A DEEP LEARNING APPROACH TO TARGET RECOGNITION IN SIDE-SCAN SONAR IMAGERY

by

Dylan Einsidler

A Thesis Submitted to the Faculty of
The College of Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Master of Science

Florida Atlantic University
Boca Raton, FL
May 2018
A DEEP LEARNING APPROACH TO TARGET RECOGNITION IN SIDE-SCAN SONAR IMAGERY

by

Dylan Einsidler

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Manhar Dhanak, Department of Ocean and Mechanical Engineering, and has been approved by the members of his supervisory committee. It was submitted of the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

Manhar Dhanak, Ph.D.
Thesis Advisor

Pierre-Philippe Beauljean, Ph.D.

Oscar Curet, Ph.D.

Javad Hashemi, Ph.D.
Chair, Department of Ocean and Mechanical Engineering

Stella Batalama, Ph.D.
Dean, College of Engineering and Computer Science

Khaled Sobhan, Ph.D.
Interim Dean, Graduate College

April 23, 2018
ACKNOWLEDGEMENTS

This work was supported as a Naval Engineering Education Center project though NAVSEA and Naval Surface Warfare Center, Panama City under grant N00174-16-C-0017. The research conducted in this thesis was made possible because of the opportunities provided by Florida Atlantic University and the U.S. Navy. The combined direction and constructive criticism provided by my advisors have been the driving reasons why the research in this thesis progressed.

I would like to acknowledge the use of the services provided by Research Computing at the Florida Atlantic University. My appreciation of the assistance of FAU’s High-Powered Computing administrators Eric Borenstein and Brian Brown cannot be expressed enough. I would like to extend this gratitude to my advisory committee for approving the thesis proposal: Dr. Manhar Dhanak, Dr. Pierre-Philippe Beaujean, and Dr. Oscar Curet. All three of these professors have been of tremendous help over the years regarding various projects, and I could not be more grateful for the time and advice invested by them in my education. I am truly thankful for the opportunity to have studied ocean engineering at FAU and the opportunity to have been guided by these mentors in producing this thesis.
ABSTRACT

Author: Dylan Einsidler
Title: A Deep Learning Approach To Target Recognition In Side-Scan Sonar Imagery
Institution: Florida Atlantic University
Thesis Advisor: Dr. Manhar Dhanak
Degree: Master of Science
Year: 2018

Automatic target recognition capabilities in autonomous underwater vehicles has been a daunting task, largely due to the noisy nature of sonar imagery and due to the lack of publicly available sonar data. Machine learning techniques have made great strides in tackling this feat, although not much research has been done regarding deep learning techniques for side-scan sonar imagery. Here, a state-of-the-art deep learning object detection method is adapted for side-scan sonar imagery, with results supporting a simple yet robust method to detect objects/anomalies along the seafloor. A systematic procedure was employed in transfer learning a pre-trained convolutional neural network in order to learn the pixel-intensity based features of seafloor anomalies in sonar images. Using this process, newly trained convolutional neural network models were produced using relatively small training datasets and tested to show reasonably accurate anomaly detection and classification with little to no false alarms.
LIST OF FIGURES

Figure 1: How an AUV “sees” an object along the seafloor using side-scan sonar imagery. [1] ........................................................................................................ 9

Figure 2: The method of the “Histograms of Oriented Gradients” (HOG) algorithm. (Creation of pixel brightness intensity gradients to detect faces). [19]............. 12

Figure 3: Common architecture of a traditional machine learning algorithm for ATR in sonar imagery. [1] ........................................................................................ 13

Figure 4: General architecture of a deep neural network. [4] ........................................... 16

Figure 5: An overview of the common CNN algorithm design. [4] .................................. 19

Figure 6: An overview of the R-CNN algorithm. [18] ..................................................... 21

Figure 7: The general process of the YOLO algorithm. In this example, since the confidence scores of the dog, bicycle, and car are all above 30%, they are kept for the final outcome. [20] ........................................................................ 23

Figure 8: A visualization of the original architecture of the YOLO convolutional neural network. [20] ................................................................................................. 24

Figure 9: The YOLOv2 layer configurations .................................................................... 25

Figure 10: Visualization of image feature regeneration due to different loss functions as training progresses. [42] .............................................................................. 27

Figure 11: Examples of image categories used in the CNN "AlexNet". [4] ....................... 31

Figure 12: Transfer-learned CNN “AlexNet” able to correctly identify new (MathWorks) office objects. ......................................................................................... 32
Figure 13: Stop sign detection (and confidence %) after reconfiguring the CifarNet CNN via TL. [13] .......................................................... 33

Figure 14: Target recognition (middle image) and classification (right image) due to transfer learning pretrained CNN models onto sonar image data in [15] ....... 36

Figure 15: A comparison of mean precision and recall values from different CNN models in the experimental trials done in [15]........................................ 36

Figure 16: An example of results from [39], where the YOLO model was repurposed to detect and count healthy scallops by a downward-pointing digital camera attached to the nose of an AUV. [39] ........................................ 38

Figure 17: The joint ROV/AUV system described in the work of [40] ......................... 38

Figure 18: An example of results from [40], where detection and tracking of the ROV is achieved with YOLO .................................................... 39

Figure 19: The Hydroid REMUS-100 with all of its built-in features and accessories. [30] .......................................................................................... 41

Figure 20: Approximate altitude and average travel speed of the REMUS-100 for all missions................................................................. 42

Figure 21: The Sea Scan Survey graphical user interface (GUI) ..................................... 43

Figure 22: The draw_box.py GUI for annotating the training images. ......................... 46

Figure 23: Line 114 and line 120 in the tiny-yolo-1c.cfg file .................................... 48

Figure 24: YOLO training parameters and descriptions that were kept consistent for all trials (except total batch size). ........................................ 50

Figure 25: Output of the loaded model at the start of the training process .................. 51

Figure 26: Output for the loading of the .xml files needed for training ....................... 52
Figure 27: The output of the beginning of the training process ........................................ 52
Figure 28: The output of the end of the training process ............................................. 53
Figure 29: The configurations for testing a newly trained model in the python script "processing_video.py" ................................................................. 54
Figure 30: Beginning of the testing process (loading of the newly created model, loading of the testing video, and outputting the frames-per-second during processing) .................................................................................................. 55
Figure 31: Line 40 of python script "get_images.py", enabling extraction and automatic downloading of google image search results ........................................... 56
Figure 32: Line 7 of the python script "rename.py" .......................................................... 57
Figure 33: A portion of the training image dataset for trial 1 ........................................ 58
Figure 34: Anomaly detection made at 64% confidence after 2700 steps of training. (Trial 1) ........................................................................................................... 59
Figure 35: Anomaly detection made at 77% confidence after 3500 steps of training. (Trial 1) ........................................................................................................... 59
Figure 36: Anomaly detection made at 87% confidence after 4500 steps of training. (Trial 1) ........................................................................................................... 60
Figure 37: Anomaly detection made at 95% confidence (and a highest core for the trial) after 5400 steps of training (full 500 epochs). (Trial 1) ................. 60
Figure 38: Multiple anomalies found simultaneously at confidences well above 50% (full 500 epochs). (Trial 1) .......................................................... 61
Figure 39: In a clutter of potential targets (or “area of pattern”), the target that casts
the largest shadow (or the most drastic change in features) will dominate
(full 500 epochs). (Trial 1) ................................................................. 62

Figure 40: One anomaly detected, while others not. Likely due to limited training
data (full 500 epochs). (Trial 1) ............................................................. 63

Figure 41: Multiple anomalies made, but some anomalies ignored. Likely due to
limited training data (full 500 epochs). (Trial 1) ................................. 63

Figure 42: An example of a false alarm detection made (right) at 61%. (Trial 1) ....... 64

Figure 43: A portion of the training image dataset for trial 2 ............................... 65

Figure 44: An example of a valuable training image with an anomaly (on the right)
for trial 2. ............................................................................................... 66

Figure 45: Anomaly detection made at 80% confidence after 500 steps of training. 1
out of 2 anomalies detected. (Trial 2) ...................................................... 67

Figure 46: Anomaly detection made at 94% confidence after 1000 steps of training. 1
out of 2 anomalies detected. (Trial 2) ...................................................... 67

Figure 47: Both anomalies detected at 74% confidence (left) and 83% confidence
(right) after 1500 steps of training. (Trial 2) ........................................... 68

Figure 48: Both anomalies detected at 93% confidence (left) and 86% confidence
(right) after 2000 steps of training. (Trial 2) ........................................... 68

Figure 49: Both anomalies detected at 98% confidence (left) and 91% confidence
(right) after 2500 steps (full 500 epochs) of training. (Trial 2) ................. 69

Figure 50: Highest anomaly detection confidence score recorded at 99% (left) after
500 epochs of training. (Trial 2) .............................................................. 69
Figure 51: 2 out of 3 anomalies detected (upper-left corner anomaly neglected). (Trial 2) ........................................................................................................... 70

Figure 52: 3 out of 3 anomalies detected moments later, but at the cost of confidence (500 epochs of training). (Trial 2) ........................................................................................................... 70

Figure 53: Anomaly detection made at 20% of a sunken tire, within in an area of pattern (from data collected on March 19th, 2018). (Trial 2) ......................................................... 71

Figure 54: Examples of what would be considered a "light anomaly" (left) and "dark anomaly" (right). (Trial 3) ........................................................................................................... 72

Figure 55: Two "light anomalies" (left) and a clutter of rocks on the right of the AUV. (Trial 3) ....................................................................................................................... 73

Figure 56: Classification made for dark anomaly (left) at 91% confidence and the clutter of rocks (right). (Trial 3) ........................................................................................................... 74

Figure 57: Classification made for light anomalies (left) at 39% and 25% confidence, and the clutter of rocks (right). (Trial 3) ........................................................................................................... 74

Figure 58: Dark anomaly detected (right) at 28% on a segment of the area of pattern that has strayed far from the clutter. (Trial 3) ........................................................................................................... 75

Figure 59: Classification made for light anomaly and dark anomaly (left) at 15% and 11%, respectively. (Trial 3) ........................................................................................................... 76
LIST OF EQUATIONS

1. Convolution Operation: \( s(t) = \int x(a)w(t-a)da \) .. ..................................................17

2. Convolution Operation (Short-hand): \( s(t) = (x * w)(t) \) .. ..................................................17

3. 2-D Matrix Convolution Operation: \( S(i, j) = (I * K)(i, j) = \sum_m \sum_n l(m,n)K(i-m, j-n) \) ................................................................. 18

4. YOLO Detection Confidence Calculation: \( \Pr(Class_i|Object) \cdot \Pr(Object) \cdot IOU_{pred}^{truth} = \Pr(Class_i) \cdot IOU_{pred}^{truth} \) .................................................................22

5. YOLO Grid Cell Predictions: \( S_x \cdot S_y \cdot (B \cdot 5 + \Pr(Class_i|Object)) \) ........................................22

6. Leaky Rectified Linear Activation: \( \phi(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases} \) ........................................26

7. General CNN Loss Function: \( L = \sum_{i=0}^{n} |y_i - h(x_i)| \) .................................................27

8. Loss for Mean Square Error: \( L(x,y) = \frac{1}{n} \sum |x_i - y_i|^2 \) .................................................29

9. Cross Entropy Formula: \( H_y'(y) = -\sum_i y_i' \log(y_i) \) .................................................29

10. YOLO Training Loss Function: \( \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \Pi_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \Pi_{ij}^{obj} \left( (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right) + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \Pi_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \Pi_{ij}^{noobj} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S^2} \Pi_{i}^{obj} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \) .................................................29

11. Ultimate Convolutional Layer Filter Amount: \( \text{filters} = (classes + 5) \times 5 \) .................49
LIST OF ABBREVIATIONS

AI    Artificial Intelligence
ATR   Automatic Target Recognition
AUV   Autonomous Underwater Vehicle
CNN   Convolutional Neural Network
DL    Deep Learning
FPS   Frames Per Second
GPU   Graphical Processing Unit
GUI   Graphical User Interface
HOG   Histogram of Oriented Gradients
ILSVRC ImageNet Large Scale Visual Recognition Challenge
IOU   Intersection Over Union
ML    Machine Learning
ReLU  Rectified Linear Units
ROV   Remotely Operated Vehicle
SVM   Support Vector Machine
TL    Transfer Learning
YOLO  You-Only-Look-Once
1. INTRODUCTION

1.1 OVERVIEW

Since the late 1960’s, computers have been equipped with vision capabilities and have been aiding mankind with vision related problems ever since. With each succeeding decade introducing more robust capabilities than the previous, computer vision has come a long way and has forged the roads for many of the amazing features of technology that we use today in our daily lives. But perhaps more important than a computer’s ability to see, is a computer’s ability to understand what it’s seeing (or in other words, to “learn”). This delves into the old question: can a machine “think”? Artificial Intelligence (AI) founder Alan Turing once was asked if a machine could think, to which he eloquently responded: “A machine is different from a person. Hence, they think differently. The interesting question is, just because something thinks differently from you, does that mean it’s not thinking?” As computer scientists have improved the ability of computers to see the world around them, so too have scientists improved computers’ ability to recognize the objects and surroundings in their field of vision, thus making AI an increasingly more realistic possibility.

There are still many challenges in the field of computer vision, and vision in the underwater setting is no exception. Because the propagation of light is limited in the water environment, to produce reliable images of the seafloor at various depths, we use sound waves (sonar signal) for underwater image generation. Although sound does not produce high quality images in comparison with light, it does provide a decent alternative
for underwater imaging. If we wish to search for an object along the seabed, we can use an AUV equipped with side-scan sonar sensors to pass along the object, and as sonar sound waves are emitted and returned, the AUV is able to produce an image representation of the received acoustic signals. To be able to gather accurate visual information of the seafloor autonomously without endangering civilian lives in the process is the ideal goal, and thus the usage of side-scan sonar imagery on Autonomous Underwater Vehicles (AUV’s) is of the utmost interest. Underwater vision with AUV’s is important for commercial and military applications such as navigation, seafloor mapping, pipeline or cable route survey, and object detection (e.g. shipwrecks, mines, and downed aircraft). [2] Object detection, commonly referred to as Automatic Target Recognition (ATR), is the application addressed in this thesis.

1.2 THESIS OUTLINE

This thesis investigates the ability of a convolutional neural network to be transfer-learned/re-trained to learn the features of an unknown object (anomaly) along the seafloor in side-scan sonar imagery. Contained in Chapter 2 is the problem statement, a more detailed discussion on the main issue at hand and the initial steps needed to formulate a corresponding solution. This will then lead to the main objectives of this thesis. Chapter 3 touches on some necessary background information as well as a brief yet relevant history/literature review of machine learning techniques and will lead into the value and importance of deep learning for computer vision. The deep learning algorithm used in this thesis will be introduced as well as a look at its inner workings, and the chapter will conclude with a look at some related work on attempting to adopt deep learning approaches for ATR in sonar imagery. Chapter 4 will serve as a review of how and where
the data used in this thesis was obtained, as well as a review of how the data was re-formatted and organized for the experimental trials. The general experimental process for all trials will be explained in full detail in Chapter 5. In Chapter 6, a description of the experimental trials and their corresponding results will be discussed individually, as well as their implications. Lastly, the thesis will be wrapped up in Chapter 7, providing a summary of the work, recommendations for future studies, and concluding remarks.
2. PROBLEM STATEMENT

2.1 STATEMENT

ATR in sonar imagery is challenging for numerous reasons. In general, sonar data by nature tends to be noisy. The signal of interest is always relative to the background noise in the ocean, and whether or not a sonar system can detect the echo received from a target or a from sound generated by an undesired source is dependent on the ocean setting. [17] Sources in the ocean such shipping noise, animals, earthquakes, volcanic eruptions, and submarines are all examples of external sources that can contribute to the collection of noisy sonar data. [17] ATR in sonar imagery is a computer vision problem at its root, and most efforts to program an AUV with ATR capabilities have involved using Machine Learning (ML) techniques. ML techniques historically have required large amounts of data, and side-scan sonar image datasets with “targets” are scarcely available for public. This is because data collection typically is expensive, time-consuming, and dangerous, and to gather a sufficient amount of unclassified side-scan sonar data for ATR research has proven to be a road full of many obstacles for researchers.

2.2 MOTIVATION

Our ability to identify objects along the seafloor autonomously (without human intervention) is important for various applications, but above all, to do so autonomously is a matter of civilian safety. Regarding Naval affairs, unexploded ordnance clearance is an application that is of the utmost importance. After World War I and II, many
explosives were unaccounted for along the seabed. To this day they still pose a threat to humans, the environment, and shipping operations. Another important application is subsea structural inspection. Having the ability to inspect the contents of structures along the seabed at sensitive locations can be useful in preventing and handling mishaps such as the Fukushima Daiichi nuclear power plant disaster in Japan and the Deepwater Horizon oilfield disaster in the Gulf of Mexico. ATR in AUV’s is also applicable to search-and-retrieval missions. Being able to autonomously search for and locate missing ships and planes in the ocean such as the Air France flight 447 and the Malaysia Airlines flight MH370 would keep search teams out of harm’s way. But the looming question remains: how is an AUV able to know the difference between a “target” and “not a target”? How can a machine, in real time, be able to know the difference between a rock, an airplane, and a mine?

Much in the same way that a human being can learn to identify familiar objects in an environment over time, so can a computer. This is the crux of the computer science subfield of ML, which has played a large role in computers’ ability to recognize objects of interest in a sonar image and perform ATR. ML gives computers the ability to learn patterns from image examples without being explicitly programmed. ML algorithms are high in popularity and have been applied to side-scan sonar ATR by numerous research teams over the years, yielding adequate results. Although ML has shown promise, there are plateaus and obstacles which prevent its application on side-scan sonar ATR imagery from being truly autonomous and reliable in noisy/turbulent settings. With ML algorithms, the more image data you have for the computer to learn (or “train”) from, the greater the potential ATR accuracy improvement. However, the learning rate of
improvement is logarithmic by nature in standard ML techniques, and thus, ML algorithms plateau in accuracy.

The response to this is the creation of Deep Learning (DL): a relatively new subtype of ML where accuracy improvement is virtually unlimited. DL has been an emerging “hot topic” in the computer vision community in recent years and is currently being used in research by tech company behemoths such as Google, Apple, and Microsoft. One of the main advantages of DL is that while traditional ML algorithms need to test different features and classifiers to achieve best results, DL algorithms learn features and classifiers automatically from the training data directly - therefore DL does not require a problem specific hand-engineered feature extraction step in the algorithm. It learns directly from the input data and discovers multiple levels of distributed representations, with higher levels representing more abstract concepts in the images. [3] Another key advantage of DL is the ability to learn from unlabeled data, making advantage of the large amounts of unlabeled data. Some examples of DL projects in work today include self-driving vehicles that slow down at pedestrian crosswalks, ATM’s that autonomously reject counterfeit bank notes, microscopes that can identify cancer cells in tissue samples, and smartphone apps that can provide instant translation of foreign street signs. [4] Advanced tools and techniques have improved DL algorithms dramatically – to the point where as of 2015, computers have been able to outperform humans at image classification. [4]

While plenty of research has been done on ATR in AUV sonar imagery using traditional ML algorithms, little research has been done using new DL algorithms in the underwater setting. Three factors in particular have hindered this research – namely, easy
access to massive sets of labeled training data, access to computing systems with high-performance GPU’s, and publicly available DL neural network models themselves. [4] Thankfully today, there are numerous resources on the internet today (GitHub, TensorFlow, Google Cloud GPU, etc.) that have made DL research easier than ever.

2.3 OBJECTIVE

The purpose of this thesis is to show that deep learning is a worthwhile approach to deploy regarding tackling the problem of ATR in side-scan sonar imagery. I aim to show that ATR can be achieved in side-scan sonar imagery via “transfer-learning” a pre-trained CNN model with relatively accurate detection using small amounts of training data. It is so often the case that limited data availability inhibits DL research, and I wish to assure the reader that good results can be achieved with small datasets. The main objective of this thesis is to show the feasibility of a state-of-the-art object detection algorithm (the “YOLO” algorithm) to be repurposed to learn the pixel-intensity based features of anomalies in side-scan sonar imagery. Another objective of this thesis is to show that ATR can be achieved using public domain Google-image search results, and thus any researcher with a connection to the internet can gather sufficient training data.
3. BACKGROUND, LITERATURE REVIEW, AND RELATED WORKS

3.1 FUNDAMENTAL PRINCIPLES OF SONAR IMAGERY

The best way to see long spans of ocean floor is to use sonar, and of all the current sonar imaging methods, side-scan sonar is the most commonly used device. [1] Side-scan sonar transducers work by transmitting a narrow sound beam at time intervals while the vehicle traverses through the water. Each sound beam sweeps through a narrow strip of seafloor to the side of the vehicle (typically on both sides), perpendicular to the direction of travel. The sound signals propagate through the water and bounce off the objects/seafloor in its path. The signals reverberate (or bounce off the objects/seafloor), and the returning signals (also known as the “backscatter”) are received by the sonar sensors and recorded. From this acoustic backscatter, an image is constructed of the two-dimensional scan of the seafloor where each row represents one strip of captured signal. The vertical dimension corresponds to the time of scanning each strip, and the horizontal dimension corresponds the time of the returning signal. The strength of the returning signal is represented by pixel intensity. [1]

If an object is found amongst the seafloor, it will appear in the image as a bright region (commonly referred to as the “highlight”) next to a dark region (commonly referred to as the “shadow”). The highlight is produced as a result of the high acoustic reflectivity of the object in comparison with the seafloor, while the shadow produced as a result of the acoustic waves being blocked - preventing the waves from reaching the area.
behind the object. Therefore, a shadow will be produced for any region where there is low (or none-at-all) acoustic reflection. All of this is illustrated in Figure 1 below (the black column in the center of the produced image represents the space occupied by the AUV). [1]

![Diagram of AUV seeing an object](image)

**Figure 1**: How an AUV “sees” an object along the seafloor using side-scan sonar imagery. [1]

Due to the typical high grazing angles of side-scan sonar transducers, objects are prone to cast shadows. AUV’s equipped with sonar systems typically are pre-programmed to traverse at low altitudes to ensure that objects produce good shadows. Shadows are often very useful in identifying objects along the seafloor as they can provide us with information about the size and the shape of the corresponding objects. The length of the shadow is related to the height of the object, its range, and the altitude of the sonar vessel at the time of capture. A shadow can only be as wide as its corresponding object - therefore, sonar images of the same object can appear to be quite
different depending on the particular conditions at the time of capture. As with any sonar system, side-scan sonar can only show objects which reflect sound back to the sonar. This reflection can be influenced by many factors including the object material, the object aspect angle, the seafloor texture, and the overall seafloor topography. [1]

As can be deduced, there are several differences between human vision and sonar imagery. Because sonar images are produced with sound instead of light, the full color spectrum does not apply in sonar imagery. Instead, a sonar image need only be produced with a gradient of one color. Most computer vision/ML algorithms capitalize on differences in color to detect certain features in an image, and although AI research in computer vision is mainly testing on practical human vision applications, the problem of ATR in side-scan sonar imagery can still be investigated by analyzing the work done in the field of computer vision - despite the differences in image production.

3.2 MACHINE LEARNING FOR OBJECT DETECTION

Before arriving to the conclusion that a DL approach holds promise for object detection in side-scan sonar imagery, it would first be of benefit to discuss a brief history of the development of the ML-based object detection algorithms that came before DL over the recent years. The first efficient and praise-worthy method for object detection was developed in 2001 and is known as the “Viola-Jones algorithm” (named after inventors Paul Viola and Michael Jones). [5] This algorithm was developed for face detection and demonstrated the ability to detect faces in real time on a live digital camera feed. They accomplished this by training on a dataset of faces, and hand-coding the distance relationships between the eyes, mouths, noses, and ears of the faces in the images. These distances would be translated into vectors, and the vectors would be fed
into a “Support Vector Machine” (SVM) classifier. An SVM is simply a linear (binary) classifier, which reduces the classification process into a binary “yes or no” method of sorting (face, or not a face). [5] This algorithm was ground breaking in the early 2000’s, and it didn’t take long before face detection became synonymous with the Viola-Jones method.

In 2005, a much more efficient detection technique was developed called “Histograms of Oriented Gradients” (HOG). HOG was invented by Navneet Dalal and Bill Triggs, and was originally developed for the purpose of pedestrian detection in traffic controls. [6] In essence, the HOG algorithm worked by comparing the brightness intensity of pixels in an image and comparing the brightness intensity of a pixel to all of its neighboring pixels. For every individual pixel in an image, the brightness gradient in relation to the surrounding pixels would be recorded into a matrix, and in a new image the gradients would be represented by an arrow pointing in the direction that the pixels would get darker. In the case of facial recognition, an image would be converted into a simple representation that captured the basic structure of a face, and then the algorithm would find the part of the image that looks the most similar to a known HOG pattern that was extracted from the dataset of face training images. Based on a set threshold value for the gradients (or arrows), the algorithm could then decide whether a face was present (or not) in the image. [6]
Figure 2: The method of the “Histograms of Oriented Gradients” (HOG) algorithm.

(Creation of pixel brightness intensity gradients to detect faces). [19]

The HOG algorithm significantly outperformed existing algorithms like Viola-Jones and would remain the preferred method for object detection for nearly a decade, until a new era began: the “Deep Learning Era” in 2012. This ushered in a paradigm shift, where traditional ML algorithms were being outperformed by a great margin, and DL caused the computer vision community to rethink their approaches to object detection from the ground up. Before diving into the details of DL, let us first take a look at the early stages of ATR in underwater vision, so that we can see the value in approaching the task with DL.
### 3.3 ATR IN UNDERWATER VISION

#### 3.3.1 EARLY APPROACHES

The design of the earliest ATR approaches in sonar imagery follows that of the traditional pattern recognition approaches described in section 3.2. In general, an overview of the common design can be described by the figure below:

![Figure 3: Common architecture of a traditional machine learning algorithm for ATR in sonar imagery. [1]](image)

This ML architecture has been the standard approach to ATR in sonar data for decades. [1] The pre-processing step transforms the raw data produced by the sonar sensor into a form suitable for the ML algorithm to process. This step is typically specific to the sensor type and can include transforming operations such as normalization, de-noising, slant range correction beam, surface return removal, and form correction (to name a few). The feature extraction step transforms the pre-processed data from the previous step into a new space referred to as the “feature space”. The purpose of the feature space is to provide better separation between classes than the original space. The feature selection step identifies the relevant features in a training image and rejects any redundant features. This reduces the feature space, which reduces the computational complexity of the algorithm. The classification step assigns a label to each data sample. The last step, post-processing, is where the algorithm seeks to reduce the number of false alarms while maintaining a high detection rate. [1]
When gathering training data for these techniques, human operators would often search for possible highlight and shadow pairs as regions of interest in sonar imagery. Many existing techniques rely on the existence of a shadow rather than the existence of a highlight in deciding whether a potential target is present or not. This is due to the high variability in the pixel intensity of highlights versus shadows in side-scan sonar imagery. Amongst previous work done in ATR for sonar imagery, techniques may also be categorized “unsupervised” or “supervised” learning techniques. Unsupervised learning techniques do not require labeled training data. Often these techniques use “a priori” models and statistics which in some cases may not require training at all, making the decision process is simpler. Supervised learning technique on the other hand learn from labelled sets of data. Rather than dealing with the data directly, these techniques often classify based on a vector of features extracted from the data. When trained on well-annotated data, supervised techniques can outperform unsupervised techniques. [1]

3.3.2 LIMITATIONS OF EXISTING APPROACHES

Although the advances in sonar ATR have been remarkable and made possible by the computer vision community, the performance of existing ATR algorithms is still full of obstacles and has yet to gain widespread acceptance, especially for critical tasks such as mine detection. Some of these obstacles relate to the practicality of the techniques, and some relate to the theoretical limitations. Existing approaches generally demand high computational power, which for many years has hindered the possibility of real-time processing onboard AUVs. Traditional algorithms have included a segmentation step which in many cases can take up to several minutes to process one sonar image. Many previous approaches result in a high number of false alarms, which is one of the biggest
obstacles in carrying out a truly autonomous mission. Regarding the training data, simulated data (or synthetically created data) is often used to increase dataset sizes, however most often the simulated data lacks the same kind of variability found in real data. Most ATR methods have been tested on data recorded in regions of simple/flat seabed and would most likely fail to perform on complex seabed regions (rock, vegetation, and/or sand ripples). Another issue that has come up with traditional ML techniques is the inability to learn cluttered areas, or “area of patterns”. Most existing approaches rely on the features of the objects only and do not utilize the full information in the background of the training data. Most approaches have required large amounts of training data. Obtaining large amounts of data for sonar ATR is troublesome, due to the high cost and limited plausibility of real underwater experiments for researchers to conduct.

Despite the limitations, ATR in sonar imagery has come a long way in a remarkably small amount of time. Although there has been much accomplished by the AI and computer vision community, further research is necessary to overcome the limitations of existing approaches in the underwater setting. This section is not to serve as a criticism of the work done over the years, but rather to serve as a means of retrospection. From prior work we can analyze the techniques laid before us, extract lessons to be learned, and from this we can work towards building an effective approach.

3.4 DEEP LEARNING

As was alluded to at the end of section 3.2, the field of AI was forever changed upon the birth of DL. DL algorithms work by training a neural network (or model) to classify data that can come in the form of images, text, or even sound. DL neural networks are
typically models with many layers of neurons, inspired by the architecture of the mammalian neocortex. DL is superior to ML in that traditional ML algorithms can contain only two or three neural network layers for classification, while DL algorithms can contain hundreds of layers, improving accuracy of object recognition tenfold. [4] A deep neural network combines multiple nonlinear processing layers using simple elements operating in parallel to each other. In general, deep neural networks consist of an input layer, several hidden layers, and an output layer. The layers are interconnected via nodes (or neurons) with each hidden layer using the output of the previous layer as its input, sequentially (as shown in Figure 1). [4]

![Figure 4: General architecture of a deep neural network.](image)

To better understand this process, consider the example: let’s say we have a set of images where each image contains one of five different categories of objects, and we want the DL network to automatically recognize which object is in each image. The images are labeled according to their category (pre-process), in order to have training data for the network. Using this training data, the network can then start to understand the object’s specific features and associate them with their corresponding category. Each
layer in the network takes in data from the previous layer, transforms it, and passes it on. The network increases the complexity and detail of what it is learning from layer to layer. [4] Finally, a new image is introduced in the output layer, and the network is then able to classify the object in the image. There are different types of deep neural networks, however the model that is best suited for image and video data is the convolutional neural network (CNN). [4]

3.4.1 DEEP CONVOLUTIONAL NEURAL NETWORKS (CNN’S)

Convolution is useful for image data because it is useful for working with inputs of variable size. In its most general form, convolution is an operation on two functions of a real-valued argument, and the operation can be defined by the equation:

Convolution Operation: \( s(t) = \int x(a)w(t - a)da \)

where both \( x \) and \( t \) are real-valued, \( x(t) \) is a single output, and \( w(a) \) is a weighting function where \( a \) is the age of a measurement. [22] Convolution can be defined for any functions for which the above integral is defined and can be used for other purposes besides taking weighted averages. The operation is sometimes denoted as the short-hand version with an asterisk:

Convolution Operation (Short-hand): \( s(t) = (x * w)(t) \)

In computer vision/machine learning terminology, the first argument (in this example, the \( x \)-function) is often referred to as the input, and the second argument (the \( w \)-function) as the “kernel”. The output is sometimes referred to as the “feature map”. [22] The feature map gives us information about which features were present at different positions.
in an image. Specific combinations of these features then make it possible to classify the image (or parts of it) into different categories. Traditional neural network layers multiply matrices by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit - thus, every layer output unit interacts with every layer input unit. CNN’s, however, typically have “sparse interactions”. Sparse interactions mean that fewer parameters are needed to be stored, which reduces the memory requirements of the model and improves its statistical efficiency. [22] This is done by making kernels smaller than their inputs. For example, when we process an image, the input image might have millions of pixels, but with sparse interactions we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. [22]

For an ML application using image data, the input will be a multidimensional array of data and the kernel will have a multidimensional array of parameters that will be adapted by the learning algorithm. These multidimensional arrays are commonly referred to as “tensors”. Because each element of the input and kernel must be stored separately, we assume that these functions are zero at values everywhere but the set of points for which we store values. Therefore, we can implement the infinite summation as a summation over a finite number of array elements. [22] We often use convolutions over more than one axis at a time. If we are to use a two-dimensional image “I” as our input, then we will use a two-dimensional kernel “K”. This can be described in the equation:

2-D Matrix Convolution Operation: \( S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \)

Due to the commutative nature of convolution, the above equation can also be written as:
2-D Matrix Convolution Operation: \( S(i, j) = (I \ast K)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n) \)

A CNN extracts features from the training data layer-to-layer by means of applying convolutional transformations. Like other neural networks, a CNN is composed of an input layer, an output layer, and many hidden layers in between. The “in between” layers are responsible for feature detection (changes in color gradients, borders, and shapes). When a computer takes an image as input, it stores an array of pixel values. Depending on the resolution and size of the image, it will see an array of dimensions ‘\(X\)’ by ‘\(X\)’ by 3 of numbers (the 3 referring to RGB (red-green-blue) values). Each of these numbers is given a value ranging from 0 to 255, which describes the pixel intensity at that specific point in the image. [44] For the deep learning algorithm used in this thesis, these numbers (derived from pixel intensity) are the only inputs available to the computer for training.

In the grand scheme, a CNN has two main sections: feature detection and classification. An overview of the common CNN algorithm design is shown in Figure 2.

![Figure 5: An overview of the common CNN algorithm design. [4]](image-url)
The feature detection layers perform one of three types of operations on the data: convolution, pooling, and rectified linear unit (ReLU). The convolution operation puts the input images through a set of convolutional filters, which activates certain features of interest from the images (as previously described). “Pooling”, in general, is a form of dimensionality reduction used in CNN design where the goal is to throw away unnecessary information and only preserve the most critical information. These layers are non-learnable and are used to reduce the spatial dimensions of the feature maps as they pass through the network. They are associated with some kernel of size $k \times k$ and a stride $s$. In simplest terms, the pooling operation simplifies the output by performing nonlinear down-sampling, reducing the number of parameters that the network needs to learn about, which in turn speeds up the process. The ReLU (Rectified Linear Units) operation allows for faster and more effective training by mapping negative values to zero and maintaining positive values. ReLU is considered an “activation function and has been regarded as a default recommendation for an activation function on hidden units in a CNN. [41] An activation function is used to limit the output of a neuron and to also introduce nonlinearities to the linear activations generated by a convolution layer, without which the model would remain a linear combination of its inputs. The input to such functions are typically the output of a preceding convolution layer. The importance of the activation function comes from its impact on the ability to minimize the cost function and the ability to propagate signals through several layers. [41] In a typical CNN, these three operations are repeated over tens or hundreds of layers, with each layer learning to detect different features. [4]
In 2014, a variant of the CNN was introduced to the computer vision community which would help to make real-time object detection even more efficient: the R-CNN. An R-CNN takes an image and creates bounding boxes (or region proposals) using a process called “Selective Search”. A bounding box describes the rectangle that encloses an object. The Selective Search process looks at the image through sliding windows of different sizes, and for each size it attempts to group together adjacent pixels according to texture, color, or intensity to identify objects. Instead of classifying every region using a sliding window, the R-CNN detector only processes those regions that are likely to contain an object. This greatly reduces the computational cost incurred when running a CNN. The images in the bounding boxes are run through a pre-trained CNN, and an SVM is used to classify what object the image in the box is. Last in the process, the bounding box is run through a linear regression model to output tighter coordinates for the box once the object has been classified. [18]

![Figure 6: An overview of the R-CNN algorithm. [18]](image)

R-CNN’s have been extremely popular in the past couple of years for object detection problems, and improvements from the original R-CNN algorithm of [18] have been made ever since. In 2016, a real-time object detection algorithm was developed which borrowed concepts from the R-CNN design and would outperform all previous methods: The “You Only Look Once” (YOLO) detection system. [19]
3.4.2 STATE-OF-THE-ART, 2018: YOU ONLY LOOK ONCE (YOLO)

YOLO takes a differing approach to its predecessors in that it is not a traditional classifier that is repurposed to be an object detector, like the algorithms before it. The most current version of YOLO, “YOLOv2” a.k.a. “YOLO9000”, is the fastest and most powerful rendition of the algorithm, and for the remainder of this thesis I will refer to the most current version simply with “YOLO”. The neural network predicts bounding boxes and their classifications directly from a full-sized image. It differs from earlier region-based methods where instead of trying to classify the same region several thousand times, the network instead looks at every part of the image once, hence the name “You Only Look Once”. The YOLO algorithm first takes its input image and divides up the image into a grid of $(S \times S)$ cells. Each of these cells is responsible for predicting $(B)$ bounding boxes and class probabilities $\Pr(Class_i|Object)$. [20] The predictions are encoded into a tensor by the equation:

$$\text{YOLO Grid Cell Predictions: } S \times S \times (B \times 5 + \Pr(Class_i|Object))$$

If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. [20] The algorithm also outputs a confidence score which tells us how certain it is that the predicted bounding box encloses a detected object. [19] The confidence score is calculated by the formula:

$$\text{YOLO Detection Confidence Calculation: } \Pr(Class_i|Object) \times \Pr(Object) \times$$

$$IOU_{pred}^{truth} = \Pr(Class_i) \times IOU_{pred}^{truth}$$
where $\Pr(\text{Object})$ is the probability that an object has been detected, $\Pr(\text{Class}_i)$ is the probability that a certain $i$th class has been detected, and $\text{IOU}_{\text{truth}}^{\text{pred}}$ is defined as the intersection over union (IOU) between the predicted box and the ground truth (training data). [20] The higher the confidence score, the thicker the bounding box will be drawn. For each bounding box, the cell also predicts a class. This works similarly to a traditional classifier in that it gives a probability distribution over all the possible classes. The confidence score for the bounding box and the class prediction are combined into one final score that tells us the probability that this bounding box contains a specific type of object. [19] An example of this process is shown below, where only detections above a confidence score threshold of 30% or higher are kept.

*Figure 7: The general process of the YOLO algorithm. In this example, since the confidence scores of the dog, bicycle, and car are all above 30%, they are kept for the final outcome.* [20]
With an NVIDIA Titan X GPU, the network can run at 45 frames per second with no batch processing, and this means that YOLO can process real-time streaming video with less than 25 milliseconds of latency. [20]

3.4.3 THE DESIGN OF YOLO

![Diagram of YOLO architecture](image)

Figure 8: A visualization of the original architecture of the YOLO convolutional neural network. [20]

The original YOLO model was trained on the “ImageNet” dataset, a 1000-class dataset used in the annual competition “ImageNet Large Scale Visual Recognition Challenge (ILSVRC)”. [20] [23] The entire network consists of a combination of convolution layers, some of which are followed by a “max pooling” layer. The convolutional layers consist of multiple filters that are defined by their parameters (or “weights”). The layer defines the number of filters and their corresponding kernel size, the size/stride in which they are applied and the amount of padding to handle image borders. The convolved output of a filter is called the “feature map”, and a convolutional layer with $n$ filters creates $n$ feature maps, which are then used as the input for the next
layer. The max pooling operation fits the maximum output of the preceding layer within a rectangular neighborhood. In this case, pooling helps to make the representation become approximately invariant to small translations of the input. [21] The initial convolutional layers of the YOLO network extract features from the image while the fully connected layers predict the output probabilities and coordinates.

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size / Stride</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv</td>
<td>32</td>
<td>3 x 3 / 1</td>
<td>416 x 416 x 3</td>
<td>416 x 416 x 32</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>416 x 416 x 32</td>
<td>208 x 208 x 32</td>
</tr>
<tr>
<td>Conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>208 x 208 x 32</td>
<td>208 x 208 x 64</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 64</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>Conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 64</td>
<td>104 x 104 x 128</td>
</tr>
<tr>
<td>Conv</td>
<td>64</td>
<td>1 x 1 / 1</td>
<td>104 x 104 x 128</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>Conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 64</td>
<td>104 x 104 x 128</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 128</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>Conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 128</td>
<td>52 x 52 x 256</td>
</tr>
<tr>
<td>Conv</td>
<td>128</td>
<td>1 x 1 / 1</td>
<td>52 x 52 x 256</td>
<td>52 x 52 x 128</td>
</tr>
<tr>
<td>Conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 128</td>
<td>52 x 52 x 256</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 256</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>Conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 256</td>
<td>26 x 26 x 512</td>
</tr>
<tr>
<td>Conv</td>
<td>256</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>Conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 512</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>Conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>Conv</td>
<td>512</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>3 x 13 x 512</td>
</tr>
<tr>
<td>Conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>Conv</td>
<td>512</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>Conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>Conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>Route 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv</td>
<td>64</td>
<td>1 x 1 / 1</td>
<td>26 x 26 x 512</td>
<td>26 x 26 x 256</td>
</tr>
<tr>
<td>Reorg</td>
<td>/2</td>
<td></td>
<td>26 x 26 x 256</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>Route 27 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1280</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>Conv</td>
<td>30</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
<td>13 x 13 x 30</td>
</tr>
</tbody>
</table>

detection

Figure 9: The YOLOv2 layer configurations.

This design was inspired by the Google’s own CNN model “GoogleNet”. [20] The original network in total consists of 24 layers (19 convolutional layers and 5 max pooling layers) followed by 2 final fully connected layers for producing output. The ImageNet
input images had a resolution of 224 x 224, however the first layer of the YOLO model doubles this to 448 x 448. This is because detection often requires fine-grained visual information, so the resolution is doubled. After the data is completely passed through the network, the final layer of the model is of the dimensions 7 x 7 (due to the choice of grid number S = 7 by the authors). [20] The final layer predicts both class probabilities and bounding box coordinates, and the bounding box is normalized so that the width and height of the box fall between values 0 and 1. A linear activation function is used for the final layer, and for all other layers, a function called “Leaky Rectified Linear Activation (Leaky ReLU)” is used:

\[
\emptyset(x) = \begin{cases} 
  x, & \text{if } x > 0 \\
  0.1x, & \text{otherwise}
\end{cases}
\]

Compared with regular ReLU, Leaky ReLU compresses the negative parts rather than mapping them to a constant zero, which makes the network allow for small, non-zero gradients when the unit is not active. The YOLO model optimizes for sum-squared error. Sum-squared error is used because it is relatively easy to optimize, although it weights localization error equally with classification error which is something that needs to be adjusted for. Another thing that needs to be accounted for is that many grid cells in an image do not contain an object to detect. This pushes the confidence scores of those cells to equal zero, often overpowering the gradient from the cells that do contain objects (which can lead to model instability). [20].

To remedy this, YOLO implements a loss function modification. When training a CNN, forward passes for a number of training images are computed. The predictions are outputted and compared to ground-truths to generate a what is known as a “loss”. Using calculus, the sensitivity of this loss with respect to changes in network parameters
can be found. In the context of CNN’s, loss functions are differentiable functions which characterize the similarity of a label with some features predicted by the network given an input. In general, a loss function for a CNN can be described by the equation:

\[
L = \sum_{i=0}^{n} |y_i - h(x_i)|
\]

where \( L \) is the loss, \( y_i \) is the ground truth and \( h(x_i) \) is the output of the model. The loss function is largely responsible for preserving the important features of the training data. It is the loss functions job to prevent valuable information from being lost during training. A good loss function is able to keep the loss low during training. Theoretically speaking, a good loss function will decrease the loss amount (as close to zero) as much as it can as time progresses during training. This is illustrated in the figure below:

![Figure 10: Visualization of image feature regeneration due to different loss functions as training progresses. [42]](image)

In the YOLO loss function, the loss from bounding box coordinate predictions is increased, and the loss from confidence predictions for boxes that don’t contain objects is decreased. This is done using two parameters: \( \lambda_{\text{coord}} \) and \( \lambda_{\text{noobj}} \). \( \lambda_{\text{coord}} \) is set to equal 5, and \( \lambda_{\text{noobj}} \) is set to equal 0.5 (these are the default YOLO values). Sum-squared error also equally weights errors in large boxes and small boxes, which needs to be accounted for since the error metric of YOLO reflects that small deviations in large boxes matter less.
than in small boxes. To address this, the square root of the bounding box width and height is predicted instead of the width and height directly. [20]

Because it is desired that only one bounding box predictor is responsible for each object during training, one predictor is assigned to be “responsible” for predicting an object based on which prediction has the highest current IOU in comparison with the ground truth. This in turn leads to specialization between the bounding box predictors. As a result, each predictor gets iteratively better at predicting certain sizes, aspect ratios, or classes of objects, improving overall recall. [20] Because YOLO directly regresses on an entire input image, its loss function captures both the bounding box locations as well as the classification label of the objects in an image. The loss function for the YOLO algorithm can be divided into five parts:

1. Loss according to the bounding box center x and center y.
2. Loss according to the square root of the width and height of the bounding boxes.
3. Penalization of predicted objects.
4. Penalization of unpredicted objects.
5. Penalization in the difference of class probabilities.

The square root of the width and height for the loss function is used to take care of differences in bounding box sizes: $\lambda_{\text{coord}}$ is a scaling factor on the bounding box coordinates to ensure bounding box penalties and class probability penalties contribute equally to the loss. $\lambda_{\text{noobj}}$ is a scaling factor to penalize object identification when there is no object. This can all described in the multi-part loss function:
YOLO Training Loss Function: $\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B^i} \mathbb{1}_{ij}^\text{obj} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B^i} \mathbb{1}_{ij}^\text{obj} \left[ \left( \sqrt{w_i} - \sqrt{\overline{w}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\overline{h}_i} \right)^2 \right] +$ 

$\sum_{i=0}^{S^2} \sum_{j=0}^{B^i} \mathbb{1}_{ij}^\text{obj} \left( C_i - \overline{C}_i \right)^2 + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B^i} \mathbb{1}_{ij}^\text{noobj} \left( C_i - \overline{C}_i \right)^2 +$ 

$\sum_{i=0}^{S^2} \mathbb{1}_{i}^\text{obj} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$ 

where the $(x, y)$ coordinates represent the location of the centroid of a bounding box relative to the bounds of the grid cell, $h$ represents the height of the bounding box and $w$ represents the width of the bounding box, $C_i$ represents the confidence score of detection, $p_i(c)$ is the classification loss, $\mathbb{1}_{i}^\text{obj}$ denotes if an object appears in cell $I$, and $\mathbb{1}_{ij}^\text{obj}$ denotes that the $j$th bounding box predictor in cell $i$ is “responsible” for that prediction. [20]

YOLO’s loss function simultaneously penalizes incorrect object detections as well as considers what the best possible classification would be, making the algorithm a great candidate for both detection and classification tasks. The function may seem intimidating at first glance, but it is simply a combination of two common loss functions used in CNN design: the mean square error and cross entropy functions (for the classification loss).

[20] The loss function for mean square error can be described as:

Loss for Mean Square Error: $L(x, y) = \frac{1}{n} \sum |x_i - y_i|^2$

where $L$ is the loss and $(x,y)$ are the value coordinates for a given image. The function for cross entropy can be described as:

Cross Entropy Formula: $H_y'(y) = - \sum y_i \log(y_i)$
Where \( H_{y'}(y) \) is the loss from cross entropy, \( y_i' \) is the ground truth label of the \( i \)th training instance and \( y_i \) is the prediction results of the classifier. [43]

### 3.5 TRANSFER LEARNING

TL is a commonly used technique on DL algorithms where you can take (or “borrow”) a pretrained network model and use it as a starting point to learn a new task. Fine-tuning a network with TL is usually much faster and easier than training a network from scratch with randomly initialized weights. You can quickly transfer the already learned features to a new task and classify new targets using a smaller, limited number of training images. By keeping the features from the early layers of a pretrained network (the transferred layer weights) and then fine tuning the final layers of the classification section of the algorithm, you can apply a working CNN model to a new problem on unfamiliar data. TL can be especially useful when faced with the problem of not having enough training data to rebuild models. Transfer learning using CNN’s has been implemented in various fields to date - in the field of medical image processing for example, where there is a lack of massive amounts of training image data, transfer learning has been an effective method when employing CNNs to medical image classification with the help of sufficient annotated natural images. Transfer learning techniques have been implemented to the problems of identifying early stages of cancer cells, lung tissue pattern classification, and recognition of a developing fetus. [7]

A good example to demonstrate this is MATLAB’s transfer learning tutorial. This example shows how to fine-tune the well-known CNN model “AlexNet” [8] in order to identify unfamiliar objects to the network. AlexNet is a model that was trained on a subset of the ImageNet database. [9]
Figure 11: Examples of image categories used in the CNN "AlexNet". [4]

The model was trained on more than a million images and can classify images into 1000 object categories (for example: keyboard, mouse, pencil, and many animals, etc.) and as a result, the model has learned rich feature representations for a wide range of images. In the example, the goal is to transfer learn the model to be able to classify between new objects (a MathWorks cap, deck of playing cards, and a screwdriver). After transfer learning is carried out, the network trains to learn the feature representations of the new objects and the algorithm is able to classify them using a live webcam. [11]
Figure 12: Transfer-learned CNN “AlexNet” able to correctly identify new (MathWorks) office objects.

Another MATLAB example shows how to retrain an R-CNN object detector for detecting a stop sign in a video. For this example, an R-CNN is pre-trained using the CIFAR-10 data set called CifarNet [12], which has 50,000 training images. This pre-trained R-CNN is then fine-tuned for stop sign detection using just 41 new training images. Without pre-training the R-CNN, training the stop sign detector would require many more images to make sufficient detections. The code returns the object bounding boxes, a detection score, and a class label for each detection in each frame of the video. The labels are useful when detecting multiple objects (stop, yield, or speed limit signs, etc.). The scores (which range in values between 0 and 1) indicate the confidence in the detection and can be used to ignore low scoring detections (if the user should choose to set a detection threshold). [13]
Figure 13: Stop sign detection (and confidence %) after reconfiguring the CifarNet CNN via TL. [13]

These TL examples made public by MATLAB offer a window into the inner workings of the DL and CNN design, and allow researchers to adopt pretrained CNN’s to their unique problems. CNN/R-CNN design itself is beyond the scope of this Master’s Thesis, and to create a custom model, tailored to feature-extract and classify specific seafloor objects within a sonar image would be a task appropriate for a Ph. D dissertation. Because of the lack of large, publicly available data sets that one can draw from online, sonar ATR researchers are limited to small sets of data provided to them by Naval research sponsors. It is for this reason, that utilizing TL techniques for the problem of sonar ATR is ideal.

3.6 DEEP LEARNING FOR ATR IN UNDERWATER VISION

A team of researchers to come to this conclusion was the team of [14]. In their work, deep CNN’s were used to perform underwater target classification in synthetic aperture
sonar (SAS) imagery. The deep networks were custom designed and were trained using a massive database of real, measured sonar data collected at sea by Bluefin’s MUSCLE AUV during different expeditions in various geographical locations. They did this using a cascaded, integral-image-based detection algorithm applied to scene-level SAS images in order to generate a set of potential objects of interest that would be examined further in a subsequent classification stage. [14] For each alarm from the detection stage, a “mugshot” of the object is extracted from the scene-level SAS image. Three binary classification experiments were conducted (the difference among them being which objects were treated as belonging to each class), however, the same CNN architecture was used in all experiments. The objects in the images consisted of dummy mine shapes, more realistic mine-like targets, random man-made objects, and calibrated rocks, and many of the images did not contain any objects. The target class consists of cylinders, truncated cones, and wedge-shaped objects, all of which mimic common mine types. The non-target class consists of specially calibrated rocks and various man-made objects (whose size and shape are similar to targets) including a washing machine, a cylindrical diving bottle, and a weighted duffel bag, among others. Experiment A sought to discriminate targets from non-targets, experiment B sought to discriminate truncated cones from the calibrated rocks, and experiment C sought to discriminate the mine type known as “mantas” from truncated cones (which optically, the two classes of objects were very similar). For each experiment, a unique deep convolutional neural network was learned in isolation. All experiments draw their training data from the same set data. [14]

In each experiment, a comparison of three different classification approaches were made. Two approaches employ the learned CNN (the first version uses the network as is
and the second variant mirrors each mugshot and takes the average of the network predictions), and the third approach uses a modified version of a relevance vector machine with the classifier parameters directly weighting a small set of traditional features that have previously been found to characterize various attributes of the objects well. The results of their experiments showed that the proposed deep networks were far superior to the traditional feature-based approaches. This can be directly attributed to the much greater capacity and complexity of the deep networks, whereas the feature-based approach is limited by its relatively tiny number of parameters. [14]

The same research team of [14] extended this work in [14] 2 years later to improve their results and more importantly, to demonstrate the feasibility of TL for AUV SAS imagery. To do this, they took CNNs trained using full-resolution mugshots but then simulated lower-resolution test mugshot imagery (by down-sampling by a factor in each image dimension). The results suggested that sonar TL was plausible, with gradual degradation in performance. This implied that the CNNs trained on the MUSCLE sensor data are relevant for exploitation with data collected by a wide range of other lower-resolution side-scan sonars (whose resolution is up to eight times poorer in both along-track and range). The results also hinted that the CNN architectures that are fully exploiting high-resolution information, and conversely, would be more suitable for successful TL.

Once it became established that TL was feasible for sonar data, the team of [15] took initiative and conducted research on various well-known pretrained CNN’s for ATR in sonar imagery. The pretrained CNNs that were used in their experiments were: VGGnet with depth 16 (VGG16), VGGnet with depth 19 (VGG19), fast VGGnet (VGG-f), and
AlexNet (Alex). To test the performance of these CNN’s, a MATLAB testbed (available for public use) called “MatConvNet” [21] was used. The data used was SAS data provided by the Naval Surface Warfare Center. Their problem was to discern between the images containing blocks, cones, spheres, and cylinders along the seabed. With this data set, they designed four trials per method using 20 training images per class and then tested on a random assortment of ten examples per class. [15] In addition to two pretrained CNN strategies, two traditional ML algorithms were and a hybrid SVM/CNN algorithm were tested for comparison. The 4 original CNN’s had the final layers designed according to their number of classes. To fine-tune a model to the new problem of sonar ATR, they simply replaced the last layers with new layers called “softmax” layers.

Figure 14: Target recognition (middle image) and classification (right image) due to transfer learning pretrained CNN models onto sonar image data in [15].

Figure 15: A comparison of mean precision and recall values from different CNN models in the experimental trials done in [15].
The results of their experiments (as shown in the figure above) revealed that when a
CNN is combined with an SVM, it can overcome the limited training data hindrance and
show impressive results (the SVM with CNN features from AlexNet and VGG19 both
had only 2 incorrect classifications out of 160 tests, each). The plot on the right shows the
results of different CNN models. While they all did relatively well when compared to the
baseline methods, a trend was observed: the more layers in the network, the better the
performance (with fine-tuned AlexNet yielding the most accurate results). [15]

Until present day and to my knowledge, there has been at least two research teams to
apply the YOLO algorithm to underwater vision aid using AUV’s (although neither to
implement YOLO on side-scan sonar imagery). The research team of [39] used YOLO to
create a system for counting wild scallops along a seabed using visual imagery to monitor
health of the scallops and their ecosystem. In their work, they focused on re-training the
model to learn only one class (“scallop”) for detection. To do this, they used a large
dataset of 170,000+ manually-annotated images captured in scallop-rich waters in order
to learn the features that represent what would be considered a healthy scallop. The
images in the dataset were collected by a downward-pointing digital camera attached to
the nose of an AUV, which moved at an altitude of a few meters above the seabed. To
account for limited visibility, the seabed was illuminated with a flash strobe. [39] This
research contained 4 trials, all of which differed in terms on partition of the training and
testing data, and how the YOLO network was modified. All trials allowed training to
endure for 10,000+ step iterations. The sizes of the datasets for each trial were each only
a small fraction of the large collection but were still to the order of thousands.
Figure 16: An example of results from [39], where the YOLO model was repurposed to detect and count healthy scallops by a downward-pointing digital camera attached to the nose of an AUV. [39]

Ultimately, the research of [39] yielded successful results in producing a system that achieved high accuracy while running fast enough to keep up with the AUV’s live image capture rate. Contrary to the work of [39], the work of [40] repurposed the YOLO algorithm to detect features in forward-looking sonar images. The purpose of this research was to enhance a remotely operated vehicles (ROV) underwater manipulation and navigation abilities via feedback from a cooperative AUV (called the “Cyclops”).

Figure 17: The joint ROV/AUV system described in the work of [40].
In their work, the YOLO algorithm was implemented onto a stream of successive sonar images to detect and track the ROV from the information provided by the forward-looking sonar transducer attached to the AUV. The ROV acted as an “end-effector robot”, which was connected to the AUV via tether control. The purposes of the ROV was to perform armless underwater manipulation or to execute docking of the entire joint-system. The training data used in this research was gathered from real-sea experiments conducted with the Cyclops AUV itself. In total, 1,607 images were used for training to detect one class (the ROV, labeled as “true”), and 1,000 images were used for testing. By having the entire cooperative system learn examples of previous ROV positions, the system was able to provide the ROV with positional awareness by sharing with it information captured by the forward-looking sonar from the AUV in real-time.

Figure 18: An example of results from [40], where detection and tracking of the ROV is achieved with YOLO.

These are some of the most recent attempts at implementing the robust, state-of-the-art available algorithm for object detection “YOLO” in the underwater setting. Both of these works accomplished positive results by training on large datasets (in the order of
thousands). Neither of these works contain experiments with small sets of data, and to my knowledge, the algorithm has yet to be tested on side-scan sonar imagery for AUV’s.
4. DATA ACQUISITION

4.1 INSTRUMENTATION

For much of the training and testing data used in the experiments, a Hydroid REMUS-100 AUV was used to obtain side-scan sonar data.

![Hydroid REMUS-100](image)

*Figure 19: The Hydroid REMUS-100 with all of its built-in features and accessories.* [30]

The REMUS-100 has many built-in sensors and equipped with numerous features, but for the sake of my experiments, I solely am interested in the side-scan sonar sensors. The AUV weighs roughly 80 lbs. and can travel at a max speed of 4 ½ knots using a DC brushless motor. It has 8-10 hours of run time and a 100-meter depth rating. The REMUS-100 turns on when a magnetic power switch is triggered, and the magnet is removed. The vehicle then can communicate via acoustic modem messages to a ‘Tow-
Fish’ connected to a “Ranger”. This provides the vehicle range to the Tow-Fish as well as health and power information and can be used to send commands manually to resurface the vehicle in case of emergency. The vehicle has two ports for data transfer; one being a low speed serial port and the other being a high-speed Ethernet port.

Figure 20: Approximate altitude and average travel speed of the REMUS-100 for all missions.

All data used was collected with a frequency of 900 kHz, an array pitch angle of $\theta_c = 10$ deg, a max slant range of $R_{\text{max}} = 30$ m, a minimum slant range of $R_{\text{gap}} = 2.7$ m, a sonar altitude of $H_F = 1.4$ m, a maximum speed of $V_{\text{max}} = 2$ m/s, and a ping rate of $f_{\text{ping}} = 16$ Hz (pings/sec). For all missions, the average speed of the vehicle was approximately $V_{\text{avg}} = 1.5$ m/s and the altitude above the seafloor was set to be $H_A = 4$ m.

The data used in this thesis are from several real-sea experiments conducted at SeaTech Oceanographic Institute of Florida Atlantic University (see appendix item 5). The surveys of each mission were identical, predefined trajectories off the coast of Fort Lauderdale (approximately at the latitude of Las Olas Blvd and 1.3 km off the coast) in approximately 10-20 m of water depth (see appendix items 6 and 7). Collectively, the AUV missions were conducted on:
4.2 DATA PREPARATION

The raw side-scan sonar data from the REMUS-100 came in the form of pairs of "Sonar Data Streaming (.sds)" files, each with a corresponding "Sea Scan Survey (.xvy)" file. For each mission, both files are needed to review the collected data. In total, side-scan sonar data from 13 separate missions were used (see appendix item 8). To view the side-scan sonar files, a software package called “Sea Scan Survey” by Marine Sonic Technology, Ltd is needed. [31] To load a mission, you simply select an .sds file. Once a mission is loaded, the side-scan sonar imagery will appear and will stream the data as the AUV progressed forward through the water on its trajectory.

Figure 21: The Sea Scan Survey graphical user interface (GUI).
The Sea Scan Survey GUI also provides latitude and longitude information on any captured position on the seafloor. All missions were converted and saved as video (.mp4) files by using the open source software “Loom Video Recorder”. From the original 13 missions, 11 video files were created. Of these video files, some would be used as a “testing video” and some would be used to extract training images from, depending on the experimental trial. The images used for training were extracted from the sonar data via the software Sea Scan Review” [31]. The partitioning of the training data and testing data will be explained in detail in Chapter 6.

Since I am interested in detecting targets, the 11 video files were then edited using Apple’s “iMovie” software in order to remove sections of the missions where the AUV did not come across a potential target (see appendix item 9). 16 separate videos with targets were created from these edits, however one video was kept (No_Targets) where the REMUS-100 traversed through the ocean and the seafloor was completely vacant of anomalies (see appendix item 10). The purpose of the video with no targets is to provide a “null set”, so that we can test to ensure that no target detection is made where there are in fact no targets.
5. SCIENTIFIC APPROACH

5.1 STEP 1: THE SOFTWARE

The software used in this research utilized several dependencies/requirements in conjunction. For all research done in this thesis, a Scientific Linux operating system was used, and all software installation was done by entering commands via the Linux terminal. The first requirement was either Python 3.5 or Python 3.6, but installing Anaconda works sufficiently. [24] [25] Most of this research was done by executing Python scripts, so having a basic understanding of the Python/Anaconda environment was necessary. The next requirement was TensorFlow (GPU version). TensorFlow is an open-source software library designed by Google for dataflow programming purposes and is among the most popular in machine learning libraries. [26] The last requirement was OpenCV. [27] OpenCV was useful during the testing phase and output the confidence scores on the detection bounding boxes.

Once all the requirements were installed, the next step was to download and install “Darkflow”. [28] Darkflow is a modified version of “Darknet”, which is an open source neural network framework compiled in C and CUDA and written for TensorFlow by one of the YOLO authors (Joseph Redmon) (see appendix item 11). [29] Despite the ominous sounding name, have no fear, for Darknet/Darkflow is safe to download and use and has no relation to the notorious “Dark Web”. After Darkflow was installed, I then gathered and modified a collection of python scripts which helped to make my life a lot easier regarding pointing to file locations running certain commands. The scripts were
originally written by AI/optical engineering researcher Mark Jay and can be downloaded from his Github.com profile. [34] Once all the scripts were gathered, they were saved in the main darkflow directory folder (see appendix item 12).

5.2 STEP 2: DATA ANNOTATION

Once all necessary software and requirements were installed, I then proceeded to annotate the training data. The reason I needed to annotate the training data is because YOLO is considered a “supervised” learning algorithm. As mentioned in section 3.3.1, this means that the user needs to tell the algorithm where the class (or target) is in each training image (it does not identify the class in each training image on its own). To start, I move all the training images to a folder in the main directory called “images” (See appendix item 13). To annotate the training images for a trial, I would start by running a python script called “draw_box.py” (see appendix item 1) which will open a GUI using “matplotlib” (part of the Python package) (shown in Figure 31).

![Figure 22: The draw_box.py GUI for annotating the training images.](image)

The GUI loads one image at a time, and to label an anomaly in an image, the user simply drags a box (from top left corner to bottom right corner) around the anomaly, for
every anomaly in the image. Once all anomalies in the loaded image have had a box drawn around them, to go to the next image, the user enters “q” on the keyboard. This will save the feature coordinates of the anomalies in the image into a “.xml” file in a folder called “annotations” in the main darkflow directory, which will be used during the training process to recall the coordinates. Once this process is done for all training images in the folder, the GUI will automatically close upon finished the last annotation, and all needed .xml files will have been created (see appendix item 14). After all images have been annotated, I then created a new folder called “train” in the main darkflow directory, where I copied the images and annotations folder to for training.

5.3 STEP 3: TRANSFER LEARNING THE YOLO MODEL

The creators of YOLO made a few different versions the original YOLO model (see appendix item 19). The “YOLOv2” neural network model was created to classify 9000 different classes, however, my experiments only require detection of one class (anomaly). Therefore, instead of transfer-learning the YOLOv2 model, it is more suitable for my research that I worked with the model “TinyYOLO”. TinyYOLO was trained to learn 80 classes and uses fewer convolutional layers (9 instead of 24) and fewer filters within those layers, which makes it more applicable for training on a much smaller number of classes (in my case, only 1 class). All training and testing parameters are otherwise the same between YOLOv2 and TinyYOLO. The original YOLO model processes images in real-time at 45 frames per second, while tinyYOLO can process at 155 frames per second, which in turn will make training and testing much faster and more considerable for real-time implementation. [20] In order to transfer-learn a pre-trained model, you need to have two files: it’s “.cfg” file and its “.weights” file. The .cfg file provides
training instructions/configurations on how the model should be created. The .weights file is a representation of the model itself, storing the numerical weighted values for the learned features on the training image set. Both the .cfg and the .weights files can be downloaded from the YOLO creators’ website. [29]

Once both files have been downloaded, I then needed to edit the configurations in the .cfg file to transfer-learn the tinyYOLO model. To do this, I made a copy of the .cfg file “tiny-yolo.cfg” (it is wise to keep the original in case we need to revert back to the original configurations) (see appendix item 15). In the cases for trial 1 and trial 2 where I wanted to transfer-learn the model to learn only one class, I renamed the file “tiny-yolo-1c.cfg” (tiny-yolo-3c.cfg for trial 3). This copy file was edited, and the original file was preserved. There are only two lines that need to be changed in the file: line 114 and line 120.

![tiny-yolo-1c.cfg](image)

Line 120 indicates the number of classes, so I simply change the classes to classes = 1. Line 114 is the number of filters in the final convolutional layer. The appropriate

**Figure 23: Line 114 and line 120 in the tiny-yolo-1c.cfg file.**

Line 120 indicates the number of classes, so I simply change the classes to classes =
number of filters corresponds to the formula: \( \text{filters} = (\text{classes} + 5) \times 5 \). Therefore, if I wanted to train the model to learn is one class, the number is changed to \( \text{filters} = 30 \) (\( \text{filters} = 40 \) for three classes in trial 3). The last step needed to be done before I began training was to tell the algorithm the name of my class/classes. To do this, I opened the file in the main darkflow directory “labels.txt”, erased any class name that previously existed, and entered the name of my class/classes (see appendix item 16). For trials 1 and 2, the label for the one class was “anomaly”. Trial 3 performed classification of three classes, and the labels for those classes were “dark anomaly”, “light anomaly”, and “rock”. This will be further elaborated upon in Chapter 6. Once all of the edited files are saved and in their respective places, I then proceeded to the training process.

5.4 STEP 4: TRAINING

For my experiments, I used four NVidia GTX Titan X (Pascal) GPUs in parallel to train the model using my training dataset. Resources were provided by Florida Atlantic University’s High Performance Computing (HPC) center, located on the Jupiter campus. Training, in any AI project, is well-known to be the most time-consuming process in the entirely of a project. Using GPU’s such as these can transform a training period of hours (even days, if on common CPU’s) to a matter of minutes.

To begin the training process and create new model, I entered the below command into the Linux terminal within the main darkflow directory:

```
flow --model cfg/tiny-yolo-1c.cfg --load tiny-yolo.weights --train --annotation train/annotations --dataset train/images --gpu 1.0 --epoch 500
```

This line tells the darkflow software a few things: where the .cfg and .weights are, where the training images and annotation files are, how much GPU percentage I would like to use (1.0 being 100%), and for how many “epochs” to train for. An epoch is one
complete pass (full iteration) through the training images. While training, the
algorithm will make many passes through the training dataset, and the higher the epoch
number, the more passes the algorithm will make over the data in creating the model.

There are five parameters for training and one parameter for testing:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>Learning rate decides how fast the error goes to the global minimum error</td>
</tr>
<tr>
<td>Momentum</td>
<td>Momentum speeds up learning rate</td>
</tr>
<tr>
<td>Decay</td>
<td>Decay is parameter to avoid overfitting</td>
</tr>
<tr>
<td>Batch size</td>
<td>How many images in one batch. In training, there is always a batch of images go through network together and use average error to update the network.</td>
</tr>
<tr>
<td>Total batch</td>
<td>How many batches of images trained.</td>
</tr>
<tr>
<td>Class Threshold</td>
<td>To eliminate detections whose class scores are less than threshold.</td>
</tr>
</tbody>
</table>

*Figure 24: YOLO training parameters and descriptions that were kept consistent for all trials (except total batch size).*

Figure 35 provides a description for all the parameters function in the algorithm. For all the trials, I used the same settings with the exception of batch size (the batch size differed for each trial). The learning rate was set to 0.001, the momentum was set to 0.9, the decay was set to 0.0001, the batch size was set to 64 images, and the class threshold is 0.1. These settings (with the exception of the class threshold) are the default settings in the YOLO configuration.

Once the command is entered, the terminal will begin to produce a series of output lines to the user. The first output is a visualization of the loaded model. Figure 29 shows
the output of the tinyYOLO model, which reassures the user that models, layers, functions, and dimensions are all correct.

<table>
<thead>
<tr>
<th>Source</th>
<th>Train?</th>
<th>Layer description</th>
<th>Output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 416, 416, 3)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 208, 208, 16)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 208, 208, 32)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 104, 104, 32)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 104, 104, 64)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 52, 52, 64)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 52, 52, 128)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 26, 26, 128)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 26, 26, 256)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 13, 13, 256)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 13, 13, 512)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>maxp 2x2 p0 2</td>
<td>(?, 13, 13, 512)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 13, 13, 1024)</td>
</tr>
<tr>
<td>Load</td>
<td>Yep!</td>
<td>conv 3x3 p1 1 +bnorm leaky</td>
<td>(?, 13, 13, 1024)</td>
</tr>
<tr>
<td>Init</td>
<td>Yep!</td>
<td>conv 1x1 p0 1 linear</td>
<td>(?, 13, 13, 30)</td>
</tr>
</tbody>
</table>

Figure 25: Output of the loaded model at the start of the training process.

The next output that will be given is the loading of the .xml files:
Figure 26: Output for the loading of the .xml files needed for training.

The algorithm will load all of the .xml files individually, and once they all have been loaded, the training will finally begin.

Statistics:
  anomaly: 220
Dataset size: 149
Dataset of 149 instance(s)
Training statistics:
  Learning rate : 1e-05
  Batch size : 16
  Epoch number : 300
  Backup every : 2000
step 1  -  loss 104.307592679 - moving ave loss 104.307592679
step 2  -  loss 102.92717749 - moving ave loss 102.92717749
step 3  -  loss 102.00707592 - moving ave loss 101.99707592
step 4  -  loss 101.952705201 - moving ave loss 101.952705201
step 5  -  loss 100.696532104 - moving ave loss 100.696532104
step 6  -  loss 99.953903516 - moving ave loss 99.953903516
step 7  -  loss 99.8115246582 - moving ave loss 99.8115246582
step 8  -  loss 99.691660289 - moving ave loss 99.691660289
step 9  -  loss 98.991508998 - moving ave loss 98.991508998
Finish 1 epoch(es)
step 10 -  loss 97.2310638428 - moving ave loss 97.2310638428
step 11 -  loss 97.9695807373 - moving ave loss 97.9695807373
step 12 -  loss 96.482515747 - moving ave loss 96.482515747
step 13 -  loss 96.3598537207 - moving ave loss 96.3598537207
step 14 -  loss 95.97134826658 - moving ave loss 95.97134826658
step 15 -  loss 95.4301877441 - moving ave loss 95.4301877441

Figure 27: The output of the beginning of the training process.
Once training begins, the output will begin to provide the loss and moving average loss for every step of training within an epoch. The number of steps within an epoch differs with the amount of training data (the more training data, the more number of steps in an epoch). To get the best results, we want to train the model until the loss number reaches a minimum value (until it won’t reduce any further).

![Figure 28: The output of the end of the training process.](image)

The training process will run until it reaches its final epoch, at which point the training terminates and checkpoints for the model will be saved in the subfolder “ckpt”.

For all of my experimental trials, I found 500 epochs to be a good amount to train to. An initial experiment was conducted using 300 epochs for 149 training images, and the loss leveled at around 2.0 well before reaching the 300 epoch checkpoint. With the resources used, the time it took to fully train the model under my settings was roughly 40 minutes. During training, the algorithm produces checkpoint files for the newly created network model. For every 125 step iterations, 4 files are created: a .data file, a .meta file, a .index
file, and a .profile file (see appendix item 17). To load the created model at any created checkpoint, all 4 of these files are needed per checkpoint.

5.5 STEP 5: TESTING

After the training process has finished, we can finally test the model to see how it performs.

```python
import cv2
from darkflow.net.build import TFNet
import numpy as np
import time

options = {
    'model': 'cfg/tiny-yolo-1c.cfg',
    'load': 25000,
    'threshold': 0.1,
    'gpu': 1.0
}

tfnet = TFNet(options)
colors = [tuple(255 * np.random.rand(3)) for _ in range(10)]
cap = cv2.VideoCapture('trial_4_testvideo.mp4')
time = 0
```

*Figure 29: The configurations for testing a newly trained model in the python script "processing_video.py".*

To do this, I executed a python script called “processing_video.py” (see appendix item 2). This script allows the user to input the models’ .cfg file, the desired checkpoint of the model to test, the detection confidence level threshold, and the percentage of GPU to use to process the video. Using 100% GPU will allow for the fastest testing possible, so I set the GPU to 1.0. This script also makes use of OpenCV, which will provide an output of the confidence scores atop each bounding box made in the testing video. To call a model checkpoint, all that is needed to input is simply the number of the checkpoint. To load a testing video, the user must input the name of the video file in line 18 of the script, and the video file itself must be placed in the main darkflow directory. Once all parameters were set and all files were placed in their correct locations, I executed the python script via the Linux terminal and the testing process began.
Figure 30: Beginning of the testing process (loading of the newly created model, loading of the testing video, and outputting the frames-per-second during processing.

The output for the beginning of the testing will appear similar to the beginning of the training output – the selected model will load, showing its inner layers and dimensions. Once the model loads, the video processing will begin, and the algorithm will output the frames-per-second (FPS) of the processing. On average, the four GPUs in parallel were able to process the testing videos at around 55 FPS. Once the processing is finished, a video file named “output.avi” is created and stored in the main darkflow directory.
6. TRIALS AND RESULTS

6.1 TRIAL 1:

6.1.1 PREFACE

The purpose of trial 1 was to show that reasonably accurate ATR could be achieved by using publicly available training data. The training data for this trial was obtained from Google image search results. To gather these training images in an efficient way, I used a python script called “get_images.py” (see Appendix item 3).

```python
def save_images(links, search_name):
    directory = search_name.replace(' ', '_')
    if not os.path.isdir(directory):
        os.mkdir(directory)

    for i, link in enumerate(links):
        savepath = os.path.join(directory, '{:03}.png'.format(i))
        urllib.urlretrieve(link, savepath)

if __name__ == '__main__':
    search_name = 'sauv side scan sonar sunken ship'
    links = get_links(search_name)
    save_images(links, search_name)
```

Figure 31: Line 40 of python script "get_images.py", enabling extraction and automatic downloading of google image search results.

This script allows the user to search Google for images and save the first 99 search results into a folder in the main darkflow directory, all with the execution of the python script command. This makes data collection from Google as fast and as simple as possible, as opposed to manually scrolling through image search results and downloading them individually. To change the name of the search, the user simply edits line 40 of the
script to reflect the desired search (shown in Figure 32). For this trial, I collected data from six separate searches, all sharing the common words “auv side scan sonar”. The Google searches were:

- “auv side scan sonar anomaly detection”
- “auv side scan sonar mine detection”
- “auv side scan sonar drowned”
- “auv side scan sonar plane”
- “auv side scan sonar sunken ship”
- “auv side scan sonar target detection”

After the six searches have executed and all 99 images for each folder have downloaded, the next thing I did was compiled all the images into one folder and renamed the images numerically and sequentially (see appendix item 18). This is done using a python script called “rename.py” (see appendix item 4) (shown in Figure 33).

```python
import os

imdir = 'images'
if not os.path.isdir(imdir):
    os.mkdir(imdir)

auv_folders = [folder for folder in os.listdir('.') if 'auv' in folder]

n = 0
for folder in auv_folders:
    for imfile in os.scandir(folder):
        os.rename(imfile.path, os.path.join(imdir, '{0}.png'.format(n)))
        n += 1
```

*Figure 32: Line 7 of the python script "rename.py".*

By editing line 7 of this script, we can search the main darkflow directory for folders with a common word in their name (in this case, “auv”), and extract the images from those folders and place them in a newly created folder “auv_folders” with each image renamed incrementally. Once this script finished executing, the final step in preparing the training data for this trial was to perform a visual inspection of the images. I examined
the images visually and removed any image that I decided would not provide useful information as training data. Hence, any image that did not contain a “target” or an “anomaly”, or any image that was not a side-scan sonar image for that matter, was deleted from the folder. Although Google does have a fair amount of side-scan sonar images in their search results, it is still a rather limited form of data to access. This is the reason why after six Google searches, I stopped searching – because after many searches the results began to produce the same outputs with little-to-no differentiation.

![Figure 33: A portion of the training image dataset for trial 1.](image)

After performing a visual inspection of the data and removing undesired images, the total amount of training images for the trial was reduced to a size of 149 images.

### 6.1.2 RESULTS

The results of this trial were produced from the testing videos described in section 4.2. The first four images below show the progression of detection confidence as training proceeds. The longer I allowed the model to train for, the higher the confidence percentage was in detecting anomalies. The four images below are segments of the same
testing video at the same moment in time, only tested at four different step checkpoints of training.

Figure 34: Anomaly detection made at 64% confidence after 2700 steps of training.

(Trial 1)

Figure 35: Anomaly detection made at 77% confidence after 3500 steps of training.

(Trial 1)
Figure 36: Anomaly detection made at 87% confidence after 4500 steps of training. (Trial 1)

Figure 37: Anomaly detection made at 95% confidence (and a highest core for the trial) after 5400 steps of training (full 500 epochs). (Trial 1)

The confidence percentage increase between each of the four testing training checkpoints was roughly 5-10%. A maximum confidence score for accurate anomaly detection in trial 1 was found to be 95% at 5400 steps of training. The above side-scan sonar images show a collection of rocks on the right-hand side, and open seafloor with a single, lone unknown object on the left-hand side. In accordance with how the training
images were annotated, the lone object on the left is what would be considered a “target” or “anomaly”. Notice how only the lone object on the left was detected, and the clutter of shadowy rocks on the right were dismissed. This was the desired outcome – to discriminate between a clutter of natural objects (such as rocks) and an object that sticks out like a sore thumb. The image below shows the algorithms ability to detect multiple shadowy anomalies simultaneously.

![Multiple anomalies found simultaneously at confidences well above 50% (full 500 epochs).](image)

*Figure 38: Multiple anomalies found simultaneously at confidences well above 50% (full 500 epochs). (Trial 1)*

This image (in contrast with the results before it) also shows that the algorithm has no problem making detections for anomalies casting shadows falling to either the left or the right. This is mostly due to the variation on training data – the training images varied in target orientation and side of the vehicle which it lays. Had this not been the case, detections would likely only be made on anomalies that cast a shadow to one direction only.
Figure 39: In a clutter of potential targets (or “area of pattern”), the target that casts the largest shadow (or the most drastic change in features) will dominate (full 500 epochs).

(Trial 1)

Figure 46 shows a case where detection was made in an area of pattern (within a clutter of rocks along the seabed). This was not desired, however what this result tells us is that the section of the clutter which has the largest shadow (or the most drastic change in features) will take precedence over the surroundings. Therefore, a bounding box was made around the part of the clutter which stuck out the most from its surroundings.

The two images below show examples of where trial 1 fell short. Trial 1 did not yield perfect results, and this is likely due to two factors: the differences between training data and testing data, and the amount of training data itself.
Figure 40: One anomaly detected, while others not. Likely due to limited training data (full 500 epochs). (Trial 1)

Figure 41: Multiple anomalies made, but some anomalies ignored. Likely due to limited training data (full 500 epochs). (Trial 1)

Figures 41 and 42 show examples where the model tested in trial 1 was able to make detections of some of the anomalies in the image, but not all. This can be solved by having a larger set of training images that have similarities to the images tested – the more similar training image you have to your testing set, the better potential your model will have to learn those particular features and accurately detect them.
Figure 42: An example of a false alarm detection made (right) at 61%. (Trial 1)

Figure 43 shows an example where a false alarm detection was made in trial 1. Although the anomaly detection made on the right of the AUV is correct (at 91% confidence), there is an anomaly detection incorrectly made within the clutter of rocks on the right of the AUV (at 61% confidence). False alarms are something we want to avoid, and with more homogeneous training data and precise annotations, false alarms like these can be reduced.

6.2 TRIAL 2:

6.2.1 PREFACE

The purpose of trial 2 was to improve upon the results of trial 1 by using testing data that is more closely related to the training dataset. The training data used in trial 1 was very different from the testing data; variations/inconsistencies include sensor models, sampling frequencies, image resolution, target shape and size, time and location of data acquisition, water conditions, and target types themselves. The training images used in trial 1 had targets in them ranging from sunken ships to lost aircrafts to potential mines. The aim of this experiment was to show that if a user desires to locate a very specific type
of target and has a set of training images which contain very similar targets within them, then a small amount of training data could be used to produce an even more accurate ATR system for an AUV.

![Figure 43: A portion of the training image dataset for trial 2.](image)

The training images for this trial were collected by extracting images from 15 out of 16 of the “Targets” videos described in section 4.2. One video was preserved for testing. The partitioning of the training and testing data was done this way to maximize the amount of features learned during training the model. Out of the 15 videos, a total of 88 images were saved for training data. In addition, the data recorded from the mission run on March 19th, 2018 was also used as a testing video.
Figure 44: An example of a valuable training image with an anomaly (on the right) for trial 2.

Figure 45 shows an example of an image that would be useful as a training image for this trial. The seafloor is open and ubiquitous, and there is an unknown object (anomaly) casting a shadow on the right-hand side of the image. Anytime a scenario like this would happen in one of the videos, an image of the moment was captured and saved to be used as a training image. All 88 training images for this trial contained a target similar to that of the image in Figure 44.

6.2.2 RESULTS

The first five images below show the progression of detection confidence as training proceeds. The longer I allowed the model to train for, the higher the confidence percentage was in detecting anomalies. The five images below are segments of the same testing video at the same moment in time, only tested at five different step checkpoints of training. Since the training dataset of trial 2 was smaller than that of trial 1, there were less training steps in between epochs for trial 2. For this trial, the number of steps reached at the 500th epoch was 2500 steps.
At the early stage of training (at step number 500), partial anomaly detection was made. The anomaly on the right-hand side is not yet detected, however the detected anomaly was found with reasonably high confidence at 80%.

From 500 to 1000 steps of training, the detection on the left-hand anomaly was increased 14% in confidence.
At step 1500, detection of both anomalies was made (with 74% confidence on the left anomaly and 83% confidence of the right anomaly). The confidence of the initial anomaly detection has dropped, but both confidences are reasonably high at this stage.

By step 2000, the confidence scores of both detected anomalies were increased (93% confidence on the left anomaly and 86% confidence on the right anomaly).
Figure 49: Both anomalies detected at 98% confidence (left) and 91% confidence (right) after 2500 steps (full 500 epochs) of training. (Trial 2)

At the 500-epoch mark (2500 steps of training), both anomalies were detected with confidence scores above 90% (98% confidence on the left anomaly and 91% confidence on the right anomaly). The highest score recorded for trial 2, a detection found at 99% confidence, is shown in the image below.

Figure 50: Highest anomaly detection confidence score recorded at 99% (left) after 500 epochs of training. (Trial 2)
The image below shows a case where on the right of the AUV there is a “dark-shadowed” anomaly and on the left of the AUV there are two “light-shadowed” anomalies.

Figure 51: 2 out of 3 anomalies detected (upper-left corner anomaly neglected). (Trial 2)

The light-shadowed anomaly in the upper left corner of the image was not detected initially - however, after one second later in the video, all three anomalies are detected (shown in figure 53).

Figure 52: 3 out of 3 anomalies detected moments later, but at the cost of confidence (500 epochs of training). (Trial 2)
Figure 53: Anomaly detection made at 20% of a sunken tire, within an area of pattern (from data collected on March 19th, 2018). (Trial 2)

Figure 54 shows a result taken from the testing video of the data captured on March 19th, 2018. The seafloor on this date was different from the seafloor in the data from the previous missions in that the seafloor was lined with sand ripples. This can be considered an area of pattern and can make object detection difficult due to the increased number of shadows produced in an image by the received sonar signals. Figure 53 shows the AUV passing along a sunken tire with an area of pattern. Although anomaly detection on the tire was made at a low confidence score (20 %), this model was successfully able to discriminate between the target and the surrounding area of pattern with accuracy. An observation to be made from this result is that in an area of pattern (where multiple acoustic shadows are present), the model will detect with a lower confidence than a setting of flat, open seafloor.
6.3 TRIAL 3:

6.3.1 PREFACE

The purpose of trial 3 was to demonstrate YOLO’s ability to be repurposed for custom classification of various objects regarding the problem of ATR in side-scan sonar imagery. While trial 1 and trial 2 showed detection of one class, trial 3 aimed to classify between three different classes. Based on the image data used in this research, the three classes were chosen to be “dark anomaly”, “light anomaly” and “rock”. Consider the image below:

![Figure 54: Examples of what would be considered a "light anomaly" (left) and "dark anomaly" (right). (Trial 3)](image)

In this example, on the right of the AUV there is an anomaly which casts a clear dark shadow, and on the left of the AUV there is an anomaly which casts a shadow that is considerably lighter. There could be many reasons for the difference in acoustic shadow production, but for the sake of the experiment, let’s assume that only anomalies which cast a dark shadow are of value to us. If a research team only wished to identify and target anomalies with dark shadows and disregard light-shadowed anomalies, then it would be best to have an ATR system that could differentiate between the two, so that an
AUV could then autonomously make decisions based on the classification. The image below shows an example (from the testing dataset) of a common scene found in the majority of the REMUS-100 data recorded at FAU:

![Image of AUV scene](image)

*Figure 55: Two "light anomalies" (left) and a clutter of rocks on the right of the AUV. (Trial 3)*

On the right of the AUV there is a clutter of rocks along the seabed and on the left of the AUV are two examples of “light anomalies” laying in the open seafloor. This rocky terrain produces numerous shadows, however if a research team knows that the object they wish to target lays in the smooth, open seafloor, then for the ATR system to detect the shadows produced by the rock clutters and identify those shadows as “anomalies” would be erroneous. Therefore, it would be ideal to have a system that differentiates between the rocky terrain and the clear anomalies that stick out amongst the seafloor.

346 training images were used in total - 173 images for “dark anomaly”, 102 images for “light anomaly”, and 71 images for “rock”. The training images gathered for this trial were a combination of the images from trials 1 and 2. The annotation process was done for one class at time, and line 18 of the “draw_box.py” python script was edited in between labeling each class so that the annotations would be named correctly. After fully
annotating the three classes, all 346 image and annotation files were compiled into one
folder each and renamed numerically and incrementally. Lastly, the image and annotation
folders were relocated to the “train” folder described in section 5.5.2.

6.3.2 RESULTS

The results in this trial were produced by testing on the same testing video that was
reserved and used in trial 2.

Figure 56: Classification made for dark anomaly (left) at 91% confidence and the clutter
of rocks (right). (Trial 3)

Figure 57: Classification made for light anomalies (left) at 39% and 25% confidence,
and the clutter of rocks (right). (Trial 3)
Figures 57 and 58 both show examples where an anomaly was detected (on the left of the AUV) and the classifier demonstrates its ability to differentiate between the anomaly and the clutter of rocks (area of pattern) on the right of the AUV. Figure 57 shows the same segment of data shown in Figure 56, after tested upon. The results in these figures show the models ability to classify between the three custom classes accurately, according to how the training data was annotated. Although the confidence scores are considerably lower than those of trial 1 and trial 2, the classification and detections made are accurate. If a segment of an area of pattern (in this case, the clutter of rocks) strays too far from the clutter, then the model may classify the strayed segment as an anomaly and not part of the area of pattern. An example of this is shown below in Figure 58:

*Figure 58: Dark anomaly detected (right) at 28% on a segment of the area of pattern that has strayed far from the clutter. (Trial 3)*

This moment in the testing video was captured moments after the image in Figure 57. A pair of rocks belonging to the clutter on the right of the AUV is separated, and the classifier identifies the segment to be “dark anomalies” at 28% confidence. This is largely due to how the training data was annotated – the model was trained to learn that any
unknown shadowy object that sits alone on the seafloor is a target of interest and was therefore given the label of dark or light anomaly.

![Classification made for light anomaly and dark anomaly (left) at 15% and 11%, respectively. (Trial 3)](image)

Figure 59: Classification made for light anomaly and dark anomaly (left) at 15% and 11%, respectively. (Trial 3)

Figure 60 shows a moment in the testing video where the model classifies between a light anomaly and a dark anomaly amongst open and smooth seafloor. The light anomaly in this case is shadow-less – in other words, a highlight is produced but not a corresponding shadow. The cause of this shadow-less light anomaly is unknown; however, it is certainly an anomaly and could be of interest to the user. Nonetheless, the model makes a detection at the largest highlight and differentiates the anomalies from the one that casts a dark shadow.
7. CONCLUSION

The capability of a state-of-the-art deep learning object detection algorithm (YOLO) to be re-purposed for detection of seafloor anomalies in AUV side-scan sonar imagery by learning pixel-intensity based features is investigated. Side-scan sonar image data captured by AUV’s (both public domain and proprietary) are collected and annotated in preparation for use in creating convolutional neural network models. Four GPU’s in parallel are used to carry out the computations needed to learn the target features in the annotated training images, and different models are created to be tested on unlearned side-scan sonar data. The testing video results show the YOLO algorithm to yield high confidence in anomaly detection using relatively low sizes of training datasets.

For future work, it would be of benefit to design a convolutional neural network from scratch specifically tailored to the problem of automatic target recognition in side-scan sonar imagery. Transfer learning a pre-trained CNN model provides users with a simple and speedy method to creating a model for the problem and is certainly a good start, but to build a model from the ground-up designed for the desired task would be the ideal scenario. This method was a supervised learning algorithm, however unsupervised learning is the preferred method for it would be an ideal case that the computer would automatically know the features of interest in each training image without the need for a user to manually label/annotate the features. It is also important to note that the features learned during training in this process were based solely on pixel-intensity. No other
information was provided to the algorithm regarding the anomalies in the training images.

One thing that was not done in this work was “k-fold cross validation”. K-fold cross validation is a way to validate that there is not a bias in the choice of testing data when partitioning a dataset into a training set and testing set. In other words, it is important to make sure that your choice of data to test on is justified, or that similar results may have been produced if you choose to test on a different portion of the net data. This would apply most to the case of trial 2, where the training and testing data came from the same batch. The video that was used for testing in trial 2 was chosen at random, and for future recommendations, it would be wise to perform a k-fold cross validation to ensure that the detection accuracies are similar throughout different testing video choices.

Another recommendation for future research would be to have a large set of training images containing specific examples of desired targets to detect would also be the ideal case. Regarding data size, many studies have supported the rule of thumb that ML and DL performance is improved by using a larger and more diverse training set. [36] [37] [38] If the user wishes to detect a very specific target shape - such as a sunken car, for example – then it would be ideal to have a large set of training images which contain targets that cast shadows like that of a sunken car. This could be done by going to known sites of a sunken cars and running multiple (tens or hundreds) of AUV missions in order to have a sufficient amount of training data similar to the task at hand. The closer related the training data is to the testing data, the greater the detection accuracy will be.

The research conducted in this thesis was done in the post-process. Meaning, the results gathered were found by testing on pre-recorded AUV missions and not on live
data. To implement this algorithm onto an AUV in real-time could have numerous extensions. An AUV equipped with live ATR capabilities could detect a potential target and then send a signal back to the user aboard a watercraft, notifying the personnel on board that the vehicle has made a detection. Another application could be multi-AUV cooperative missions; for example, an AUV equipped with ATR using side-scan sonars could detect an anomaly and then send an acoustic signal to a cooperative AUV equipped with a magnetometer to investigate the estimated location of the anomaly and search for ferro-magnetic properties along the seabed. Future researchers are encouraged to attempt live/real-time implementation for applications such as these. Although the research conducted in this thesis was done in the post-process, the results from this research support that deep learning provides a robust alternative for users to create anomaly detectors of their own and incorporate them into target recognition dependent AUV missions.
APPENDICES
APPENDIX A: PYTHON SCRIPTS

1. Python Script: “draw_box.py”

   # originally written by Mark Jay and edited by Dylan Einsidler

   import os
   import matplotlib.pyplot as plt
   import cv2
   from matplotlib.widgets import RectangleSelector
   from generate_xml import write_xml

   # global constants
   img = None
   tl_list = []
   br_list = []
   object_list = []

   # constants
   image_folder = 'images'
   savedir = 'annotations'
   obj = 'anomaly'

   def line_select_callback(clk, rls):
       global tl_list
       global br_list
       global object_list
       tl_list.append((int(clk.xdata), int(clk.ydata)))
       br_list.append((int(rls.xdata), int(rls.ydata)))
       object_list.append(obj)

   def onkeypress(event):
       global object_list
       global tl_list
       global br_list
       global img
       if event.key == 'q':
           print(object_list)
           write_xml(image_folder, img, object_list, tl_list, br_list, savedir)
           tl_list = []
           br_list = []
           object_list = []
           img = None
           plt.close()

   def toggle_selector(event):
       toggle_selector.RS.set_active(True)
if __name__ == '__main__':
    for n, image_file in enumerate(os.scandir(image_folder)):
        img = image_file
        fig, ax = plt.subplots(1)
        mngr = plt.get_current_fig_manager()
        mngr.window.setGeometry(250, 120, 1280, 1024)
        image = cv2.imread(image_file.path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        ax.imshow(image)
        toggle_selector.RS = RectangleSelector(
            ax, line_select_callback,
            drawtype='box', useblit=True,
            button=[1], minspanx=5, minspany=5,
            spancoords='pixels', interactive=True
        )
        bbox = plt.connect('key_press_event', toggle_selector)
        key = plt.connect('key_press_event', onkeypress)
        plt.show()

2. Python Script: “processing_video.py”

# originally written by Mark Jay and edited by Dylan Einsidler

import cv2
from darkflow.net.build import TFNet
import numpy as np
import time

options = {
    # 'model': 'cfg/tiny-yolo-1c.cfg',
    # 'load': 2500,
    # 'threshold': 0.9,
    # 'gpu': 1.0
}

tfnet = TFNet(options)
colors = [tuple(255 * np.random.rand(3)) for _ in range(10)]

cap = cv2.VideoCapture('trial_4_testvideo.mp4')
size = (int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
        int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
)
codec = cv2.VideoWriter_fourcc(*'DIVX')
out = cv2.VideoWriter('output.avi', codec, 60.0, size)
while (cap.isOpened()):
    stime = time.time()
    ret, frame = cap.read()
    results = tfnet.return_predict(frame)
    if ret == True:
        for color, result in zip(colors, results):
            tl = (result['topleft']['x'], result['topleft']['y'])
            br = (result['bottomright']['x'], result['bottomright']['y'])
            label = result['label']
            confidence = result['confidence']
            text = '{}: {:.0f}%'.format(label, confidence * 100)
            frame = cv2.rectangle(frame, tl, br, color, 5)
            frame = cv2.putText(
                frame, text, tl, cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2)
            out.write(frame)
            # cv2.imshow('output.avi', frame)
            print('FPS {:.1f}'.format(1 / (time.time() - stime)))
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

    cap.release()
    out.release()
    cv2.destroyAllWindows()

3. **Python Script: “get_images.py”**

# originally written by Mark Jay and edited by Dylan Einsidler

import os
import urllib.request as ulib
from bs4 import BeautifulSoup as Soup
import json
url_a = 'https://www.google.com/search?ei=1m7NWePfFYaGmQG51q7IBg&hl=en&q={}'
url_b = '
&tbm=isch&ved=0ahUKEwjovnD7sjWAhUGQyYKHTmrC2kQuT0l7gEoAQ&start={}'
url_c = '
&yv=2&vet=10ahUKEwjovnD7sjWAhUGQyYKHTmrC2kQuT0l7gEoAQ.1m7NWePfFYaGmQG51q7IBg'
url_d = '\.i&ijn=1&asearch=ichunk&async=_id:rg_s,_pms:s'
url_base = '\.join((url_a, url_b, url_c, url_d))

headers = {'User-Agent': 'Chrome/41.0.2228.0 Safari/537.36'}

def get_links(search_name):
    search_name = search_name.replace(' ', '+')
    url = url_base.format(search_name, 0)
request = ulib.Request(url, None, headers)
json_string = ulib.urlopen(request).read()
page = json.loads(json_string)
new_soup = Soup(page[1][1], 'lxml')
images = new_soup.find_all('img')
links = [image['src'] for image in images]
return links

def save_images(links, search_name):
    directory = search_name.replace(' ', '_')
    if not os.path.isdir(directory):
        os.mkdir(directory)

    for i, link in enumerate(links):
        savepath = os.path.join(directory, '{:06}.png'.format(i))
        ulib.urlretrieve(link, savepath)

if __name__ == '__main__':
    search_name = 'auv side scan sonar sunken ship'
    links = get_links(search_name)
    save_images(links, search_name)

4. Python Script: **“rename.py”**

# originally written by Mark Jay and edited by Dylan Einsidler

import os

imdir = 'images'
if not os.path.isdir(imdir):
    os.mkdir(imdir)

auv_folders = [folder for folder in os.listdir('.') if 'auv' in folder]

n = 0
for folder in auv_folders:
    for imfile in os.scandir(folder):
        os.rename(imfile.path, os.path.join(imdir, '{:06}.png'.format(n)))
    n += 1
APPENDIX B: ADDITIONAL FIGURES

5. Site of operation for all AUV missions from SeaTech:

6. Predefined trajectory of the AUV missions.

7. Myself and fellow graduate student colleagues deploying the Hydroid REMUS-100 along the latitude of Las Olas Blvd, Fort Lauderdale on March 19th, 2018.
8. The raw REMUS-100 side-scan sonar data files.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 8:52 PM</td>
<td>Side Scan Mission</td>
<td>10,878 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 3:06 PM</td>
<td>Side Scan Survey File</td>
<td>1 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>3,105 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 10:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
<tr>
<td>REMUS_MERGE_082010</td>
<td>3/20/2018 1:04 AM</td>
<td>Side Scan Survey File</td>
<td>2,954 KB</td>
</tr>
</tbody>
</table>

9. The original AUV missions converted to .mp4 video files.
10. The final edited video clips of the AUV missions, where the AUV passed across a potential target.

11. "Darknet" - the original framework for the YOLO object detection system.

[29]
12. The main darkflow directory folder.

![Diagram of the main darkflow directory folder]

13. All training images compiled into one folder called "images".

![Diagram showing the directory structure with "images" folder]

14. The created .xml files, containing feature coordinates for the corresponding training images.

![List of .xml files]

15. Creating a copy of the tinyYOLO .cfg file.

16. Creating the labels.txt file, enabling the network to label the detected anomalies.

17. The checkpoint files (in the subfolder “ckpt”) for the newly created network models, created at step iterations of 125.
18. The folders from each of the six Google image searches of Trial 1.

19. The different versions of the YOLO model. [29]
REFERENCES


[8] BVLC AlexNet Model.


[25] https://anaconda.org/anaconda/python

[26] https://www.tensorflow.org/

[27] https://opencv.org/

[28] https://github.com/thtrieu/darkflow


[31] https://www.marinesonic.com/


[33] https://www.apple.com/imovie/

[34] https://github.com/markjay4k/YOLO-series


[43] https://mlblr.com/includes/mlai/index.html#scary-loss-function