A Comparison of Single-Stage and Two-Stage Object Detection Algorithms in Autonomous Vehicles Against Adversarial Attacks

Research Question: How do the object detection algorithms YOLOv5 and Faster R-CNN, used for automated vehicle technology, compare their robustness against adversarial attacks?

Subject: Computer Science

Word count: 3939

Table of Contents

I. Introduction

With the increase in the use of autonomous vehicles, robust control systems must be built to guarantee the safety of drivers and pedestrians. Autonomous cars are built with sensors and algorithms to detect their environment, find their track, and make fast and accurate decisions. While it is an advantage that the systems can make logical decisions in a fast manner, when the object detection algorithms fail to perform accurate detections of the environment, the driver's safety becomes at risk. Recent studies on adversarial attacks showed that unnoticeable intended perturbations (see Appendix A for definition) in visuals such as t-shirt designs(Kaidi, et al.)[1] or stickers on stop signs(Kevin, et al.)[2] cause misclassifications of object detectors and lead to wrong decision making of the vehicle.

Autonomous vehicles utilize single-stage or two-stage object detection algorithms for making robust detection. While single-stage detectors detect the location of objects and classify them in the same process, two-stage detectors first predict the placement of an object, and then classify it. This difference in the algorithms has different strengths and weaknesses when faced with adversarial images. Comparing and finding the more robust algorithm type is worth investigating as it provides insights for making the right algorithm choice for autonomous vehicles and provides further observations on areas for improvement.

This essay will examine the following research question: "*How do the object detection algorithms YOLOv5 and Faster R-CNN, used for automated vehicle technology, compare their robustness against adversarial attacks?"* by utilizing a custom database generated by road images from Google Maps that are labeled manually over Roboflow.

Word count: 260

[1] Xu, Kaidi, et al. "Adversarial t-shirt! evading person detectors in a physical world." Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16. Springer International Publishing, 2020.
[2] Eykholt, Kevin, et al. "Robust physical-world attacks on deep learning visual classification." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

II. Background Information

II.I. Object Detection

Autonomous vehicles use sensors like vision cameras, LiDAR, and ultrasonic sensors. These sensors provide information about the vehicle's surroundings not only in binary image pixels but also information about the texture, temperature, etc. of the objects around it. This data is then transferred into object detection algorithms where the features from the data are extracted and classified inside segments or boxes. For object detection, neural networks such as Convolutional Neural Networks(CNNs) are used. CNNs try to detect objects according to the pre-set features of the object (for example a stop sign's red color is a feature for one layer and its circular shape is a feature for another).

When an image is input into a CNN, it undergoes two key processes called the forward pass and the backward pass. In the forward pass, the network applies filters to the image to detect features and uses pooling operations to reduce the size of the data by summarising areas of the image into single values, which helps in handling smaller, simplified data. After processing the image, the network evaluates how well it performed using a loss function, essentially calculating the error between the predicted outcomes and the actual values.

In the backward pass, the network adjusts its internal settings or weights based on this error, aiming to reduce it through a method called gradient descent. This involves mathematically determining how each weight should be changed to decrease the error, and updating them accordingly. **Equation 1** represents the mathematical representation of gradient descent. w_f is the final weight applied to the feature, w_i is the initial weight of the feature, α is the learning rate determining the incremental step, and $\frac{\delta L}{\delta w}$ is the gradient of the loss function with respect to the weight.

$$w_f = w_i - \alpha\frac{\delta L}{\delta w} \quad (1)$$

The process of adjusting weights continues from the last layer of the network back to the first, allowing the network to improve its accuracy in feature detection and overall prediction. The

operation is illustrated in **Table 1**. This cycle of forward and backward passes repeats, enabling the CNN to learn from the data progressively.

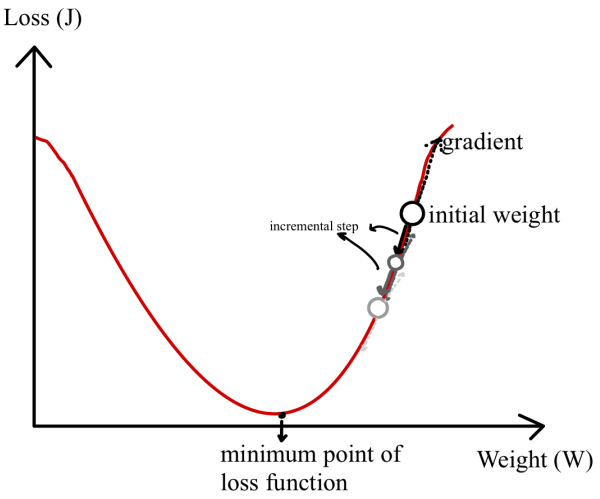


Table 1. Representation of gradient descent in a Loss(J) vs. Weight(W) graph

II.I.I. Two-Stage vs. Single-Stage Detectors

Two-stage detectors operate by first generating a set of region proposals that might contain objects (see **Table 2**), and then classifying these regions and refining their locations. This two-step process allows for high accuracy and precision, as the initial stage focuses on identifying potential object locations, and the second stage refines these predictions. Examples of two-stage detectors include the Region-based Convolutional Neural Network (R-CNN) family, such as Fast R-CNN and Faster R-CNN. These models typically achieve superior performance in terms of detection accuracy but at the cost of higher computational complexity and longer processing times.

In contrast, single-stage detectors streamline the object detection process by performing both object localization and classification in a single step (see **Table 2**). This approach leads to faster detection times and lower computational requirements, making it suitable for real-time applications. Single-stage detectors, such as the You Only Look Once (YOLO) series, prioritize speed over precision, making them ideal for scenarios where quick decision-making is critical.

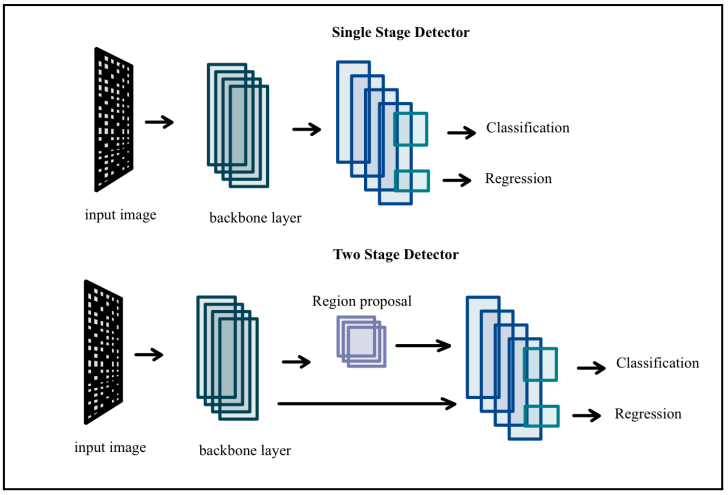


Table 2. A diagram representation of Single Stage and Two Stage Detectors

In the context of autonomous driving, object detection algorithms are crucial for identifying pedestrians, vehicles, traffic signs, and other obstacles. The ability to quickly and accurately detect objects in real-time ensures the safety and efficiency of autonomous navigation systems. Whether using two-stage or single-stage detectors, the choice of algorithm depends on the specific requirements of the application, balancing the trade-offs between detection accuracy and computational speed.

II.I.I.I. YOLOv5 Algorithm

YOLOv5(You Look Only Once version 5)[3] is a single-stage object detection algorithm. The structure of the algorithm enables making fast and accurate real-time detections and provides accuracy for small and close object detections. Additionally, it is a common model used for autonomous vehicle technologies. **Table 3** shows the steps of training a YOLOv5 model.

Initialization of YOLOv5	• Loading pre-trained weights of YOLOv5 • Setting parameters of the image size, anchors(predefined bounding box shapes), normalizing pixels, and turning the image into a tensor (see Appendix A for definition)

[3] Jocher, Glenn, et al. "ultralytics/yolov5: v7. 0-yolov5 sota realtime instance segmentation." Zenodo (2022).

Pre-Processing Input Image	● Resizing the image, normalizing (see Appendix A for definition) pixels ● Turning the image into a tensor
Forward Propagation (repeated for each layer)	● Applying convolution, batch normalization, and linearizing using Sigmoid (SiLU)(see Appendix A for definition) ● Applying special layers for feature enhancement ● Using detection heads which are final predictions from extracted features by determining bounding box coordinates, defining an objectness score, predicting class probabilities
Post-Processing	● Making sure bounding boxes do not overlap by applying the Non-Max Suppression method ● Adjusting bounding box coordinates from the resized image to the original image scale
Output	● Outputting bounding boxes, printing confidence scores, and labeling the classes of the objects

Table 3. Steps of training a YOLOv5 algorithm

See Appendix B to find the Python code I wrote for training the YOLOv5 algorithm with the custom dataset.

II.I.I.II. Faster R-CNN Algorithm

Faster RCNN[4] is a two-stage object detection algorithm that combines a region proposal network (RPN) with a robust CNN-based detection network. This architecture enables Faster R-CNN to deliver high accuracy and robustness in object detection, making it particularly effective for complex and varied environments. **Table 4** shows the steps of training a Faster RCNN model.

[4] Ren, Shaoqing, et al. "Faster R-CNN: Towards real-time object detection with region proposal networks." IEEE transactions on pattern analysis and machine intelligence 39.6 (2016): 1137-1149.

Initialization of Faster R-CNN	● Loading Pre-trained Weights of Faster R-CNN. ● Setting Parameters: setting image size, anchors such as RPN (Region Proposal Network), normalization standards, and the conversion of images into tensors.
Pre-Processing Input Image	● Resizing the Image: Adjust the input image to a fixed size to maintain consistency in input dimensions across different images. ● Normalizing Pixels: Scale pixel values to a range that improves convergence during training (see Appendix A for definition). ● Turning the Image into a Tensor
Forward Propagation (repeated for each layer)	● Applying Convolutions and Normalizations: Process the image through convolutional layers with batch normalization to extract features. ● Linearizing using Activation Functions: Use ReLU activation to introduce non-linearity into the feature maps, aiding in learning complex patterns. ● Region Proposal Network (RPN): Use the feature maps from initial layers to propose candidate object regions. RPN applies a set of anchors to predict the presence of objects and their boundaries. ● RoI Pooling: Apply Region of Interest (RoI) Pooling to normalize the features extracted from each candidate region to a consistent size, which is necessary for classification and bounding box regression in the subsequent layers.

Applying Special Layers for Feature Enhancement	● Using Detection Heads: Process the normalized features from each proposed region using fully connected layers to refine bounding box coordinates and classify the contents of each bounding box.
Post-Processing	● Non-Max Suppression (NMS): Apply NMS to reduce redundancy and overlap among the proposed bounding boxes, ensuring each detected object is represented by the best possible bounding box. ● Adjusting Bounding Box Coordinates: Scale the coordinates of the bounding boxes from the processed image size back to the original image dimensions.
Output	● Outputting Bounding Boxes: Return the final bounding boxes around detected objects. ● Printing Confidence Scores: Provide a confidence score for each detection to indicate the certainty of the model regarding each object's presence and classification. ● Labeling the Classes of the Objects: Assign class labels to each detected object based on the classification performed by the detection heads.

Table 4. Steps of training a Faster R-CNN algorithm

See Appendix B to find the Python code for training the Faster R-CNN algorithm with the custom dataset.

II.III. Adversarial Attacks

Adversarial Attacks mean intended attacks to manipulate an algorithm to produce wrong or misleading results. There are three ways of adversarial attacks: data poisoning, model extraction

attacks, and noise attacks. Data poisoning attacks refer to attacks that train an algorithm with adversarially manipulated images. These attacks are often targeted against open-source databases to affect large collections of algorithms and find cybersecurity gaps in these algorithms by sending inputs that are trained to be manipulated due to data poisoning. Model extraction is the method that's going to be used in this paper. When a pre-trained model is known, different methods can be used to grow the weight function of an object in an image by applying the opposite mathematics of gradient descent algorithms to its pixels. Fast Gradient Sign Method is an example of the model extraction attack technique. Finally, for attacks that don't have access to the detection model, noise (such as Gaussian Noise or Salt and Pepper Noise) can be added to the input image to confuse the network[5] and produce worse results, even though a difference can not be detected by the human eye.

II.III.I. Fast Gradient Sign Method

Fast Gradient Sign Method (FGSM) is a model extraction attack. FGSM works by exploiting the gradients of the model's loss function to create perturbations that maximize the loss. Perturbation means changes or patches on an input image by altering its pixels or changing the values of the pixels as shown in **Equation 2.** This method is particularly notable for its simplicity and efficiency in creating effective adversarial attacks.

$$x_{adv} = x + \epsilon \cdot sign(\nabla_x J(\theta, x, y)) \quad (2)$$

In this equation x_{adv} is the adversarial output, x is the original input, ϵ is the magnitude of perturbation, $J(\theta, x, y)$ is the loss function where θ is the predetermined weight of the model that's being attacked and y is the actual class of the input, and $sign(\nabla_x J(\theta, x, y))$ is a mathematical function extracting the sign of a real number to indicate if it is negative, positive, or zero. Additionally, ∇_x represents the gradient of the loss function, gathering many inputs, and can be re-written as:

$$\nabla_x J(\theta, x, y)) = \left[\frac{\delta J}{\delta x_1}, \frac{\delta J}{\delta x_2}, ..., \frac{\delta J}{\delta x_n}\right] \quad (3)$$

[5] Renkhoff, Justus, et al. "Exploring adversarial attacks on neural networks: An explainable approach." 2022 IEEE International Performance, Computing, and Communications Conference (IPCCC). IEEE, 2022.

Connecting **Equation 2** and **Equation 3**, it can be seen that applying FGSM is the exact opposite of applying gradient descent, but instead of changing the weights, this time perturbations are made to alter the input image pixels.

Word count: 1842

III. Methodology

As the paper aims to study the robustness of object detection algorithms to improve the safety of drivers and pedestrians in fully autonomous cars, the experiment will use a dataset of images that are taken from a car's view in rodes and will utilize two algorithms that are commonly in use for autonomous vehicle's object detection tasks (YOLOv5 and Faster R-CNN). Firstly, a dataset of 384 images labeled with cars, pedestrians, and traffic signs will be used to train YOLOv5 and Faster R-CNN. The loss functions and mAP scores of these models will be collected. Then two FGSM programs will be built to create perturbation images of an image to generate an adversarial image. The adversarial image and the input image will be tested using the object detection algorithms and the confidence levels and precision accuracies will be compared.

III. I. Images in the Dataset

The dataset is built by taking screenshots of Google Maps' "Street View" images. The images were randomly selected from the map to provide diversity for the model's feature extractions. **Image 1** is an example input image that is used to train the object detection algorithms.

Image 1. A sample input image taken by screenshotting Google Maps' Street View[6]

The input images were selected to have cars, pedestrians, and traffic signs and lights in them so that more labels with fewer images could be utilized to train an accurate model. All of the images had daylight conditions, which might cause the model to make wrong predictions for low light conditions. However, in all steps of the experiment, only high-resolution images with enough lighting were used.

III. II. Labeling the Dataset

After 384 images were selected, the images were uploaded to *Roboflow* to label them. Ten classes were created: car (class 1), pedestrian (class 2), green traffic light (*go-traffic-sign*) (class 3), red traffic light (*stop-traffic-sign*) (class 4), pedestrian (class 5), pedestrian-crossing (class 6), pedestrian-crossing-sign (class 7), speed-limit-20 (class 8), speed-limit-30 (class 9), and big vehicles (*vehicle-big*) (class 10). **Image 2** shows a sample input image being labeled in the *Roboflow* interface.

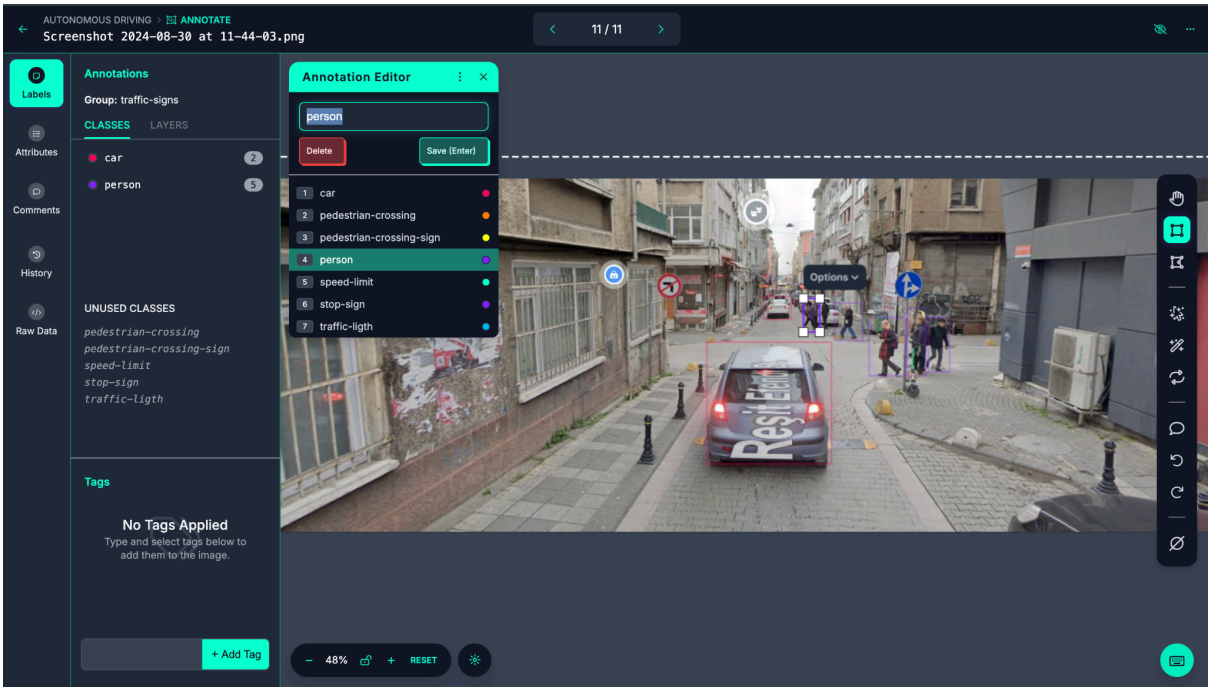


Image 2. Sample input image being labeled in Roboflow interface

[6] Google Maps. Country of Turkey. Map, Google, March 2023. https://maps.app.goo.gl/ZkoLYVKFLoKJnqCP7

III. III. Training YOLOv5

A YOLOv5 model will be trained by the algorithm mentioned in section II.I.I.I. with the Roboflow dataset. The code can be found in **Appendix B**. For training the YOLOv5 model, the Roboflow dataset will be integrated with the YOLOv5 PyTorch format, meaning the input labels will be in .txt format. The model was trained with a 416x416 pixel image size, batch size of 16 (meaning 16 images were processed together in one pass of forward and backward propagation through the model), epoch of 100 (meaning there were 100 training cycles), and weights gathered from yolov5 model. **Image 3** is the command line where these values were determined.

```
!python train.py --img 416 --batch 16 --epochs 100 --data {dataset.location}/data.yaml --weights yolov5s.pt --cache
```

Image 3. The code line for training the YOLOv5 model.

These values were selected specifically for enriching the accuracy of the system. The image size was chosen as 416x416 as 416 is divisible by 32 (which is the number used to divide the pixels for downsampling in this model) and creating a balance between speed and accuracy in training, the batch was chosen 16 as it is an ideal size to maximize GPU memory usage and it's not too large so it does not cause overfitting, and epochs were 100 to increase the accuracy and minimize the time of the training. Even though there were 100 batches, the model took over 7 hours to complete training indicating its complexity.

III. IV. Training Faster R-CNN

A Faster R-CNN model will be trained by the algorithm mentioned object in section II.I.I.II. with the Roboflow dataset. The code can be found in **Appendix B**. For training the Faster R-CNN model, the Roboflow dataset will be integrated with the COCO format, meaning the input labels this time will be in JSON format.

III. V. Generating FGSM Perturbation Images

For generating a Fast Gradient Sign Method perturbation image two programs are written in Python (see Appendix B), one for YOLOv5 and the other for Faster R-CNN. The program first gathers the trained model with its predetermined weights, classes, and loss functions. It will then

gather an input image. By performing the equations mentioned above in section II.III.I. It will generate an output adversarial image. After downloading the image, both models will be tested to classify original and adversarial images and their accuracy will be compared according to their confidence in detecting the right objects.

IV. Results

IV. I. Accuracy of YOLOv5 Model

After training the YOLOv5 model 157 layers, including convolutional layers, batch normalization layers, and activation layers were built, 7,034,398 parameters were created, and computational complexity was measured as 15.8 GFLOPs(Giga Floating Point Operations Per Second). Additionally, **Table 5** shows general and specific scores gained by the model.

Class	Images	Instances	Precision	Recall	mAP50	mAP50-95
all	32	286	0.853	0.529	0.529	0.300
car	32	164	0.879	0.793	0.849	0.422
green-light	32	20	0.789	0.700	0.733	0.336
pedestrian	32	48	0.754	0.510	0.558	0.218
pedestrian crossing	32	6	0.922	0.833	0.836	0.465
pedestrian crossing sign	32	11	0.973	0.727	0.728	0.424
speed limit 20	32	2	1.000	0.000	0.000	0.000
speed limit 30	32	5	1.000	0.666	0.995	0.511
red-light	32	15	0.646	0.200	0.177	0.0988
big-vehicle	32	15	0.717	0.333	0.456	0.225

Table 5. The statistics of the trained YOLOv5 model.

Class means the categories provided to the dataset and the model to detect as objects. Image is the number of images used to train the model for a specific class. The instance is the number of occurrences of the class in the set of images. Precision is the measure of the accuracy of the prediction according to the formula $\frac{true\ positive}{true\ positive + false\ positive}$ where true positive means that there was detection and it was correct and false positive means there was a detection but normally there was no object there. Recall is the measure of the model's ability to find all relevant instances using the formula $\frac{true\ positive}{true\ positive + false\ negative}$, in this formula false negative means that there was an object to detect but the model did not detect anything. mAP50 is the mean average precision (see Appendix A for definition) at 50% Intersection over Union(IoU). This metric evaluates the model's accuracy by calculating the average precision at 50% overlap between the predicted bounding box and the ground truth. mAP50-95 is the average mAP value calculated at 50% to 95% IoU thresholds with 5% increments. It provides a more comprehensive measure of model performance across different levels of detection difficulty. Looking at **Table 5** it can be seen that while the overall mAP score was moderate due to not having enough training images for all classes, the scores for car and pedestrian crossing were much higher.

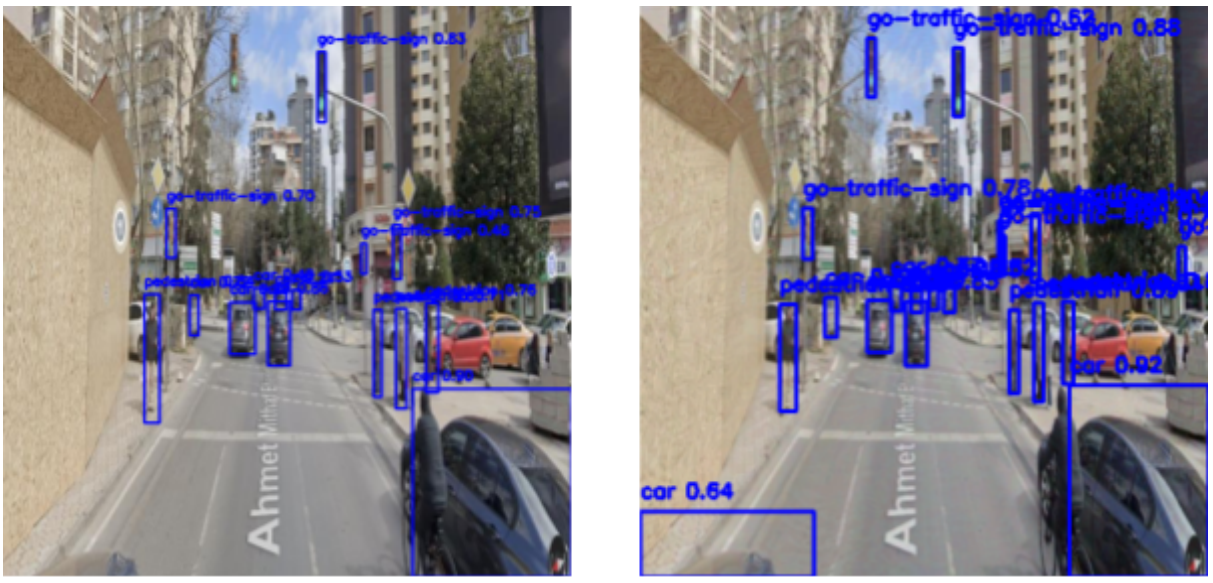
IV.I.I. Robustness of YOLOv5

To analyse the robustness of the YOLOv5 model a Fast Gradient Sign Method was used. The code for the model can be found in Appendix B. **Image 4** shows the original versus the adversarial attack.

Original Image Adversarial Image

Image 4. Original and Adversarial images after a FGSM perturbation change.

It can be seen that the changes in the images are almost invisible to the human eye. After the image was modified to adversarially attack the model, the model was tested (see Appendix B for the testing code). **Image 5** shows the original and adversarial images after being tested by the YOLOv5 model. See Appendix C to find the high-resolution version of **Image 5**.



Original Image Adversarial Image

Image 5. Original and Adversarial images after being tested by the YOLOv5 model.

After testing the image it can be seen that the adversarial image, even though the changes are not visible to the human eye, makes the model detect false positives. Firstly, the adversarial image detected a car with 0.64 confidence while it did not detect anything at that spot in the original image. Secondly, in the middle right corner, the adversarial image caused the model to detect an object on the tree, but actually, there were no objects that could be categorized as one of the classes that appeared there. Even though the algorithm made false positives, there were almost no false negatives nor drops in confidence scores in true positives.

IV. II. Accuracy of Faster R-CNN Model

The Faster R-CNN model was trained in three epochs. Each epoch's loss function is shown in **Table 6**.

Epoch	Loss Function
1	0.9819401502609253
2	0.9940546154975891
3	0.8315337896347046
4	0.8252518773078918
5	0.8702759742736816
6	0.674719512462616
7	0.7058202624320984
8	0.48121678829193115
9	0.5538797378540039
10	0.29297319054603577

Table 6. Loss functions of Faster R-CNN model.

The decreasing trend in the loss function shows that the model learned in every epoch. Especially the sharp decrease in the final epoch pointed out the high convergence of the model. Furthermore, the same mAP scores were collected from the Faster R-CNN model as shown in **Table 7**.

Average Precision (AP)	IoU=0.50:0.95	Area = all	maxDets=100	= 0.466
Average Precision (AP)	IoU=0.50	Area = all	maxDets=100	= 0.761
Average Precision (AP)	IoU=0.75	Area =all	maxDets=100	= 0.520
Average Precision (AP)	IoU=0.50:0.95	Area = small	maxDets=100	= 0.355
Average Precision (AP)	IoU=0.50:0.95	Area = medium	maxDets=100	= 0.674
Average Precision (AP)	IoU=0.50:0.95	Area = large	maxDets=100	= 0.756
Average Recall (AR)	IoU=0.50:0.95	Area = all	maxDets=1	= 0.310
Average Recall (AR)	IoU=0.50:0.95	Area = all	maxDets=10	= 0.533
Average Recall (AR)	IoU=0.50:0.95	Area = all	maxDets=100	= 0.536
Average Recall (AR)	IoU=0.50:0.95	Area = small	maxDets=100	= 0.444
Average Recall (AR)	IoU=0.50:0.95	Area = medium	maxDets=100	= 0.731
Average Recall (AR)	IoU=0.50:0.95	Area = large	maxDets=100	= 0.791

Table 7. The AP and AR scores of the Faster R-CNN Model.

The code for evaluating Faster R-CNN can be found in Appendix B.

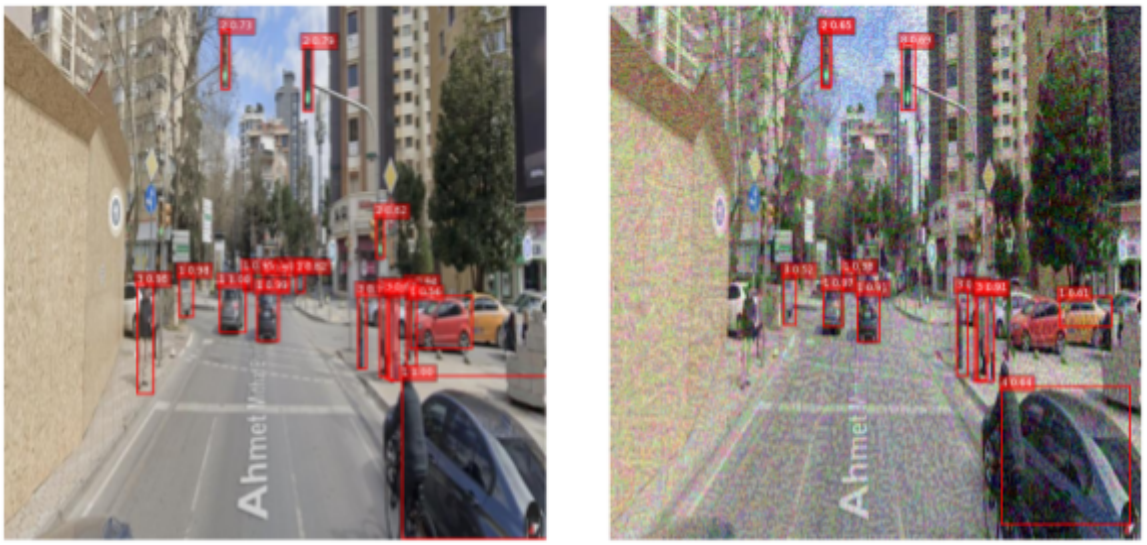
IV.I.II. Robustness of Faster R-CNN

To determine the Faster R-CNN model's robustness, an FGSM program was utilized that took all the dataset images and added perturbation to them with 0.1 epsilon (epsilon is the ϵ value see **Equation 2**) value and downloaded them to Google Drive. See Appendix B for the FGSM image-generating code. **Image 5.** shows the perturbation and original image.

Original Image **Adversarial Image**

Image 5. Comparison between the original and adversarial images.

While there appears some noise in the image, it's still possible to clearly see the objects with the human eye. To assess the robustness of the Faster R-CNN model both images were tested. **Image 6** illustrates the outputs. The image quality decreased when creating the illustration, so see Appendix C for higher-quality images.



Original Image **Adversarial Image**

Image 6. Comparison of confidence levels of original and adversarial images.

As can be inferred from the image, applying the Fast Gradient Sign Method to an original image decreased the accuracy and confidence of the model and even produced highly confident

misclassification results. To analyze deeper, it's plausible that the algorithm did not see the pedestrian on the left and identified a car (a class 1 object that had 1.00 accuracy in the original picture) as a pedestrian crossing line(class 4 object) with 0.644 confidence. These exemplify how the model made false negatives. Moreover, the green traffic light (a class 2 object that had 0.79 accuracy in the original image) was misclassified as a red traffic light (a class 8 object) with 0.69 confidence.

IV.III. Robustness Comparison

Both algorithms showed clear signs of misclassification. If such an attack were performed in a real fully autonomous vehicle, the car might even cause traffic accidents by not seeing the pedestrians and misclassifying traffic lights. Comparing the robustness of the two algorithms, it can be seen that YOLOv5 made few to no false negative errors while most errors of Faster RCNN were false negatives. On the other hand, YOLOv5 lost less confidence in true positive detections compared to Faster R-CNN. Nonetheless, Faster R-CNN made few to no false positive errors and almost all of its bounding boxes were perfectly capturing an object (even though oftentimes the object was not accurately classified).

IV.IV. Data Analysis

Comparing the original trained YOLOv5 and Faster R-CNN models as a whole from **Table 6** and **Table 7**, it can be seen that Faster R-CNN's mAP@0.50-0.95 score (0.466) was higher than YOLOv5's (0.300). And Faster R-CNN had higher AP at IoU=0.75 (0.520) compared to YOLOv5 (0.300). This comparison showed that Faster R-CNN had higher accuracy in stricter conditions. Faster R-CNN performed consistently well across small, medium, and large objects with APs of 0.355, 0.674, and 0.756, respectively, while YOLOv5's accuracy was variant on different classes.

Additionally, the adversarial image of Faster R-CNN had much more visible noise than the one of YOLOv5. This suggests that Faster R-CNN had a more complex model and required more perturbations to fool.

V. Limitations

The research compared single-stage and two-stage object detection algorithms in terms of their robustness against adversarial attacks. Both models were trained on a small dataset with limited complexity. Due to not having enough samples for each class and having too many classes, some classes had greater precision while others were not even detected in most cases. Moreover, the model used static images only. On the other hand, novel papers such as Amirhosein, et al.[7]'s recent discoveries show that adversarially manipulating dynamic object detection algorithms is possible.

Furthermore, this experiment can be applied to real life by printing out adversarial images that can fool object detection algorithms and observing how decision-making algorithms of fully autonomous vehicles operate against them.

VI. Conclusion

The experiment conducted in this paper showed how applying the model extraction method of adversarial attacks, specifically the FGSM causes the object detection algorithm to misclassify with and produce high confidence scores for false positive and false negative detections. Even though when training the Faster R-CNN object detection model the model was faster and had a greater mean average precision score, its confidence level dropped significantly more compared to YOLOv5 when detecting true positives. Additionally, Faster R-CNN's two-stage object detection model helped its bounding boxes to only be placed around existing objects and only make classification errors or not catch any object at all. On the other hand, the YOLOv5 algorithm only made additional wrong detections while it held high confidence in true positive detections. YOLOv5 algorithm's adversarial attack was harder to spot both being an advantage that it can not fool the system as much and a disadvantage that the model is less complex.

[7] Chahe, Amirhosein, et al. "Dynamic adversarial attacks on autonomous driving systems." arXiv preprint arXiv:2312.06701 (2023).

Given the results of the experiment, it can be concluded that both algorithms have weaknesses and strengths against adversarial attacks. In situations where an autonomous vehicle was to choose a model, Faster R-CNN would be preferable if the vehicle could hold higher complexity; however, in a situation where the vehicle would have to trust and find a safe stop to not hit any object, YOLOv5 would be more dependable as it did not make any false negatives, but only false positives giving it an advantage to find stops where there are no objects.

VII. Works Cited

Abdulghani, Abdulghani Mawlood, and Gonca Gökçe Menekşe Dalveren. “Moving Object Detection in Video with Algorithms Yolo and Faster R-CNN in Different Conditions.” European Journal of Science and Technology, 2022, https://doi.org/10.31590/ejosat.1013049.

Chahe, Amirhosein, et al. "Dynamic adversarial attacks on autonomous driving systems." arXiv preprint arXiv:2312.06701 (2023).

Eykholt, Kevin, et al. "Robust physical-world attacks on deep learning visual classification." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

Jocher, Glenn, et al. "ultralytics/yolov5: v7. 0-yolov5 sota real-time instance segmentation." Zenodo (2022).

Mahendrakar, Trupti, et al. “Performance Study of yolov5 and Faster R-CNN for Autonomous Navigation around Non-Cooperative Targets.” 2022 IEEE Aerospace Conference (AERO), 2022, https://doi.org/10.1109/aero53065.2022.9843537.

OpenAI. "ChatGPT-3.5." ChatGPT, 2024. OpenAI.

Ren, Shaoqing, et al. "Faster R-CNN: Towards real-time object detection with region proposal networks." IEEE transactions on pattern analysis and machine intelligence 39.6 (2016): 1137-1149.

Renkhoff, Justus, et al. "Exploring adversarial attacks on neural networks: An explainable approach." 2022 IEEE International Performance, Computing, and Communications Conference (IPCCC). IEEE, 2022.

Xu, Kaidi, et al. "Adversarial t-shirt! evading person detectors in a physical world." Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16. Springer International Publishing, 2020.

Yusro, Muhamad Munawar, et al. “Comparison of Faster R-CNN and Yolov5 for Overlapping Objects Recognition.” Baghdad Science Journal, 20 Nov. 2022, https://doi.org/10.21123/bsj.2022.7243.

VIII. Appendix

A. Definitions

Perturbation: Perturbation refers to small changes made to input data (like an image) intended to trick a machine learning model into making mistakes. These changes are usually so slight that humans can't notice them, but they can cause the model to give the wrong answer.

Tensor: A tensor is a term for a data container that can hold numbers in multiple dimensions. It's similar to a spreadsheet but can have more layers and be more complex. It's used a lot in machine learning to handle and process data.

Normalization: Normalisation is the process of adjusting data to fit within a certain range, usually to make calculations more straightforward and stable. Machine learning often involves changing input values (like pixels in an image) to make them all fall between 0 and 1, making it easier for the model to learn effectively.

Sigmoid: The Sigmoid function is used in machine learning to turn any number into a value between 0 and 1, making it useful when the model needs to decide between two things (like yes/no questions).

SiLU (Sigmoid Linear Unit): The SiLU function is a newer version used for similar purposes but can keep information flowing through the model better when the numbers involved are negative or very large, potentially leading to better learning by the model.

Mean Average Precision: "Mean Average Precision (mAP) is a metric used to evaluate how accurately a model can identify and locate objects within images. It averages the precision (the correctness of the objects it identifies) across all types of objects the model can detect, providing a single score to summarise its overall effectiveness."[8]

[8] OpenAI. "ChatGPT-3.5." ChatGPT, 2024. OpenAI.

B. Code

YOLOv5 Training Code in Python:

```python
# -*- coding: utf-8 -*-
"""YOLO_EE.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1kNejb3jaI2ipiJ30ceFlFKZwAqPRXQTJ
"""

# Commented out IPython magic to ensure Python compatibility.
!git clone https://github.com/ultralytics/yolov5
# %cd yolov5
# %pip install -qr requirements.txt
# %pip install -q roboflow

import torch
import os
from IPython.display import Image, clear_output

print(f"Setup complete. Using torch {torch.__version__} ({torch.cuda.get_device_properties(0).name if torch.cuda.is_available() else 'CPU'})")

os.environ["DATASET_DIRECTORY"] = "/content/datasets"

!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="RSULTlbgzcou9wRPFxvB")
project = rf.workspace("eextended-essay").project("extended-essay-o1oqj")
version = project.version(1)
dataset = version.download("yolov5")

!python train.py --img 416 --batch 16 --epochs 100 --data {dataset.location}/data.yaml --weights yolov5s.pt --cache

# Commented out IPython magic to ensure Python compatibility.
# Look at training curves in tensorboard:
# %load_ext tensorboard
# %tensorboard --logdir output

import glob
from IPython.display import Image, display

i=0
for imageName in glob.glob('/content/yolov5/runs/detect/exp/*.jpg'):
  i+=1
  if i<10:
    display(Image(filename=imageName))
    print("\n")

from google.colab import files
files.download('./runs/train/exp/weights/best.pt')

```

```python

from google.colab import drive
drive.mount('/content/drive')

import shutil

# Path to the trained model weights
src_model_path = 'runs/train/exp/weights/best.pt'
dest_model_path = 'yolov5_model.pt'

# Copy the model to the desired location
shutil.copy(src_model_path, dest_model_path)

print(f"Model saved to {dest_model_path}")

!cp yolov5_model.pt /content/drive/MyDrive/

```

Faster R-CNN Training Code in Python:

```python
# -*- coding: utf-8 -*-
"""faster_rcnn_final.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1y_bMwqQF0Pyd-Arz1X1aHU2g7dOSlQbU
"""

# Install required libraries
!pip install torch torchvision
!pip install roboflow
!pip install pycocotools
!pip install opencv-python

!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="RSULTlbgzcou9wRPFxvB")
project = rf.workspace("eextended-essay").project("extended-essay-o1oqj")
version = project.version(2)
dataset = version.download("coco")

root_dir = "/content/extended-essay-2/train"  # Path to the folder containing images
ann_file = "/content/extended-essay-2/train/_annotations.coco.json"  # Path to the annotation file

!pip install opencv-python
import os
import torch
from torchvision import datasets, transforms
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator
from torch.utils.data import DataLoader
from pycocotools.coco import COCO
import cv2 # Import the OpenCV library

class CustomCocoDataset(torch.utils.data.Dataset):
    def __init__(self, root, annFile, transforms=None):
        self.root = root
        self.coco = COCO(annFile)
        self.ids = list(self.coco.imgToAnns.keys())
        self.transforms = transforms

    def __getitem__(self, index):
        coco = self.coco
        img_id = self.ids[index]
        ann_ids = coco.getAnnIds(imgIds=img_id)
        anns = coco.loadAnns(ann_ids)
        path = coco.loadImgs(img_id)[0]['file_name']

        img = cv2.imread(os.path.join(self.root, path)) # Now cv2 is defined
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        if self.transforms is not None:
            img = self.transforms(img)

        num_objs = len(anns)
        boxes = []
        labels = []
        for i in range(num_objs):
            xmin = anns[i]['bbox'][0]
            ymin = anns[i]['bbox'][1]
            xmax = xmin + anns[i]['bbox'][2]
            ymax = ymin + anns[i]['bbox'][3]
            boxes.append([xmin, ymin, xmax, ymax])
            labels.append(anns[i]['category_id'])

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels

        return img, target

    def __len__(self):
        return len(self.ids)

# Define your dataset and dataloader
try:
    dataset = CustomCocoDataset(root_dir, ann_file, transforms=transforms.ToTensor())
    data_loader = DataLoader(dataset, batch_size=4, shuffle=True, num_workers=4, collate_fn=lambda x: tuple(zip(*x)))
except Exception as e:
    print(f"Error loading dataset: {e}")
```

```python

from torchvision.models.detection import fasterrcnn_resnet50_fpn

# Load a pre-trained Faster R-CNN model
model = fasterrcnn_resnet50_fpn(pretrained=True)

# Replace the classifier head with a new one for your specific number of classes
num_classes = len(dataset.coco.getCatIds()) + 1  # Add 1 for the background class
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

import torch.optim as optim

# Move model to the GPU if available
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)

# Define the optimizer
params = [p for p in model.parameters() if p.requires_grad]
optimizer = optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    for images, targets in data_loader:
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs} - Loss: {losses.item()}")

from google.colab import drive
drive.mount('/content/drive')

# Save the trained model
torch.save(model.state_dict(), 'faster_rcnn_model.pth')

# Optionally, save to Google Drive
!cp faster_rcnn_model.pth /content/drive/MyDrive/
```

Evaluating Faster R-CNN

```python
# -*- coding: utf-8 -*-
"""faster_rcnn results.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1TLZfl_UnStvPWFlG8t3MPFy0EkbA_m_s
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="RSULTlbgzcou9wRPFxvB")
project = rf.workspace("eextended-essay").project("extended-essay-o1oqj")
version = project.version(2)
dataset = version.download("coco")

root_dir = "/content/extended-essay-2/train"  # Path to the folder containing images
ann_file = "/content/extended-essay-2/train/_annotations.coco.json"  # Path to the annotation file

import torch
import torchvision
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torch.utils.data import DataLoader
import torchvision.transforms as T
from PIL import Image
import os
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from tqdm import tqdm


```

```python
# Custom COCO Dataset
class CocoDataset(torch.utils.data.Dataset):
    def __init__(self, root, ann_file, transforms=None):
        self.root = root
        self.coco = COCO(ann_file)
        self.ids = list(self.coco.imgs.keys())
        self.transforms = transforms

    def __getitem__(self, index):
        coco = self.coco
        img_id = self.ids[index]
        ann_ids = coco.getAnnIds(imgIds=img_id)
        anns = coco.loadAnns(ann_ids)
        path = coco.loadImgs(img_id)[0]['file_name']

        img = Image.open(os.path.join(self.root, path)).convert('RGB')
        if self.transforms:
            img = self.transforms(img)

        # Prepare the annotation in the format the model expects
        boxes = []
        labels = []
        for obj in anns:
            xmin, ymin, w, h = obj['bbox']
            boxes.append([xmin, ymin, xmin + w, ymin + h])
            labels.append(obj['category_id'])

        target = {}
        target['boxes'] = torch.as_tensor(boxes, dtype=torch.float32)
        target['labels'] = torch.tensor(labels, dtype=torch.int64)
        target['image_id'] = torch.tensor([img_id])  # Include image_id in the target

        print(f"Created target: {target}")  # Debugging: print the target dictionary

        return img, target

    def __len__(self):
        return len(self.ids)


```

```python
# Load the dataset
root_dir = "/content/extended-essay-2/train"
ann_file = "/content/extended-essay-2/train/_annotations.coco.json"

# Initialize dataset and data loader
dataset = CocoDataset(
    root=root_dir,
    ann_file=ann_file,
    transforms=T.Compose([T.ToTensor()])
)

data_loader = DataLoader(
    dataset,
    batch_size=4,
    shuffle=False,
    num_workers=2,
    collate_fn=lambda x: tuple(zip(*x))
)

# Load the model
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
num_classes = 11  # Number of classes including background

model = fasterrcnn_resnet50_fpn(pretrained=False)
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
model.load_state_dict(torch.load('/content/drive/My Drive/faster_rcnn_model.pth'))
model.to(device)
model.eval()
```

```python

# Evaluation Function
def evaluate(model, data_loader, device):
    print("Starting evaluation...")  # To confirm that the function is being called
    model.eval()
    coco_gt = COCO("/content/extended-essay-2/train/_annotations.coco.json")  # Ground truth COCO

    results = []

    for batch_idx, (images, targets) in enumerate(tqdm(data_loader)):
        print(f"Processing batch {batch_idx + 1}/{len(data_loader)}")  # Print current batch

        images = list(img.to(device) for img in images)
        outputs = model(images)

        for i, output in enumerate(outputs):
            print(f"Target {i}: {targets[i]}")  # Debugging target structure

            image_id = targets[i].get('image_id')
            if image_id is None:
                raise ValueError(f"Image ID not found in target: {targets[i]}")

            image_id = image_id.item()
            boxes = output['boxes'].detach().cpu().numpy()  # Use detach() before converting to numpy
            scores = output['scores'].detach().cpu().numpy()  # Use detach() before converting to numpy
            labels = output['labels'].detach().cpu().numpy()  # Use detach() before converting to numpy

            # Convert boxes to COCO format
            boxes[:, 2:] -= boxes[:, :2]

            for box, score, label in zip(boxes, scores, labels):
                result = {
                    "image_id": image_id,
                    "category_id": int(label),
                    "bbox": box.tolist(),
                    "score": float(score)
                }
                results.append(result)

        print(f"Completed processing batch {batch_idx + 1}/{len(data_loader)}")  # Confirm batch completion

    print("Finished processing all batches, now evaluating...")  # Before COCO evaluation
    # Load results into COCO format
    coco_dt = coco_gt.loadRes(results)

    # Run COCO evaluation
    coco_eval = COCOeval(coco_gt, coco_dt, iouType='bbox')
    coco_eval.evaluate()
    coco_eval.accumulate()
    coco_eval.summarize()
    print("Evaluation complete.")  # Confirm completion of evaluation
```

```python

# Run the evaluation
evaluate(model, data_loader, device)

```

Testing YOLOv5

```python
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Clone YOLOv5 repository (if not already cloned)
!git clone https://github.com/ultralytics/yolov5
%cd yolov5
%pip install -qr requirements.txt  # Install dependencies

import torch
from PIL import Image
import torchvision.transforms as T
import matplotlib.pyplot as plt
import cv2
from models.common import DetectMultiBackend  # Import after cloning YOLOv5
from utils.general import non_max_suppression
import numpy as np
```

```python
# Helper function to scale coordinates
def scale_coords(img1_shape, coords, img0_shape, ratio_pad=None):
    if ratio_pad is None:  # calculate from img0_shape
        gain = min(img1_shape[0] / img0_shape[0], img1_shape[1] / img0_shape[1])  # gain  = old / new
        pad = (img1_shape[1] - img0_shape[1] * gain) / 2, (img1_shape[0] - img0_shape[0] * gain) / 2  # wh padding
    else:
        gain = ratio_pad[0]
        pad = ratio_pad[1]

    coords[:, [0, 2]] -= pad[0]  # x padding
    coords[:, [1, 3]] -= pad[1]  # y padding
    coords[:, :4] /= gain
    coords[:, :4] = torch.clamp(coords[:, :4], min=0, max=img0_shape[1])  # clip coords
    return coords

# Load model and set it to evaluation mode
print("Loading model...")
model_path = '/content/drive/My Drive/yolov5_model.pt'
model = DetectMultiBackend(weights=model_path)
model.eval()
print("Model loaded successfully.")

# Load and preprocess the adversarial image
image_path = '/content/original.png'
image = Image.open(image_path).convert('RGB')

transform = T.Compose([
    T.Resize((416, 416)),  # Resize image to YOLOv5 input size
    T.ToTensor(),
])

input_tensor = transform(image).unsqueeze(0)
print("Image preprocessed.")
```

```
# Forward pass through the model
output = model(input_tensor)
print("Model inference complete.")

# Apply Non-Max Suppression (NMS) to filter boxes
conf_thresh = 0.25  # Confidence threshold
iou_thresh = 0.45  # IoU threshold for NMS
pred = non_max_suppression(output, conf_thresh, iou_thresh)
print("Non-Max Suppression complete.")

# Rescale boxes to original image size
img1_shape = input_tensor.shape[2:]  # Resized image shape
img0_shape = image.size[::-1]  # Original image shape
pred[0][:, :4] = scale_coords(img1_shape, pred[0][:, :4], img0_shape).round()

# Load image with OpenCV for drawing boxes
img_np = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)

# Draw boxes on the image using OpenCV
for det in pred:  # detections per image
    if len(det):
        for *xyxy, conf, cls in det:
            label = f'{model.names[int(cls)]} {conf:.2f}'
            x1, y1, x2, y2 = map(int, xyxy)
            cv2.rectangle(img_np, (x1, y1), (x2, y2), color=(255, 0, 0), thickness=2)
            cv2.putText(img_np, label, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), thickness=2)
print("Bounding boxes drawn.")

# Convert back to RGB for plotting in matplotlib
img_np = cv2.cvtColor(img_np, cv2.COLOR_BGR2RGB)
```

```
# Save the resulting image with detections
output_path = '/content/detected_adversarial_image.png'
Image.fromarray(img_np).save(output_path)
print(f"Detected image saved to {output_path}")

# Display the resulting image with detections
plt.imshow(img_np)
plt.title('Detected Objects in Adversarial Image')
plt.show()
```

Testing Faster-RCNN

```
from google.colab import drive
drive.mount('/content/drive')
```

```python
import torch
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
from pathlib import Path

# Define paths
model_path = '/content/drive/My Drive/faster_rcnn_model.pth'  # Path to trained Faster R-CNN model weights
image_path = 'sample.png'  # Path to the image
output_path = 'output_sample.png'  # Path to save the output image

# Load the Faster R-CNN model architecture
model = fasterrcnn_resnet50_fpn(pretrained=False)

# Modify the classifier head to match the number of classes in your model
num_classes = 11  # Update this to match the number of classes in your trained model
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

# Load the trained weights into the model
state_dict = torch.load(model_path, map_location=torch.device('cpu'))
model.load_state_dict(state_dict)
model.eval()

# Define the image transformations
transform = transforms.Compose([
    transforms.ToTensor(),
])

# Load and transform the image
image = Image.open(image_path).convert("RGB")
image_tensor = transform(image).unsqueeze(0)  # Add batch dimension
```

```python
# Perform inference
with torch.no_grad():
    outputs = model(image_tensor)

# Extract results
boxes = outputs[0]['boxes']
labels = outputs[0]['labels']
scores = outputs[0]['scores']

# Display results
plt.figure(figsize=(12, 12))
plt.imshow(image)
ax = plt.gca()

# Draw bounding boxes and labels
for box, label, score in zip(boxes, labels, scores):
    if score >= 0.5:  # Adjust the threshold as needed
        xmin, ymin, xmax, ymax = box
        ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin, edgecolor='red', facecolor='none', linewid
        ax.text(xmin, ymin, f'{label.item()} {score:.2f}', fontsize=12, color='white', bbox=dict(facecolor='red', alp

# Save and display the output image
plt.axis('off')
plt.savefig(output_path, bbox_inches='tight')
plt.show()

print(f"Output image saved as {output_path}")
```

Generating Adversarial Images with FGSM for YOLOv5

```python
from google.colab import drive
drive.mount('/content/drive')
```

```python
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

import torch
from PIL import Image
import torchvision.transforms as T
import numpy as np
import matplotlib.pyplot as plt
from models.common import DetectMultiBackend  # Import after cloning YOLOv5

# Load model and set it to evaluation mode
model_path = '/content/drive/My Drive/yolov5_model.pt'
model = DetectMultiBackend(weights=model_path)
model.eval()

# Load and preprocess the image
image_path = '/content/original.png'
image = Image.open(image_path).convert('RGB')

transform = T.Compose([
    T.Resize((416, 416)),
    T.ToTensor(),
])

input_tensor = transform(image).unsqueeze(0)
input_tensor.requires_grad = True

# Forward pass through the model
output = model(input_tensor)
output = output[0]  # Extract the relevant part of the output
```

```python
objectness_score = output[..., 4]  # Example for objectness score
class_scores = output[..., 5:]  # Example for class scores

# Combine or focus on a specific part of the output
loss = torch.mean(objectness_score)  # Example: focus on reducing objectness

model.zero_grad()
loss.backward()

# Generate FGSM adversarial image
epsilon = 0.01
perturbation = epsilon * input_tensor.grad.sign()
adversarial_image = input_tensor + perturbation
adversarial_image = torch.clamp(adversarial_image, 0, 1)

# Convert adversarial tensor to image and save it
adversarial_image_np = adversarial_image.squeeze().detach().numpy().transpose(1, 2, 0)
adversarial_image_pil = Image.fromarray((adversarial_image_np * 255).astype(np.uint8))
output_path = '/content/adversarial_image.png'
adversarial_image_pil.save(output_path)

# Show the adversarial image
plt.imshow(adversarial_image_np)
plt.title('Adversarial Image')
plt.show()
```

Generating Adversarial Images with FGSM for Faster R-CNN

```python
from google.colab import drive
drive.mount('/content/drive')
```

```python
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="RSULTlbgzcou9wRPFxvB")
project = rf.workspace("eextended-essay").project("extended-essay-o1oqj")
version = project.version(2)
dataset = version.download("coco")

root_dir = "/content/extended-essay-2/train"  # Path to the folder containing images
ann_file = "/content/extended-essay-2/train/_annotations.coco.json"  # Path to the annotation file
```

```python
# Step 1: Setup the Environment
import torch
import torchvision
from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torch.utils.data import DataLoader
import torchvision.transforms as T
from PIL import Image
import os
import numpy as np
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from tqdm import tqdm
```

```python
# Custom COCO Dataset
class CocoDataset(torch.utils.data.Dataset):
    def __init__(self, root, ann_file, transforms=None):
        self.root = root
        self.coco = COCO(ann_file)
        self.ids = list(self.coco.imgs.keys())
        self.transforms = transforms

    def __getitem__(self, index):
        coco = self.coco
        img_id = self.ids[index]
        ann_ids = coco.getAnnIds(imgIds=img_id)
        anns = coco.loadAnns(ann_ids)
        path = coco.loadImgs(img_id)[0]['file_name']

        img = Image.open(os.path.join(self.root, path)).convert('RGB')
        if self.transforms:
            img = self.transforms(img)

        # Prepare the annotation in the format the model expects
        boxes = []
        labels = []
        for obj in anns:
            xmin, ymin, w, h = obj['bbox']
            boxes.append([xmin, ymin, xmin + w, ymin + h])
            labels.append(obj['category_id'])

        target = {}
        target['boxes'] = torch.as_tensor(boxes, dtype=torch.float32)
        target['labels'] = torch.tensor(labels, dtype=torch.int64)
        target['image_id'] = torch.tensor([img_id])  # Include image_id in the target

        return img, target
```

```
    def __len__(self):
        return len(self.ids)

# Step 2: Load the Dataset
root_dir = "/content/extended-essay-2/train"
ann_file = "/content/extended-essay-2/train/_annotations.coco.json"

dataset = CocoDataset(
    root=root_dir,
    ann_file=ann_file,
    transforms=T.Compose([T.ToTensor()])
)

data_loader = DataLoader(
    dataset,
    batch_size=1,  # Use a batch size of 1 for adversarial generation
    shuffle=False,
    num_workers=2,
    collate_fn=lambda x: tuple(zip(*x))
)

# Step 3: Load the Pre-trained Model
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
num_classes = 11  # Number of classes including background

model = fasterrcnn_resnet50_fpn(pretrained=False)
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
model.load_state_dict(torch.load('/content/drive/My Drive/faster_rcnn_model.pth'))
model.to(device)
model.eval()
```

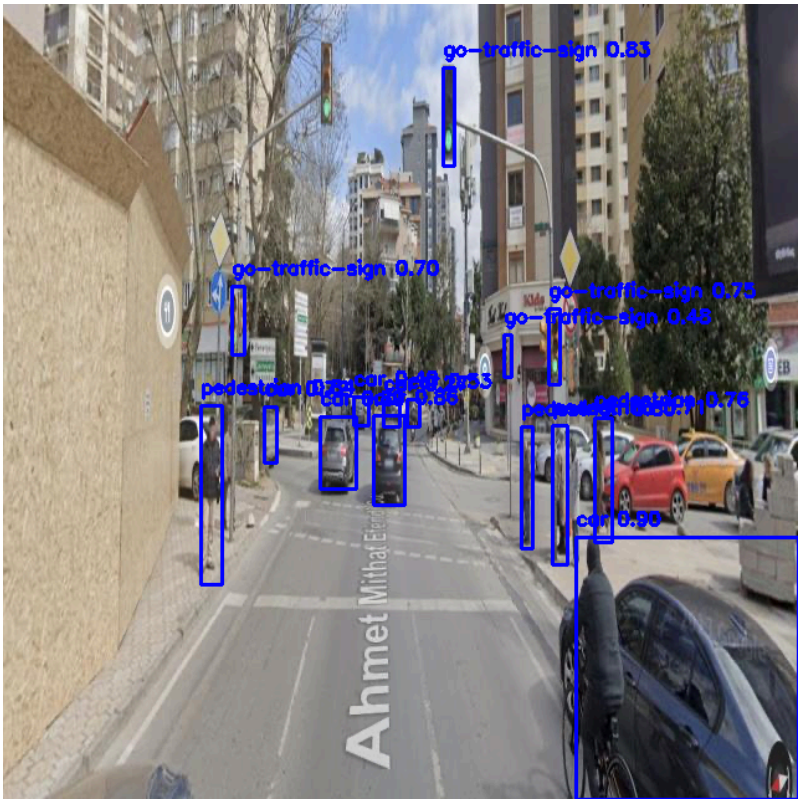
```
# Step 4: Define FGSM Attack Method
def fgsm_attack(image, epsilon, data_grad):
    # Collect the sign of the gradients
    sign_data_grad = data_grad.sign()
    # Create the perturbed image by adjusting each pixel of the input image
    perturbed_image = image + epsilon * sign_data_grad
    # Adding clipping to maintain [0,1] range
    perturbed_image = torch.clamp(perturbed_image, 0, 1)
    return perturbed_image
```

```
# Step 5: Generate Adversarial Examples
def generate_adversarial_examples(model, data_loader, device, epsilon):
    model.eval()  # Ensure the model is in evaluation mode initially
    adversarial_examples = []

    for images, targets in data_loader:
        images = list(img.to(device) for img in images)

        # Set requires_grad attribute of tensor. Important for Attack
        for img in images:
            img.requires_grad = True

        # Switch model to training mode to calculate loss
        model.train()

        # Forward pass the data through the model
        loss_dict = model(images, targets)

        # Calculate the total loss
        losses = sum(loss for loss in loss_dict.values())

        # Zero all existing gradients
        model.zero_grad()

        # Backward pass to calculate gradients
        losses.backward()

        # Collect the gradient of the input image
        data_grad = images[0].grad.data

        # Call FGSM Attack
        perturbed_image = fgsm_attack(images[0], epsilon, data_grad)
```

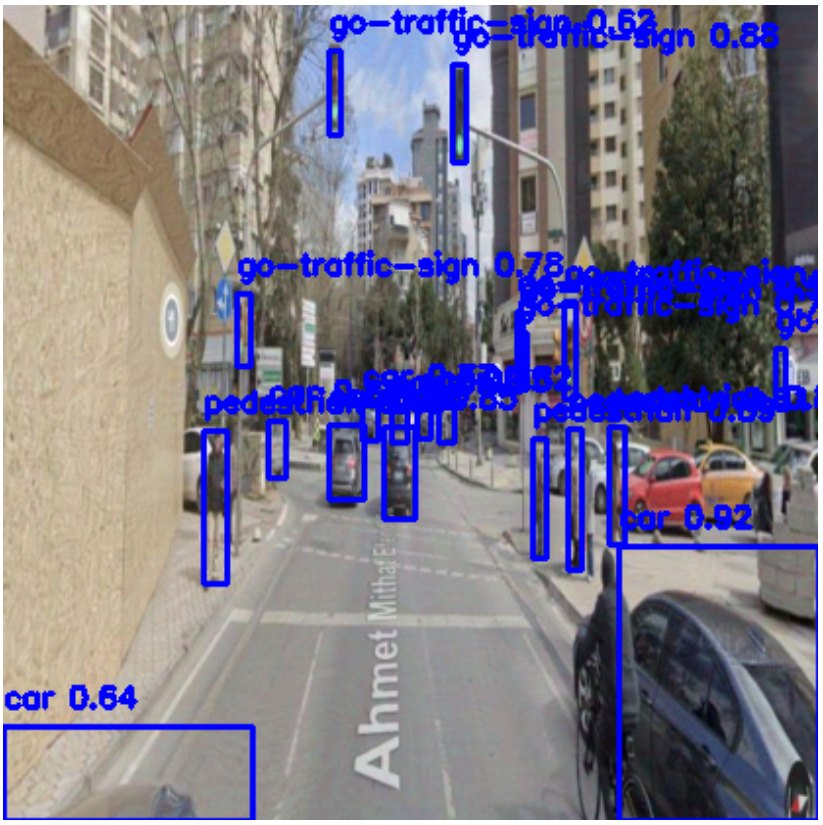
```
        # Save the adversarial example for later
        adversarial_examples.append(perturbed_image.cpu().detach().numpy())

        # Switch back to evaluation mode
        model.eval()

    return adversarial_examples


# Step 6: Set Epsilon and Generate Examples
epsilon = 0.1  # Adjust epsilon as needed
adversarial_examples = generate_adversarial_examples(model, data_loader, device, epsilon)

# Step 7: Save or Display Adversarial Images
import matplotlib.pyplot as plt

# Display a few adversarial examples
for idx in range(min(5, len(adversarial_examples))):
    img = adversarial_examples[idx]
    img = np.transpose(img, (1, 2, 0))  # Convert from CxHxW to HxWxC
    plt.figure()
    plt.imshow(img)
    plt.title(f"Adversarial Example {idx + 1} with epsilon {epsilon}")
    plt.show()

# Optionally, save adversarial examples
for idx, adv_img in enumerate(adversarial_examples):
    adv_img = (adv_img * 255).astype(np.uint8)  # Convert to uint8 for saving
    adv_img = np.transpose(adv_img, (1, 2, 0))  # Convert from CxHxW to HxWxC
    adv_img_pil = Image.fromarray(adv_img)
    adv_img_pil.save(f"/content/drive/My Drive/adversarial_example_{idx}.png")
```

C. Images

Original Image of Faster R-CNN testing:

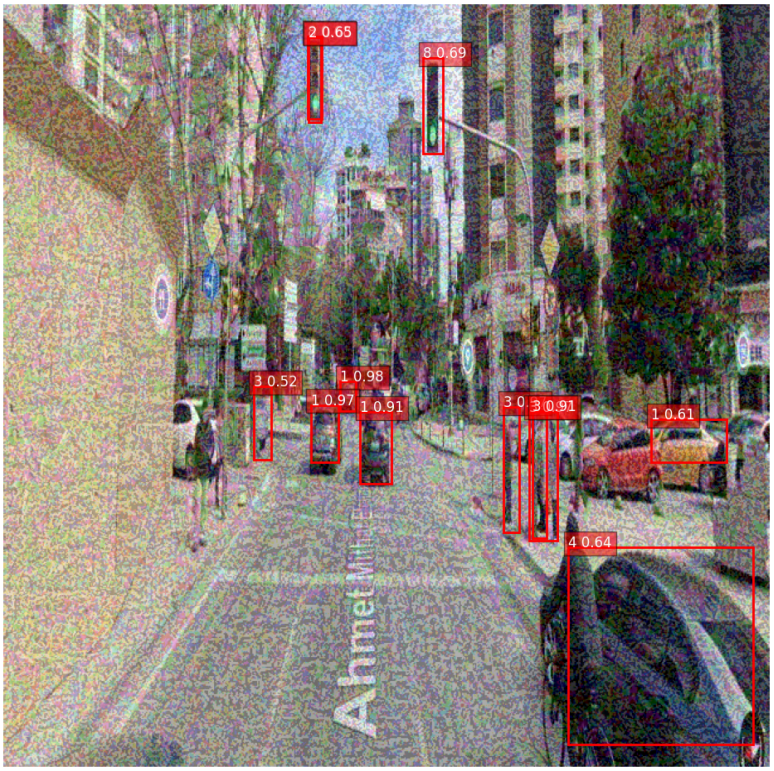


Adversarial Image of YOLOv5 Testing:

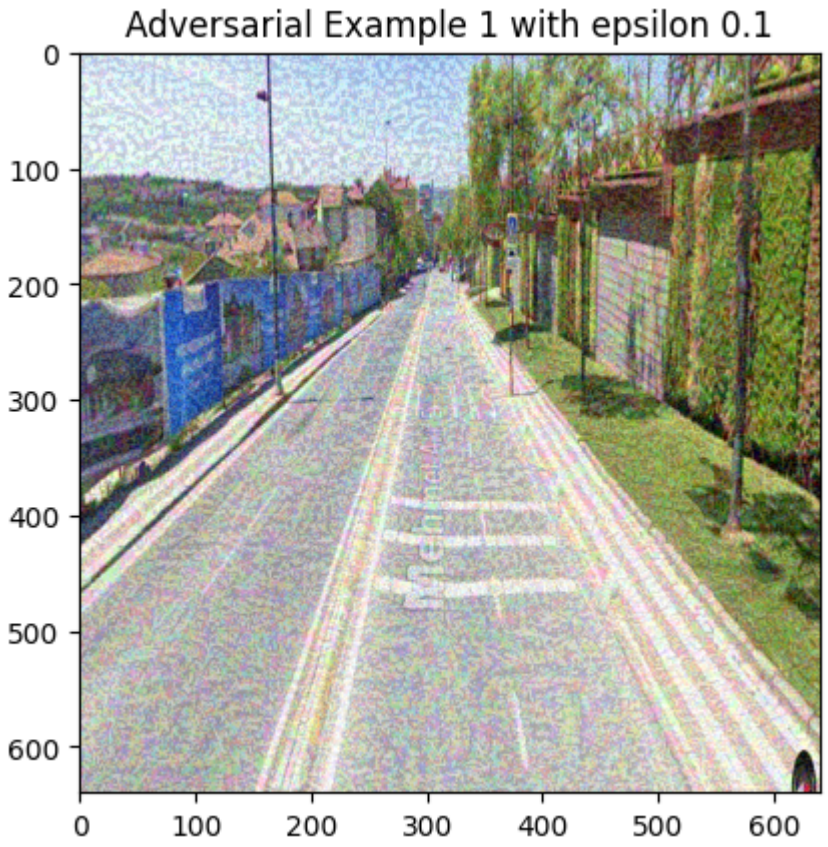


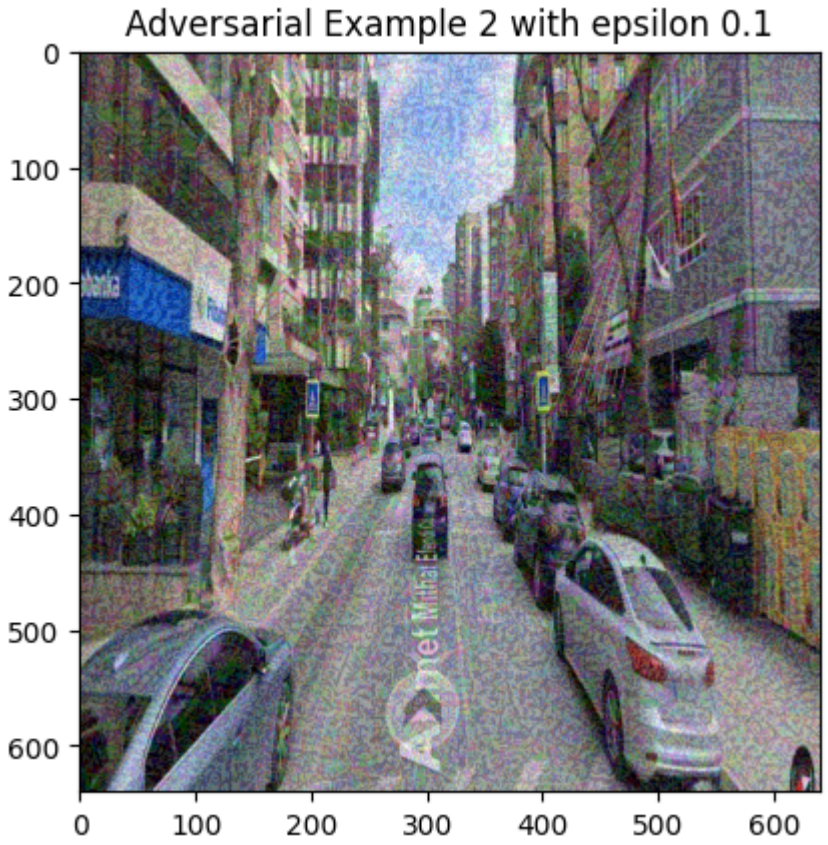
Original Image of Faster R-CNN testing:

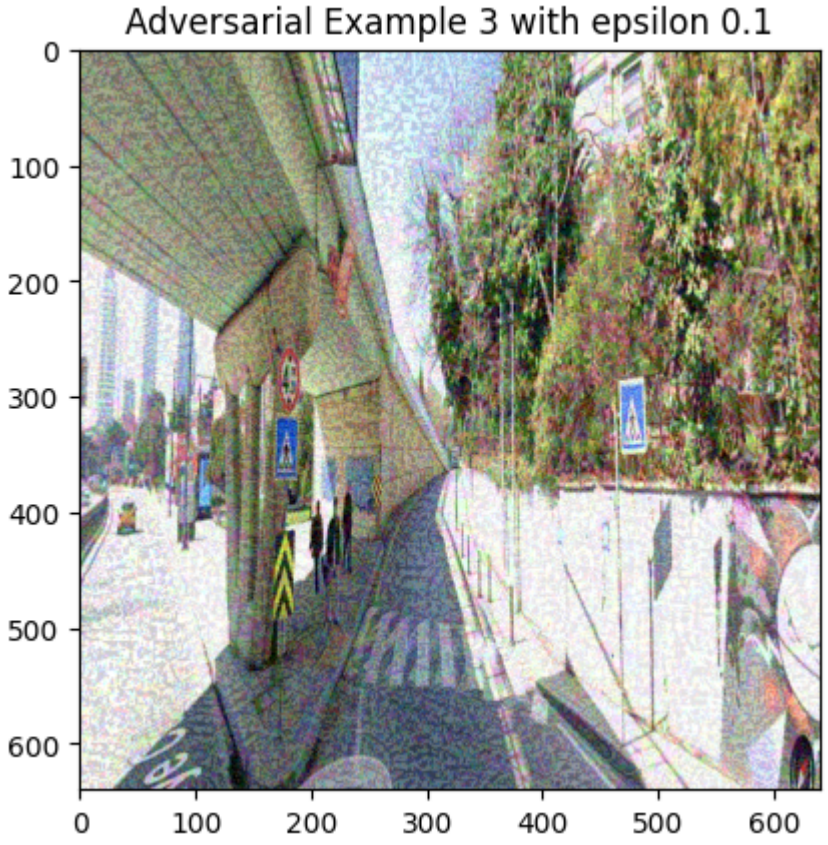
Adversarial Image of Faster R-CNN Testing:



Other Examples Adversarial Images




Adversarial Example 2 with epsilon 0.1
0
100
200
300
400
500
600
0
100
200
300
400
500
600


Adversarial Example 3 with epsilon 0.1
0
100
200
300
400
500
600
0
100
200
300
400
500
600