110612117 張仲瑜 HW2 report

I. Introduction

1. Goal

In this task, we aim to achieve as high mAP and accuracy as possible in a digit detection competition with Faster-RCNN based model. There are two tasks in the competition, including digit bounding box prediction and digits recognition in images prediction.

2. Environment

- ✧ Laptop: Windows / RTX4060 * 1
- ✧ Kaggle account * 3: Linux / Tesla T4 * 2

3. Core Idea

I built a fast RCNN pipeline [1,2] with custom components, including backbone, anchor generator [3] and ROI_pooler, which allows me to select a pretrained ResNet101d as the backbone model while designing the FPN (Anchor generater as I want to conduct experiments.

II. Method

1. Data augmentation set

- ✧ Resize
  After observing the data that are confusing to the model, which are either small or low resolution, I perform the upsampling with bicubic interpolation to the data and adjust the label simultaneously to increase the mAP [4] in task one.
- ✧ Contrast Enhancement, Sharpening
  Some of the data are difficult since lights are dim in those images, I perform the upsampling with bicubic interpolation

to increase the performance of the pipeline.

2. Model Architecture

✧ Backbone Model: ResNet101 (Pretrained on imageNet1k)

Output size	ResNet-101	
144 × 144	conv, 7 × 7, 64, stride 2	
72 * 72	max pool, 3 × 3, stride 2	
72 × 72	conv, 1 × 1, 256 conv, 3 × 3, 256 conv, 1 × 1, 256	x3
36 x 36	conv, 1 × 1, 512 conv, 3 × 3, 512 conv, 1 × 1, 512	x4
18x18	conv, 1 × 1, 1024 conv, 3 × 3, 1024 conv, 1 × 1, 1024	x23
9x9	conv, 1 × 1, 2048 conv, 3 × 3, 2048 conv, 1 × 1, 2048	x3
1x1	global average pooling, 2048-d fc, softmax	

✧ FPN: I use all four 4 layers of the model

```python
def forward(self, x):
    x = self.stem(x)
    c2 = self.layer1(x)
    c3 = self.layer2(c2)
    c4 = self.layer3(c3)
    c5 = self.layer4(c4)
    return {
        '0': c2,
        '1': c3,
        '2': c4,
        '3': c5
    }
```

- Anchor size map and aspect ratio:
  Ascending pixels size as layers go deeper, with the common digit aspect ratio.

```python
anchor_generator = torchvision.models.detection.rpn.AnchorGenerator(
    sizes = ((4, 8, 16), (8, 16, 32), (16, 32, 64), (32, 64, 128), (64, 128, 256)),
    # sizes = ((4, 12, 20), (28, 36, 44), (52, 60, 68), (68, 72, 80), (100, 128, 256)),
    aspect_ratios = ((0.5, 1.0, 2.0),) * 5,
)
```

- ROI pooler:
  4 feature maps with default settings.

```python
roi_pooler = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0', '1', '2', '3'],
    output_size=7,
    sampling_ratio=2
)
```

3. Hyperparameters (best one)
   - Pretrained on imageNet1k
   - Training/ testing box NMS threshold [5]: 0.5
   - Training/ testing box score threshold: 0.65
   - Anchor map: As image above
   - Aspect ratios: As image above
   - Sharpening factor = 1.5
   - Learning rate: 5e-5 and * 0.5 decay factor whenever both validation accuracy and loss didn’t decrease in 2 consecutive epochs.
   - Loss: Cross Entropy (classification) + Smooth L1(Regression)
   - Optimizer: AdamW
   - Batch size: 4
   - Epoch: Max is 50 but with early-stopping patience = 7 epochs, stopping at epoch 28.
   - Weight decay = 1e-3

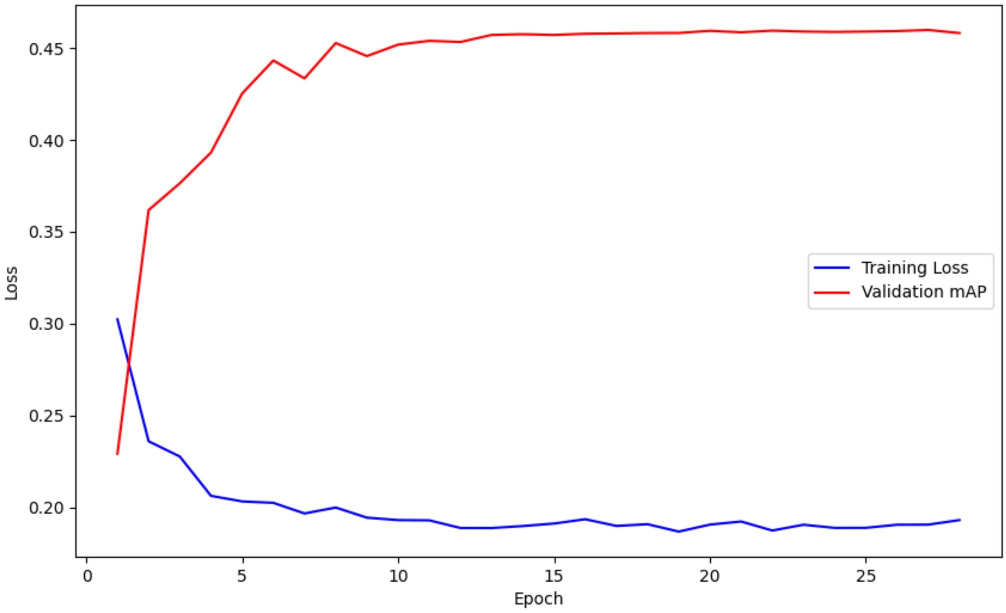
III. Result:

1. Best performance

- ✧ Task 1 mAP: 0.3767 on private testing
- ✧ Task 2 Accuracy: 0.8477 on private testing
- ✧ Other COCO indicators

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.456
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.907
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.384
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.446
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.497
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.597
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.504
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.537
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.537
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.527
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.572
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.604
```

- ✧ Learning curve (epoch vs Training loss & Val mAP)



IV. Additional experiment [6]:

1. Hypothesis: Adjust the custom anchor size stride in each feature map to 4, that is ((4, 12, 20), (28, 36, 44), (52, 60, 68), (68, 72, 80), (100, 128, 256)), instead of double one ((4, 8 ,16)...) will increase the performance on small targets.

    ✧ I think this may work because most of our digit sizes lie in a specific dense range.

    ✧ Result

Anchor stride	AP@IoU=0.50:0.95 area= small maxDets=100
Stride = +4	0.446
Stride = x2	0.443

    ✧ Implication
    There's only a slight difference between them; I think that's because in latter layers, the receptive fields are too large that the influence of particular small area is little, that is, the boost only comes from the stride change in first two layers.

2. Hypothesis: The higher the scale factor in supe resolution is, the better the performance is.
   - ✧ I think this may work after observing images that are difficult to recognize, and most of them are small.

   - ✧ Result

Scale Factor	AP@IoU=0.50:0.95 area= small maxDets=100
4	0.446
2	0.438

   - ✧ Implication
     It seems that under the same settings, larger scale factors did help the model learn slightly better in small areas; however, it also increased lots of computational cost. Personally, I think it's not worth doing it.

3. Hypothesis: The higher the box NMS threshold is, the more digits we will get in the prediction, which means that we can observe the wrong data type (missing/repeat digits) to adjust the threshold.
   - ✧ I think this may work since NMS will remove boxes that have high IoU with the one which has the highest confident score. Sometimes numbers like 117 or 112 will be recognized as 17 ,1117,12 and 1112 depending on this threshold
   - ✧ Result

Box_NMS_threshold	result
0.7	7117/1112
0.5	117/112
0.3	117/112

   - ✧ Implication:
     It did work and I successfully boost the performance through adjusting the threshold to train a better model.

V. Github

https://github.com/sharkccy/NYCU_CV_2025_Spring/tree/main/hw2

VI. Thoughts

It's my first time running object detection models locally, which makes me learn more about how to build a Faster RCNN pipeline given CNN model. The process of observing and barnstorming is interesting as usual, and I think data processing is still the key in this kind of task. Having great datasets is far more effective than other training strategies.

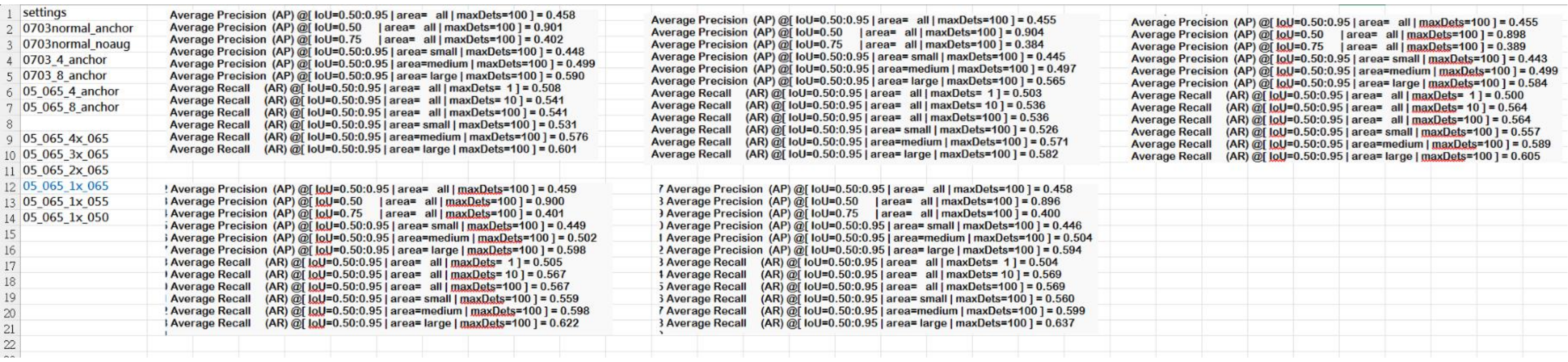


Fig. Some of my experiments

VII. Reference:

1. https://ivan-eng-murmur.medium.com/object-detection-s3-faster-rcnn-%E7%B0%A1%E4%BB%8B-5f37b13ccdd2
2. https://didi3310781.medium.com/faster-r-cnn-%E5%AD%B8%E7%BF%92%E7%AD%86%E8%A8%98-4917d59edb55
3. https://github.com/pytorch/vision/issues/3246
4. https://reurl.cc/Dq48ee
5. https://reurl.cc/0K9yn9
6. https://pytorch.org/vision/main/_modules/torchvision/models/detection/faster_rcnn.html#fasterrcnn_resnet50_fpn_v2