FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Computer vision model for table tennis player detection
Student:	Yannick Daniel Gibson
Supervisor:	Ing. Karel Klouda, Ph.D.
Study program:	Informatics
Branch / specialization:	Artificial Intelligence 2021
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

This bachelor thesis aims to design and implement a machine vision model that focuses on the detection of players and ping pong paddles during a table tennis game. The result should be an application that allows easy data retrieval, application of the resulting model and visualization of the results.

Specific tasks:
1) Create and describe a suitable dataset usable for training.
2) Familiarize yourself with existing computer vision algorithms applicable to the task (e.g., YOLOv8).
3) For the selected model, use the created dataset for fine-tuning or transfer learning. Evaluate the results and select the best model.
4) Based on the best model, build an application that, among other things, visualizes the model results appropriately.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 8 December 2023 in Prague.

Bachelor's thesis

COMPUTER VISION MODEL FOR TABLE TENNIS PLAYER DETECTION

Yannick Daniel Gibson

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Karel Klouda, Ph.D.
May 16, 2024

Citation of this thesis: Gibson Yannick Daniel. *Computer vision model for table tennis player detection.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I want to thank the supervisor, Ing. Karel Klouda, Ph.D., for taking on the challenge of leading my bachelor's thesis, for the mentorship and patience. Furthermore, I want to thank my family and friends for the ability to brainstorm my ideas with them.

Declaration

In Prague on May 16, 2024

Abstract

This bachelor's thesis focuses on identifying specific targets in a ping-pong match. Among these targets are ping-pong paddles and players. Furthermore, we also decided to detect a ball and a scorekeeper in matches. We applied a computer vision system in the ubiquitous Python programming language for object detection with the architecture **YOLOv8** (**Y**ou **O**nly **L**ook **O**nce version 8) based on YOLOv5 paper. This project gets a video input, draws the enclosing bounding boxes around objects of interest, and displays the video with predictions. We acquire unlabeled data and annotate it manually while also utilizing the pre-annotation method with a pre-trained model. In addition, we supply a plethora of data manipulation techniques and analysis of our results. We end with a robust model detecting all four defined classes at the inference speed of 72 Frames Per Second (FPS).

Keywords object detection, YOLO, ping-pong, fine-tuning, deep learning, Python

Abstrakt

Tato bakalářská práce se zaměřuje na detekci objektů v pingpongovém zápase. Mezi tyto objekty patří pingpongové pálky a hráči. Dále jsme se v zápasech rozhodli detekovat míček a zapisovatele skóre. Pro detekci objektů jsme využili programovací jazyk Python s architekturou **YOLOv8** (**Y**ou **O**nly **L**ook **O**nce verze 8) vycházející z YOLOv5 architektury. Tento projekt získá vstupní video, nakreslí ohraničující rámečky kolem objektů zájmu a zobrazí video s predikcemi. Získáváme neoznačená data a anotujeme je ručně, ale zároveň využíváme metodu pre-annotation s naším natrénovaným modelem. Kromě toho jsme vytvořili množství technik manipulace s daty a analýzu našich výsledků. Výsledkem je robustní model detekující všechny čtyři definované třídy při rychlosti 72 snímků za sekundu (FPS).

Klíčová slova detekce objektů, YOLO, ping pong, fine-tuning, hluboké učení, Python

List of abbreviation

FPS	frames per second
mAP	mean average precision
SOTA	state of the art

Introduction

In the last decade, the field of computer vision has undergone revolutionary changes, spurred by significant advances in machine learning algorithms and hardware performance. This progression has made it feasible for computer systems to process images and video data at unprecedented speeds, enabling real-time applications that were previously unimaginable.

One of the key developments has been the refinement of machine learning systems, which are particularly well suited for analyzing visual imagery. Furthermore, the advent of specialized hardware has dramatically accelerated the computational capabilities required to process large volumes of image data. This has not only improved the efficiency of existing applications but has also paved the way for the development of sophisticated new technologies such as autonomous vehicles and real-time video analysis systems.

The result of this thesis will be primarily useful in the company Tipsport a. s. which will have access to use this model in their systems. Furthermore, the data processing, labeling setup, and the training data will be applied for development of other computer vision models.

As for the structure, we will first delve into establishing background knowledge with the reader, followed by looking into the available architectures that are suited for our problem. We will briefly explain the most prominent architectures and conclude by choosing one. Then we follow with an in-depth explanation of the selected architecture. It is also important to note the software licenses that are relevant to our solution. In the end, we will focus on data acquisition and model training.

As for the goals of this thesis, the theoretical part of this work aims to build common ground and introduce the reader to the concepts of computer vision and machine learning theory used in this thesis. Following the assignment details of this thesis, we will create and describe a suitable dataset for training a computer vision model, suited to use in ping-pong matches. Our primary goal is for the model to be able to responsibly identify ping-pong paddles and players in the video, while running in real time. The next task is to familiarize the viewer with the existing computer algorithms and to further investigate the selected architecture. After the selection, we will apply the acquired dataset to fine-tune the model specifically for our specific needs. The last point concludes by creating a visualization displaying models results.

Chapter 1

Background

Before delving into the thesis, it is important to establish a background with the reader. We assume that the recipient has background knowledge and understanding of Artificial Intelligence and Machine learning. Here, we will provide general information of the technical terms that are relevant to comprehending the contents of the following chapters.

1.1 Computer vision

Computer vision constitutes a significant area of study within the domain of artificial intelligence that empowers machines to process and interpret visual data from the surrounding environment. This field leverages algorithms and deep learning models to analyze images and videos captured by cameras, enabling automated detection, classification, and response to visual stimuli. Its applications are diverse, ranging from the enhancement of consumer technology with facial recognition systems to advancing critical sectors such as autonomous driving and medical diagnostics. As such, computer vision is pivotal in advancing the capabilities of machines to mimic human visual perception, thereby facilitating a deeper integration of artificial intelligence into practical and innovative applications.

1.2 Image Classification

Image classification is a task that aims to classify images into predefined classes or categories based on their visual content. This process involves training a machine learning model, typically a convolutional neural network (CNN), to recognize patterns and features within images that distinguish one class from another. Once trained, the model can predict the class of unseen images by analyzing their features and comparing them to those learned during training. Image classification finds applications in various fields, including object recognition, medical diagnosis, autonomous vehicles, and content-based image retrieval.

1.3 Localizer

A localizer is a type of algorithm or model that is used to perform object localization tasks. It is designed to identify and precisely locate objects within an image by drawing bounding boxes around them. Localizers are typically part of larger systems for object detection or recognition.

1.4 Object detection

Object detection is a computer vision task that aims to identify and locate objects within an image or a video sequence. Unlike image classification, which assigns a single label to an entire image, object detection identifies multiple objects within an image by drawing bounding boxes around them and predicts a class for each object.

1.5 Feature maps

Feature maps are outputs from applying filters to an input image or another feature map in CNNs. They highlight specific features such as edges, textures, or patterns. As data moves through successive layers of a CNN, feature maps represent increasingly complex features, aiding in tasks like image recognition and classification.

1.6 Convolutional operation, kernel, stride

In a convolutional neural network, the convolutional operation involves the application of a kernel to an input image or feature map to produce a transformed feature map. The kernel is typically a small matrix of weights that slides over the input image spatially (left to right, top to bottom). Strides define how much the filter moves across the image or input feature map after each operation. A stride of 1 means that the filter moves one pixel at a time. A larger stride, such as 2 or more, means that the kernel skips positions. In Figure 1.1 we can see a visual example of convolutional operation with 2×2 kernel and stride of 1.

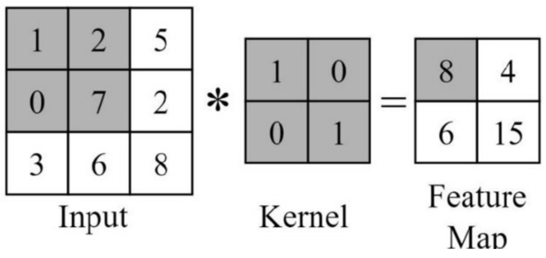


Figure 1.1 Example of a convolutional operation from [1]

1.7 Convolutions over volumes

Convolutions can be applied not only to two-dimensional images but also to three-dimensional volumes, such as RGB images, where each image has three color channels (Red, Green, Blue). In the case of RGB images, the convolution is performed using a 3D filter (e.g., $3 \times 3 \times 3$), matching the three dimensions of the image, to effectively process all color channels simultaneously. The convolution of a 3D volume results in an output that loses the depth dimension, resulting in a 2D output (e.g., from a $6 \times 6 \times 3$ image with a $3 \times 3 \times 3$ filter, the output is a 4×4 image), as the convolution aggregates information across the depth. Multiple filters can be applied to detect various features (e.g., vertical, horizontal, diagonal edges) within the same convolution layer, producing multiple output channels that represent different detected features. This multi-filter approach allows for the construction of complex feature detection systems in convolutional neural networks, crucial for processing images with varying characteristics and enhancing the network's ability to recognize diverse patterns and objects.

1.8 Normalization and batch normalization

Normalization in convolutional neural networks is a technique that enhances the performance and stability of these networks during training. It involves modifying the activations of a layer to ensure that they have a mean close to zero and a standard deviation close to one. This process helps to reduce the internal covariate shift [2], where the distribution of network activations varies significantly during training, often leading to slower convergence.

The batch normalization process normalizes the activations for each batch, calculating the mean and variance used for normalization of the batch. Let us take an example of normalizing a feature map of size $500 \times 500 \times 3$ with a batch size of 8 in a convolutional neural network using batch normalization. Assume that the convolutional layer outputs a feature map of size $500 \times 500 \times 3$ for each example in the batch. The number three, in this case, represents the depth of the feature map. The batch contains 8 such feature maps. Therefore, the input to the normalization layer from this batch would be a shape tensor $8 \times 500 \times 500 \times 3$. The mean μ is calculated separately for each of the 3 channels in all 500×500 pixels and 8 images in the batch. This results in three mean values, one for each channel. Similarly, the variance σ^2 is calculated for each channel across the same elements used to calculate the mean. For each pixel in each channel of each image in the batch, the pixel value is normalized using the formula:

$$\text{Normalized Value} = \frac{\text{Pixel Value} - \mu}{\sqrt{\sigma^2 + \epsilon}}.$$

Here, ϵ is a small constant added for numerical stability to avoid division by zero. After normalizing, each value is then scaled and shifted using trainable parameters specific to each channel:

$$\text{Output Value} = \gamma \times \text{Normalized Value} + \beta,$$

Where γ and β are parameters learned during training that allow the network to scale and shift the normalized data in ways that best aid the learning process. They are also specific to each channel. The output of the batch normalization process will still be of shape $8 \times 500 \times 500 \times 3$, preserving the dimensionality of the input feature maps but with each pixel value standardized and adjusted by the learned parameters.

Through these steps, batch normalization helps to ensure that the values throughout the deep network are standardized and well-conditioned, leading to better convergence behaviors during training [2].

1.9 Dropout layer

Dropout is a regularization technique used in neural networks, particularly effective in preventing overfitting. The concept is simple yet powerful: randomly deactivate a subset of neurons during each training step. This is achieved by element-wise multiplying the layer outputs by a binary mask. Notably, the technique is not commonly used in inference. The dropout layer encourages the network to develop a more robust set of features that are not dependent on any small group of neurons.

1.10 Bounding box

A bounding box is a rectangular box that is used to define the position and spatial extent of an object within an image in the context of image processing and computer vision. The primary purpose of a bounding box is to highlight and isolate the region of an image where a particular object is located.

Typically, a bounding box is defined by a set of coordinates that specify its position on the image. These coordinates are usually the x, y, width, and height coordinates. Alternatively, the bounding box can also specify its rotation. The definitions of the coordinates can differ; the x and y coordinates might represent the center of an object in some cases, and in others, they may refer to the top-left corner.

1.11 Intersection Over Union (IOU)

Intersection over Union (IOU) is a metric used to evaluate the accuracy of an object detector on a particular dataset. It measures the overlap between the predicted bounding box and the ground truth bounding box. The IOU is calculated by dividing the area of the overlap between the predicted bounding box and the ground truth bounding box by the area of the union of these two boxes. The formula for IOU is:

$$\text{IOU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}. \tag{1.1}$$

1.12 Mean Average Precision (mAP)

Mean Average Precision (mAP) is a popular metric used in computer vision, particularly for evaluating the performance of models involved in object detection tasks, like identifying and locating objects within images. To understand mAP, we first need to grasp the concepts of precision and recall. Precision measures the accuracy of predictions, defined as the ratio of true positive predictions to the total number of predicted positives (true positives + false positives). Recall measures the model's ability to detect all relevant instances, defined as the ratio of true positives to the total number of actual positives (true positives + false negatives). For a single class, Average Precision (AP) is calculated by plotting a Precision-Recall, curve across different thresholds of detection confidence, typically, the threshold is in the range $[0, 1]$. We can see an example of the Precision-Recall curve in Figure 1.2. This curve shows precision values for

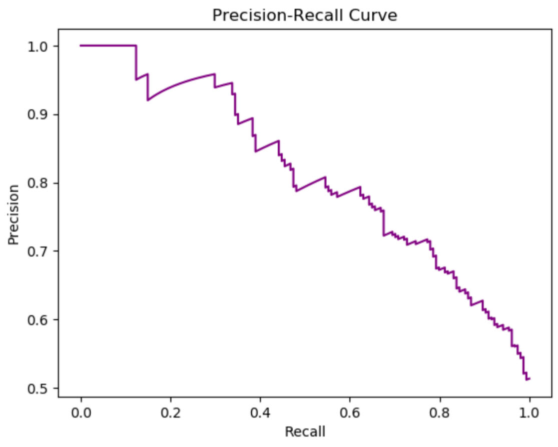


Figure 1.2 Precision-Recall curve example from [3]

different recall levels. AP is then computed as the area under the Precision-Recall curve. A higher area under the curve indicates better performance of the model at all levels of recall. In tasks where multiple object classes (such as cars, dogs, trees, etc.) are to be detected, mAP is used. mAP is the mean of the AP values for each class.

For the IOU metric, a prediction is typically considered correct if the output value is above a certain threshold (common thresholds are 0.5, 0.75, etc.). For different IOU thresholds, the mAP can be calculated, leading to metrics like mAP@0.5, mAP@0.75. We can also specify the start threshold, end threshold, and a step, following by averaging the outputs. For example, the COCO mAP [4], also denoted as mAP@[.50:.05:.95], starts with a threshold of 0.5, and by 0.05 increments ends at a threshold of 0.95.

1.13 Multinomial distribution

A multinomial distribution is a generalization of the binomial distribution. It describes the probabilities of observing counts among multiple categories. An output of a neural network must adhere to specific rules to be considered a multinomial distribution.

For example, in a classification task with k classes, the network output should correspond to the probabilities of each class being the correct classification. Each predicted probability for the classes must be non-negative, i.e., $p_i \geq 0$ for all i. Furthermore, the sum of all the probabilities must be equal to 1, $\sum_{i=1}^{k} p_i = 1$. The multinomial distribution also assumes the trials to be independent of each other. Furthermore, the output should be uniform (of a consistent shape) and yield the same result for identical inputs.

This setup is commonly modeled in neural networks through the use of a softmax output layer, which transforms the raw output scores (logits) into probabilities, ensuring that they meet the probability constraints of a multinomial distribution. Below is the softmax activation function:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}},$$

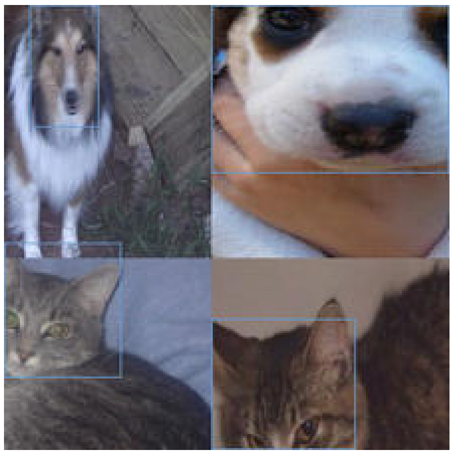
Where z_i represents the logit corresponding to the i-th class, and the denominator is the sum of the exponentials of all logits, ensuring that the output is a valid probability distribution that sums up to one.

1.14 Mosaic augmentation

The Mosaic data augmentation was first introduced in YOLOv4 [5] in 2013. It is a useful data augmentation technique for improving the robustness of object detection models in the training process. The process begins by taking four distinct images and resizing them so that they can be easily manipulated. These resized images are then stitched together to form a single composite image. This stitching is typically done in such a way that each of the original images occupies one quadrant of the composite image, essentially creating a large, new image made up of four smaller ones.

The next step involves taking a random cutout from this stitched composite. This means selecting a random section of the combined image, which effectively creates a new image that may contain parts of one or more of the original images. The size and position of the cutout are typically random. The corresponding ground truth bounding boxes need to be corrected and repositioned on the new image. If a bounding box exceeds the limits of the image, the box is sliced at the edge of the image, and the coordinates are reassigned respectively.

The mosaic augmentation technique can be combined with other data augmentation techniques such as adjusting saturation, brightness, et cetera. In Figure 1.3 we observe an example of a training input after the application of mosaic augmentation.



■ **Figure 1.3** Example of mosaic augmentation [6]

Chapter 2

Requirements and Environment

The goal of this thesis is to create a robust machine-learning model that will be used in live table tennis videos. Its primary focus is to detect the positions and sizes of paddles and people in a ping-pong match. The model will run when a match starts and stop when a match ends. The objective is to develop a program that loads video, detects bounding boxes in each frame of the video, and finally displays the bounding boxes around people and paddles. To further constrict our model's architecture, this system needs to run in real time on specialized graphics cards. The Figure 2.1 further demonstrates our desired result in a single frame of a video 2.1.

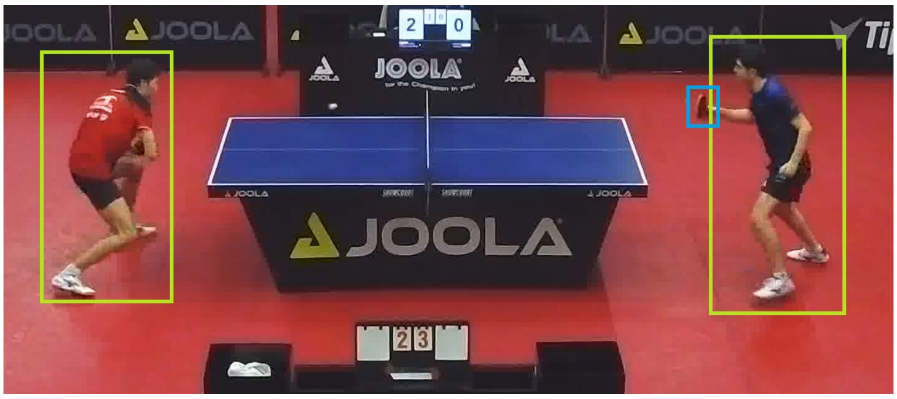


Figure 2.1 Desired result showing one paddle and two people

2.1 Speed requirements

This task constricts our choice of architecture model, since we require the model to run inference in real time, which constricts the video frame rate. Frame rate, typically measured in frames per second (FPS), is the frequency at which consecutive images, called frames, are displayed in a video. The term is particularly relevant in areas like film making, video games, and video streams, where a higher frame rate contributes to a smoother appearance of motion. The concept of "real time" in terms of FPS refers to the frame rate that is sufficient for the human eye to perceive smooth motion. In the world of cinema, a traditional frame rate is 24 FPS. This rate has been standard since the sound film era, as it provides a cinematic look with natural motion blur, which most audiences find visually pleasing [7]. Television broadcasts often utilize a frame rate of 30 FPS, offering a good balance between fluidity of motion and economical data usage, making it ideal for many standard broadcasts [8]. For applications requiring very smooth motion, such as fast-paced video games, a frame rate of 60 FPS is commonly used, since it tends to improve performance and enjoyment for users [9]. For us, we define "real time" as being 30 FPS or higher, so that is going to be our target model's goal.

2.2 Environment

In this section, we will talk about the resources at our disposal to train and run interference with the to-be-created models.

2.2.1 Development

We do not expect to have the fastest computer processing unit (CPU) nor Graphics Processing Unit (GPU) for local development. In this environment, we will make various preprocessing and postprocessing algorithms which include data transformation, visualization, et cetera. It is important to be able to iterate quickly because that enables fast and effective development. For this environment, we will use a laptop, which unfortunately does not have a dedicated graphics card, but only an integrated one. The CPU is as follows:

- **CPU:** Intel(R) Core(TM) i7-13700H 2.40 GHz,

2.2.2 Training and Testing

We will be training and testing on the following GPU and CPU:

- **GPU:** NVIDIA Tesla V100,
- **CPU:** Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz, 2993 Mhz.

We will use the remote Window Remote Desktop Connection application to access the machine with these specifications. It is very convenient to connect to the remote computer, which makes training the models seamless, not requiring us to operate via the Secure Shell (SSH) protocol.

2.2.3 Production and benchmarks

For production and benchmarking, we will use an environment which is set up on the Kubernetes network. It is not difficult to see that developing or training in this system will be time inefficient. To specify, the benchmarks we will run are not the ones to test accuracy of our model that we can do on the Tesla V100, we could even use CPU in the notebook we used. The important statistic will be how fast our production system runs interference on the set of images. The GPU and CPU available are:

- **GPU:** Nvidia A40,
- **CPU:** Intel(R) Xeon(R) Gold 5317 CPU @ 3.00GHz.

2.3 Desired result

The main goal is to have a model with its average FPS to be over 30. The model needs to be robust (with minimal throttling, or speed drops) and have visually acceptable results. We want to have a program that loads video, predicts in real time on each frame and display the results as bounding boxes drawn around people and paddles currently visible. In addition, we do not want to detect objects hidden behind other objects. This problem falls into the popular field of object detection.

Chapter 3

Picking an architecture

Before diving into working on the model for object detection, it is important to search for relevant solutions that have already been published online with the goal of reapplication or at very least inspiration.

This requires us to look through websites, articles, and scientific papers to find optimal solutions. One of such websites is `paperswithcode.com` [10], which offers a comprehensive list of SOTA models used for the specific problem of our interest, object detection. It offers multiple benchmarks using various datasets. There are several widely recognized datasets for their importance in training and evaluating object detection algorithms. These datasets are crucial because they provide a diverse range of images, annotations, and challenges that help in developing robust models. Some of the most popular are Common Objects in Context (COCO) [11], Pascal Visual Object Classes (VOC) 2007 [12] and ImageNet [13] datasets.

The first requirement upon which we need to base our choice of architecture is the inference speed of the picked model architecture. We have defined that for the results of the practical part to be considered as a success, we need to achieve an average Frames Per Second (FPS) of 30 or above. This is a speed at which the model predictions will look smooth and visually pleasing. Also, when comparing model performance in the field of object detection, one of the most crucial metrics used is the mean Average Precision (mAP). This metric is elaborated upon in Chapter 1 of this thesis. The mAP provides a comprehensive measure that evaluates how well an object detection model predicts both the presence and the precise location of objects across different classes. We will aim to maximize this metric right after the FPS requirement.

3.1 Faster-R-CNN

The first architecture we found prominent is Faster Region-Based Convolutional Neural Network (Faster-R-CNN) [14] that improves many areas of its predecessor Fast-R-CNN [15]. Faster R-CNN comprises a backbone part, like VGG [16] or ResNet [17], which extracts features from the input image such as edges, textures, colors, and shapes. These features are then fed into a Region Proposal Network (RPN), noted in the Faster-R-CNN paper, that generates potential location of objects. The RPN outputs bounding box coordinates and confidence scores. These proposals are refined and classified by a detection network, resulting in the final output: detected objects with bounding box coordinates and class labels. When we went over the precision for the architecture on common datasets, we found out that the Faster-R-CNN is very impressive, achieving an mAP of 73.2 on the popular evaluation dataset Pascal VOC, which outperforms many modern detectors. The mAP used here corresponds to the notation mAP@[0.0:0.1:1.0]. Although the accuracy is brilliant, the inference speed of the architecture is quite the opposite.

The FPS for the Faster-R-CNN hovers around 7 FPS for the most accurate variations and 18 FPS for the less precise ones. Since our resulting model needs to run on a live video, where we set the minimum FPS to 30, this model is not a good fit. In addition, the training pipeline is very complex, making the training process very slow since we need to tune hyper-parameters for many parts of the pipeline.

3.2 YOLO

One prominent architecture we explored is You Only Look Once (YOLO) [18], which revolutionized object detection tasks with its streamlined approach. YOLO condenses the detection pipeline into a single neural network, eliminating the need for separate region proposals and detection stages such as Faster-R-CNN. The network directly predicts bounding boxes and class probabilities for each grid cell across the image. This holistic approach enables YOLO to achieve remarkable speed without compromising on accuracy. By integrating the entire detection process into a single forward pass, YOLO achieves real-time performance, making it ideal for applications that require rapid object detection. The YOLO architecture achieves a mAP of 63.4 while maintaining 45 FPS on the Pascal VOC dataset. This is a considerable upgrade in speed while maintaining high accuracy; this is a completely suitable solution for usage in live streams. There is also an alleviated model, containing fewer convolutional layers, called Fast YOLO that achieves an accuracy of 52.7 while maintaining FPS of 155.

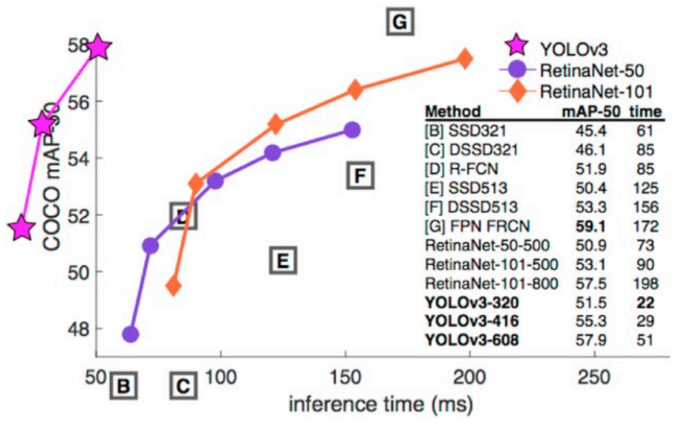
3.3 SSD

The Single Shot Detector (SSD) architecture [19] simplifies object detection by integrating the entire process into a single forward pass, eliminating the need for complex multi-stage procedures. This method divides the image into a grid, predicting bounding boxes and class probabilities for each grid cell concurrently. One of SSD's primary strengths is its use of multi-scale feature maps, which allows for effective detection of objects across various sizes; an improvement over architectures using single-scale feature maps. The SSD300 architecture has achieved a mAP of 74.3 while running 46 FPS on the Pascal VOC dataset [19]. SSD300 denotes the SSD architecture with an input shape corresponding to $300 \times 300 \times 3$. While SSD introduced substantial gains in processing objects of varying sizes with competitive accuracy, its detection speed is generally slower in comparison with other single-pass detectors such as newer versions of the YOLO architecture lineup [20].

3.4 RetinaNet

RetinaNet is a popular object detection model known for its efficiency and effectiveness, introduced by researchers from Facebook AI Research (FAIR) in their 2017 paper titled *Focal Loss for Dense Object Detection* [21]. This model addresses the challenge of detecting objects across a range of scales and, most importantly, handles the class imbalance problem, which is characterized by the under-representation of some classes in the training dataset. To combat this, RetinaNet introduces a new loss function known as Focal Loss that is designed to focus training on hard-to-classify examples. It reduces the relative loss for well-classified examples, putting more emphasis on correcting misclassified examples. The architecture adds a subnet scheme, which splits the layers into two parts, class subnet and box subnet, outputting class probabilities and bounding box coordinates accordingly. This enables the neural network's subnets to specialize in specific output tasks.

The Focal Loss technique will not be fully utilized in our assignment, as the benefit of this function is that it helps to balance the loss of less occurring classes. In a ping-pong match, we assume that at most times there are going to be all the target classes. Moreover, the accuracy and inference speeds are good but not brilliant, especially compared to the newest object detection neural networks. The Figure 3.1 compares RetinaNet implementations with YOLOv3 [22] architecture using the mAP@0.5 metric on a COCO dataset.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Figure 3.1 Comparing YOLOv3 with RetinaNet [23]

3.5 Choice of the architecture

The assignment of this thesis was to choose one computer vision model architecture that will be suitable for proceeding to the next steps of the practical parts. Our main requirements were to choose a model that performs to our speed standards, which we set as being 30 FPS, while maximizing the model accuracy. Based on our research, we concluded that the YOLO architecture is the best fit for our problem, since it provides the perfect balance of inference speed and accuracy. It offers processing speed far beyond the other detectors.

Chapter 4

YOLO Architecture

Here we will explain the basics regarding the YOLO (You Only Look Once) paper and how the model works internally. This section serves as an introduction to the matter; for further details, you are advised to read the source [18]. We will also list some of the YOLO architecture variations.

4.1 YOLO architectures

Since its inception in 2015, the "You Only Look Once" (YOLO) architecture has undergone several iterations, each introducing significant advancements in real-time object detection. From the groundbreaking YOLO to the latest iterations like YOLOv9 and beyond. These architectures have reshaped the landscape of computer vision research and practical applications. This introduction provides an overview of the evolution of YOLO architectures.

Below are bullet points highlighting several YOLO versions with their respective citations.

- **YOLOv1 (2015):** The first publication of YOLO architecture [18].
- **YOLOv2 (2016):** *YOLO9000: Better, Faster, Stronger* [24].
- **YOLOv3 (2018):** *An Incremental improvement* [22].
- **YOLOv4 (2020):** *Optimal Speed and Accuracy of Object Detection* [5].
- **YOLOv5 (2020):** *State-of-the-art real-time object detection system* [25].
- **YOLOv6 (2022):** *A Single-Stage Object Detection Framework for Industrial Applications* [26].
- **YOLOv7 (2022):** *Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors* [27].
- **YOLOv8 (2023):** *Real-Time Flying Object Detection with YOLOv8* [28].
- **YOLOv9 (2024):** *Learning What You Want to Learn Using Programmable Gradient Information* [29].

4.2 Introduction

Let us focus on the most ground-breaking model, YOLOv1. It was presented in 2015 as a new approach to object detection. Previously, object detection algorithms repurposed classifiers to perform object detection. YOLO instead frames object detection as a regression problem to spatially separated bounding boxes and associated class probabilities. Compared to previous advancements, this architecture utilizes a single neural network that predicts bounding boxes and class probabilities in one forward pass. The whole detection pipeline is a single neural network, hence it can be optimized end-to-end directly on detection performance.

In the original paper, the authors split the input image in a 7×7 grid where each cell is responsible for detecting an object that has its center in the cell. This grid is defined by how we interpret the output of our network. Here lies the main benefit of single network architecture, each grid cell has the context of the whole image; since we are not running a classifier on a specific location in the image, we are not losing context information. In Figure 4.1 we can see an example of an image separated into a 13×13 grid.

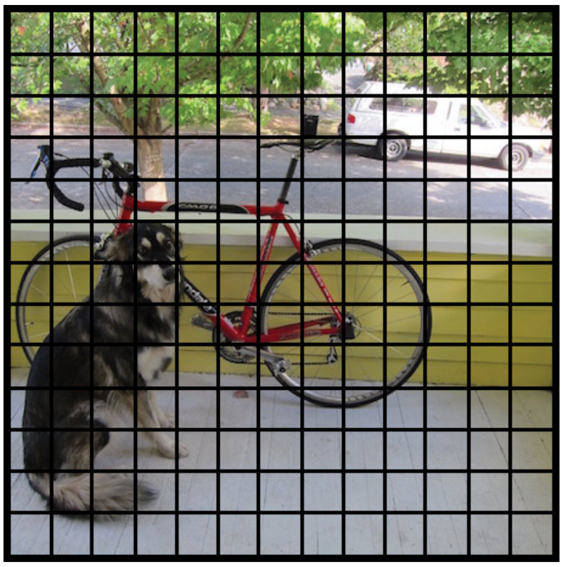


Figure 4.1 YOLO Grid Map from [30]

The input is a colorful image with a shape of arbitrary width and height with a color channel of size three. The neural network itself has an input shape of $448 \times 448 \times 3$. Trivially, the image has to be transformed to fit this input shape. However, simply resizing the image to completely fill the input size may distort the aspect ratio. To address this issue, the standard approach is to resize the image while preserving its aspect ratio. This is done by resizing the longer side of the image to 448 pixels to maintain the original aspect ratio. This ensures that the resized image fits into the fixed input size of the YOLOv1 network without distortion. You may notice that if an image does not have its width equal to its height, it will not fill the whole network input. What this preprocess procedure does is that it fills the remaining pixels with zeros on the left and right or top and bottom, depending on which side needs to be padded.

We provide an example 4.2 from a website, where we input $1280 \times 720 \times 3$ image into a neural network accepting $416 \times 416 \times 3$ as input. First, we resize the input image while preserving the aspect ratio. That transforms the image to $416 \times 234 \times 3$, then we pad the remaining space with zeros. The neural network will perceive the padded parts of the image as black.

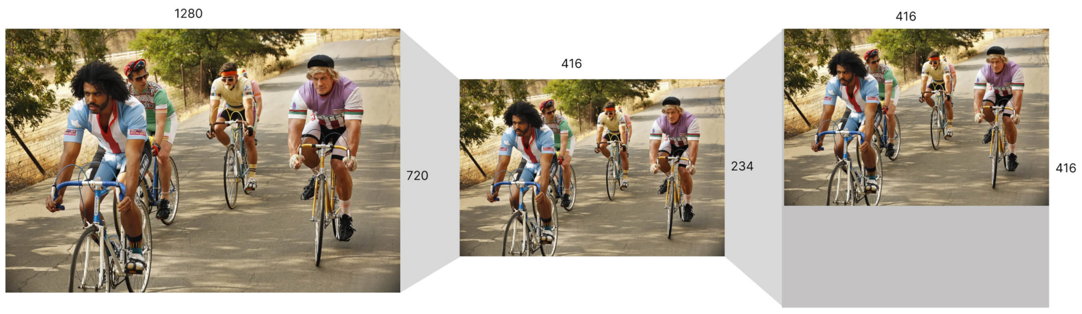


Figure 4.2 Resizing and padding [31]

4.3 Architecture

The architecture of YOLOv1 consists of 24 convolutional layers followed by 2 fully connected layers. The early convolutional layers within the network are responsible for extracting features from the image, whereas the fully connected layers are tasked with predicting the probabilities and coordinates of the output. Compared to GoogLeNet [32], YOLO uses a 1×1 reduction convolutional layer. The final layer is of size 1470 and can be represented as a $7 \times 7 \times 30$ tensor. In Figure 4.3 we can see the YOLOv1 architecture.

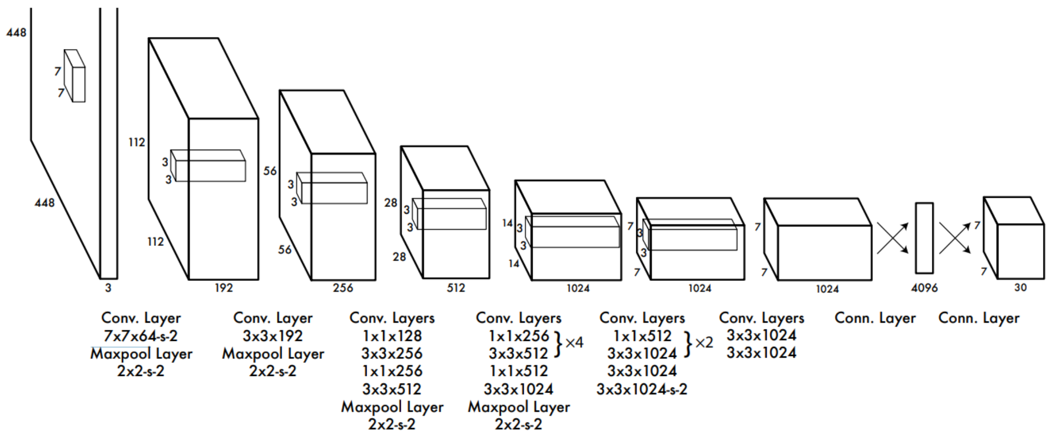


Figure 4.3 YOLOv1 layers from the [18] paper

The architecture includes convolutional layers, which are characterized by their kernel size, number of output channels, stride, as well as max pooling layers, which are defined by their kernel size and stride. Finally, the YOLOv1 model also encompasses connected layers, which are defined by their number of units. It should be noted that the architecture omits batch normalization layers, something that YOLOv2 includes. Although the authors of the YOLO paper conducted all training and inference using the Darknet framework [33], this process can also be reimplemented in libraries such as TensorFlow and PyTorch.

A linear activation function was employed for the final layer, while all other layers utilized the following leaky rectified linear activation:

$$\phi(x) = \begin{cases} x, & \text{if } x > 0, \\ 0.1x, & \text{otherwise.} \end{cases} \tag{4.1}$$

The architecture supports a parameter S denoting the size of the grid cell to be $S \times S$. A denser grid is ideal for detecting smaller objects in the scene but will take longer to train and perform a forward pass. Additionally, the parameter B indicates the number of bounding boxes to be predicted in a grid cell. Finally, there is a parameter that specifies the number of classes to predict per grid cell. For the sake of clarity, we choose the following parameters in the rest of our examples: $S = 7$, $B = 2$, $C = 20$. The authors chose these parameters in training and benchmarks, so it will be convenient to keep them consistent in our examples as well.

The following bullet list is a comprehensive and detailed account of each layer in the YOLOv1 architecture used for inference.

- Convolutional Layer: 7×7 filters, 64 out channels, stride of 2
- Max Pooling Layer: 2×2, stride of 2
- Convolutional Layer: 3×3 filters, 192 out channels, stride of 1
- Max Pooling Layer: 2×2, stride of 2
- Convolutional Layer: 1×1 filters, 128 out channels, stride of 1
- Convolutional Layer: 3×3 filters, 256 out channels, stride of 1
- Convolutional Layer: 1×1 filters, 256 out channels, stride of 1
- Convolutional Layer: 3×3 filters, 512 out channels, stride of 1
- Max Pooling Layer: 2×2, stride of 2
- 4x Convolutional Block
  - Convolutional Layers: 1×1 filters, 256 out channels, stride of 1
  - Convolutional Layers: 3×3 filters, 512 out channels, stride of 1
- Convolutional Layer: 1×1 filters, 512 out channels, stride of 1
- Convolutional Layer: 3×3 filters, 1024 out channels, stride of 1
- Max Pooling Layer: 2×2, stride of 2
- 2x Convolutional Block
  - Convolutional Layers: 1×1 filters, 512 out channels, stride of 1
  - Convolutional Layers: 3×3 filters, 1024 out channels, stride of 1
- Convolutional Layer: 3×3 filters, 1024 out channels, stride of 1
- Convolutional Layer: 3×3 filters, 1024 out channels, stride of 2
- Convolutional Layers: 3×3 filters, 1024 out channels, stride of 1
- Convolutional Layers: 3×3 filters, 1024 out channels, stride of 1
- Connected Layer: 4096 units
- Connected Layer: $7 \times 7 \times 30$ units

4.3.1 Interpreting the output

The neural network produces a one-dimensional vector as an output of size $7 * 7 * 30$ as seen in the bullet list above. More generally, the final tensor size is $S * S * (B * 5 + C)$. We interpret the shape of the result as a $7 \times 7 \times 30$ tensor, which is a three-dimensional cube corresponding to Figure 4.4.

There are 7×7 grid cells that are responsible for detecting objects in the corresponding area. Each grid cell has 30 floating-point values ranging from 0 to 1 used to predict a bounding box. Among the 30 numbers are 2 bounding boxes, each specifying 5 values. The choice of two bounding boxes instead of one is made because then the loss function will enable them to specialize for different types of objects. Within the 5 values are the normalized width and height relative to the image size, and normalized x and y coordinates relative to the grid cell position within the grid.

Additionally, each bounding box contains a confidence score, which corresponds to the probability that an object's center is in the grid cell and is accurately predicted by the bounding box. Formally, we define bounding box confidence as $\Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}}$. If there is no object, the confidence should be zero. This behavior is a consequence of the loss function used.

After the bounding box values, there are 20 conditional class probabilities, $\Pr(\text{Class}_i|\text{Object})$ where $i \in \{1, \ldots, 20\}$, which after normalization add up to 1. The probabilities do not correspond to any bounding box, but rather to the entire grid cell. When running inference, the bounding box confidence is multiplied by conditional class confidence defining class-specific confidence for each bounding box as seen below. The Class_i specifies the name of one of the 20 predefined classes.

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}} \tag{4.2}$$

These scores represent both the likelihood of the class being present within the box and the accuracy with which the predicted box matches the object. The table 4.1 further illustrates the complete output.

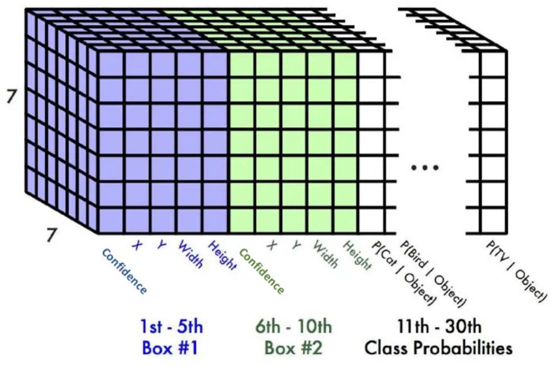


Figure 4.4 YOLO output interpreted as a three dimensional matrix with arbitrary classes from [34]

Let us circle back to Figure 4.1. We will walk through the process of interpreting the result; note that the images supplied are only illustrative. Considering that an inference was run, first image in Figure 4.5 displays all the bounding boxes in the output irrespective of their grid cell class. The thickness of the bounding box borders is chosen according to the bounding box confidence (highest confidence corresponds to the highest thickness). The middle image in the figure shows colored bounding boxes corresponding to the class with the highest conditional probability in the grid cell. The thickness is chosen relatively to the value returned from previously stated formula (4.2). The third image is a result of a non-max algorithm run on the neural networks output.

4.4 Limitations

First, we will clarify the concept "center" of an object, we begin by identifying it as the midpoint of the bounding rectangle that encompasses all the visible portions of the object. This bounding rectangle serves as a spatial framework that outlines the object's extent on a plane.

The YOLO model predicts two bounding boxes with one class for each grid cell. After this step, only one bounding box is selected based on the two confidence scores in the grid cell corresponding to each bounding box.

In scenarios where multiple objects appear within a single image, and each object's center falls into distinct grid cells on the predefined grid, this spatial separation allows a computational

Table 4.1 Thirty values in a single grid cell of the output.

	Name	Range	Description
Bounding Box 1	Confidence Score	$[0, 1]$	Certainty of the bounding box..
	X	$[0, 1]$	Relative X position of the object's center in the cell.
	Y	$[0, 1]$	Relative Y position of the object's center in the cell.
	Width	$[0, 1]$	Width relative to the entire image.
	Height	$[0, 1]$	Height relative to the whole image.
Bounding Box 2	Confidence Score	$[0, 1]$	Certainty of the bounding box.
	X	$[0, 1]$	Relative X position of the object's center in the cell.
	Y	$[0, 1]$	Relative Y position of the object's center in the cell.
	Width	$[0, 1]$	Height relative to the entire image.
	Height	$[0, 1]$	Width relative to the entire image.
Probs.	$\Pr(\text{Class}_1 \mid \text{Object})$	$[0, 1]$	Conditional probability, which together adds up to 1.
	...	...	...
	$\Pr(\text{Class}_{20} \mid \text{Object})$	$[0, 1]$	Conditional probability, which together adds up to 1.

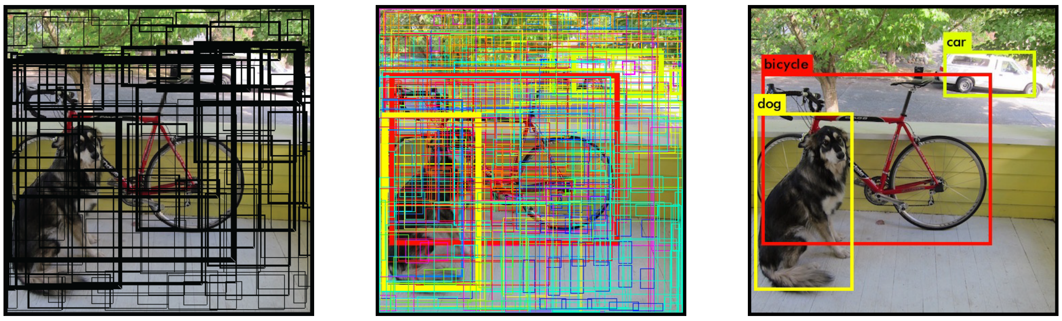
model to effectively discern and predict the presence of each individual object and their class. The distinct grid cells help to isolate the center of one object from another, thus facilitating precise predictions.

Conversely, complications arise when two or more objects share a center that falls into the same grid cell. In such cases, the underlying architecture of the model encounters limitations. Multiple centers in a single grid cell will make it impossible to predict both objects in the cell. That happens because there is only one bounding box prediction per grid cell at the end. This issue is more likely to occur when the desired prediction objects are smaller, as the smaller the objects, the greater the likelihood of more objects appearing in a single grid cell, which in turn results in worse overall predictions. This limitation is inherent in the design of the model's architecture. Issues like these can be limited or even avoided by using a more densely distributed grid or a newer model from the YOLO line-up.

4.5 Pre-training

The researchers started pre-training only the first 20 convolutional layers using the ImageNet 1000-class competition dataset [35], which was a prominent competition in the field of computer vision. The primary task of this competition was image classification. Participants developed algorithms that were capable of accurately classifying images into one of 1,000 predefined categories. These categories included a wide range of objects, animals, and scenes.

This setup was trained for approximately one week, achieving a single crop top-5 accuracy of 88%, which is comparable to other SOTA models at the time. Single crop top-X accuracy is a specific performance metric used in image classification. The metric expects the network to output a multinomial distribution, most commonly, the network's last layer will be a softmax activation function. Let us take an example. For the top-1 score, the evaluation involves checking whether the class predicted with the highest probability matches the target label. Regarding the top-5 score, it assesses whether the target label appears within the five predictions which the network is most certain with. In the top-5 accuracy metric, it is irrelevant whether the target label matches the first or, let us say, the fifth prediction; both scenarios are valued equally.



(a) Thickness corresponding to bounding box certainty

(b) Thickness corresponding to 4.2 and color matching the highest conditional class probability

(c) Resulting image after performing Non-maximum Suppression

Figure 4.5 Interpreting YOLO Result [30]

Using an arbitrary X, we proceed analogously. We will call the metric evaluation of a single image a task. The top score is calculated by dividing the number of tasks that exceed the top-X threshold, by the total number of tasks evaluated. Single Crop refers to taking only one version of the image for predicting classes. This is in contrast to Multi Crop, where multiple crops are taken (often corners and the center of the image). The predictions for multiple crops would then typically be averaged to give a final prediction for the image.

For absolute clarity, let us take an example with a single crop top-2 accuracy metric. The first image out of two contains a dog. Our network, which predicts for the classes *cat, dog, horse, mouse*, has the respective output $0.0, 0.4, 0.5, 0.1$. When we sort the probabilities, we find that *dog* is the second most confident prediction, which satisfies the single crop top-2 metric. The next image contains a mouse and our network outputs $0.3, 0.5, 0.0, 0.2$. Here, *mouse* is the third most confident prediction, that does not meet the specified metric. We have assessed all the images, the result of the metric is the number of predictions that satisfy the condition, divided by the number of tasks in total, the result is $\frac{1}{2} = 50\%$.

4.6 Training

Following the 20 convolutional layers, the researchers added four convolutional layers and two fully connected layers, this procedure was influenced by the paper *Object Detection Networks on Convolutional Feature Maps* [36]. The final layer of the network predicts both class probabilities and bounding box coordinates.

Optimization was focused on minimizing the sum-squared error in the model output. This approach does not ideally align with the goal of maximizing average precision, since it equally weights localization and classification errors. In addition, many grid cells in each image typically do not contain any object centers. This often overpowers the gradient from cells that contain objects, leading the training to diverge early. To address this issue, researchers increased the penalization of grid cells that contain objects and decreased the penalization for grid cells that do not contain objects. That corresponds to the parameters λ_{coord} and λ_{noobj}. In the loss function used to train the model is also the notation $\mathbb{1}_i^{\text{obj}}$, which tells us if an object is in a cell i. And $\mathbb{1}_{ij}^{\text{obj}}$ denotes that the jth bounding box predictor in cell i is "responsible" (has the highest Intersection Over Union (IOU) with the ground truth) for that prediction. Defining which bounding box object is "responsible" for predicting an object allows the bounding boxes to specialize for different sizes and ratios.

Moreover, the sum-squared error does not differentiate between errors in large versus small boxes. To more accurately reflect the impact of size discrepancies, the loss function takes into account the square root of the dimensions of the bounding box instead of their direct measurements. This results in size deviations being more penalized for smaller objects.

This particular loss function was used:

$$\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&\quad + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&\quad + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&\quad + \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned} \tag{4.3}$$

1. The first term penalizes inaccurate localization of centers of objects.
2. The second term penalizes the bounding box with bad width and height. The penalization is higher for smaller bounding boxes.
3. Within the grid cell, forces the most accurate bounding box confidence score to be close to the ground truth confidence score (if there is an object).
4. If there is no object, this term compels the confidence score to be close to 0.
5. The last term aims to align the class of the grid cell with the ground truth (if there is an object).

The values with a hat correspond to labels and the ones without are predictions. The measure $\hat{C}_{ij}$ is the confidence score of the bounding box which has the highest IOU with the ground truth.

The training was optimized for over approximately 135 epochs using training and validation datasets from Pascal Visual Object Classes (VOC) 2007 and 2012 [12]. For the first few epochs, the learning rate has gradually increased from 10^{-3} to 10^{-2}. Next, they continued training with a learning rate of 10^{-2} for 75 epochs, followed by 10^{-3} for 30 epochs, and concluded with 10^{-4} for the final 30 epochs.

To mitigate overfitting, a dropout layer was introduced at a rate of 0.5 after the initial connected layer, for inference, the dropout layer is not used. Data augmentation practices were employed, including random scaling and translations up to 20% of the original image size, along with adjustments to image exposure and saturation up to a factor of 1.5 in the HSV (Hue, Saturation, Value) color space.

4.7 Comparing performance

At the time of publication of the YOLO paper, object detection continued to be a fundamental challenge in the field of computer vision. The detection processes typically initiate with the extraction of robust features from input images, employing techniques such as Haar [37] and

HOG [38] features. Following feature extraction, objects within the feature space are identified using classifiers or localizers, which are executed across the entire image or specific regions in a sliding window fashion or selective approach.

The researchers compared the YOLO detection system with several leading detection frameworks, highlighting notable similarities and differences. Deformable parts models (DPM) [39], for example, adopt a sliding window mechanism for object detection and use a disjoint pipeline for feature extraction and region classification. In contrast, the YOLO system consolidates these processes into a singular convolutional neural network, enhancing speed and accuracy by performing feature extraction, bounding box prediction, non-maximal suppression, and contextual reasoning concurrently.

The Region-based Convolutional Neural Network (R-CNN) [40] and its derivatives represent another method, relying on region proposals rather than sliding windows to detect objects. First, Selective Search [41] is used to create many bounding boxes, a convolutional network extracts features, and a Support Vector Machine (SVM) [42] scores the bounding boxes. A linear model is then used to adjust the bounding boxes, with non-max suppression to remove duplicate detections. Although this setup seems rather thought through, it lacks a fundamental characteristic that the YOLO architecture fully supports – real-time inference speed. The trained R-CNN model requires a processing time of more than 40 seconds per image. Additionally, the pipeline is very complex and requires each parameter to be tuned independently, which results in a tedious training process.

The YOLO system shares certain similarities with R-CNN, such as proposing potential bounding boxes and using convolutional features for scoring. However, it introduces spatial constraints on grid cell proposals to reduce redundant detections and integrates these components into a single model optimized jointly. It also proposes significantly fewer bounding boxes per image. Using these YOLO parameters, $S = 7, B = 2$, the number of predicted bounding boxes per image is $S * S * B = 7 * 7 * 2 = 98$, compared to R-CNN, which has about 2000. The large amount of bounding boxes for the R-CNN architecture is proposed by the Selective Search method.

Beyond these, other fast detectors like Fast [15] and Faster R-CNN [14] aim to accelerate the R-CNN framework by sharing computations and employing neural networks for region proposal instead of using Selective Search, although they still do not achieve real-time performance.

Other efforts to expedite the detection process focus on the DPM pipeline, enhancing the computation of HOG features, and utilizing GPUs instead of CPUs. Nonetheless, real-time performance is achieved only by specific configurations such as the 30Hz DPM [43].

We can see Frames Per Second (FPS) and accuracy using (mean Average Precision) mAP differences between the relevant models listed in Table 4.2.

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [43]	2007	16.0	100
30Hz DPM [43]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [44]	2007	30.4	15
Fast R-CNN [15]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[14]	2007+2012	73.2	7
Faster R-CNN ZF [14]	2007+2012	62.1	18

Table 4.2 Real-Time Systems on Pascal VOC 2007. Accuracy and speed performance comparing the fastest detectors in 2015. Fast YOLO is the fastest detector on the record for the year 2015 Pascal VOC detection and is still twice as accurate as any other real-time detector. YOLO is 10 mAP more accurate than the fast version while still well above real-time in speed. For a more recent comparison see newer articles [45], [46].

4.8 YOLOv8

In January 2023, Ultralytics [47] released YOLOv8 [28], the latest version of its YOLO series. Unfortunately, the authors in Ultralytics did not accompany the release with a scientific paper. Compared to YOLOv2 and other predecessors, YOLOv8 has shifted to an anchor-free design that means it does not rely on predefined bounding boxes to detect objects (with fixed sizes and or scales) similarly to YOLOv1. Comparably, YOLOv8 predicts fewer bounding boxes. It introduces an enhanced and quicker non-maximum suppression process. Furthermore, Ultralytics incorporated mosaic augmentation during the training phase, but it is deactivated in the final ten epochs to prevent potential negative impacts. In mosaic augmentation, four different training images are combined into one composite image (as described in Chapter 1). This approach helps the model learn to detect objects under varying conditions and contexts.

YOLOv8 is available in five scaled versions: YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra-large) varying primarily in the amount of convolutional layers. In tests on the MS COCO dataset test-dev 2017 [11], YOLOv8x achieved an impressive 53.9% AP at a 640-pixel image size, outperforming YOLOv5's [25] 50.7% at the same resolution and maintaining high speed.

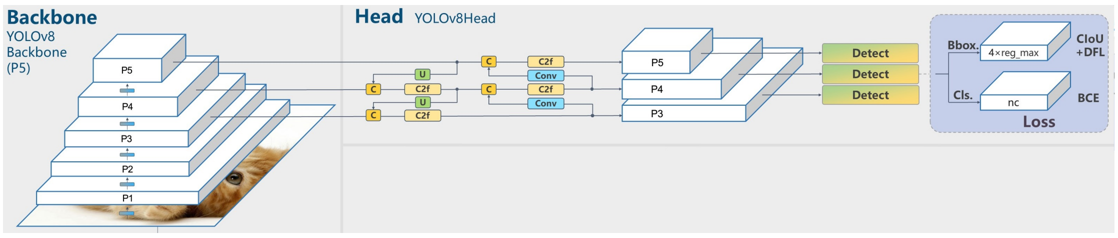


Figure 4.6 YOLOv8 Architecture [48]

In Figure 4.6, we can see the high-level overview of the YOLOv8 architecture. It separates the scheme into a backbone part responsible for extracting feature maps, and a head part. The block `C` denotes concatenation, `Cf2` specifies a custom block containing residual connections, similarly to ResNet [17]. The letter `U` indicates an upsampling convolutional layer, its main purpose is to increase the spatial dimensions (i.e., width and height) of the input feature maps. The blue block with the text "Conv" describes a convolutional layer followed by a batch normalization layer followed by a Sigmoid Linear Unit (SiLU). In convolutional networks, we usually observe a Rectified Linear Unit (ReLU) activation function, in Figure 4.7 we can see the two functions side by side.

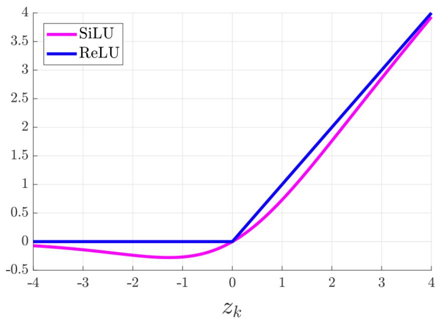


Figure 4.7 SiLU vs ReLU [49]

Chapter 5
Licenses

This chapter will guide us through the usage of licenses and their constraints. We are going to go through some of the most popular licenses used in computer software and end with a comparison of the discussed licenses.

When developing software or an algorithm in an academic environment, there is not much of a need to spend time deciphering licenses. Our thesis accompanies a commercial solution, so it is very important to look through licenses and decide which software components to use and which ones to avoid, since a misuse could lead to legal affairs costing the company a substantial amount of funds. This scenario we want to avoid at all costs.

The language used in these licenses is rather complex, incorporating intricate sentence structures and extensive clauses, which might pose significant challenges to those without legal training. The licenses are crafted to address a variety of legal scenarios, enhancing their legal thoroughness but simultaneously reducing their accessibility. Consequently, many individuals find themselves compelled to seek legal advice to decipher their rights and responsibilities under these agreements, incurring additional effort and expenses. This complexity contributes to difficulties for non-legal audiences when navigating around software licenses. It should be noted that the following text is for informational purposes only and should not be taken as legal advice.

Before we delve into the relevant licenses, let us first create a background about the terms used in this chapter.

- **distributing**: refers to the act of providing copies of a software to others, either physically or electronically, for their use. For example, providing downloadable software, installing software on client's servers, or sending copies over CDs.
- **contributor**: An individual or entity that has contributed to the source code, documentation, or other materials to a project distributed under a specific license.
- **trademark**: A trademark is a legally protected symbol, word, or phrase used to represent a product or company, distinguishing it from others in the marketplace.
- **licensee**: The licensee is the party who is allowed to use a product or service under the terms of a license agreement.
- **user**: A licensee that interacts directly with the licensed software.
- **copyleft license**: A license that enforces that if certain conditions are met, the derivative work has to be distributed under the same copyleft license.
- **copyright license**: A license outlining if a certain conditions are met, the derivative work may be distributed under different license terms.

5.1 MIT License

The Massachusetts Institute of Technology (MIT) license [50] starts by stating that a copyright applies to the software, which means that the original creators or copyright holders maintain ownership over the code. This acknowledgment serves as the foundation for the permissions granted in the MIT License. Specifically, it underscores that users must include the copyright notice in all copies or substantial portions of the software, ensuring proper attribution to the copyright holders.

The core of the license allows any person to use, copy, modify, merge, publish, distribute, sublicense, and sell copies of the software without any restrictions. This means that users have extensive freedom regarding the usage and distribution of the software.

The license includes a disclaimer stating that the software is provided “as is” without any warranty. This means that the software is provided without any guarantees regarding its performance or suitability for any particular purpose. For example, if users of software licensed under the MIT license suffer financial losses caused by the software, the copyright holders are not liable. Users are advised to use the software at their own risk.

The MIT License is suitable for various types of projects, including commercial and non-commercial ones. It is often chosen for libraries, frameworks, and utilities due to its minimalistic requirements and permissive nature.

5.2 Apache License

The Apache License 2.0 [51] is an open-source license developed by the Apache Software Foundation. It is widely used for software projects as it provides a balance between permissive and protective terms, fostering collaboration and innovation within the software community. In comparison to the MIT license, the Apache License 2.0 is considerably more verbose, comprising approximately ten times the number of words. It is arranged into multiple sections, starting with definitions, where the license defines key terms used throughout the text, like “Contributor”.

Contributors grant a broad royalty-free copyright license to use, reproduce, prepare derivative works of, display, perform, sublicense, and distribute the work and any derivative works. This means that we can use the software freely and include it in our projects, provided that we adhere to the terms of the license.

This license provides an express grant of patent rights from contributors. It explicitly allows users to utilize the patented technologies without the risk of infringement claims. The Apache License 2.0 offers clarity and legal protection for both the contributors and the users of the software.

For example, imagine that you have contributed to a scientific paper that introduces a novel object detection algorithm. As part of your contribution to the paper, you have also filed for a patent for the algorithm that is still pending. In the meantime, you decide to make the implementation of this algorithm open source and license it under the Apache License 2.0. If, for example, another researcher, Alice, comes across your open-source library and decides to use it in her project, then if the patent is approved, Alice still has the right to freely use the library in her software. Alice would then be granted a license for the underlying patent. In conclusion, Apache License 2.0 protects users from infringing on potential patents that might be attached to parts of the open source.

If modifications were made to the licensed software, it is necessary to explicitly present them. Usually, this means adding a notice to the modified source code or documentation, such as appending it to the end of the `LICENSE.txt` file of the licensed software. This notice should clearly outline the changes made and the date of modification. Additionally, it is important to provide contact information or references to additional documentation for users to refer to. The contact information provided in the notice may include an email address for support inquiries and a project website link for additional documentation. Users can also engage with project maintainers through social media handles or community forums to seek assistance or provide feedback on the modifications.

The Apache License 2.0 also clearly defines the limitations of trademark usage. It requires that the trademarks of the licensed software be easily accessible, which especially applies for distributed software. This could mean putting credit in the About section, where the user mentions that his software does utilize the licensed software. On the contrary, you are not allowed to use the trademark to boast about your software, for example, if you use a library licensed under the Apache License 2.0 owned by Google [52], you cannot show your clients or potential clients that the software is a

Google product. That also applies to logos, so you are not allowed to list a Google logo on the front page of your product that would potentially influence the product's success. This way, the licensees do not have to worry about attaching their reputation to the products of users.

Analogously to the MIT license, the Apache License 2.0 does not provide any warranty to users. The license owners do not have the responsibility to fix a bug in their software, let alone ensure that the software functions. Following the analogy, the Apache License 2.0 is not liable for any losses users might experience as a consequence of using the licensed software.

5.3 GPL 3.0 License

The GNU General Public License version 3.0 (GPLv3) [53] is a significant open-source copyleft license developed by the Free Software Foundation (FSF) [54]. It inherits the principles of its predecessors while addressing contemporary concerns in software licensing. For example, compared to GPLv2, this license protects users from patent litigation, similar to the Apache License 2.0, making it more feasible to use the license for commercial purposes. GPLv3's predecessors did not state anything in their licenses about patent protection, leaving that term ambiguous and allowing for user exploitation within the patent world. This constricts the licensee's activities, compelling them to decide whether this license truly suits their needs because if any patents are associated with the licensed software, any user must be automatically granted the patent license.

The GPLv3 seeks to ensure that software remains free for all users, protecting freedoms to run, copy, modify, and distribute software while preventing any restrictions that may limit these freedoms. For modifications specifically, as usual, the user is required to note them in the source code. The license emphasizes the distinction between "free" as in liberty, not necessarily as free of cost. The term "free" refers to the essential rights to run, study, modify, and distribute the software. These freedoms are fundamental to the philosophy of the Free Software Foundation, which promotes user autonomy and community-oriented development of software. The GPLv3 aims to protect these freedoms by ensuring that all users have the legal right to control what their software does and to share their modifications with the community. Although GPLv3 software can be distributed free of charge, the license does not require it to be free. "Free" in this context refers primarily to freedom, not to price. The license allows developers to charge for the closed-source distribution of software or to provide paid support and services related to the software. An example of a free open-source software that provides paid support is Red Hat Enterprise Linux (RHEL) [55]. This model not only ensures wide accessibility due to its open-source nature but also generates revenue through premium support services, catering to enterprise needs and ensuring system reliability and security.

As this is a copyleft license, as opposed to a copyright one, it requires all the derivative work to be released under the same license terms. This ensures that derivative works remain open for modification and redistribution, thereby supporting the open-source community.

The license includes specific provisions that facilitate compatibility with other free software licenses, which is a significant enhancement over its predecessors. This compatibility is crucial for developers who want to combine or integrate code from various projects that are licensed under different free software licenses. For example, if we would like to develop a project using the libraries using the MIT and GPLv3 licenses, we are allowed to do so, since the GPLv3 is more restrictive and compatible with the MIT license, which will result in a project that needs to be released under GPLv3 license. The same goes for the Apache License 2.0.

The act of distributing the software means, for example, sharing a compiled form of the project with end users via a physical or digital medium, such as online downloads and CDs. The license requires, if you distribute the derivative work, stemming from the licensed code, you must also release its source code under the GPLv3. But if you are not distributing your software, then you do not need to release your source code under the GPLv3 license. This is a loophole that some companies may use to their benefit when employing GPLv3-licensed software in a server-side, Software as a Service (SaaS) model. By running the software on their servers and providing access to it via a web interface without actually distributing the software itself, companies can utilize the licensed components without the obligation to publicly release their source code under the same license. This means that they can maintain the proprietary nature of their added code and functionalities, which do not leave the server environment. This approach is legally valid under the terms of the GPLv3 license but is addressed and closed by the Affero General Public License version 3(AGPLv3) which we will discuss in the AGPLv3 Section.

If a licensee violates any of the conditions of the license, their rights granted under the license are automatically terminated. However, the GPLv3 license also offers the opportunity to redeem and reinstate these rights, reflecting a forgiving approach to compliance. If the copyright holder notifies a user about the violation by some reasonable means and this is also his first breach of the license, the user has a 30-day period within which he has time to fix the inadequacies that were leading to the license violation and send a receipt of notice to the copyright holder. After curing the violation issues, the user's right to the license is permanently reinstated. If the user has breached the license more than once, then the user's license will be terminated, but after completing the breach fixes, the license is provisionally reinstated until or if the licensor finally terminates the user's license. Under the condition that the licensor fails to notify the user by some reasonable means within 60 days after cessation, the user's license is permanently renewed. This provision is particularly important, as it encourages compliance by providing licensees with an opportunity to correct their mistakes without facing permanent exclusion from the use of the software.

5.4 AGPL 3.0 License

The Affero General Public License version 3.0 (AGPLv3) [56] is a critical open-source copyleft license developed by the Free Software Foundation (FSF). It builds upon the principles established by its predecessors, including the GNU General Public License (GPLv3), while introducing measures to address the use of software over a network—an area not covered by earlier versions. Like the GPLv3 license, the AGPLv3 license provides robust protection against patent litigation, adopting similar provisions to those in the Apache License 2.0. This makes the AGPLv3 license particularly viable for commercial applications, ensuring that all patents associated with the licensed software automatically grant a patent license to all users.

The AGPLv3 aims to keep software free for all users, safeguarding the freedom to run, copy, modify, and distribute software, even when the software is used over a network. This addition closes the so-called "SaaS loophole", requiring that modifications and additions be made available to all users of the software, not just those who receive a copy of the original program. The distinction between "free" as in freedom and "free" of cost is maintained, underlining the rights to use, study, modify, and share the software—cornerstones of the Free Software Foundation's ethos, which promotes user autonomy and collaborative development.

If a licensee fails to comply with the terms of AGPLv3, their rights under the license are terminated, but these rights may be reinstated if the breach is remedied. This reflects the forgiving nature of the license, which aims to encourage compliance and allow for the correction of errors, thereby fostering a more inclusive and cooperative development environment. This provision underscores the license's goal of promoting ongoing collaboration and innovation in the software community, ensuring that all enhancements remain freely available.

We will now discuss a real-world example, a software called Nextcloud [57], it is a self-hosted file server that allows users to store, access, and share files securely. It provides features similar to popular cloud storage services such as Dropbox [58] or Google Drive [59] but offers users greater control over their data by enabling them to host the server themselves. With Nextcloud, users can collaborate on documents, manage calendars and contacts, even run additional apps to extend its functionality. Users can self-host their files internally using Nextcloud without needing to open-source their proprietary files or data. The requirement to provide access to the source code applies to the Nextcloud software itself, not to the data or files stored within Nextcloud instances. Users retain control and ownership of their data and are not obligated to open source their proprietary files or information. But if a user decides to modify the Nextcloud software and repurpose it into a hosting service that they either distribute as a program or make available online to end users, then the licensees are allowed to do that only under the condition that they also license their program under the same license – AGPLv3. As a consequence, the code will need to be publicly available, which might not be ideal for a person or entity.

5.5 Ultralytics license

Ultralytics [47] is a popular open-source project led by Glenn Jocher that follows the Affero General Public License (AGPLv3) [56] licensing model. It serves as a comprehensive platform for computer vision, machine learning, and artificial intelligence applications. Similarly to Nextcloud, Ultralytics provides a host of enterprise-grade features and services to cater to the needs of organizations. Through its AGPLv3 license, Ultralytics empowers users to harness cutting-edge computer vision and AI capabilities while ensuring transparency and openness. Organizations can leverage Ultralytics for tasks such as object detection, image classification, and video analysis, as shown in Figure 5.1. This approach ensures a sustainable business model using an open-source-driven environment.

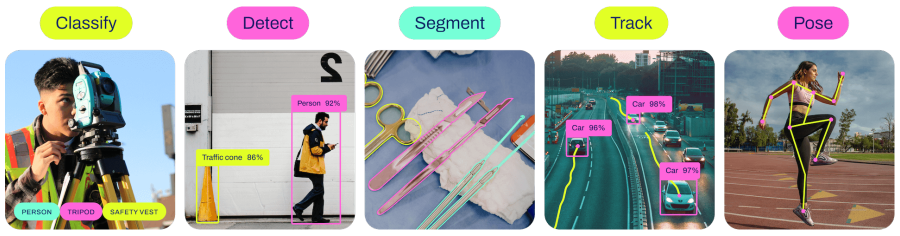


Figure 5.1 Some of the tasks supported by Ultralytics [60]

Ultralytics software offers two main licenses, AGPLv3, which is ideal for students and enthusiasts, and the second is their custom paid Enterprise license [61] which is designed for commercial use, bypassing the limitations of the AGPLv3 license. Some of the limitations are not being able to distribute the extended software without open-sourcing it or providing the extended software as a service over the network.

It is clear that to use our solution as Software as a Service (SaaS) in the company in which the solution will be provided to, we need to acquire the Enterprise license. This may not be the ideal course of action, since the price depends on negotiations that can easily result in a license price of tens of thousands of dollars. Models we trained using this licensed library could end up being used as an internal testing prototype. Additionally, there is a possibility for the Enterprise license to be acquired by the company and also to be used in other spaces in machine learning, which could make for a great addition to the tool stack.

One of the things that the library Ultralytics under the AGPLv3 license can be used for is, when we have a trained model, we can use it to generate pre-annotations and store them internally. After that, we can manually correct the annotations and train another model that is licensed under a permissive license. This usage is in alignment with the AGPLv3 license.

Although the original YOLO [18] implementation is free to use, the various easy-to-use implementations of the versions of the YOLO lineup may be licensed under differing licenses. The Ultralytics Python library, which implements the YOLO paper, exclusively uses the AGPLv3 license on their package solution. It is an open-source **copyleft** license.

The issue with this license is that, if a company uses this code in their distributed product, to comply with the license terms, it needs to provide their code to the public. That means releasing all the source code under the AGPLv3 license. The definition of providing the code publicly can also be stretched. The license defines that the code must be publicly available but does not precisely state how that must be accomplished. It would be unwise to specify in the license that the source code needs to be uploaded on a public website, since that might not be applicable in the distant future. A company can avoid uploading the code to popular publicly available platforms like GitLab [62]. For example, a company can provide the code by stating that the person or entity requesting the code needs to come directly to their office to receive a CD with the source code. Using this approach, the company allows the code to be shared, but virtually disables a large portion of potential requesters to get the source code. One of the reasons for that are the travel costs which the company does not have to cover.

5.6 Comparison

When entering the software development world, one must not only be adept at writing code but also at navigating the variety of software licenses that come with it. Each license carries specific terms that dictate how software can be used, shared, and modified. Understanding the differences among popular software licenses is not just a matter of legal compliance; it is a strategic decision that can influence the adoption, evolution, and success of a software project.

5.6.1 Popularity

Researchers from WhiteSource Software (now also known as Mendl.io) [63] have collected a database, which includes over 3 million open-source components and 70 million files, covering over 200 programming languages. The ultimate goal was to uncover the most popular open-source software licenses chosen by licensors in 2018. They also used data from previous years to compare two popular license types. The first type is a **permissive** license, which generally minimizes restrictions on how software can be used, modified, and distributed, thus allowing for a greater freedom and incorporation into both open and proprietary software. The second type, a **copyleft** license, requires that derivative works of the software be distributed under the same license terms, ensuring that modifications remain open and accessible to the community. They discovered that permissive licenses gradually started to become dominant, and in the year 2016 finally overthrew the copyleft licenses. In 2018, the prevalence of permissive licenses was 64%.

In Figure 5.2 we can see ten of the most popular software licenses in the year 2018 in a pie chart. In the previous sections), we have covered the top 3 software licenses. We can also see that the GNU General Public License (GPL) family represents more than 32% of the pie chart. There are licenses that we have covered or their previous versions, but also a Lesser General Public License (LGPL) [64] license, which compared to the GPL licenses is more permissive, as it allows users to implement the licensed software as a library under the condition that they will not modify its contents. We can also see other licenses, but they are not directly relevant to our research, so we are not going to focus on them. We can see that the MIT license leads in the forefront, which is not very surprising since the license, aiming for clarity, is very brief and concise, therefore it requires less cognitive function for the average developer to fully comprehend the full extent of the license.

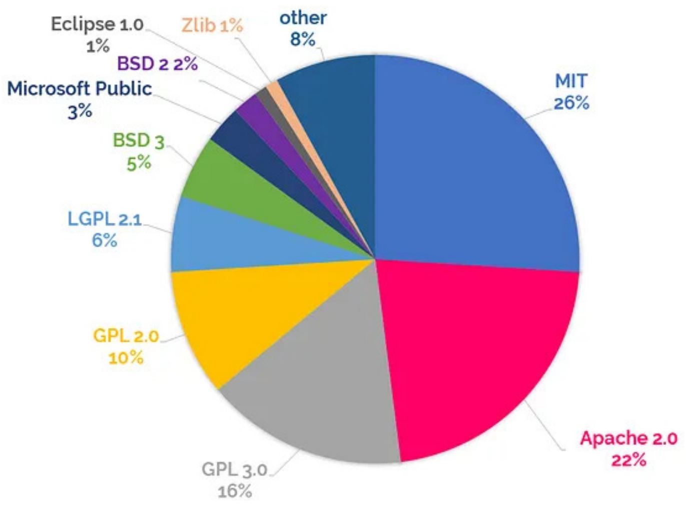


Figure 5.2 Top 10 Open Source Licenses in 2018 [63]

We will also look through the popular open-source licenses for the programming language python [65] according to the Open Source Initiative [66]. Their research focuses on more recent events, as they base their claims on the year 2023. Specifically, the dataset used for their research is a dataset created by the organization ClearlyDefined [67] made in September 2023. The dataset can also be conveniently aggregated by package managers such as npm [68], Nuget [69], Pypi [70] and Maven [71] for languages JavaScript, C#, Python and Java respectively.

They found that overall, the most popular licenses are again MIT and Apache 2.0. The licensing landscape varies among different package managers, with each programming language exhibiting distinct preferences for licenses within its ecosystem. For example, the JavaScript community commonly favors the MIT license, whereas Go developers tend to prefer Apache 2.0.

In Figure 5.3 we can see the most popular licenses for licensors for the Pypi package manager, which is used in unison with the Python programming language. The MIT and Apache 2.0 licenses are the most prevalent, representing 29.14% and 23.98% respectively. Newly, we can observe that the AGPLv3 license has appeared in the pie chart, contributing to roughly 2% of the pie chart. Compared to the pie chart in Figure 5.2, this graph lists unlicensed software components that make up a little under 24% of the chart.

5.6.2 Comparing licenses side by side

In Figure 5.1 we can see the main points we underlined in the previous sections between licenses. We can observe the main differences and similarities in the license terms in a true false table. We see that the permissions for commercial use, modification, and distribution are the same, and all the outlined licenses support these points. Since the MIT License is very brief, it, for example, excludes a patent grant, which the others support. The license also does not force the licensees to submit changes. We can also see which license is copyleft.

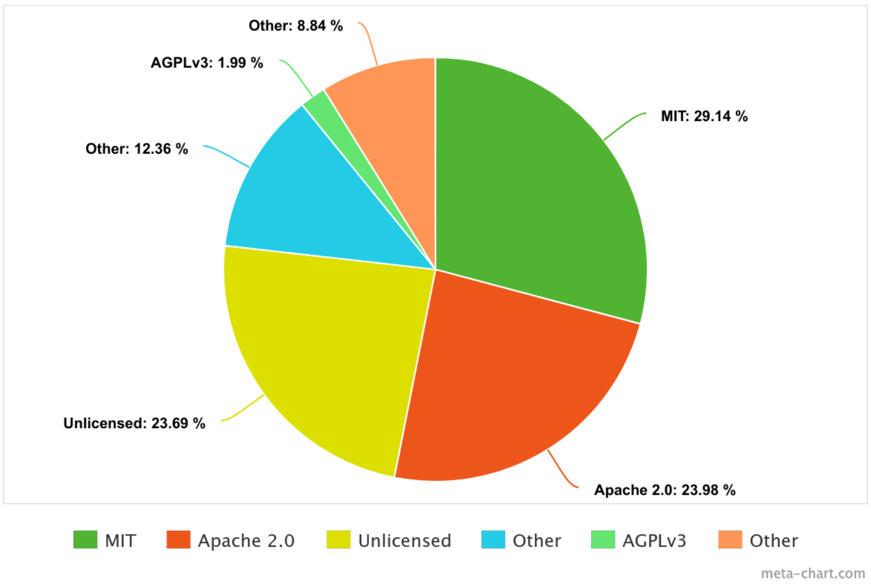


Figure 5.3 Top licenses for Pypi package manager for Python as of 2023 [65]. This is a remade graph we made using the tools from [72].

Table 5.1 Outline of license differences [73]

		MIT	Apache 2.0	GPLv3	AGPLv3
Permissions	Commercial Use	√	√	√	√
	Modify	√	√	√	√
	Distribute	√	√	√	√
	Patent Grant	×	√	√	√
Conditions	Copyright notice	√	√	√	√
	State Changes	×	√	√	√
	Same License	×	×	√	√
	Make Public if Distributing	×	×	√	√
	Network use is Distribution	×	×	×	√
Limits	Waives Warranty	√	√	√	√
	Waives Liability	√	√	√	√
	Trademark Restriction	×	√	×	×

Chapter 6

Implementation and Results

6.1 Dataset

The first thing we need to consider is the dataset; it will dictate the further steps we are going to take. All the datasets used contain videos recorded in a ping-pong hall owned by the Tipsport company, helping with narrowing down the input variability. Let us separate the data into two parts; the first part of the data we will use for training, and the second part we will use for inference in the production environment. It is also important to choose the training and testing datasets from a similar distribution [74]. For example, training a model to detect cats only on images of white cats with black background and then testing the model on images of black cats with white background is an example of choosing a different distribution, which will not lead to optimal results. Additionally, the distribution of the testing data set should closely reflect the production data. The camera, the ping-pong table, and the scorekeeper's table are expected to also remain at the same position and angle.

6.1.1 Issues

When we started working on this problem, an issue arose stemming from the company's procedures. We did not have data from the camera mounted in the ping-pong hall, as there were delays; this was by the company's partner firm, LIVEBOX. The Czech firm LIVEBOX, located in Brno, specializes in delivering video stream services to other businesses. As is often the case, the corporate world moves very slowly, which is also true at Tipsport. On the bright side, we have a splendid amount of ping-pong video data from a previously used camera, which was also used for training models not related to this thesis. There were also human-made annotations, but neither of them usable for our goal of detecting paddles and people. The problem with the video was that we wanted to minimize the difference between distributions for the training and production datasets. Considering that, we had two choices, either not doing anything or experimenting with the data we already have. We decided to use the data that are available to us.

6.1.2 Old videos

We got a hold of many long videos that contained many ping-pong matches, let us call these videos "old videos". The videos date back to the year 2018, this fact also reflects the camera parameters. This first parameter of the captured videos is its resolution; it is only a HD resolution (1280 × 720 pixels). This resolution is not the most cutting edge technology that was available at the time. Although the footage was taken in 2018, the camera must be dated back even further. To refer back to the picked model schema, the input shape for the YOLOv1 architecture is $448 \times 448 \times 3$ and the input shape for the YOLOv8 architecture is 640×640. This means that the input resolution will ultimately be resized to fit the input shape, therefore, increasing the video resolution will not necessarily lead to great improvements. The other important parameter is that the videos have a frame rate of 24 FPS, which is one of the reasons why the camera was replaced at the end. For the scene, the camera is positioned at a roughly 45 degree angle, which captured all the necessary components; paddle and paddles. The total length of all the old videos exceeded 6 hours. The videos commonly contain many segments in which no one is playing (for example, a game pause). In Figure 6.1 we can see an example frame taken from this dataset.

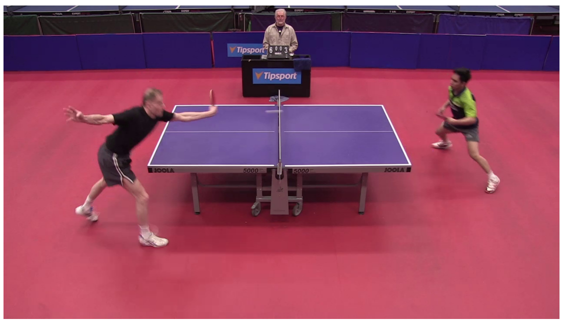


Figure 6.1 An example frame from an old video

6.1.3 New videos

The second set of videos came to us with a month-long delay after the planned date and was shot with a different camera. The delay was in part caused by the location of the external company, LIVEBOX. This is the company responsible for providing videos and ensuring the streaming of the production data. The videos, let us call them “new videos”, are of a higher quality, specifically the resolution is Full HD (1920×1080 pixels). This is a pleasant upgrade, however it is not too significant. The second parameter is the frame rate, which is a studding 120 FPS. This many frames per second are excessive for our usage. Although this would make the detected bounding boxes smoother, it comes at a cost of choosing a less accurate model, leading to worse overall results. It may seem that the only case where the higher frame rate would bring value is playing the footage in slow motion. Now we could apply a more accurate model since, for example, a 4 times slowed down video with FPS of 120 ends up being a 30 FPS video. The combined video length is roughly 2 hours and 53 minutes. Consisting of two videos, one spanning 2 hours and 11 minutes, and the second one spanning 42 minutes. In Figure 6.2, an illustrative frame is depicted from this dataset.

6.1.4 Production videos

The production videos will unfortunately be supplied after the deadline for this thesis, but we still have some basic information about the production videos. Note that these videos will be used purely for machine learning pipelines, so there is no concern for aesthetics. The videos will have 60 FPS with practically infinite supply, and the resolution will be fixed to Full HD. Although there is no video available from the camera, we have access to a frame taken from this camera as seen in Figure 6.3. The exposure time was specifically set to be very low in order to prevent the ball from blurring. In Figure 6.1 we can see an example frame taken from this dataset.

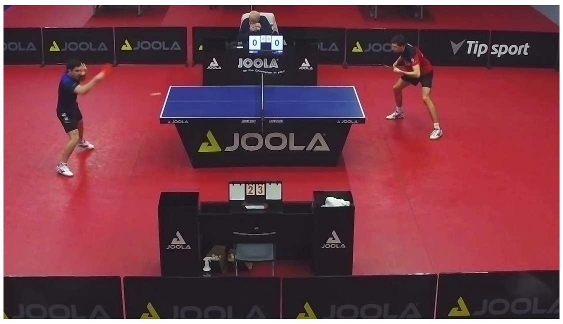


Figure 6.2 An example frame from a new video

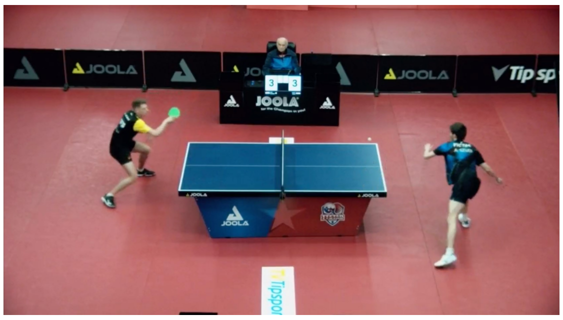


Figure 6.3 An example frame from the camera to be used in production

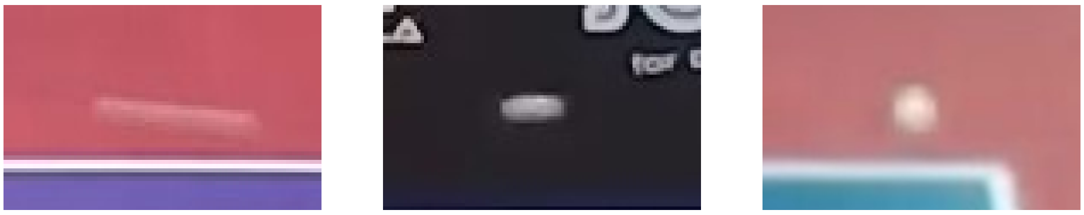
6.1.5 Exposure time

We have mentioned various datasets available to us with varying frame rates and exposure times. Exposure time refers to the duration during which a camera's sensors collect light to create an image. Having a camera that records video at high FPS is especially useful, since the camera can capture frames with a lower exposure time while retaining the frame quality. We can calculate the minimal exposure time given by the video FPS. The following equation calculates the minimal exposure time for a 120 FPS video:

$$\begin{aligned} \text{min\_exposure}(\text{frame\_rate}) &= \frac{1}{\text{frame\_rate}}\ \text{s}, \\ \text{min\_exposure}(120) &= \frac{1}{120}\ \text{s} = 8.\overline{3}\ \text{ms}. \end{aligned}$$

For our trained model to be able to accurately detect fast-moving objects such as paddles and balls, it is valuable to keep the exposure time as low as possible. The exposure time ought to also remain consistent across the training, evaluation, and test set, which reduces the dataset distribution variability. Figure 6.4 shows three images in which we can see the difference between high, low, and very low exposure times and the impact that it has on the blurriness of a quickly moving ball. The image on the left was captured on a camera with 24 FPS and the one on the right was captured on a camera with 60 FPS. The image in the middle has 120 FPS, and we can see that even though the camera has higher FPS, this does not necessarily mean that the exposure time is going to be lower. The image on the right, originating from the production dataset, has the lowest exposure time of all.

6.2 Annotation



(a) High exposure time **(b)** Low exposure time **(c)** Very low exposure time

Figure 6.4 Comparison of low and high exposure time

6.2.1 Defining what to annotate

The assignment of this thesis defined that we need to create a model that detects paddles and players by drawing bounding boxes around them. That is our minimum, which we need to accomplish. Frankly, it would also be noteworthy to be able to differentiate between the score-keeper (or referee) and a player. Detecting people in a video is one of the most popular tasks in object detection [75], so there are many datasets and models that can do exactly this, which would not be too impressive. Eventually, we have decided that we want to differentiate between scorekeepers and players. Combining sports analysis and computer vision, the pursuit of accurately detecting balls stands out as a particularly promising and fruitful endeavor. To increase the utility of the resulting model, we have added ball detection as our last annotation object. The following list shows all of the labels that we are going to focus on:

- paddle,
- player,
- ball (additionally),
- scorekeeper (additionally).

6.2.2 Looking for available datasets

Unfortunately, the datasets provided to us do not contain labels. Before we can proceed to the training step, we need to create annotations for our object detection task. Typically, assignments on popular machine learning sites such as Kaggle [76] provide labeled datasets. However, such labels are often not readily available for projects aimed at developing bespoke commercial solutions. Before delving into manual annotating, we will search the Internet for possible options.

First, we came across Roboflow [77]. They offer a comprehensive set of labeled datasets that are available for anyone to download. Each dataset is contributed by a user with a Roboflow account and is free to choose the license they desire. When searching for ping-pong datasets, Roboflow displayed a plethora of labeled frames, varying in camera position, camera angle, environment, labels, and lighting. For example, the “ping-pong ImageNet” dataset contains 837 images with varying ping-pong images ranging from professional ping-pong games to casual matches with friends and family. Two issues here are that the camera angle keeps on changing (we do not need a general model) and that the dataset labels only ball bounding boxes. Another dataset from the Roboflow user “pingpang” [78], contains ping-pong paddles, which is what we need. There is an absence of detecting bounding boxes around people, but that could be fixed with using a pre-trained detecting people and creating a branched off dataset. However, the tables in the images are not consistent, and the angle of the video is from the back of one player, these features do not make the dataset a viable option. Moreover, the attached license is not open source. Eventually, we ended our research by concluding that there is no available Roboflow dataset that meets our needs.

Looking through many other sites, we discovered that there were not any suitable datasets for detecting paddles and players. Most of the dataset solutions focus purely on ping-pong balls. We were now ready to create the label data ourselves.

6.2.3 Choosing a labeling environment

Firstly, we have looked for tools that would help us to create annotations, starting with Landing AI [79] created by the researcher and university professor Andrew NG. Landing AI offers a variety of tools to help with the annotation process. It provides advanced machine learning algorithms that can automate part of the annotation process, significantly reducing the manual effort required and increasing the speed of dataset preparation. These tools are particularly beneficial for projects involving large volumes of data, where manual annotation would be time-consuming and potentially error-prone. Moreover, Landing AI supports a variety of data types, including images, videos, and text, allowing for versatility in research applications. The platform also includes features for quality control and collaboration, enabling teams to work together effectively and maintain high standards of data integrity. We have tested the service and found the results to be satisfactory. All it took was annotating 10 videos and then letting a hidden object detection neural network train on this small set of data. The result was a model that we could use via an Application Programming Interface (API) they provide. The model was notably able to detect paddles and people with an acceptable visual margin of error. Additionally, one of the downsides was that the internal model used must have been very complex since the inference speeds were not capable of running in real time. Excluding the last fact, this process could easily be used to pre-annotate or even annotate data used in our training. The only issue was that the service is not free, they provide a few plans based on the company size. Our company falls into the most expensive plan, the price would be decided upon emails and meetings between the two companies, which again, could easily reach high amounts. This was one of the reasons why we decided not to use this technology to help us annotate data.

There are several other services that offer comparable functionality, including Roboflow Annotate [80] but none of them offer a free service. So we had no choice but to pivot to annotating our data manually.

We looked through various pages comparing free programs for annotating, like [81], where they compare proprietary and open-source software. Among the open-source software, the most popular one is Label Studio [82]. The license attached to this code is Apache License 2.0, which is a permissive open-source license that protects the licensee. This is good news since we can use the app for commercial purposes. One of the benefits of this service is that they provide source code, enabling any company to host this service within their internal systems. For example, in our company, we can simply put this service on a private URL, allowing employees to collaborate on annotating needed data. The service is shared across many users, allowing for simultaneous work on the same dataset. This is one of the things that will be implemented in our company, but as with anything in the corporate world, that will take some time, which prompts us to create only a local environment. The Label Studio software allows for easy deployment via the platform Docker [83], which encapsulates the application and its dependencies in a consistent environment across different systems. This approach enhances flexibility and scalability, enabling quick setups and maintenance, while ensuring robust security through isolation from other applications. The extensive ecosystem of Docker also allows for easy integration of additional tools and services, making it an ideal solution for machine learning projects that demand rapid deployment and scalability. As described within the Label Studio source code, there is one Command Line Interface (CLI) command that creates a Docker-encapsulated environment showing a demo of the Label Studio program. We quickly find that the software allows for many labeling tasks ranging from image classification to text annotation, and audio transcription, catering to a broad spectrum of data labeling needs across different types of data. The software is intuitive and easy to use and supports a variety of output formats such as the YOLO format [84].

6.2.4 Defining rules

The following list outlines the annotation rules used in training.

- **Rules for all classes**:
  - Annotate if most of the object is visible.
  - Enclose the object as tightly as possible.
  - Annotate only visible parts.
- **Class-specific rules**
  - **paddle**:
    * If a hand covers the paddle, annotate the hand as well (for example, the hand covers the whole handle).
  - **player**:
    * Exclude paddles from the player's bounding box if possible.
  - **ball**:
    * Annotate the ball only when it is in the air.
    * Fixed size of the bounding box; its width will be approximately 35% of the image width and height 25% of the image height. The ball must be in the center of the underlying bounding box.
  - **scorekeeper**:
    * No additional rules.

We chose to annotate objects only when the majority of the object is visible, this should reduce the models amount of false positives. We also want to enclose objects as tightly as possible. Creating a bounding box that includes unnecessary space or does not encompass the entire object will increase the inaccuracy of the IOU metric. Even a well-performing model may penalize itself for failing to predict regions where gaps were left during labeling [85]. And finally, we are going to annotate only visible parts of objects, so we do not force the model to associate the target object with a background. For example, if the legs of a player are hidden behind a table, we annotate only the visible parts, even though we can safely assume that his legs must be behind the table.

If a hand covers a paddle, for example, its handle, we want to annotate the whole paddle even though it is hidden. But if it is hidden behind other objects, we do not annotate it. A person's hand is a common object covering the paddle; we assume that our model will quickly learn this fact. For the player class, we are going to precisely outline his visible parts, excluding the paddle even though a hand is hidden behind him. There is no particular reasoning behind this, other than ensuring consistency throughout annotations.

In addition, while researching the best annotation methods [86], we found that YOLO architectures have trouble detecting smaller objects, those represented by a few pixels. To avoid this issue, we made the decision that we would detect the ball class with its background. The model will need more training data to learn the possible backgrounds, but eventually, this will increase the model's performance on the ball class. Also, we decided to fix the width and height of the ball bounding box to be 25% and 35% to the image width and height, respectively. This will make it easier for the model to specialize for the ball class. For completeness, the actual ball in the image has to be in the very center of the bounding box.

Defining the rules for annotating a single model is crucial to optimize its learning capabilities especially when there are multiple annotators participating in the development of the model.

6.2.5 Annotation Setup

We choose to create an environment using a `docker-compose.yaml` file instead of a simple one-line command in Docker. This approach allows for a concise definition of environmental variables, local ports, permissions, image version, and path to where the program will save files on our local machine. For clarity, environmental variables are variables, analogous to variables in programming, passed to a process on its initialization. Docker image is a read-only template that contains instructions for creating a Docker container.

Additionally, we also modified the `.env` file describing the keyboard shortcuts we will use to annotate. The default shortcuts were too difficult to press, combining letters with `shift`, `ctrl`, and others. The new shortcuts are processed with a single button press.

6.2.6 Annotation Results

Using the old video, we have trimmed the first minute. We only annotated players and paddles and ended up with approximately 1500 labeled frames. For fine-tuning (training an already pre-trained model), it is usually sufficient to have around 1000 frames [87]. The second video we annotated was a subsection from the 2-hour video (mentioned earlier). We have trimmed the video from the first to the second image, which makes for $120 \times 60 = 7200$ frames, which is way too many to annotate. Additionally, because this is 120 frames per second video, similar frames add less value to the training dataset, so we decided to pick every fourth frame instead, making the total number of frames $7200/4 = 1800$. We have annotated all the 1800 frames, this time with all the 4 target classes: players, people, ball, and scorekeeper. The third created dataset was from the 42-minute video, where we randomly selected 1000 throughout its full duration, making sure that all frames were unique. This process increases the variability in the frames, making the dataset more valuable for the training process.

Labeling images with a random sequence of frames is more difficult because we cannot utilize the interpolation feature in Label Studio [88], forcing us to annotate in the Images mode instead of the Video mode. This also changes the format in which we need to supply frames to the software.

During annotation, we estimate the average speed of making accurate labels to be roughly 150 frames per hour. Considering the total number of annotated frames to be over $1500+1800+1000 = 4300$, the time it took to purely label images was analogously $4300/150 = 28.\overline{6}$ hours.

6.3 Data manipulation

To help transform data into various formats, we have created Python scripts, utilizing popular libraries like pandas [89] and numpy [90].

First, we needed to trim our videos, convert them from Transport Stream (ts) to MPEG-4 Part 14 (mp4) format, and be able to change FPS. To achieve that, we opt for the open-source HandBrake software [91]. The benefits of this software are its intuitive use, open-source nature, and wide support for the operating system. However, when converting a 120 FPS video to a 30 FPS video, we find that the video is becoming blurrier. Handbrake software automatically applies interpolation when converting a higher frame rate video to a lower frame rate one, this option cannot be disabled. This shows the program's inflexibility and makes it a poor match for our use case. We have also tried the versatile tool ffmpeg [92], but despite our efforts, we were unable to maximize the potential of the tool due to compatibility issues. We have used this tool only for trimming videos. Eventually, we decided to write our own script for the following tasks: video trimming, choosing the nth frame, changing speed, and grabbing frames at random. We are using the Python library called OpenCV [93] for most of our operations. By reading the input video at a specific start to the specified end and saving the frames to an output video, we achieve the trimming operation. Choosing the nth frame is done by the modulo operation in Python, allowing us to omit other frames in the output. The video speed depends on the output frames per second and the actual output video frames per second. The core of grabbing frames at random is the function originating from the numpy module, `numpy.random.choice`, allowing to choose a distinct set of numbers from a range of integers.

Part of the data manipulation tools is also a Python script that merges two datasets in a YOLO format together, making a combined one. No external libraries were needed for this process, only stock ones that come with the programming language. The script copies one dataset and appends the second one with different properties, so it is compatible with the YOLO format. Before we allow the script to copy datasets, it ensures that all classes of the datasets are the same, including their arrangements, this information is accessible in the `classes.txt` file. If the right conditions are not met, the process fails notifying the user of the script. There was a situation where we wanted to merge two datasets, but they were not compatible. The class order did not match, although the classes were identical. One dataset needed to be reconfigured. This gave rise to another utility, which focuses on reindexing classes in all prediction files inside the designated dataset in the YOLO format.

The Label Studio software returns different outputs depending on what mode we used to annotate. If we annotate in the images mode, we can export the annotations using the YOLO format. However, using the video mode, the only viable option is the Label Studio export option. This fact introduces a further complication, necessitating the need for a procedure to convert between the two formats. Particularly, we needed to deploy a script to convert annotations from a Label Studio JSON-MIN format to a YOLO format. The input format is a JSON file that contains frame numbers and their class labels. The script extracts these labels along with a frame from the video and creates two files, a label file and an image file.

6.4 Pre-annotation

The annotation process alone is still tedious, we looked into ways that would speed it up. That is where pre-annotation comes in. This technique uses a pre-trained model to create annotations before a real person goes to manually label the images. Pre-annotation frees the annotators from creating all the annotations; they will create fewer new bounding boxes, adjust some predictions, and leave the rest of the correct predictions untouched. This process also gives the annotator feedback, telling him what images the model struggles with and which ones he does not. This can then be used to focus on the problematic images, speeding up the training process. Label Studio natively supports this feature only in image labeling mode. To pre-annotate, we need to add unlabeled data, and then pass a JSON file with bounding box predictions from an object detection model of our choice. The JSON format for Label Studio pre-annotations is mentioned here [94]. To create the pre-annotations, we were required to create yet another script to handle this. It loads a specified model, extracts its predictions, and returns them to an output file in the outlined format.

When we applied pre-annotations for unseen data, we noted that the annotation process got approximately 1.4x faster. This improvement benefits other annotators in future projects.

6.5 Training

To train our YOLOv8 [28] model, we utilized the Ultralytics module in Python. The company Ultralytics [47] located in Los Angeles developed this module and even introduced the YOLOv8 architecture. Their software is well-known for its robust support and wide recognition within the machine-learning community, making it an ideal choice for implementing advanced object detection systems. Ultralytics facilitates seamless integration with various programming environments, enhancing the adaptability and efficiency of our development process.

The module's compatibility with numerous data formats and its comprehensive pre-built functionalities accelerate the model training and evaluation phases. It simplifies complex procedures such as model tuning and real-time data processing, which are critical for optimizing the performance of YOLOv8. Furthermore, Ultralytics provides extensive documentation and a supportive community, offering valuable resources that aid in troubleshooting and refining model configurations. This combination of ease of use and powerful capabilities makes the Ultralytics module a cornerstone in our approach to deploying a highly effective YOLOv8 model.

We have used these parameters as default values:

- batch size of 16,
- weight decay of 1e-5,
- If the number of training iterations is smaller than 10000 (number of iterations is equal to the number of epochs times the number of training images divided my batch size), Adam with weight decay (AdamW) [95] optimizer will be used with momentum 0.9 and learning rate 0.00125.
- If the number of training iterations is larger than 10000, Stochastic Gradient Descent (SGD) [96] optimizer with weight decay will be used with momentum 0.9 and learning rate 0.1,
- If 50 epochs show no improvement, the early stopping technique stops the training.
- Automatic Mixed Precision (AMP) [97] is used, which uses mixed float precision for some operations to speed up linear and convolutional operations during training.

Additionally, we have done all the training on Tesla v100 Graphics Processing Unit (GPU)

First, we annotated a dataset that comes from the old video. We decided to take one continuous minute of the 24 Frames Per Second (FPS) video, containing over 1400 frames. In this dataset, we annotated two classes: paddles and people. We trimmed the video to the part where players had a match. The score of the ping-pong battle starts at 0:0 and ends at 1:2. The scene contains two people wearing mostly black clothes, both wearing white shoes. There is an unsuccessful serving attempt, and a few times ball hits the ground.

We chose a `yolov8n` (nano) model, it has an input size of $640 \times 640 \times 3$ and performs 37.3 mAP@[.50:.05:.95] on the COCO val 2017 dataset [11], specified in the Ultralytics source code repository [47]. The speed of this model is one of its main advantages; it offers a speed that is right under 1 ms inference on the A100 TensorRT GPU. We used this model by connecting via a remote desktop program. We have trained the model on the mentioned data for 10, 100, and 1000 epochs. While training 1000 epochs, we stopped at epoch 775 due to the early stopping regulation technique. On the Tesla V100, the training took 30 seconds, 0.45 hours, and 3.3 hours, respectively. We used the 80/20 train/test split; this proportion leaves the majority of data for training, while leaving sufficient amount of images in the other set.

Next, we took a new video of length of 2 hours. The video is at 120 FPS, making the frames very close to each other. We made a script that trims the video and takes every fourth frame, which resulted in a minute long video (first to second minute) at 30 FPS. From now on, we started focusing on all four classes: paddle, player, ball, and scorekeeper. After annotating 1800 images manually, we used the `yolov8n` (nano) model to train on 10, 100, and 1000 epochs, where the last 1000 epochs got cut short to 820 epochs because of early stopping. Training with 822 epochs took 6.4 hours. We also used `yolov8l` (large) modification for 20 epochs. We kept the train, test split at 80/20 ratio.

Following that, we used a 42-minute new video and randomly took 1000 and 100 distinct frames using data tools we had developed. We used the larger dataset for training and the smaller one for testing. We then ran 1000 epochs, which got interrupted at 222 because there was no progress in the last 50 epochs. The model we used was `yolov8x` (extra-large).

For the preprocessing procedure we have also used the Mosaic augmentation method [98]. The Figure 6.5 shows four post processed inputs that we used in training.

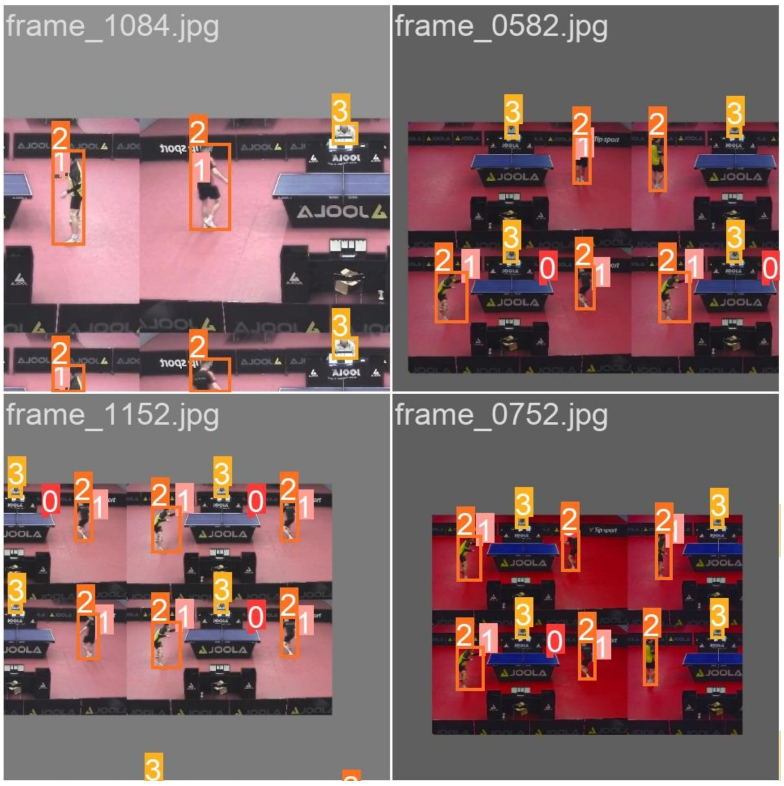
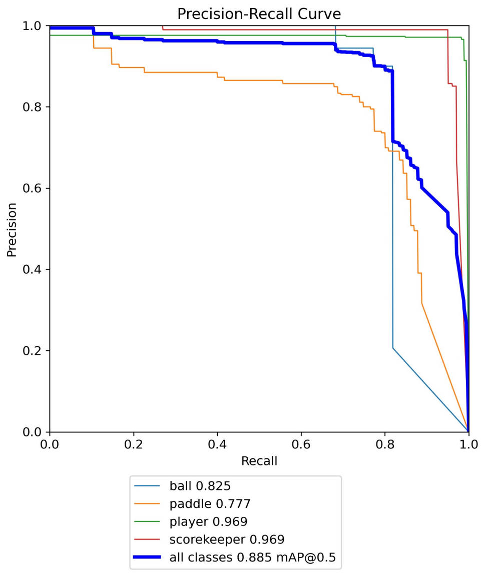


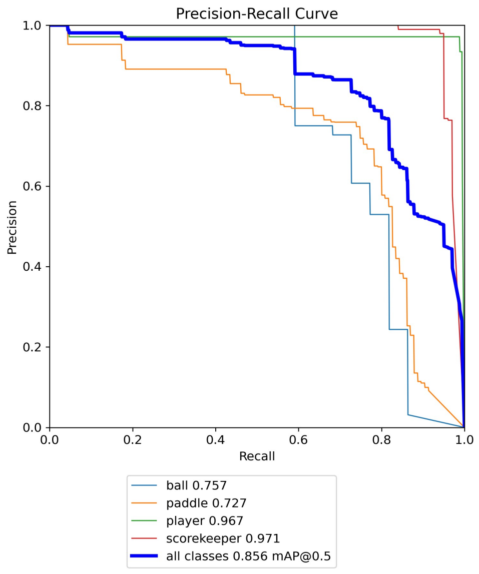
Figure 6.5 Mosaic augmentation during training

6.6 Displaying the results

Our most precise models using the architectures YOLOv8m (medium) and YOLOv8x were trained on 150 and 222 epochs, respectively. We provided both models with the same data: 1000 randomly selected frames from the 42-minute video, and as testing data, we provided 100 different frames from the same video. In Figure 6.6, we can see both model's precision compared for each class using the Precision-Recall (PR) curve. Both models have no issues learning where players are, but they struggle to identify paddles. For the larger and more complex model YOLOv8x, we can see that the mAP@0.5 precision for all classes is a bit higher than that of YOLOv8m. The legends show the recall values for each class separately for mAP@0.5. Interestingly, the smaller model is negligibly better at predicting the scorekeeper class. Overall, it is essential to have numerical results, but the final choice should also be decided upon by visualization of the results on a video.



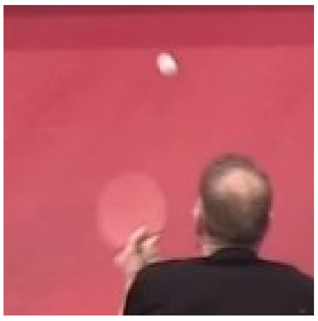
(a) PR curve for YOLOv8x



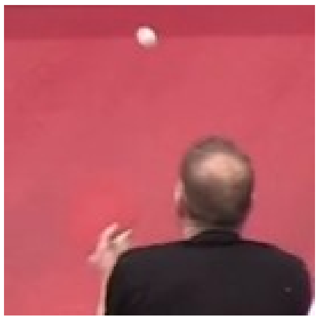
(b) PR curve for YOLOv8m

Figure 6.6 Precision-Recall curves for YOLOv8x and YOLOv8m

The image in Figure 6.7 shows that the paddle seems to disappear entirely in some frames. That makes it difficult for even an ordinary human to determine the situation. This problem is caused by the ground color, imperfect lighting, and a slight Joint Photographic Experts Group (JPEG) compression.



(a)

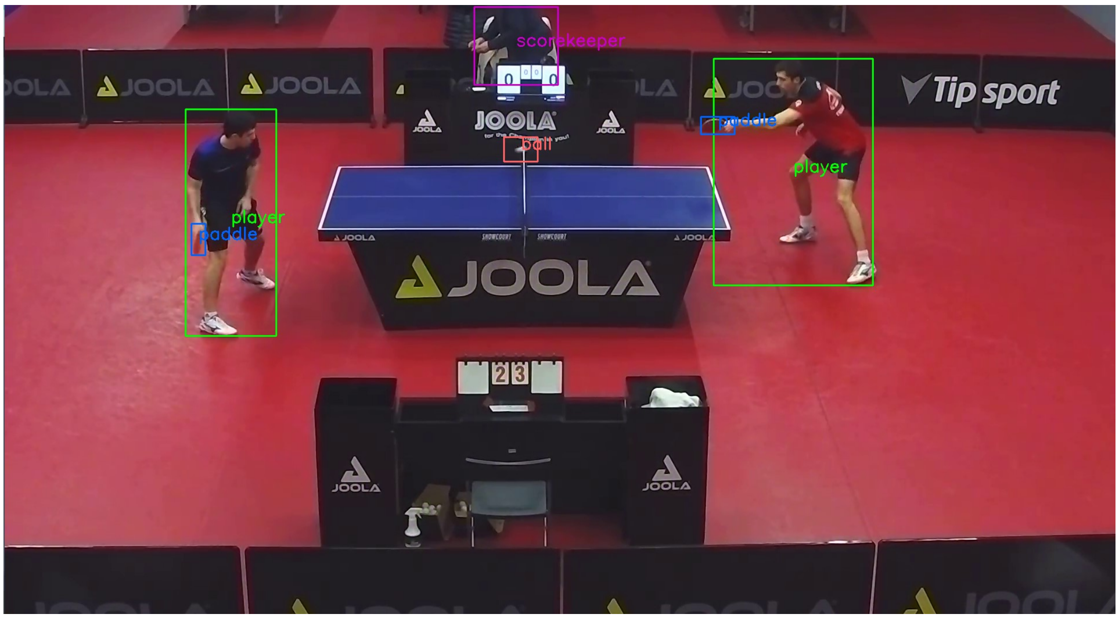


(b)

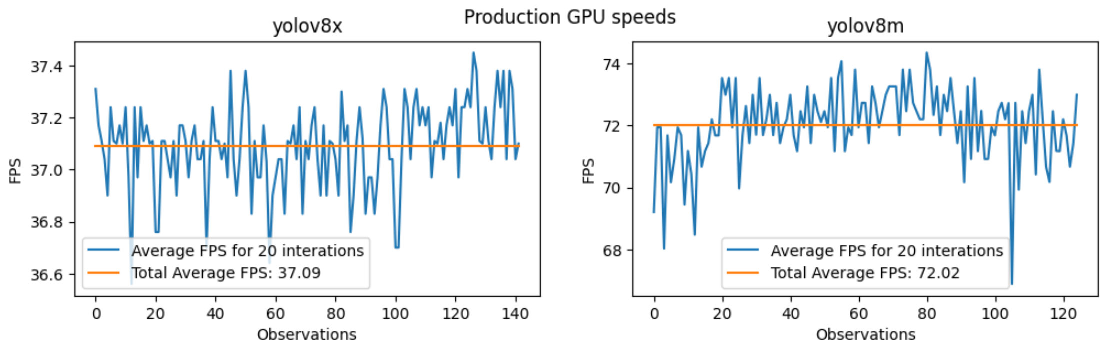
Figure 6.7 Disappearing paddle

After running the benchmarks we achieved Frames Per Second (FPS) of 37 on production (Graphics Processing Unit) GPU, with visually pleasing results as seen in Figure 6.8. The FPS is the inverted value of the average time of inference for 20 iterations.

We also used the same setup to train a smaller model, YOLOv8m (medium). The inference speed has more than doubled; we compare the inference speeds between these two models in the Figure 6.9. The YOLOv8m (medium) converged in 155 epochs.



■ **Figure 6.8** Result on YOLOv8x with 222 epochs



■ **Figure 6.9** Average FPS on production GPU

6.7 Experimentation

We are aware of the high demand for detecting balls in sports [99]. We have decided to look at the games from the 42-minute new video. We have applied the YOLOv8n model we trained for 10 epochs as it offers the balance of iteration speed and precision. Due to the small epoch size, the small architecture does not experience overfitting. We predicted the ball on 42 minutes of a 120 (Frames Per Second) FPS video, that is over three hundred thousand frames that the model processed. This many annotations would take me a very long time. Using the previous finding of the time it took me to annotate 150 annotations using the pre-annotation method, the total time would be 2000 hours or 83 days non-stop. We saved the midpoint of the bounding box, the confidence associated with it, and the frame number. We saved the predictions to a Comma-Separated Values (CSV) file.

First, we want to see the frequency of ball detections throughout the 42-minute video. To achieve this, we use a histogram, where we count the value of confidence scores. We use floating point number certainty instead of 0 or 1 values to draw a more telling histogram. In Figure 6.10, we can see the ball detection histogram with 1000 bins.

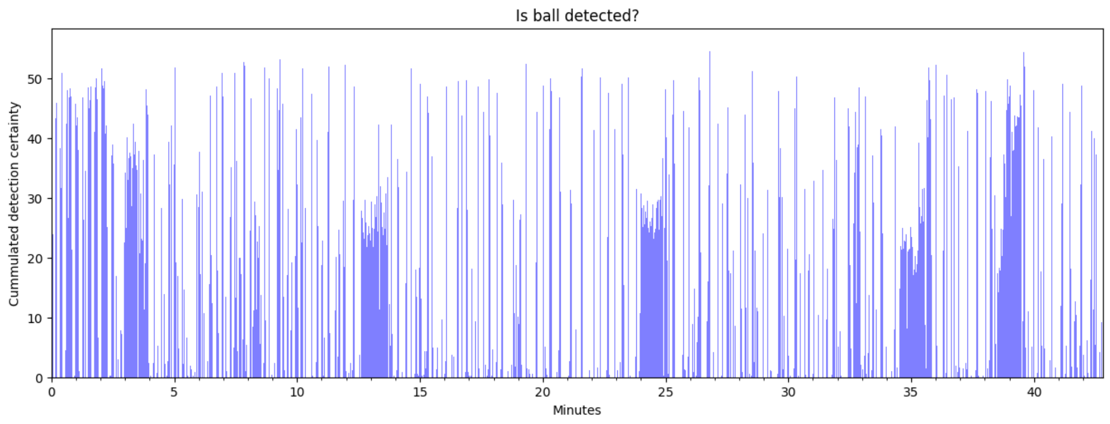


Figure 6.10 Ball detection histogram for trained model YOLOv8n on 10 epochs

The figure shows relatively random ball detections except for the five clustered regions. Since there were many ping-pong games throughout the video, we decided to investigate these regions. We look into frames in these regions; in Figure 6.11, we can see an example of such a frame.

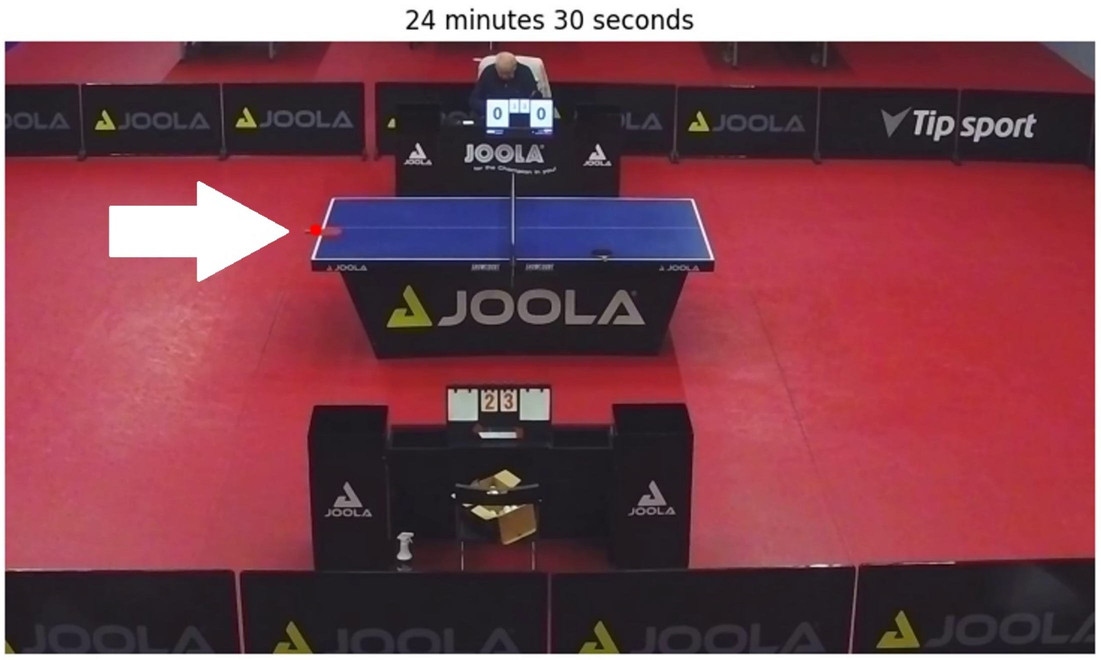


Figure 6.11 Incorrectly classified red side of A paddle facing up on the table

In all the regions we can observe that the detections are caused by a red side of a paddle facing up on the table. This situation is not included in the training dataset, as the dataset contains only one minute of the 2-hour video, with constant ball movement. In Figure 6.11 we can see one such frame, the large white arrow points at the point, where the prediction was evaluated, the red dot on the paddle is the precise predicted position. We combated these issues using our dataset manipulation technique of randomly extracting frames from our videos, which offers larger frame diversification.

It would also be nice to see what positions were frequent and which were not. One solution would be to create a 2D matrix correlating with the video resolution and set all values in the radius of ball positions to one. We include such visualization only in our practical part for briefness. Another solution is that instead of creating a 2D matrix of the same size as the video resolution, we would make a 2D histogram of a smaller size. We would add 1 to all the bins that a ball position falls into. In the next step, we would normalize the matrix values and display the heatmap. But by doing this, we will be overwhelmed with the clustered values previously seen in Figure 6.10. To combat this, we can use a logarithmic scale. The Figure 6.12 demonstrates the 2D histogram of ball positions throughout the 42-minute video. We can notice the most frequent points are the edges of the table where the paddles were lying.

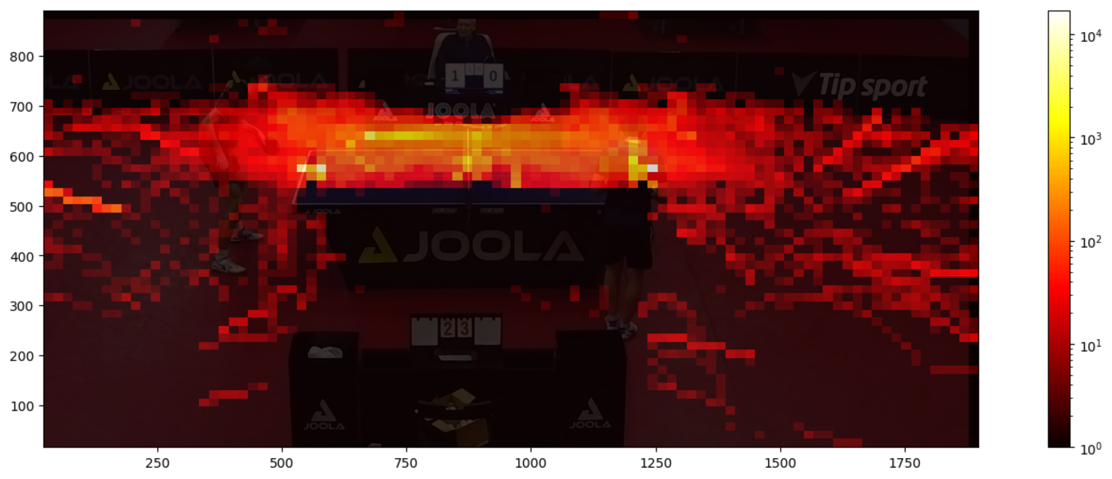


Figure 6.12 2D histogram heatmap of ball positions in a 42-minute video

We can also clip the larger values with appropriate parameters instead. For this, we used a Gaussian kernel to approximate the ball detection for a smoother heatmap; this can be seen in Figure 6.13

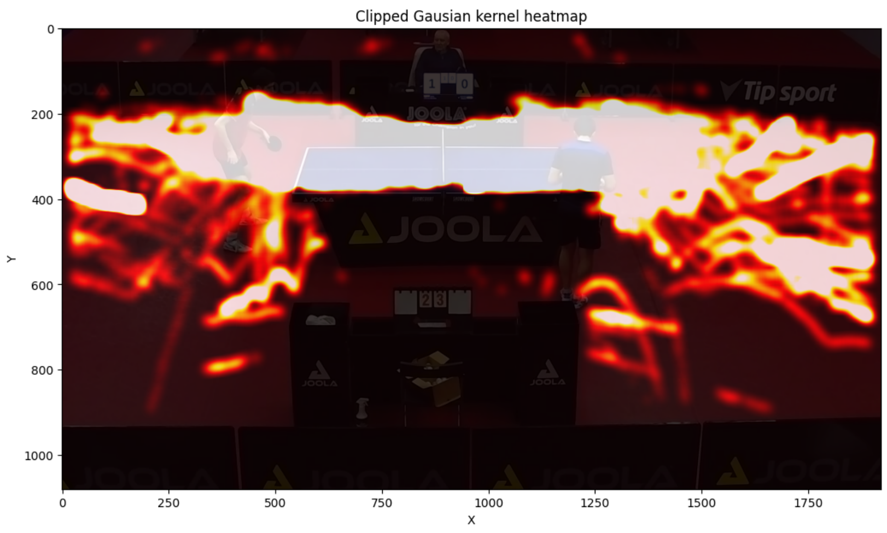


Figure 6.13 Gaussian heatmap of ball positions in a 42-minute video

6.7.1 Kalman filter

When looking at our model predictions on a continuous video, we notice that sometimes, the model does not make any predictions. When watching the video, the ball positions were flickering at times. We looked into the Kalman filter [100], which offers protection from noise and, importantly, allows us to calculate/predict the future position of an object if the present position is not known. This technique is often used in the Global Positioning System (GPS). We have included the implementation in the repository attached to this thesis. When the model does not predict ball detections, we allow the Kalman filter to determine the ball's position for four frames. We also set positive vertical velocity to simulate gravitation. This approach could be improved since false negative detections destabilize the Kalman filter.

6.7.2 Benchmark of SSD

The Single-Shot Detector (SSD) [19] is not particularly impressive in terms of speed and accuracy, but it offers a good balance between those two. Compared to the newest You Only Look Once (YOLO) architectures, it looses in both speed and accuracy departments. Nevertheless, we have tried measuring the speed of the SSD with input size 300×300 and the VGG-16 [101] backbone without fine-tuning using PyTorch library. VGG16 is a convolutional neural network model originally developed for the task of image recognition and classification. It was introduced by the Visual Geometry Group (VGG) [16] from the University of Oxford in 2014, and it became popular because of its simplicity and strong performance on the ImageNet challenge, a large-scale visual recognition competition. The number 16 signifies the number of layers of the backbone.

We run inference on the production Graphics Card Unit (GPU), and as seen in Figure 6.14, the average speed is just a bit lower than our minimal threshold of 30 Frames Per Second (FPS). The slow prediction speed would be a bottleneck when used in production. We know that other processes also take time when running inference. Some of these processes are loading the video stream and the time to send the data elsewhere. The additional functionalities would require us to make a multi-process application, complicating the development unnecessarily. For these reasons, we decided not to pursue this path further.

In addition, we also include a file that focuses on manual calculation of Average Precision (AP) metric and the PR curve in object detection. It describes how to calculate true positives, false positives in object detection and using that we can determine the score of a prediction or set of predictions.

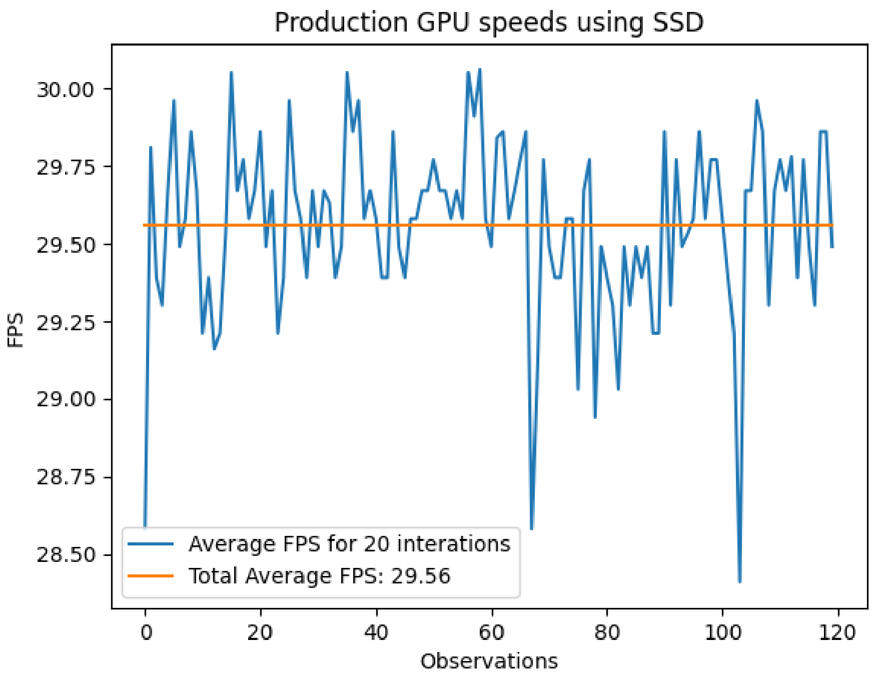


Figure 6.14 Average FPS on production GPU using SSD

Chapter 7

Conclusion

Our research presents a dataset comprising two videos; the first video's length is 42 minutes, and the second is over 2 hours. Both videos are 120 frames per second (FPS), corresponding to over 302400 and 864000 frames, respectively. To annotate for object detection, we used fitting algorithms to select frames of interest. We used the open-source software Label Studio [82], suitable for commercial use, set it up appropriately, and labeled extracted images. We settled on the following detection classes: person, paddle, ball, scorekeeper. We researched to check available datasets online to find that none of them particularly suited our needs and proceeded to annotate our data, which succumbed to rules that we had defined for the optimal learning process. The annotation process comprises our manual labeling and assisted labeling, which consists of pre-annotation using a trained model on manually labeled data following manual correction.

This solution accompanies many preprocessing and postprocessing tools ranging from algorithms for picking ideal frames to video conversion. Since our labeling software has a specific output format, there also was a need to implement an annotation conversion tool. We researched the most relevant approaches to our problem by comparing and experimenting with various model versions, such as "medium" and "large".

We used a fine-tuning method for a machine learning model from the YOLO line-up, precisely YOLOv8, developed by the authors of Ultralytics [47].

Achievement of our work is a robust model trained on custom data we created detecting a player, paddle, ball, and the scorekeeper in a ping-pong match in real time running on our hardware A100 TensorRT with an average FPS of 72.

Bibliography

1. BETTS, Rebecca; DIERKING, I. Machine learning classification of polar sub-phases in liquid crystal MHPOBC. *Soft Matter*. 2023, vol. 19. Available from DOI: 10.1039/D3SM00902E.
2. IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, vol. abs/1502.03167. Available from arXiv: 1502.03167.
3. *How Can I Create A Precision-Recall Curve In Python?* [Online]. 2024. Available also from: https://scales.arabpsychology.com/stats/how-can-i-create-a-precision-recall-curve-in-python/. Accessed on 05/14/2024.
4. *mAP (mean Average Precision) for Object Detection | by Jonathan Hui | Medium* [online]. 2018. Available also from: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173. Accessed on 05/09/2024.
5. BOCHKOVSKIY, Alexey; WANG, Chien-Yao; LIAO, Hong-Yuan Mark. YOLOv4: Optimal Speed and Accuracy of Object Detection. *CoRR*. 2020, vol. abs/2004.10934. Available from arXiv: 2004.10934.
6. *mosaic_10.jpg (224×224)* [online]. 2024. Available also from: https://www.ccoderun.ca/darkmark/mosaic_10.jpg. Accessed on 05/09/2024.
7. PAZHOOHI, Farid; KINGSTONE, Alan. The Effect of Movie Frame Rate on Viewer Preference: An Eye Tracking Study. *Augmented Human Research*. 2021, vol. 6. Available from DOI: 10.1007/s41133-020-00040-0.
8. *Understanding FPS Values: The Advanced Guide to Video Frame Rates* [online]. 2024. Available also from: https://www.dacast.com/blog/frame-rate-fps. Accessed on 04/302024.
9. JANZEN, Benjamin; TEATHER, Robert. Is 60 FPS better than 30? The impact of frame rate and latency on moving target selection. *Conference on Human Factors in Computing Systems - Proceedings*. 2014. ISBN 978-1-4503-2474-8. Available from DOI: 10.1145/2559206.2581214.
10. *Object Detection | Papers With Code* [online]. 2024. Available also from: https://paperswithcode.com/task/object-detection. Accessed on 04/302024.
11. LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge J.; BOURDEV, Lubomir D.; GIRSHICK, Ross B.; HAYS, James; PERONA, Pietro; RAMANAN, Deva; DOLLÁR, Piotr; ZITNICK, C. Lawrence. Microsoft COCO: Common Objects in Context. *CoRR*. 2014, vol. abs/1405.0312. Available from arXiv: 1405.0312.

12. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007)* [online]. 2007. Available also from: `http://host.robots.ox.ac.uk:8080/pascal/VOC/voc2007/`. Accessed on 04/302024.

13. DENG, Jia; DONG, Wei; SOCHER, Richard; LI, Li-Jia; LI, Kai; FEI-FEI, Li. ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. Available from DOI: `10.1109/CVPR.2009.5206848`.

14. REN, Shaoqing; HE, Kaiming; GIRSHICK, Ross B.; SUN, Jian. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*. 2015, vol. abs/1506.01497. Available from arXiv: `1506.01497`.

15. GIRSHICK, Ross B. Fast R-CNN. *CoRR*. 2015, vol. abs/1504.08083. Available from arXiv: `1504.08083`.

16. *VGG Explained | Papers With Code* [online]. 2024. Available also from: `https://paperswithcode.com/method/vgg`. Accessed on 04/302024.

17. *ResNet Explained | Papers With Code* [online]. 2024. Available also from: `https://paperswithcode.com/method/resnet`. Accessed on 04/302024.

18. REDMON, Joseph; DIVVALA, Santosh Kumar; GIRSHICK, Ross B.; FARHADI, Ali. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*. 2015, vol. abs/1506.02640. Available from arXiv: `1506.02640`.

19. LIU, Wei; ANGUELOV, Dragomir; ERHAN, Dumitru; SZEGEDY, Christian; REED, Scott E.; FU, Cheng-Yang; BERG, Alexander C. SSD: Single Shot MultiBox Detector. *CoRR*. 2015, vol. abs/1512.02325. Available from arXiv: `1512.02325`.

20. ZHU, Yanzhao; YAN, Weiqi. Traffic sign recognition based on deep learning. *Multimedia Tools and Applications*. 2022, vol. 81, pp. 1–13. Available from DOI: `10.1007/s11042-022-12163-0`.

21. LIN, Tsung-Yi; GOYAL, Priya; GIRSHICK, Ross B.; HE, Kaiming; DOLLÁR, Piotr. Focal Loss for Dense Object Detection. *CoRR*. 2017, vol. abs/1708.02002. Available from arXiv: `1708.02002`.

22. REDMON, Joseph; FARHADI, Ali. YOLOv3: An Incremental Improvement. *CoRR*. 2018, vol. abs/1804.02767. Available from arXiv: `1804.02767`.

23. PRUSTY, Manas; TRIPATHI, Vaibhav; DUBEY, Anmol. A novel data augmentation approach for mask detection using deep transfer learning. *Intelligence-Based Medicine*. 2021, vol. 5, p. 100037. Available from DOI: `10.1016/j.ibmed.2021.100037`.

24. REDMON, Joseph; FARHADI, Ali. YOLO9000: Better, Faster, Stronger. *CoRR*. 2016, vol. abs/1612.08242. Available from arXiv: `1612.08242`.

25. ULTRALYTICS. *State-of-the-art real-time object detection system* [online]. 2024. Available also from: `https://github.com/ultralytics/yolov5`.

26. LI, Chuyi; LI, Lulu; JIANG, Hongliang; WENG, Kaiheng; GENG, Yifei; LI, Liang; KE, Zaidan; LI, Qingyuan; CHENG, Meng; NIE, Weiqiang; LI, Yiduo; ZHANG, Bo; LIANG, Yufei; ZHOU, Linyuan; XU, Xiaoming; CHU, Xiangxiang; WEI, Xiaoming; WEI, Xiaolin. *YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications*. 2022. Available from arXiv: `2209.02976 [cs.CV]`.

27. WANG, Chien-Yao; BOCHKOVSKIY, Alexey; LIAO, Hong-Yuan Mark. *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*. 2022. Available from arXiv: `2207.02696 [cs.CV]`.

28. REIS, Dillon; KUPEC, Jordan; HONG, Jacqueline; DAOUDI, Ahmad. *Real-Time Flying Object Detection with YOLOv8*. 2023. Available from arXiv: `2305.09972 [cs.CV]`.

29. WANG, Chien-Yao; YEH, I-Hau; LIAO, Hong-Yuan Mark. *YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information*. 2024. Available from arXiv: `2402.13616 [cs.CV]`.

30. *Object Detection Using OpenCV YOLO | Great Learning* [online]. 2024. Available also from: `https://www.mygreatlearning.com/blog/yolo-object-detection-using-opencv/`. Accessed on 04/22/2024.

31. *The history of YOLO: The origin of the YOLOv1 algorithm | SuperAnnotate* [online]. 2023. Available also from: `https://www.superannotate.com/blog/yolov1-algorithm`. Accessed on 04/23/2024.

32. SZEGEDY, Christian; LIU, Wei; JIA, Yangqing; SERMANET, Pierre; REED, Scott; ANGUELOV, Dragomir; ERHAN, Dumitru; VANHOUCKE, Vincent; RABINOVICH, Andrew. *Going Deeper with Convolutions*. 2014. Available from arXiv: `1409.4842 [cs.CV]`.

33. REDMON, Joseph. *Darknet: Open Source Neural Networks in C* [online]. 2016. Available also from: `http://pjreddie.com/darknet/`.

34. *Review: YOLOv1 — You Only Look Once (Object Detection) | by Sik-Ho Tsang | Towards Data Science* [online]. 2018. Available also from: `https://towardsdatascience.com/yolov1-you-only-look-once-object-detection-e1f3ffec8a89`. Accessed on 04/23/2024.

35. RUSSAKOVSKY, Olga; DENG, Jia; SU, Hao; KRAUSE, Jonathan; SATHEESH, Sanjeev; MA, Sean; HUANG, Zhiheng; KARPATHY, Andrej; KHOSLA, Aditya; BERNSTEIN, Michael; BERG, Alexander C.; FEI-FEI, Li. *ImageNet Large Scale Visual Recognition Challenge* [online]. 2015. Available also from: `http://image-net.org/challenges/LSVRC/2015/`.

36. REN, Shaoqing; HE, Kaiming; GIRSHICK, Ross B.; ZHANG, Xiangyu; SUN, Jian. Object Detection Networks on Convolutional Feature Maps. *CoRR*. 2015, vol. abs/1504.06066. Available from arXiv: `1504.06066`.

37. PAPAGEORGIOU, C. P.; OREN, M.; POGGIO, T. A general framework for object detection. In: *Computer Vision, 1998. Sixth International Conference on*. IEEE, 1998, pp. 555–562.

38. DALAL, Navneet; TRIGGS, Bill. Histograms of oriented gradients for human detection. In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. IEEE, 2005, vol. 1, pp. 886–893.

39. FELZENSZWALB, Pedro F.; GIRSHICK, Ross B.; MCALLESTER, David; RAMANAN, Deva. Object Detection with Discriminatively Trained Part-Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2010, vol. 32, no. 9, pp. 1627–1645. Available from DOI: `10.1109/TPAMI.2009.167`.

40. GIRSHICK, Ross; DONAHUE, Jeff; DARRELL, Trevor; MALIK, Jitendra. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. Available from arXiv: `1311.2524 [cs.CV]`.

41. UIJLINGS, Jasper; SANDE, K.; GEVERS, T.; SMEULDERS, A.W.M. Selective Search for Object Recognition. *International Journal of Computer Vision*. 2013, vol. 104, pp. 154–171. Available from DOI: `10.1007/s11263-013-0620-5`.

42. HEARST, M.A.; DUMAIS, S.T.; OSUNA, E.; PLATT, J.; SCHOLKOPF, B. Support vector machines. *IEEE Intelligent Systems and their Applications*. 1998, vol. 13, no. 4, pp. 18–28. Available from DOI: `10.1109/5254.708428`.

43. SADEGHI, Mohammad Amin; FORSYTH, David. 30Hz Object Detection with DPM V5. In: FLEET, David; PAJDLA, Tomas; SCHIELE, Bernt; TUYTELAARS, Tinne (eds.). *Computer Vision – ECCV 2014*. Cham: Springer International Publishing, 2014, pp. 65–79. ISBN 978-3-319-10590-1.

44. YAN, Junjie; LEI, Zhen; WEN, Longyin; LI, Stan. The Fastest Deformable Part Model for Object Detection. In: 2014. Available from DOI: `10.1109/CVPR.2014.320`.

45. SRIVASTAVA, Shrey; DIVEKAR, Amit Vishvas; ANILKUMAR, Chandu; NAIK, Ishika; KULKARNI, Ved; PATTABIRAMAN, V. Comparative analysis of deep learning image detection algorithms. *Journal of Big Data*. 2021, vol. 8, no. 1, p. 66. ISSN 2196-1115. Available from DOI: `10.1186/s40537-021-00434-w`.

46. TERVEN, Juan; CÓRDOVA-ESPARZA, Diana-Margarita; ROMERO-GONZÁLEZ, Julio-Alejandro. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Machine Learning and Knowledge Extraction*. 2023, vol. 5, no. 4, pp. 1680–1716. ISSN 2504-4990. Available from DOI: `10.3390/make5040083`.

47. JOCHER, Glenn. *GitHub ultralytics repository* [online]. W3C, 2024. Available also from: `https://github.com/ultralytics/ultralytics`.

48. *Brief summary of YOLOv8 model structure · Issue #189 · ultralytics/ultralytics* [online]. 2023. Available also from: `https://github.com/ultralytics/ultralytics/issues/189`. Accessed on 05/02/2024.

49. *Screen_Shot_2020-05-27_at_4.17.41_PM.png (842×620)* [online]. 2020. Available also from: `https://production-media.paperswithcode.com/methods/Screen_Shot_2020-05-27_at_4.17.41_PM.png`. Accessed on 05/09/2024.

50. *MIT License* [online]. [N.d.]. Available also from: `https://opensource.org/licenses/MIT`. Accessed: 2024-04-28.

51. *Apache License 2.0* [online]. [N.d.]. Available also from: `https://opensource.org/license/apache-2-0`. Accessed: 2024-04-28.

52. GOOGLE LLC. *Google* [online]. 2024. Available also from: `https://www.google.com`. Accessed: 2024-04-28.

53. *GNU General Public License Version 3 (GPLv3)* [online]. [N.d.]. Available also from: `https://www.gnu.org/licenses/gpl-3.0.html`. Accessed: 2024-04-28.

54. FOUNDATION, Free Software. *Free Software Foundation* [online]. 2024. Available also from: `https://www.fsf.org/`.

55. *Frequently Asked Questions about Linux and the GPL* [online]. 2024. Available also from: `https://www.redhat.com/en/blog/frequently-asked-questions-about-linux-and-gpl`. Accessed on 04/27/2024.

56. *GNU Affero General Public License Version 3 (AGPLv3)* [online]. [N.d.]. Available also from: `https://www.gnu.org/licenses/agpl-3.0.html`. Accessed: 2024-04-28.

57. *Nextcloud - Open source content collaboration platform* [online]. 2024. Available also from: `https://nextcloud.com/`. Accessed on 04/27/2024.

58. *Dropbox.com* [online]. 2024. Available also from: `https://www.dropbox.com/`. Accessed on 04/27/2024.

59. *Personal Cloud Storage & File Sharing Platform - Google* [online]. 2024. Accessed on 04/27/2024.

60. *Ultralytics YOLOv8 Tasks - Ultralytics YOLOv8 Docs* [online]. 2024. Accessed on 04/27/2024.

61. *Ultralytics Licensing* [online]. 2024. Available also from: `https://www.ultralytics.com/license`. Accessed on 04/28/2024.

62. GITLAB INC. *GitLab* [online]. 2024. Available also from: `https://www.gitlab.com`. Accessed: 2024-04-28.

63. *Top 10 Open Source Licenses in 2018: Trends and Predictions | by WhiteSource | Medium* [online]. 2018. Available also from: https://medium.com/@WhiteSourceSoft/top-10-open-source-licenses-in-2018-trends-and-predictions-404fdcdca7bb. Accessed on 04/28/2024.
64. FREE SOFTWARE FOUNDATION, INC. *GNU Lesser General Public License, version 2.1 (LGPL-2.1)* [online]. 1999. Available also from: https://www.gnu.org/licenses/lgpl-2.1.html.
65. *The most popular licenses for each language in 2023 – Open Source Initiative* [online]. 2023. Available also from: https://opensource.org/blog/the-most-popular-licenses-for-each-language-2023. Accessed on 04/28/2024.
66. *Open Source Initiative* [online]. [N.d.]. Available also from: https://opensource.org. Accessed: 04/28/2024.
67. *ClearlyDefined* [online]. 2024. Available also from: https://clearlydefined.io/about. Accessed on 04/282024.
68. NPM, INC. *npm: A package manager for JavaScript* [online]. 2024. Available also from: https://www.npmjs.com. Accessed: 2024-04-28.
69. MICROSOFT CORPORATION. *NuGet Package Manager* [online]. 2024. Available also from: https://www.nuget.org. Accessed: 2024-04-28.
70. PYTHON SOFTWARE FOUNDATION. *PyPI - the Python Package Index* [online]. 2024. Available also from: https://pypi.org. Accessed: 2024-04-28.
71. APACHE SOFTWARE FOUNDATION. *Apache Maven* [online]. 2024. Available also from: https://maven.apache.org. Accessed: 2024-04-28.
72. *Meta-Chart - Free online graphing tool. Visualize data with pie, bar, venn charts and more* [online]. 2024. Available also from: https://www.meta-chart.com/. Accessed on 04/282024.
73. *Licenses | Choose a License* [online]. 2023. Available also from: https://choosealicense.com/licenses/. Accessed on 04/292024.
74. *5 min Recap for Andrew Ng Deep Learning Specialization-Course 3 | by Rishav Sapahia | Medium* [online]. 2017. Available also from: https://medium.com/@rishavsapahia/5-min-recap-for-andrew-ng-deep-learning-specialization-course-3-90e89a332b0c. Accessed on 04/302024.
75. *Object Detection and Person Detection in Computer Vision* [online]. 2024. Available also from: https://www.plugger.ai/blog/object-detection-and-person-detection-in-computer-vision. Accessed on 05/03/2024.
76. *Kaggle: Your Home for Data Science* [online]. 2024. Available also from: https://www.kaggle.com/. Accessed on 05/03/2024.
77. *Roboflow Universe: Open Source Computer Vision Community* [online]. 2024. Available also from: https://universe.roboflow.com/. Accessed on 05/03/2024.
78. STONX. *pingpang Dataset* [online]. Roboflow, 2021. Available also from: https://universe.roboflow.com/stonx/pingpang-2oqfz. visited on 2024-05-03.
79. *Data-Centric AI | What is Data-Centric AI & Why Does It Matter?* [Online]. 2024. Available also from: https://landing.ai/data-centric-ai. Accessed on 05/04/2024.
80. *Roboflow Annotate: Label Images Faster Than Ever* [online]. 2024. Available also from: https://roboflow.com/annotate. Accessed on 05/04/2024.
81. *Free and Paid Data Annotation and Labelling Tools Comparison* [online]. 2023. Available also from: https://maddevs.io/blog/data-annotation-tools-comparison/. Accessed on 05/04/2024.

82. *Open Source Data Labeling | Label Studio* [online]. 2024. Available also from: `https://labelstud.io/`. Accessed on 05/04/2024.

83. *Docker: Accelerated Container Application Development* [online]. 2024. Available also from: `https://www.docker.com/`. Accessed on 05/04/2024.

84. *Object Detection Datasets Overview - Ultralytics YOLOv8 Docs* [online]. 2024. Available also from: `https://docs.ultralytics.com/datasets/detect/`. Accessed on 05/04/2024.

85. *How to Annotate with Bounding Boxes [Guide & Examples]* [online]. 2021. Available also from: `https://www.v7labs.com/blog/bounding-box-annotation`. Accessed on 05/04/2024.

86. *How to Detect Small Objects: A Guide* [online]. 2020. Available also from: `https://blog.roboflow.com/detect-small-objects/`. Accessed on 05/05/2024.

87. *Transfer Learning in Image Classification: how much training data do we really need? | by Michele Zanotti | Towards Data Science* [online]. 2020. Available also from: `https://towardsdatascience.com/transfer-learning-in-image-classification-how-much-training-data-do-we-really-need-7fb570abe774`. Accessed on 05/05/2024.

88. *Label Studio — Video Object Detection Data Labeling Template* [online]. 2024. Available also from: `https://labelstud.io/templates/video_object_detector%5C#Fine-grained-control-actions`. Accessed on 05/05/2024.

89. TEAM, The pandas development. *pandas-dev/pandas: Pandas*. Zenodo, 2020. Latest. Available from DOI: `10.5281/zenodo.3509134`.

90. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON, Pearu; GÉRARD-MARCHANT, Pierre; SHEPPARD, Kevin; REDDY, Tyler; WECKESSER, Warren; ABBASI, Hameer; GOHLKE, Christoph; OLIPHANT, Travis E. Array programming with NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: `10.1038/s41586-020-2649-2`.

91. *HandBrake: Open Source Video Transcoder* [online]. 2023. Available also from: `https://handbrake.fr/`. Accessed on 05/05/2024.

92. *FFmpeg* [online]. 2024. Available also from: `https://ffmpeg.org/`. Accessed on 05/05/2024.

93. BRADSKI, Gary; KAEHLER, Adrian. *Open Source Computer Vision Library* [online]. 2021. Available also from: `https://opencv.org`. Accessed: 2023-05-05.

94. *Label Studio Documentation — Import pre-annotated data into Label Studio* [online]. 2024. Available also from: `https://labelstud.io/guide/predictions`. Accessed on 05/05/2024.

95. LOSHCHILOV, Ilya; HUTTER, Frank. *Decoupled Weight Decay Regularization*. 2019. Available from arXiv: `1711.05101 [cs.LG]`.

96. ROBBINS, Herbert; MONRO, Sutton. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*. 1951, vol. 22, no. 3, pp. 400–407. Available from DOI: `10.1214/aoms/1177729586`.

97. MICIKEVICIUS, Paulius; NARANG, Sharan; ALBEN, Jonah; DIAMOS, Gregory; ELSEN, Erich; GARCIA, David; GINSBURG, Boris; HOUSTON, Michael; KUCHAIEV, Oleksii; VENKATESH, Ganesh; WU, Hao. *Mixed Precision Training*. 2018. Available from arXiv: `1710.03740 [cs.AI]`.

98. HAO, Wang; ZHILI, Song. Improved Mosaic: Algorithms for more Complex Images. *Journal of Physics: Conference Series*. 2020, vol. 1684, no. 1, p. 012094. Available from DOI: `10.1088/1742-6596/1684/1/012094`.

99. *AI Ball Tracking for Sport Analysis* [online]. 2023. Available also from: `https://www.mercity.ai/blog-post/ball-tracking-for-sport-analysis`. Accessed on 05/11/2024.

100. KÁLMÁN, Rudolf Emil. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*. 1960, vol. 82, no. 1, pp. 35–45. ISSN 0021-9223. Available from DOI: `10.1115/1.3662552`.

101. SIMONYAN, Karen; ZISSERMAN, Andrew. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. Available from arXiv: `1409.1556 [cs.CV]`.

Contents of the source code

`data`	Folder for data used throughout the repository
`analysis`	Analyzing models
`annotation`	Folder with pre-processing and labeling
`experimentation`	Folder for experiments
`pre-annotation`	Folder that uses trained models to generate annotations
`play_model.py`	File to run trained models
`train.py`	Fine-tuning YOLO models
`utils.py`	Utilities used in multiple Python files
`README.md`	Description of the project with examples
`thesis.pdf`	Text of the thesis