

Conceiving a rocket tracker

Abstract

The overall task aims to conceive a mechanism that can record the launch of a rocket from the ground station. It will be equipped with one or more cameras on a moving support controlled by a software. This project specifically designs the software. The program must implement an efficient real-time tracking algorithm to follow the rocket on the camera. It must also receive metrics from the rocket such as speed, altitude, etc. Using those information, the software moves the camera accordingly.

Introduction

The EPFL Rocket Team was founded in 2016 [0] to conceive and launch soundrockets 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 head moved by motors. The software controlled the motors to orient the head to the flying rocket. Many cameras and an antenna are mounted on the stand. The tracker achieves two goals: the antenna being always in the direction of the rocket, it receives a signal of maximum strength. Additionally, the cameras can capture a high quality footage of the flight. This project focus on the detection and tracking of the rocket on an image to provide to the software the information needed to move the head correctly. I explore many technologies, mostly coming from the 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.

Related work

Unquestionably, space agencies around the world develop their own tracker for their space vehicles. However, they use state-of-the-art and extremely costly hardware in order to track the device at very high altitude and thus are out of our scope of research. Moreover, none of them publish the software and the methods used in the trackers. For further information, you can refer to European Space Agency (ESA) and their Estrack system[1] which is one of the rare resource 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. A international competition, Euroc[2], 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 don't build one or they're keeping the tracker proprietary. For instance, Wüspace [3] is a german association that partly focus on conceiving a tracker, 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 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" [4] 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"[5] is a more recent work. The publication presents a custom signal processing algorithm that controls a movable antenna. Despite those prior researches, we decided to rely on a camera and video processing instead of antennas. Our goal is to capture a 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" in which they display how to process a video footage 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 nosetip of the rocket and track it across the video frames.

Methodology

Objectives and constraints

The tracker software receives the live image from the camera and output the position of the rocket using a bounding box. It must respect the following requirements:

- The tracker follows the rocket in the image.
- The tracker is robust against the noise in the image such as sun, clouds, birds, etc.
- The tracker is able to recover if the rocket is lost.
- The tracker is fast enough, namely achieve 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.

Finding an object detection algorithm

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

In the past years, the research on object detector has been very active and thanks to that interest, the choices are numerous. The detection methods improve their accuracy constantly in a though competition, but the accuray 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 in order to run in real time. YOLO is a famous object detector existing since 2016 and being at its fifth generation (YOLO v5). YOLO uses a one-pass method which predict bounding boxes and assign class probabilities with a single convolutional network. Previous methods like R-CNN would have a two-pass 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. We choose to train a YOLO detector that would detect sounding rocket and that could be use for two purposes in the project:

- The initial detection in the video to start tracking the rocket
- Recover the rocket if the tracker fails to track it

Training

We choose to train the YOLOv4 detector which was published in 2020[6]. The implementation is found the darknet repository [7]. 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 knew 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 should be carefully chosen to reflect every aspect of real world situation in which the tracker would operate. Thankfully, the EPFL Rocket Team maintain an archive of pictures and videos through the years and it contains footages of launches and rockets in-flight.

We seeked through the content and selected 113 training pictures for our first training of YOLO. We also selected 17 images to form a testing dataset. The images were manually annotated using the Computer Vision Annotation Tool (CVAT)[8]. CVAT is a web-based annotation application that let you import your collection, annotate it and export in various popular format including the YOLO one. The configuration, the dataset and the annotation can be found in our Github repository. The most impacting setting is 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 converged with a mean average precision (mAP) of 96%. On our test dataset, rockets could be detected on 9 images out of 17, resulting in a 52% accuracy. To be noted that the threshold of the probability to be a rocket is set to 50%. We 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.

We analyzed the successes and the failures to extract which type of images we needed to get a complete coverage. We defined x categories:

- A rocket on its launch rails. The rocket is vertical, but there might 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 background rocket is very high. In this phase, the tracker will eventually lose the rocket.

From these conclusion, I formed a new training dataset containing 150 images. Apart from adding new pictures to make sure each of the described situations was well covered, I also took existing ones and manually rotated it to have rockets with different orientations in the sky. Another important consideration is the size of the object relatively to the image. I must ensure that the set covers a range of scales that match the application settings in real condition.

The second training was done using the same settings as the first, but with the new dataset. The results were rather encouraging. YOLOv4 achieved a mAP of 100% in the training phase, reached after around 2400 iterations of 6000. The outcome of testing set sounds about the same: 100%. Following these satisfying training on YOLO, I also trained 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 ran on limited hardware, namely an embedded platform. The training achieved an mean average precision of 94%, marking a net difference with the full YOLO. The tiny detectors successfully found 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. Follow the instruction in the README.

Finding a tracking algorithm

As of the detectors, tracking methods are numerous and evolving quickly thanks to the active research. The explosion of trackers makes the work of finding a suitable algorithm challenging. Therefore, we selected a few trackers from distinct subfields to compare their performance solving our problem. A repository [12] created by Qiang Wang containing up-to-date classification and benchmarks of tracking methods guided my research.

Correlation filters

Trackers based on correlation filter have existed since 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 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 techniques 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 object. To test this category, I selected the MOSSE tracker [9] which is an old (2010) but quick method. I used the OpenCV implementation. OpenCV [10] stands for Open Computer Vision and is a popular, large and open source library of tools. The library can be found in most of the 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.

SiameseFC networks

As for detectors, 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 (Bromley et al., 1993) and consist of comparing a template image and 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) [11] 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 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 tool and the SiamRPN tracker, which is the fastest of the available ones, can be tested against any video.

Others

More sophisticated networks for tracking exists. Notably, ATOM [21] is 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.

Comparison

Both the MOSSE and the SiamRPN were tested on a few videos to compare their output. These results can also be found in the repository. As anticipated, the MOSSE often fails to track the rocket across the whole footage while SiamRPN has no problem following the rocket, even with smokes or sun rays, or if the rocket leaves the frame. Their respective speeds will be compared in the following section.

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 are often on 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 run 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 selected and compared three of them from the biggest actors in the field: Intel, Google and Nvidia.

Intel actively push its architecture to run and accelerate machine learning models as the market is demanding. They developed their framework OpenVINO [14] to optimize and deploy ML engine on their platform. Along with OpenVINO, they offer specialized hardware going from their 5000\$ FPGA to a 100\$ USB stick, the Intel Neural Compute Stick 2 (Intel NCS2) [15]. 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.).

Google is prominent and at the edge of machine learning researches. They created Tensorflow in 2015 which is now one of the most famous library in the field. Similarly to Intel, Google released their Coral products [16] which ranges from a 25\$ PCI accelerator, a 60\$ USB accelerator to a 100\$ board. However, Google's hardware only understand 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 place them at the front when considering hardware for computer vision. They manufacture the Jetson family [17], 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 at 1000\$. The boards embed each a resized NVIDIA GPU, which have from 128 CUDA cores to 512 cores. In comparison, a NVIDIA GTX 1080 have 2560 of them. These boards sharing the classic NVIDIA architecture and CUDA enables flexibility when executing software. A software that execute on a desktop platform can in theory also run on a Jetson without much work, at the difference that the CPU is ARM based. However, NVIDIA also developed a framework TensorRT[18] which

aims to optimize model inferencing on their platform. Like Intel's OpenVINO, TensorRT supports to import existing models or one can define one from scratch.

Each of the presented solutions have their pros and cons. First of all, I eliminated Google Coral's hardware as its dependency to Tensorflow is too limiting. We do not know which exact models will be using for the tracker yet and 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 thourger decision. NVIDIA provides benchmarks [19] which compares the Jetson Nano to Intel NCS2 (and accessorily to Google's Coral) on various object detection networks. The numbers gives a clear advantage to NVIDIA's board that runs at least twice faster. Even considering that NVIDIA chose situations that favor its hardware, the gap is large enough to give them as clear winner. On a side note, other independent benchmarks confirm this result. Overall, the Jetson Nano was chosen for the following reasons:

- Performance: models run faster on the Jetson Nano
- Compatibility: NVIDIA's CUDA technology is widely supported without additional work
- Support: NVIDIA's toolkit, support, documentation and its contribution to the field make it the perfect partner
- Camera support: the Jetson boards integrate MIPI CSI-2 interface. From their website, "MIPI CSI-2 is the most widely used camera interface in mobile and other markets" [20]. Practically, one can easily add various camera modules (sensor, lenses) which can be read and processed out of the box by the board.

Here the main characteristics of the Jetson Nano:

- 128-core NVIDIA Maxwell
- Quad-core ARM A57 @1.43 GHz
- 2 GB of memory
- 1x MIPI CSI-2 connector with 4Kp30 encoding and 4Kp60 decoding

Result

Settings

Setting up the Nano board was 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 architecure is ARM. Thus, the packages must be built from source most of the time. This was the case for OpenCV for example.

I benchmarked the previously mentionned 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 were 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 recomputed and verified using the provided program which has also a benchmark mode.

Comments

The benchmarking results mostly confirmed our appreciation. First of all, deep learning models are really affected by the use of a GPU, even a small one. SiamRPN runs at a low 3 fps on a top class CPU but achieved an impressive 50 fps on a GPU and was 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 number of cores and 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 could merely use a YOLO detector to detect the initial frame for the tracker which must be a MOSSE or a KCF tracker because they are the only ones running with at least 24 fps (with a few tweaks). This does not satisfy the robustness requirement, as we have no means of recovery if the tracker fails.

Optimization

TensorRT is NVIDIA's framework to define machine learning models that target their hardware. TensorRT can help us accelerate the inference time of YOLO or SiamRPN on the Jetson Nano. They 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 used with C++ or Python. Secondly, existing models can be imported from popular formats. I chose the second method as it is easier, quicker and requires less knowledge of the model you're importing. YOLOv4 already has many TensorRT implementations, TODO. However, I found no 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 at 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 from PyTorch. Instead, the framework can import ONNX models. ONNX stands for Open Neural Network Exchange [22] and is an open format that represents ML models. Its goal is to be an universal format for developers to share their work and thus help the 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 was 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. This tool works by tracing the network which follows the input and the applied transformation applied until the output. The ONNX model can then be visualized as a graph. The PyTorch SiamRPN could not be exported out of the box as it is composed of two subnetworks, a backbone (an AlexNet) and a RPN head. Thus, I inspected the input dimension needed and both were exported independently from the main model. I encountered no special 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 were executed using ONNX runtime and yielded similar performance, as expected. As an advantage,

one can run the SiamRPN tracker by installing ONNX runtime. The next step is to convert from ONNX to TensorRT. In theory, it is as straightforward converting to ONNX: load the model and use the provided parser. The backbone was successfully converted to a so-called TensorRT engine. However, the RPN head ran into a problem when parsing: the parser encountered 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 `search` as input and `kernel` as the kernel. Finally, the last output is computed using the cross-correlation output and a last subnetwork. TensorRT does not support convolution with multiple dynamic inputs, which is precisely what the cross-correlation step is. TensorRT being a relatively new framework, a lot of features are still missing and only a few resources are available. No solution on the NVIDIA's forum or the Github issues were found to circumvent this incompatibility.

Discussion

Future work

[0]<https://epflrocketteam.ch/fr/>

[1]https://www.esa.int/Enabling_Support/Operations/ESA_Ground_Stations/Estrack_ground_stations

[2]<https://euroc.pt> [3]<https://www.wuespace.de> [4] [5] [6]<https://arxiv.org/abs/2004.10934>

[7]<https://github.com/AlexeyAB/darknet/> [8]<https://github.com/openvinotoolkit/cvat> [9] [10]

[11]<https://arxiv.org/pdf/1606.09549v2.pdf> [12]https://github.com/foolwood/benchmark_results

[13]<https://github.com/STVIR/pysot> [14]<https://www.intel.com/content/www/us/en/internet-of-things/openvino-toolkit.html> [15]<https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html> [16]<https://www.coral.ai> [17]<https://developer.nvidia.com/embedded/jetson-modules>

[18]<https://developer.nvidia.com/tensorrt> [19]<https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks> [20]<https://www.mipi.org/specifications/csi-2> [21]<https://arxiv.org/abs/1811.07628>

[22]<https://onnx.ai>