深 圳 大 学 实 验 报 告

课程名称：人工智能课程实训

实验项目名称：实验 4 模型加速实践

学院：计算机与软件学院

专业：软件工程（腾班）

指导教师：王旭

报告人：洪子敬 学号：2022155033 班级：腾班

实验时间：2024 年 11 月 07 日至 11 月 27 日

实验报告提交时间：2024 年 11 月 23 日

教务处制

实验目的与要求：

目标：

1. 了解常见模型加速基本方法、原理；
2. 掌握常见模型压缩的方式，并且自行比较不同方法对模型的影响；
3. 掌握基于 TensorRT 或 PyTorch 模型优化工具包的模型加速和压缩方法。

基本要求：

1. 在常见方法中（如：包括结构优化、剪枝（Pruning）、量化（Quantization）、知识蒸馏（Knowledge Distillation））选择至少 2 种方式对实验 3 中训练的模型进行优化；
2. 完成模型的基本压缩和加速推理；并给出性能对比。

高级要求：

3. 尝试进一步提升模型准确度、压缩率和推理速度；并给出性能对比。

方法、步骤：

1. 在实验三模型部署的基础上，利用量化（Quantization）和剪枝（Pruning）对 Yolov5s 模型进行优化；
2. 对量化后不同精度的模型对给定的数据集进行推理，比较推理时间和推理结果，并给出性能对比；
3. 对剪枝后不同规模的模型在 coco val2017 上进行测试，比较推理时间、精度和召回率等指标，并给出性能比较。

实验过程及内容：

1. 模型优化

（1） 基础模型

在实验三的基础上，我们利用 coco128 数据集对 Yolov5s 模型进行训练，得到最佳权重 best.pt 用于后续的模型推理和优化；并人为挑选了用于模型推理的图片数据集存放在 inference/images 目录下，该数据集如下所示：

（2） 模型量化（Quantization）

模型量化主要指的是通过将模型中的浮点数权重和激活值转换为低精度格式（如整数）来减少模型的存储需求和计算复杂度。这种方法在需要在资源有限的设备上部署模型时特别受用，已经广泛用于深度学习网络中。

在 Yolov5s 的官方文档中，我们可以看到其本身就支持模型的量化，包括 FP32、FP16 和 INT8 三种精度。因此本实验我们将从这三种精度分别进行模型

的推理和测试，并对结果进行比较。

（3） 模型剪枝（Pruning）

剪枝是一种用于优化深度学习模型的方法，旨在减少模型的复杂性和计算需求，同时尽可能保持模型的性能。通过去除不重要或冗余的神经元、连接或权重从而提高模型的效率和速度。

主要的剪枝方法包括：

a. **权重剪枝**：根据权重的大小，去除那些绝对值较小的权重。这通常是最常见的剪枝方法。

b. **神经元剪枝**：去除一些神经元（也称为通道）及其相关的连接，通常基于其对模型输出的贡献。

c. **结构化剪枝**：按层或模块进行剪枝，而不是单个权重，使得剪枝后的模型更易于在硬件上实现。

（本实验采用的主要是第一种权重剪枝）

2. 性能对比

（1）模型量化（Quantization）

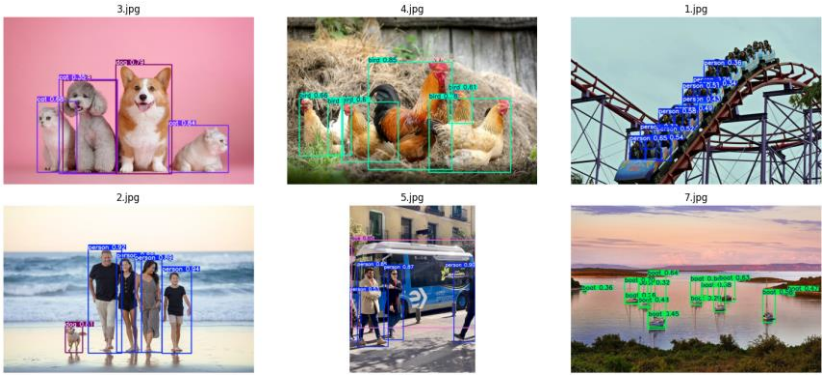
a. 对于原模型

对于指定数据集的推理结果如下：

```
image 1/7 /content/yolov5/inference/images/1.jpg: 448x640 11 persons, 27.7ms
image 2/7 /content/yolov5/inference/images/2.jpg: 448x640 4 persons, 1 dog, 9.5ms
image 3/7 /content/yolov5/inference/images/3.jpg: 448x640 3 cats, 2 dogs, 9.4ms
image 4/7 /content/yolov5/inference/images/4.jpg: 448x640 6 birds, 9.4ms
image 5/7 /content/yolov5/inference/images/5.jpg: 640x480 4 persons, 1 bus, 30.1ms
image 6/7 /content/yolov5/inference/images/6.jpg: 384x640 2 persons, 2 ties, 29.1ms
image 7/7 /content/yolov5/inference/images/7.jpg: 448x640 14 boats, 9.5ms
Speed: 0.4ms pre-process, 17.8ms inference, 103.5ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp
```

由结果可知对于给定数据集中的每张图片推理耗时 17.8ms。

部分检测结果如下：



b. 对于 FP32 量化

使用 Yolov5s 的 export 文件导出 32 位存储的 engine 和 onnx 格式模型：

```
# FP32量化
!python export.py --weights yolov5/experiment3/weights/best.pt --include onnx engine --device 0
```

运行可以得到 29.7MB 大小的 engine 文件和 28.0MB 的 onnx 文件：

```
TensorRT: export success ✅ 170.5s, saved as yolov5/experiment3/weights/best.engine (29.7 MB)

ONNX: starting export with onnx 1.17.0...
ONNX: export success ✅ 0.7s, saved as yolov5/experiment3/weights/best.onnx (28.0 MB)
```

使用得到的 engine 文件对给定的数据集进行推理：

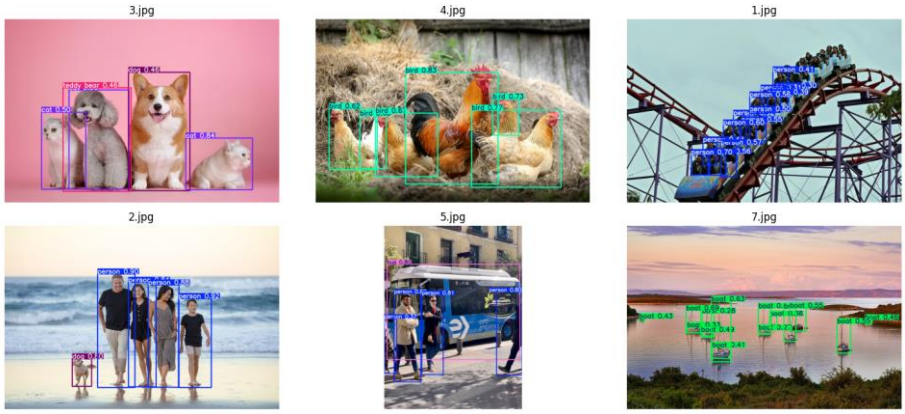
```
# FP32引擎推理
!python detect.py --weights yolov5/experiment3/weights/FP32.engine --source inference/images
```

运行后可以得到以下结果：

```
image 1/7 /content/yolov5/inference/images/1.jpg: 640x640 13 persons, 10.9ms
image 2/7 /content/yolov5/inference/images/2.jpg: 640x640 4 persons, 1 dog, 10.9ms
image 3/7 /content/yolov5/inference/images/3.jpg: 640x640 3 cats, 1 dog, 1 teddy bear, 10.9ms
image 4/7 /content/yolov5/inference/images/4.jpg: 640x640 6 birds, 10.8ms
image 5/7 /content/yolov5/inference/images/5.jpg: 640x640 4 persons, 1 bus, 10.9ms
image 6/7 /content/yolov5/inference/images/6.jpg: 640x640 2 persons, 2 ties, 10.8ms
image 7/7 /content/yolov5/inference/images/7.jpg: 640x640 14 boats, 10.9ms
Speed: 4.8ms pre-process, 10.9ms inference, 82.8ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp3
```

由结果可知对于给定数据集中的每张图片推理耗时 10.9ms，通过对比可以知道模型的速度对于原模型提升接近一倍。

FP32 模型部分检测结果如下：



c. 对于 FP16 量化

在 FP32 的基础上加上“--half”命令，转为浮点数 16 位：

```
# FP16位量化
!python export.py --weights yolov5/experiment3/weights/best.pt --include onnx engine --half --device 0
```

运行可以得到 15.7MB 大小的 engine 文件和 14.0MB 大小的 onnx 文件，模型大小相比 FP32 减少了接近一半：

```
TensorRT: export success ✅ 471.1s, saved as yolov5/experiment3/weights/best.engine (15.7 MB)

ONNX: starting export with onnx 1.17.0...
ONNX: export success ✅ 0.5s, saved as yolov5/experiment3/weights/best.onnx (14.0 MB)
```

同样使用得到的 engine 文件对给定的数据集进行推理：

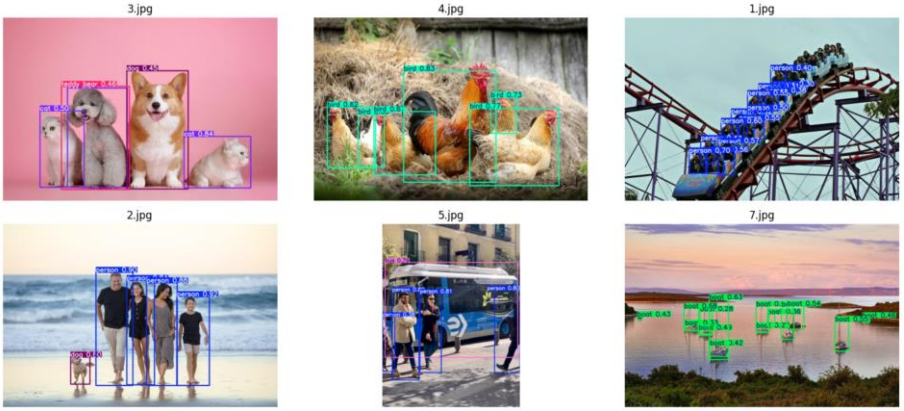
```
# FP16引擎推理
!python detect.py --weights yolov5/experiment3/weights/FP16.engine --source inference/images
```

运行后可以得到以下结果：

```
image 1/7 /content/yolov5/inference/images/1.jpg: 640x640 13 persons, 4.1ms
image 2/7 /content/yolov5/inference/images/2.jpg: 640x640 4 persons, 1 dog, 4.2ms
image 3/7 /content/yolov5/inference/images/3.jpg: 640x640 3 cats, 1 dog, 1 teddy bear, 4.1ms
image 4/7 /content/yolov5/inference/images/4.jpg: 640x640 6 birds, 4.2ms
image 5/7 /content/yolov5/inference/images/5.jpg: 640x640 4 persons, 1 bus, 4.1ms
image 6/7 /content/yolov5/inference/images/6.jpg: 640x640 2 persons, 2 ties, 4.1ms
image 7/7 /content/yolov5/inference/images/7.jpg: 640x640 14 boats, 4.1ms
Speed: 3.6ms pre-process, 4.1ms inference, 82.7ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp5
```

由结果可知对于给定数据集中的每张图片推理耗时 4.1ms，通过对比可以知道模型的速度对于 FP32 模型提升一倍之多。

FP16 模型部分检测结果如下：



观察结果可知，此时 16 位模型的检测效果与 32 位的效果基本一样，在模型体积减小，推理速度加快的情况下，能够达到检测效果基本一样，说明 16 位量化效果十分成功。

d. 对于 INT8 量化

在 FP32 的基础上加上“—int8”命令，转为无符号整数 8 位：

```
# int8位量化
!python export.py --weights yolov5/experiment3/weights/best.pt --include onnx engine --int8 --device 0
```

运行可以得到 31.9MB 大小的 engine 文件和 28.0MB 大小的 onnx 文件，模型大小与 FP32 接近，模型大小减少的效果不好：

```
TensorRT: export success ✅ 116.0s, saved as yolov5/experiment3/weights/best.engine (31.9 MB)

ONNX: starting export with onnx 1.17.0...
ONNX: export success ✅ 0.9s, saved as yolov5/experiment3/weights/best.onnx (28.0 MB)
```

同样使用得到的 engine 文件对给定的数据集进行推理：

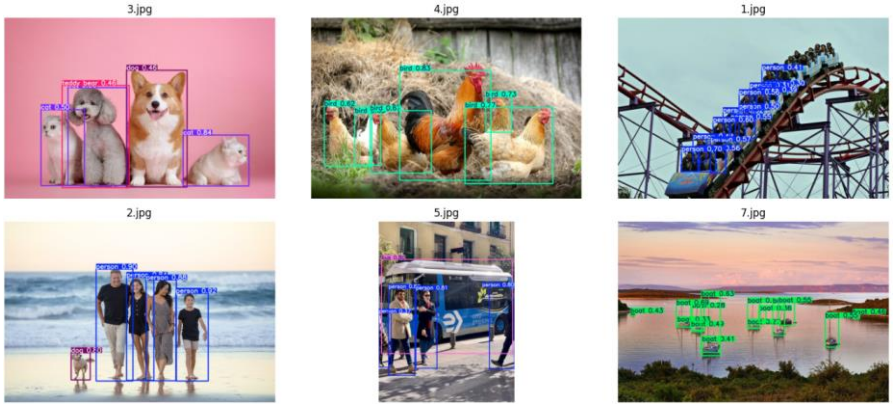
```
# int8位引擎推理
!python detect.py --weights yolov5/experiment3/weights/int8.engine --source inference/images
```

运行后可以得到以下结果：

```
image 1/7 /content/yolov5/inference/images/1.jpg: 640x640 13 persons, 10.9ms
image 2/7 /content/yolov5/inference/images/2.jpg: 640x640 4 persons, 1 dog, 10.9ms
image 3/7 /content/yolov5/inference/images/3.jpg: 640x640 3 cats, 1 dog, 1 teddy bear, 10.8ms
image 4/7 /content/yolov5/inference/images/4.jpg: 640x640 6 birds, 10.8ms
image 5/7 /content/yolov5/inference/images/5.jpg: 640x640 4 persons, 1 bus, 10.8ms
image 6/7 /content/yolov5/inference/images/6.jpg: 640x640 2 persons, 2 ties, 10.8ms
image 7/7 /content/yolov5/inference/images/7.jpg: 640x640 14 boats, 10.8ms
Speed: 3.7ms pre-process, 10.9ms inference, 86.3ms NMS per image at shape (1, 3, 640, 640)
Results saved to runs/detect/exp6
```

由结果可知对于给定数据集中的每张图片推理耗时 10.9ms，与 FP32 模型的推理速度基本一致。

INT8 模型部分检测结果如下：



观察结果可知，此时 INT8 位模型的检测效果与 FP32 位的效果基本一样，而模型大小和推理速度基本一致的情况下，达到检测效果基本一样，说明 INT8 位量化效果并不佳，可能是 Yolov5s 自带的检测效果并不好有关。

（2）模型剪枝（Pruning）

阅读 Yolo5vs 官方文档可知，通过测试不同剪枝稀疏程度的模型在 coco val2017 数据集上的表现来进行性能比较。

a. 对于原模型

使用 val 文件进行数据集的验证，还是使用之前训练好的权重，同时指定精度为 FP16（前面已经验证过效果比较好）提升速度：

```
# 正常测试
! python val.py --weights yolov5/experiment3/weights/best.pt --data ./data/coco.yaml --img 640 --half
```

运行命令后可以得到以下结果，一张图片 2.6nms，总共耗时 2.8ms：

```
val: New cache created: /content/datasets/coco/val2017.cache
               Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:18<00:00,  1.99it/s]
                 all       5000      36335      0.66      0.512      0.554      0.355
Speed: 0.1ms pre-process, 2.8ms inference, 2.6ms NMS per image at shape (32, 3, 640, 640)
```

设定基础的 mAP 为 0.02（选定阈值在 0.5-0.95 之间）：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.020
 Average Precision  (AP) @[ IoU=0.50       | area=   all | maxDets=100 ] = 0.044
 Average Precision  (AP) @[ IoU=0.75       | area=   all | maxDets=100 ] = 0.016
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.017
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.038
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.017
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.043
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.084
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.097
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.036
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.096
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.127
Results saved to runs/val/exp3
```

b. 对于 10%剪枝模型

修改 val.py 文件，添加下面部分：

```
# 修建模型至0.1稀疏度
from utils.torch_utils import prune
prune(model, 0.1)
# Configure
model.eval()
is_coco = isinstance(data.get('val'),str) and data['val'].endswith('coco/val2017.txt')
nc =1 if single_cls else int(data['nc'])    # number of classes
iouv = torch.linspace(0.5, 0.95, 10).to(device)    # iou vector for mAP@0.5:0.95
niou = iouv.numel()
```

运行命令后可以得到以下结果，一张图片 2.6nms，总共耗时 2.7ms：

```
val: Scanning /content/datasets/coco/val2017.cache... 4952 images, 48 backgrounds, 0 corrupt: 100% 5000/5000 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:16<00:00,  2.05it/s]
                   all       5000      36335       0.66      0.512      0.551      0.355
Speed: 0.1ms pre-process, 2.7ms inference, 2.6ms NMS per image at shape (32, 3, 640, 640)
```

此时 mAP 为 0.358，相比基础平均检测精度提升了很多，速度变化不大，mAP 图如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.358
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.557
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.385
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.197
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.467
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.302
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.503
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.554
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.371
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.611
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.703
Results saved to runs/val/exp7
```

c. 对于 20%剪枝模型

同样修改 val.py 文件，把传入 prune 参数改为 0.2 即可；运行命令后可以得到以下结果，一张图片 2.8nms，总共耗时 2.7ms：

```
val: Scanning /content/datasets/coco/val2017.cache... 4952 images, 48 backgrounds, 0 corrupt: 100% 5000/5000 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:17<00:00,  2.02it/s]
                   all       5000      36335      0.649      0.502      0.539      0.344
Speed: 0.2ms pre-process, 2.7ms inference, 2.8ms NMS per image at shape (32, 3, 640, 640)
```

此时 mAP 为 0.347，相比基础平均精度也提升了很多，与 10%剪枝效果接近，稍逊色一点，mAP 图如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.347
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.546
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.370
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.192
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.396
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.444
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.296
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.496
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.548
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.359
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.607
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.694
Results saved to runs/val/exp6
```

d. 对于 30%剪枝模型

同样修改 val.py 文件，把传入 prune 参数改为 0.3 即可；运行命令后可以得到以下结果，一张图片 2.8nms，总共耗时 2.6ms：

```
val: Scanning /content/datasets/coco/val2017.cache... 4952 images, 48 backgrounds, 0 corrupt: 100% 5000/5000 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:19<00:00,  1.97it/s]
                   all       5000      36335      0.631      0.466      0.503      0.309
Speed: 0.2ms pre-process, 2.8ms inference, 2.6ms NMS per image at shape (32, 3, 640, 640)
```

此时 mAP 为 0.312，相比基础平均精度也提升了很多，但比 10%、20%剪枝效果还是差一点，mAP 图如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.312
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.510
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.328
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.166
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.360
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.409
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.277
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.462
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.514
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.325
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.571
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.662
Results saved to runs/val/exp2
```

e．对于 40%剪枝模型

同样修改 val.py 文件，把传入 prune 参数改为 0.4 即可；运行命令后可以得到以下结果，一张图片 2.8nms，总共耗时 2.6ms：

```
val: Scanning /content/datasets/coco/val2017.cache... 4952 images, 48 backgrounds, 0 corrupt: 100% 5000/5000 [00:00<?, ?it/s]
               Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:14<00:00,  2.11it/s]
                 all       5000      36335      0.541      0.316       0.34      0.196
Speed: 0.2ms pre-process, 2.6ms inference, 2.5ms NMS per image at shape (32, 3, 640, 640)
```

此时 mAP 为 0.198，相比基础平均精度提升了较多，但比 10%-30%剪枝效果差的还是比较多，mAP 图如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.198
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.344
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.203
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.117
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.236
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.252
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.204
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.355
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.402
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.235
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.440
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.524
Results saved to runs/val/exp4
```

f．对于 50%剪枝模型

同样修改 val.py 文件，把传入 prune 参数改为 0.4 即可；运行命令后可以得到以下结果，一张图片 1.9nms，总共耗时 2.7ms：

```
val: Scanning /content/datasets/coco/val2017.cache... 4952 images, 48 backgrounds, 0 corrupt: 100% 5000/5000 [00:00<?, ?it/s]
               Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 157/157 [01:05<00:00,  2.39it/s]
                 all       5000      36335      0.271     0.0821     0.0576     0.0261
Speed: 0.2ms pre-process, 2.7ms inference, 1.9ms NMS per image at shape (32, 3, 640, 640)
```

此时 mAP 为 0.02，此时已经和基础模型差不多了，说明此时剪枝效果已经没起到作用，继续剪枝下去可能会适得其反，mAP 图如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.020
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.044
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.016
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.017
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.038
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.017
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.043
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.084
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.097
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.036
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.096
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.127
Results saved to runs/val/exp3
```

g．性能比较

由前面结果可知，在到 10%-50%（10%为间隔）的剪枝中，10%-30%效果比较好，接下来以一张图片的检测结果进行分析比对：

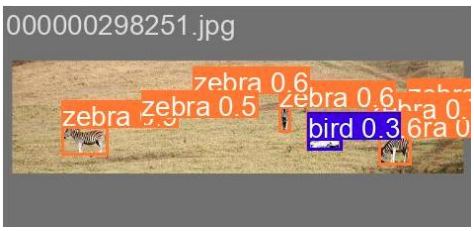


图 1. 原模型

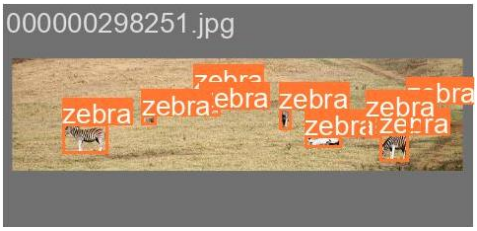


图 2.真实标签

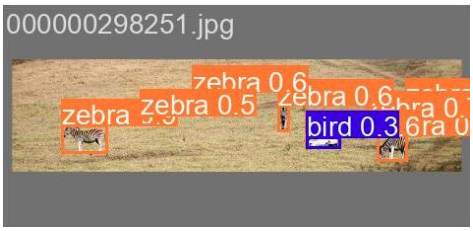


图 3. 10%剪枝

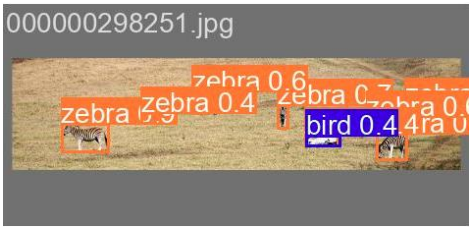


图 4. 20%剪枝

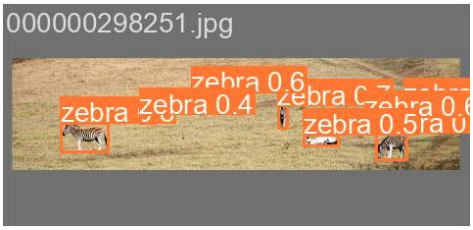


图 5. 30%剪枝

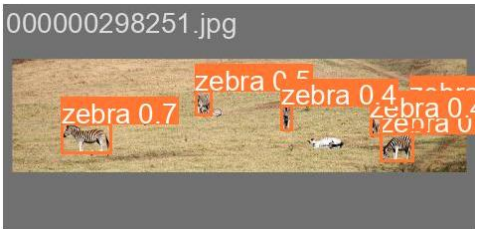


图 6. 40%剪枝

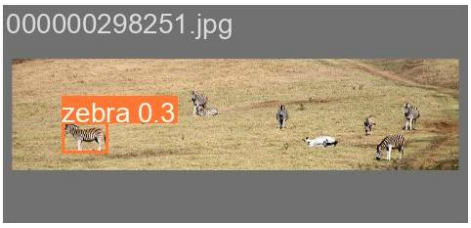


图 7. 50%剪枝

观察实验结果可知，原本模型对于中间的一只趴下的斑马识别错误成了鸟类，在 10%和 20%剪枝下仍是错误分类，但在 30%剪枝下成功分类，说明合理剪枝可以提升模型的速度和性能，像 50%剪枝下模型效果变差了很多，一定程度上说明合理剪枝的重要性。

实验结论：

本次实验进行了 Yolov5s 模型的加速和优化，具体从模型量化和剪枝出发，成功进行了不同程度的量化以及不同规模的剪枝，并成功进行了性能比较，验证了在选择正确的量化方式和合理的剪枝规模下对于模型的速度和性能的提升还是较大的。本次实验圆满结束。

心得体会：

本次实验初步接触了模型的加速和量化，花了较多的时间去阅读文档和搜集相关的资料，还得是借用 colab 上限量免费的 GPU 才能较快的跑出结果。通过本次实验，我真正体会到模型量化和剪枝对模型带来的功效，不管是利还是弊也好，至少从理论迈入实践了，收获还是蛮大的，继续努力。

指导教师批阅意见：   成绩评定：   指导教师签字：王旭 2024 年 11 月 28 日
备注：