Development of an Object Detection Application that Monitors Vessel Traffic, using Neural Networks

Nikolaos Chatzistamatiou

Diploma Thesis

School of Naval Architecture and Marine Engineering
National Technical University of Athens

Supervisor: Assistant Prof. G. Papalambrou

Committee Members:

Prof. G. Grigoropoulos
Associate Prof. C. Papadopoulos

July 2021

This page is intentionally left blank.

Acknowledgements

This thesis concludes my years of studies at the School of Naval Architecture and Marine Engineering of the National Technical University of Athens. It has been, without a doubt, a great academic journey, where i gained great knowledge in multiple engineering fields and further developed my soft skills.

It was, also, an opportunity for me to practice computer programming and machine learning, which I personally enjoy as a hobby. Reaching the point of training an object detection model, certainly was not easy. Throughout the past year, I learned a new programming language and developed several machine learning and deep learning projects that provided me with the necessary knowledge for this thesis's goals.

I owe my greatest appreciation to my thesis supervisor, Assistant Professor George Papalambrou, for giving me the opportunity to work on this project and I would like to thank him for his patience and continuous support, throughout this COVID-19 era.

I would also like to thank Mr. Vasilis Mentogiannis (UFR Team) and Mr. Kostas Katsioulis (NGUE) for providing us with the vessel pictures that were used in the training process of our model and Elli Tsakopoulou for her support and her assistance in the laboring process of labeling our dataset.

Finally, I would like to dedicate this thesis to my late dog Lady, that kept me company every time I was getting frustrated over my non-working codes and to my mother, who showed immense courage and won the most difficult battle this past year. I express my sincere gratitude to my whole family and friends, for the unceasing encouragement, support and motivation throughout my studies.

Abstract

This thesis investigates the implementation of an object detection application using neural networks to monitor vessel traffic over the Peristera shipwreck. Our study's goal is to provide a monitoring system cheaper, but as robust as the traditional solutions, ensuring the safe operation of the underwater shipwreck museum and the preservation of its national treasure.

The algorithm receives video stream from a pre-installed remote based land camera on Peristera island, applies machine learning and visualizes the results on real time. For that purpose, experiments are conducted, with a variety of images and videos, in order to choose an efficient and fast object detection model, resulting to YOLOv3. This model is then used in two different implementations.

In the first implementation, the algorithm uses standard YOLOv3 pre-trained on COCO Dataset to perform vessel traffic monitoring, delivering quite good results and frame rate of 0.8 FPS. Although COCO Dataset contains a "boat" class, it is not specialized in ship detection, resulting in excessive computations for other classes and insufficient performance under not ideal lighting conditions. Moreover, it does not provide any information about the specific type of the detected vessel.

This leads to the second implementation of performing transfer learning to YOLOv3 with a small dataset of vessel images and videos captured by the land camera on Peristera. The whole procedure of creating, labeling and pre-processing our dataset, along with the creation of the code pipeline and a custom loss function that are used for training the new model, consist a challenging aspect that is described thoroughly. Although the loss value is decreasing after each epoch, suggesting that the model is "learning", the training phase could not be completed due to a memory leak in the code, that crashes the program after a certain number of iterations.

Περίληψη

Η παρούσα διπλωματική εργασία διερευνά την ανάπτυξη μιας εφαρμογής Αναγνώρισης Αντικειμένων, μέσω της χρήσης Νευρονικών Δικτύων ώστε να ελέγχεται η κίνηση των θαλάσσιων σκαφών που διέρχονται πάνω από το Ναυάγιο της Περιστέρας. Βασικός στόχος της έρευνας αποτελεί η εύρεση ενός εναλλακτικού συστήματος που είναι ταυτόχρονα οικονομικό αλλά και εξίσου αξιόπιστο με τα παραδοσιακά συστήματα παρακολούθησης, έτσι ώστε να εξασφαλίζεται η ομαλή λειτουργία του Υποβρύχιου μουσείου και η συντήρηση του θησαυρού που αυτό φιλοξενεί.

Ο αλγόριθμος που δημιουργήθηκε λαμβάνει τη ροή βίντεο από την κάμερα που βρίσκεται εγκατεστημένη στο νησί της Περιστέρας, και με χρήση τεχνικών Machine Learning οπτικοποιεί τα αποτελέσματα, σε πραγματικό χρόνο. Για να επιτευχθεί αυτή η διαδικασία, πραγματοποιήθηκαν πειράματα με ένα πλήθος εικόνων και βίντεο, ώστε να επιλεγεί εν τέλει το YOLOv3 ως το ταχύτερο και αποτελεσματικότερο μοντέλο για την αναγνώριση αντικειμένων. Το μοντέλο αυτό, χρησιμοποιήθηκε στη συνέχεια σε δύο διαφορετικές εφαρμογές.

Στην πρώτη εφαρμογή, ο αλγόριθμος χρησιμοποιεί standard YOLOv3 ήδη εκπαιδευμένο σε βάση δεδομένων COCO, ώστε να παρακολουθήσει την κίνηση των θαλάσσιων σκαφών, παράγοντας σχετικά επιτυχή αποτελέσματα με ταχύτητα ανανέωσης frame 0.8 FPS. Αν και η βάση δεδομένων COCO περιλαμβάνει μια κλάση η οποία ονομάζεται “boat”, καθώς δεν ειδικεύεται στην αναγνώριση πλεούμενων σκαφών, το μοντέλο καταλήγει να πραγματοποιεί περίσσιους υπολογισμούς για άλλες κλάσεις, ενώ δεν παράγει επαρκή αποτελέσματα όταν οι συνθήκες φωτός δεν είναι ιδανικές. Επιπλέον, δεν παρέχει πληροφορίες που αφορούν τον συγκεκριμένο τύπο του σκάφους που εντοπίζει.

Έτσι, οδηγούμαστε στη δεύτερη εφαρμογή, στην οποία πραγματοποιείται transfer learning στο YOLOv3, με τη χρήση μιας μικρής βάσης δεδομένων που αποτελείται από εικόνες και βίντεο σκαφών, τραβηγμένα από την εγκατεστημένη κάμερα. Η διαδικασία της δημιουργίας, του labeling και της επεξεργασίας της βάσης δεδομένων, καθώς και η σχεδίαση του κώδικα και μιας νέας προσαρμοσμένης συνάρτησης απόκλισης, ώστε να μπορέσει να εκπαιδευτεί το μοντέλο, αναλύονται διεξοδικά στα επόμενα κεφάλαια, καθώς αποτέλεσαν μια ισχυρή πρόκληση, καθ’ όλη τη διάρκεια της έρευνας. Η εκπαίδευση του προγράμματος συνάντησε ωστόσο ένα εμπόδιο και δεν μπόρεσε να ολοκληρωθεί: Παρόλο που παρατηρήθηκε ότι η τιμή της απόκλισης μειώνεται έπειτα από κάθε epoch – κάτι που υποδεικνύει ότι το πρόγραμμα εκπαιδεύεται – μια διαρροή μνήμης στον κώδικα αδρανοποιεί το πρόγραμμα έπειτα από ένα χρονικό διάστημα.

Abbreviations

ADAM Adaptive Moment Estimation.

AI Artificial Intelligence.

CNN Convolutional Neural Network.

CONV Convolutional Layer.

CV Computer Vision.

DL Deep Learning.

FC Fully-Connected Layer.

FPS Frames Per Second.

ML Machine Learning.

NN Neural Network.

NOUS Undersea Vision Surveillance System.

OS Operating System.

POOL Pooling Layer.

ReLU Rectified Linear Unit.

RGB Red Green Blue.

SGD Stochastic Gradient Descent.

SSD Single Shot MultiBox Detector.

YOLO You Only Look Once.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes briefly the scope of the investigation. It also provides useful information about the underwater museum in Peristera and states the dangers that create a need for a monitoring system at the selected area. Finally, it lists the advantages of our selected method and describes briefly the thesis outline.

1.1 Peristera Shipwreck

Figure 1.1: The Peristera shipwreck underwater museum.[1]

Peristera shipwreck was a large commercial vessel that sank around 425 BC while carrying thousands of wine amphorae. Today it lies at a depth of 25 meters (80 feet) near the Greek island of Alonissos, right off the islet of Peristera, in the Aegean sea. Besides the site's scenic value, the wreck shifted our understanding of the ancient world's shipbuilding techniques. Marine archaeologists revealed a significant discovery in terms of the technology

used to construct the ship. Before discovering the shipwreck, the size of this barge was thought to have been introduced by the Romans 3 centuries after the final voyage of the Peristera ship. The size of the wreckage proves that Greek shipbuilding technologies were ahead of their time. It is thought to be one of the most significant shipwrecks ever found in the Mediterranean Sea, and it can inform us about aspects of history otherwise not known. The archaeological excavation is still ongoing, and experts still have a lot to learn from the shipwreck of Peristera. Since august 2020, the site is accessible to recreational divers, making it Greece's first underwater museum.[12]

The safe operation of the museum and the preservation of this national treasure is made possible with the successful deployment of the NOUS Undersea Vision Surveillance System. The system consists of five prototype submarine housings equipped with cameras and windshield wipers for the camera lenses. The whole operation is controlled by a number of multitasking computing units. The network of the underwater cameras ends up in a submarine hub, which is powered by a 200 meter long cable from the shore, via a purpose-built solar power station on Peristera. NOUS is equipped with Artificial Intelligence capabilities in order to distinguish, classify, associate and perceive significant differences in measurable parameters that take place and are considered of interest within the area of sea wreck. Machine Learning algorithms are implemented for intelligent information processing from images and video streams. State of art methods are used for the customization (training) of neural networks and their preparation for real time operation. Functionality has been enhanced by a weather station and a *remotely-controlled 360°camera* for sea and land monitoring. The system transfers its data via fiber optic and RF link to an internet connection, to a server and cloud.[2]

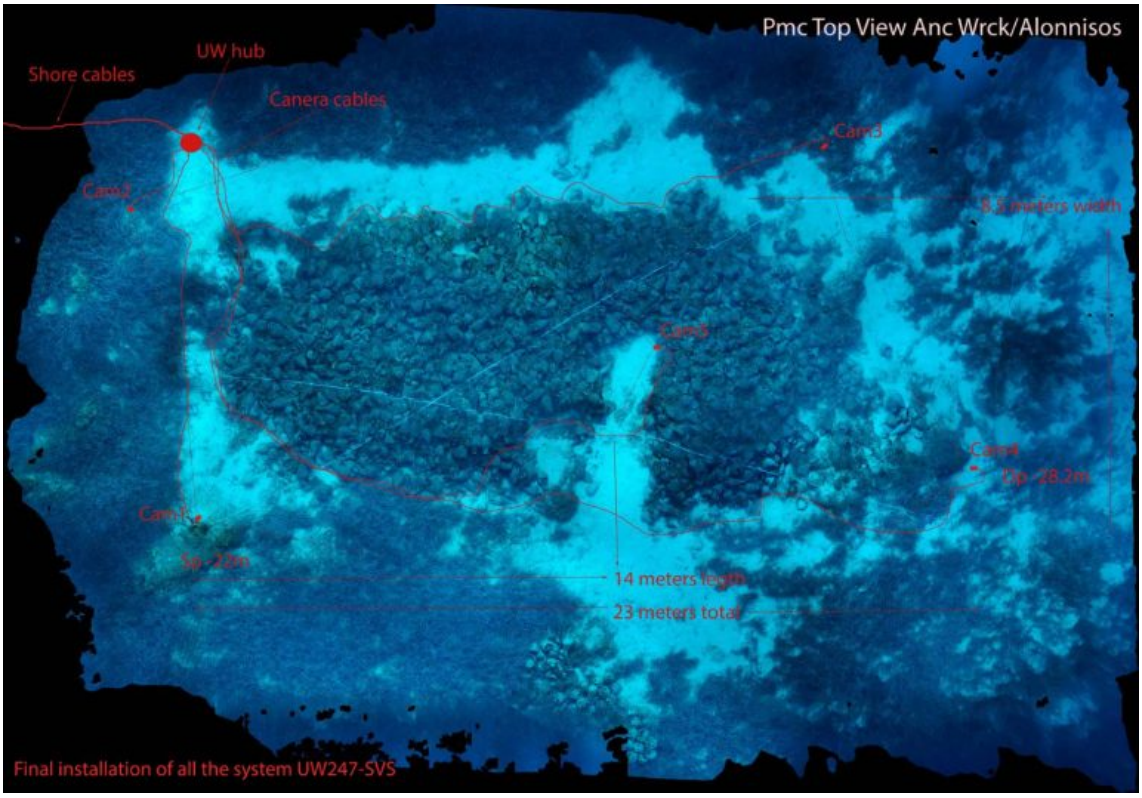


Figure 1.2: The five cameras and underwater hub of Peristera's shipwreck NOUS system.[2]

1.2 Problem Description and Motivation

The main objective of this thesis is to develop a land-based cheap but reliable monitoring system that will detect vessels passing over the Peristera shipwreck.

Statistical data in Figure 1.3 show high vessel traffic through the passageway between Alonissos and Peristera, especially considering the short distance between the two islands. Art smuggling and marine pollution are two major threats for the preservation of the ancient findings that can be tackled with proper surveillance.

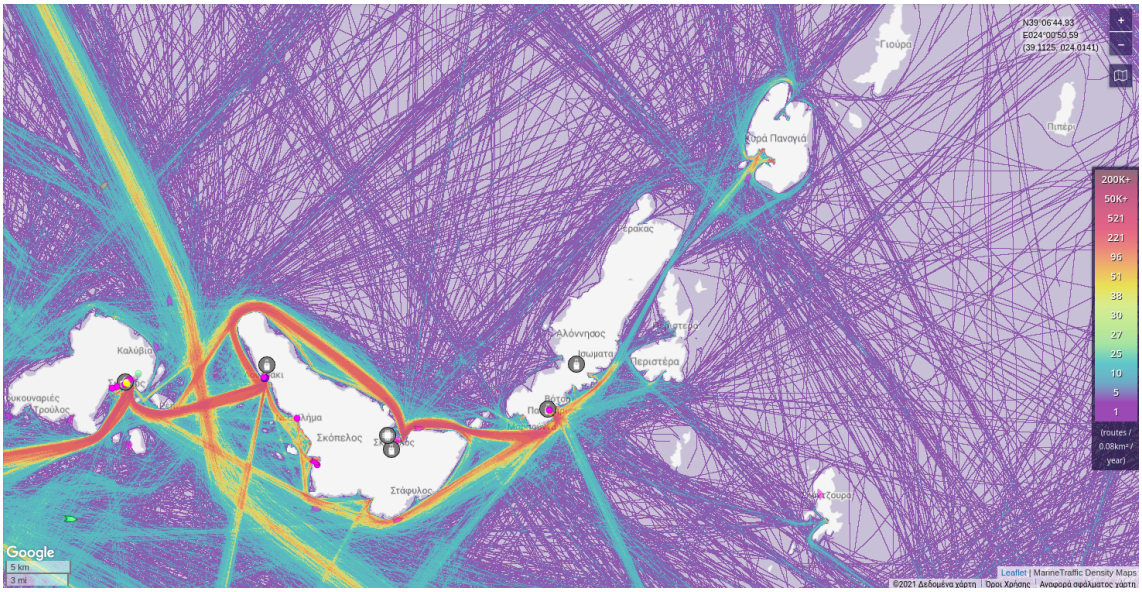


Figure 1.3: Vessel routes over the Peristera shipwreck.[2]

We decided to take advantage of the already-installed remotely-controlled 360°camera and create a vessel detection algorithm using deep neural networks. The main advantages of this solution in comparison to other monitoring systems can be listed below:

- There is no need for additional equipment.
- The development cost is essentially zero.
- Video and image sensors resemble and optimize human abilities.
- Visualizing a detection provides a direct connection between the program and the user.
- Machine learning is an adaptable and –if done correctly– extremely precise tool.

A 2020 research named *The Special Issue (SI) "Remote Sensing in Vessel Detection and Navigation"* highlights a variety of topics related to remote sensing with navigational sensors. The sequence of articles included in this Special Issue is in line with the latest scientific trends and the latest developments in science, including artificial intelligence. The authors of this issue, while comparing Video sensors with other monitoring sensors state: "Video surveillance is becoming increasingly popular in the process of detecting ships at short distances. It is particularly useful in lintel systems both on sea and inland waterways."[13] This issue includes also a number of Convolutional Neural Network models that have achieved state-of-the-art performance on detecting and classifying vessels.

Figure 1.4: The power station and the land-based remotely-controlled 360°camera on Peristera.[2]

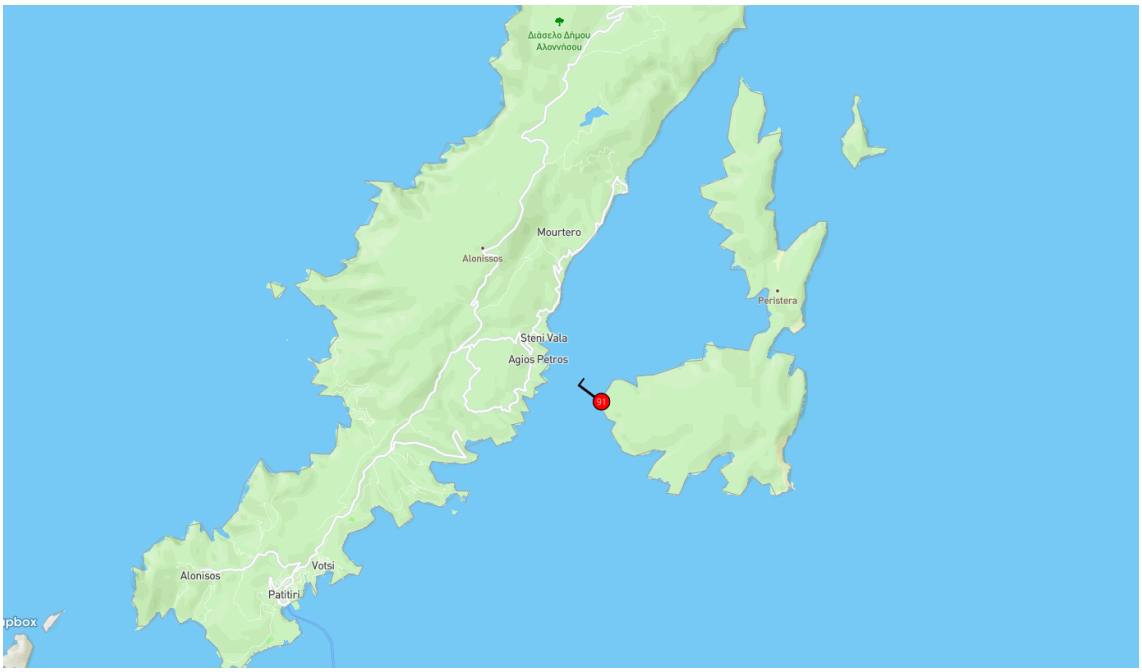


Figure 1.5: The location of the land-based remotely-controlled 360°camera on Peristera.[2]

1.3 Contributions and Engineering Challenges

Based on the predefined project goal, the contributions and Engineering Challenges of this thesis can be summarized in the following list:

- Choose the most convenient tools for developing the application.
- Perform a comparison experiment to choose an efficient and fast object detection model.
- Develop a user friendly application that performs object detection on the video stream captured by the camera on Peristera.
- Test the limits of the selected model.
- Create a vessel dataset from images and videos captured by the camera.
- Create a custom loss function that is used for model optimization.
- Develop a transfer learning pipeline code.
- Train the new model.

1.4 Thesis Outline

- **Chapter 2** introduces the basic principles of artificial intelligence, machine learning, deep learning and computer vision, analyzes the NN and CNN architectures and explains the model training phase.
- **Chapter 3** lists the tools used for the development of this thesis and presents the results of a comparison experiment between two object detection models, YOLO and SSD. In addition, YOLOv3 architecture is explained thoroughly.
- **Chapter 4** explains in depth the code pipeline that uses the pre-trained YOLOv3 model to detect ships on the Peristera video stream. It, also, presents extracted images as results that lead to important conclusions.
- **Chapter 5** analyzes the vessel dataset creation procedure, the transfer learning training performed on YOLOv3 and the loss function used for computing deviation from the ground truth labels.
- **Chapter 6** presents the concluding thoughts about this thesis's targets based on the extracted results and suggests future work for the next steps of the project.

Chapter 2

Theoretical Background

This chapter's purpose is to inform the reader about the theoretical background this thesis is based upon. The descriptions are brief and cover only terms and methods implemented in the project. For deeper comprehension, visiting the listed citations is advised. Specifically this chapter introduces the basic principles of artificial intelligence, machine learning, deep learning and computer vision, analyzes the NN and CNN architectures and explains the model training phase.

2.1 The AI Roadmap

Phrases like Artificial Intelligence, machine learning, neural networks, and deep learning mean related but different things. In this section, we will explain briefly all the above terms, as well as, analyze the Neural Network and Convolutional Neural Network structure.

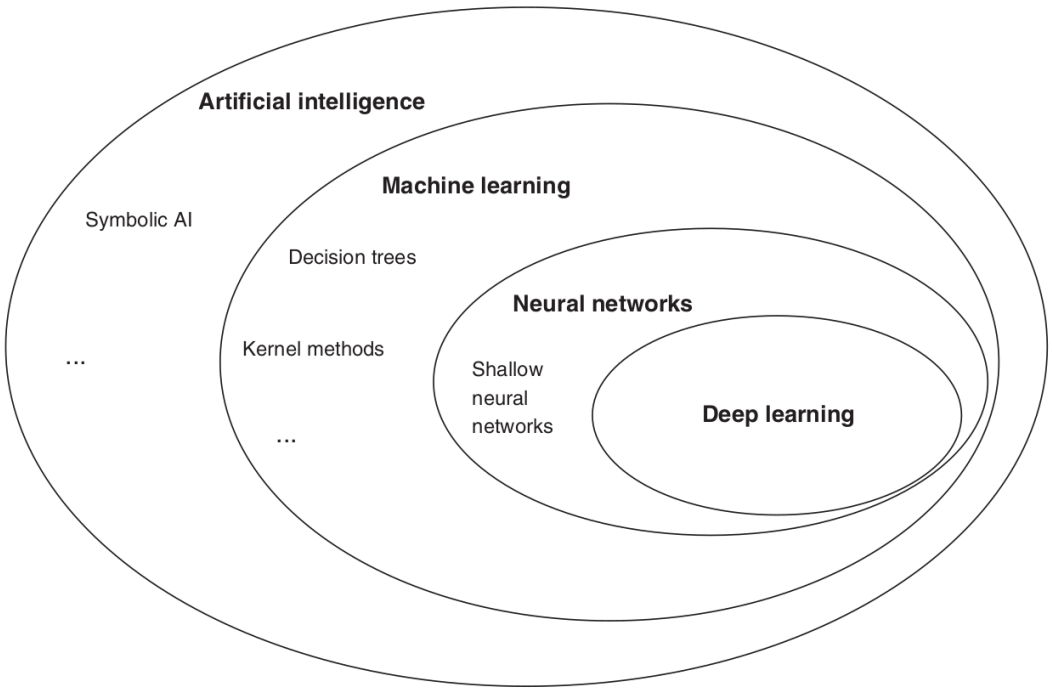


Figure 2.1: Relations between AI, machine learning, neural networks, and deep learning.[3]

2.1.1 Artificial Intelligence

Artificial Intelligence (AI) is often perceived as an extremely complicated term that is only related to computers and robots, but nothing could be further from the truth. Although some sub-fields, indeed, include computer science and robotics, AI is a broad term that can be approached by the following four phrases : "Thinking Humanly, Acting Humanly, Thinking Rationally, Acting Rationally"[14]. Combining the above approaches, we can summarize that Artificial intelligence refers to the simulation of human intelligence in machines that are programmed to think like humans and mimic their actions. The term may also be applied to any machine that exhibits traits associated with a human mind such as learning and problem solving.[15]

As the Venn diagram in Figure 2.1 shows, AI is a broad field. As such, AI encompasses machine learning, neural networks, and deep learning, but it also includes many approaches distinct from machine learning. Early chess programs, for instance, involved hard-coded rules crafted by programmers. Those didn't qualify as machine learning because the machines were programmed explicitly to solve the problems instead of being allowed to discover strategies for solving the problems by learning from the data. For a long time, many experts believed that human-level AI could be achieved through handcrafting a sufficiently large set of explicit rules for manipulating knowledge and making decisions. This approach is known as *symbolic AI*, and it was the dominant paradigm in AI from the 1950s to the late 1980s."[3]

2.1.2 Machine Learning

The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as machine learning. The introduction of machine learning allowed computers to tackle problems involving knowledge of the real world and make decisions that appear subjective.[8] Basically, *Machine Learning (ML)* is a sub-field of AI, where, a computer learns to do a specific task by each own. This learning process happens through feeding examples in data, applying machine learning and let machines discover the set of rules on their own.

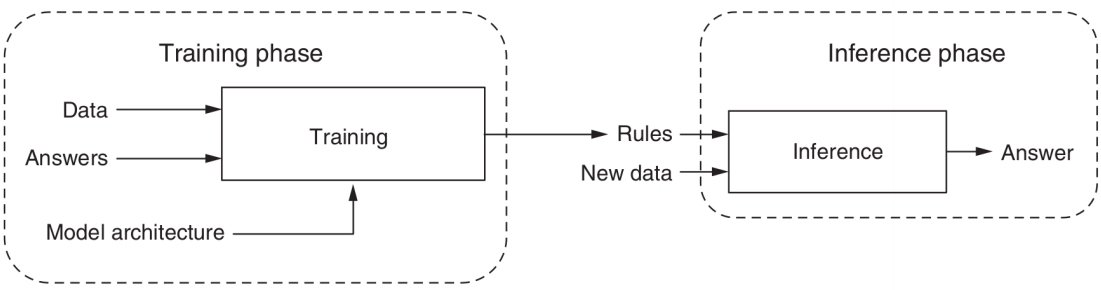


Figure 2.2: A detailed view of the Machine Learning paradigm.[3]

In figure 2.2, we take a closer look at the steps involved in machine learning. There are two important phases. The first is the *training phase*. This phase takes the data and answers, together referred to as the *training data*. Each pair of input data and the desired answer is called an *example*. With the help of the examples, the training process produces the automatically discovered rules. Although the rules are discovered automatically, they

are not discovered entirely from scratch. In other words, machine learning algorithms are not creative in coming up with rules. In particular, a human engineer provides a blueprint for the rules at the outset of training. The blueprint is encapsulated in a *model*, which forms a *hypothesis space* for the rules the machine may possibly learn.[3] The training data answers are also called *labels* and are used in order to calculate and progressively reduce the error in a model's outputs. This style of Machine Learning is called *supervised learning*. Applications in which the training data comprises examples of the input vectors along with their corresponding target vectors are known as *supervised learning problems*.[4] There are other styles of Machine Learning, that are not implemented in this thesis, such as *unsupervised learning, self-supervised learning, reinforcement learning* etc.

Once we have the trained model, we are ready to apply the learned rules on new data that the training process has never seen. This is the second phase, or *inference phase*.[3] The ability to categorize correctly new examples that differ from those used for training is known as *generalization*.

2.1.3 Neural Networks

The term *neural network* (NN) has its origins in attempts to find mathematical representations of information processing in biological systems (Widrow and Hoff, 1960; Rosenblatt, 1962; Rumelhart et al., 1986). Indeed, it has been used very broadly to cover a wide range of different models, many of which have been the subject of exaggerated claims regarding their biological plausibility. From the perspective of practical applications of pattern recognition, however, biological realism would impose entirely unnecessary constraints. Our focus is on neural networks as efficient models for statistical pattern recognition. In particular, we shall restrict our attention to the specific class of neural networks that have proven to be of greatest practical value, namely the *multi-layer perceptron*.[4] These layers are usually stacked on top of each other, with connections only between adjacent ones.

2.1.4 Anatomy of a Neural Network

As mentioned above, a Neural Network consists of a number of layers. A layer is similar to a mathematical function in that it is a mapping from an input value to an output value. However, neural network layers are different from pure mathematical functions in that they are generally *stateful*. In other words, they hold internal memory. A layer's memory is captured in its weights. *Weights* are simply a set of numerical values that belong to the layer and govern the details of how each input representation is transformed by the layer into an output representation.[3] Figure 2.3 shows a simple neural network with two layers. The input, hidden, and output variables are represented by *nodes (neurons)*, and the *weight parameters* are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. Arrows denote the direction of information flow through the network.[4]

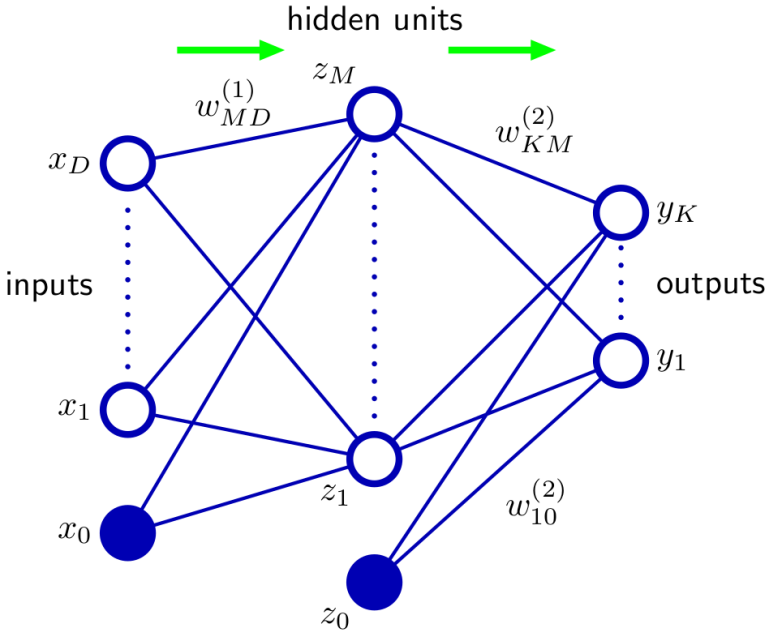


Figure 2.3: Illustration of neural network nodes and weight parameters.[4]

Activation Function

In order to understand how data get transformed through different kinds of layers, we have to isolate a node and take a look at how it functions. As shown in Figure 2.4, the weighted sum of inputs is a linear transformation. This linear value z is then passed through a non-linear function $f(z)$ which creates a new input for the next layer. This function is called *Activation Function*. Most commonly encountered activation functions are briefly analyzed in the next paragraphs.

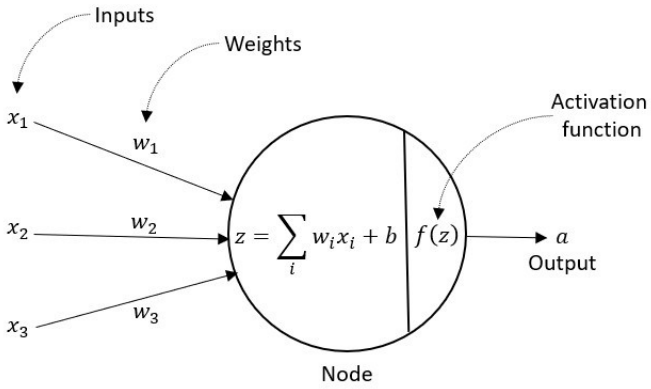


Figure 2.4: Neural Network node.[5]

As it is implied by the mathematical form shown by the equation 2.1, a *sigmoid* activation function takes a real-valued number and "squashes" it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. Although frequently used historically, nowadays, sigmoid activation function is rarely used due to two major drawbacks: sigmoids saturate and kill gradients and sigmoid outputs are not zero-centered.[6]

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

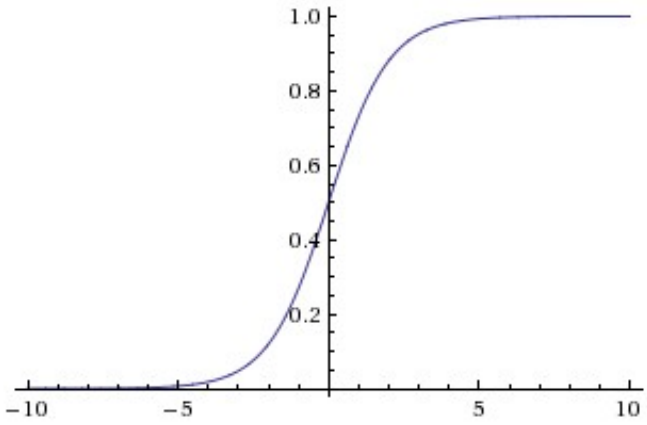


Figure 2.5: Sigmoid Activation Function.[6]

ReLU is an activation function thresholded at zero, which means, that it replaces every negative number with zero and leaves positive numbers unconverted. The above implementation can be seen in the figure 2.6. The Rectified Linear Unit has become very popular in the last few years, mainly due to the acceleration it offers during the convergence of stochastic gradient descent (see subsection 2.1.4) and the simplicity of necessary calculations. Unfortunately, if the learning rate is set too high, ReLU can make many nodes ineffective, causing as much as 40% of the neural network to be dead.[6]

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

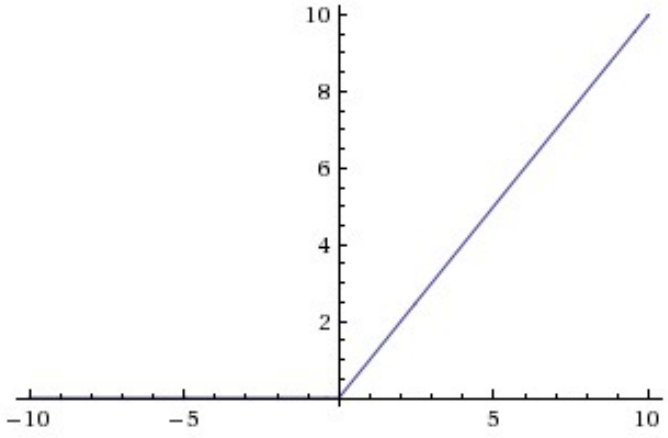


Figure 2.6: ReLU Activation Function.[6]

Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes:

$$f(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x \geqslant 0)(x) \tag{2.3}$$

where α is a small constant.

Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each node. However, the consistency of the benefit across tasks is presently unclear.[6]

Except for Sigmoid, ReLU and Leaky ReLU that are implemented in this thesis, there are several other activation functions which are widely used among the machine learning community, such as *Tanh, Softmax, Maxout* etc.

Loss Function

As the input gets transformed though the layers, it finally reaches a desired output form. At this point we need to be able to measure how far off this output is from what it is expected. For this purpose, we define a new term that is vital for the functionality of a neural network. A *loss function* measures the deviation of the estimated values from the actual values.[5] The effectiveness of a neural network depends on minimizing the loss function output. A neural network that has multiple outputs might have multiple loss functions (one per output), but the gradient-descent (see subsection 2.1.4) process must be based on a single scalar loss value; so, for multi-loss networks, all losses are combined (via averaging) into a single scalar quantity.[7] The loss function used depends on the goal of the model.

Choosing the right loss function is crucial for the proper function of a model, because if the calculated deviation does not express the properties of the true values, the model will be trained on doing wrong things. Fortunately, there are simple guidelines that can be followed.

For *regression problems*, where the model has to predict certain values, the loss function commonly used is *Mean Squared Error (MSE).*[5]

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

For *classification problems*, where the model has to classify between two classes, the loss function commonly used is *Binary Cross-entropy.*[5]

$$BCE = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{2.5}$$

For *multi-class classification problems*, where the model has to classify between more than two classes, the loss function commonly used is *Cross-entropy.*[5]

$$CE = -\sum_{i=1}^{k} \log(\hat{y}) \tag{2.6}$$

Sometimes the problem might request a more complex loss function or a combination of the above. In these cases, a *custom loss function* can be created that satisfies the criteria. One of these cases is the *object detection* problem targeted in this thesis.

Optimizer

An *Optimizer* is an algorithm that determines how the weight parameters will be changed during training, in order to minimize the output of the loss function. To determine how to update the weight parameters, an optimizer must first compute the *gradient* of the loss function. This concept could be easier to understand, if we think an optimizer as a blindfolded hiker that wants to abseil a mountain as fast as possible. So, he has to step by step feel the slope of the hill below his feet and descent the direction that feels steepest. The procedure of repeatedly evaluating the gradient and then performing a parameter update is called *gradient descent.*[6] The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step. Choosing the step size (also called the *learning rate* is one of the most important (and most headache-inducing) hyperparameter settings in training a neural network.[6] Most commonly encountered optimization algorithms are briefly analyzed in the next paragraphs.

Usually, gradient is calculated over a number of training data called *batch.* In case a batch contains only one training data per training cycle, the process is called *Stochastic Gradient Descent (SGD).* As we can see in the equation 2.7, the weight w is altered by the gradient $\nabla_w L$ of the loss function L (calculated for training example $x^{(i)}$ and label $y^{(i)}$) multiplied by the learning rate number η.[16]

$$w = w - \eta \nabla_w L(w; x^{(i)}; y^{(i)}) \tag{2.7}$$

In practice *Adam (Adaptive Moment Estimation)* is currently recommended as the default algorithm to use. In addition to storing an exponentially decaying average of past squared gradients v_t , Adam also keeps an exponentially decaying average of past gradients m_t as we can see below:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.8}$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \tag{2.9}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). These biases are counteracted by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.10}$$

$$\hat{u}_t = \frac{u_t}{1 - \beta_2^t} \tag{2.11}$$

Then, these are used to update the parameters, which yields the Adam update rule:

$$w_{t+1} = w_{t+1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{2.12}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ϵ.[16]

Except for SGD and Adam that are implemented in this thesis, there are several other optimizer algorithms which are widely used among the machine learning community, such as *Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, AdaMax, Nadam, AMSGrad* etc.[16]

Finally the anatomy of a Neural Network model can be summarized in the Figure 2.7.

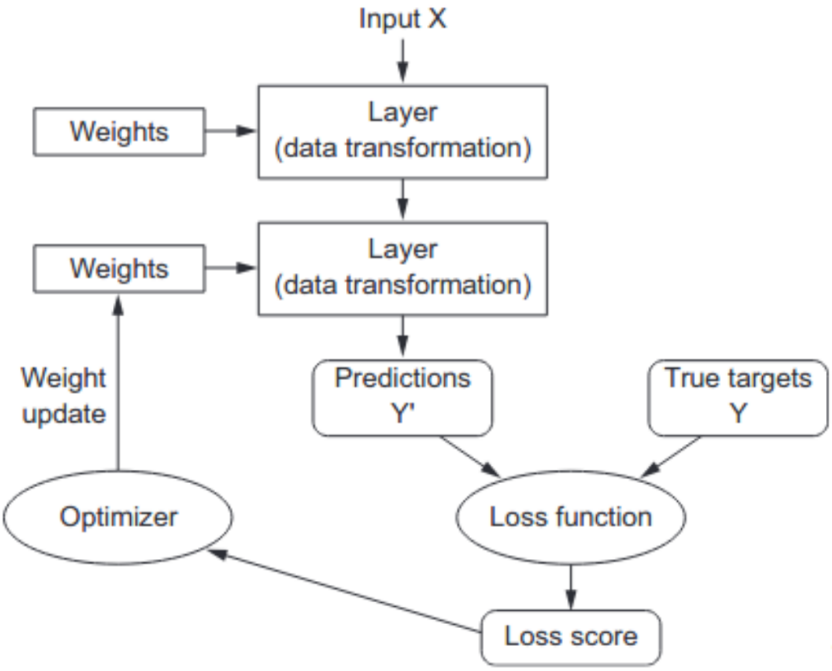


Figure 2.7: Neural Network model summary.[7]

2.1.5 Deep Learning

Deep Learning (DL) is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts and representations, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.[8] Basically, Deep Learning is the study and application of *deep neural networks*, which are, quite simply, neural networks with many layers (typically, from a dozen to hundreds of layers). Here, the word deep refers to the idea of a large number of successive layers of representations. The number of layers that form a model of the data is called the *model's depth*. Modern deep learning often involves tens or hundreds of successive layers of representations and they are all learned automatically from exposure to training data. Meanwhile, other approaches to machine learning tend to focus on learning only one or two layers of representations of the data; hence, they are sometimes called *shallow learning*.[3]

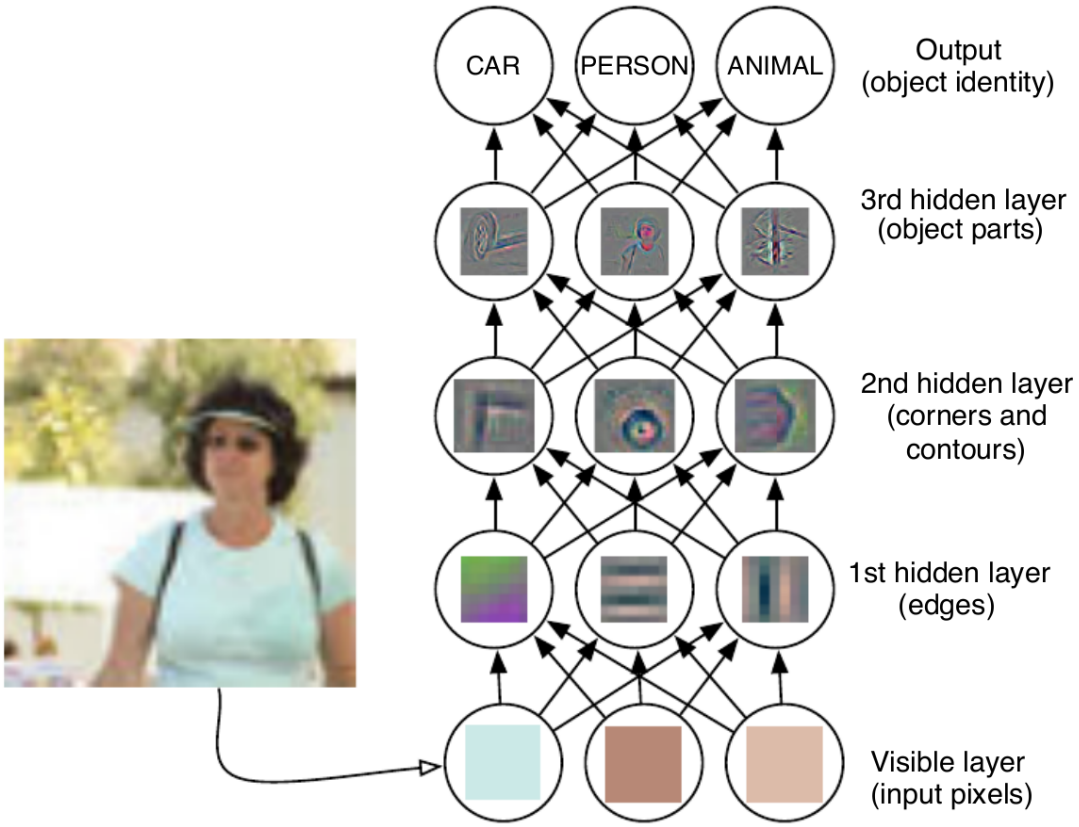


Figure 2.8: Illustration of a deep learning model.[8]

2.1.6 Computer Vision and Convolutional Neural Networks

Computer Vision (CV)

As humans, we perceive the three-dimensional structure of the world around us with apparent ease. Think of how vivid the three-dimensional percept is when you look at a vase of flowers sitting on the table next to you. You can tell the shape and translucency of each petal through the subtle patterns of light and shading that play across its surface and effortlessly segment each flower from the background of the scene. Looking at a framed group portrait, you can easily count (and name) all of the people in the picture and even guess at their emotions from their facial appearance.[17] The overly complicated science behind teaching a computer to perform all the above human tasks is called *Computer Vision (CV)*. Computer vision is itself a broad field, some parts of which use non machine learning techniques beyond the scope of this thesis and therefore, some of its most common uses today are indicatively mentioned below:[17]

- Optical character recognition (OCR)
- Machine inspection
- Retail
- 3D model building (photogrammetry)
- Medical imaging

- Automotive safety
- Match move
- Motion capture (mocap)
- Surveillance
- Fingerprint recognition and biometrics

The CV sub-field that this thesis is based on is called *deep learning for computer vision* and it is performed by a special type of neural network called *Convolutional Neural Network (CNN or ConvNet).*

Regular NN vs CNN

Convolutional Neural Networks are very similar to ordinary Neural Networks, meaning that they are made up of nodes that have learnable weights and biases. Each node receives some inputs, performs a dot product and optionally follows it with a non-linearity.[6] The whole network is still evaluated by a loss function, chosen according to the specific task the model is intended to do, the result of which then passes through an optimizer in order to update the weight parameters. So what is the difference between an ordinary Neural Network and a Convolutional Neural Network?

The main difference comes down to the way the nodes are arranged through the layers. In a regular NN, each *hidden layer* (layers between input and output layer) is made up of a set of nodes, where each node is fully connected to all nodes in the previous layer, and where nodes in a single layer function completely independently and do not share any connections. This leads to a large number of weight parameters, that stacks up as the size of the input image increases. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have nodes arranged in 3 dimensions: *width, height, depth.*[6] The difference between the two architectures is visualized in Figure 2.9.

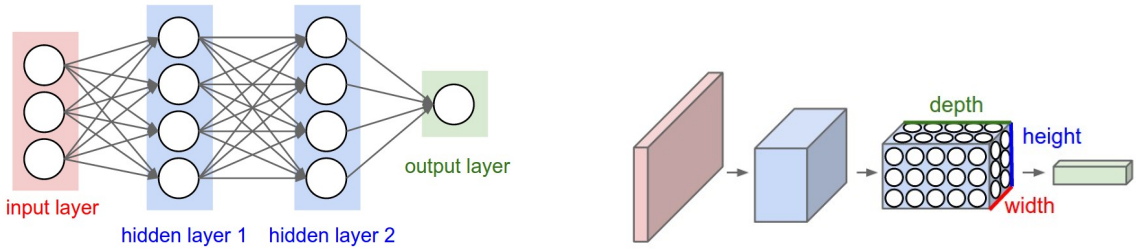


Figure 2.9: Visualization of the main difference between regular NN (left) and CNN (right).[6]

Anatomy of a Convolutional Neural Network

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: *Convolutional Layers (CONV), Pooling Layers (POOL), and Fully-Connected Layers (FC)* . These layers stacked, form a full *ConvNet architecture.*[6] As it is depicted in Figure 2.10, a typical CNN performing image classification is structured by:

- An *INPUT* layer that holds the raw pixel values of the image, with shape [imageWidth, imageHeight, 3]. The number 3 represents the RGB (red, green, blue) color values of a pixel at given coordinates.
- A repeated pattern of *CONV, RELU, POOL* layers. The RELU layers apply the element-wise activation ReLU function described in the subsection 2.1.4.
- An *FC* layer. Nodes in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. In some cases, FC layer is not necessary (e.g. YOLOv3).

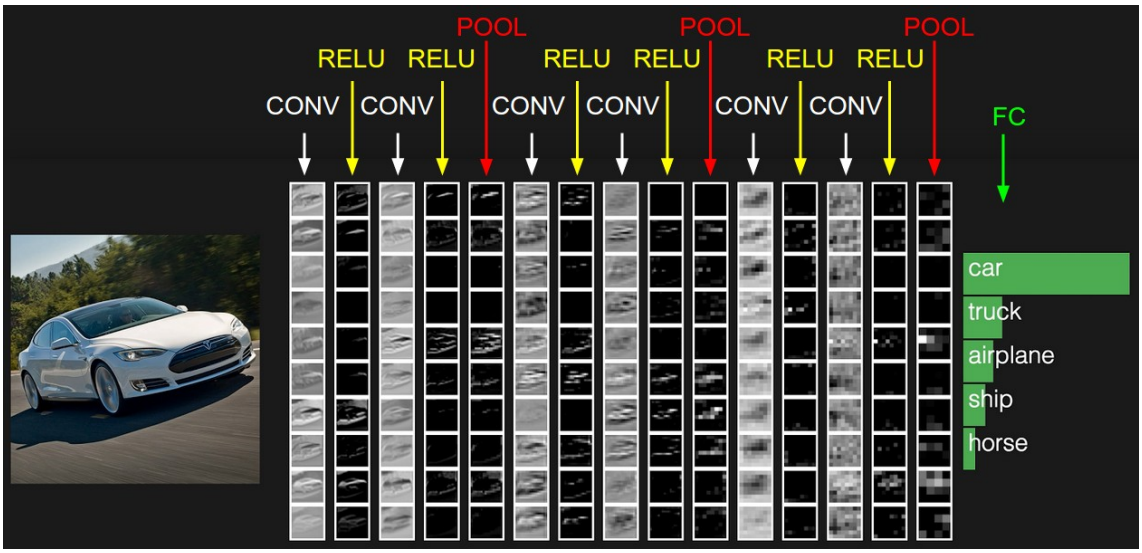


Figure 2.10: Typical ConvNet Architecture.[6]

The *Convolutional Layer* is the core building block of a Convolutional Network that does most of the computational heavy lifting. The CONV layer's parameters consist of a set of learnable filters called *convolutional kernels*, or simply *kernels*. Every kernel is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical kernel on a first layer of a ConvNet (also called input layer) might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each kernel across the width and height of the input volume and compute dot products between the entries of the kernel and the input at any position. As we slide the kernel over the width and height of the input volume we will produce a 2-dimensional *activation map* that gives the responses of that kernel at every spatial position. Intuitively, the network will learn kernels that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of kernels in each CONV layer, and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the *output volume*.[6] The key-parameters that define a convolutional layer are the following:

- *Kernel size*: e.g. 3x3, 5x5

- *Depth*: the number of filters we would like to use, each learning to look for something different in the input. (Unlike the input image, the depth in the output tensor doesn't actually have to do with colors.)
- *Stride*: the size of the kernel's step.
- *Zero-padding*: the amount of zeros padded around the output volume.

Figure 2.11 illustrates a convolutional layer in greater detail. The input image, for the sake of simplicity, is of shape [width: 5, height: 3, color chanels: 2]. The convolutional layer is defined by kernel size: 3x3, depth: 3, stride: [1, 1] and zero-padding: 0. Here we can see only the kernel output at position [2, 1].

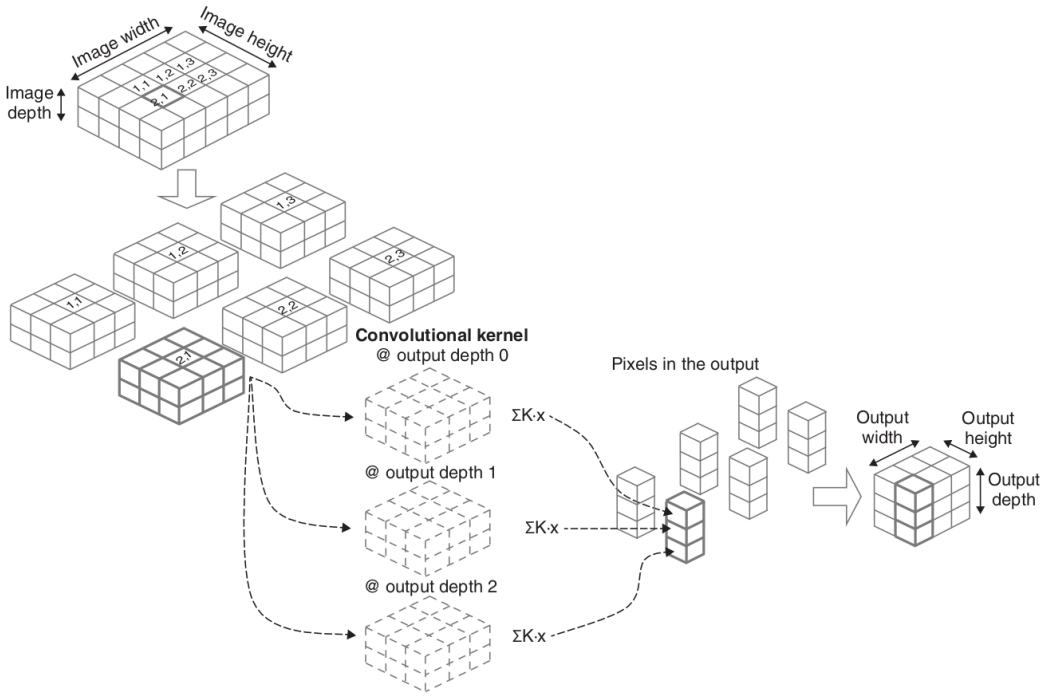


Figure 2.11: A Convolutional Layer in action.[3]

A *Pooling Layer* is much simpler than a convolutional layer. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting (see 2.2.2), meaning, to make the ConvNet less sensitive to the exact location of key features in the input image. This property is called *positional invariance.*[3] The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the *MAX operation.* More rarely, a pooling unit can also perform other functions, such as average pooling or even L2-norm pooling. A pooling layer is also defined by:

- *Pool Size*
- *Stride*

In Figure 2.12 we can see a Pooling Layer with poolSize: 2x2 and stride: [2, 2] perform MAX operation.

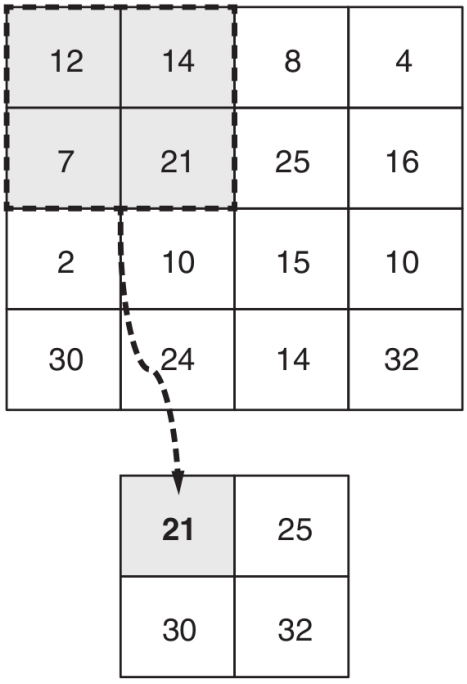


Figure 2.12: A Pooling Layer in action.[3]

In some models (e.g YOLOv3) pooling layers are replaced by more convolutional layers, in order to prevent loss of low-level features.

Common Computer Vision tasks performed by ConvNets

Image Classification is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, Image Classification refers to images in which only one object appears and is analyzed. Some of the most efficient Image Classification models include *ViT (Vision Transformer)*, *EfficientNet*, *MobileNet*, *VGG* etc. In Figure 2.13 we can see image classification predictions performed by a ConvNet trained on MNIST [18] dataset, an image database of handwritten digits.

Figure 2.13: Image Classification prediction examples made by a ConvNet, shown alongside the input MNIST images.[3]

In contrast to Image Classification, *Object Detection* involves both classification and localization tasks, and is used to analyze more realistic cases in which multiple objects may exist in an image. More specifically, object detection is the task of detecting instances of objects of a certain class within an image, and then highlighting them, by usually delimiting them in boxes. The state-of-the-art methods can be categorized into two main types: one-stage methods and two stage-methods. One-stage methods prioritize infer-

ence speed, and example models include *YOLO*, *SSD* and *RetinaNet*. Two-stage methods prioritize detection accuracy, and example models include *Faster R-CNN*, *Mask R-CNN* and *Cascade R-CNN*. In Figure 2.14 we can see object detection predictions performed by YOLOv3 in an "object-dense" environment.

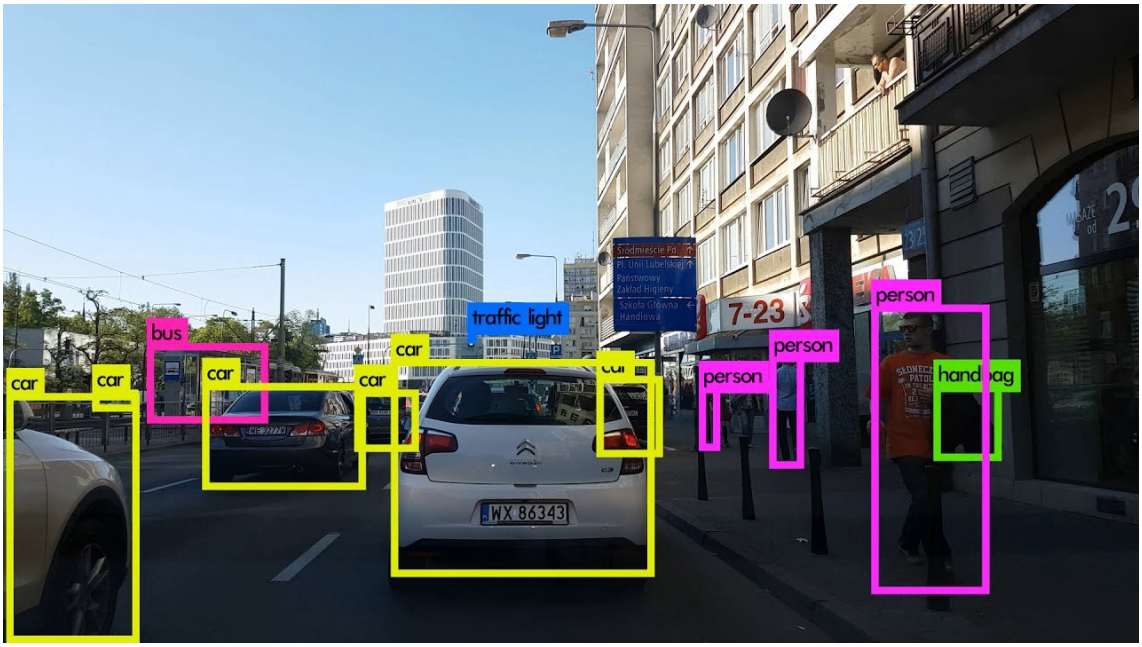


Figure 2.14: Object Detection example by YOLOv3.[9]

Other common Computer Vision tasks performed by ConvNets include *Semantic Segmentation*, *Pose Estimation*, *Image Generation*, *Depth Estimation*, *Style Transfer* etc.

2.2 Training Neural Networks

As it is already mentioned, we can teach a neural network to perform a specific task by feeding many examples to it. The network processes the input data through the layers and produces an output. The output then passes to a loss function, where it is compered with the labels' true values. The scalar output of the loss function, then, helps the optimizer to determine how the weight parameters of each layer will be updated. This cycle is repeated until we are satisfied with the accuracy of our model. This whole procedure is called *training*. In order to train a model, we must first define some parameters:

- *Input Data*: the input data that we feed to the model. In case of CNN, input data are usually images.
- *Labels*: the desired outputs of our model for each input data, created by a time-consuming procedure called *labeling*. In case of, for example object detection, a label might consists of class indicators and bounding box coordinates and dimensions.
- *Batch Size*: the number of input data our model processes at once. In general, the benefit of using larger batch sizes is that it produces a more consistent and less variable gradient update to the model's weights than a smaller batch size. But the larger the batch size, the more memory is required during training. We should also keep in mind that given the same amount of training data, a larger batch size leads to a small number of gradient updates per epoch.[3]

- *Epoch*: the number of iterations over each batch.
- *Validation Split*: the percentage of data that are used for evaluating the model, usually around 15%. While training, it is important to monitor validation loss and accuracy, in order to avoid *overfitting* (see 2.2.2).

2.2.1 Transfer Learning

In some cases, we might encounter a situation, where the available dataset examples intended to train our model to do a complicated task such as object detection, are insufficient. A common strategy to tackle this problem is to use *data augmentation*, a technique, where images are manipulated through various techniques (e.g. rotation, crop, flip, brightness, hue/saturation, exposure, gray-scaling etc.), in order to create new data. The two main drawbacks of this technique are: 1) it does not eliminate the problem of overfitting, 2) big dataset means time-consuming and memory-demanding computations.

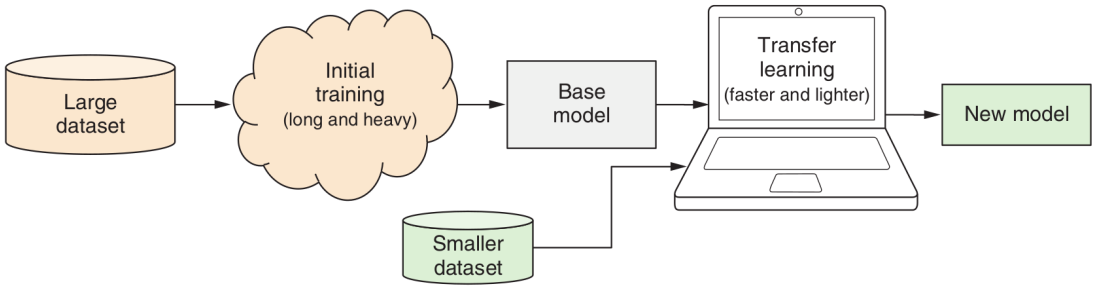


Figure 2.15: The general workflow of transfer learning.[3]

Another technique that is used instead/along with data augmentation is *Transfer Learning*. In essence, transfer learning is about speeding up a new learning task by reusing the results of previous learning. It involves using a model already trained on a dataset to perform a different but related machine-learning task. The already-trained model is referred to as the *base model*. Transfer learning sometimes involves retraining the base model and sometimes involves creating a new model on top of the base model. We refer to the new model as the *transfer model*.[3] In Figure 2.15 we can see the general workflow of transfer learning. We can implement transfer learning with the following three techniques:

Freezing Layers

The Freezing layers technique is used when the desired output shape of the transfer model is compatible with the base model's. For example, we can use the this technique on a CNN that is trained to recognize handwritten digits from 0 to 4, to create a new model that recognizes digits 5 to 9. In order to implement the freezing layer technique, before we feed the new dataset, we have to "freeze" all the layers except for the last ones. By freezing, we mean restricting the weight parameters of the *frozen layers* (also called *feature layers*) to be updated during the training phase. This significantly reduces the training time, because during backpropagation the only weight parameters that will be updated are the ones of the last layers (also called *head* of the model). [3] In Figure 2.16 we can see schematic explanation for why freezing some layers of a model speeds up training.

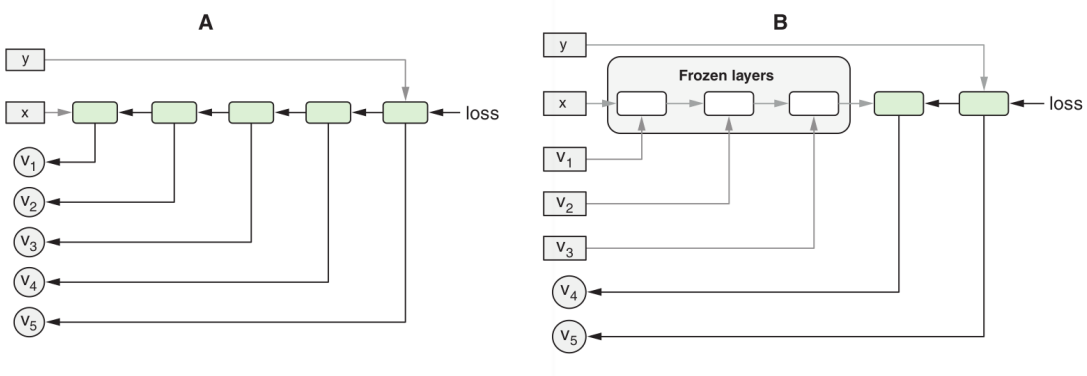


Figure 2.16: The Freezing Layers technique.[3]

Two Models

The Two Models technique is used in cases that require an output shape different from the original one. In order to implement this technique we have to remove the head of the base model, creating a new one referred to as *truncated model*. The output of the truncated model is then used as an input to a new smaller model, that functions as an independent head and produces the desired output shape. One of the main advantages of the two models technique is that the outputs of the truncated model (also called *embedding tensors* are directly accessible, giving the ability to perform many different methods of classification. The two disadvantages are, that the complexity increases, because we have to manage two models and that it is difficult to fine-tune layers of the original model.[3] In Figure 2.17 we can see a schematic of the two model technique implemented on a webcam controlled Pac-Man model that uses MobileNet as the base model.

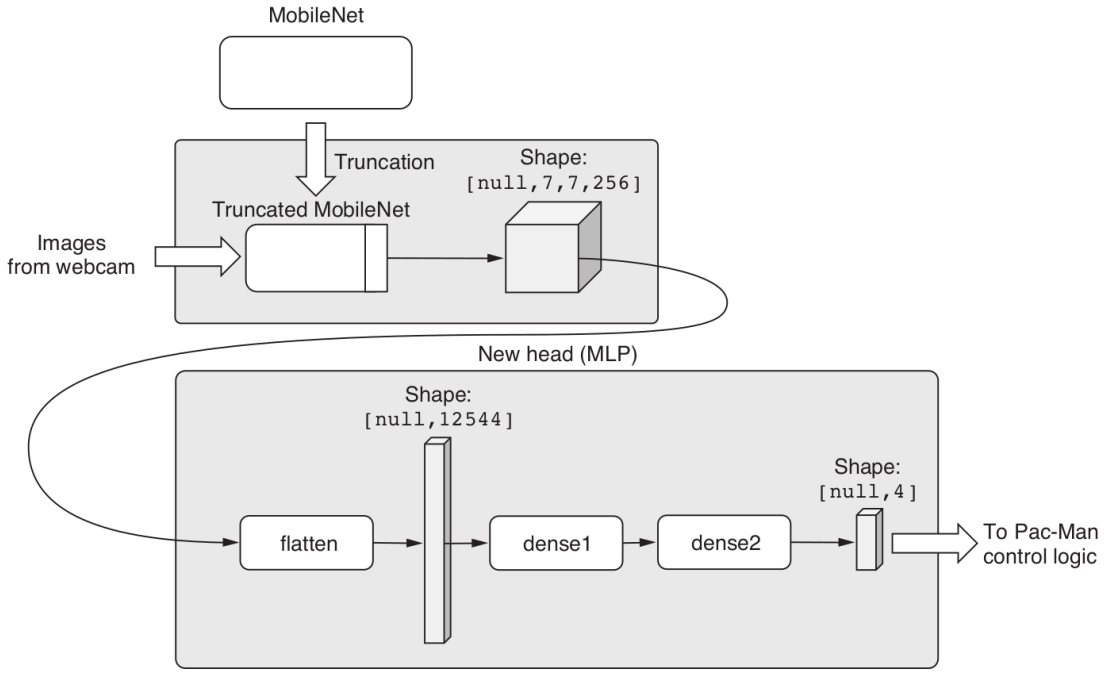


Figure 2.17: The Two Models technique.[3]

Single Model

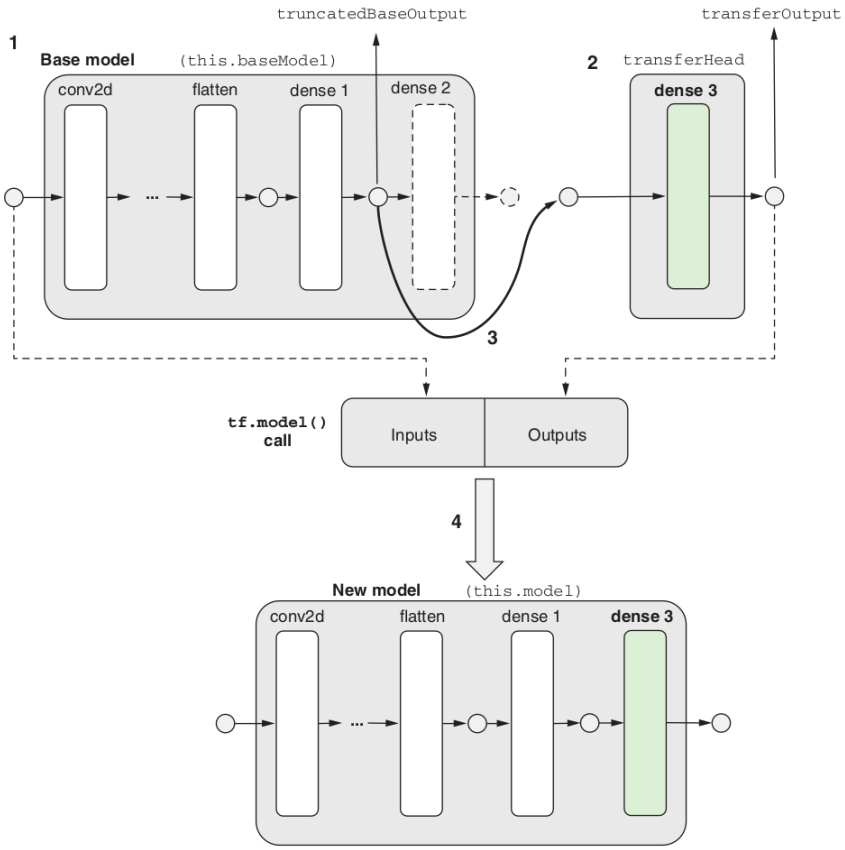


Figure 2.18: The Single Model technique.[3]

The Single Model technique, just like the two models technique, is used in cases that require an output shape different from the original one. The idea behind this technique is the creation of a new model that contains the feature-extracting layers of the original model and the layers of the new head. To achieve that we add a step to the two layers technique, where the truncated model and the new head are combined into a single model that is defined by the input and the output of the two models respectively. This simplifies the training procedure and enables fine-tuning of feature-extracting layers.

Fine-tuning is an optional step of transfer learning that follows an initial phase of model training. As previously mentioned, in the initial phase, all the layers from the base model are frozen, and weight updating happens only to the head layers. On the contrary, during fine-tuning, some of the layers of the base model are unfrozen, and then the model is trained on the transfer data again. [3] Fine-tuning achieves a more robust connection between the truncated model and the new head that increases accuracy. This layer unfreezing is shown schematically in Figure 2.19.

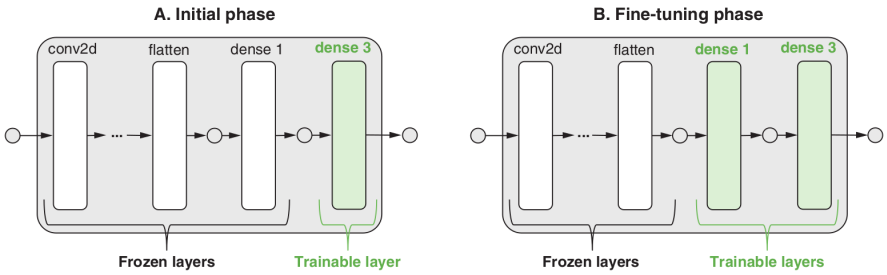


Figure 2.19: Illustration of the Fine-tuning phase.[3]

During the Single Model technique, internal activations (embeddings) are not directly accessible. That means, that in order to teach our model a completely different task (e.g. teach an image classification model to perform object detection), we might have to compile the model with an entirely different loss function, or like in our case, create a custom one, which can be really challenging.

2.2.2 Overfitting and Underfitting

During the training phase of a machine-learning model, we want to monitor how well our model is capturing the patterns in the training data. A model that doesn't capture the patterns very well is said to be *underfit*; a model that captures the patterns too well, to the extent that what it learns generalizes poorly to new data, is said to be *overfit*.[3]

Underfitting

Underfitting is usually a result of using an insufficient representational capacity (power) to model the feature-target relationship. To overcome underfitting, we usually increase the power of the model by making it bigger. Typical approaches include adding more layers (with nonlinear activations) to the model and increasing the size of the layers.[3]

Overfitting

Overfitting is a state in which the model pays too much attention to the fine details of the data it has seen during training—so much so that its prediction accuracy on data not seen during training is negatively affected.[3] An overview of commonly used methods for reducing overfitting can be seen below:

- *L2 regularizer*: Assigns a positive loss (penalty) to the weight by calculating the summed squares of parameter values of the weight. It encourages the weight to have smaller parameter values.
- *L1 regularizer*: Like L2 regularizers, encourages the weight parameters to be smaller. However, the loss it assigns to a weight is based on the summed absolute values of the parameters, instead of summed squares. This definition of regularization loss causes more weight parameters to become zero (that is, "sparser weights").
- *Combined L1-L2 regularizer*: A weighted sum of L1 and L2 regularization losses.
- *Dropout*: Randomly sets a fraction of the inputs to zero during training (but not during inference) in order to break spurious correlations among weight parameters that emerge during training.

- *Batch normalization*: Learns the mean and standard deviation of its input values during training and uses the learned statistics to normalize the inputs to zero mean and unit standard deviation as its output.
- *Early stopping of training based on validation-set loss*: Stops model training as soon as the epoch-end loss value on the validation set stops decreasing.

Chapter 3

Tools and Model Selection

The complicated goal of this thesis requires the most suitable tools. This chapter lists the hardware and software used for the development of this thesis and presents the results of a comparison experiment between two object detection models, YOLO and SSD. In addition, YOLOv3 architecture is explained thoroughly.

3.1 Hardware and Software

Deep Learning and especially object detection is an extremely computationally intensive and complicated task for any personal computer or laptop. Multinational corporations use datasets in size of terabytes in order to achieve state-of-art results. To train a model, they usually use TPUs (Tensor Processing Units), especially designed to perform deep learning computations, instead of CPUs (Central Processing Units) and GPUs (Graphics Processing Units) our computers usually have. In addition, our final object detection model has to receive the video stream from Peristera island through the internet and that creates some extra limitations. Considering the above, it is crucial to choose the most suitable tools, in order to simplify the procedure and overcome/adapt to the limitations.

Hardware

Designing, development, coding, data manipulation, training and testing were done in a **personal laptop** with the following specifications:

Table 3.1: Machine Specifications

Specification	**Value**
Laptop Model	Lenovo Z51-70
CPU	Intel® Core™ i7-5500U CPU @ 2.40GHz × 4
GPU	AMD® Verde / Mesa Intel® HD Graphics 5500 (BDW GT2)
RAM Memory	8 GB
Disk Capacity	1 TB

Operating System

A key parameter of this project was that we needed a system that is simple to configure to our needs and yet easy to program in it. So we chose **Ubuntu 20.04 LTS**, the latest (at the time) version of Ubuntu distributors. Ubuntu is a Linux based OS that is pretty

commonly used for Machine Learning, hence there is a vast community in sites like Stack Overflow, GitHub etc. In order to use Linux, we had to convert the laptop to a dual boot machine with both Ubuntu 20.04 LTS and the pre-installed Windows 10.

Programming Language

We made the odd choice to develop our application in **JavaScript**. Machine learning, like other branches of AI and data science, is usually done with traditionally backend-focused languages, such as Python and R, running on servers or workstations outside the web browser. This status quo is not surprising. The training of deep neural networks often requires the kind of multicore and GPU-accelerated computation not directly available in a browser tab; the enormous amount of data that it sometimes takes to train such models is most conveniently ingested in the back-end: for example, from a native file system of virtually unlimited size.[3] But, as mentioned in the introduction of this section, the expected data are available in the browser and JavaScript is the default programming language running in it. Besides, with the release of TensorFlow.js library for machine learning and Node.js for backend-development, training and deploying a top-notch model is not impossible. In Figure 3.1 we can see a brief summary of the benefits of doing deep learning in JavaScript.

Consideration	Examples
Reasons related to the client side	• Reduced inference and training latency due to the locality of data • Ability to run models when the client is offline • Privacy protection (data never leaves the browser) • Reduced server cost • Simplified deployment stack
Reasons related to the web browser	• Availability of multiple modalities of data (HTML5 video, audio, and sensor APIs) for inference and training • The zero-install user experience • The zero-install access to parallel computation via the WebGL API on a wide range of GPUs • Cross-platform support • Ideal environment for visualization and interactivity • Inherently interconnected environment opens direct access to various sources of machine-learning data and resources
Reasons related to JavaScript	• JavaScript is the most popular open source programming language by many measures, so there is an abundance of JavaScript talent and enthusiasm. • JavaScript has a vibrant ecosystem and wide applications at both client and server sides. • Node.js allows applications to run on the server side without the resource constraints of the browser. • The V8 engine makes JavaScript code run fast.

Figure 3.1: The benefits of doing deep learning in JavaScript.[3]

Deep Learning Library

We decided to use **TensorFlow.js**, a library that enables you to do deep learning in JavaScript. As its name suggests, TensorFlow.js is designed to be consistent and compatible with TensorFlow, the Python framework for deep learning. TensorFlow.js is not

the only JavaScript library for deep learning; neither was it the first one to appear (for example, brain.js and ConvNetJS have a much longer history). So, why did we choose TensorFlow.js among similar libraries?

The first reason is its comprehensiveness. TensorFlow.js is the only currently available library that supports all key parts of the production deep-learning workflow:[3]

- Supports both inference and training.
- Supports web browsers and Node.js.
- Leverages GPU acceleration (WebGL in browsers and CUDA kernels in Node.js).
- Supports definition of neural network model architectures in JavaScript.
- Supports serialization and deserialization of models.
- Supports conversions to and from Python deep-learning frameworks.
- Compatible in API with Python deep-learning frameworks.
- Equipped with built-in support for data ingestion and with an API for visualization.

The second reason is the ecosystem. Most JavaScript deep-learning libraries define their own unique API, whereas TensorFlow.js is tightly integrated with TensorFlow and Keras. Tight integration with non-JavaScript frameworks not only boosts interoperability but also makes it easier for developers to migrate between the worlds of programming languages and infrastructure stacks.[3] And finally, been most widely used programming language right now, Javascript has a great support community and so does TensorFlow.js.

3.2 Model Selection

As mentioned in the first chapter, the main target of this thesis is to use an object detection model to monitor vessel traffic over the Peristera shipwreck. Creating and training an object detection model from scratch is something that exceeds the purpose of this thesis. Another common way this problem can be approached is by using a pre-developed state-of-art object detection model in its standard state or by modifying it. Following this approach, we compared two models: YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector).

Comparing in detail YOLO and SSD could be another project on each own. Opinions are very much divided on this issue. For example, after comparing YOLOv3 with other models on threshold score 0.5, Joseph Redmon and Ali Farhadi (creators of YOLO) stated: "... YOLOv3 is very strong. It is almost on par with RetinaNet and far above the SSD variants".[10] Other researchers disagree, like Chandrakala Busireddy, who compared many versions of the two models with the same hardware and came to the conclusion that: "The SSD network ran both faster and had superior performance to YOLO".[19]

However, unfortunately, all comparison researches we found and studied are characterized by the same three problems listed below:

- The tests were not executed in JavaScript or Node.js
- The machine learning library used was not TensorFlow.js

- The tests were not executed in browser, but instead in powerful computer systems.

It was apparent that we could not exclude conclusions without testing. So we decided to conduct our own experiment, with a limited amount of images and videos as dataset, that we thoroughly selected in order to cover a wide range of resolution, brightness, blurriness, object density etc.

3.2.1 Model's Configuration for Comparison

More specifically, we decided to compare *YOLOv3* and *SSD Mobilenet v2*. The two models were compared in their default configuration that is explained below.

SSD

In order to test SSD model, we adapted the *Object Detection (COCO-SSD) Demo* [20] to perform prediction on images and videos uploaded by the user. The model was tested in standard configuration as shown in the table below:

Table 3.2: SSD Comparison Configuration

Configuration	**Value**
Model Version	SSD Mobilenet v2
Input Shape	[n, 300, 300, 3]
Max Boxes	20
Score Threshold	0.5
Trained on	COCO Dataset [21]

YOLO

In order to test YOLO model, we adapted the *tfjs-yolo* code [22] to perform prediction on images and videos uploaded by the user. The model was tested in standard configuration as shown in the table below:

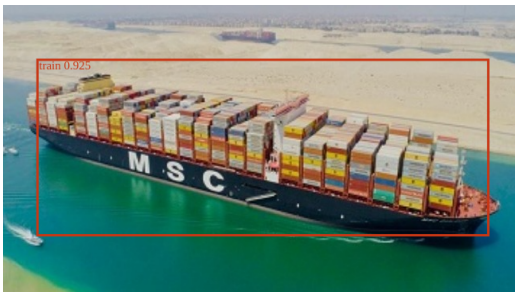
Table 3.3: YOLO Comparison Configuration

Configuration	**Value**
Model Version	YOLO v3
Input Shape	[n, 416, 416, 3]
Max Boxes	20
Score Threshold	0.5
Trained on	COCO Dataset [21]

3.2.2 Comparison Results

The results of the comparison test can be seen in Figure 3.2 where the YOLOv3 outputs are presented in the left column and the SSD Mobilenet v2 in the right. Some conclusions that can be pointed out are listed below:

- YOLOv3 is more accurate when an image contains multiple objects.
- YOLOv3 is more accurate on blurry video frames (e.g. images: *s/t, u/v, w/x*)
- The two models performed decently on 3D rendered ship models (e.g. images: *m/n*)
- SSD Mobilenet v2 showed a higher frame rate speed averaging at 5.6 FPS (Frames Per Second), while YOLOv3 averaged 0.8 FPS.
- In images *o* and *p* we can observe that the two models recognized correctly only one of the two depicted boats. This is an actual low resolution image, captured by the camera installation on Peristera island.

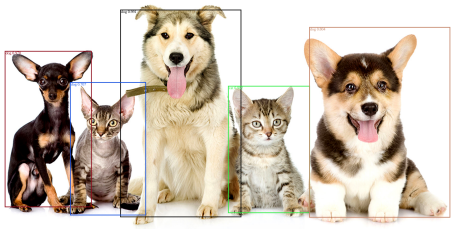


(a)

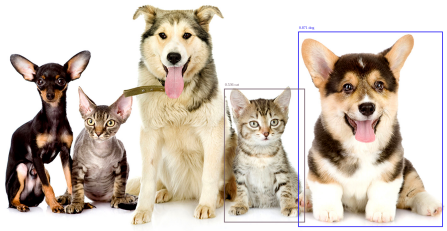
(b)

(c)

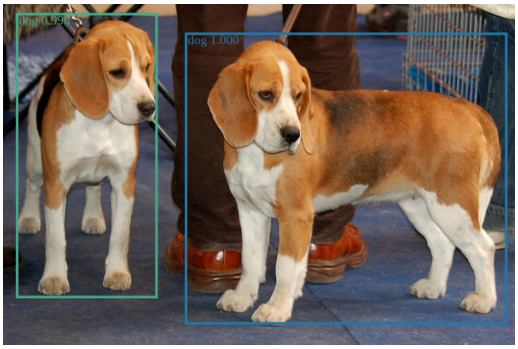
(d)



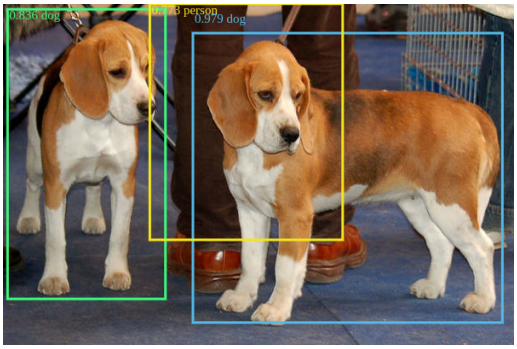
(e)



(f)


dog 1.000

(g)


0.979 dog

(h)


boat 0.995

(i)


0.873 boat

(j)

(k)

(l)


Costa Smeralda

(m)


Costa Smeralda

(n)


boat 0.768

(o)


0.677 boat

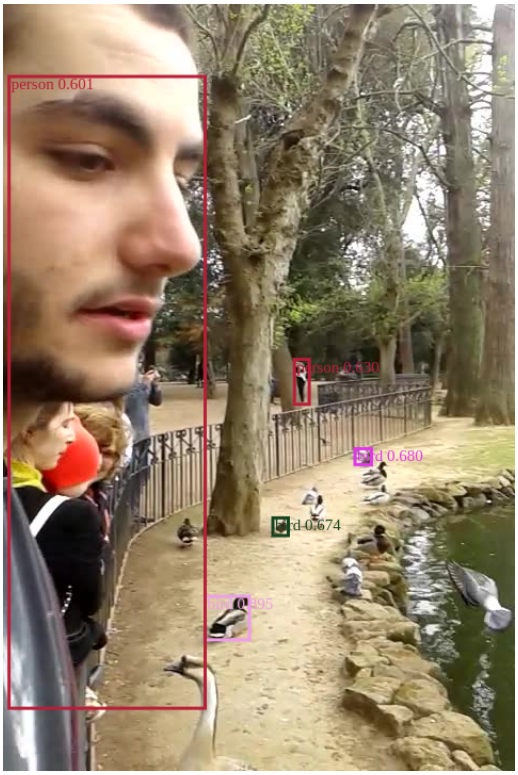
(p)


person 0.915
person 0.993
person 0.984

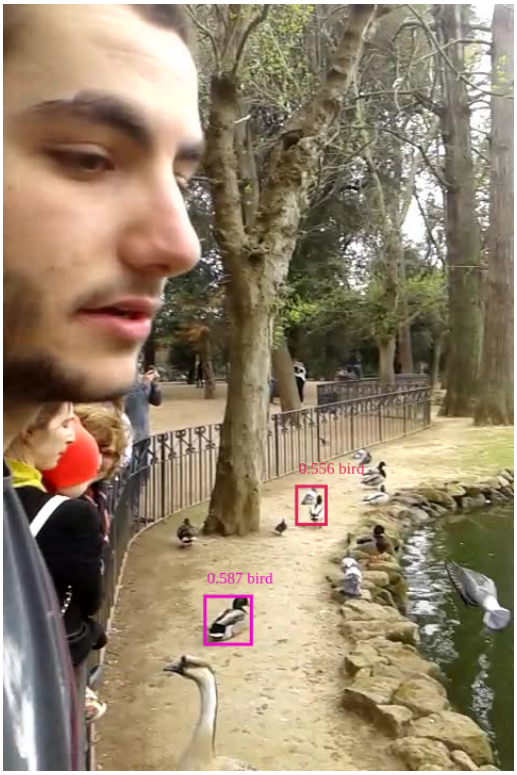
(q)


0.944 kite
0.837 kite
0.513 kite
0.831 person
0.729 person
0.674 person

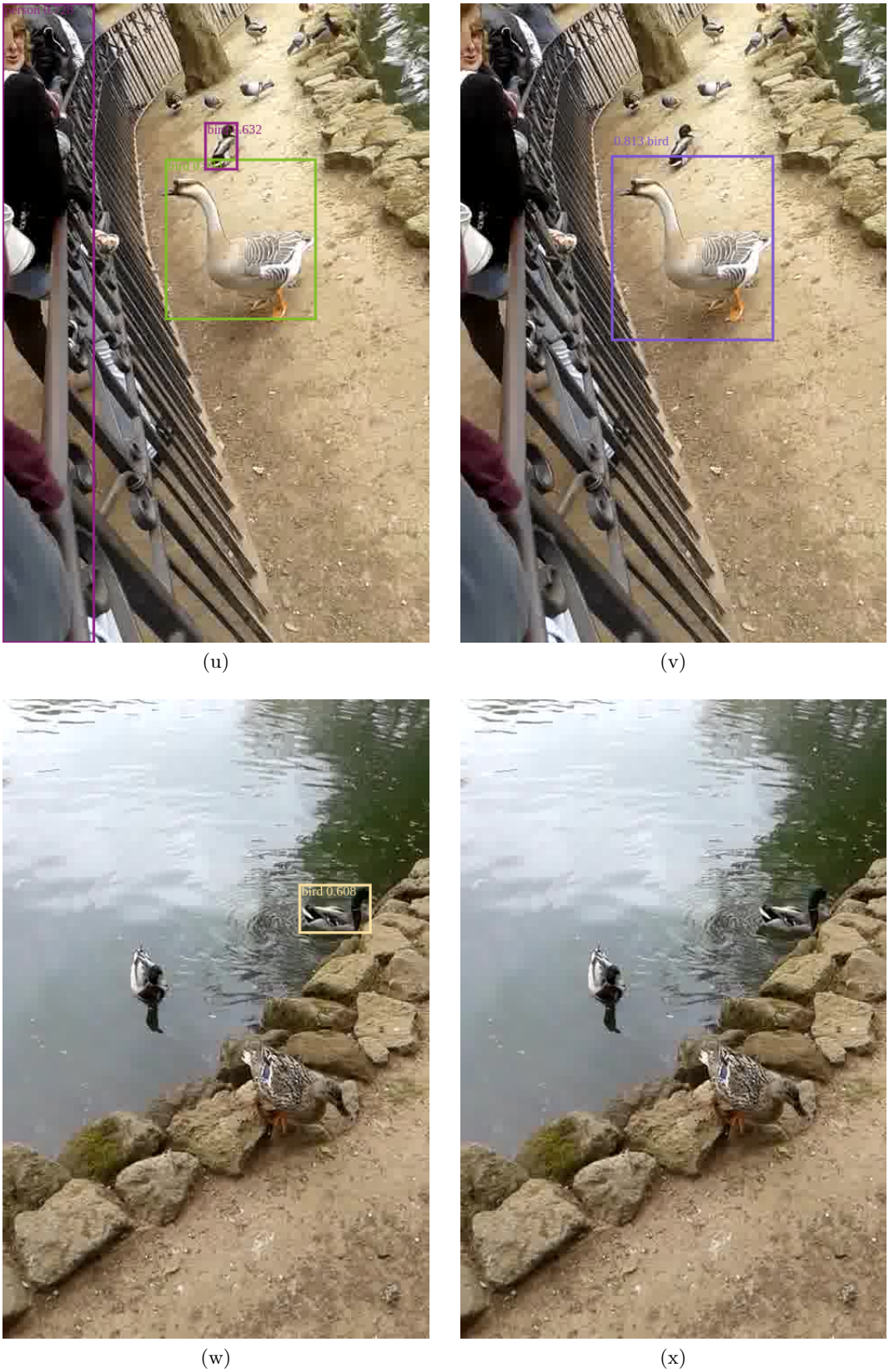
(r)


person 0.601
person 0.630
bird 0.680
bird 0.674

(s)


0.556 bird
0.587 bird

(t)



(u) (v)

(w) (x)

Figure 3.2: Object detection predictions by YOLOv3 (left) and SSD Mobilenet v2 (right)

In conclusion, we chose to base our thesis on **YOLOv3** object detection model. This choice leads us to adapt to the following complications:

- YOLOv3 is a computationally intensive model.
- It was built on Darknet [23] framework, so it needs some adaptations in order to run on TensorFlow.js.
- While testing, it showed a relatively low Frame Rate of 0.8 FPS.

3.3 YOLOv3 Model

YOLOv3 was created by Joseph Redmon and Ali Farfadi in 2018 and it is part of YOLO ("You Only Look Once") model family.[10] As its predecessors, YOLOv3 uses Convolutional Neural Networks to perform object detection. It can detect multiple objects in a single image, meaning, that apart from predicting objects' classes, it also estimates their locations. In order to achieve that, YOLOv3 applies a single CNN that divides the image into *grid cells* and produces probabilities for each one. Then, it predicts a number of *bounding boxes* that cover some cells and chooses the best ones according to the probabilities.[24]

3.3.1 YOLOv3 Architecture

YOLOv3 consists of 53 convolutional layers that are also called *Darknet-53*. Each convolutional layer is followed by a batch normalization layer and a Leaky ReLU activation function. There are no pooling layers, but instead, additional convolutional layers with stride 2, are used to down-sample feature maps, preventing loss of low-level features that pooling layers just exclude.[24]

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 3.3: YOLOv3 feature extractor network architecture Darknet-53.[10]

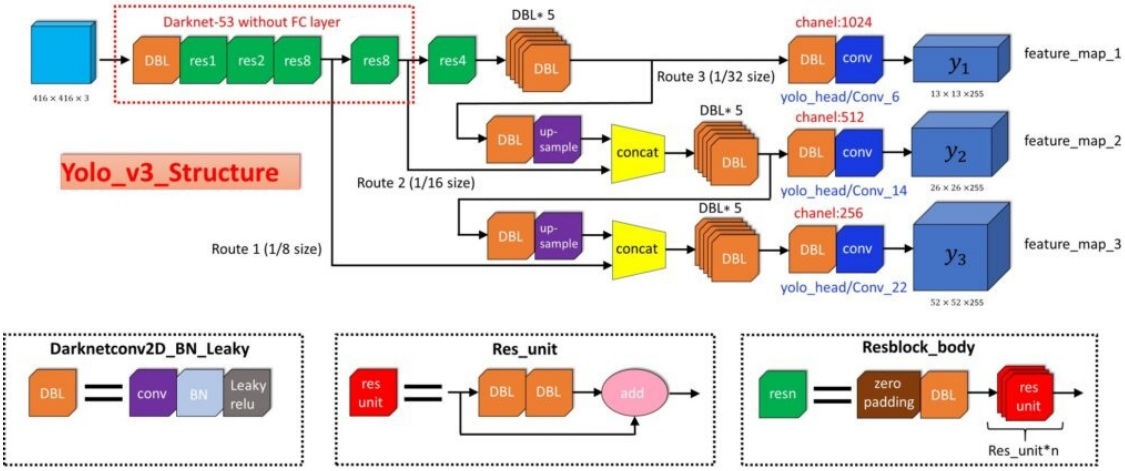


Figure 3.4: The complete YOLOv3 model architecture.[11]

Input

The default input of the Network is a batch of images of shape [n, 416, 416, 3], where n is the number of images, the next two numbers are width and height and the last one is the number of color channels (RGB). The width and height can be changed and set as any other number that is divisible by 32 without leaving a remainder, for example 320x320 or 608x608. Input images themselves can be of any size, meaning, that they don't have to be resized manually before being fed to the network. YOLOv3 will adjust their size according to the input shape.[24]

Output

As it is depicted in Figure 3.4, YOLOv3 performs detection at 3 separate layers in the Network, producing a *triple output*. At these 3 layers, the input image is downsampled by factors 32, 16, 8 respectively. These numbers are called *strides* of the network. For instance, if we consider stride 32 and input size 416x416, then the output will be of size 13x13. Consequently, for stride 16 the output will be 26x26 and for the stride 8 the output will be 52x52. 13x13 is responsible for detecting large objects; 26x26 is responsible for detecting medium-sized objects and 52x52 is responsible for detecting small objects.[24]

To produce this output, YOLOv3 applies 1x1 *detection kernels* at these three separate places in the Network. The depth of the detection kernels is calculated by the following equation:

$$depth = b \cdot (5 + c) \tag{3.1}$$

Where:

- b : represents the number of bounding boxes that each cell of the produced feature map can predict. YOLOv3 predicts 3 bounding boxes for every cell of these feature maps hence, b is equal to 3.
- $5 + c$: represents the attributes of each bounding box: center coordinates (t_x, t_y), width t_w, height t_h, objectness score p_0 and a size c list of confidences for every class this bounding box might belong to.

For example if YOLOv3 is trained on COCO dataset that has 80 classes, then c is equal to 80. Considering a default input of 416x416, the shapes of the three outputs will be [n, 13, 13, 255], [n, 26, 26, 255] and [n, 52, 52, 255].[24]

Grid Cells

We already know that YOLOv3 predicts 3 bounding boxes for every cell of the feature map. Each cell, in turn, predicts an object through one of its bounding boxes if the center of the object belongs to the receptive field of this cell. And this is the task of YOLOv3 while training: identify this cell that falls into the center of the object. When YOLOv3 is training, it has one ground truth bounding box that is responsible for detecting one object. That's why and firstly, we need to define which cells this bounding box belongs to. And to do that, let's consider the first detection scale, where we have 32 as stride of the Network. The input image of 416x416 is downsampled into 13x13 grid cells. This grid, now, represents the produced output feature map. When all cells, that the ground truth bounding box belongs to, are identified, the center cell is assigned by YOLOv3 to be responsible for predicting this object. The objectness score for this cell is equal to 1.[24]

Anchor Boxes

To predict bounding boxes YOLOv3 uses pre-defined default bounding boxes that are called *anchors* or *priors*. These anchors are used later to calculate predicted bounding box's real width and real height. In total, 9 anchor boxes are used. Three anchor boxes for each scale. This means that at each scale, every grid cell of the feature map can predict 3 bounding boxes by using 3 anchors.[24] For the COCO dataset the 9 Anchor Boxes are: (10×13), (16×30), (33×23), (30×61), (62×45), (59×119), (116×90), (156×198), (373×326).[10]

3.3.2 How does YOLOv3 Predict ?

Bounding Box Prediction

YOLOv3 uses dimension clusters as anchor boxes. As it was mentioned above, The network predicts 4 coordinates for each bounding box: t_x, t_y, t_w, t_h. If the cell is offset from the top left corner of the image by (c_x, c_y) and the anchor box has width and height p_w, p_h, then the predictions correspond to [10]:

$$b_x = \sigma(t_x) + c_x \tag{3.2}$$

$$b_y = \sigma(t_y) + c_y \tag{3.3}$$

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

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

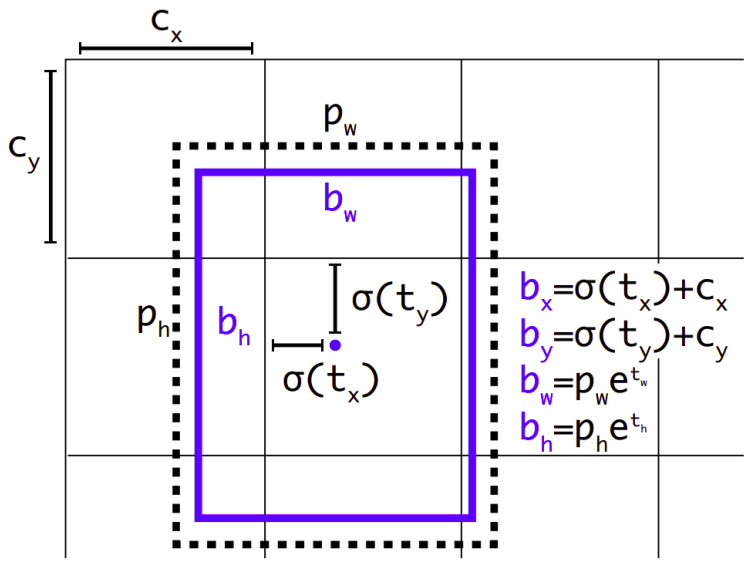


Figure 3.5: Bounding boxes with dimension priors and location prediction.[10]

YOLOv3 doesn't predict absolute values of width and height. Instead, as mentioned above, it predicts offsets to anchors. Why? Because it helps to eliminate unstable gradients during training. That's why values c_x, c_y, p_w and p_h are normalized to the real image width and height, and the center coordinates t_x, t_y are passed through sigmoid function that gives values between 0 and 1. Consequently, to get absolute values after predicting, we simply need to multiply them to the real and whole image width and height.[24]

All the predicted bounding boxes are filtered with *non-maximum suppression*. Non-maximum suppression takes a bounding box with the highest probability and then looks at other bounding boxes that are close to the first one and the ones with the highest overlap with this one will be suppressed. The overlap is measured with a method called *Intersection Over Union (IoU)* that is illustrated in Figure 3.6.

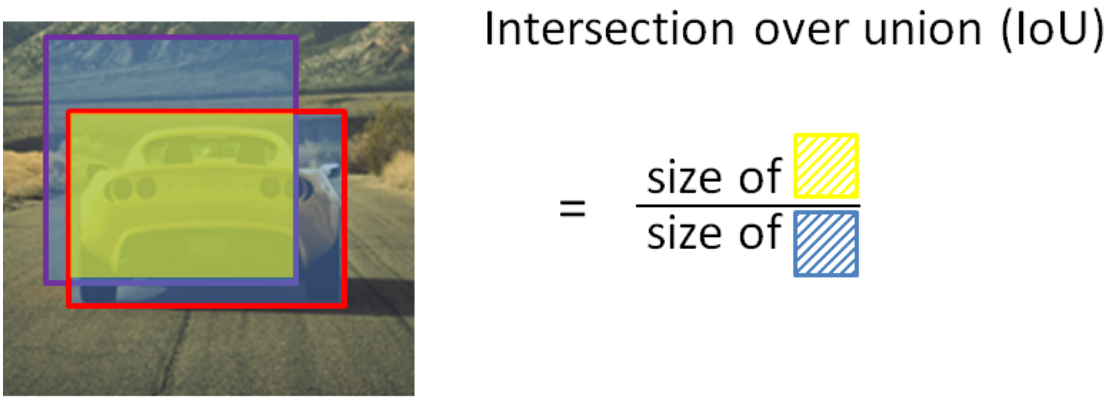


Figure 3.6: Visualization of Intersection over Union.[11]

Objectness Score

The model predicts an objectness score (p_0) for each bounding box using logistic regression. This should be 1 if the anchor box overlaps a ground truth object by more than any other

anchor. If the anchor box is not the best but does overlap a ground truth object by more than some threshold (usually 0.5), the prediction is ignored. The system only assigns one anchor box for each ground truth object. If an anchor box is not assigned to a ground truth object it incurs no loss for coordinate or class predictions, only objectness.[10]

Class Prediction

Each box predicts the classes the bounding box may contain using multilabel classification. For that, independent logistic classifiers are used instead of softmax as it is found that it is unnecessary for good performance. During training binary cross-entropy loss is used for the class predictions. This formulation helps when we move to more complex domains. Using a softmax imposes the assumption that each box has exactly one class which is often not the case. A multilabel approach better models the data.[10]

3.3.3 Loss Function

Although it is not clearly defined in the official tech report published in 2018, YOLOv3 during training, as its predecessors optimizes the following, multi-part loss function:

$$
\begin{aligned}
Loss = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 \\
+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
+ \sum_{i=0}^{S^2} \left[\mathbb{1}_{i}^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c) \right)^2 \right]
\end{aligned}
\tag{3.6}
$$

Where:

- λ_{coord} and λ_{noobj} : represent constants. λ_{coord} is highest in order to focus more on detection.
- S^2 : represents the size of the grid.
- B : represents the number of bounding box predictions for each grid cell.
- $\mathbb{1}_{ij}^{obj}$, $\mathbb{1}_{ij}^{noobj}$ and $\mathbb{1}_{i}^{obj}$: represent masks that indicate if there is an object in a cell or not. $\mathbb{1}_{ij}^{obj}$ is 1 when there is an object in the cell i, else 0. $\mathbb{1}_{ij}^{noobj}$ is 1 when there is no object in the cell i, else 0. $\mathbb{1}_{i}^{obj}$ is 1 when there is a particular class predicted, else 0.
- x_i, y_i : represent the location of the center of the anchor box.
- w_i, h_i : represent the width and height of the anchor box.

- C_i : represents the objectness.
- $p_i(c)$: represents the classification loss.
- $\wedge$: represents the predicted values.

Chapter 4

Implementation of Standard YOLOv3

This chapter explains in depth the code pipeline that uses standard COCO-trained YOLOv3 to detect ships on the Peristera video stream and lists the encountered obstacles that occurred during development. It, also, presents extracted images as results that lead to important conclusions.

4.1 Standard YOLOv3 Code

As mentioned in the previous chapter, standard YOLOv3 achieved great performance even on blurry or low resolution images. The version of YOLOv3 we used was pre-trained on an 80 class dataset called COCO [21]. COCO trained YOLOv3 is considered a "classic" among the machine learning community and is widely used for plenty of object detection applications. Besides, its measured performance shown in Figure 4.1 confirms it.

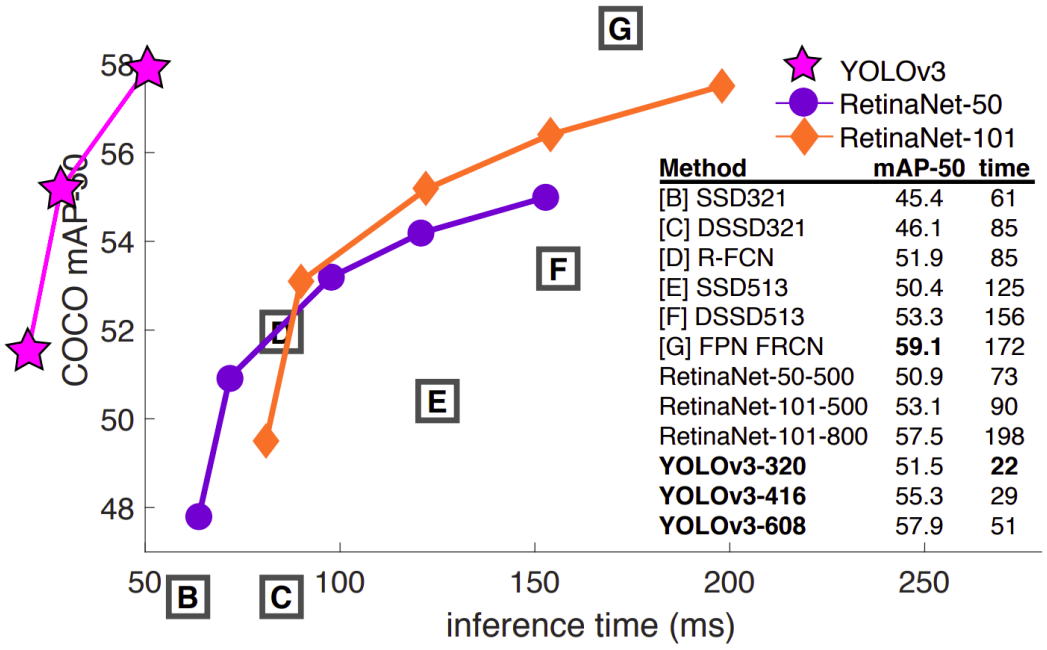


Figure 4.1: Visualization that compares YOLOv3 performance against other models.[10]

One of COCO's classes is called *"boat"* and it is meant to include most vessel types. So, we decided to take advantage of this fact and created an application that receives a video stream from the camera on Peristera and uses standard COCO-trained YOLOv3 to detect vessels. The pipeline of this application is explained in depth below.

4.1.1 In depth Overview

Once the window loads, a new image is created, requesting video feedback from the camera on Peristera. The video stream passes to the main control web-page of the camera, and from there, through a NodeJS proxy to our JavaScript code. When the stream is loaded, an HTML *start* button gets enabled for the user to click and start loading YOLOv3 model. The model is loaded via the *tf.loadLayersModel* command of the TensorFlow.js API [25], by requesting the URL of the model from an HTTP server [26]. Once YOLOv3 is fully loaded, a second HTML *predict* button is enabled for the user to press and trigger the prediction loop. The loop starts by capturing the current date and continues by drawing the current frame of the video stream in an HTML input canvas. Then, the input canvas, along with the user-defined maxBoxes and *scoreThreshold* parameters, pass to the *Modified tfjs-yolo Code*. There, the input canvas gets reshaped to predefined input shape 416x416 and its pixel data get extracted via the *tf.browser.fromPixels* command. The pixel tensor of shape [1, 416, 416, 3] and values normalized between 0 and 1, is then used in the *predict* command that produces the output of the model. The output is an array that contains the three tensors produced by each output layer. These tensors have the following shapes: [1, 13, 13, 255], [1, 26, 26, 255], [1, 52, 52, 255]. Afterwards, the post process function, following the YOLOv3 prediction algorithm described in chapter 3 (see also Fig. 4.3), uses the output array, the maxBoxes, scoreThreshold and iouThreshold parameters, the COCO classes, the original image shape and the predefined anchor boxes to produce and select the best bounding boxes for the given frame. Since we used standard YOLOv3 trained on COCO dataset, we also used the standard COCO anchor boxes: (10×13), (16×30), (33×23), (30×61), (62×45), (59×119), (116×90), (156×198), (373×326).[10] The post process function then scales the selected boxes back to the original image shape and returns them to the main thread of the code, along with the score value and the predicted classes' names. Finally, the predictions and the extracted frame are drawn to a separate HTML output canvas. The code loops by *requestAnimationFrame* command. At the beginning of each loop the new captured date is compared to the previous one in order to calculate the *frame rate (FPS)* that is updated in the inner HTML text. During the whole process, the user gets informed about the progress of each task via HTML text.

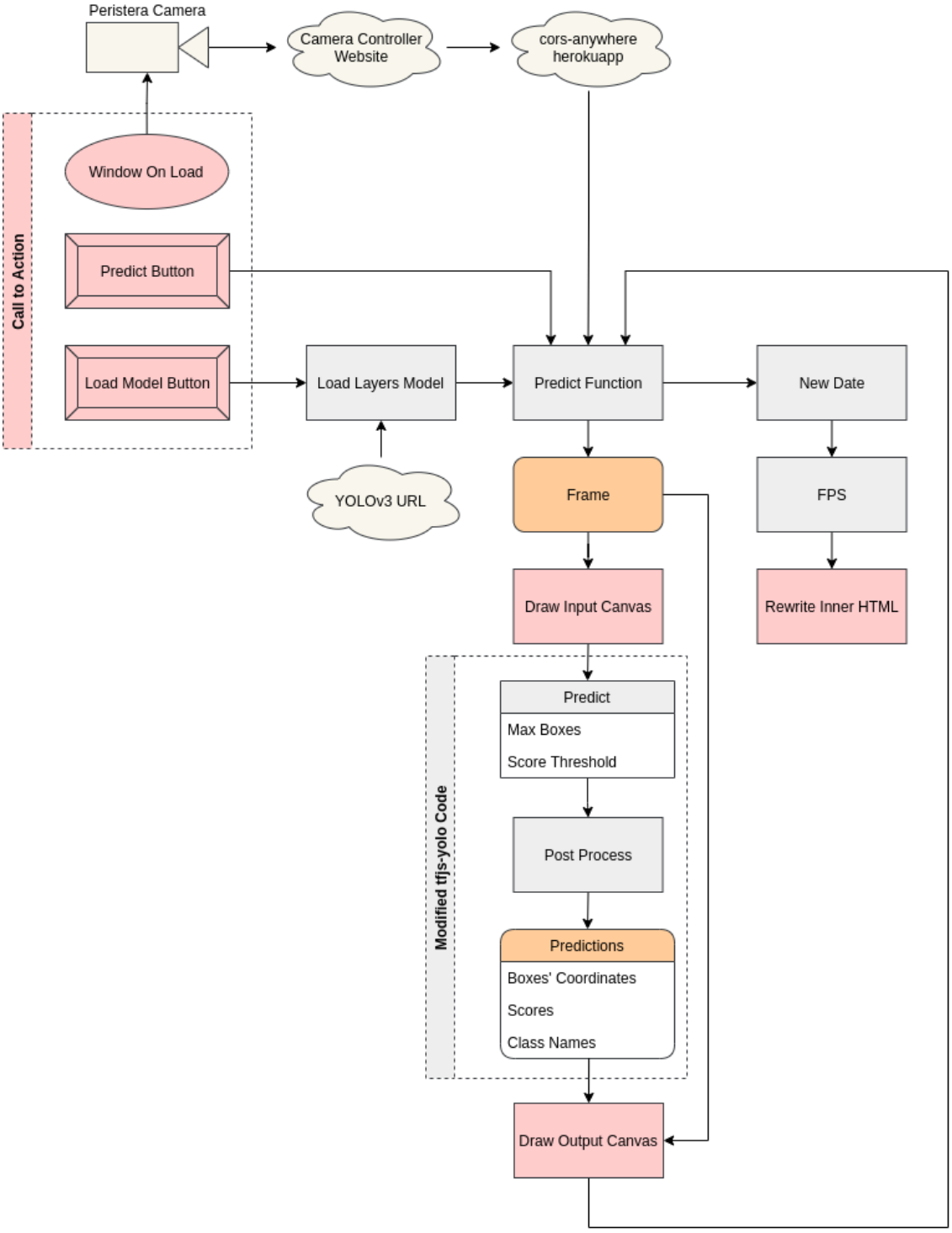


Figure 4.2: Object Detection on Peristera Stream with Standard YOLOv3 Code Pipeline.

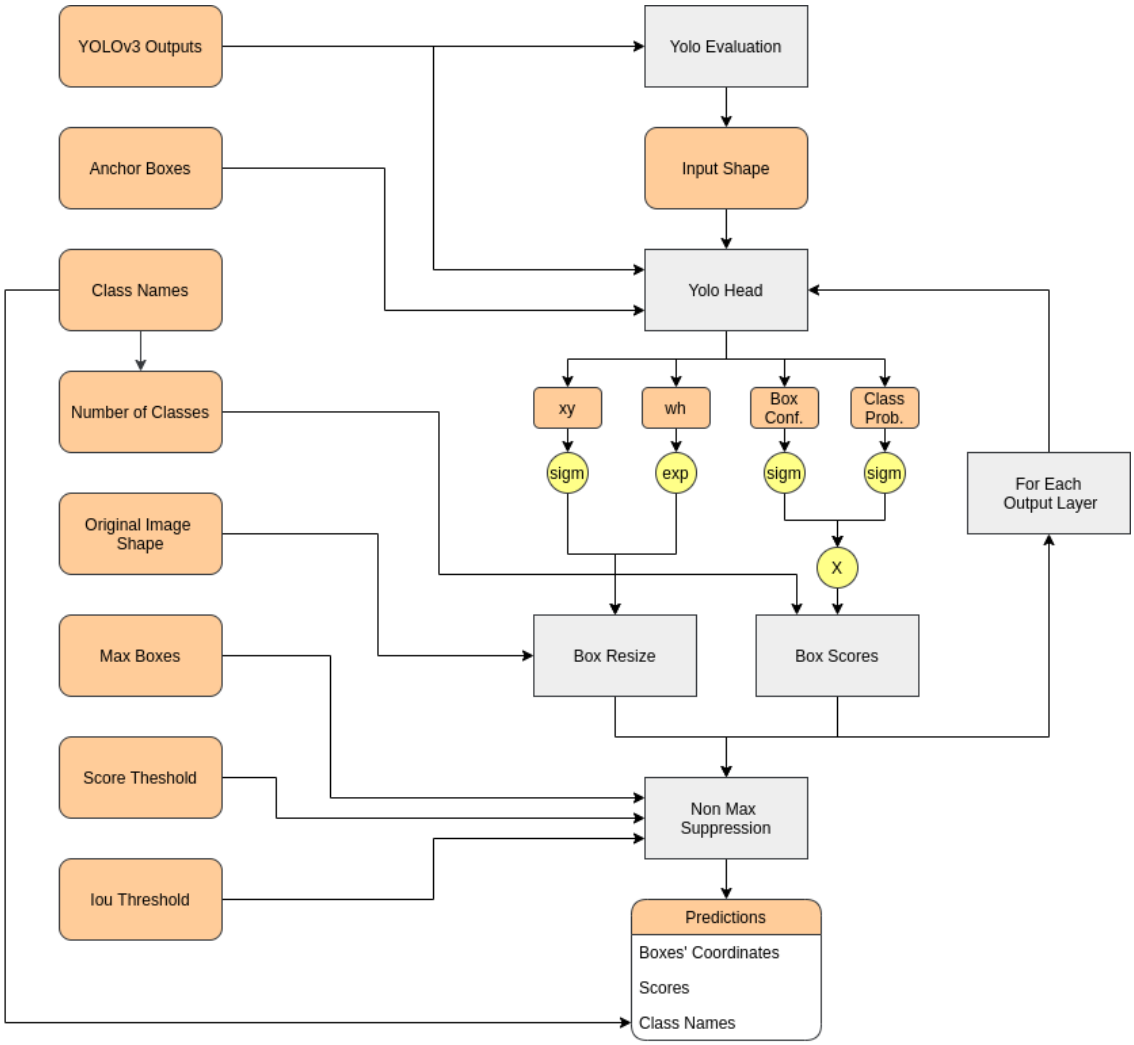


Figure 4.3: Pipeline of YOLOv3 Outputs' Post Process Code.

4.1.2 Encountered Obstacles and Further Discussion

CORS Policy

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request. For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.[27]

As mentioned above, the video stream of Peristera camera passes to its control web-site. So, trying to load the video stream directly from source to localhost results to the following error:

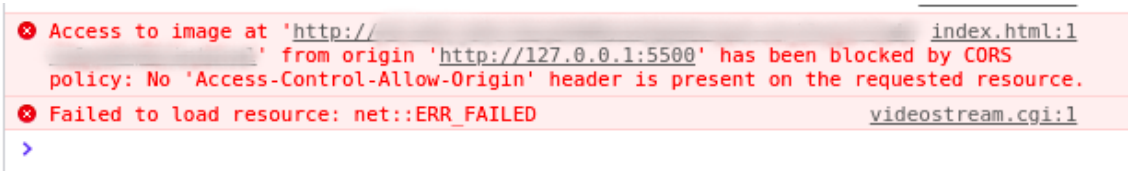


Figure 4.4: CORS Policy error while loading video stream directly from source.

To overcome this obstacle, we ended up to the solution of using an external NodeJS proxy called *CORS Anywhere* which adds CORS headers to the proxied request.[28] As mentioned by the Developer, this demo should only be used for development purposes, which makes this solution not sustainable while deploying the application.

Synchronous Nature of JavaScript

In its most basic form, JavaScript is a synchronous, blocking, single-threaded language, in which only one operation can be in progress at a time. This is convenient when we want things to load and happen right away. For example when applying some user-defined styles to a web-page we will want the styles to be applied as soon as possible. However, if we are running an operation that takes time it is better to push this off the main thread and complete the task asynchronously.[29]

In our case, loading YOLOv3 model was especially time consuming, considering that its weight data are about 237 MB in size.[10] So, we tackled this issue by using *async/await* which creates a promise that is fulfilled when the model is fully loaded. Upon fulfillment, the *predict* button gets enabled for the user to press and trigger the prediction loop.

Looping and Frame Refresh

Instead of using the *setInterval* command that calls the predict function after a specified number of milliseconds (equal in each loop), we decided to go with the *requestAnimationFrame* command that calls the predict function as soon as the previous computations are over. We found out that by using this looping command we eliminated the errors that occurred due to unfinished computations and we accelerated the FPS because the program does not have to wait for the time interval to finish in each loop. This meant that FPS value is not constant and had to be calculated via the following equation:

$$FPS = \frac{1000}{Date_{thisLoop} - Date_{lastLoop}} \tag{4.1}$$

In some cases we noticed that between each loop Pristera video stream was not able to refresh (probably due to low internet speed), leading to unchanged predictions.

Double Canvases

We noticed that by using a single canvas to draw both the input frame and the predictions we stumbled upon a logical error, where as soon as the predictions are drawn to the canvas, the predict function immediately loops and draws on top the next input frame. So, we came up with the solution to use two canvases, one for the old input frame and the predictions and one for the new input frame.

4.2 Results

This section presents extracted images as results from our code that lead to important conclusions.

Throughout the tests we did, we had to readjust the maxBoxes and *scoreThreshold* parameters resulting to different outputs. We also noticed that, although the width and height of the video stream remained the same, there was deviation in the received frame's resolution. There are several extracted images that appear to be pixelated and blurry. For example in Figures 4.5 and 4.6 we can see two correct prediction one on high and one on low resolution. The main difference is depicted in the score values, where the low resolution image recognized the person in front with probability 10.5%, which is quite low compared to the high resolution score of 89.7%.

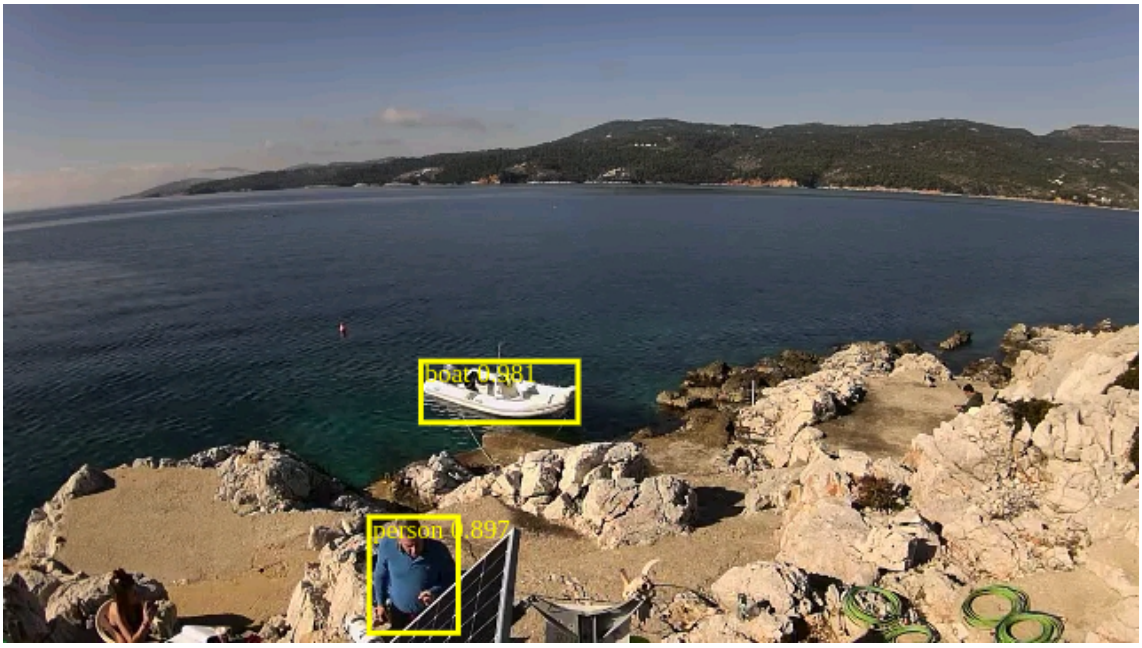


Figure 4.5: Correct predictions with standard YOLOv3 on high resolution image.

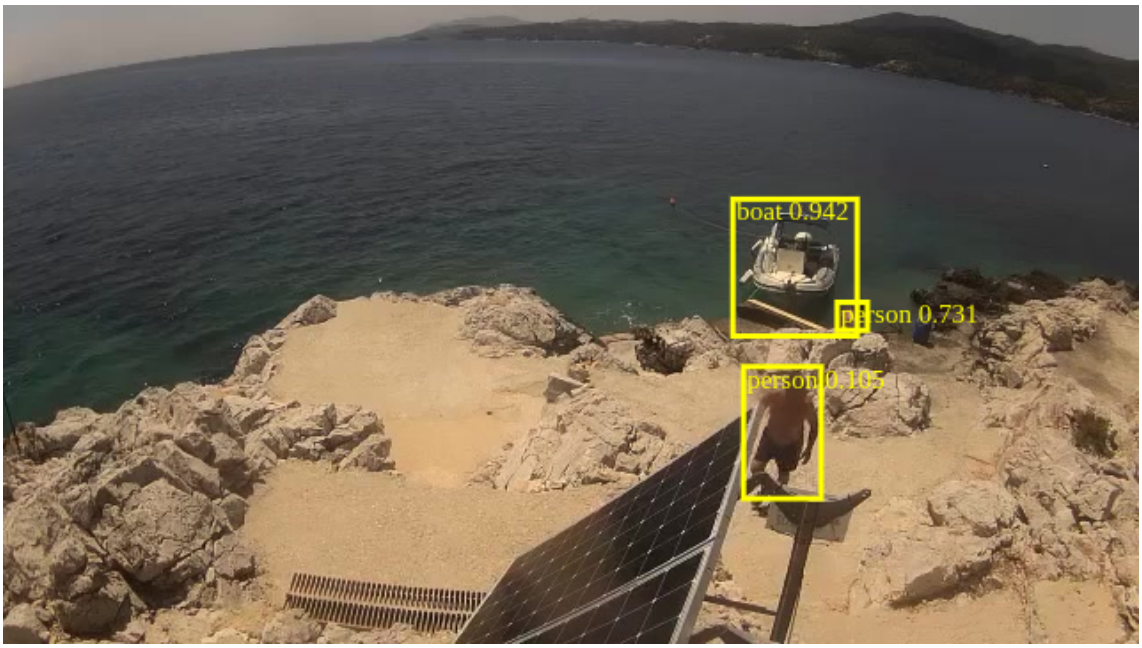


Figure 4.6: Correct predictions with standard YOLOv3 on low resolution image.

In some cases the resolution was so low that the model could not recognize the inflated boat that was perfectly detected a few frames ago in Figure 4.6.

Figure 4.7: Insufficient predictions with standard YOLOv3 on low resolution image.

As can be seen in Figure 4.8, we also managed to capture predictions with two vessels at great distance from the camera. The achieved scores are relatively low (4.6% and 10.3%) but considering the image's resolution this is a really impressive result.

Figure 4.8: Predictions with standard YOLOv3 of small objects on low resolution image.

The video from which the frame in Figure 4.8 was extracted was quite helpful for our experiments because it led us to important discoveries. First of all, we were able to test the limits of standard YOLOv3 in not ideal lighting conditions. As the vessel on the right passed through the reflection of the sun's light beam (Figure 4.9), YOLOv3 was struggling to detect it. Only when we lowered the score threshold to a value under 0.01 YOLOv3 successfully detected it, but also made incorrect predictions, as it is depicted in Figure 4.10.

Figure 4.9: Standard YOLOv3 predictions with one vessel in not ideal lighting conditions.

Figure 4.10: Predictions of standard YOLOv3 on lowered score threshold with one vessel in not ideal lighting conditions.

Secondly, in cases where two vessels were too close to each other and at great distance from the camera (Figure 4.11), YOLOv3 was unable to detect them, even in low score threshold values.

Figure 4.11: Predictions of standard YOLOv3 with two vessels close to each other.

Finally, we can summarize the conclusions we extracted from our experiments in the following list:

- YOLOv3 is a robust and fast object detection model that can perform well even on low resolution images.

- It also showed great results detecting small objects.
- It showed moderate results on images with not ideal lighting conditions.
- It was unable to detect small objects whose outline shape was not clear because they interfered with each other.
- YOLOv3 is a computationally demanding model, especially for a web browser, resulting in a relatively low frame rate of 0.8 FPS.
- COCO trained YOLOv3 exports tensors containing information about 80 classes, which adds to the need for more complicated computations.
- With COCO dataset we cannot extract information about the specific type of a detected vessels.

All the above conclusions led us to a second object detection application with YOLOv3, where we performed transfer learning to the already COCO trained model with a small custom dataset of images and videos captured by the camera on Peristera. This implementation is analyzed thoroughly in the next chapter.

Chapter 5

Transfer Learning on YOLOv3

The results of the previous chapter showed that, although COCO Dataset contains a "boat" class, it is not specialized in ship detection, forcing YOLOv3 to make excessive computations for other classes and to perform insufficiently under not ideal lighting conditions. Moreover, the standard YOLOv3 implementation does not provide any information for the detected vessel's type. This leads to the second implementation of performing transfer learning to YOLOv3 with a small dataset of vessel images and videos captured by the land camera in Peristera. The vessel dataset creation procedure, the transfer learning training pipeline and the custom loss function used for computing deviation from the ground truth labels, are analyzed in this chapter. Additionally, the final section presents training results and important conclusions on them.

5.1 Dataset Creation

The first step to train a great object detection model is to have good quality data. This section presents the steps we followed to create a small but comprehensive and efficient vessel dataset.

5.1.1 Classes Selection

The first step someone has to follow in order to create an object detection dataset is to determine what kind of objects the model will predict. COCO dataset contains a "boat" class (among 80 different classes) which is mostly centered around big commercial ships. This vague one-class approach limits the information we can extract from a prediction and lowers the probabilities for the model to successfully detect smaller vessels such as sailboats, plane-crafts etc.

We decided to take a different approach by choosing a multi-class dataset. By analyzing the usual liner passenger ship routes (see Figure 5.1) and considering the narrowest width of the passageway between Peristera and Alonissos (about 600 meters), we opted to exclude classes that describe big ships.

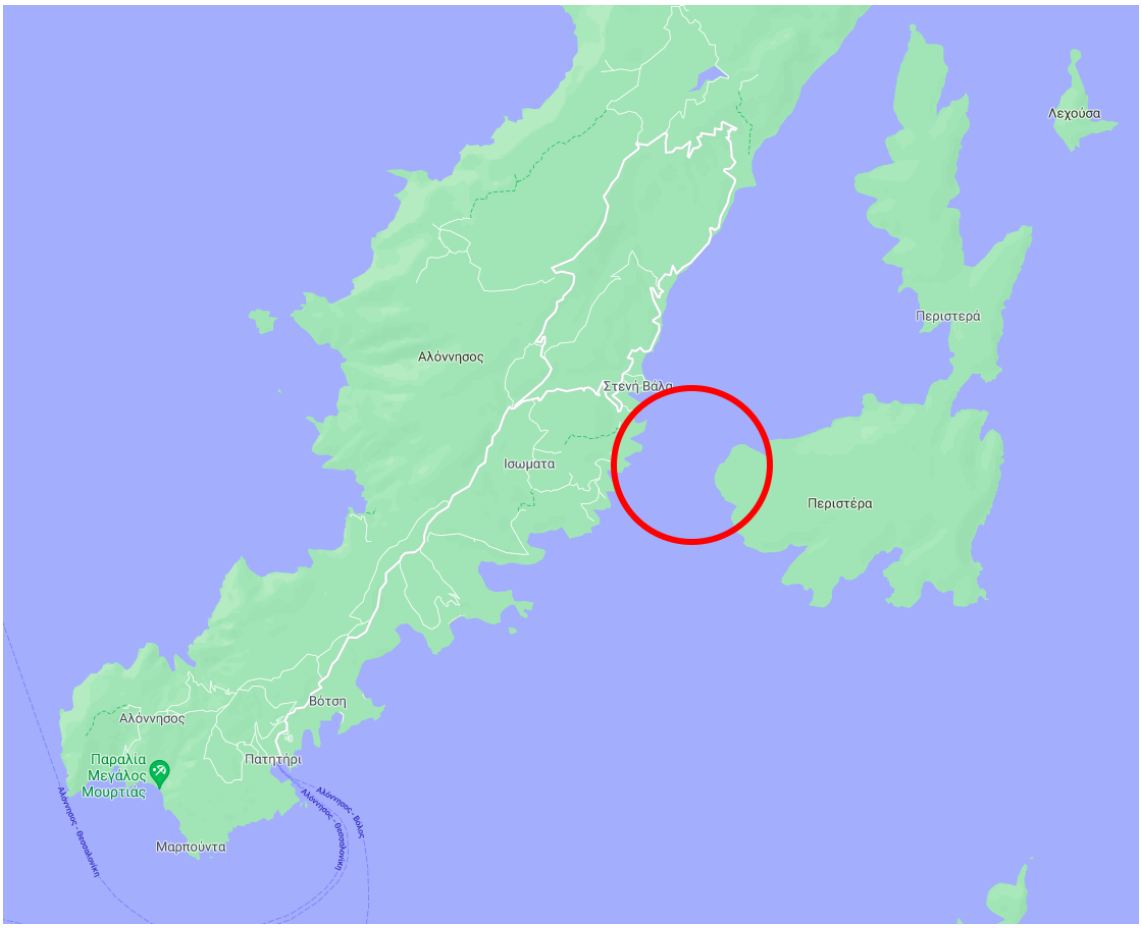


Figure 5.1: Map that illustrates routes of liners around Alonissos and Peristera.

Finally, after examining the available videos and images captured by the Peristera camera, we concluded to the four vessel classes listed below:

- **Fishing Vessel**
- **Inflatable Boat**
- **Sailboat**
- **Planing Craft**

5.1.2 Labelling

To build our dataset, we gathered in total 728 images captured by the Peristera camera. Many of them were extracted by converting videos into individual frames using the *FFmpeg*[30] application. After renaming them to a $x.jpg$ format, where x is a whole number between 0 and 727 (convenient in code), we labeled all of them by hand using the *LabelImg*[31] application.

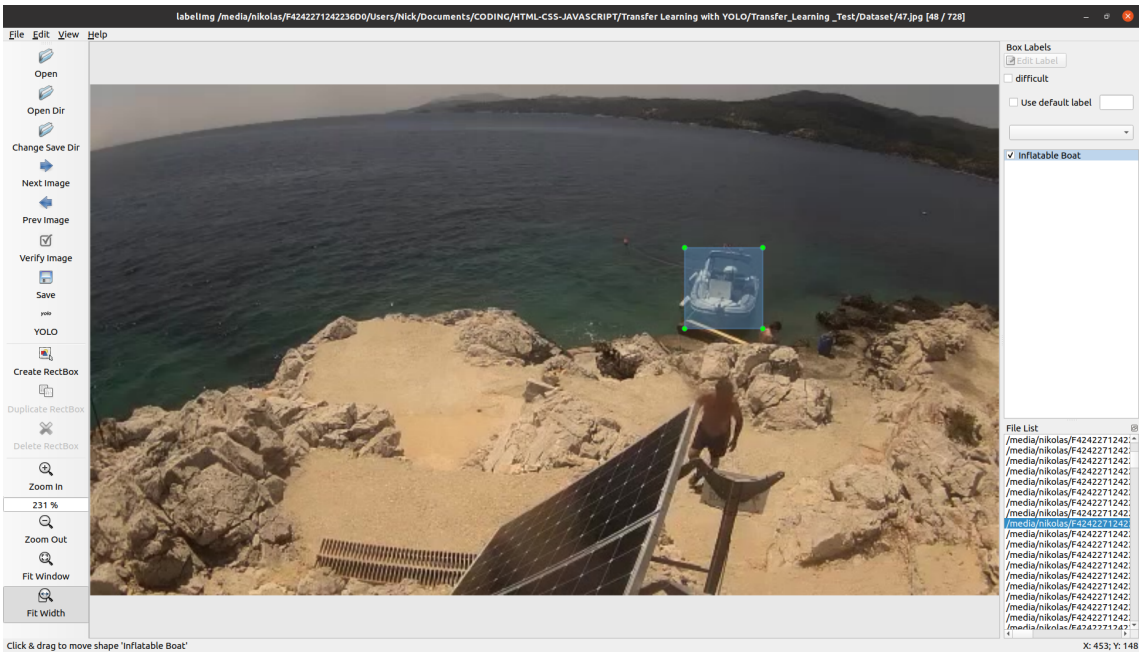


Figure 5.2: The LabelImg workspace.

Fortunately, LabelImg offers a YOLO compatible save format that exports xml.txt files for each image. A typical example with 4 labels in one image can be seen in Figure 5.3, where the the first column contains the number that refers to the labeled class for each box, the second and third columns contain the normalized center coordinates and the last two columns contain the normalized width and height.

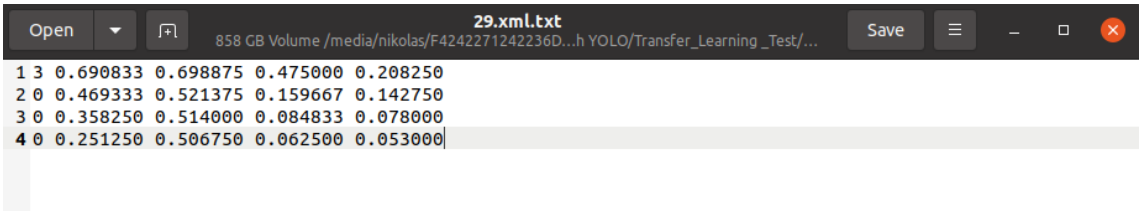


Figure 5.3: LabelImg YOLO compatible output file for an image that contains 4 labels.

5.1.3 Label Preparation

Some images contained more than one vessel and so, consequently, were characterized by more than one boxes. Each box is represented by a different line in the extracted xml.txt file of the image. As we imported these files in the main transfer learning pipeline, each image's boxes resulted to a tensor of different shape. So, when we tried to apply mathematical operations on a batch of images represented by tensors of different shape, errors occurred. This is an obvious mistake that originates from the way the TensorFlow.js tensor system works, which is different from how nested arrays work, in that tensors should have constant size in a dimension.

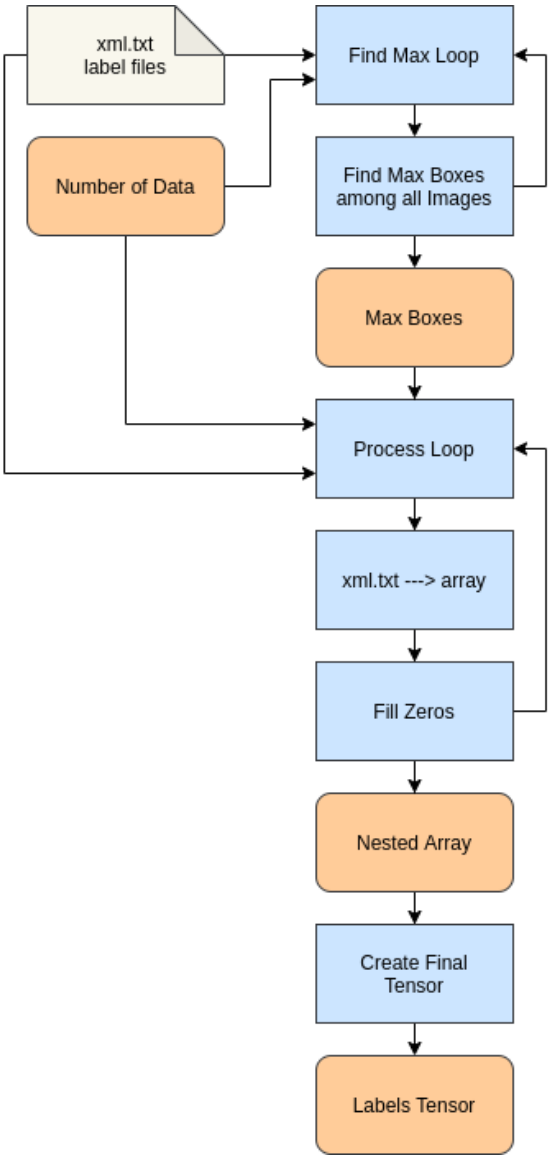


Figure 5.4: Label preparation code pipeline.

To tackle this issue, we created a label preparation pipeline that is visualized in Figure 5.4. We decided to use Node.js instead of JavaScript, because we found it more suitable to deal with local storage file accessibility. A *Find Max Loop* loads one by one the xml.txt files via the *file system (fs)* library, reads each line via the *readline* library and updates a *MaxBoxes* variable. After all files are examined a new *Process Loop* loads them again, transforms them into arrays and fills with zeros any spare positions until it matches the size of the array containing the most boxes. The loop, finally, produces a total nested array that was originally written to a JSON file. This approach was effective in tests we did with a few images (about 10), but created a lot of memory problems while trying to load the whole dataset. More specifically, this code was used at first as a separate pipeline that produced a JSON file which was loaded into the main transfer learning thread and then into the *Pre-process of True Labels* module (see 5.2.2). This module, which is responsible for transforming the comprehensible labels to YOLOv3 compatible outputs, could not fulfill the task for the whole dataset at once, resulting to the error of Figure 5.5.

Figure 5.5: Error while trying to load all labels via a single JSON file.

The solution we came up with, was to incorporate the label preparation pipeline into the main thread which now extracts a tensor of shape [728, 4, 5] instead of a JSON file. Then, a loop slices parts of this tensor via the *tf.slice* command and feeds them to the Pre-process of True Labels module. The module is called multiple times instead of one, but has to apply computations on smaller batches, resulting to better memory allocation.

5.1.4 Anchor Boxes

As discussed in chapter 3 the purpose of the anchor boxes is to detect multiple objects of different sizes for which the center is located in the same cell. Anchor boxes are computed specifically for a whole dataset with *K-Means Clustering* [32].

The field of view of the camera in Peristera is quite wide and the distance between the two islands exceeds 1 km. Considering that there are, also, frequent visits by the technical team to repair and inspect the equipment in Peristera, we have images containing vessels at many different perspectives and distances from the camera. In addition, sailboats are contained in more square-like boxes than the other types of vessels, due to the height of their sails. This results to a need for anchor boxes that cover a wide range of object shapes and sizes.

Unfortunately, the tests we did with custom anchor boxes showed poor results, which is expected, considering what the YOLOv3 author states: "Only if you are an expert in neural detection networks - recalculate anchors for your dataset for width and height. If many of the calculated anchors do not fit under the appropriate layers - then just try using all the default anchors."[33] So, for our case, we decided to use the already calculated COCO anchor boxes that cover a wide range of object shapes and sizes. The shapes of our dataset's anchor boxes are: (10×13), (16×30), (33×23), (30×61), (62×45), (59×119), (116×90), (156×198), (373×326).[10]

5.2 Transfer Learning Pipeline

After creating the vessel dataset, the next step was to train our model. Considering the limited computer power of our machine, we decided to take advantage of the effectiveness the COCO-trained YOLOv3 shows and to not train our model from scratch. So, we followed the transfer learning procedure described in chapter 2 and more specifically the single model approach. The tests we did with JavaScript showed quite slow training speed and even unexpected termination in cases where the internet disconnected. So, we decided to use Node.js to train our model.

5.2.1 Main Thread

The main thread of our code is responsible for loading the dataset, creating, compiling and then training the new model. During the whole process, the user gets informed about the progress of each task via logs in the terminal.

Loading Labels

The label xml.txt files are loaded in the *label preparation* module (see 5.1.3) which transforms them into a three-dimensional tensor of shape [m, T, 5], where m is the number of dataset images, T is the max boxes per image constant computed by the label preparation code and number 5 represents the parameters of each box (class index, x and y center coordinates, width and height). Slices of the tensor is then used along with the input shape, the anchor boxes and the number of classes in a loop that calls a separate *pre-process true boxes* (see 5.2.2) module which transforms the inputs into YOLOv3 compatible nested arrays, that are then reshaped into the final shapes: [m, 13, 13, 27], [m, 26, 26, 27], [m, 52, 52, 27].

Creating and Compiling the new Model

YOLOv3 is loaded via the *tf.loadLayersModel* command, by requesting the URL of the model from an HTTP server [26]. This happens in an asynchronous way via the *async/await* format because of the weights' large size. Once the promise is fulfilled, a loop scans the whole model to find the target layers that will be cut off. After thorough research we decided to discard only the last layers of each prediction thread, specifically the layers: *"conv2d_59"*, *"conv2d_67"* and *"conv2d_75"*. A new loop creates the three new layers with depth 27 ($num_{layers} \cdot (num_{classes} + 5)$) that will from the new model head. The truncated YOLOv3 along with the new head are combined via the *tf.model* command to create our new model. Before compiling the model all the layers except for the new head are given the attribute *trainable = false*. In that way we can still take advantage of the already gained "knowledge" of YOLOv3, that is mostly connected with basic image data manipulation like recognizing edges, shapes etc. and stays the same whether we intend to predict 4 or 80 classes. The amount of time and computational power needed while training are also reduced significantly, since only the new head weights are updated. Specifically, we managed to reduce the trainable parameters from 61592497 to just 48465. Finally, to compile our model, we decided to use the following configuration:

Table 5.1: Model Configuration for Training.

Configuration	**Value**
Optimizer	ADAM w/ learning rate = 0.001
Loss	Custom Loss Function (see 5.2.3)
Metrics	Accuracy

Training the new Model

For training, we decided to use the *fitDataset* command instead of the usual *fit*. The difference between this two commands is that fit requires two tensors, one for the inputs of the model and one for the targeted outputs that contain the whole dataset, while

fitDataset generates small batches of dataset every time it is called. Generally, fitDataset is widely used in TensorFlow.js when the training dataset is relatively large. Considering our model has input shape 416x416, each tensor containing the pixel data of each dataset image will be of shape [1, 416, 416, 3]. These tensors are quite big to use them in intensive computations, especially stacked together. In fact, while experimenting with the fit command we reached a limit of 197 images where any more would crash the program.

While training, fitDataset calls a *generator function* which returns a *generator object* that contains a batch of images' pixel data tensor and their respective labels, represented as outputs of each of the model's three prediction layers. Generators are functions that can be exited and later re-entered. Their context (*variable bindings*) will be saved across re-entrances. Generators in JavaScript – especially when combined with Promises – are a very powerful tool for asynchronous programming as they mitigate – if not entirely eliminate – the problems with callbacks.[34] The number of examples the generator function creates and feeds to the fitDataset command is determined by the *batch* attribute of the *tf.data.generator* command. When the generator function is called, the next image in line is loaded via the *file system (fs)* library, gets decoded via the *tf.node.decodeImage* command and resized to the model's input shape by the *tf.image.resizeBilinear* command. The generated labels are the reshaped outputs of the *pre-process true boxes* module. For example, if we consider batch equal to 2, the generator object will have the following format:

- **Generator Object:**
  - xs property:
    - pixel tensor of shape [2, 416, 416, 3]
  - ys dictionary property:
    - "conv2d_0_NEW": label tensor of shape [2, 13, 13, 27]
    - "conv2d_1_NEW": label tensor of shape [2, 26, 26, 27]
    - "conv2d_2_NEW": label tensor of shape [2, 52, 52, 27]

The memory allocation while training is monitored via the *tf.memory* command and the whole procedure is timed and then transformed to *hrs : mins : secs : ms* format.

Once the model has finished training, the *save* command is called which generates a JSON file that contains the model's architecture and a .bin file that contains the weight parameters.

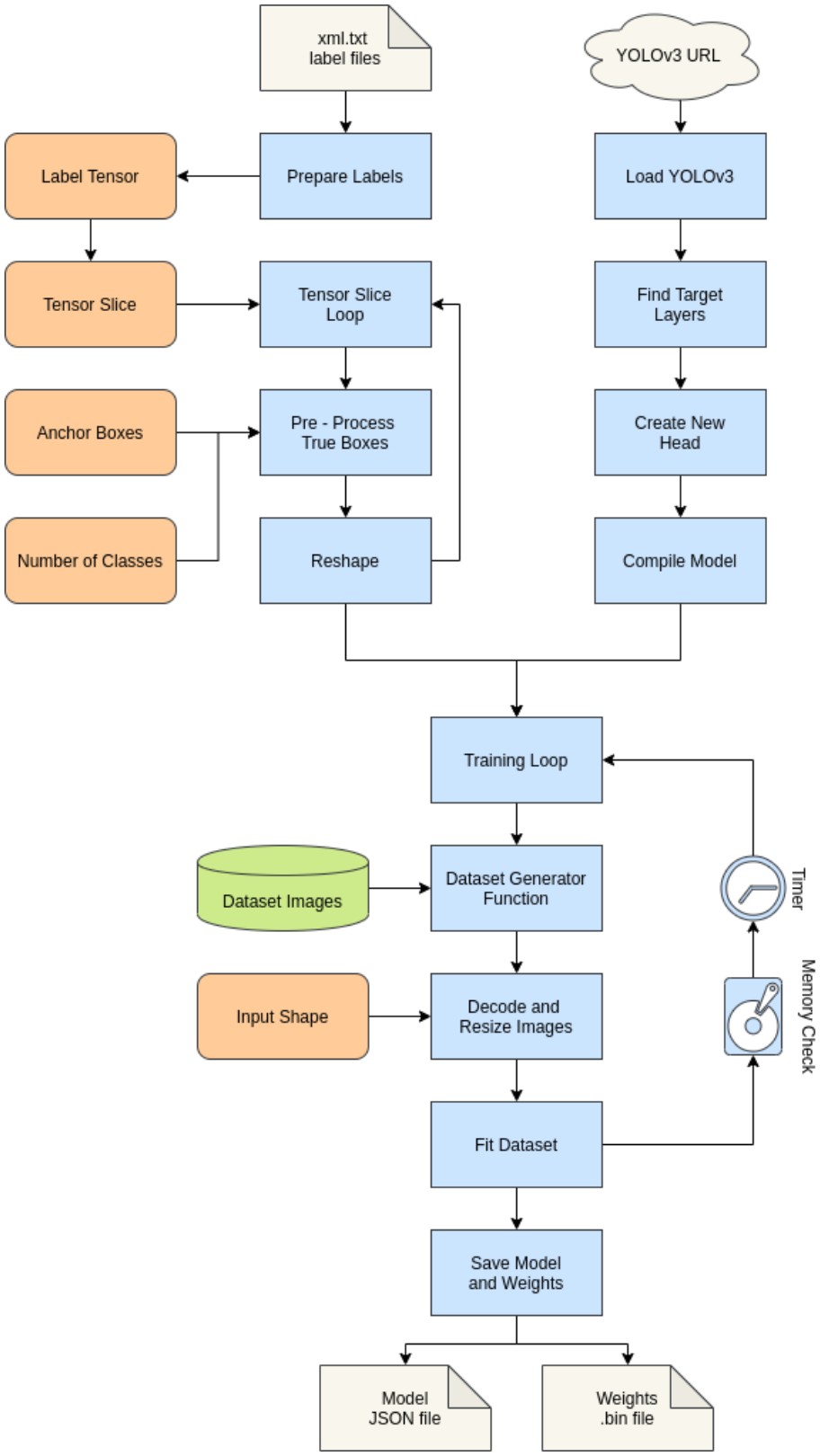


Figure 5.6: Transfer learning main thread code pipeline.

5.2.2 Pre-process of True Labels

The *pre-process true boxes* module is responsible for transforming the comprehensible labels created by the LabelImg application to YOLOv3 compatible outputs. We created this module based on a 2018 YOLOv3 training project developed in Python with TensorFlow/Keras.[35] To achieve that, we had to convert parts of the Python code to JavaScript and in some cases to replace them entirely with JavaScript equivalent code snippets.

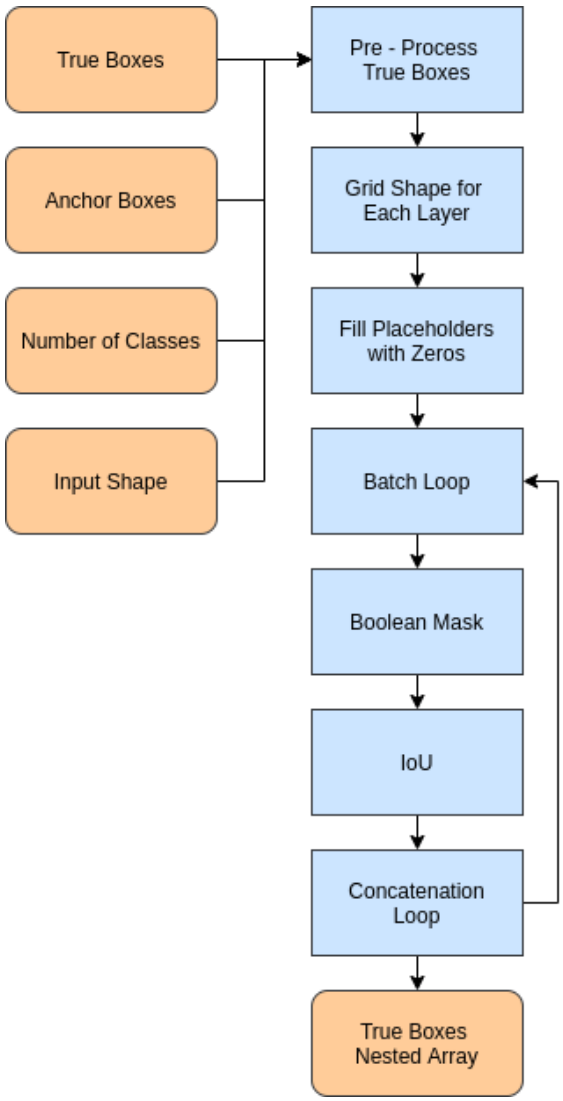


Figure 5.7: Visualization of Pre-process true labels code pipeline.

Firstly, the module multiplies the normalized boxes' parameters with the input shape of the model in order to map them between values 0 and 416 and continues by creating the grid shapes 13x13, 26x26 and 52x52 for each prediction layer. These shapes are used in the *tf.zeros* command that fills the three output placeholder tensors with zeros. A loop, then, receives the boxes of each image separately and applies *tf.booleanMaskAsync* to discard zero rows. The new box tensors are reshaped and used along with the anchor boxes to calculate IoU (see Figure A.1). The anchor box with the best IoU score is picked as the most suitable for a particular box and determines the position of the box parameters in

the output placeholder tensors. Finally, all box tensors are converted to arrays and get placed into three nested arrays of shape [m, 13, 13, 3, 9], [m, 26, 26, 3, 9] and [m, 52, 52, 3, 9] that represent the true outputs of our model for each prediction layer respectively.

5.2.3 Loss Function

The most crucial component of training a CNN is the loss function, that should describe as best as possible the deviation of the predictions from the targeted labels. A model with a mismatched loss function is not able to learn regardless of the training time and the size of the dataset. The loss function created for an object detection task is more complicated than a one created for image classification, because the model not only has to predict correctly the class, but also the center coordinates, the width and the height of the boxes.

Standard YOLOv3, although it is not clearly defined in the official tech report published in 2018, uses as loss function the equation 3.6. The difference with its predecessors is that YOLOv3 produces 3 outputs and so, has to slightly modify and apply the loss function for each one of them.

TensorFlow.js does not support an equivalent pre-made loss function that covers the needs of YOLOv3, which is expected considering that YOLO is written in a completely different framework called Darknet [23] which is developed in C. So, we created a custom loss function in JavaScript based on a 2018 YOLOv3 training project developed in Python with TensorFlow/Keras.[35] To achieve that, we had to convert parts of the Python code to JavaScript and in some cases to replace them entirely with JavaScript equivalent code snippets. One of the first obstacles we encountered was that the Python code feeds all three predicted tensors to the loss function as a list, whereas TensorFlow.js applies the loss function for each output tensor separately. So, instead of creating three different custom loss functions, we inserted *if* and *switch* statements in a single one, that depending on the tensor shape lead to different code snippets. The loss value that will be minimized by the model is calculated as the sum of all individual losses.

Another challenging aspect was to combine the asynchronous character of JavaScript with the limitations of TensorFlow.js library. Training a model is an extremely memory demanding procedure and so, it is crucial that no memory leaks (periodic increase in memory usage) occur. TensorFlow.js offers a tool called *tf.tidy* that executes the provided function contained in it and after it is executed, cleans up all intermediate tensors allocated by the function except those returned by it. The problem is that tf.tidy can not contain any asynchronous code. That means that the whole loss function must be created as one synchronous continuous thread with no external modules. In other cases, such as the *tf.booleanMaskAsync* command, TensorFlow.js offers no synchronous versions making it even more challenging to bypass parts of the Python code.

The final pipeline of our loss function can be seen in Figure 5.8. After the model applies convolutional computations to a batch of pixel tensors, the produced output and the target tensor for each prediction layer are fed to loss function. Firstly, the function creates an *object mask* $\mathbb{1}^{obj}$, that essentially determines which parts of the label tensor contain useful information. The prediction tensor gets processed in a *yolo head* function same as the one depicted in Figure 4.3. Loss function continues by calculating the *box loss scale* λ_{coord}. The raw label tensor width and height values are divided by the anchor boxes, multiplied by the input shape and then passed through a log function. With the help of the object

mask and the *tf.where* command, unwanted values are replaced with zeros. The final width and height values are multiplied and subtracted from 2 to produce the box loss scale. A batch loop, then, calculates IoU (see Figure A.1) between the predicted and true box for each image and calculates the *ignore mask* $\mathbb{1}^{noobj}$ based on a *ignore threshold* value. Finally, loss function applies the equation 3.6 where *binary crossentropy* is used to avoid exp overflow and sums the calculated batch-averaged loss values for box center coordinates, width and height, confidence and classification.

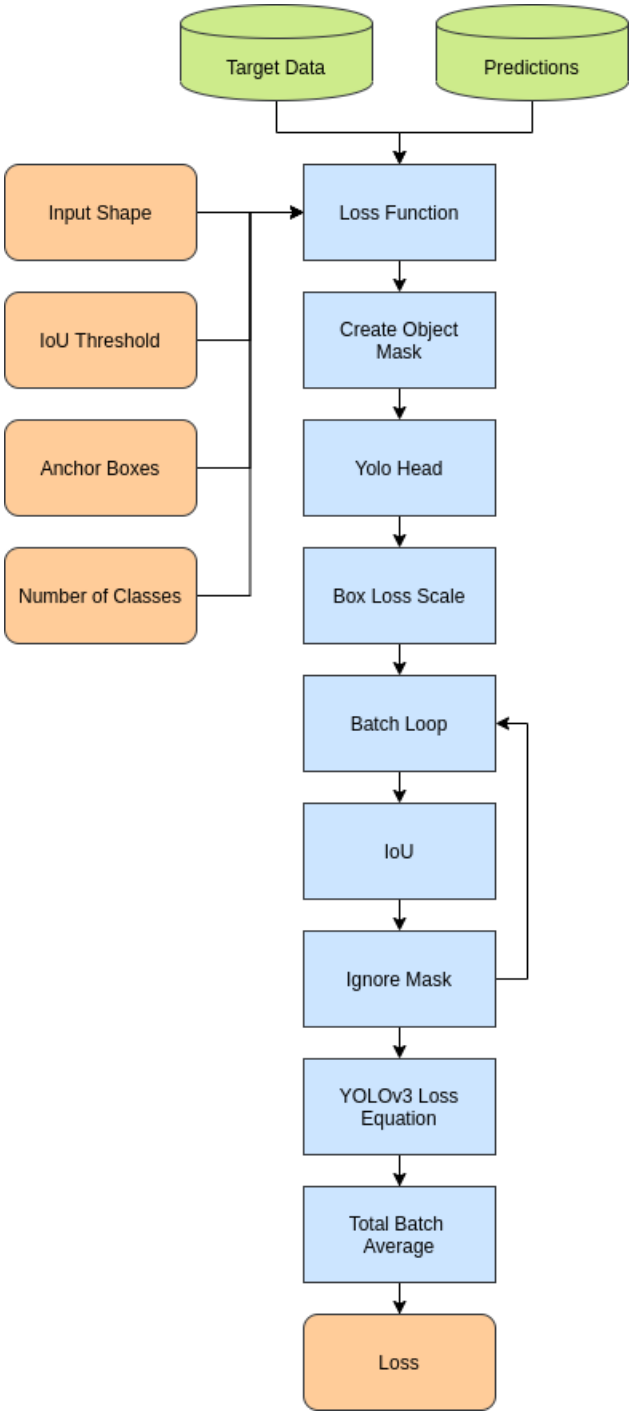


Figure 5.8: Loss Function pipeline.

5.3 Training Results

The whole transfer learning procedure was meant to optimize YOLOv3 for vessel detection. The new model would replace standard YOLOv3 in the detection pipeline described in chapter 4. In addition to being able to detect vessels, as is standard YOLOv3, the new model would classify the predictions between the four selected classes: "Fishing Vessel", "Inflatable Boat", "Sailboat" and "Planing Craft".

Unfortunately, the training procedure was never completed due to a *memory leak* in our code. In computer science, a memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in a way that memory which is no longer needed is not released.[36] While using TensorFlow.js, a memory leak is usually caused by a periodic increase in the amount of created tensors. The library provides a tool called *tf.memory* that returns memory info at the time called in the program. The result is an object with the following properties:[25]

- *numBytes:* Number of bytes allocated (undisposed) at this time.
- *numTensors:* Number of unique tensors allocated.
- *numDataBuffers:* Number of unique data buffers allocated (undisposed) at this time, which is equal to, or lower than the number of tensors.
- *unreliable:* True if the memory usage is unreliable.

The most "suspicious" parts of a code that can cause memory leaks are loops. In our case we can confirm that there are no loops outside of the training function that generate additional tensors. So, that means that the cause of our problem is located in the training loop. We can also confirm that the generator function does not create excessive tensors, as we performed tests with the *fit* command (that does not use a generator function) and the memory leak still remained. Finally, we can probably also exclude the custom loss function as the whole code snippet is contained in a *tf.tidy* command that deletes unused tensors. The only value that the loss function returns is the loss value itself, that is passed to the optimizer and then supposedly is deleted automatically. To be honest, we were unable to figure out what causes the memory leak. Figure 5.9 depicts the increase in the amount of tensors after each training iteration.

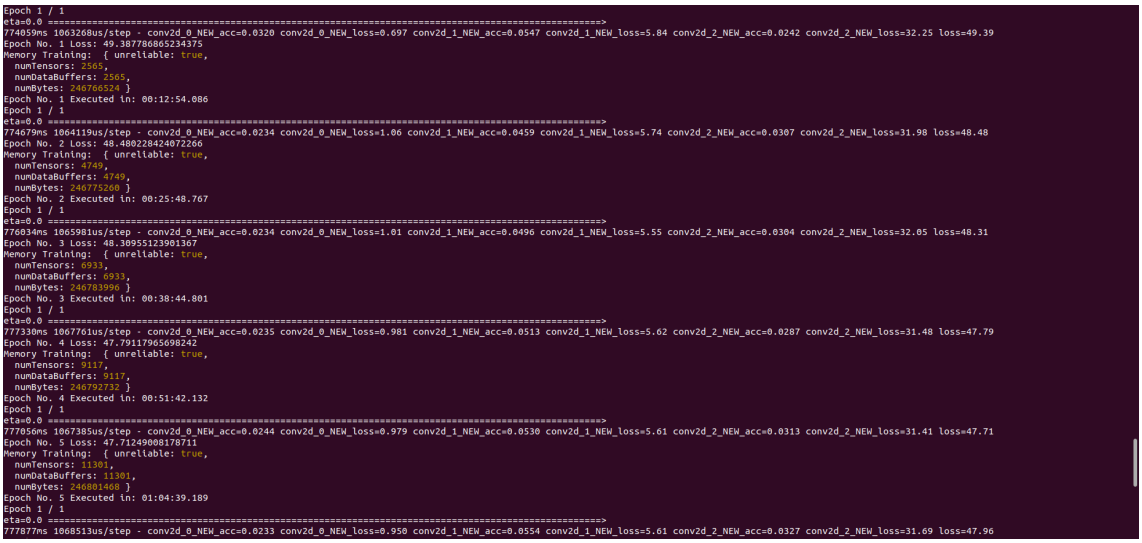


Figure 5.9: Screenshot that shows the memory leak occurring while training the model.

In addition, we encountered a problem where the program crashed when the batch number was set too high (e.g. 10). This is caused mainly due to the capabilities of our machine, but also due to the computational intensity of YOLOv3 and the limitations of Node.js (JavaScript) and TensorFlow.js.

Nevertheless, we decided to proceed with training the model several times, adjusting the learning rate, the batch size and the number of data every time the program crashed. We managed to complete a training session with 11 epoches, using all 728 images with batch size equal to 1 and a learning rate equal to 0.001. As it is depicted in Figure 5.10, the average loss value (far right) is decreasing after each epoch, suggesting that the model is "learning".

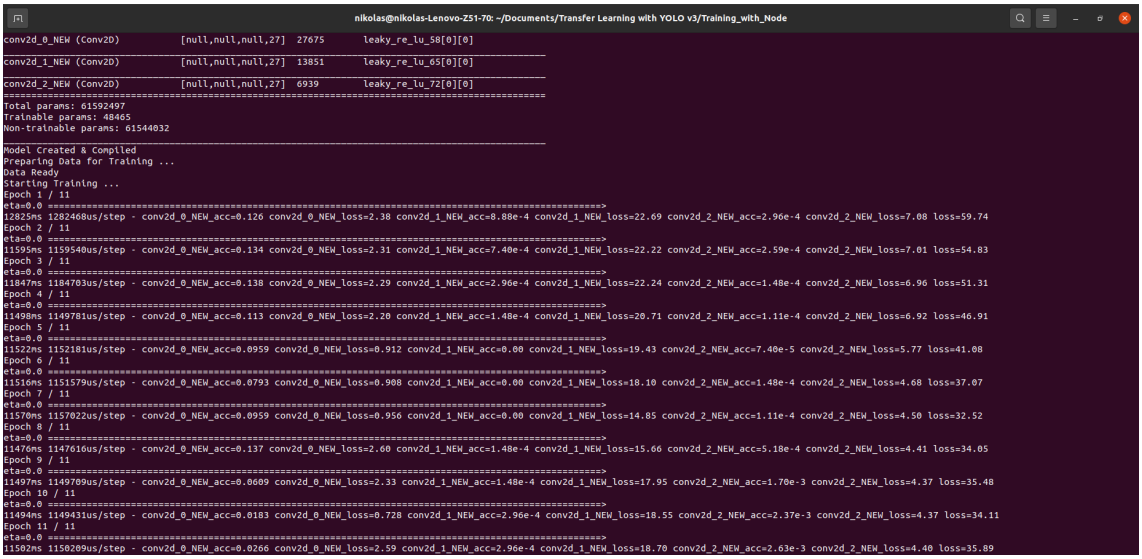


Figure 5.10: Training with 11 epoches and learning rate equal to 0.001.

As it is obvious, this training session is not enough for our model to be capable of performing vessel detection on the video stream from Peristera.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, the implementation of an object detection application using neural networks to monitor vessel traffic over the Peristera shipwreck was investigated. Specifically, after thorough research, the most convenient tools were selected for the development state and a comparison experiment was conducted between YOLO and SSD declaring YOLOv3 model as the most suitable choice for the goal. The model was used in two implementation: as standard YOLOv3 pre-trained on COCO dataset and in a transfer learning pipeline with a custom dataset. The extracted conclusions can be summarized in the following paragraphs.

JavaScript and TensorFlow.js

As mentioned in the previous chapters, JavaScript is not commonly used for machine learning purposes. The reasons are more apparent while trying to develop deep learning applications, especially using computationally intensive models like YOLOv3. Even in Node.js the performance was far more inferior than in other backend programming languages like Python and C++. Data manipulation is also, much more difficult in JavaScript than in Python, resulting to unnecessary computations only to transform the data in the desired shape. On the other hand, certainly, the availability of DOM data (videos, images, canvases etc.) on the web allows easier accessibility to data not stored locally.

TensorFlow.js is a relatively new, but ever evolving machine learning library, following the steps of Python's TensorFlow. Although we can confirm that it is a "well-rounded" library, we cannot help but notice that some of the Python's TensorFlow features are missing, complicating tasks, such as developing a custom loss function in JavaScript. In addition, the fact that some commands do not support both the synchronous and asynchronous character (for example *tf.booleanMaskAsync*), clashes with the synchronous nature of JavaScript.

All things considered, we can definitely suggest JavaScript and TensorFlow.js for developing object detection applications that use pre-trained models, but for training purposes, we recommend using another programming language, such as Python or C++, and then convert the model to JavaScript compatible format, or switching to a "lighter" model, such as the *Tiny YOLO* series [10].

YOLO vs SSD

Although the comparison experiment we conducted declared YOLOv3 as the most efficient of the two object detection models, this conclusion should not be generalized. The results refer to the needs of our project and definitely do not reflect all strong and weak points of the two models. The experiment images and videos were quite limited and tested the models on very specific configurations and not even on all COCO dataset classes.

YOLOv3

The YOLO model series is ever evolving, with each version overcoming its predecessors. YOLOv3 is a powerful, efficient and fast object detection model. Pre-trained on COCO dataset, YOLOv3 was able to detect even the smallest vessels in low resolution images, with the only limiting factors being the not ideal lighting conditions and the outline distortion that occurs when vessels are too close to each other. On the other hand, due to its computationally intensive character in combination with JavaScript, we could not achieve a higher frame rate than 0.8 FPS.

Performing transfer learning on YOLOv3 in JavaScript was an extremely complicated task. The fact that TensorFlow.js does not provide a pre-developed loss function that covers YOLOv3's needs, led us to develop a custom one. Unfortunately the training phase was never completed due to a memory leak in our code. So, although after a certain number of iterations the loss value keeps decreasing, we cannot draw definite conclusions about the success of our transfer learning pipeline. An alternative and certainly more efficient method for custom training YOLOv3, would be to use C++ and Darknet framework [23], the original machine learning library YOLOv3 was written with.

6.2 Future Work

This thesis was part of a project, which is still active, and covered some of the required final targets. Consequently, based on the concluding results, several improvements can be made to further enhance the next steps and, ultimately, achieve all the project's goals. Furthermore, machine learning and especially deep neural networks are fields that evolve rapidly, creating space for further optimization of the codes. The suggested future work is summarized in the next paragraphs.

Data Augmentation

The dataset we used to train our model consists of only 728 images, which is quite a small amount for such a complicated task. We estimate that more than approximately 3000 images should be considered a sufficient dataset. Collecting all these images is a time consuming procedure that could take several years. An alternative solution could be to perform data augmentation, a technique that multiples the available images by flipping, rotating, scaling, cropping, applying Gaussian blur, translating them etc. Data augmentation is widely used among the machine learning community in cases where the available data are limited, showing robust results.

Detection on Night Vision

The camera on Peristera island has the capability of switching to night vision after a certain time in a day. A complete monitoring system should always be able to detect decently under any lighting conditions. Unfortunately the night vision images containing

at least one vessel we had use for were quite few. Using them in the training process would simply "confuse" our model. A method that could be used in conjunction with image collection, is to artificially generate night vision-like images by applying filters on the daylight dataset.

Figure 6.1: Image extracted from the video stream of Peristera while the camera was on night vision mode.

Main Detection Code Pipeline Optimization

While testing standard YOLOv3 on the Peristera video stream, it managed to achieve a relatively low frame rate of 0.8 FPS. This is a limiting factor for real-time monitoring. We suggest that further optimization of the main pipeline code and even modifications on YOLOv3 itself should be made, in order to better fit the main goal of this thesis.

In addition, using CORS Anywhere to overcome the CORS policy limitation while deploying the application is not a sustainable solution. The obstacle could be tackled by hosting the application in the same server as the camera control website. This could also limit the deviation of the video stream's resolution.

Fine-tuning

A common step of transfer learning that follows an initial phase of model training is Fine-tuning. In our case it could not be implemented because training was never completed. Granted that the memory leak obstacle is tackled in the future, fine-tuning could produce a more robust model that shows increased accuracy.

Expanding the Application Features and Field

Once the final model is able to produce accurate results, the application could be used for collecting statistical data, that could contribute to control the vessel traffic in the Sporades area and the Aegean sea. In addition, provided that the frame rate is increased, it could also be used for estimating a vessels speed by comparing the coordinates of a box

between each frame. Such an implementation requires an algorithm that performs specific mathematical operations that eliminate lens distortion and calculate the distance from the camera.

The concept of this thesis could also be applied to the five underwater cameras that encircle the underwater museum. By the time writing, the implemented machine learning algorithms are able to perform only image classification. Using YOLOv3 trained to detect divers could additionally provide information about their position in the 3D space and even prevent accidents.

Appendices

Appendix A

IoU Code Pipeline

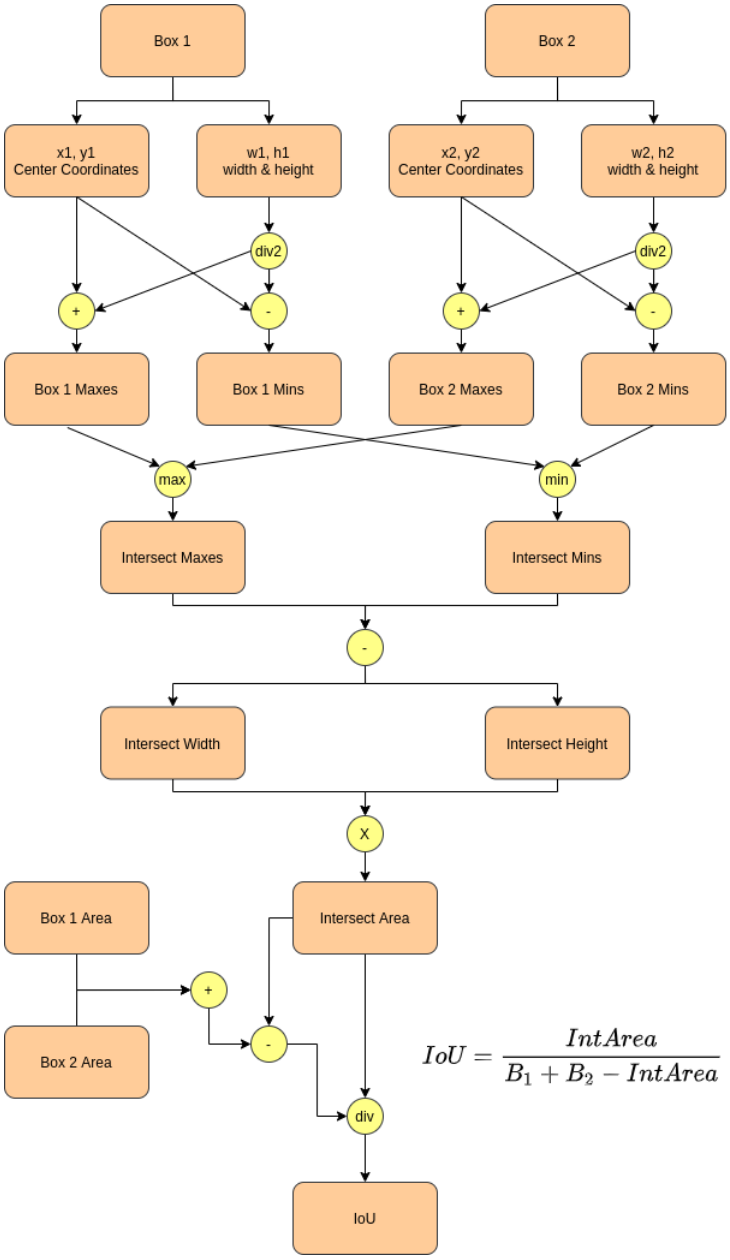


Figure A.1: Code pipeline that computes intersection over union of two boxes.

Bibliography

[1] Timo Dersch. Peristera Wreck. https://bestdivingingreece.com/persitera-wreck/.

[2] Peristera's ancient ship wreck. https://nous.com.gr/naxly_project/peristeras-ancient-ship-wreck/.

[3] Shanqing Cai, Stanley Bileschi, Eric D. Nielsen, and François Chollet. *Deep Learning with JavaScript: Neural Networks in Tensorflow.js*. Manning Publications Co., 2020.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, 2006.

[5] Devraj Agarwal. A Review of the Math Used in Training a Neural Network. https://levelup.gitconnected.com/a-review-of-the-math-used-in-training-a-neural-network-9b9d5838f272, 2020.

[6] Fei-Fei Li, Ranjay Krishna, and Danfei Xu. CS231n: Convolutional Neural Networks for Visual Recognition. http://cs231n.stanford.edu/, 2021.

[7] Stefanos Georgis. Position estimation of multiple sea vessels using a stereo-based camera system and neural networks. Master's thesis, NTUA, 2020.

[8] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep Learning*. 2015.

[9] Ayoosh Kathuria. What's new in YOLO v3? https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b, 2018.

[10] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[11] datahacker.rs. TF YOLO V3 Object Detection in TensorFlow 2.0. http://datahacker.rs/tensorflow2-0-yolov3/, 2019.

[12] PERISTERA SHIPWRECK. http://www.ikiondiving.gr/event/peristera-shipwreck/?lang=en.

[13] Henning Heiselberg and Andrzej Stateczny. Remote sensing in vessel detection and navigation. *Sensors*, 2020.

[14] Peter Russell, Stuart J.and Norvig. *Artificial Intelligence: A Modern Approach, Third Edition*. Pearson Education, Inc., 2010.

[15] Jake Frankenfield. Artificial Intelligence (AI). https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp, 2021.

[16] Sebastian Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/, 2016.

[17] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer Science+Business Media, LLC, 2010.

[18] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[19] Chandrakala Busireddy. is YOLO really better than SSD? `https://www.linkedin.com/pulse/yolo-really-better-than-ssd-chandrakala-busireddy`, 2019.

[20] Na Li. Object Detection (COCO-SSD) Demo. `https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd/demo`, 2021.

[21] COCO Dataset. `https://cocodataset.org`.

[22] Sha Qian. tfjs-yolo. `https://github.com/shaqian/tfjs-yolo`, 2019.

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

[24] Valentyn Sichkar. Introduction into YOLO v3. `https://www.youtube.com/watch?v=vRqSO6RsptU&t=312s`, 2020.

[25] TensorFlow.js API. `https://js.tensorflow.org/api/latest/`.

[26] YOLOv3 JSON file. `https://raw.githubusercontent.com/shaqian/tfjs-yolo-demo/master/dist/model/v3/model.json`.

[27] Cross-Origin Resource Sharing (CORS). `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

[28] Rob Wu. CORS Anywhere. `https://github.com/Rob--W/cors-anywhere#demo-server`.

[29] Introducing asynchronous JavaScript. `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing`.

[30] FFmpeg. `https://www.ffmpeg.org/`.

[31] Tzutalin. LabelImg. `https://github.com/tzutalin/labelImg`, 2015.

[32] Miguel Mota. K-Means Clustering in JavaScript. `https://miguelmota.com/blog/k-means-clustering-in-javascript/`, 2015.

[33] AlexeyAB. Yolo v4, v3 and v2 for Windows and Linux. `https://github.com/AlexeyAB/darknet#how-to-improve-object-detection`, 2020.

[34] function*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*`.

[35] qqwweee. keras-yolo3. `https://github.com/qqwweee/keras-yolo3`, 2018.

[36] Wikipedia. Memory leak — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Memory%20leak&oldid=1032107406`, 2021.

[37] David Flanagan. *JavaScript The Definitive Guide*. O'Reilly, 2011.

[38] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[39] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[40] Daniel Smilkov and Nikhil Thorat. Tensorflow.js: Machine learning for the web and beyond. *arXiv*, 2019.

[41] N. Wang, Y. Wang, and M.J. Er. Review on deep learning techniques for marine object recognition: Architectures and algorithms. Control Engineering Practice. `https://doi.org/10.1016/j.conengprac.2020.104458`, 2020.

[42] Hugo Zanini. Custom object detection in the browser using TensorFlow.js. `https://blog.tensorflow.org/2021/01/custom-object-detection-in-browser.html`.

[43] TensorFlow Core v2.5.0. `https://www.tensorflow.org/api_docs/python/tf/all_symbols`.

[44] Eddie Forson. Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning. `https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab`, 2017.

[45] Yolo v3 loss function. `https://stats.stackexchange.com/questions/373266/yolo-v3-loss-function`.

[46] Shanqing Cai. TensorFlow.js: Custom Loss. `https://codepen.io/caisq/pen/QZYZEZ?editors=1111g`.

[47] Vincent Mühler. 18 Tips for Training your own Tensorflow.js Models in the Browser. `https://itnext.io/18-tips-for-training-your-own-tensorflow-js-models-in-the-browser-3e40141c9091`, 2018.