Deploying Fast Object Detection for Micro Aerial Vehicles

by

Nicholas Florentino Villanueva

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aerospace Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Signature redacted

Author ....
Department of Aeronautics and Astronautics
May 24, 2018

Signature redacted

Certified by.......................
Nicholas Roy
Professor of Aeronautics and Astronautics
Thesis Supervisor

Signature redacted

Accepted by ....
Hamsa Balakrishnan
Associate Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Deploying Fast Object Detection for Micro Aerial Vehicles

by

Nicholas Florentino Villanueva

Submitted to the Department of Aeronautics and Astronautics
on May 24, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Aerospace Engineering

Abstract

In this thesis, we present an evaluation of four state of the art convolutional neural network (CNN) object detectors, and a method to incorporate temporal information into an object detection pipeline for a micro aerial vehicle (MAV). This work was done as part of the Defense Advanced Research Projects Agency's (DARPA) Fast Lightweight Autonomy (FLA) program with the goal of creating an autonomous MAV that could explore and map an unknown urban environment. We tested four CNN-based object detectors on a range of compact deployable compute devices in order to select the best detector-hardware pair for our flight vehicle. We chose to use the MobileNetSSD object detector running on an Intel NUC Skull Canyon. Additionally, we developed an efficient object detection method that incorporates temporal information found in sequential camera frames by selectively utilizing an object tracker when the object detector fails. Our temporal object detection method shows promising results improving the recall of the base object detector on two of three datasets while maintaining a high framerate.

Thesis Supervisor: Nicholas Roy
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank my advisor, Nicholas Roy, Professor of Aeronautics and Astronautics at MIT. Nick has been my advisor at MIT through both my undergraduate and graduate careers and has been a great mentor to me both academically and through his leadership. I would also like to thank Jake Ware, John Carter, the FLA team, and the Robust Robotics Group for their support, without which this wouldn't have been possible.

Additionally, I would like to thank the Lemelson Foundation for their assistance this year.

Lastly, to my friends and family who were with me for all the good times and the bad, thank you for sticking with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem

Robots are becoming an ever larger part of our daily lives. Their applications are becoming broader as hardware gets smaller and more powerful and algorithms become more effective and efficient. However, in order for robots and humans to work together, machines must have an understanding of the world similar to that of their human counterparts. The necessity for a common understanding highlights the importance of semantics for robots, or meaningful labels assigned to the perceived environment. For example, it is important for a home assistant robot to know that an apple is different from a plate when a human asks for a snack. Furthermore, the robot must be able to recognize both the apple and the plate in order to connect their meanings with the immediate environment. This requires a semantic perception system for the classification of objects. We propose such a semantic perception system, specifically, an object detection pipeline, to be used by deployed robots to recognize objects in their environments.

This object detection pipeline was used as part of a larger program called Fast Lightweight Autonomy (FLA) put on by the Defense Advanced Research Projects Agency (DARPA). In the first phase of the program, the goal was to use a micro aerial vehicle (MAV) to locate an item (a red barrel) at the end of a trajectory and return to the start, moving as fast as possible (20m/s) while dodging obstacles. This environment could range from an outdoor field, to a forest, or to the inside of a warehouse.

In the second phase of the program, the challenge was to explore an urban environment and create a semantic map of the objects that the flight vehicle encountered during its exploration. To complete this mission, the flight vehicle would need state estimation, depth estimation, local mapping, flight control, behavior selection, object detection, and object localization. The goal was to complete these tasks using only monocular camera methods, the motivation being that images contain a rich source of information that is commonly underutilized. The advent of Convolutional Neural Networks (CNNs), along with advances in classical computer vision techniques, have made using a single camera for the tasks stated above much more reliable for flight vehicles. CNNs in particular have made hard coded feature functions for image analysis largely a thing of the past.

Object detection was used to report back what the vehicle encountered on its exploration of the urban environment. The class of objects that the vehicle was to report were predetermined ahead of deployment. For this mission, the objects to detect were mainly people and cars. These objects are typical in urban environments and knowledge of their locations may be of importance for soldiers deployed in an unknown city assessing possible dangers or points of interest.

An object detection is made by placing a bounding box around each object of interest in a given image with an associated semantic label. An example is shown in Figure 1-1. The detections are then sent to a mapper that localizes the object in the global map to be reported back to the user. While seemingly straightforward, the task of object detection comes with a host of challenges. For example, it is easy for humans familiar with cars to pick them out of an image whether the car is big or small, and regardless of the car's orientation. Cars alone come in all kinds of shapes, sizes, and colors. Additionally, when encountered in the real world they can be found in many different orientations. A large part of the difficulty in the object detection task is generalizing. Not only are there many types of cars, they can be encountered in many different situations. Often times, cars in an environment can be occluded or

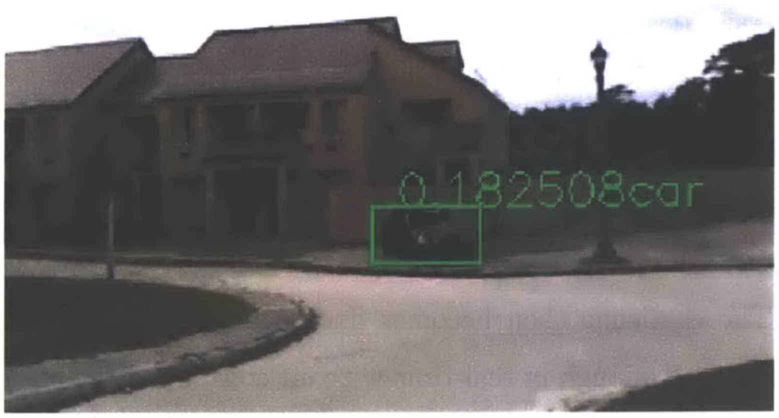


Figure 1-1: An object detection consists of a bounding box, a semantic label, and a confidence score.

found in cluttered environments and the object detection system would have to be able to pick out the car from the clutter. This problem becomes even more difficult when the object detection system is asked to classify other objects in addition to cars. There are classical computer vision techniques that can perform object detection in many ways whether it be by breaking an object down into parts [17] or computing hand crafted features [74]. Unfortunately, these methods can be tedious and often times do not perform well in a variety situations.

A CNN, a type of artificial neural network, on the other hand, is able to learn useful feature functions for analyzing images, eliminating the need for handed coded feature functions. CNNs, once trained on enough data, can learn general representations of many classes of objects. This makes them very accurate detectors outperforming methods that preceded them. The ability of CNNs to generalize is why we chose to use CNN-based object detectors for the flight vehicle. Although many existing object detectors exist, such as Faster R-CNN [58], You Only Look Once v2 (YOLOv2) [57], and the Single Shot MultiBox Detector (SSD) [42], there are still challenges for deployment.

Part of the DARPA FLA program was to put all the autonomy and image processing stated above onto a single MAV. A main concern when deploying CNN-based methods is their computational load onboard the vehicle, which often has much less compute power than the environments where the methods were developed. CNNs, while state of the art, use millions of parameters when computing detections from a single image. This can come at a significant cost especially on a vehicle with a small computational budget. Decreasing the size of the CNN could speed up computation, but would come at a cost to accuracy. On top of this, in an autonomous flight system, the detections need to be sent in real-time to the object localizer to be placed in the global map. The challenge then becomes finding a CNN object detection pipeline that can perform well enough in real-time without consuming the entire computation budget of the vehicle. Many state of the art detectors boast short computation times and high accuracy, but do not tackle the issue of deploying these systems on size-, weight-, and power-constrained devices.

Additionally, the state of the art object detectors are made to detect objects on single images. During deployment, the detector will process sequential images in a live video stream. Due to vehicle movement, lighting changes, and viewpoint changes, some objects detected in one camera frame may not be detected in the next. This can cause object detections to be dropped and may provide too few detections to properly localize the object. Specifically for the DARPA challenge, the MAV is flying through an urban environment where objects are generally sparse. In many cases, the vehicle will encounter objects at a distance or in the periphery of the camera frame, which could mean that several detections are required to localize the object.

To address these problems, we present an evaluation of the runtime and resource consumption of some of the fastest state of the art object detectors. This evaluation can aid in the selection of a particular method for devices with relatively lower computation throughput. Additionally, we propose a selective tracking-by-detection method that uses temporal object representations to create more consistent object detections over time.

1.2 Our Approach

The following sections will discuss how we selected an appropriate neural network object detector for our hardware and incorporated temporal information in the detection pipeline to increase object recall and temporally consistent detections.

1.2.1 Neural Network Object Detector Evaluation

The first step involved selecting an object detection method to put on the vehicle using an analysis of the current state of the art in object detection. Our main criteria in this evaluation were speed, accuracy, and computational resource consumption. To this end, we narrowed down the search space of detectors to CNN-based methods because CNNs are the top performing in terms of accuracy and are highly generalizeable given enough training data. Additionally, all of our proposed hardware on which to deploy the object detector had a GPU, which allowed the detector to reduce its inference time.

Using past evaluations as a starting point such as the one done by Huang et al. [28] and the experimental results from the original object detection papers, we narrowed down our search to the You Only Look Once (YOLO) methods (YOLOv2 [57] and Tiny YOLO [54]) and the Single Shot MultiBox Detector method [42]. We chose these methods because they were designed to be both fast and accurate.

The object detection methods were evaluated across selected deployable compute devices that could be used on the flight vehicle. When testing the object detection methods on the devices, we were mainly looking at the test time frame rate and GPU, CPU, and memory usage. The methods were first tested on a desktop machine with an NVIDIA TitanX GPU and Intel Core i7-6700K CPU. After initial testing on a machine without size, weight, and power constraints, we tested the methods on a range of deployable hardware. This included a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon.

After the evaluation of these methods on different hardware, we decided to move forward with the Single Shot MultiBox Detector using MobileNet [27] as its feature extractor. We will refer to this combination as MobileNetSSD. This setup ran the fastest while still maintaining reasonable accuracy performance.

We used a version of MobileNetSSD [3] implemented with the Caffe deep learning framework [33]. The network runs entirely on the Intel NUC Skull Canyon using an OpenCL version of Caffe [71] so that we could take advantage of the NUC's GPU for inference.

1.2.2 Object Detection for Micro Aerial Vehicles

Although deploying a CNN object detector onboard a MAV works generally well given that it can run in real-time, most state of the art detection methods were made to address the PASCAL Visual Object Classes detection challenge [15], which measures the accuracy of detectors on a test set of still images. However, when detectors are running on a robot, they process sequential camera frames, and also need to deal with varying object poses and possibly blurred images from motion.

To tackle this problem, we leveraged information present across camera frames to augment the object detection pipeline. Specifically, we increased the confidence of detections on objects the robot has seen before by using a similarity metric based on object color, size, location, and bounding box dimensions. This metric was created by using an adaptive weighting scheme for each of the metric components based on the size of the bounding box in a given frame. Since we know the general shape and size of the objects we are detecting and the environment in which we are flying, the bounding box size roughly relates to the distance from which an object is viewed. Based on this size, we can dynamically weight how much the system should rely on color, bounding box shape and size, and location.

We also leveraged an object tracker in the detection pipeline by selectively applying it to find objects that the detector found in a previous frame, but failed to detect in the current frame. The tracker was selectively applied by using the similarity metric to to determine if the object was detected or not. In this way, we only applied the tracker to an object that needed to be detected.

The augmented object detection pipeline was designed keeping into consideration the limited computational capacity onboard the vehicle, and the need to maintain a real-time system. The entire pipeline consists of a detection step, a rescoring step, a tracking step, and a non-maximum suppression step. Although the detection pipeline could be setup with any arbitrary single image object detector and image based object detector, we implemented the system with the MobileNetSSD object detector and GOTURN object tracker [24]. Together, the system runs at a minimum of 40 frames per second on the Intel NUC Skull Canyon.

1.2.3 Contributions

Our first contribution is an evaluation of several state of the art CNN-based object detection networks tested across a range of deployable compute hardware. The evaluated object detection networks included YOLOv2, Tiny YOLO, SSD, and MobileNetSSD. The hardware for the evaluation included a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon. The test results can help in selecting an appropriate CNN-based detector and compute-device pair.

The second contribution is a proposed method to incorporate temporal information into an object detection pipeline that is both effective at increasing detector recall and maintaining a high frame rate. This is done by combining an object detector and an object tracker in an efficient framework where the object tracker is only applied when necessary.

1.2.4 Thesis Outline

This thesis is organized into six chapters. Chapter 2 provides a background for CNNs as well as past and current methods for object detection, object detection evaluation, object tracking, and object detection in videos. Chapter 3 gives a summary of the methodology of the evaluated neural network object detectors. It then provides the results of the object detector evaluation across the tested hardware. Chapter 4 describes how the object detector was deployed on the flight vehicle. Lastly, Chapter

5 introduces the temporal object detector and presents results of the method using data collected in three different environments.

Chapter 2

Background

2.1 Convolutional Neural Networks

At the core of the object detector analysis and our temporal object detector, is the use of convolutional neural networks (CNN). CNNs are a type of artificial neural network that take 2D signals as input; this can include things such as an audio signal or an image. Neural networks are useful because they can approximate complex nonlinear functions. Neural networks are made up of a series of units often referred to as neurons stacked into layers. Each neuron consists of a series of weights w_i, a bias term b, and an output function. In a network, a neuron takes in inputs x_i from other connected neurons. The output of the neuron is the output function applied to the sum of the weights, w_i, times the inputs, x_i, plus the bias b. If the output function is a linear function, then the sum of the weighted inputs plus the bias is exactly the output. To produce a nonlinear result, a nonlinear function is applied. Common nonlinear output functions used in neural networks include the $\tanh(x)$ function shown in Equation 2.1, the rectified linear unit (ReLU) shown in Equation 2.2, and the leaky rectified linear unit shown in Equation 2.3.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.1}$$

$$f(x) = \max(x, 0) \tag{2.2}$$

$$f(x) = \begin{cases} \max(x, 0) & x >= 0 \\ \alpha \min(x, 0) & x < 0 \end{cases} \tag{2.3}$$

The nonlinear function can be chosen depending on the desired output of the neuron, for instance if we wanted to have the output bounded between -1 and 1, or to act as a threshold function on some property of the input. The output of a neuron with an applied nonlinear function, taking n inputs x_i, with n weights w_i, a bias b, and a nonlinear function $f(x)$ is shown in Equation 2.4.

$$y = f\left(\sum_{i=1}^{n} x_i w_i + b\right) \tag{2.4}$$

A neural network is then a series of layers of neurons stacked onto one another as shown in Figure 2-1. The network in Figure 2-1 takes in a column vector x and outputs a single value y. The first layer of the network is the input layer, the middle layer is a hidden layer, and the last node is the output layer.

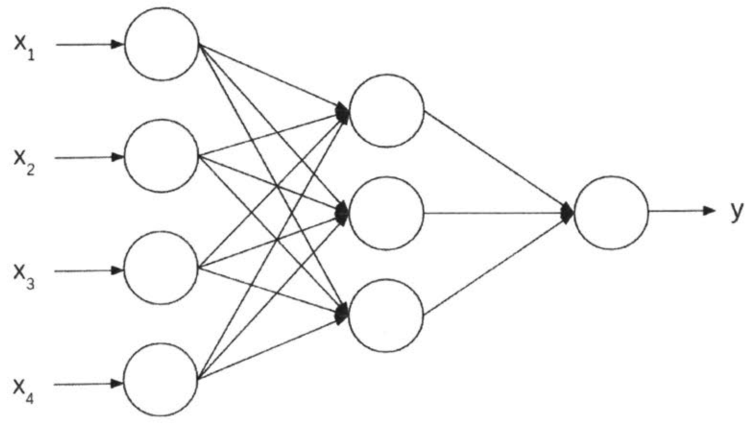


Figure 2-1: A fully connected three layer neural network.

A neural network is trained through a process called stochastic gradient descent using backpropagation. During training, a network is given an input and a label. The input is passed through the network and a loss is computed between the output of the network and the label. The loss function can be thought of as a function that makes a comparison between the output of the network and the label. The goal then is to minimize this function. This minimization is done using stochastic gradient descent, which calculates gradients for the network parameters with respect to the loss function. Since each layer of the network depends on its parameters and input, the gradients can be found by recursively applying the chain rule to each layer of the network; this use of the chain rule is known as backpropagation. The network parameters θ can then be updated using the calculated gradients with Equation 2.5 where t is the iteration and η is a scalar typically called the learning rate.

$$\theta^{t+1} \leftarrow \theta^{t} + \eta \nabla \theta \tag{2.5}$$

Convolutional neural networks are similar to the previous example network except that they take 2D inputs and have volumetric layers of filters instead of layers of neurons. The filters are convolved with the input signal to produce a volumetric feature map output. Convolution makes CNNs spatially invariant allowing the filters to be trained to respond to a feature anywhere in the input signal. This is especially useful in image processing where key features may appear anywhere in the image. In an example, similarly done by Howard et al. [27], given an input image of size $D_w \times D_w \times p$, where D_w is the width of the image and p is the number of channels, to produce an output feature map of size $D_w \times D_w \times N$, where N is the number of output channels, N number of $D_k \times D_k \times p$ filters are applied to the input image.

Like the neurons of a neural network, the volumetric layers, also known as convolutional layers, are typically followed by some type of nonlinear function like those mentioned previously. There are also other types of common layers like max pooling, which selects the maximum value from a portion of a feature map. These convolutional layers, nonlinear functions, and pooling layers make up the basic building blocks of CNNs.

2.2 Object Detection

Object detection tackles the problem of locating object of interest in an image. In the time before CNNs were applied to the problem, solutions usually involved using some sort of hand-coded features to detect objects of interest. One such method by Viola and Jones was presented in [74]. Their method used rectangular features quickly computed using an integral image. Machine learning was used to select descriptive features and train a classifier. Using a hierarchy of classifiers, they took a coarse to fine approach to find the object in the image. The detection algorithm was able to run at 15 frames per second (fps) on a 700 MHz Intel Pentium III, however it could only detect one class of objects.

To better incorporate detecting more than one class of object, Torralba et al. in [70] proposed training classifiers jointly. They did this by looking for common features between object classes. This method could also help generalize the detector for a single object class by finding common features between different viewpoints. Another pre-CNN-based object detector used histograms of oriented gradients or HOG features presented by Dalal and Triggs [6]. Dalal and Triggs used their histogram of oriented gradients method for human detection using a linear SVM. They found that objects can be characterized by a local distribution of intensity gradients. Their detector worked so well that the authors created their own more difficult dataset to demonstrate its capabilities. They also pointed out a few advantages of their method over others. They found that using HOG descriptors over SIFT [43] features reduced the number of false positive detections by an order of magnitude. HOG descriptors are also good at generalizing the shape and structure of an object because they can be invariant to small translations or rotations.

Although the work done by Dalal and Triggs was groundbreaking at the time, it was still limited. Like the work done by Viola and Jones, it was only designed to detect one object. Both of these techniques could be used to detect multiple objects, but would need separate classifiers for each object class.

An object detection method by Felzenszwalb, Girshick, McAllester, and Ramanan

[17] used parts-based models to find objects. The idea behind the parts-based models is that objects can be represented by their individual parts. Their method uses mixture models making it possible to detect objects at different orientations and deformability such that objects parts can be in different places with relation to each other. This formulation helps their method generalize to different object representations. The discriminatively trained part-based model [17] achieved state-of-the-art performance on PASCAL VOC challenges [14, 11, 12] and the INRIA Person dataset [6]. Despite this performance, it took the method approximately 2 seconds to evaluate an image with a 2.8Ghz 8-core Intel Xeon Mac Pro.

Soon CNNs dominated the object detection space. One of the early CNN based object detectors by Sermanet et al. [62] was called OverFeat. Their method combined classification, localization, and object detection. The authors first trained a CNN classifier using stochastic gradient descent (SGD) on the ImageNet 2012 dataset [8], composed of 1.2 million images and 1,000 classes. The first few layers had a similar setup to the network made by Krizhevsky et al. [38] with ReLU nonlinearities and max pooling. For localization, they replaced the classification layers with a regression network to make bounding box predictions at different locations and scales on the image. The detection task was done similarly except that they have to train for a background class when there are not any objects of interest in the image. Their method came in fourth for the classification task, first in the localization task, and first in the detection task for the ILSVRC 2013 dataset [60].

After Overfeat, came a family of region-based based convolutional neural network object detectors. These were R-CNN [21], Fast R-CNN [20], and Faster R-CNN [58]. The idea behind these methods is to combine region proposals with CNNs, hence R-CNN. The first R-CNN [21] uses selective search [72] to create 2,000 region proposals. The regions are then warped to a specific size to be put through a CNN like the Krizhevsky et al. network [38]. The CNN produces a feature vector that was scored by a class-specific SVM for classification. The authors found that performance improved if the CNN was first pretrained on ILSVRC 2012 [7] dataset and then finetuned on the data from VOC, composed of only 20 classes.

This method did well on the VOC 2012 challenge scoring a mAP of of 53.3%, 30% higher than the nearest competitor at the time. Although this method had good accuracy and scaled fairly well for many classes, it still had a long run time of nearly 13 seconds for each image using a GPU. The next version, Fast R-CNN, was designed to address this limitation.

The creators of Fast R-CNN pointed out some issues with R-CNN. The first problem was that R-CNN had multiple stages; the CNN and the SVM. Second, training was slow since these parts had to be independently trained. Lastly, R-CNN was very slow during inference time since it did a forward pass through a CNN for every region. Fast R-CNN aimed to unify the process. Given an image and regions of interest, Fast R-CNN generates confidences for each class and bounding box offsets in one pipeline. They accomplish this in part by having a multi-purpose loss function that has both class and location components.

This method increased accuracy from R-CNN and greatly increase both training and test speed. They reported an inference time of 0.3 seconds per image and an mAP on PASCAL VOC 2012 [13] of 66%.

After Fast R-CNN came Faster R-CNN. The idea behind Faster R-CNN was to share the computation of the region proposals with the detector, by creating a region proposal network (RPN) where all but the last few layers are shared with the detector. The RPN then returns region proposals and objectness scores, whether or not the region contains an object, to the detector. The RPN uses a sliding window across a feature map and for each location it predicts a certain number of regions and objectness scores.

This combination of RPN and detection layers led to a substantial speed up for an entire object detection pipeline. This created one of the first CNN object detection methods that could run in real time running at 5 fps on a GPU while also scoring state of the art accuracy on the PASCAL VOC 2007 [11] and PASCAL VOC 2012 [13].

Another architecture similar to Faster R-CNN by Dai et al. was the Region-based fully connected networks or R-FCN [5]. Their idea was to share more computation between the region proposal network and the detector. The last convolutional layers of the network create a set of position sensitive score maps for each class. A position sensitive region of interest (RoI) pooling layer comes after. These scores could then be used to classify a region and refine the bounding box. This method was able to achieve an inference time of 170ms or roughly 5.88 fps and an mAP of 83.6% on the VOC 2007 set [11].

Many of these proposed CNN-based object detectors had two parts, region proposals and classification. Other methods have more of a unified implementation. One such method is the You Only Look Once (YOLO) [56] object detector.

YOLO presented by Joseph Redmon et al. in [56] is a neural network architecture that has a unified bounding box proposal and object recognition pipeline. YOLO does this by representing the object detection problem as a regression problem. This method is able to look at the entire image at once, as opposed to sliding window methods. For bounding box proposals, looking at the entire images gives it the ability to take in contextual information. Compared to the R-CNN type object detectors, YOLO is very fast running at a reported 45 fps.

2.3 Object Detection Evaluation

In order to determine which object detection method to use for a particular task, it is important to have a way to compare them. One of the popular object detection benchmarks is the PASCAL Visual Object Classes (VOC) Challenge [15]. The PASCAL VOC Challenge provides a way to evaluate and compare the accuracy of different object detectors. The challenge provides annotated training and testing images for 20 different classes. Using the same data for training and testing across different methods of object detectors offers a more even playing field for comparison. The PASCAL VOC challenge evaluates object detectors using Average Precision (AP). AP is found by taking the average of the measured precision of the method over different recall levels. Recall measures how many true positive detections were found out of the total above a rank and precision measures the proportion of true positive detections over all the detections made above a rank. A detection is deemed a true positive if the ratio of the overlap area, the area shared by the detection bounding box and the ground truth bounding box, to the total area of both bounding boxes is above 0.5.

While the evaluation method used in the PASCAL VOC Challenge is widely used and useful when comparing object detectors, it does not measure other useful metrics like object detector runtime. Another detector evaluation by Dollár et al. [9] looked at different pedestrian detectors in street settings. Although they were only measuring the performance of the methods detecting one class, they looked at more metrics geared towards deployment performance. The detectors were tested on data recorded by driving cars around a city. They split the annotation of their data by the distance of the pedestrian from the car using three levels: near, medium, and far. This way they could do an in-depth evaluation on the detector performance at these different distance levels. Instead of using AP, the authors used the log-average miss rate as their metric which averages the miss rate of the detector at nine different false positives per image rates. This was also an evaluation that looked at the runtime performance of the detectors.

A recent evaluation of state of the art CNN object detectors by Huang et al. [28] was a very thorough analysis of several different object detection architectures and feature extractors. The authors tested Faster R-CNN, R-FCN, and the Single Shot MultiBox Detector (SSD) with different combinations of feature extractors. The tested feature extractors included VGG [64], Residual Networks [23], and MobileNets [27]. This analysis not only evaluated the object detectors on accuracy, but also GPU time, memory usage, and FLOPs or multiply-adds. In order to do this evaluation they had to implement many of the network combinations themselves in the deep learning framework Tensorflow [1]. This evaluation most closely resembled the evaluation done in this thesis. However, their evaluation was not tested across different hardware and they also did not include the YOLOv2 [57] network in their evaluation.

2.4 Object Tracking

When object detection is used for real-world applications, the input data is a stream of images rather than on disconnected images. This is why we chose to incorporate an object tracker into our temporal object detector since this is the domain in which they function. One popular tracking method was Kernelized Correlation Filters or KCF [25] from Henriques et al.

The authors break down the problem of tracking objects in images as trying to train a classifier on a training target image and then finding the target in the next frame by successfully being able to separate it from the background. The classifier can then be updated with the new target model. A key insight of the KCF tracker is that they are able to exploit a property of circulant matrices that allow them to quickly and efficiently train on thousands of translated samples. Their tracker performs well and can be used on either raw pixels or on HOG [6] features mentioned earlier. Using raw pixels they achieve a mean precision of 56% and with HOG features they achieve an even better result of 73.2%. In addition to the high accuracy, this method is also very fast. With raw pixels the method can run at an average of 278 frames per second and using HOG features it can run at an average of 292 frames per second.

Another tracking algorithm, from Comaniciu et al. [4], also focuses on target representation and correlation. Their goal is to optimize a smooth similarity function to localize the target in a new frame. They do this by applying a spatial isotropic kernel to the target object in a frame. The object representation is a histogram of features, which for example could be color. The isotropic kernel weights the pixels in the center of the target region more then the edges since those pixels are the most representative of the target. They then maximize a likelihood score derived from the Bhattacharyya coefficient to localize the target from a search region in a following frame. The method achieves a frame rate of 150 frames per second on a 1GHz PC.

Somewhat of an alternative to using a single representation of the target object, Reliable Patch Trackers (RPT) from Zhu et al. [39] uses a particle filter to weight different trackable regions of an object. The authors propose using these regions to track their trajectories and separate them from the image background. Although a robust method, achieving an intersection over union over 0.5 on 38 of 51 videos compared to KCF [25], which only achieved this on 31, this method only runs at an average of 4.2 frames per second.

Similar to breakthroughs in object detection, the use of convolutional neural networks has also made progress for object tracking. One such tracker by Ma et al. [44] learns correlation filters over the features extracted from each level of the feature maps. They use a hierarchical view of the CNN layers in a network in that the last layer contains semantic information, which is more invariant to object deformation, while the previous layers contain more spatial information for the object. Using the correlation filters on this hierarchical structure of identification to spatial localization they can track a desired object. While a fairly robust tracker in terms of its accuracy on tracking datasets, it is slower than other trackers like KCF running at 10.7 frames per second while using a GeForce Titan GPU for forward passes of the CNN.

Akin to the hierarchical method, the fully convolutional network based tracker of FCNT [75] also exploits the differences in feature map layers to track objects. They switch between using two different layers of the network to track the target object depending on the presence of distractors or deformations. Again, like the hierarchical method this method is robust, but is only capable of running at 3 frames per second using a Titan GPU.

Although the previous methods were relatively slow, some CNN based trackers can run at much higher frame rates. Two related trackers, SiamFC [2] and CFNet [73] by Jack Valmadre and Luca Bertinetto depart from the previous CNN-based trackers by using a two input CNN, or siamese network, for the target and search region, trained to find the similarity between two image patches. This shift in methodologies led to higher frame rates while maintaining relatively high accuracy. Both methods can run between 58 and 86 frames per second. CFNet is based on SiamFC, but they are able to incorporate a correlation filter as part of one side of the siamese network and train both the filter and network offline. This allows them to use a shallower network than just the siamese network and maintain a similar accuracy.

2.5 Object Detection in Video

Some methods address object detection in video directly. One of the challenges in the ImageNet Large Scale Visual Recognition Challenge 2015 [61] is to detect objects in video. One method that tackles this problem is Seq-NMS proposed by Han et al. [22]. This is a simple add-on to existing object detection methods where detections are linked over time based on their overlap with a detection with a previous frame. They create a sequence of detections over an entire video and reassign scores across the sequence. In this way they can maintain some temporal information in the final detections. This is similar to the method we use, however Seq-NMS can reason over an entire video sequence and it only uses an overlap criteria for object association.

The winner of the object detection in video challenge, T-CNN created by Kang et al. [35] uses both single frame object detection methods and tracking methods in a single pipeline. Their method consists of using two still image detection networks. The detections produced from these detectors are then sorted based on score to suppress low scoring classes that are unlikely to be in the video sequence. They use object motion to propagate detections from previous to subsequent frames and trackers are applied to create tubelets across the video clip. These are then re-scored based on the detection scores. Finally, the proposals from each detector are combined by non-maximum suppression to produce the final result.

Another class of methods that also tackle the object detection in video challenge use tubelet proposals. These include "Object Detection from Video Tubelets with Convolutional Neural Networks," by Kang et al. [36], "Object Detection in Videos with Tubelet Proposal Networks" by Kang and Li et al. [34], and "Object Detection in Videos by Short and Long Range Object Linking" by Tang et al. [65]. Tubelets are linked candidate detections across video frames. The work by Kang et al. uses a three step tubelet proposal pipeline that consists of object proposals, object proposal scoring, and tracking of high scoring proposals. The authors then perturb, score, and run a non-maximum suppression step for each detection generated by the tubelet proposal pipeline. The last step is to use a Temporal Convolutional Network that takes in as input information from the tubelet such as detection and tracking score and it outputs a probability of whether or not the proposed detection overlaps a ground truth detection. In this way they can capture more temporal information over an entire tubelet.

Both the work by Kang and Li et al. and Tang et al. use neural network based tubelet proposals. Kang and Li et al. combine a tubelet proposal network that generates tubelets across the video taking into account motion learned from training. A Long Short-Term Memory network then generates object confidences using information from the entire video. Tang et al. on the other hand, use a cuboid proposal network that creates a bounding box extending over several frames containing an object where a tubelet is generated and theses tubelets are then linked across video segments.

Detect and Track or D&T is a method created by Feichtenhofer et al. [16] that brings together advances in CNN object detection and object tracking. D&T builds on top of the R-FCN [5] detection network and combine it with a correlation layer and ROI tracking. Feature maps of different scales are used by the correlation layer to find the movement of the object from one frame another. In this way they can both detect and track objects in a unified manner. This method scores well on the detection task with an online version running at roughly 7.9 frames per second and achieving a mean average precision of around 79%.

The previous methods assume that the entire video sequence is available, as is the case on the object detection in video challenge, however another method created by Galteri et al. [18] only uses a previous frame to help locate object from one frame to another. The method uses a feed back loop where information from the previous frame can be fed into the current. Region proposals from the previous frame are fed back into the current frame and they rescore the current proposed regions based on their overlap. This has a dual effect of helping to find previously detected images and narrowing down the proposals in a current frame. Although this is a fairly simple method to incorporate temporal information it does require that the baseline detector uses region proposals in its detection pipeline.

A method used for deployed object detection was developed by Pillai and Leonard as described in [52]. The method assumes that the robot builds a semi-dense reconstruction of the world it views. They partition the semi-dense reconstruction using spatial and edge color information. Once the scene is segmented, they can use this as object proposals for object recognition by projecting the segmentation into the camera frames. The object can then be classified using a bag-of-visual-words method and aggregated over multiple frames. This was shown to be a fairly robust method for object detection. It does however assume that the robot is able to create a good reconstruction of the world around it. Many times this type of reliable information is not available to the robot.

Chapter 3

Convolutional Neural Network Object Detector Evaluation

In this chapter we will give an overview of the evaluated object detectors and their runtime performance. The first section will give a summary of the methodology for the evaluated detectors: You Only Look Once v2 (YOLOv2), Tiny YOLOv2, the Single Shot MultiBox Detector (SSD), and a variant of SSD called MobileNetSSD. The second section will discuss the tested hardware: a desktop setup with an NVIDIA TitanX GPU and Intel Core i7-6700K CPU, a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon. The stated hardware specifications were reported by the respective maker or directly queried from the device unless otherwise noted.

The main goals for selecting an object detector were to minimize computation time and load as well as maintain an acceptable level of detection accuracy. The object detection network needed to run on size-, weight-, and power-constrained hardware onboard a micro aerial vehicle (MAV) along with other processes for autonomous estimation, planning, and control.

Specifically, the first requirement for the object detector was that it had to run at a minimum of 2 frames per second (fps) on the deployed hardware. Since the MAV is moving through the environment, it might miss important objects in the scene if the frame rate of the object detector were slower. While 2 fps was set as

Table 3.1: Object Detector Requirements

Specifications	Objectives
Frames Per Second	$\geq$ 2 fps
GPU Usage	$\leq$ 70%
Classes	$\geq$ 5 classes (People, Cars, Doors, Windows, Dumpsters, etc)

the minimum, faster is better. To successfully localize an object, the vehicle would need 3-5 detections of that object. The speed of the vehicle, distance to the object, and layout of the environment influence how long and at what viewpoints objects are seen. The faster the detector, the higher the chance the vehicle will have more detections for localization.

The second main requirement was that the object detection network could only take up roughly 70% of the GPU to allow enough room for other processes to function. The desktop setup and Jetsons were able to run with CUDA from NVIDIA, a programming extension framework that allows processes to utilize NVIDIA GPUs, while the Skull Canyon utilized OpenCL, a similar framework for heterogeneous processor program execution for GPU utilization. These differences affected the performance of the networks on the respective hardware.

The third requirement was accuracy. This was a difficult requirement to determine precisely since the vehicle would be flying through varying environments. As an initial benchmark, the trained networks were judged on their mean Average Precision (mAP) on the PASCAL VOC 2007 test set [15]. The final network was evaluated by its average precision on the test set of its training data. This number was then broken down by object class to ascertain which objects the network struggled to detect.

The last requirement for the detector was that it needed to be able to detect $\geq$ 5 classes. All the tested object detectors could be trained to detect at least 20 classes, so this was an easy requirement to meet. A break down of the requirements can be found in Table 3.1.

3.1 You Only Look Once v2 (YOLOv2)

YOLOv2 [57] is a successor object detection network to the original YOLO [56] created by Joseph Redmon et al. The YOLOv2 network was made with speed in mind, making it a natural network to evaluate on our hardware. YOLOv2 made several improvements to the first YOLO pulling together successes of other object detection networks and the author's own innovations.

The YOLOv2 network architecture consists of a base classification network of 18 convolution layers and 5 maxpooling layers followed by 3 3×3 convolution layers and two 1×1. As opposed to YOLO, YOLOv2 uses batch normalization [32] after every convolution layer. This helps to normalize the input to each layer and regularize. Batch normalization also helps to prevent overfitting so the authors decided to remove the dropout layers from the network.

Another change from YOLO, is that YOLOv2 uses anchor boxes, or priors, over bounding box shapes and sizes, similar to Faster R-CNN [58] and SSD [42]. Using anchor boxes slightly decreases the mAP of the network from 69.5 to 69.2, but the recall shows a larger increase going from 81% to 88%. To incorporate the anchor boxes, YOLOv2 has a nominal input size of 416×416 so that the output feature map is 13×13. Having an odd-sized feature map ensures that there is an anchor box in the center of the image where, in many datasets, an object is located. Although anchor boxes were similarly used in other methods, the authors of YOLOv2 made some notable differences. For one, Faster R-CNN and SSD use hand-picked anchor boxes, whereas YOLOv2 finds its anchor boxes by running k-means clustering on the training data ground truth boxes. The clustering is done using the distance function in Equation 3.1, where d is the calculated distance and IOU is the intersection over union. The intersection over union is the ratio of the area of the intersecting region between two bounding boxes and the total area of the intersecting bounding boxes as shown in Figure 3-1. This distance formulation creates equal weighting for small and large boxes. The authors choose $k = 5$ for the k-means clustering. They show that with $k = 5$, the network achieves similar performance to methods that use 9 anchor

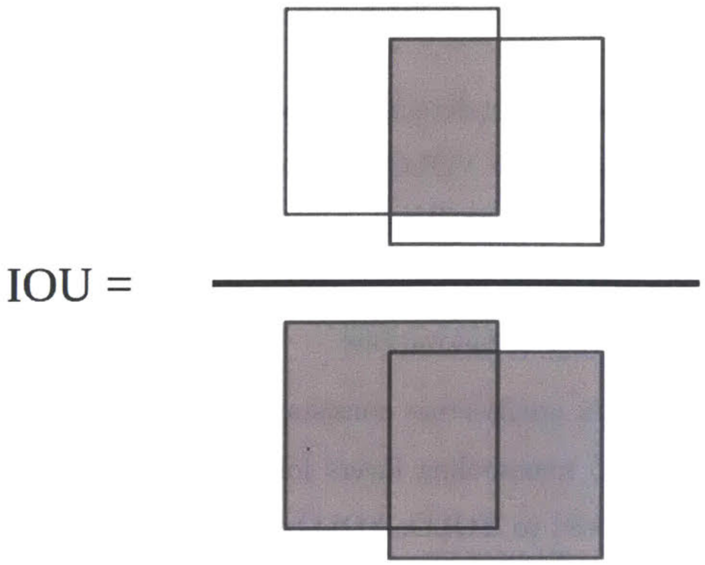


Figure 3-1: The intersection over union between two bounding boxes is found by dividing the area of the intersecting region of the boxes by the total area of the boxes.

boxes. Additionally, using anchor boxes causes the spatial, and class and objectness predictions to be separated. For the case of both YOLO and YOLOv2, objectness is the IOU of the predicted bounding box and the ground truth bounding box and the class prediction is given by the conditional probability of an object of that class residing in the given bounding box given that the bounding box contains an object.

$$d(box, centroid) = 1 - IOU(box, centroid) \tag{3.1}$$

An additional difference between the implementations of Faster R-CNN and SSD, and YOLOv2, is that instead of predicting anchor box offsets, YOLOv2 predicts bounding box coordinates relative to the anchor box cell location. Also, the bounding box size is made relative to the image size. This constrains the ground truth boxes to be within 0 and 1 and the network output is also constrained to be between 0 and 1 using a logistic activation. The detection output is given by the following equations:

$$b_x = \sigma(p_x) + a_x \tag{3.2}$$

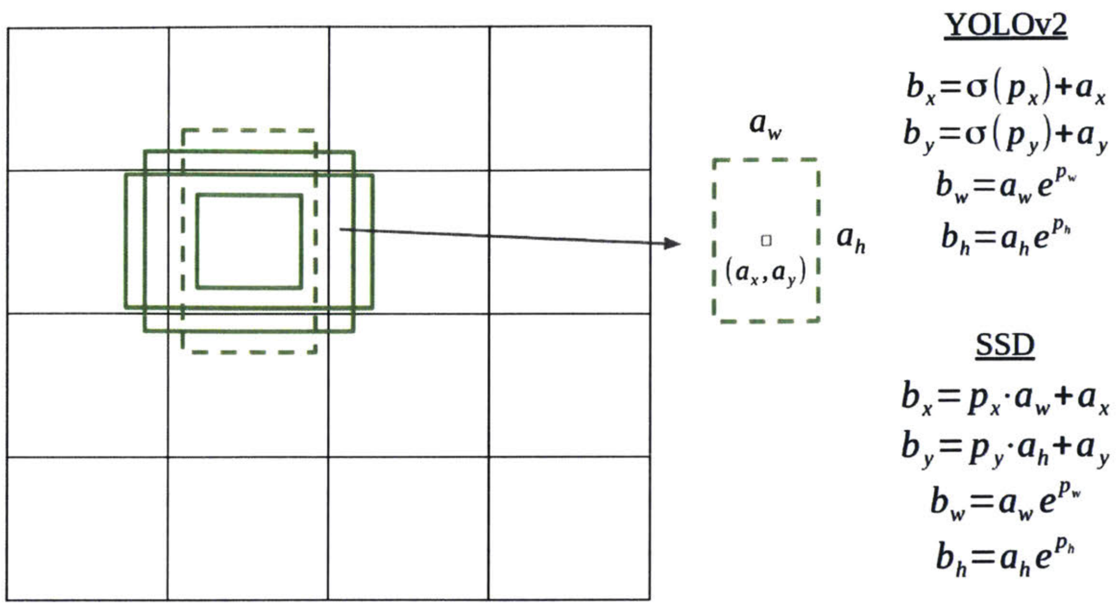


Figure 3-2: Anchor boxes, in green, are used as priors for predicting bounding boxes. The equations for producing the bounding box coordinates for YOLOv2 and SSD are shown on the right.

$$b_y = \sigma(p_y) + a_y \tag{3.3}$$

$$b_w = a_w e^{p_w} \tag{3.4}$$

$$b_h = a_h e^{p_h} \tag{3.5}$$

$$Pr(object) \cdot IOU(b, object) = \sigma(p_o) \tag{3.6}$$

where b_x and b_y are the centers of the predicted bounding box, b_w and b_h are the width and height of the predicted bounding box, a_x and a_y are the image coordinate centers of the anchor box center, a_w and a_h are the anchor box's width and height, b is the predicted bounding box, σ is the logistic activation function, and p_x, p_y, p_w, p_h, and p_o are the outputs of the network. An example of anchor boxes on a feature map is shown in Figure 3-2.

YOLOv2 uses a passthrough layer to incorporate a feature map layer with a resolution of 26×26 to help find objects of different sizes. The passthrough layer stacks adjacent features into different channels matching the 13×13 size. The base feature extractor, Darknet-19, requires almost a fourth of the floating point operations than the VGG-16 [64] base network used by Faster R-CNN at a slight cost of 2% accuracy on ImageNet.

YOLOv2 also has a useful feature in that it is trained to work with input images of different sizes. This adaptability of input size allows trade-off between speed and accuracy. The inference time decreases with lower resolution images at the cost of some accuracy. The authors train YOLO2 on images ranging from 320×320 to 608×608 with multiples of 32 in between. In this evaluation, we only tested YOLOv2 at its default 416×416 input size.

Another feature not explored in our evaluation of the network, is that when YOLOv2 is incorporated with a variable loss function and a tree version of WordNet called WordTree, the network can be trained on both detection and classification datasets. The training process uses a hierarchical tree of object classes and only back-propagates the classification loss when the input to the network is from a classification dataset and back-propagating the full loss when the input is from a detection dataset.

3.2 Tiny YOLO

Tiny YOLO is a faster version of the YOLOv2 network. It gains this speed at a cost to its accuracy, decreasing from a reported 76.8 mAP for YOLOv2 to 57.1 mAP on the 2007 test set [54]. Tiny YOLO is derived from the Darknet reference network [55]. The Darknet reference network has similar top 1 and top 5 ImageNet classification as the AlexNet network [38] with only 1/10 of the parameters. Tiny YOLO uses the first seven convolutional layers and six max pooling layers from the Darknet network. This is much shallower compared to the 18 layers of the base network used by YOLOv2. Additionally, Tiny YOLO only has one added 3×3 convolutional layer followed by a 1×1 convolutional layer compared the YOLOv2's three added 3×3 layers. These reductions in size make Tiny YOLO fast, but also less accurate. Despite the drop in accuracy, this lightweight network made a good candidate for our system and we chose to include it in our evaluation.

3.3 The Single Shot MultiBox Detector (SSD)

The other network architecture we evaluated was the Single Shot MultiBox Detector also known as SSD created by Liu et al. [42]. SSD, like YOLO and YOLOv2, was designed to be fast. SSD uses a base classification network followed by convolution layers of different sizes for multiscale detections. SSD uses these added layers, whereas YOLOv2 has a passthrough layer to use features from a previous feature map.

Predictions are made from these added layers by applying a $3 \times 3 \times p$ kernel, where p is the number of channels at each cell in the feature layer. Again, like Faster R-CNN and YOLOv2, SSD uses anchor boxes of different scales and aspect ratios at each feature map cell. The network then predicts anchor box offsets, as opposed to YOLOv2's cell-relative bounding box coordinates, and class confidence scores for every anchor box in each cell. The class scores for each anchor box correspond to confidence of the presence of that class in the anchor box. Along with the four bounding box offsets, this is a total of $(c + 4)k$ outputs for each cell where k is the number of anchor boxes per cell. The equations for finding the bounding box coordinates are given by:

$$b_x = p_x \cdot a_w + a_x \tag{3.7}$$

$$b_y = p_y \cdot a_h + a_y \tag{3.8}$$

$$b_w = a_w e^{p_w} \tag{3.9}$$

$$b_h = a_h e^{p_h} \tag{3.10}$$

where b_x and b_y are the centers of the predicted bounding box, b_w and b_h are the width and height of the predicted bounding box, a_x and a_y are the image coordinate centers of the anchor box center, a_w and a_h are the anchor box's width and height, b is the predicted bounding box, and p_x, p_y, p_w, and p_h are the outputs of the network. A comparison of the network outputs is shown in Figure 3-2.

The anchor box scale is chosen to be different for each size feature layer so that each layer learns to correspond with differently scaled objects in the image. The aspect ratios of the anchor boxes are selected to be $1, 2, 3, \frac{1}{2}$, and $\frac{1}{3}$. Equation 3.11 sets the scale, s_k, for the anchor boxes of feature layer m:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), \; k \in [1, m] \tag{3.11}$$

where s_{min} and s_{max} are set to be 0.2 and 0.9 respectively. A sixth anchor box is also created with an aspect ratio of 1 and scale according to Equation 3.12.

$$s'_k = \sqrt{s_k s_{k+1}} \tag{3.12}$$

The width and height of the anchor boxes are given by Equations 3.13 and 3.14 where a is an aspect ratio from the set above.

$$w_k^a = s_k \sqrt{a} \tag{3.13}$$

$$h_k^a = \frac{s_k}{\sqrt{a}} \tag{3.14}$$

Training SSD involves matching ground truth bounding boxes with anchor boxes. First, the anchor box that has the highest IOU with a ground truth bounding box is found. Then, all other anchor boxes with an IOU greater than 0.5 are also matched to a given ground truth bounding box. Adding additional matching anchor boxes helps to simplify what the network has to learn and the network can predict more scores for these overlapping bounding boxes.

The loss function for SSD is similar to the loss function from the MultiBox detector [10], but includes more classes. The high level loss function is a weighted combination of a confidence loss, L_{conf}, and a localization loss, L_{loc}. The loss function is shown in Equation 3.15 where x is 1 if an anchor box is matched with a ground truth and is 0 otherwise. c is the class confidence, l is the predicted bounding box, g is a ground truth box, and α is a weight assigned to the localization loss. The confidence loss is a softmax loss over the class confidences and the localization loss is a smooth L1 loss as in [20].

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \tag{3.15}$$

Since the anchor boxes tile the entirety of the feature maps, there are a lot of negative examples presented to the network. To help alleviate this bias, the authors set the ratio between negative and positive examples to be 3:1 by ordering the anchor boxes from highest confidence loss to lowest confidence loss and they pick the highest boxes until they reach the desired ratio.

During training, the data shown to the network is augmented to help it generalize. The data is augmented by switching between using the original image, sampling a patch of the image with a minimum IOU with an object, and randomly sampling a patch from the image. To constrain the augmentation patches range in scale from 0.1 to 1 and the aspect ratio is kept between $\frac{1}{2}$ and 2. On top of these augmentations, some of the images are also flipped horizontally and undergo photometric distortions.

The standard version of SSD uses VGG-16 [64] as its base network feature extractor. VGG-16 is a 16-layer cousin of a classification network that won the 2014 ImageNet Challenge. The idea behind the design of the VGG networks is that they use 3×3 filters, which are small but also still capture the concepts of up, down, left, and right. They then stack these filters with max pooling layers and end with three fully connected layers. The authors of SSD make a few changes to the original VGG-16 network to incorporate it into the detector by changing two of the fully connected layers to convolution layers and removing the dropout layers and the last fully connected layer.

3.3.1 MobileNetSSD

While the authors of SSD used VGG-16 as their feature extractor, advancements in CNN network architectures have created faster and more efficient networks. One such development was the creation of MobileNets by Howard et al. [27] for the purpose of lightweight applications. The core of MobileNets is the use of depthwise separable convolution layers. Using depthwise separable convolutions is not a particularly new idea and the authors note that it was first introduced in [63]. Depthwise separable convolution is a type of factorization of standard convolution layers requiring less computation and parameters. The factorization divides the standard convolution operation done by a convolution layer into two steps by first applying the depthwise filters to each channel and then combining the output.

The factorization is carried out by breaking apart the standard $D_k \times D_k \times M$ filter, where D_k is the filter size and M is the input channel size, used to create each new layer of the feature map output. The filter is broken into M number of $D_k \times D_k \times 1$ filters which are applied to each channel of the input. The second piece is a $1 \times 1 \times M$ filter applied to the output of the M number of $D_k \times D_k \times 1$ filters effectively creating a linear combination of the output. For a full convolutional layer of N number of filters, this turns into M number of $D_k \times D_k \times 1$ filters followed by N number of $1 \times 1 \times M$ filters, also known as pointwise convolutions. Each set of filters is its own layer followed by a batchnorm layer and a ReLU nonlinearity layer. This factorization can be seen in Figure 3-3.

The savings of this factorization can be easily calculated as shown by Howard et al. [27]. The computational cost of a standard convolutional layer is shown in Equation 3.16 where $c_{standard}$ is the cost of a standard convolutional layer and D_f is the size of the input and output assuming a stride of 1.

$$c_{standard} = D_k \cdot D_k \cdot M \cdot N \cdot D_f \cdot D_f \tag{3.16}$$

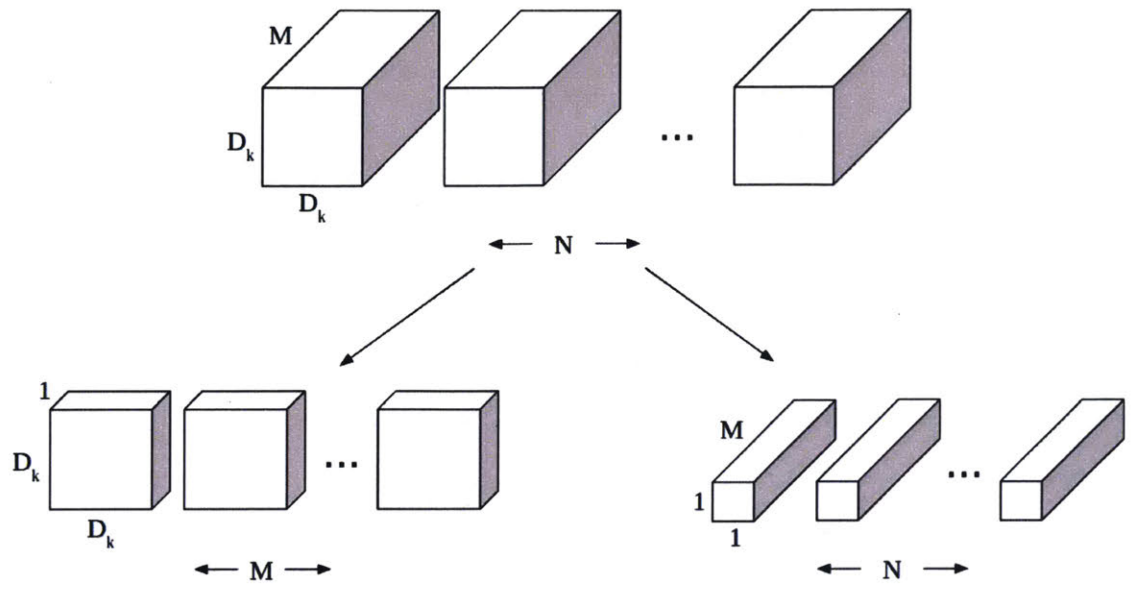


Figure 3-3: The standard size convolutional layer (top) is factorized into depthwise convolutions (bottom left) and 1×1 convolutions (bottom right).

The computational cost of a depthwise separable convolutional layer is given by Equation 3.17 where c_{dws} is the cost of a depthwise separable convolutional layer.

$$c_{dws} = D_k \cdot D_k \cdot M \cdot D_f \cdot D_f + M \cdot N \cdot D_f \cdot D_f \tag{3.17}$$

In Equation 3.17, $D_k \cdot D_k \cdot M \cdot D_f \cdot D_f$ is the cost of the depthwise convolution and $M \cdot N \cdot D_f \cdot D_f$ is the cost of the N number of $1 \times 1 \times M$ convolutions. By taking the ratio of the cost of the depthwise separable convolutions and the full convolutions we can find the computational savings of this factorization shown below:

$$\frac{D_k \cdot D_k \cdot M \cdot D_f \cdot D_f + M \cdot N \cdot D_f \cdot D_f}{D_k \cdot D_k \cdot M \cdot N \cdot D_f \cdot D_f} \tag{3.18}$$

which can be simplified to:

$$\frac{1}{N} + \frac{1}{D_k^2} \tag{3.19}$$

The authors compared several MobileNet classifiers to other high performing classifiers like VGG-16. In addition to making a standard size classifier, the authors also tested shallower and skinnier networks. When compared to VGG-16, the standard MobileNet classifier network uses 569 million multiply-add operations while VGG-16 uses 15300 million. Additionally, the MobileNet network uses 4.2 million parameters while VGG-16 uses 138 million. MobileNet reduces the multiply-add operations by more than $\frac{1}{26}$ and reduces the parameters by $\frac{1}{32}$. The cost to accuracy for using fewer parameters is only 0.9% with VGG-16 scoring 71.5% accuracy on ImageNet and MobileNet scoring 70.6%.

The standard size MobileNet architecture uses a full sized convolutional layer for the first layer followed by alternating depthwise convolutions and pointwise convolutions. There are a total of nine pairs of depthwise and pointwise convolutions followed by an average pooling layer, a fully connected layer, and then a softmax layer. To incorporate this network into the SSD architecture, the pooling layer, fully connected layer, and softmax layer were removed. Then four extra depthwise separable layers were added.

3.4 Test Setup

Each of the four object detection networks described in this chapter were tested across five hardware platforms: a full desktop setup with a NVIDIA TitanX GPU and Intel Core i7-6700K CPU, a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon. The desktop setup and Jetsons had NVIDIA GPUs allowing the use of the NVIDIA CUDA framework to exploit GPU computation. The original networks from the authors' Github repositories and website [54, 41, 3] were used for testing on the desktop and Jetsons because they were implemented on platforms that used CUDA. To be used by the Intel Movidius Stick, the networks had to be converted to a special graph format. Caffe implementations of all the networks were used for the evaluation done on the NUC since it does not have an NVIDIA GPU. All the networks were evaluated using weights trained on the PASCAL VOC 2007 and 2012 training sets to detect 20 classes.

We evaluated each of the networks on its runtime frame rate, GPU usage, CPU

Table 3.2: Intel Core i7-6700K CPU Specifications

Cores	4
Process Base Frequency	4.00 GHz

Table 3.3: NVIDIA Titan X GPU Specifications

TFLOPs	11
CUDA Cores	3584
Base Clock Rate	1417 MHz

usage, and memory usage. Each network was tested in its GPU configuration and its CPU-only configuration. The frame rate timing on all the platforms except for the NUC included the pre- and post-processing steps for the detector consisting of image resizing and normalization, raw network output to detection bounding boxes, and non-maximum suppression. For timing, we used the same timer as the authors of GOTURN [24]. Additionally, the evaluations on all the hardware, except for the NUC, processed images from a webcam through the network. The evaluation on the NUC used the Caffe timing tool, which does not include the pre- and post-processing steps.

The following sections will describe the testing setup for each platform and the methods used for gathering the evaluation data. The final section will give an overview of the cumulative results.

3.5 NVIDIA TitanX GPU and Intel Core i7-6700K CPU Network Evaluation

The full desktop machine we ran our initial evaluation on was able to utilize a NVIDIA TitanX GPU and an Intel Core i7-6700K CPU. The specifications for each are shown in Tables 3.2 and 3.3. This setup used the CUDA toolkit version 8.0 from NVIDIA [49].

We used this setup to be our baseline performance for the networks. The hardware most closely matched those described in the networks' papers. Although this hardware setup was not deployable for our application, it provided what we might expect to be close to peak performance. To test YOLOv2 and Tiny YOLO, we used the original implementation from the author [54]. This implementation was written in C using the author's own deep learning framework called Darknet [55]. The evaluation used the author's detection demonstration, which feeds the network images from a live video stream and shows the detection results. We used a Play Station Eye camera to provide the video.

The results for the SSD networks were collected in a similar way to the YOLO networks. The same type of demonstration was made that would feed the networks images from a video stream captured by the Play Station Eye camera. The demo was written in C++ and the networks were implemented using the Caffe deep learning framework using the C++ API.

Across the four networks evaluated on this setup, we collected results for the networks' frame rate in frames per second (fps), GPU usage as a percentage, GPU memory usage in MiB, CPU usage as a percentage, and CPU memory as a percentage. The frame rate was recorded by taking the average fps of the network over one minute of runtime. The fps was recorded after the network had already processed 30 frames in order to discard any extraneous initial inference rates. The GPU usage was recorded using the nvidia-smi tool [47]. The GPU usage was queried approximately every 15 seconds for five samples and then averaged to create the final usage percentage. The GPU memory usage was recorded the same way as the GPU usage. CPU usage and CPU memory usage were recorded using the htop tool [26] in the same way as the GPU measurements in 15 second intervals. For GPU and CPU measurements, the network was allowed to "warm up" for more than 10 seconds to remove any initial transient values. Any reported values above 100% for the CPU usage means that more than one core is being used. The results of the evaluation on this setup are shown in Tables 3.4 and 3.5. Table 3.4 shows the results when the network is able to utilize the GPU and Table 3.5 shows the results when the network is only allowed to use the CPU.

Table 3.4: Desktop GPU Network Evaluation Results

Network	FPS	GPU Usage	GPU Memory Usage [MiB]	CPU Usage	CPU Memory Usage
YOLOv2	32.5	32.80%	1015	116.80%	3.50%
Tiny YOLO	31.9	18.20%	553	113.60%	2.70%
SSD	45.6	34.40%	1234	52.50%	2.10%
MobileNetSSD	40	32.40%	532	50.60%	1.70%

Table 3.5: Desktop CPU Network Evaluation Results

Network	FPS	CPU Usage	CPU Memory Usage
YOLOv2	0.3	100.00%	1.70%
Tiny YOLO	1	101.20%	0.80%
SSD	0.4	99.28%	4.30%
MobileNetSSD	6.5	84.86%	2.10%

The baseline results are not exactly what was expected. From their respective papers we would have expected YOLOv2 and Tiny YOLO to outmatch the default SSD network in terms of speed, but the SSD network had the highest frame rate on this setup. This discrepancy of reported tested frame rates can be explained by the GPU usage of each network. None of the networks used the full GPU, and SSD used the most amount of the GPU at 34.40%. Had the other networks been utilizing more of the GPU, it would have been expected that they could more closely reach their reported speeds. The hardware for this setup was kept at its default values for this evaluation and was not optimized. Another result to note is the GPU memory usage for each network. As expected, the smallest networks, Tiny YOLO and MobileNetSSD, used the least amount of memory and SSD used the most.

Running the networks in CPU-only mode was also illuminating. Here, MobileNetSSD outperformed the other networks in frame rate and CPU usage. MobileNetSSD was over 16× faster than the default SSD and was over 6× faster than Tiny YOLO. Additionally, MobileNetSSD used the least amount of the CPU.

3.6 NVIDIA Jetson TX1 Network Evaluation

After the initial baseline was taken with the full desktop setup, we tested the networks on a NVIDIA Jetson TX1. The Jetson TX1, albeit small, is still a very capable computing device popular for applications with size, weight, and power constraints. The size of the TX1 itself measures a mere 50mm×87mm. We tested the Jetson TX1 using its development board. An image of the testing setup is shown in Figure 3-4. The Jetson TX1 specifications are shown in Tables 3.6 and 3.7. The TX1 has a NVIDIA Maxwell GPU and a quad-core ARM Cortex-A57 CPU. Additionally, the TX1 we used was flashed with the NVIDIA Jetpack version 3.2 [46], which comes with the CUDA toolkit version 9.0 [45]. The networks were tested using the same demo methods used on the previous full setup.

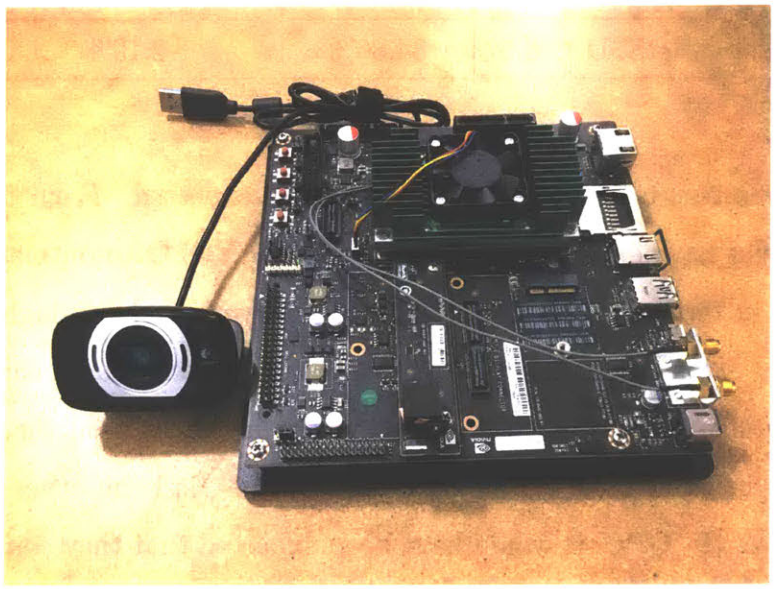


Figure 3-4: The Jetson TX1 setup with development board and camera. The Jetson TX1 was tested with a Logitech webcam.

Table 3.6: QUAD ARM CPU Specifications

Cores	4
Process Base Frequency	1.73 GHz

Table 3.7: Maxwell GPU Specifications

TFLOPs	1
CUDA Cores	256
Max Clock Rate	998 MHz

The Jetson TX1 is a small, but still very capable computing device. However, it is still much smaller than its full-sized Titan X cousin. It has $\frac{1}{14}$ the number of CUDA cores and achieves 11× fewer TFLOPs. Additionally, its CPU cores' clock rates are more than 2× slower than the desktop setup.

We gathered similar statistics from the desktop setup on the TX1 with a few additions. These include the network frame rate (fps), the GPU usage as a percentage, the GPU frequency during runtime, the RAM usage, CPU usage as a percentage, the CPU frequency, and the average usage for each CPU core. The networks were also run and tested in their CPU mode configurations. The results were taken using the tegrastats function [48] on the TX1. The networks were again allowed a warm up period before the results were taken. The data was then taken across 60 seconds with the device being queried roughly every second, and each data point was averaged across the 60 second interval. The results of the network evaluation on the Jetson TX1 are shown in Tables 3.8, 3.9, and 3.10.

Table 3.8: Jetson TX1 GPU Network Evaluation Results

Network	FPS	GPU Usage	GPU Frequency [MHz]	RAM Usage [MB]	Average CPU Usage	CPU Frequency [MHz]
YOLOv2	4.5	91.17%	998	2767.73	36.35%	1734
Tiny YOLO	13.3	88.35%	998	2213.02	52.42%	1734
SSD	10.34	70.25%	964.67	2479	14.03%	311.1
MobileNetSSD	6.5	50.72%	587.1	1849	30.64%	1465.4

Table 3.9: Jetson TX1 GPU Network Evaluation Results (CPU Core Usage)

Network	CPU 1	CPU 2	CPU 3	CPU 4
YOLOv2	42.98%	37.62%	35.27%	29.52%
Tiny YOLO	55.98%	53.43%	47.38%	52.88%
SSD	13.15%	10.60%	9.52%	22.85%
MobileNetSSD	37.87%	34.57%	25.97%	24.17%

Table 3.10: Jetson TX1 CPU Network Evaluation Results

Network	FPS	CPU Usage	CPU Frequency [MHz]	CPU 1	CPU 2	CPU 3	CPU 4
YOLOv2	0.10	26.34%	1734	43.53%	10.35%	17.53%	33.95%
Tiny YOLO	0.20	26.02%	1734	23.85%	30.73%	2.78%	46.72%
SSD	0.095	99.83%	1734	100.00%	100.00%	99.97%	99.37%
MobileNetSSD	0.84	98.93%	1734	97.02%	99.98%	100.00%	98.73%

Utilizing the GPU, the Tiny YOLO network ran at the highest frame rate of 13.3 fps; the next fastest was the default SSD network. The least demanding network, however, was MobileNetSSD, which only consumed about 50% of the GPU even while the GPU was not operating at its highest clock rate. Additionally, MobileNetSSD used the least amount of RAM to run.

In CPU-mode, MobileNetSSD performed the best. Using the Jetson's CPUs, it was able to achieve a frame rate of 0.84 fps. One difference to note between the Caffe framework and Darknet, is that YOLO and Tiny YOLO (both implemented using Darknet) only used one core at a time while the Caffe implementations of SSD and MobileNetSSD would use all the available CPU cores.

3.7 NVIDIA Jetson TX2 Network Evaluation

After testing on the Jetson TX1, we tested the next compact computing device developed by NVIDIA, the Jetson TX2. An image of the TX2 is shown in Figure 3-5. Instead of using a development board as we had done with the TX1, we used the Connect Tech Orbitty carrier board with the TX2. This configuration shows off the compact size of the TX2. The TX2 uses a Pascal GPU and a Dual Denver CPU and a Quad ARM CPU for a total of six CPU cores. The TX2 also came pre-flashed using Jetpack version 3.1 [50], which uses the CUDA toolkit 8.0. The CPU and GPU specifications are shown in Tables 3.11 and 3.12. NVIDIA is less specific on the TFLOP capability of the TX2, but NVIDIA does report that it can do >1 TFLOP, making it more powerful than the TX1. The clock rates for both the CPUs and GPU are also greater than the TX1. In addition, the TX2 has two more CPU cores.

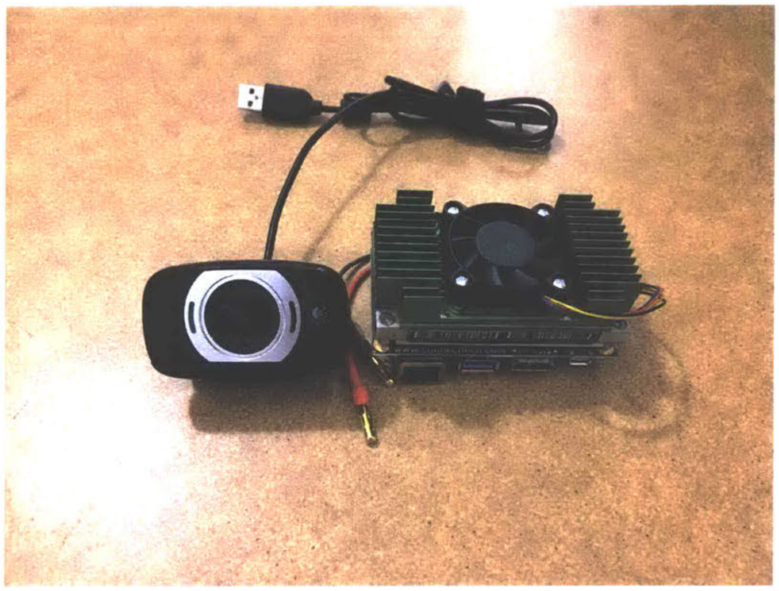


Figure 3-5: Jetson TX2 with Connect Tech Orbitty Carrier Board. The TX2 was also tested with the Logitech webcam shown on the left.

Table 3.11: QUAD ARM and Dual Denver CPU Specifications

Cores	6
Process Base Frequency	2.0 GHz

Table 3.12: Pascal GPU Specifications

TFLOPs	>1
CUDA Cores	256
Max Clock Rate	1300 MHz

The same statistics for the networks were taken with the TX2, network frame rate, GPU usage, GPU operating frequency, RAM usage, CPU usage, and CPU operating frequency. Additionally for the TX2, we tested the networks using the TX2's jetson_clocks function, which maxes out the GPU and CPU clocks. Both of these results are shown below. We gathered these values using the same method used on the TX1.

Table 3.13: Jetson TX2 GPU Network Evaluation Results

Network	FPS	GPU Usage	GPU Frequency [MHz]	RAM Usage [MB]	Average CPU Usage	CPU Frequency [MHz]
YOLOv2	6.1	95.60%	1132.3	3283.7	25.02%	2028.97
Tiny YOLO	14.6	76.48%	1069.4	3142	33.05%	2013.13
SSD	4.37	76.83%	222.8	2946	7.16%	358.47
MobileNetSSD	4.97	90.33%	137.8	2820.17	11.95%	604.38

Table 3.14: Jetson TX2 GPU Network Evaluation Results (CPU Core Usage)

Network	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6
YOLOv2	33.55%	0.00%	0.00%	56.10%	32.47%	28.03%
Tiny YOLO	56.97%	0.00%	0.00%	50.27%	45.43%	45.61%
SSD	8.42%	0.00%	0.00%	3.93%	14.73%	15.58%
MobileNetSSD	18.73%	0.00%	0.00%	13.28%	20.52%	19.18%

Table 3.15: Jetson TX2 CPU Network Evaluation Results

Network	FPS	CPU Usage	CPU Frequency [MHz]	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	RAM Usage [MB]
YOLOv2	0.10	18.05%	2027	1.70%	0.00%	0.00%	61.07%	40.93%	4.62%	3113
Tiny YOLO	0.20	26.02%	2026	9.53%	0.00%	0.00%	41.58%	45.28%	7.35%	2818
SSD	0.11	99.83%	2026.05	99.45%	0.00%	0.00%	99.75%	100.00%	100.00%	3696
MobileNetSSD	0.92	98.93%	2027.7	95.38%	0.00%	0.00%	100.00%	100.00%	100.00%	2995

Table 3.16: Jetson TX2 GPU Network Evaluation Results with Max Clock Rate

Network	FPS	GPU Usage	GPU Frequency [MHz]	RAM Usage [MB]	Average CPU Usage	CPU Frequency [MHz]
YOLOv2	6.1	96.27%	1134	3335	24.92%	2027
Tiny YOLO	14.9	76.43%	1134	3195.45	33.43%	2029.73
SSD	27.7	82.70%	1134	2952	3.62%	2027.03
MobileNetSSD	10.99	54.53%	1134	2827	12.55%	2028.77

Table 3.17: Jetson TX2 GPU Network Evaluation Results (CPU Core Usage) with Max Clock Rate

Network	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6
YOLOv2	30.67%	0.00%	0.00%	56.52%	33.97%	28.37%
Tiny YOLO	51.10%	0.00%	0.00%	48.52%	54.95%	46.00%
SSD	6.50%	0.00%	0.00%	3.78%	5.58%	5.58%
MobileNetSSD	31.52%	0.00%	0.00%	9.95%	15.08%	18.75%

Table 3.18: Jetson TX2 CPU Network Evaluation Results with Max Clock Rate

Network	FPS	CPU Usage	CPU Frequency [MHz]	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	RAM Usage [MB]
YOLOv2	0.10	17.57%	2027.58	32.05%	0.00%	0.00%	2.87%	39.76%	30.72%	3148.05
Tiny YOLO	0.20	17.29%	2029.22	31.55%	0.00%	0.00%	20.20%	13.80%	38.20%	2853
SSD	0.11	66.52%	2029.93	100.00%	0.00%	0.00%	100.00%	99.10%	100.00%	3704
MobileNetSSD	0.95	65.72%	2027.17	99.48%	0.00%	0.00%	100.00%	94.90%	99.93%	3004.08

These results show that the TX2 is indeed more powerful than the TX1, as all the networks showed an increase in frame rate. When the TX2 is at maximum GPU clock rate, SSD shows a vast increase in frame rate. This is a surprising result, although it seems reasonable that at the nominal clock setting it only uses 76% of the GPU at 222.8 MHz. Even though MobileNetSSD did not have the highest frame rate, it was still very efficient at both clock rates in its frame rate to resource usage.

The results also show that the two Denver cores were not used in this evaluation. The power and rate settings were the default settings for the TX2. We tested a few of the other power settings that used the two Denver CPU cores, but using the four ARM cores showed competitive results since they were still operating at high clock rates. More testing with the other power settings is left for future work.

3.8 Intel Movidius Neural Compute Stick Network Evaluation

Moving away from NVIDIA specific hardware, we also tested the detection networks on a new deployable hardware aimed at deep learning applications made by Intel called the Movidius Neural Compute Stick. The Movidius Stick is the size of a large flashdrive making it ideal for size-, weight-, and power-constrained applications. The Movidius Stick also only needs a USB port to communicate with the host hardware. The neural compute stick is shown in Figure 3-6. Similar to CUDA cores, at the heart of the Movidius Stick are 12 SHAVE processors that can parallelize the network computations in the stick's Vision Processing Unit or VPU. Intel reports that this compact device can achieve 100 GFLOPS [29].

The compute stick currently supports networks implemented with Caffe and Tensorflow [1], another popular deep learning framework. .Within these two frameworks, the Movidius Stick API only supports certain network layers. This is very limiting for this hardware since it cannot support custom layers, which are popular for new emerging networks. The custom layers can't be used because the weights files for the networks, for example a ".caffemodel" for Caffe, needs to be converted into a "graph" that the Movidius Stick can use. This is also limiting for using YOLO networks which to be used by the Movidius stick, first need to be converted into either a Caffe or Tensorflow model and then into a graph. Luckily, the Movidius Stick has support for SSD and MobileNetSSD, as well as Tiny YOLOv1. It does not, however, have official support for either YOLOv2 or Tiny YOLOv2 at the time of this writing. There is also no "CPU Only" configuration using the Movidius stick so that the only collected statistics are frame rate, CPU usage, and CPU Memory usage of the host machine, which was the same as the full setup described earlier. The results of the Movidius Stick evaluation are shown in Table 3.19.

Figure 3-6: Intel Movidius Neural Compute Stick

Table 3.19: Intel Movidius Neural Compute Stick Network Evaluation Results

Network	FPS	CPU Usage	CPU Memory Usage
YOLOv2	Not Supported	Not Supported	Not Supported
Tiny YOLOv2	Not Supported	Not Supported	Not Supported
Tiny YOLOv1	7.3	27.72%	0.90%
SSD	0.74	10.84%	0.80%
MobileNetSSD	10.6	27.18%	0.70%

Again using this hardware MobileNetSSD outperforms the other networks in terms of framerate. The networks also put little load on the on the host machine during runtime.

3.9 Intel NUC Skull Canyon Network Evaluation

Our last evaluation was done on an Intel NUC Skull Canyon. Like the Movidius Stick, the NUC doens't have any NVIDIA hardware. This creates a significant problem for neural network deployment since most popular deep learning frameworks utilize the CUDA framework and NVIDIA GPUs. The NUC, on the other hand, uses an Intel Iris Pro Graphics 580. Additionally, in order to use this, the deep learning framework needs to use OpenCL instead of CUDA. Currently, there are few stable deep learning frameworks that use OpenCL. There is, however, a fairly stable version of Caffe, clCaffe [71], that uses an OpenCL backend that can utilize the NUC's GPU. This OpenCL distribution of Caffe also supports all four of the networks we are evaluating. The NUC uses an Intel Core i7-6770HQ Quad Core CPU. The CPU and GPU specifications are shown in Tables 3.20 and 3.22. Figure 3-7 shows the Intel NUC Skull Canyon setup.

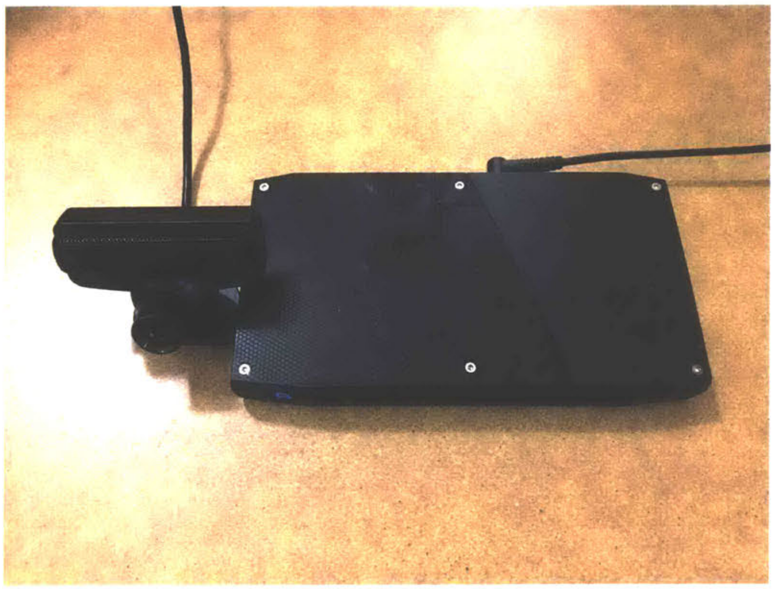


Figure 3-7: Intel NUC Skull Canyon setup with Play Station Eye Camera.

Table 3.20: Intel Core i7-6770HQ Quad Core CPU Specifications

Cores	4
Process Base Frequency	2.60 GHz (max 3.50 GHz)

Table 3.21: Iris Pro Graphics 580 Specifications

TFLOPs	1.15[1] [67]
CUDA Cores	N/A
Max Clock Rate	950 MHz

The statistics on the NUC were taken using the intel_gpu_top tool, which queries the NUC's GPU usage. This tool reports render busy as a percentage, which we have taken to be GPU usage, and render space, which take for memory usage. We also use htop to record CPU usage and CPU memory usage. As before, every 100% of CPU usage is one full CPU core. For example, 350% of CPU usage indicates that 3.5 CPU cores were used. Since we used the Caffe versions of all the networks for the NUC, we used the Caffe timing function for forward network passes to take timing data. The timing results were taken in milliseconds [ms] and were averaged over five samples. The networks were allowed to warm up to prevent recording transient effects. The timing and usage results are shown in Tables 3.22 and 3.23.

Table 3.22: Intel NUC Skull Canyon GPU Network Evaluation Results

Network	Forward Time [ms]	GPU Usage	Render Space Usage	CPU Usage	CPU Memory Usage
YOLOv2	73.43 (13.62 fps)	99.80%	9.61%	78.00%	2.20%
Tiny YOLO	18.62 (53.71 fps)	95.80%	4.63%	77.64%	1.00%
SSD	106.94 (9.35 fps)	100.00%	10.37%	79.36%	1.50%
MobileNetSSD	14.47 (69.11 fps)	95.20%	7.50%	99.96%	0.50%

Table 3.23: Intel NUC Skull Canyon CPU Network Evaluation Results

Network	Forward Time [ms]	CPU Usage	CPU Memory Usage
YOLOv2	2101.71 (0.48 fps)	770.60%	2.20%
Tiny YOLO	569.89 (1.75 fps)	771.60%	1.00%
SSD	4170.21 (0.24 fps)	783.20%	1.60%
MobileNetSSD	631.09 (1.58 fps)	783.40%	0.80%

On this setup, MobileNetSSD and Tiny YOLO were the top performers in terms of inference time and render space usage. MobileNetSSD edges out Tiny YOLO in frame by a little more than 15 fps. The results also show that YOLOv2 outperforms SSD, but these two networks are still much slower to their lighter counterparts.

[1]Value not from manufacturer

3.10 Testing Results

Altogether, this evaluation covered 43 different network and hardware configurations. The tested hardware spanned from our baseline full desktop setup to possible deployable hardware including a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon. Based on the evaluation results and the existing hardware already on the flight vehicle, we decided to move forward with the MobileNetSSD detector on the Intel NUC Skull Canyon. The NUC was already being used on the flight vehicle as the main computer for the planning and state estimation stacks. The NUC's GPU was also free real estate in the sense that no other processes were directly using it. There was the possibility that a monocular state estimation pipeline might come online and utilize the GPU, so it was still important the detector not take up the entire GPU during run time. This will be discussed more in the next chapter.

Although the Movidius Stick was also a very competitive choice to use with the MobileNetSSD detector, we decided not to use it and opted for the higher frame rate on the NUC so that we could add other components to the detection pipeline for temporal consistency and maintain a large margin to keep the system running in real-time. This is also part of the reason we did not use the YOLOv2 detector. The other is that the MobileNetSSD detector was the most flexible in terms of its performance across different hardware, making it robust to an ever-evolving flight vehicle. Additionally, MobileNetSSD was still competitive with the other detectors in terms of accuracy. Table 3.24 shows a summary of the reported accuracy results of the detectors on the PASCAL VOC dataset where all the networks were trained on a concatenation of the 2007 and 2012 training and validation data.

Table 3.24: Network VOC07+12 Accuracy [mAP]

Network	Accuracy [mAP]
YOLOv2	76.8 [57]
Tiny YOLO	57.1 [54]
SSD	74.1 [42]
MobileNetSSD	68 [3]

Using MobileNetSSD as our detector only comes at a small cost to accuracy from the most accurate, YOLOv2. MobileNetSSD only decreases in mAP by 8.8, but we get a 5$\times$ speed up on the NUC. However, when pretrained on the Microsoft Common Objects in Context (COCO) dataset [40], MobileNetSSD was able to achieve a mAP of 72.7. Additionally, MobileNetSSD was shown to have one of the smallest memory demand across the devices showing the other benefit of the depthwise separable convolution factorization.

Despite the expansive evaluation across hardware and a few configurations, there are still some shortcomings of this evaluation. For one, the hardware configurations were not optimized. This can be seen in the results for the full desktop setup where the networks were not fully utilizing the hardware. Additionally, we did not look at how each network scaled with the number of classes it was trained on. We did train a YOLOv2 network and an SSD network on five classes. However, the network structure was not downsized to account for the reduction in classes. The final MobileNetSSD network we used for deployment was downsized for our reduction in classes, which we will show later. We leave the hardware optimization and network class size evaluation for future work.

Chapter 4

Object Detection Flight Vehicle Deployment

This chapter will give an overview on how MobileNetSSD was deployed as the object detector for the FLA program to fulfill our exploration and semantic mapping mission. The full FLA quadcopter is shown in Figure 4-1. The first step for deployment was to train the network on the classes of interest. There were several iterations of object classes, but the latest network was trained to detect people, cars, doors, and windows. After training, we installed the necessary software on the Intel NUC Skull Canyon to use the OpenCL backend of the clCaffe deep learning framework.

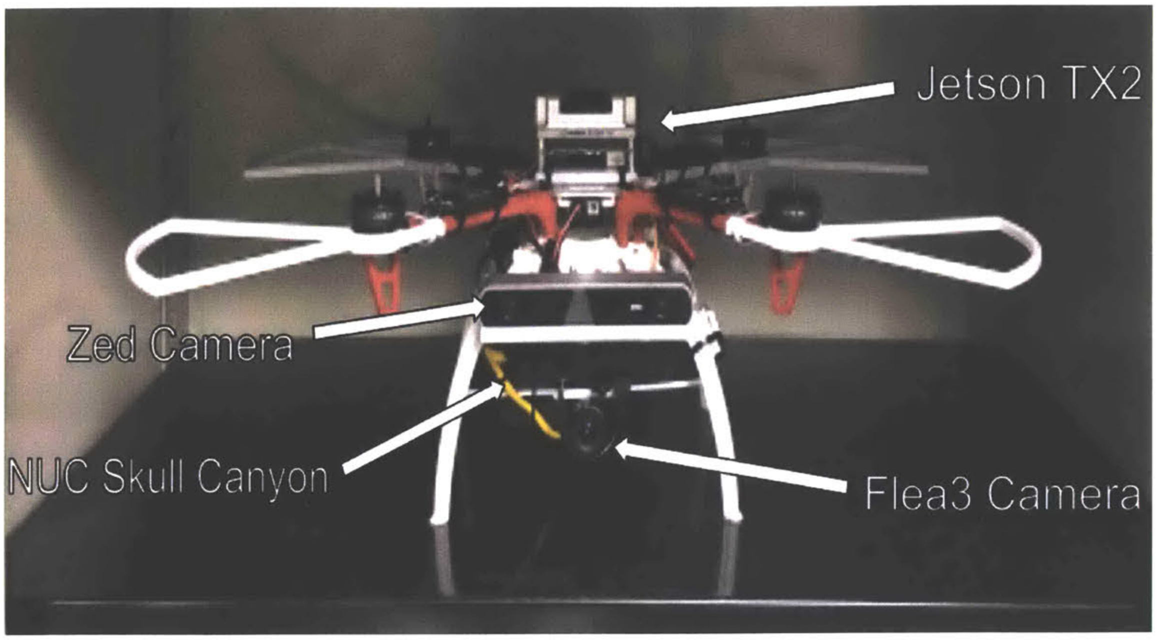


Figure 4-1: FLA Quad setup. The main components are labeled including the Jetson TX2, the Zed Camera (runs on the TX2), the Flea3 Camera, and the Intel NUC Skull Canyon.

4.1 Training MobileNetSSD

The first step to deploying MobileNetSSD for the mission was to train it on our objects of interest. Finding relevant training data for CNNs can be difficult. Training datasets for the object detection task are especially sparse. For example, the image classification task only requires a minimum of one label per image. The detection task, however, requires a bounding box and a label for each object of interest in the image. An example of an annotated image is shown in Figure 4-2. Figure 4-2 shows several hand-labeled annotated objects in an image including a door and several vehicles.

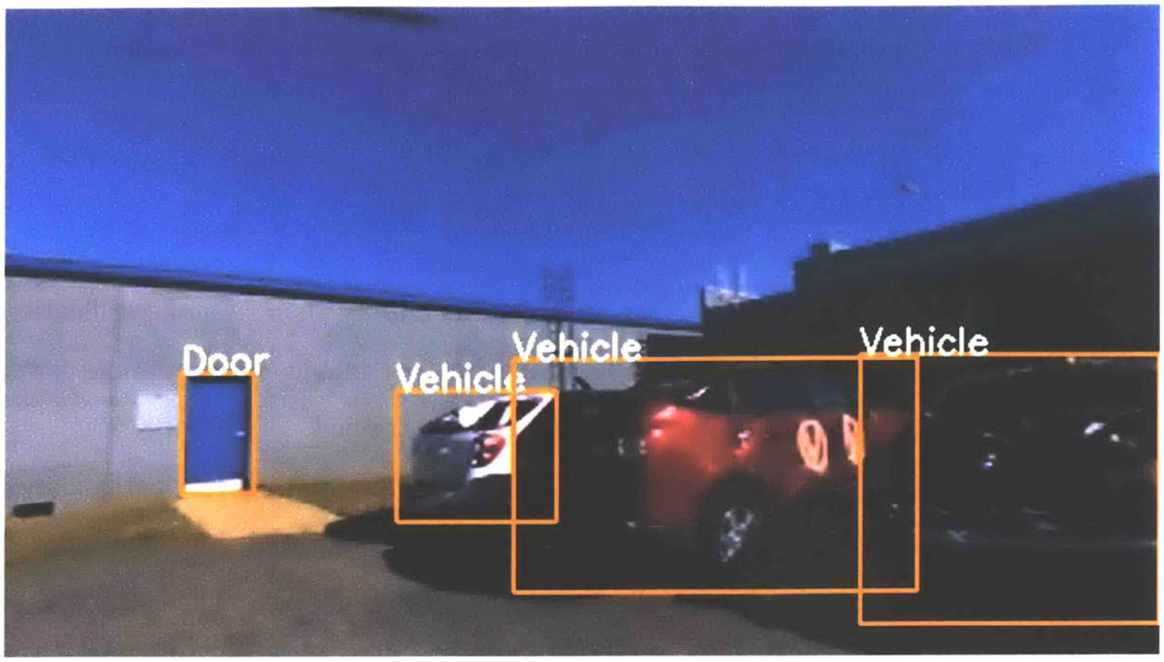


Figure 4-2: An example of an annotated image. This image was created using the Figure Eight annotation platform. The image was created using visualization code provided by Figure Eight.

4.1.1 Dataset: OpenImages

We used the OpenImages dataset [37] to train the network. At the time of training, the V2 dataset of OpenImages was the available version. The most recent version of the dataset is V4 at the time of this writing. In total, the V2 dataset boasts having 545 trainable classes with a total of over 669,000 images and over 1.2 million bounding boxes in the training set alone.

Papadopoulos et al. in [51] describe the method they used to create the V2 dataset. The dataset was machine-created and human-validated, thus reducing the bounding box annotation time by 6-9 times. They annotated the dataset by creating a feedback loop between a learner and humans. In a standard loop of their method, localized bounding boxes were verified by human annotators. The positively and negatively determined bounding boxes were then used to train the learner and reduce the object localization space for possible objects, respectively. With this method, no human

Table 4.1: OpenImages Downsampled Dataset and Class Distribution

Image Sets	Total	Person	Door	Car	Window
Test	32675	39906	517	28195	4994
Train	86563	57796	1792	22582	6869
Validation	10892	12978	160	9182	1603

had to draw a bounding box. However, in order to simplify their human verification step, their method only produces one bounding box per image. This presents fewer positive examples to the object detector that is being trained.

The authors verified their annotation method by comparing the performance of two object detectors—one trained on their machine-annotated, human-validated dataset and the other trained on a fully human-annotated dataset. The learner, Fast R-CNN [20] was trained with one annotation per image from both the fully annotated set and the human verified set. The loss in accuracy between the fully annotated data and the human verified only dropped by 6%. Although there were some shortcomings to this dataset, mainly that every object is not annotated, it did offer a wide range of classes and training examples. Using this dataset allowed us to keep the same training pipeline for a range of different object classes as opposed to combining different datasets.

In order to use this dataset, we took a subsample of the full dataset containing only our desired classes. This downsized dataset contained almost 100,000 images on which the network could be trained. The distributions of the image sets and class instances of the downsampled dataset are shown in Table 4.1.

The data was converted from the OpenImages CSV format to the VOC [15] XML annotation format. Once the data was converted to the VOC format, it could be easily integrated into the existing training procedure used by MobileNetSSD.

4.1.2 Training Setup

The first step to training the MobileNetSSD detector was to adjust the network parameters to accommodate the reduction in detectable classes from 20 to 4. There is a convenient template for adjusting the network parameters available from [3]. We

Table 4.2: MobileNetSSD mAP on OpenImages

Class	mAP
Person	0.40
Door	0.26
Car	0.70
Window	0.14

also used the pretrained weights from the same Github repository. The pretrained model was first trained on the Microsoft Common Objects in Context (COCO) dataset [40] and then trained on the 2007 and 2012 VOC datasets [15].

Our final model was trained for 120,000 iterations with a batch size of 24. The training was done using RMSProp [19], a method that helps with network optimization by normalizing the back-propagated gradients. RMSProp normalizes the gradient with a running average of gradients for a particular weight so that one gradient doesn't overpower others. The accuracy of the final trained network is shown in Table 4.2.

The final network accuracy leaves much to be desired. The person and car classes are suitable for deployment, but the network struggles with detecting doors and windows. This also reflects the bias in the training data. The network is shown many more images of people and cars than doors and windows.

4.2 Vehicle Integration

After the network was trained on the specified classes, we needed to integrate the software onto the flight vehicle. This turned out to be quite a challenge since it needed to build easily across all the flight vehicles. Each flight vehicle NUC also needed to have the proper OpenCL driver and SDK versions installed. We used the Intel SRB4.1 Linux driver package [30] and the Intel OpenCL SDK for Ubuntu v7.0 [31]. Additionally, clCaffe required additional dependencies compared to the main branch Caffe. These included ViennaCL [59], a linear algebra library that is functional with both CUDA and OpenCL, and ISAAC [69], an input-aware basic linear algebra subprogram (BLAS).

During runtime, the network could use either color images from the Zed Camera or grayscale images from the Flea3 Camera. Both have shown comparable results to each other. The object detection pipeline was integrated into the overall system using the Robot Operating System (ROS) Kinetic [53].

One issue when deploying the detector, was the input frame rate from the camera. Directly feeding every image from the camera to the detector would create a time lag if the camera frame rate was faster than the inference time of the network. Additionally, if we formulated the object detection ROS node to only accept an image once it was done with the previous frame, the network would consume all the GPU resources. To lower the time-averaged GPU usage and reduce possible time lag, we throttled the incoming image message rate to 30 Hz.

The overall goal for the flight vehicle was to create a semantic map of objects of interest in the environment. The detector would report its detections to the mapper. The mapper, however, required several detections (> 5) in order to localize the object. This also meant that there needed to be a way to filter some of the raw detections from the detector to be used by the mapper. We applied a confidence threshold to each class based on the class's mAP score. The threshold value for each class was determined using Equation 4.1, where T_l is the per-class threshold and m_l is the mAP value for each class on the OpenImages V2 test set.

$$T_l = \frac{1 - m_l}{2} \tag{4.1}$$

Setting the per-class confidence thresholds using this Equation 4.1 provided promising results in reported detections as shown in Chapter 5. We deployed this configuration of MobileNetSSD on the Intel NUC Skull Canyon for several missions. However, one weakness of this system was that MobileNetSSD only operates on single images. There is a lot of information that is lost by using only single camera frames, and the detector will sometimes find an object in one frame and miss it in the next. Chapter 5 discusses how we approached solving this problem and presents an evaluation of our method.

Chapter 5

Towards Temporally Consistent Object Detection

We propose a lightweight method to incorporate temporal information into a detection pipeline running on a live camera feed called Temporal Object Detection or TOD. Figure 5-1 shows the motivation for this type of system. Each row of Figure 5-1 shows consecutive camera frames where a detection in one frame was lost in the next. By leveraging object image representations across camera frames our method can determine if an object was dropped and recover it. We can recover detections by raising the confidence of low confidence detections or, in the absence of any detections, use an object tracker to find the object. Existing methods that address the problem of object detection in video such as [22, 36, 34, 35] reason over the entire video sequence at run time to find objects. Other methods like [16, 18] build off networks that require a region proposal stage, which is not as efficient as single shot detectors like SSD [42] or YOLO [56, 57]. Our proposed method incorporates a base object detector, a similarity metric based on bounding box size, bounding box dimensions, intersection over union (IOU), and color, and a selectively employed object tracker.

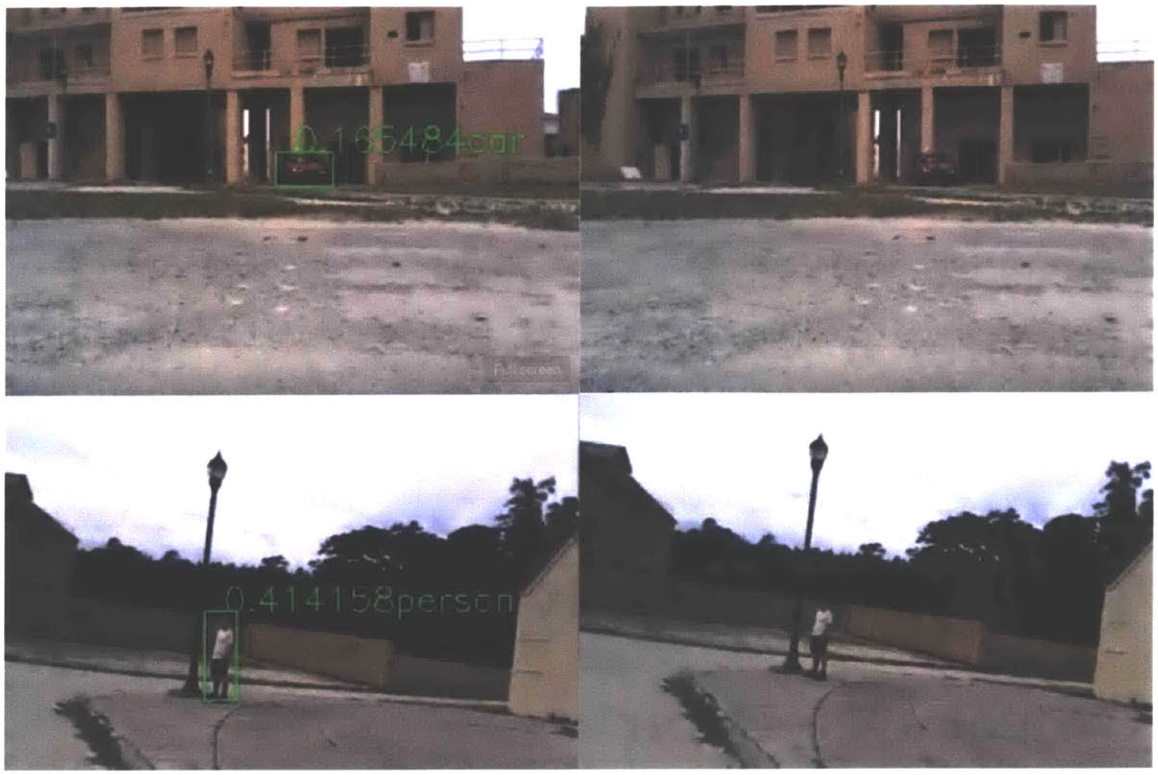


Figure 5-1: The top and bottom rows show consecutive camera frames where the detector was able to detect objects above the confidence threshold, but in the next frame the objects were either not detected or their confidence dropped below the reportable threshold.

5.1 Object States

At the core of our method is the idea of determining the state of an object that the system has previously detected. The object states are detected, in-frame, recovered, tracked, failed, and their inverses. The detected state means that the object has been directly detected by the detector with a confidence above the class threshold set by Equation 4.1. If an object is in the in-frame state, it is likely still within the given image frame. Out-of-frame means that the object is likely now outside the current frame. The recovered state means that the object was found using a low confidence detection. The tracked state means that the object tracker was able to find the object. Lastly, failed means that the system is not able to find the object. This is a similar

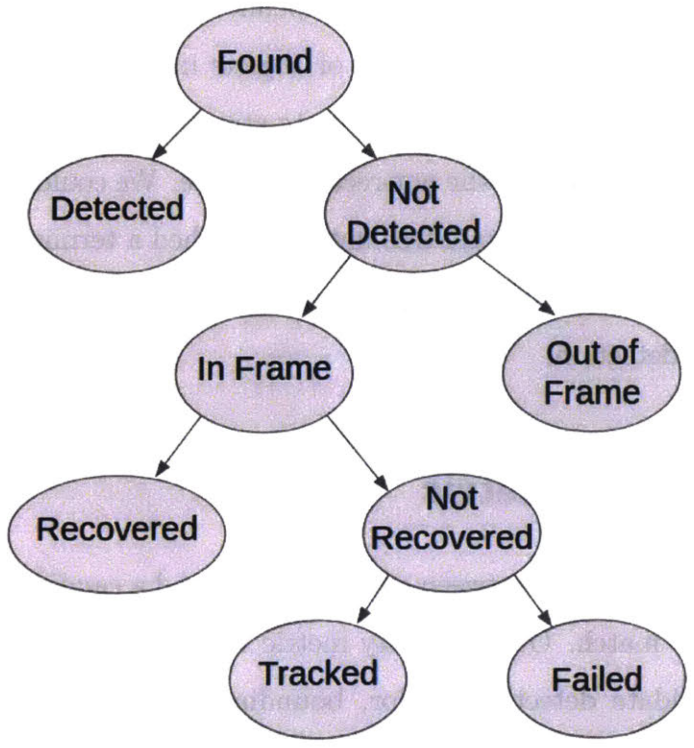


Figure 5-2: This is the representation of the object states used by our method. Every previously found object starts in the found state.

idea used by [76] where they can determine a state of an object using an Markov Decision Process. TOD however does not require learning a policy. The states can be thought of as being conditionally dependent on one another. For example, an object that is recovered must be in-frame. These conditional dependencies are shown in Figure 5-2 where every previously found object starts in the found state. This tree informs how or if TOD searches for an object.

Using this concept of object states, our method reduced the amount of computation needed to find lost objects compared to a method that applies a tracker to every object. For example, Detect and Track [16] finds correlation maps for every location in a feature map. For TOD, since we knew the object distribution was relatively sparse in the environments our vehicle was deployed, we could avoid doing computation over the entire image by selectively applying portions of our object detection pipeline to a subset of the objects in a given frame depending on the state of the object. The object tracking step in our pipeline was the most expensive portion. By only running the tracker on a subset of objects in an image, we could save time and computational resources. Using our state structure, we only applied the object tracker when the object was in the not-recovered state. We could determine an object state by traversing down the state tree until we reached a terminal state. The object state was chosen at each split by using a similarity metric between a target object and a candidate detection.

5.2 Similarity Metric

We used a similarity metric between a target object and a candidate detection to help determine if they match. Our similarity metric was based on the similarity between target and candidate detections' color, bounding box dimensions, IOU, and size. These four components were combined into an overall score to measure the similarity between a target object and candidate detection. The following sections will break down how we calculate the contribution from each one of the metric components.

5.2.1 Object Color

The color similarity component was based off the Hellinger Distance shown in Equation 5.1 where P and Q are discrete distributions and N is the number of bins. The Hellinger Distance gives a distance between these two discrete distributions. We created HSV color space histograms using the H and S channels for each detected object using its bounding box to bound all the pixels used in the histogram.

$$d(P,Q) = \sqrt{1 - \frac{1}{\sqrt{\bar{P}\bar{Q}N^2}} \sum_{i=1}^{N} \sqrt{p_i \cdot q_i}} \tag{5.1}$$

Using color histograms to represent objects allowed the representation to be rotation invariant, a useful property when images are coming from a constantly moving flight vehicle. The color histogram representation however also came with the downside that it lost the spatial component of the object. To maintain some sense of spatial consistency, each object is represented by two histograms, one for the left side of the bounding box and the other for the right. The total score for this component then became the average of the Hellinger Distances of the left and right sides between two object bounding boxes. Additionally, since the Hellinger Distance is a distance metric we instead used $1 - d$ for each side of the bounding boxes, where d is the Hellinger Distance between either the left or right side of two bounding boxes. The distance was calculated using OpenCV [66].

5.2.2 Intersection Over Union (IOU)

Another component of the similarity metric was the intersection over union or IOU of the candidate bounding box and the target bounding box of the previous frame. This component of the metric exploited the idea that the location of an object between consecutive frames would not drastically change. The IOU also accounted for object size in its calculation such that even if two bounding boxes overlap, they also need to have similar sizes to have a large IOU. The IOU value was directly used as part of the overall similarity metric

5.2.3 Bounding Box Dimensions

We used bounding box dimensions as part of the similarity metric since this was a characteristic of the object that should not have large changes between frames. Additionally, while the IOU component took into account bounding box size, it did not account for the dimensions of the bounding box. This component helped push TOD to choose bounding boxes of similar dimensions. This component was calculated by averaging the ratio of the widths and heights of two bounding boxes as shown in Equation 5.2, where w_1 and w_2 are the widths of the two bounding boxes and h_1 and h_2 are the heights of the two bounding boxes. The final component score was then calculated from Equation 5.3 where $\mu = 1$ and $\sigma = 0.25$. These values were chosen empirically.

$$A = \frac{1}{2} \cdot (\frac{w_1}{w_2} + \frac{h_1}{h_2}) \tag{5.2}$$

$$R(A) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(A-\mu)^2}{2\sigma^2}} \tag{5.3}$$

5.2.4 Bounding Box Size

We included the bounding box size into the similarity metric as a weight between the color component, and the IOU and aspect ratio components. Since we knew the relative size of the objects in the environment, we made the assumption that the larger the bounding box was, the closer the vehicle was to the object. Using this bounding box size and object distance relation, we further assumed that objects with smaller bounding boxes would move less between two consecutive frames than objects with large bounding boxes. We exploited this idea to create a weight between the color and dimension/location components of the similarity metric based on the object's bounding box size. We called this weight β. If the bounding box was large, we would rely more on the color component of the metric and if the bounding box was small we would rely more on the dimension/location components of the similarity metric. β was found using a linear function shown in 5.4 where b_A is the bounding box area, I_A is the image area, m is chosen to be 0.8, and c is chosen to be 0.5.

$$\beta = \frac{b_A}{I_A} \cdot m + c \tag{5.4}$$

We also limited β to be between 0.5 and 0.8, so that the metric was more biased towards the color component. Putting all of the components together, the final similarity metric became Equation 5.5 where d is the Hellinger distance between two image crops, R is the aspect ratio component, IOU is the intersection over union of two bounding boxes in consecutive frames, and β is the weight from Equation 5.4.

$$S = d \cdot \beta + \frac{(R + IOU)}{2} \cdot (1 - \beta) \tag{5.5}$$

All the components of the similarity metric were between 0 and 1 so the final metric was also between 0 and 1. Using this metric, our method could compare consecutive bounding boxes to help determine the state of the object.

5.3 Pipeline Overview

At a high level, our method worked by traversing down the tree shown in Figure 5-2 for each previously detected object until it reaches a terminal state. Determining an object state was broken up into several distinct steps. Initially, the system started with no target detections. Once valid detections were made by the base detector that were over the class confidence threshold, we set them up to be targets, meaning they were objects in the current frame that we would like to detect in the next frame. After the initial targets were found, for every target from the previous frame, we used the similarity metric to determine if we successfully detected the target in the current frame by the object detector. If the method determined that the object was dropped, we tried to find the next best detection in the vicinity of the target by using the similarity metric. Then we determined if the object was likely in the camera frame or not. If it was likely in the frame, we looked to see if the similarity metric score was above a certain threshold. If it was above the threshold, we took that detection to be a valid detection. Else, we would try harder to find the object. In this case, we applied an object tracker to find the object. The tracker returned a candidate detection that we scored with the similarity metric and if it was above a threshold we took it as a valid detection. If the candidate detection was below the threshold, we determined that we could not detect the dropped object. With this formulation, we only applied the tracker as a last resort. Each of the following sections will explain each step of the system including finding dropped objects, finding close detections, scoring and selecting detections, determining if an object was in the frame, determining if the object has been recovered, and tracking the object.

5.3.1 Find Dropped Objects

The first step of our method was to determine if the object detector has already found a target object from the previous frame. To do this, we took all the detections output by the base object detector in the current frame that were in the vicinity of the target location from the previous frame. We then scored these detections against the target object using the similarity metric. If the highest scoring detection was above a threshold, we determined that the target object was found. If not, we passed the target object to the next step of the pipeline. Although this step created scores for each detection based on the similarity metric, we always kept detections above the class confidence threshold.

5.3.2 Find Close Detections

In order to further determine the state of a target object, we would see if the object was found by any low confidence detections we would not normally accept. To do this, we first needed to find all the detections in the vicinity of the target object in the current frame. At this step, we looked at all low confidence detections that overlapped the target object bounding box in the current frame and were of the same class of the target and add them to our list of candidate detections.

If there were not any low confidence detections in the vicinity of a target area we ran the object tracker so that the target would have at least one candidate detection.

5.3.3 Determine In-Frame

Next, we would determine if the object was likely to be in the current frame or not. This was done by simply creating a buffer around the edge of the image. If the center of the target object was in this buffer, it was determined that the object was likely out of the frame. We set the buffer to be 15% of the image width and height because it showed good results for the current configuration of the flight vehicle. It was important to have this as part of the pipeline because there was a tendency for TOD to place detections at the edge of the image when the object had already moved

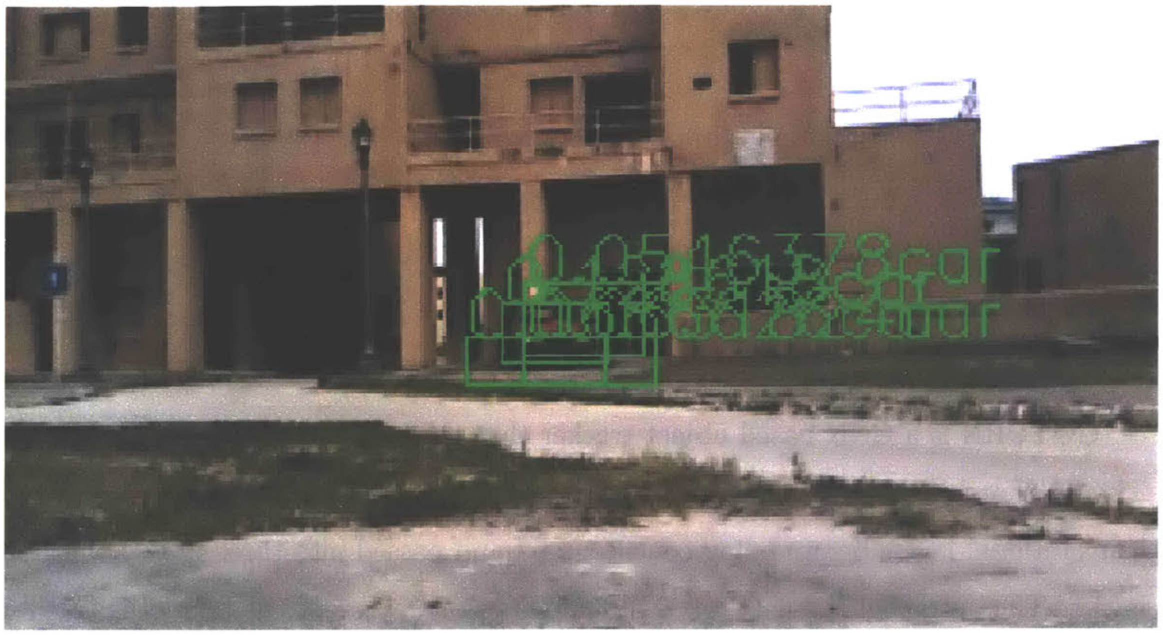


Figure 5-3: An example of multiple overlapping low confidence detections on an object.

out of the frame. The size of the buffer could also be tuned. A smaller buffer could yield a higher recall, but at the cost of some precision.

5.3.4 Object Recovered

Given that the object was still in the frame, we would see if any of our candidate detections found from "find close detections" might be a suitable detection for the target object. A visual depiction of this is shown in Figure 5-3. Figure 5-3 shows that at a low confidence there may be several boxes in the vicinity of an object we would like to detect and we need to pick the best one.

These nearby detections were scored against the target object using the similarity metric and then the highest valued detection was selected as a candidate detection for the target object. If the candidate detection score was above a threshold, we considered the detection a valid detection and it took on the confidence value of its matched target object. If the score was below the threshold, we moved to the tracking step.

5.3.5 Object Tracking

After all the other steps to find the target object had been exhausted, we attempted to find the object with an object tracker. The object tracker could be any algorithm that takes a target image and a search image as input and returns a bounding box. This includes methods such as [25, 2, 73]. For our implementation, we used the GOTURN tracker by Held et al [24].

GOTURN is a CNN based object tracker that takes in two image crops as input. The CNN architecure is unique in that each input is passed through separate parallel convolutional layers before being concatenated and passed into three fully connected layers. The output of the network is four points that represent the bounding box for the tracked object. The network is trained entirely offline on frames from videos and requires no online training. This object tracking method is both fast and accurate. The authors claim that it is capable of running at 100 fps on a GTX 680 GPU. These features made GOTURN a promising component to add as part of the object detection pipeline. Akthough, on the Intel NUC Skull Canyon the GOTURN network had a forward pass time of 30.9 ms. The inference time of GOTURN was a significant cost for the overall pipeline since the inference time of the detector was only 14.5 ms. This is why we wanted to limit the use of the tracker as much as possible.

After the tracker found a bounding box for the object, the candidate detection still needed to be verified. Again we used the similarity metric to score the candidate detection. If the candidate score was above a threshold we took the candidate detection as a valid detection and it took on the confidence value of its matched target object. If the score was not above the threshold, we assumed that the object could not be found. Once we added our final detections, we set these as targets for the next frame.

During testing, we set the threshold for the similarity metric for accepting candidate detections to be 0.75. This threshold could determine how risky the system would be. A lower threshold would accept more detections, while a higher threshold would be more conservative. We found 0.75 to be a good middle ground.

At the end of the pipeline, we needed to add two more post processing steps. Since we knew that the objects in our deployed environment were somewhat sparse, it was unlikely that objects will be stacked or heavily occluding one another. With this in mind we added a non-maximum suppression step that is based on overlap area rather than IOU. This step eliminated spurious detections where one object was found to be inside the bounding box of another object of the same class. This would helped prevent the pipeline from tracking some false positive detections. The second step, was updating the histograms of the targets by Equation 5.6 to incorporate past object representations, where H^m is the new histogram, H^t is the histogram of the object in the current frame, H^{t-1} is the histogram of the object in the previous frame, and α is a scalar. We set alpha to be 0.8. This was similarly done by Tian et al. [68].

$$H^m = \alpha H^t + (1 - \alpha) H^{t-1} \tag{5.6}$$

5.4 Results

To evaluate our method, we tested TOD on three ROS bags recorded on board the flight vehicle across three different locations using the base detector, MobileNetSSD, as the baseline to evaluate against. Our ground truth labels were created using the Figure Eight platform. The Figure Eight platform uses human annotators to label raw data. We needed several objects of different classes to be labeled in each image. Figure Eight was able to provide this service by going through one of their partners. The data that was labeled using Figure Eight was a subsample of the total amount of image message collected in the ROS bag. The labeled data was taken at roughly 10 Hz, however the image rates in the ROS bags ranged from 20 Hz to 60 Hz depending on the camera being used. To better match the performance during runtime, we tested our method and the baseline with images collected at higher framerates and then only used the frames with an associated ground truth in the evaluation.

We evaluated our method and baseline, on the annotated ROS bags by measuring the total true positives, total false positives, recall, and precision across the entire

ROS bag at different confidence thresholds. We looked at thresholds from 0.15 to 0.95 at a step size of 0.05. Following the VOC model [15], a true positive detection is one that has an IOU greater than 0.5 with a ground truth bounding box. A false positive either has an IOU less than 0.5 with a ground truth box or do not overlap any ground truth box. Recall is calculated by Equation 5.7 where t is the total amount of true positives at a given confidence threshold and n is the total amount of ground truth objects in the dataset. Precision is found using Equation 5.8 where t is the amount of true positives and f is the amount of false positives.

$$r = \frac{t}{n} \tag{5.7}$$

$$p = \frac{t}{f+t} \tag{5.8}$$

5.4.1 Camp Lejeune

One of the flight testing locations was at the Marine Corps Base Camp Lejeune in North Carolina. An example image from the ROS bag recorded during a flight at Camp Lejeune is shown in Figure 5-4. This image was taken using an Intel Realsense, which was serving as the camera on the flight vehicle at that time. The Realsense collected images at roughly 60 Hz. During runtime we throttle the rate of accepted image messages to the object detector to 30 Hz so that the network does not constantly use the GPU. In our evaluation we collected images at around 30 Hz to more closely match runtime performance. We then evaluated the method on frames that were most closely associated with the frames that had ground truth information available.

Figure 5-4 shows that the environment the vehicle was flying in was quite unique and was not a typical urban environment. There weren't typical windows or doors and all of the cars are crushed with their windows broken in. Additionally, as might be expected in a semi-urban environment, the amount of windows vastly outnumbered any other of the detectable objects. This created a difficult job for the object detector. Since MobileNetSSD already had a low mAP for windows and doors, it was not

Figure 5-4: An image taken on the flight vehicle at Camp Lejeune during a flight test.

suitable for detecting those objects and will be left out of this evaluation. The results for people and cars are shown in Figures 5-5 and 5-6.

The results show that our method, Temporal Object Detection or TOD, outperformed the baseline detector in recall over all the confidence thresholds up to 0.8 and in some places precision. It also shows that our nominal operating confidence threshold of 0.3 for the people class was well-suited for the baseline detector on the Camp Lejeune dataset. While this threshold was good for the baseline detector, with TOD, adjusting the threshold to 0.4 gave a slight improvement to precision at a small cost to recall compared to the baseline. Figure 5-7 shows a sequence of images where TOD was able to pick up detections that would have otherwise been rejected. The top row shows the results from TOD and the bottom row are the results from the baseline. In the top row, blue bounding boxes are detections below the threshold that our method has determined match our target object and we accept it as a valid detetion. The red box represents a detection added by the GOTURN tracker. The GOTURN detection was also determined to be better than a bounding box returned by the detector.

The best results of TOD were seen on the people class. While TOD did achieve

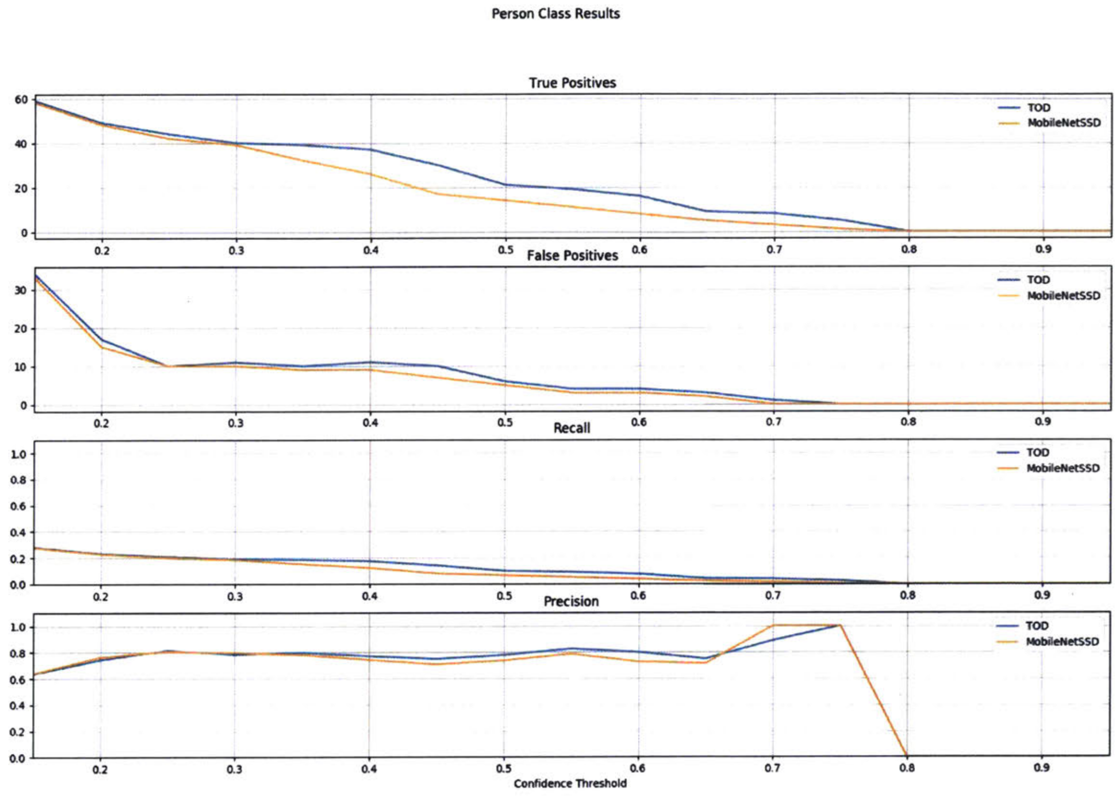


Figure 5-5: True positives, false positives, recall, and precision plotted against accuracy thresholds for the person class on the Camp Lejeune dataset. TOD is able to achieve a higher recall across all confidence thresholds than the baseline detector at a small cost to precision.

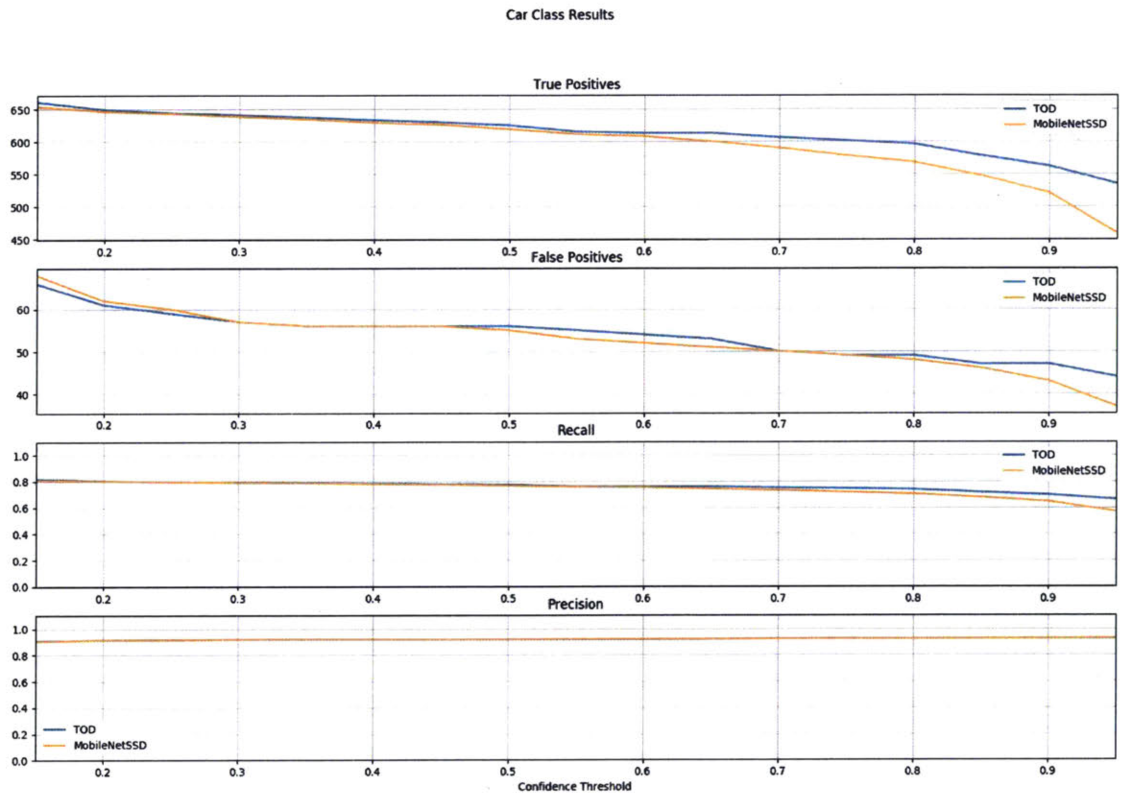


Figure 5-6: True positives, false positives, recall, and precision plotted against accuracy thresholds for the car class on the Camp Lejeune dataset.

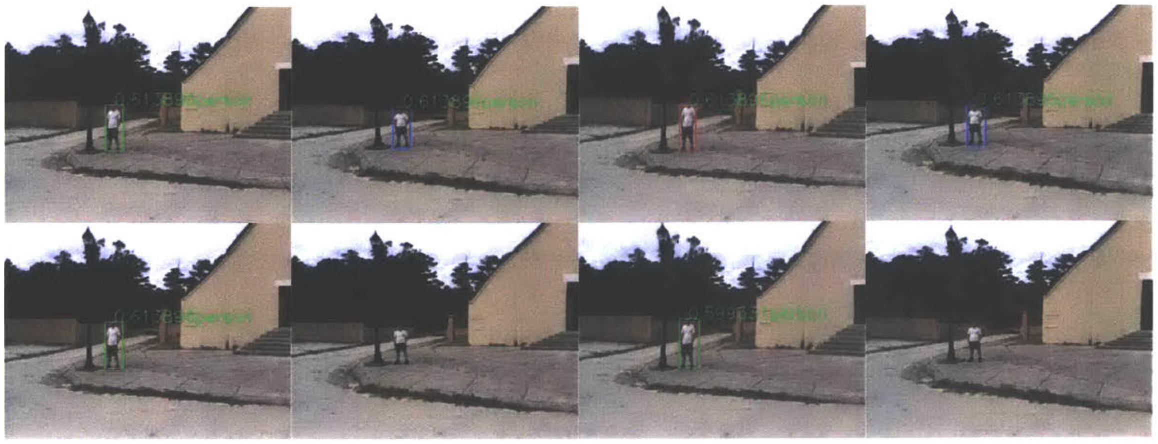


Figure 5-7: The top row shows detections from TOD and the bottom row shows detections from the baseline detector. Blue detections are low confidence detections matched with previously detected object. Red detections were found by GOTURN.

a higher recall than the baseline on the car class, the baseline detector is still very proficient at detecting cars.

A downside to TOD, was that when the confidence threshold increased, the number of false positives did not necessarily decrease as it does with the baseline. This was because a detection that would normally have detected the object was rejected by the threshold causing TOD to try and find the target object, producing an invalid detection. An example of this is shown in Figure 5-8. The top row shows the results of TOD running at the 0.4 threshold and the bottom row shows TOD running at the 0.35 threshold. The poor detection results of TOD on the top row were propagated from previous frames since the threshold rejected the detections that would have otherwise detected the object shown in the bottom row.

The higher recall of TOD comes at a modest cost to framerate despite using another neural network. Figure 5-9 shows a comparison of the framerates of the baseline detector and TOD at increasing confidence thresholds. TOD was still very fast running at over 60 fps for confidence thresholds over 0.2.

5.4.2 Medfield State Hospital

Another flight location for the vehicle was at the Medfield State Hospital in Medfield Massachusetts. This was a fairly different environment compared to Camp Lejeune.

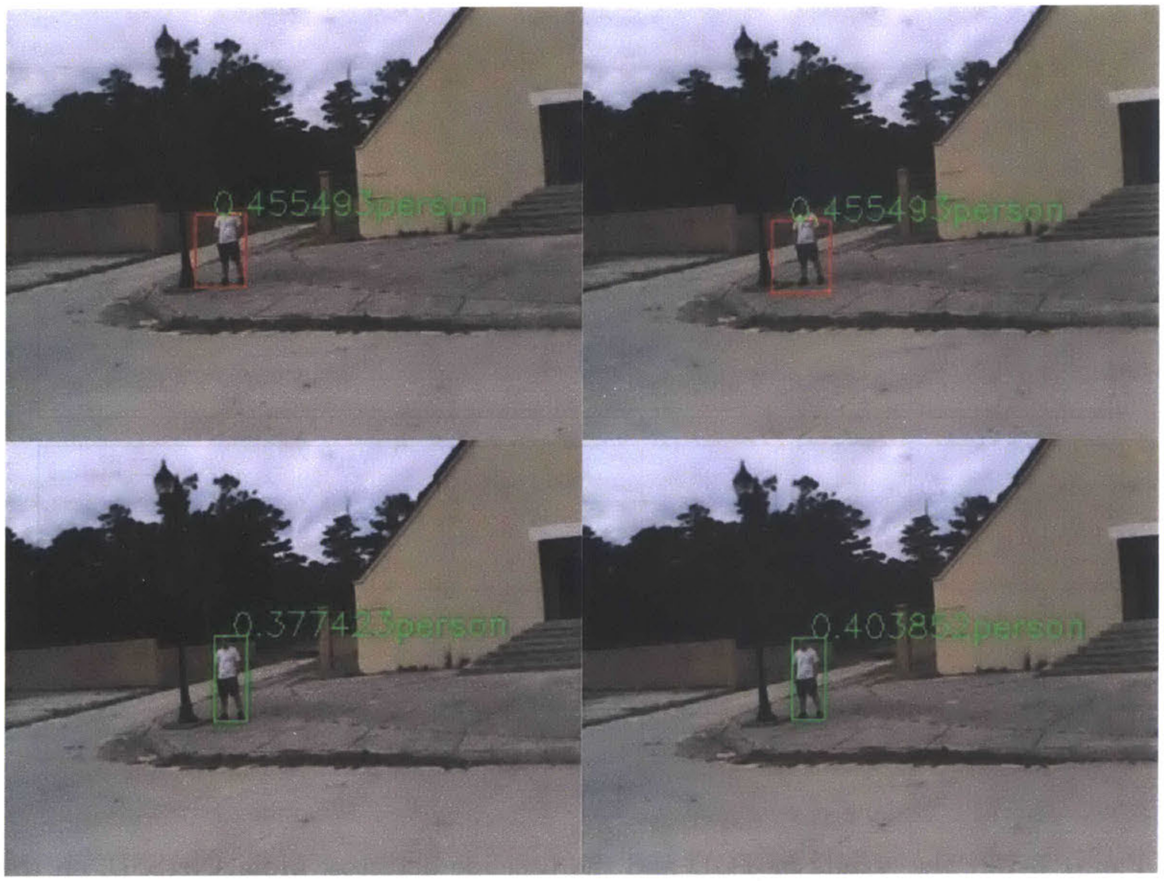


Figure 5-8: A poor result of TOD. The top row is from TOD running at a confidence threshold of 0.4 and the bottom row shows TOD running at a confidence threshold of 0.35. The red boxes are detections from the GOTURN tracker. The top row has poor detection results because of drift in the tracker from a bad target image. This does not occur in the bottom row because the detection output was not rejected by the threshold.

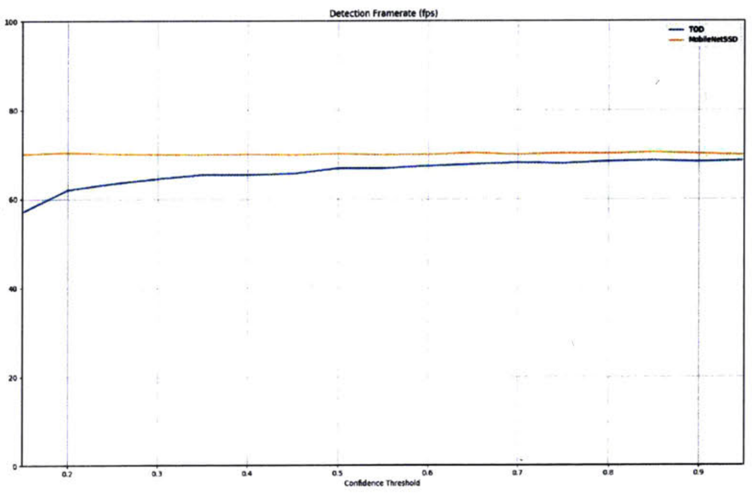


Figure 5-9: Average framerate of TOD vs the baseline on the Camp Lejeune dataset

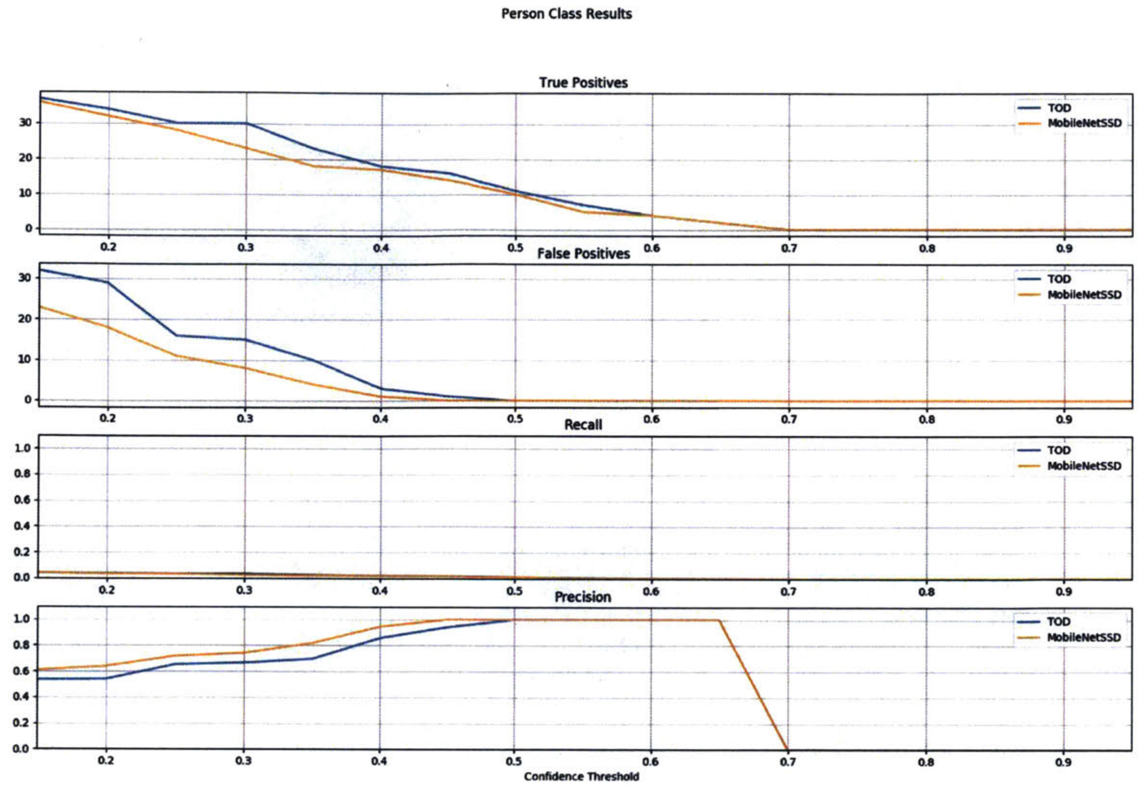


Figure 5-10: The results of TOD and the baseline detector, MobileNetSSD, on the Medfield data for the person class. TOD is able to add more valid detections at all the threshold levels up to 0.6. Additionally, several of the false positives are due to mislabeling of the ground truth data.

Instead of many cement buildings and broken cars, Medfield had brick buildings with boarded up windows and intact cars. Another difference was that at this stage in the program, the flight vehicle color camera switched from the Intel Realsense to the Zed camera. One challenge with this camera was that it would publish color image messages at irregular framerates ranging from 20-25 Hz. This was much slower when compared to the 60 Hz output of the Realsense. Like the Camp Lejeune results, we evaluated TOD and the baseline on total true positives, total false positives, recall, and precision across different confidence thresholds for the person and car classes. The ground truth was also provided by Figure Eight. The results are shown in Figures 5-10 and 5-11.

As with the Camp Lejeune dataset, TOD was able to find more valid detections than the baseline while only adding a small number of false positives. Figure 5-10

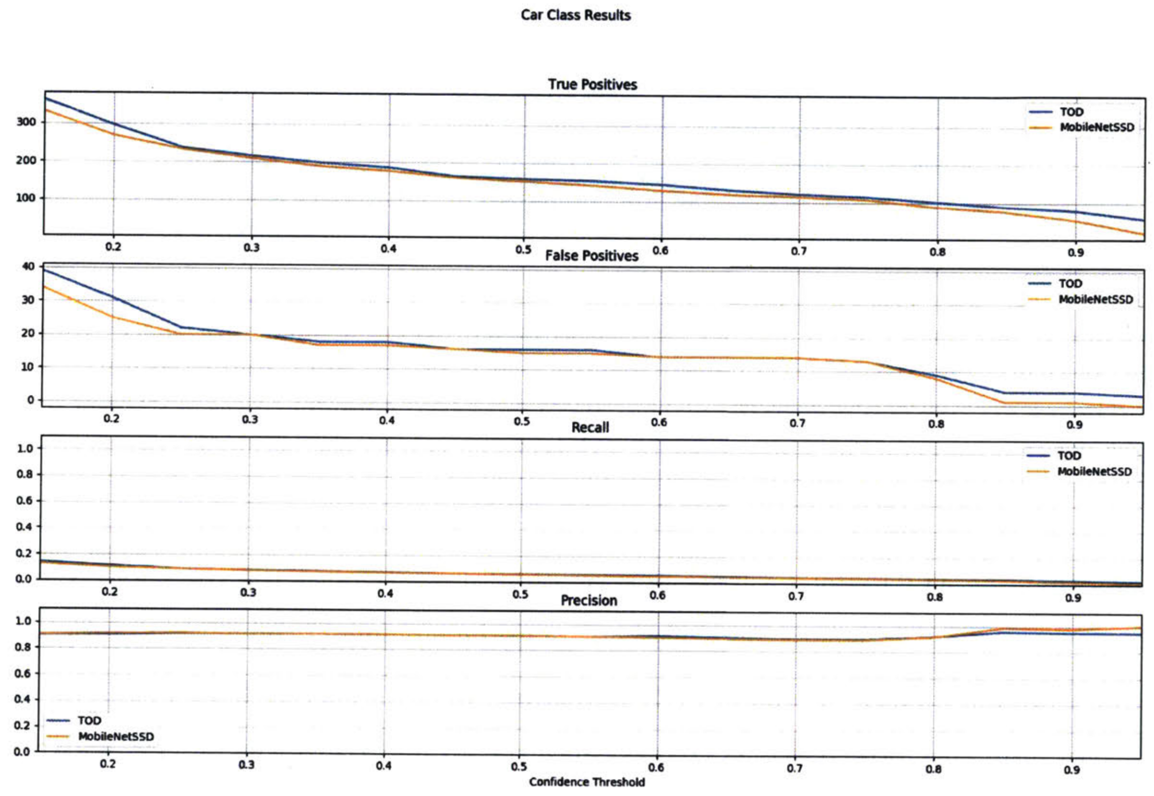


Figure 5-11: The results of TOD and the baseline detector, MobileNetSSD, on the Medfield data for the car class.

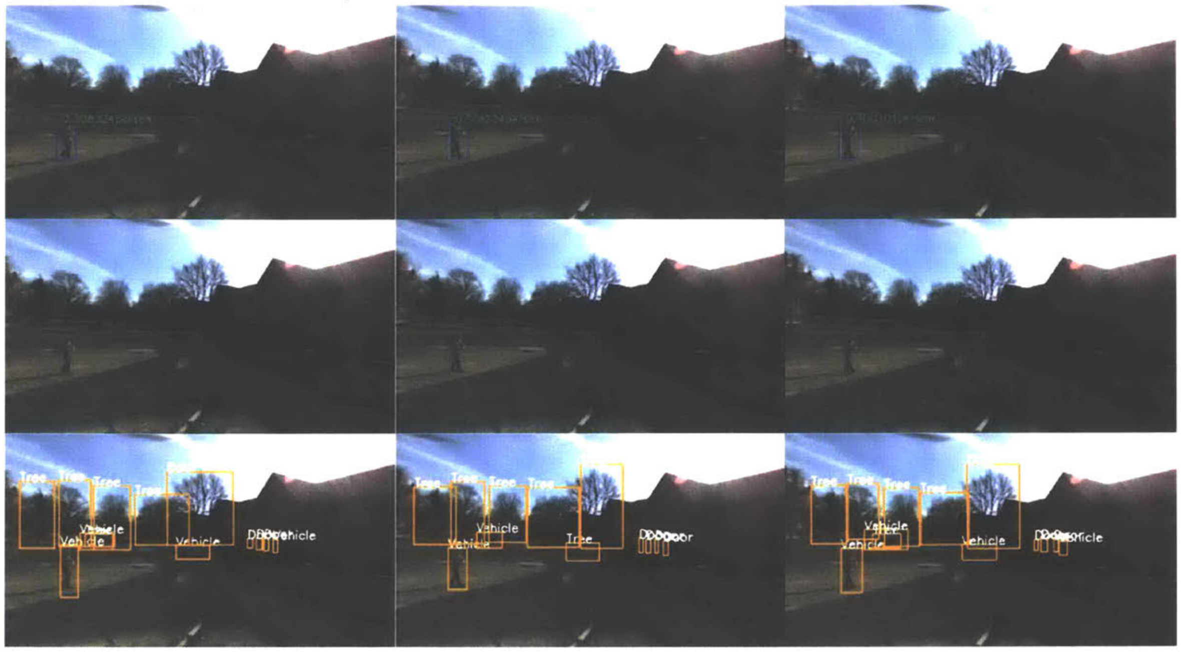


Figure 5-12: The top row shows the results from TOD and the second row are the baseline results using a confidence threshold of 0.3 over a sequence of three frames. The bottom row are the associated ground truth labels from Figure Eight. The blue boxes are low confidence boxes that have increased confidence by matching with previous detections. The three positives detections across the sequence are counted as false positives because of the mislabeling.

shows that the TOD added a large portion of false positives at the threshold of 0.3. This is an example of where there were errors in the labeling of the data. Figure 5-12 shows example images of where TOD detections were labeled as false positives. In this case, the false positives occurred because the ground truth was labeled with the wrong classes.

The cost to the achievable framerate of TOD on the Medfield data more closely matches the baseline. This was because the tracker was used fewer times in this dataset and more matching low confidence boxes were found. The comparison in framerates of TOD and the baseline are shown in Figure 5-13.

5.4.3 Guardian Center

The last location where we ran an evaluation of our method was at the Guardian Center in Perry, Georgia. The data taken at this location shows some of the weak

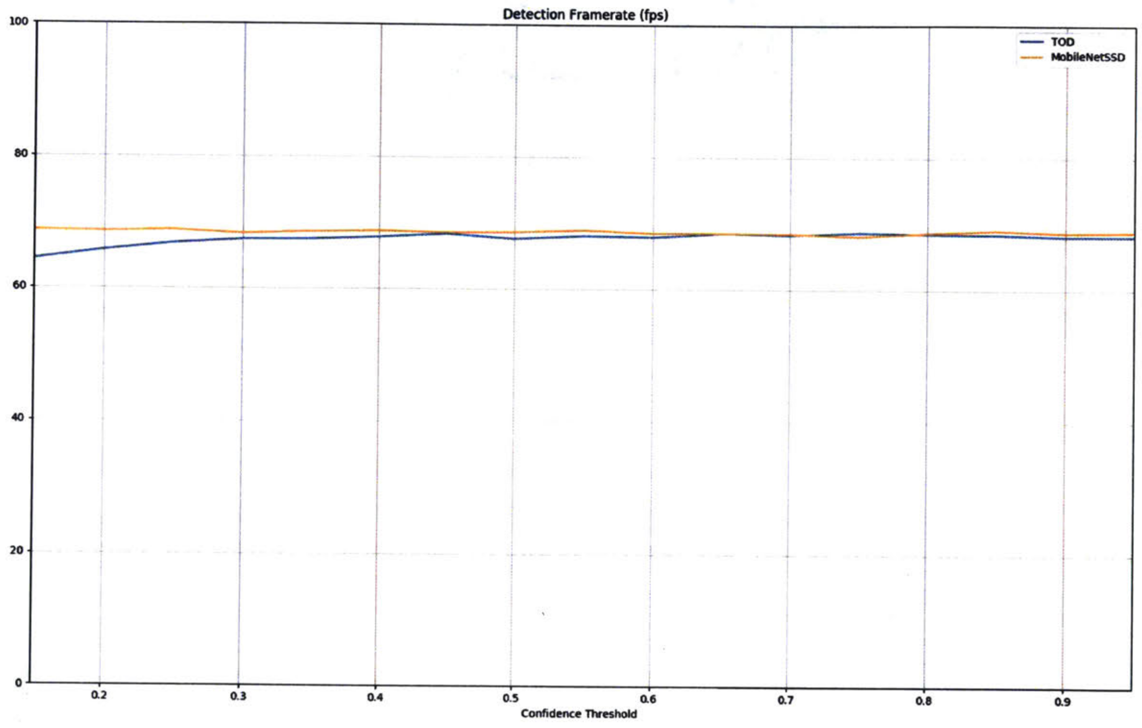
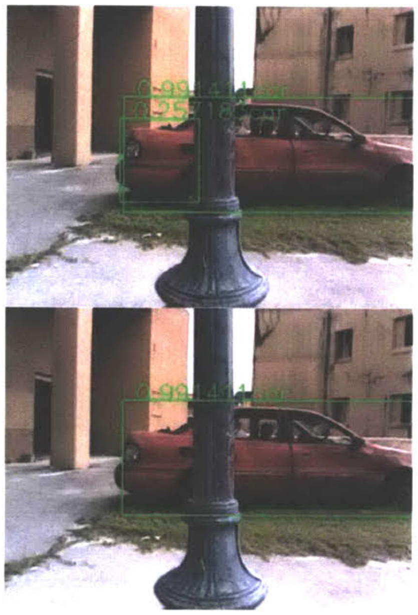


Figure 5-13: A comparison in frame rates of TOD vs the baseline MobileNetSSD detector. TOD performs very close to the framerate of the baseline while adding more valid detections.


a) Example of the overlap non-maximum suppression step removing a false positive detection.

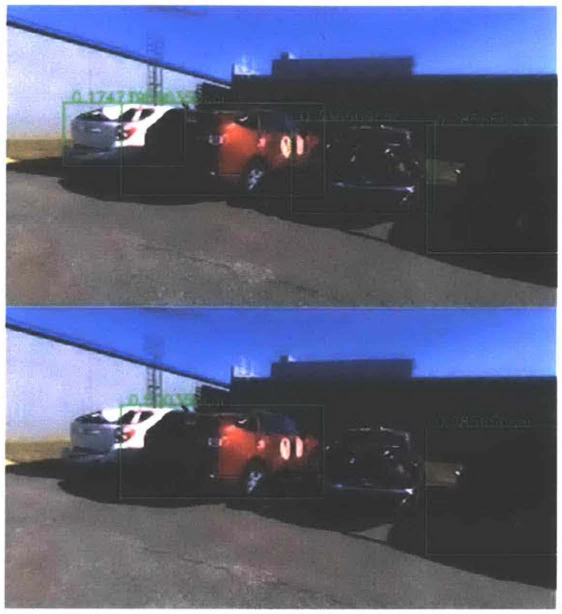

b) Example of the overlap non-maximum suppression step removing valid detections.

Figure 5-14: Two cases where the overlap based non-maximum suppression step gets rid of a false positive detection in a) and removes valid detections in b). The top frames show the result before the step is applied, the bottom shows the effect afterwards.

spots of our method. For one, there were several false positive detections that our method started to track, creating invalid detections. This highlights that TOD is only as good as the base detector. Additionally, there are several cars grouped together creating overlapping detections. This created a situation where our overlap based non-maximum suppression step would reject otherwise valid detections. Since many of the environments we flew in had a wide spatial distribution of objects, this step helped to reject false positive detections. Figure 5-14 shows two situations, one where this non-maximum suppression step helped eliminate false positive detections and another where it rejected valid detections.

The results of TOD and the baseline detector are shown in Figures 5-15, 5-16, and 5-17. Our method performed the worst on this dataset. It was unable to add

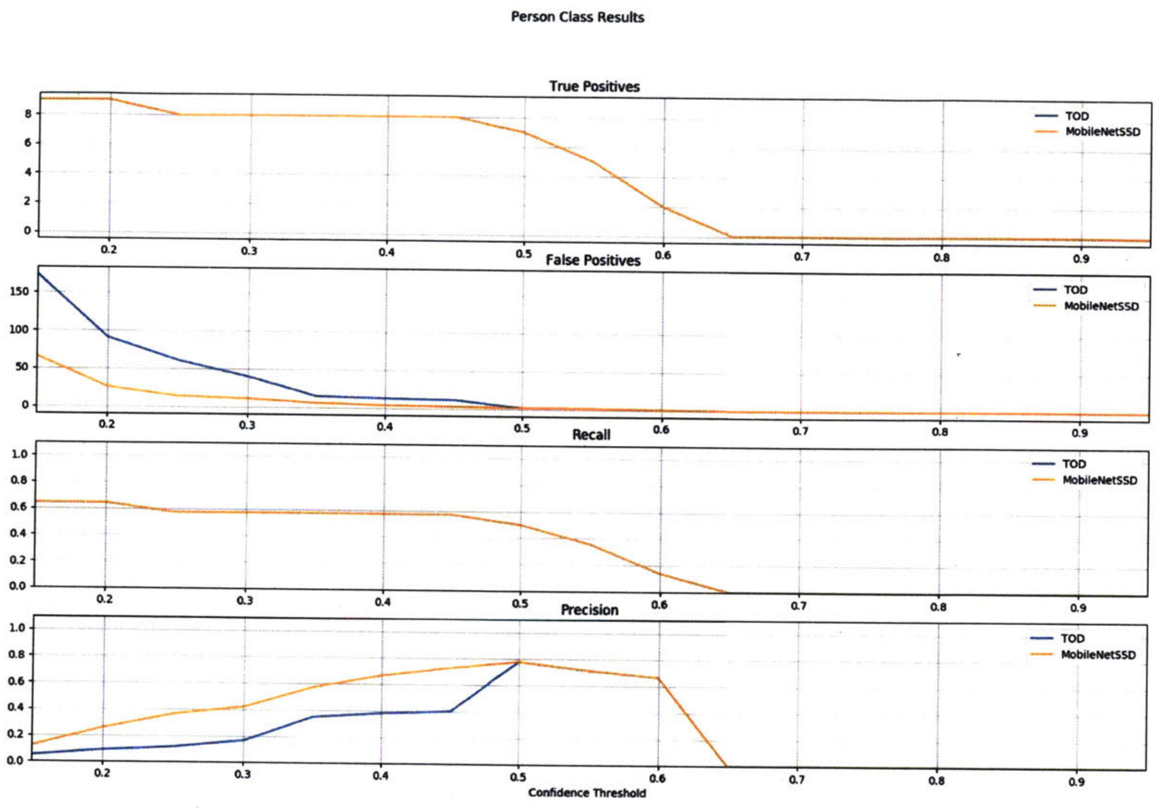


Figure 5-15: TOD compared to the baseline detector results on true positives, false positives, recall, and precision for varying confidence thresholds for the person class.

valid person detections to the baseline. At its worst, the overlap based non-maximum suppression step rejected several valid detections. The method also started to track a tree that was falsely detected as a person.

5.4.4 Discussion

Overall our method, Temporal Object Detection (TOD), performed well on the Lejeune and Medfield datasets, but had a difficult time on the Guardian dataset. TOD showed the most improvement on the people class, where it was able pick up lost detections. Additionally, by selectively applying the object tracker, the runtime of TOD was only slightly slower than the baseline detector. TOD has also proven to be very flexible. It can use any network as the base detector without any requirements on an object proposal stage as well as with any object tracker. The method is also tunable by raising or lowering the similarity score threshold providing a trade off

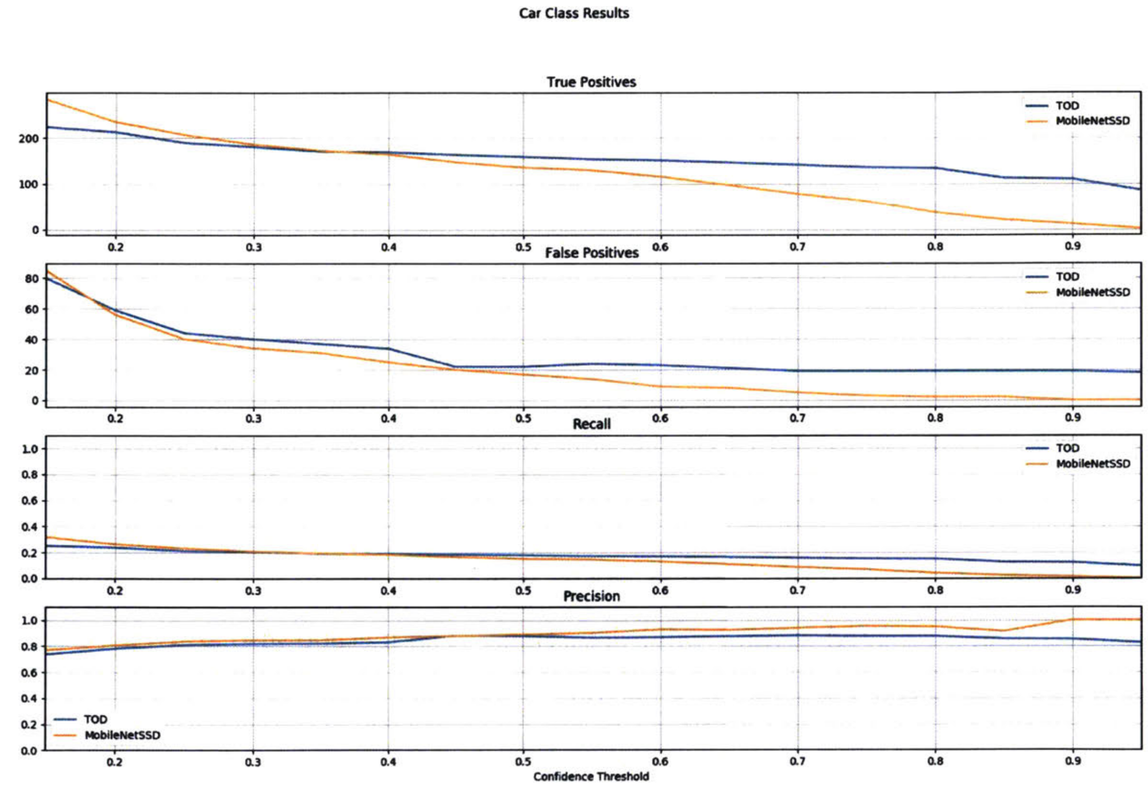


Figure 5-16: TOD compared to the baseline detector results on true positives, false positives, recall, and precision for varying confidence thresholds for the car class.

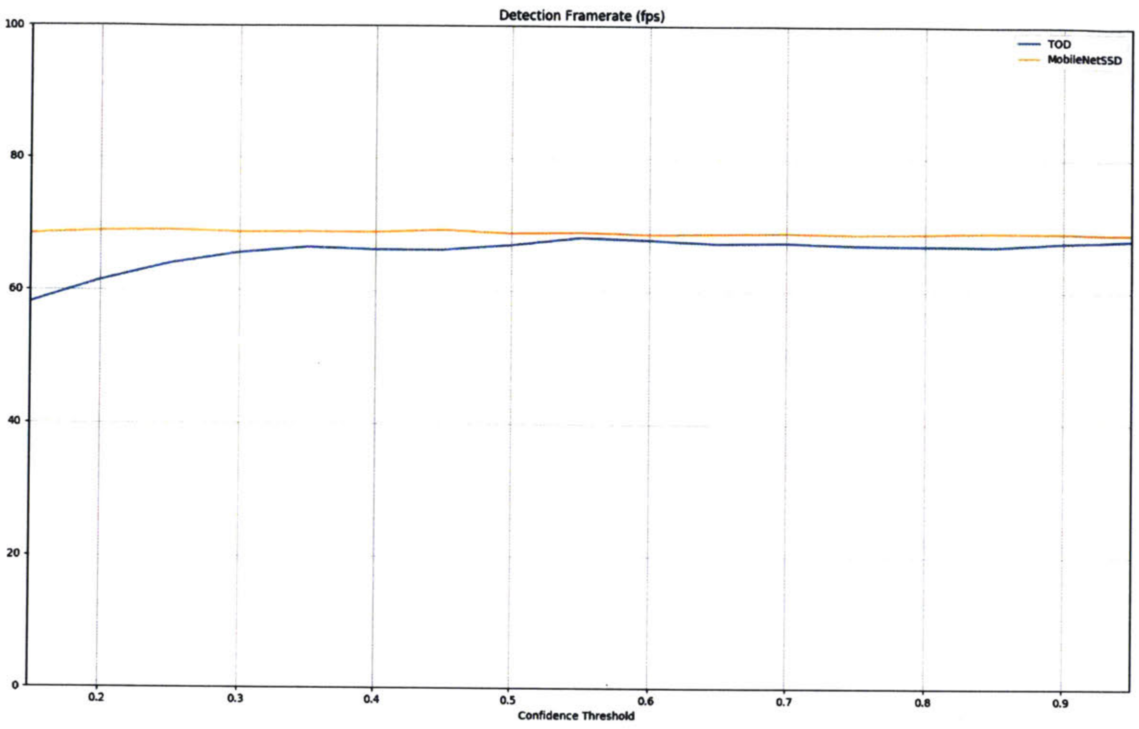


Figure 5-17: Average framerate of TOD vs the baseline on the Guardian dataset.

between recall and precision.

The method as it currently stands assumed that objects are spread fairly sparsely throughout the map. Using this assumption, we applied a non-maximum suppression step based on the ratio of the overlap area between two boxes and a box's area eliminating small detections inside larger detections. This does however eliminate positive detections if objects are close together. The method could be improved by being more selective about what objects it decides to track, but we leave this for future work.

As the results across the datasets show, although TOD often offers improvements in both recall and precision, the baseline detector also performs very well on its own. TOD becomes more important on edge case objects where the detector may only get one or two high confidence detections. TOD can then help provide subsequent detections providing enough for the object to be localized in the map by the object mapper later on in the pipeline.

For improvements, the method could likely be paired with a motion model to help track the objects. Additionally, instead of using color similarity as a main portion of the similarity metric, appearance features could be incorporated into the model. The tracking step could also be improved by creating a criteria for which objects should be tracked. We leave these possible improvements for future work.

Chapter 6

Conclusion

6.1 Summary

The research presented in this thesis was motivated by the DARPA Fast Lightweight Autonomy (FLA) program. The goal of the program was to have an autonomous micro aerial vehicle (MAV) explore and map an urban environment. The map needed to include semantically labeled objects of interest such as cars and people that the vehicle encountered in its exploration. In pursuit of this goal, the flight vehicle needed a real time object detection system to find and recognize objects in the environment.

To develop this capability, we first evaluated of several state of the art object detectors measuring speed and resource consumption across different hardware. This evaluation included recently developed convolutional neural network (CNN) object detectors. We tested You Only Look Once v2 (YOLOv2), Tiny YOLO, the Single Shot MultiBox Detector (SSD), and another version of SSD, MobileNetSSD. These networks were tested across five different hardware setups. The hardware included a desktop setup with a NVIDIA TitanX GPU and Intel Core i7-6700KCPU, a NVIDIA Jetson TX1, a NVIDIA Jetson TX2, an Intel Movidius Neural Compute Stick, and an Intel NUC Skull Canyon. The evaluation also included testing the networks in a GPU mode and a CPU only mode. In total, we tested 43 different configurations. Throughout the evaluation we looked at the networks' framerate in frames per second (fps), GPU utilization, CPU utilization, and memory usage. We found after the evaluation that MobileNetSSD provided the most flexibility and efficiency across the various hardware setups.

After we chose MobileNetSSD as the object detector for the flight vehicle, we worked on integrating it with the software stack and hardware. Since the flight vehicle was already using the Intel NUC Skull Canyon, we decided to also use it for object detection. In order to utilize the GPU on the NUC we had to use a special version of Caffe called clCaffe, which uses an OpenCL backend. For initial flight tests, MobileNetSSD used images from the onboard Flea3 camera since MobileNetSSD had similar performance using both color and grayscale images.

During deployment, the object detector would look at a single frame from an image stream being collected by the camera. Since the detector only looks at one image at a time, it would frequently detect an object in one frame and not in the next. To take advantage of the temporal information in the image stream, we presented Temporal Object Detection (TOD). TOD used MobileNetSSD as a base object detector and a similarity metric to either link detections between frames or determine when a previously detected object had been lost. If an object was lost, TOD employed the object tracker GOTURN to try and find the lost object. By selectively applying the object tracker we maintained a high average framerate. Our results show that in two of the three datasets TOD outperforms the baseline detector in finding valid detections while still remaining fast.

6.2 Contributions

The main contributions of this work are an evaluation of state of the art CNN based object detectors across a range of hardware and TOD, a method for using temporal information in the object detection task. Previous evaluations of the tested CNN object detectors including both an independent analysis [28] and analyses done in the detectors' respective papers [57, 42], did not provide runtime results for different hardware of the kind presented in this thesis. Additionally, the evaluation looked at the performance of the hardware in supporting computationally demanding CNNs.

The results of this evaluation can help guide future deployment configuration decision by finding the appropriate balance in accuracy, speed, resource usage.

We also proposed TOD, a fast object detection method that incorporates previously detected object representations to both determine when an object has been lost and to guide the search for the object using low confidence detections and an object tracker. TOD showed promising results on the tested datasets. The evaluation of TOD also showed some weaknesses in using annotation services to label testing data.

6.3 Future Work

There are several areas for future work. For one, the object detection evaluation did not involve optimizing the tested hardware configurations. This could shine more light on the full potential of the hardware. Additionally, different versions of TOD could be explored using various combinations of object detectors and object trackers. A new similarity metric could also be used that takes advantage of features and a motion model.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Luca Bertinetto, Jack Valmadre, Joao F Henriques, Andrea Vedaldi, and Philip HS Torr. Fully-convolutional siamese networks for object tracking. In *European Conference on Computer Vision*, pages 850–865. Springer, 2016.

[3] chuanqi305. Mobilenet-ssd. https://github.com/chuanqi305/MobileNet-SSD.

[4] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):564–577, 2003.

[5] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016.

[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005.

[7] J Deng, A Berg, S Satheesh, H Su, A Khosla, and L Fei-Fei. Ilsvrc-2012, 2012. *URL http://www. image-net. org/challenges/LSVRC*, 2012.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[9] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: An evaluation of the state of the art. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):743–761, 2012.

[10] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2154, 2014.

[11] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[12] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2008 (VOC2008) Results. http://www.pascal-network.org/challenges/VOC/voc2008/workshop/index.html.

[13] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[14] M. Everingham, A. Zisserman, C. K. I. Williams, and L. Van Gool. The PASCAL Visual Object Classes Challenge 2006 (VOC2006) Results. http://www.pascal-network.org/challenges/VOC/voc2006/results.pdf.

[15] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[16] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Detect to track and track to detect. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3038–3046, 2017.

[17] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, Sept 2010.

[18] Leonardo Galteri, Lorenzo Seidenari, Marco Bertini, and Alberto Del Bimbo. Spatio-temporal closed-loop object detection. *IEEE Transactions on Image Processing*, 26(3):1253–1263, 2017.

[19] Tijmen Tieleman Geoffrey Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. 2012.

[20] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[21] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[22] Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S Huang. Seq-nms for video object detection. *arXiv preprint arXiv:1602.08465*, 2016.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[24] David Held, Sebastian Thrun, and Silvio Savarese. Learning to track at 100 fps with deep regression networks. In *European Conference on Computer Vision*, pages 749–765. Springer, 2016.

[25] João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-speed tracking with kernelized correlation filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(3):583–596, 2015.

[26] Hisham Muhammad. htop, v2.0.1. https://hisham.hm/htop/index.php?page=faq, 2016.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[28] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016.

[29] Intel. *Intel Democratizes Deep Learning Application Development with Launch of Movidius Neural Compute Stick*, 2017.

[30] Intel Corporation. Intel opencl linux graphics device driver, v r4.1. https://software.intel.com/en-us/articles/opencl-driverslatest$_linux_SDK_release$, 2016.

[31] Intel Corporation. Intel sdk for opencl applications, v7.0. https://software.intel.com/en-us/intel-opencl, 2017.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[33] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[34] Kai Kang, Hongsheng Li, Tong Xiao, Wanli Ouyang, Junjie Yan, Xihui Liu, and Xiaogang Wang. Object detection in videos with tubelet proposal networks. In *Proc. CVPR*, volume 2, page 7, 2017.

[35] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-cnn: Tubelets with convolutional neural networks for object detection from videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 2017.

[36] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 817–825, 2016.

[37] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[39] Yang Li, Jianke Zhu, and Steven CH Hoi. Reliable patch trackers: Robust visual tracking by exploiting reliable patches. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*, pages 353–361. IEEE, 2015.

[40] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[41] Wei Liu. Ssd: Single shot multibox detector. https://github.com/weiliu89/caffe/tree/ssd.

[42] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[43] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004.

[44] Chao Ma, Jia-Bin Huang, Xiaokang Yang, and Ming-Hsuan Yang. Hierarchical convolutional features for visual tracking. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3074–3082, 2015.

[45] NVIDIA Corporation. Cuda toolkit 9.0. https://developer.nvidia.com/cuda-90-download-archive.

[46] NVIDIA Corporation. Jetpack 3.2. https://developer.nvidia.com/embedded/jetpack.

[47] NVIDIA Corporation. Nvidia system management interface program, v384.111. https://developer.nvidia.com/nvidia-system-management-interface.

[48] NVIDIA Corporation. tegrastats, 28.2.

[49] NVIDIA Corporation. Cuda toolkit 8.0. https://developer.nvidia.com/cuda-80-ga2-download-archive, 2017.

[50] NVIDIA Corporation. Jetpack 3.1. https://developer.nvidia.com/embedded/jetpack-3_1, 2017.

[51] Dim P. Papadopoulos, Jasper R. R. Uijlings, Frank Keller, and Vittorio Ferrari. We don't need no bounding-boxes: Training object class detectors using only human verification. *CoRR*, abs/1602.08405, 2016.

[52] Sudeep Pillai and John Leonard. Monocular slam supported object recognition. *arXiv preprint arXiv:1506.01732*, 2015.

[53] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, page 5. Kobe, Japan, 2009.

[54] Joseph Redmon. Yolo: Real-time object detection. https://pjreddie.com/darknet/yolov2/.

[55] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[56] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[57] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[58] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[59] Karl Rupp. Viennacl, v1.7.1. http://viennacl.sourceforge.net/, 2016.

[60] O Russakovsky, J Deng, J Krause, A Berg, and L Fei-Fei. Large scale visual recognition challenge 2013 (ilsvrc2013), 2013.

[61] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[62] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[63] Laurent Sifre and PS Mallat. *Rigid-motion scattering for image classification.* PhD thesis, Citeseer, 2014.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[65] Peng Tang, Chunyu Wang, Xinggang Wang, Wenyu Liu, Wenjun Zeng, and Jingdong Wang. Object detection in videos by short and long range object linking. *arXiv preprint arXiv:1801.09823*, 2018.

[66] OpenCV Developers Team. Opencv, 2017.

[67] TechPowerUp. Intel iris pro graphics 580. https://www.techpowerup.com/gpudb/2788/iris-pro-graphics-580.

[68] Shu Tian, Fei Yuan, and Gui-Song Xia. Multi-object tracking with inter-feedback between detection and tracking. *Neurocomputing*, 171:768–780, 2016.

[69] Philippe Tillet. Isaac. https://github.com/ptillet/isaac, 2017.

[70] Antonio Torralba, Kevin P Murphy, and William T Freeman. Sharing visual features for multiclass and multiview object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(5):854–869, 2007.

[71] Fabian Tschopp. Efficient convolutional neural networks for pixelwise classification on heterogeneous hardware systems. *CoRR*, abs/1509.03371, 2015.

[72] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013.

[73] Jack Valmadre, Luca Bertinetto, João Henriques, Andrea Vedaldi, and Philip HS Torr. End-to-end representation learning for correlation filter based tracking. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5000–5008. IEEE, 2017.

[74] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–511–I–518 vol.1, 2001.

[75] Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu. Visual tracking with fully convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3119–3127, 2015.

[76] Yu Xiang, Alexandre Alahi, and Silvio Savarese. Learning to track: Online multi-object tracking by decision making. In *2015 IEEE International Conference on Computer Vision (ICCV)*, number EPFL-CONF-230283, pages 4705–4713. IEEE, 2015.