Spring 4-2017

Feasibility of Neural Networks for Maritime Visual Detection on a Mobile Platform

Robert Goring
Embry-Riddle Aeronautical University

Feasibility of Neural Networks for Maritime Visual Detection on a Mobile Platform

by

Robert Goring

A Thesis Submitted to the College of Engineering, Department of Electrical, Computer, Software, & Systems Engineering for the partial fulfillment of the requirements of the degree of

Master of Science in Electrical and Computer Engineering

Embry-Riddle Aeronautical University

Daytona Beach, Florida

April 2017

Feasibility of Neural Networks for Maritime Visual Detection on a Mobile Platform

by

Robert Goring

This thesis was prepared under the direction of the candidate's thesis committee Chair, Dr. Brian Butka, Professor, Daytona Beach Campus, and Thesis Committee Members Dr. Eric Coyle, Professor, Daytona Beach Campus, and Dr. Jianhua Liu, Professor, Daytona Beach Campus, and has been approved by the thesis committee. It was submitted to the Department of Electrical, Computer, Software, & Systems Engineering and was accepted in
partial fulfillment of the requirements for the degree of Masters of Master of Science in Electrical and Computer Engineering.

THESIS COMMITTEE

Chairman, Dr. Brian Butka

Member, Dr. Eric Coyle

Member, Dr. Jianhua Liu

Department Chair, Dr. Timothy A. Wilson

27 APR 2017
Date

Dean of College of Engineering, Dr. Maj Mirmirani

4/27/17
Date

Vice Chancellor for Academics, Christopher Grant, PhD

4/27/2017
Date

Acknowledgments

I would like to begin by expressing my sincerest gratitude to everyone who assisted me in the completion of this thesis. First and foremost, my advisor Dr. Butka who ignited my interest in computer vision, and who pushed me to succeed. I would also like to thank my committee members, Dr. Coyle and Dr. Liu for serving on my thesis committee and providing invaluable feedback and inspiration.

Thanks goes to the Robotics Association at Embry-Riddle as well as the faculty advisors for providing the hardware and opportunities required for the completion of this thesis. This project would not have been possible without their support and resources. Special acknowledgment goes to ONR grant #N000141512746 for partial support.

Lastly, I need to convey my thanks to my family and friends. Without their support, I would not have made it wo where I am today. To my parents, for their years of support and love throughout my educational career. Thanks goes to my brother Andrew for assistance in editing, whose editing skills have been invaluable. Also, I must thank all my friends for caring and always being there for me. Finally, I want to thank to my beautiful girlfriend Tara. I could not have completed this thesis without all her support and encouragement.

Abstract

Researcher: Robert Goring

Title: Feasibility of Neural Networks for Maritime Visual Detection on a Mobile Platform

Institution: Embry-Riddle Aeronautical University

Degree: Master of Science in Computer and Electrical Engineering

Year: 2017

Object detection through computer vision has traditionally been difficult to reliably implement due to various lighting conditions caused by weather and time of day. Any changes in conditions can be detrimental to the detector's ability to accurately identify objects. A modern approach implements deep learning techniques to classify and train a neural network. While highly effective, this approach can be cumbersome and computationally intensive. This project will investigate the feasibility of using deep learning to detect, classify, and track objects in near real-time while being processed on a mobile platform. I will investigate the feasibility of these processes on a small embedded system, such as the NVIDIA Jetson TX1. I will investigate several promising algorithms such as Faster R-CNN, TensorBox, DetectNet, and YOLO. This research is beneficial because it will transition deep learning techniques developed primarily for research in a lab environment to a real-world situation in which high accuracy and fast processing are vital. The work solved through this research will greatly benefit platforms that require object detection capabilities, but do not have the space, budget, or power capabilities for large GPUs or GPU clusters.

Table of Contents

List of Tables

List of Equations

List of Figures

1. Introduction

1.1. Background

Due to the harshness of the marine environment, there is a heavy cost, as well as danger, associated with conducting manned surface and subsurface missions. To help combat these issues, unmanned vehicles have been used to remove the human element from the situation. Unmanned marine vehicles have been used by the U.S. Navy since 1932 [1]. The effectiveness of these vehicles, however, is limited by their need to be operated and controlled by humans. Modern researchers are designing new ships that can be operated autonomously.

There are countless naval applications for Autonomous Surface Vehicles (ASVs), and Autonomous Underwater Vehicles (AUVs), which can cover a wide range of tasks. These autonomous vehicles can be used for mine countermeasures, harbor monitoring, inspection, and Intelligence, Surveillance and Reconnaissance (ISR) [2]. While there is extensive research for the controls aspects of these vehicles, their autonomy is still limited by their ability to sense and detect objects within their surroundings. Using radar, LIDAR, or sonar, it is possible to detect an object's presence. Classification of these objects is a more complicated process.

Classification is the process of algorithmically determining which class an object may be. For naval vehicles, many objects that would be encountered in their use cases will have a pre-defined appearance, such as navigational markers and buoys, but there is a nearly unlimited range of objects that could be placed within the path of these vehicles.

This makes the task of classifying surrounding objects incredibly difficult. For this thesis, it will be assumed that any object that will be classified is pre-defined.

Often during non-autonomous classification, data is required to be transmitted back to the operator at the ground station, who must analyze the data, and decide what the object is. This process, however, would induce human error into the equation. This would be much more optimized if the vehicle could make these decisions autonomously. As classification allows for autonomous navigation, the human resources needed for operation of the vehicle could be significantly reduced by onboard classification. This would decrease the response time of the vehicle, as well as lower the communication required of the vehicle. In the case of ROVs, high speed underwater communication is extremely complex and expensive. If an AUV could sense and act on its own, a data tether would not be necessary. To increase the autonomy of ASVs or AUVs, they should be able to detect and classify objects within their surroundings through the use of cameras, sonar, or LIDAR systems. Sonar, and LIDAR solutions however are often much more expensive than a camera, and are not well suited to classifying colors or shapes on a flat surface. One limitation of using cameras for detection and classification is that most algorithms are often not robust enough to handle varying lighting conditions.

This thesis will investigate the use of deep learning techniques to create an image classifier for an ASV or an AUV. The classifier should run on the vehicle, and not rely on transmitting data to a ground station for processing. This classifier will autonomously classify course elements in near real time through the use of a camera system. The developed methods were tested on two autonomous vehicles. A prototype of this software suite was tested on the ASV, Minion at the 2016 RobotX Challenge. Figure 1 shows Minion during a test on the Halifax River in Daytona Beach.

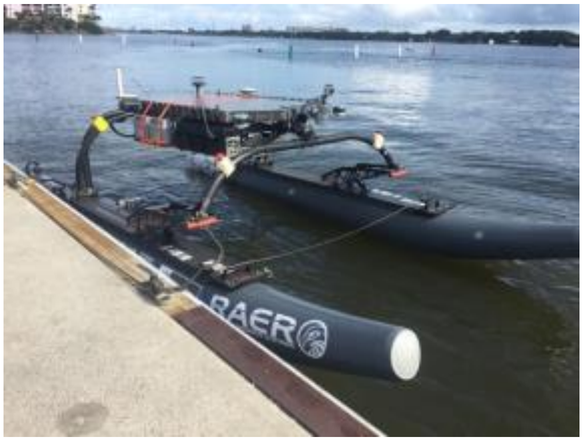


Figure 1 - Minion ASV During a test on the Halifax River in Daytona Beach, FL

This software has also been tested on the AUV Blackfinn for the Association for Unmanned Vehicle Systems International's (AUVSI's) RoboSub competition. The Embry-Riddle RoboSub team plans to use the results of this project for their entry in the 2017 AUVSI RoboSub Competition. A rendering of Blackfinn is shown in Figure 2.

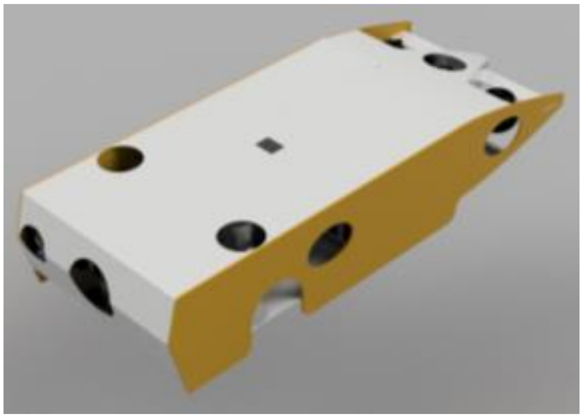


Figure 2 - 3D rendering of Blackfinn

Both the RobotX and RoboSub competitions are held on a closed course. This means that every element within the course is well documented and known ahead of time. This allows for the classifier to be pre-trained on any elements that require classification.

1.2. Motivation

Machine learning is a field of computer science that studies how to analyze data and build models, without needing to explicitly program the system how to complete the task [3]. This field of computer science is rapidly growing with the onset of new hardware and software discoveries. Machine learning is a major subset of artificial intelligence. Recently, one subset of machine learning that has rapidly developed is deep learning. This rapid development is largely due to the advances in graphics processing units, GPUs [4]. Deep learning uses of many layers to compute an output.

Some models consist of 1000+ hidden layers [5]. These layers require many gigabytes of RAM to store, as well as millions of operations to compute. Due to their architecture, GPUs are perfectly suited to handle these large numbers of layers. The architecture of these layers is shown in Figure 3. Each network has an input layer, and an output layer. These layers are connected by a series of hidden layers [6]. Popular layer types for deep learning are convolution layer, pooling layer, dropout layer, relu, tanh, and sigmoid layer [7]. These layers are composed of nodes, which contain values based on the outputs from convolutions certain weights and parameters.

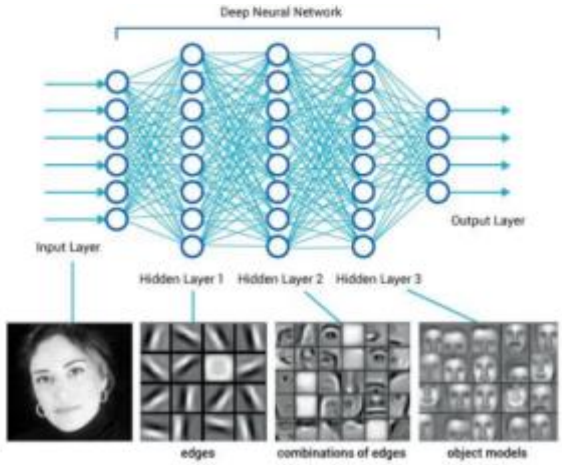


Figure 3 - Architecture of a Deep Neural Network with visualizations of each layer [7]

Deep learning can be used to complete a multitude of complicated tasks such as optical character recognition (OCR), voice recognition, medical diagnosis, and financial trends [3] [8]. Another application that can greatly benefit from deep learning techniques is computer vision. Deep learning when applied to computer vision can perform extremely powerful image classification and detection algorithms. An example of the results from these algorithms is shown in Figure 4.

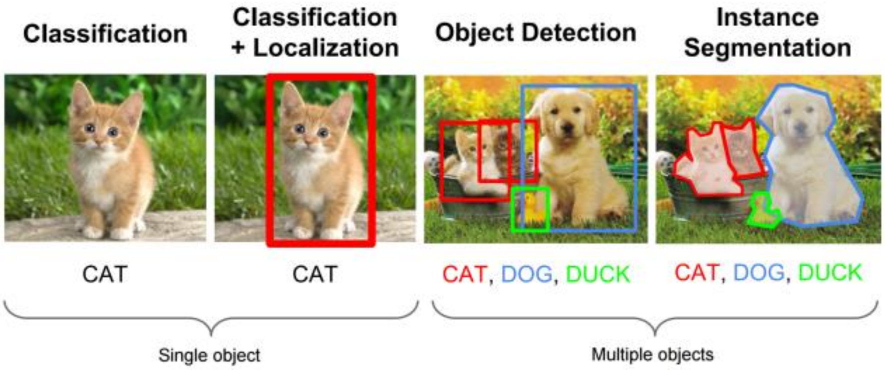


Figure 4 - Examples of object classification and Detection [7]

Classification is the process in which an image is analyzed and a determination is made of what object, or objects, are in the picture. Such applications include Google's Reverse Image Search, Facebook's facial recognition, and Pinterest's similar object recognition [9] [10] [11]. While these uses are significant, another field that can greatly benefit from deep learning is robotics. Detection is an important part of these algorithms. Without detection capabilities, for example, Facebook's facial recognition could not work on an image containing more than just a person's face. Detection allows for the classifier to localize objects throughout the image frame. This is often done through a process called Selective Search [12].

For a large majority of systems, computer vision is an important part of the perception suite on any autonomous or robotic system. When relying on this system, it is imperative that it is as robust and accurate as possible. One method to increase the robustness and accuracy of computer vision is to use deep learning. Deep learning however, is an extremely computationally intensive technique. While this processing can be handled by a central processing unit (CPU) massive performance increases are achieved by processing on a GPU. This allows for parallel processing. GPUs are twenty to fifty times more efficient than CPUs when performing deep-learning computations [4]. To handle the processing required for deep learning, it is standard to perform processing on large servers with multiple workstation grade GPUs. This processing requirement provides a practical limitation of the applications in which deep learning can be used. Using current practices, it is highly impractical to perform deep learning on a mobile platform.

Though deep learning appears to solve all the processing issues for mobile robotic systems, there are still several issues that must be circumvented before implementation is feasible. Some issues with deep learning computations occurring on a mobile system is the strict size and power requirements. These deep learning processes are often run on large workstation grade GPUs, such as the NVIDIA M6000 or NVIDIA Plex 7000 [13]. These GPUs can draw up to 600 watts of power as well as require proper ventilation to cool. These requirements make integration a difficult process.

The purpose for this research is to demonstrate the feasibility of using deep learning techniques to perform object detection and classification while on a mobile platform. For this to be feasible, implementation should be performed on a graphics card with a low power consumption. This would increase the runtime of the vehicle, as well as reduce the amount of heat the vehicle must dissipate. This is crucial in the maritime field as an ASV may be exposed in the hot sun, and an AUV may have limited internal airflow, due to a sealed hull. This research investigates ways to accurately and efficiently run object recognition processes on mobile platforms.

1.3. Research Objectives

This report will focus on object detection and classification, with an emphasis on naval applications. When developing classification techniques, they are often tested on a select few popular databases. Popular databases are ImageNet, MNIST, CIFAR 10, VOC2012, and STL-10 [14]. These databases often contain millions of images, with thousands of images of each class. A class is an object that has been manually classified to be recognized. For example, the ImageNet database contains over 14 million images and nearly twenty-two thousand individual classes [15].

To give this research a practical application, testing will not be completed on one of these previously mentioned publicly available datasets; testing will be performed on self-obtained datasets. These datasets used for this research will be images taken from the AUVSI's Maritime RobotX Competition, as well as the AUVSI's RoboSub competition. The RobotX dataset that was used was collected at the 2016 RobotX competition, which was held in Honolulu, HI. The RoboSub dataset is composed of images from the 2014 and 2015 RoboSub competitions, which were held at the TRANSDEC facility in San Diego, CA. I was present at these competitions, as a member of the Robotics Association at Embry-Riddle, to obtain these images.

Deep learning vision processing techniques have widely been used for object detection and classification on high powered computers. This research employs many new computational techniques, which were adapted for use with image processing. A key objective of this research is to prove that object detection through deep learning is feasible while running on a mobile platform. The primary technical objectives of this research are as follows:

1. The detector and classifier should be trainable on a custom dataset.
2. The software should be able to run on a consumer grade computer with consumer grade GPU.
3. The software should be able to run at least at a moderate frame rate.

Some methods of detection or classification are not capable of being trained on a custom dataset, meaning that if a desired object is not in that trained database, it cannot be detected. Therefore, when determining a method to use for a detector, it is a requirement that it is capable of being trained on a custom dataset. This method of training is called supervised learning. While it helpful for some research to run a detection algorithm on a pre-trained dataset, and find pre-classified objects, it is often necessary to detect objects not available in a dataset. Datasets such as the VOC2012 have only 20 trained classes [16]. If a desired object is not trained in a dataset, it is necessary to train another classifier to include it. Figure 5 demonstrates image detection of the twenty classes within the VOC2012 data set.

Figure 5 - VOC2012 Classes [16]

For this project to be considered feasible for a robotic system, it should be able to run on a consumer grade GPU. The definition of this is a card that is less than that of a “Workstation” GPU. Work station GPUs are commonplace in the deep learning community. This project should be able to run on a card that is of low thermal dissipation. Ideally, this system would utilize the NVIDIA Jetson TX1. The Jetson TX1 uses NVIDIA's Maxwell architecture. This board has 256 CUDA cores, and can provide over 1 TeraFLOP of performance on a 64-bit CPU while only dissipating 15 watts while under full load [17]. The specifications of the TX1 are shown in Table 1.

Table 1 - NVIDIA Jetson TX1 Specifications [17]

Part	**Specification**
GPU	NVIDIA Maxwell ™, 256 CUDA cores
CPU	Quad ARM® A57/2 MB L2
Video	4K x 2K 30 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (10-Bit Support)
Memory	4 GB 64 bit LPDDR4 25.6 GB/s
Display	2x DSI, 1x eDP 1.4 / DP 1.2 / HDMI
CSI	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.1 (1.5 Gbps/Lane)
PCIE	Gen 2 | 1x4 + 1x1
Data Storage	16 GB eMMC, SDIO, SATA
Other	UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0
Connectivity	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
Mechanical	50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)

The final requirement for this project is frame rate. When selecting a method to be used, it should be able to run at a moderate framerate A moderate frame rate in this context is defined as at least five frames per second (FPS). While this framerate is slower than that of the camera that is used, it is still sufficient. Five FPS is sufficient because neither Minion or Blackfinn travel at high speeds. When in motion, it is likely there will be many frames taken of each object before the object is out of frame. Minion travels at a maximum of 7 knots, which can be converted to 3.5 meters per second. Therefore, Minion would never travel more than 0.7 meters between frames. Blackfinn travels at a maximum of 0.5 meters per second. This allows for Blackfinn to travel 0.1 meters between frames. As all the task objects are stationary, this allows for sufficient visual coverage. Due to the anticipated reliability of image detection, it is only necessary to have an object visible for one frame to correctly detect it. While traditional computer vision methods would be expected to process at a faster rate, it is reasonable to accept a slower frame rate for this project due to the boost of accuracy and precision.

1.4. RobotX Dataset

The Maritime RobotX Competition is an international competition sponsored by the AUVSI Foundation. The goal of this competition is to design an autonomy and propulsion package for an ASV that can complete a variety of navigational, detection, and classification based tasks. Each team is given a WAM-V, which is a 16-foot-long inflatable pontoon boat as a platform, which is used to complete six tasks [18]. These tasks require the capability to autonomously detect objects to complete each task's objective. This data for this competition was collected in December 2016, on Sand Island in Honolulu, Hawaii. This dataset contains 24 unique classes, which are found in the following tasks. These classes are listed in Table 2. The person class was added as a side experiment and was not intended or used for this research.

Table 2 - Table of Objects for the RobotX dataset

RobotX Class Names
black_ball
black_buoy
black_tower
blue_buoy
blue_circle
blue_cruciform
blue_tower
blue_triangle

green_buoy
green_circle
green_cruciform
green_tower
green_triangle
orange_ball
person
red_buoy
red_circle
red_cruciform
red_tower
red_triangle
white_buoy
yellow_buoy
yellow_tower

1.4.1. Navigation

The first task in the RobotX Competition is navigating the qualifying gates. This task requires the ASV to travel between two navigational buoys, travel a variable distance, and then exit through two more identical buoys. These buoys are placed in the configuration shown in Figure 6.

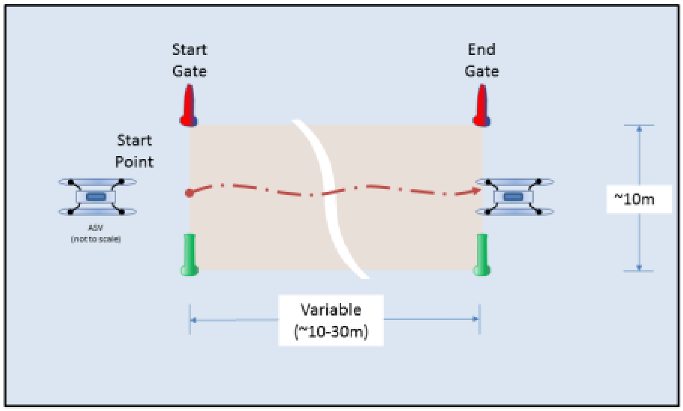


Figure 6 - RobotX Qualifying Gate Task [19]

These buoys are Taylor Made Products Sur-Mark Can Buoys, which have a distinct shape and color [19]. The two shapes of buoys used are shown in Figure 7. For this task, each pair consists of a red and a green buoy, which the ASV is required to detect; without detection, it would not be possible to navigate between them. As this is a qualifying task, it is required to be completed before any other task can be attempted. As these are similar shape, and only differ by color, it is imperative that the classifier algorithm is color independent. Other tasks require the capability to detect blue, yellow, and white buoys.

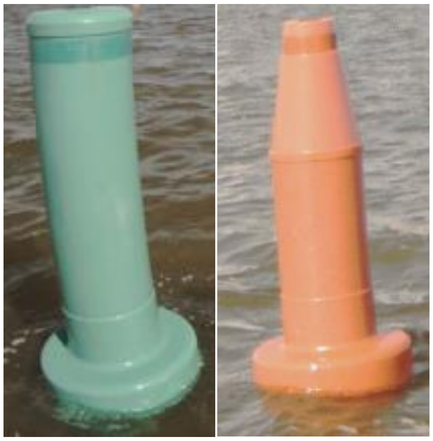


Figure 7 - Green and Red Taylor Made Navigational Buoy, Daytona Beach, FL

1.4.2. Scan the Code

The Scan the Code task is comprised of a light tower on a floating buoy, with three outward facing LED panels. The three LED panels are arranged in a triangle, and are between 1 and 3 meters above the water. Each panel is 15.2” by 7.6” and can display red, green, yellow, blue, or black [19]. The tower has a randomly generated sequence of four colors that is repeated every 5 seconds. The first color in every sequence must be black and the following three colors are randomly selected. Each color is only displayed for 1 second before switching to the next. In between sequences the panels display black for two seconds. The goal of this task is to correctly identify the color sequence of the panels. A rendering of this task is shown below in Figure 8.

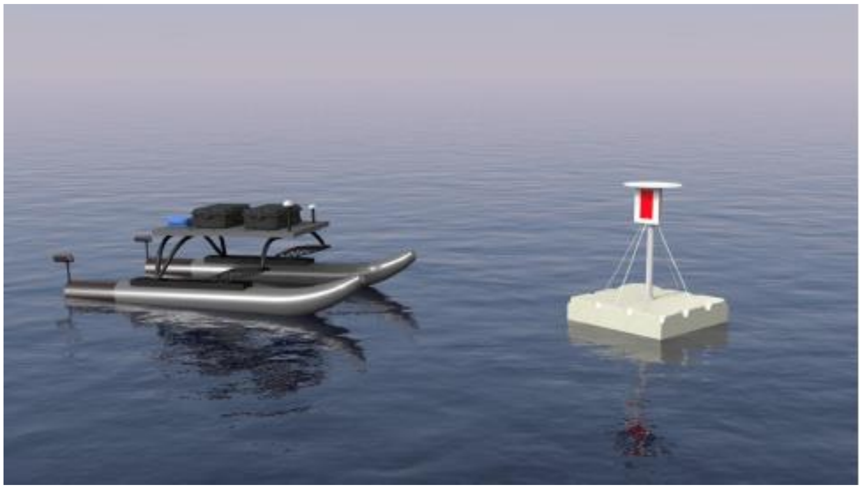


Figure 8 - Scan the Code, Rendering

This task requires detection for two components. The primary component requires the light tower to be detected. This would allow for the ASV to approach the obstacle and begin the task. The second component of this task is identifying the color sequence. This requires the classifier to be able to recognize the color of each LED panel. This is a particularly challenging task since the panel quickly changes color and the colors can appear to drastically change due to weather conditions.

1.4.3. Identify Symbols and Dock

The Identify Symbols and Dock requires the ASV to locate the correct docking bay, navigate into it, stop, and then back out. Each docking bay has a randomly selected sign associated with it. These signs have large geometric shapes which can have different orientations and colors. The possible shapes are either a cruciform, circle, or triangle, and the colors can be red, blue, or green. The judges, before the start of the run, determine which symbol is associated with the correct bay. Because of the combination of shapes and colors, this task requires the detector to detect and classify nine permutations. These signs should be detected and classified in a single image frame. A rendering of this task is shown in Figure 9.

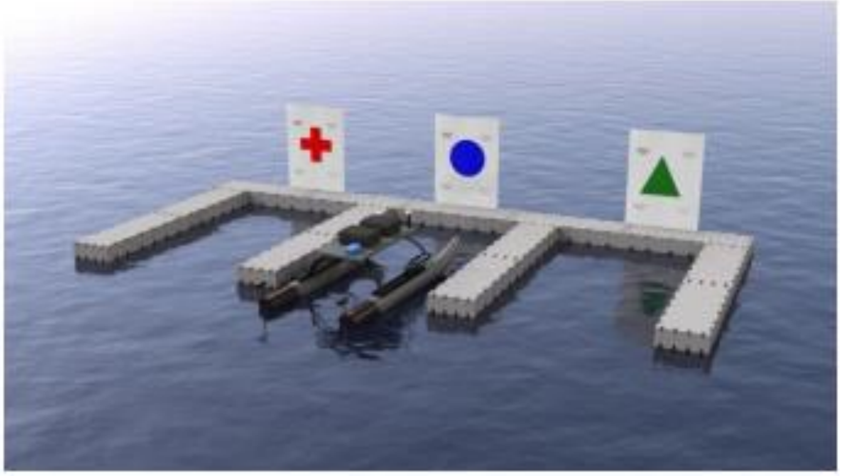


Figure 9 - RobotX Docking task, rendering [19]

1.4.4. Detect and Deliver

As with the docking tasks, Detect and Deliver requires the ability to detect and classify signs with varying shapes and colors, and the sign will be determined by the judges prior to the start of the run. The vessel must circumnavigate a floating tower to find the correct. This adds complexity to the detector since there is a greater chance the camera will be off axis from the sign. Being off axis makes the shapes appear to be skewed, which increases the difficulty of classification. Figure 10 shows a rendering of this task.

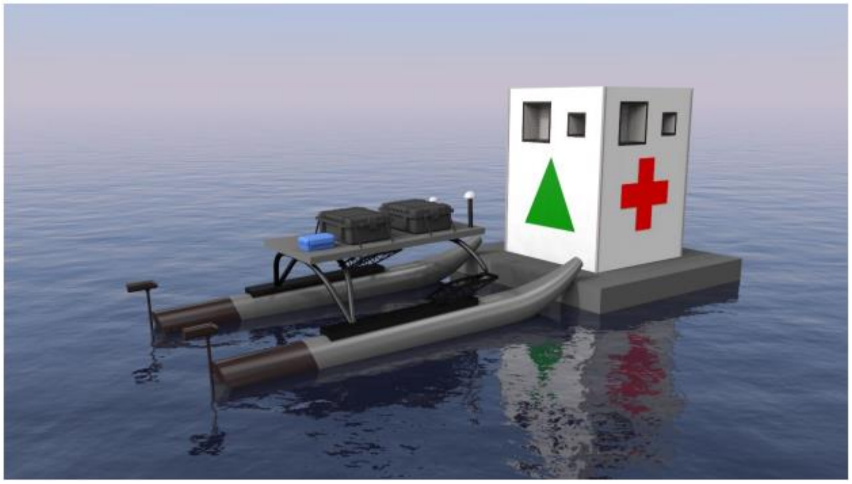


Figure 10 - Detect and Deliver, rendering [19]

1.5. RoboSub Dataset

The RoboSub Competition is an international competition sponsored by the AUVSI Foundation. This competition is composed of approximately forty teams from the United States, and around the world and takes place in San Diego [20]. Each team is tasked to develop an AUV, which is fully autonomous and has no outside communication during a mission. This competition is composed of navigational and manipulation tasks. These tasks are shown in the course diagram, Figure 11. This competition utilizes the TRANSDEC Anechoic Pool which is divided into two identical complete courses. These courses are composed of various tasks, six of which require visual detection. This dataset contains 11 unique classes, which are found in the following tasks. These classes are listed in Table 3.

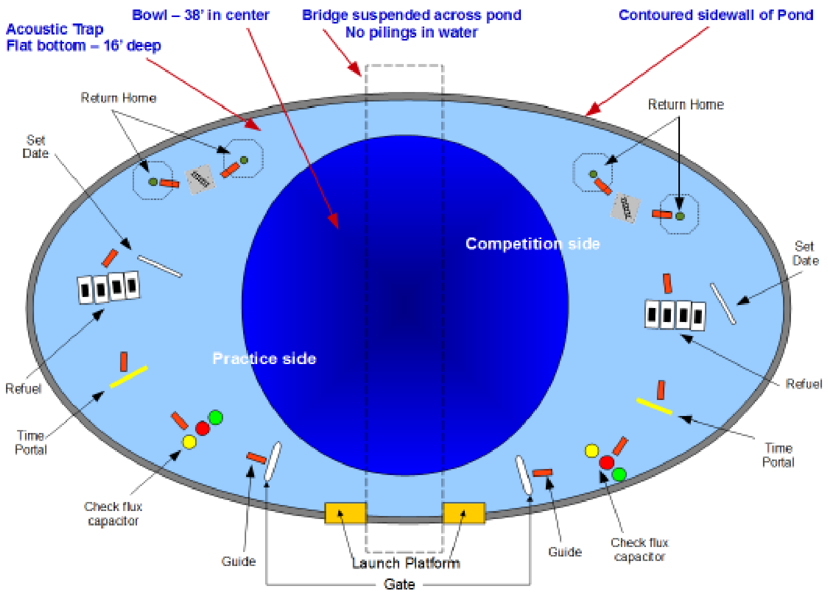


Figure 11 - RoboSub Course Layout [21]

Table 3 - Table of Objects for the RoboSub dataset

RoboSub Object Name
bin_ banana
bin_can
bin_lightning
bin_orange
gate
gate_inv
green_buoy
path
red_buoy
torpedo_board
yellow_buoy

1.5.1. Validation Gate

The Validation Gate is the first task that is encountered, and the only required task. This Validation Gate is a five-foot-tall, ten-foot-wide arch built from 3-inch diameter orange PVC pipe [21]. The goal of this task is to maneuver the AUV through the center of the gate. To complete this task, it is important to have a classifier that can detect the gate. This gate is a difficult object to detect due to its width. Due to the water quality, it is difficult to detect the gate from far away, but when close, it is difficult to fit the entire object into a frame. The diagram of this task is shown in Figure 12.

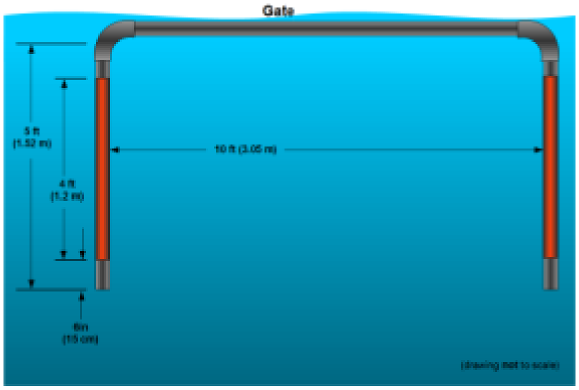


Figure 12 - RoboSub Validation Gate [21]

1.5.2. Buoys

The buoy task is the second course element to be encountered. This task is composed of three 9-inch diameter buoys, suspended from the pool bottom [21]. For this task, there will be three separate colors: red, yellow, and green. These buoys will be within a three-foot-tall vertical box, and have four feet of separation between them. This task requires a classifier to correctly identify the color of each buoy simultaneously. Points for this task are awarded for bumping one specified color buoy, backing up, and then bumping another color. This requires for the classifier to be able to detect the buoys from a far distance, as well as close. This task is challenging due to the similarity of colors. Due to sediment in the water, as well as color absorption, the red and yellow buoys appear similar in color. Additionally, the green buoy can blend in with the murkiness of the water. A diagram of this task is shown in Figure 13.

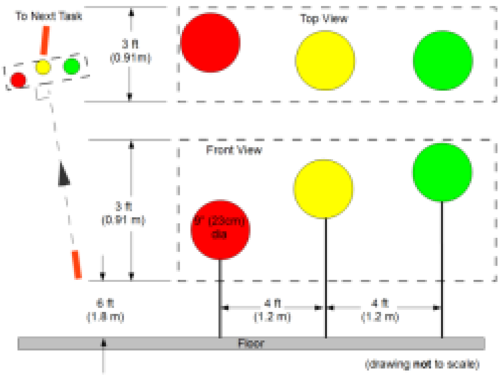


Figure 13 – Diagram of RoboSub Buoy Task [21]

1.5.3. Inverted Gate

The inverted gate differs from the validation gate as it is smaller and inverted. This gate is only four feet tall and eight feet wide [21]. This task is constructed from two-inch diameter PVC pipe, and is yellow in color. The goal of this task is to travel through the center of the posts. Bonus points are awarded if the vehicle is beneath the top of vertical posts. To complete this task, it is important that the classifier can detect both this obstacle's width, but also its height. For the Validation Gate only the object's width was required to be detected. A diagram of this task is shown in Figure 14.

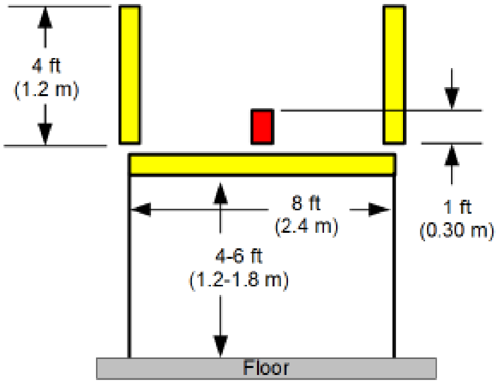
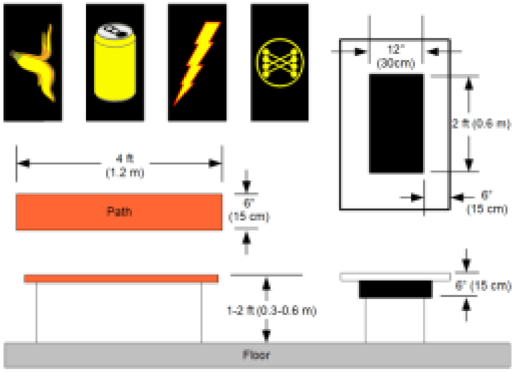


Figure 14 - Diagram of the Inverted Gate Task [21]

1.5.4. Dropper Bins

The dropper bins are perhaps the most advanced detection task in the competition. This task requires the capability to not only detect where the dropper bins are, but to individually classify which one is which. Unlike the other tasks, which are in front of the vehicle, this one is beneath. To observe this task, a downwards viewing camera is required. This adds complexity to the detector as the pool's bottom is coated in sediment and debris. All four bins are surround by a white rectangle, which makes it possible to observe due to the contrast from the pool floor. However, in each bin is a different symbol. These symbols change every year to meet the theme of the competition. In the 2015 competition, which is shown in Figure 15, the shapes were a banana, soda can, lightning bolt, and a flux capacitor. These shapes are yellow on a black background. This requires an advanced classifier to be capable of correctly identifying these abstract shapes.



(drawing not to scale)

Figure 15 - Dropper Bin symbols and dimensions [21]

1.5.5. Torpedoes

The requirement for the torpedo task is that the AUV is required to launch a torpedo through one of the four holes on the board. There are two sets of square holes. Each set has a 12-inch-wide hole and a 7-inch-wide hole. A one inch red border is placed around each hole [21]. Each set is identified by a number above both holes. At the time of the run, one of these sets is designated the primary target. Extra points will be awarded for getting a torpedo through this set. Additional points will be awarded by getting the torpedo through the smaller of the holes. This task not only requires for the yellow board to be detected in the pool, but also the holes and the identifying numbers. A diagram of this task is shown in Figure 16.

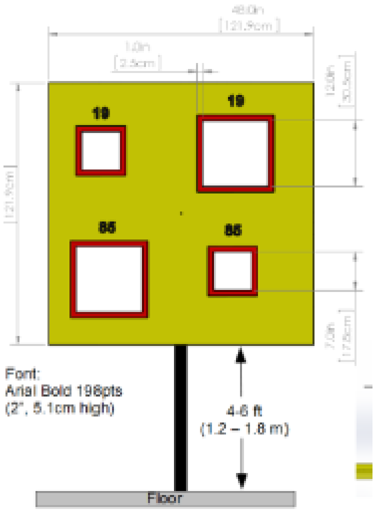


Figure 16 - Torpedo Board Diagram [21]

1.5.6. Path Markers

The Follow the Path task consists of several blaze orange Path Markers throughout the course. These Path Markers are four feet long, half a foot wide and are suspended one to two feet from the pool floor [21]. These Path Markers are used to guide the AUV from one course element to the next. These markers point in the direction of the next task. Therefore, not only is it required to detect these elements, but their orientation must also be calculated. This research will focus on just detection of the paths. However, since the detector will give a bounding box around the object, in the future it would be simple to calculate the angle of the path. A diagram of this task is shown in Figure 17.

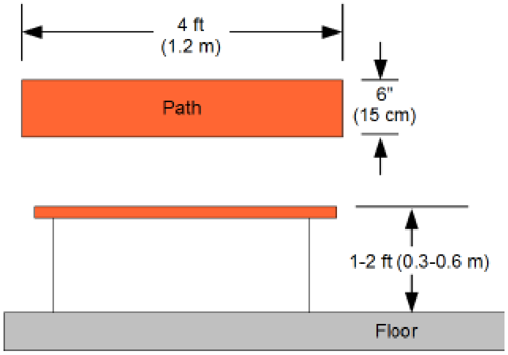


Figure 17 - Path Marker diagram [21]

2. Literature Review

Computer vision is a field that has been used since the early 1970s for object recognition, detection, and tracking [22]. While early researchers thought it would be simple to write an algorithm to take a camera stream and “describe what it saw”, this was not the case [22]. At the time, there was not enough research in artificial intelligence or the hardware to support such processing. Since then, significant research has been performed on creating detectors that are capable of this high level of detection. This section will discuss published research projects which are relevant to this research.

Students at the International Research Institute MICA investigated the use of background subtraction techniques to detect boats [23]. The methods used where Mixture of Gaussians (MOG) and Visual Background Extractor (VIBE). Results show that background subtraction is not sufficient on its own for reliable maritime detection. Both methods had difficulty detecting stationary vessels. If the vessel was not moving, it would be classified as background and subtracted. While VIBE was more efficient, both methods had a low detection rate due to background clutter and movement.

To perform maritime monitoring, a hybrid foreground detection algorithm was tested [24]. This method combined an existing foreground object detection method with image color segmentation techniques to boost accuracy. This method attempted to perform foreground detection, then filter the results with color segmentation and thresholding. This method requires a background reference image. While this works for stationary detection, it is not possible to implement with a moving camera. It is hard to evaluate the success of this method as no formal results are given. The only results given are several images demonstrating the effectiveness of the method. This method is capable of detecting that an object is moving in the frame, though there is no capability to detecting what the object is. Additionally, the accuracy is extremely poor and detects other background noise such trees and waves, and is extremely susceptible to the moving object's wake. While this research offered a good attempt at maritime object detection, other reviewed literature showed more promising results.

Research at the University of Reading performed maritime object detection with visual saliency [25]. Once a saliency map is created, it is filtered through adaptive hysteresis thresholding. After thresholding, a binary image of regions of interests remains. This method could provide reliable object detection and tracking with few false positives. Unlike the background subtraction attempts, this method is capable of filtering anomalies such as waves and boat wakes. However, it had no ability to classify the detected objects.

Joint research between the Naval Research Lab, University of Nevada, and Knexus Research Corporation investigated the usage of machine learning to detect and classify several types of boats. Investigated techniques included Histogram of Oriented Gradients (HOG), Exemplar-SVM (ESVM), and Latent-SVM with Deformable Part Models (LSVM) [26]. These classifiers were trained to detect the following classes: cabin_cruiser, canoe, kayak, motorboat, paddleboard, raft, rowboat, sailboat, and water_taxi. This research showed L-SVM was the highest performing method, though still lacked the ability to classify vessels with less identifying features, such as canoes and kayaks. Under the best circumstances, L-SVM had a MAP of 0.453.

Research was conducted to evaluate the feasibility of Fast R-CNN for sign classification and detection [27]. A set of six SLR cameras were attached to a car to create panoramic images along roads. One hundred thousand images were collected from five different cities in China. Several thousand annotations were used for each class. This method could successfully detect and classify signs with an accuracy of 0.88 and a processing time of 0.3 seconds per image. While slow, this method provided promising results.

In conclusion, there has been a lot of research conducted on maritime detection with the use of visual imagery. However, most of this research is only concerned with detecting the presence of another vessel, and not classifying the type of vessel. No research could be found on the autonomous detection of maritime navigational markers, though road signs are common. Techniques used were background subtraction, foreground object detection, salient detection, L-SVM, and Fast R-CNN. While a few sources can classify the detected object, either no performance metrics were given, or the method was not capable of performing at speeds sufficient for implementation. Little to none research could be found of autonomously classifying objects in near-real-time while on a mobile vessel.

3. Classification and Detection Algorithms Methodology

While image detection and classification sound similar, they are two separate processes. Image classification is the process of algorithmically determining which object, or objects, appear within an image frame. Classification cannot determine where in the fame the object is. In Figure 18 the left image is an example image classification, while the right image shows detection as well as classification. Depending on the method, it is possible to classify more than one object per frame, though most classification methods only allow for one object to be classified per frame.

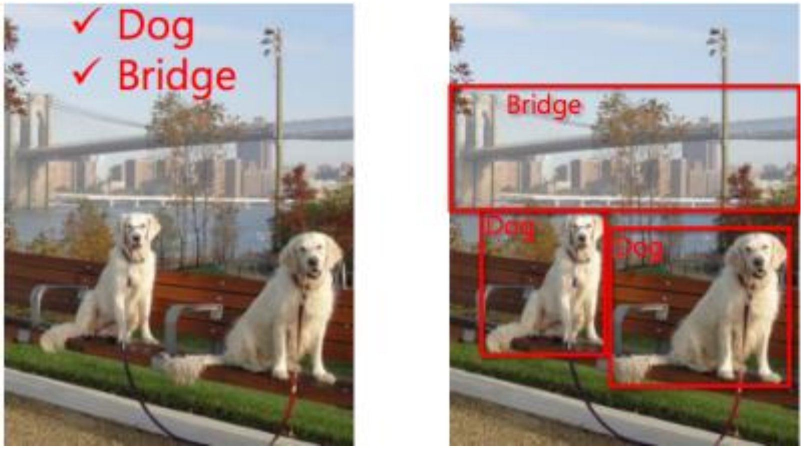


Figure 18 - Image Classification vs Detection and classification [28]

Most methods that are capable of image detection, also include classification. Detection allows for the ability to locate where in the image certain objects may be. This can be paired with a classifier to determine what the object is. Most methods display the location by providing a bounding box around the detected object. In Figure 18, the picture on the right is an example of detection and classification. Depending on the method used, it can either be a single object detector, or a multiple object detector. For this thesis, it would be advantageous to have a method that can detect and classify multiple objects in each frame.

3.1. Methods under Consideration

The four algorithms under consideration in this thesis are Faster R-CNN, TensorBox, DetectNet, and YOLO. Each of these methods are formally published and documented [29] [30] [31] [32]. This was a requirement for a method to be considered. Without documentation, it would be much harder to work with, and implement these algorithms in the time frame of this project. It is a requirement of the selected method that it is both Linux and Windows compatible. Linux is required as the NVIDIA Jetsons have an ARM processor that only supports Linux. Windows is a requirement of the selected method as Minion operates on Windows 7. Currently Blackfinn operates on Windows, though is being switched to Linux. When selecting a method, it is required that it can perform multiple object detection. Multiple object detection is the ability for the detector to detect the location of multiple objects in each frame. Multiple object detection is much more practical in a real word situation because it cannot always be guaranteed that there will not be more than one known object visible in an image frame. Additionally, certain cases may require for there to be multiple objects detected at once. In addition to detection, it is a requirement that these methods can also perform classification on each of the detected regions. For instance, in the RoboSub competition, it is required to identify the red, yellow, and green buoys. This task would not be feasible if the detector could only determine if one of the buoys was in frame. Figure 19 shows an image captured of this task from the 2015 competition, in the TRANSDEC facility. This image shows the Path, Inverted Gate, and Red, Green, and Yellow Buoys.

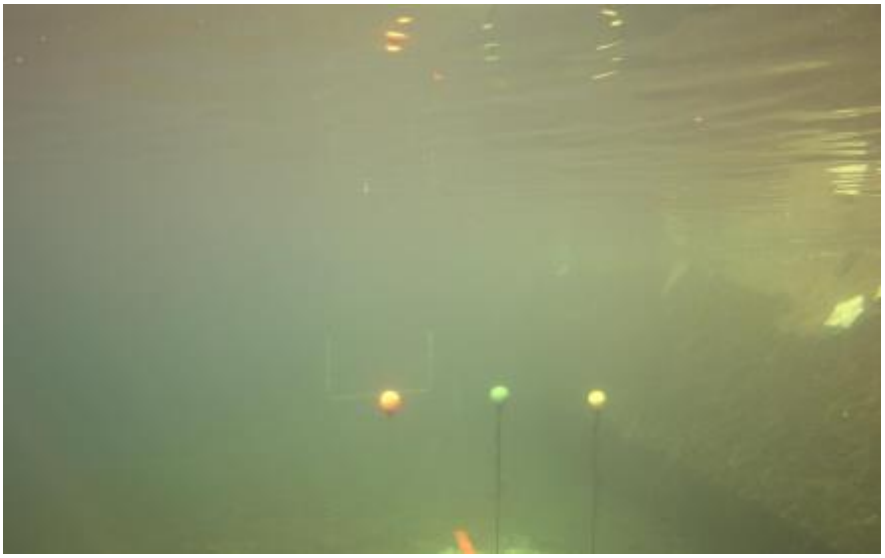


Figure 19 - Path, Inverted Gate, and Red, Green, and Yellow Buoys in the TRANSDEC

3.2. Faster R-CNN

Faster R-CNN is a project that was created by Shaoqing Ren, Kaiming He, Ross Girshick and Jian Sun [33]. It was initially published in their NIPS 2015 paper. Originally, this project was written and published in MATLAB. After publication, a Python reimplementation of their method was released. This reimplementation, py-faster-rcnn, can achieve a similar mean average precision (MAP) as the MATLAB version. This method can achieve a 66.9 percent MAP using VGG16 model on the VOC 2007 dataset [34]. Additionally, the Python reimplementation on average is ten percent slower [29]. This speed decrease is due to the inability for Python to use GPU acceleration on all layers. Faster R-CNN can be run either with or without a GPU. The Python reimplementation is the version that will be considered. This is due to the ease of implementation as well as integration with other software systems. Additionally, it would be less complicated to run the Python version on a Linux system, compared to the MATLAB version. While MATLAB has Ubuntu support, it is known to have support issues. Faster R-CNN allows for the training of a custom dataset, which meets the qualifications. Figure 20 shows an example of Faster R-CNN's capability of detecting and classifying multiple objects in an image.

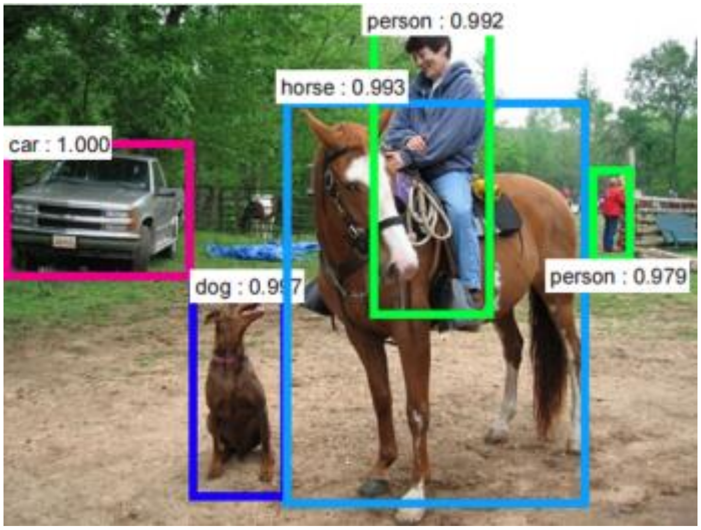


Figure 20 - Faster R-CNN Example [35]

Faster R-CNN is built upon the Caffe framework. Caffe is a deep learning framework that was created by a team at the Berkeley Vision and Learning Center (BVLC), as well as by community contributors [36]. Caffe is open sourced under the MIT License project and is a popular framework for many deep learning projects. As Caffe is written in C, using it, or any projects based off it, would be easy to implement for any project requiring cross platform compatibility. The Caffe framework allows for CUDA acceleration when an NVIDIA GPU is accessible.

Faster R-CNN is an improvement over its predecessor algorithm, Fast R-CNN. Fast R-CNN is another project by Ross Girshick. Faster R-CNN is on average one hundred and forty-seven times faster than Fast R-CNN [34]. This speed increase was achieved through the usage of region proposal networks. Additionally, Fast R-CNN is also an improvement over Girshick's R-CNN algorithm. Fast R-CNN is two hundred and thirteen times faster at runtime than R-CNN [33]. R-CNN was initially published in 2013, while Fast R-CNN was published in 2015.

3.3. TensorBox

TensorBox is an open sourced project that uses the TensorFlow framework to implement Google's GoogLeNet-OverFeat algorithm [37]. TensorBox was initially uploaded to GitHub on January 23, 2016 by user kupel. TensorBox is an image detector that is written in Python and implements Tensorboard. Tensorboard is TensorFlow's, graphical user interface (GUI) which is used to visualize the learning process of the network. TensorBox is capable of being trained on a custom dataset with multiple classes. TensorBox uses the JSON file format. These files contain the filename, and the bounding boxes for each class within each image. Figure 21 shows an exaple of TensorBox detecting multiple objects in a image.

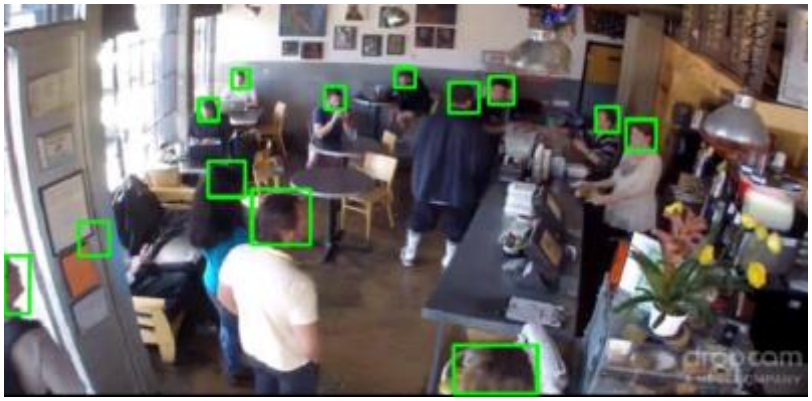


Figure 21 - TensorBox head detector example [37]

TensorFlow is an open sourced software library originally created by engineers at Google and was released in November 2015 [38] [39]. TensorFlow can run on one or more GPUs for CUDA acceleration [40]. When an NVIDIA GPU is not available, TensorFlow may also be run on one or more CPUs; however, it will be significantly slower. TensorFlow is compatible with Windows, Mac OS X, and Linux [41].

3.4. DetectNet

DetectNet is an image detection algorithm created by developers at NVIDIA. DetectNet is capable of image object detection as well as image segmentation. This method however does not support classification. DetectNet is implemented by using a network that is derived from the GoogLeNet model. This network was modified for improved object detection [42]. Figure 22 shows an example of DetectNet being used to detect construction vehicles on a work site.

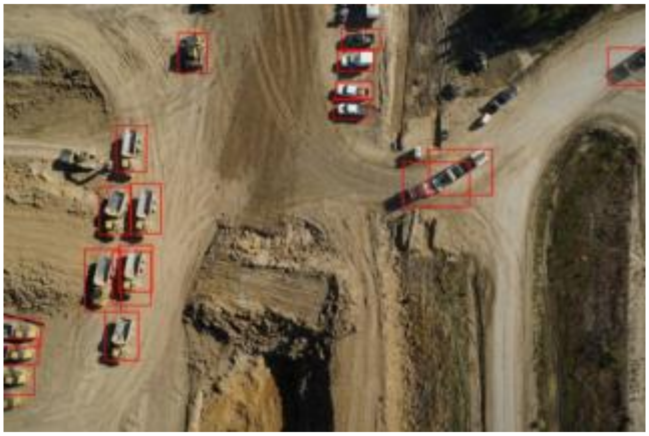


Figure 22 - DetectNet Example, Vehicle Detection [31]

DetectNet is based upon the DIGITS framework. DIGITS is an open sourced project that is supported and maintained by NVIDIA. NVIDIA DIGITS version 1 was initially released on June 26th 2015, though it is currently on version 5 [43]. At the time of selecting a method, the most recent version was version 4. DetectNet was initially released with DIGITS version 4RC, on June 21st, 2016 [44]. The DIGITS library supports CUDA acceleration, as well as CPU processing.

Through the DIGITS platform, it is possible to train a network on a dataset. The DIGITS framework provides support on inputting annotation files, training datasets, as well as validation datasets [31]. Additionally, there is graphical support for editing training parameters. Editable training parameters include batch size, learning rate, and snapshot intervals. The DIGITS frame work is only supported for Ubuntu 14.06 and Ubuntu 16.04 [44].

3.5. YOLO Version 1

The final method that was investigated was the use of YOLO. YOLO, You Only Look Once, is based on the Darknet framework [45]. Darknet is an open sourced framework for neural networks that is written in C and CUDA [32]. YOLO claims to be one hundred times faster than Faster R-CNN. YOLO makes claims to perform detection at 45 FPS, while the Tiny YOLO model can perform at 155 FPS [45]. This Tiny YOLO model only requires 516MB of GPU memory. These framerates were achieved while running on a NVIDIA Titan X. The specifications of the Titan X are shown in Table 4 [46].

Table 4 - NVIDIA Titan X Specifications

Component	Specification
GPU Architecture	Pascal
Frame Buffer	12 GB G5X
Memory Speed	10 Gbps
Boost Clock	1531 MHz
Graphics Card Power	250 Watts

YOLO when trained on the VGG-16 network could achieve a 66.4 percent MAP, at 21 FPS. While this is much faster than Faster R-CNN, it is considerably less accurate. Figure 23 demonstrates how YOLO performs analysis on the entire image at once (top image).

Once weights for the entire image is obtained, the highest weight for each region is saved (bottom image). The final detection step (right image) uses these classified regions and places a bounding box around each separate region.

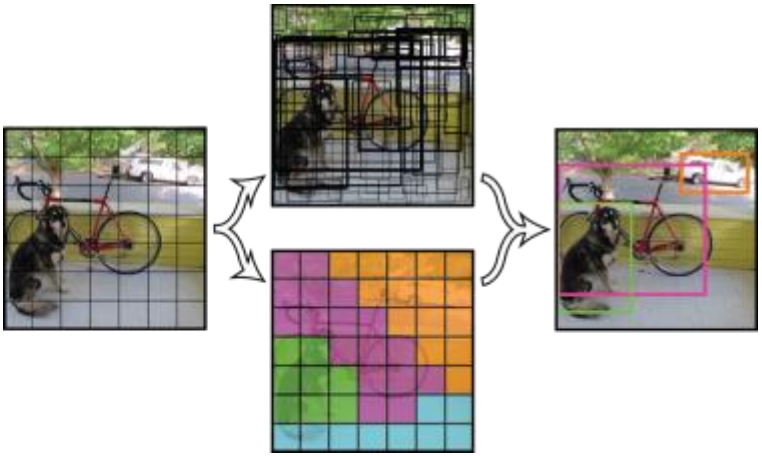


Figure 23 - YOLO Image Detection Example

3.6. Selection Process

When considering which algorithm to select, there were many factors to consider. The three main criteria for evaluation was the operating system it can run on, speed in which it can process an image, and precision of its detection. The selected method would have greater practicality, and usage, if it can run on multiple operating systems. All four of these selected methods are compatible with Linux, though they are not all officially supported on Windows. This section will investigate if there are any unofficial ports that enable windows support. Speed is an important metric to evaluate. A higher frame rate would increase the feasibility of this project. Unfortunately, it proved difficult to find performance metrics for these methods. Therefore, to get a speed metric it would be required to download and install every method, and then train a network on the RobotX or RoboSub dataset. As this is an extremely time consuming and tedious process, it was not done for every method. Faster R-CNN and YOLO were the only methods that provided speed information. The third requirement is precision. All the methods that were investigated have similar precision and detection capabilities.

For this research, it was decided to implement the Faster R-CNN library for Python. DetectNet was eliminated as a possibility as it does not perform classification, as well as its lack of support for Windows compatibility. This was because classification and Windows capability were a requirement of the selected detector. TensorBox was not further considered as there was no published frame rate. This lack of performance metrics makes comparison a difficult process. Additionally, TensorBox appeared to be difficult to integrate. As per the previous research, it was found that Faster R-CNN had a lower frame rate than YOLO, but a considerably higher MAP. Through experimentation it was found that Faster R-CNN had a more straightforward training process, as well as clearer defined models, and more online support. These reasons made Faster R-CNN the top choice for this project.

This is a suitable method as it was determined that it would be the easiest to implement, had a high ratio of performance to accuracy ratio, and is capable of being added upon. Faster R-CNN was originally written for Linux capability only, but there have been successful ports to Windows. This has been run and tested on both Windows 7 and Windows 10.

3.7. Faster R-CNN Operation

Faster R-CNN is a regional convolutional neural network used for object detection and classification. This object detector is composed of two modules that interact together [33]. The first module is a fully convolutional neural network that creates region proposals. This module is the basis of the detection capability. The second module is the Fast R-CNN detector, which is created by the same author. This detector is executed on each of the proposed regions. The interaction of these two modules is shown in Figure 24.

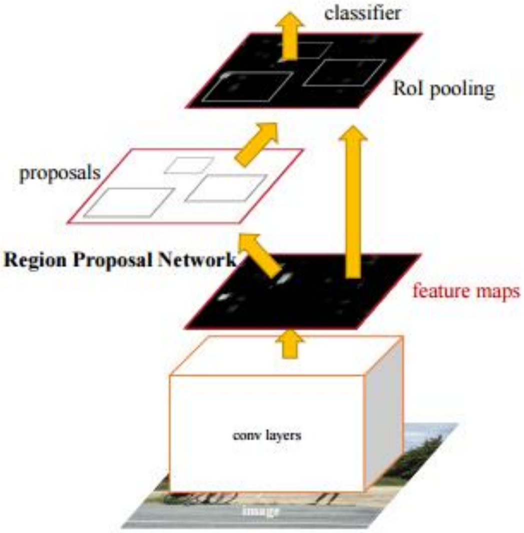


Figure 24 - Faster R-CNN modules

This figure shows a bottom up approach of classification. The bottom-most layer is the input image. This image is sent through a series of convolutional layers. These layers are like those shown in Figure 3. The output layer of this network is displayed as the feature maps layer. This allows for the Fast R-CNN detector module to locate regions of interest in which it should perform on. This allows for greater accuracy and performance increases since the classifier does not need to perform over the entire image. This classifier can take a rectangular input region of any size. For this research the VGG16 model was used, which has 13 shareable convolutional layers. These layers are segmented by a sliding window that has 512 dimensions. A diagram depicting these sliding windows is shown in Figure 25. In this figure the intermediate layer is represented as 256 dimensions from the ZF model, but is easily exchanged for the VGG16 model.

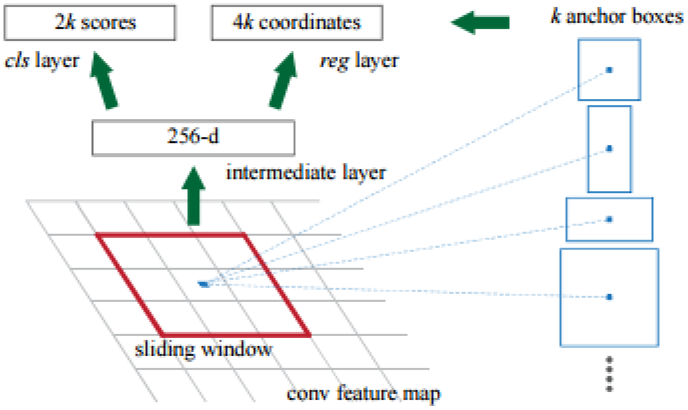


Figure 25 - Faster R-CNN Region Proposal Network (RPN)

For each of these regions cls scores and coordinates are generated. By default, 9 anchor boxes are used for each position of the sliding window. These anchor boxes use 3 scales and 3 aspect ratios. The resulting values from these anchors are used to classify each region.

4. Methodology

To implement an image detector on Minion or Blackfinn, the only sensor required is a digital camera. Visual imagery was provided by using two PointGrey Blackfly cameras. Two cameras were used on each of these vehicles to increase the field of view. These cameras are power over Ethernet (PoE) powered, and GigE compatible. This allows for them to easily be powered and connected on any system. When working in the marine environment, waterproofing these cameras is drastically easier since there is only one cable required for the camera's connection. These 2.3Mp cameras provide 1900x1200 pixel images at an average of 27 FPS [47]. Due to bandwith and processing limitations, these cameras were only sampled at 10 FPS. These Blackfly cameras, paired with Fujinon CS-Mount 2.2-6mm Varifocal Lens, were measured to have a field of view of nearly 100 degrees. On Minion, the cameras are focused outwards at 86-degree angle to achieve a field of view of approximately 200-degrees. A rendering showing this orientation is shown in Figure 26. The cameras are encased inside the tubes and face ± 43 degrees from forward. Blackfinn uses a forward-facing camera and one downward facing camera. This is due to the need to detect objects both in front of, and below the vehicle. While the same cameras were used for obtaining both the RoboSub and RobotX datasets, any digital camera can be used with this project, if the focal length is known.

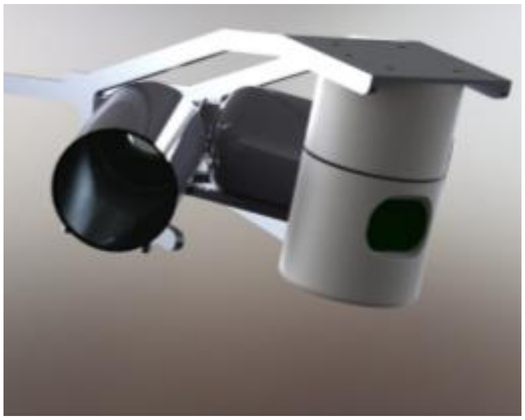


Figure 26 - Minion's Camera orientation

4.1. Installing Faster R-CNN Algorithm

The first step of implementing Faster R-CNN is to set up the programing environment. As previously stated, the goal of this project is a system capable of running on the Linux and Windows operating systems. Linux will be used on Blackfinn, while Minon runs Windows. For the ease of research, most programing will be completed in Linux. This section will be describing the implementation for a Linux system. The usage and compatibility of Windows will be discussed later in this section. Linux was the primary operating system for this project. The Linux operating system used for testing was Ubuntu 16.04 LTS. This was selected because Ubuntu is free and well supported. Additionally, Ubuntu met all the dependency requirements of Faster R-CNN.

The most important dependency for this project were the CUDA drivers. Another reason for Ubuntu section is that NVIDIA makes a version of CUDA 8.0 for Ubuntu. Compatible CUDA drivers are a requirement for this project as they are needed for the CUDA code to be compiled and run on the GPU. Without CUDA drivers, CUDA accelerated code could not be run on the GPU, and therefore could only run on the CPU. This would render the project unfeasible, as CPU operations are 20x slower. In testing it was determined that processing an image on the GPU takes 0.174 seconds while the same method takes 3.242 seconds when processed on the CPU.

CUDA 8 was selected as the version of CUDA to be used. This was done because it is the latest version, and offers support for the Pascal GPU Architecture [48]. This is required to run on NVIDIA's Pascal GTX 10-Series GPUs. As Minion uses a GTX 1080 for vision processing, this was a must. The NVIDIA Jetson TX1 also requires CUDA 8. Therefore, this project needs to be built on CUDA 8 so that it is compatible on a TXI. Additionally, CUDA 8 has other benefits including NVCC compiling that is twice as fast and expanded developer platform which allows for Visual Studio 2015 on Windows and GCC 5.4 on Ubuntu 16.04 [48].

In addition to the many required dependencies, such as the CUDA drivers, Faster R-CNN has many other dependencies. Faster R-CNN requires Caffe as well as the Python libraries such as cython, python-opencv, and easydict [29]. In Ubuntu, these dependences are all available through the APT package handling utility.

As previously mentioned, Faster R-CNN is built on top of the Caffe framework. This requires that Caffe is installed to be able to run Faster R-CNN. Since Caffe is a framework, most projects will implement it, and then build upon it, while making changes to its structure. This process is also followed when using Faster R-CNN. Faster R-CNN adds several layers to the Caffe layers. This requires for Faster R-CNN to only be compatible with a modified version of Caffe. This modified version is available from the same location as Faster R-CNN on rbgirshick's GitHub repository [29].

To install Caffe, rbgirshick's Caffe repository needs to be cloned, and then Make can be used to build the project. The project's build properties can be configured using the Makefile.config parameter file. This configuration file has parameters for the compiler to include other dependencies and corresponding data layers. Important parameters in this file are WITH_PYTHON_LAYER, USE_CUDNN, USE_OPENCV, USE_LEVELDB, USE_LMDB, and CUDA_ARCH. This file should be used to include paths of external dependencies that are also required, such as hdf5.

It is important to enable the Python layer when building Caffe. The Python layer is required because Faster R-CNN is a Python implementation and requires Python layers. If this parameter is not set to true, Faster R-CNN would not be able to be run and an exception will be thrown. Caffe would have to be re-compiled with Python layers enabled.

CUDA Deep Neural Network library (cuDNN), is a GPU-accelerated library created by NVIDIA to improve deep neural network performance. cuDNN provides optimized implementations for common routines such as forward and backward convolution, pooling, normalization, and activation layers [49]. The latest version is cuDNN version 5.1, which was released January 20, 2017 [50]. Version 5.0 and greater supports the Pascal architecture, which is a requirement for this project. cuDNN versions 4 and earlier do not support the Pascal architecture [50]. cuDNN is supported on both Linux and Windows. As Faster R-CNN was released before cuDNN V5, Faster R-CNN does not officially support cuDNN V5. However, there are community created forks of the Faster R-CNN project that support cuDNN V5. For this project, a fork created by GitHub user, TheTesla was used [51]. It is recommended to compile Caffe with cuDNN as it offers many optimizations and speed increases. With the use of cuDNN, training a network that uses 3x3 convolutions is 2.7 times faster. These speed benefits are shown in

Figure 27. This is useful since the VGG network would benefit from this increase. Additionally, cuDNN claims to increase the training speed up to 44 percent faster on a Pascal GPU [52].

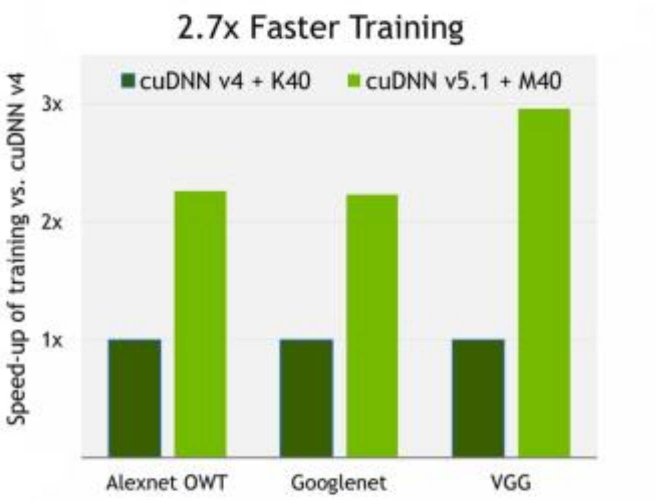


Figure 27 - Speed benefits of cuDNN V4 vs V5.1 on a M40 [49]

Perhaps the most important benefit of cuDNN is reduced memory usage. When training or implementing a VGG16 network, it was found that 5 GB of VRAM is required. Using cuDNN though, this is reduced to only 3 GB. This is massive improvement, and greatly increases the usability of this project. Many popular GPUs, such as the GTX 970, 980, and 1050, only have 4 GB of VRAM. Also, the NVIDIA Jetson TX1 has 4GB of shared RAM. Using cuDNN, these platforms can run this project because they meet the minimum RAM/VRAM requirement.

It is required to enable OpenCV in the makefile.config file. OpenCV is needed for Caffe, because Faster R-CNN uses OpenCV for all image inputs and outputs. OpenCV also allows for efficient and streamlined methods to manipulate images. For this project, OpenCV is used to read and save images, as well as video files. OpenCV also is used to modify loaded images. OpenCV can read and write any standard image (.jpg, .bmp,

.png), as well as read or write any standard video formats (.avi, .mp4). OpenCV was chosen to manipulate images loaded into memory. Manipulation includes adding text and shapes to the image frame. Faster R-CNN was written with support to use the Matplotlib library for these manipulations, as well as displaying images. This library however is limited with its ability to refresh plot windows, to give a smooth viewing experience for a video steam. Due to this, OpenCV was used to rewrite all image manipulation functions. Using OpenCV, the image detector was written so that it can operate on a single image, a folder of multiple images, a video file, or a video stream. Configuration parameters are set to easily allow for the program to be switched between input methods. A camera can be added by either OpenCV's VideoCapture function, or a GigE Camera that can be added by an open-sourced wrapper for FLIR's FlyCapture API (formerly PointGrey). The FlyCapture SDK can be acquired on FLIR's website [53]. The FlyCapture SDK is Ubuntu and Windows compatible, which makes it a good choice for this project.

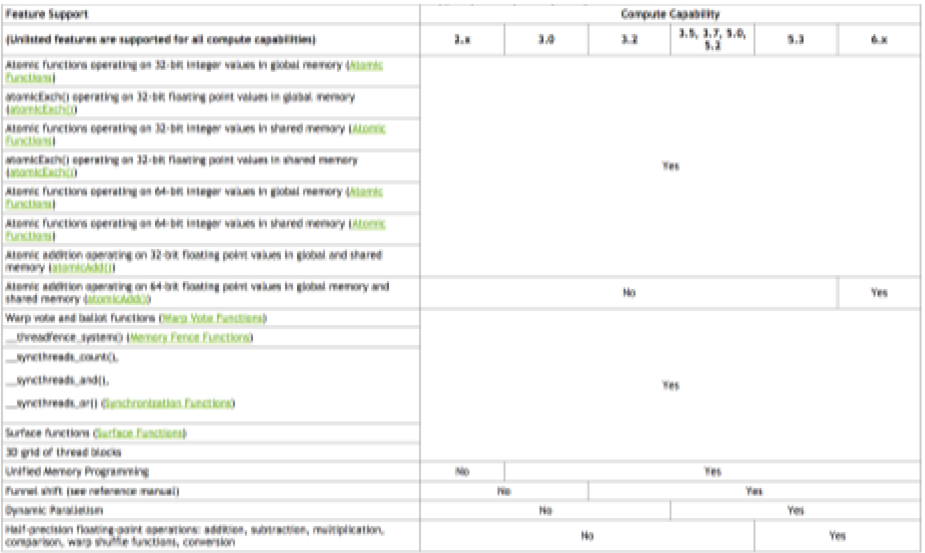
Both LevelDB and LMDB are database files. These should be enabled to increase the efficiency of the code. These formats are used to store the layer data, and are required by Faster R-CNN.

CUDA_ARCH is an important parameter to verify in the makefile.config. This parameter lets the NVCC compiler know what Compute Capability the code should be compiled for. Compute Capability is a metric that is used to identify the capabilities of NVIDIA GPUs. Table 5 shows the Compute Capability of several popular GPUs, as well the NVIDIA Jetsons. Since this project was compiled for a GTX 1080, Compute Capability 6.1 was used. In the configuration file this is dictated as compute_61.

Table 5 - Compute Capabilities of Popular GPUs [54]

GeForce Desktop Products	
GPU	**Compute Capability**
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 1050	6.1
GeForce GTX TITAN X	5.2
GeForce GTX TITAN Z	3.5
GeForce GTX TITAN Black	3.5
GeForce GTX TITAN	3.5
GeForce GTX 980 Ti	5.2
GeForce GTX 980	5.2
GeForce GTX 970	5.2
GeForce GTX 960	5.2
GeForce GTX 950	5.2
CUDA-Enabled TEGRA /Jetson Products	
GPU	**Compute Capability**
Jetson TX1	5.3
Jetson TK1	3.2
Tegra X1	5.3
Tegra K1	3.2

It is important to verify that the selected Compute Capability is correct. If a lesser capability is selected than the card that will be used, performance losses will be severe. If the project was compiled for a lower Compute Capability, then it would have a similar performance to a card of that Compute Capability, despite possibly being a card of higher capability. Figure 28 shows the features of each Compute Capability.



Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)						
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)						
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())						
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No					Yes
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming	No	Yes				
Funnel shift (see reference manual)	No		Yes			
Dynamic Parallelism	No			Yes		
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No				Yes	

Figure 28 - Compute Capability Features [55]

After the parameters for the Caffe Make configuration file is set, the project can be built. Depending on the processor speed and number of processor threads available, this can take up to a few minutes to build. To verify this is built correctly, open a Python terminal and execute “import caffe.” If this is successful, Caffe was properly built and installed. It is likely that the Caffe Python folder’s destination needs to be added to the PYTHONPATH. After Caffe is built, Faster R-CNN can be built. This is done by running Make in the /lib/ source file. Once this builds, Faster R-CNN should be properly installed. This can be verified by running a demo script on a pre-trained network. How to run a network will be discussed in the next section.

When installing on Windows, there are a few differences that make the process feasible, but much more complex. The first issue is Faster R-CNN, or its Caffe dependency, cannot be natively compiled in Windows. Since Caffe is written mainly in C, the code can be ported to work on Windows. When testing on Windows the repository Caffe, by ShaoqingRen was used [56]. This repository is a fork of the original BLVC/Caffe repository. This project was forked on October 1st, 2014 [56]. This project was then modified to be Windows compatible. In addition to code modifications, Visual Studios projects were added. This allowed for Visual Studios 2013 to compile the project.

The “SPP_net” branch of this repository was modified to include the Faster R-CNN layer types. Once this repository is cloned, Visual Studios can be used to build the project. Once the build file for the Python folder is added to Windows PYTHONPATH, the project should be ready to be imported. As with Linux, this can be tested by opening a Python terminal and executing “import caffe.” To install Faster R-CNN, the Linux repository can be used, with some modification. On GitHub, MrGF uploaded a repository called py-faster-rcnn-windows [57]. This repository contains a modified version of the lib directory. This modified directory contains an altered setup.py file that allows for the C and CUDA code to be compiled. This altered setup.py file should overwrite the original.

Another issue with Windows support is the lack of cuDNN V5 support. There is not a Faster R-CNN fork that was modified for Windows, as well as modified for cuDNN V5 support. As the official version of Faster R-CNN has not been officially updated to allow for cuDNN V5, it is necessary for a community member to make this modification. This issue was solved for the Linux version, but not for Windows yet. As of the time of publication, a solution was not available. Without cuDNN V5 support for Windows, this project is still feasible if a GPU with more than 6 GB VRAM is available. cuDNN V4 is compatible with this version, though is not CUDA 8 compatible.

To run Faster R-CNN, there are no official documented minimum system requirements. However, due to the intensive nature of the process, it is obvious that there is a minimum system requirement to handle the computation. Through testing, it was determined that the system must have at least 4GB of RAM. This because the trained

Caffemodel using VGG16 requires 3.5GB of RAM to load. If the system does not have a GPU, computation can all be handled on the CPU, though performance would be greatly reduced. Detailed performance results will be covered in the results section, as previously mentioned. Through testing it has been found that CPU only processing is approximately 20x slower than on a GPU. Therefore, it is highly recommended to have a GPU to boost performance with parallel processing. The smallest popular model for Faster R-CNN is the VGG16 model. This model requires 6GB of VRAM. However, with the inclusion of cuDNN, optimizations can be made which reduce the requirement to approximately 3.5GB of RAM. It is also recommended to have an equal amount of RAM as VRAM. Therefore, the minimum requirement is 4GB RAM and 4GB VRAM. An exception for this is made for the NVIDIA Jetson TX1. The TX1 uses shared memory, so 4GB of RAM is sufficient.

4.2. Running Faster R-CNN

To run Faster R-CNN, a prototxt file, and caffemodel file are needed. These two files are the core of the Caffe framework, and allow Caffe to have such flexibility. The prototxt file is a Caffe file structure used to construct the different layers of the network. The caffemodel file is used to store the trained model data. For this project to run on a custom dataset, the network must have been trained. This is a rigorous and time consuming process. How to train a model will be discussed later in section 4.3 Training.

Before implementing the Faster R-CNN algorithm into another project, or training a network for it, it is recommended to test its performance on a pre-trained network. This would allow for verification if everything was configured properly. This additionally gives a subjective performance metric of how a system compares to another, based on online metrics. When tested, if the speeds are much different than somebody else's speeds, it could be assumed that something was not configured properly. If there was an issue with the CUDA drivers, and this were to be run on the CPU, processing times would be much longer. Faster R-CNN comes with a demo script, which runs detection on five images. After running the demo.py script on the pre-trained Faster R-CNN demo models, the results of the five classified images are shown. Figure 29 shows an example image of this process. Each image took 0.194s, 0.153s, 0.167s, 0.161s, 0.182s, respectfully, to detect. When running this test on CPU an average 3.242 seconds per frame was calculated.

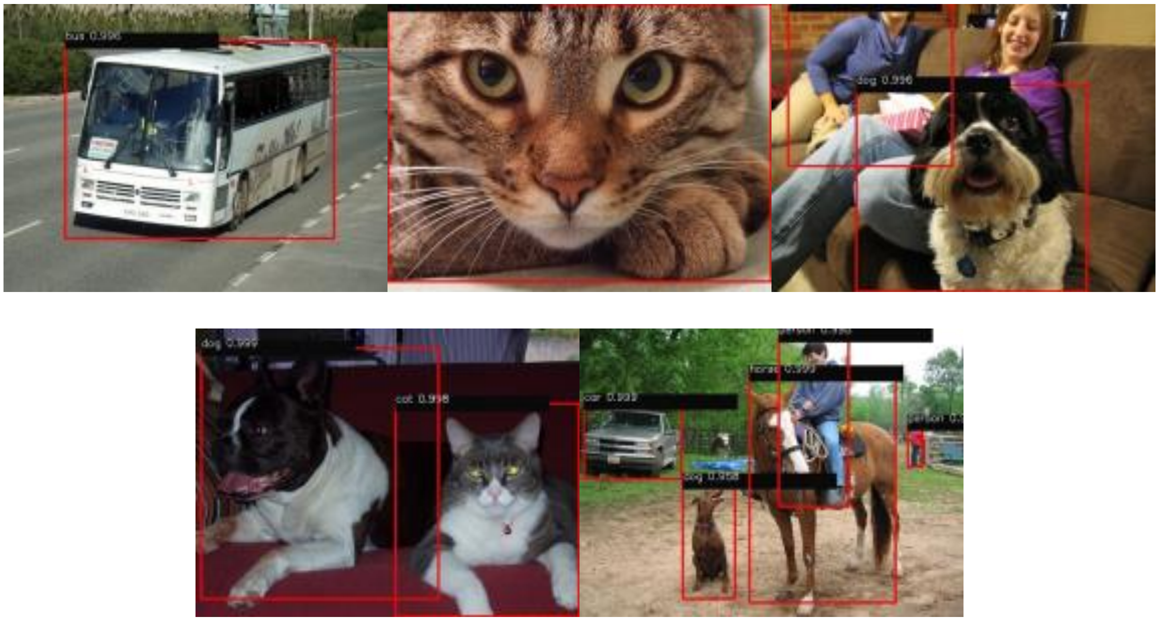


Figure 29 - Results of Demo.py

4.3. Training

Training a network is the process in which the classifier and detector learn to recognize different objects. To train a model on an object, a data set containing many images of that object must be acquired. In this dataset, it is recommended that the images are against various backgrounds, and of multiple orientations of the object. This will ensure that the training is correctly tuned to the object, and not also the background. To train a model for Faster R-CNN, there are a few requirements. The first requirement is that the dataset must be annotated.

Annotation is the process of manually segmenting the objects in each image. Annotations can only be performed on an image, and not to a video file. This was an issue because Minion's camera system stores logged frames as an .avi file. This requires for the frames to be extracted from the .avi for annotation and training. To do this, a Python script was written. This script uses OpenCV to scan a folder, and extract frames from every contained video file. The frames from each video are organized in folders with corresponding names.

Faster R-CNN supports several formats of annotation files. The annotation process requires a separate annotation file for each image. To create these annotations files, the program LabelImg was used. LabelImg can be downloaded on GitHub from Tzutalin's repository [58]. This repository is active, and is continually being updated with new features. This program was modified to include additional shortcuts and hotkeys to speed up the annotation process. LabelImg was selected because it saved the annotation files to the PASCAL VOC format. This format is the same format used by ImageNet, and is compatible with Faster R-CNN. An example of LabelImg annotating an image from the RobotX dataset is shown below in Figure 30. This image shows the blue_circle, red_triangle, and red_cruciform classes being annotated. This image was taken at the 2016 RobotX Competition.

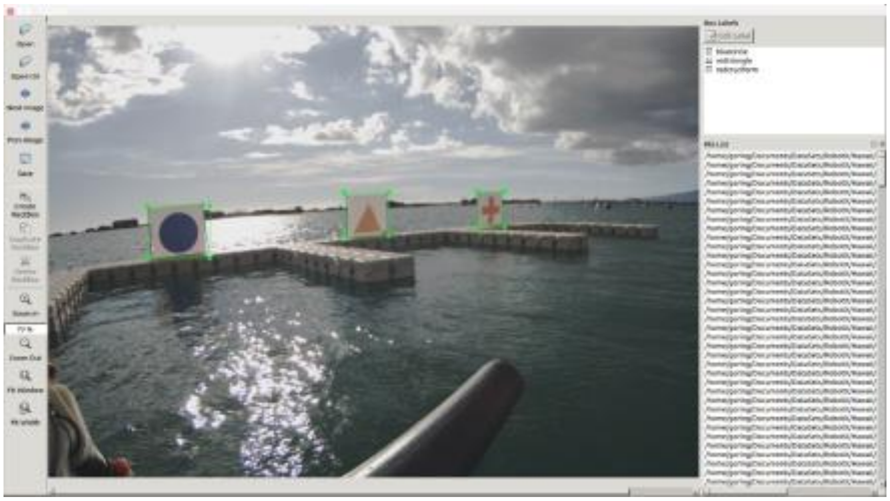
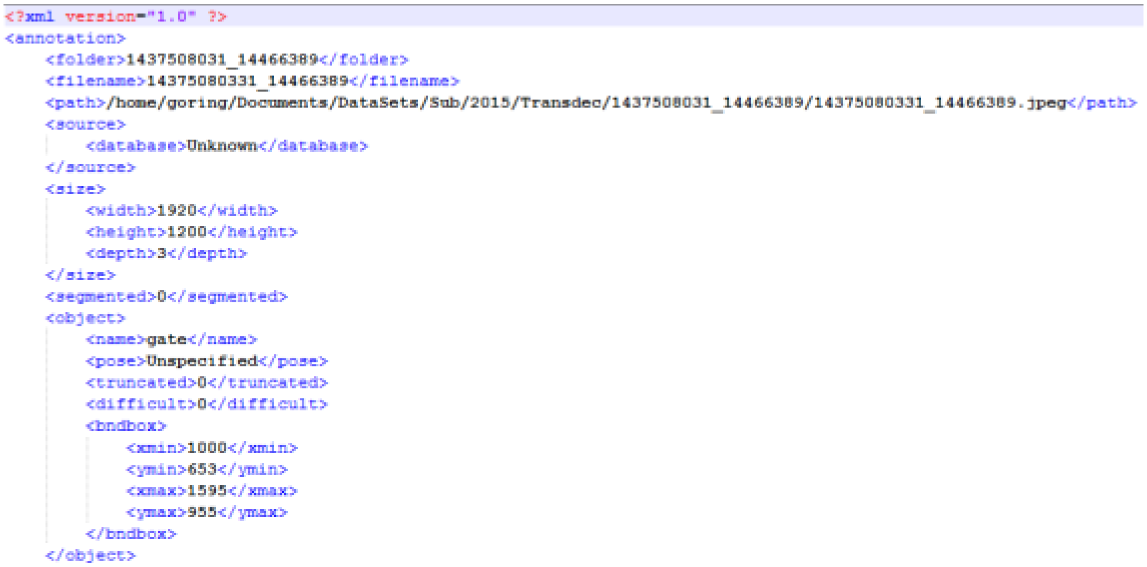


Figure 30 - Using LabelImg to annotate an image from the RobotX dataset.

The annotation files created by LabelImg use the .XML file format. This file has data fields for the source image width, height, and depth. More importantly, for each annotated object, there is an object field. This field contains the name of the object as well as coordinates for the bounding box. The bounding box is listed as xmin, ymin, xmax, ymax. By recording two coordinate pairs, a rectangle around the object can be drawn. This format allows for multiple objects to be included in one .XML file. An example of an annotation file is shown in Figure 31.



```xml
<?xml version="1.0" ?>
<annotation>
    <folder>1437508031_14466389</folder>
    <filename>14375080331_14466389</filename>
    <path>/home/goring/Documents/DataSets/Sub/2015/Transdec/1437508031_14466389/14375080331_14466389.jpeg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>1920</width>
        <height>1200</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>gate</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>1000</xmin>
            <ymin>653</ymin>
            <xmax>1595</xmax>
            <ymax>955</ymax>
        </bndbox>
    </object>
```

Figure 31 - Example Annotation File created using LabelImg

The RoboSub model consists of 11 individual classes. To train this model, 833 images were annotated for a total of 1,355 annotations. These annotations were from a dataset of 92,447 images. All these images were manually analyzed to see if they would be beneficial in the training set. Most these images were not usable as there were no course elements in them. When analyzing these images, a set was pulled aside for accuracy verification and testing. A set from both the forward, and downward facing camera was set aside for this. This set was composed of 4,188 images. While this is large set, most of the frames do not contain any course elements. The breakdown of these annotations is shown in Table 6. These annotations are plotted in Figure 32.

Table 6 - Annotation Summary for RoboSub

Object Name	Number of Annotations
gate	290
gate_inv	138
red_buoy	190
green_buoy	186
yellow_buoy	168
path	287
torpedo_board	15
bin_bannana	24
bin_lightning	27
bin_can	21
bin_orange	9

As the images for this dataset were taken during a competition, it was not possible to obtain equal data for every task. There are the most images available for annotation of the gate, as every run begins with it. This is what allows the detection of the gate to be incredibly accurate. Unfortunately, classes such as the bins, were only trained on logs from one run. Despite a much lower number of annotations, the bins can still be classified with a high MAP. The bins were only trained from one run, as there were only two runs of data for these obstacles. Therefore, one was reserved for training as the other was reserved for testing.

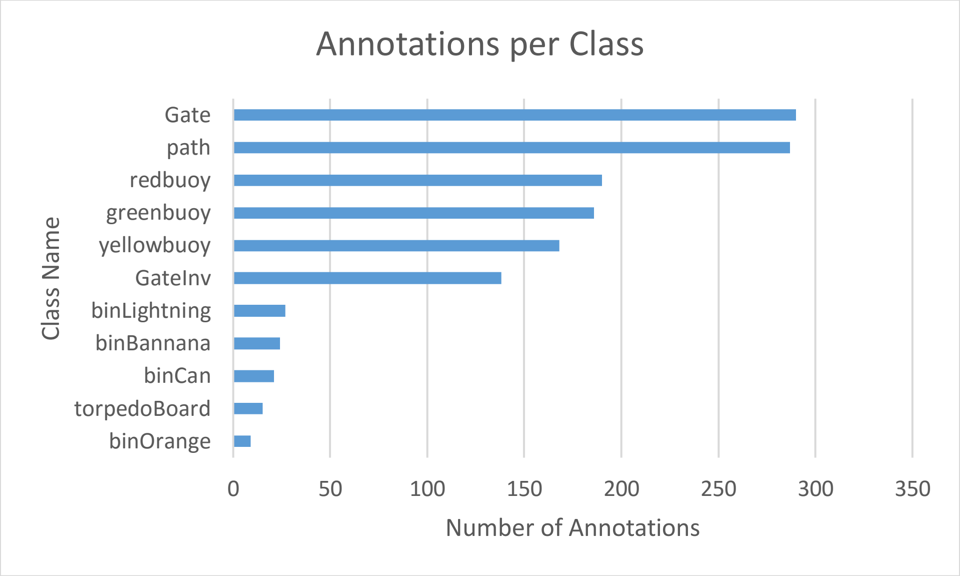


Figure 32 - Plot of Number of Annotations Per Class

Additional experimentation on detecting the torpedo board symbols was performed. Since this was an isolated test, the annotations from this experiment were not included in the final trained model. The summary of annotations for this test is shown below in Table 7. The performance of this test will be evaluated later in 4.5.2 RoboSub Accuracy.

Table 7 - Annotation Summary for Torpedo Board Test

Object Name	Number of Annotations
torpedoboard2016	36
W	21
S	20
N	7
E	7
torpedoboard2016cover	7

While the RobotX dataset has twice the number of classes, it is much smaller than the RoboSub dataset. This dataset is comprised of 23 unique classes. These classes were annotated with 2,365 annotation files. There is a total of 3,207 annotations in this dataset.

This annotated dataset was composed of 50,887 images. This was a tedious process to sort through and annotate. Unfortunately, there is a large discrepancy between the number of annotations taken for each class. As with the RoboSub dataset, this is the result of logging competition data. Included in this dataset is data from tests in Daytona Beach, in addition to the data from Hawaii. As this project was planned to be used for the Light Tower portion of the competition, there is a large focus of Light Tower data. One of the Daytona testing datasets was focused on the blue_circle, making it by far the most popular class. The distribution of annotations can be found in Table 8 as well as plotted in Figure 33. When sorting images for annotation, a set was reserved for verification and testing. This set was composed of 4,105 images. These images were selected from image sets that were not in the training set. These images were selected as they contained all course elements.

Table 8 - Annotation Summary for RobotX Dataset

Object Name	Number of Annotations
blue_circle	382
black_tower	320
blue_tower	239
red_buoy	209
green_tower	202
red_tower	201
green_buoy	196
red_triangle	189
yellow_tower	176
black_ball	163
person	132
red_cruciform	110
white_buoy	101
blue_cruciform	98
blue_triangle	83
black_buoy	82
green_triangle	74
blue_buoy	71
green_circle	50

yellow_buoy	50
green_cruciform	41
orange_ball	22
red_circle	16

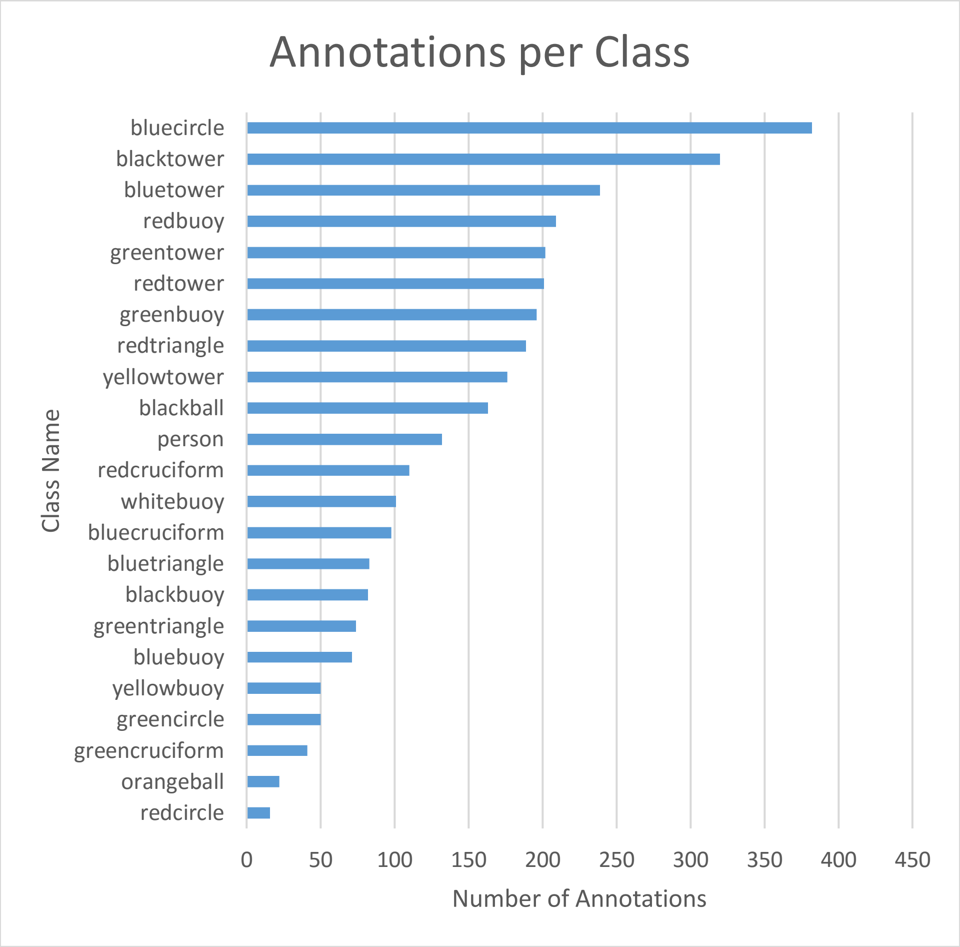


Figure 33 - Plot of Number of Annotations Per RobotX Class

Once each image has an associated annotation file, the training process can begin. As Faster R-CNN is based on the Caffe framework, the training process is like that of training a Caffe network. To train a network, both Caffe and Faster R-CNN must be installed on the computer. While not necessary, it is highly recommended that training be performed on a computer with one or more GPU. It is not required for the network to be trained on the same device that it will be run on. This is especially useful for a system that uses a Jetson TX1, because the TX1 would train a network at a much slower rate than on a computer with a GPU that has a higher clock rate. Additionally, as the TX1 has only has 4GB RAM, training would be a slow process. To limit the RAM used, smaller batch sizes would be needed. By reducing the number of images trained simultaneously in a batch, the training process will take longer. The training process is a long and cumbersome one, that can take several hours to several days depending on the GPU(s) used, and the size of the dataset.

When training a network, it is recommended to use another trained model as a weights model. Due to the relatively small size of the datasets used for this project, less than 100,000 images, training would benefit from using a pre-trained weights file. The weights parameters for each layer are learned through the back-propagation phase. To eliminate the need for this propagation to start from scratch, a pre-trained weight file can be used. Through initial testing it was determined that this was necessary. Due to the limited size of the training and validation sets, the weights would not properly initialize. To resolve this issue, the pre-trained weights file VGG16.V2.caffemodel was used. This file can be obtained from the Faster R-CNN repository. To use the pre-trained weights file the name of the last layer must be changed [59]. When changing this name, it should be done in the .prototxt file. However, for Faster R-CNN to recognize this new layer name as the final layer, it must also be changed in the code for loading the Faster R-CNN model. To change, this the variable “box_deltas” should be changed. This variable is found in ./libfaster/rcnn/test.py. This variable should be changed to the same as the new layer name. The layer name should be changed in the test.prototxt file and the train.prototxt file. The parameter that needs to be changed in both files is the argument “bottom,” for the layer cls_prob. This name can be changed to anything that is not a current layer name. For this project, the last layer's name was changed from cls_score to cls_score2.

When training a model, it is crucial to ensure the train.prototxt file is properly configured. When training a model on a custom dataset, it is likely the training prototxt file would need to be changed from the default. The required changes are as follows:

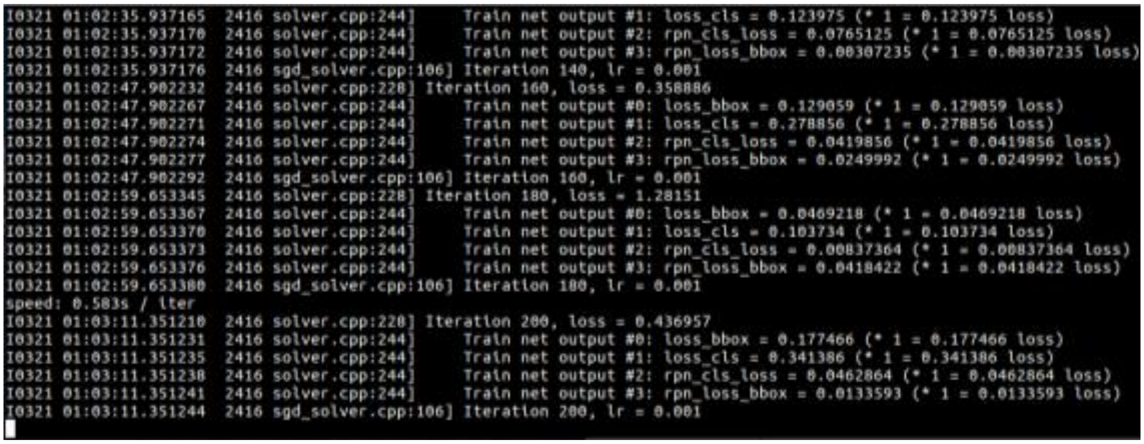
- In 'VGG_ILSVRC_16_layer' the python_param param_str 'num_classes' should be changed to the number of unique classes + 1. For the RobotX Dataset there is 23 unique classes, plus the background class. Therefore, this parameter is 24.
- In 'roi-data' the python_param param_str 'num_classes' should be changed to the number of unique classes +1. Therefore, this parameter is 24.
- IN 'bbox_pred' the inner_product_param num_output should be changed to the (number of unique classes + 1) * 4. Therefore, this parameter is 96.

The test prototxt file also needs to be altered before runtime. The test prototxt file requires the number of classes to be changed as well as the number of parameters for the final layer.

When choosing the parameters, it is required to add one to the number of classes. This is done to account for the background class. This class, which is named '__background__', is used as a negative image for the classifier. This class is automatically set up and is used to reduce proposals in the background region.

Training a network is started by calling the script, faster_rcnn_end2end.sh. This script takes three input arguments. The first argument is the identification number of the GPU(s) to train with. If only one GPU is installed, this argument would be index 0. The second argument is the network type. For this training, the VGG16 network was used.

The third and final argument is the annotation format. For this training the pascal_voc format was used.



```
I0321 01:02:35.937165  2416 solver.cpp:244]     Train net output #1: loss_cls = 0.123975 (* 1 = 0.123975 loss)
I0321 01:02:35.937170  2416 solver.cpp:244]     Train net output #2: rpn_cls_loss = 0.0765125 (* 1 = 0.0765125 loss)
I0321 01:02:35.937172  2416 solver.cpp:244]     Train net output #3: rpn_loss_bbox = 0.00307235 (* 1 = 0.00307235 loss)
I0321 01:02:35.937176  2416 sgd_solver.cpp:106] Iteration 140, lr = 0.001
I0321 01:02:47.902232  2416 solver.cpp:228] Iteration 160, loss = 0.358886
I0321 01:02:47.902267  2416 solver.cpp:244]     Train net output #0: loss_bbox = 0.129059 (* 1 = 0.129059 loss)
I0321 01:02:47.902271  2416 solver.cpp:244]     Train net output #1: loss_cls = 0.278856 (* 1 = 0.278856 loss)
I0321 01:02:47.902274  2416 solver.cpp:244]     Train net output #2: rpn_cls_loss = 0.0419856 (* 1 = 0.0419856 loss)
I0321 01:02:47.902277  2416 solver.cpp:244]     Train net output #3: rpn_loss_bbox = 0.0249992 (* 1 = 0.0249992 loss)
I0321 01:02:47.902292  2416 sgd_solver.cpp:106] Iteration 160, lr = 0.001
I0321 01:02:59.653345  2416 solver.cpp:228] Iteration 180, loss = 1.28151
I0321 01:02:59.653367  2416 solver.cpp:244]     Train net output #0: loss_bbox = 0.0469218 (* 1 = 0.0469218 loss)
I0321 01:02:59.653370  2416 solver.cpp:244]     Train net output #1: loss_cls = 0.103734 (* 1 = 0.103734 loss)
I0321 01:02:59.653373  2416 solver.cpp:244]     Train net output #2: rpn_cls_loss = 0.00837364 (* 1 = 0.00837364 loss)
I0321 01:02:59.653376  2416 solver.cpp:244]     Train net output #3: rpn_loss_bbox = 0.0418422 (* 1 = 0.0418422 loss)
I0321 01:02:59.653380  2416 sgd_solver.cpp:106] Iteration 180, lr = 0.001
speed: 0.583s / iter
I0321 01:03:11.351210  2416 solver.cpp:228] Iteration 200, loss = 0.436957
I0321 01:03:11.351231  2416 solver.cpp:244]     Train net output #0: loss_bbox = 0.177466 (* 1 = 0.177466 loss)
I0321 01:03:11.351235  2416 solver.cpp:244]     Train net output #1: loss_cls = 0.341386 (* 1 = 0.341386 loss)
I0321 01:03:11.351238  2416 solver.cpp:244]     Train net output #2: rpn_cls_loss = 0.0462864 (* 1 = 0.0462864 loss)
I0321 01:03:11.351241  2416 solver.cpp:244]     Train net output #3: rpn_loss_bbox = 0.0133593 (* 1 = 0.0133593 loss)
I0321 01:03:11.351244  2416 sgd_solver.cpp:106] Iteration 200, lr = 0.001
```

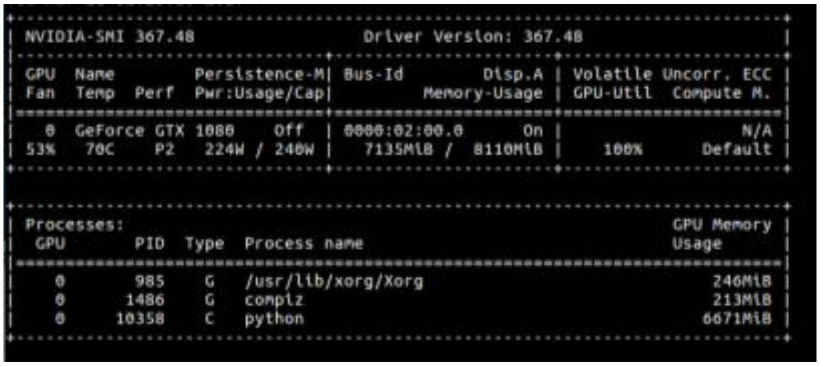
Figure 34 - Terminal window showing output during training

Figure 34 shows the output of the terminal screen while training a network. This screen shows the current iteration that is being trained on. In this screenshot, iteration 200 is being finished. This model is being trained on a computer with the specs shown in Table 9.

Table 9 - Desktop System Specifications

Part	**Specification**
CPU	i7 4790K – 4.3Ghz
RAM	16GB 1600Mhz
GPU	NVIDIA GTX 1080 - 2560 CUDA cores – 8GB RAM
OS	Ubuntu 16.04 LTS

In the screenshot of the training, Figure 34, it is shown that it takes 0.583 seconds per iteration. As shown in the data from Table 9, this was trained on a NVIDIA GTX 1080. While training, the process took 6671 MB of VRAM, which is shown in Figure 35. Additionally, this screen shows other pertinent information such as GPU usage, GPU temperature, and power consumption. This data is accessible by running ‘nvidia-smi’ from the terminal window.



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 367.48                 Driver Version: 367.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    Off  | 0000:02:00.0      On |                  N/A |
| 53%   70C    P2   224W / 240W |   7135MiB /  8110MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0       985    G   /usr/lib/xorg/Xorg                              246MiB |
|    0      1486    G   compiz                                          213MiB |
|    0     10358    C   python                                         6671MiB |
+-----------------------------------------------------------------------------+
```

Figure 35 - Results of NVIDIA-SMI while training

When training, it is recommended to do several thousand iterations. This is necessary for the network to converge. For the RobotX dataset, there were 24 unique classes. To train this model, 100,000 iterations were performed. This training took over 16 hours to complete. Faster R-CNN creates a log file during the training process. This log file can be used to plot the loss curve for the training process. The curve for the training of this network is shown in Figure 36. This plot shows that for this training, the loss quickly settles around 10,000 iterations, though continues to drop until around 65,000 iterations. After this, the lost begins to gradually increase. Additionally, while training snapshot files are created. These files are used to resume training if an interruption occurs. The interval for which these files is saved can be set in a configuration script. When using the model trained from this training, it was decided to use snapshot from iteration 65,000.

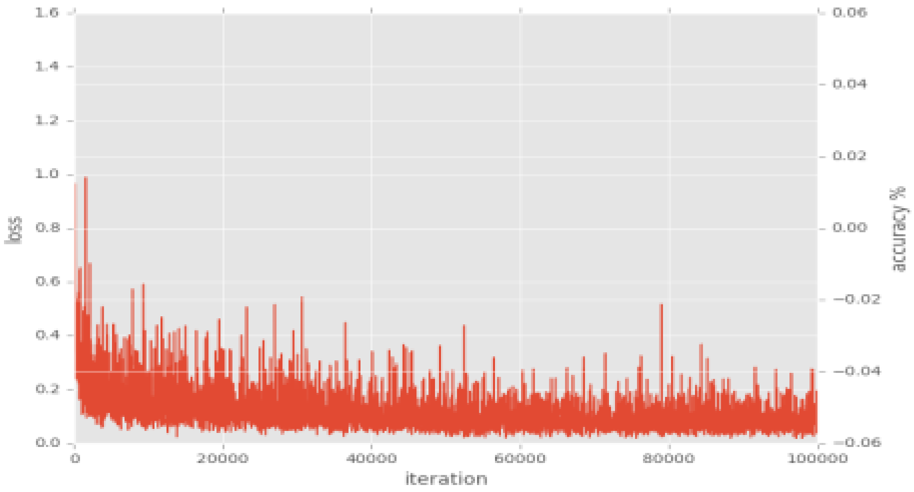


Figure 36 - Loss curve for RobotX training

4.4. Tracking and Localization

In addition to providing classification, this project additionally tracks each detected object. This is done to increase rate of true positives as well as reduce the probability of false negatives. This tracking increases the rate of true positives, because the confidence threshold can be lowered. This allows for the detector to be more sensitive, and classify objects that have lower confidence. While this should raise the false positive rate, this is not an issue. This tracking reduces the probability of false positives as its persistently keeps track of each objects from frame-to-frame. Faster R-CNN’s algorithm is set up to be able to sufficiently detect an object with a single frame, but there is no verification process done to ensure that there is no abnormal change from one frame to another. This is useful because for example, imagine a red buoy in a location. This buoy has been detected for fifty frames consistently and correctly, however the next frame has a sun flare, or another anomaly that causes for the detector to incorrectly classify the buoy as a yellow buoy. By having a persistent tracker, the algorithm would know it is not possible to instantaneously switch to another object. This feature however is disabled for the light panels on the Light Tower. This is because these panels are constantly changing color every second.

To track each found object, a Python class is made for each potential object. This class allows for a memory location to store pertinent information about each found object. Each classified object in each frame, is searched through the database of previously detected objects. If it is determined that the object was not previously found, a new class object is initialized. If the object was previously found, its parameters are updated. To determine which objects in the current frame that has already been found, an exhaustive search is conducted for each object in the database. During the search, the position of each object in the current frame is compared against the position of each stored object. A vector is created between the new pixel coordinate and each old pixel coordinate. As the frames are recorded at 10 FPS, and all course elements are static, meaning they do not move around, there will not be much change in the pixel location of each object between frames. If the magnitude of the vector is less than 10 pixels, it is assumed to be the same object. This value was tested, and determined to be accurate, though can be changed at any point. For each frame an object is determined to be in frame, its life parameter is increased by one. The life parameter counts the number of frames an object has been detected for. This parameter is useful when eliminating false positives. If an object is only detected for one frame, then disappears, it was likely a false positive. This parameter is also used to clear old objects from the database. If an object is not seen for 300 frames, or 30 seconds, it is removed from the database. This is done to purge past objects from the database, reducing computational load and memory usage.

The object class stores many parameters about each detected object. To detect if the object has already been found, the objects pixel coordinates are recorded. Other parameters for each object include, width, height, and area in pixels of the surrounding bounding box. These values are used to calculate a distance from the vehicle to the object. As all the course elements are documented in the task descriptions, the dimensions of all objects are known. Additionally, the object's average width, height, and coordinates are stored in the class. These averages are calculated by a moving average of a first in, first out queue.

To calculate the distance to an object, it is required to know the actual width of the object, the perceived width in pixels, and the focal length of the camera [60]. As the lens used for recording this imagery is a variable focus lens, it is possible for it to have a range of focal lengths. The Fujinon lens that was used has a focal length of 2.2-6 mm [61]. This poses an issue, because at the time of recording, the current focal length was never measured. Through checking different values in the range, it appears the focal length was close to 4.0 mm. While this might not be exact, for this research it demonstrates that the distance scales correctly as the object is approached. Equation 1 is used to make this calculation. In this equation, W_K is the known width, L_f is the focal length of the lens, and W_P is the perceived width of the targeted object.

Equation 1 - Distance Formula

$$Distance = \frac{W_K * \ L_f}{W_P}$$

In this equation W_K is the width for each object in millimeters. These widths are found in the task descriptions for each course. The value of L_f used was 4.0 mm. W_P is the width in pixels of the object that is being calculated. This value is stored in the object’s class.

Through calculating the distance to the object, an estimation of the object’s position can be derived. This calculation is performed by correlating the size of the object, to the observed size of the object. This will give a result that has units of meters per pixel. Next, the pixel difference between the center of the frame and the center of the object is calculated. This allows for the X and Y position to be determined. The Z position is already known, which is distance.

While this calculation is close, there will be error induced due to the rotation of the object. This is because the object’s width is pre-programed, and does not account for off-axis. This could be fixed by adding a LIDAR or imaging sonar to the vehicle. By knowing the actual width of the object, position could more accurate be measured. However, if that were the case, position would already be known, rendering this calculation unnecessary. Figure 37 shows the inverted gate from the 2015 RoboSub competition. These calculated values about the gate are displayed above the object. Due to the rotation of the object, it will appear to be smaller, and therefore calculated to be further.

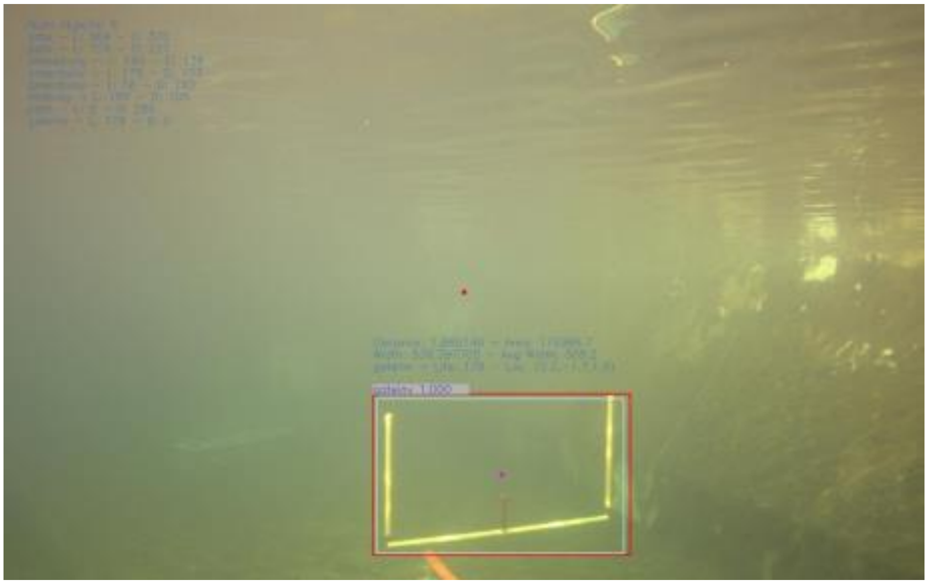


Figure 37 - Detected Inverted Gate

These displayed values are as follows:

- Distance – Distance in meters to the object
- Area – Area of the surrounding bounding box in pixels
- Width – Width of the surrounding bounding box in pixels
- Avg Width – Moving average of width
- ObjectID – Class name
- Life – Number of frames in which the object was detected
- Location – (x, y, z) position of the object.

The location of each object is given as Cartesian coordinates. These are in the format of (X, Y, Z). The axis convention used is shown in Figure 38. This system was decided upon, over traditional right handed coordinates, as it is more intuitive when viewing it as a forward-looking image.

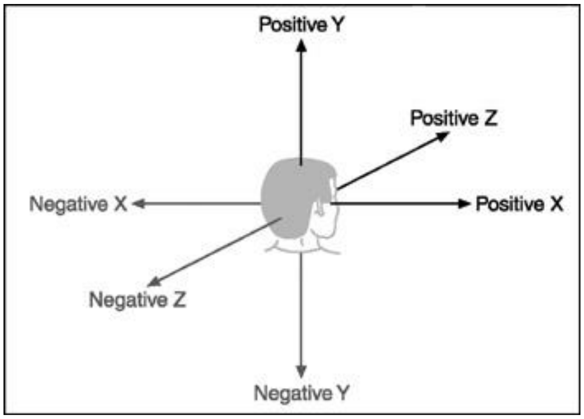


Figure 38 - Coordinate Frame Conventions used for position [62]

The display includes addition information about the detected objects. The red box around the detected object is the detected bounding box. This box is the result of Faster R-CNN's object detection layer. This bounding box, moves around constantly, and is not consistently the same size. To smooth this out, a moving average of both the bounding box's position, and the bounding box's size is calculated. This moving average uses a bin size of 5. This value can easily be adjusted to suite the vehicles movements. This averaging makes the object's position much more consistent, which is useful when creating a map from the data. This calculated average bounding box is shown as the light blue box surrounding the detected object. Shown above the bounding box, contrasted against a grey background, is the detected objects class name, as well as the confidence for the class. The top left corner of the image is used to list all the previously and currently detected objects. Next to each object name is two fields, 'L' and 'D'. 'L' is the number of frames in which that object was detected. 'D' is the number of frames since the object has last been detected. This is a useful metric to have, to clear out old object's that have past. The red dot in the image is the center of the image, while the purple dots are the center of the detected object's bounding box.

4.5. Results

The results of this project are very promising. This detector has proven to be capable of successfully detecting and classifying all the course elements for both the RobotX and RoboSub competition. In addition to detection, the objects are successfully tracked and stored in the class list. This results section will discuss the success of both competitions, as well as performance metrics.

4.5.1. Accuracy

For both data sets the detector algorithm proved to have an extremely high mean average precision (MAP). This precision is defined as the ratio of true positives to the sum of false positives and false negatives. A true positive is defined as classifying an object by the correct name. A false positive is defined as detecting an object that is not present. A false negative is defined as not detecting an object that is present. To obtain a MAP the detector was run on a testing set for each dataset. After the detector was run, the number of frames that each course element was correctly detected and the number of frames the element was either falsely classified or failed to be detected, after the object was in range, were counted. The detectors range was defined to be the distance in which the object is first detected. The accuracy was defined as the sum of false negatives and false positives divided by the number of true positives less than one. This equation is shown as Equation 2. Additionally, while the detector is capable of partial frame detection, it is not expected for the object to be detected if more than half of the object is out of frame. These images will be disregarded when calculating accuracy.

Equation 2 - Equation to Calculate Accuracy (MAP)

$$Accuracy = 1 - \frac{FN + FP}{TP}$$

Through testing and experimentation, it was found that a confidence threshold of 0.80 should be used for both datasets. This threshold value can be demonstrated with a frame consisting of 3 buoys and a path. Figure 39 shows the region proposals that were calculated for this image. This image was created by setting the confidence threshold to .0001. This effectively allowed for all the proposal regions to be shown. This image illustrates how the proposals are focused on objects, and disregard the background class.

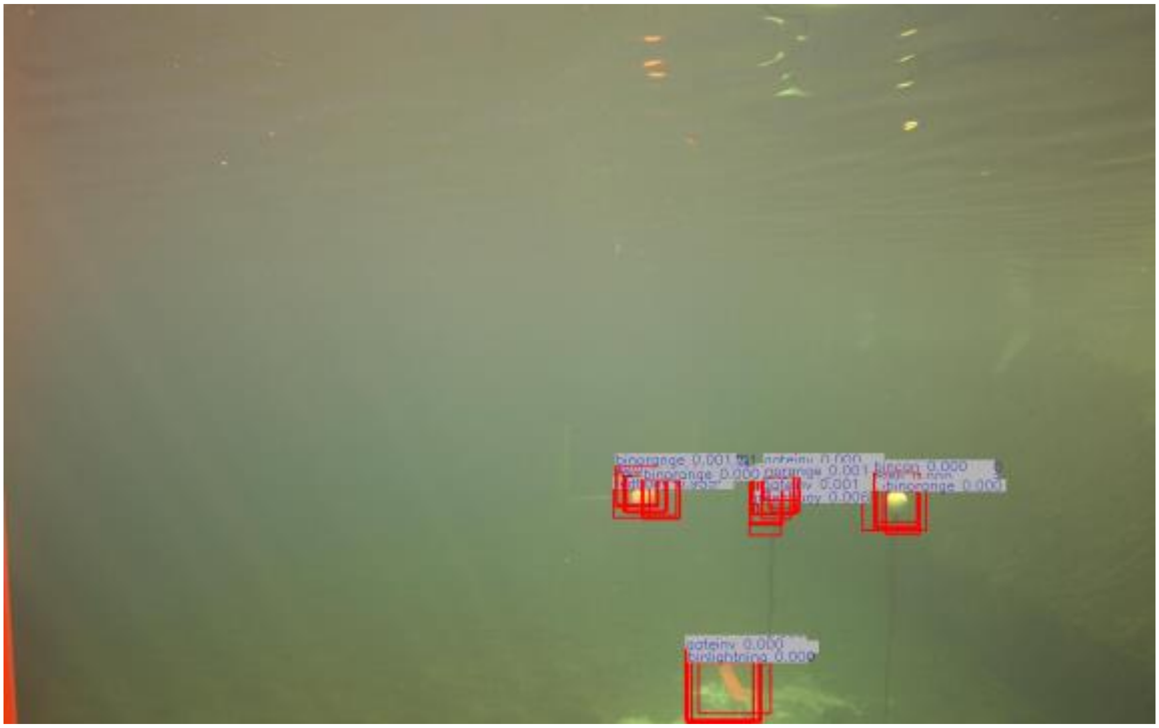


Figure 39 - Region Proposals

While this image shows that many overlapping proposals are calculated, it is difficult to see what the actual detected results are, as all the bounding boxes overlap. To make these results more apparent, the confidences for each object were logged, and then plotted. This plot is shown in Figure 40. In this example, 47 proposals were generated, these proposals are shown along the X-axis, while their confidence is on the Y-axis.

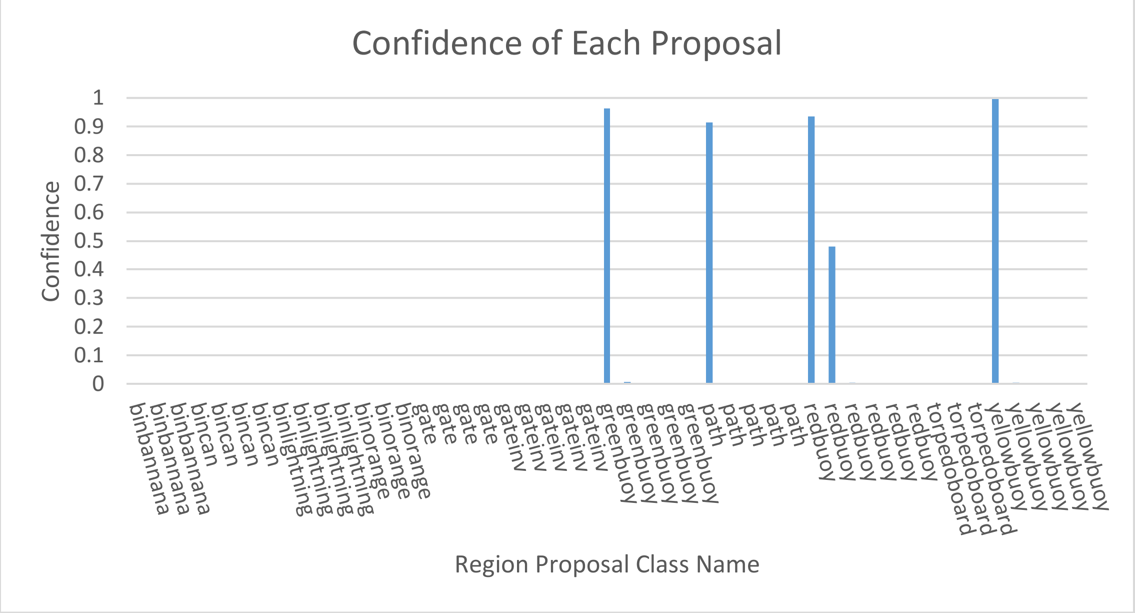


Figure 40 - Region Proposal Confidences

As seen in Figure 40, there are four objects with high confidences, and one object with a mid-range confidence, while the rest are negligible. The confidence threshold for this project was set to be 0.80. This value was determined to be most accurate when detecting all objects, and eliminating false positives. As there are only four objects in the frame, there is another object that could be a false positive with a confidence of 0.48. This object is a double detection of the red buoy. This however is not an issue as 0.48 is below the threshold of 0.80. This is due to the NMS threshold being too low for this case. NMS stands for Non-Max Suppression. NMS controls the possibility of an object being classified more than once. This parameter however is a tradeoff. If the value is too high, then it is a risk of losing precision of the bounding box. If the value is too low, then multiple true positives would occur [63]. Due to the confidence threshold, low confidence detection is not an issue. To see the results of the negligible proposals, another chart must be used with a different Y-axis range. This new plot is shown in Figure 41.

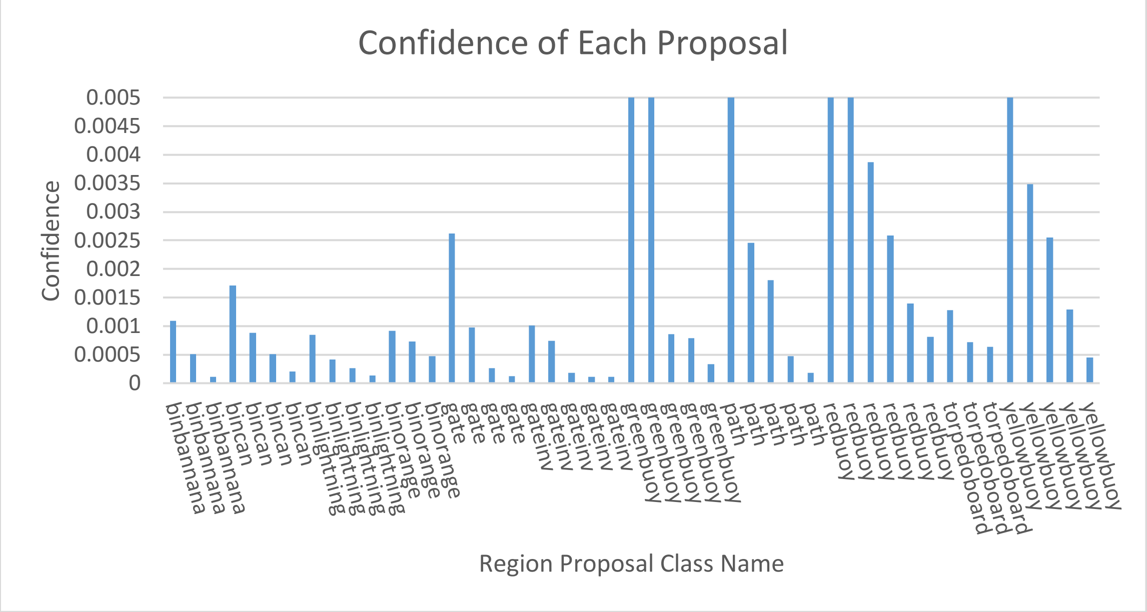


Figure 41 - Plot of confidences with reduced Y-Axis

This graph has a range of 0 to 0.005. This was done so that the values of the other proposals could be seen. These were not viewable on the initial graph due to their near zero values. This graph shows the other proposals, outside the correct ones, they have a minuscule confidence. The classified image from this experiment is shown as Figure 42. This experiment demonstrates the classifiers accuracy. The classifier has a low chance of classifying regions with the wrong class name.

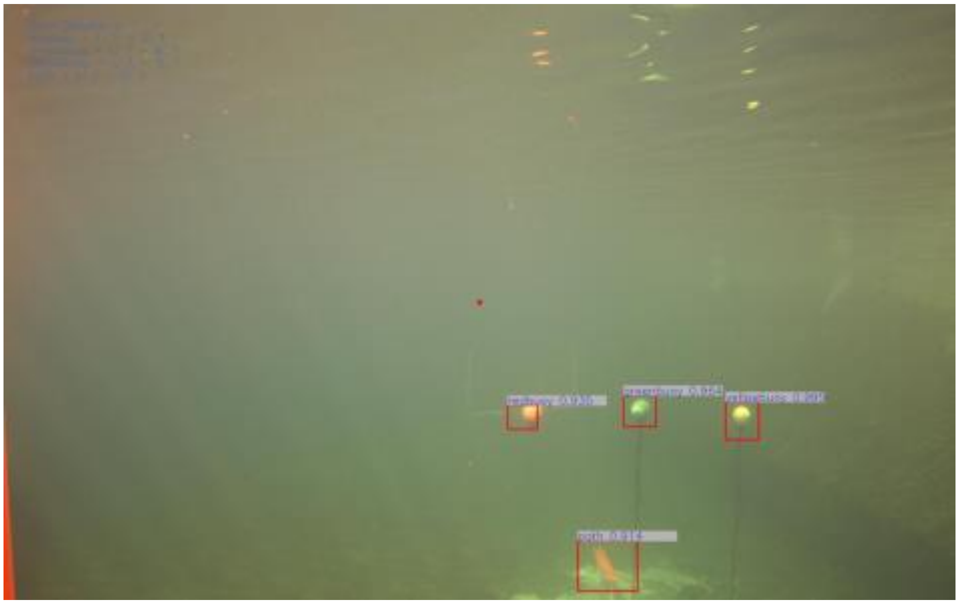


Figure 42 - Image demonstrating detection of the red buoy, green buoy, yellow buoy, and path

4.5.2. RoboSub Accuracy

From the RoboSub datasets 13 vision logs were deemed sufficient to be used as results. As these datasets were taken at competitions, the logs cannot be perfect. The quality of the logs depends on the ability for the vehicle to function, and correctly navigate the course. The logs deemed not sufficient, were ones in which the vehicle never left the starting dock. From these 13 logs, 26,879 images were processed through the detector. Processing at an average rate of 5.83 FPS, this detection took seventy-two minutes to perform. This processing was performed on a GTX 1080. More details on this timing performance will be discussed in section 4.5.4 Processing time. While the results of all these images were viewed, and deemed successful in classifying, they were not all used for the accuracy calculation. To fairly test the accuracy of the detector, two image sets were omitted from the training set. These two sets were selected as between both they contained all the classes. One of these sets was of a forward view, while the other was a downwards view.

Through analysis of this data, the detector proved to be successful and feasible for the RoboSub competition. Through extensive testing, it was proven that this detector can identify all 11 classes. Other classes were experimented and with and proved to be successful, but this will be discussed later in this section. As previously mentioned, the accuracy was calculated as the number of true positives divided by the sum of false negatives and false positives. This was shown in Equation 2. The results of this testing are shown in Table 10.

Table 10 - RoboSub Dataset accuracy

Class	**True Positives**	**False Negatives**	**False Positives**	**Accuracy**
gate_inv	300	0	0	100%
torpedo_board	231	0	0	100%
bin_orange	283	0	0	100%
gate	369	5	0	99%
bin_banana	468	7	0	99%
bin_lightning	628	12	0	98%
bin_can	469	13	0	97%
path	1193	65	1	95%
red_buoy	160	10	2	94%
yellow_buoy	193	24	1	88%
green_buoy	201	65	0	68%

As the data shows, the detector performs with a high accuracy. The average accuracy of all classes was calculated to be 94 percent. This accuracy is average of all classes. This is significantly higher than Faster R-CNN’s documented performance of 66.9 percent MAP using the VGG16 model on the VOC2007 dataset [34]. The trained detector had a very low rate of false positives. Of the 4,342 images between both image sets, there was only four false positives. This is a 0.0009 percent chance of a false positive occurring. One source of false positives was the detector incorrectly classifying air bubbles as red buoys. An image of one of these false detections is shown in Figure 43.

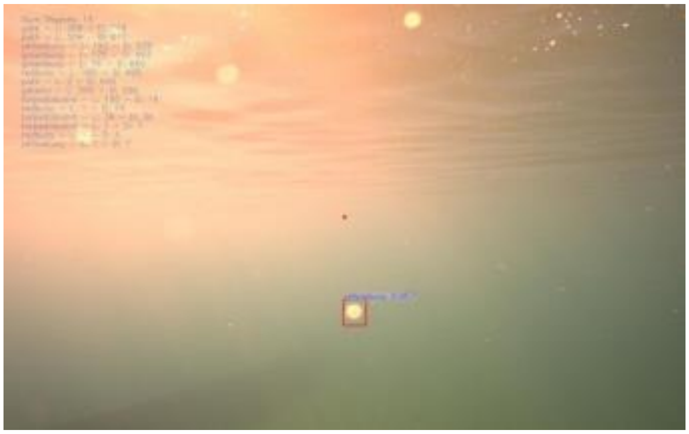


Figure 43 - False positive of air bubble as being classified as a red buoy

Figure 44 demonstrates the ability to train a detector on the various elements on the 2016 torpedo board. This experimented was omitted from the evaluation of the RoboSub training process. This shows that it is feasible to train a network on small intricate details with few images in the training set.

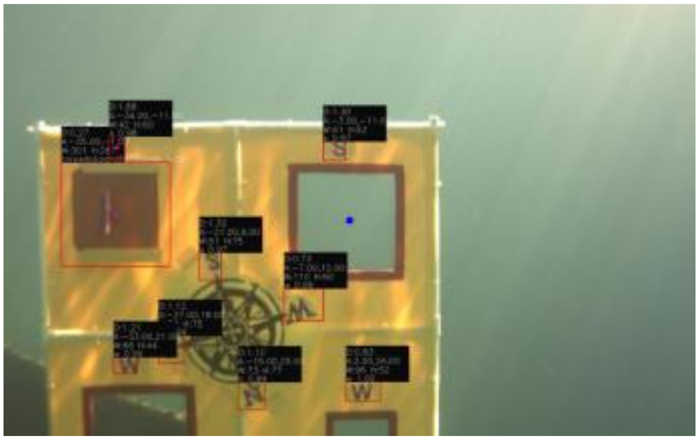


Figure 44 - 2016 Torpedo Board Detection

In addition to success with the detector correctly classifying each object, the detector was extremely accurate with detecting the object's size and location. Each object had a tight bounding box correctly placed around it. Figure 45 shows an example of all classes being detected. The path is shown being detected in both the forwards and downwards camera images. While the path can be, and often is, detected in the forward camera, the forward-facing path was not counted towards accuracy. This because the model was never trained on the path in the forward camera because it is not necessary, or even useful, to the competition. The imagery from the bottom facing camera was used for the path, and bin objects.

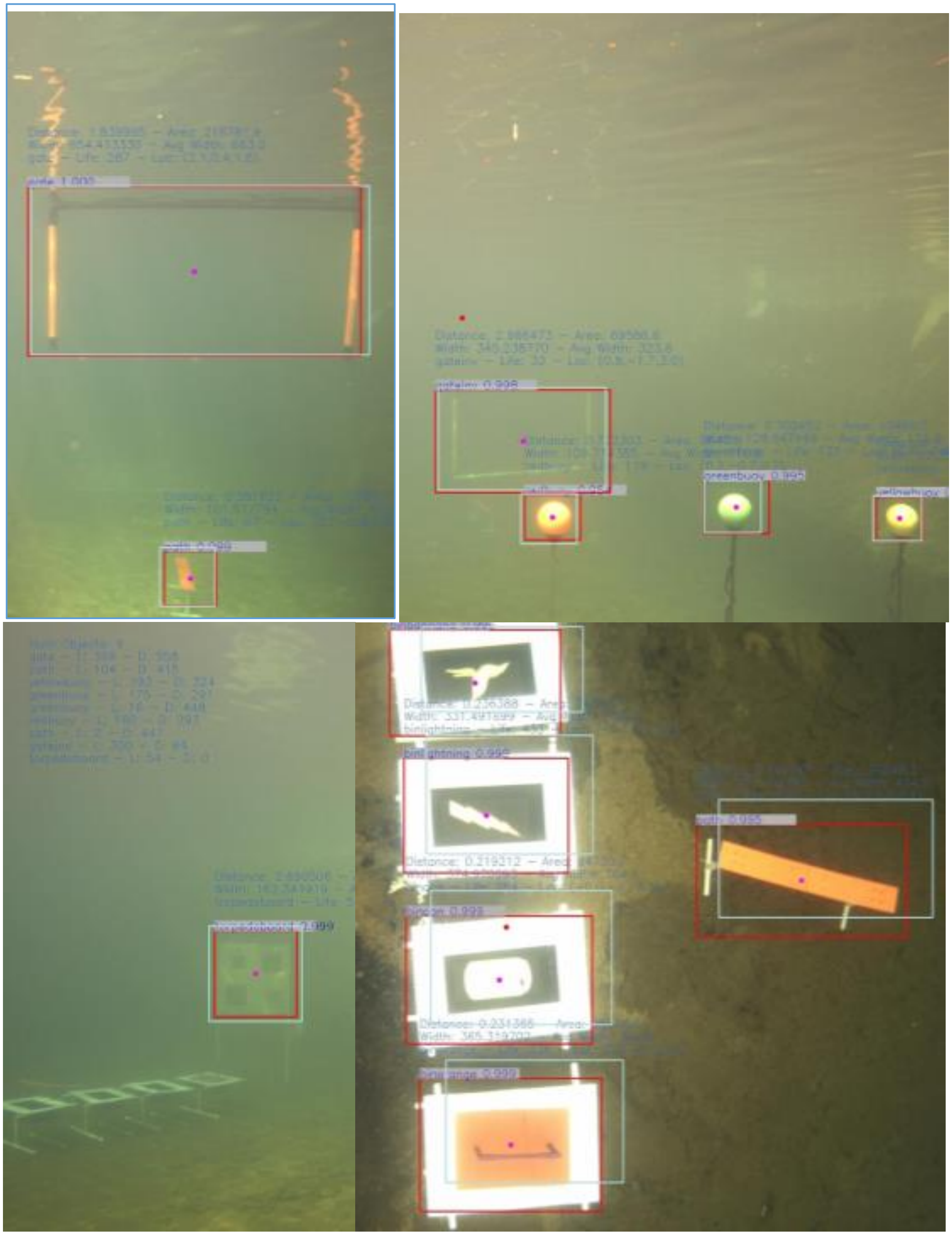


Figure 45 - Detection of All RoboSub Course Elements. Top Left: Gate, Path. Top Right: Red Buoy, Yellow Buoy, Green Buoy, Inverted Gate. Bottom Left: Torpedo Board. Bottom Right: Bin Banana, Bin Lightning, Bin Can, Bin Orange, Path.

Overall, this detector was extremely accurate when detecting these objects. Accuracy for some objects, such as the torpedo board identifiers, was high despite the lower number of annotations. Overall, there is no correlation between accuracy and number of annotations. This is possibly due to the relatively low sample set when testing. If more validation sets were performed with a wider range of lighting variation, this result may have been different. Unfortunately, there is no collected data to support this. Figure 46 shows a scatter plot that plots number of annotations for each class verses the percent accuracy for that class. There appears to be no trends demonstrated by this figure. It is hypothesized that this is because of the limited training set for this project. As the bins and torpedo boards had few annotations, they were over trained to be accurate to the only test case, which also had a similar appearance to the training set. This differs from the buoys, which had more annotations from different orientations and conditions. This figure was created by combining data from Table 6 and Table 10.

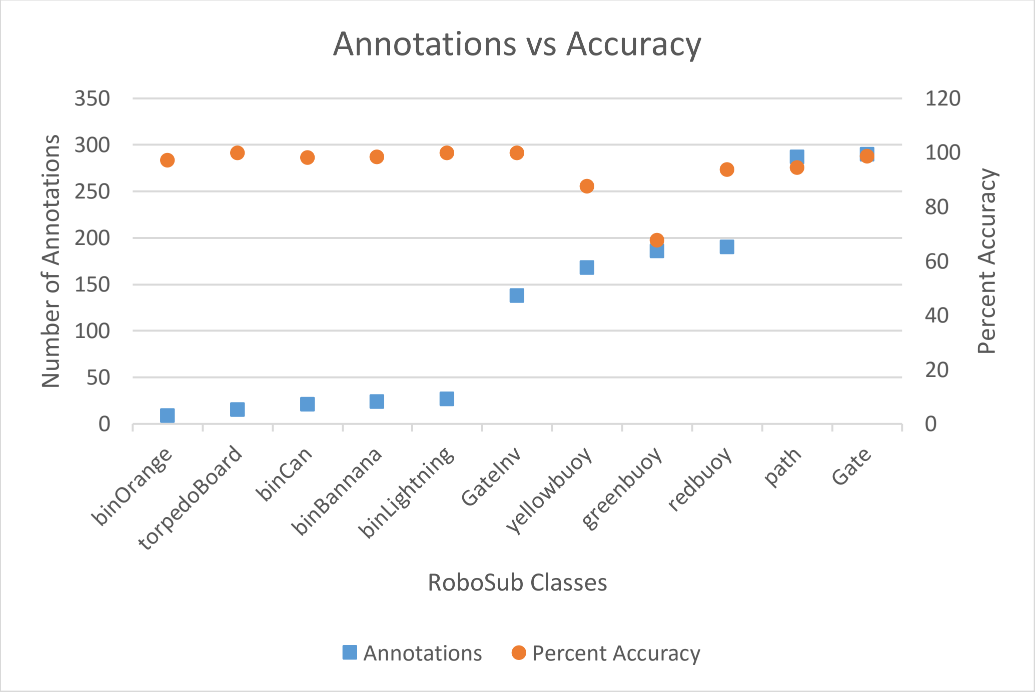


Figure 46 - Plot of Annotations vs Accuracy for the RoboSub Dataset

4.5.3. RobotX Accuracy

For RobotX Dataset, there is sixty-nine image sets that where processed through the detector algorithm. Within these image sets there are a total of 33,653 images. Processing of these images took ninety-seven minutes. Though the algorithm was tested on all these images, most them were in the training set. Despite training images being tested on, they were not used for any formal results. This was done to give subjective reasoning that the detector could work on more than just the one case that is the testing set. No recorded data was used from this testing. As validation on a training set does not give a fair evaluation, several images sets were reserved for testing purposes. These image sets were selected as they contain all the classified classes. 4,105 images were reserved from the testing set, and were only used for calculating the accuracy of the detector. These images were selected as they contained all the course elements. These images were all selected from separate runs than the training set. This was done to ensure there is enough difference in each image.

Through analysis of these validation sets, this detector proved to be extremely successful at detecting the various course elements. As with the RoboSub dataset, the number of true positives, false positives, and false negatives were counted. False negatives were counted after initial detection of the object. This is because the cameras have an incredibly long line of sight in open water. With more training data, the range could reliably increase, but this was not feasible due to the limited dataset. The results of these tests are shown in Table 11. Note, while the accuracy for some classes is 100%, it is not guaranteed there will be 100% accuracy for all test cases, this is simply for these test cases.

Table 11 - RobotX Dataset Accuracy

Class	True Positives	False Negatives	Accuracy
red_circle	142	0	100%
orange_ball	91	0	100%
green_cruciform	130	0	100%
blue_triangle	216	0	100%
red_triangle	410	0	100%
red_tower	40	0	100%
blue_circle	143	0	100%
white_buoy	816	8	99%
red_cruciform	404	4	99%
green_buoy	348	4	99%
green_triangle	259	3	99%
person	3217	60	98%
black_tower	491	10	98%
blue_cruciform	306	10	97%
black_buoy	143	4	97%
red_buoy	25	1	96%
yellow_tower	131	6	95%
green_tower	105	5	95%
green_circle	208	11	95%
blue_tower	63	7	89%
yellow_buoy	23	4	83%
black_ball	430	84	80%
blue_buoy	73	32	56%

Through validation of the detector it proved the detectors high accuracy. On average the detector had a 94% accuracy. This accuracy is coincidentally the same as the RoboSub accuracy. There however were no false positives, or incorrectly classified objects. Due to this, it is possible the confidence threshold could have been reduced slightly, which could have increased the range of the detector. For these test cases, there is a significantly lower accuracy for all the buoys. This is believed to be due to the image sets used, and not the training process. Initial tests before competition showed the detector having a significantly higher accuracy. For these tests, however, the buoys were much closer. During the RobotX competition the boat was rarely logging camera data on a course that implemented buoys. This resulted in most buoy images being at a long distance, and often from another course. While the detector had difficulties detecting the buoys at a long range, the shape signs excelled at long range. Figure 47 demonstrates the detectors ability to correctly detect the blue triangle from another course. Using Equation 1, this object was estimated to be 14 meters away. This is only an estimate though, as this equation relies on the camera’s focal length, which estimated.

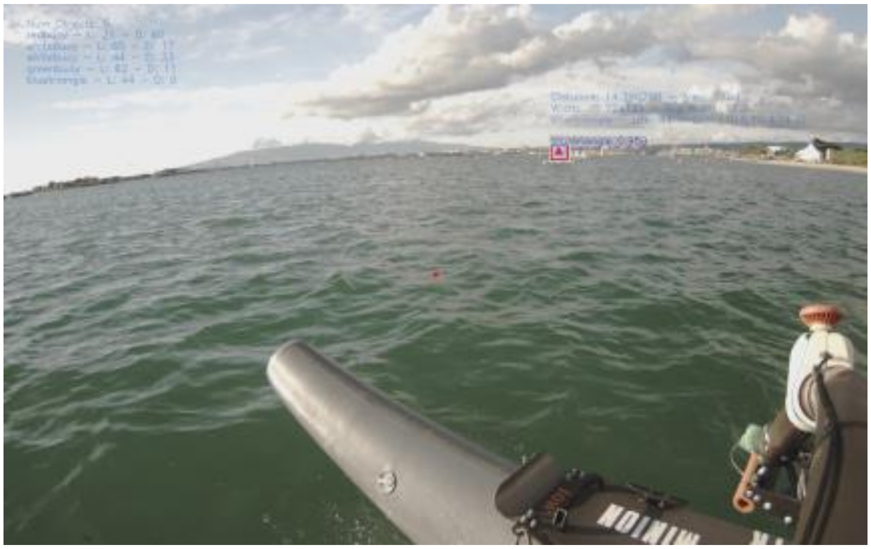


Figure 47 - Blue Triangle Detection

To demonstrate the detection of the Light Tower, images of the tower under various conditions were selected. The circle at the panel’s centroid was not drawn so the panel could be seen. While testing did not show the panels to have perfect accuracy, there was no errors if the object was within 3 meters. Within 3 meters the classifier correctly detected the panel in every frame. The classifier did an exceptional job at detecting and classifying the Light Tower’s panels for each color. The results in Figure 48 show the blue panel was the hardest to detect. The reasoning for is assumed to be because the only test set containing the blue panel was a difficult scenario. This validation set had the Light Tower positioned in front of the sun, as well as being from a further average distance than the other Light Tower test sets. These panels are a difficult object to detect, as they can appear to be completely different colors depending on the circumstances. If the object is in front of the Sun, the camera's exposure will wash out the image. This differs greatly from if the camera is looking away from the sun. Images of various Light Tower conditions are shown in Figure 48. There is an added difficulty to this classification due to the nature of LED panels. The bottom right image in Figure 43 shows the panel displaying green, which appears to be a striping of blue and yellow.

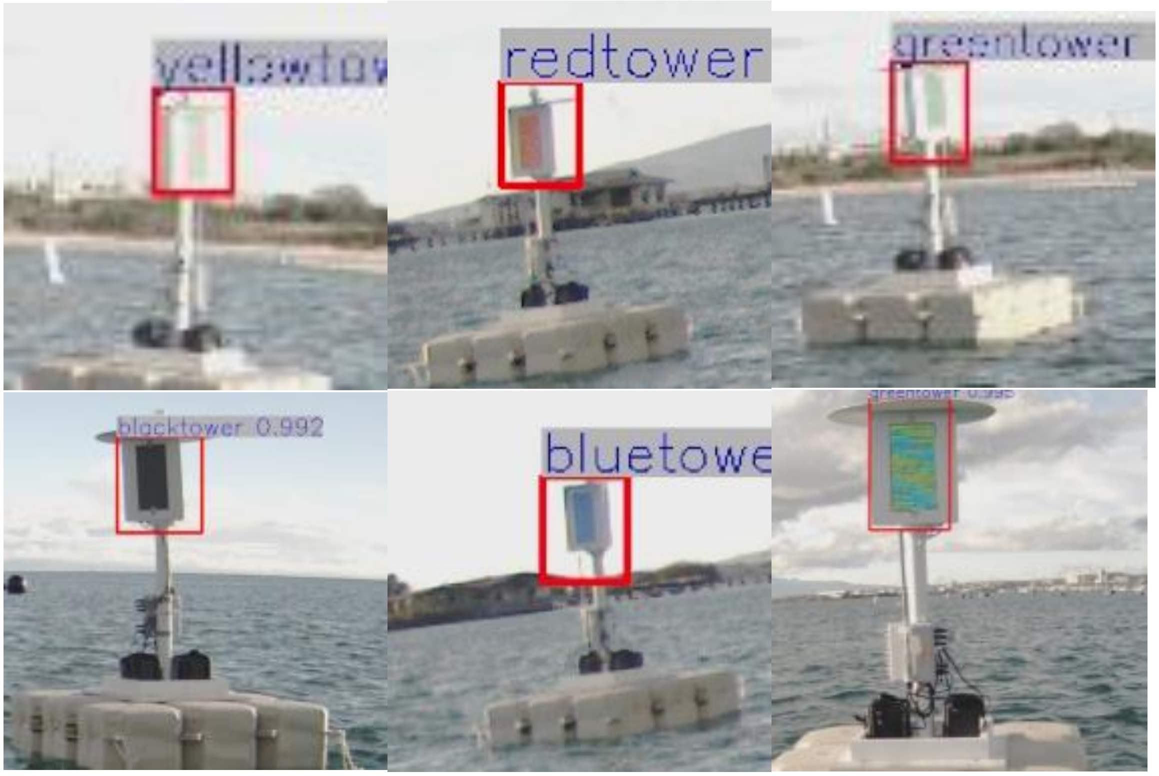


Figure 48 - Light Tower Detection for Yellow, Red, Green, Black, and Blue Panels

In addition to a high accuracy when detecting the light tower, this detector was also very accurate with identifying the dock symbols. Figure 49 and Figure 50 show the dock symbols being detected. The detector works both up near and at far distances.

Between these two images it can be shown how the detector can detect the triangle signs when they are rotated.

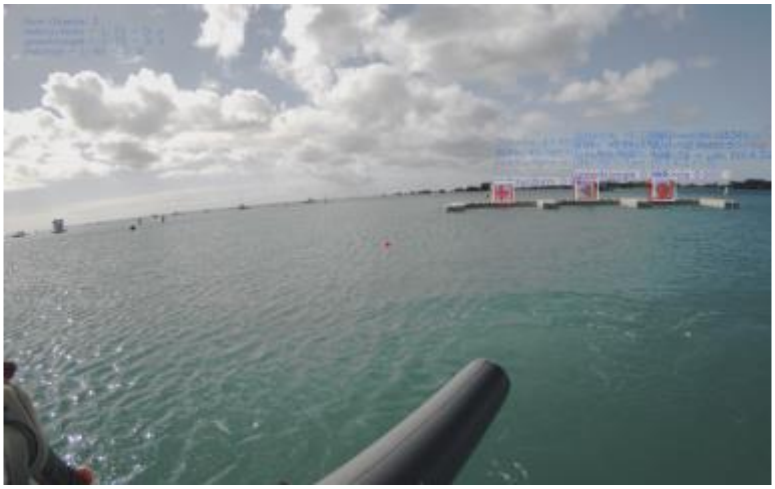


Figure 49 - Detected Dock Symbols. Red Cruciform, Green Triangle, and Red Circle

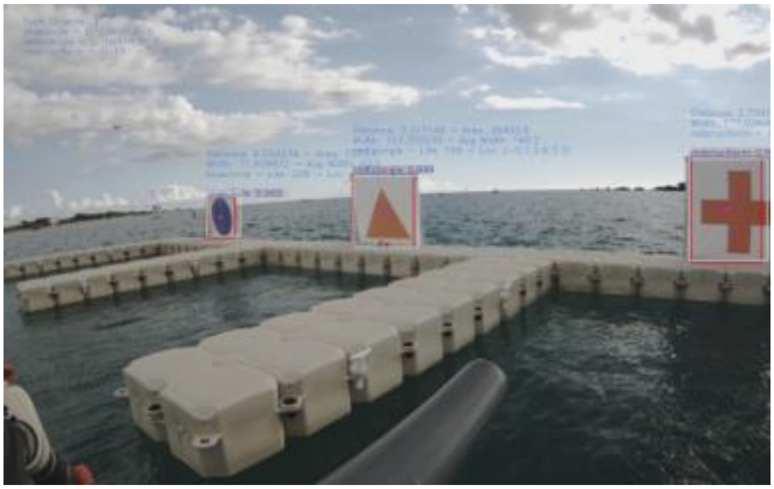


Figure 50 - Detected Dock Symbols. Blue Circle, Red Triangle, and Red Cruciform

For this detector to correctly identify the buoys, it was required that they be close to the vehicle. The detector appeared to have an effective range of 10 meters. Unfortunately, much of the images were further than this range. Figure 51 shows the detector attempting to detect black buoys and black balls.

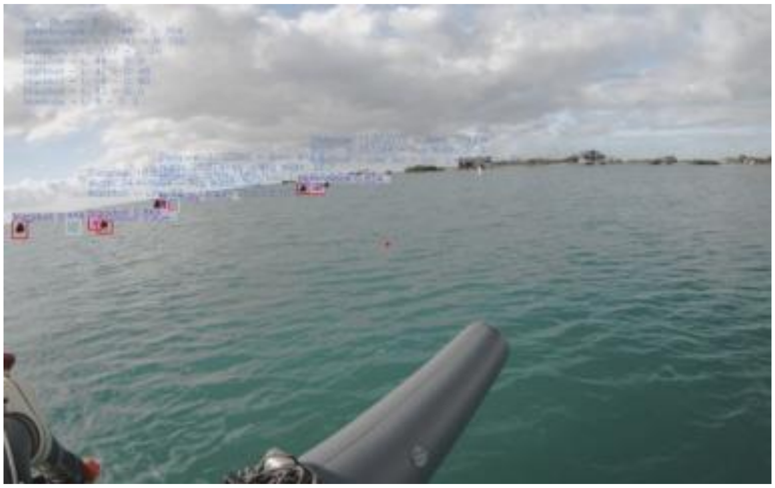


Figure 51 - Detection of Black Buoy and Black Balls

Over all, this detector performed very well for the RobotX dataset. This detector was successful in detecting the objects in varying weather and lighting conditions. Additionally, the detector averaged a 94% accuracy. This detector is even more accurate when it can be guaranteed the object will be within several meters of the boat.

4.5.4. Processing time

As with accuracy, processing time is also an important measure when evaluating the feasibility of using deep learning methods for object detection. When processing these images, the average time per frame was 0.174 seconds. This achieves a frame rate of 5.83 FPS. The processing time however depends on the system in which it is running on. To achieve an average processing time of 0.174 seconds, processing was performed on a NVIDIA GTX 1080. This is the same GPU as Minon has onboard. Minion proved that this is feasible to process neural networks while on a mobile platform. Initial testing was performed on a NVIDIA GTX 970. This card could process a frame in 0.28 seconds. This gives an average rate of 3.57 FPS. Depending on the application this is more than sufficient.

Additional testing was performed to evaluate the feasibility of running this process on a NVIDIA Jetson TX1. It was determined that it is possible to run this on a Jetson TX1, though it is not practical. The TX1 has 4GB shared RAM, which is sufficient, as Faster R-CNN only needs 3.5GB of ram to operate. However, due to memory leaks, and other inefficiencies the program can only run once, and then the Jetson must be restarted to completely purge the RAM. Processing took on average 1.8 seconds per frame. This is too slow to use for near real-time processing. Testing was also attempted on a Jetson TK1, though due to a limitation of 2GB RAM, it was highly unsuccessful. Due to its small form factor, and power requirements, the TX1 can be a powerful processing platform. A processing time of 1.8 seconds per frame is slow compared to the performance of a GTX 1080. However, processing on the TX1 is still much more efficient than processing on the CPU. The TX1 could achieve twice the frame rate, while maintaining a significantly lower thermal dissipation. CPU testing was performed on an Intel Core i7-4790K Processor, which has a thermal dissipation of 88 watts [64]. As previously mentioned, when processing only on a CPU, this method has an average time per frame of 3.242 seconds. Therefore, a TX1 is twice as fast, as well as having five times less thermal dissipation, than operating on a CPU. In this scenario, a TX1 is a sensible substitution.

5. Conclusions and Future Recommendations

ASVs and AUVs can be used to perform an important role in the maritime environment. One important capability of these vessels, is the ability to perform autonomous object detection and classification. There has been considerable research in performing detection using LIDAR, Radar, and imaging sonar. However, these methods perform poorly when attempting to detect color, or shapes on flat surfaces. In order to detect objects in these circumstances a camera is desirable. Unfortunately, there has not been much successful research conducted in maritime object detection or classification.

To create a detector for maritime usage, it was decided to pursue a method implementing a neural network. Through research it was decided to implement Faster R-CNN as the framework for this research. This open sourced project was built upon to add in persistent object tracking as well as position estimates. Faster R-CNN was capable of successfully training a model to detect and classify all the course elements in both the RoboSub and RobotX competitions.

This project resulted in the creation of a highly effective object detector and tracker. Using Faster R-CNN a detector could be trained on both the RoboSub and RobotX datasets. For both datasets, a mean average precision of 94% was achieved. This provides a strong backing for an object tracker. To track objects, it is often necessary to detect and classify them first. This project demonstrated it is feasible to use deep learning to both detect maritime objects, and run on mobile maritime platforms.

If this project were to be conducted again, there will be several factors that are recommended to change. If I were to do this project again, I would reconsider the use of YOLO Version 2. During the phase of researching plausible methods, YOLO was still released as version 1. At that time YOLO did not appear to be as credible, or as promising of a method. However, in November 2016, after much research and work was already conducted on this project, YOLO Version 2 was released. This update offered significant performance and accuracy increases. This new version advertises 40-90 FPS and a 78.6% MAP on VOC 2007 [32]. When testing YOLO V2 using the Tiny YOLO model, 13 FPS could be achieved on a Jetson TX1, while 250 FPS could be achieved on a GTX 1080. With the release of version 2, YOLO is both faster and more accurate than Faster R-CNN. Obtaining 13 FPS would be more than feasible for implementing a system using a Jetson TX1. Unfortunately, YOLO V2 was released too late into the research process to be feasible of switching methods.

Another recommendation to future projects would be to use updated hardware. During the length of this project NVIDIA announced both a new Jetson model, as well as releasing the GTX 1080 Ti. These two upgrades are a significant improvement over their predecessor models. The GTX 1080 Ti has approximately 40% more memory, and 30% faster performance [66]. With these increases, it could be presumed that Faster R-CNN would operate at over 8 FPS. Additionally, other models that require 12 GB VRAM could be used. NVIDIA claims that the Jetson TX2 is twice as fast as the TX1. With this performance increase it can be presumed that YOLO could operate at 25 FPS. This would make this a perfect solution, that would be feasible for object detection [67]. Additionally, the TX2 will have 8 GB of RAM, instead of 4 GB. This would allow for the capability to load most networks, with no issues.

Future work on this project could investigate methods of reducing the processing time per image. This processing time could be reduced by altering the layers of the model used. The VGG-16 layer used has 41 layers, 16 of which have learnable weights, 14 are convolutional and 3 are fully connected layers [65]. As most the computational time for each frame is spent processing each of these layers, the model could be speed up if some layers were removed or altered. This however could have an impact on the accuracy, though it presumed that not all layers are necessary for a model this small.

To build upon this project, the code has been made open sourced, and can be found online. The code for this project has been made publicly available at https://github.com/rtgoring/py-faster-rcnn-thesis. This repository is public, and can be accessed by anybody. This project will remain as it for archival purposes, though a fork will be created in this repository for all future work. The models trained for this research have also been made available. Note, due to GitHub's 100 MB file size limit, these models have been uploaded to Google Drive. A document containing a link to these files was created in the 'Output' folder.

Works Cited

[1] "Utah I (Battleship No. 31)," 18 Febuary 2016. [Online]. Available: https://www.history.navy.mil/research/histories/ship-histories/danfs/u/utah.html.

[2] AUTONOMOUS SURFACE VEHICLES LTD, "Unmanned Marine Systems," [Online]. Available: http://www.unmannedsystemstechnology.com/company/autonomous-surface-vehicles-ltd/.

[3] SAS, "Machine Learning What it is and why it matters," [Online]. Available: https://www.sas.com/en_us/insights/analytics/machine-learning.html#machine-learning-today-world.

[4] R. Parloff, "fortune," 28 September 2016. [Online]. Available: http://fortune.com/ai-artificial-intelligence-deep-machine-learning/.

[5] K. He , Z. Xiangyu , S. Ren and J. Sun, "Deep Residual Learning for Image Recognition".

[6] ufldl Stanford, "Multi-Layer Neural Network," [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/.

[7] L. A. d. Santos, "Introduction," [Online]. Available: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/neural_networks.html.

[8] "Deep Learning Use Cases," [Online]. Available: https://deeplearning4j.org/use_cases.

[9] S. Kiran, "What is the algorithm used by Google's reverse image search (i.e. search by image)?," 9 August 2014. [Online]. Available: https://www.quora.com/Algorithms/What-is-the-algorithm-used-by-Googles-reverse-image-search-i-e-search-by-image.

[10] A. Abdulkader, A. Lakshmiratan and J. Zhang, "Facebook," 1 June 2016. [Online]. Available: https://code.facebook.com/posts/181565595577955/introducing-deeptext-facebook-s-text-understanding-engine/.

[11] D. Kislyuk, "medium," 27 June 2016. [Online]. Available: https://medium.com/@Pinterest_Engineering/introducing-automatic-object-detection-to-visual-search-e57c29191c30#.mt7b1seyj.

[12] Uijlings, "Selective Search for Object Recognition," [Online]. Available: http://vision.stanford.edu/teaching/cs231b_spring1415/slides/ssearch_schuyler.pdf.

[13] techpowerup, "NVIDIA Quadro Plex 7000," 2011. [Online]. Available: https://www.techpowerup.com/gpudb/902/quadro-plex-7000.

[14] "Datasets," 14 June 2014. [Online]. Available: http://deeplearning.net/datasets/.

[15] Stanford Vision Lab, Stanford University, Princeton University , 2016. [Online]. Available: http://www.image-net.org/.

[16] "The PASCAL Visual Object Classes Challenge 2012," [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/.

[17] "NVIDIA Jetson," March 2017. [Online]. Available: http://www.nvidia.com/object/jetson-tx1-module.html.

[18] WAM-V Marine Advanced Research, "WAM-V," [Online]. Available: http://www.wam-v.com/16-wam-v-usv/.

[19] Association for Unmanned Vehicle Systems International Foundation, "2016 Maritime RobotX Challenge Task Descriptions," 18 November 2016. [Online]. Available: http://robotx.org/images/files/2016-MRC-Tasks-2016-11-28.pdf.

[20] AUVSI Foundation, "Robosub," 2017. [Online]. Available: http://www.robonation.org/competition/robosub.

[21] Association for Unmanned Vehicle Systems International Foundation, "RoboSub," 2015. [Online]. Available: http://higherlogicdownload.s3.amazonaws.com/AUVSI/fb9a8da0-2ac8-42d1-a11e-d58c1e158347/UploadedFiles/RoboSub%20Competition%20Official%20Rules%20and%20Mission%20-%202015.pdf.

[22] R. Szeliski, Computer Vision: Algorithms and Applications, Springer London, 2010.

[23] T.-H. Tran and T.-L. Le , "Vision based boat detection for maritime surveillance," IEEE, Hanoi, 2016.

[24] D. Socek, D. Culibrk, O. Marques, H. Kalva and B. Furht, "A Hybrid Color-Based Foreground Object Detection Method for Automated Marine Surveillance," Florida Atlantic University, Boca Raton.

[25] T. Cane and J. Ferryman, "Saliency-based Detection for Maritime Object Tracking," 2016.

[26] M. Chua, D. W. Aha, B. Auslander, K. Gupta and B. Morris, "Comparison of Object Detection Algorithms on Maritime Vessels," 2013.

[27] Z. Zhu, D. Liang, S. Zhang, X. Huang, B. Li and S. Hu, "Traffic-Sign Detection and Classification in the Wild," CVPR, Beijing, China; Bethlehem, USA, 2016.

[28] R. Girshick, "Fast R-CNN Object detection with Caffe," [Online]. Available: http://tutorial.caffe.berkeleyvision.org/caffe-cvpr15-detection.pdf.

[29] rbgirshick, "py-faster-rcnn," 8 March 2016. [Online]. Available: https://github.com/rbgirshick/py-faster-rcnn.

[30] R. Stewart and M. Andriluka, "End-to-end people detection in crowded scenes," 2015.

[31] A. Tao, J. Barker and S. Sarathy, "DetectNet: Deep Neural Network for Object Detection in DIGITS," 11 August 2016. [Online]. Available: https://devblogs.nvidia.com/parallelforall/detectnet-deep-neural-network-object-detection-digits/.

[32] J. Redmon, "Darknet: Open Source Neural Networks in C," *http://pjreddie.com/darknet/,* 2013-2016.

[33] R. Girshick, "Fast R-CNN," *girshickICCV15fastrcnn,* 2015.

[34] R. Shikler and G. Elbaz, "webcourse," 26 May 2016. [Online]. Available: https://webcourse.cs.technion.ac.il/236815/Spring2016/ho/WCFiles/RCNN_X3_6pp.pdf.

[35] D. Wilding-McBride, "the cleverness of deep learning," 12 6 2016. [Online]. Available: http://dius.com.au/2016/12/06/the-cleverness-of-deep-learning/.

[36] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick , S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093,* 2014.

[37] Russell91, "TensorBox," 15 March 2017. [Online]. Available: https://github.com/TensorBox/TensorBox.

[38] Tensor Flow, "Image Recognition," 8 March 2017. [Online]. Available: https://www.tensorflow.org/tutorials/image_recognition.

[39] TensorFlow, "TensorFlow White Papers," 8 March 2017. [Online]. Available: https://www.tensorflow.org/about/bib.

[40] TensorFlow, "Using GPUs," 8 March 2017. [Online]. Available: https://www.tensorflow.org/tutorials/using_gpu.

[41] TensorFlow, "Installing TensorFlow," 9 March 2017. [Online]. Available: https://www.tensorflow.org/install/.

[42] g. j. lukeyeager, "Using DIGITS to train an Object Detection network," 17 January 2017. [Online]. Available: https://github.com/NVIDIA/DIGITS/tree/master/examples/object-detection .

[43] NVIDIA, "NVIDIA® DIGITS™ Downloads," 1 Febuary 2017. [Online]. Available: https://developer.nvidia.com/rdp/digits-download.

[44] lukeyeager, "GitHub," 29 November 2016. [Online]. Available: https://github.com/NVIDIA/DIGITS/releases.

[45] J. Redmon, "YOLO: Real-Time Object Detection," November 2016. [Online]. Available: https://pjreddie.com/darknet/yolov1/.

[46] nvidia, "NVIDIA TITAN X," 2016. [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[47] PointGrey, "Blackfly 2.3 MP Color GigE PoE (Sony IMX136)," 2016. [Online]. Available: https://www.ptgrey.com/blackfly-23-mp-color-gige-vision-poe-sony-imx136-camera .

[48] M. Harris, "CUDA 8 Features Revealed," 5 April 2015. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/.

[49] NVIDA, "NVIDIA cuDNN," 20 January 2017. [Online]. Available: https://developer.nvidia.com/cudnn.

[50] NVIDIA, "cuDNN Download," 20 January 2017. [Online]. Available: https://developer.nvidia.com/rdp/cudnn-download.

[51] TheTesla, "GitHub," 15 December 2016. [Online]. Available: https://github.com/TheTesla/caffe-fast-rcnn.

[52] NVIDIA, "Overview | What's New," 2017. [Online]. Available: https://developer.nvidia.com/cudnn-whatsnew.

[53] FLIR, "FlyCapture SDK," [Online]. Available: https://www.ptgrey.com/flycapture-sdk.

[54] NVIDIA, "CUDA GPUs," March 2017. [Online]. Available: https://developer.nvidia.com/cuda-gpus.

[55] NVIDIA, "CUDA C Programming Guide," 12 January 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-5-x.

[56] ShaoqingRen, "GitHub," 26 June 2015. [Online]. Available: https://github.com/ShaoqingRen/caffe/tree/SPP_net.

[57] MrGF, "GitHub," 11 July 2016. [Online].

[58] tzutalin, "GitHub," 9 March 2017. [Online]. Available: https://github.com/tzutalin/labelImg/.

[59] D. Curro, "Borrowing Weights from a Pretrained Network," 23 Febuary 2016. [Online]. Available: https://github.com/BVLC/caffe/wiki/Borrowing-Weights-from-a-Pretrained-Network.

[60] A. Rosebrock, "Find distance from camera to object/marker using Python and OpenCV," 15 January 2015. [Online]. Available: http://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/.

[61] bhphoto, "inon CS-Mount 2.2-6mm Varifocal Lens," [Online]. Available: https://www.bhphotovideo.com/c/product/970556-REG/fujinon_yv2_7x2_2sr4a_sa2_cs_mount_2_2_to.html.

[62] F. K. C., "flylib," [Online]. Available: http://flylib.com/books/en/2.416.1.16/1/.

[63] J. Hosang, R. Benenson and B. Schiele, "A Convnet for Non-Maximum Suppression," 2016.

[64] Intel, "Intel® Core™ i7-4790K Processor," Intel, 2014. [Online]. Available: http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz.

[65] MATLAB, "vgg16," 2017. [Online]. Available: https://www.mathworks.com/help/nnet/ref/vgg16.html.

[66] GPU Boss, "GPU Boss," March 2017. [Online]. Available: http://gpuboss.com/gpus/GeForce-GTX-1080-Ti-vs-GeForce-GTX-1080.

[67] NVIDIA, "NVIDIA Jetson TX2 Module," March 2017. [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-tx2.

[68] AUVSI Association, "MEET THE 2016 TEAMS," 2016. [Online]. Available: http://robotx.org/index.php/teams/2016-teams.