École Polytechnique Fédérale de Lausanne

Conception of a rocket tracker: detect and track a sounding rocket on a live feed

by Keran Kocher

Bachelor Project Report

Examining Committee:

Prof. Pascal Fua
Project Advisor

Joachim Hugonot
Project Supervisor

EPFL CVLAB
BC 309 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 17, 2021

Abstract

The EPFL Rocket Team aims to conceive a mechanism that can follow and record the launch of a sounding rocket from the ground station. The device will be equipped with one or more cameras on a moving stand controlled by software. This bachelor's project specifically designs the software. The program must implement an efficient real-time tracking algorithm to catch the rocket on the live feed that can run on battery-powered hardware. Then, the tracking information is used to move the stand's head accordingly. This report explores the latest advances in computer vision to find a suitable solution. The code of the project is on Github `https://github.com/EPFLRocketTeam/MantisTracker`.

Contents

Chapter 1

Introduction

The EPFL Rocket Team was founded in 2016 [11] to conceive and launch sounding rockets almost entirely built by students. Year after year, they improve their rocket technology to fly better and higher. Building a rocket tracker is part of improving the ground station, the facilities that are responsible for setting the launch and receiving telemetric data from the flying rocket. Essentially, the tracker is a portable stand with a rotating head moved by motors (see figure 1.1). The software controls the motors to orient the head towards the flying rocket. Cameras and an antenna are mounted on the stand. The tracker achieves two goals: the antenna is always in the direction of the rocket and receives a signal of maximum strength. Additionally, cameras can capture high-quality footage of the flight. My work focuses on the detection and the tracking of the rocket on an image, to provide the software with the information needed to move the head correctly. I explore many technologies, mostly coming from recent advances in machine learning. The project also addresses the challenge of running the software on embedded hardware, because the rocket tracker is a portable, battery-powered unit.

Figure 1.1: 3D modeling of the tracker stand

Chapter 2

Related Work

To find potential usable work, I cover the effort already done on the topic. Both the industry, and the literature can help guide the project such that it contributes to advance the research. Unquestionably, space agencies around the world develop their tracker for their space vehicles. However, they use state-of-the-art and extremely costly hardware to track the device at a very high altitude, and thus this is out of my scope of research. Moreover, none of them publish the software and the methods used in the trackers. For further information, you can refer to the European Space Agency (ESA) and their Estrack system [12] which is one of the rare resources given by a space agency.

In Europe, a network of amateur rocketry associations is well developed. The goal is to build sounding rockets using the knowledge or skills of students and space enthusiasts. An international competition, Euroc [13], gather them for launching and comparing the different aerial vehicles. Unfortunately, none of these associations publish their work on a custom rocket tracker. There might be two reasons for that: either they do not build one or they are keeping the tracker proprietary. For instance, Wüspace is a german association that partly focuses on conceiving a tracker [34], but their project is not open-source.

Fortunately, object detection and tracking is a buzzing field in computer vision. The research is progressing at a fast pace with new methods emerging every year. Many detection and tracking algorithms will be presented in the following sections. Concerning public research on the specific challenge of rocket tracking, not much content has been published in the literature. Many publications tackle the tracking question by using radars and signal processing. The paper « Kalman Filter Embedded in FPGA to Improve Tracking Performance in Ballistic Rockets » [15] from 2013 analyses the possibility to use radars and Kalman filters to process the signal and predict the rocket trajectory. Following the same idea, « Long range radio location system employing autonomous tracking for sounding rocket » [7] is a more recent work. The publication presents a custom signal processing algorithm that controls a movable antenna. Despite those prior researches, the Rocket Team decided to rely on a camera and video processing instead of

antennas. Our goal is to capture video footage of the rocket launch and therefore image quality and framing must be optimized. Besides, the cost of capable radars or antennas is higher than the one of simple cameras and computers. In 2017, researchers explored this idea in « Rocket launch detection and tracking using EO sensor » [33] in which they display how to process a video to track missiles and rockets for a military purpose. Their approach is to track the rocket using feature matching with the ORB algorithm. They specifically detect the nose tip of the rocket and track it across the video frames. Unfortunately, their proposed method is not applicable because our viewpoint is not static.

Chapter 3

Methods

3.1 Objectives and constraints

Before diving into the experiments, we define the exact requirements we should meet. The tracker software receives the live image from the camera and outputs the position of the rocket using a bounding box. It must respect the following requirements:

- The tracker knows the position of the rocket at every frame, either by tracking or detection.
- The tracker is robust against the noise in the image such as sun, clouds, birds, etc.
- The tracker can recover if the rocket is lost.
- The tracker is fast enough, namely achieves enough frames per second (fps), to follow the rocket when launching.
- The tracker can run on the given hardware, that might be limited.

The software uses a combination of object detection and tracking methods to achieve the best accuracy and speed possible given the hardware limitations.

3.2 Object detection

The object detection's goal is to find and delimit a given object on an image. For each object we need to detect, the framework must be trained with an annotated set of object images. The annotation process consists of drawing bounding boxes around the object and assign them a label, the name of the object's class. In our case, we detect only one class: a rocket.

In the past years, the research on object detectors has been very active, and thanks to that interest, the choices are numerous. The detection methods improve their accuracy constantly in a tough competition, but the accuracy often comes at a performance cost. Indeed, the best methods are often very deep networks with many layers and weights that need a powerful GPU to run in real-time. The table 3.1 compares 4 popular detectors from the past two years: EfficientDet [30], YOLOv4 [4], Faster R-CNN [32], and MnasFPN [8]. The speed is corresponding to the inference time on a NVIDIA V100, expect for MnasFPN which is a detector that targets mobiles and is an order of magnitude faster than the others. To compare their performance, the average precision on the COCO dataset [9] is computed.

	EfficientDet	YOLOv4	Faster R-CNN	MnasFPN
AP	55.1	43.5	42.4	26.1
Speed	6 fps	62 fps	8.6 fps	Mobile

Table 3.1: Comparison between popular object detectors

The average precision is defined as following. Each detection is given a score using intersection over union (IoU) which is the ratio of the intersection of the predicted box with the ground-truth box and the union of both.

$$IoU = \frac{intersection}{union}$$

Given the IoU score, a threshold is defined to accept or reject the predicted box. To measure the performance, we compute the precision and the recall. Precision is the ratio of true positive and the total number of predicted positives. It represents the reliability of a positive sample. Recall is the ratio of true positive and the ground truth positives. It represents the ability to detect a positive sample.

$$Precision = \frac{True_{positive}}{True_{positive} + False_{positive}}$$

$$Recall = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

There is a natural trade-off when maximizing precision and recall. Therefore, the precision-recall curve can be used. Precision is on the y-axis, recall on the x-axis and their value for thresholds between 0 and 1 is plotted (figure 3.1). On top of that, the average precision (AP) summarizes the recall and the precision with the area under the curve which is often approximated with k points:

$$AP = \sum_{k} P(k)\Delta R(k)$$

YOLO is a famous object detector, existing since 2016 and being at its fifth-generation (YOLO v5). YOLO uses a one-pass method that predicts bounding boxes and assigns class probabilities with a single convolutional network. Previous methods like R-CNN would have a two-pass

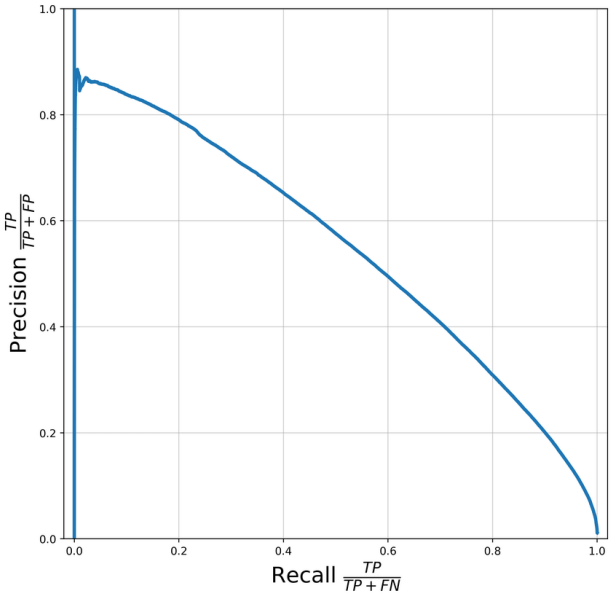

Figure 3.1: An example of precision-recall curve

algorithm: first a region proposal and then a classifier for the proposed boxes. YOLO and its approach enables a similar accuracy while boosting its speed, as we can observe in the table 3.1. We choose to train a YOLO detector that would detect sounding rockets and that could be used for two purposes in the project: the initial detection in the video to start tracking the rocket and recover the rocket if the tracker fails to track it.

3.2.1 Training

We choose to train the YOLOv4 detector which was published in 2020 [4]. The implementation is found in the Darknet repository [1]. The most challenging task is the creation of the training dataset, because it will determine the accuracy of detection in the real-world setting. In our case, the dataset is particularly difficult to gather, because the object is not common. From the start, we know that it would be impossible to collect thousands of sounding rocket's pictures and that our dataset would be rather hundreds of images. Therefore, the images must be carefully chosen to reflect every aspect of encountered situations when operating the tracker. Thankfully, the EPFL Rocket Team maintains an archive of pictures and videos through the years and it contains footages of launches and rockets in-flight.

I explore the content and select 113 training pictures for the first training of YOLO. I also select 17 images to form a testing dataset. The images are manually annotated using the Computer Vision Annotation Tool (CVAT) [28]. CVAT is a web-based annotation application that let you import your collection, annotate it, and export the data in various popular formats including the YOLO one. The configuration, the dataset, and the annotations can be found in the Github

repository. There are many settings to determine when training a network. The most impacting ones are the resolution of the network, which I choose to be 416x416, and the number of iterations which is 6000. Here are the results after more than a day of computation. The algorithm converges with a mean average precision (mAP) of 96%. YOLO computes the mAP as defined by PascalVOC competition [14]. It is the mean of the average precision of each classes. I also compute the accuracy of the test dataset. Rockets can correctly be detected on 9 images out of 17, resulting in a 52% accuracy. It must be noted that the threshold of the probability to be a rocket is set at 50%.

Figure 3.2: A sample of outputs from the test set

I draw the following conclusions from the first training. First, the chosen YOLO convolutional network is complex enough to suit our pictures, which means we do not need to head for a more complex detector or backbone. On the other hand, the training dataset does not reflect all situations a rocket can be found in, explaining the poor result on the testing dataset. The performance of the detector suffers from the small size of the set. To mitigate the lack of universality, I analyze the successes and the failures to extract which type of images we need to get better coverage. I define 3 categories:

- A rocket on its launch rails. The rocket is vertical, but there could be a lot of background noise (the rail, people, the landscape, etc).

- A rocket right after its take-off. This is the easiest part, as the rocket is still vertical, but the background, the sky, tends to be simpler. The difficulty comes from the smoke the rocket emits.
- End of flight. The rocket is far away in the sky and is reaching its maximum altitude. Thus, the rocket becomes smaller and is not necessarily vertical. The bounding boxes are rectangles, such that when a rocket is skew the ratio between the background and rocket is very high. In this phase, the tracker will eventually lose the spacecraft.

From these conclusions, I form a new training dataset containing 150 images. Apart from adding new pictures to make sure that each of the described situations are well covered, I also take existing ones and manually rotate them to have rockets with different orientations in the sky. Another important consideration is the size of the object relative to the image. I must ensure that the set covers a range of scales that match the application settings in real condition. Lastly, pictures without any rockets are added.

The second training is done using the same settings as the first, but with the new dataset. The results are rather encouraging. YOLOv4 achieves an mAP of 100% in the training phase, reached after around 2400 iterations out of 6000. The outcome of the testing set sounds about the same: 100%. Following this satisfying training on YOLO, I also train a Tiny YOLOv4 detection network. Tiny YOLOv4 is a simplified version of the YOLOv4 network, enabling it to run much faster while keeping a solid accuracy (in theory). Tiny YOLOv4 is worth trying in our case as the tracker will be run on limited hardware, namely an embedded platform. The training achieves a mean average precision of 94%, marking a net difference with the full YOLO. The tiny detector successfully finds 13 rockets out of 15 in the testing set (86%). These two detectors can be tested on any image using the tool in the Github repository.

3.3 Tracking

As for the detectors, tracking methods are numerous and evolving quickly thanks to active research. The explosion of trackers makes the work of finding a suitable algorithm challenging. Therefore, we select a few trackers from distinct subfields to compare their performance in solving our problem. A repository [16] created by Qiang Wang containing up-to-date classification and benchmarks of tracking methods guided my selection.

3.3.1 Correlation filters

Trackers based on correlation filters have existed for more than a decade and are still developed today despite the rise of deep learning. Basically, the goal is to train a filter such that for each frame, the filter is correlated over a search window, and then, the location corresponding to

the maximum value of the correlation indicates the new position of the target. Nowadays, their principal advantage is their speed thanks to their relatively simple computation in comparison to deep learning. They can often run in real-time (> 24fps) on modern CPUs and are simply deployable. On the other hand, their capability is limited. For instance, they cannot recover if the object disappears, which is inherent to the technics used. Robustness is also a weakness: changes in the background, lighting variation, or cloud/smoke are all potential failure reasons for the tracker. This problem could be mitigated by using a detector in pair with the tracker to recover in case of a lost target. To test this category, I select the MOSSE tracker [5] which is an old (2010) but a quick method. I used the OpenCV implementation. OpenCV [27] stands for Open Computer Vision and is a popular, large, and open-source library of tools. The library is present in most computer vision projects as it includes tools to read, write, and modify images/videos as well as detectors, trackers, classifiers, etc. Thanks to OpenCV, deploying a MOSSE is a matter of minutes. The MOSSE tracker can be easily run on any video using the developed tool on Github along with other OpenCV trackers.

3.3.2 SiameseFC networks

As for detectors, a few years ago deep learning networks also began to be deployed to solve the tracking challenge. Lots of architectures have been developed since and the SiameseFC family is one of them. The siamese networks were introduced in the 1990s [6] and consist of comparing a template image with another image to determine if the images are alike. The outputs of the two input images are computed using the same network with the same weights, and the training is such that similar images have similar outputs. In the case of tracking, an initial frame is fed to the tracker. Then, given the next frame, the tracker proposes possible regions where the object could be located and these are compared to the template image, using a siamese network. Finally, the region with the highest score is chosen. The first proposal was released in 2016 (SiameseFC) [3] and ever since the family keeps growing with enhancements in the region proposals, for instance. The particularity of the method, which makes it relatively fast while being very accurate, is the offline training. Traditionally, the features of the target would be learned online, while tracking it. On the other hand, SiameseFC is trained beforehand and the tracking only needs to infer with the given trained weights. Overall, SiameseFC is a very accurate method, while still being able to run in real-time on a GPU. It is more robust and accurate than the MOSSE tracker presented in the previous section. To deploy such a tracker, the « SenseTime » team provides a repository [13] which provides implementations of many Siamese trackers in Pytorch. They also supply the pre-trained models. Thus, it can be used out of the box. This library is included in my program and the SiamRPN tracker [22], which is the fastest of the available ones, can be tested against any video.

3.3.3 Others networks

More sophisticated networks for tracking do exist. Notably, ATOM [10] is a tracker that aims to improve the target estimation. Its main difference is that ATOM introduces a classification component that is trained online to better understand the object. This method effectively results in a state of the art precision, while trading off the inference speed. I choose not to integrate this tracker, because running a SiameseFC tracker, which is faster, in real-time is already a challenge on limited hardware.

3.3.4 Comparison

MOSSE and SiamRPN, as well as the KCF tracker [18] (also found in OpenCV), are tested on a few videos to compare their output. The results can be found in the repository. The figure 3.3 shows 6 frames of one video. Each of the trackers exhibits different behavior. MOSSE and KCF tend to correctly bound the rocket at the beginning. On the contrary, SiamRPN is deceived by the smoke and bounds the rocket and its trace. Nonetheless, the siamese tracker always perfectly bounds the target, resizing and adapting as the rocket flies away. KCF looses the rocket and fails as soon as the smoke occludes the camera, while MOSSE keeps tracking the target but its bounding box becomes very imprecise. On the whole, MOSSE and SiamRPN do track successfully until the end. Although SiamRPN set an erroneous target, the method behaves smarter across the video and it might indicate a greater potential in unfavorable situations. The second comparison in figure 3.4 shows that KCF fails again while the other two display the same behavior as the first comparison. The inference time of the trackers will be compared in the following section.

3.4 Hardware

One critical choice for the software design is the hardware on which it will run. The tracker software is embedded on a portable tripod that can be deployed anywhere for a rocket launch, which is often on an isolated site. Therefore, the computer must be an embedded platform powered by a battery, limiting the range of possible hardware. One obvious choice would be a single-board computer like a Raspberry Pi, which carries a quad-core ARM processor. However, choosing such a solution eliminates the use of any deep learning algorithm, as they need a GPU to hopefully operate at a reasonable speed. Thankfully, due to the rise of deep learning in many fields, vendors started to ship specialized embedded hardware with ML accelerators. A typical use case is computer vision for autonomous cars (e.g Tesla). I select and compare three of them from the biggest actors in the field: Intel, Google, and NVIDIA.

Intel actively pushes its architecture to run and accelerate machine learning models as the market is demanding. They develop their framework OpenVINO [21] to optimize and deploy

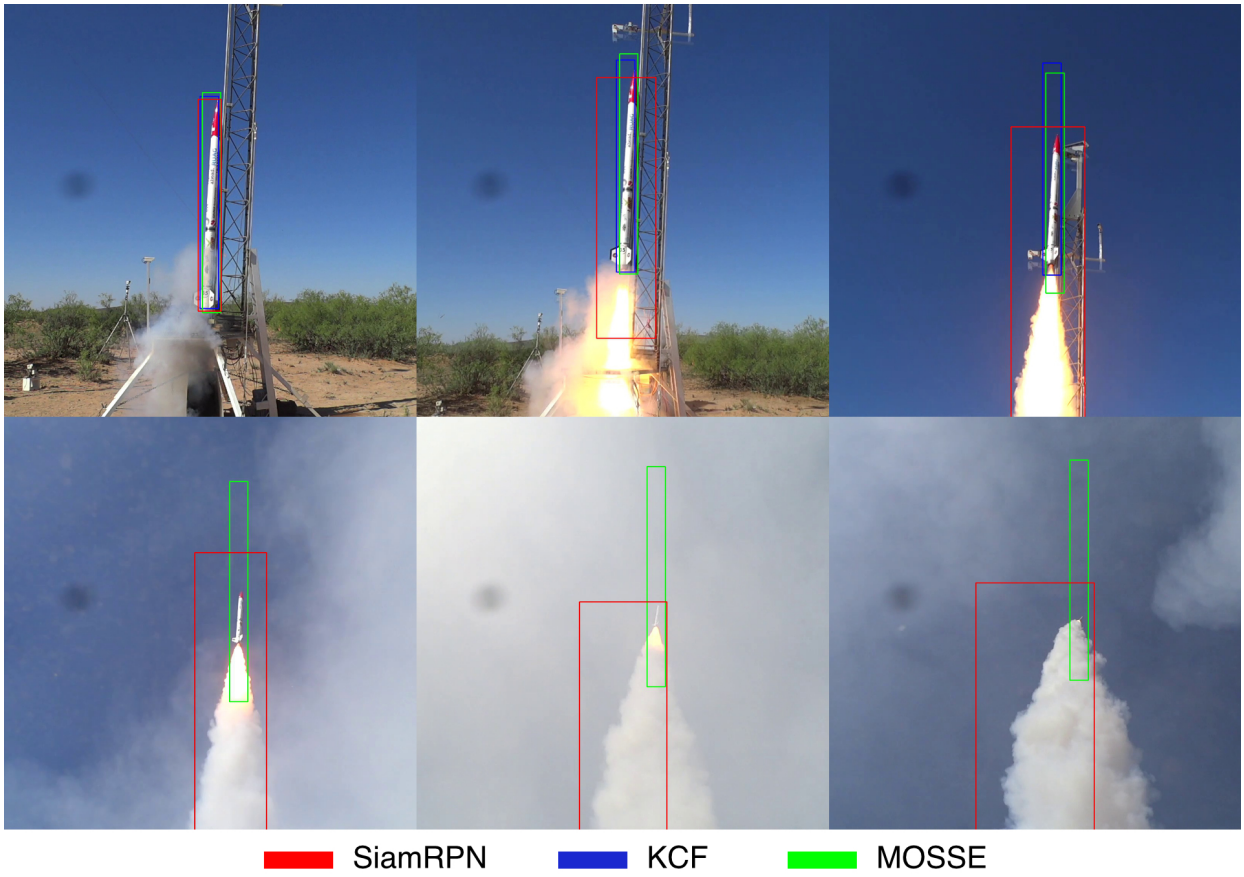

Figure 3.3: First comparison of SiamRPN, KCF and MOSSE

ML engine on their platform. Along with OpenVINO, they offer specific hardware, going from their 5000$ FPGA to a 100$ USB stick, the Intel Neural Compute Stick 2 (Intel NCS2) [20]. The latter is particularly interesting for its ease of deployment. The NCS2 stick contains a deep learning inference accelerator that can be plugged and used on any computer, even a Raspberry Pi. OpenVINO offers to convert existing models from many popular frameworks (ONNX, Caffe, Tensorflow, etc.) to run them efficiently.

Google is prominent and at the edge of machine learning researches. They created Tensorflow in 2015, which is presently one of the most famous libraries in the field. Similar to Intel, Google released their Coral products [17] which range from a 25$ PCI accelerator, a 60$ USB accelerator to a 100$ board. However, Google's hardware only understands Google's framework Tensorflow, which is a huge limitation, even if Tensorflow is well-developed and already includes many tools.

Finally, NVIDIA is also a serious contender. Their GPU and CUDA technology is unanimously used to train networks across the globe, which places them at the front when considering hardware for computer vision. They manufacture the Jetson family [23], a line up of boards to address various needs in the industry. It starts with the Jetson Nano at 100$ to the Jetson AGX Xavier

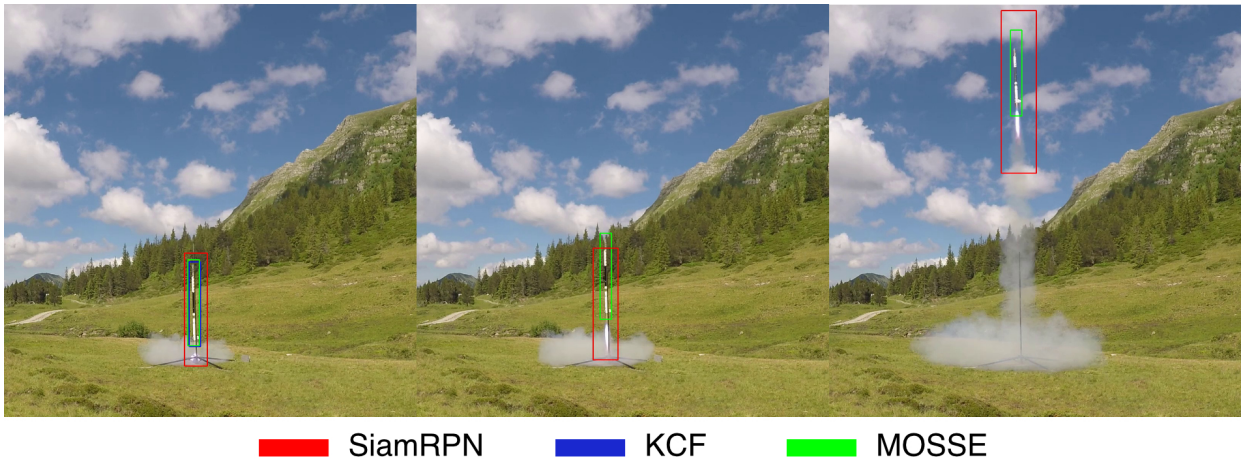

Figure 3.4: Second comparison of SiamRPN, KCF and MOSSE

at 1000$. The boards embed each a resized NVIDIA GPU, which has from 128 CUDA cores to 512 cores. In comparison, an NVIDIA GTX 1080 has 2560 of them. These boards sharing the classic NVIDIA architecture and CUDA enable flexibility when executing software. Software that executes on a desktop platform can in theory also run on a Jetson without much additional work, at the difference that the CPU is ARM-based. However, NVIDIA also develop a framework TensorRT [25] which aims to optimize model inferencing on their hardware. Like Intel's OpenVINO, TensorRT supports to import existing models or one can define a model from scratch.

Each of the presented solutions has its pros and cons. First of all, I eliminate Google Coral's hardware, as its dependency on Tensorflow is too limiting. We do not know which particular models we will be using for the tracker yet. Therefore it could be frustrating to stick to Tensorflow because we aim to test as many methods as possible. Then, deciding between Intel and NVIDIA is a tough decision. NVIDIA provides benchmarks (figure 3.2 that compares the Jetson Nano to Intel NCS2 (and accessorily to Google's Coral) on various object detection networks. The numbers give a clear advantage to NVIDIA's board that runs at least twice faster. Even considering that NVIDIA exposes situations that favor its hardware, the gap is large enough to declare them as the clear winner. On a side note, other independent benchmarks confirm this result.

Overall, the Jetson Nano is adopted for the following reasons:

- Performance: inference is faster on the Jetson Nano
- Compatibility: NVIDIA's CUDA technology is widely supported without additional work
- Support: NVIDIA's toolkit, support, documentation, and contribution to the field make it the perfect partner
- Camera support: the Jetson boards integrate MIPI CSI-2 interface. Their website advertises MIPI as "the most widely used camera interface in mobile and other markets" [2]. Practi-

	Jetson Nano	Intel NCS2	Google TPU
Tiny YOLOv3 (416x416)	25	DNR	DNR
Mobilenet-V2 (480×272)	64	30	130
ResNet-50 (224x224)	36	16	DNR
DNR = did not run (hardware or support limitations)			

Table 3.2: NVIDIA comparison of Jetson Nano, Intel NCS2, and Google's TPU board [24]

cally, one can easily add various camera modules (sensor, lenses) which can be read and processed out of the box by the board.

Here are the main characteristics of the Jetson Nano: 128-core NVIDIA Maxwell, Quad-core ARM A57 @1.43 GHz, 4 GB of memory, 2x MIPI CSI-2 connectors with 4Kp30 encoding and 4Kp60 decoding.

Chapter 4

Results

4.1 Settings

Setting up the Nano board is a straightforward task. NVIDIA provides a custom OS built on top of Ubuntu which includes the necessary drivers, CUDA toolkit, and TensorRT. However, one should be careful when installing libraries, because the CPU architecture is ARM. Thus, the packages must be built from source most of the time. This was the case for OpenCV for example.

I benchmark the previously mentioned methods, object detectors, and trackers, on the following hardware:

- Desktop CPU: 2x Intel Xeon E5-2650v2 2.60GHz (2x8 cores)
- Desktop GPU: Geforce GTX Titan Z (5760 CUDA cores)
- Mobile CPU (Jetson Nano): ARM A57 @1.43 GHz (4 cores)
- Mobile GPU (Jetson Nano): NVIDIA Maxwell (128 CUDA cores)

The following methods are tested: YOLOv4, Tiny YOLOv4, MOSSE, KCF, SiamRPN

For object detectors, I obtain the number of frames by second (fps) by measuring the average inferencing time on 15 different images. For trackers, they are applied on a video lasting from 5 to 20 seconds and the total time is used along with the number of frames in the footage. These results can be seen in the table 4.1. They can be recomputed and verified by using the provided program, which has also a benchmark mode.

	Detectors		Trackers		
	YOLO	Tiny YOLO	MOSSE	KCF	SiamRPN
Desktop CPU	0.7	3.2	145	45	3
Desktop GPU	7	10	145	45	30
Mobile CPU	0.05	0.5	80	20	1.3
Mobile GPU	1.15	3.5	80	20	7

Table 4.1: Benchmark results (in fps)

4.2 Comments

The benchmark results mostly confirm our appreciation. First of all, deep learning models are dramatically affected by the use of a GPU, even a small one. SiamRPN runs at a low 3 fps on a top-class CPU but achieves an impressing 50 fps on a GPU and is even more than 2.5x quicker on the Jetson Nano with 7 fps. Concerning the built-in OpenCV trackers (KCF, MOSSE), the GPU makes no difference, as they do not use it. Consequently, the performance directly depends on the number of cores and the clock speed of the CPU. In any case, the Jetson can run these in real-time (or almost). A more surprising result is the runtime of YOLO. Even on a desktop GPU, we do not reach the number promised by the YOLOv4 framework. We temporarily conclude that it must be due to the benchmarking method, the structure of the tool, and the Python language introducing an overhead. Nevertheless, the numbers can be compared relatively: while the desktop GPU marks a clear difference compared to others, the Nano board has no clear advantage over the desktop CPU. Both are far from reaching real-time frames per second. At this point, our rocket tracker can merely use a YOLO detector to detect the initial frame for the tracker. The latter must be at this point a MOSSE or a KCF tracker because they are the only ones running with at least 24 fps (with a few tweaks). However, this combination of methods does not satisfy the robustness requirement, as we have no means of recovery if the tracker fails.

4.3 Optimization

TensorRT is NVIDIA's framework to define machine learning models that target their hardware. TensorRT can help accelerate the inference time of YOLO or SiamRPN on the Jetson Nano. There are two distinct ways to create a TensorRT model. First of all, one can build a network from scratch using their API, which can be done with C++ or Python. Secondly, existing models can be imported from popular formats. I choose the second method as it is easier, quicker, and requires less knowledge of the model you're importing. YOLOv4 already has many TensorRT implementations. I use the tkDNN library [31] which provides a tool to export a trained YOLO network and operate it using TensorRT. The export of YOLOv4 does not work on the Jetson Nano because of hardware resources limitiation, but I successfully obtain a TensorRT engine for Tiny

YOLOv4. Singularly, the performance is satisfying. The inference runs at 26 fps and the rocket is captured on most of the frames if the threshold is low enough. The outputs can be seen in the repository. Having achieved such numbers, I would like to also apply these optimizations to the tracker. However, I did not find such implementation of SiamRPN or any Siamese tracker to this day. Therefore, the goal is to convert the SiamRPN network to a TensorRT model, to determine if, in the end, it can be used in our rocket tracker software. The SiamRPN implementation used in the project is done with PyTorch. Unfortunately, TensorRT provides no tool to parse directly from PyTorch. Instead, the framework can import ONNX models. ONNX stands for Open Neural Network Exchange [26] and is an open format that encodes ML models. Its goal is to be a universal format for developers to share their work and thus help collaboration and research. ONNX is supported by many tech actors such as Facebook, Microsoft, or Intel. Its companion ONNX Runtime makes it easy to run any ONNX model on numerous platforms including the Jetson family. The first step is to export the PyTorch model to ONNX. Following their documentation, the workflow looks like the following: import a PyTorch model, load the trained weights, and invoke the ONNX export tool with the model and a dummy input. The tool defines the model by tracing the network: it observes the input and the applied transformations until the output. The ONNX model can then be visualized as a graph. The Pytorch SiamRPN can not be exported out of the box, as it is composed of two subnetworks, a backbone (an AlexNet) and an RPN head. Thus, I extract them from the main model, inspect the needed input dimension and both are exported independently. I encounter no particular difficulties apart from the backbone model, which accepts two different input dimensions. The models and the export tool is available in the repository. The models are executed using ONNX runtime and yield similar performance, as expected. As an advantage, one can run the SiamRPN tracker by only installing ONNX runtime and skip the other dependencies. The next step is to convert from ONNX to TensorRT. In theory, it is as straightforward as converting to ONNX: load the model and use the provided parser. The backbone is successfully converted to a so-called TensorRT engine. However, the RPN head runs into a problem when parsing: the parser encounters an unsupported operation. To understand the issue, we have to understand the head network structure. On a given input, two outputs are computed from two subnetworks, named search and kernel. Then, a cross-correlation is computed with search as the input and kernel as the kernel. Finally, the last output is computed by using the cross-correlation output and another subnetwork. The figure 4.1 shows an illustration of the network, which is doubled (classification and regression).

TensorRT does not support convolution with multiple dynamic inputs, which is precisely what the cross-correlation step is. TensorRT is a relatively new framework, a lot of features are still missing and only a few resources are available. No solution on NVIDIA's forum or the Github issues were found to circumvent this incompatibility.

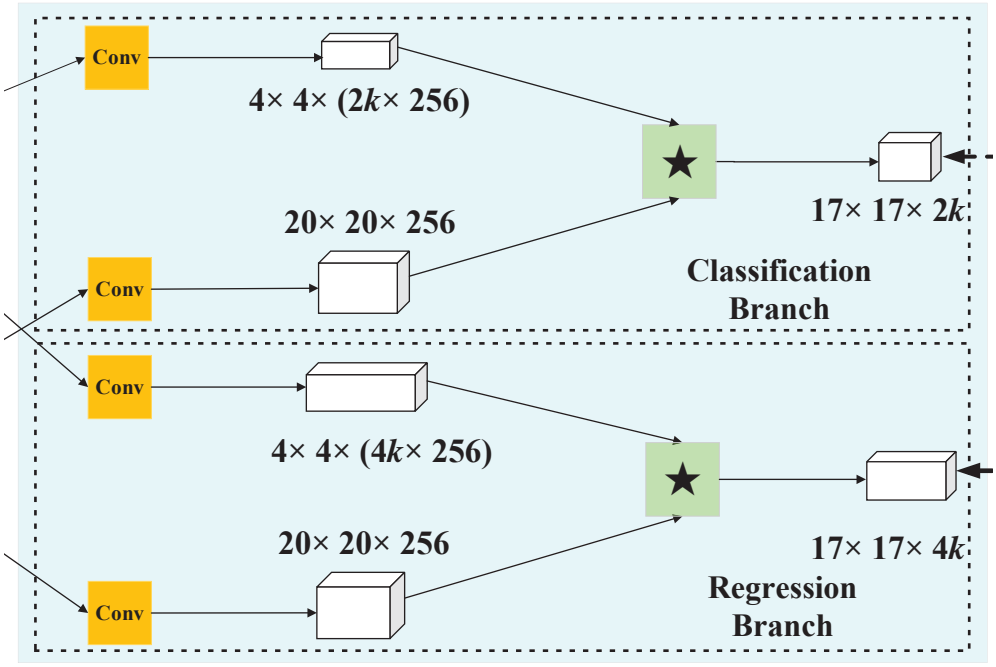

Figure 4.1: Diagram of the RPN head [22]

Chapter 5

Discussion

In summary, the project explored many paths to solve the tracking problem. For detection, I only focused the research on the YOLO method. The Darknet framework was convenient. It offered the necessary tools to train a model and facilitate the customization of the network (e.g YOLO vs Tiny YOLO). The training using a very small number of images (150) proved that YOLO is an object detector that can reach first-class accuracy. However, neither the full nor the tiny network did meet our speed requirements on limited hardware at first. Thankfully, TensorRT allowed a Tiny YOLO to run in real-time (26 fps) on the Jetson Nano board. The reduced network comes at the cost of accuracy and we need extended testing with a rocket launch and the stand to assess the feasibility. A comparison with other lightweight detectors could worth the effort, YOLOv3 [29] or MobileNet-based detectors [19].

Unlike detectors, I compared more trackers, because they are easily deployable. OpenCV implements many trackers and deep learning ones are pretrained, such that they can be used out of the box. Again, the trade-off is between the accuracy and the inference speed. Pysot framework provides the trained models and SiameseFC networks including SiamRPN. SiamRPN offers great performance because the method handles well smokes, clouds, occlusion, rescaling, and direction variations. However, SiamRPN only operated at 7 fps on the board, which is not sufficient. On the other hand, KCF and MOSSE can be adopted for the software. In particular, the MOSSE tracker yields surprising results despite its age: the framing is better than SiamRPN and tracking does not fail at any time in the example videos. The speed leap achieved when converting Tiny YOLO to TensorRT is promising. SiamRPN could be optimized to obtain a similar boost and thus eventually meet the requirements.

A key lesson of the experimentation is the importance of hardware. With the actual knowledge of the hardware and the possible optimizations (e.g. TensorRT), the selection of detectors/trackers could be more oriented. For instance, I could have verified that a given deep learning framework had a TensorRT implementation. The results indeed proved that optimization is a necessary step to use deep learning models on the Jetson Nano. Anyhow, the rise of deep

learning on mobile platforms is still at an early stage and companies like NVIDIA actively push the research in this field. The capability of TensorRT grows rapidly. Thus, solutions to our current problem may appear in the following months. For now, the rocker tracker software can work with the following settings: the MOSSE algorithm tracks the rocket and the Tiny YOLO detector recovers the target in case of failure and readjusts the framing periodically.

Chapter 6

Future work

The report focused on the computer vision challenge that an overall rocket tracker stand poses. A lot of work remains to deploy the software. Namely, the program must process the input, the bounding boxes around the rocket, and then control the stand's motors to follow the rocket as smoothly as possible. The interaction between a mechanical piece and software is itself a difficult task. The hazards are numerous and it must be properly handled by robust software.

Obviously, the detection and tracking can be improved and many ways were already mentioned. The dataset of rockets could be grown to be more effective and reflect as many situations as possible. Furthermore, more detectors could be trained and compared in terms of speed and accuracy with YOLO. The new methods must definitely be compatible with TensorRT. As for trackers, the research on converting SiameseFC networks to NVIDIA's framework can continue. Furthermore, more testing is necessary to compare the studied possibilities regarding the criteria previously set. The software needs to be assessed with more videos, and on the tracker stand with actual rocket launches.

Bibliography

[1] AlexeyAB. *Darknet on Github*. URL: https://github.com/AlexeyAB/darknet/. (accessed: 14.01.2021).

[2] MIPI Alliance. *Camera Serial Interface 2*. URL: https://www.mipi.org/specifications/csi-2. (accessed: 14.01.2021).

[3] Luca Bertinetto, Jack Valmadre, Joao Henriques, Andrea Vedaldi, and Philip Torr. "Fully-Convolutional Siamese Networks for Object Tracking". In: vol. 9914. Oct. 2016, pp. 850–865. ISBN: 978-3-319-48880-6. DOI: 10.1007/978-3-319-48881-3_56.

[4] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[5] David Bolme, J. Beveridge, Bruce Draper, and Yui Lui. "Visual object tracking using adaptive correlation filters". In: June 2010, pp. 2544–2550. DOI: 10.1109/CVPR.2010.5539960.

[6] Jane Bromley, James Bentz, Leon Bottou, Isabelle Guyon, Yann Lecun, Cliff Moore, Eduard Sackinger, and Rookpak Shah. "Signature Verification using a "Siamese" Time Delay Neural Network". In: *International Journal of Pattern Recognition and Artificial Intelligence* 7 (Aug. 1993), p. 25. DOI: 10.1142/S0218001493000339.

[7] Tomasz Chelstowski, Karol Dobrzyniewicz, Przemyslaw Kant, and Jerzy Michalski. "Long range radio location system employing autonomous tracking for sounding rocket". In: May 2018, pp. 400–403. DOI: 10.23919/MIKON.2018.8405237.

[8] Bo Chen, Golnaz Ghiasi, Hanxiao Liu, Tsung-Yi Lin, Dmitry Kalenichenko, Hartwig Adams, and Quoc Le. "MnasFPN: Learning Latency-aware Pyramid Architecture for Object Detection on Mobile Devices". In: Dec. 2019.

[9] *Common objects in context COCO*. URL: https://cocodataset.org/#home. (accessed: 14.01.2021).

[10] Martin Danelljan, Goutam Bhat, Fahad Khan, and Michael Felsberg. "ATOM: Accurate Tracking by Overlap Maximization". In: June 2019, pp. 4655–4664. DOI: 10.1109/CVPR.2019.00479.

[11] *EPFL Rocket Team*. URL: https://epflrocketteam.ch/fr/. (accessed: 14.01.2021).

[12] *Estrack ground stations*. URL: https://www.esa.int/Enabling_Support/Operations/ESA_Ground_Stations/Estrack_ground_stations. (accessed: 14.01.2021).

[13] *Euroc European Rocketery Challenge*. URL: https://euroc.pt. (accessed: 14.01.2021).

[14] Mark Everingham, Luc Van Gool, Christopher Williams, John Winn, and Andrew Zisserman. "The Pascal Visual Object Classes (VOC) challenge". In: *International Journal of Computer Vision* 88 (June 2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[15] J.V. Fonseca, Roberto Oliveira, J.A.P. Abreu, Ernesto Ferreira, and Madson Machado. "Kalman Filter Embedded in FPGA to Improve Tracking Performance in Ballistic Rockets". In: Apr. 2013, pp. 606–610. ISBN: 978-1-4673-6421-8. DOI: 10.1109/UKSim.2013.149.

[16] Foolwood. *Benchmark results on Github*. URL: https://github.com/foolwood/benchmark_results. (accessed: 14.01.2021).

[17] Google. *Coral AI*. URL: https://www.coral.ai. (accessed: 14.01.2021).

[18] Joao Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. "High-Speed Tracking with Kernelized Correlation Filters". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37 (Apr. 2014). DOI: 10.1109/TPAMI.2014.2345390.

[19] Andrew Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017).

[20] Intel. *Neural Compute Stick 2*. URL: https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html. (accessed: 14.01.2021).

[21] Intel. *OpenVINO*. URL: https://www.intel.com/content/www/us/en/internet-of-things/openvino-toolkit.html. (accessed: 14.01.2021).

[22] Bo Li, Junjie Yan, Wei Wu, Zhu Zheng, and Xiaolin Hu. "High Performance Visual Tracking with Siamese Region Proposal Network". In: June 2018, pp. 8971–8980. DOI: 10.1109/CVPR.2018.00935.

[23] NVIDIA. *Jetson modules*. URL: https://developer.nvidia.com/embedded/jetson-modules. (accessed: 14.01.2021).

[24] NVIDIA. *Jetson Nano: Deep Learning Inference Benchmarks*. URL: https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks. (accessed: 14.01.2021).

[25] NVIDIA. *TensorRT*. URL: https://developer.nvidia.com/tensorrt. (accessed: 14.01.2021).

[26] *ONNX*. URL: https://onnx.ai. (accessed: 14.01.2021).

[27] *OpenCV*. URL: https://opencv.org. (accessed: 14.01.2021).

[28] OpenVINOtoolkit. *CVAT on Github*. URL: https://github.com/openvinotoolkit/cvat. (accessed: 14.01.2021).

[29] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: (Apr. 2018).

[30] Mingxing Tan, Ruoming Pang, and Quoc Le. "EfficientDet: Scalable and Efficient Object Detection". In: June 2020, pp. 10778–10787. DOI: 10.1109/CVPR42600.2020.01079.

[31] Micaela Verucchi, Gianluca Brilli, Davide Sapienza, Mattia Verasani, Marco Arena, Francesco Gatti, Alessandro Capotondi, Roberto Cavicchioli, Marko Bertogna, and Marco Solieri. “A Systematic Assessment of Embedded Neural Networks for Object Detection”. In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 937–944.

[32] Jingdong Wang, Sun ke, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. “Deep High-Resolution Representation Learning for Visual Recognition”. In: Aug. 2019.

[33] Chee Wong and Oleg Yakimenko. “Rocket launch detection and tracking using EO sensor”. In: Apr. 2017, pp. 766–770. DOI: 10.1109/ICCAR.2017.7942801.

[34] *WüSpace T-REX2*. URL: https://www.wuespace.de/trex/trex2. (accessed: 14.01.2021).