Institute for Computer Science VII
Robotics and Telematics

Bachelor's thesis

Implementation and evaluation of object detection using neural networks for use in an automated optical rocket tracking system

David Baldsiefen

March 2022

First reviewer: Prof. Dr. Andreas Nüchter
Supervisor: Prof. Truong Thu Huong

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Nowadays, large-scale rocket launch campaigns are often documented on film for public outreach and error analysis purposes. This footage is created using complex and manually operated machinery. An *automated optical rocket tracker* would give model and sounding rocket enthusiasts access to the same kind of visual documentation. However, such a system requires an image processor which is capable of reliably detecting rockets at high speeds. This work proposes a neural network based approach which achieves a throughput of over 50 frames per second and a latency of 18.30ms on an NVIDIA Jetson AGX Xavier embedded computing board, while being able to correctly identify the target rocket in nine out of ten cases. Therefore, it is fast enough for use in an automated optical rocket tracker while also being very accurate. As the image processor is embedded in the Robot Operating System (ROS) framework, it is compatible with a wide range of robtics systems, including the optical rocket tracker developed by WüSpace e. V. and the TU Wien Space Team as part of the T-REX project.

Zusammenfassung

Konventionelle, groß angelegte Raketenstarts werden häufig visuell dokumentiert, um Filmmaterial zur Fehleranalyse und für Werbezwecke zu erstellen. Diese Aufnahmen werden üblicherweise mit Hilfe mit komplexen, handbedienten Anlagen erstellt. Ein *automatisierter optischer Raketen-Tracker* würde Modellraketen-Enthusiasten Zugang zu ähnlich hochqualitativem Bildmaterial ermöglichen. Allerdings stellt ein solches System hohe Ansprüche an die zugrundeliegende Bildverarbeitung, da Raketen auch bei großen Geschwindigkeiten zuverlässig erkannt werden müssen. Diese Arbeit stellt einen auf neuronalen Netzen basierenden Ansatz vor, welcher auf einem NVIDIA Jetson AGX Xavier Modul einen Durchsatz von 50 Bildern pro Sekunde bei einer Latenzzeit von 18.30ms erreicht. Dabei erkennt der Algorithmus in durchschnittlich neun von zehn Bildern die Zielrakete. Der vorgeschlagene Ansatz ist somit sehr genau und schnell genug, um in einem automatisierten optischen Raketen-Tracker eingesetzt zu werden. Da die Software im Robot Operating System (ROS) eingebettet ist, ist sie zu einer Vielzahl von Robotik-Systemen kompatibel. Dies beinhaltet auch den Raketen-Tracker, welcher vom WüSpace e. V. und dem TU Wien Space Team im Rahmen des T-REX Projektes entwickelt wird.

The NVIDIA Jetson AGX Xavier used in this work was generously sponsored by the NVIDIA Corporation as part of the NVIDIA Academic Hardware Grant Program.

Contents

Chapter 1

Introduction

We are so used to seeing high quality videos of rocket launches every month on the TV, computer or smartphone, that it is easy to forget how complicated it is to produce that footage. At any given moment, the rocket, which is travelling at very high speeds at an ever-rising altitude, has to be centered perfectly within the camera frame. This process of keeping the rocket centered at all times is described as rocket *tracking*.

Nowadays, conventional rocket launches are usually manually filmed, using heavy and expensive equipment, which was often originally developed for military purposes and consists of complex tracking mounts and large cameras [27]. Thus, the many advantages of gathering high-quality footage of rocket launches, such as error analysis and publicity, remain reserved to large-scale launch campaigns that can afford to acquire, maintain and operate such systems. Meanwhile, model rocket enthusiasts often have to resort to insufficient hand-recorded videos. The higher acceleration of model and sounding rockets provides a special challenge when trying to visually document these launches. The fact that model rocket events often take place in varying multi-purpose locations, such as meadows or off-season crop fields, adds an additional portability requirement to a potential tracker.

As of yet, there is no single system combining the requirements of portability, affordability and automation that need to be met to make optical tracking of model and sounding rockets viable. The T-REX project, a cooperation between WüSpace e. V.[1] and the TU Wien Space Team[2], has the goal of creating such a system.

However, the project's previous approaches to image processing have proven insufficient to reliably detect model and sounding rockets. While basic properties such as colour and contrast depend strongly on the rocket being clearly discernible from its environment, feature-based algorithms such as Haar cascades are too simplistic to reliably differentiate between rockets and similarly-shaped objects. Therefore, another approach is needed. One promising candidate is the use of neural networks. In recent years, neural network based object detection algorithms have become increasingly fast and accurate, meaning they can be trained on small datasets while being deployable in real-time speeds on edge devices.

[1] `https://wuespace.de/` (Accessed 27.02.2022)
[2] `https://spaceteam.at/` (Accessed 27.02.2022)

1.1 Thesis Goal

The goal of this thesis is to develop a neural network based image processor, that is well suited for use in an automated optical rocket tracker. As such, it should be

- accurate enough to reliably detect model and sounding rockets in a variety of settings and environments.
- fast enough to reliably track model and sounding rockets when deployed on an NVIDIA Jetson AGX Xavier embedded computing board.
- integrated into the Robot Operating System (ROS) to enable compatibility with a wide variety of robotics systems.

1.2 Thesis Outline

Chapter 2 gives an overview of the technological and scientific background of the thesis, describing the inner workings of neural networks in the process. In the third chapter, a performance requirement for the image processor is derived mathematically based on the specifications of common model and sounding rockets. This requirement is used to choose a suitable neural network based object detection model in the fourth chapter, which is then trained using footage from the T-REX project. Finally, an image processor that is capable of loading and running the trained model is implemented in the ROS framework in chapter 5, and its performance evaluated on the NVIDIA Jetson AGX Xavier. The results are then summarised in chapter 6, with a conclusion being drawn in regards to the initial thesis goal.

Chapter 2

Background and Terminology

2.1 T-REX

The Tracking Rocket EXperiments (T-REX) project is a cooperation between WüSpace e. V. and the TU Wien Space Team, with the goal of creating an affordable, portable and reliable automated optical rocket tracking system. The project was initiated in 2018 as a direct result of the failed REXUS 24 launch. During the launch, a failing retention mechanism caused the payload to unintentionally separate from the motor, striking the tail fin on its fall down and ultimately causing the rocket to disintegrate after around nine seconds of flight[40]. Even though many spectators were filming the launch manually using smartphones and DSLR cameras, it happened to be that no single videographer actually captured the moment of the explosion. Thus, the idea of an automated rocket tracker for error analysis was born.

The T-REX software is built on the Robot Operating System (ROS) framework[28]. It can be deployed on any conventional computer running Ubuntu 18.04. While it is possible to run the tracking-software and graphical user interface (GUI) on a single computer, the most common setup consists of a dedicated tracking computer, with the GUI being connected via network.

As of January 2022, the tracking-software supports up to two cameras. Image processing is performed based on colour, contrast, brightness or Haar features. It is supplemented with a Kalman filter, which helps mitigate the effects of false positives as well as false negatives. The outputs of the image processor are then fed into a PID-controller, which finally communicates with the motor interface.

On the hardware side, two stepper motors are used to move the dual-axis camera mount. Additional external cameras can be mounted on the tracker. A NVIDIA Jetson AGX Xavier serves as tracking computer.

2.2 Object Detection

Object detection is a field of computer vision that combines both object classification and object localization. As such, the goal of object detection is to identify different object classes in an image (e.g. 'car', 'cat', 'rocket') and draw bounding boxes around these objects.

Historically, simple features such as colour, edges and Haar features were used to perform

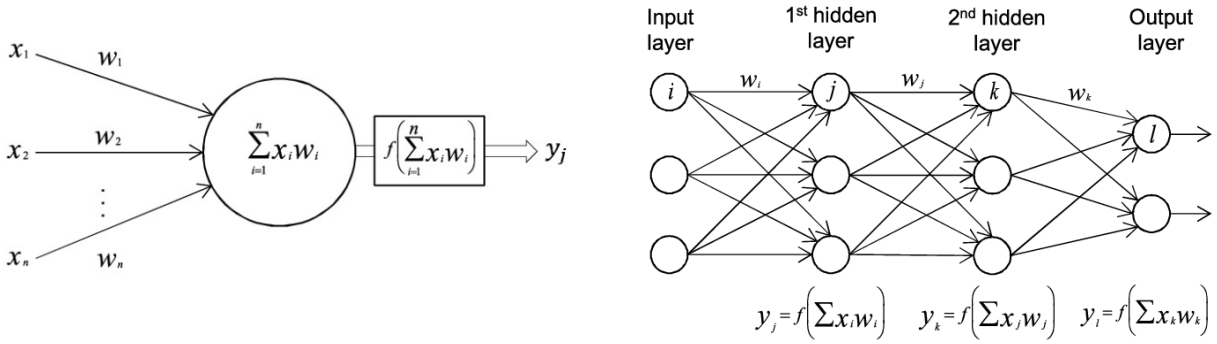
Figure 2.1: Picture of the T-REX rocket tracker.
Picture by courtesy of Antonius Adler.

both of these tasks. While these algorithms can be very fast, they require the user to define the desired features beforehand, making it hard to reliably detect complex objects. But in recent years, advances in the fields of artificial neural networks and machine learning introduced neural-network based approaches as meaningful alternative. Despite often being slower, they offer high reliability when detecting complex structures, given sufficient training beforehand. The act of producing a certain set of outputs from a given input is commonly referred to as *inference*.

2.3 Neural Networks

In recent years, artificial neural networks have become one of the most prominent approaches to machine learning. Based off of their biological counterparts, neural networks consist of thousands to millions of neurons, which are connected by weighted edges to produce a complex and deeply connected network. Multiple neurons are usually grouped into structures called ”layers” and only connected to neurons in the preceding and following layer. The first layer contains all the neurons activated by the input data, while the last layer contains the neurons representing the model's output. The remaining layers are referred to as ”hidden layers”, as they contain the inner structure of the network and are not ”visible” from the outside. The number of hidden layers is commonly referred to as the networks ”depth”, thus coining the term *Deep Learning*. Figure 2.2 gives an overview of a neural network's structure.

Using different learning algorithms, neural networks are trained on sample data with the goal of transferring the gained ”experience” to unfamiliar problems. The advantage of neural networks is that they can find optimal solutions to problems which can not be easily defined mathematically. In object detection, this means that neural networks learn to identify certain common features among all labeled objects which are often not recognised as such by human brains. The disadvantage is that the model's creator never knows the exact problem the neural network found the optimal solution to. This is often the result of incomplete or insufficient



(a) Neurons are the core building block of neural networks. Each neurons value ("activation") is determined based on a weighted sum of all its inputs, which is then passed through an activation function f. The bias C is omitted in this figure for clarity. Neural networks are created by connecting many neurons.

(b) Every network contains at least one input and one output layer, with none or multiple hidden layers in-between. Each layer contains hundreds to thousands of neurons. The two hidden layers in this case are so-called *fully connected* layers, as every neuron in one layer is connected to every neuron in the next.

Figure 2.2: Main components of neural networks. Left: Structure of a single neuron. Right: Topology of neural networks.[39]

training data. For example, a model may have learned to identify rockets in the given training dataset based solely on the colour of their exhaust fumes while ignoring all other features, thus failing to detect rockets that have left the burning phase. A solution to this simplified problem is to include rockets of all flight stages in the training data.

The goal of this section is to provide some insight into the mathematical inner workings of neural networks. In addition, prominent neural network models used in object detection are presented.

2.3.1 Components of Neural Networks

Neurons

Neurons are the fundamental component of every neural network. Each neuron receives n inputs $x_{1...n}$ and produces a single output y. The inputs originate from the networks input data, or other neurons. The output is usually normalised to a range of 0 to 1, and called the neurons *activation*. Each connection between neurons is associated with an individual weight factor w_i.

In order to calculate a neuron's activation, all the inputs are multiplied by the weight of their connections and then added together. This sum is extended by one more summand, the *bias* C. Lastly, the total sum is passed through an activation function $f(s)$, which usually restricts the neurons activation to a range of 0 to 1.

$$y = f\left(\left(\sum_{i=1}^{n} x_i w_i\right) + C\right) \tag{2.1}$$

The actual activation function used can vary significantly between different models and even layers. For example, You Only Look Once (YOLO) uses a linear activation function for the final layer and a leaky Rectified Linear Unit (ReLU) activation function for all other layers[31].

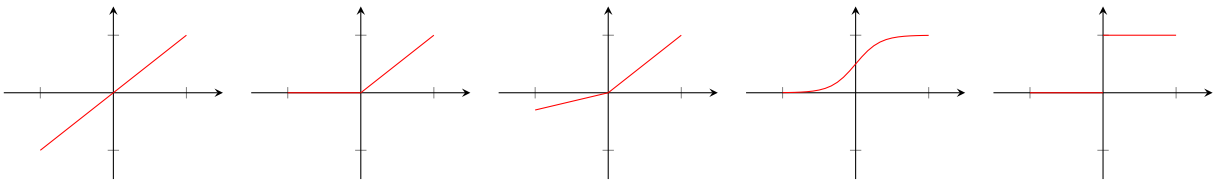


Figure 2.3: Common activation functions (left to right): Linear, ReLU, Leaky ReLU, Sigmoid, Heavyside

Other activation functions commonly used in neural networks are the Sigmoid function and the Heavyside step function (see figure 2.3).

Layers

As mentioned in the introduction to this section, neurons are usually grouped together in layers. In *feedforward* networks, neurons only receive information from the previous layer, and only send information to the next, thus causing a single-directional flow of information. In *recurrent* networks, neurons also connect to neurons of the same or previous layers, allowing cycles and loops. When every neuron in one layer is connected to every neuron in the next, they are called *fully connected layers*. When a group of neurons in one layer is combined into a single neuron in the next, thus reducing the amount of neurons in that layer, it is called a *pooling layer*. This has a downsampling effect. In *maxpool* layers, the weighted sum is replaced by a `max()` operation, whereas it is replaced by an unweighted average in *avgpool* layers.

2.3.2 Training Neural Networks

At its core, training a neural network refers to finding optimal weights and biases for all connections and neurons, so that a given input produces the desired outputs. As even a very small neural network consisting of only two fully connected 10-neuron layers already has $10 \times 10 = 100$ connections and ten biases, tuning those parameters by hand is rarely an option (and also not the point of using neural networks). Instead, training algorithms based on mathematical optimisation are used to adjust the weights and biases according to real input data.

The most common training approach used in object detection is that of *supervised learning*. In supervised learning, the training dataset consists of pairs of inputs and desired outputs. For example, an object detection training dataset contains images together with labels and bounding box coordinates. Other types of learning include *unsupervised learning*, where the model is not provided with desired outputs initially (useful for pattern recognition), and *reinforcement learning*, where the computer is placed in a dynamic environment with the goal of maximising some form of feedback-based reward[6]. The most common algorithm used for training neural networks is *backpropagation*, and was first introduced by Rumelhart, Hinton and Williams in 1986[34]. Its core principle is described in the following paragraph.

First, one image from the training dataset is fully processed by the neural network. After this *forward pass*, the actual and desired outputs are compared, and an error (or *cost*) is cal-

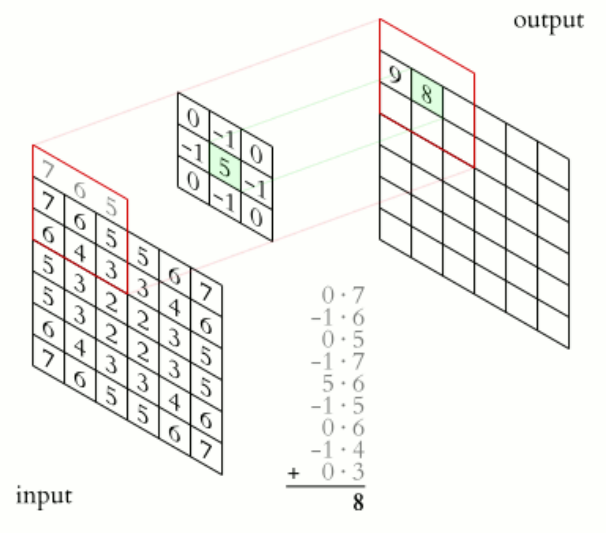


Figure 2.4: Visualization of the structure of a convolutional layer with a 2D input and output map. The kernel in the center contains the weights of the layer. [29]

culated using a *loss function*. By estimating the gradient of this loss function, local minima are determined with respect to the model's weights and biases. Thus, this gradient "points" towards the direction that the weights need to be tuned in order to achieve better results. This process is repeated for every layer starting from the last, until one full *backward pass* has been completed. Now, all parameters are tuned according to the gradients, and the entire process is repeated with the next image[34].

This procedure is performed hundreds or thousands of times for the entire dataset, until optimal weights and biases have been found. During the process, the network is evaluated on an independent set of data in regular intervals, to get a sense of how well it is progressing, and to notice when it stops improving.

2.3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural network which are particularly useful for computer vision tasks. The reason for this is that they significantly reduce the amount of weights between different layers by using a special layer structure called *convolutional layer*. Instead of connecting all neurons in one to all neurons in the next layer, convolutional layers employ a sliding window approach, where a square matrix of weights (called the kernel) is moved over equally sized pixel-squares of the input matrix. The value of the neuron in the output layer is then computed by calculating the dot product of the kernel's inputs with the kernel's weights (see figure 2.4). This concept can be easily applied to multi-channel inputs, by extending the kernel to a third dimension. The reason why this approach is particularly useful for computer vision tasks is that it significantly reduces the amount of weights per layer. Instead of growing exponentially with the resolution of the input image, the amount of weights in a convolutional

layer is of fixed size, defined only by the kernel's dimensions. At the same time, the sliding window approach means that each neuron in a new layer is only calculated based on small, spatially restricted regions in the preceding layer. This mimics the biological setup of the visual cortex of animals, and is particularly good for detecting localised features such as edges[20]. By adjusting the stride of the kernel, the overlap between each kernel window can be reduced, as well as the size of the output layer.[3]

Most CNNs are composed of multiple convolutional layers, followed by pooling and fully connected layers. Adjusting the structure and setup of each layer is a core part of neural network design.

2.3.4 State of the Art in Object Detection using Neural Networks

Up until the publication of YOLO[31] by Joseph Redmon et. al in 2016, there were two major ways to perform object detection using neural networks. First, there are deformable parts models (DPM), which involve moving a sliding window over the image and running a classifier on each possible position [14]. Secondly, there are Region Based Convolutional Neural Networks (R-CNNs) which use region proposal methods like selective search for generating potential bounding boxes, extract features using CNNs and finally run classifiers on these features to finalise the object detection[16][38].

Instead of improving on this multiple-step approach, Joseph Redmon et. al reframed object detection as a regression problem, where a single neural network predicts bounding boxes and class probabilities straight from the input image[31]. This is achieved by dividing the input image into a $S \times S$ grid, where each grid cell is responsible for detecting any object whose center falls within said cell. Therefore, each cell predicts B bounding boxes together with a confidence score. In addition, each cell determines a single set of C class confidences for its content. These class confidences are combined with each bounding box prediction, resulting in a total of $S \times S \times B$ predictions. Finally, overlapping bounding boxes are eliminated using non-maximum suppression.[31]

The network, consisting of 24 convolutional layers followed by two fully connected layers, proved to be very fast, outperforming any other real-time object detectors at the time in terms of speed and accuracy. In addition, the fact that it evaluated the whole image at once while performing classification and localization means that contextual information is implicitly encoded in the model as well, leading to a reduced amount of background errors. Its structure also allowed it to perform better on generalised representations of objects, such as artwork. However, the model does struggle with small structures, especially when they appear in groups.[31]

Because of these results, many researchers started adapting and improving the existing models, leading to the development of advanced models such as (Scaled-)YOLOv4[9, 41], YOLOv5[21], YOLOX[15] and PP-YOLO[19, 25]. As of 2021, most real-time object detectors are based on the YOLO approach[36].

2.4 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier is an embedded computing board developed by NVIDIA, and, as of March 2022, the most powerful module in the NVIDIA Jetson family. It is optimised for accelerating artificial intelligence, deep learning and computer vision tasks, and offers comparatively high performance at a small form factor (105mm × 105mm) and low power consumption levels.[11]

The module is equipped with a 8-Core ARM CPU in combination with a 512-core NVIDIA Volta GPU with 64 Tensor Cores. Together with two Deep Learning Accelerators (DLA), these processors can compute up to 16 trillion half-precision floating point operations per second (TFLOPS) for deep learning and computer vision tasks. In comparison, the Tesla V100 GPU commonly used for neural network benchmarking offers "only" twice the performance (32 TFLOPS) at roughly 8 times the power consumption (250W vs. 30W)[10]. The developer kit used in this work comes equipped with 32GB of RAM, and runs the latest JetPack SDK (version 4.6). All of the performance tests in chapters 4 and 5 are performed in the 30W power mode with jetson_clocks enabled, which locks the CPUs clocks to their maximum.[11]

The computing board's high performance and power efficiency at a comparatively low cost and form factor make it a good candidate for any optical rocket tracking system, which is why it was chosen as the main hardware platform for the T-REX project.

Chapter 3

Performance Criteria for the Image Processor

3.1 General Performance Criteria

The image processor developed in this work needs to reliably detect model and sounding rockets of all shapes and sizes. In order to allow the image processor to be used for automated optical tracking, this detection has to be performed at high speeds and accuracy. Detection speed is measured based on how many frames can be fully processed every second. Accuracy is measured based on several factors which are discussed in detail in chapter 4.

While tracking shall be possible during all stages of flight, the launch phase is the most critical and challenging detection phase. This is because a good initial estimate of the rockets velocity and acceleration are necessary to allow the controller to predict its future movement. The acceleration of the tracker motors is also highest during the launch phase, the relative angular velocity of the rocket in relation to the tracker decreases with increasing height and distance. The much more varied and chaotic background during the launch phase adds an additional challenge to accurately detecting rockets in the camera frame.

Table 3.1 contains a selection of model and sounding rockets commonly used today. As model rockets are usually custom combinations of motors, propellants, fairings and payloads, it is nearly impossible to define the ”average” model rocket. Therefore, table 3.1 contains popular pre-built sets sold by three of the most well-known manufacturers. Their launch accelerations are manually calculated based on the thrust- and weight specifications of the model in combination with the listed motor. The table also includes specifications of some of the rockets encountered during the T-REX project.

Rocket	Manufacturer	Launch acceleration	Length
Crossfire ISX (C6-5)	Estes	26g	0.40m
Pegasus (D9-7)	Raketenmodellbau Klima	13g	0.50m
Aspire (F10-8)	Apogee	21g	0.74m
Vanadium	TU Wien Space Team	12g	0.90m
The Hound	TU Wien Space Team	30g	4.00m
REXUS	Alcojet, ZARM	20g	5.60m
Black Brant IX	Magellan Aerospace	17g	12.20m

Table 3.1: Selected model and sounding rocket specifications. [4, 5, 17, 24, 35]

3.2 Calculating Speed Requirements of the Image Processor

3.2.1 Latency and Throughput

Before any calculations start, it is important to understand the two most relevant speed metrics used in image processing: Latency and throughput.

Latency describes how long it takes to fully process a single image. In this case, it is the time difference between the moment a new frame is captured and the moment the image processor outputs its results.

Throughput describes how many frames can be processed in a given timeframe. It is usually measured in frames per second (fps).

Even though for basic image processors both metrics describe essentially the same thing (fps being the inverse of the latency), they can differ greatly in more complex scenarios. For example, a multi-threaded image processor may be able to compute two frames at a time, thus doubling its throughput even though the latency is unaffected. Many neural network models also offer batch processing, a technique where multiple frames are loaded into the GPU and processed ”in one go” before outputting any results. This reduces throughput due to reduced memory calls, but at the same time increases latency, as the results for the first frame of the batch are only available once processing of the other frames has finished too. Therefore an inference batch size of one is usually the best option for real-time applications.

3.2.2 Setting up a Formula

In order to make the footage produced by the tracker usable for error detection, it is necessary for the rocket to fill up a significant portion of the image at all times. While there are many considerations flowing into the correct positioning of the tracker relative to the rocket, the main factors relevant for the design of the image processor are the size of the rocket in the frame as well as its acceleration within the frame.

Let’s assume that for decent error detection, the rocket needs to initially fill up one fifth of

the frame. This means that the rocket can cover four times its own length before starting to move out of frame, as long as it was positioned at the bottom initially. Although the actual relation between pixel-size on the frame and real height is not linear due to optical distortions, this effect is negligible at small fields of view such as that of a zoomed-in tracking camera.

Despite the fact that the acceleration of most rockets fluctuates over time due to the burned fuel reducing the overall mass and due to the fuel input in general being often non-linear, a near linear acceleration in the very first milliseconds of the launch can be assumed for model rockets. This is because the amount of fuel burned in that time is usually relatively low compared to the total rocket's weight. For example, the total mass of the Crossfire ISX model rocket decreases by around 20% during the 1.6 second thrust duration. As the C6-5 engine has a constant burn rate, this means that the acceleration of the rocket only increases by roughly 2.6% in the first 200ms. This work generally assumes linear launch acceleration for the remaining chapter, but the inaccuracy is covered for by choosing a higher k in subsection 3.2.3.

The distance covered by a linearly accelerating object in the time frame t is described by the formula $s = \frac{1}{2}at^2$. Let r be the portion of the frame the rocket fills up. Using the length l and acceleration a of the rocket, the total time T it spends completely in the frame is calculated by

$$T = \sqrt{\frac{2s}{a}} = \sqrt{\frac{2l}{a}(r^{-1} - 1)} \qquad 0 < r < 1, \quad a > 0 \tag{3.1}$$

This means that once T seconds have passed, the rocket tracker needs to start moving in order to keep the rocket in the camera frame. Thus, within T, enough information needs to be gathered by the image processor to allow the controller to send appropriate commands to the motors.

Let's define k as the minimum number of detections the controller needs to accurately predict the rockets future movement. Then the image processor needs to *finish* processing k frames before T is over. At a throughput of f, the k^{th} frame starts being processed after $(k-1) \cdot f^{-1}$ seconds, and finishes after another latency-period of t seconds. This leads to the following relation between throughput, latency and T:

$$\frac{k-1}{f} + t < T \tag{3.2}$$

As discussed in subsection 3.2.1, the relation between latency and throughput for a basic image processor can be described by $t = f^{-1}$. Using this, the formulas for minimum throughput and maximum latency are defined as follows:

$$f_{min} > \frac{k}{T} = k \left(\sqrt{\frac{2l}{a}(r^{-1} - 1)} \right)^{-1} \qquad k, f, T > 1, \quad a > 0, \quad 0 < r < 1 \tag{3.3}$$

$$t_{max} < \frac{T}{k} = k^{-1} \left(\sqrt{\frac{2l}{a}(r^{-1} - 1)} \right) \qquad k, f, T > 1, \quad a > 0, \quad 0 < r < 1 \tag{3.4}$$

However, any image processor that satisfies equation 3.2 is still suitable. Note how both requirements get more strict when the length-to-acceleration relation of the rocket minimises.

3.2.3 Choosing Suitable Parameters

Among the rockets listed in table 3.1, the Crossfire ISX model rocket has the lowest length-to-acceleration ratio at $\frac{l}{a} \approx 0.0016\text{s}^2$. While rockets with even lower ratios do certainly exist, it serves as a good estimate for what an optical rocket tracker is likely to encounter.

The portion of the screen the rocket shall fill up depends largely on the personal preference of the user, the resolution of the camera and the intended use case of the footage. This work uses $r = 1/5$. Please note that this parameter is only used to calculate the overall performance target, and that the image processor is still able to detect smaller and larger rockets.

According to equation 3.1, the total time the rocket spends within the frame at the given parameters is $T = 112\text{ms}$.

The amount of consecutive detections k that the controller requires to produce good tracking results varies greatly based on the controller used. Naturally, more input data increases the accuracy of the controller. If the very initial acceleration of the rocket is actually near-linear, a well-tuned PID-controller may be able to reliably predict its future movements from just three measurements. Experience during the T-REX project confirms this assumption. Nevertheless, this is not always be the case, and as table 3.1 shows, the different rockets vary greatly in regards to their initial acceleration. Thus, a higher k is a more suitable choice. In this work, the fps target is calculated with $k = 5$.

Inserting these parameters into equations 3.3 and 3.4 returns a throughput requirement of 44.6 fps at a latency of 22.4ms. However, these calculations do not account for the fact that the rocket tracker needs some additional time to pass the detections through the controller and set the motors in motion. Thus, I define the final performance requirement as 50.0fps at a latency of 20.0ms, which gives the tracker 12 additional milliseconds to start moving before T is reached.

While the image processor developed in this work is optimised for this performance target, it is important to note that by adjusting camera zoom (and thus r) or the controller (and thus k) it is still usable for rockets exceeding the length-to-acceleration relation defined at the beginning of this section.

Chapter 4

Model Selection and Training

While the previous chapter defined general performance criteria for an automated optical rocket tracker's image processor, the goal of this chapter is to identify promising neural network models which are capable of fulfilling these requirements. In addition, this chapter covers the creation of the training dataset, as well as the subsequent training of the best-performing model. This trained model is then used as the foundation for the image processor developed in chapter 5.

4.1 Selecting Promising Neural Network Models

There are many promising neural network models being developed with the goal of offering high-accuracy real-time object detection. In relevant literature, the real-time limit is commonly defined as 30 fps. However, as the speed of any model largely depends on the computing power available, many models far surpass this limit in order to still enable real-time detection on low-end devices. The goal of this section is to identify the most accurate candidates for object detection at 50fps on the NVIDIA Jetson AGX Xavier.

The most common metric used to compare the accuracy of different object detection model's is the *mean Average Precision* (mAP), which describes the model's detection performance on a certain task. In most cases, this means running the object detector on a large, predefined and standardised dataset. Microsofts Common Objects in Context (COCO) is one of the most common datasets used to evaluate and compare different models. Microsoft also defines specific guidelines on how to perform the measurements[23]. In order to understand how the mAP is calculated, it is necessary to define Precision, Recall and Intersection over Union (IoU) first:

- **Precision** describes how many of the model's predictions are actually true. It is equal to the number of true positives divided by the total number of predictions
- **Recall** describes how good the model is at correctly finding all the existing positives. It is equal to the number of true positives divided by the number of ground truths.
- **Intersection over Union** describes how well a predicted bounding box and the actual bounding box overlap. It is equal to the area of overlap divided by the area of union of both bounding boxes (see figure 4.1).

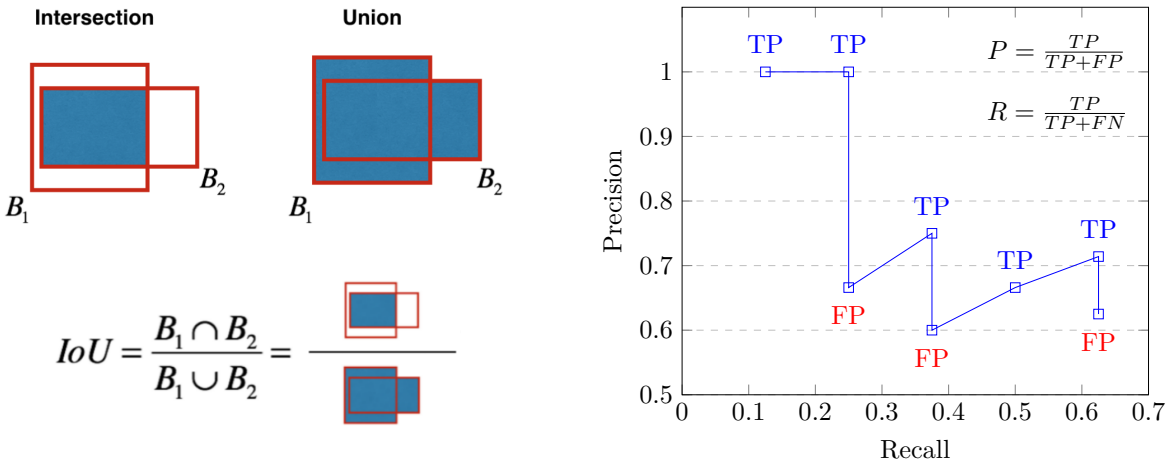


Figure 4.1: Left: Visualization of Intersection over Union (IoU) determination. Right: Example of AP-calculation using the precision-recall curve. The predictions are sorted in descending order according to their confidence score, and then classified as either true or false positive (TP or FP). Precision and recall are calculated based on the accumulated count of true and false positives. The Average Precision is equal to the area below the curve.

When determining whether a given prediction made by an object detector is a true positive, the IoU of the prediction is compared to a predefined threshold. The prediction is classified as true positive only if the classification is correct and the IoU is larger or equal than that threshold. Every other prediction counts as a false positive.

In order to calculate the *Average Precision* (AP) of a model, all the model's predictions in a given test set are classified as either true or false positive, and then sorted in descending order according to their confidence score. Next, the the number of true and false positives are accumulated in the same order, and then used to calculate the precision and recall of each element. This results in a precision-recall curve as shown in figure 4.1. The Average Precision is equal to an approximation of the area under the precision-recall curve. For the COCO challenge, it is approximated based on a 101-point interpolation[23].

To get the *mean* Average Precision of a given object detection model, the AP value is averaged over all object classes and different IoU thresholds. For the COCO challenge, this includes 80 classes, with the AP being calculated for each class with ten different IoU values (0.5 to 0.95 in steps of 0.05). The mAP at a single IoU threshold of 0.5 is often determined in addition to the "general" mAP value (mAP@0.5 or mAP_{50}). Some researchers also specify mAP values for differently sized objects.

In literature, mAP and AP are often treated like synonyms, even though both have different meanings. When this occurs, it is much more common that mAP is wrongly referred to as AP, than vice versa.

Model	Backbone	Size	mAP	mAP_{50}	Throughput (fps)	
					w/o TRT	w/ TRT†
YOLOv4[9]	CSPDarknet53	608	43.5%	65.7%	62	105.5
YOLOv4-CSP[41]	CSPDarknet53s	640	47.5%	66.2%	73	-
YOLOv5s[19][21]	-	640	36.7%	55.4%	113.0	-
YOLOv5m[19][21]	-	640	44.5%	63.1%	88.2	-
PP-YOLO[25]	ResNet50-vd-dcn	608	45.9%	65.2%	72.9*	155.6
PP-YOLOv2[19]	ResNet50-vd-dcn	640	49.5%	68.2%	68.9	106.5
PP-YOLOv2[19]	ResNet101-vd-dcn	640	50.3%	69.0%	50.3	87.0
YOLOX-S[15]	Mod. CSP v5	640	39.6%	-	102.0†	-
YOLOX-M[15]	Mod. CSP v5	640	46.4%	65.4%	81.3†	-
YOLOX-DarkNet53[15]	DarkNet53	640	47.4%	67.3%	90.1†	-
EfficientDet-D1[37]	EfficientNet-B1	640	40.5%	59.1%	74.1*	-

Table 4.1: Comparison of throughput and accuracy of different neural network based object detection models. All measurements were performed on the MS-COCO test set (test-dev 2017) on a Tesla V100 GPU at batch size 1. Fields where no official data is available are marked with '-'. Timings marked with '†' were measured in FP16 precision. '*' indicates that the measurement includes bounding box decode time (1-2ms).

It is important to point out that for the rocket tracker itself, mAP may not be the ideal accuracy metric. While mAP is calculated based on *all* predictions of a neural network, a rocket tracker can only follow one rocket at a time, meaning that it is only necessary for the *most confident* prediction to be actually correct. Therefore, it does not matter for the tracker if predictions with a lower confidence score are correct as well. Nevertheless, this chapter mainly focuses on optimising mAP, as it is the de facto industry standard and still a good indicator of a model's general performance.

Table 4.1 contains an overview of some of the most popular and promising object detection model's for the given use case. The presented model size refers to the maximum width and height of the input image, and is chosen based on the frame-size of the current T-REX camera (480x640). However, most models can be scaled up and down at a cost of accuracy. For YOLOv4 and PP-YOLO, the original papers only contain measurements at a model size of 608[9, 25]. It is important to note that among the papers, different authors often employ very different ways of measuring the speed of their models. For example, some[37, 41] include postprocessing in the final timings, while others[9, 31] do not disclose the details of what is included and what is not. That is why these values always have to be compared with care. As of March 2022, no official paper for YOLOv5 has been released, which is why the corresponding rows contain results obtained by Xin Huang et al. as part of PP-YOLOv2 in 2021[19].

Tiny object detectors, such as PP-PicoDet-L[42], PP-YOLO-Tiny[42] and YOLOv4-tiny[41], are intentionally disregarded, as they lack the accuracy of larger models while being optimised for weaker hardware.

Some models are directly ruled out as candidates based on their metrics. For example, YOLOv4 and PP-YOLO are outperformed by YOLOv4-CSP both in terms of speed and accuracy. Furthermore, benchmarks in the official repository for PP-YOLO[1] reveal the model to reach just 20fps on the Jetson AGX Xavier when using TensorRT, which is significantly slower than the 50fps required for the rocket tracker. Based on this information, PP-YOLOv2 is also ruled out, as it is even slower than PP-YOLO according to the original benchmarks of the paper[19]. EfficientDet-D1 is 1.1fps faster but 7.0% points less accurate than YOLOv4-CSP.

YOLOX-M, YOLOX-DarkNet53 and YOLOv5m are all accurate and fast, even though it has to be noted that throughput of the YOLOX models was measured with FP16 precision. The fastest models are the small object detectors YOLOv5s and YOLOX-S, even though they are much less accurate than their full-sized counterparts.

Finally, I chose YOLOv4-CSP, YOLOv5 and YOLOX for further investigation due to their good speed/accuracy trade-off.

4.2 Deploying and Evaluating Performance on NVIDIA Jetson AGX Xavier

In this section, the different models are deployed on the NVIDIA Jetson and compared in terms of speed and accuracy. To make best use of the Jetsons capabilities and its limited hardware, each model is first converted to a half-precision TensorRT engine, and then evaluated using the `trtexec` tool provided by NVIDIA. In addition, accuracy is determined by testing the converted model against the COCO2017-validation dataset.

The results of each measurement are listed in table 4.2. Where possible, tests are performed with the newest available version based on the official GitHub repositories. Accuracy is only determined for models that are able to achieve at least 45fps.

4.2.1 Technical Background

ONNX

Open Neural Network Exchange (ONNX) is an open source software ecosystem with the purpose of enabling and simplifying compatibility and interoperability between machine and deep learning projects and formats. ONNX also defines an extensible computation graph model, which is used to transfer neural network models between different frameworks. All of the models listed in table 4.1 are provided with tools for ONNX conversion.[7]

TensorRT

TensorRT is a SDK developed by NVIDIA with the purpose of optimising deep learning inference on NVIDIA GPUs. By utilizing NVIDIA's CUDA platform, which allows software to perform highly parallelizable computations on the GPU, software can be sped up significantly

[1] `https://github.com/PaddlePaddle/PaddleDetection/blob/release/2.3/deploy/BENCHMARK_INFER_en.md` (Accessed 16.02.2022)

when compared to conventional CPU deployment. In addition, TensorRT allows neural network models to be calibrated for 16-bit floating-point (FP16) and 8-bit integer (INT8) formats. As a lower precision reduces requirements for storage and bandwidth significantly, it can cause a significant speedup for tasks where large numbers of calculations have to be performed. For most neural network models, the loss in accuracy is far outweighed by the performance gains (this is also shown in the measurements performed in this work, see subsection **??**). TensorRT supports most major neural network frameworks, such as PyTorch, TensorFlow and ONNX.[12]

trtexec

`trtexec` is a command line tool provided as part of TensorRTs Open Source Software (OSS) components. It includes basic functionalities for engine serialization and benchmarking. New engines can be generated from ONNX, UFF or Caffee protoxt models. [13]

4.2.2 Testing Methodology

All TensorRT engines are generated based off of ONNX models using `trtexec` and FP16 precision. Modifying and calibrating the engine for INT8 precision is not necessary to get a representative performance comparison. The benchmarks are performed with a warmup period of 10s and an execution period of 3000 iterations. A 3x640x640 tensor representing a three channel input image is used as input shape. The batch size is 1. Accuracy is measured on the COCO validation dataset using the tools provided in the YOLOv5 and YOLOX repositories.

During the entire process, the jetson is operating in MAXN power mode with jetson_clocks enabled.

4.2.3 Results

Model	mAP^{val}	mAP^{val}_{50}	Throughput
YOLOv4-CSP	-	-	34.6
YOLOv5s	36.0% (36.4%)	55.4% (56.5%)	113.0
YOLOv5m	43.1% (44.3%)	62.7% (63.8%)	54.7
YOLOX-S	40.3% (40.5%)	58.9% (59.3%)	104.7
YOLOX-M	46.6% (46.9%)	65.1% (65.6%)	50.4
YOLOX-DarkNet53	47.4% (47.7%)	66.5% (67.0%)	49.6

Table 4.2: Throughput and accuracy measurements made on the NVIDIA Jetson AGX Xavier. All models were converted to TensorRT engines with FP16 precision. For engines that surpassed 45fps, accuracy was determined based on the COCO2017-validation dataset, with the accuracy of the default (non-TensorRT) models being listed in brackets.

Of the tested engines, four reach the performance target of 50fps at an input resolution of 640x640. Of those, YOLOX-M achieves the highest accuracy at 46.6%mAP on the COCO

validation dataset. YOLOv5m comes in second at 43.1%mAP and 54.7fps, which is 4.3fps faster than YOLOX-M. The fastest engine is YOLOv5s at 113fps. However, YOLOX-S is 4.3 percent points more accurate than YOLOv5s while only being 7% slower. The conversion to FP16 precision caused the mAP to drop by 0.5% points on average

Overall, YOLOv5m offers the best speed/accuracy trade-off: it is fast enough for detecting rockets in real-time, while still offering some margin for potential performance losses during pre- and postprocessing. However, YOLOX-S is a promising alternative when a faster model is of need.

4.3 Dataset Creation

The key component to successfully training any neural network is the dataset. In order to be able to learn generalised representations of an object, the neural network needs to be provided with specific examples from all possible situations first. This includes (but is not limited to) all the angles, lighting conditions, sizes and colours in which the object is likely to appear. That is why it is not uncommon for big datasets to contain hundreds of thousands of labeled images.

4.3.1 Collecting Data

The dataset used to train the neural network needs to contain imagery that is as close as possible to the real conditions the tracker will encounter. While a human is able to easily draw a connection between a picture of the space shuttle in space and a model rocket on the ground, a neural network will just be confused by these very different scenes, as they do not share a lot of visual features. That is why it is not a feasible approach to just label the first 1000 results returned by entering "Sounding Rocket" in a search engine.

Instead, the imagery in this work is derived from pictures and videos gathered during the T-REX project. The advantage of this approach is that the images are very close to future imagery in regards to setting, lighting conditions, camera properties, rockets etc. The disadvantage is that the available footage is quite limited. Over the second phase of the T-REX project, five rocketry events were visited, resulting in 216 videos totaling around 45 minutes of raw footage.

By extracting selected frames out of these videos, the dataset is created. These frames are chosen to be as diverse as possible, representing all phases of the launch (rocket sitting on launchpad, launching, flying and dropping). While it is possible to extend the dataset by including more frames with higher similarity, training advantages would get increasingly limited. The fact that the footage was created with a diverse set of cameras, optics and setups increases variety among the images extracted. At the same time, it improves the neural network's ability to handle different setups in the future.

As final step in the dataset creation process, it is necessary to decide on how to resize differently-sized images. As discussed before, the chosen model is trained for an input size of 640x640, meaning that larger imagery needs to be downscaled before being usable for training. While there are many different methods to choose for this process (e.g. *stretching*, *cropping* or

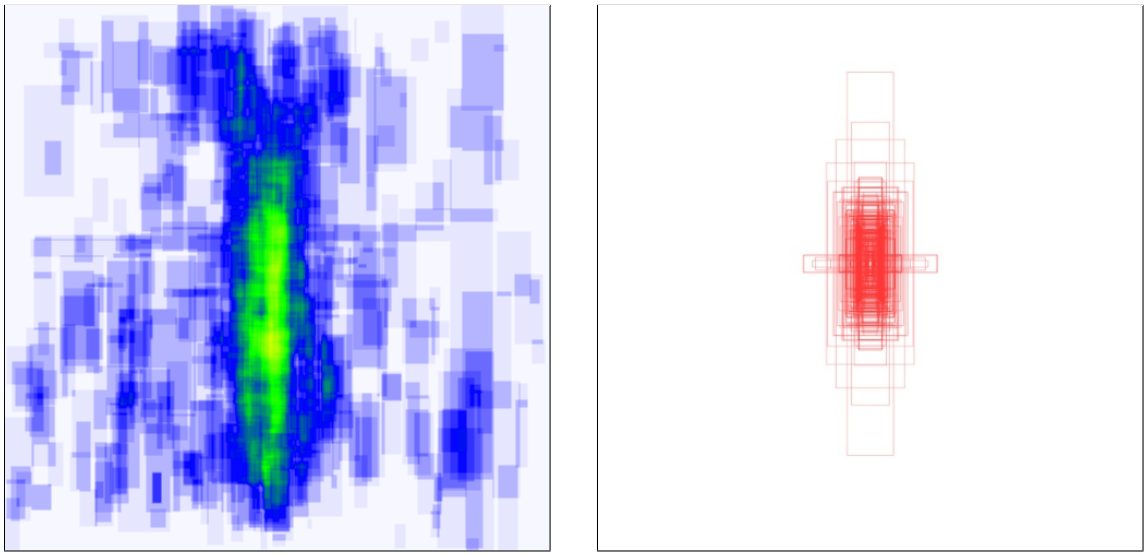


Figure 4.2: Annotation heatmap of the dataset (left) and bounding box sizes (right). The average bounding box size of the dataset is 18x66 pixels. Images created using Roboflow Annotate[33].

fitting), most of them distort the image, meaning that the network learns incorrect representations of the objects it is supposed to recognise. In this work, fitting is used, which means that large images are rescaled wile maintaining their aspect ratio. The same method is used to preprocess images in the image processor developed in chapter 5.

4.3.2 Labeling

Image labeling is the process of identifying and marking different object classes present in an image. As a matter of fact, most people label images on a day-to-day basis without even noticing, as it is one way how companies like Google perform their "I am not a robot"-checks. For the present work, image labeling includes drawing bounding boxes around all rockets present in each image of the dataset, including those which are not fully visible or occluded partly. The tool used in this work is the open-source Computer Vision Annotation Tool (CVAT) developed by Intel[2].

The final dataset consists of 525 images, containing 627 annotated rockets. Figure 4.2 shows a heatmap of all bounding boxes across all images. It is clearly visible that the rockets are mainly focused around a vertical column in the center of the image. This information can be used when considering cropping images to gain performance. The second image in figure 4.2 contains a centered representation of all bounding boxes, giving a good depiction of average bounding box dimensions. This information is often used by neural network developers to determine so-called *anchor boxes*, which can be understood as predefined bounding boxes for which the model searches first. Appendix A contains examples of labeled images.

[2]https://cvat.org/ (Accessed 04.01.2022)

4.3.3 Splitting Training, Validation and Test Sets

Before the actual training can start, it is necessary to split the dataset into subsets for training, validation and testing. This is necessary to avoid a phenomenon called *overfitting*. Overfitting occurs when the model starts aligning too closely to the training data, meaning that it learns to "memorise" the data instead of recognizing general patterns. Thus, the model's effectiveness when processing new images it wasn't trained on starts to decrease.[32]

The *training* set is the largest portion of the dataset and used for the actual model training. After each training iteration, the model is evaluated based on its performance on the *validation* set. This allows the developer or algorithm to get a sense of how well the model performs on images it never saw before. The results of this evaluation are also generally used to further enhance the model, e.g. by tuning its parameters. A *test* set is used to make a final assessment of the model's effectiveness. This prevents the final evaluation from being skewed towards the validation set as a result of the developer manually overfitting the model in its design process. In many object detection challenges, the labeled test data is not available publicly for the same reason.

There is no clear consensus on how to ideally divide the data into the three subsets. It largely depends on the total size of the dataset as well as the general task at hand, and can be considered an optimisation problem itself. For this work, the data is divided into 70% for training, 20% for validation and 10% for testing, which is the split recommended by the developers of YOLOv5[32].

4.4 Training

There are three main ways how the general performance of a neural network based object detection model can be improved.

The first and most obvious step is **extending and improving the training dataset**. As mentioned before, a diverse and complete dataset is the key component to decently training any model. Increasing variety among images and excluding those which divert too much from the sought-after objects almost certainly improve a model's training performance.

Secondly, developers can **adjust training parameters** to change *how* the model learns. For example, one can change the number of epochs, the batch size or the training-related hyperparameters such as learning rate and decay.

Lastly, it is possible for experienced developers to **modify the model itself**. This includes changing, adding and removing layers from the model, modifying the activation function or performing similar adjustments to the model's structure and topology. Many new neural network models are created through this way.

This work focuses on adjusting training parameters.

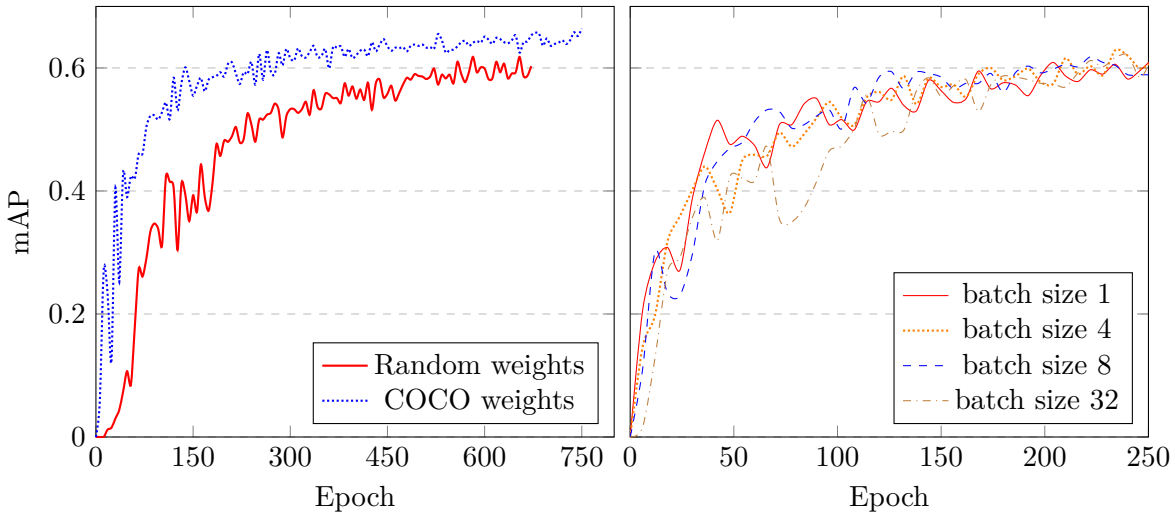


Figure 4.3: Accuracy development during model training. The left plot visualises how much longer it takes for the model to converge when initiating it with random weights. The right side visualizes how different batch sizes affect the initial training performance.

4.4.1 Initiating Training with Random vs. Pretrained Weights

The first training is performed with the default YOLOv5m training configuration with batch size 16. Instead of starting from scratch with random weights, it is initiated based on an existing model trained on the COCO dataset. The training is performed over 1000 epochs, meaning the entire dataset passes through the network just as often. However, the early-stopping algorithm implemented in YOLOv5 stops the training earlier if no improvement is seen over 100 consecutive epochs. The first training automatically stops after 720 epochs, with the best results being observed at epoch 619. The final accuracy values on the validation dataset are 66.4%mAP and 93.1%mAP_{50}.

The next training is executed with the same parameters, but with random weights instead of starting from a pretrained model. After 683 epochs, the model stopped training at an accuracy of 61.8%mAP (92.2%mAP_{50}). The results are visualised in figure 4.3. It is interesting to see how it takes significantly longer for the model to start converging when the weights are initiated randomly. These results clearly confirm that, at least in the given case, it is better to start from a pretrained state instead of starting from scratch when training a new model.

4.4.2 Varying Batch Sizes

Next, the performance of the model when trained at different batch sizes is compared. The batch size defines how many samples are passed through the neural network before adjusting any weights. A larger batch size usually allows for a more accurate estimate of the gradient, but comes at a higher memory cost and often leads to degrading generalization performance[22]. At the same time, training usually converges faster when using smaller batch sizes, as weights are

Batch Size	Initial Weights	Hyper-parameters	mAP^{val}	mAP^{val}_{50}	Epochs
1	COCO	default	66.1%	92.1%	714
4	COCO	default	66.0%	92.7%	627
8	COCO	default	66.3%	94.0%	677
8	COCO	evolved	66.3%	92.5%	960
16	random	default	61.8%	92.2%	683
16	COCO	default	**66.4%**	93.1%	720
16	COCO	evolved	65.7%	**94.5%**	933
32	COCO	default	65.0%	94.3%	683

Table 4.3: YOLOv5 performance on the rocket validation dataset after training with different configurations.

updated after each pass[18]. For YOLOv5 however, it is important to note that the developers decided to automatically adjust two hyperparameters together with the batch size: loss and weight decay. This was done as to make training performance of the model as indifferent to the batch size as possible.

In this work, training is evaluated for batch sizes 1, 4, 8, 16 and 32. A higher batch size is not possible due to the limited memory availability on the NVIDIA Jetson. The results are listed in table 4.3 and visualised in figure 4.3.

The final accuracy is quite similar across all batch sizes. The highest mAP is observed at batch size 16, even though mAP_{50} was slightly higher for batch size 32 at 94.3%. While the very initial gain in accuracy differs greatly across the batch sizes, it converges after around 200 epochs. It is interesting to note that the total training time was very different across the different batch sizes. At a batch size of 32, the algorithm took an average of 20 seconds per epoch, which is almost six times faster than the 116 seconds it took to compute one epoch at a batch size of one. This can be explained by the higher amount of forward- and backwards passes necessary per epoch (the number of passes necessary equals the amount of training samples divided by the batch size).

4.4.3 Hyperparameter Evolution

There are about 25 hyperparameters that can be tuned to optimise the training performance of YOLOv5. While adjusting them all manually is always an option, it requires a lot of experience and research to find optimal values by hand. That is why YOLOv5 offers a feature called *hyperparameter evolution*, a hyperparameter optimisation method using a genetic algorithm at its core. As genetic algorithms are a whole big topic on their own, this section only describes the methodology used in the YOLOv5 implementation. A more generalised introduction can be found in [26].

The YOLOv5 hyperparameter evolution algorithm works as follows. First, the model is trained for a specific set of epochs with default parameters. Then, these parameters are *mutated* according to the formula described below. Next, a new generation is trained based on the new parameters. This procedure has to be repeated for many generations, with every generation choosing its starting parameters based on the best performing generations that preceded it. This *selection* process is slightly randomised on its own, so that there is a slight chance of choosing the second or third best generation as a starting point instead of the best. During mutation, there is a 80% chance of a parameter value being changed. If that is the case, a mutation factor is calculated based on a normal distribution (mean = 1, variance = 0.04) and a parameter-specific *gain* value. The original hyperparameter is then multiplied by this factor and limited to predefined constraints.[21]

While there is a good chance of hyperparameter evolution producing optimised parameters, it is a very time-intensive process that cannot guarantee ideal outcomes. In this work, evolution is performed for 300 generations of 20 epochs each (6000 epochs in total). The batch size is 16. The evolved hyperparameters (as well as the defaults) can be found in table A1 of the appendix.

Using these parameters, training is re-initiated at batch sizes 8 and 16. For batch size 8, the evolved parameters proved to be worse than the standard ones, leading to a lower mAP and mAP_{50} (table 4.3). For batch size 16, only mAP_{50} increased slightly by 1.4% points while mAP degraded by 0.7% points. Overall, these results indicate that the default hyperparameters may already be near-optimal for the given use case, and that much more generations might be necessary to find better values.

4.5 Results and Discussion

To get a final and unbiased evaluation of the trained models, the most promising two are converted to FP16 TensorRT engines again using `trtexec` and evaluated using the test set of the rocket dataset. This includes the models that were trained at batch size 8 and 16 with otherwise default configurations.

The model trained at batch size 8 achieves a final mAP of 53.9% (86.9% mAP_{50}), compared to slightly better 55.9% for the model trained at batch size 16 (87.4% mAP_{50}). The lower accuracies on the test set can be explained by the low amount of images within the set: It is unlikely that just 50 images properly represent all possible detection scenarios, and if only five images contain exceptionally difficult settings, 10% of the set are already affected. Considering this, the viability of using such small subsets can be questioned. As long as the dataset is not significantly larger, it might make more sense to omit the test set and use its contents for training and validation instead.

Nevertheless, the final model is still very accurate and well suited for use in an optical rocket tracker. Figure 4.4 shows that the model performs decently well at detecting all rockets, but

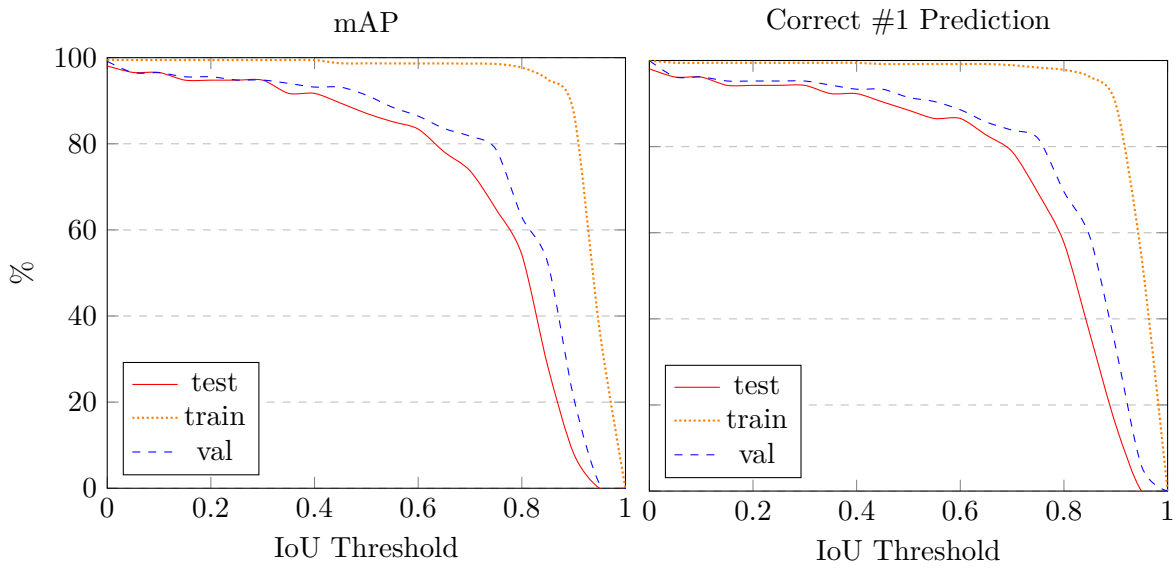


Figure 4.4: Left: mAP of the final model at different IoU thresholds. Right: Percentage of cases where the most confident prediction of the model is correct. Note how the model performs almost ideally on the training set, whereas it struggles more with images that it has never seen.

struggles with accurately fitting bounding boxes, leading to worse performance at higher IoU thresholds. For a potential tracker, this means that general trends - like the rocket's acceleration and direction - are most likely reproducible, even though the detected position of the rocket at a given moment might deviate slightly from the actual one.

When evaluating just the model's most confident predictions (as it will be the case during real-world deployment), it is able to correctly detect rockets in 90.4% of the test and validation images at an IoU of 50%. Actively including this metric into the selection, training and validation process would be an interesting approach to investigate in future works.

Chapter 5

Implementation in the ROS Framework

The goal of this chapter is to develop an image processor which is capable of loading and running the YOLOv5m model trained in the previous chapter in real-time on a NVIDIA Jetson AGX Xavier. Using ROS ensures a high level of compatibility with different robotics software, making the image processor ideal for use in an automted optical rocket tracker. The code is fully available on GitHub (`https://github.com/DavidBaldsiefen/rocket_tracker`) including instructions on how to reproduce the results of section 5.4.

5.1 Robot Operating System

5.1.1 General

The Robot Operating System (ROS) is an open source software framework for robotics applications. Next to an extensive set of software libraries and interface definitions, the software development kit also includes several tools for logging, debugging, visualization and more. The modular architecture facilitates fault isolation, separation of independent code blocks and clear communication interfaces between different sensors, actuators and software elements.[28, 30]

ROS supports multiple programming languages such as Python and C++, though only the latter is used in this work. This section gives an overview of the frameworks key important components.

Nodes are the core component of any ROS-based software project. Each node can be understood as an individual process that is responsible for performing a specific task. In order to allow for high parallelization and synchronization, each node spawns multiple threads responsible for receiving, storing and processing incoming messages. This requires little to no user interaction as it is mostly handled by the ROS backend.

Topics are the primary way of communication between different nodes. By creating a message *publisher*, associated with a specific datatype and unique name, nodes can broadcast

information to all other nodes on the same network, provided they have a corresponding *subscriber*. Again, communication is entirely handled by the ROS backend. It is also possible to have different nodes communicate across devices using TCP. Once a node receives a message through a topic, it is handled in a user-defined *callback* method. By default, those callbacks are protected by mutexes, meaning that only one callback on a specific topic is executed at a time. However, if a node subscribes to multiple topics simultaneously, their callbacks may be executed in parallel. Naturally, any level of parallelization is limited by the amount of threads the underlying hardware is able to execute in parallel. If a receiving node is busy while new messages are arriving in one of its subscribers, those messages are collected in a queue of user-defined length, with the oldest ones being discarded first once that length is exceeded. It is possible for multiple nodes to publish and subscribe on the same topics.

The **parameter server** is another communication interface ROS offers. However, unlike topics, it is neither optimised for performance nor large chunks of data. Instead, as the name indicates, its main purpose is to share configuration parameters between different nodes. Every parameter can be set and retrieved by any node based on a unique identifier. In addition, it is possible to load and store parameters from and to configuration files.

Launchfiles are the most convenient way of starting several nodes at once. They contain information about the nodes to spawn, as well as arguments and parameters to set initially. They are written in XML format and executed by using the `roslaunch` command line tool.

5.1.2 T-REX Node Structure

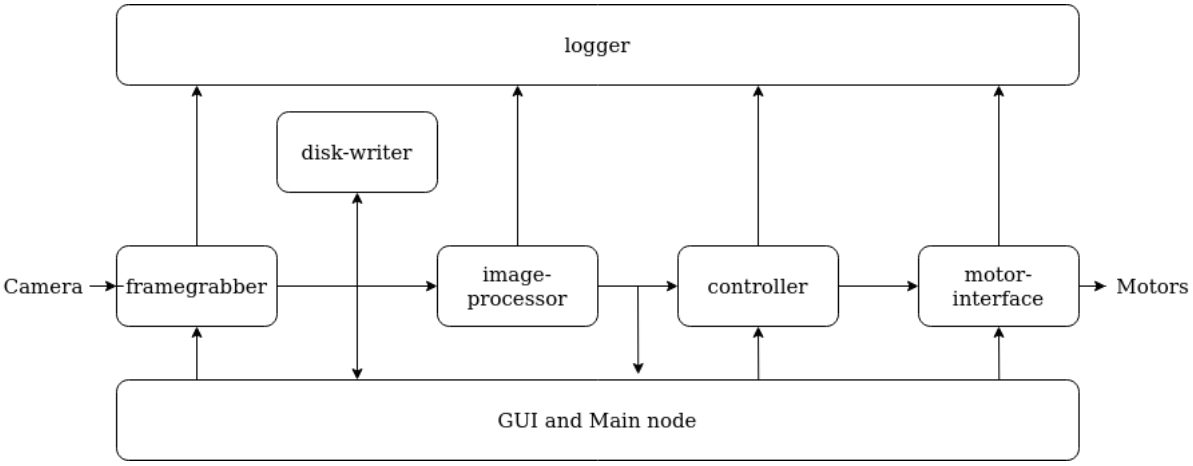


Figure 5.1: T-REX node composition. Incoming frames are processed by the framegrabber, and then published to the GUI, diskwriter and image processor. The diskwriter is responsible for saving any new images to the hard drive. The image processor is responsible for detecting rockets in the incoming images. Whenever a rocket is found, its coordinates within the frame are published to the controller node, which calculates the way the motors shall move based on the rockets current and past positions. Lastly, the motor interface communicates those new commands to the tracker motors.

As noted in section 2.1, the T-REX software is built on the ROS distribution Melodic Morenia[28]. OpenCV 4[2] serves as a library to perform most tasks related to image acquisition and processing. A modified version of cv-bridge allows ROS Melodic and OpenCV4 to compile together. The graphical user interface (GUI) was developed in the form of a rqt-plugin.

The core software of T-REX (excluding the GUI) consists of up to ten nodes. Independent from the amount of cameras used, the software always spawns one logger node, one controller node, one motor interface node and one main node. While the *controller node* is responsible for calculating the way the motors shall move next based on the rockets coordinates within the camera frame, the *motor interface node* is responsible for communicating the controllers output to the actual motors. The *main node* is managing communication and synchronization between different nodes, but is planned to become deprecated in a future release. The *logger node* logs information of the other nodes to logfiles and the console output.

Apart from those base nodes, the software spawns three nodes per input camera, with up to two cameras being supported.

The first node spawned is the *framegrabber* node. It initialises the video capture using OpenCV, gathers new frames from the capture at a predefined rate and publishes those to the other nodes using an `image_transport` publisher.

The second node is the *diskwriter* node. Its only task is to subscribe to the image topic and store frames on the hard drive.

The third node is the *image processor*. It also subscribes to the image topic, but processes every incoming frame with the goal of identifying rockets. Depending on the configuration, a Kalman filter supports the algorithm in that process. All current image processing methods are based on colour, contrast, brightness, Haar-like features or a combination of the four. Once a likely target has been identified in the image, its coordinates within the frame are published together with a confidence score and metadata such as the frame ID and timestamps.

As the software for the framegrabber and image processor is identical for each camera, this work considers single-camera setups only. The image processor developed in this chapter is loosely based on the T-REX node structure.

5.2 Testing Methodology

This section describes the testing methodology behind all performance measurements obtained on the NVIDIA Jetson AGX Xavier as part of this chapter.

In every performance test, a 480x640 pixel video is used as input. The video consists of training and validation images of the dataset from chapter 4. It is played at 50fps and looped constantly. For every processed frame, the image processor logs the latency as well as time measurements of individual processing steps. Latency is always determined based on the time difference between frame capture and detection publication. After 1000 processed frames, averages over the last 1000 measurements are calculated and printed to the console. In addition, throughput of the image processor is calculated based on the time difference between the first

and last frame of each batch. The image processor also checks incoming frames for continuity. Whenever a frame ID is skipped, the frame is considered "dropped". It is important to note that throughput is always limited by the publishing rate of the framegrabber. All time measurements are taken using `ros::Time` .

5.3 Implementation in ROS and Deployment on NVIDIA Jetson

The software developed in this chapter consists of three nodes, which are based on their T-REX counterparts: A *framegrabber* that is responsible for reading new frames from the input video, an *image processor* which is responsible for processing the images and publishing detections, and an optional *evaluator* node which can be used as a GUI to visualise the detection process. As the evaluator node is only receiving information and not at all relevant to the detection process, it is not described further in this work.

Because chapter 4 showed that TensorRT offers significant performance gains at only small losses in accuracy, neural network inference of the software is also based on TensorRT.

5.3.1 Basic Implementation

Before any programming starts, it is necessary to install and link all required libraries, which includes TensorRT 8.0, CUDA 10.2+, CuDNN 8.2, Boost 1.65 and OpenCV 4. All of these come bundled with the NVIDIA Jetpack 4.6 installed on the NVIDIA Jetson. The libraries are linked by the `CMakeLists.txt` of the rocket tracker software. In addition, it is necessary to install and compile a modified version of CV_Bridge that is compatible with OpenCV 4. One such version can be acquired at `https://github.com/DavidBaldsiefen/vision_opencv`.

The **framegrabber node** only performs one task. After loading the video from a source which can be configured by command line arguments, its main loop spins at a rate defined by the `rocket_tracker/fg_fps_target` ros parameter (default: 50). Upon every loop, a new frame is captured from the video using OpenCV, appended with a frame-ID and timestamp and then published through an `image_transport` publisher. When the end of the video is reached, the capture-position is set to the first frame again causing an infinite loop.

The **image processor node** is responsible for the actual detection process. First, a TensorRT runtime is created and initialised using the TensorRT API. This involves loading the engine file and deserializing it. Next, the input and output buffers of the engine need to be prepared so they can be accessed by the GPU to read and write data. The TensorRT API offers several interfaces through which the engine's input and output bindings can be accessed, returning information about their names, datatypes and dimensions. For example, the bindings of the base YOLOv5m engine in FP16 precision look as follows:

```
Binding "images", Datatype kFLOAT, Dimensions [1 3 640 640]
 Binding "528", Datatype kHALFS, Dimensions [1 3 80 80 85]
 Binding "594", Datatype kHALFS, Dimensions [1 3 40 40 85]
 Binding "660", Datatype kHALFS, Dimensions [1 3 20 20 85]
Binding "output", Datatype kFLOAT, Dimensions [1 25200 85]
```

The input and output bindings contain important information about the engine itself. For example, the input binding contains information about the input image format (here 3-channel 640x640 images, the ”1” denoting batch size). The output binding contains information about the class count (here 80; For each of the 25200 detections, there are four coordinates, one box confidence and 80 class confidences). It can also be seen how the input and output bindings need to be in kFLOAT format, so that they are compatible with C++ floats, even though the rest of the engine is using FP16 precision.

The buffer memory for each binding is allocated using `cudaMallocHost`. This leads to a small performance gain compared to `cudaMalloc`, as it allocates page-locked memory which in turn accelerates memory access by the GPU. The size of each buffer equals the product of every dimension of its binding (e.g. 3×640×640×sizeof(float)=4800KiB for the input).

Once everything is initialised, the node constantly listens for new incoming messages from the framegrabber, and handles all new images in the subscribers callback method. To make sure that the image processor always evaluates the newest frame, its subscribers queue size is set to one. For every new image, the following three steps need to be taken: preprocessing, model inference and postprocessing.

Preprocessing

During **preprocessing**, the image needs to be rescaled to fit into the model input shape, and then written to the input buffer. Rescaling is performed using OpenCV, and only done when either of the images' dimensions exceeds that of the model (no upscaling). During rescaling, the aspect ratio of the original image is maintained. All of the remaining preprocessing steps are combined into the following code snippet:

```
1  img.forEach<cv::Vec3b>([&](cv::Vec3b &p, const int *position) -> void{
2      // p[0-2] contains bgr data
3      // position[0-1] contains row-column location
4      // model_size contains the product of the model's width and height
5      // TensorRT expects input buffer in order [RRR...GGG...BBB]
6
7      int index = model_height * position[0] + position[1];
8      inputBuffer[index] = p[2] / 255.0f;
9      inputBuffer[model_size + index] = p[1] / 255.0f;
10     inputBuffer[2 * model_size + index] = p[0] / 255.0f;
11 });
```

Snippet 5.1: Input preprocessing

First of all, the image of type `cv::Mat` is iterated using the `forEach` expression. This is considerably faster than any other iteration method, such as pointer-access or `cv::Mat::at`, the reason being parallel execution of the lambda operator by OpenCV[1]. Specifying the reference operator "&" in line 1 also ensures that objects are passed by reference and not by value, saving additional time. Then, the buffer array is subsequently filled up with the RGB data of the image. As OpenCV stores the video frame in BGR order internally, it needs to be accessed in reverse order during this process. Again, combining these steps is much faster than using `cv::cvtColor` separately. Lastly, YOLOv5 expects the input array to contain floats in a range of 0.0 to 1.0, which means that the elements of the `cv::Mat` need to be divided by 255.

Model Inference

Once the image data is written to the input buffer, the actual **inference** is initiated on the GPU using `nvinfer1::IExecutionContext::executeV2(buffers)`. Unlike `enqueueV2`, this method call is fully synchronous, meaning it only returns once the inference is complete.

Postprocessing

After inference is complete, **postprocessing** can start on the CPU. As this call is fully synchronous, the output buffers can be accessed directly without copying the contents to a different location first, saving time. Each detection follows the following format:

`[centerX, centerY, width, height, box_confidence, n×class_confidences]`

Thus, for the default YOLOv5 models trained on the COCO dataset, each detection occupies 85 datapoints. Because the rocket tracker can always only track one object at a time, it is sufficient to determine the most likely target from all the detections, by iterating over all confidence values and selecting the highest one. This offers significant speed advantages compared to "normal" post processing, where additional steps such as non-maximum suppression need to be taken to eliminate overlapping bounding boxes. For single-class models such as the one developed in this work, the class confidence is disregarded.

Once the most likely target has been determined, its coordinates, confidence, class ID and a timestamp are collected and published by the image processor's output publisher. The detection process is now complete and the image processor ready to process the next frame. With this approach, the trained model achieves an average latency of 19.35ms at a throughput of 50fps, which means it satisfies the performance criteria of chapter 3.

5.3.2 Using Shared Memory

Stripping down the time measurements provides additional insight into potential areas of improvement. Even though the average latency is 19.35ms, the actual sum of preprocessing, inference and postprocessing is significantly lower at 17.39ms. Thus, 1.96ms can be accounted to neither of the three processing steps. Considering that only a few operations are not covered by the time measurements, this discrepancy is most likely caused by ROS' communication pipeline.

This subsection describes the attempt to circumvent this bottleneck by replacing the publisher/subscriber pipeline of ROS with a shared memory approach. Even though this goes against the core design principles of ROS, it offers two significant advantages: First, using shared memory allows both the framegrabber and image processor threads to communicate extremely fast, skipping the publisher/subscriber pipeline which contains several serialization and deserialization steps. This promises latency improvements. Second, it opens up the opportunity of performing the preprocessing asynchronously inside of the framegrabber instead of the image processor. Therefore, new frames can be preprocessed simultaneously to old frames being inferenced or postprocessed, which allows the system to work at a potentially higher throughput. It is theoretically possible to do the same for postprocessing, but the advantage would be minimal considering it already takes less than 0.1ms for the custom model.

Implementation Steps

The shared memory approach is implemented using the `boost::interprocess` library. First, two vectors are constructed inside of a `managed_shared_memory` segment, which is directly accessible by both threads. The first vector contains the preprocessed input, while the second contains meta information about the new frame, such as frame ID, timestamp and preprocessing time.

Upon every loop in the framegrabber, the newly captured frame is preprocessed and written into the image vector using the same code as in snippet 5.1. However, instead of accessing the vector elements through traditional ways such as `vector::at` or `vector::operator[]` , it is first cast to a float pointer and then accessed like a normal array. This causes a measurable speedup of around 0.5ms per frame, as all vector-based exception handling is skipped.

Last, the frame ID inside of the metadata vector is incremented by one, which informs the image processor that a new frame is ready for processing. Copying the image vectors contents into the engine's input buffer is achieved fastest by using `std::copy` . Two alternative methods were also evaluated for comparison: For the 480x640 test input, `cudaMemcpy` took an average of 0.86ms, compared to `std::memcpy` at 0.75ms and `std::copy` at 0.53ms. Once the input buffer is prepared, the image processor starts computing the new frame in the same way as before.

Despite the additional copying operation that has to be performed for every frame, the usage of shared memory reduces the overall latency by roughly 1ms, resulting in a total latency of 18.30ms for the trained model. At the same time, the potential throughput without dropping frames increases by roughly 4fps (cf. figure 5.2).

5.3.3 Adding YOLOX Support

As YOLOX and YOLOv5 share very similar input and output structures, support for YOLOX engines can be added to the software with only few modifications to the original code.

For the input buffer, the only difference is that YOLOv5 expects the RGB information in form of floats with a range of 0.0 to 1.0, while YOLOX uses floats with a range of 0.0 to 255.0. This difference is accounted for by replacing the `255.0f` divisor of snippet 5.1 with a custom float variable, which is then set to `1.0f` whenever the software is launched with the `yolox_model` launch parameter.

As for the output buffer, the only difference is that the bounding box coordinates are stored in a different way when using YOLOX. Instead of absolute coordinates, the model uses local coordinates within each predictions respective grid cell. By applying the same output decode logic that is used in the official YOLOX repository, the coordinates can be easily converted to the coordinate system used by the remaining tracker software. As this additional transformation is only necessary for the most likely bounding box, the impact on performance is rather minimal. At the same time, YOLOX-S only makes 8400 individual predictions, which is much lower than the 25200 predictions of YOLOv5s. Thus, postprocessing is performed at an overall faster rate for YOLOX (see table 5.2).

5.3.4 Things That Did Not Work

Despite the success of the shared memory approach, it was preceded by several unsuccessful attempts at improving performance, which are listed here in short detail.

Running Pre- and Postprocessing Asynchronously Inside of the Image Processor

The first attempt at increasing throughput of the image processor was to perform pre- and postprocessing asynchronously. This is achieved by using TensorRTs `enqueue` method instead of `execute`, which returns control to the CPU immediately after starting inference. The core pipeline looked as follows:

1. Upon receiving a new frame, perform preprocessing and store it in a local buffer.
2. Constantly poll the GPU and when it is available, copy the contents of the local buffer into that of the GPU to start inference.
3. Immediately after starting a new inference job, perform postprocessing of the engine's (old) output and publish the results.

In theory, this allows the GPU to work on inference for almost the entire time, with the CPU performing pre- and postprocessing in parallel. However, this approach proved to be even slower than the base approach, caused by the fact that the three steps regularly got out of sync. In addition, it did not tackle the latency issues caused by the ROS publisher/subscriber pipeline.

Combining the Framegrabber and Image Processor in a Single Node

Another attempt consisted of merging the framegrabber and image processor into a single node. The advantage of this approach is that the entire publisher/subscriber pipeline is skipped, removing the bandwidth bottleneck altogether. However, it meant that all image processing was

now performed by a single thread, which was under high load constantly. In addition, the entire thread was blocked whenever TensorRT started inference, meaning that no new frames could be captured in that time. Overall, this approach proved to be slower than the base approach, while also being unable to even *capture* new frames at a rate of 50fps. Therefore it is unfit for use in a rocket tracker.

Using the Shared Memory Segment for TensorRT Buffers

Last but not least, it was attempted to place the entire TensorRT input buffer inside of a shared memory segment. This would allow the framegrabber to write directly into the input buffer after preprocessing, making the additional call to `std::copy` obsolete. However, the technical hurdles of this approach could not be successfully overcome, as the GPU drivers are very strict about where the memory segments may be that they access. Thus, this approach is still subject of further research.

5.4 Performance of Different Models and Setups

As mentioned before, both implementations allow the trained YOLOv5m model in FP16 precision to be deployed at a latency of under 20ms while maintaining a throughput of 50fps, making them well suited for use in an optical rocket tracker according to chapter 3. The shared memory implementation even exceeds these criteria, allowing the model to run fluidly up until 58fps. The purpose of this section is to take a more detailed look at the image processor's performance when using different models at different frame rates. For that reason, tables 5.1 and 5.2 also contain measurements for the base YOLOv5m model in FP16 and FP32 precision, as well as for the small object detectors YOLOv5s and YOLOX-S. Figures 5.2, 5.3 and 5.4 visualise the performance of different models at varying framegrabber publishing rates.

5.4.1 Publisher/Subscriber vs. Shared Memory Implementation

The shared memory approach outperforms the publisher/subscriber implementation in all relevant metrics. At a framegrabber publishing rate of 50fps, latency is on average 1.19ms lower for all models. At the same time, the shared memory implementation allows for a higher potential throughput and less dropped frames across a wide range of framegrabber fps, as shown in figures 5.2 and 5.3. Likewise, the total processing time for five frames is more than 5ms faster for the trained YOLOv5m model before dropping frames. These results clearly indicate that the publisher/subscriber pipeline is a significant performance bottleneck when developing a real-time image processor in ROS.

Furthermore, the figures neatly visualise what happens when the image processor's latency exceeds the time it takes for the framegrabber to publish a new frame. As soon as this happens, it has a snowball effect on following frames, causing each consecutive frame to start and finish processing increasingly late. This effect causes a significant spike in the average latency, as seen in figure 5.2. In addition, the saturated throughput means that frames start being dropped at regular intervals. This can be imagined like two marathon runners running laps around a small

football field: When one runner is just slightly slower, the other one will end up overtaking him several times before finishing the marathon. In our case, this relates to the framegrabber publishing more frames in a given amount of time than the image processor is able to process, causing them to be dropped. Knowing the point where the latency exceeds the publishing rate is therefore essential when trying to maximise a model's real-time performance.

5.4.2 FP16 vs. FP32 Precision

As expected, all FP16 engines significantly outperform their FP32 counterparts in terms of latency and throughput. On average, the latency of half-precision engines is over 50% lower with throughput more than doubling. When considering the comparatively low loss in accuracy of around 0.5% mAP (cf. chapter 4), these results show that half-precision TensorRT engines have a much better speed vs. accuracy trade-off, making them well suited for real-time applications such as the one discussed in this work.

5.4.3 Pre- and Postprocessing

Postprocessing for the trained YOLOv5m model takes 0.06ms on average, making it significantly faster than the 2.85ms it takes to perform postprocessing for the base models. This can be explained based on the different model output sizes. The output array of the base model contains up to two million datapoints due to the higher class count, compared to only 100.000 datapoints in the custom model. Similarly, postprocessing for YOLOX-S is more than two times faster than for YOLOv5s, as it only makes 8400 predictions compared to YOLOv5's 25200. As a result, the otherwise faster YOLOv5s model is outperformed by YOLOX-S in both implementations.

While the output arrays vary greatly in size based on the amount of classes and predictions made, input arrays are of identical size and do not depend on the model used. Thus, the expected results are almost identical preprocessing times across all models for both implementations. However, that is not the case. For the shared memory approach, preprocessing of the small object detectors YOLOv5s and YOLOX-S is around 14% slower when compared to YOLOv5m. Meanwhile on the publisher/subscriber implementation, preprocessing appears consistent at around 1.0ms across all FP16 engines, whereas the FP32 engines are 70% slower at an average of 1.7ms.

Despite a lot of experiments with code and model modifications, I was unable to find the root cause of these variations. One theory is that it is related to the CPU cache being exhausted due to the larger engine size. The profiling tool *Valgrind* confirmed the timings, but did not reveal any more information about possible causes. Thus, they are subject for further research. They are especially interesting as finding the origin of the anomalies might open up room for further performance enhancements.

5.4.4 Performance of Small Object Detectors YOLOv5s and YOLOX-S

Both small object detectors easily exceed the latency and throughput requirements defined in chapter 3. In the shared memory implementation, YOLOv5s and YOLOX-S achieve a total processing time for five frames of 62.63ms and 60.26ms respectively, meaning that they are able to process up to three additional frames before reaching the time limit of 100ms. Overall, the lower postprocessing time of YOLOX-S allows it to reach a throughput of up to 84fps before dropping more than 0.2% of frames, making it 4fps faster than YOLOv5s. Considering the fact that on the COCO dataset, YOLOX-S is 4.3% points more accurate than YOLOv5s and only 2.8% points less accurate than YOLOv5m, this indicates that YOLOX-S is more suitable for use in an automated optical rocket tracker than YOLOv5s, and may even outperform YOLOv5m in scenarios that require higher speeds.

5.5 Lessons Learned

Even though the image processor developed in this chapter is already capable of running neural networks at sufficient speeds for use in an automated optical rocket tracker, there are many lessons to be learned for potential future improvements.

First of all, this chapter shows that in terms of speed, ROS may not be the ideal framework to use in an automated optical rocket tracker. Although the publisher/subscriber implementation is still able to fulfill chapter 3's performance criteria, it is consistently slower than the shared memory implementation, leading to the question whether there are additional bottlenecks hidden in the ROS framework.

Secondly, subsection 5.3.4 shows that there are many structural code modifications which can still be explored to offer additional performance gains. For example, solving the technical hurdles of placing TensorRT buffers in the shared memory would most likely lead to an immediate latency improvement of over 0.5ms, as it eliminates the need to perform an additional copying operation for every frame.

Lastly, developing this image processor made clear that it is very important to carefully examine each line of code when looking for potential performance improvements on such a high-speed level. There are many seemingly irrelevant changes which actually lead to measurable performance increases, such as iterating over vectors using pointers instead of method calls. It is likely that many more such "tricks" exist which allow the software to run even faster. Similarly, section 5.4 revealed that there is still unexpected behaviour in the preprocessing code which may be caused by such details.

Overall, the image processor developed in this work is not to be understood as a "definite solution", but more as a basic foundation upon which projects such as T-REX may improve.

Engine	Latency (ms)	PRE (ms)	INF (ms)	PST (ms)	Throughput (fps)	Dropped Frames
YOLOv5m custom (FP16)	19.35	0.80	16.54	0.05	50.0	0
YOLOv5m custom (FP32)	55.95	1.93	42.83	0.06	22.2	1251
YOLOv5m base (FP16)	33.03	1.02	18.10	2.84	45.1	108
YOLOv5m base (FP32)	61.74	1.46	45.26	2.88	20.1	1488
YOLOv5s base (FP16)	14.48	1.02	8.86	2.82	50.0	0
YOLOX-S base (FP16)	13.60	0.98	9.88	0.93	50.0	0

Table 5.1: Final performance measurements of different TensorRT engines in ROS when using the publisher/subscriber implementation. All values are averages over 1000 processed images of size 480x640.

Engine	Latency (ms)	PRE (FG IP) (ms)	INF (ms)	PST (ms)	Throughput (fps)	Dropped Frames
YOLOv5m custom (FP16)	18.30	1.64 (1.17 0.47)	16.53	0.05	50.0	0
YOLOv5m custom (FP32)	54.47	1.59 (1.08 0.51)	42.85	0.06	23.0	1175
YOLOv5m base (FP16)	32.45	1.54 (0.98 0.56)	18.17	2.86	46.2	81
YOLOv5m base (FP32)	59.66	1.55 (0.99 0.56)	45.16	2.84	20.6	1427
YOLOv5s base (FP16)	13.52	1.83 (1.29 0.54)	8.82	2.85	50.0	0
YOLOX-S base (FP16)	12.60	1.78 (1.22 0.56)	9.87	0.93	50.0	0

Table 5.2: Final performance measurements of different TensorRT engines in ROS when using the shared memory implementation. Preprocesssing times include the time spent in the framegrabber (FG) and in the image processor (IP). All values are averages over 1000 processed images of size 480x640.

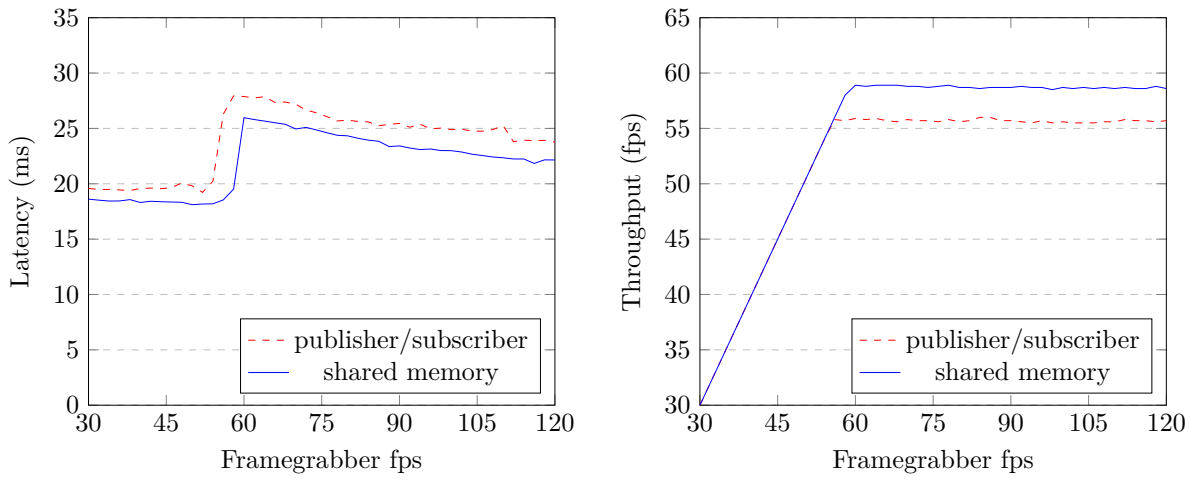


Figure 5.2: Latency and throughput of both implementations based on different publishing rates of the framegrabber. The model used is the trained YOLOv5m version in FP16 precision.

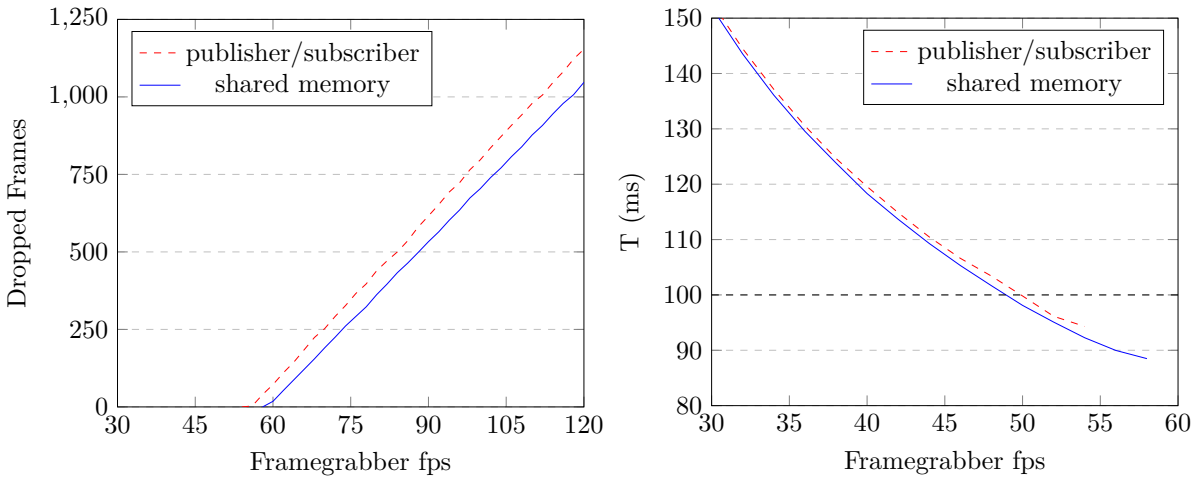


Figure 5.3: Dropped frames and total processing time for five frames for both implementations based on different publishing rates of the framegrabber. Measurements where more than 0.2% of frames were dropped are disregarded for the total processing time. The model used is the trained YOLOv5m version in FP16 precision.

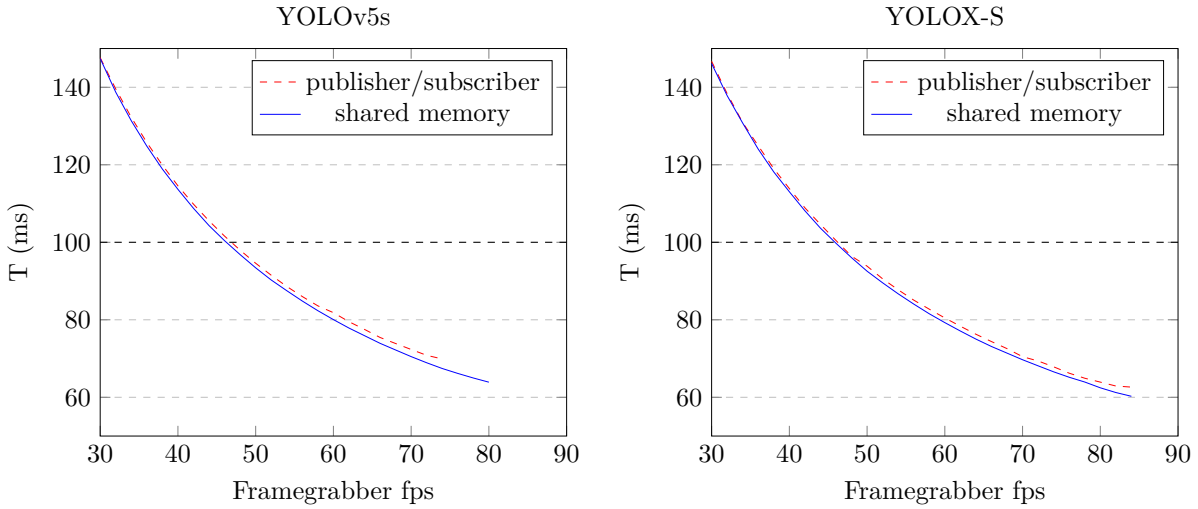


Figure 5.4: Total processing time for five frames for both implementations based on different publishing rates of the framegrabber. Measurements where more than 0.2% of frames were dropped are disregarded. On the left are results for the base YOLOv5s model in FP16 precision. On the right are results of base YOLOX-S model in FP16 precision. Everything above the dashed black line does not pass the performance requirements of chapter 3.

Chapter 6

Summary, Conclusion and Outlook

6.1 Summary

Automatically detecting and tracking rockets in real-time is a challenging task which requires a fast and accurate image processing algorithm. To make automated optical rocket tracking viable for use in model rocketry, the algorithm needs to be deployable on portable and affordable hardware platforms, such as embedded computing boards. This work investigated the viability of neural network based approaches in combination with a NVIDIA Jetson AGX Xavier.

While chapter 1 introduces the goals and motivation of the thesis, chapter 2 offers an insight into the theoretical background of rocket tracking, object detection and neural networks. It also contains a description of the NVIDIA Jetson module used in this work, and explains why it is a viable choice for an automated optical rocket tracker.

In chapter 3, performance criteria for such a tracker are defined based on common model and sounding rocket specifications. Latency and throughput are presented as the two most important speed metrics in image processing, and a formula to calculate both is derived mathematically. The chapter concludes with the finding that the image processor needs to be able to process five frames in less than 100ms, suggesting a throughput requirement of 50fps at a latency of less than 20ms.

Chapter 4 details the neural network selection and training process. At the beginning of the first half, mean Average Precision (mAP) is presented as the most common accuracy benchmark for object detection tasks. This metric is then used to select promising neural network models, which are subsequently deployed on the NVIDIA Jetson to evaluate their performance in FP16 precision. YOLOv5m proves to be the model with the best speed/accuracy trade-off, at a throughput of 54.7fps and a mAP of 43.1%. The speed-tests also illustrate the advantages of converting the models to half-precision TensorRT engines. Furthermore, the tests reveal YOLOX-S as a faster, but slightly less accurate, alternative to YOLOv5m. The second half of the chapter outlines the dataset creation and training process. First, footage from the T-REX project is used to create the training dataset, which is split into three subsets for training, validation and testing. Then, different training configurations are evaluated including varying batch sizes and hyperparameter evolution. The best results were achieved by training at batch size 16 with otherwise default configuration, leading to a final accuracy of 55.9%mAP. However,

the model's ability to correctly identify a rocket with its most confident prediction is much higher at 90.4%. This is relevant because the image processor exclusively uses the model's most confident detection. Overall, the chapter shows that extending the dataset is the most promising way of increasing the model's performance.

Finally, chapter 5 describes the implementation of an image processor capable of loading and running the trained neural network model. It is fully embedded in the ROS framework, which is introduced at the beginning of the chapter. Because ROS' communication interface turns out to be a measurable performance bottleneck in real-time scenarios, two possible implementations are proposed at the end of the chapter. The first uses the default publisher/subscriber pipeline and achieves a latency of 19.35ms at a throughput of 50fps with the trained model. The second one uses shared memory, and achieves a latency of 18.30ms in the same scenario. A comparison of different neural network models in both implementations proves that the shared memory approach outperforms its counterpart in every metric, and once again demonstrates the speed advantages of half-precision TensorRT engines. It also reveals YOLOX-S to be faster than YOLOv5s within the full detection pipeline. The small object detector is able to fully process five frames in 60.26ms, meaning it is able to process up to three additional frames before surpassing 100ms. This is especially interesting, as YOLOX-S is just 2.8% less accurate than YOLOv5m on the COCO dataset. Finally, the chapter concludes with a list of learned lessons, which point out some ways in which the proposed image processor's performance could be improved even further.

6.2 Conclusion and Outlook

The goal of this thesis is to develop a neural network based image processor which is fast and accurate enough to be used in an automated optical rocket tracker. The final implementation achieves speeds of over 50fps when deployed on a NVIDIA Jetson AGX Xavier, satisfying the performance criteria of chapter 3. Therefore, it is fast enough to be used in an automated optical rocket tracker. At the same time, the trained model correctly identifies a target rocket in over nine out of ten cases. This is most likely more accurate than any of the conventional object detection methods currently used in the T-REX project, such as colour, contrast and brightness. However, this assumption still needs to be verified empirically, and is therefore up for further research. Overall, it can be concluded that the image processor developed in this work is fast enough to be used in an automated optical rocket tracker, while also being very accurate. The usage of the ROS framework allows the software to work seamlessly with a variety of robotics systems, including the rocket tracker developed as part of the T-REX project.

Furthermore, some findings of this work already found their way into the real world: As chapters 4 and 5 show, half-precision TensorRT engines are on average more than two times faster than their FP32 counterparts while only being slightly less accurate (-0.5% points mAP). This prompted the author to submit a corresponding change request to the official YOLOv5 repository, which makes FP16 the default option while also allowing the user to customise the engine's input binding. The change request was merged into the main code base on 6th March 2022.[8]

Even though the results of the thesis are more than satisfactory, there are several approaches that can be taken to improve upon this work. For example, chapter 4 indicates that accuracy of the model can most likely be enhanced by further extending the training dataset. It also suggests the use of a custom accuracy metric that reflects the fact that the tracker can only target one object at a time. On the performance side, switching to YOLOX-S promises speed improvements of over 30fps while coming at a potentially insignificant accuracy loss. Besides upgrading the hardware, switching to a faster model is probably the most feasible way of increasing the image processor's detection speed. Looking beyond the neural network based approach suggested in this work, there are many alternatives which may be even better suited for the given use case. For example, feature trackers examine differences between consecutive images to identify moving objects, and are comparatively fast. It is also conceivable to combine different object detection algorithms, for example by using feature trackers to generate region proposals and then running classifiers on those regions. Once again, this underlines the importance of investigating the accuracy and performance of different approaches empirically.

Appendix A

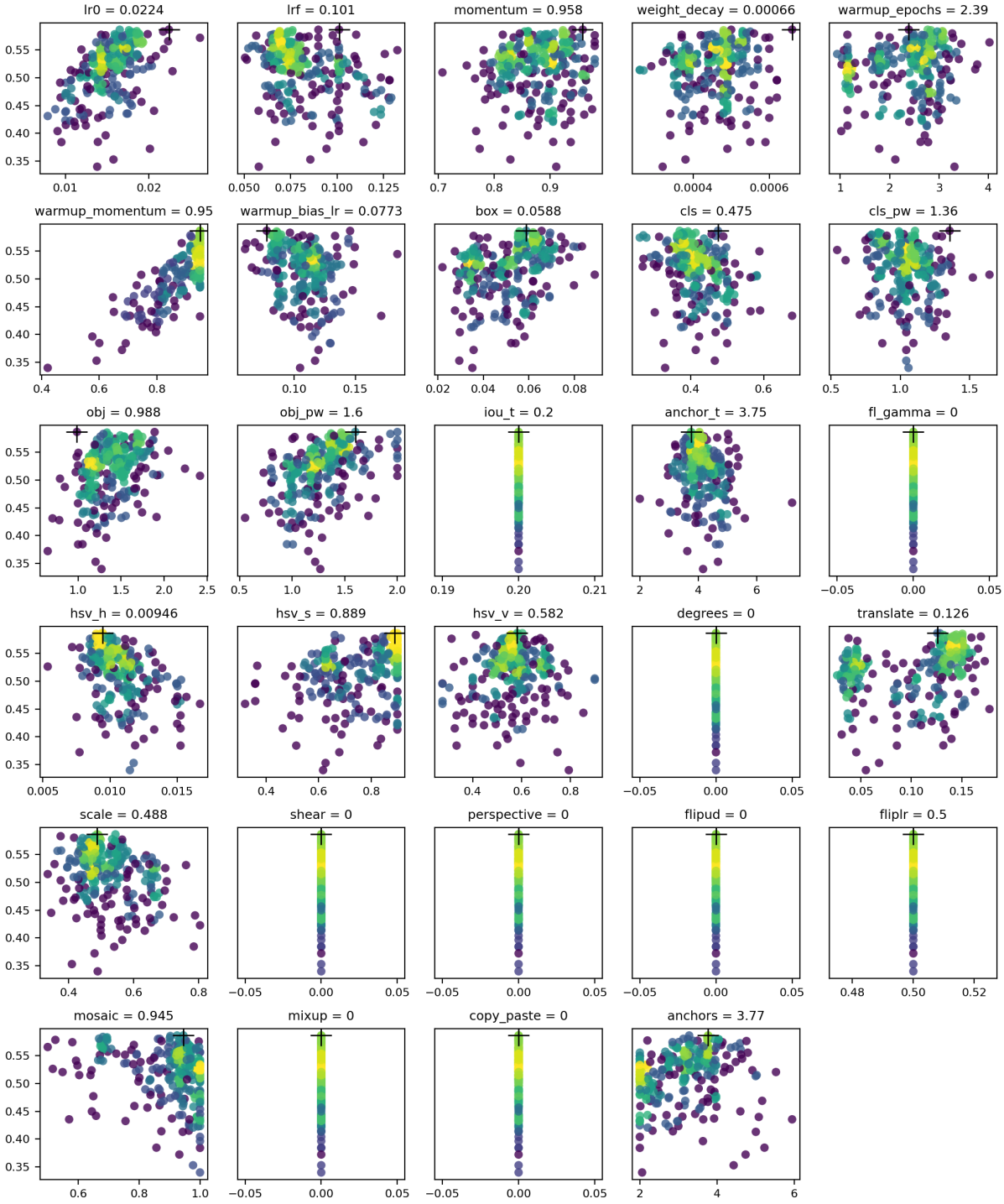


Figure A1: Hyperparameter evolution visualization, with accuracy (mAP) on the y-axis and the parameter value on the x-axis. Yellow indicates higher concentrations, while straight lines indicate that the parameter was disabled during evolution. The parameters were evolved for 300 generations of 20 epochs each.

Parameter	Default	Evolved	Description
lr0	0.01	0.02241	initial learning rate (SGD=1E-2, Adam=1E-3)
lrf	0.01	0.10105	final OneCycleLR learning rate (lr0 * lrf)
momentum	0.937	0.95843	SGD momentum/Adam beta1
weight_decay	0.0005	0.00066	optimizer weight decay
warmup_epochs	3.0	2.3886	warmup epochs
warmup_momentum	0.8	0.95	warmup initial momentum
warmup_bias_lr	0.1	0.07734	warmup initial bias lr
box	0.05	0.05885	box loss gain
cls	0.5	0.47486	cls loss gain
cls_pw	1.0	1.3565	cls BCELoss positive_weight
obj	1.0	0.98752	obj loss gain (scale with pixels)
obj_pw	1.0	1.6004	obj BCELoss positive_weight
iou_t	0.20	0.2	IoU training threshold
anchors	3	3.771	anchors per output layer
anchor_t	4.0	3.7503	anchor-multiple threshold
fl_gamma	0.0	0.0	focal loss gamma
hsv_h	0.015	0.00946	image HSV-Hue augmentation (fraction)
hsv_s	0.7	0.88938	image HSV-Saturation augmentation (fraction)
hsv_v	0.4	0.58234	image HSV-Value augmentation (fraction)
degrees	0.0	0.0	image rotation (+/- deg)
translate	0.1	0.12582	image translation (+/- fraction)
scale	0.5	0.48823	image scale (+/- gain)
shear	0.0	0.0	image shear (+/- deg)
perspective	0.0	0.0	image perspective (+/- fraction)
flipud	0.0	0.0	image flip up-down (probability)
fliplr	0.5	0.5	image flip left-right (probability)
mosaic	1.0	0.94532	image mosaic (probability)
mixup	0.0	0.0	image mixup (probability)
copy_paste	0.0	0.0	segment copy-paste (probability)

Table A1: Overview of the final evolved hyperparameters and their default values. Evolution was performed for 300 generations of 20 epochs each. Descriptions from [21].

Figure A2: Examples of labeled images in the final dataset.

Figure A3: Predictions made by the final model, together with their confidence score.

Bibliography

[1] Opencv 4.1.1 documentation. https://docs.opencv.org/4.1.1/, July 2019. Accessed February 21st, 2022.

[2] OpenCV. Version 4.x. Open source computer vision library. https://opencv.org/, 2018.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.

[4] Inc. Apogee Components. Aspire. https://www.apogeerockets.com. Accessed February 21st, 2022.

[5] Encyclopedia Astronautica. Black brant. http://www.astronautix.com/b/blackbrant.html. Accessed March 8th, 2022.

[6] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. *New advances in machine learning*, 3:19–48, 2010.

[7] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2022.

[8] David Baldsiefen and Glenn Jocher. Default fp16 tensorrt export. https://github.com/ultralytics/yolov5/pull/6798, 2022.

[9] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[10] NVIDIA Corporation. Nvidia tesla v100 gpu architecture, version 1.1. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017.

[11] NVIDIA Corporation. Jetson agx xavier series system-on-module datasheet, version 1.6. https://developer.nvidia.com/jetson-agx-xavier-series-datasheet, December 2021.

[12] NVIDIA Corporation. Nvidia tensorrt. https://developer.nvidia.com/tensorrt, 2022.

[13] NVIDIA Corporation. Nvidia tensorrt open source software. https://github.com/NVIDIA/TensorRT, 2022.

[14] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 32(9):1627–1645, 2009.

[15] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.

[16] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[17] Raketenmodellbau Klima GmbH. Pegasus rtf modellrakete. `https://www.raketenmodellbau-klima.de/`. Accessed February 21st, 2022.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] Xin Huang, Xinxin Wang, Wenyu Lv, Xiaying Bai, Xiang Long, Kaipeng Deng, Qingqing Dang, Shumin Han, Qiwen Liu, Xiaoguang Hu, et al. Pp-yolov2: A practical object detector. *arXiv preprint arXiv:2104.10419*, 2021.

[20] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574, 1959.

[21] Glenn Jocher, Alex Stoken, Ayush Chaurasia, Jirka Borovec, NanoCode012, TaoXie, Yonghye Kwon, Kalen Michael, Liu Changyu, Jiacong Fang, Abhiram V, Laughing, tkianai, yxNONG, Piotr Skalski, Adam Hogan, Jebastin Nadar, imyhxy, Lorenzo Mammana, AlexWang1900, Cristi Fati, Diego Montes, Jan Hajek, Laurentiu Diaconu, Mai Thanh Minh, Marc, albinxavi, fatih, oleg, and wanghaoyang0106. ultralytics/yolov5: v6.0 - YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support, October 2021.

[22] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[23] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[24] Estes Industries LLC. Crossfire isx. `https://estesrockets.com/`. Accessed February 21st, 2022.

[25] Xiang Long, Kaipeng Deng, Guanzhong Wang, Yang Zhang, Qingqing Dang, Yuan Gao, Hui Shen, Jianguo Ren, Shumin Han, Errui Ding, et al. Pp-yolo: An effective and efficient implementation of object detector. *arXiv preprint arXiv:2007.12099*, 2020.

[26] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[27] NASA. Shuttle launch imagery. https://www.nasa.gov/pdf/167722main_LaunchImagery06.pdf, 2006.

[28] Inc. Open Source Robotics Foundation. Robotic operating system. version melodic morenia. https://www.ros.org, 2018.

[29] Michael Plotke. 2d image-kernel convolution animation. https://commons.wikimedia.org/wiki/File:2D_Convolution_Animation.gif, Jan. 28, 2013. CC BY-SA 3.0 License.

[30] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[31] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[32] Inc. Roboflow. Roboflow blog: The train, validation, test split and why you need it. https://blog.roboflow.com/train-test-split/, 2020.

[33] Inc. Roboflow. Roboflow annotate. https://roboflow.com/annotate, 2022.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[35] Katharina Schuettauf, Rainer Kirchhartz, et al. Rexus user manual, 2017.

[36] Shrey Srivastava, Amit Vishvas Divekar, Chandu Anilkumar, Ishika Naik, Ved Kulkarni, and V Pattabiraman. Comparative analysis of deep learning image detection algorithms. *Journal of Big Data*, 8(1):1–27, 2021.

[37] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

[38] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

[39] Sandra Vieira, Walter HL Pinaya, and Andrea Mechelli. Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. *Neuroscience & Biobehavioral Reviews*, 74:58–75, 2017.

[40] Stefan Völk, Mikael Viertotak, Stefan Krämer, Simon Mawn, and Katharina Schüttauf. Rexus 24 failure investigation. 2019.

[41] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13029–13038, June 2021.

[42] Guanghua Yu, Qinyao Chang, Wenyu Lv, Chang Xu, Cheng Cui, Wei Ji, Qingqing Dang, Kaipeng Deng, Guanzhong Wang, Yuning Du, et al. Pp-picodet: A better real-time object detector on mobile devices. *arXiv preprint arXiv:2111.00902*, 2021.

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Hanoi, 11th of March 2022