Deep Learning 2 - Project

Interaktionstechnik und Design
HSHL Sommersemester 2020

Bruno Berger, MatrNr: 2170706

22. 06. 2020

Contents

1 Introduction

This paper is about the Object Detection project for the elective subject Deep Learning 2 in the "Interaktionstechnik und Design" course. The project was worked on from April to June 2020. This text is a report about the research that was conducted into the topic, the concept that was created, the requirements that were set and the final implementation and some decisions behind it.
The source code and all its stages can be found in the GitHub Repository. [1]

1.1 The Task

The goal of this project was to develop a tool that detects objects and then saves their GPS-coordinates with a timestamp. This type of system could be used to improve the autonomous driving systems in cars, although the task was not solely centered on this scenario.
An alternative use case could be projects that try to map objects in a big area, like mapping animal stock or movement, maybe with footage from a drone.

During the phase of implementing the system, a second goal was added. The user should be able to display the detected objects on a map, to visualise their GPS-position.

Due to the restrictions around COVID-19 at the time and the semester being a pure online-semester, it was clear from the beginning, that this project would only consist of a software-part.
Otherwise, it would have been extended to a hardware-part to implement the scenario of a car driving around and detecting its surrounding objects.

This meant, that there would be no real GPS-module, attached to something like a Raspberry Pi, to get the actual GPS-coordinates. So it was also clear, that this part of the task would have be replaced by a software simulation.

The structure of the project was partly laid out, by additional submissions that were due along the way, that helped organise the project and provided status reports to the professors.
First, research into projects with similar goals was to be done. Based on that, formal requirements and a concept were to be developed. All these submissions and the source code should be done as commits to a GitHub Repository, as a way to train using version control.

Additionally to this paper, a presentation on the project will be held.

[1] https://github.com/BrunoBerger/DeepLearning2

2 Research

The initial research into the field focused on basic object detection and the various ways to achieve it. This research quickly lead to the blog posts of Adrian Rosebrock on his website PyImageSearch. Rosebrock writes about various topics around Computer Vision and Deep Learning, using the OpenCV-Library for Python. They are mostly aimed at beginners regarding the subject, with examples to code along. One of his blogs had already helped in a "Blink-Detection" project in the previous semester.
These blogs later provided the foundation for the Object-Detection-part of the project, most notably, the blog about real-time object detection.[2]

2.1 Related Works

After the initial look into the basics of object detection, research was done into similar works, that had similar goals or aspects as this project. Surprisingly, there were few projects with the exact same premise. Most of the time, their goal was simply to detect objects, and not to catalog them.

Many of the projects about mapping objects were not using real-time video for their object detection, but rather existing images from sources like Open Street Map or Google Street View. But they also faced a lot of the same issues and had solutions, that could also be useful for a real-time detecting- and mapping-project.

For example, researchers from Dublin proposed a new triangulation process. As an example, they were checking if the official maps for traffic lights and telegraph poles matched the real world positions they detected from Google Street View.[4]
Instead of using the GPS-position of the car, they calculated the exact position of the masts. This was achieved by using the depth information of multiple images from different positions, as visualised in Figure 1.

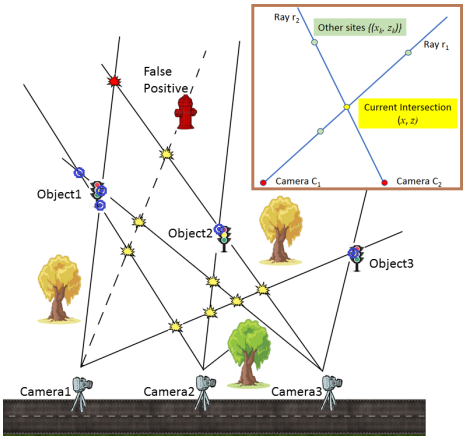

Figure 1: Triangulation [4]

Further information on this method and more related works can be found in the Wiki of this projects GitHub-repository. [3]

[2] https://www.pyimagesearch.com/2017/09/18/real-time-object-detection-with-deep-learning-and-opencv/
[3] https://github.com/BrunoBerger/DeepLearning2/wiki/Related-Works

A little bit of research was also done into the simulation of GPS-data. There was of course always the way of generating two totally random numbers, but there was also a project that simulated the coordinates of a car driving around.[4] It worked by giving the package two locations and then it periodically returned coordinates along a route, that was generated by Google Maps.

2.1.1 Conclusions

One thing that connected all the object detection projects was the use of Python. Python one of the most or the most popular programming languages, but it is even more widely used in the Machine and Deep Learning fields.[6]

Reason for that is partly the simplicity of the syntax, but also the gigantic number of packages, that are easy and free to use.[6]
Another commonality was the use of the "YOLO" object detector. It seemed to be the newer model, that was currently taking the place of older object detectors.

2.2 Different Object Detectors

During the research into deep learning based object detection, a few different models made reoccurring appearances in other projects. The following section will now describe a few of the major ones and take a look at their operating principles, as well as their benefits and weaknesses.
The eventual models that were used in the implementation and the reason for choosing them, will be discussed later, in Section 4.3.

2.2.1 R-CNN and Faster R-CNN

R-CNN's were one of the first deep learning based detectors, proposed first in 2013.[12][2] They are *Two-Stage*-Detectors. This means that the process of detecting objects in a image is split into two separate actions.

In the first step, the regions of interest are extracted from the image, delivering a rough "bounding box", which could contain an object. Then, each of the bounding boxes are run through a classifier, to detect an object that the model was trained on.[7]
This means that the image has to be re-sampled two times, which makes two-stage detectors generally more accurate, but very slow.

Multiple versions of R-CNN have been developed over the years, each improving the accuracy and performance. First versions of R-CNN even relied on an external algorithm to deliver the initial bounding boxes.[12] This was removed in the latest version "Faster R-CNN", proposed in 2015.[10] Even with all the incremental improvements, the performance of Faster R-CNN is still too poor for real time video throughput, only managing around seven frames per second.[11]

[4] https://github.com/hashedin/gps-simulator

2.2.2 YOLO

"YOLO", as mentioned in Section 2.1.1, is very popular and modern detection model. It stands for "**Y**ou **O**nly **L**ook **O**nce" and was initially developed in 2015, by researchers from Washington University.

As the name suggests, it is a one-stage detector. This means that, as depicted in Figure 2, both detection steps are done at the same time for each S input square.

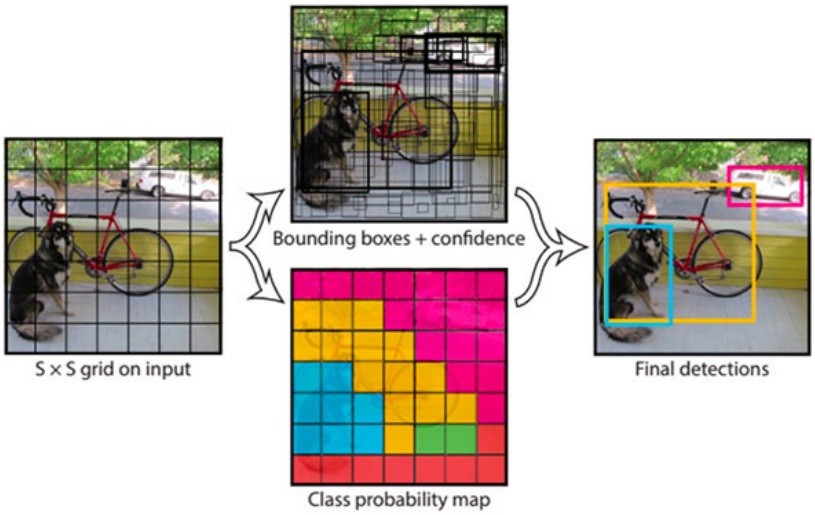

Figure 2: YOLO model design [9]

This, plus the fact that the system approaches the detection as a regression problem, means that the performance of YOLO is much better compared to R-CNN models.[9] Using a Titan X GPU, the model was able to achieve a throughput of 45 frames per second.[9] The drawback is the much lower accuracy, compared to two-stage detectors like Faster R-CNN.[12]

YOLO has also seen multiple updated or slightly adjusted versions, up to YOLOv3, released in 2018[8] and at the time of research, the newest version. Since then YOLOv4 has been released, and is promising drastic improvements to accuracy and speed.[1]
Version 2, called YOLO9000, was capable of detecting 9000 different classes, although with a much weaker accuracy.[12]
For the versions 1 and 3, a "tiny"-version was also released, promising up to triple the performance[11], in trade off for accuracy.

2.2.3 SSD

The "Single Shot Detector" or "SSD", was developed in 2016, by Google researchers. Like the name heavily suggests, it is also a one-stage detector, and promised to be "easy to train and straightforward to integrate"[5].

It achieved an accuracy somewhere between R-CNN's and YOLO and even better performance than YOLO.[11] The drawback was a significantly smaller pool of detectable objects.

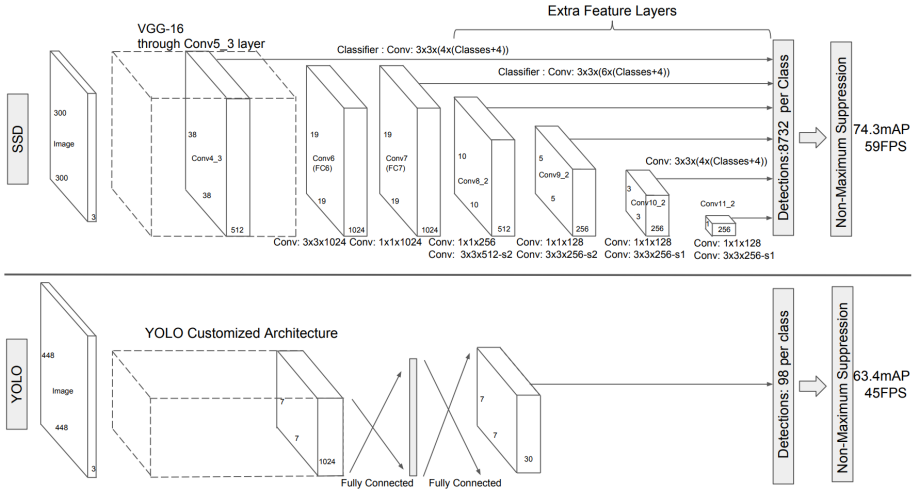

Figure 3: Comparison between YOLO and SSD [5]

Figure 3 shows how in SSD, several feature layers are added to the end of the base network.[5] Those layers try to predict the offset of the scale and aspect ratio of the initial bounding boxes.[5] This method outperformed the YOLO model in accuracy, even while using a lower resolution of 300 x 300 pixels, while also improving speed.[5]

To get to the final model, that was used in the final version of this project, "MobileNets" have to be introduced first:

Normally, object detection models are build on an existing architecture, but the commonly used architectures are not that resource efficient. This is a problem for a lot of real-world applications, where the resources on the device are constrained, like on smartphones or systems like a Raspberry Pi.
For those situations, Google researchers in 2017 developed a new architecture, called "MobileNet".[3] In short, they changed the convolutional layer, where the input gets read into a feature map, to be much faster but less accurate.[11] For an in-depth explanation, see the paper [3] and this video-explanation[5].

[5]https://www.youtube.com/watch?v=T7o3xvJLuHk

This architecture was combined with a SSD detector, by Zehao Shi to create the "**MoblileNetSSD**" model.[6] This model was then implemented for the Caffe deep learning framework and trained on the COCO dataset, by GitHub user chuanqi305.[7] It can detect 20 different objects and has a mean average precision of 72.7%.[11]

3 Concept and Requirements

Next, it was time to create a concept and formal requirements for the implementation. Both were due as a delivery, committed to the Wiki of the GitHub repository.[8]

Parallel to the research, small test implementations of the examples from the blogs of Adrian Rosebrock were done, which helped shape the concept and getting a grasp on the boundaries of what was reasonable to achieve. Those first implementations were written in Python, using the computer-vision package "Open CV" to handle the detection model. Because of that, this was continued for the rest of the project.

As mentioned in Section 1.1, the project only consisted of a software part, so the program would only be running on a PC or Laptop. This system would have to be equipped with either an integrated or external webcam, to deliver the live video feed.
As also mentioned, there would be no GPS-module on the system, meaning that the coordinates of the objects would have to be simulated. A more elaborate simulation, like the one presented in Section 2.1, was considered, but ultimately dismissed. It was simply unnecessary effort, for no real functional benefit. Although it would have presented the possible use case of a autonomous car nicely.

A concept for the process of visualising the detected objects on a map was also needed. First, all the objects that were wanted would be detected and saved. Afterwards, a map of this data would be generated in a separate step. For this, both the Python package "matplotlib" and the JavaScript library "Leaflet" were considered as possible solutions for the task.

As a potential way of storing the data from the detections, CSV files were chosen. They are a relatively easy format to read and write, lightweight, and understandable just by looking at them. Although not fully standardised, they are supported by almost all systems that are used to handle datasets. This could be usefully, if a user wants to process the data in their own way.

[6] https://github.com/Zehaos/MobileNet
[7] https://github.com/chuanqi305/MobileNet-SSD
[8] https://github.com/BrunoBerger/DeepLearning2/wiki/Concept and
https://github.com/BrunoBerger/DeepLearning2/wiki/Requirements

3.1 Choice of detection model

The initial choice of object detector was based on the first test implementations. There, real-time-video object detection was tested with both a YOLO and a MobileNetSSD model. They clearly showed the difference in performance and detection quality. The MobileNetSSD model had sometimes difficulties to detect objects properly, but the test PC, equipped with relatively modern hardware, struggled to get a smooth framerate with YOLO.
So it was decided, that the user would be able to chose, which of those two models they want to use. This way, they could adjust the detector according to their systems resources and their need for accuracy.

3.2 Requirements

Based on the research, early tests and the developing concept, the requirements listed in Figure 4, were self-set for the project, on April 24th.
They were designed to be relatively general, but to also strictly define every essential function, of each wanted feature. If they where all successfully implemented, will be looked at later in Section 5.2.

Context	ID	Requirement Description
Object Detection	OD1	The Object Detection has to be performed in Real Time
	OD1.1	A live video has to be an option for Input
	OD2	The program shall only detect one instance of an object once
	OD3	The User has to be able to choose between different Models, depending on his systems performance
GPS + Time	G1	A detected Object shall be saved with the current GPS-Position of the system
	G1.1	The GPS-Position has to be at least accurate to 4 decimal places
	G2	A detected Object shall be saved with the current Timestamp of the system
Safety	S1	The user gets a warning if Object Detection has an Error
General	R1	Detected Objects can be visualised on a map
	R2	The user has to have access to the raw detection-data
	R3	A live video has to be an available output
	R3.1	Detected objects have to be marked in the video output
	R3.2	The live video has to be above 20 frames per second

Figure 4: Requirements, published in this format on the GitHub Wiki

At that time it was also already clear, that there would have to be necessary to recognise, whether the object in front of the camera was already recognised. If not, a big number of duplicate detections would be saved. So this was also added as requirement ID.OD2.

4 Implementation

This section will describe the different aspects of the final implementation and some of the decisions that were made along the way. First, the general structure of the whole program will be explained, by going through each part of the system and how it is connected to the next part.

The starting point is the *main.py* file. This is the file that the user has to execute in order to start the program All basics are set up here, like the command-line arguments that the user can opt to use, or the setting of the working directory. It also has the job of closing all threads that may still be running when the program gets closed. As the final step in the setup, the window, depicted in Figure 5, is created from the *UI* section of the system.

4.1 User Interface

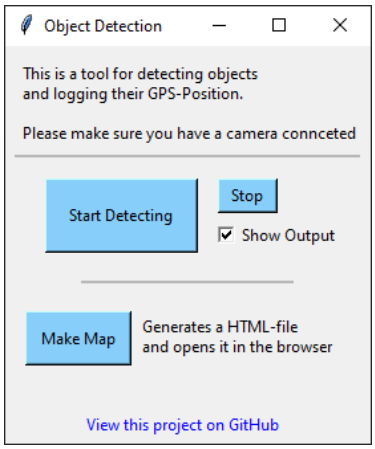

Figure 5: Main tkinter Window

This window is the center for all user operations. Here they can start or stop the object detection, or generate a map of the detected objects. They can also choose if they want to see the output from the webcam or not.
There is a bit of text, to explain the main functions of the program and a link at the bottom to the GitHub repository, where the user can find the source code and more information about the project.

The window and its content is created via the tkinter package for python. A more powerful GUI-Builder like Qt for Python was considered, but tkinter was chosen because of its simplicity and lack of need for external tools.

4.2 Object Detection

Now, the sequence of detecting an object will be explained. If the user presses the *Stat Detecting* button in the window from Figure 5, a new thread for the object detection function will be started. For this, the multiprocessing package is used and it ensures, that the user can still interact with the rest of the system, while the detection is running.

First, the standard print function is overwritten, to always flush the sys.stdout buffer, in which normal prints from a thread are written into. Then, the detector model and its configurations are loaded in. A new file for the saving of the detected objects is created, and a video stream from the webcam is started

The code after this runs in a loop, until the user either presses the stop button or closes the main window.

At the start of a detection-loop, a frame from the webcam is read. From this frame a "blob" is created, using the OpenCV package. It contains basic contour information of the frame, and resizes the image from the webcam to a resolution that the detector can read. Now, the blob is send through the detector, producing a set of detected objects. The function then loops over each of these object from the frame, and executes the following steps for each one.

First it checks if the confidence in the detection is high enough. This threshold can be changed with a command line argument, if maybe a bad lighting situation requires it. If accepted, a box around the object and the confidence are drawn onto the image, as shown in Figure 6.

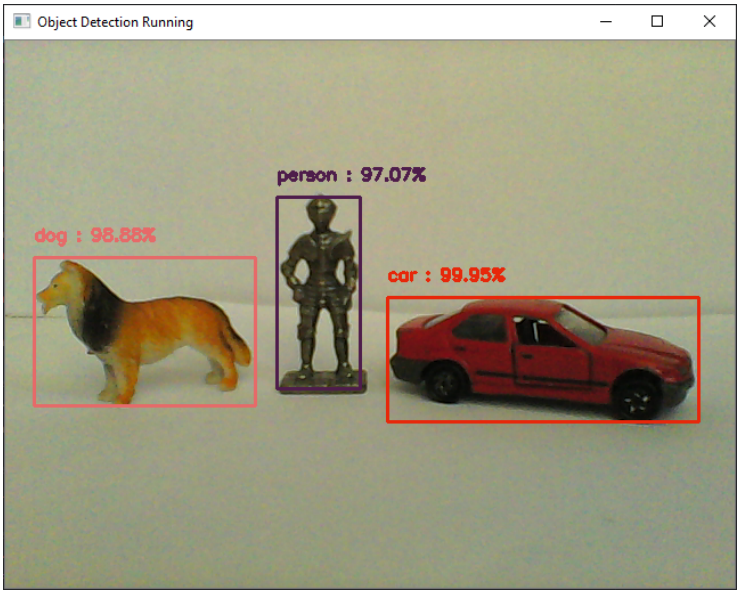

Figure 6: Output window, displaying 3 detected objects and their confidence

Additional information about the detection is then gathered, like the objects position in the image and a timestamp. This is also the point where the system gets the GPS position of the object, which is currently, as discussed earlier in Section 3, just a random coordinate.. With all known information, a new python object is created from the custom *detectedObject* class and the system has to decide now, if this detection should be written into the save file.

4.2.1 Filter

A filter is needed, so that the system doesn't just log every detection it sees, even if the object has been always the same one. For this, the last 30 of the python objects are stored in a list.
The filter then compares each new object to the old ones in the list. It checks if there is an object with the same name and how much time has passed since the last time a detection with this name was made. It also checks how much the object has moved in the image, so a person slowly walking through the picture is still recognised as the same instance of a person.

If the filter can't find an object that almost matches the new one, it writes a new entry, with the detections name, its confidence value, its coordinates and a timestamp, into the CSV log file.

After this, if the *Show Output* Check Box from Figure 5 is currently ticked, the final image, with all detections from this frame drawn onto it, gets displayed in a separate window, seen in Figure 6.
Then, the loop starts again, a new frame is grabbed and the whole detection sequence starts anew.

4.3 Choice of Model

As intended in Requirement ID.OD3, the object detection was at first implemented with both the YOLOv3 and MobileNetSSD models. But this resulted in a few unforeseen complications.
The two models are based on different Deep Learning frameworks, DarkNet and Caffe. Although they are both supported by the OpenCV package and the way they are used is very similar, they each require slightly different code at some stages.
This meant that the whole object detection function had to be written twice, with only minor differences between them. This made the readability of the code considerably worse and new features had to implemented twice. Efforts to combine both models proved fruitless or would have resulted in even messier code.

So, in consultation with the research assistant, the decision was made to include only one detection model. This would make possible future work based on this project much easier and shift the focus more onto things that actually brought new or improved features.
As the final model, MobileNetSSD was chosen, largely based on its major performance benefits, which would be even more important, if the system would be implemented on mobile hardware one day. YOLOv3's accuracy was not big enough of an improvement and only the also selectable "Tiny" sub-version of YOLOv3 was producing smooth frame rate. Finally, the large file size of 240 MB, meant that it couldn't even be uploaded to GitHub.

A separate Git branch was added for the old version with both models implemented, in case the functionality would become relevant again in the future. It lacks the newer features like the filter. But its very basic UI makes it possible to select the wanted model before starting a new detection-session.

4.4 Generating a Map

For the visualisation part of the system, the Python package *folium*[9] was used. It uses the JavaScript library Leaflet, mentioned in Section 3. The maps themselves are interactive and use OpenStreetMap as a source.

Because the program always creates a new CSV file to log the objects each time the detection process is started, the user is able to choose, which detection-session they want to see on a map.
This is done through a file dialog, that opens when the *Make Map* button is pressed. Here the user can choose one of their CSV files, which are named as their time of creation. After a file is chosen, folium creates a map as a HTML file with the Leafleat library.
The system then loops over detection in the CSV file and adds a marker to the map, at the simulated position. The markers are given extra information on the detection, for mouse-over and click-interactions, both seen in Figure 7.

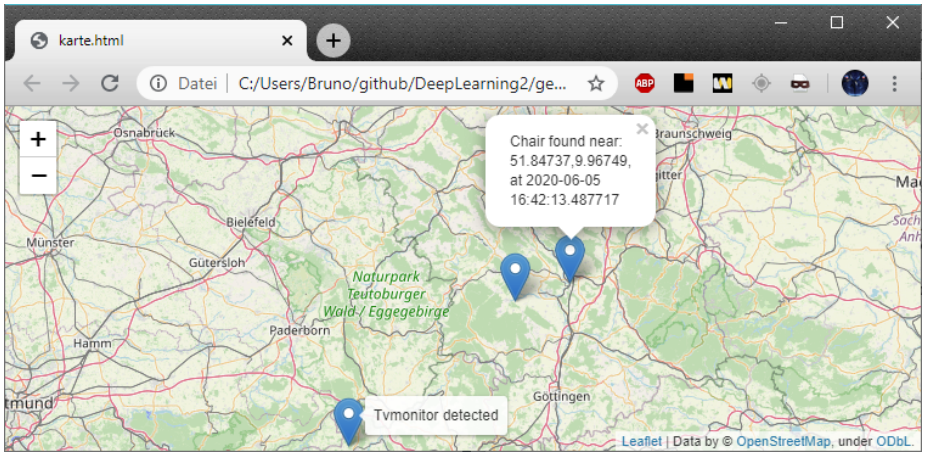

Figure 7: Interactive Leaflet map in the browser

The HTML file is then opened in a new tab of the users default browser, which will look similar to Figure 7. The user can zoom in and out of the map and click on or hover over a marker.
The performance of the map is soft-limited by the resources of the device, bu even a few ten thousand markers shouldn't be a problem.

[9] https://github.com/python-visualization/folium

5 Conclusion

5.1 Results

Although there hasn't been a large-scale real-world test, the system performed really good with small toys in the simple test runs. The UI and the mapping work consistently as they should. The object detection with MobileNetSSD works also really good, with only a few hiccups, that are to be expected from current detection systems.
The filter has, as of writing, still a few problems and still detects objects multiple times. But this should be a relatively easy to improve, with the right adjustment to the parameters.

5.2 Fulfillment of Requirements

Nearly all of the requirements, set in Figure 4, Section 3.2, are satisfied by the implementation. Only requirement ID.OD3, the feature to choose between different models, isn't implemented, for the reasons discussed in Section 4.3.

Requirement ID.OD2, the ability to filter duplicate detections, proved to be not defined enough. So although there is a filter in place, it is hard to quantify the results, unless it works perfectly. Meaning that the requirement could be seen as not fulfilled, from a certain point of view.

5.3 Future Work

While the project is formally done for now, there are still a few interesting things that could be added in the future. The major one would be to develop a small mobile system with a real GPS module, to actually get the coordinates of the objects. This could even be expanded by using triangulation like mentioned in Section 2.1, which would greatly improve the accuracy of the positions They could then be added to the filter, making the it potentially much more reliable.

The functionality of the map could also be expanded, by giving the markers even more information. The icon of the marker could be changed, depending on the type of object and the pop-ups could display a small image of the detection. Folium supports passing any HTML object and getting the image would be easy, since the system already has the bounding box of the objects. An efficient way to store so many images would have to be found, but way is already laid out.

Another possible extension could be using a GPU for the detection. This can potentially give a huge boost to the performance. The support for this is already in the code, but the OpenCV package has to compiled from source on the device.[10] It also requires the device to have a Nvidia GPU with CUDA cores, which is not always the case, especially on something like a Raspberry Pi.

[10] https://www.pyimagesearch.com/2020/02/03/how-to-use-opencvs-dnn-module-with-nvidia-gpus-cuda-and-cudnn/

5.4 Final Thoughts

Personally, the experience of working on this project has been very positive. Computer Vision is a very interesting subject and provides even beginners with rewarding experiences I really enjoyed putting together and improving the program and I am happy with end result.
Working with Python and Git was a bit new, but created pretty good workflows after getting the hang of it. I liked that the task was relatively open, which allowed for much freedom when thinking of new features.

It would be great to see this project on mobile hardware someday, maybe with some of the suggested features. I'll try to keep working on it a bit longer.

References

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

[2] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[3] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[4] Vladimir A. Krylov, Eamonn Kenny, and Rozenn Dahyot. "Automatic Discovery and Geotagging of Objects from Street View Imagery". In: *CoRR* abs/1708.08417 (2017). arXiv: 1708.08417. URL: http://arxiv.org/abs/1708.08417.

[5] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *ECCV*. 2016.

[6] Prince Patel. *Why Python is the most popular language used for Machine Learning*. https://medium.com/@UdacityINDIA/why-use-python-for-machine-learning-e4b0b4457a77. Accessed: 2020-06-17. Udacity India, Mar. 2018.

[7] Patrick Poirson et al. "Fast Single Shot Detection and Pose Estimation". In: (Sept. 2016).

[8] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[9] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[10] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[11] Adrian Rosebrock. *Object detection with deep learning and OpenCV*. https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/. Accessed: 2020-06-18. PyImageSearch, Sept. 2017.

[12] Adrian Rosebrock. *YOLO object detection with OpenCV*. https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/. Accessed: 2020-04-13. PyImageSearch, Nov. 2018.