Ethan Davis, Robert Che, Yasmine Subbagh, Kyle Manning

Survey of Object Detection

***Abstract*—Autonomous vehicles require both reactive and proactive responses to their environment. In this paper we research one-stage and two-stage object detection with YOLO, Fast RCNN, Mask RCNN, and SSD. In general, one-stage object detectors are faster and have lower accuracy. On the other hand, two-stage models have higher accuracy and are slower. We use COCO as the backbone of our models. We use Ultralytics to train YOLO, Detectron2 to train Fast and Mask RCNN, and TorchVision to train SSD. Our results show that YOLO has the best speed performance, while Mask RCNN has the best accuracy. Additionally, we find support to explain why Fast and Mask RCNN models have greater accuracy in general. We believe that a wider variety of data and more training epochs for two-stage object detectors would provide better accuracy. Training and testing data practices that we were limited to due to resource constraints for our experiment are likely to have had great effect on our results that can be addressed in future work.**

***Index Terms*—Autonomous Vehicles, Fast RCNN, Mask RCNN, Object Detection, SSD, YOLO.**

I. Introduction

A. Background

Object detection is different from classification because the purpose of object detection is to classify multiple instances of objects in images whereas classification aims to classify images with a single instance of an object in them. In general, objection detection is either classified as two-stage or one-stage. The main difference between these two methods is that two-stage is accuracy-focused and one-stage is speed-focused.

Two-stage object detectors break down prediction into two distinct steps: firstly, region proposal which identifies potential regions of an image that might contain objects, and secondly, classifying objects in the proposed regions and refining their location with bounding boxes. Two distinct stages allow resources to be focused on each stage which allows better proposals, classification, and bounding box refinement. Separation of region proposal and classification allows the second stage to zoom in and detect small objects and objects in crowded scenes; two things that one-stage object detectors struggle with.

Notable two-stage object detection models are RCNN (Regional CNN), Fast RCNN, Faster RCNN, and Mask RCNN. Fast and Faster RCNN improved the speed of RCNN, where Fast introduced region of interest (ROI) pooling, and Faster RCNN introduced the region proposal network (RPN) creating a nearly end-to-end trainable model. Mask RCNN extended Faster RCNN to perform instance segmentation for each detected object [1].

Real-world applications of two-stage object detectors that require high accuracy include medical image analysis, autonomous driving, detailed scene understanding, and quality control in manufacturing. Detecting small tumors or lesions in CT or MRI scans requires high accuracy and precise localization is critical for treatment planning. Companies like Waymo rely heavily on robust detection and localization of pedestrians, cyclists, other vehicles, and traffic signs, all of which can be small or occluded. Identifying small defects or verifying component placement in complex assemblies demands high precision. Training these models requires large, labeled datasets, such as COCO, and careful tuning.

One-stage object detectors treat prediction as a regression problem. A CNN is used to extract features such as edges that are directly fed into a detection head that predicts bounding box coordinates, class probabilities, and confidence scores simultaneously across the image grid or feature map [2].

Real-world applications of one-stage object detectors include autonomous vehicles, security and surveillance, robotics, traffic management, retail analytics, and manufacturing quality control. Companies like Waymo rely heavily on real-time object detection. Monitoring systems can identify intrusions, unauthorized individuals, crowd density, and trigger security alarms. Traffic cameras can be monitored for vehicle flow, detection of accidents, and optimization of traffic signal timing.

B. Case Study

We apply object detection to autonomous vehicle research. One-stage object detection is needed because reaction is required in cases like a grocery cart rolling into the road. Alternatively, two-stage object detection is needed because proactive response is required to correctly take action for road signs. In the former case, classifying an object doesn’t matter as much as detecting it, and in the latter case, both detection and classification are required.

We use object detection models pretrained on COCO. The object detectors we researched are Fast RCNN, Mask RCNN, YOLO, and SSD. The first two object detectors are two-stage and the latter two are one-stage. Therefore, we hypothesize that the first two models will have the slowest and most accurate car detection, and vice versa for the latter two models.

Our research questions are as follows: How many true and false positives are detected? How many true and false negatives? Key metrics for our research are accuracy, precision, recall, and mAP. We also consider training and testing speed to be a key metric. Since we used Google Colab with public resources for computation, our measurements for speed are at worst rough approximations.

This case study models a real-world problem and its potential solutions. Solutions to this problem have two dimensions: speed and accuracy. On one hand, one-stage object detectors are fast and can significantly improve speed of inference. On the other hand, two-stage models can significantly improve accuracy. Both aspects are critical to safe autonomous driving.

II. Methods

This section focuses on presenting prominent deep learning models for object detection: Fast RCNN, Mask RCNN, YOLO, and SSD. These architectures represent a range of design approaches, each balancing detection speed, accuracy, and computational cost.

A. Fast RCNN

a. Background

The original Vanilla RCNN architecture described in 2013 paper [3] consisted of the Selective Search algorithm for region proposal, followed by a CNN to extract features for each proposal. The extracted features were then passed into a separate classifier and linear regressor model to classify the objects in the image and refine the bounding boxes.

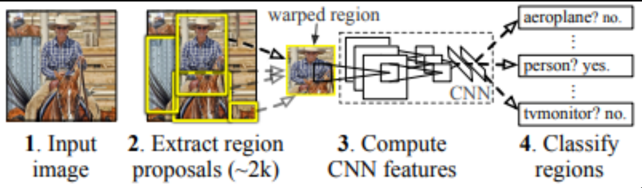


Figure 1.1.1: Vanilla RCNN Framework

Unlike its predecessor, Fast RCNN first performs convolution over the entire image, producing a shared feature map, which is then forwarded into the Region of Interest (RoI) pooling layer [4]. Although the RoIs are still determined by the Selective Search algorithm, Fast R-CNN allows for end-to-end training and supports batch processing since the classifier and regressor layers can directly branch off the RoI pooling layer.

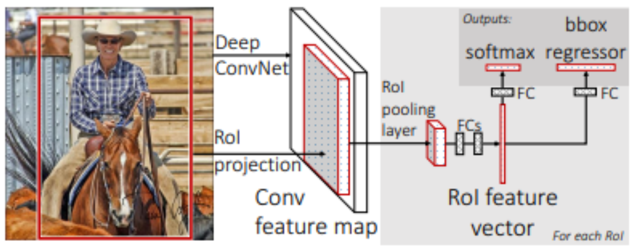


Figure 1.1.2: Fast RCNN Framework

b. Case Study

Since the original Vanilla RCNN paper did not clearly specify the CNN backbone used, we employed a ResNet-50 model as a substitute for feature extraction. Although ResNet50 was introduced approximately two years after the original R-CNN, it was chosen as a candidate because it served as a reasonable approximation of the CNN architectures commonly used at the time.

In the second iteration of our implementation, we used [5] an untrained version of Detectron2's Fast RCNN, which features a ResNet-50-FPN backbone. The model weights were randomly initialized using a fixed seed to ensure reproducibility.

The Fast R-CNN model was trained for 100 epochs using a batch size of 16 and a learning rate of 0.00025 on the same subset of data, the first 5,000 images from the COCO 2017 dataset. Initial testing over few epochs revealed explosive loss curves. To stabilize training, we enabled norm gradient clipping with a clip value of 1.

Initially our goal was to implement a Vanilla RCNN to establish a baseline of comparison with its other flavors. However, during the implementation process, we found that the region proposal component was incompatible with batching, as it required separate, non-parallelized processing for each image. To ensure standard batch sizes for all models, we switched to Fast RCNN as the closest equivalent replacement for Vanilla RCNN.

B. Mask RCNN

This subsection focuses on Mask RCNN, outlining its architectural components and the specific steps taken to implement and fine-tune the model for our case study.

a. Background

The Mask RCNN architecture is built upon the Faster RCNN architecture, adding pixel wise segmentation to combine both object detection and instance segmentation. It was first presented by Facebook AI Research at the Computer Vision Conference in 2017, it was the first to provide both object detection and instance segmentation in a single unified framework. The original Mask RCNN outperformed every existing single-model entry on every task at the time, including the Common Objects in Context (COCO) 2016 challenge winners.

The architecture of Mask RCNN is built upon the Faster RCNN framework, with the addition of an extra "mask head" branch, as seen in figure 1.2.1. While the pipeline of the framework produces object detection, the additional branch enables fine-grained pixel-level boundaries for accurate and detailed instance segmentation.

The model first extracts features from the input image using a backbone convolutional neural network, our specific version uses ResNet50. These features are then used by a region proposal network (RPN) to identify a set of candidate object regions, also known as regions of interest (RoIs). For each RoI, the RoIAlign layer extracts a small, fixed size feature map. This features map is then fed into two parallel branches. One branch predicts the object's class and defines the bounding box coordinates (Faster RCNN). The second branch generates a binary mask for each RoI, which effectively separated the object from the background at the precise pixel level.

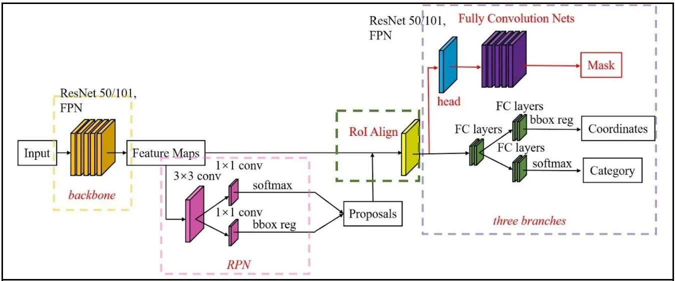


Figure 1.2.1: Mask RCNN Framework

The Mask RCNN network architecture consists of the ResNet50 CNN backbone for feature extraction, the RPN for generating proposals, and the two separate heads for object classification and bounding box regression, and for mask prediction.

b. Case Study

The Mask RCNN for this implementation used a ResNest50 FPN 3X backbone trained on the COCO dataset, consisting of 44 million parameters [6]. In addition to the pretrained backbone, it was fine-tuned, provided by Dectron2 on the car subset of the COCO dataset. Afterwards, the fine-tuned model on cars subcategory, is used to detect vehicles within the entirety of the COCO validation set. Object detection is completed with confidence scores ranging between 0.7 and 0.95 to observe the tradeoff of precision and recall.

The initial pre-fine tune training was done over 10 epochs with 16 samples per batch using the first 5,000 sample of COCO's training dataset. The stochastic gradient descent (SGD) optimizer with momentum started with a learning rate of 0.00025. The fine-tuning of the model was done again over 10 epochs, and with 16 samples per batch to mitigate the risk of RAM over-consumption on the GPU. The dataset for fine tuning were samples pulled from COCO that included cars, roughly 9,000 images were sampled for training.

The goal of using Mask RCNN over the other models presented in this case study is the added pixel level instance segmentation. While we are aware of the performance speed tradeoffs of this model, we hope to gain more precision and clarity in the results. The addition of masks rather than just bounding boxes are key in fields such as autonomous self-driving vehicles where data fuels safety and trust.

C. YOLO

a. Background

You only look once (YOLO) was first published by Redmon et al from the University of Washington, Allen Institute for AI, and Facebook AI Research in 2016. At that time, other real-time object detectors already existed. However, YOLO separated itself from the rest with its ability to detect objects with accuracy similar to two-stage techniques.

Real-time image processing is defined as 30 frames per second (fps) or higher. The original base model of YOLO processed at a rate of 45 fps. Its fast model, Fast YOLO, processed 155 fps while still achieving double the mAP of other real-time detectors [7].

YOLO also differed from detectors such as RCNN that repurpose classifiers to perform detection. For example, deformable parts model (DPM) used a sliding window where a classifier is run at evenly spaced locations over the entire image. RCNN was a newer approach than DPM and used region proposal methods to first find potential bounding boxes and then run a classifier on these boxes. Object detection was reframed by YOLO to be a regression problem that goes straight from image pixels to bounding box coordinates and class probabilities.

The original YOLO learned generalizable representations of objects. When trained on natural images and artwork, YOLO outperformed methods like DPM and RCNN by a wide margin. YOLO was less likely to break down when applied to unexpected inputs [7].

Although YOLO was fast, it did not have state-of-the-art accuracy. It also struggled to detect small objects. These properties were tradeoffs of using YOLO.

The network was implemented as a CNN. The initial convolutional layers extracted features from images and fully connected layers predicted output probabilities and coordinates. The original YOLO architecture was inspired by the GoogleNet model for image classification. There are 24 convolutional layers followed by 2 fully connected layers [7].

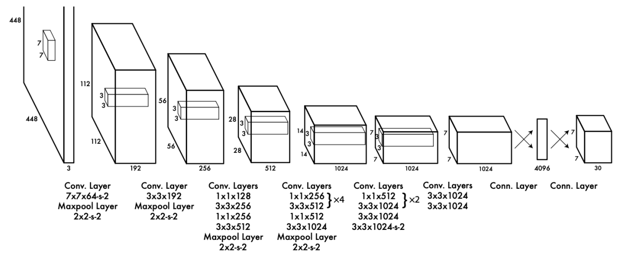


Figure 1.4.1: Original YOLO backbone with 24 convolutional layers and 2 fully connected layers.

The original YOLO network was trained for about 135 epochs with batch size of 64, momentum 0.9, and decay 0.0005. Overfitting was avoided by using a layer dropout rate of 0.5. Additionally, data augmentation was used with random scaling and translations up to 20% of the original image, and random exposure and saturation of the image by a factor of 1.5 in the HSV color space [7].

YOLO took input images and divided them in grid cells that predict bounding boxes in parallel. The architecture of YOLO imposed strong constraints on bounding box predictions. Each grid cell could only predict two boxes, and this limits the number of nearby objects it can predict. YOLO struggled with small objects in groups. The model used coarse features for predicting bounding boxes since it had multiple down sampling layers from the input image.

Additionally, the loss function treated all errors the same whether from small or large bounding boxes. In general, a small error in a large bounding box is benign but affects IOU much greater for a small bounding box. The main source of error for YOLO is incorrect localization.

Since it was originally published, YOLO has gone through many major versions. Between 2015 and 2020 YOLO went through 5 major versions. Currently, the latest version of YOLO is v11.

YOLO v2, also known as YOLO 9000, was introduced in the same year as YOLO and was designed to be faster and more accurate. It used anchor boxes as a set of predefined bounding boxes. When predicting bounding boxes, YOLO v2 used a combination of the anchor boxes and predicted offsets to determine the final bounding box. Also introduced were batch normalization to improve accuracy and stability of the model, multi-scale training to improve the detection performance of small objects, and a new loss function better suited to object detection tasks [8].

YOLO v3 introduced in 2018 used a new backbone, Darknet-53, whereas YOLO v2 used Darknet-19. The new YOLO v3 also used anchor boxes with different scales and aspect ratios which improved its ability to predict objects of different shapes and sizes because YOLO v2 anchor boxes all had the same size. To refine its ability to detect small objects, YOLO v3 also introduced feature pyramid networks (FPN) that are used to detect objects at multiple scales [8].

All YOLO versions v4 and higher are not considered the "official" YOLO because Joseph Redmon, the creator of YOLO, left the AI community prior to their development. YOLO v5 was introduced as an open-source project and is maintained by Ultralytics. Researchers from Taiwan and China have also released versions of YOLO. Currently, the latest version of YOLO is v11 that was released by Ultralytics.

For this case study we use YOLO v8 which has a CSPDarknet53 backbone [9, 10]. It also has neck and head structure that offers learning over multiple image resolutions. A detection head is also used that offers finely tuned anchor box assignment for IOU to better minimize the loss function. See Figure 1.4.2 that displays the YOLO v8 architecture.

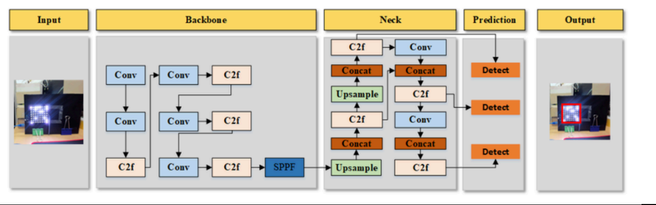


Figure 1.4.2: YOLO v8 architecture with backbone, neck and head, and detection head

b. Case Study

An Ultralytics YOLO yolov8m model pretrained on COCO was fine-tuned with the Stanford Cars training dataset. Afterwards the model was used to detect cars in the COCO validation dataset. This object detection is done with confidence scores ranging between 0.7 to 0.95 to observe the tradeoff of precision and recall.

Our YOLO model was fine tuned for 50 epochs with batch size 16. Larger batch sizes risk GPU out-of-memory errors. The Adam optimizer was used with initial and final learning rates as 0.0003 and 0.01 respectively. Generally, the Stanford Cars dataset is designed to be used to classify types of cars, however we used it to fine tune detection of the COCO car class.

Our goal was to fine tune our YOLO model well enough that it would maintain high precision and recall for car detection when filtering multi-object and multi-class images in COCO. It was a matter of convenience to use a YOLO model pretrained on COCO. Our intention with fine tuning using Stanford Cars was to suppress the other classes learned during pretraining.

D. SSD

a. Background

The Single Shot MultiBox Detector (SSD) architecture was originally proposed in 2015 by Wei Liu et al from UNC, Zoox, Google, and University of Michigan. In 2016, their paper was presented at the European Conference on Computer Vision.

SSD is a single stage grid-based object detection method that utilizes predefined anchor boxes instead of a separate region proposal network [12]. To handle objects of different shapes and sizes, the anchor boxes have varying aspect ratios and scales, as illustrated in Figure 1.5.1 [12].

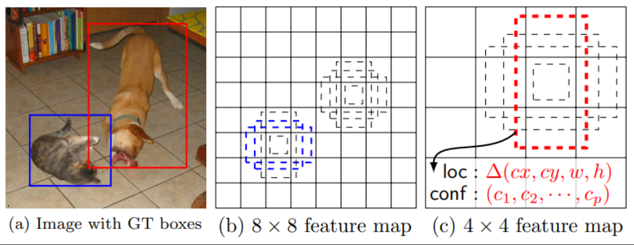

loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$
(a) Image with GT boxes
(b) 8×8 feature map
(c) 4×4 feature map

Figure 1.5.1: SSD framework

Additionally, the SSD architecture adds hierarchical auxiliary layers to the truncated base network (in the case of the original paper, a VGG16 backbone) [12]. These layers allow for improved object detection at different scales [12]. Finally, small kernels produce class scores and offset values relative to the default boxes [12].

b. Case Study

We employed a TorchVision SSD300 model with a VGG16 backbone to match the original paper. The model was pretrained on COCO and then fine-tuned with the Stanford Cars dataset to improve detection of the COCO car class.

For fine-tuning, we used the AdamW algorithm for the optimizer with a learning rate of 0.001 and a weight decay coefficient of 0.0001. We additionally used a batch size of 32 samples. The fine-tuning started with a maximum epoch count of 50, but we implemented early stopping based on mAP for the validation set with a patience limit of 15 epochs and validation mAP checks every 5 epochs. This resulted in the fine-tuning stopping early at 20 epochs and our saved model being fine-tuned for only 5 epochs.

III. Results

In the following section, we discuss the results, both qualitative and quantitative, obtained from evaluating the four object detection models. We analyze their performance in terms of precision, recall, and mean average precision (mAP), and provide visual examples to highlight each model's strengths and limitations.

A. Fast RCNN

Fast R-CNN demonstrates linear improvement over the course of 100 training epochs. However, as illustrated in Figures 2.1.1 and 2.1.2, the rate of progress was sufficiently gradual enough to suggest suboptimal convergence during gradient descent. Furthermore, the relatively lower mean average precision scores compared to foreground classification accuracy indicate that the model has a high rate of false positives.

Given that, as previously mentioned, the Fast R-CNN architecture has been deprecated in favor of Faster R-CNN, the stability issues encountered during training are understandable. Since gradient clipping was applied to mitigate the exploding gradient problem, the slow convergence observed in gradient descent suggests that the selection of hyperparameters in gradient clipping was suboptimal.

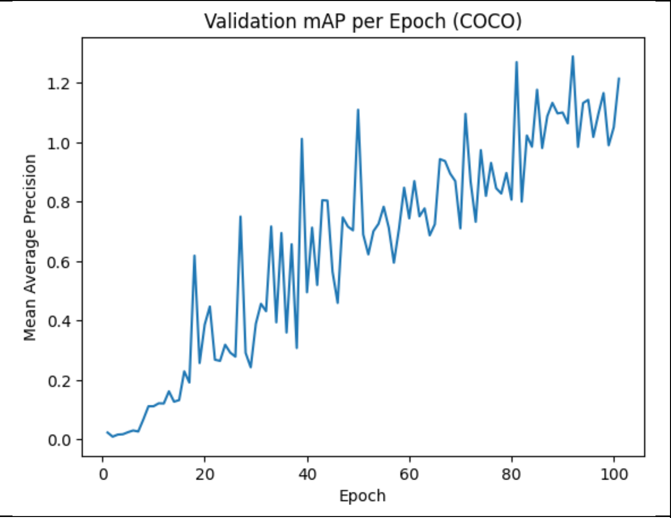


Figure 2.1.1 Total mean average precision of Fast RCNN over 100 Epochs for all classes on the COCO2017 validation set with increasing performance.
Note: Detectron2 reports mAP scores with a range between 0 to 100.

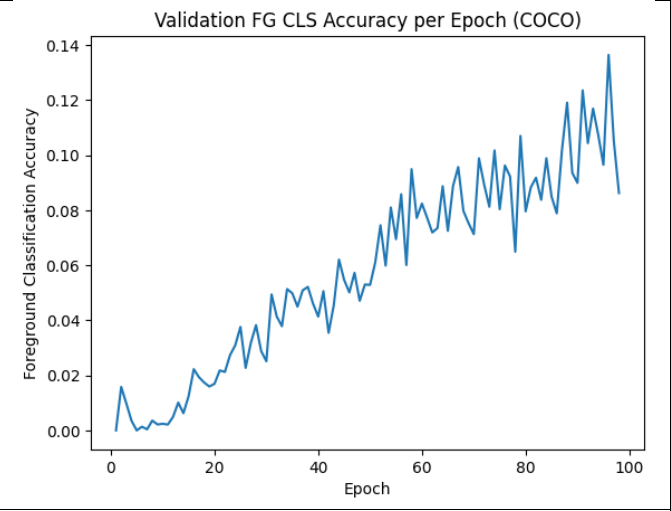
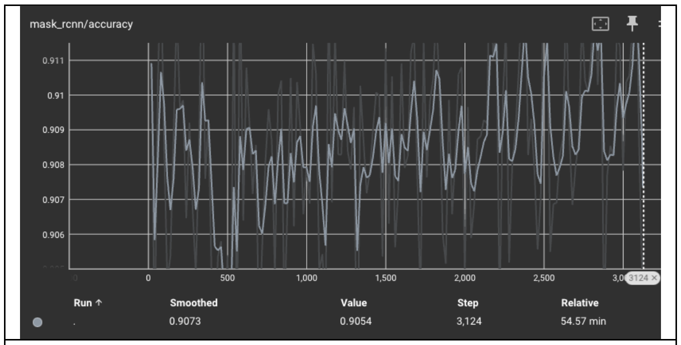


Figure 2.1.2 Total classification accuracy of foreground objects of Fast RCNN over 100 epochs on COCO2017 validation set with increasing performance

B. Mask RCNN

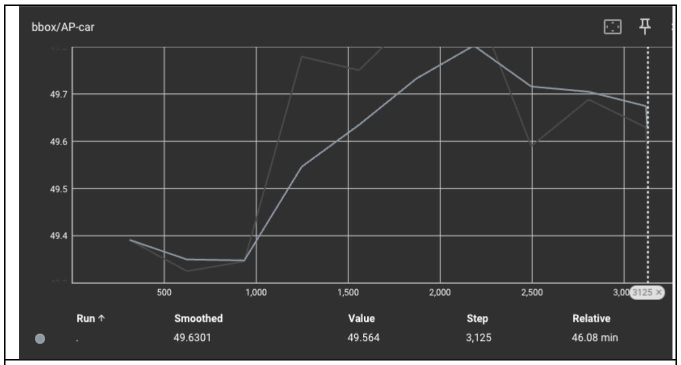
Our Detectron2 Mask R-CNN ResNet50 RPN 3X was initially tuned on a Google Colab A100 GPU for approximately just less than one hour. The model performs decently well across all classes. Figure 2.2.1 depicts that while the model is slowly on average increasing its accuracy, there is no consistency. Additionally, the mean average precision (mAP) score after this round of training was 76.1%.



Run ↑	Smoothed	Value	Step	Relative
.	0.9073	0.9054	3,124	54.57 min

Figure 2.2.1: Total classification accuracy of foreground object on Mask RCNN over 10 epochs on COCO 2017 validation set

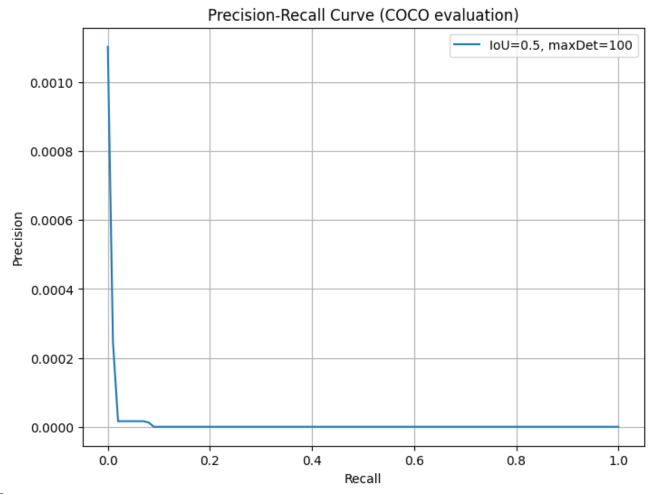
The model was then fine-tuned on the COCO car subset training data over 10 epochs, taking roughly 1.25 hours to complete. First results showed that the model was performing much better at classification for cars compared to the rest of the COCO classes, having an AP score as much as 20 times higher.



Run ↑	Smoothed	Value	Step	Relative
.	49.6301	49.564	3,125	46.08 min

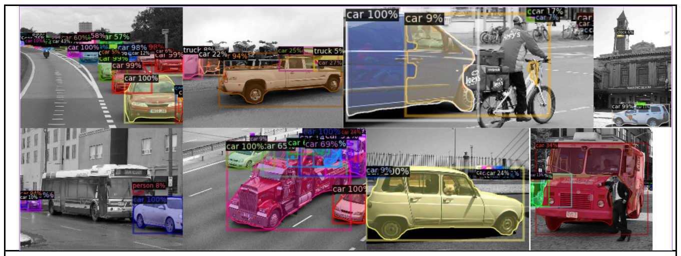
Figure 2.2.2: Average precision for the car class of Mask RCNN over 10 epochs on the COCO 2017 validation set

However, as we completed testing inference on the COCO dataset. We can see better under the hood that the model is not performing as well as we would like. Figure 2.2.3 depicts this prediction mismatch much better.



2.2.3: Precision-recall PR curve for Mask RCNN fine-tuned for cars

The model seems to have a strong lean towards cars after the fine-tuning on the car class. Often labeling items that are not cars, as cars, and even with higher confidence. Additionally, the model seems to no longer classify other classes as often, and if it does, with much lower confidence than before. We can see this in figure 2.2.4, where nominal items in the background are being classified as cars, and more prominent people in the foreground are not classified at all.



2.2.4: Examples of predicted bounding boxes and masks from COCO validation set, fine-tuned Mask RCNN on the COCO car subset

Overall, our validation results for our fine-tuned Mask R-CNN are extremely poor, even worse than our initial tuning stage on COCO. The model appears to be overconfident and heavily biased in "car" as a class, predicting a lot of false positives. The mAP score has dropped significantly, all the way down to 13.1%, a loss of 63%. The bar chart in figure 2.2.5 shows the heavy class imbalance, the low AP score across the board reflect the confusion that the model has, predicting other classes as "cars".

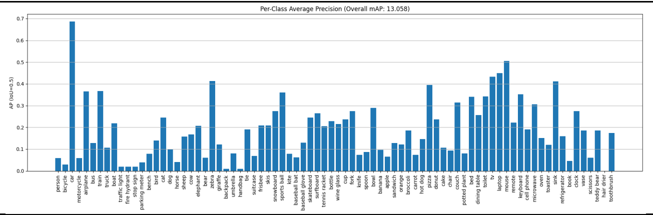


Figure 2.2.5: AP scores for all COCO classes after fine-tuning Mask RCNN on COCO cars subset

C. YOLO

Our Ultralytics yolov8m model was fine tuned in Google Colab with a T4 GPU for approximately 4 hours. For nearly every epoch out of 50, the loss decreased without flattening. This suggests that fine tuning would benefit from additional epochs.

Figure 2.3.1 shows the precision and recall (PR) curve from fine tuning. It shows that fine tuning goes remarkably well. In all cases precision and recall are nearly equal to 1.0. Figures 2.3.2 and 2.3.3 show examples of ground truth and predicted boxes from fine tuning on Stanford Cars.

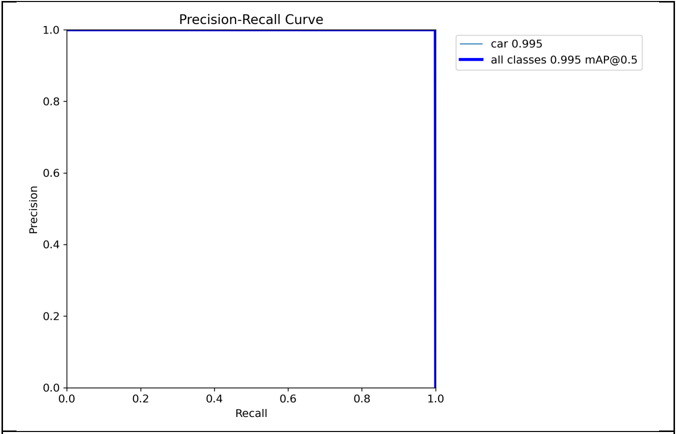


Figure 2.3.1: PR curve from fine tuning Ultralytics YOLO on Stanford Cars with high performance results

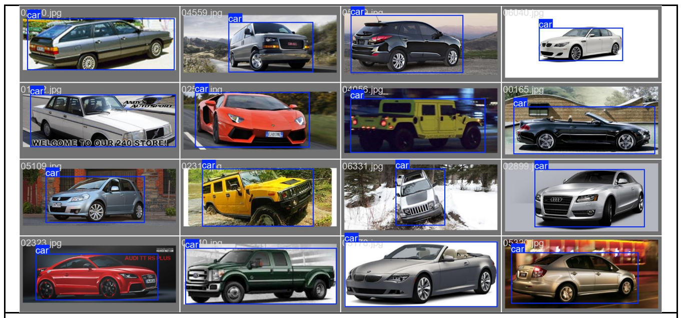


Figure 2.3.2: Example of ground truth bounding boxes from Stanford Cars

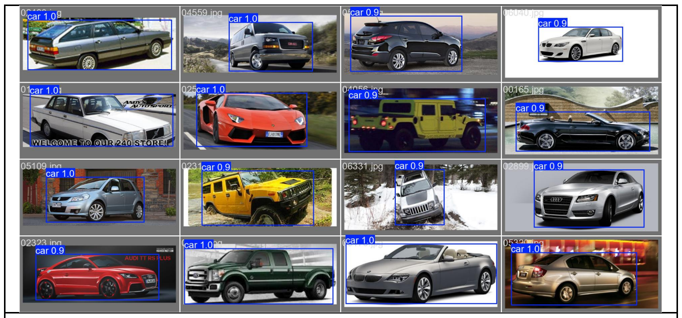


Figure 2.3.3: Example of predicted bounding boxes from fine tuning YOLO on Stanford Cars

Our first impression of the results from fine tuning YOLO was that they were promising. However, our results from prediction on the COCO validation set were unlike our results from fine tuning. See Figure 2.3.4 for our PR curve from prediction on the COCO validation set with mAP 0.028. Compare this to our mAP 0.995 that was from fine tuning.

Figures 2.3.5 and 2.3.6 show examples of ground truth and predicted bounding boxes on the COCO validation set. We see cases where YOLO mistakes the front of planes for cars, and this is understandable. However, we also see cases where groups of scooter or motorcycle wheels are detected as cars, and this is less understandable since the body of the car is not represented.

Additionally, we see cases where cars are not detected in images when we are predicting on the COCO validation set. These are images where cars are small relative to the scale of the image. We know that YOLO struggles to detect small objects in general. Even since its original architecture YOLO has struggled to detect small objects. Recall that the reason for this is related to the design decision that has grid cells predict classes for a limited number of neighbor cells. This design is a reason why YOLO is fast but also why it can be less accurate and miss small objects.

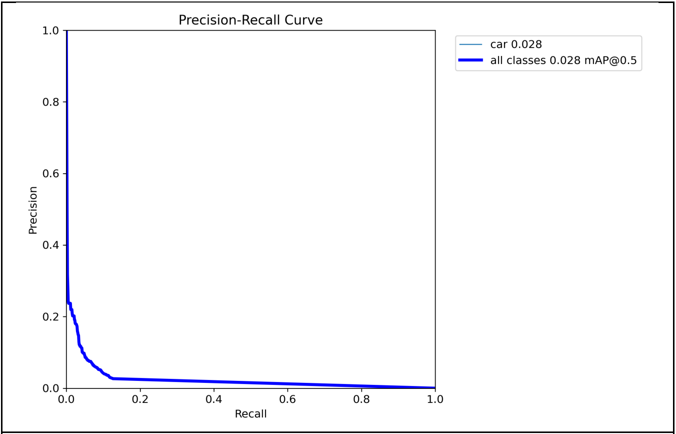


Figure 2.3.4: PR curve from inference on COCO validation data with low performance

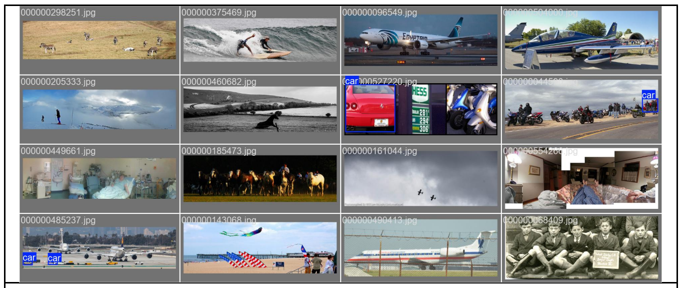


Figure 2.3.5: Example of ground truth bounding boxes of cars from the COCO validation set

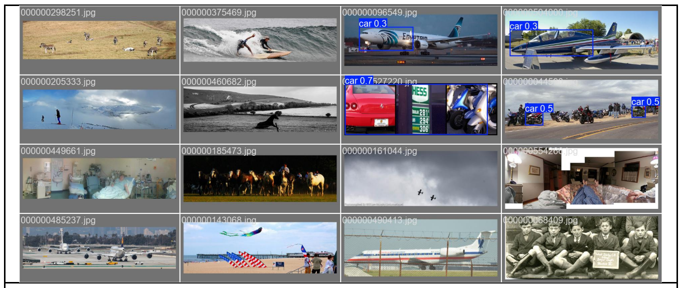


Figure 2.3.6: Example of predicted bounding boxes of cars on the COCO validation set using our fine-tuned YOLO model

Not pictured here are cases where plates or bowls of food are detected as cars. It is possible that this happens because plates, bowls, and cars generally have rounded edges. Also, food and the body of cars is often colorful. Since YOLO solves a regression problem for object detection, it is possible that these features of cars and food make them similar. Again, this design choice of YOLO architecture makes it fast but at the same time less accurate.

Our test results from YOLO after 50 epochs are poor. However, they are better than our first YOLO model that was fine-tuned on Stanford Cars for just 10 epochs. Also remember that our loss function from fine tuning YOLO had not flattened after 50 epochs. This suggests that we have not fine-tuned YOLO well enough to accurately detect cars in the YOLO validation set.

D. SSD

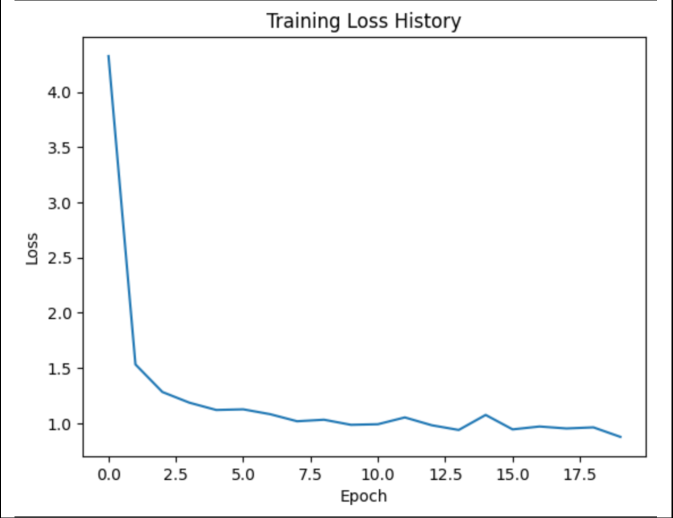


Figure 2.4.1: Training loss history for SSD

As illustrated in Figure 2.4.1, the loss during fine-tuning slowly decreased over the 20 epochs before early stopping triggered. However, the validation mAP after just 5 epochs (0.6734) was higher than the validation mAP values after 10, 15, and 20 epochs (0.6684, 0.6486, and 0.6506 respectively). With the model fine-tuned for 5 epochs, the mAP for the Stanford Cars test set was 0.6749.

Moving on to inference with COCO, our SSD model's performance was extremely poor. Even at a very low confidence threshold of 0.05, the model was unable to produce any true positives across almost 10,000 images.

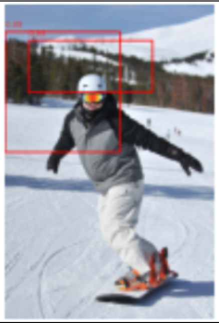


Figure 2.4.2: An example of false positives produced by SSD (detecting cars where there are none)

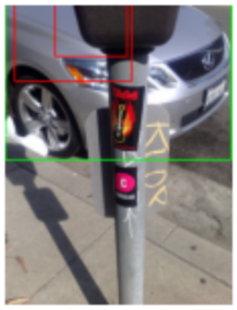


Figure 2.4.3: An example of false positives produced by SSD (correctly identifying a car, but the predicted bounding boxes are under the IoU threshold in relation to the ground truth box)

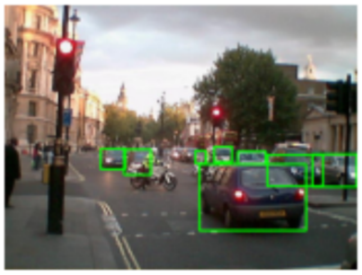


Figure 2.4.4: An example of false negatives produced by SSD (not detecting ground truth cars)

Figure 2.4.2 displays an example of the type of false positives that the SSD model commonly struggles with. The model frequently detects cars where there are none. Meanwhile, Figure 2.4.3 displays a different kind of false positive that the model has issues with. Even though the model appears to recognize certain components of a car here, the predicted bounding boxes are well under the IoU threshold of 0.5 in relation to the ground truth bounding box. Figure 2.4.4 shows an example of several false negatives (i.e. not detecting ground truth cars). The model appears to struggle with recognizing cars from certain angles such as the backside, as well as recognizing cars that are further away.

IV. Discussion

In the following discussion, we summarize the key findings from our evaluation of the four object detection models and reflect on the implications of their performance. We also address the limitations encountered during the project, including dataset constraints, class imbalances, and resource limitations that may have impacted the results.

A. Summary

We tested four solutions to detect cars in images: YOLO, Fast RCNN, Mask RCNN, and SSD. The key dimensions of our solutions are speed and accuracy. In other words, our central research question was how quickly and accurately can we detect cars in images?

Fast RCNN was trained with validation from scratch using the first 5000 images from the COCO training set and first 625 images from the COCO validation set. The model was tested using the second 625 images from the COCO validation set. Training took 6 hours over 10 epochs and was performed with a Google Colab T4 GPU. Testing took 1 minute. The mAP from testing across all classes was 0.016.

Mask RCNN was pretrained on COCO. It was fine-tuned with validation using the 9000 images from the COCO training set that contain car annotations. The first 5000 car images were used for training and the next 1000 car images were used for validation. The model was tested using the 5000 images from the COCO validation set. Fine tuning took 1 hour over 10 epochs and was done with a Google Colab A100 GPU. Testing took 12 minutes. The mAP from testing for car detection was 12.552.

YOLO was pretrained on COCO. It was fine-tuned using the first 6000 images from the Stanford Cars training set and the first 600 images from the Stanford Cars validation set. The model was tested using the 5000 images from the COCO validation set. Fine tuning took 4 hours over 50 epochs and was done with a Google Colab T4 GPU. Testing took 2 minutes. The mAP from testing for car detection was 0.0215.

SSD was pretrained on COCO. It was fine-tuned using the first 4000 images from the Stanford Cars training set, and it was tested using roughly 10,000 images from COCO. Using a Google Colab L4 GPU, fine-tuning took 30 minutes over 20 epochs and testing took 7 minutes. The model was unable to produce any true positives.

As we expected before our experiment, these results show that YOLO and SSD have faster inference than both Fast and Mask RCNN. We expected this because YOLO and SSD are one-stage models and both RCNN models are two-stage. This supports the idea that one-stage models should be used in cases where autonomous vehicles must react to their environment.

Additional factors that make YOLO faster than Fast and Mask RCNN, as well as SSD, is its architecture. YOLO v8 that we used for our experiment uses a CSPNet53 backbone and the RCNN models use ResNet50. CSPNet53 is designed to be faster than ResNet50 [9, 11]. It splits features maps so that some go through residual layers with heavy computation and others go through lighter computation. ResNet is a standard benchmark in CNN architectures that CSPNet53 was designed to be faster than.

Unexpectedly YOLO also has higher mAP than the Fast RCNN model. We believe Fast RCNN has a lower mAP because it has limited training data and epochs. However, considering that Fast RCNN's mAP is close to YOLO's, we believe that Fast RCNN's mAP would be higher with more training and testing resources. Mask RCNN outperformed YOLO in our evaluation. However, its mAP remains relatively low, likely due to suboptimal training. We believe that with a more diverse training dataset and additional fine-tuning epochs, Mask RCNN could achieve significantly better results. As a two-stage detector, it has the potential for higher accuracy when properly trained. With more experimentation in the training routine, we also believe that SSD could achieve much better results and offer a middle ground between the RCNN models and YOLO in terms of speed and accuracy.

B. Limitations

a. Model Testing

Our model backbones were trained on COCO which is commonly used for object detectors with popular libraries like PyTorch, Detectron2, and Ultralytics. We also use the COCO validation set to test our models. This is a limitation of our experiments because the COCO validation set is indirectly used to train our models.

Since we test our models on the COCO validation set, we introduce bias into our experiments. To maintain internal validity for our experiments, we should test on an entirely unseen dataset. Pascal VOC datasets are real-life images similar to COCO and since they were not used during training, they would be better datasets to test our object detection models.

b. YOLO

Ultralytics trains YOLO models on COCO and all its 80 classes. This dataset has a wide range of scenes, scales, and hues from real-life. Our YOLO model was fine-tuned on the Stanford Cars dataset that has well-centered high-quality images of cars. Unfortunately, Stanford Cars and COCO do not have similar enough images for fine tuning YOLO to be generally effective during predictions.

Fine tuning on Stanford Cars overfits our model so that it best detects cars when they are well-centered and high-quality in images. However, it also makes false positive predictions of other objects like food when they have these features too. See Figure 3.1.1 where a well-centered high-quality image of food with rounded edges of the plate and colorful body is misclassified as a car. Also see Figure 3.1.2, one of few images from the COCO validation set that has a well-centered high-quality car.

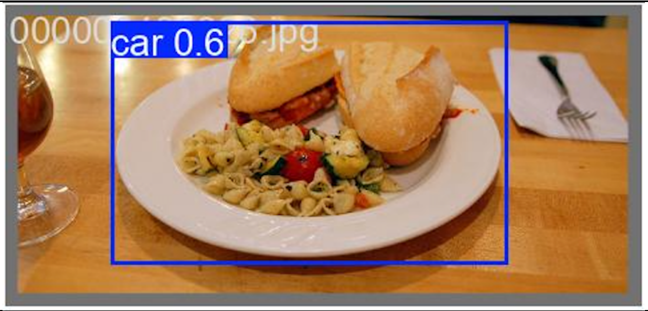


Figure 3.1.1: YOLO misclassifies well-centered high-quality image of food like Stanford Cars images are formatted

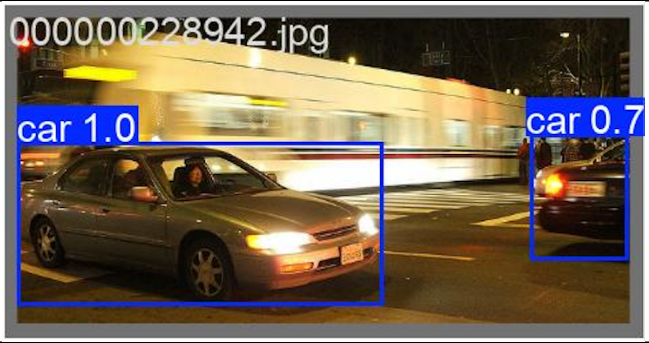


Figure 3.1.2: YOLO model that was fine tuned on Stanford Cars classifies a well-centered high-quality image of a car from the COCO validation set with 1.0 confidence

It is possible that fine tuning on the Stanford Cars dataset could perform better predictions on the COCO validation set if techniques like random scaling, jitter, and hue augmentation were used during fine tuning. Furthermore, if we know we are doing object detection of cars in the COCO validation set, then maybe using Pascal VOC would be better for fine tuning since its images are similar to COCO. The COCO dataset is messy, has occlusion, and low-resolution images of cars. The Stanford Cars dataset was not similar enough out of the box to detect cars in real-life images.

c. Fast RCNN

Like Vanilla RCNN, very few libraries still support the Fast RCNN architecture. In addition, both models are known for their slower training and processing speeds. Since gradient clipping was not employed as a hyperparameter in the other models, it was difficult to directly observe the advantage of Fast R-CNN's generally higher accuracy within the same number of training epochs.

Additionally, gradient clipping was approached heuristically in our experiment. It may be feasible to evaluate the model's training performance across a wide range of gradient clipping values over an extended number of epochs. A gradient clipping value of 0.01 is generally recommended when training over larger numbers of epochs to maintain training stability. A two-way ANOVA can be conducted by varying the choice of optimizer.

d. Mask RCNN

Due to the nature of Mask RCNN's architecture and outputs, any data used for either training or validation requires masking input. While the goal for all the models was to be fine-tuned on car specific data that was different than the data used for initial training, most datasets did not include masking or required a lot of preprocessing. While the other models fine-tuned on Standford Cars, the lack of mask data and limited time forced us to use a subset of COCO. The subset was created by only including samples that had cars as ground truths. It is understood that this is not good practice, to be using the same dataset for multiple levels of training, as well as used for testing. Due to the complex architecture of Mask RCNN and the added cost of instance segmentation, it required a significant amount of GPU memory and training time. As a result, we had to make compromises in terms of batch size, training iterations, and augmentation strategies to ensure the model could be trained within our available hardware and time constraints. These adjustments may have limited its full potential performance in our experiments.

e. SSD

Compared to YOLO (another one-stage architecture), SSD is generally expected to trade some speed in exchange for higher accuracy. The model did provide a middle ground between the RCNN models and YOLO in terms of speed, but it was unable to produce any true positives. This is likely attributable to the training routine we utilized. SSD may have required more extensive fine-tuning on the Stanford Cars dataset and a less restrictive early-stopping mechanism. The SSD architecture also could have struggled with the shift from the Stanford Cars dataset (centered and high-quality car images) to the more diverse COCO dataset.

Additionally, our SSD model would possibly have benefited from an architecture different from that described in the original paper. For instance, the NVIDIA SSD300 model replaces the VGG16 backbone with a ResNet-50 one due to the VGG16 backbone being "obsolete" [13].

C. Future Work

In this project, we primarily used the COCO dataset for both training and evaluation. While COCO provides a rich and diverse set of object categories and annotations, using the same dataset for both stages limits our ability to assess the model's generalization capabilities on truly unseen data.

Moving forward, we aim to evaluate our model on external benchmark datasets such as Google Open Images, Pascal VOC, and Cityscapes. These datasets offer a variety of scenes, object distributions, and annotation styles that would allow us to better understand the robustness and transferability of our model. Unfortunately, due to time constraints, we were unable to integrate and evaluate against these datasets in the current iteration.

Expanding to other datasets will help identify potential overfitting to COCO-specific features, assess performance on rare or unseen classes, and guide improvements in training strategies to enhance generalization in real-world scenarios.

Furthermore, YOLO is well supported through Ultralytics that makes training models easy. Mask RCNN is also well supported through libraries like Detectron2 and PyTorch. However, Vanilla and Fast RCNN are deprecated because Faster RCNN is now faster and more accurate. To get results with better accuracy with RCNN models, we should limit future experiments to current RCNN architectures.

D. Conclusion

We have trained, tested, and compared one-stage and two-stage object detection models for the purpose of autonomous vehicle research. Using YOLO, Fast RCNN, Mask RCNN, and SSD we hypothesized that one-stage models would be faster and have lower accuracy, and vice versa for two-stage object detectors. Our results showed that YOLO performed best in terms of speed, and Mask RCNN performed best in terms of accuracy. However, we provided reasons why our experiment still supports two-stage object detection over single-stage for our case study and the improvements for all models.

References

[1] Ultralytics. (n.d.). Two-Stage Object Detectors - Discover the power of two-stage object detectors—accuracy-focused solutions for precise object detection in complex computer vision tasks. https://www.ultralytics.com/glossary/two-stage-object-detectors

[2] Ultralytics. (n.d.-a). One-Stage Object Detectors - Discover the speed and efficiency of one-stage object detectors like YOLO, ideal for real-time applications like robotics and surveillance. https://www.ultralytics.com/glossary/one-stage-object-detectors

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," arXiv (Cornell University), Nov. 2013, doi: https://doi.org/10.48550/arxiv.1311.2524

[4] R. Girshick, "Fast R-CNN," 2015 IEEE International Conference on Computer Vision (ICCV), pp. 1440–1448, Dec. 2015, doi: https://doi.org/10.1109/iccv.2015.169.

[5] "Detectron2," ai.facebook.com. https://ai.meta.com/tools/detectron2/

[6] E. Hassan, N. El-Rashidy, and F. M. Talaa, "Review: Mask R-CNN Models," Nile Journal of Communication and Computer Science, vol. 3, no. 1, pp. 17–27, May 2022, doi: 10.21608/njccs.2022.280047.

[7] Joseph, R., Santosh, D., Ross, G., & Ali, F. (2015). You only look once: Unified, Real-Time Object Detection. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1506.02640

[8] Kundu, R. (n.d.). YOLO: Algorithm for Object Detection Explained [+Examples]. V7. https://www.v7labs.com/blog/yolo-object-detection

[9] Torres, J. (2025, January 2). YOLOv8 Architecture; Deep Dive into its Architecture -Yolov8. YOLOv8. https://yolov8.org/yolov8-architecture/#2_YOLOv8_Architecture_Overview

[10] Yaseen, M. (2024). What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2408.15857

[11] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick, "Mask R-CNN," presented at the Conference on Computer Vision, 2017, pp. 2961–2969. Accessed: Jun. 06, 2025. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2017/papers/He_Mask_R-CNN_ICCV_2017_paper.pdf

[12] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, September). Ssd: Single shot multibox detector. In European conference on computer vision (pp. 21-37). Cham: Springer International Publishing. https://doi.org/10.48550/arXiv.1512.02325

[13] "SSD v1.1 for PyTorch." NVIDIA NGC Catalog catalog.ngc.nvidia.com/orgs/nvidia/resources/ssd_for_pytorch