DLBAIPCV01 – PROJECT: COMPUTER VISION

STUDENT: HUGO ALBUQUERQUE COSME DA SILVA

ID:92126125

TABLE OF CONTENTS

1 – Introduction

Object detection in the context of computer vision refers to a task where computers are prompted to identifying objects in images or videos. It is a very active research area, because its applications add a lot of value in different industry sectors. Identifying people in security cameras, identifying pedestrians, cars, and other vehicles in self driving cars are prominent applications of object detection.

The goal of this project is to implement three SOTA(State Of The Art) approaches that perform object detection videos. In order to do this, open source libraries and pre-trained models made available by researchers will be used. An evaluation of the strengths and weaknesses of each approach will be performed. Benchmark results will be presented, and to simulate my own benchmark, I will run the same video sequence in the three models and compare the results.

2 - Methodology

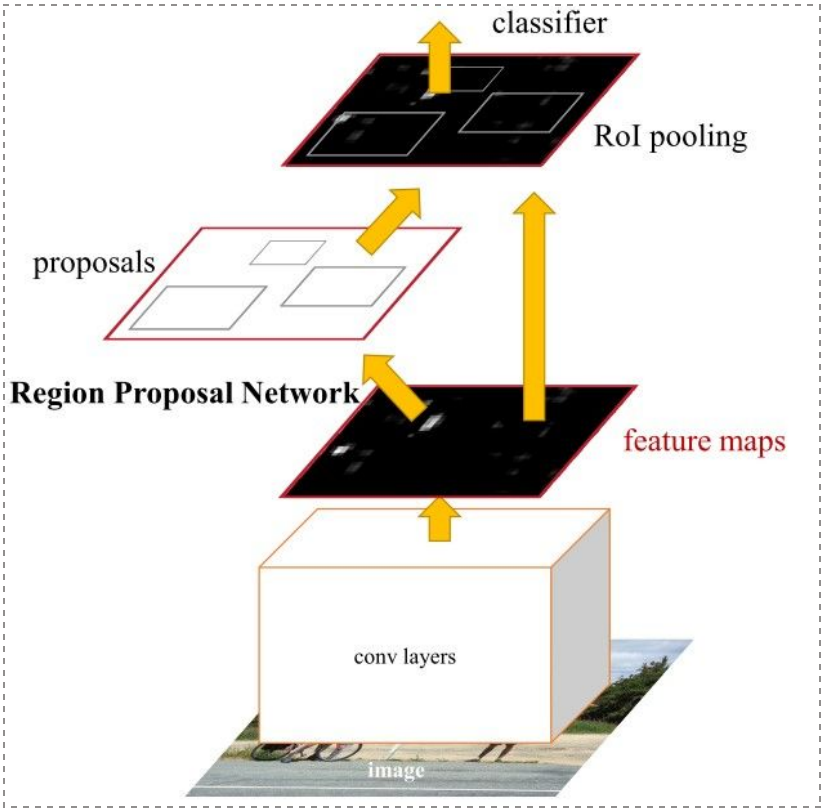
2.1 - Models to be analyzed

The three models chosen to be analyzed are: FASTER RCNN [1], Retina Net [2] and YOLO_v8 [3]. Links to the model's original papers can be found in the reference section, but I will provide a brief overview of the architecture of each model down below. In Computer Vision , the first part of the model that receives the input images and generate a feature map is called a "backbone". In this study, I used the same backbone for Faster RCNN and Retina net: A pre-trained imagenet model(ResNet 50) with a Feature Pyramid Network(FPN), which is a feature extractor that takes a single-scale image of an arbitrary size as input, and outputs proportionally sized feature maps at multiple levels, in a fully convolutional fashion[4]. YOLO uses a CSPDarknet53 as the backbone of its architecture.

2.1.1 - Faster R-CNN

This model was developed by researchers at Microsoft, and it is based on Convolutional Neural Networks. It introduced a Region Proposal Network(RPN) that generate region proposals that are then fed to the detention model. Each extracted feature vector is used to assign a region to either the background or a specific object class. The architecture has three main components: the Region Proposal generator, the feature extraction, and classification module:

Figure 1 - Fast rcnn High-level overview



2.1.2 - RETINA NET

Retina Net is a one stage object detection model, whose main innovation was the use of a technique called "Focal loss", a dynamically scaled cross entropy loss function that allow the network to focus on classifying harder examples correctly. Besides the backbone, it has two specific subnetworks: one for classifying the objects in the bounding-boxes, and other for estimating its coordinates(regression task). Here is a high level overview:

Figure 2 - RetinaNet high level overview

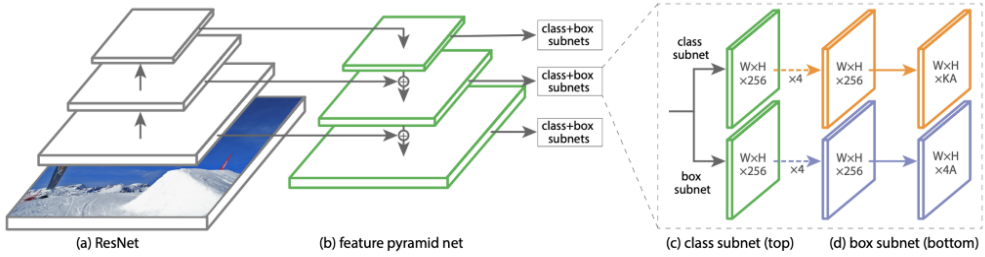


Figure 3. The one-stage **RetinaNet** network architecture uses a Feature Pyramid Network (FPN) [20] backbone on top of a feedforward ResNet architecture [16] (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The network design is intentionally simple, which enables this work to focus on a novel focal loss function that eliminates the accuracy gap between our one-stage detector and state-of-the-art two-stage detectors like Faster R-CNN with FPN [20] while running at faster speeds.

2.1.3 - YOLO

YOLO stands for "You Only Look Once" and, as the name suggests, it is a single shot detector. But while other algorightms like Faster RCNN and RetinaNet use auxiliary subnetworks to help identifying regions of interest, YOLO perform all of its predictions with the help of a single fully connected layer. It divides the image into an SxS grid. Each grid cell will pass through the network, and it will output a vector that contains the probability that the cell contains an object, the object coordinates of the center, the predicted width and height of the object, and the probabilities that the object belongs to a specific class. After this, a IOU(Intersection over Union) filter is applied to help distinguishing different objects from the same class(let's say a picture with two dogs). At last, a technique called Non Maximal Supression(NMS) is applied to select the final bounding box. Below is high level overview of the workflow:

Figure 4 - YOLO workflow

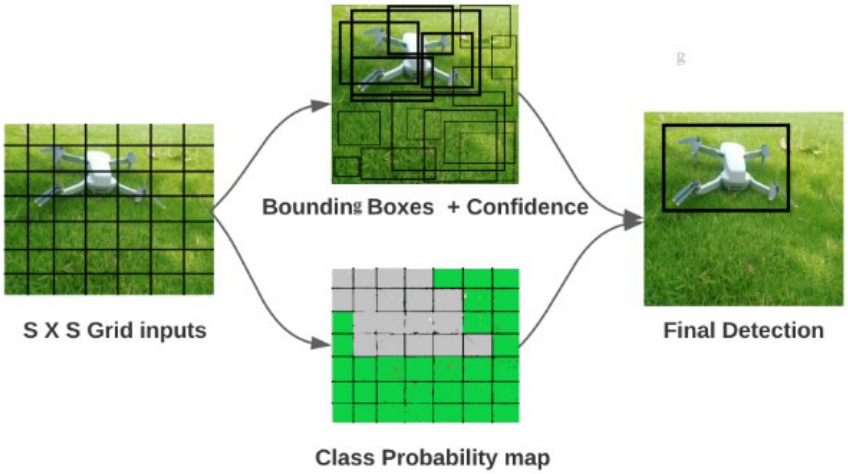


Figure 5 - YOLO architecture

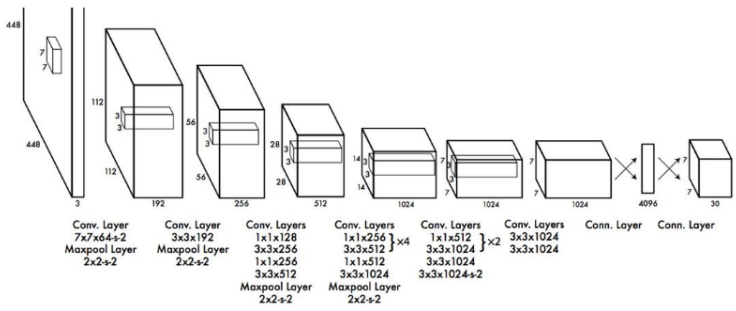


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

2.2 - Libraries and platform used

To perform the study, I have used google colab pro and its jupyter notebooks. I have used a runtime environment with a Nvidia V100 GPU, to speed up the predictions and get experiments results faster. To avoid spending a long time training these models from scratch, I have used pre-trained versions made available in open source libraries.

Detectron2 is an open source library published by Facebook AI Research team. It contains a "model zoo" for several common computer vision tasks, like object detection, image segmentation, facial keypoint detection, etc. I have used its python API to implement both Fast RCNN and RetinaNet.

Ultralytics is python library made available by the the company that holds the same name. This library include different implementations of the YOLO algorithms and a simple an intuitive API. For the analysis I will use Yolov8m, a medium size version of the model with 25.9 M parameters.

What is interesting here for benchmarking purposes is that all above mentioned models, were trained using COCO(Common Objects in Context) 2017 train dataset.

2.3 - Experimental Procedure

To compare the results in a fair manner, I have selected a 30fps video from another benchmark dataset, MOT16. The video contains several people walking in the street, besides cars, bicycles and other objects[5]. after downloading and uploading the video to COLAB's environment, the procedure was mainly:

1. Instanciate the object detection models

2. perform the predictions
3. Convert the output frames with bounding boxes to video format with the help of openCV functions when necessary.

The detailed step by step workflow can be found in the project's notebook[6].

2.4 - Results discussions

2.4.1 - Official Benchmarks

let's start by presenting the official benchmarks of the models in the COCO dataset:

Figure 6 - Faster R-CNN Detectron 2 official benchmark [7]

Faster R-CNN:

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	model id	download
R50-C4	1x	0.551	0.102	4.8	35.7	137257644	model | metrics
R50-DC5	1x	0.380	0.068	5.0	37.3	137847829	model | metrics
R50-FPN	1x	0.210	0.038	3.0	37.9	137257794	model | metrics
R50-C4	3x	0.543	0.104	4.8	38.4	137849393	model | metrics
R50-DC5	3x	0.378	0.070	5.0	39.0	137849425	model | metrics
R50-FPN	3x	0.209	0.038	3.0	40.2	137849458	model | metrics
R101-C4	3x	0.619	0.139	5.9	41.1	138204752	model | metrics
R101-DC5	3x	0.452	0.086	6.1	40.6	138204841	model | metrics
R101-FPN	3x	0.286	0.051	4.1	42.0	137851257	model | metrics
X101-FPN	3x	0.638	0.098	6.7	43.0	139173657	model | metrics

Figure 7 - RetinaNet Detectron 2 official benchmark[7]

RetinaNet:

Name	lr sched	train time (s/iter)	inference time (s/im)	train mem (GB)	box AP	model id	download
R50	1x	0.205	0.041	4.1	37.4	190397773	model | metrics
R50	3x	0.205	0.041	4.1	38.7	190397829	model | metrics
R101	3x	0.291	0.054	5.2	40.4	190397697	model | metrics

Figure 8 - YOLOv8 ultralytics benchmark [8]

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

- **mAP^{val}** values are for single-model single-scale on COCO val2017 dataset.
  Reproduce by `yolo val detect data=coco.yaml device=0`
- **Speed** averaged over COCO val images using an Amazon EC2 P4d instance.
  Reproduce by `yolo val detect data=coco128.yaml batch=1 device=0|cpu`

Looking at the Mean Average Precision of these models , we see that YOLOv8 is the most accurate in predicting bounding box locations, followed by Faster RCNN in second and Retina Net in third.

Comparing inference speed, YOLOv8 again gets the first place with 0.00183 sec/img. This is much faster than the second place, Faster RCNN, that has a speed of 0.038 sec/img. RetinaNet comes in third with a speed of 0.041 s/img.

2.4.1 - Personal Benchmarks

The link to the predictions made in the same video by the three different models are below:

- Faster RCNN
- RetinaNet
- Yolov8m

Analyzing speed performance, Fast RCNN took almost 7 minutes to predict every frame in the video:

Figure 9 - Faster RCNN inference speed

Performing inference...

```
[ ]  1 results = []
     2 for imagePath in tqdm(frames):
     3     pred = detector.onImage(imagePath, return_prediction=True)
     4     results.append(pred)
```

100% 1050/1050 [06:46<00:00, 2.64it/s]

RetinaNet took 11 minutes:

Figure 10 - RetinaNet inference speed

```
1 results = []
2 for imagePath in tqdm(frames):
3     pred = detector.onImage(imagePath, return_prediction=True)
4     results.append(pred)
```

```
100% 1050/1050 [11:02<00:00, 1.63it/s]
```

Yolo took 1 minute and 40 seconds:

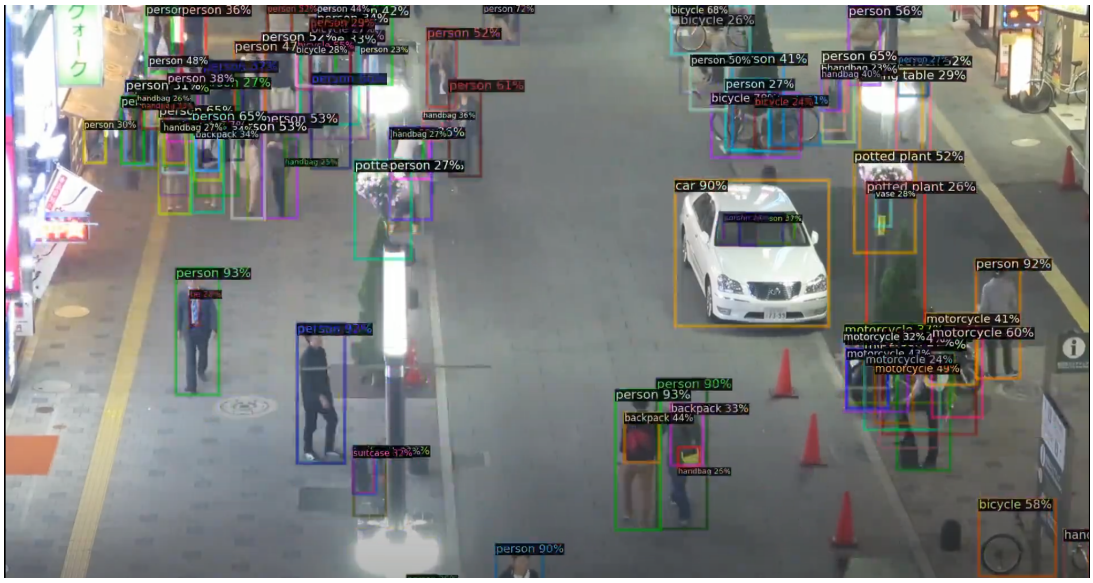
Figure 11 - Yolo inference speed

```
Speed: 2.3ms preprocess, 8.1ms inference, 1.5ms postprocess per image at shape (1, 3, 384, 640)
Results saved to runs/detect/predict
CPU times: user 1min 34s, sys: 5.31 s, total: 1min 39s
Wall time: 1min 40s
```

Analyzing the output videos we can check the following:

- Yolo and FasterRCNN do an excelent job at classifying objects. They practically don´t label anything incorrectly. RetinaNet is the only one that sometimes label things that are not in the image, like in this frame where it detected people in the seat of the car, but the car was empty:

Figure 10 - Frame at 24 seconds in RetinaNet video output



- Faster RCNN is the model that makes predictions with the highest confidence. In contrast Retina net is the model capable of identifying the biggest variety of objects in the video, but the confidence of the detections is not high, and there is a considerable amount of false positives. For example, at 15 seconds, the model misclassifies a manhole by a frisbee, and some coverage in the floor by a suitcase.

Figure 11 - Misclassifications in Retina net

- Faster RCNN suffers from not making detections consistently across time. In the below example, we see part of the frame at 4s and the same part again at 22s. The person is not classified on both occasions.

Figure 12 - Detections of some objects are not consistent across time in faster RCNN

- Yolo is the model that shows the most consistent performance across the frames. Nonetheless for smaller objects like backpacks, handbags and ties, the model is only capable of detecting it when they are close to the camera.

3 - Conclusion

This project examined the performance of three different object detection models, namely, Faster-RCNN, RetinaNet anf YoloV8. All models were trained on COCO dataset and made available in python open source libraries. Besides showcasing their official benchmark, a personal benchmark analysis was conducted, where the models were used to detect objects in the same video sequence.

As showcased in this document and the work done in the project's notebook, YOLO wins the battle, since it is by far the fastest model, its detections are consistent across the entire video, and it classifies the objects with correct labels.

4 - Bibliography

1. Ren, S., He, K., Girshick, R., & Sun, J. (2016, January 6). Faster R-CNN: Towards real-time object detection with region proposal networks. arXiv.org. https://doi.org/10.48550/arXiv.1506.01497

2. Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2018, February 7). Focal loss for dense object detection. arXiv.org. https://doi.org/10.48550/arXiv.1708.02002

3. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016c, May 9). You only look once: Unified, real-time object detection. arXiv.org. https://doi.org/10.48550/arXiv.1506.02640

4. Papers with code - FPN explained. FPN Explained | Papers With Code. (n.d.). https://paperswithcode.com/method/fpn#:~:text=A%20Feature%20Pyramid%20Network%2C%20or,in%20a%20fully%20convolutional%20fashion.

5. MOT16-04. MOT Challenge - Visualize. (n.d.). https://motchallenge.net/vis/MOT16-04

6. benchmarking_video_object_detection_models. GitHub. (n.d.-a). https://github.com/hualcosa/IU_DLBAIPCV01/blob/main/benchmark_video_object_detection_models.ipynb

7. Detectron2 Model Zoo and baselines. GitHub. (n.d.). https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md

8. Ultralytics. (n.d.). Ultralytics/ultralytics: New - yolov8 🚀 in PyTorch > ONNX > OpenVINO > CoreML > TFLite. GitHub. https://github.com/ultralytics/ultralytics