与目标物体匹配。

（8）后处理：最后，需要对检测结果进行后处理，以去除重叠的候选框和错误的检测结果等。常见的后处理算法包括非极大值抑制（NMS）等。

结果：

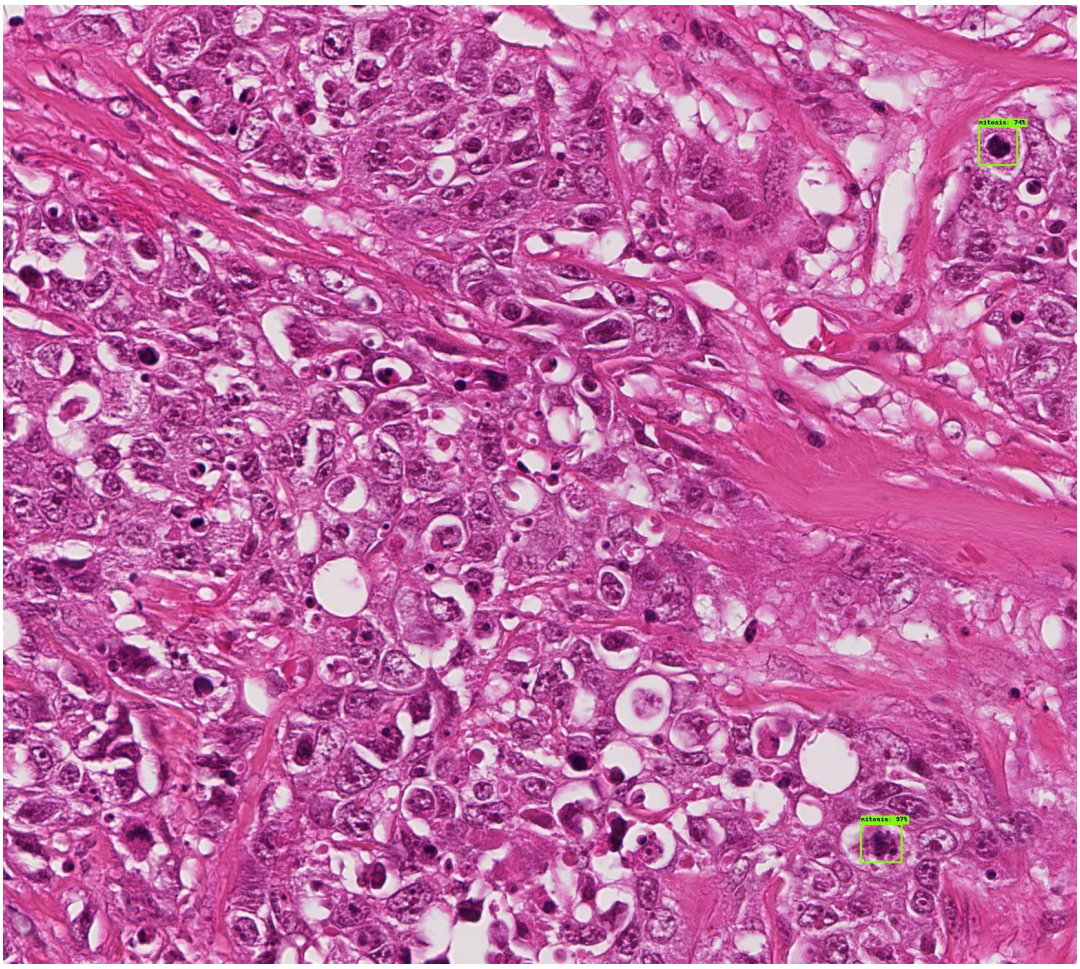
（1）使用 resnet50 作为 faster-rcnn 的 backbone，使用 fpn



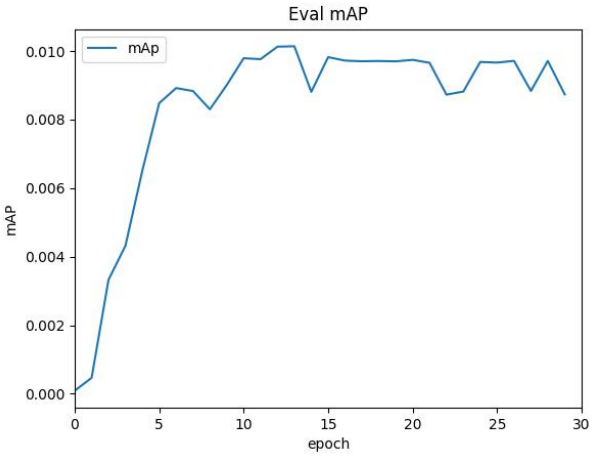


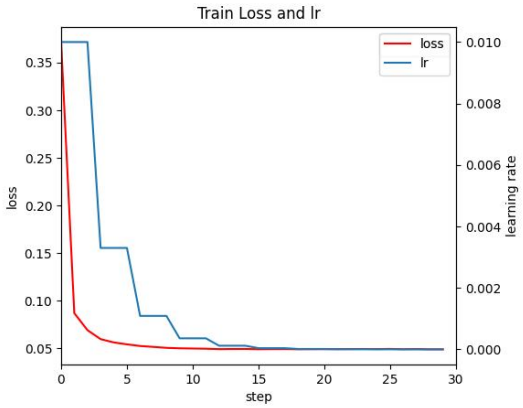

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.1874
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.4133
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.1598
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.1874
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.1530
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.3163
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.3163
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.3163
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 mAP 为 0.4133，recall 为 0.3163



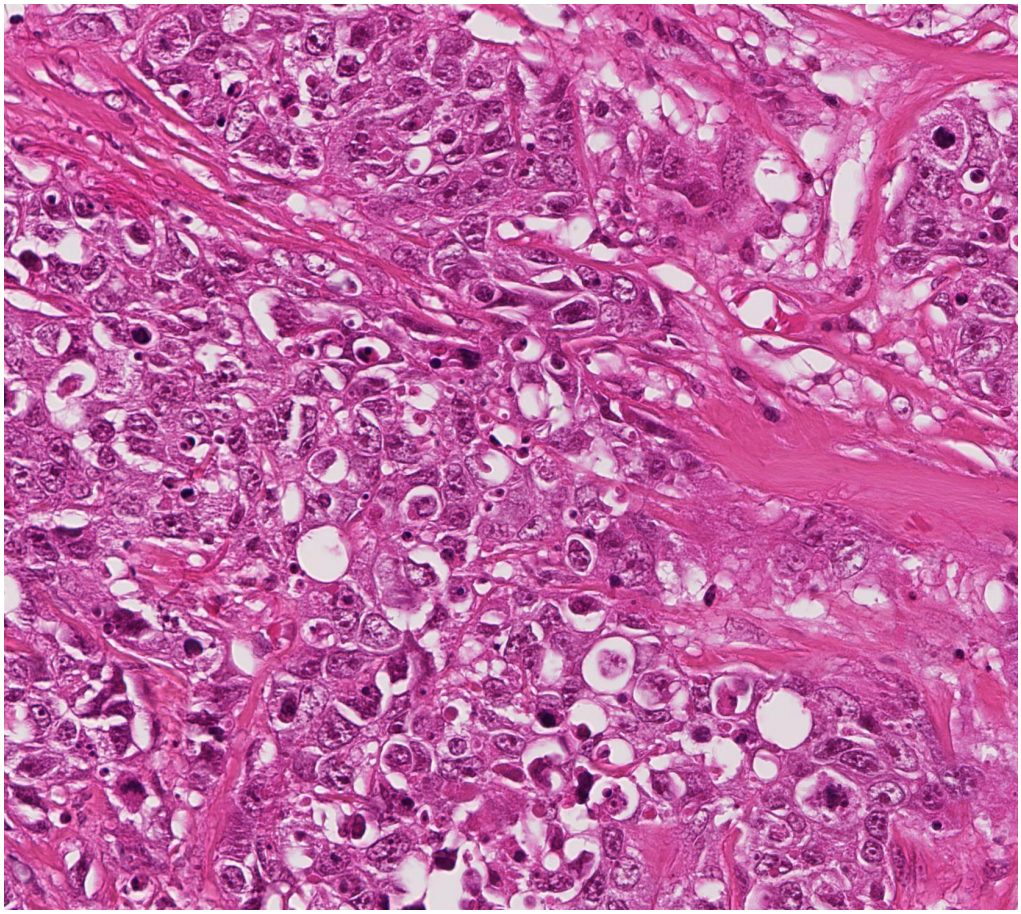
（2）使用 mobilenet_v2 作为 faster-rcnn 的 backbone，不使用 fpn



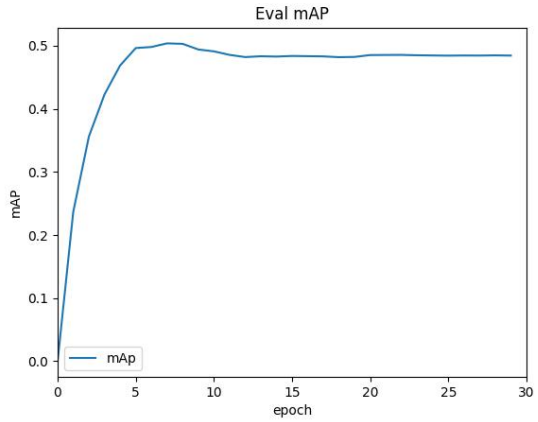


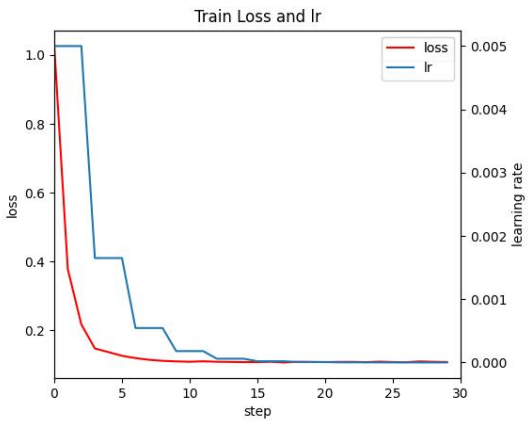
```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.0021
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.0087
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.0003
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.0025
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.0127
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.0410
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.0554
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.0554
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 **mAP 为 0.0087，recall 为 0.0554**



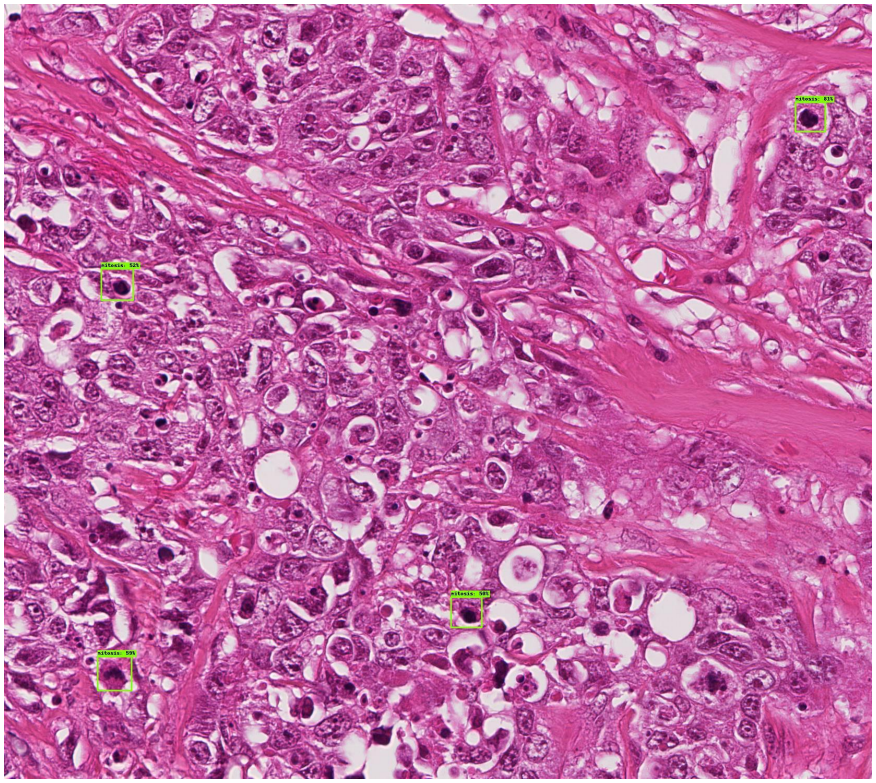
（3）使用 mobilenet_v3_large 作为 faster-rcnn 的 backbone，使用 fpn



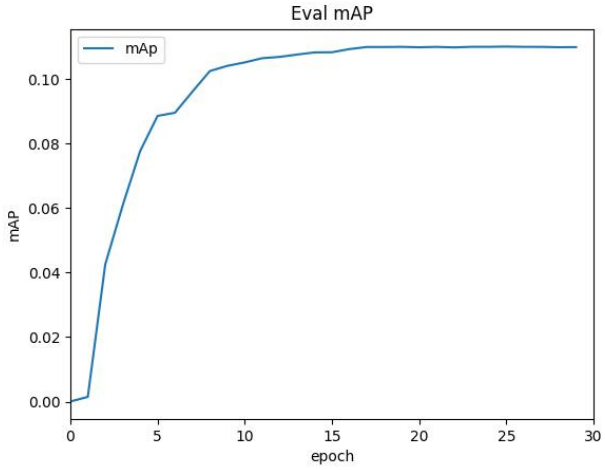


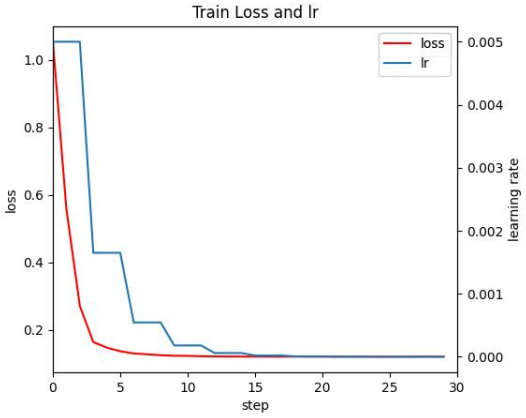

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.2243
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.4845
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.1609
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.2244
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.1584
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.4271
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.4639
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.4639
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 **mAP 为 0.4845，recall 为 0.4639**



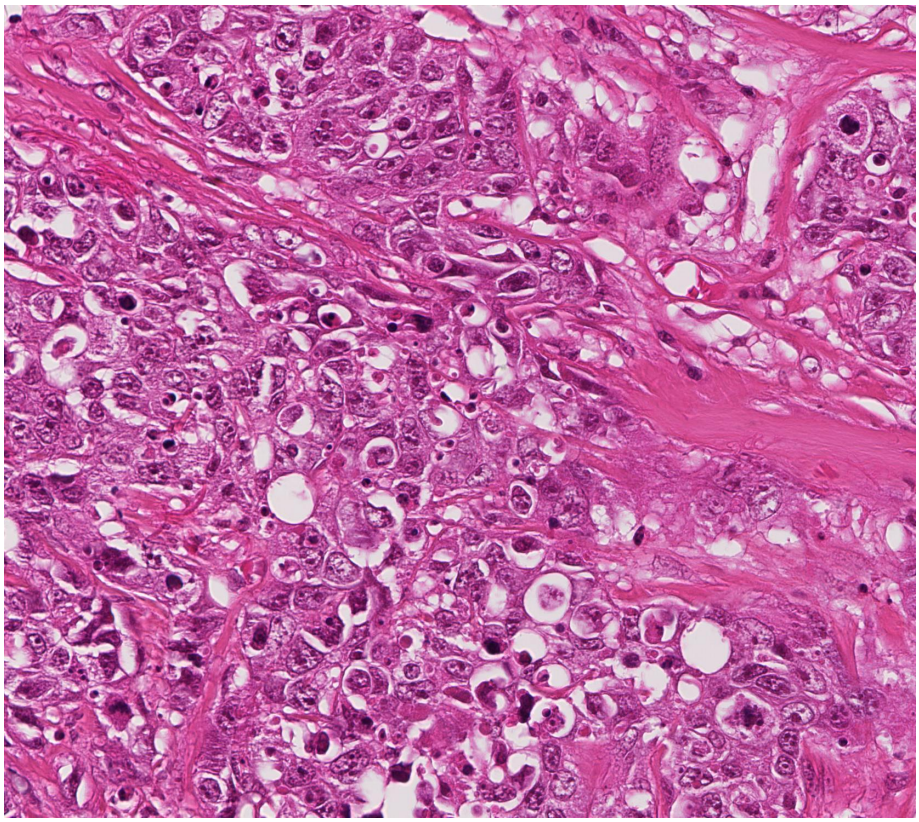
（4）使用 vgg16 作为 faster-rcnn 的 backbone，不使用 fpn



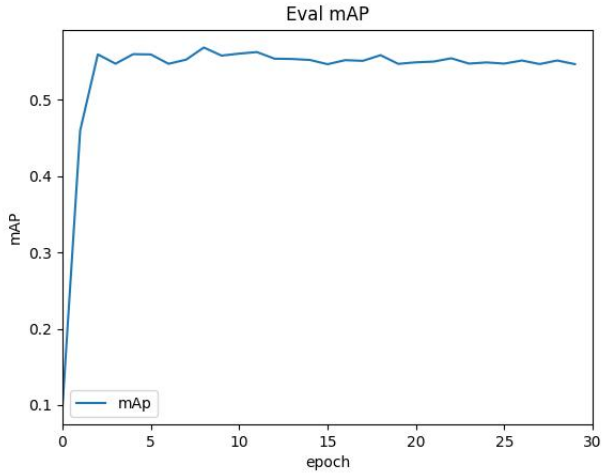


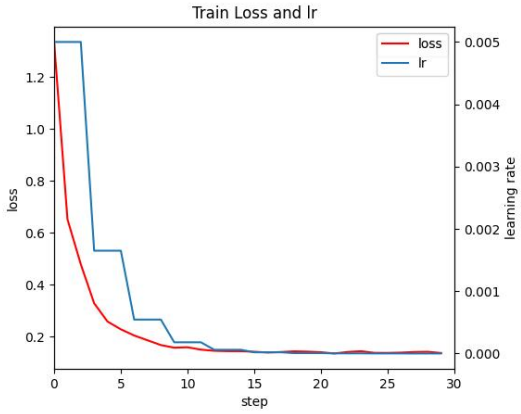

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.0276
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.1099
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.0076
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.0277
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.0458
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.1361
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.1488
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.1488
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 mAP 为 0.1099，recall 为 1488



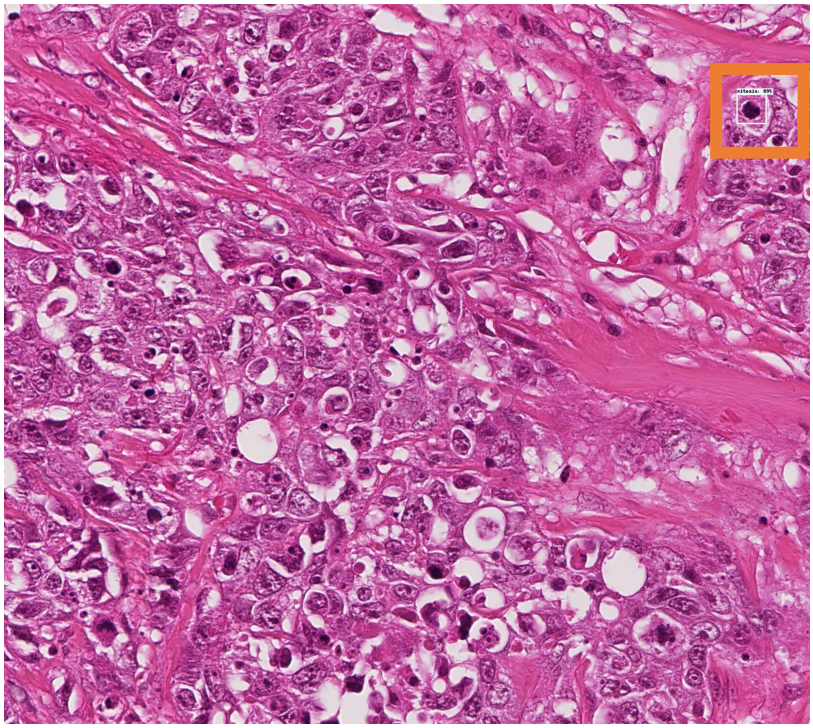
（5）使用 resnet50 作为 retinanet 的 backbone



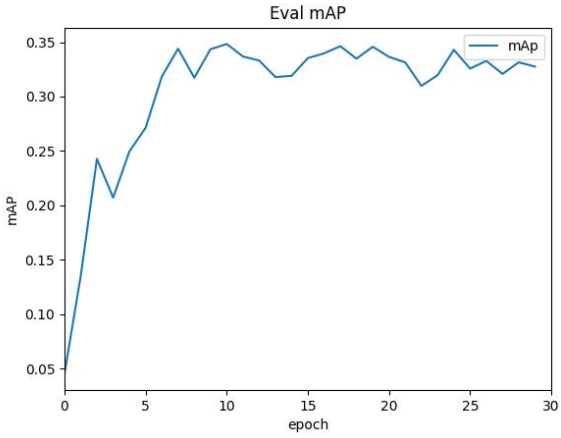


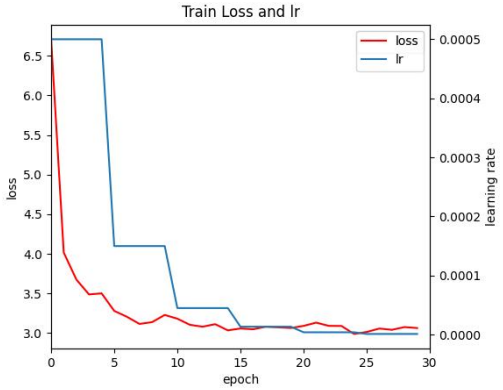
```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.2649
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.5469
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.2037
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.2649
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.1783
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.5301
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.5494
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.5494
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 **mAP 为 0.5469，recall 为 0.5494**



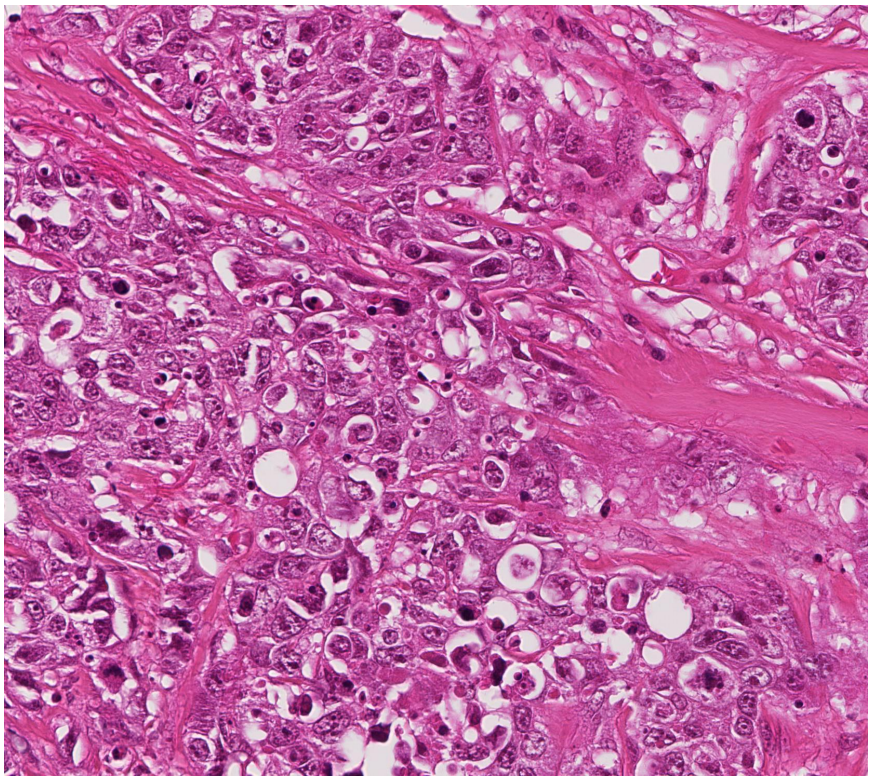
（6）使用 SSD





```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.139
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.3276
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.0806
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.14
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.1108
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.3482
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.4283
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.4283
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = -1.000
```

在最后一个测试集的输出我们可以看到 **mAP** 为 **0.3276**，**recall** 为 **0.4283**



代码：（以 **resnet50** 作为 **backbone**，使用 **faster-rcn** 为例）

```python
def create_model(num_classes, load_pretrain_weights=True):
    # 注意，这里的backbone默认使用的是FrozenBatchNorm2d，即不会去更新bn参数
    # 目的是为了防止batch_size太小导致效果更差(如果显存很小，建议使用默认的FrozenBatchNorm2d)
    # 如果GPU显存很大可以设置比较大的batch_size就可以将norm_layer设置为普通的BatchNorm2d
    # trainable_layers包括['layer4', 'layer3', 'layer2', 'layer1', 'conv1']， 5代表全部训练
    # resnet50 imagenet weights url: https://download.pytorch.org/models/resnet50-0676ba61.pth
    backbone = resnet50_fpn_backbone(pretrain_path="pytorch_object_detection/faster_rcnn/backbone/resnet50.pth",
                                     norm_layer=torch.nn.BatchNorm2d,
                                     trainable_layers=3)
    # 训练自己数据集时不要修改这里的91，修改的是传入的num_classes参数
    model = FasterRCNN(backbone=backbone, num_classes=91)

    if load_pretrain_weights:
        # 载入预训练模型权重
        # https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
        weights_dict = torch.load("pytorch_object_detection/faster_rcnn/backbone/fasterrcnn_resnet50_fpn_coco.pth", map_location='cpu')
        missing_keys, unexpected_keys = model.load_state_dict(weights_dict, strict=False)
        if len(missing_keys) != 0 or len(unexpected_keys) != 0:
            print("missing_keys: ", missing_keys)
            print("unexpected_keys: ", unexpected_keys)

    # get number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

```python
def main(args):
    device = torch.device(args.device if torch.cuda.is_available() else "cpu")
    print("Using {} device training.".format(device.type))

    # 用来保存coco_info的文件
    results_file = "results{}.txt".format(datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))

    data_transform = {
        "train": transforms.Compose([transforms.ToTensor()]),
        "val": transforms.Compose([transforms.ToTensor()])
    }

    VOC_root = args.data_path
    # check voc root

    # load train data set
    # VOCdevkit -> VOC2012 -> ImageSets -> Main -> train.txt
    train_dataset = VOCDataSet(VOC_root, data_transform["train"], "train.txt")

    # 是否按图片相似高宽比采样图片组成batch
    # 使用的话能够减小训练时所需GPU显存，默认使用
    if args.aspect_ratio_group_factor >= 0:
        train_sampler = torch.utils.data.RandomSampler(train_dataset)
        # 统计所有图像高宽比例在bins区间中的位置索引
        group_ids = create_aspect_ratio_groups(train_dataset, k=args.aspect_ratio_group_factor)
        # 每个batch图片从同一高宽比例区间中取
        train_batch_sampler = GroupedBatchSampler(train_sampler, group_ids, args.batch_size)

    # 注意这里的collate_fn是自定义的，因为读取的数据包括image和targets，不能直接使用默认的方法合成batch
    batch_size = args.batch_size
    nw = min([os.cpu_count(), batch_size if batch_size > 1 else 0, 8])  # number of workers
    print('Using %g dataloader workers' % nw)
```

```
train_data_loader = torch.utils.data.DataLoader(train_dataset,
                                                batch_size=batch_size,
                                                num_workers=nw,
                                                collate_fn=train_dataset.collate_fn)

# load validation data set
# VOCdevkit -> VOC2012 -> ImageSets -> Main -> val.txt
val_dataset = VOCDataSet(VOC_root, data_transform["val"], "test.txt")
val_data_set_loader = torch.utils.data.DataLoader(val_dataset,
                                                  batch_size=1,
                                                  shuffle=False,
                                                  pin_memory=True,
                                                  num_workers=nw,
                                                  collate_fn=val_dataset.collate_fn)

# create model num_classes equal background + 20 classes
model = create_model(num_classes=args.num_classes + 1)
# print(model)

model.to(device)

# define optimizer
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params,
                            lr=args.lr,
                            momentum=args.momentum,
                            weight_decay=args.weight_decay)
```

```
# learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.33)

# 如果指定了上次训练保存的权重文件地址，则接着上次结果接着训练
if args.resume != "":
    checkpoint = torch.load(args.resume, map_location='cpu')
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    lr_scheduler.load_state_dict(checkpoint['lr_scheduler'])
    args.start_epoch = checkpoint['epoch'] + 1
    if args.amp and "scaler" in checkpoint:
        scaler.load_state_dict(checkpoint["scaler"])
    print("the training process from epoch{}...".format(args.start_epoch))

train_loss = []
learning_rate = []
val_map = []
```

```python
for epoch in range(args.start_epoch, args.epochs):
    # train for one epoch, printing every 10 iterations
    mean_loss, lr = utils.train_one_epoch(model, optimizer, train_data_loader,
                                          device=device, epoch=epoch,
                                          print_freq=10, warmup=True,
                                          scaler=scaler)
    train_loss.append(mean_loss.item())
    learning_rate.append(lr)

    # update the learning rate
    lr_scheduler.step()

    # evaluate on the test dataset
    coco_info = utils.evaluate(model, val_data_set_loader, device=device)

    # write into txt
    with open(results_file, "a") as f:
        # 写入的数据包括coco指标还有loss和learning rate
        result_info = [f"{i:.4f}" for i in coco_info + [mean_loss.item()]] + [f"{lr:.6f}"]
        txt = "epoch:{} {}".format(epoch, '  '.join(result_info))
        f.write(txt + "\n")

    val_map.append(coco_info[1])  # pascal mAP

    # save weights
    save_files = {
        'model': model.state_dict(),
        'optimizer': optimizer.state_dict(),
        'lr_scheduler': lr_scheduler.state_dict(),
        'epoch': epoch}
    if args.amp:
        save_files["scaler"] = scaler.state_dict()
    torch.save(save_files, "./save_weights/resNetFpn-model-{}.pth".format(epoch))
```

```python
        # save weights
        save_files = {
            'model': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'lr_scheduler': lr_scheduler.state_dict(),
            'epoch': epoch}
        if args.amp:
            save_files["scaler"] = scaler.state_dict()
        torch.save(save_files, "./save_weights/resNetFpn-model-{}.pth".format(epoch))

    # plot loss and lr curve
    if len(train_loss) != 0 and len(learning_rate) != 0:
        from plot_curve import plot_loss_and_lr
        plot_loss_and_lr(train_loss, learning_rate)

    # plot mAP curve
    if len(val_map) != 0:
        from plot_curve import plot_map
        plot_map(val_map)
```

```python
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(
        description=__doc__)

    # 训练设备类型
    parser.add_argument('--device', default='cuda:0', help='device')
    # 训练数据集的根目录(VOCdevkit)
    parser.add_argument('--data-path', default='/mnt/d/Users/19692/Desktop/dataset/', help='dataset')
    # 检测目标类别数(不包含背景)
    parser.add_argument('--num-classes', default=1, type=int, help='num_classes')
    # 文件保存地址
    parser.add_argument('--output-dir', default='./save_weights', help='path where to save')
    # 若需要接着上次训练，则指定上次训练保存权重文件地址
    parser.add_argument('--resume', default='', type=str, help='resume from checkpoint')
    # 指定接着从哪个epoch数开始训练
    parser.add_argument('--start_epoch', default=0, type=int, help='start epoch')
    # 训练的总epoch数
    parser.add_argument('--epochs', default=30, type=int, metavar='N',
                        help='number of total epochs to run')
    # 学习率
    parser.add_argument('--lr', default=0.01, type=float,
                        help='initial learning rate, 0.02 is the default value for training '
                             'on 8 gpus and 2 images_per_gpu')
    # SGD的momentum参数
    parser.add_argument('--momentum', default=0.9, type=float, metavar='M',
                        help='momentum')
    # SGD的weight_decay参数
    parser.add_argument('--wd', '--weight-decay', default=1e-4, type=float,
                        metavar='W', help='weight decay (default: 1e-4)',
                        dest='weight_decay')
    # 训练的batch size
    parser.add_argument('--batch_size', default=8, type=int, metavar='N',
                        help='batch size when training.')
    parser.add_argument('--aspect-ratio-group-factor', default=3, type=int)
    # 是否使用混合精度训练(需要GPU支持混合精度)
    parser.add_argument("--amp", default=True, help="Use torch.cuda.amp for mixed precision training")
```

三、总结（心得体会）

1.训练所得模型的 precision 和 AP 值都不高，这可能是因为训练集较小，只有 313 张图，且模型只训练了一次，并未进行参数调优，导致模型预测的边界框太多，precision 较低，从而使得计算的 AP 也比较小。

2.精度较低也有一个原因可能是训练的时间不够，因为我为了控制变量，每一个网络都只训练了 30 轮，可能有些网络还是欠拟合。

3.在这几项实验之中，我们可以看出 retainnet 确实像在论文中介绍的这个样子，在检测小目标的时候效果较好 mAP 为 0.5469，recall 为 0.5494

使用 mobilenet_v3_large 作为 faster-rcnn 的 backbone，使用 fpn，mAP 为 0.4845，recall 为 0.4639 效果也是很好

但是如果不适用 fpn 那么网络的结果就不是很好 mAP 都在 0.1 附近

4.这个细胞的展示图只是为了展示使用，不代表效果好的网络在本张图片上面展示效果很好