UNIVERSITAT ROVIRA I VIRGILI (URV) Y UNIVERSITAT OBERTA DE CATALUNYA (UOC)

Computational Engineering and Mathematics

Master's Thesis

Area: Artificial Intelligence

Underwater Species Detection in Images and Videos

Author: Ricard Fos Serdà

Consultant teacher: Antonio Burguera Burguera

Submission date: 01/06/2022

TECHNICAL SPECIFICATIONS

TITLE:	Underwater species detection in images and videos
Author:	Ricard Fos Serdà
Consultant teacher:	Antonio Burguera Burguera
Due date (mm/aaaa)**:**	06/2022
Master's Degree Program:	Computational Engineering and Mathematics
Final Project Field of Study:	Artificial Inteligence
Language:	English
Key Words:	Object Detection, Deep Learning, Computer Vision

Abstract

English:

The topic of this study belongs to the area of Deep Learning, which means using neural networks to create predictive models. Concretely, this thesis is about object detection models using Deep Learning to be able to detect and classify underwater species on images and on video in real-time.

The study of Deep Learning methods is more relevant each year, as it is being used on more kinds of problems as the research expands. Object detection is a very useful tool and is becoming the backbone for autonomous robots. Detecting underwater species is an important task given the difficulty of humans to stay in that habitat. Underwater cameras can be used to find given species and can be used to count them and study their behavior.

A study of the origins and evolution of object detection models has been conducted and two detection models and one instance segmentation model have been retrained for the detection of underwater species on a public dataset. The resulting models have been evaluated in terms of detection speed and precision using the standard metric for object detection called Mean Average Precision. Resulting videos have been generated with drawn detections and segmentation masks.

Additionally, a simple tracking algorithm has been developed that is able to assign unique ids to each fish on camera and track them until they get out of sight.

By the end of the thesis the best detection model for underwater videos has been found to be YOLOv5, which achieves good accuracy and framerate.

Spanish:

El tema de este estudio pertenece al área de Deep Learning, que significa usar redes neuronales para crear modelos predictivos. Concretamente, esta tesis trata sobre modelos de detección de objetos usando Deep Learning para poder detectar y clasificar especies submarinas en imágenes y en video a tiempo real.

El estudio de métodos de Deep Learning es cada vez más relevante dado que se está usando cada vez en más tipos de problemas a medida que se expande su investigación. La detección de objetos es una herramienta muy útil y está convirtiéndose en la base para crear robots autónomos. Detectar especies submarinas es una tarea importante dada la dificultad del ser humano de permanecer en ese hábitat. Se pueden usar cámaras submarinas para encontrar las especies deseadas y se puede usar para contarlas o estudiar su comportamiento.

Se ha llevado a cabo un estudio de los orígenes y evolución de modelos de detección de objetos y dos modelos de detección y uno de segmentación se han reentrenado para la detección de especies submarinas en un dataset público. Los modelos resultantes se han evaluado en cuanto a velocidad de detección y precisión usando la métrica estándar para detección llamada Mean Average Precision. Se han generado los videos resultantes con las detecciones y segmentaciones dibujadas.

Además, se ha desarrollado un algoritmo de seguimiento simple que es capaz de asignar identificadores únicos a cada pez enfocado por la cámara y seguirlo hasta que ya no es visible.

Al final de la tesis se ha encontrado que YOLOv5 es el mejor modelo de detección para videos submarinos ya que consigue buena tasa de imágenes por segundo y precisión.

“The human world, it's a mess
Life under the sea
Is better than anything they got up there”

-- Horatio Thelonious Ignacious Crustaceous Sebastian

This work is dedicated to my sister Helena, who loves the sea.

Table of Contents

Table of Figures

1. Introduction

1.1. Context and Relevance of the Research Topic

Deep learning [1] is changing the landscape of **Artificial Intelligence** (**AI).** The increase in computational power and the use of **Graphics Processing Units (GPU)** have made this research topic more competitive. State-of-the-art solutions tend to become obsolete in a year or two. As the world is becoming increasingly aware of the advantages of AI, it is sadly not used only for good. Deep Learning can be used for a variety of purposes, such as:

- **Image Classification**: Given a photograph tell, for example, if it is a photo of a dog or a cat.
- **Natural Language Processing (NLP)**: Resolve different kind of language problems given a word or sentence, such as translation, sentiment analysis or understanding a sentence.
- **Price Forecasting**: Predict the stock price of the next day given the stock price of the last few days.
- **Motion Planning**: Make a robot or a simulated character learn where and how to move by its own, such as autonomous cars learning how to drive inside their lane.
- **Object Detection**: Given an image, be able to detect the presence of desired objects and their position in the image. For example, object detection can be used on a camera to detect faces.
- **Marketing**: By tracking online activities from users a model can learn their habits, personality and preferences. That information can be used to select the most appropriate adds to show to each user. Companies can access that information for morally dubious purposes as well. The use of that information could be seen as privacy attacks.
- **More**…

Object Detection is one of the most researched topics in Deep Learning and is the focus of this project. It is also a great example of the speed at which research is going, as new solutions improve previous ones in a matter of months. Object detection focuses on finding objects on an image by finding a rectangular "**bounding box**" that encapsulates each of them. Each bounding box is paired with the class to which the object pertains, such as it being a dog or a cat.

Detection can be used by autonomous cars to detect and avoid obstacles, satellites to find anomalies, video cameras to detect faces and much more. One of the fields where it is heavily used is in the military, using it on drones and missiles to detect their target. Such uses of this technology do not interest me, and I wanted to find more morally fulfilling fields to apply AI.

I chose to focus on **detection of underwater species.** I got inspiration from an online competition [2] with the objective of detecting invasive species of starfish, the Crown of Thorns Starfish, which threatens the coral in the great barrier reef in Australia. I thought that being able to detect **invasive underwater species in real time videos** could be useful. For example, a model could be used on robots to identify the areas where those species are located.

Moving away from invasive species exclusively, a model could be trained to identify and count salmons in a salmon fish farm. Additionally, it could be used to detect damaged or unhealthy salmons that should be removed from the farm. However, the idea of finding invasive species is more appealing to me for moral reasons. Either way, using object detection on underwater species is interesting and not as common as, for example, pedestrian and vehicle detection.

The main purpose of this work is to carry out a **study of object detection models** since their inception and find the most suited for real-time detection. Finally, chosen detector models are going to **trained and evaluated on a public underwater fish detection dataset**.

1.2. Objectives

The specific objectives of this thesis are:

1. Design a real-time object detection model for underwater species

The main goal of the project is to train an accurate object detection model that can be used on real-time video. It should detect underwater species accurately at each frame while keeping an acceptable video speed in terms of frames per second (FPS). This objective includes the following subobjectives:

1.1. Evaluate the data and make a good training and validation Split

A model needs data for training, so the first objective is to find a public dataset containing videos of underwater species prepared for object detection problems. If no such dataset could be found, a custom dataset should be created using public videos and annotating it manually. Deep Learning models are trained on training data and evaluated on different data, called validation or test data. For that reason, the dataset should be split into train and validation sets.

1.2. Preprocess the data before training the model

The data should be fed to the model in the correct format. The original format could be ill fitted for a specific model, have errors or simply be hard to understand. That is why the data should be preprocessed to be given the correct format.

1.3. Find the Object Detection model that better suits the problem

The technology of object detection should be carefully studied to be able to understand how and why it works and which models are more fitted for the problem and how to use them. The chosen model must be trained on the dataset with the proper configuration.

1.4. Evaluate the model

The model precision on validation data must be evaluated using the correct metrics for object detection.

1.5. Make sure that the model is fast enough to detect objects in real time during a video

The model should be able to make predictions to an acceptable number of images per second so that it can be used on real time videos. A minimum of **25 fps** should be achieved.

2. (Optional) Design a Segmentation model

While object detection is about finding rectangular bounding boxes around the detect object, segmentation [3] is about finding each pixel that pertains to that object. An optional objective to extend object detection to segmentation is proposed. The deep learning segmentation model will be tested for real time video as well.

3. (Optional) Extract extra information from detections

Detection information is usually used for a purpose, such as counting the number of objects, their size, movement speed or others. The focus of this optional objective is to apply **object tracking** to underwater species on real-time video. Tracking objects can be used to count how many fishes of certain species appear during a time frame.

4. (Optional) Create my own detection dataset from public underwater videos

The Optional objective of creating my own dataset by manually inputting detection data on each frame from a public video of underwater species.

1.3. Followed Approach

Before this thesis, my only experience with Deep Learning was almost nonexistent. I knew how traditional Neural Networks worked and completed a basic tutorial about image classification. I did not know anything about Object Detection models and knew very little about image classification. The first step was to study deep learning in the topic of image classification before studying object detection.

In the studying process, I learned about **transfer learning**. This method consists in using pre-trained models on popular datasets with lots of images and classes and repurpose it for another dataset. As I learned about Object Detection, it became clear that the best and most usual approach was **using state-of-the-art models instead of building one from scratch**.

Although the idea of designing a model by hand was interesting, I found it unnecessary. By the time of writing this thesis, object detection models have become so complex that trying to implement and train a model from scratch would be a very complicated and time-consuming task.

After studying the history and evolution of object detection models, I select some of the most prominent ones to train, evaluate and compare on a public detection dataset of underwater species. The precision and speed on real time videos are taken in consideration.

For the implementation phase, I used the **Python** programming language [4] and studied the most common deep learning libraries for python before sticking with **Pytorch** [5]. After training the models I generated videos of underwater species with drawn detection boxes and fps values.

1.4. Work Planning

The terms **Prueba de Evaluación Continuada (PEC)** refers to continuous assessment activities and serve as milestone in the project. Figure 1 shows the Gantt chart of this thesis planning based on PEC deadlines. PEC0 and PEC1 do not appear as they were the preplanning.

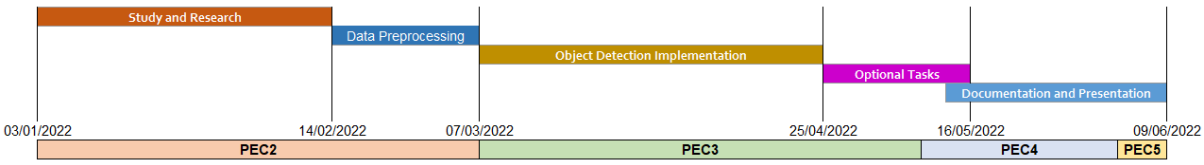

Figure 1: Gantt chart with the work planning.

The timeframe for **PEC2** includes the **study and research** of the topic of Deep Learning, image classification and object detection as well as the **gathering and preprocessing of a public dataset** that fits the objective of this project. Online courses, videos, websites and research papers are to be used for the study and research phase. For data gathering, a public dataset consisting of videos underwater species annotated for detection purposes should be found. Data preprocessing should include studying the dataset, fixing its errors and converting it into the appropriate format for model training.

By the end of **PEC3**, the **object detection model is planned to be trained and evaluated**, thus completing the main objective of this thesis. Deep learning libraries for object detection should be studied and implement chosen models with the preprocessed data. Multiple models should be trained and evaluated to choose the best ones afterwards. The final model should output a video with drawn detections and fps values.

Additionally, **optional objectives** should be almost finished by the end of PEC3 and finished by 16/05/2022. The priority order of optional tasks is the same at which they are mentioned in section 1.2. If an optional task is not finished after that deadline its implementation will be cancelled.

PEC4 consists of the **full documentation of the project**, which should be finished by 1/06/2022. The study of object detection models will be included as well as an explanation for the implemented python code.

PEC5 is the final delivery of the **presentation slides with my explanation in video form,** which should consist of 20 slides and last 20 minutes at most. Finally, the **public defense** of the project will be done before 22/06/2022.

Resources used in this project include:

- Computer equipped with a GPU
- Python
- Deep Learning Library (Pytorch)
- Additional python libraries

1.5. Summary of the Products Obtained

As a result of completing this thesis, multiple object detection models have been trained for real time detection of underwater species, including the resulting videos with drawn bounding boxes and fps values. A model for instance segmentation has also been trained including a video with drawn segmentation masks. Finally, an object tracking algorithm has been developed, which uses detections from the object detection model. A video with the tracking results has been generated.

1.6. Brief Description of the Other Chapters

The overall structure of this document is as follows:

Chapter 2 starts introducing **convolutional neural networks** for image classification, which are the backbone for object detection models. It also includes the study of **the history and evolution of object detection models**, from slow to faster models for real-time detection.

Chapter 3 starts with the **exploration of public datasets** to be used for this project, and shows the final dataset chosen and how it was preprocessed. It also includes the documentation of **my implementation of object detection models**: Faster R-CNN and YOLOv5, on that dataset. It also includes the optional tasks of segmentation and object tracking.

Chapter 4 shows the conclusions of the thesis, with tables comparing speed and precision of each model and concluding with which is the best approach for real-time detection.

2. State of The Art

This section introduces the most important topics related to this thesis, including convolutional neural networks, object detection models and evaluation methods for object detection.

2.1. Study of Convolutional Neural Networks.

Image classification is a common topic when learning about deep learning and computer vision. It consists of using one image as input to a neural network and receiving a prediction of its class. Is the image a dog or a cat? A type of neural network called **Convolutional Neural Network** (**CNN**) [6], is usually used to solve those kinds of problems as represented in Figure 2.

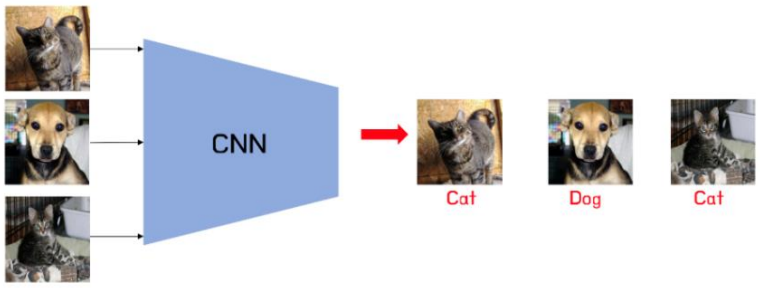

Figure 2: CNNs can be trained to predict classes from given images. Source: Google Images.

A **CNN** is a neural network that includes **convolutional layers**, which consist of a list of a set number of kernels with a given shape (n x n) that slide through the image with a step of a given number of pixels called "**stride**", computing for each step the dot product of the n x n values of the kernel with the corresponding pixel values of the image in that step. Each kernel is passed through all the images of the input, which can be three for the first layer if the input is an RGB image, having three color channels as shown in Figure 3, or could be the number of kernels in the previous convolutional layer. The partial results of the kernel used on each of the input channels are added alongside an additional value called **bias**, which is unique to each kernel.

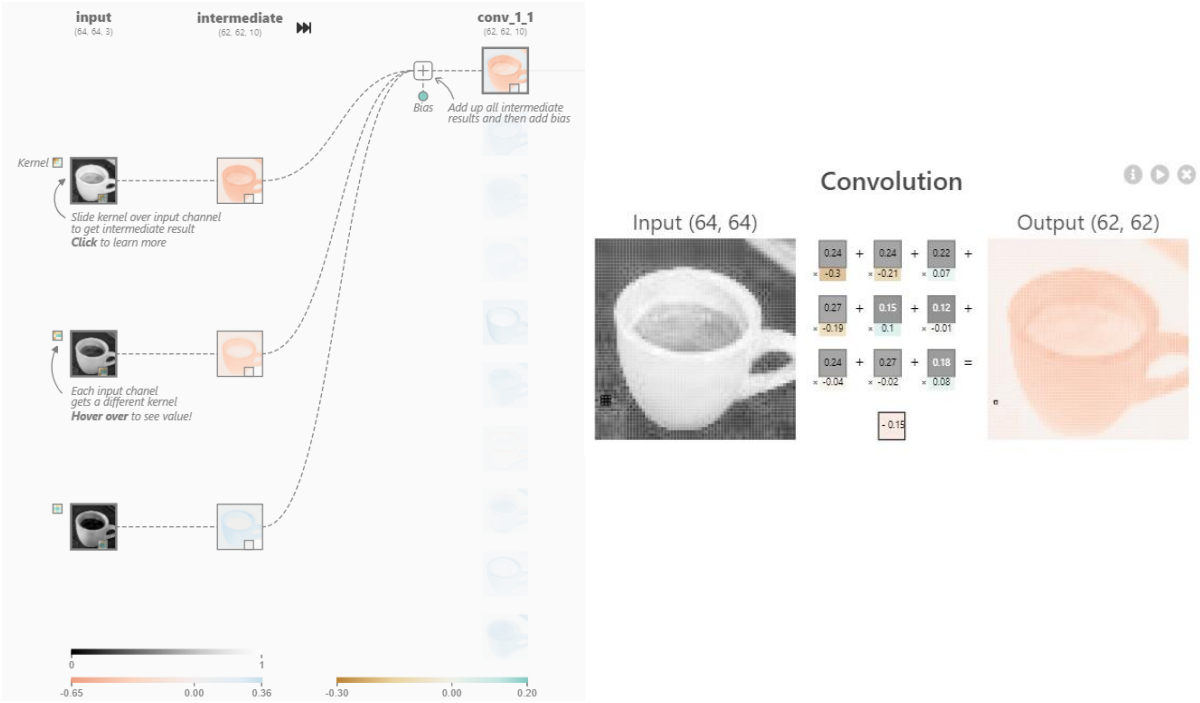

Figure 3: The input layer and first convolutional layer from a CNN (left) and a close view of one of its kernels (right). Source: [6].

The values of each of the n x n values and the bias in each kernel are the **trainable parameters, also called weights,** of the neural network. Weights, which are initially initialized randomly, are "trained" by optimizing (or minimizing) the final **loss function** of the network, which computes the "distance" between the predicted output and the truth (ground truth). There are different loss functions aimed at different kind of problems. If the network must predict a class, a **Categorical Crossentropy** loss is commonly used, which is defined as follows:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Where y equals 1 if the true label of that example is the class *i* and 0 if not. ŷi stands for the output probability of the example being the class *i* according to the network. This formula increases if the true class has a low prediction probability, as it results in the negated log of that probability, which increases as the probability is closer to 0.

There are different **optimization algorithms**, such as **Stochastic Gradient Descent (SGD)** or **Adam** [7], used to minimize the loss function (see Figure 4) using partial derivatives in respect to each weight by applying the chain rule from the loss function back to that weight, by multiplying the partial derivatives of all the functions in the way. That is why the process of optimizing the network is called "**backpropagation".** After computing the partial derivatives, each weight is updated by multiplying a set parameter known as **Learning Rate,** usually lower than 1, to the result of its partial derivatives (depends on the optimization algorithm) and subtracting that number from the original weight.

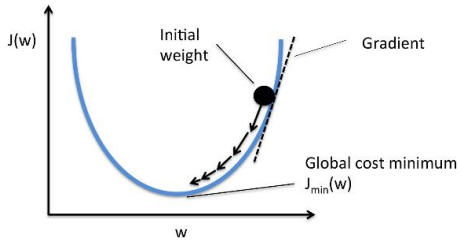

Figure 4: Example of a simple case of gradient descent with one weight. The partial derivative (Gradient) of the final function J(w) with respect to 'w' is positive if J(w) increases as w does, or negative if it decreases when 'w' increases. To minimize J(w), the derivative multiplied by a learning rate is subtracted from 'w', ensuring that the next value of J(w) will be closer to the minimum, as a positive derivative indicates that 'w' should decrease and a negative one indicates it should increase. Source: Google Images.

The final output of each kernel is the sum of the results for all the inputs plus a bias, which then is passed through a non-linear activation function, such as **ReLU** = max (input, 0). The output shape can be different from the input depending on the kernel size and stride. For example, a 3x3 kernel going through a 14x14 image outputs a 12x12 image, as the right side of the kernel reaches the end of the image in 12 steps, and the bottom of the kernel reaches the bottom of the image in 12 steps as well. Reshaping can be avoided with the use of **padding**, like adding 0 to each border of the image, thus becoming a 16x16 image as shown in Figure 5. Output sizes can also be higher than the original depending on the stride and padding.

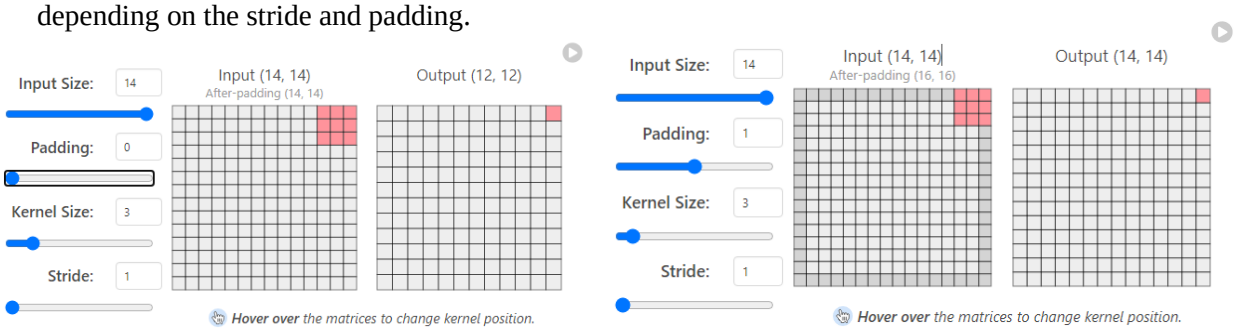

Figure 5: 3x3 kernels can generate an output of the same size as the original image if a padding of 1 pixel is used.Source: [6].

CNNs include other layers as well, typically the **Pooling layers**, whose main function is to reduce the output size of the previous layer. For example, **MaxPooling** layers use a n x n kernel that slides through the input image, like a convolutional kernel, and the output is simply the maximum value of the pixel in that region for each step, as shown in Figure 6.

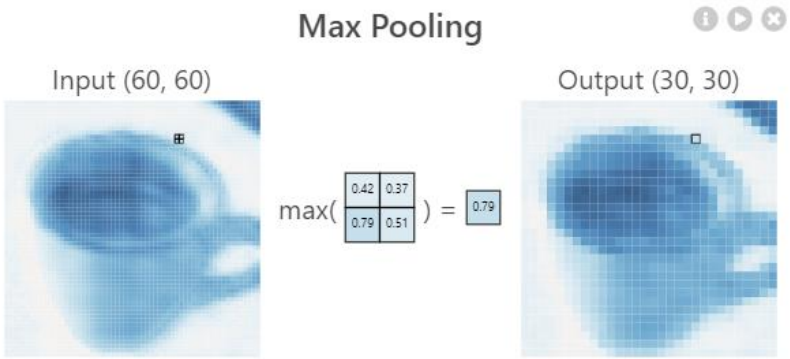

Figure 6: Max Pooling in detail. Source: [6].

The final output of a layer in a CNN is a concatenation of all the kernel outputs and is called **feature map,** which is a 3-dimensional matrix. After all the convolutional and pooling layers, the last feature map is flattened and becomes a 1D array including all the values of the feature map. For example, if the last layer is a Convolutional layer of 10 kernels, each generating a 5x5 output, the output would be a **5x5x10 feature map** and the flattened output would be an array of 5x5x10 = **250 values**. CNNs can also have fully connected layers at the top (end) after the flatten layer. The last layer is usually a fully connected layer with **the number of neurons equal to the number of classes to predict**, giving the final prediction as output, which can be used as input to a **Softmax** activation function that returns the probabilities of the image pertaining to each one of the classes. The Softmax activation function is defined as follows:

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Where x_i is the output of one of the neurons of the output layer, which pertains to one class prediction. The function divides the exponential of that value by the sum of the exponentials of all values for that prediction, converting raw values, called logits, into probability values.

Figure 7 shows a representation of a full traditional CNN.

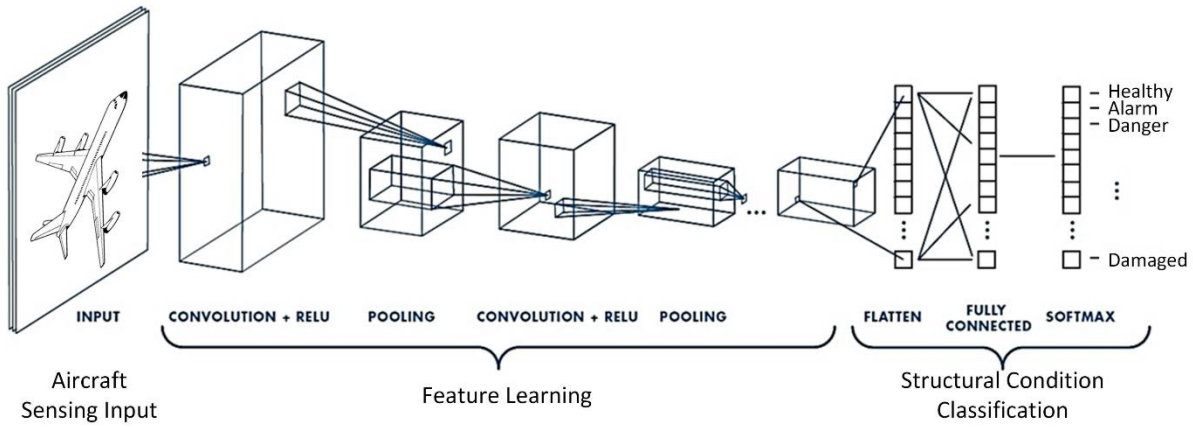

Figure 7: Architecture of a simple CNN. Source: Google Images.

2.2. Study of Two-Stage Object Detection

CNNs solve the problem of image classification, but what happens if there are different objects in an image that we want to identify? If both a cat and a dog appear in an image, the CNN will only output one predicted class. This problem can be solved with **Object Detection**, which consists of finding the area of the image where the object is, and its class, as shown in Figure 8. The output of object detections models is a list of each class detected with its bounding box, which can be in the form of the coordinate x and y of the center of the box and its width and height.

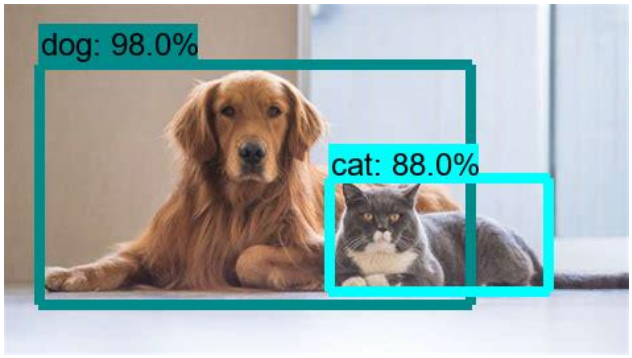

Figure 8: An example of object detection. Source: Google Images.

Early models of object detection such as **Viola-Jones** [8] did not use neural networks at all. They used fixed size **sliding windows** of kernel filters that moved through the whole image, using the extracted features on a linear classifier, which would be equivalent to one neural network with one fully connected layer of one neuron.

OverFeat [9] improved those models with the introduction of CNNs. having a sliding window crop the image and send the cropped region to a CNN at each iteration to classify that part of the image. If a class was found, the model considered there was an object in that place. This method has the problem of being too slow, as lots of images were passed to the CNN.

2.2.1. Region-Based CNN (R-CNN)

Region-Based CNN (R-CNN) [10][9] solved this problem by reducing the number of cropped images to feed from the initial image to the CNN by using **Selective Search [11]** before the CNN. Selective search is an algorithm that tries to segment all the objects of an image at different scales and containing components of different color and texture, like the images in Figure 9, more precisely than traditional image segmentation algorithms.

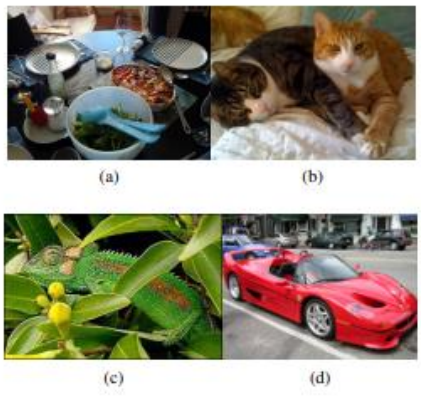

Figure 9: With simple image segmentation the image is segmented by color and texture. The car could have its wheels in a different group than its chassis, or it would be difficult to detect the whole span of the table containing dishes. Source: [11].

After an initial segmentation and saving the bounding boxes spanning each group, selective search uses a greedy algorithm to combine each group to its closest neighbor, computing that "distance" in terms of color, texture, size, and shape. The bounding boxes of the new combined groups are saved as well. Selective Search keeps doing iterations of combining groups and saving their bounding boxes until only one group remains. Figure 10 shows the results from Selective Search at different iterations.

Figure 10: The left most image shows the original segmentation, with its corresponding bounding boxes. Each iteration gets fewer groups and bounding boxes as they combine with each other, being able to have the whole woman from the right in one group and bounding box. Source: [11].

R-CNN takes the bounding boxes from the selective search as input, which would be around 2000, using each image inside them on a CNN, which is called "**backbone**", to classify them and predict which object contains, thus reducing the number of images to feed to the CNN compared to the sliding window method. The backbone tends to be a popular pretrained image classification model, such as **ResNet** [12] or **EfficientNet** [13], which have good accuracy scores on big image datasets like **ImageNet** [14]. After the classification of each region is done, the four values describing the bounding box position are used as input for a regressor that adjusts their values using the ground truth to improve the precision of the detection. Figure 11 shows a representation of the R-CNN architecture.

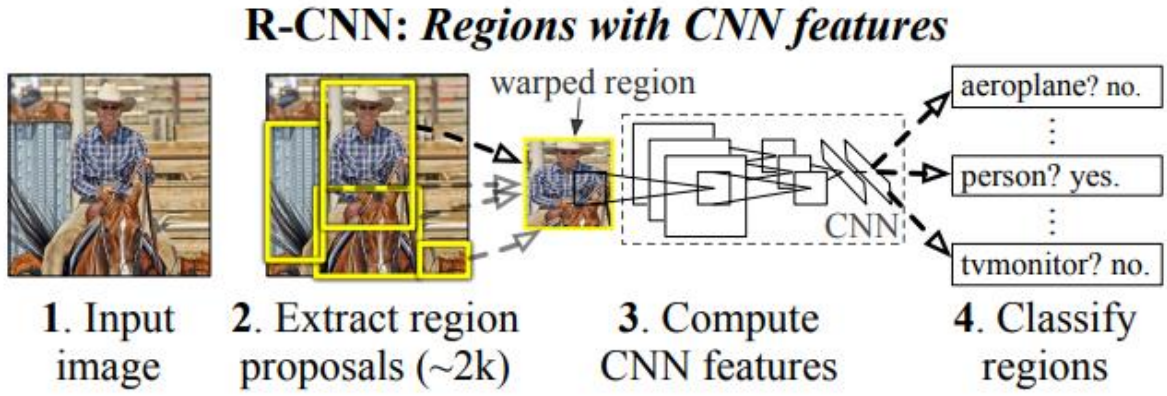

Figure 11: Architecture of a R-CNN. An extra bounding box regression layer is added next to the classifier. Source: [10].

During training, the **Intersection over union (IoU)** (see Figure 12 and section 2.2.1) between predicted boxes and ground truth boxes is computed. If the prediction and the ground truth are the same, the IoU will be equal to 1. Predicted bounding boxes with IoU > 0.5 **are considered true for their detected class** and the rest are considered negatives, by assigning them the class "background". Bounding boxes that have an IoU higher than 0.3 with respect to another bounding box with higher IoU with the same ground truth box are considered overlaps and classified as negative.

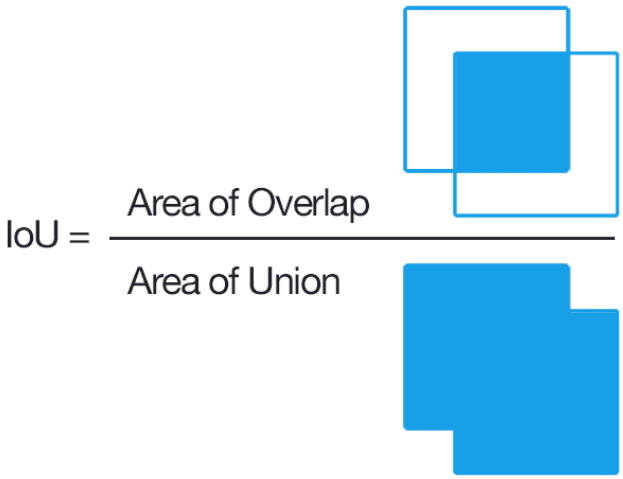

Figure 12: Visual representation of the IoU formula. Source: Google Images.

During inference, a similar method is commonly used in Object Detection models in general to only output the most relevant bounding boxes, called **Non-Max Suppression (NMS)**. This algorithm starts removing bounding boxes with a lower **classification score** than a specified threshold, then, for each class, it sorts the remaining bounding boxes using their probabilities and pick the highest scoring one, which is removed from the list and marked as the "current element".

The **IoU** of that element with each of the remaining boxes from the list is computed and if it is above a specified threshold, the box from the list is deleted. Finally, the "current element" is added to the "final list". The process of picking the highest scoring bounding box and deleting the rest depending on their IoU is repeated until there are no boxes left on the initial list, and the "final list" is returned as output. Figure 13 shows the result of using NMS on multiple boxes.

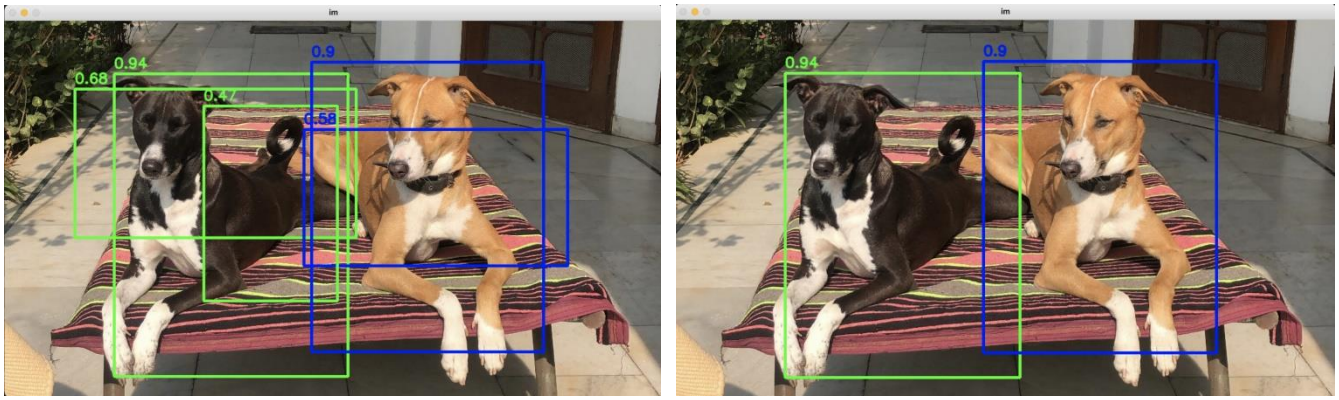

Figure 13: Result of using Non-Max Suppression. Source: Google Images.

2.2.2. Fast R-CNN

R-CNN has the drawback of still being very slow, as feeding the CNN with 2000 regions per image is still very computationally expensive and depends on the output of the selective search algorithm, which is not trainable and can generate bad region proposals from the start.

Fast R-CNN [15] tries to solve some of those problems by using the whole image as input for a CNN backbone that **produces a convolutional feature map** which, like the input, is an "image" encoded as a matrix with three dimensions: height, with and color channels. Then, for each region from the **selective search**, instead of cropping the original image, the region of interest (RoI) is cropped from the convolutional feature map and goes through a **RoI Pooling Layer,** which **turns the RoI into a fixed-length (h x w) feature vector** by dividing the RoI into (h x w) subsections and picking the maximum value out of each. Each feature vector is fed to a set of fully connected layers that branch into two output layers: one that classifies the object and the other that adjusts the 4 values that encode the bounding box. Figure 14 shows the Fast R-CNN architecture.

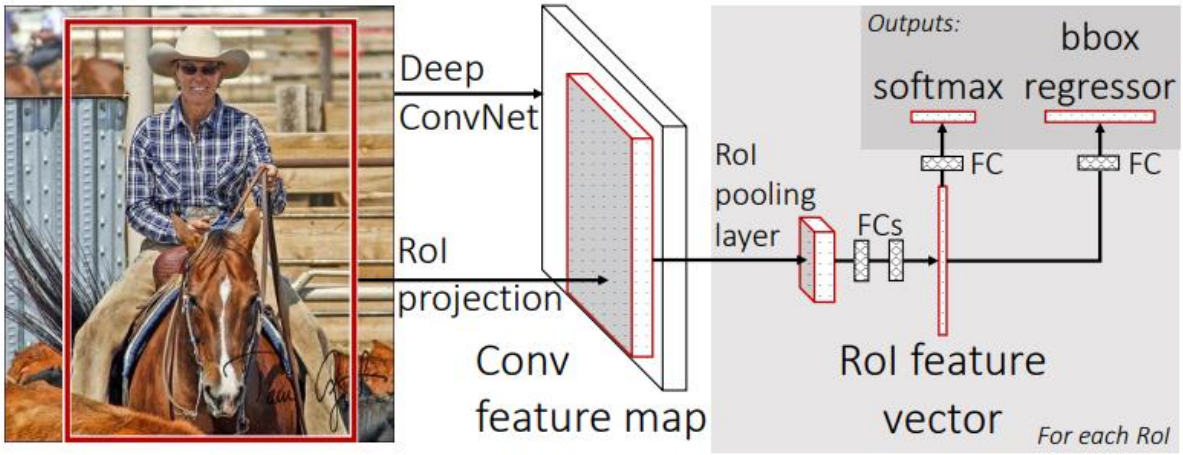

Figure 14: Fast R-CNN architecture. Source: [15].

Fast R-CNN does not feed 2000 images to the CNN, as the convolution operation is only done once per image, making it much faster than the original R-CNN at test time. However, it still depends on region proposals from the selective search algorithm, which are computationally expensive and affects its performance.

2.2.3. Faster R-CNN

Faster R-CNN [16] avoids using region proposal methods such as Selective Search by, after generating an initial convolutional feature map like Fast-RCNN, using **Region Proposal Networks (RPN)**, a convolutional network that takes the convolutional feature map (of any size) as input and outputs a set of region proposals. Each region proposal has an **objectness score** measuring how likely that region pertains to a class and not to the background, which is computed by checking the IoU of the region with the ground truth boxes. Those region proposals are used as the input of a **Fast-RCNN** network that outputs the final detections. Figure 15 shows a representation of the Faster R-CNN architecture.

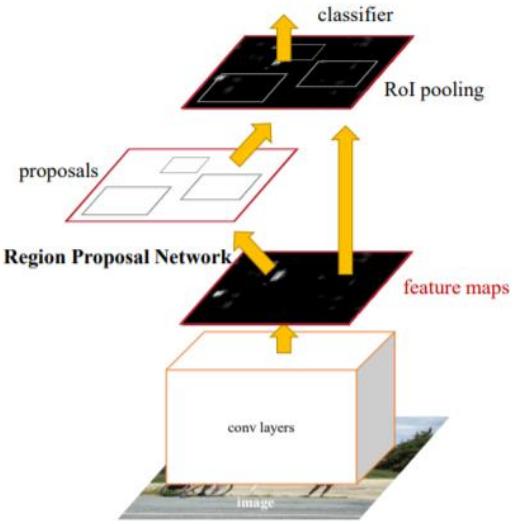

Figure 15: Representation of a Faster R-CNN. Feature maps are used both in the RPN and the Fast R-CNN. Source: [16].

The **Region Proposal Network (RPN)** uses the output of the backbone convolutional neural network, which takes the input image after resizing it such that its shortest side is 600 pixels and the longer side not exceeding 1000 pixels. The backbone total stride is 16, meaning that for each consecutive pixel in the backbone output, the corresponding pixels in the original image are 16 pixels apart.

RPN uses by default **3 fixed scales and 3 aspect ratios to create 9 "anchors"** centered in a 3x3 sliding window that slides through the feature map. These anchors indicate possible objects of various sizes and aspect ratios in the corresponding location at the original image in respect to that point in the feature map, as shown in Figure 16. The network then must check if each anchor corresponds to a real object and refine its coordinates to create region proposals that will be fed to the **Fast R-CNN**.

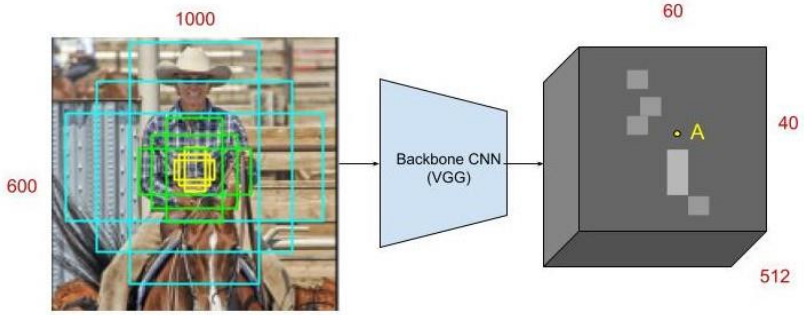

Figure 16: At point A in the feature map, 9 anchors using 3 scales and aspect ratios are generated centered at the corresponding point in the input image. Source: [17].

The 3x3 sliding window has the form of a 3x3 convolution with 512 units, which is applied to the HxWx512 feature map, meaning each of the 512 units uses 512 kernels with size 3x3. The output is fed to two sibling 1x1 convolutional layers that carry out the "objectness" classification and the bounding box regression, respectively.

The classification layer has 18 units as it outputs two values for each of the 9 anchors: The probability of that anchor containing an object and the probability of containing the background, obtained by using the **softmax** activation. Thus, the final output is a HxWx18 feature map, that indicates the probability of there being an object inside of each of the 9 anchors in each point of the convolutional feature map.

The regression layer has 36 units, encoding the coordinates of the 9 anchors of each point of the feature map with 4 values. The regression is used to improve the coordinates of the initialized anchors that have fixed scales and aspect ratios and is only activated if the ground truth of the box containing an object is true. Figure 17 shows the detailed architecture of the RPN.

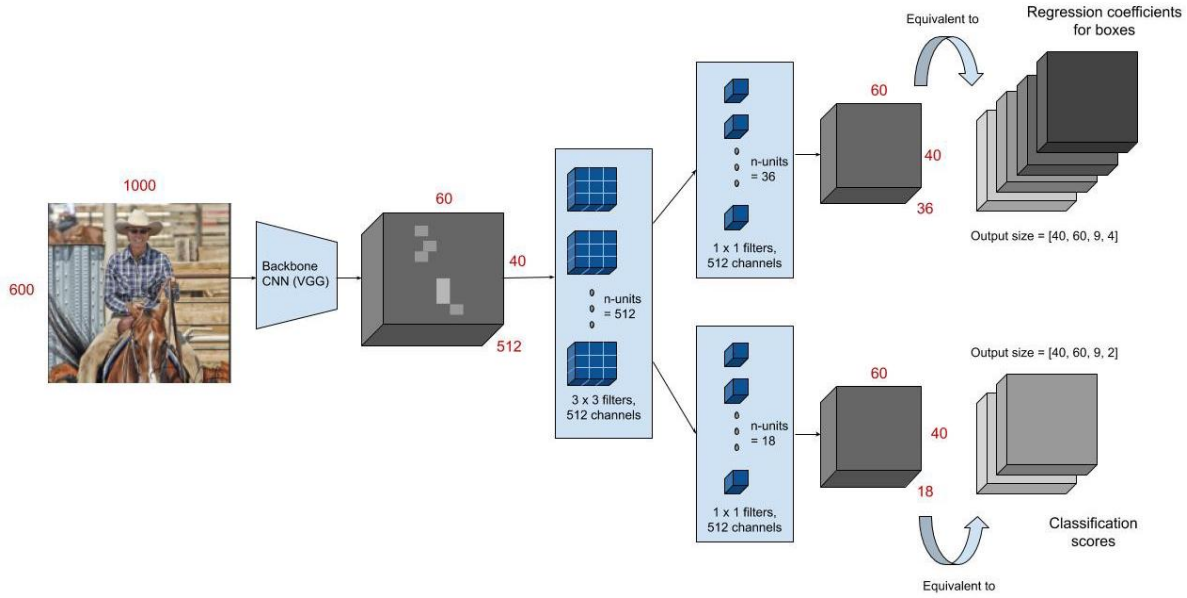

Figure 17: Architecture of a Region Proposal Network (RPN). Source: [17].

The total output consists of about 40x60 locations with 9 anchors in each, meaning around **20.000 anchors per image**. **During training**, however, anchors that cross the image edges are not used, leaving around 6000 anchors per image. From those anchors, RPN uses only the ones with the highest IoU with a ground truth box, or the ones with more than 0.7 IoU with any ground truth box. During test time, Non-Max Suppression is applied to the 20.000 bounding boxes, removing boxes with more than 0.7 IoU with a higher scoring one, giving around 2000 region proposals by the end.

Finally, those region proposals are fed to a Fast-RCNN, extracting them from the initial feature map, applying RoI pooling, being fed to a set of fully connected layers and then to the final bounding box regressor and classifier. Unlike the RPN regressor that has 4 different regressions (one for each coordinate) for each scale and aspect ratio combination, the bounding box regressor has 4 different regressions for each of the final possible classes.

RPN only takes around 10 milliseconds to compute region proposals, a considerable speed up compared to Selective Search, which takes more than one second. Faster R-CNN test time for a single image is below half a second, being able to run detections on real time videos at around **17 frames per second** depending on the GPU.

2.2.4. Mask R-CNN

Mask R-CNN [18] builds up on Faster R-CNN, to solve a different kind of problem, called **Instance Segmentation,** while still detecting objects using bounding boxes**.**

Segmenting an image, as seen in the Selective Search explanation, means dividing the image in groups, usually depending on color, shape, and texture. **Semantic Segmentation** tries to group each object of the same class in the same group, painting the objects pixel by pixel, and coloring the objects of the same class with the same color. Those pixel-by-pixel groups are called segmentation masks. **Instance Segmentation** decouples those groups of the same class to have a **separate segmentation mask for each singular object**, having a different color each, as shown in Figure 18.

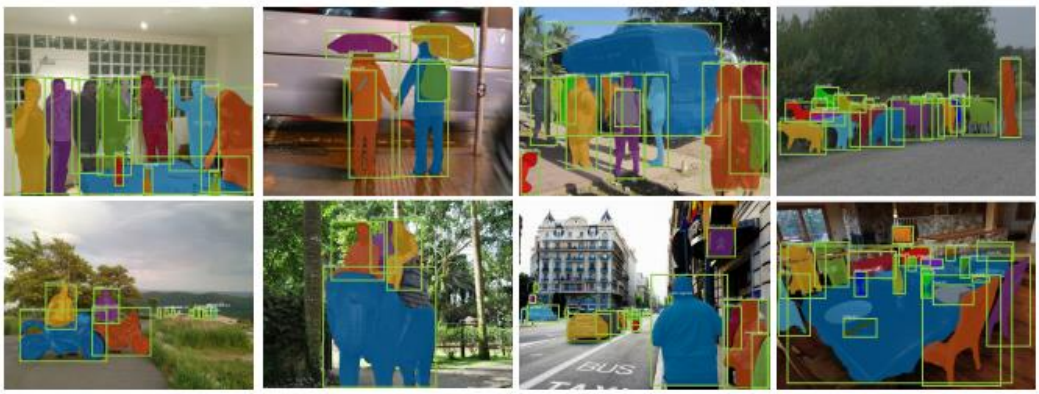

Figure 18: Examples of instance segmentation with bounding boxes. Source: [18].

To predict segmentation masks, Mask R-CNN adds a **parallel fully convolutional network** to **Faster R-CNN,** using the same extracted RoIs that are used as input for the image classification and bounding box regression. In Faster R-CNN, those extracted RoIs go through a RoI Pooling layer that converts each extracted RoI into a feature vector of the same size, as described section 2.2.3. Figure 19 shows the Mask R-CNN architecture.

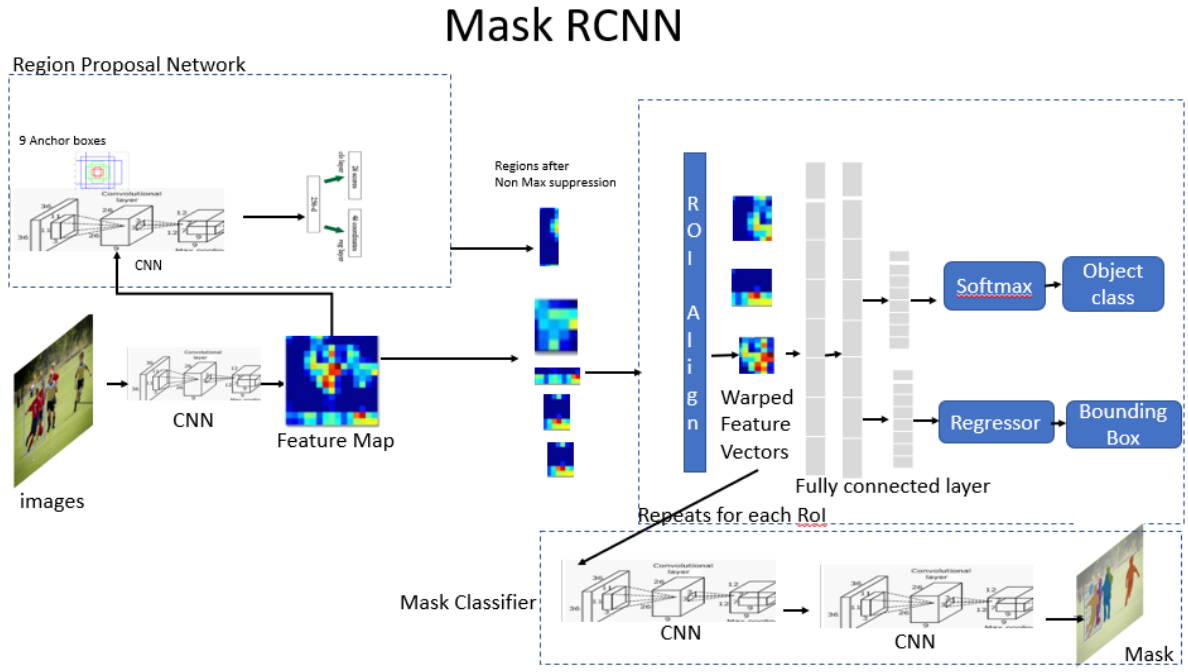

Figure 19: Mask R-CNN architecture. Source: [19].

RoI Pooling causes misalignments from the feature map as it uses rounding, which is not important for classification and bounding box detection but has a negative impact when trying to predict pixel-by-pixel segmentation masks. For example, if the RoI upper left corner is at pixel (26, 37), and the CNN that outputs the feature map has a total stride of 16, the corresponding pixels in the feature map would be at (24/16, 37/16) = (1.625, 2.3125), which, when using RoIPooling is rounded to (2, 2) and the value from that pixel from the feature map is used.

Mask R-CNN replaces RoI Pooling with RoI Align, which avoids rounding the coordinates by using bi-linear interpolation, so that the output is aligned correctly with the input (feature map). In the previous example, the value for the upper-left corner of the roi would be the **bi-linear interpolation** of the four nearest pixels, in that case the pixels at (1, 2) , (1, 3), (2, 2) and (2, 3) in the feature map, which is the **weighted average of the values of those four pixels**, the weights are determined by the distance between the target point (1.625, 2.3125) and each of the 4 nearby pixels. Figure 20 shows an example of RoI Align.

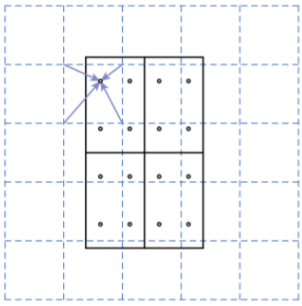

Figure 20: If the final RoI size is 2x2 (solid line), the original RoI is divided in 4 sections, and 4 regularly sampled points are used in each section (dots). The value of each sampling point is obtained using the bi-linear interpolation with its four nearest points in the feature map (dashed grid). Finally, MaxPooling or AveragePooling is used on those four sampling points to get a final RoI size of 2x2. Source: [18].

From the RoI align output, a **CNN branches out to predict the segmentation masks in parallel** to the object classification and bounding box regression, as shown in Figure 21. The architecture of this CNN is straight-forward and depends on the backbone used for the feature map generation, which can be ResNet50, which is the one used originally by Faster R-CNN, or **FPN** (**Feature Pyramid Network**) which uses convolutions of different scales and connect them horizontally to a neural network. FPN has residual layers like ResNet so the backbone is called **ResNet-FPN** and is the main backbone for Mask R-CNN.

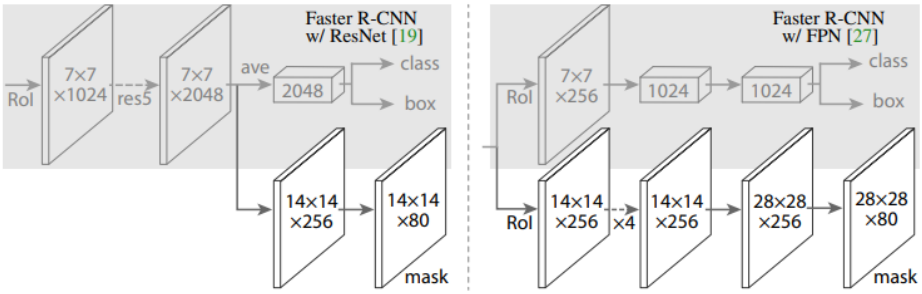

Figure 21: The CNN branch that predicts segmentation masks, depending on the backbone. Source: [18].

For each RoI the Mask network outputs **k segmentation masks**, one for each possible class, and the result of the Faster R-CNN object classifier decides which mask will be the final output. During inference, only masks from the 100 highest scoring predicted bounding boxes are generated, speeding up the test time. Overall, Mask R-CNN adds an overhead of 20% to the Faster R-CNN computation time, and generally runs at **5 fps** in real time detection.

2.2.5. Feature Pyramid Networks (FPN)

Historically, image pyramids were used for testing in object detection tasks, which consisted in using copies of the same image at different scales and detect objects in each of them separately to check if the model can detect the object at different scales. The feature maps generated at each scale would be different, with bigger scales resulting in more **low-level features** (meaning, more specific and less generalizable), than smaller scales.

To train more precise object detectors for multiple scales, **Feature Pyramid Networks** [20] emulate and improve image pyramids by generating feature maps of the same image at multiple scales, created by a bottom-up CNN with a scaling step of 2 in a process called **bottom-up pathway**. Although many layers produce feature maps of the same size, those layers are considered the same network stage. The output of the final layer of each stage is considered one of the feature maps to be used. Those feature maps extract low-level features at the bottom scales, and **higher-level features** as they go up. Figure 22 compares FPN with other pyramidal methods.

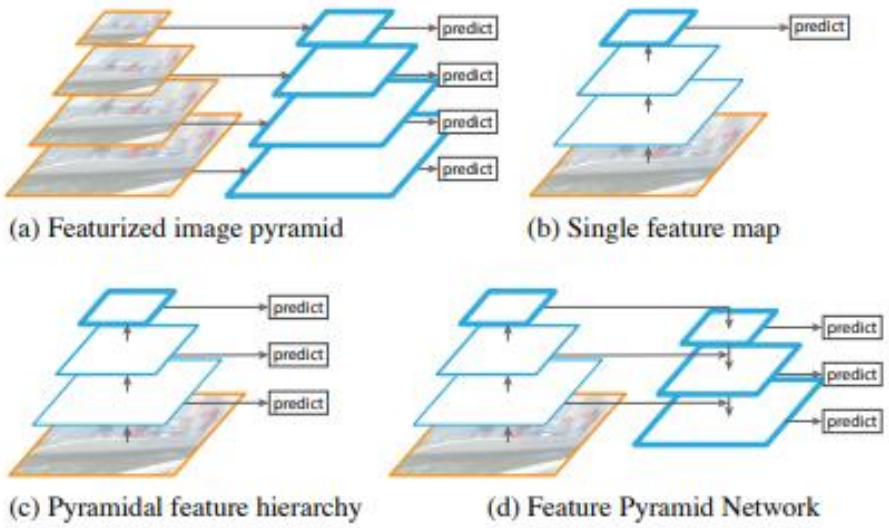

Figure 22: Differences between using convolutions on different scales of the same images (a), a normal multi-layer convolutional feature map extraction (b), using the pyramidal feature hierarchy of a convnet for predictions to emulate image pyramids (c) and Feature Pyramid Networks (c). Source: [20].

FPN use residual blocks like **Resnets** [12]**,** meaning that the outputs of a layer are added to the inputs of other layers. The activations of the final feature maps of each stage are used as residual inputs and are called (C2, C3, C4, C5) for the convolutional layers conv2, conv3, conv4 and conv5, which have strides of 4, 8, 16 and 32 pixels with respect to the original image. The output of conv1 is not used as it would need more use of memory. FPNs often use a **ResNet-50** as a bottom-up pathway.

The network follows the first stage with a **top-down pathway with lateral connections,** which take the last output of the bottom-up pathway, which is the smaller in scale, passes it through a 1x1 convolutional layer to reduce its channel dimensions and upsamples its spatial resolution by a factor of 2, generating the next upscaled feature map. Each upscaled feature map has a lateral connection with its sibling feature map from the bottom-up path, which has the same resolution. The sibling bottom-up feature map goes through a 1x1 convolution layer to reduce the cannel dimensions and is **added** to the upsampled result from the previous top-down feature, as shown in Figure 23. Finally, the merged maps go through a 3x3 convolution resulting in the final next top-down feature map, which is upsampled again and the process is repeated until the largest map is generated. The final set of feature maps is called (P2, P3, P4, P5) corresponding to (C2, C3, C4, C5).

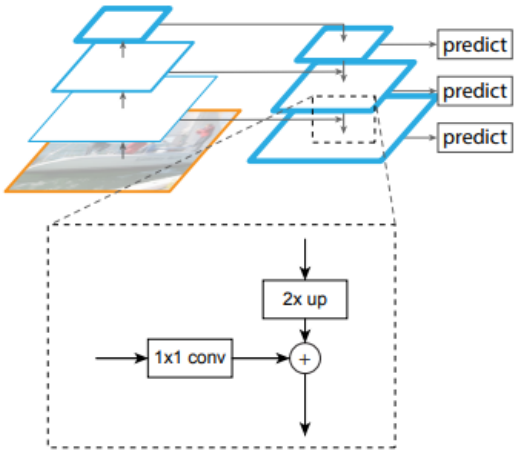

Figure 23: Lateral connections add the bottom-up feature maps to the top-down upsampled features. Source: [20].

Feature Pyramid Networks are not a complete object detection model, instead, they are used as a new backbone feature map for **Region Proposal Networks (RPN)** for Faster R-CNN and Mask R-CNN models. The RPN layers that predict the bounding box and objectness score are attached to each of the feature maps from the FPN, sharing the same trained parameters. Instead of using anchors of multiple scales at each step of the RPN sliding window, it only uses one scale for each of the feature pyramid levels as they already capture features at different scales. When used for RPN, FPN generates one extra feature map called P6, which is simply a stride two upsampling from P5, with no lateral connections. Fixed scales with areas of (32^2, 64^2, 128^2, 256^2, 512^2) are used for (P2, P3, P4, P5, P6) respectively, with 3 aspect ratios of (1:2, 1:1, 2:1), resulting in a total of 15 total anchors over the pyramid. Figure 24 shows the difference of predicting different scales instead of using the last output.

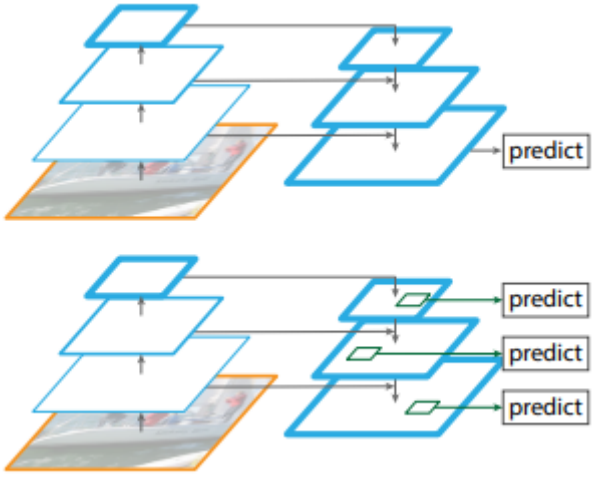

Figure 24: Predictions are carried out separately on each level (bottom) and not only on the final output (top). Source: [20].

FPN is also used as the backbone feature map for **Fast R-CNN**, which is used as the final object detection stage of Faster R-CNN and Mask R-CNN models. Instead of extracting ROIs from one single feature map, they **are extracted from one of the FPN levels depending on the height and width of the ROI**. Larger ROIs are extracted from lower, bigger levels, and smaller from higher ones. Those ROIs go through ROI pooling to become a 7x7 feature and are fed to two fully connected layers to predict bounding box coordinates and class labels as usual.

2.3. Study of One-Stage Object Detection

Although Faster R-CNN gets closer to real-time object detection, it does not get near to the standard 24 fps in movies or 60 fps that most video players can achieve. Even after optimizing the test time, Faster R-CNN and similar models still rely on a region proposal stage before the final classification and bounding box regression stage, thus being called two stage object detection models.

One-Stage object detection models try to detect objects without the region proposal stage, increasing the detection speed and achieving real time object detection.

2.3.1. You Only Look Once (YOLO)

While models such as Faster R-CNN use a pretrained object classification backbone on region proposals and bounding box regression separately **YOLO** [21] approaches object detection as a single regression problem by predicting the class and bounding box coordinates in the same layer and by processing the input image **only once**.

To do this, YOLO splits the image into an **S x S grid**, for each grid cell, it predicts **B bounding boxes centered at that cell** encoded as x, y, w, h, where (x, y) are the coordinates for the **center of the bounding box**, and (w, h) represent its width and height, respectively. For each bounding box, YOLO predicts one **confidence score** that reflects how confident the model is that the box contains an object, which is like the "objectness" score from the RPN. This score is equal to the highest IOU of that box with any ground truth box. Finally, at each cell the model also predicts **C class probabilities**, which are multiplied by the confidence score at test time to get a final detection probability. Final output boxes are filtered using the usual method of **non-max suppression** and their class is the highest scoring class probability in the grid cell where the box is centered. Figure 25 shows a visual example of the process used by YOLO.

YOLO was trained with the parameters S=7, B=2 on the Dataset Pascal VOC, which has 20 labeled classes, so C=20 and the final output tensor shape was 7 x 7 x (5 x 2 + 20) = 7 x 7 x 30.

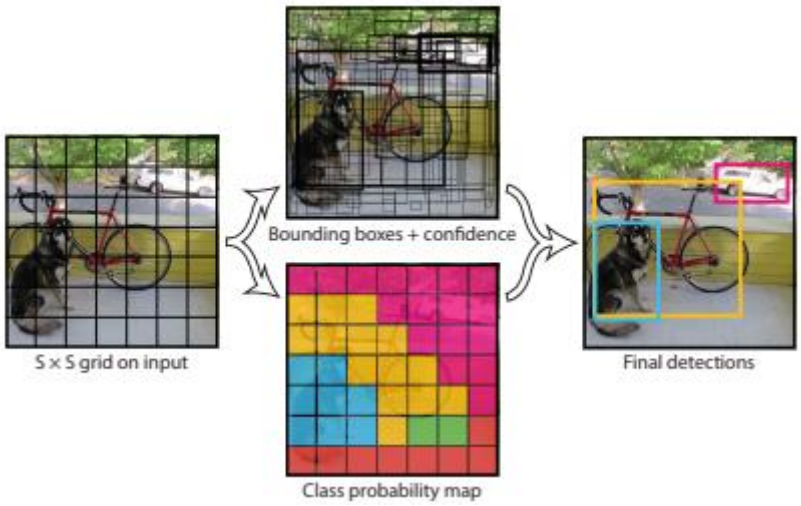

Figure 25: YOLO divides the original resized image into a S X S grid, predicts B bounding boxes and C class probabilities for each cell and a confidence score for each box. Predictions are encoded as an S x S x (5 x B + C) tensor. Source: [21].

YOLO is implemented as a convolutional neural network with no pretrained backbone. It has 24 convolutional layers followed by 2 fully connected layers, with the final layer generating the output as a 7 x 7 x 30 tensor. Figure 26 shows the architecture of YOLO.

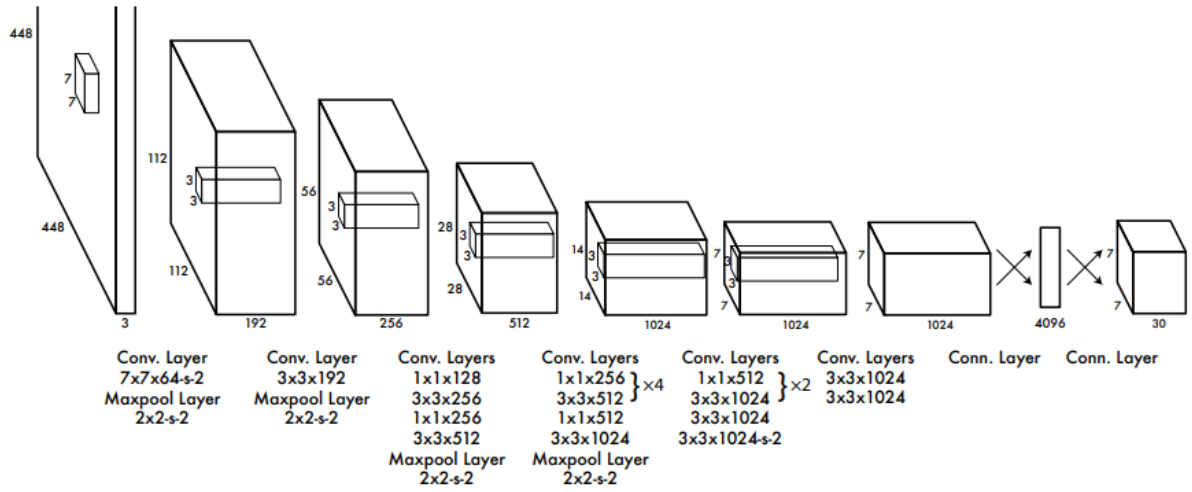

Figure 26: The YOLO network architecture. The image only goes through the convolutional network once. Source: [21].

As YOLO has no pretrained backbone, the first 20 layers, followed by an average pooling layer and a fully connected layer, were first trained on the classification dataset **Imagenet** [14] as a classification model, and then those learned features were used on the final YOLO model to learn object detection faster. The process of using pretrained models and changing the final layers to get a different output is called **Transfer Learning** or **fine tuning** and is useful to get good results with only a few trainable layers as the general features were learned on the pretrained ones.

By avoiding splitting the problem into two stages, YOLO achieves an inference speed of **45 frames per second**, beating Faster R-CNN by a big margin and achieving real time object detection with good detection accuracy of **63.4 mean Average Precision (mAP)**, while **Faster R-CNN had more than 70 mAP**.

The main limitation of YOLO is that it has spatial constrains, **as each grid cell can only predict one class** and two bounding boxes, struggling with small objects that appear in groups such as flocks of birds. YOLO also struggles with bounding box sizes and aspect ratios that has not learned on its training dataset, as it **learns the boxes coordinates from scratch**, instead of using anchor boxes like Faster R-CNN.

2.3.2. YOLOv2

YOLOv2 [22] improves the original YOLO accuracy and speed by tuning the neural network layers, training the network with randomized image rescaling, adding a residual layer from a bigger feature map to the final one and adding **batch-normalization layers,** which **standardize** the layer outputs of a batch, making them have a mean of 0 and a standard deviation of 1.

But the biggest change in YOLOv2 is the adoption the **anchor box** system from Faster-RCNN (RPN) instead of having the network predict the boxes coordinates from scratch. YOLOv2 also **decouples the class prediction from the cell location**, predicting it for each anchor box along with the objectness score instead. As the original YOLO only predicted 2 boxes per cell grid on a 7x7 grid, it predicted a total of 98 bounding boxes. With anchor boxes, YOLOv2 predicts more than a thousand.

Instead of setting anchor box sizes using handpicked scales and aspect ratios, a **k-means algorithm** is run during test time on the test dataset to find the most common bounding box shapes, using the k-means centroids as the anchor-box sizes [Figure 27]. YOLOv2 also tunes the anchor box system by making it predict the box locations relative to the location of their grid cell instead of the full image.

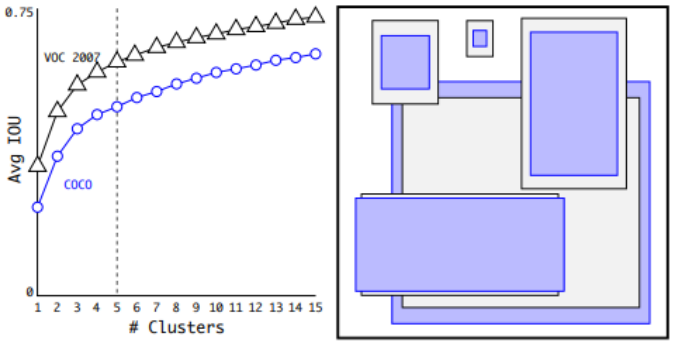

Figure 27: Using K-means to get the main box shapes. Source: [22].

YOLOv2 has a modified backbone from YOLO, called **Daknet-19** is trained on the image classification dataset ImegeNet with **1000 different classes**. Using this pretrained network before training the model for detection after changing the last layers makes the model able to detect objects of classes that are not included on the detection dataset. Figure 28 shows the detailed architecture of Darknet-19.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 28: Darknet-19 architecture with the image classification head. Source: [22].

2.3.3. YOLOv3

YOLOv3 [23] improves YOLOv2 mainly by using a new backbone, called **Darknet-53**, which makes use of **residual layers** and does predictions at **different scales** in a similar way to **Feature Pyramid Networks** (FPN). Figure 29 shows the detailed architecture of Darknet-53.

	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 29: Darknet-53 architecture with the image classification head. Source: [23].

Like YOLOv2, this iteration uses anchor boxes with predicted sizes using k-means. YOLOv3 will predict 3 bounding boxes at 3 different scales, so this time, 9 centroids are used as pre-chosen anchor box dimensions, called priors, and are divided evenly across 3 arbitrarily chosen scales. The last backbone feature map, which has the lower scale, will go through added detection convolution layers to predict 3 anchors boxes, where each has 4 coordinates, 1 objectness score and 80 class probabilities, resulting in a tensor of dimensions N x N x [3 * (4 + 1 + 80)] where N x N is the number of grid cells the image is divided into.

After the first set of detected boxes, the feature map from two layers before that result is upsampled by 2 and is added to a feature map from earlier in the network, which goes through the detection layers and predicts three more bounding boxes, resulting in a tensor twice the size of the previous one. This process is repeated with a previous feature map and a result of 9 bounding boxes is predicted.

YOLOv3 achieves an inference time of less than 50 milliseconds, achieving between 30 and 50 fps on videos, while having a mean average precision mAP on the COCO dataset [52] (also called COCO AP) between 28 and 33. At the time of its release, YOLOv3 was slightly less precise than the state-of-the-art detection models (in terms of precision) such as Faster R-CNN, but was a lot faster, as seen in Figure 30.

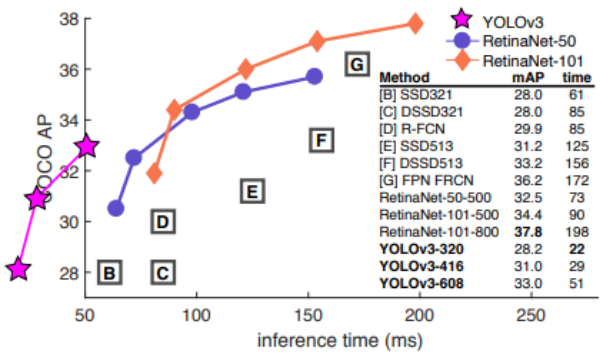

Figure 30: Comparison between precision and inference time of detection models in respect to YOLOv3. Source: [23].

2.3.4. YOLOv4

By the time **YOLOv4** [24] was released (2020), most object detector models had a similar structure where there was a backbone, neck, dense prediction and sparse prediction. The **neck** are layers between backbone and head that carry out the collection of feature maps at different stages of the backbone, having bottom-up and top-down paths like an FPN. The **dense prediction** stage is where bounding boxes are predicted from the layers of different scales of the neck, like **RPN** for two stage detectors and **YOLO** for one stage The **sparse prediction** stage is exclusive to two stage detectors, where the dense predictions (**ROI**s) go through the final detection layer, like in Fast R-CNN. Figure 31 shows a representation of standard object detection architecture at the time.

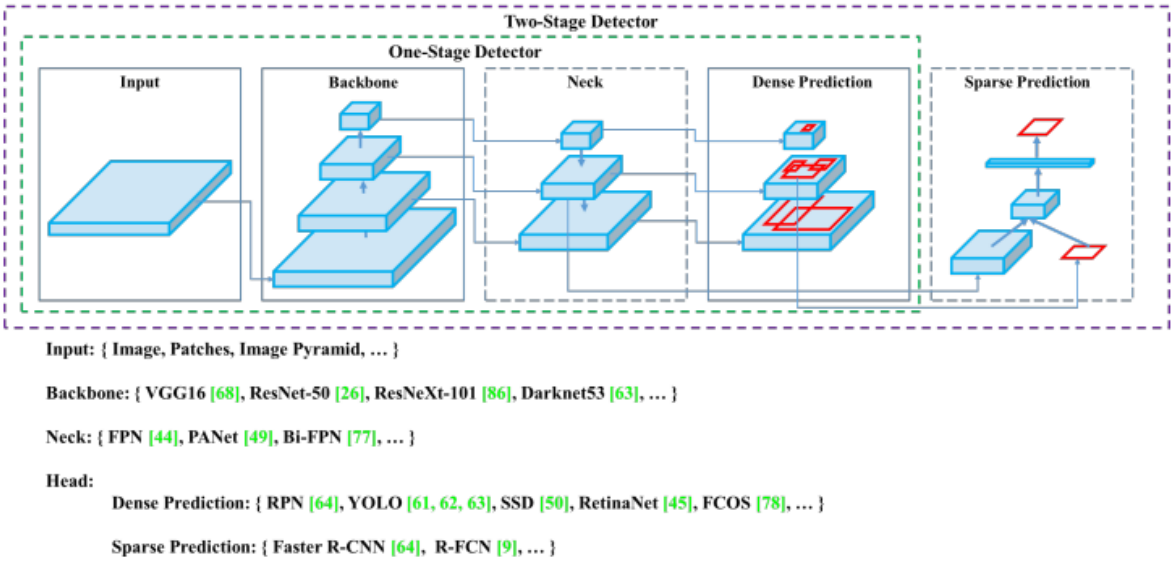

Figure 31: The standard Object Detection Architecture. Source: [24].

YOLOv4 groups its improvement methods into two categories. The first one, called **bag of freebies,** includes methods that increase the model accuracy without increasing the inference cost by only affecting the training stage. **Data augmentation and Regularization** methods are part of this group. Data augmentation makes the model less prompt to overfit on the training data, as the data is changing randomly with random transformations such as rotations, change of brightness, size and more. YOLOv4 uses **CutMix** [25], which cuts a random region from an image and fills them with a patch from other one, and **Mosaic** data augmentation, which takes blocks of 4 different images and merges them into one. Another data augmentation technique used by YOLOv4 is Self-Adversarial Training (SAT), which modifies images in a similar way to Generative Adversarial Networks. Figure 32 shows an example of CutMix and Mosaic.

Figure 32: Examples of CutMix (left) and Mosaic (right) data augmentation. Source: Google Images.

There are techniques like data augmentation applied to feature maps and not to the original image that serve as **regularization** methods, whose function is to avoid overfitting on the training data. On popular method in deep learning is **DropOut** [26], where random pixels from feature maps are converted to 0 (black), or **DropBlock** [27], where blocks of pixels are set as 0 instead of individual ones. YOLOv4 uses DropBlock regularization. **Class label smoothing** is another method of regularization, which consists of altering the class prediction results to prevent having one class probability being much larger than the others. Figure 33 shows an example of DropOut and DropBlock used on feature maps.

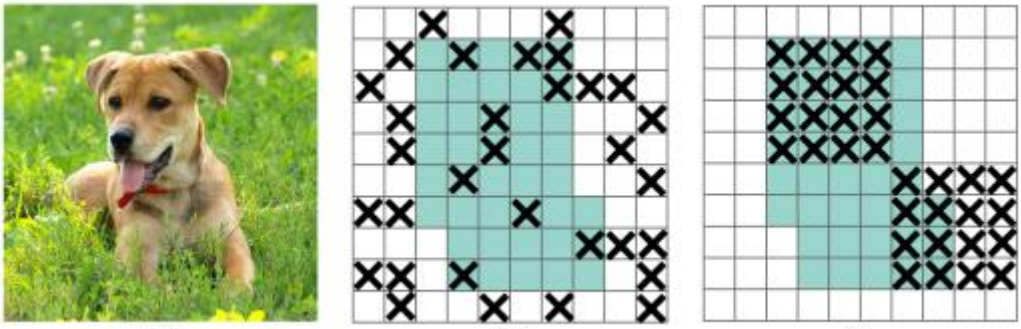

Figure 33: Examples of DropOut (b) and DropBlock (c) regularization on feature maps extracted from an image (a). Source: [27].

Another element from the bag of freebies is Cross Mini-Batch normalization, which is a modified version of batch normalization where the mean and the standard deviation are computed, normalized and acummulated to subsequent mini-batches inside the same batch.

The last element of the bag of freebies is the loss function for bounding box regression. Traiditional object detectors use the **Mean Squared Error (MSE)** of the 4 bounding box coordinates. However, this method treeats each coordinate as an independent variable and does not take into account the integrity of the object. To fix this issue, **loss functions based on IoU** are proposed. YOLOv4 uses **Complete IoU Loss (CIoU)** [28], which takes into account the IoU, the distance between center points and the aspect ratio similarity between the predicted and the **ground truth box**. The CIoU loss is defined as follows:

$$\mathcal{L}_{CIoU} = 1 - IoU + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2} + \alpha v$$

where $\rho(b, b^{gt})$ is the **Euclidean Distance** between the boxes central points, c is the diagonal length of the smallest enclosing box covering the two boxes, α is a positive trade-off parameter, and v measures the consistency of aspect ratio with the following formula:

$$v = \frac{4}{\pi^2}(arctan\frac{w^{gt}}{h^{gt}} - arctan\frac{w}{h})^2$$

Where w and h are the width and height of bounding boxes.

The other group of improvement methods is called **bag of specials**, and include methods that improve the model precision while increasing inference time by a small ammount.

One of the modules in the bag of specials is the **enhancement of the receptive field**, which is the size of the input image pixels that a final feature map output value depends on. For example, the result of one convolutional iteration with a 3x3 kernel in the original image has a 3x3 receptive field. If one feature has a larger receptive field, it means its information contains a bigger context of the image. Figure 34 shows a visual representation of the receptive field concept.

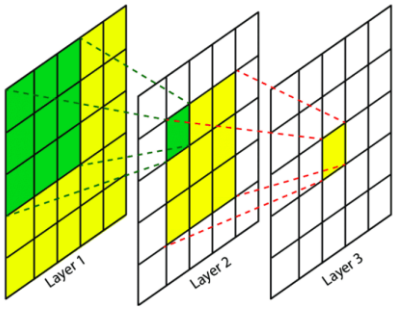

Figure 34: Example of receptive fields with 3x3 kernel size convolutions. Green cells in Layer 1 represent the receptive field of the green cell in layer 2. Green and yellow cells from Layers 1, 2 represent the receptive field of the yellow cell in Layer 3. Source: Google Images.

Spatial Pyramid Pooling (SPP) [29]**¡Error! No se encuentra el origen de la referencia.** is one of the methods that enhances the receptive field. SPP is similar to ROI Pooling but with multiple scale outputs, it uses a feature map and divides it into multiple N x N grids with different values of N, and then uses max-pooling on the pixels inside each separate cell, flattens each grid output and concatenates them in a single one dimensional vector. As a one dimensional output is not fit for using it as a feature map, YOLOv4 uses a modified version of SPP by converting it into a concatenation of max-pooling outputs of k x k sized kernels where k = {1, 5, 9, 13} and stride 1. This results in multiple feature maps of the same size but with different receptive field sizes. The SPP module is added after the final layer of the backbone. Figure 35 shows the original and modified version of SPP.

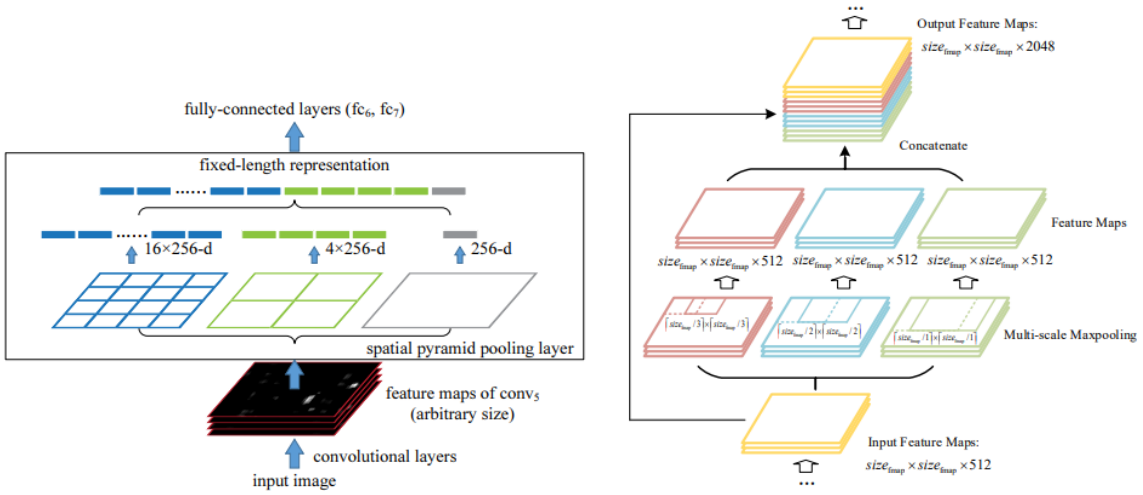

Figure 35: Representation of Spatial Pyramid Pooling (SPP) (left) and the YOLOv4 modified version (right). Source: [29].

Like **FPN**, YOLOv4 uses bottom-up paths and top-down paths with lateral connections. Instead of using FPN like YOLOv3, YOLOv4 uses a modified version of **Path Aggregation Networks (PANet)** [30], that includes another bottom-up path after the top-down path with lateral connections, called **bottom-up path augmentation**. YOLOv4 modifies PAN lateral connections to **concatenation** instead of addition as shown in Figure 36. The PAN takes outputs from different layers of the backbone as well as the SPP output as its smallest scale input (the top of the top-down path), and **the final feature maps from each PAN stage are fed to a YOLOv3 head** that uses 3 anchor boxes on each stage to predict a total of 9 bounding boxes per pixel, each with their objectness score and class prediction.

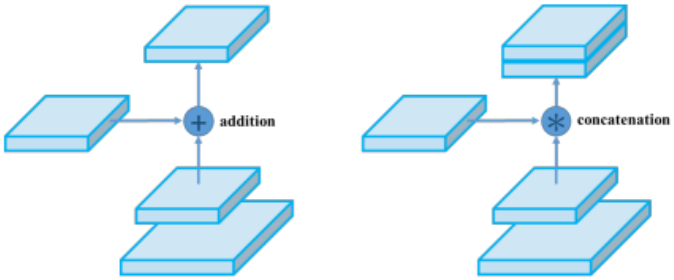

Figure 36: Original PAN aggregation method (left) and the YOLOv4 PAN modification (right). Source: [24].

An additional block is added to the neck, the **Spatial Attention Module (SAM)** [31], which makes use of the attention mechanism. Attention is referred as a mechanism in deep learning where one feature "looks" at previous inputs. For example, in a neural network that translates sentences from one language to another, the output translation of a certain word might depend on previous words, so the network has an attention mechanism where it considers the importance of previous words for that input, usually in the form of a vector with the same size as the input size, with the values representing the importance ratio between 0 and 1. Spatial Attention translates this mechanism to images or feature maps, where each pixel has its attention value.

SAM does this by concatenating the results of a max-pooling and average-pooling layers over the input, and applying a convolution with a **sigmoid** activation function, resulting in the spatial wise 2D attention map, which is multiplied with the original 3D input, having each channel value multiplied by the attention value of its (x, y) position, to get the result. YOLOv4 modifies SAM by avoiding the max-pooling and average-pooling layers, and instead uses one convolution layer with the output of the same size as the input with a **sigmoid activation** to get a **3D attention map**, which multiplies each pixel of the input by the attention pixel at its (x, y, z) position, as shown in Figure 37. This modified SAM block is added in the **PAN** module after each lateral connection between the bottom-up path augmentation and the previous top-down path. Figure 38 shows the full detailed architecture of the PAN neck for YOLOv4, as well as its YOLOv3 detection head.

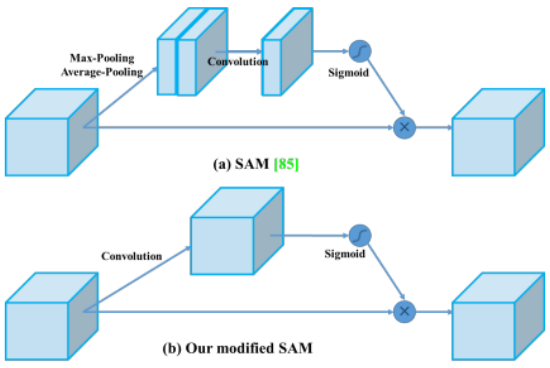

Figure 37: The original SAM (top) and modified SAM for YOLOv4 (bottom). Source: [24].

Another element in the bag of specials is the change of the **activation function** for most layers from ReLU, which outputs 0 for every negative input, to **Mish**, which can make the gradient more efficiently propagated without much extra computational cost and proved to be more effective for YOLOv4. The **Mish activation function** is defined as: **f(x) = x*tanh(softplus(x))**, where the softplus function is defined as follows: **softplus(x) = ln(1+e^x).**

The last element in the bag of specials is the modification of the **Non-Max Suppression (NMS)** algorithm. YOLOv4 uses **DIoU NMS ¡Error! No se encuentra el origen de la referencia.** instead, which uses **Distance IoU (DIoU)** to filter out boxes instead of normal IoU. DIoU is the same as the **CIoU** formula without the aspect-ratio parameter. DIoU is defined as follows:

$$\mathcal{R}_{DIoU} = \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2}$$

And **DIoU NMS** has the following logic:

$$s_i = \begin{cases} s_i, & IoU - \mathcal{R}_{DIoU}(\mathcal{M}, B_i) < \varepsilon, \\ 0, & IoU - \mathcal{R}_{DIoU}(\mathcal{M}, B_i) \geq \varepsilon, \end{cases}$$

Where Ɛ is a set threshold parameter.

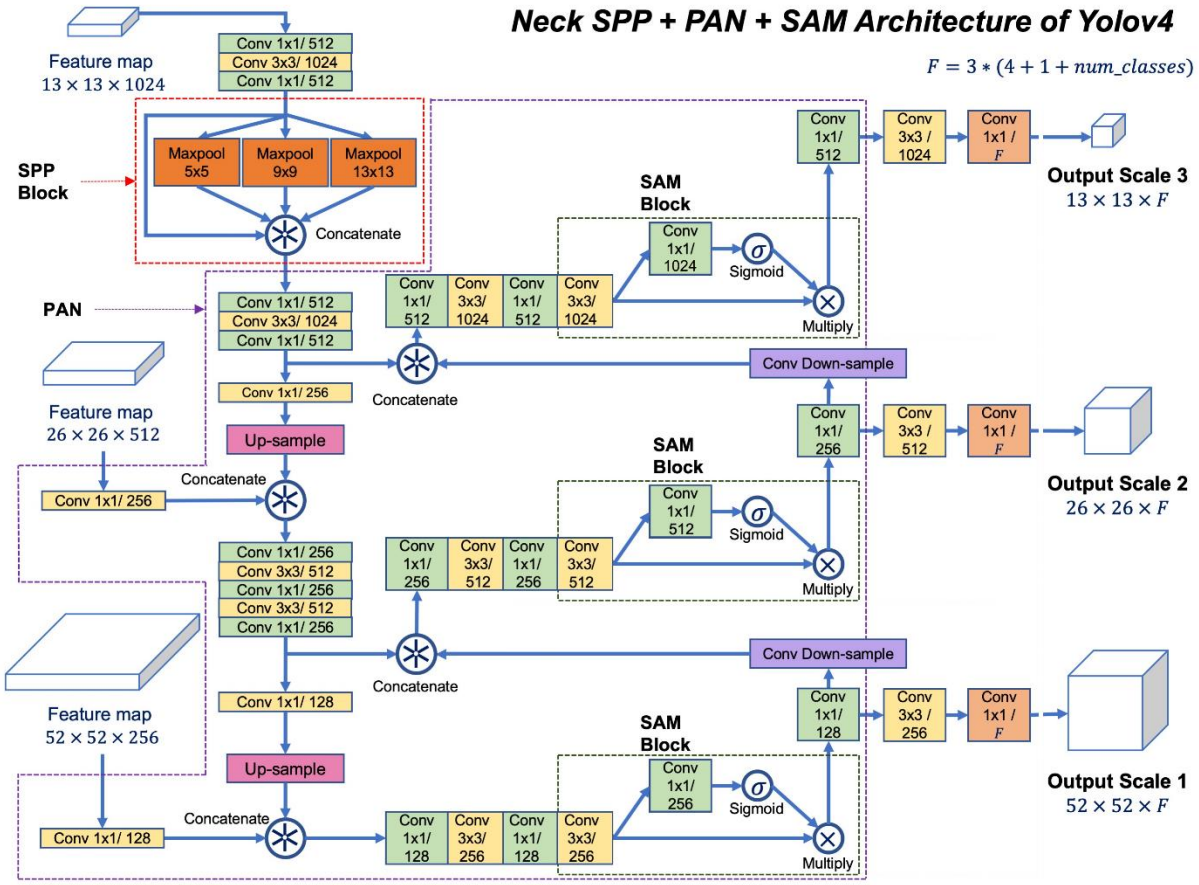

Figure 38: Inside the YOLOv4 SPP + PAN neck and YOLOv3 head. Source: corrected image personally from [33].

YOLOv4 uses the same backbone as YOLOv3, the DarkNet-53, but it modifies it by using **Cross-Space Spatial connections (CSP)** [32]**,** which splits feature maps into two copies, has each copy go through convolutional layers and concatenates the results of both copies afterwards. The backbone is then renamed to **CSPDarkNet-53**. Figure 39 shows the architecture of the CSPDarknet-53.

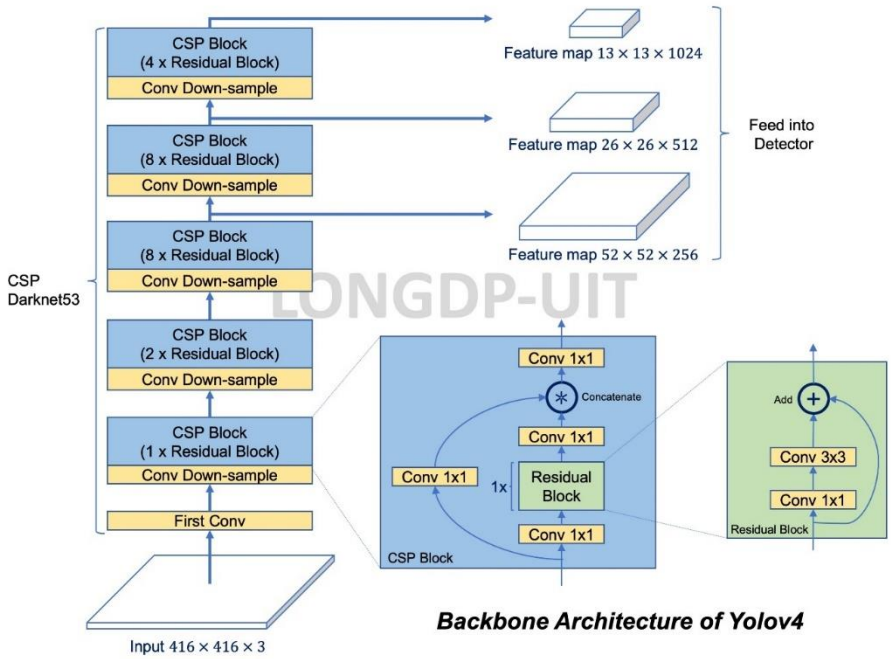

Figure 39: Architecture of CSPDarknet-53. It has 5 CSP blocks that contain a different number of residual blocks. The outputs of the last 3 CSP blocks is fed to the PAN neck, having the last output go through the SPP first. Source: [33].

YOLOv4 has the mindset of being able to be trained and tested on single standard GPUs while achieving real time inference and became the start of the art of real-time detectors when it released in 2020, achieving **43.5% mAP** on COCO dataset at a real time speed of **65 FPS** on a Tesla V100 GPU as shown in Figure 40.

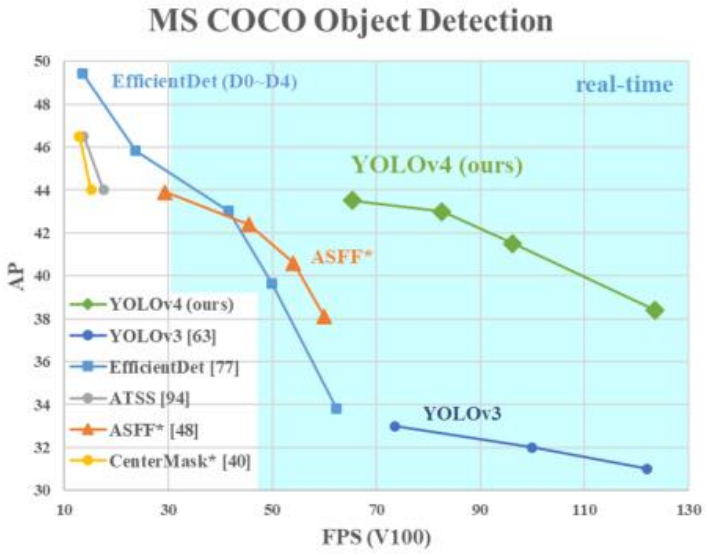

Figure 40: Comparison of YOLOv4 and other state of the art object detectors at the time of its release. Source: [24].

2.3.5. YOLOv5

What once started as a **Pytorch YOLOv4** implementation with few changes has now become the most popular real-time detector. The nature of **YOLOv5** [34] is quite different from previous models, as **it is not a research project and does not have a research paper**, instead, it has a whole company behind its **continuous development** and is being constantly updated with tweaks to the architecture or adding support with different third-party software. YOLOv5 also features models of different "size" options, from faster and less precise models to slower and more precise ones that uses more feature map scales for anchor box prediction. Figure 41 shows the different size options for YOLOv5.

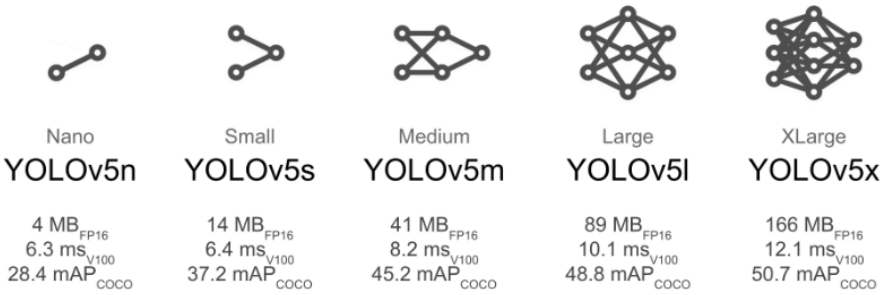

Figure 41: Size options for YOLOv5 with their COCO precision and inference speed on a Tesla V100 GPU. Source [34].

The continuous development and improvement of YOLOv5 makes it difficult to describe its changing architecture with detail, which can be checked in the source code, but **its general structure is the same as YOLOv4** [Figure 42**]** with small changes to layer placement and parameters. The bigger options of YOLOv5 at the 6.0 version include **1 extra output layer**, changing the downsample and upsample rates to use 4 feature maps with lateral connections and predicts boxes at **4 different scales** instead of 3.

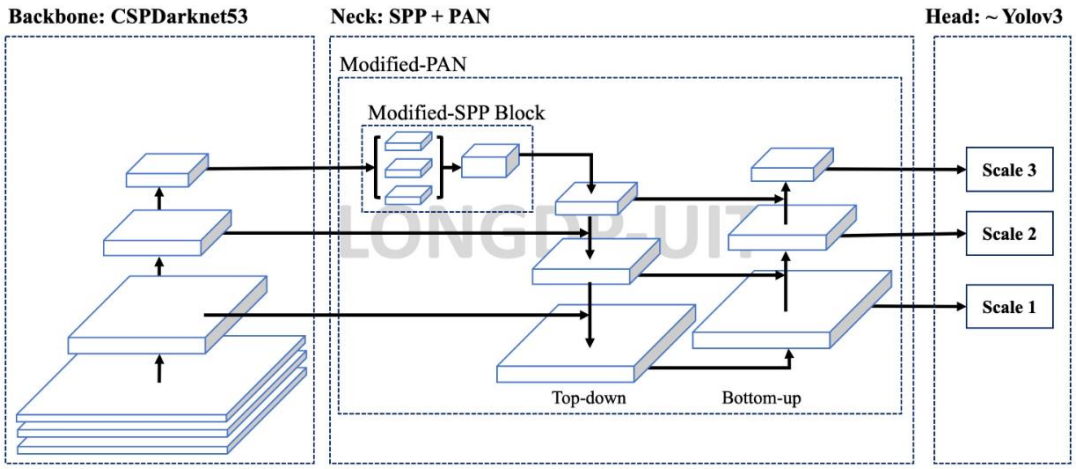

Figure 42: The general architecture of YOLOv5 is the same from YOLOv4. Source: [33].

YOLOv5 replaces the **Mish** activation function with **SILU**, which is equal to the **sigmoid function** multiplied by its input, giving a similar output to RELU, has a modified SPP block, called **SPPF,** uses different formulas for box coordinates calculation and uses a different method for anchor box templates assignment to grid cells by having cells use ground truth boxes centered at the edge of an adjacent cell. YOLOv5 also uses more data augmentation methods. Figure 43 shows the detailed architecture of YOLOv5 version 6.1.

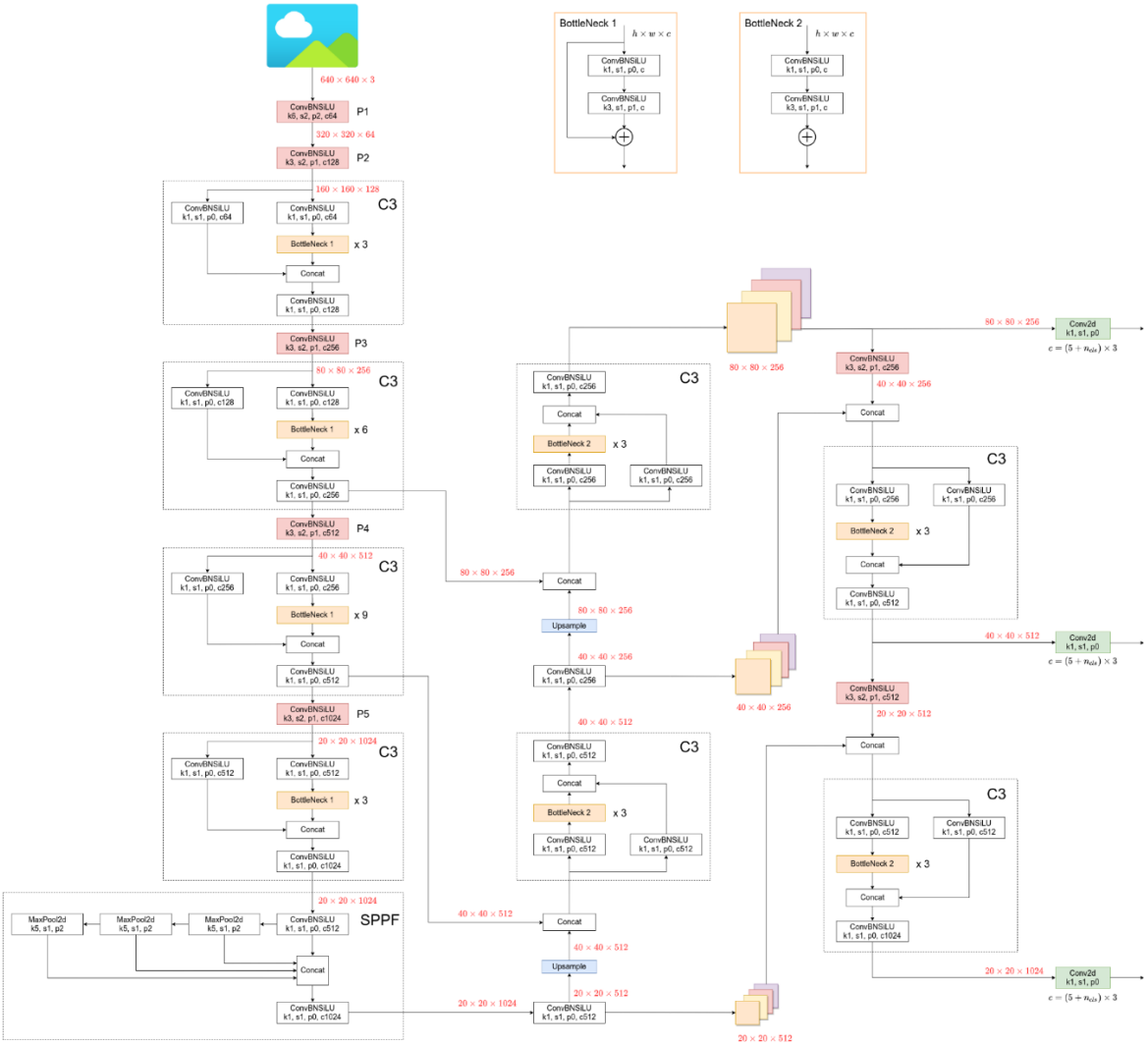

Figure 43: YOLOv5 6.1 detailed architecture, YOLOv5l (large option) in particular. Check the source for a higher resolution and more info: [34].

2.3.6. Other One-Stage Object Detectors

Single Shot Detection (SSD)

SSD [35] was released shortly after the first version of YOLO and was a faster and more precise model. Its main difference was using a system like **anchor boxes**, which the first YOLO lacked, and carried out **predictions on different scales**, in a similar way to the first bottom-up path from FPN. Figure 44 shows the SSD architecture.

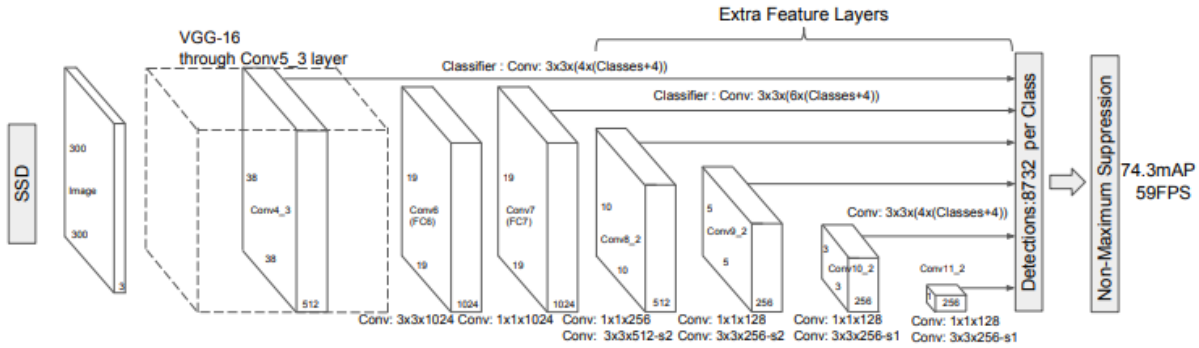

Figure 44: SSD architecture. The 74.3 mAP is the AP with 0.5 IOU threshold on the VOC2007 test dataset. Source: [35].

RetinaNet

RetinaNet [36] introduced the use of **FPN** on one-stage detectors **before YOLOv3** appeared. It used the FPN outputs on two subnetworks that predicted the class and bounding box regression **separately**. Figure 45 shows the RetinaNet architecture.

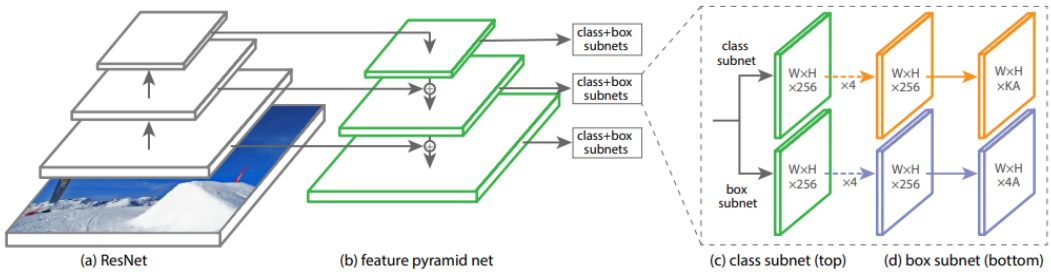

Figure 45: The RetinaNet architecture. Source: [36].

EfficientDet

The main feature of **EfficientDet** [37] is the use of a modified FPN called **Bi-directional Feature Pyramid Network (BiFPN),** which has multiple top-down and bottom-up paths like **PANet,** but with different lateral connections [Figure 46]**.** EfficientDet uses **EfficientNet** [13] as its backbone as its name suggests.

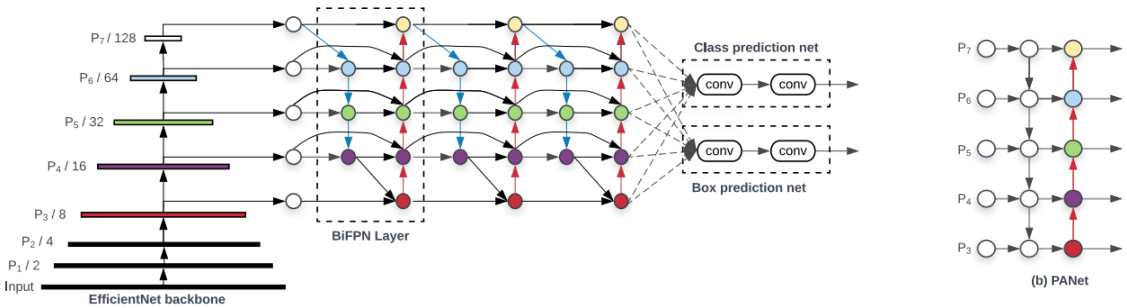

Figure 46: EfficientDet architecture (left). The BiFPN Layer is a modificed version of PANet (Right). Source: [37].

2.4. Evaluation Methods for Object Detection Models

Although this project focuses on the inference speed of object detection models to apply them on real time videos, it is also important to evaluate how accurate those models are at detecting objects.

For normal image classification, metrics such as **Accuracy, Precision and Recall** are usually used. For **object detection**, precision and recall are more important, and the **Intersection over Union (IoU)** is used to compute them. This section describes these metrics as well some additional ones used on the results of the Precision and Recall metrics.

2.4.1. Accuracy

Accuracy is the most straight forward metric as it simply measures the percentage of correct predictions by comparing them to the ground truth. The total number of correct predictions is divided by the total number of predictions to get the accuracy.

2.4.2. Precision

The model precision for a given class is the **ratio** of **correctly** made **positive predictions of that class**, also called **True Positives (TP), over the total number of positive predictions for that class,** which could include incorrect positive predictions, also called **False Positives (FP).** The precision is defined by the following formula:

$$Precision(c) = \frac{TPc}{TPc + FPc}$$

Where c is a given class, TPc is the number of True Positives assuming that c is the positive class and all the other classes are considered negatives. FPc the number of false positives of that class. Precision measures how close the model is to being right in all its positive predictions and is not affected by **incorrect negative predictions,** called **False Negatives (FN)**, which are positive in the ground truth. That means that precision will be equal to 1, its maximum value, if the model only predicts one True label out of hundreds and that one prediction is correct.

2.4.3. Recall

Recall is, for a given class label, **the ratio of correctly made positive predictions of that class (TP) over the total number of ground truth instances of that class**. The total number of positive ground truth instances of a class can also be computed as the total number of True Positives (TP) plus the total number of **False Negatives (FN)**. The recall is defined by the following formula:

$$Recall(c) = \frac{TPc}{TPc + FNc}$$

Where c is a given class, TPc is the number of True Positives of that class, and FNc the number of false negatives of that class. Recall measures how close the model is to not miss any ground truth True label of that class and is not affected by incorrect True predictions, called **False Positives (FP)**. This means that Recall will be 1, its maximum possible value, if a model predicts every instance as positive arbitrarily.

2.4.4. Average Precision (AP)

If a model arbitrarily predicts only one True label, and is a True Positive, Precision will be 1 but recall will be close to 0, as it will have lots of False Negatives (FN). On the other hand, if the model predicts all instances as positive, recall will be maximized but precision will be close to 0, as there will be lots of false positives (FP).

Thus, one way to evaluate a model is by leveraging the **Precision-Recall Curve**, which is a graph with precision and recall as axes, with each iteration having a different **confidence threshold** for the model predictions to be considered positive. With a confidence threshold of 0.1, all predictions that have 0.1 confidence or more of an object pertaining to that class will be considered positive.

To evaluate the Precision-Recall Curve, the area under the graph is computed, which will be always between 0 and 1 given the maximum values for Precision and Recall is 1, and that value is called **Average Precision (AP)**, as shown in Figure 47**.**

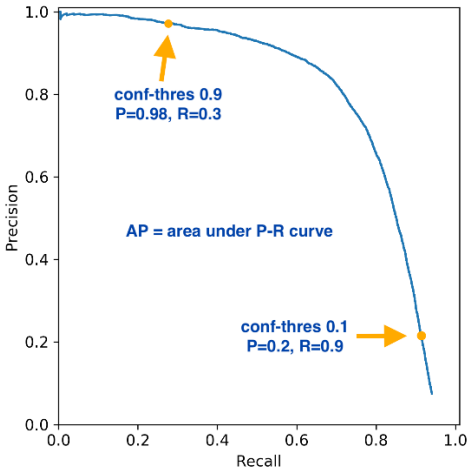

Figure 47: Example of a Precision-Recall curve. Source: Google Images.

2.4.5. Intersection Over Union (IoU)

It is trickier to know if a predicted bounding box should be considered a TP or FP in comparison to simple image classification, where each class is given a confidence score. In object detection the bounding box shape and position are important as well to determine if the prediction is correct, and the IoU is used for that. **The Intersection over Union (IoU)** evaluates the similarity between two bounding boxes which is computed by **dividing the area of their intersection by the area of their union.** Considering that the area of the union is the sum of the two areas minus the area of the intersection. We can define IoU as follows:

$$IoU(b1, b2) = \frac{|b1 \cap b2|}{|b1| + |b2| - |b1 \cap b2|}$$

Where b1 and b2 are bounding boxes and $|\cdot|$ is the area. IoU ranges from 0, where the two boxes do not intersect at all, to 1, where the boxes are identical and in the same position. For evaluation metrics, **IoU thresholds are used to determine if one detected box is a True Positive (TP) or a False Positive (FP)**. If a detected box IoU **with a ground truth box with the same class label is above the IoU threshold**, **that detected box is considered a TP, otherwise it is considered a FP**. Ground truth boxes with no TP detected boxes attached are considered FN.

2.4.6. Mean Average Precision (mAP)

Originally, the **Mean Average Precision (mAP)**, also called **mAP@0.5** of object detection models was the **mean of Average Precisions over all classes** with an **IoU threshold of 0.5**. To plot the Precision-Recall-Curve of a class, all the predicted boxes are **sorted by their confidence score from maximum to minimum** and a loop iterates over them, considering one prediction as a **TP if the IoU with the ground truth box is over 0.5 and as a FP otherwise**, and the precision and recall for that given instant is computed and saved in an array. If two predicted boxes are attached to the same ground truth box with IoU above the threshold, the one with higher confidence is considered a TP and the other a FP.

The precision is computed by dividing the TP by (TP + FP) **at that given iteration**, which is usually equal to 1 on the first iteration where the confidence score is higher and there is only one TP and zero FP. The recall is always computed by dividing TP at that iteration by the total number of ground truth boxes, being close to 0 in the first iteration as there would be 1 TP and a larger number of ground truth boxes. Figure 48 shows an example of the process for precision-recall computation.

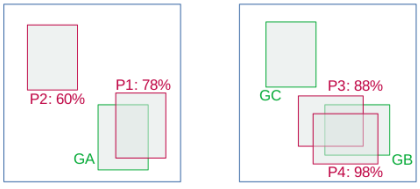

Image	Detection	Confidence	IoU	Ground Truth	TP/FP	Acc TP	Acc FP	Precision	Recall
Image 2	P4	98%	> 0.5	GB	TP	1	0	1	0.33
Image 2	P3	88%	> 0.5	GB	FP	1	1	0.5	0.33
Image 1	P1	78%	> 0.5	GA	TP	2	1	0.67	0.67
Image 1	P2	60%	< 0.5	-	FP	2	2	0.5	0.67

Figure 48: Representation of the process of creating a Precision-Recall for object detection. The values from the Precision and Recall columns are used to plot the curve. Source: [38].

The Precision-Recall curve is plotted using the results for each iteration, and the Average Precision (AP) is computed by finding the area under the curve. The process is repeated for each class and then the mean AP is computed, resulting in **mAP@0.5.**

If a different IoU threshold is used, for example 0.75, the resulting metric would be **mAP@0.75**. Other IoU thresholds would result in different curves, as shown in Figure 49. The **COCO Dataset** [52], which is a benchmark of object detection models, created a metric called **mAP@0.5:0.05:0.95**, also called **COCO AP** or simply **mAP or AP**, which is the **mean of mAPs over a range of 10 IoU thresholds starting at 0.5 and ending at 0.95 with a step of 0.05**.

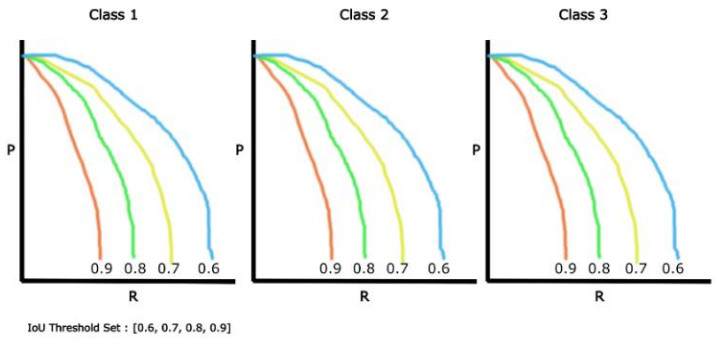

Figure 49: Example of Precision-Recall curves with different IoU thresholds. Source: Google Images.

3. Implementation and Evaluation

This chapter includes the work completed for this project to achieve the objectives from section 1.2.

3.1. Data Gathering

After the study of two-stage and one-stage object detection models, it is time to use them on videos to study their performance, both speed and precision-wise. **For training, all detection models require a labeled dataset** for detection in the form of images with their annotations in the form of bounding box coordinates and class labels. It is important to **split the data in a training and validation sets**, and optionally an extra **test set**, as the model must be able to generalize what it learned during training to unknown data. If the model is trained for too long on a training dataset it will memorize that data and will probably have poor results on the validation, this problem is called **overfitting.** One way to reduce overfitting is to **augment the training data** by applying **random transformations** to the images at each training iteration, like random flips, rotations and more. This topic was discussed in section 2.3.4.

To train my models, I searched online for **public annotated datasets** for underwater species detection on real-time videos, which are not very common. I found three main options: the **Help Protect the Great Barrier Reef Kaggle competition dataset** [2], the **FishCLEF-2015 dataset** [39]**,** and the **Luderick-Seagrass dataset** [40].

3.1.1. Help Protect the Great Barrier Reef Dataset

The aim of the **Help Protect the Great Barrier Reef** competition was training a model to detect **Crown of Thorns Starfish** in images, which were extracted frames from video files. This competition served as the inspiration for this project, and I started participating in the competition with little to no background knowledge about object detection models.

I trained an **EfficientDet** model with the **TensorFlow Object Detection API**, which was unintuitive for me at the time and had lots of compatibility problems with other libraries. The model was poor at detecting objects outside the training set as I did not shuffle nor augmented the training data and trained the model for too long, resulting in overfitting. The code for this model is not included in this project as it was a training experience, and **I did not use this dataset further for this project** as I thought it was less interesting than others due to the starfish being **motionless**. Figure 50 shows one of the best results of my trained EfficientDet model on an image from the validation set.

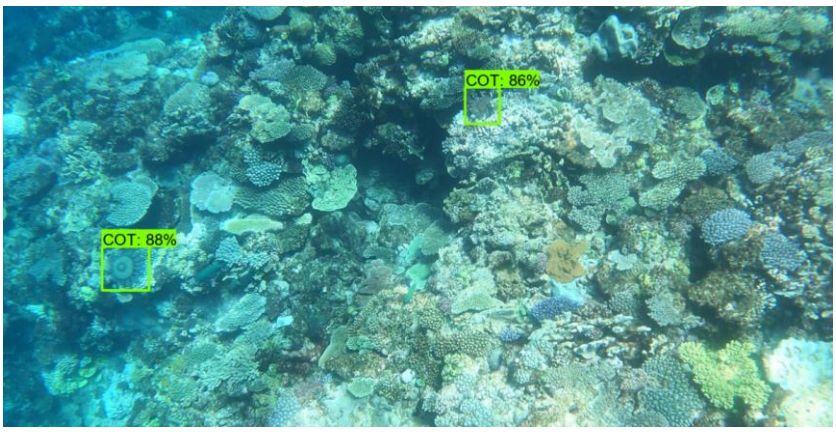

Figure 50: One output from my model for the Help Protect the Great Barrier Reef competition.

3.1.2. FishCLEF-2015 Dataset

The **FishCLEF-2015** dataset consists of **93 videos** of underwater scenery, with detection annotation data for each frame that includes bounding boxes around **15 underwater species**. The data is unbalanced as there are more than 3000 annotation for one species while there are less than 100 for others. This dataset could have been used for this project but was discarded in the end due to its **poor image quality** due to video compression artifacts, as it can be seen in Figure 51, which would have resulted in the model learning features that do not exist in reality, and real features and details from the species would be hidden due to the image quality.

Figure 51: One frame from the FishCLEF-2015 dataset. Source: video file downloaded from: [39].

3.1.3. Luderick-Seagrass Dataset

The Luderick-Seagrass dataset includes pre-extracted frames from video footage of the species ***Girella Tricuspidate***, also called **Luderick**, from Queensland, Australia. The dataset includes the original **video files** and the reduced number of annotated images, extracted at a rate of **5 frames per second** from the videos. Annotations include a few instances of another species, the Bream, but there are only 53 annotations for that class while there are more than six thousand annotations for Ludericks, so it cannot be considered a multi class detection dataset. **Annotations include segmentation data as well as bounding boxes,** in the form of pixel coordinates at the contours of each fish, as Figure 52 exemplifies, making it possible to train an Instance Segmentation model such as **Mask R-CNN** with this dataset.

The Luderick dataset has been chosen for this project for having a good amount of data in video form in **high definition at 1920 x 1080 resolution**, **easy to preprocess annotation data and the inclusion of segmentation data**. The downside of this dataset is that there is only one class to detect if we ignore the Bream.

Figure 52: An example image from the Luderick-Seagrass dataset with drawn segmentation annotations. Source: [41].

3.2. Data Preprocessing

The Luderick-Seagrass dataset includes 9429 annotations (including Bream) and 4280 annotated images for object detection and instance segmentation. The images have been extracted from the video at a rate of 5 frames per second. The dataset has been already split into training, validation and test sets. Table 1 shows information about each split, where the class unbalance can be noticed between ludericks and Breams.

Dataset	Split	Luderick Annotations	Bream Annotations	Total
Luderick Seagrass Jack Evans A	Training	6649	53	6702
Luderick Seagrass Jack Evans B	Validation	1632	29	1661
Luderick Seagrass Tallebudgera	Test	1023	43	1066
Total		9304	125	9420

Table 1: Details of the Luderick-Seagrass dataset splits.

All images and videos have a resolution of 1920 x 1080 pixels. Images are in JPG format and videos are in MP4 format running at 30 frames per second (fps). Figure 53 shows one image from the training set where three ludericks can be seen. All images are included in the same folder and their total size is 3.5 Gigabytes. The video files can be downloaded separately, and they are only used for video inference, as there is not annotation data for each frame.

Figure 53: Image example from the Luderick dataset.

Annotations are included in two formats: The JSON format, and the **CSV format**. Both formats contain the same information, and I used the CSV format for better compatibility with the **Pandas** library for python, which can read CSV files and convert them to **DataFrames**, a data structure in a tabular form that is often used in data analysis or machine learning tasks. The CSV annotations include one row for each single object instance and includes the bounding box coordinates, the contour coordinates for segmentation, the class label, box area and the image where it appears. All images include at least one instance of a Luderick or a Bream, so there are no images with 0 annotations. Table 2 shows the structure and format of each row in the CSV annotation file format.

Column	Description
id	INT annotation ID
catecory	STR name of category (luderick/bream)
category_id	INT category ID
image	STR image file name
image_id	INT image ID
bbox_x	INT minimum x pixel coordinate of bounding box
bbox_y	INT minimum y pixel coordinate of bounding box
bbox_w	INT width of bounding box in pixels
bbox_h	INT height of bounding box in pixels
area	INT area of bounding box in pixels squared
segmentation	STR segmentation polygon coordinates in format "[[x, y, x, y, ...]]"

Table 2: Structure of the CSV annotations file. INT and STR refer to Integer and String format for their respective columns.

I created the **Data Preprocessing Luderick Seagrass Jupyter Notebook.** Jupyter Notebook is a file format that offers the option to run python code divided by cells and lets the user write text-based cells as well. I start by loading the CSV annotations files for each split with the Pandas library, converting them to a Pandas DataFrame. Figure 54 shows the training split in the dataframe format.

	id	category	category_id	image	image_id	bbox_x	bbox_y	bbox_w	bbox_h	area	segmentation
0	0	luderick	1	04C1_Luderick_1.mov_5fps_000001.jpg	0	1778	425	141	185	26085	[[1778, 515, 1789, 498, 1806, 479, 1835, 457, ...
1	1	luderick	1	04C1_Luderick_1.mov_5fps_000002.jpg	1	1659	406	260	239	62140	[[1659, 509, 1675, 488, 1696, 465, 1720, 437, ...
2	2	luderick	1	04C1_Luderick_1.mov_5fps_000003.jpg	2	1578	400	337	220	74140	[[1578, 530, 1585, 512, 1594, 496, 1592, 477, ...
3	3	luderick	1	04C1_Luderick_1.mov_5fps_000004.jpg	3	1508	398	304	228	69312	[[1812, 624, 1795, 578, 1795, 542, 1797, 507, ...
4	4	luderick	1	04C1_Luderick_1.mov_5fps_000005.jpg	4	1465	390	234	236	55224	[[1691, 591, 1687, 544, 1688, 503, 1691, 473, ...
...	...	...	...	...	...	...	...	...	...	...	...
6697	6699	luderick	1	C4_Luderick_9_014000.jpg	2669	471	127	228	233	53124	[[471, 360, 491, 319, 492, 280, 487, 236, 489,...
6698	6700	luderick	1	C4_Luderick_9_014200.jpg	2670	561	79	268	234	62712	[[578, 313, 592, 247, 618, 252, 656, 230, 679,...
6699	6701	luderick	1	C4_Luderick_9_014400.jpg	2671	573	33	398	228	90744	[[578, 261, 594, 213, 601, 183, 594, 146, 581,...
6700	6702	luderick	1	C4_Luderick_9_014600.jpg	2672	650	3	476	213	101388	[[650, 216, 676, 182, 673, 133, 674, 90, 673, ...
6701	6703	luderick	1	C4_Luderick_9_014800.jpg	2673	797	6	489	178	87042	[[797, 184, 811, 150, 822, 123, 811, 70, 800, ...

Figure 54: The training dataset in Pandas DataFrame format printed in the data preprocessing notebook.

The annotation structure is not very convenient for analyzing the data nor for training models, which or each training example usually require one image with the list of bounding boxes and class labels included in it. To correct that, I **preprocessed** the original data to create new **preprocessed dataframes** where each row included one image and the list of **all bounding boxes** with the format [xmin, ymin, xmax, ymax]**, class labels and segmentation data**. For better data analysis, I included the name of the video from which each image is extracted, and the frame number of that image inside the video running at 5 fps. Table 3 describes the details of each column in the new preprocessed dataframe structure.

Column	Description
image_name	STR image file name
video_name	STR video name
frame_number	INT Frame number from its respective video
number_boxes	INT number of bounding boxes in the image
labels	ARRAY(INT) class id of each bounding box (1 = luderick, 0 reserved for background)
bounding_boxes	ARRAY(ARRAY(INT)) list of bounding box coordinates in format [[x_min, y_min, x_max, y_max], ...]
area	ARRAY(INT) area of each bounding box in pixels squared
segmentation	ARRAY(ARRAY(ARRAY(INT))) segmentation polygon coordinates in format [[[x, y, x, y, ...]], [[x, y,...]]]

Table 3: Column descriptions for each row of the new Preprocessed dataframe structure.

To preprocess the original dataframes I created a function called **preprocess_data**, which uses Pandas functions such as **group by** to aggregate the original rows by image name and count how many bounding boxes per image or how many images per video are included. It also filters out all bream instances as the dataset is considered a single class dataset for simplicity, so images with only bream appearances will not be included in the preprocessed dataframe. The function also fixes some labeling errors where negative coordinates were given to bounding box and segmentation coordinates. Figure 55 shows the results of the preprocessed dataframe from the training set. Figure 56 shows the implementation of the preprocess_data function.

	image_name	video_name	frame_number	number_boxes	labels	bounding_boxes	area	segmentation
0	04C1_Luderick_1.mov_5fps_000001.jpg	04C1_Luderick_1	1	1	[1]	[[1778, 425, 1919, 610]]	[26085]	[[1778, 515, 1789, 498, 1806, 479, 1835, 457, ...
1	04C1_Luderick_1.mov_5fps_000002.jpg	04C1_Luderick_1	2	1	[1]	[[1659, 406, 1919, 645]]	[62140]	[[1659, 509, 1675, 488, 1696, 465, 1720, 437, ...
2	04C1_Luderick_1.mov_5fps_000003.jpg	04C1_Luderick_1	3	1	[1]	[[1578, 400, 1915, 620]]	[74140]	[[1578, 530, 1585, 512, 1594, 496, 1592, 477, ...
3	04C1_Luderick_1.mov_5fps_000004.jpg	04C1_Luderick_1	4	1	[1]	[[1508, 398, 1812, 626]]	[69312]	[[1812, 624, 1795, 578, 1795, 542, 1797, 507, ...
4	04C1_Luderick_1.mov_5fps_000005.jpg	04C1_Luderick_1	5	1	[1]	[[1465, 390, 1699, 626]]	[55224]	[[1691, 591, 1687, 544, 1688, 503, 1691, 473, ...
...	...	...	...	...	...	...	...	...
2667	C4_Luderick_9_014000.jpg	C4_Luderick_9	21	1	[1]	[[471, 127, 699, 360]]	[53124]	[[471, 360, 491, 319, 492, 280, 487, 236, 489,...
2668	C4_Luderick_9_014200.jpg	C4_Luderick_9	22	1	[1]	[[561, 79, 829, 313]]	[62712]	[[578, 313, 592, 247, 618, 252, 656, 230, 679,...
2669	C4_Luderick_9_014400.jpg	C4_Luderick_9	23	1	[1]	[[573, 33, 971, 261]]	[90744]	[[578, 261, 594, 213, 601, 183, 594, 146, 581,...
2670	C4_Luderick_9_014600.jpg	C4_Luderick_9	24	1	[1]	[[650, 3, 1126, 216]]	[101388]	[[650, 216, 676, 182, 673, 133, 674, 90, 673, ...
2671	C4_Luderick_9_014800.jpg	C4_Luderick_9	25	1	[1]	[[797, 6, 1286, 184]]	[87042]	[[797, 184, 811, 150, 822, 123, 811, 70, 800, ...

Figure 55: The resulting preprocessed training dataframe.

```python
def preprocess_data(luderick_df):
    """
    Convert an original luderick DataFrame to the DataFrame format described above
    """
    # Remove the bream entries from the original dataframe
    luderick_df = luderick_df[luderick_df['category'] == 'luderick']
    # Create the dataframe
    output_df = pd.DataFrame(luderick_df['image'].unique(), columns=['image_name'])
    # get the list of unique images
    images_list = luderick_df['image'].unique()
    # get the list of box numbers
    number_boxes = luderick_df.groupby(['image'])["bbox_x"].count().to_list()
    # get the video names
    output_df['video_name'] = output_df['image_name'].apply(lambda x: get_video_name(x))
    # get the total frames in each video
    frame_numbers = output_df.groupby(['video_name'])["video_name"].count().to_list()
    # get the frame number for each frame in each video
    frame_numbers_array = np.array([np.arange(frames)+1 for frames in frame_numbers], dtype=object)
    # flatten the frame number array
    frame_numbers_array = np.hstack(frame_numbers_array)
    output_df['frame_number'] = frame_numbers_array
    # save the number of boxes
    output_df['number_boxes'] = number_boxes

    all_labels = []
    all_boxes = []
    all_areas = []
    all_segmentations = []
    for i, image in enumerate(images_list):
        # get all the rows of the image (frame), each row has info of one bounding box
        image_rows = luderick_df[luderick_df['image'] == image]
        # get all the class labels in the frame
        frame_labels = image_rows['category_id'].values
        # add frame labels to total label list
        all_labels.append(frame_labels)

        # get all the bounding boxes
        frame_boxes = image_rows[["bbox_x", "bbox_y", "bbox_w", "bbox_h"]].values
        # fix annotation errors where bbox coordinates are negative or higher than the image resolution
        frame_boxes[:, 0] = np.clip(frame_boxes[:, 0], 0, 1920).tolist()
        frame_boxes[:, 1] = np.clip(frame_boxes[:, 1], 0, 1080).tolist()
        # convert the bbox_w and bbox_h to x_max and y_max by adding those to bbox_x and bbox_y
        frame_boxes[:, 2] += frame_boxes[:, 0]
        frame_boxes[:, 3] += frame_boxes[:, 1]
        # fix annotation errors (there's at least one case of a coordinate at 1921 after adding the width)
        frame_boxes[:, 2] = np.clip(frame_boxes[:, 2], 0, 1920).tolist()
        frame_boxes[:, 3] = np.clip(frame_boxes[:, 3], 0, 1080).tolist()

        # add frame boxes to the total boxes list
        all_boxes.append(frame_boxes)
        # get all box areas of the frame
        frame_areas = image_rows['area'].values
        # add frame areas to the total areas list
        all_areas.append(frame_areas)
        # get the segmentations of that frame
        # ignore the rows that have no segmentation data (segmentatio = '[]')
        df_segmentations = image_rows[image_rows['segmentation'] !='[]']['segmentation'].values
        # clip negative values of segmentations to 0 and maximum to 1920
        frame_segmentations = [np.clip(ast.literal_eval(segm)[0], 0, 1920).tolist() for segm in df_segmentations]
        # add frame segmentations to the total segmentations list
        all_segmentations.append(frame_segmentations)

    output_df['labels'] = all_labels
    output_df['bounding_boxes'] = all_boxes
    output_df['area'] = all_areas
    output_df['segmentation'] = all_segmentations

    return output_df
```

Figure 56: The preprocess_data function, which takes one dataframe in the original annotation structure.

After obtaining the preprocessed dataframes for each split, they are saves as CSV files to be loaded and used by other notebooks that carry out training and inference on models.

3.3. Training a Faster R-CNN Model

The first model to try on this project is Faster R-CNN, one of the fastest **two-stage object detection models**. The objective is to retrain this model using **transfer learning** for the Luderick-Seagrass dataset and evaluate it during inference both in terms of speed and accuracy. As mentioned on section 2.3.1, transfer learning is the process of changing the last few layers of a model, usually called the head, with layers fitted for the number of classes of our dataset, while keeping the trainable parameters learned by previous layers, which should be able to generalize image classification and detection problems for new datasets easily. To use transfer learning on a Faster R-CNN, the model with pretrained weights on the COCO dataset will be loaded and then its Fast R-CNN detection head will be replaced with a new one with random weights and the correct output size given the number of classes to detect. The weights from the head will be the only trainable parameters in the model.

The two main options considered for training a Faster R-CNN model were the **TensorFlow Object Detection API**, and the **Torchvision** library from **Pytorch**. After experimentation with both options, the Pytorch option was chosen for being more intuitive, robust and customizable than TensorFlow in terms of object detection models. The Pytorch library offers functions to train deep learning models easily, with a system called **autograd** that offers **automatic differentiation** of functions, needed for the **backpropagation** step of neural networks. The steps to train a neural network on Pytorch are usually the following:

1. Create the custom dataset class.
2. Create data loaders for the dataset to create batches.
3. Create or load the deep learning model.
4. Implement the training and validation loop functions for one epoch.
5. Iterate over each epoch calling the training and validation functions.

The implementation of the Faster R-CNN model training on the Luderick dataset is included in the **Luderick Detection Faster R-CNN - Training** notebook. The notebook starts by loading the training and validation preprocessed luderick dataframes and setting some global variables for training, like the batch size and number of epochs. For training I used 20 epochs for all experiments. I also created some utility functions which save the best model during training and save plots with each iteration loss. Pytorch can take advantage of **Graphic Processor Units (GPU)** from graphic cards for faster training and inference, and the DEVICE variable is set to GPU if available or CPU otherwise and is used to "send" data to that device. Figure 57 shows the global variables used for one of the experiments. I used image resizing on one experiment, which allowed using larger batch sizes as it allowed to keep more images on memory at the same time but discarded it for the last experiment. The CLASSES variable is a list of the classes from the dataset, where the class at position 0 corresponds to the background class, needed for the Faster R-CNN model. The following sections describe the rest of the notebook.

```
BATCH_SIZE = 2 # increase / decrease according to GPU memeory
#MAX_SIZE = 512 # resize the image for training and transforms
NUM_EPOCHS = 20 # number of epochs to train for

DEVICE = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
# training images and XML files directory

IMAGES_DIR = "Fish_automated_identification_and_counting/luderick-seagrass"

# classes: 0 index is reserved for background
CLASSES = ['background', 'Luderick']
NUM_CLASSES = len(CLASSES)

# location to save model and plots
OUT_DIR = 'faster_rcnn_outputs'
!mkdir {OUT_DIR}
```

Figure 57: Global variables, including paths to input and outputs directories.

3.3.1. Data Augmentation

It is important to use **data augmentation** on training data to **avoid overfitting**, as mentioned on sections 2.3.4 and 2.4. I created the **get_train_transform** function which transforms images with random flips, 90-degree rotations and blur using the albumentations library, which applies the images transformations to the ground truth bounding boxes, so they are kept in sync. The function also converts the image into the tensor class, which shape is (number of color channels, image height, image width) and is needed for pytorch models. I also create the get_valid_transform, which is used for the validation set and only transforms the image into tensors. Figure 58 shows the code for the training and validation transforms, and Figure 59 displays an image after applying data augmentation.

```python
# define the training transforms for Data Augmentation
def get_train_transform():
    return A.Compose([
        A.Flip(0.5),
        A.RandomRotate90(p=0.2),
        A.MedianBlur(blur_limit=3, p=0.1),
        A.Blur(blur_limit=3, p=0.1),
        ToTensorV2(p=1.0),
    ], bbox_params={
        'format': 'pascal_voc',
        'label_fields': ['labels']
    })

# define the validation transforms
def get_valid_transform():
    return A.Compose([
        ToTensorV2(p=1.0),
    ], bbox_params={
        'format': 'pascal_voc',
        'label_fields': ['labels']
    })
```

Figure 58: Training a validation transform functions.

Figure 59: A transformed image with data augmentation. The image has been flipped horizontally. Bounding box coordinates are modified accordingly to the new image.

3.3.2. The Luderick Dataset Class

The next step is the implementation of the luderick dataset class. Dataset classes in pytorch must extend the torch.utils.data.Dataset class and implement the following functions:

- **__init__:** Initializes the dataset with the given parameters
- **__getitem__:** Given a unique id. fetches and returns one item from the dataset with the correct format for training the model.
- **__len__:** Returns the number of items in the dataset, in this case the number of images.

Figure 60 shows the implementation of the **LuderickDataset** class. It is instantiated with a preprocessed luderick dataframe, the relative path to the images and the list of classes, max_size for image resizing and tranforms (like get_train_transform) as input parameters for the **__init__** function.

```
class LuderickDataset(torch.utils.data.Dataset):
    def __init__(self, luderick_df, images_dir, classes, max_size=None, transforms=None):
        self.luderick_df = luderick_df
        self.images_dir = images_dir
        self.max_size = max_size
        self.classes = classes
        self.transforms = transforms

    def __getitem__(self, idx):
        # load images and masks
        row = self.luderick_df.iloc[idx]
        image_path = os.path.join(self.images_dir, row['image_name'])

        #read the image
        image = cv2.imread(image_path)

        # convert BGR to RGB color format
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB).astype(np.float32)

        resize_ratio = 1
        if self.max_size:
            height, width, _ = image.shape
            max_dim = max(height, width)
            resize_ratio = self.max_size / max_dim
            image = cv2.resize(image, (int(width*resize_ratio), int(height*resize_ratio)))

        image /= 255.0

        num_boxes = row['number_boxes']
        boxes = row['bounding_boxes']
        labels = row['labels']
        area = row['area']

        # convert everything into a torch.Tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        # there is only one class
        labels = torch.as_tensor(labels, dtype=torch.int64)


        image_id = torch.tensor([idx])
        area = torch.as_tensor(area, dtype=torch.int64)
        # suppose all instances are not crowd
        iscrowd = torch.zeros((num_boxes,), dtype=torch.int64)

        target = {}
        # resize bounding box coordinates according to the image resizing
        boxes = boxes*resize_ratio
        target["boxes"] = boxes
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = area*resize_ratio**2 # ratio*h*ratio*w = h*w*ratio^2
        target["iscrowd"] = iscrowd

        # apply the image transforms
        if self.transforms:
            sample = self.transforms(image = image,
                                     bboxes = target['boxes'],
                                     labels = labels)
            image = sample['image']
            target['boxes'] = torch.Tensor(sample['bboxes'])

        return image, target

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

Figure 60: The LuderickDataset class.

The __getitem__ function retrieves an image and its annotation data in the correct format for the Faster R-CNN model implemented by Pytorch. This format should include the image in tensor format and a dictionary, called target, which includes the following keys with values in the tensor format:

- **"boxes":** A list of bounding box coordinates in the form of [xmin, ymin, xmax, ymax]
- **"labels":** A list with the class ids of each bounding box.
- **"image_id":** A unique numerical identifier for the image.
- **"area":** A list with the area of each bounding box.
- **"iscrowd"** A list of binary values that indicate if there are lots of instances of the same object in the same detection. Some object detection models use this information. It is always set to 0.
- **"masks"** (optional): In the case of using an instance segmentation model such as Mask R-CNN, masks should include a list of binary bitmaps with the same size as the image with values of 1 on the coordinates where there are pixels pertaining to that object and 0 elsewhere. This key is not used for Faster R-CNN.

As the preprocessed dataframe includes a list of boxes with the same format, a list of labels and areas for each image, it is easy to generate the target dictionary. **__getitem__** simply uses the inputted id on the class saved preprocessed luderick dataframe, using the pandas function iloc, which retrieves the ieth row. In the case of using image resizing, a **resize_ratio** is computed as the ratio at which the image has been resized. Bounding box coordinates are multiplied by it to be adjusted to the new size, as well as the areas which are multiplied by the squared value of resize_ratio. The final step in the __getitem__ function is to use the transform functions used for **data augmentation** in the case of the training set, and just converting the image into tensor in the validation set.

The **__len__** function simply returns the number of rows in the preprocessed luderick dataframe.

Figure 61 shows how the LuderickDataset class is instantiated to create the train_dataset and valid_dataset objects.

```
# create the dataset instances
train_dataset = LuderickDataset(train_df, IMAGES_DIR, CLASSES, transforms=get_train_transform())
valid_dataset = LuderickDataset(validation_df, IMAGES_DIR, CLASSES, transforms=get_valid_transform())
```

Figure 61: Creating the dataset objects.

Figure 62 shows an example output of the "target" dictionary with annotation data for the transformed image from Figure 59.

```
# visualize the bounding boxes data and how it got resized accordingly
target
```

```
{'boxes': tensor([[ 505.0000,    5.0000, 1427.0000,  250.0000],
        [1356.0000,  294.0000, 1560.0000,  584.0000],
        [1431.0000,  330.0000, 1911.0000,  993.0000],
        [ 688.0000,  908.0000,  915.0000, 1062.0000],
        [ 487.0000,  665.0001,  864.0000,  903.0000],
        [ 176.0000,  761.0001,  299.0000,  836.0000],
        [   0.0000,  877.0000,  110.0000,  995.0000]]),
 'labels': tensor([1, 1, 1, 1, 1, 1, 1]),
 'image_id': tensor([90]),
 'area': tensor([225890,  59160, 318240,  34958,  89726,   9225,  12980]),
 'iscrowd': tensor([0, 0, 0, 0, 0, 0, 0])}
```

Figure 62: The target dictionary with annotation data from the LuderickDataset class.

3.3.3. Data Loaders

Instances of the **DataLoader** class from pytorch are used for the training and validation datasets, which group data into **batches** with a given **batch size** and can **randomize the data order** with the **shuffle** parameter, which is useful to avoid overfitting on video data, as each sequence would train the model with a list of similar images if they were fed by order. Figure 63 shows the code for the train and validation dataset loaders.

```
train_loader = DataLoader(train_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=True, # shuffle the dataset to avoid overfitting on long sequences
                          num_workers=0,
                          collate_fn=collate_fn)

valid_loader = DataLoader(valid_dataset,
                          batch_size=BATCH_SIZE,
                          shuffle=False,
                          num_workers=0,
                          collate_fn=collate_fn)
print(f"Number of training samples: {len(train_dataset)}")
print(f"Number of validation samples: {len(valid_dataset)}\n")
```

```
Number of training samples: 2672
Number of validation samples: 824
```

Figure 63: Train and validation data loaders.

3.3.4. Creating the Model

Pytorch offers an implementation of the Faster R-CNN model which can be loaded from the library with pretrained weights on the **COCO dataset** [52], which includes images with detection labels of 80 different classes and is a benchmark of object detection models. As explained in section 2.2.3, Faster R-CNN consists of an **RPN** that generates **ROIs** and a **Fast R-CNN detection head** that uses those ROIs for a final bounding box regression and object classification. The loaded model from Pytorch has the **Resnet50-FPN backbone**, which is the default for Faster R-CNN since Mask R-CNN. To apply **transfer learning,** the Fast R-CNN head is replaced with a new instance with a different number of classes, 2 in the case of the luderick dataset, including the background and luderick classes. Figure 64 shows the create_model function, which returns an instance of the Faster R-CNN model with a Fast R-CNN detection head that predicts the given number of classes.

```
def create_model(num_classes):

    # load Faster RCNN pre-trained model
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

    # get the number of input features
    in_features = model.roi_heads.box_predictor.cls_score.in_features

    # define a new head for the detector with the required number of classes
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

Figure 64: The create_model function returns an instance of the Faster R-CNN model.

3.3.5. The Training and Validation Functions

The model is to be trained for **20 epochs**. One epoch means that all the images from the training dataset have been fed to the model once. To train the model for one epoch, I created the train function, which iterates over the training dataset batch by batch using the train data loader, which returns one batch from the dataset at each iteration, until all batches have been used. I use the tqdm library to iterate over batches, which creates a visual representation of a progress bar updating in real time while the process is running.

The Pytorch **automatic differentiation (autograd)** records the partial derivatives of all the trainable parameters for each function they go through, including convolution layers, activation functions and loss functions. By using the model with a batch as input, those partial derivatives can be used to update weights using an **optimizer**, like **Stochastic Gradient Descent (SGD)**, with a learning rate and momentum, which adjust the speed at which the weights are updated, and weight decay, which is a **regularization** method that adds a penalty to weight values, resulting in smaller weights.

In one iteration of the **train function**, images and target dictionaries are loaded from **one batch**, and used as input to the Faster R-CNN. Pytorch models can be used in training or inference mode, when used on training mode they return the loss value obtained from the predictions. The Faster R-CNN model returns the losses for the batch inputs, **including objectness and anchor box regression losses from the RPN**, and the **final detection boxes regression and object classification losses** from the Fast R-CNN head. After retrieving those losses, the train function adds them to a single total loss value, which is used by **autograd** by calling the **backward** function on the total loss variable to compute the partial derivatives of the weights and using the **step** function from the **optimizer** to update the weights. For each new batch at the start of each iteration, partial derivatives are resetted by calling the **zero_grad** function from the optimizer as one **batch should not affect the next**. The loss from each iteration is saved into a global variable for plotting. Figure 65 shows the implementation of the train function.

```
# function for running training iterations
def train(train_data_loader, model, device):
    print('Training')
    global train_itr
    global train_loss_list

    # initialize tqdm progress bar
    prog_bar = tqdm(train_data_loader, total=len(train_data_loader))

    for i, data in enumerate(prog_bar):
        # reset the gradient accummulation from the previous batch
        optimizer.zero_grad()
        images, targets = data

        # use the device from the configuration (gpu if afailable)
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        # the model returns the losses in training mode, including the
        # classification loss, rpn and final bounding box regression loss
        # and rpn objectness loss
        loss_dict = model(images, targets)

        # add all the losses to get the total loss
        losses = sum(loss for loss in loss_dict.values())
        loss_value = losses.item()
        train_loss_list.append(loss_value)

        train_loss_hist.send(loss_value)

        # backpropagate the total loss
        losses.backward()
        # update the parameters according to their gradients
        optimizer.step()

        train_itr += 1

        # update the loss value beside the progress bar for each iteration
        prog_bar.set_description(desc=f"Loss: {loss_value:.4f}")
    return train_loss_list
```

Figure 65: The train function, which trains the model for one epoch.

After each epoch training by calling the train function, the model must be validated to see how well it performs on validation data. The simple way is to simply compute the loss, called validation loss, which would demonstrate that the model is generalizing what it is learning on the training set. When the validation loss stops decreasing while the training loss still does, it means the model starts **overfitting.** Another way of evaluating the model at each epoch would be to compute the **mAP** as seen in section 2.4, but I left the mAP evaluation for once the model is fully trained.

The **validate function** is the validation version of the train function, which iterates over all batches of the given data loader and feeds each batch to the model, retrieving losses, adding them and saving them to a validation loss list. As the model should not be trained on validation data, the model is called inside a nested pytorch function called **no_grad**, called as **with torch.no_grad():** which disables the computation of derivatives for that call, and the losses.backward() and optimizer.step() function are not called. Figure 66 shows the implementation of the validate function.

```
# function for running validation iterations
def validate(valid_data_loader, model, device):
    print("Validating")
    global val_itr
    global val_loss_list

    # initialize tqdm progress bar
    prog_bar = tqdm(valid_data_loader, total=len(valid_data_loader))

    for i, data in enumerate(prog_bar):
        images, targets = data

        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        with torch.no_grad():
            loss_dict = model(images, targets)

        losses = sum(loss for loss in loss_dict.values())
        loss_value = losses.item()
        val_loss_list.append(loss_value)

        val_loss_hist.send(loss_value)

        val_itr += 1

        # update the loss value beside the progress bar for each iteration
        prog_bar.set_description(desc=f"Loss: {loss_value:.4f}")
    return val_loss_list
```

Figure 66: The validate function, which saves the validation loss for each batch.

3.3.6. The Main Code

With all the functions implemented, the **main training loop** can be created. The main code, shown in Figure 67, starts by creating the Faster R-CNN model instance with the create_model function and the SGD optimizer with 0.001 learning rate, 0.9 momentum and 0.0005 weight decay, and initializes variables such as the train a validation loss lists. An average class is used to get the average losses for each epoch instead of each batch.

A for loop is called with iterations equal to the number of epochs to train, in this case 20, and calls the train and validate function at each iteration. After the validation, the model is saved and if the validation loss is the lowest from all the iterations, a copy called “best_model…” is saved as well. Saving the best model is important as the model could start overfitting, making the last saved model unsuitable for data outside the training dataset.

```python
# initialize the model and move to the computation device
model = create_model(num_classes=NUM_CLASSES)
model = model.to(DEVICE)
# get the mdel parameters
params = [p for p in model.parameters() if p.requires_grad]
# define the optimizer
optimizer = torch.optim.SGD(params, lr=0.001, momentum=0.9, weight_decay=0.0005)

# initialize the Averager class
train_loss_hist = Averager()
val_loss_hist = Averager()
train_itr = 1
val_itr = 1
# train and validation loss lists to store loss values of all...
# ... iterations till end and prot graphs for all iterations
train_loss_list = []
val_loss_list = []

# mean losses for train and validation at every epoch
epoch_train_losses = []
epoch_validation_losses = []

# name to save the trained model with
MODEL_NAME = "MODEL_1920_aug_1"

# initialize SaveBestModel class
save_best_model = SaveBestModel(model_name=MODEL_NAME)

# start the training epochs
for epoch in range(NUM_EPOCHS):
    print(f"\nEPOCH {epoch+1} of {NUM_EPOCHS}")

    # reset the training and valitation loss histories for the current epoch
    train_loss_hist.reset()
    val_loss_hist.reset()

    # create two subplots, one for each, training and validation
    figure_1, train_ax = plt.subplots()
    figure_2, valid_ax = plt.subplots()

    # start timer and carry out training and validation
    start = time.time()
    train_loss = train(train_loader, model, DEVICE)
    val_loss = validate(valid_loader, model, DEVICE)
    print(f"Epoch #{epoch+1} train_loss: {train_loss_hist.value:.3f}")
    print(f"Epoch #{epoch+1} validation_loss: {val_loss_hist.value:.3f}")
    end = time.time()
    print(f"Took {(end - start):.3f} seconds for epoch {epoch+1}")

    # save the mean train and validation losses of this epoch
    epoch_train_losses.append(train_loss_hist.value)
    epoch_validation_losses.append(val_loss_hist.value)

    # save the best model until now if we have the minimum val loss this epoch
    save_best_model(val_loss_hist.value, epoch, model, optimizer)

    #save the current epoch model
    save_model(epoch, model, optimizer, model_name=MODEL_NAME)

    #save the loss plot
    save_loss_plot(OUT_DIR, train_loss, val_loss, model_name=MODEL_NAME)
```

Figure 67: The main code.

As Pytorch takes advantage of **GPU computing** with the **CUDA** library which **parallelizes** the computation, training time will change depending on the graphic card and other components. If no GPU is available, online python environments such as Google Colab can be used, although they are limited for free users. I have evaluated model performances in terms of inference time in my own computer, which uses **NVIDIA GeForce GTX 960 graphics card** and has **8GB RAM**. However, I used another computer equipped with an **NVIDIA GeForce RTX 2080 Ti** and **16GB RAM** for faster training. Training for 20 epochs lasted more than 3 hours where the training of each epoch lasted around 10 minutes.

Figure 68 shows a **plot** of the **mean train a validation losses at each epoch**, also called **Loss Curves**. The loss curves show how the model learned to detect ludericks on the training set over 20 epochs. However, it did not struggle as much on the validation set, where the loss started low and decreased slower than the training loss. The best validation loss was achieved after training for 17 epochs with a validation loss value of 0.1367. The model was saved on epoch 17 to be used for evaluation. Another model with image resizing was trained as well, with the 16th epoch achieving the lowest validation loss, and was saved as a separate model for evaluation**.**

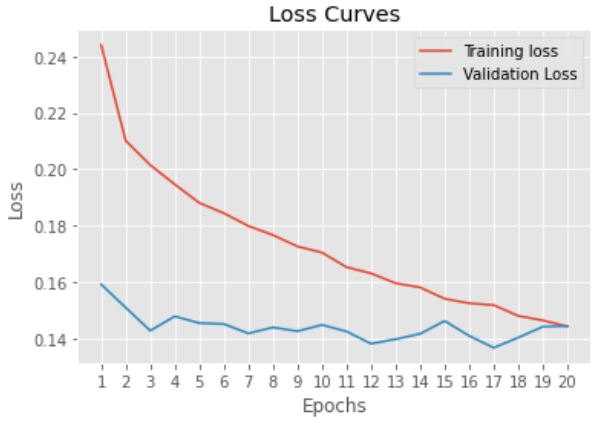

Figure 68: A plot of the loss curves for 20 epochs.

3.4. Using and Evaluating the Trained Faster R-CNN Model

After training the model, it is time to use and evaluate it on data outside the training set. I created the **Luderick Detection Faster R-CNN – Inference** Jupyter Notebook. It starts by loading the preprocessed luderick dataframe and setting the proper global variables like in the training notebook.

3.4.1. Loading the Trained Model

The Faster R-CNN model is loaded like in the training notebook, but the saved parameters trained in the previous notebook are loaded to that model as well. The load_model function returns that model with the trained parameters, also called weights, from the given path to the saved model.

A **min_size** parameter can also be used on the function to load a Faster R-CNN model that resizes the input images so that their minimum size is equal to that parameter. A smaller min_size should make the model faster at inference time but also make it potentially less precise. The model is loaded in inference mode using the eval function, so that it returns detections instead of losses and the parameters are no longer trained. Figure 69 shows the implementation of the load_model function for the inference jupyter notebook.

```
def load_model(num_classes, saved_model_dir, device, min_size=800):

    # load Faster RCNN pre-trained model
    # min_size = minimum size of the image to be rescaled before feeding it to the backbone (default = 800)
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True, min_size=min_size)

    # get the number of input features
    in_features = model.roi_heads.box_predictor.cls_score.in_features

    # define a new head for the detector with the required number of classes
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
    checkpoint = torch.load(saved_model_dir, map_location=device)
    model.load_state_dict(checkpoint['model_state_dict'])
    model.to(device).eval()

    return model
```

Figure 69: The load_model function.

3.4.2. Predict Bounding Boxes

To detect ludericks in an image it should be converted into the tensor and use it as input to the model. The model returns a dictionary with a list of bounding boxes, class labels and confidence scores in tensor format.

I created a **predict** function that filters out detections depending on a given confidence threshold, and returns boxes, labels and scores in the array format, as well as the inference time for that detection. With the inference time, the fps at that given moment can be computed by dividing 1 by that number. Figure 70 shows the implementation of the predict function.

```python
def predict(image, model, device, detection_threshold):
    """
    Return, as arrays, the detection bounding boxes, labels,
    pred_scores and inference time in an PIL Image (RGB) using a
    given model above a given detection_threshold.
    """
    # transform the image into a tensor
    image = transform(image).to(device)
    image = image.unsqueeze(0) # add a batch dimension
    start_time = time.time()
    with torch.no_grad():
        outputs = model(image) # get the predictions on the image
    end_time = time.time()
    inference_time = end_time - start_time

    # get the score for all the predicted objects
    pred_scores = outputs[0]['scores'].detach().cpu().numpy()
    high_score_indices = pred_scores >= detection_threshold

    pred_scores_high = pred_scores[high_score_indices]

    # get all the predicted labels
    pred_labels = outputs[0]['labels'].cpu().numpy()
    pred_labels_high = pred_labels[high_score_indices]

    # get all the predicted bounding boxes
    pred_bboxes = outputs[0]['boxes'].detach().cpu().numpy()
    # get the boxes above the threshold score
    pred_bboxes_high = pred_bboxes[high_score_indices].astype(np.int32)

    return pred_bboxes_high, pred_labels_high, pred_scores_high, inference_time
```

Figure 70: The predict function.

Once the predicted bounding box coordinates are obtained, they can be painted on the original image to get a visual representation of the model results. The class name and confidence score should be painted on their respective box to get a better idea of the detection. To do that, the draw_boxes function was implemented. Given an image, a list of bounding box coordinates, class labels and prediction scores, it will draw each detection box alongside the class name and score. **OpenCV** [42] is a powerful library for image processing and has been used for this project. The OpenCV **rectangle** function will draw a rectangle with the given coordinates in the same format as the ones from our bounding boxes. Figure 71 shows the implementation of the draw_boxes function.

```python
def draw_boxes(boxes, classes, labels, image, pred_scores, color=(255, 0, 0)):
    """
    Returns an image array that contains the predicted bounding boxes, labels and scores
    """
    # read the image with OpenCV
    image = np.asarray(image)
    for i, box in enumerate(boxes):
        cv2.rectangle(image,
                      (int(box[0]), int(box[1])), (int(box[2]), int(box[3])),
                      color, 2)
        # showing probability scores
        bbox_text = f"{classes[labels[i]]} {str(round(pred_scores[i]*100, 2))}"
        cv2.putText(image, bbox_text, (int(box[0]), int(box[1] - 5)),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2,
                    lineType=cv2.LINE_AA)

    return image
```

Figure 71: The draw_boxes function.

For faster inference time, I loaded the trained Faster R-CNN model with the min_size parameter set at 512. After that, an image from the validation set is loaded and the prediction function is used to get the predicted boxes. Finally, those predictions are drawn using the draw_boxes function. Figure 72 shows the code cell carrying out that with its output, including the inference time and the drawn detected boxes.

```
# use a faster model with min_size = 512 and check its inference time
model_fast = load_model(NUM_CLASSES, model_path, DEVICE, min_size=512)

image = Image.open(f"{IMAGES_DIR}/C1_Luderick_14.mov_5fps_000013.jpg").convert('RGB')
boxes, labels, pred_scores, inf_time = predict(image, model_fast, DEVICE, 0.8)
image_boxes = draw_boxes(boxes, CLASSES, labels, image, pred_scores)
print(f'Inference time = {inf_time} seconds')
view_image(image_boxes)
torch.cuda.empty_cache()
```

```
Inference time = 0.12499570846557617 seconds
```

Figure 72: Code cell and output using inference on one image.

Now that the proper functions have been created, the whole validation dataset can be used on the model by looping over the rows of the preprocessed validation dataframe and using the image names alongside the path where they are stored. I implemented the **preditct_whole_dataset** function, which given a preprocessed luderick dataframe and the trained model, will apply inference on all the images from the dataset and save them in the given folder. The function also prints the average **fps** by dividing the sum of the inference time for each of the detections by the total number of images.

Processing the full validation dataset with the trained **Faster R-CNN** model on my computer, equipped with an **NVIDIA Geforce GTX 960 graphics card,** resulted in an average of **7.6 fps.** Figure 73 shows the implementation of the predict_whole_dataset function. Figure 74 displays some of the resulting images from that function used on the validation dataset.

```python
def predict_whole_dataset(luderick_df, model, images_dir, output_dir, score_threshold=0.8):
    """
    Carry out object detection on all the images from a preprocessed luderick dataset
    and save them with bounding boxes, labels and prediction scores printed at the desired folder
    """
    image_names = luderick_df['image_name'].values
    total_fps = 0
    frame_count = 0

    for i, image_name in enumerate(image_names):
            # load the image
            image = Image.open(f"{images_dir}/{image_name}").convert('RGB')
            # use the model to get the predictions
            boxes, labels, pred_scores, inf_time = predict(image, model, DEVICE, score_threshold)
            # draw the image boxes
            image_boxes = draw_boxes(boxes, CLASSES, labels, image, pred_scores)
            # get the current fps
            fps = 1 / inf_time
            # increase total fps
            total_fps += fps
            # increment frame count
            frame_count += 1
            image_boxes = cv2.cvtColor(image_boxes, cv2.COLOR_BGR2RGB).astype(np.float32)
            cv2.imwrite(f"{output_dir}/{image_name}", image_boxes)
            print(f"Detection on image {i+1} finished...")
            print('-'*50)

    print('Dataset detection completed.')
    # calculate and print the average FPS
    avg_fps = total_fps / frame_count
    print(f"Average FPS: {avg_fps:.3f}")
```

Figure 73: The predict_whole_dataset function.

Figure 74: Some resulting images from the predict_whole_dataset function.

3.4.3. Real-Time Inference on Videos

The main goal of this thesis is about real-time detection on videos. Although the average fps obtained from the predict_whole_dataset already answers the question of real-time speed, it would be a waste to not try the model on real videos. The **detect_video** function has been created to use a real video on the model and output another with the drawn bounding boxes, classes, confidence scores and the fps at each frame. The Luderick-Seagrass includes videos from which the images have been extracted, and those can be used with this function.

The function uses the OpenCV library to open the video and loop over each frame, using each as an image and using the predict function to get detections. The draw_boxes function is called and the final image is added to an output video that is saved at 30fps. I chose the video with more simultaneous appearances of ludericks to test the model and it ran at an average of **7.5 fps**. The time to draw boxes is not considered. The resulting video was delivered alongside this document. Figure 75 shows the implementation of the detect_video function.

```
def detect_video(video_path, model, device, output_dir='.', detection_threshold=0.8, save_video=True):
    """
    Carries out detection on a video frame by frame, showing it in real time in an opencv window and showing the framerate.
    It also saves the video at 30 fps as a .mp4
    """
    cap = cv2.VideoCapture(video_path)

    if (cap.isOpened() == False):
        print('Error while trying to read video. Please check path again')

    # get the frame width and height
    frame_width = int(cap.get(3))
    frame_height = int(cap.get(4))

    save_name = f"{video_path.split('/')[-1].split('.')[0]}"
    # define codec and create VideoWriter object
    if save_video:
        out = cv2.VideoWriter(f"{output_dir}/{save_name}.mp4",
                              cv2.VideoWriter_fourcc(*'mp4v'), 30,
                              (frame_width, frame_height))

    frame_count = 0 # to count total frames
    total_fps = 0 # to get the final frames per second

    # Reading Frames and Detecting Objects
    # read until end of video
    while(cap.isOpened()):
        # capture each frame of the video
        ret, frame = cap.read()
        # while the video isn't finished...
        if ret:
            with torch.no_grad():
                # get predictions for the current frame
                boxes, labels, pred_scores, inf_time = predict(frame, model, device, detection_threshold)

            image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB).astype(np.float32)
            # draw boxes and show current frame on screen
            image = draw_boxes(boxes, CLASSES, labels, frame, pred_scores, color=(0, 0, 255))
            # get the inference time fps
            fps = 1 / inf_time
            # add fps tot total fps
            total_fps += fps
            # increment frame count
            frame_count += 1
            image = cv2.putText(image, f"{fps:.1f} FPS",
                        (image.shape[1]-200, 25),
                        cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255),
                        2, lineType=cv2.LINE_AA)

            # Display frames in a window
            # press 'q' to exit
            wait_time = max(1, int(fps/4))
            cv2.imshow('image', image)
            if save_video:
                out.write(image)
            if cv2.waitKey(wait_time) & 0xFF == ord('q'):
                break

        else:
            break

    # release VideoCapture()
    cap.release()
    # close all frames and video windows
    cv2.destroyAllWindows()

    # calculate and print the average fps
    avg_fps = total_fps / frame_count
    print(f"Average FPS: {avg_fps:.3f}")
```

Figure 75: The detect_video function.

3.4.4. Evaluating the Model with Mean Average Precision

The final step in the inference notebook is to evaluate a model using the **mAP** as explained in section 2.4. There are already implemented libraries and functions that can compute the mAP of a given model, but there are some drawbacks. They often require downloading third party code and repurposing it for our problem, as well as having to convert data in the correct format for those functions.

It would be interesting to be able to see the whole process, including the **precision-recall curves** of the model. An example of such curves was provided in Figure 47. That is why I decided to implement the mAP computation from scratch and use the same method for both Faster R-CNN and YOLOv5.

To compute the mAP, as was shown in Figure 48, we need a dataframe where each row is one detected bounding box and with the following columns:

- **Confidence Score:** The probability score for that box given by the model prediction
- **Ground Truth Box:** A **unique ID** for the real, ground truth box to which the prediction is associated. That box will be the ground truth box with the highest IoU with the predicted box.
- **IoU**: The Intersection over Union of the detected box with its ground truth box.
- **TP/FP:** Whether that detected box is a True Positive or a False Positive
- **Precision:** Total Precision from the top of the table to that row.
- **Recall:** Total Recall at that row.

Figure 48 shows two columns that were used as counters of TP and FP boxes, but simple variables outside the table can be used instead. The table must be **sorted from highest to lowest confidence score**.

The first step to obtain the dataframe is to implement the **IOU function**, which given two bounding boxes computes the area of the intersection divided by the area of their union, as detailed in section 2.4.

To get the intersection area, the right-most left edge between the two bounding boxes will be the left edge of the intersection. The left edge from the intersection is thus computing by getting the maximum value of the first coordinate (xmin) between the two boxes. The left-most right edge between the two boxes will be the right edge of the intersection. This is computed by getting the minimum third coordinate (xmax) between the two boxes.

The top edge of the intersection will be the lowest top edge from the two boxes, which is the maximum second coordinate (ymin) between the two boxes. The bottom edge will be the top-most lower edge between the two boxes, which is the minimum of the fourth coordinate (ymax) between the two boxes.

Once the left, right, top and bottom edge of the intersection box are found, as they are coordinates on the image, the area will be computed by simply subtracting the left edge from the right, subtract the bottom edge from the top, and multiplying both results. If the two boxes do not intersect, at least one of the resulting subtractions will be negative. In that case, the returned area must be 0.

To find the area of the union, the area of each detected box is computed the same way. Then, the areas of the two boxes are added and the intersection area is subtracted, as the area corresponding to the intersection is counted twice when adding the two boxes together.

Finally, the intersection area is divided by the union area, giving the IoU as result. Figure 76 shows the implementation of the IOU function.

```python
def IOU(box_1, box_2):
    """
    Computes the Intersection Over Union (IOU) between two bounding boxes
    formatted as [x_min, y_min, x_max, y_max]
    """
    # get the right-most left edge between the two bounding boxes
    max_left = max(box_1[0], box_2[0])
    # get the left-most right edge between the two bounding boxes
    min_right = min(box_1[2], box_2[2])
    # compute the intersection width
    intersection_width = min_right - max_left

    # get the lowest top edge (y is bigger as it goes lower)
    max_top = max(box_1[1], box_2[1])
    # get the highest bottom edge
    min_bottom = min(box_1[3], box_2[3])
    # compute the intersection height
    intersection_height = min_bottom - max_top

    # if the boxes don't intersect, intersection_width or height will be negative
    intersection_area = max(0, intersection_width) * max(0, intersection_height)
    box_1_area = (box_1[2] - box_1[0]) * (box_1[3] - box_1[1])
    box_2_area = (box_2[2] - box_2[0]) * (box_2[3] - box_2[1])
    # get the union area by adding the boxes areas and substracting their intersection
    union_area = box_1_area + box_2_area - intersection_area
    # get the final iou
    iou = intersection_area / union_area

    return iou
```

Figure 76: The IOU function.

Next, the dataframe with the detected boxes information should be created. As the modern **mAP@0.5:0.05:0.95** requires assigning TP or FP labels separately to compute the AP over a range of IOU scores before computing its mean, the base dataframe will only include the columns with the confidence score, IoU and ground truth box id. To geta unique id for each ground truth box of the image, I added a column with the name of the image where the bounding box was detected, and the ground truth boxes from that image have an id starting at 0. The unique ID will be created by adding that id at the end of the image name. Detected boxes that do not match with any true box are given a ground truth box ID of -1.

To generate that base dataframe, I created the **get_boxes_evaluation_dataframe** function, that iterates over the given dataset and predicts detections in a similar way to the **predict_whole_dataset** function. After getting each prediction, the function iterates over the total number of classes and sends detected boxes and ground truth boxes of that image pertaining each class to the **get_image_box_evaluation_dataframe**, that will iterate over those boxes to match them by finding the max IoU between a true and a detected box. Figure 77 and Figure 78 show the implementation of both functions. Figure 79 shows the resulting dataframe using the trained model and validation dataset. The dataframe is no sorted from maximum to minimum confidence score yet.

```python
def get_image_box_evaluation_dataframe(pred_boxes, true_boxes, true_labels, pred_label, pred_scores):
    """
    Returns the box evaluation dataframe of a single image and class
    """
    ious_gt_box_index = np.array([-1] * len(pred_boxes))
    ious = np.zeros(len(pred_boxes))

    for i, pred_box in enumerate(pred_boxes):
        max_iou = 0
        max_iou_box_index = -1
        for j, true_box in enumerate(true_boxes):
            iou = IOU(pred_box, true_box)
            if iou > max_iou and true_labels[j] == pred_label:
                max_iou = iou
                # save the index of the true box with highest iou
                max_iou_box_index = j

        ious[i] = max_iou
        ious_gt_box_index[i] = max_iou_box_index

    output_df = pd.DataFrame({'confidence': pred_scores,'IOU': ious,'GT_Box': ious_gt_box_index})

    return output_df
```

Figure 77: The get_image_box_evaluation_dataframe function.

```python
def get_boxes_evaluation_dataframe(model, device, luderick_df, images_dir, num_classes, detection_threshold):
    """
    Returns the box evaluation dataframe
    """
    image_names = luderick_df['image_name'].values
    num_images = len(image_names)
    all_boxes_eval_df = pd.DataFrame()
    for i, image_name in enumerate(image_names):
        if i % int(num_images/4) == 0:
            print(f'{(i/num_images)*100}% of images processed')
        # load the image
        image = Image.open(f"{images_dir}/{image_name}").convert('RGB')
        # use the model to get the predictions
        pred_boxes, pred_labels, pred_scores, inf_time = predict(image, model, device, detection_threshold)
        true_boxes = luderick_df.iloc[i]['bounding_boxes']
        true_labels = luderick_df.iloc[i]['labels']
        for label in range(num_classes):
            # check if each box is a true positive or false positive
            class_indices = np.array(pred_labels) == label
            # use all the boxes from class i as input to get an array of their ious
            # and a boolean list indicating if the predicted class is correct
            image_boxes_eval_df = get_image_box_evaluation_dataframe(pred_boxes[class_indices],
                                        true_boxes, true_labels, label, pred_scores[class_indices])
            image_boxes_eval_df['image_name'] = [image_name]*len(pred_boxes[class_indices])
            image_boxes_eval_df['label_id'] = [label]*len(pred_boxes[class_indices])
            image_boxes_eval_df = image_boxes_eval_df[['image_name', 'label_id', 'confidence', 'GT_Box', 'IOU']]
            all_boxes_eval_df = pd.concat([all_boxes_eval_df, image_boxes_eval_df], ignore_index=True)

    return all_boxes_eval_df
```

Figure 78: The get_boxes_evaluation_dataframe function.

	image_name	label_id	confidence	GT_Box	IOU
0	04C2_Luderick_16_000400.jpg	1.0	0.993775	0.0	0.948097
1	04C2_Luderick_16_000400.jpg	1.0	0.220970	-1.0	0.000000
2	04C2_Luderick_16_000400.jpg	1.0	0.122255	-1.0	0.000000
3	04C2_Luderick_16_000600.jpg	1.0	0.996549	0.0	0.885086
4	04C2_Luderick_16_000600.jpg	1.0	0.080668	-1.0	0.000000
...	...	...	...	...	...
2107	C1_Luderick_14.mov_5fps_000016.jpg	1.0	0.998660	2.0	0.891118
2108	C1_Luderick_14.mov_5fps_000016.jpg	1.0	0.998467	1.0	0.854480
2109	C1_Luderick_14.mov_5fps_000016.jpg	1.0	0.997568	-1.0	0.000000
2110	C1_Luderick_14.mov_5fps_000016.jpg	1.0	0.996231	0.0	0.838008
2111	brim.mov_5fps_000002.jpg	1.0	0.990456	0.0	0.683378

2112 rows × 5 columns

Figure 79: The resulting dataframe from the get_boxes_evaluation_dataframe function.

The next step is to assign a TP or FP label to each detected box depending on the give IoU threshold. If a detected box has higher IoU with its ground truth box than the threshold, it will be considered a TP if there it is the first detected box considered a TP for that ground truth box. The dataframe will be ordered from highest to lowest confidence score, so detected boxes with higher score will have priority. I created the **assign_tp_fp** function, which and sorts the dataframe from highest to lowest confidence score and ads the TP column to the dataframe with value 1 if the detected box is a TP and 0 if it is a FP. Figure 80 shows the implementation of the assign_tp_fp function.

```python
def assign_tp_fp(boxes_eval_df, iou_thresh):
    """
    Check if every predicted box is a true positive (TP) or a false positive (FP) for a given IOU threshold
    and set the TP column to 1 or 0 accordingly
    """
    boxes_eval_df = boxes_eval_df.sort_values(by='confidence', ascending=False)
    boxes_eval_df = boxes_eval_df.reset_index(drop=True)
    gt_boxes_checked = []
    for index, row in boxes_eval_df.iterrows():
        gt_box_id = f"{row['image_name']}_box_{row['GT_Box']}"
        # assignt TP if the pred_box intersects with a GT box and has higher IOU than the threshold
        if row['GT_Box'] != -1 \
            and row['IOU'] > iou_thresh \
                and not(gt_box_id in gt_boxes_checked): # dont mark as TP if another pred_box is TP on the same GT box
                    boxes_eval_df.at[index, 'TP'] = 1
                    gt_boxes_checked.append(gt_box_id)

        else:
            ## add false positive
            boxes_eval_df.at[index, 'TP'] = 0

    return boxes_eval_df
```

Figure 80: The assign_tp_fp function.

Once we know if a box is a TP or FP, the precision and recall values can be computed. The **assign_class_precision_recall** function iterates over the dataframe with the TP column included from the assign_tp_fp function and computes the precision and recall from the top row to the bottom. In each row, the precision will be the total number of True Positive boxes seen at that iteration divided by the number of detected boxes seen at that moment. The recall will be computed by dividing the number of TP boxes seen at that moment by the total number of ground truth boxes in the dataset. The function returns the dataframe with the precision and recall columns added

Although both assigning TP/FP and computing the precision and recall could be made in the same loop, I decided to split those functions, as having the TP/FP dataframe could be used to compute other metrics if needed.

Figure 81 shows the implementation of the assign_class_precision_recall function and Figure 82 shows the resulting dataframe after using the dataframe from Figure 79 on the assign_tp_fp function with an IoU threshold of 0.5 and then using it on assign_class_precision_recall function. Figure 83 shows the plotted precision-recall curves for that dataframe, using the precision and recall values.

```python
def assign_class_precision_recall(TP_df, class_id, num_gt_boxes):
    """
    Assign precision and recall ot the dataframe iteratively
    """
    # use only the boxes with the deriser class label
    prec_rec_df = TP_df[TP_df['label_id'] == class_id]
    prec_rec_df = prec_rec_df.sort_values(by='confidence', ascending=False)
    prec_rec_df = prec_rec_df.reset_index(drop=True)

    tp_count = 0
    for index, row in prec_rec_df.iterrows():
        # increase the total true positive count
        tp_count += row['TP']

        # precision = true positives / (true positives + false positves)
        # each row is either a true positives or a false positive,
        # so precision = tp_count / index+1
        prec_rec_df.at[index, 'Precision'] = tp_count / (index+1)

        # recall = true positives / num_gt_boxes
        prec_rec_df.at[index, 'Recall'] = tp_count / num_gt_boxes

    return prec_rec_df
```

Figure 81: The assign_class_precision_recall function.

	image_name	label_id	confidence	GT_Box	IOU	TP	Precision	Recall
0	04C4_Luderick_25.mov_5fps_000011.jpg	1.0	0.999672	0.0	0.956599	1.0	1.000000	0.000613
1	27C3_Luderick_30.mov_5fps_000019.jpg	1.0	0.999657	0.0	0.941160	1.0	1.000000	0.001225
2	2205_Luderick_25.mov_5fps_000091.jpg	1.0	0.999654	0.0	0.854590	1.0	1.000000	0.001838
3	2205_Luderick_25.mov_5fps_000141.jpg	1.0	0.999631	0.0	0.899627	1.0	1.000000	0.002451
4	27C3_Luderick_30.mov_5fps_000018.jpg	1.0	0.999626	0.0	0.945412	1.0	1.000000	0.003064
...	...	...	...	...	...	...	...	...
2107	2205_Luderick_50.mov_5fps_000006.jpg	1.0	0.050522	-1.0	0.000000	0.0	0.735769	0.950368
2108	2205_Luderick_25.mov_5fps_000041.jpg	1.0	0.050379	5.0	0.491430	0.0	0.735420	0.950368
2109	2205_Luderick_25.mov_5fps_000061.jpg	1.0	0.050093	13.0	0.353808	0.0	0.735071	0.950368
2110	2205_Luderick_25.mov_5fps_000198.jpg	1.0	0.050078	3.0	0.671362	1.0	0.735197	0.950980
2111	04C2_Luderick_22_005400.jpg	1.0	0.050068	-1.0	0.000000	0.0	0.734848	0.950980

2112 rows × 8 columns

Figure 82: Resulting precisión-recall dataframe with an IOU threshold of 0.5.

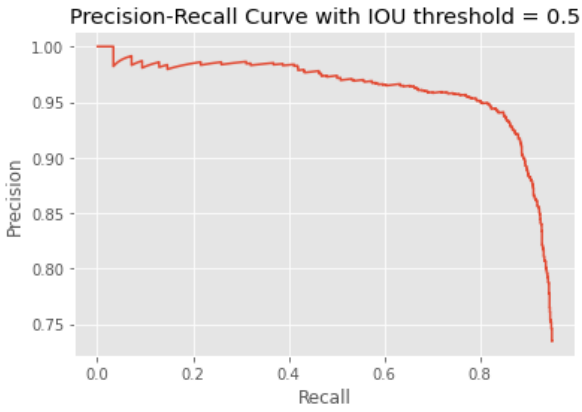

Figure 83: The precisión-recall curves of the validation dataset with 0.5 IoU threshold.

As the final mAP needs precision-recall curves for multiple IoU thresholds, I created the **get_threshold_prec_recall**, which joins the assign_tp_fp and the assign_class_precison_recall functions. Figure 84 shows the **get_threshold_prec_recall** function and Figure 85 shows precision-recall curves for the validation dataset with different IoU thresholds. When the IoU threshold is higher, the curve falls faster as there are less detected boxes considered True Positives.

```
def get_threshold_prec_recall(box_eval_df, iou_threshold, class_id, num_gt_boxes):
    """
    Returns the final Precision and Recall dataframe given a box_eval_df and an IOU threshold
    """
    out_df = assign_tp_fp(box_eval_df, iou_threshold)
    out_df = assign_class_precision_recall(out_df, class_id, num_gt_boxes)
    return out_df
```

Figure 84: The get_threshold_prec_recall function.

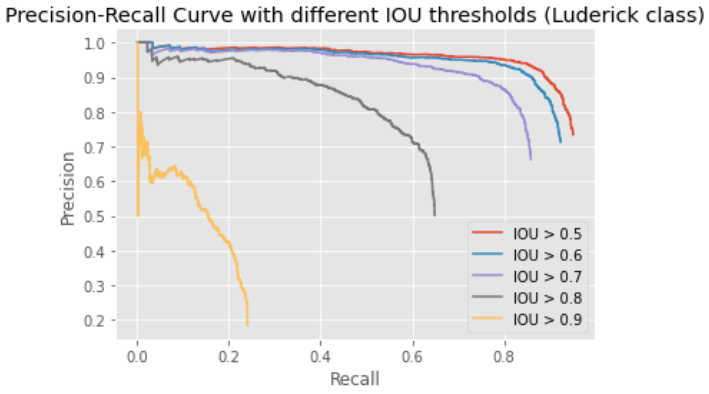

Figure 85: Precision-Recall curves for different IoU thresholds.

Now that we have the precision-recall curves, the AP can be found by computing the area under the curve, which is the same as computing the curve integral. I created the **compute_trapezoid_integral**, which computes the numerical integral using the trapezoidal rule with 2 points. The trapezoidal rule takes two points and computes the area of the trapezoid where the altitude is the x-axis distance between the two points, and the y-axis of each point is each of the bases. The area of a trapezoid is ½*(base1 + base2) * height.

The **compute_class_AP** function will return the area under the given precision and recall lists by lopping over them and calling the compute_trapezoid_integral at each iteration and accumulating the results until the final area under the curve is obtained. Figure 86 shows the implementation of the compute_trapezoid_integral and the compute_class_AP functions.

```
def compute_trapezoid_integral(point_a, point_b):
    """
    Computes the trapezoidal numerical integration between two points
    Which is the area of the trapezoid formed by the two points and the x-axis
    where the altitude is (point_b_x - point_a_x) and point_a_y and point_b_y are the two bases
    area_trapezoid = (1/2)*(point_b_x - point_a_x)*(point_a_y + point_b_y)
    """
    return (1/2)*(point_b[0] - point_a[0])*(point_a[1] + point_b[1])
```

```
def compute_class_AP(precision_list, recall_list):
    """
    Returns the average precision of a class with a list of precisions and recalls as input
    """
    total_ap = 0
    # start at i = 1 as the trapezoidal rule needs the previous point
    for i in range(1, len(precision_list)):
        # recall is the x-axis and precision the y_axis
        point_a = (recall_list[i-1], precision_list[i-1])
        point_b = (recall_list[i], precision_list[i])
        # compute the area under the two points
        ap = compute_trapezoid_integral(point_a, point_b)
        # add to the total area
        total_ap += ap

    return total_ap
```

Figure 86: The compute_trapezoid_integral and the compute_class_AP functions.

The resulting AP for the precision-recall curves shown in Figure 85 are the following:

- (IOU > 0.5) AP = 0.9159
- (IOU > 0.6) AP = 0.8845
- (IOU > 0.7) AP = 0.8106
- (IOU > 0.8) AP = 0.5684
- (IOU > 0.9) AP = 0.1287

The final step is to compute the **mean average precision**, by computing for each class the mean between the AP with 10 IoU thresholds, **ranging from 0.5 to 0.95 with a step of 0.05**, and then compute the mean between all classes. As the preprocessed Luderick dataset only has one class apart from the background, the luderick, the mean between classes is not necessary. Either way, the **compute_mAP** function I implemented is prepared for multi-class problems.

This compute_map function shown in Figure 87 returns the **mAP@0.5:0.05:0.95** as well as the **mAP@0.5** and **mAP@0.75**, which are the mean between the class Aps with IoU threshold 0.5 and 0.75, respectively. mAP@0.5 was the main metric for object detection evaluation before mAP@0.5:0.05:0.95 became the standard and shows how accurate is the model at detecting objects even if the bounding box shape is not accurate. It is a good metric if the objective is to detect the presence of the object or its approximate position. mAP@0.75 adds more box shape accuracy to the metric and mAP@0.5:0.05:0.95 considers a higher range of box shape accuracy for its final value. The final mAP value is multiplied by 100 to convert it into a percentage.

```python
def compute_mAp(box_eval_df, preprocessed_df, num_classes):
    """
    Returns the following metrics of a given box evaluation dataframe and the original preprocessed dataframe:
    - mAP@0.5:0.05:0.95
    - mAp@.50
    - mAP@.75
    """
    # initialize the trhesholds array [0.5, 0.55, ... 0.90, 0.95]
    iou_thresholds = np.arange(0.5, 1, step=0.05)

    class_mAP_list = []
    class_mAP50_list = []
    class_mAP75_list = []
    # iterate over the classes avoiding the 0 class (background)
    for i in range(1, num_classes):
        iou_trheshold_ap_list = []
        num_gt_boxes = get_total_gt_boxes(preprocessed_df, i)
        for iou_threshold in iou_thresholds:
            # get the precision-recall dataframe
            prec_rec_df = get_threshold_prec_recall(box_eval_df, iou_threshold, i, num_gt_boxes)
            # get the average precision for this threshold
            ap = compute_class_AP(prec_rec_df['Precision'], prec_rec_df['Recall'])
            iou_trheshold_ap_list.append(ap)

        # get the mean of all the threshold APs and add it to the class_mAP list
        class_mAP = sum(iou_trheshold_ap_list) / len(iou_trheshold_ap_list)
        class_mAP_list.append(class_mAP)
        # save the mAP with IOU threshold = 0.5 to its class mAP@50 list
        class_mAP50_list.append(iou_trheshold_ap_list[0])
        # save the mAP with IOU threshold = 0.75 to its class mAP@50 list
        class_mAP75_list.append(iou_trheshold_ap_list[5])

    # get the final mAP by computing the mean of all the class mAPs
    mAP = sum(class_mAP_list) / len(class_mAP_list)
    mAP50 = sum(class_mAP50_list) / len(class_mAP50_list)
    mAP75 = sum(class_mAP75_list) / len(class_mAP75_list)
    # multiply by 100 to be in percentage format
    mAP *= 100
    mAP50 *= 100
    mAP75 *= 100
    return mAP, mAP50, mAP75
```

Figure 87: The compute_map function.

The final step is to create a single function that uses all the previous ones to get a final mAP evaluation given a model and a dataset. The **evaluate_model_mAP** function shown in Figure 88 will use all previous functions and show the three mAP values found by compute_mAP. Those values will be returned in a dictionary format as well. Figure 89 shows the results of that function on the fast model and the validation set.

The results of my mAP implementation were validated using a library with the **official COCO mAP implementation** used to evaluate models on the COCO benchmark. My mAP results had a difference lower than 1% in respect to the official implementation, possibly due to the method used for numerical integration or the initial detection threshold used on the model.

```python
def evaluate_model_mAP(model, preprocessed_df, num_classes, device, images_dir):
    """
    Returns the Mean Average Precision (mAP) metrics of an object detection model on a apreprocessed dataframe
    with the number of classes, device and images directory as extra parameters
    """
    print("Starting detection on images...")
    start_time = time.time()
    # get the box evaluation dataframe with a low detection threshold
    box_eval_df = get_boxes_evaluation_dataframe(model, device, preprocessed_df, images_dir,
                                                 num_classes, detection_threshold=0.01)
    print("Finished image processing")
    print("Starting mAP evaluation...")
    # get the mAP
    mAP, mAP50, mAP75 = compute_mAp(box_eval_df, preprocessed_df, num_classes)
    end_time = time.time()
    print ('Evaluation finished')
    final_time = end_time - start_time
    print(f'Evaluation time = {final_time:.2f} seconds')
    print('-'*50)
    print(f'Mean Average Precision')
    print(f'mAP@0.5:0.05:0.95 = {mAP:.2f}')
    print(f'mAp@.50 = {mAP50:.2f}')
    print(f'mAp@.75 = {mAP75:.2f}')
    print('-'*50)
    torch.cuda.empty_cache()
    metrics_dict = {}
    metrics_dict['mAP@0.5:0.05:0.95'] = mAP
    metrics_dict['mAP@0.50'] = mAP50
    metrics_dict['mAP@0.75'] = mAP75
    return metrics_dict
```

Figure 88: The evaluate_model_mAP function.

```
metrics = evaluate_model_mAP(model_fast, validation_df, NUM_CLASSES, DEVICE, IMAGES_DIR)
```

```
Starting detection on images...
0% of images processed
25% of images processed
50% of images processed
75% of images processed
Finished image processing
Starting mAP evaluation...
Evaluation finished
Evaluation time = 169.97 seconds
--------------------------------------------------
Mean Average Precision
mAP@0.5:0.05:0.95 = 61.62
mAp@.50 = 91.60
mAp@.75 = 71.86
--------------------------------------------------
```

Figure 89: Example results from evaluate_model_mAP.

Two different models were evaluated with evaluate_model_mAP on the validation and test sets. One model was trained with training images resized to 512 x 288 pixels, while on the other model training images were left at their original size of 1920 x 1080 pixels. Each model was evaluated on three configurations: The default one, where inference images are resized to a minimum size of 800, meaning that the lowest value between width and height of the image is converted to 800 and the other dimension is scaled with the same ratio. Another configuration is a faster model where they were the minimum size is 512. Finally, an even faster model with inference image resizing to 256 minimum size was evaluated as well. Table 4 shows the evaluation results.

Model	Inference img resize	Dataset	mAP@0.5	mAP@0.75	mAP@0.5:0.05:0.95
MODEL_512_aug_1	800	Validation	94.05	76.77	65.08
MODEL_512_aug_1	800	Test	91.51	76.60	63.99
MODEL_512_aug_1	**512**	**Validation**	**91.60**	**71.86**	**61.62**
MODEL_512_aug_1	**512**	**Test**	**91.35**	**77.60**	**64.06**
MODEL_512_aug_1	256	Validation	56.39	40.09	35.72
MODEL_512_aug_1	256	Test	62.82	46.16	40.10
MODEL_1920_aug_1	800	Validation	94.17	75.88	64.70
MODEL_1920_aug_1	800	Test	92.00	77.32	64.34
MODEL_1920_aug_1	512	Validation	90.94	70.70	60.76
MODEL_1920_aug_1	512	Test	90.37	77.45	63.71
MODEL_1920_aug_1	256	Validation	54.52	38.07	33.76
MODEL_1920_aug_1	256	Test	61.83	43.78	39.22

Table 4: mAP Evaluation results for two Faster R-CNN models with three configurations on the validation and test datasets.

From the results in Table 4, there is not much difference on rescaling images during **training**, and that reducing them to 512 pixels during inference barely reduces the model precision while being faster, achieving around **7 fps** on videos. On the other hand, reducing images to 256 pixels during inference clearly reduces precision considerably. An **mAP@0.5:0.05:0.95** above 60% is a good result considering that the best models on the COCO dataset achieve around 60% as well. An mAP@0.5 above 90% shows that the model can detect almost all correct instance of ludericks with few false positives even if the bounding box shapes are not perfect.

3.5. Training a YOLOv5 Model

Although the Faster R-CNN model achieved good results in terms of precision, its speed was very far from the real-time speed I was looking for, achieving only 7 fps on my **NVIDIA GeForce GTX 960 graphics card.** As seen in section 2.3, one stage object detection models are built for real-time detection and **YOLOv5** is one of the most fast and precise models at this time. Additionally, it is being continuously updated and is very user-friendly, offering easy to understand tutorials for its use in the official repository [34].

3.5.1. Preparing the Dataset

I created the **Luderick Detection YOLOv5 – Training** notebook to train a YOLOv5 model with the luderick dataset. YOLOv5 requires the input data to have a specific format. Each image file needs a sibling text file with the same name where annotation data is included. The text file should have one row of text for each bounding box. Each row should have the following format:

class x_center y_center width height.

Where class is the class label, starting at 0 for the first class and not including the background, x_center and y_center are the coordinates of the bounding box center and width and height are the width and height of the bounding box. Figure 90 shows an example of the annotation file format for YOLOv5.

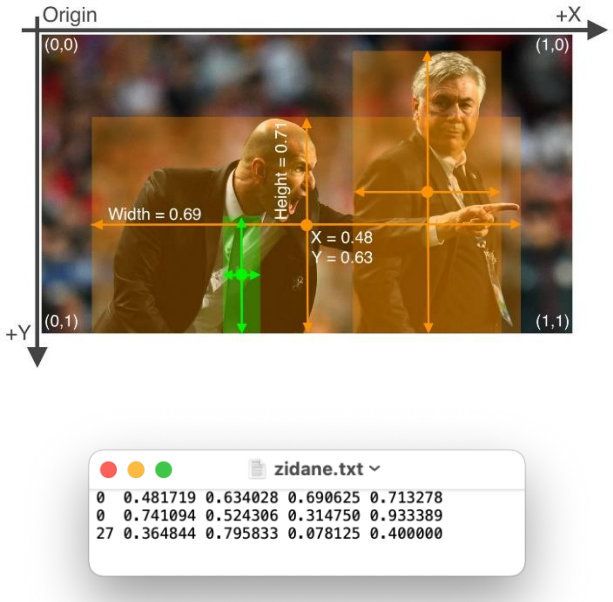

Figure 90: Annotation format for YOLOv5. Source: [34].

After loading the preprocessed luderick training and validation datasets, I implemented the **convert_to_yolo_dataset** function to generate the annotation files in the YOLOv5 format. The function takes a preprocessed luderick dataframe and generates the text file for each image by computing the center coordinates and the width and height of the bounding boxes. Although the luderick class had the label 1 in the original dataset, it was set to 0 for the text annotations for YOLOv5. Figure 91 shows the implementation of the convert_to_yolo_dataset function.

```
# Copy images and create .txt annotations
def convert_to_yolo_dataset(luderick_df, images_dir, split_name):
    """
    Creates the yolo formatted labels from the luderick dataframe into datasets/luderick/labels/split_name
    and copies the images itno datasets/luderick/images/split_name
    """
    print("Starting dataset conversion...")
    sep = os.sep
    labels_out_path = f"datasets{sep}luderick{sep}labels{sep}{split_name}"
    img_out_path = f"datasets{sep}luderick{sep}images{sep}{split_name}"

    for index, row in luderick_df.iterrows():
        if index % (len(luderick_df)/4) == 0:
            print(f"{index/len(luderick_df)*100}% completed")
        image_path = os.path.join(images_dir, row['image_name'])
        image = Image.open(image_path).convert("RGB")
        # copy original image into yolo dataset path
        !copy {image_path} {img_out_path}

        width, height = image.size
        boxes = np.array(row['bounding_boxes'])
        labels = row['labels']
        image_name = row['image_name']
        # initialize the file
        # delete the .jpg from the image name
        label_file = open(f"{labels_out_path}{sep}{image_name[:-4]}.txt", 'a')
        for i, box in enumerate(boxes):
            # get width and height
            box_width = box[2] - box[0]
            box_height = box[3] - box[1]
            # get the box center
            x_center = box[0] + (box_width / 2)
            y_center = box[1] + (box_height / 2)
            # normalize by dividing by image width and height
            x_center /= width
            box_width /= width
            y_center /= height
            box_height /= height
            # yolo class labels start from 0 and does not include the background
            class_label = labels[i] - 1
            # create the yolo box line
            yolo_box = f"{class_label} {x_center} {y_center} {box_width} {box_height}"
            # add yolo box to a new line in the label file
            label_file.write(f'{yolo_box}\n')

        label_file.close()
```

Figure 91: The convert_to_yolo_dataset function.

3.5.2. Downloading YOLOv5

YOLOv5 uses Pytorch and requires the user to download its source code. To download it, I cloned the official GitHub repository [34]. After downloading the source code, the library dependencies included in the repository as a text file should be installed as well. Once installed, YOLOv5 can be loaded and used as a Pytorch model.

3.5.3. Calling The Train Script

YOLOv5 offers python scripts to train models, making the training process quite simple compared to Faster R-CNN. Before training the model, I must create a file with the YAML format that includes paths and information about our dataset. Figure 92 shows the code for the YAML file generation.

```
yaml_text = \
"""# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]
path: ../datasets/luderick  # dataset root dir
train: images/train  # train images (relative to 'path') 128 images
val: images/validation  # val images (relative to 'path') 128 images
test:  images/test # test images (optional)

# Classes
nc: 1  # number of classes
names: [ 'luderick' ]  # class names"""

with open("luderick.yaml", 'a') as file:
    file.write(yaml_text)
```

Figure 92: Generating the luderick.YAML file.

After that, training the YOLOv5 model is as easy as calling the training script with some arguments, such as image resizing, batch size, epochs and the generated YAML file path. The training script shuffles the dataset and augments the data using methods such as Mosaic, so there is no need to implement data augmentation manually like I did for Faster R-CNN. One of the arguments is the size option of the YOLOv5 model, from the options that were seen in Figure 41. I trained the Small (yolov5s), Medium (yolov5m) and Large (yolov5l) models separately for 20 epochs with the same parameters. Training YOLOv5 is faster than Faster R-CNN, which took less than 2 hours on the same GPU. Figure 93 shows the command used to train the yolov5m model.

```
# Train YOLOv5m for 20 epochs, image resize to 640 and 4 batches
!python train.py --img 512 --batch 2 --epochs 20 --data luderick.yaml --weights yolov5m.pt --workers 1
```

Figure 93: Calling the train.py script from YOLOv5 to train a medium model for 20 epochs.

YOLOv5 computes the mAP on the validation dataset during training as well as training and validation losses. Figure 94 displays the mAP and loss plots for each epoch and model size where blue, orange and red lines represent small, medium and large model, respectively.

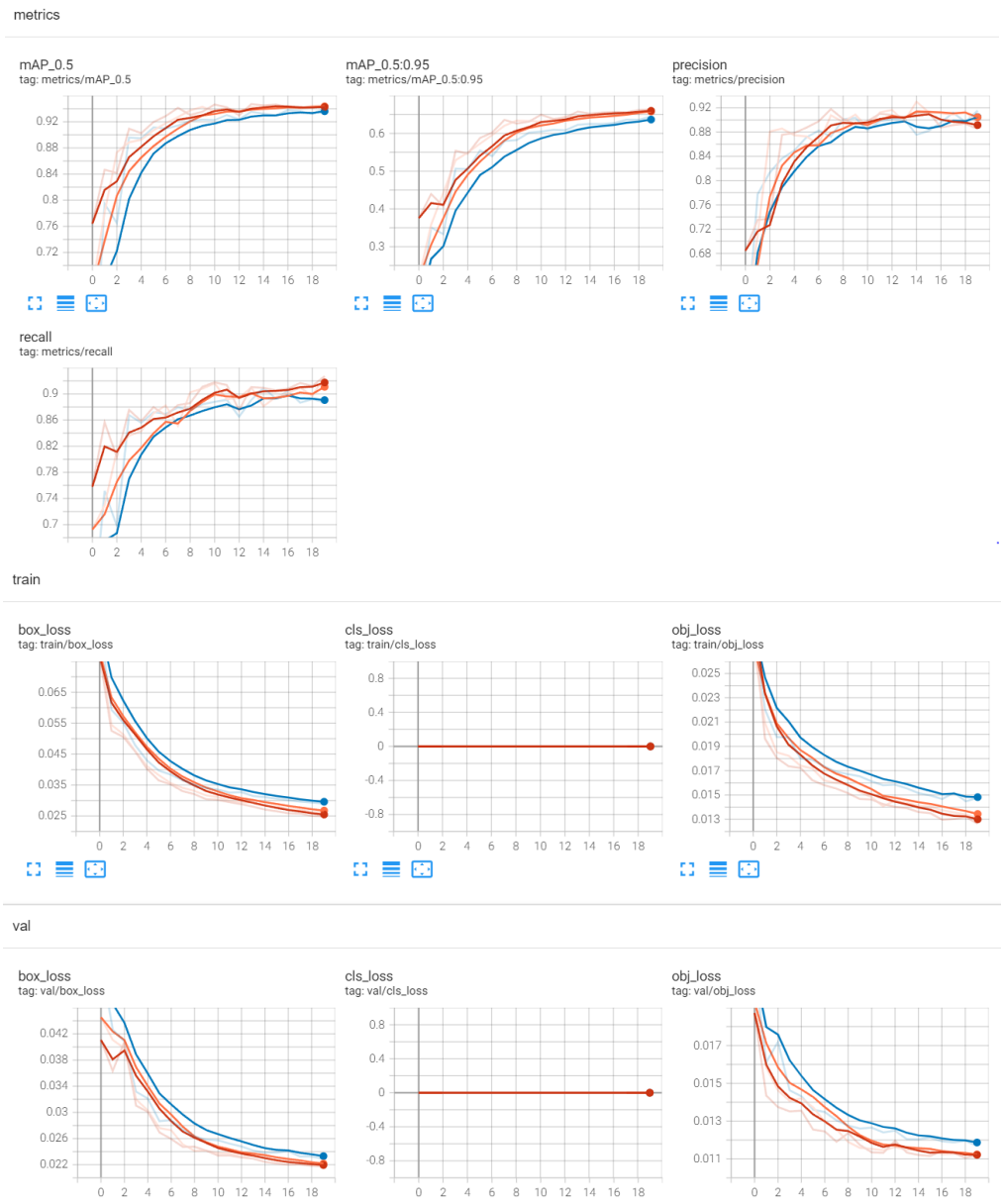

Figure 94: mAP and loss plots for each epoch and model size.

3.6. Using and Evaluating the Trained YOLOv5 Model

Most of the functions for using and evaluating the Faster R-CNN have been reused for YOLOv5 in the **Luderick Detection YOLOv5 – Inference** notebook. The main difference is loading the model and the prediction format.

3.6.1. Loading the Trained Model

To load the trained model, I use the Pytorch library to load a pretrained YOLOv5 model on the COCO dataset and use the trained weights file generated in the training notebook to update the weights. I loaded the three trained models: small, medium and large into the variables **model_s**, **model_m** and **model_l** respectively. Figure 95 shows the code used to load the small trained YOLOv5 model.

```
model_path = f"{MODELS_DIR}/best_YOLOv5s_model_1.pt"
model_s = torch.hub.load('ultralytics/yolov5', 'custom', path=model_path)  # local model
model_s.to(DEVICE).eval();
```

Figure 95: Loading the small trained YOLOv5 model.

3.6.2. Predict Bounding Boxes

Although YOLOv5 used the center points of bounding boxes along their height and width for training, predicted boxes are returned in the same format as Faster R-CNN with the xmin, ymin, xmax and ymax coordinates along their confidence score and class label. The main difference is that the luderick class label is 0 instead of 1. Results obtained from a yolov5 model can be converted to a pandas dataframe easily. I reimplemented the predict function from the Faster R-CNN notebook to return detections as a dataframe. Figure 96 shows the predict function along one example output dataframe from using it.

```
def predict(image, model):
    """
    Return, as arrays, the detection bounding boxes, labels, pred_scores and inference time in an
    PIL Image (RGB) using a given model above a given detection_threshold.
    """
    start_time = time.time()
    with torch.no_grad():
        results = model(image) # get the predictions on the image
    end_time = time.time()
    inference_time = end_time - start_time

    # get the results as a pandas dataframe
    results_pandas = results.pandas().xyxy[0]

    return results_pandas, inference_time
```

	xmin	ymin	xmax	ymax	confidence	class	name
0	397.627563	236.689697	697.171753	437.110107	0.943445	0	luderick
1	1582.460205	437.462463	1865.663574	587.593262	0.927670	0	luderick
2	1516.050781	318.521057	1729.309570	451.021301	0.904255	0	luderick

Figure 96: the predict function and one example result in the dataframe format.

I reused most of the same code from Figure 73 to apply inference on the whole validation dataset using the medium yolov5 model. The medium model ran at an average **23.6 fps**. Figure 97 displays some resulting detection images from the validation dataset using the medium YOLOv5 model.

Figure 97: Results from the YOLOv5 medium model on the validation dataset.

The same code from Figure 75 was used for inference on videos. The resulting average fps on the same video as in the Faster R-CNN notebook on the same GPU were the following:

- model_s (small) = 47.3 fps
- model_m (medium) = 33.7 fps
- model_l (large) = 21.4 fps

The small and medium models are fast enough to be considered real-time detections. The increase in fps in comparison to Faster R-CNN, which ran at 7.5 fps on the same video, is huge. All that remains is to compute the precision of the YOLOv5 models to see if that speed increase has a precision cost.

3.6.3. Evaluating the Model with Mean Average Precision

The same code for the mAP computation used on the Faster R-CNN notebook was used for the YOLOv5 model. Only small changes were applied to consider the class label for the luderick class being 0 instead of 1. Table 5 contains the evaluation results for the three YOLOv5 models. Results show that there is barely a difference in mAP between the small and large YOLOv5 models, while the small model is way faster at 47 fps. Comparing those results with Faster R-CNN bolded results in Table 4 demonstrates how the **small** YOLOv5 model is **more precise** on the validation set and almost as precise on the test set while being almost 7 times faster, **achieving detections at real-time speed**.

Model	Dataset	mAP@0.5	mAP@0.75	mAP@0.5:0.05:0.95
model_s (Small)	**Validation**	**93.31**	**73.91**	**63.43**
model_s (Small)	**Test**	**88.62**	**71.29**	**59.93**
model_m (Medium)	Validation	93.45	75.06	64.72
model_m (Medium)	Test	88.94	74.98	62.19
model_l (Large)	Validation	93.66	74.15	64.27
model_l (Large)	Test	88.27	72.83	61.55

Table 5: mAP evaluation results for the three YOLOv5 models.

3.7. Instance Segmentation Using Mask R-CNN

The first **optional objective** for this project is to train an **instance segmentation** model. As explained in section 2.2.4, mask R-CNN only adds a parallel segmentation CNN to a faster R-CNN with ROI-align instead of ROI-Pooling. The Faster R-CNN model used in section 3.3 already used ROI-Align, so the only difference is the segmentation CNN.

3.7.1. Training a Mask R-CNN Model

The **Luderick Segmentation Mask R-CNN – Training** notebook is a copy of the Faster R-CNN one with some changes. First, we need to generate segmentation mask data for the Mask R-CNN model. The model expects segmentation data in the form of a list of binary matrix, one for each object instance, in this case ludericks. Those matrices have the same size as the original image and have values of True where the pixels are part of the object and False otherwise.

The first step to get binary segmentation masks is to create one single segmentation image where each object instance has different pixel values and the background pixels are 0 (black). I implemented the **draw_image_segmentation** function which returns the segmentation image given the original image and the segmentation data. Each segmentation instance from the preprocessed dataset consists of a list of x and y values that represent the contours of each luderick. Using the **drawContours** function from OpenCV with the parameter thickness=cv2.FILLED, those contours are drawn in the image and filled with the desired color. Figure 98 shows the draw_image_segmentation function and its output.

```
def draw_image_segementation(image, segmentations):
    """
    Returns the segmentation masks from an image in a gray scale imageusing the segmentaiton contourn points
    Each segmentation has a different grayscale color
    """
    height, width, channels = np.array(image).shape

    # create a black image with the same size as the input
    segmentation_image = np.zeros((height, width, 1))

    for i, segmentation in enumerate(segmentations):
        x_coords = []
        y_coords = []
        # iterate over the coodinates of the segmentation
        for j, coord in enumerate(segmentation):
            # coordinates at even positions represent an x position
            # coordinates at odd positions represent an y position
            if j%2 == 0:
                x_coords.append(coord)
            else:
                y_coords.append(coord)
        # create an array of points (x, y)
        segm_points = [list(coord) for coord in zip(x_coords, y_coords)]

        # draw the contours of the segmentation mask and fill with color
        # each segmentation mask from the image has a different color
        cv2.drawContours(segmentation_image,
                         np.array([segm_points], dtype='int32'),
                         -1,
                         i+1, # paint each segmentation in a different color, starting from 1 (0 is black)
                         thickness=cv2.FILLED)

    # convert the shape from (height, width, 1) to (height, width)
    segmentation_image = np.squeeze(segmentation_image)

    return segmentation_image
```

Figure 98: The draw_image_segmentation function and one example result.

Once we have the segmentation image, we can convert it into the desired array of binary masks to use it on Mask R-CNN. The **get_binary_masks** function shown in Figure 99 takes a segmentation image created by draw_image_segmentation and returns one binary matrix for each unique number in the segmentation image excluding 0, which pertains to the background.

```
def get_binary_masks(segm_mask):
    """
    Converts a gray scale segmentation mask image into an array of binary boolean matrices
    """
    # create an array with all the different gray scale colors in the segmentation image
    object_ids = np.unique(segm_mask)
    # remove the background color (0)
    object_ids = object_ids[1:]

    # create a binary mask matrix for each different object with True where the pixels of each objects are
    binary_masks = [segm_mask == i for i in object_ids]

    return np.array(binary_masks)
```

Figure 99: The get_binary_masks function.

Segmentation data can be used to find the bounding box coordinates by taking the coordinates of pixels at the extremes to the right, left, bottom and top of the segmentation. This is useful if we apply data augmentation as we can use the augmented segmentation data to get the augmented bounding box. The **masks_to_boxes** function shown in Figure 100 takes a list of binary masks as input and returns their bounding box coordinates.

```
def masks_to_boxes(binary_masks):
    """
    Return the bounding box coordinates from a list of binary masks
    """
    num_objects = len(binary_masks)
    boxes = []
    for i in range(num_objects):
        # np.where returns the positions where there are True values in the matrix
        # in the form of a list of y coordinates and another of x coordinates
        pos = np.where(binary_masks[i])
        # get the minimum and maximum x and y positions
        x_min = np.min(pos[1])
        x_max = np.max(pos[1])
        y_min = np.min(pos[0])
        y_max = np.max(pos[0])

        # save the box in the list
        boxes.append([x_min, y_min, x_max, y_max])

    return boxes
```

Figure 100: The mask_to_boxes function.

We are ready to create the new **LuderickDataset** class to train the Mask R-CNN model with, shown in Figure 101. It is a bit different from the one in the Faster R-CNN notebook. This time, the segmentation data is added to the target dictionary. The same data augmentation is applied to both the original image and the segmentation image, from which the binary masks and bounding boxes are extracted.

The last step before training is to load the Mask R-CNN model. This is done in a similar way to the Faster R-CNN notebook. The Mask R-CNN model is loaded from the Pytorch Library and the prediction and mask heads are replaced with new ones with the desired number of classes, 2 in our case for background and ludericks. Figure 102 shows the **create_model** function, which creates the Mask R-CNN model given the number of classes.

```python
class LuderickDataset(torch.utils.data.Dataset):
    def __init__(self, luderick_df, images_dir, classes, transforms=None):
        self.luderick_df = luderick_df
        self.images_dir = images_dir
        self.classes = classes
        self.transforms = transforms

    def __getitem__(self, idx):
        # load images and masks
        row = self.luderick_df.iloc[idx]
        image_path = os.path.join(self.images_dir, row['image_name'])

        #read the image
        image = Image.open(image_path).convert("RGB")

        num_objects = row['number_boxes']
        segmentations = row['segmentation']
        labels = row['labels']
        area = row['area']
        # get the segmentation image
        segmentation_image = draw_image_segementation(image, segmentations)

        # transform the image and the segmentation_image with the same random transforms
        seed = np.random.randint(2147483647) # make a seed with numpy generator
        random.seed(seed) # apply this seed to image tranfsorms
        torch.manual_seed(seed) # needed for torchvision 0.7
        # apply the image transforms
        if self.transforms is not None:
            image = self.transforms(image)

        random.seed(seed) # apply this seed to image tranfsorms
        torch.manual_seed(seed) # needed for torchvision 0.7
        if self.transforms is not None:
            segmentation_image = self.transforms(segmentation_image)

        # convert the segmentation image back to an array and desnormalize it (multiply by 255)
        segmentation_image = np.array(segmentation_image*255)[0]
        # convert the segmentation image into segmentation mask array
        masks = get_binary_masks(segmentation_image)
        # get the boxes out of the binary masks
        boxes = masks_to_boxes(masks)

        # convert everything into a torch.Tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        # there is only one class
        labels = torch.as_tensor(labels, dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)

        image_id = torch.tensor([idx])
        area = torch.as_tensor(area, dtype=torch.int64)
        # suppose all instances are not crowd
        iscrowd = torch.zeros((num_objects,), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["masks"] = masks
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

        return image, target

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

Figure 101: The LuderickDataset class for instance segmentation.

```python
def create_model(num_classes):

    # load mask r-cnn model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True)

    # get the number of input features
    in_features = model.roi_heads.box_predictor.cls_score.in_features

    # define a new head for the detector with the required number of classes
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    # get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask,
                                                       hidden_layer,
                                                       num_classes)

    return model
```

Figure 102: The créate_model class for Mask R-CNN.

The training function is the same as the Faster R-CNN notebook. The Mask R-CNN model was trained for 20 epochs, from which the 9th epoch had lowest validation loss, as seen in the loss curves in Figure 103.

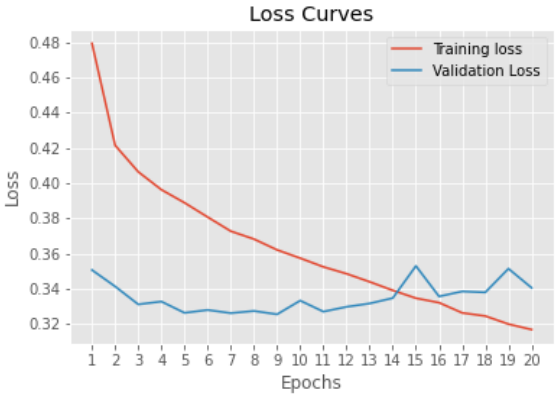

Figure 103: Loss curves for the Mask R-CNN model.

3.7.2. Using and Evaluating the Trained Mask R-CNN Model

The **Luderick Segmentation Mask R-CNN – Inference** notebook loads the trained Mask R-CNN model and uses it on the validation and test datasets. Being an instance segmentation model, it returns predicted segmentation masks as well as bounding boxes. The masks are returned as a list of matrices of the same size as the image, one for each detected luderick. The values in the matrices represent the probability of that pixel pertaining to the luderick instance. I modified the **predict** function to return **binary segmentation masks** by setting the pixels from the predicted segmentation masks that are above a given mask confidence value as True. Figure 104 shows the modified predict function.

```
def predict(image, model, device, detection_threshold, mask_pixel_confidence=0.5):
    """
    Return, as arrays, the detection bounding boxes, segmentation masks, labels, pred_scores
    and inference time in an PIL Image (RGB) using a given model above a given detection_threshold.
    """
    # transform the image into a tensor
    image = transform(image).to(device)
    image = image.unsqueeze(0) # add a batch dimension
    start_time = time.time()
    with torch.no_grad():
        outputs = model(image) # get the predictions on the image
    end_time = time.time()
    inference_time = end_time - start_time

    # get the score for all the predicted objects
    pred_scores = outputs[0]['scores'].detach().cpu().numpy()
    high_score_indices = pred_scores >= detection_threshold

    pred_scores_high = pred_scores[high_score_indices]

    # get all the predicted labels
    pred_labels = outputs[0]['labels'].cpu().numpy()
    pred_labels_high = pred_labels[high_score_indices]

    # get all the predicted bounding boxes
    pred_bboxes = outputs[0]['boxes'].detach().cpu().numpy()
    # get the boxes above the threshold score
    pred_bboxes_high = pred_bboxes[high_score_indices].astype(np.int32)

    # get all the predicted masks pixels with confidence above the mask_pixel_confidence
    pred_masks_high = outputs[0]['masks'] > mask_pixel_confidence
    pred_masks_high = pred_masks_high[high_score_indices]

    # convert to numpy
    pred_masks_high = pred_masks_high.detach().cpu().numpy()

    return pred_bboxes_high, pred_masks_high, pred_labels_high, pred_scores_high, inference_time
```

Figure 104: The predict function for Mask R-CNN.

Once we have the predicted masks, we can proceed to paint those pixels on the original image, with a different color for each detected instance. I implemented the **draw_binary_masks** function for that task. It takes the returned binary masks and multiplies the pixels of each masks, which are treated as ones if they are True, by a random color assigned to each instance in the image. Those colored masks are added to the original image and divided by 2 to avoid overflow, as higher numbers than 255 start from 0 again. The function has the option of painting all segmentations as red, which is used for video inference as each frame would show different colors for the same fish.

The code to apply inference on the whole dataset as well as in video form was reused from the Faster R-CNN notebook using the new draw_binary_masks function. The average fps using the fast model version with a minimum size rescaling of 512 was **7.6 fps on the whole validation set** and **7 fps on the sample video**, which has been submitted along this document, named **Mask RCNN video.mp4.** Figure 105 shows the implementation of the draw_binary_masks and two predictions.

```
def draw_binary_masks(binary_masks, classes, labels, image, instance_colors=True):
    """
    Returns an image array that contains the predicted instance segmentation masks with a different color for each
    instance mask if instance_colors = True
    """
    num_objects = len(binary_masks)
    # initialize the colors for each instance segmentation
    if instance_colors:
        # create random RGB colors for each segmentation mask
        colors = np.random.uniform(0, 255, size=(num_objects, 3))
    else:
        # if instance_colors = False, draw all masks with the same color
        colors = np.array([[128., 0., 0.]] * num_objects)

    # initialize a black image with the same shape as the original image and 3 color channels
    width, height = image.size
    image_masks = np.zeros([height, width, 3])
    for i, mask in enumerate(binary_masks):
        # add the mask pixels multiplied by each color to each color channel from image_masks
        for channel in range(3):
            image_masks[:, :, channel] += mask[0]*colors[i, channel]

    # convert the original iamge to float, add the segmentation masks and divide by to to avoid color values > 255
    image_masks = (np.array(image).astype('float32') + image_masks.astype('float32')) / 2
    # convert the merged image back to type int
    image_masks = image_masks.astype('uint8')

    return image_masks
```

Figure 105: The draw_binary_masks function and two example results.

The same mAP evaluation functions from the Faster R-CNN notebooks were used for Mask R-CNN. the evaluation results for two Mask R-CNN model configurations are showed on Table 6**.**

Model	Inference img resize	Dataset	mAP@0.5	mAP@0.75	mAP@0.5:0.05:0.95
Mask_RCNN	800	Validation	93.78	75.35	64.19
Mask_RCNN	800	Test	91.72	72.41	61.15
Mask_RCNN	512	Validation	90.89	71.37	61.34
Mask_RCNN	512	Test	90.97	74.57	62.00

Table 6: mAP Evaluation results of the Mask R-CNN model.

3.8. Tracking Individual Objects in Videos

In section 3.7.2 I talked about painting all segmentation masks in the video with the same color, as using random ones for each instance would result in fishes changing colors each frame. That would happen because a detection model works on individual images and cannot tell if a detected fish in one frame is the same as a detected fish in the previous frame. That issue can be solved with **tracking.**

3.8.1. Introduction to Object Tracking

Object Tracking methods involve matching bounding boxes from one image with boxes from another. A unique ID is assigned to each detection in an image, and if the next image has matching boxes, those are given the same ID as their match. In real-time video, each frame can be compared to the previous one to assign the ids to the boxes. Object Tracking is a deep subject and could be studied in a thesis of its own. The purpose of introducing the concept in this thesis is to show how can real-time object detection in videos be used for.

One way to match a bounding box with one from a previous frame is by comparing the **IoU** between them, as represented by Figure 106. If it is higher than a given threshold, the two boxes can be considered to have the same object inside. A popular method for matching boxes is to create a table with one row for each bounding box from the previous frame and one column for each new detected box in the new frame and populate the table with the IoU value between the boxes from rows and columns. Box matching is considered an **assignment problem** [43] where the rows should be matched with the columns so that the **total cost** is the **minimum** possible. In the case of IoU, the maximum total cost is desired, so the negative IoU is used. One optimized method to resolve this problem is the **Hungarian Algorithm** [45]**,** which uses a table as the one described above as input and minimizes the problem. Implementation details about the algorithm are out of the scope of this thesis.

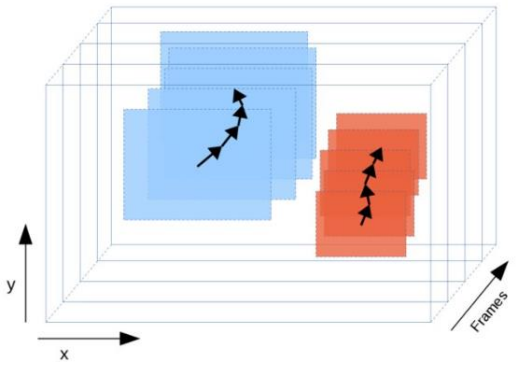

Figure 106: Visual Representation of IoU tracking. Source: [44].

Simple Online and Realtime Tracking (SORT) [46] is a simple tracking method that works on results from object detection models. SORT proposes using the Hungarian Algorithm on IoU tables matching new detections with predicted box positions and shapes of previous-frame detections. **Kalman Filters** [47] are used to predict the new position and shape of a previous box in the new frame. A Kalman Filter is a widely used algorithm to predict future values, such as object positions, given the speed of change from previous instances. SORT predicts the next position and shape of a bounding box from the previous frame, and the IoU matching with the new detected boxes is resolved with the Hungarian Algorithm. Kalman Filters are not used in this thesis.

Faster and more precise methods have been implemented, some of which use Deep Learning, such as **Deep Sort** [48] and **ByteTrack** [49], but those are out of the scope of this thesis.

3.8.2. Simple Object Tracking Implementation

The **Luderick Tracking** notebook includes a **simple SORT implementation** that does not use Kalman Filters. An **IoU matching system** has been developed as detailed in section 3.8.1. where an IoU table is created from old and new detections and matched using a Hungarian Algorithm variation.

First, the **tracked_object** class is created, as seen in Figure 107. Each instance of this class has a unique id and includes the coordinates of its bounding box on the last frame where it was matched, a color for display purposes, the number of frames where that object has not been matched to a new detection, and the number of times it has been matched with a new detection.

```
class tracked_object():
    def __init__(self, track_id, box, color, unmatched_iterations, matched_iterations):
        self.track_id = track_id
        self.box = box
        self.color = color
        self.unmatched_iterations = unmatched_iterations
        self.matched_iterations = matched_iterations
```

Figure 107: The tracked_object class.

The main idea behind the algorithm is to maintain a list of tracked objects and compare them to detections on a new frame, check if they match and update the tracked objects list. This functionality has been implemented in the **track_new_object** function shown in Figure 108. The function takes a results dataframe obtained from a **YOLOv5** model, an IoU threshold and the number of maximum iterations to track an object.

The tracked_objects global variable is used, which consists of a list of tracked objects. A table with as many rows as the number of tracked objects in the list and as many columns as new detections is created and filled with the IoU between tracked objects and new detections. Next, a Hungarian Algorithm variant, the **Jonker–Volgenant algorithm** [50] from the SciPy library [51] called **linear_sum_assignment** is used on the negation of the IoU matrix to get matches between tracked objects and new detections.

If a new detection is matched to an existing tracked object and the IoU is above the given threshold, the bounding box and the number of matched iterations of that tracked object is updated. If not, a **new tracked object is created with a new id and a random color**. If one tracked object is not matched to any new detections in a new frame, its number of unmatched iterations increases and the tracked object is deleted if this number becomes higher than a given threshold, which is 2 in this case. This means that the algorithm can keeps tracks of objects that are not detected for 2 frames before being detected again.

A similar function to the ones used on previous notebooks is used to apply the object tracking algorithm on videos. The global variables tracked_objects and track_id_iterator are initialized as an empty list and 1, respectively. At each frame, the trained YOLOv5 small model from section 3.5 is used on each video frame and the results are sent to the track_new_objects function to update the list of tracked objects. The tracked objects are drawn in the image using a similar function to the one used to draw bounding boxes on previous sections, where the boxes are painted with the color of the tracked object and the id of each tracked object is shown above it. Finally, the video with drawn colored and identified bounding boxes is saved. The video has been submitted along this document, named **tracking_video.mp4.** Figure 109 displays three close frames from the resulting video.

The **average fps** of the combined YOLOv5 (small) detection and the tracking algorithm were **23.6 fps** on the sample video. YOLOv5 (small) by its own ran at an average of 47.3 fps on the same video. Faster and more precise tracking methods exist, as explained in the last section, but the focus of this goal was to try to implement something simple by hand as a proof of concept.

```python
def track_new_objects(result_df, IOU_threshold, max_untracked_iterations):
    """
    Update the global variable tracked_objects using detection results from a new frame
    """
    global tracked_objects
    global track_id_iterator
    new_boxes_ids = np.arange(len(result_df))

    # initialize the iou matrix with size (num tracked objects, num new boxes)
    iou_matrix = np.zeros((len(tracked_objects), len(new_boxes_ids)))
    # populate the iou_matrix
    for i in range(iou_matrix.shape[0]):
        # get the old box row from the results df
        old_box = tracked_objects[i].box
        for j in range(iou_matrix.shape[1]):
            # get the new box row from the results df
            new_box = result_df.iloc[new_boxes_ids[j]]
            # use only he coordinates
            new_box = new_box[['xmin', 'ymin', 'xmax', 'ymax']].values
            # multiply IOU by 100 and round to get clean percentages
            iou_matrix[i,j] = round(IOU(old_box, new_box)*100)

    # use the negative iou_matrix as the algorithm tries to minimize the cost.
    # setting the paramenter maximize=True lets us use the positive iou_matrix as well.
    row_ids, col_ids = linear_sum_assignment(-iou_matrix)

    # combine the row and col ids to get the final assignments
    assignments = np.array([list(zipped) for zipped in list(zip(row_ids, col_ids))])

    # filter out assignments with lower iou than threshold and save new detections
    for pair in assignments:
        # save on unmatched boxes if iou is lower than threshold
        if iou_matrix[pair[0], pair[1]] < IOU_threshold:
            # update untracked_object number of unmatched iterations
            tracked_objects[pair[0]].unmatched_iterations += 1
            # remove the tracked object if it has been unmatched for X iterations
            #if tracked_objects[pair[0]].unmatched_iterations > max_untracked_iterations:
            #    tracked_objects.remove(tracked_objects[pair[0]])
        else:
            # update the tracked object
            new_box = result_df.iloc[new_boxes_ids[pair[1]]]
            new_box = new_box[['xmin', 'ymin', 'xmax', 'ymax']].values
            # update tracked_object
            tracked_objects[pair[0]].box = new_box
            tracked_objects[pair[0]].matched_iterations += 1
            tracked_objects[pair[0]].unmatched_iterations = 0

    # find unmatched old detections and delete if has been unmatched for X iterations
    for index, old_track in enumerate(tracked_objects):
        if (len(assignments) == 0 or index not in assignments[:, 0]):
            # update untracked_object number of unmatched iterations
            tracked_objects[index].unmatched_iterations += 1
        # remove the tracked object if it has been unmatched for X iterations
        if tracked_objects[index].unmatched_iterations > max_untracked_iterations:
            tracked_objects.remove(tracked_objects[index])

    # find unmatched new detections
    for new_box_id in new_boxes_ids:
        if (len(assignments) == 0 or new_box_id not in assignments[:, 1]):
            # create new tracked object with random color
            new_box = result_df.iloc[new_box_id]
            new_box = new_box[['xmin', 'ymin', 'xmax', 'ymax']].values
            new_tracked = tracked_object(track_id = track_id_iterator,
                                         box = new_box,
                                         color = np.random.uniform(0, 255, size=3),
                                         unmatched_iterations = 0,
                                         matched_iterations = 0)
            tracked_objects.append(new_tracked)
            # increase the id iterator
            track_id_iterator += 1
```

Figure 108: The track_new_object function.

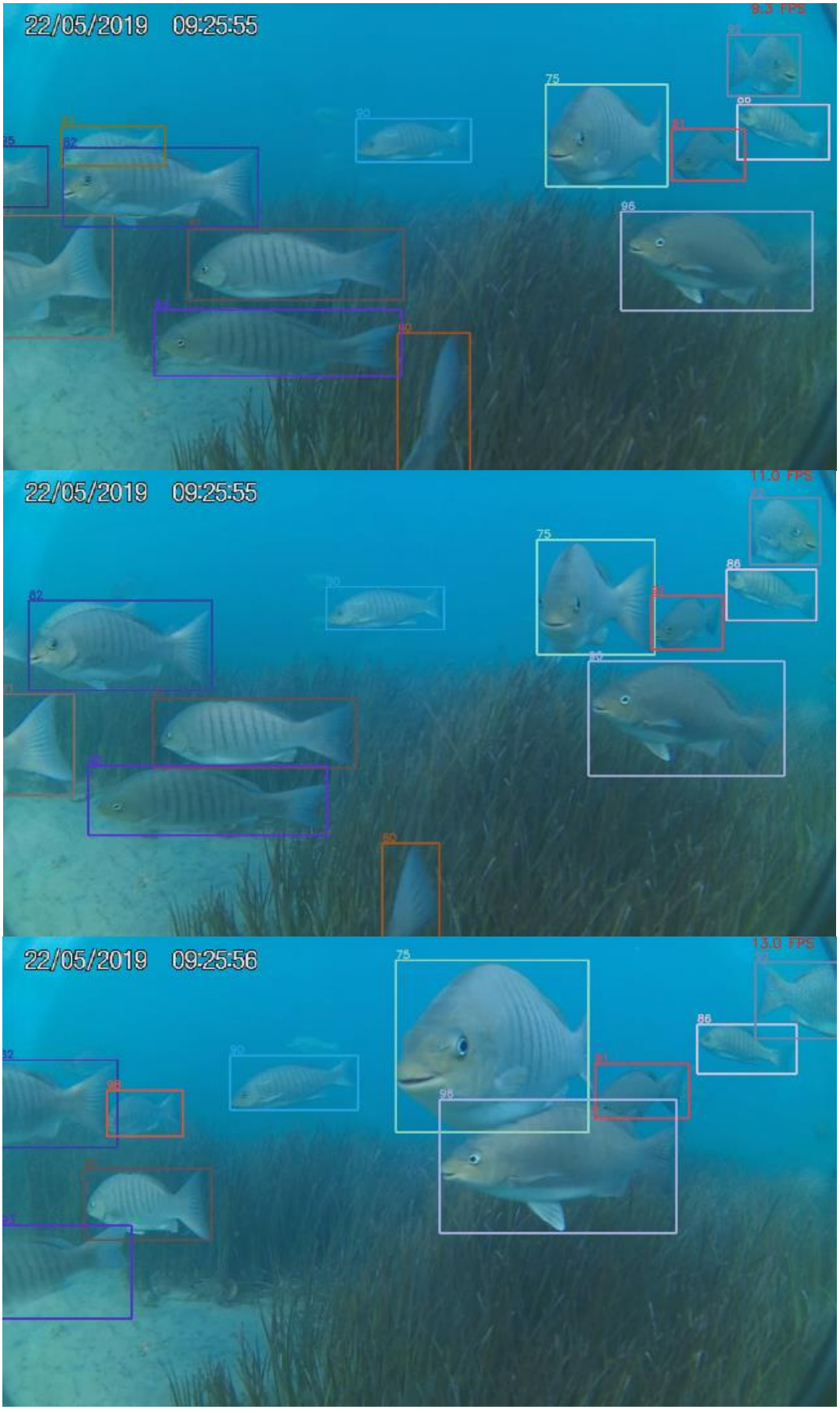

Figure 109: Three close frames from the resulting tracking_video.mp4 in chronological order.

4. Conclusions and Further Work

Over the course of this thesis, the history and evolution of object detection models has been studied. Detection models started with slow two-stage detectors which used no deep learning methods at the beginning and ended up solving all the steps with neural networks. Faster but less precise one-stage object detection models appeared using neural networks based on two-stage object detectors and their neural network architecture became increasingly complex until they were even more precise than two-stage detectors while keeping the faster inference times. This study has made me realize of the potential of Deep Learning on image data, as it is steadily replacing traditional computer vision methods to get more accurate results.

For this thesis I have trained two novel two-stage and one-stage object detection models on a public underwater dataset and tested their inference speed and precision. Clearly, the one-stage mode, YOLOv5, which achieves above 40 average fps, is much better than the two-stage options such as Faster R-CNN or Mask R-CNN, which achieve around 7 average fps with similar precision.

Instance segmentation proved to be quite easy to implement on top of a Faster R-CNN and showed accurate results. The concept of object tracking has been explored with a simple algorithm that has already given good results by being able to track most fishes while they are on camera while keeping an average framerate above 20 fps. Object tracking has proven to be a useful tool that could be used to count species, their speed or behavior in general, and demonstrates how can object detection be used for.

All the main objectives for this thesis have been completed as planned on the task schedule, including two optional objectives. The study of deep learning and object detection models proved to be the most complex task. The optional objective of creating my own detection dataset of underwater species in videos could not be completed in time. Manually labeling detection data on videos comprised of thousands of frames is a very time-consuming task and was not deemed necessary for the completion of this thesis.

Using object detection on underwater environments has a lot of potential given the difficulty of humans to explore those areas freely. Autonomous or remotely controlled robots equipped with cameras could be used to look for a specific species, count them, analyze their behavior, sizes and other data. While object detection relies heavily on GPUs for optimal speed, YOLOv5 has showed that it does not require very powerful GPUs for a real-time speed, and other solutions such as life-streaming video data from the robot to the cloud, where the detection processing would be done, could be proposed.

Further work on the topic of object detection could focus on trying new, emerging object detection models for video inference, given how fast the research is going, each year will result in more than one new architecture. Existing models could also be used with more precise but slower new classification backbones such as transformers that are becoming the state-of-the-art in terms of precision. To expand this work, an object detection model could be created and trained from scratch, layer by layer, and use it on a personally labeled dataset by completing the last optional objective of this thesis. Finally, the study of object tracking could be expanded with newer, deep learning-based methods or/and apply Kalman filters on the implemented simple SORT algorithm.

Glossary

- **AI**: Artificial Intelligence: the intelligence demonstrated by machines.
- **Bounding Box**: A rectangle in an image that contains a detected object.
- **CNN**: Convolutional Neural Network, a neural network with convolutional layers.
- **Feature Map**: The two or three-dimensional output of a layer in a neural network.
- **FPN**: Feature Pyramid Network, a neural network with bottom-up and bottom-down feature map pyramids with lateral connections
- **FPS**: Frames Per Second, the number of frames included in a single second of a video or the number of images a model can apply inference to in a single second.
- **GT:** Ground Truth. The manually assigned labels of the dataset, considered the truth.
- **IoU**: Intersection over Union
- **Kernel**: A square grid with decimal values that multiply the input pixels.
- **Loss**: A function that symbolizes the error of the model prediction.
- **mAP or AP**: mean Average Precision, an evaluation method for object detection models that computes the area under the precision-recall curve.
- **MSE**: Mean Squared Error. A popular loss function that is computed the same way as it sounds.
- **NMS**: Non-Max Suppression. Filtering out bounding boxes with lower IOU with the same ground truth box.
- **Overfitting**: Process in which a model memorizes the training data and performs poorly on outside data.
- **R-CNN**: Region based CNN. A two-stage object detection model that uses CNN and ROIs
- **Regularization**: Methods that help a model generalize what it learns to avoid overfitting.
- **ROI**: Region of Interest. A rectangle fragment from an image where there is a potential object.
- **RPN**: Region Proposal Network. A neural network that predicts ROIs.
- **SGD**: Stochastic Gradient Descent. An optimization algorithm to train neural networks based on partial derivatives.
- **TP**: True Positive. A positive prediction that is also positive in the ground truth.
- **YOLO**: You Only Look Once. A family of one stage object detection models that only process an image once.

Bibliography

[1] Deep Learning Wikipedia https://en.wikipedia.org/wiki/Deep_learning
[2] Kaggle Competition: TensorFlow - Help Protect the Great Barrier Reef https://www.kaggle.com/c/tensorflow-great-barrier-reef
[3] Image Segmentation Wikipedia https://en.wikipedia.org/wiki/Image_segmentation
[4] Python Programming Language: https://www.python.org/
[5] Pytorch: https://pytorch.org/
[6] CNN Explainer: https://poloclub.github.io/cnn-explainer/
[7] Optimizers Wikipedia: https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[8] Viola-Jones detector Wikipedia: https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework
[9] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, Yann LeCun: "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks", 2013; arXiv:1312.6229.
[10] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik: "Rich feature hierarchies for accurate object detection and semantic segmentation", 2013.
[11] Uijlings, Jasper & Sande, K. & Gevers, T. & Smeulders, A.W.M.. (2013). Selective Search for Object Recognition. International Journal of Computer Vision.
[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: "Deep Residual Learning for Image Recognition", 2015.
[13] Mingxing Tan, Quoc V. Le: "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", 2019, International Conference on Machine Learning, 2019.
[14] Imagenet Dataset website: https://www.image-net.org/
[15] Ross Girshick: "Fast R-CNN", 2015.
[16] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun: "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", 2015.
[17] Shilpa Ananth: Faster R-CNN for object detection blog post. https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46
[18] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick: "Mask R-CNN", 2017.
[19] Renu Khandelwal: Computer Vision: Instance Segmentation with Mask R-CNN blog post: https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1
[20] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie: "Feature Pyramid Networks for Object Detection", 2016.
[21] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi: "You Only Look Once: Unified, Real-Time Object Detection", 2015.
[22] Joseph Redmon, Ali Farhadi: "YOLO9000: Better, Faster, Stronger", 2016.
[23] Joseph Redmon, Ali Farhadi: "YOLOv3: An Incremental Improvement", 2018.
[24] Alexey Bochkovskiy, Chien-Yao Wang, Hong-Yuan Mark Liao: "YOLOv4: Optimal Speed and Accuracy of Object Detection", 2020.
[25] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, Youngjoon Yoo: "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features", 2019.
[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov: "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", 2014
[27] Golnaz Ghiasi, Tsung-Yi Lin, Quoc V. Le: "DropBlock: A regularization method for convolutional networks", 2018.
[28] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, Dongwei Ren: "Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression", 2019.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition”, 2014.
[30] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, Jiaya Jia: “Path Aggregation Network for Instance Segmentation”, 2018.
[31] Sanghyun Woo, Jongchan Park, Joon-Young Lee, In So Kweon: “CBAM: Convolutional Block Attention Module”, 2018.
[32] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh: “CSPNet: A New Backbone that can Enhance Learning Capability of CNN”, 2019.
[33] YOLOv4 architecture images: https://github.com/AlexeyAB/darknet/issues/5891#issuecomment-695117608
[34] YOLOv5 official repository: https://github.com/ultralytics/yolov5
[35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg: “SSD: Single Shot MultiBox Detector”, 2015.
[36] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár: “Focal Loss for Dense Object Detection”, 2017.
[37] Mingxing Tan, Ruoming Pang, Quoc V. Le: “EfficientDet: Scalable and Efficient Object Detection”, 2019, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2020).
[38] Evaluation metrics for object detection and segmentation: mAP blog post: https://kharshit.github.io/blog/2019/09/20/evaluation-metrics-for-object-detection-and-segmentation
[39] FishCLEF Dataset: https://github.com/perceivelab/FishCLEF-2015
[40] Ditria, E. M., Connolly, R. M., Jinks, E. L., & Lopez-Marcano, S. (2021). Annotated video footage for automated identification and counting of fish in unconstrained marine environments (Version 1.0.0) [Computer software]. https://doi.org/10.1594/PANGAEA.926930
[41] Luderick-Seagrass dataset Repository: https://github.com/globalwetlands/luderick-seagrass
[42] OpenCV Library: https://opencv.org/
[43] Assignment Problem Wikipedia: https://en.wikipedia.org/wiki/Assignment_problem
[44] Bochinski, Erik & Eiselein, Volker & Sikora, Thomas. (2017). High-Speed Tracking-by-Detection Without Using Image Information [Challenge winner IWOT4S].
[45] Hungarian Algorithm Wikipedia: https://en.wikipedia.org/wiki/Hungarian_algorithm
[46] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, Ben Upcroft: “Simple Online and Realtime Tracking”, 2016
[47] Kalman Filter Wikipedia: https://en.wikipedia.org/wiki/Kalman_filter
[48] Nicolai Wojke, Alex Bewley, Dietrich Paulus: “Simple Online and Realtime Tracking with a Deep Association Metric”, 2017.
[49] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, Xinggang Wang: “ByteTrack: Multi-Object Tracking by Associating Every Detection Box”, 2021.
[50] Jonker, R.; Volgenant, A. (December 1987). "A shortest augmenting path algorithm for dense and sparse linear assignment problems".
[51] SciPy Library: https://scipy.org/
[52] COCO Dataset website: https://cocodataset.org/#home