YOLOv5 & Learning with Different Backbones:

Trade-offs & scaling relationships in object detection

Yuxiang Chai, Dan Zhao

Abstract

Object detection is a key area of focus in computer vision and has seen significant work over numerous iterations in the search for improved performance, speed, etc., especially over the past few years. Generally speaking, there are two mainstream model architectures: two-stage models (including RCNN(Girshick et al. 2014), Fast and Faster RCNN(Ren et al. 2016) and its variants) and one-stage models (including SSD(Wei Liu et al. 2016), its variants, and the YOLO series).

While there exist comparative benchmarks documenting the relative performance across different object detection models in the past, YOLOv5, having debuted recently, has seen less attention—especially with respect to what effects different backbones, and therefore feature representations, have on YOLOv5's object detection performance, efficiency, and the tradeoffs between the two. We run exhaustive experiments using popular neural networks as backbones for YOLOv5—about one from every year from 2016-2020—to document and evaluate the trade-offs between empirical performance (mAP, recall, etc) and model efficiency/complexity. Additionally, we use these trade-offs to fit a simple model that captures the relationship of this performance/efficiency trade-off among YOLOv5-based models in the hopes of illustrating how performance scales with efficiency in YOLOv5 across different backbone architectures. This model's predictions are used to estimate an efficient frontier that shows how the newest YOLOv5 model/backbones (2020/21) achieve gains in both performance and efficiency that are un-achievable with other neural network backbones.

Executive Summary

Our main focus is to investigate the relationship and potential trade-offs between the efficiency, complexity, and performance in object-detection with a specific focus on a very popular framework used in object-detection tasks: YOLO ("You Only Look Once") and its most recent (2021), SOTA iteration YOLOv5.

With YOLOv5, given its relative novelty, we systematically document its complexity, efficiency, and performance across several feature-extractors/backbones by capturing the number of parameters, FLOPs, and other metrics across all backbone combinations. We then document the comparative trade-offs in terms of performance vs. efficiency across various popular network architecture backbones for YOLOv5.

We experiment across these backbones using the PASCAL VOC dataset—a common benchmark standard for object detection–and evaluate against several metrics that capture a model's performance, efficiency, and complexity.

Finally, we derive an empirical scaling relationship that can be described by a power-law, which illustrates and predicts how different YOLO models (new and old) might scale in their trade-offs between performance & efficiency/complexity. We derive this relationship/frontier via a log-log regression model to capture the power-law exponent that governs the trade-offs between increased model performance and increased model complexity/inefficiency.

Problem & Motivation

Beginning from LeNet, researchers have developed countless model architectures, evaluative benchmarks, datasets, modules, and training procedures for computer vision, especially for image classification. Compared to another area of computer vision, widely used, well-performing models for object detection are relatively limited, for example, to Faster RCNN and its variants as well as YOLO. A natural question is: if we replace the backbones of YOLOv5 with other representative backbones (i.e. other neural network architectures) that are used in image classification, what will be the effects on the efficiency and performance of this new resulting

model? Can we derive an explicit relationship of the performance and efficiency trade-offs to better understand how performance and efficiency scale across different YOLOv5 backbones?

Background & Prior Work

Object Detection

For object detection, the main task at hand is, given an image as input, can a model the model will output the bounding box and its corresponding class or category.

Two-stage Models

The famous RCNN(Girshick et al. 2014) model was published in 2014 and it first proposed a two-stage method that first gives “proposals” and then it will train the model to extract features from those proposals and classify them. Later, Fast RCNN(Girshick 2015) and Faster RCNN(Ren et al. 2016) were published in 2015 and 2016 respectively. They both improved the performance of two-stage object detection on the base of RCNN and Faster RCNN proposed a network called RPN to select “proposals” and gave the idea of “anchor”. They follow a pattern that the model will have two detection heads, one for bounding box regression and one for classification.

One-stage Models

YOLO(Redmon et al. 2016) was first published in 2015 and its name is a contrast to RCNN because YOLO is an end-to-end model that doesn't need “proposals”. In 2016, SSD was published, and later YOLO series was developed to YOLO9000(Redmon and Farhadi 2017), YOLOv3, and YOLOv4, and they all published papers except for YOLOv5, the SOTA model developed by ultralytics on GitHub. They follow a pattern that the model will directly output a bounding box as well as class probability.

Technical Challenges

Backbone Selection

Due to the rapid development of network structures, there are so many models and modules we can get online, and many of them don't clearly specify their FLOPs and number of parameters, which increases the complexity for us to choose proper model backbones. In our project, we are supposed to choose those models whose number of parameters and flops are various and sparse, in order to form a robust regression model. Finally, we chose models on the basis of published year and model size. Those models are representative of lightweight and middleweight models and they are all from different years.

Dataset Selection

For object detection, computer scientists usually use two datasets. One is the MS COCO, which is more than 25 GB in total, and the other is the Pascal VOC dataset, which is much smaller. Because we planned to run our project on NYU HPC, the dataset couldn’t be too large or too small. So we decided to use a combination of several VOC datasets from different years. In the following section, we will present the exact dataset combination.

Training Time

The training time is the main challenge for our project. Each model would take more than 13 hours to complete the training process and before that, we needed to wait for available nodes on NYU HPC, and sometimes it took more than half an hour.

Environments

The original YOLOv5 repository requires many dependencies, and we spent a lot of time handling the environment we needed to build on NYU HPC, and during this time, Dan's HPC broke due to space/memory constraints, and with the help of HPC administrators, we finally built the environment successfully.

Approach

1. We planned to first find and modify the code of backbones we chose for YOLOv5 feature extraction, and the backbones are RepVGG, MobileNetV2/V3, GhostNet, DenseNet, ResNet18, and InceptionV4.
2. We planned to integrate the modified backbones into YOLOv5's structure and train each of the models (including the YOLOv5 itself) with the same hyperparameters and record the performance metrics, such as training time, training loss, mAP, and recall.
3. After the training process, we planned to calculate the efficiency metrics of each model, such as a number of parameters, FLOPs, MACs.
4. We planned to derive a relationship of performance and efficiency trade-offs among all these model combinations.

Implementation

Main Frameworks, Environments, and Packages

- PyTorch
- Thop (to calculate the number of parameters and FLOPs)
- Python 3.8
- PyTorch 1.10
- CUDA 11.3

Settings / Hyperparameters

- Initial learning rate: 0.01
- Scheduler: One Cycle Scheduler, with max learning rate 0.1 and max epoch of 200
- Batch size: 32
- Optimizer: SGD
- Image size: 320x320
- Epochs: 200

Datasets

- Train: VOC 2007 train and VOC 2012 train - Total 19,910 objects and 8,218 images
- Test: VOC 2007 test - Total 12,032 objects and 4,952 images

Experimental Design

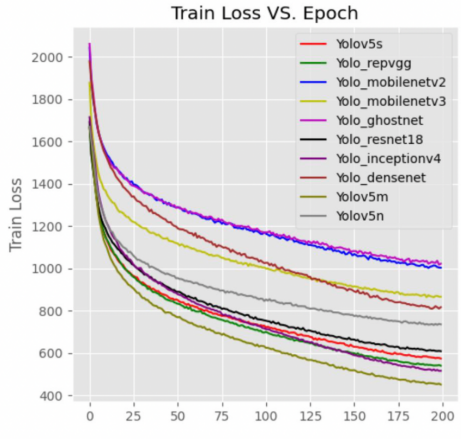
- To ensure a representative sample of models, we use well-known neural networks that debuted each year from 2015-2020 with different model sizes varying from lightweight to middleweight.
- During the training process, the hyperparameters should be set the same among models.
- We should record the performance metrics after each epoch.
- We would run those models on both V100 and RTX8000 with the same settings to ensure the robustness of our results.
  (Not fulfilled. Due to the inadequate compute nodes on NYU HPC)
- We would run those models on the same GPU several times and calculate the mean value to ensure the robustness of our results.
  (Not fulfilled. Due to the inadequate compute nodes on NYU HPC)

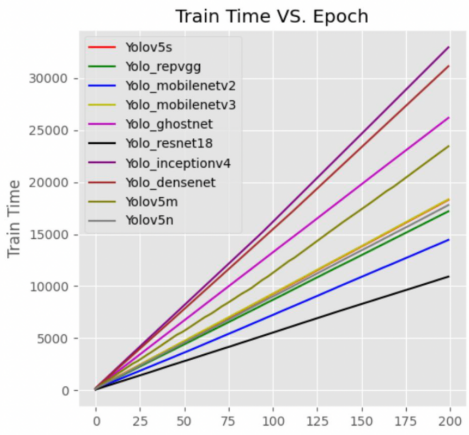
Evaluation

To evaluate our work, we monitor performance metrics (during and after training) as well as other metrics trying to capture each underlying model/backbone's complexity and efficiency. To capture performance, we capture training loss, training time, test mean average precision (mAP) across a variety of intersection-over-union thresholds (0.5 to 0.95) and test recall. As a proxy for complexity, we use the model's number of parameters and FLOPs (floating-point operations per single input/instance to perform inference with the model) to proxy for the model's efficiency; the higher the number of parameters, the more complex the model and the higher the FLOPs, the less efficient the model.

While imperfect, these measures/metrics attempt to capture the degree to which any model is a "good" one: relatively quick to train, low training loss, high mAP and recall on the test set, and efficient all without being too complex.

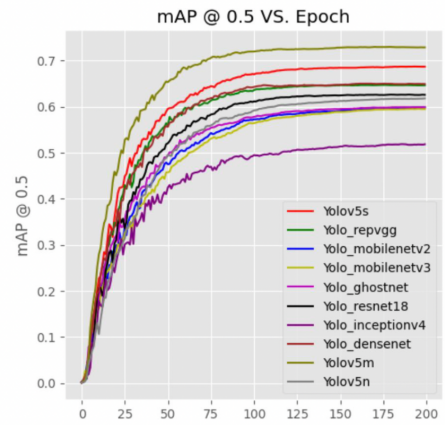
Training Loss & Training Time

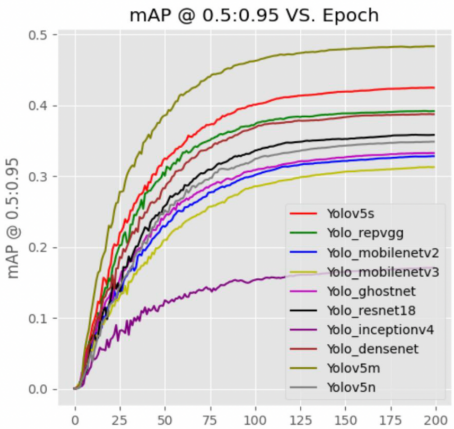




As we train our different backbones/models with the experimental settings described above, we track the training loss and training time epoch by epoch. Note that YOLOV5n/s/m describes different variants of YOLOv5—namely, nano, small, and medium, respectively, which describe the size of YOLOV5 in terms of parameter sizes. In ascending order, the smallest is nano, followed by small and then by the medium. In the figures above, we see that YOLOV5m does the best in achieving the lowest training loss across almost all epochs during training while its counterparts do less well. However, we see that there is no 1:1 correspondence between achieving low training loss and speed of training; compared to other model/backbones, YOLOV5m is middle-of-the-road in terms of required training time.

Test Performance: Mean Average Precision

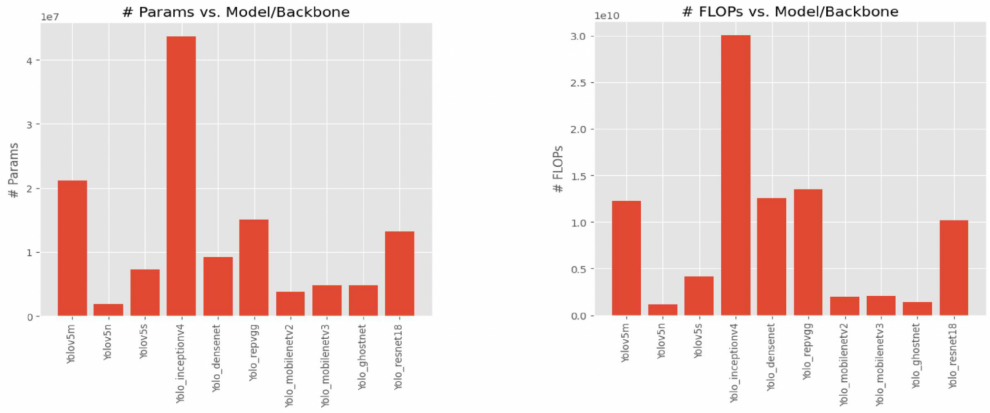




Mean average precision (mAP) is one of the key metrics used to benchmark performance on object detection tasks. The plots above show the mAP across all the different model/backbone combinations for a variety of intersection-over-union (IOU) thresholds from 0.5 to 0.95. Typically, an IOU threshold of 0.5 is used to calculate mAP but we include other thresholds as well to be holistic in understanding a model's performance across the board. For the thresholds "0.5 : 0.95", we calculate mAP for each threshold level from 0.5 to 0.95 and take the average mAP overall thresholds to arrive at the plot on the right.

In terms of test mAP, just like in terms of training loss, we see that throughout all 200 epochs of training, YOLOv5m again takes the top spot, achieving the best mAP at the 0.5 IOU threshold and at thresholds throughout 0.5 : 0.95. The second-best performer is yet another closely related model, the YOLOv5s (i.e. the small variant rather than the medium).

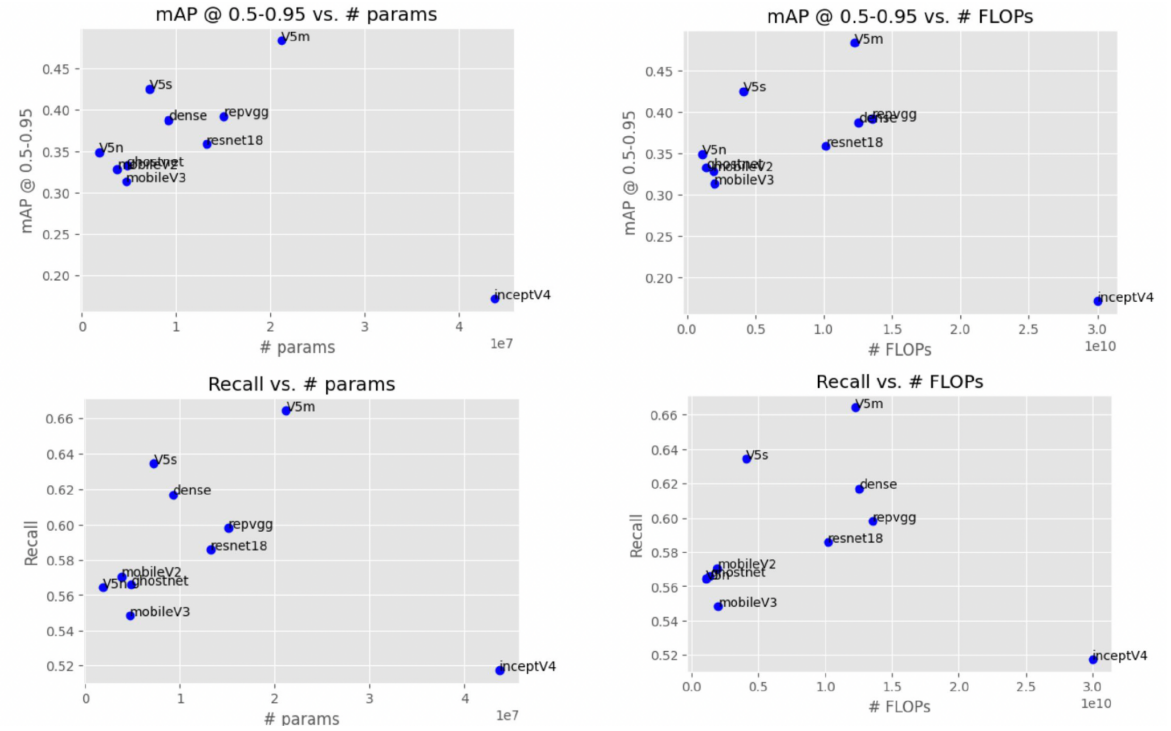
Complexity and Efficiency



In terms of model selection, while performance in terms of mAP and training loss are important—and perhaps the most important consideration in some cases—model efficiency is an important factor especially if deployed into online settings or compute/resource constrained environments. Model complexity is another factor to consider as it influences not just model efficiency but also the quality and length of time for training, re-training, fine-tuning, etc.

From the figures above, we see that the most complex model and the most inefficient model—as measured by the number of parameters and number of FLOPs, respectively—is the inceptionV4 backbone. Despite being so heavily over-parameterized and FLOP-heavy, the inceptionV4 backbone does not deliver great performance given such inefficiencies. Again, we see that while YOLOV5m tends to be slightly more over-parameterized than the other backbones, it does deliver SOTA performance (as seen in earlier figures) at half, if not less, the cost of the inceptionV4 backbone in both the number of parameters and the number of FLOPs.

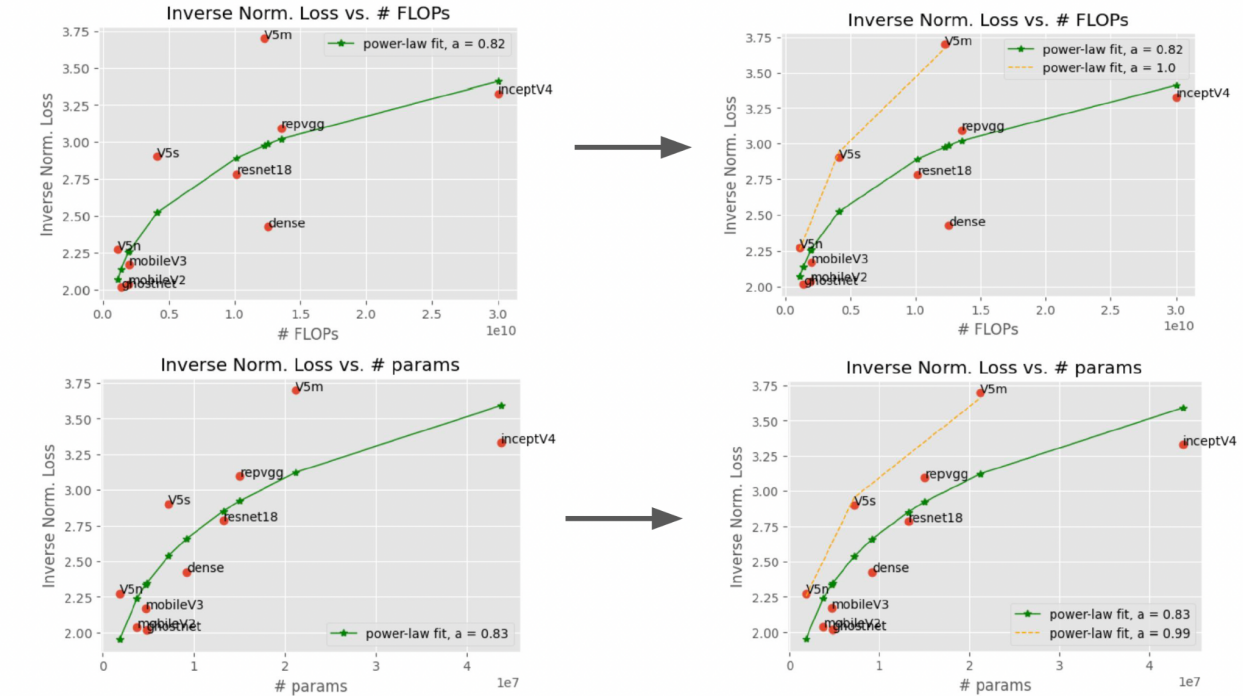
Performance vs. Complexity+Efficiency



Now, instead of looking at just performance or efficiency in isolation, we explore potential relationships that may exist between the two. As such, in the figures above, we show how different model/backbone combinations fare when looking jointly at both their respective performance—in mAP and recall—and their respective efficiency/complexity in the form of the number of parameters/FLOPs.

An initial glance reveals no clear trend or relationship: while increasing the number of FLOPs/parameters of a model is somewhat associated with an increase in either mAP or recall, we see two main issues. The first is the outlier that is the inceptionV4 module: as we have seen earlier, the inceptionV4 backbone is the most expensive/inefficient in terms of the number of parameters/FLOPs among the others but does not deliver in terms of achieving good performance as seen here. Secondly, while there is some positive association for the rest of the backbones between the number of parameters/FLOPs and increased performance (i.e. higher recall or mAP), the functional form of this relationship is unclear and tentative. Nonetheless, we continue to see the YOLOV5m, and YOLOV5s to a lesser extent, do well as they share a similar number of parameters/FLOPs as other backbones but do much better in recall and mAP.

Scaling and Efficient Frontier



When previously comparing recall and mAP against a model's number of FLOPs and number of parameters, we saw that there was no clear relationship that could capture the effect of the increased number of FLOPs/parameters on performance as measured by recall and mAP. However, if we define a metric called the inverse normalized loss, we see a clearer relationship.

The inverse normalized loss is calculated by taking the inverse of the trajectory of training loss values over the training process normalized be between 0 and 1 (i.e. one over the normalized loss, where the normalized loss is simply the training loss divided by the maximum loss encountered for that model/backbone during training). The reason why this normalization is necessary is due to the large absolute scale with which losses occur in training (e.g. loss values can span anywhere from a few thousand to a few hundred or even on the magnitude of ten). As such, in order to use training loss as a performance metric as we do with mAP and recall (which are already normalized to be between 0 and 1 by definition), this normalization is necessary for a standardized comparison. Taking the inverse of this normalized loss is a stylistic choice in order to keep comparisons with other figures the same (e.g. in recall and mAP vs. # FLOPs/parameters plots, the higher the recall/mAP the better), but otherwise does not affect our conclusion or calculations.

We suspect a power-law relationship between inverse normalized loss and the number of parameters/FLOPs across the different model/backbones and so we fit a power-law curve to examine how well the fit is (green line). We fit a log-log regression—namely, taking the log of the number of FLOPs/parameters and regressing it against the log of the inverse normalized loss—to calculate the power-law exponent, as shown in the legend of the figures above. The predicted values of this log-log regression are plotted (again, the green line) to form an "efficient frontier" for performance/efficiency/complexity trade-offs; in other words, the green line represents an estimate of how much of a performance gain (in terms of inverse normalized loss) one can expect given any model with some specified number of parameters/FLOPs. This estimate can help predict or inform how model complexity/efficiency scales with performance (and vice versa). Given the variety of different backbones we train here, we argue that this also captures the efficient frontier: points on the green line represent the maximum, optimal level of performance (in terms of inverse normalized loss) one can expect given a model with some number of parameters/FLOPs. Similarly, points close to or below the efficient frontier

or the green line indicate models that are close to optimal or sub-optimal in terms of performance given their number of parameters/FLOPs.

We note that our efficient frontier is unable to capture some models: namely, YOLOV5n/s/m and repVGG lie beyond the estimated green curve. However, these models are all relatively recent (YOLOV5 debuting in 2021, repVGG in around 2020). If we fit the power-law relationship only using the YOLOV5 series (the orange line), we see a very different efficient frontier forming. Although speculation, we believe that the YOLOV5 framework has architectural changes that can induce a shift in the scaling behavior of older backbone networks, resulting in an efficient frontier/scaling law that is much better than the older one (green line). If so, then continuing to build on the YOLOV5 framework without relying too much on these other backbone networks may be the path forward for better performance/efficiency trade-offs and gains.

Conclusion

Our work examined and experimented with a diverse set of backbones in the form of popular neural networks that debuted from 2016 to 2021, almost one per year, for YOLOV5 to examine the performance, efficiency, and complexity of these resulting models and their performance. After documenting these results for YOLOV5, we then derive an efficient frontier using a simple predictive model, illustrating the trade-offs between performance & efficiency/complexity for these different model/backbones, taking advantage of a power-law behavior to derive scaling relationships. Ideally, this derived predictive model and its estimated efficient frontier can be used to predict how new, unseen models—based on the number of parameters/FLOPs alone— might scale in terms of expected performance gains. For instance, in designing a new model architecture, one can calculate the number of parameters/FLOPs of said model and use the efficient frontier to estimate, without running/training the model, the maximum likely inverse normalized loss (or other performance metrics) said model can realize. With more time, we would have liked to test the predictive power of this estimated efficient frontier by collecting more data points (in the form of more trained models/backbones and their performance) to test against.

References:

Girshick, Ross. 2015. “Fast R-CNN.” In ICCV 2015.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation.” In CVPR 2014.

Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. “SSD: Single Shot MultiBox Detector.” In ECCV 2016.

Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. “You Only Look Once: Unified, Real-Time Object Detection.” In CVPR 2015.

Redmon, Joseph, and Ali Farhadi. 2017. “YOLO9000: Better, Faster, Stronger.” In CVPR 2017.

Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2015. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.” NIPS, 2015.