Object Detection:
A comparison of YOLOv3 and Faster RCNN algorithm on the IndyCar Series dataset

Imtiaz Ahmad [1]

Abstract

The modern era is overwhelmed with the presence of devices that collects real time data. Hence, the necessity of Real Time data analysis is getting more importance than ever before. One of the exciting and open problem in this field is Real Time Object Detection. In this project, I use two well known deep learning models to detect cars in the dataset provided by a racing car event: IndyCar Series.

1. Introduction

One of the most fundamental and challenging problems in image processing and computer vision is **Object Detection** (Everingham et al., 2010; Lin et al., 2014). It is a technique used to identify the location of objects in an image. It is increasingly getting popularity in fields such as: intelligent vehicles, identity recognition, intelligent surveillance etc. According to (Wang et al., 2017), the goal of object detection is to learn a visual model for concepts such as cars and use this model to localize these concepts in an image. If there is a single object in the image and we want to detect that object, it is known as image localization. But in general, there are more than one object in an image and things get complicated when it comes to detecting objects in video data.

Recently the focus of the research community has shifted to Real Time Object Detection as it has so many possibilities. Fast and accurate algorithms for object detection like human visual systems would allow computers to drive cars in any weather without specialized sensors. This would also enable assistive devices to convey real-time scene information to human users. Due to the recent surge of breakthroughs in deep learning and computer vision, we can lean on object detection algorithms to not only detect objects in an image – but to do that with almost the speed and accuracy of humans.

An ideal real-time object detection model should be able to sense the environment, parse the scene and identify what types of objects are present in the scene. On top of that it should declare the position of the objects by defining a bounding box around each object. However, there are some pitfalls to implementing these models. Deploying object detection models on our local machines is a hard task as they generally take a lot of memory and computation power. Also, the input data comes in various shapes and formats. In summary, the challenge while using these deep learning models for real time object detection lies in keeping a balance between detection performance and real-time requirements. The normal trend is if the real-time requirements are met, there is generally a drop in performance and vice versa.

In this project the goal is to compare the performance of two deep learning models on a unique dataset that contains images of sports cars. The Indy500 is a racing event that occurs each year. It has huge data gathered from each race. It has attracted data scientist for analysing various aspects like rank prediction, anomaly detection etc. One thing that is still little explored is object detection in live video feed. Although there are general car detection models that perform very well but the unique feature about this data set is it requires detection of sports cars which are slightly different from normal cars.

The deep learning models that are going to be used are: (1) YOLOv3(You Only Look Once) and (2) Faster RCNN(Region-based Convolutional Network). Each of the algorithms have their own pros and cons. The goal is to see which model performs better in terms of performance on this unique dataset of sports cars.

2. YOLO : You Only Look Once

The traditional two stage detection models are based on region proposal method which increases the time to make inference. To reduce this inference time came YOLO (Redmon et al., 2015) object detection model. This model detects objects by dividing an image into several grid units. The feature map of the output layer is designed to output bounding (bbox) coordinates, the objectness score, and the class scores. This is how multiple objects are detected with a single inference using YOLO. Therefore, we see a significant increase in the detection speed compared to the conven-

Figure 1. Indy 500 race.

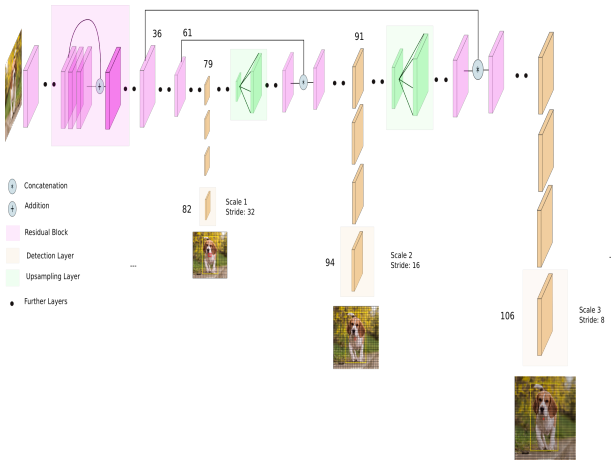

Figure 3. YOLO v3 network architecture.

tional detection models. However, the main drawbacks are

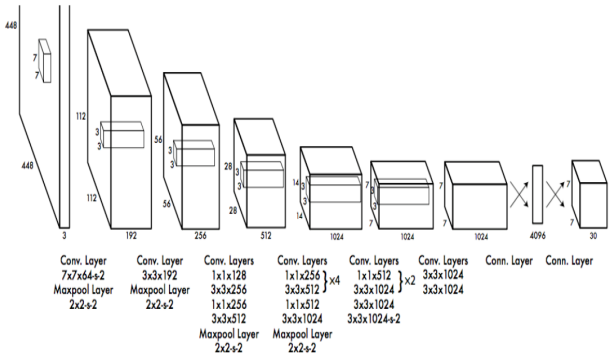

Figure 2. YOLO network architecture.

the existence of localization errors and the low accuracy when detecting objects. Localization errors happen because of processing of the grid unit. To address these problems, YOLOv2 (Redmon & Farhadi, 2016) was proposed. To improve detection accuracy, YOLOv2 uses batch normalization for the convolution layer. Along with fine-grained features and multi-scale training, it also applies anchor box technique to detect the regions. One downside is that it's detection accuracy is still low for small or dense objects.

The next iteration that has been proposed aiming at overcoming this disadvantage is YOLOv3 (Redmon & Farhadi, 2018)[1].

[1]image courtesy: medium.com

YOLOv3 consists of convolution layers and is constructed of a deep network for an improved accuracy. YOLOv3 applies a residual skip connection to solve the vanishing gradient problem of deep networks and uses an up-sampling and concatenation method that preserves fine-grained features for small object detection. It uses a variant of Darknet, which originally has 53 layer network trained on Imagenet. For the task of detection, the underlying architecture of YOLOv3 adds an additional 53 layers are onto it, giving a total 106 layers.

The most prominent feature is the detection at three different scales. In more detail, when an image of three channels of R, G, and B is input into the YOLOv3 network, as shown in Figure 3, information on the object detection (i.e., bbox coordinates, objectness score, and class scores) is output from three detection layers. Actually these detection layers make prediction at three scales by down-sampling the dimensions of the input image by 32, 16 and 8 respectively. The first detection is made by the 82nd layer. This layer has a stride of 32 i.e. an input image of 416 x 416, would produce a feature map of size 13 x 13. Use of the 1 x 1 detection kernel yields a detection feature map of 13 x 13 x 255.

Then, the feature map from layer 79 passes through a few convolutional layers before being up sampled to the dimension of 26 x 26. This feature map is then depth concatenated with the feature map from layer 61 which passes through a few 1 x 1 convolutional layers. Then, the second detection is made by the 94th layer, yielding a detection feature map of 26 x 26 x 255.

A similar procedure is followed again, upsampling the feature maps and passing through a few convolutional layers before being depth concatenated with a feature map from layer 36. We make the final detection at the 106th layer, yielding feature map of size 52 x 52 x 255.

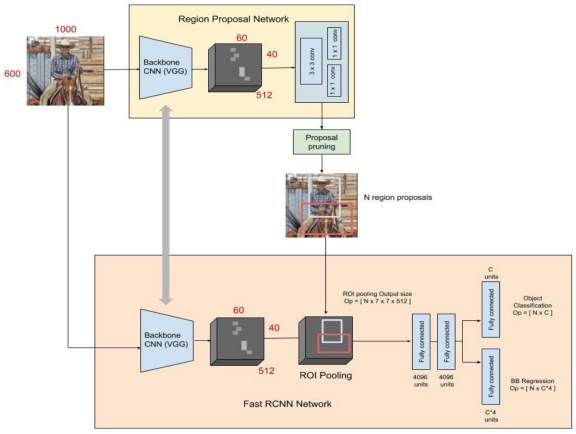

Figure 4. Faster R-CNN framework.[3]

For final classification, YOLOv3 uses logistic regression instead of softmax which was used by the earlier versions. Logistic regression is helpful for multilabel classification. It performs better than YOLOv2 by detecting smaller objects as it uses different detection layers for detecting different objects: the 13 x 13 layer is responsible for detecting large objects, the 26 x 26 layer detecting medium objects, and the 52 x 52 layer detects the smaller objects. In terms of the trade-off between accuracy and speed, YOLOv3 is suitable and widely used for real time object detection.

3. Faster RCNN

The second model which is going to be used is an iteration of a two stage detection model called RCNN : Region-based Convolutional Neural Networks. It's called Faster RCNN because as the name suggests, it's much faster than the previous versions of the RCNN model. In particular it trumps the earlier versions in terms of computational efficiency, reduction in test time, and improvement in performance (mAP).

The Faster R-CNN (Ren et al., 2015) is a single and unified network for object detection which consists of the following two modules:

- A deep and fully convolutional network that proposes regions (RPN)
- A Fast R-CNN detector that uses the proposed regions.

The region proposal network takes an image as its inputs, and a set of regions proposals along with their objectness scores as outputs. To generate these regions proposals, RPN slides a small network over the convolutional feature map output by the last convolutional layer. At each window, this small network predicts an objectness score and four regression coordinates scores. Sliding window is an essential method used by RPN to generate region proposal boxes, and RPN can realize end-to-end detection. RPN generates anchor boxes in the input image with three scales (1:1,1:2,2:1) and three resolutions (128,256,512), thus generating 9 anchor boxes for each position. These are then passed through two parallel fully connected layers: box classification layer (cls) and box regression layer (reg) to obtain the classification and location information. The structure of RPN is illustrated in Figure 5[4].

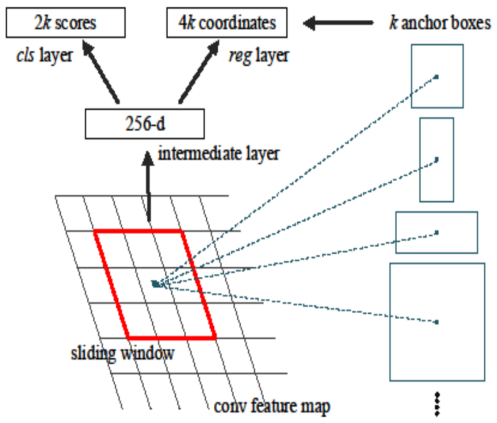

Figure 5. Region Proposal Network.

As the network moves through each pixel in the output feature map, it has to check whether these k corresponding anchors spanning the input image actually contain objects, and refine these anchors coordinates to give bounding boxes as Object proposals or regions of interest. First, a 3 x 3 convolution with 512 units is applied to the backbone feature map as shown in the top part of Figure 4, to give a 512-d feature map for every location. This is followed by two sibling layers: a 1 x 1 convolution layer with 18 units for object classification, and a 1 x 1 convolution with 36 units for bounding box regression. The 18 units in the classification branch give an output of size (H, W, 18). This output is used to give probabilities of whether or not each point in the backbone feature map (size: H x W) contains an object within all 9 of the anchors at that point. The 36 units in the regression branch give an output of size (H, W, 36). This output is used to give the 4 regression coefficients of each of the 9 anchors for every point in the backbone feature map (size: H x W). These regression coefficients are used to improve the coordinates of the anchors that contain objects.

The Fast R-CNN detector consists of a CNN backbone, an ROI pooling layer and fully connected layers followed by two sibling branches for classification and bounding box regression as shown in Figure 4. The input image is first

[4]image courtesy: github.com

passed through the backbone CNN to get the feature map (Feature size: 60, 40, 512). The bounding box proposals from the RPN are used to pool features from the backbone feature map. This is done by the ROI pooling layer which has a size of (N, 7, 7, 512) where N is the number of proposals from the region proposal algorithm. Next, these ROIs are sent to fully connected layers of size 4096. Finally, a class score layer gives the probability of the object belonging to each class and a bounding box regression layer gives the bounding box around the detected object in order to make the proposal boxes more precise.

To merge RPN and Fast R-CNN, the authors propose a 4-stage training method: The first stage trains a RPN initialized with weights from ImageNet models; The second stage trains a Fast R-CNN also initialized with weights from ImageNet models and feeding the Fast R-CNN with region proposals from RPN; The third stage uses Fast R-CNN weights to initialize a RPN but only fine-tune the layers unique to RPN; The fourth stage fine-tunes Fast R-CNN unique layers.

4. Dataset

The IndyCar series is the premier level of open-wheel car racing in the US. With more than 500 sensors, large amount of data, including timing and scoring log data are collected.

The images were generated from different cameras on vehicles in the racing tracks, with some of the them with no cars at all. There are 2,910 training images and 187 test images. Both the train and test images had the dimension of 1280 x 720. The annotation files were of XML format. We were supplied with this data-set by our lab instructor. The data-set format was PASCAL VOC2007. The main folder had two sub-folders named as test and train; which contained the test and train images. The folders also contained the annotation files. Each of the test and train folders had three folders: Annotations, JPEGImages, Imagesets. As the name suggest, the annotation files were kept in the Annotation folder, the test and train images were of .JPG format and kept in the JPEGImages folder.

5. Hardware

The experiment for both the models were run in two different hardware settings to compare performance. First they were run on a CPU in futuregrid server. The CPU was a 64 bit system with Red Hat Linux as the operating system which had a total memory of 262GB. It has three banks of DD4 RAM each of size 16GB and having a speed of 2666MHz. After that they were run using a GPU in futuresystems server. This CPU too was a 64 bit system with Red Hat Linux as the operating system which had a total memory of 65GB. The GPU had a memory of 4705MB and a computation capability of 3.7.

6. Results

6.1. YOLOv3 on CPU

First, the code was run on a single node CPU on futuregrid server. The training involved a total number of 53 epoches, with 3 warm-up epoches. An early stop approach was followed if the loss function doesnot change significantly. The training ran for more than 24 hours. The loss was stable after epoch 37; it did not increase from 9.79472. So, the training had an early stop after epoch 37. The **mAP**(Average Mean Precision) was 0.9740. Below the Loss and Learning Rate is displayed:

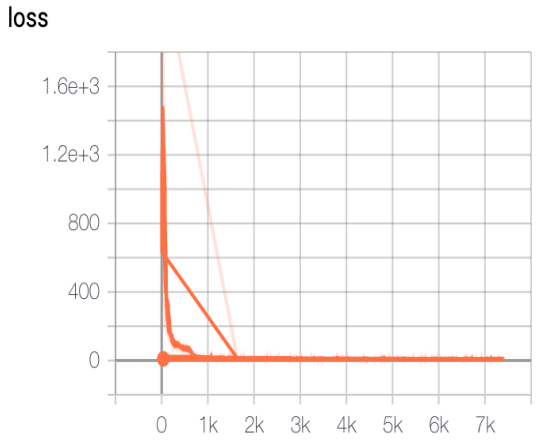

Figure 6. Loss function.

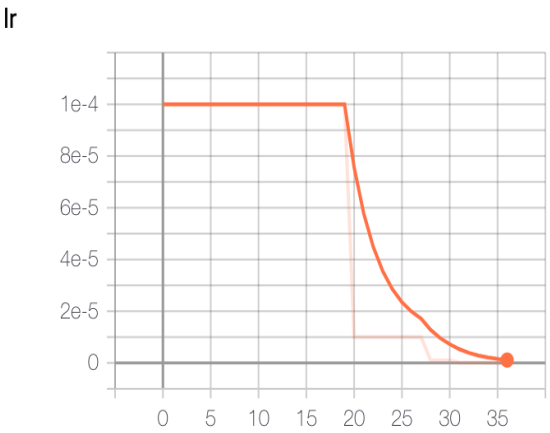

Figure 7. Learning rate.

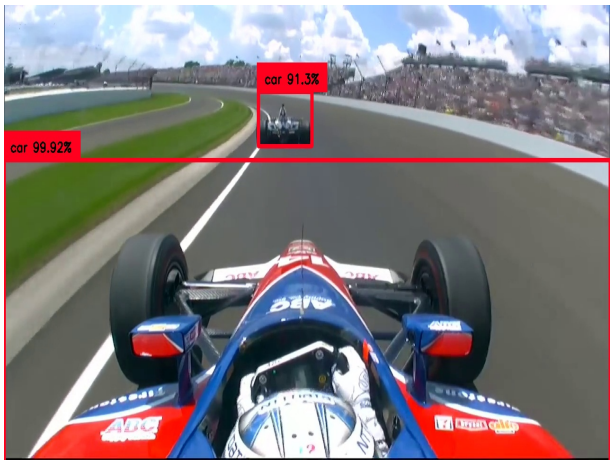

Figure 8. Example inference using YOLOv3.

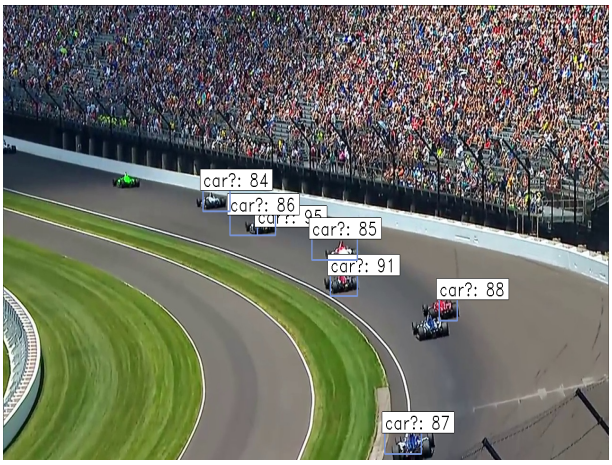

Figure 9. Example inference using faster rcnn.

The test was done on a set of 187 test images. It took on an average 6.5 seconds to infer each image.

6.2. YOLOv3 on GPU

The same code was then run on a GPU enabled server. As expected, the training was faster; taking only 9 hours. The mAP was 0.965. However, there was a significance improvement in the inference time. On an average, it took only 0.27 seconds to infer each image.

6.3. Faster-RCNN on CPU

The faster RCNN code was first run on a node on CPU futuregrid server. Instead of training the whole model at the same time, training was done in two steps. In the first step, the regional proposal network (rpn) model was trained. This step was done fairly quickly; it took 9 hours to finish training the rpn model and the weights were saved. The number of epochs was 50 and each epoch length was 1000. For rpn, the number of region of interest (ROI) was chosen to be 10.

Next, the faster-rcnn (frcnn) model was trained. Unlike yolov3, no early stopping was used in this training. Out of the available options, resnet50 was used for both the rpn and frcnn model as the back-end network.

The total number of epochs was 100 with each epoch having a length of 1000. At the beginning the individual epochs were pretty fast but after 70 epochs as the learning rate significantly reduced, each epoch took much longer to finish. The number of region of interest (ROI) was chosen to be 32. The total training time was 9 days. The **mAP**(Average Mean Precision) was 0.272. Single image inference took about 9.43 seconds.

6.4. Faster-RCNN on GPU

The GPU version basically had the same code. The training was again done in two steps. This time the training of RPN was quite quick as expected. It took only 4 hours to train the RPN model. The major difference was instead of 10 ROIs, this time the training was done with 32 ROIs.

For the actual RCNN model training, 100 epochs each with a length of 700 were used. The ROIs were same as before; with 32 ROIs. Unlike the training with CPU, this training took only 30 hours to finish. So, there is a huge gain in time requirements in the GPU version.

The inference time on an average was 1.04 seconds for each test image.

7. Discussion

Empirically, it is expected that YOLOv3 should have a higher speed in inference compared to FRCNN because YOLO is a single step model while FRCNN is a two step model. In this experiment this trend is followed. As described in the table 1, YOLOv3 takes less inference time both in the case of using a CPU and using a GPU compared to FRCNN.

	YOLOv3(secs)	FRCNN(secs)
CPU	6.5	9.43
GPU	0.27	1.04

Table 1. Comparison of inference time.

Also, the training was much faster when done using the GPU compared to CPU. As shown below in table 2.

The accuracy of YOLOv3 was a bit lower compared to FRCNN. It can be seen from the two following examples

	YOLOv3	FRCNN
CPU	24hrs	9days
GPU	9hrs	30hrs

Table 2. Comparison of training time.

where Yolo puts bounding boxes over a flower vase and persons and identifies them as cars while running FRCNN in these same frames did not show anything like this.

Figure 10. YOLOv3 inferring a car.

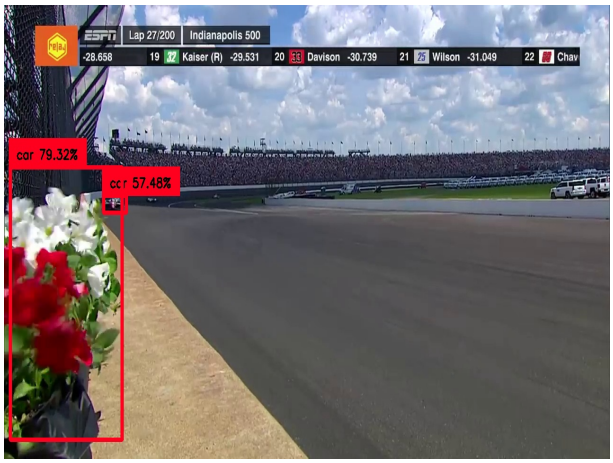

Figure 11. FRCNN not inferring a car.

However, there are cases where YOLOv3 could successfully detect a car while FRCNN could not. For example, the following two images shows an example of this scenario.

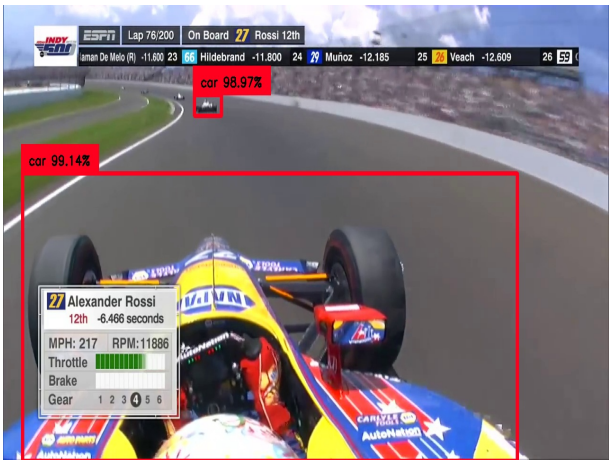

Figure 12. YOLOv3 inferring a car.

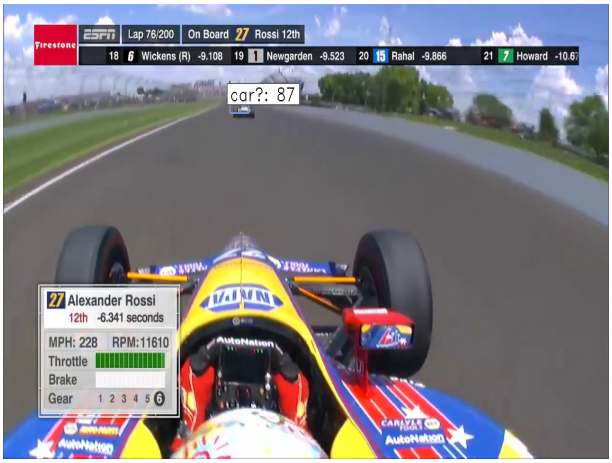

Figure 13. FRCNN not inferring a car.

This outcome could be linked to the performance of training of the RPN model. If the car is comparatively closer to the camera, FRCNN at times fails to identify the car. The reason might be due to the number of ROIs, which was 32 but it would be interesting to see how the model performs if the number of ROIs is increased. Another reason could be the limited number of training images. In future works, it would be worth training the models with a handsome number of images and then see how they perform.

8. Conclusions

Both YOLOv3 and Faster-RCNN has been heavily used for object detection in real life scenarios. However, in this project these models were used to detect a special kind of car: racing cars. As these cars are structurally different from traditional cars, it was a hard challenge to train the models. Especially, with the few number of train data available,

it was a challenging task to get the best result out of the models. Using CPU and GPU, the performance of these two models on the IndyCar dataset was measured. The outcome followed the expected pattern. The FRCNN has high accuracy compared to YOLOv3 but the YOLOv3 has higher speed in inference compared to FRCNN. The trade-off between speed and performance has been prominent since the very beginning of using deep learning models in real life problems. I hope this project gives a direction for future researches in detection of sports cars and which models to use in different contexts.

Acknowledgements

This project is done as a class project for the course CSE B649: High Performance Computing. I am really grateful to Prof and Associate Instructor Selahttin Akkas for the continuous help with the understanding of several concepts related to deep learning and the implementations of different deep learning models.

References

Everingham, M., Gool, L., Williams, C. K., Winn, J., and Zisserman, A. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010. ISSN 0920-5691. doi: 10.1007/s11263-009-0275-4. URL http://dx.doi.org/10.1007/s11263-009-0275-4.

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL http://arxiv.org/abs/1405.0312.

Redmon, J. and Farhadi, A. Yolo9000: Better, faster, stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2016.

Redmon, J. and Farhadi, A. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL http://arxiv.org/abs/1804.02767.

Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. URL http://arxiv.org/abs/1506.02640.

Ren, S., He, K., Girshick, R. B., and Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL http://arxiv.org/abs/1506.01497.

Wang, X., Shrivastava, A., and Gupta, A. A-fast-rcnn: Hard positive generation via adversary for object detection. *CoRR*, abs/1704.03414, 2017. URL http://arxiv.org/abs/1704.03414.