Thesis Dissertation

ROBOT VISUAL IDENTIFICATION AND LOCATION ALGORITHMS FOR INDUSTRIAL IOT APPLICATIONS

Ilias Kalaitzidis

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2022

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Robot Visual Identification and Location Algorithms For Industrial Iot Applications

Ilias Kalaitzidis

Supervisor

Dr. George Pallis

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

Acknowledgements

I would like to thank Dr. George Pallis who gave me the opportunity to work on a very interesting project from which I learned a lot, and his support throughout the duration of the thesis.

I would also like to thank research assistant Joanna Georgiou who helped me from the beginning to find all relevant information for the implementation and make the experimentation process easier.

Abstract

Autonomous systems using Internet of Things devices are becoming more popular every day. Specifically, the use of robots in industrial scenarios, like assembly lines, has increased in the last couple of years. The goal of this thesis is to simulate this scenario using a Dobot Magician which will pick and place cubes, sorting them based on their color. To achieve this, we developed multiple different recognition techniques to compare them based on different criteria and see what advantages and disadvantages each method has. In addition, we want to see how a system performs when changing its different parameters such as workload and speed of operation.

Contents

Chapter 1

Introduction

1.1 Motivation

One of the most prominent subjects in the technology world is Internet of Things. Millions of devices are connected to the internet exchange information and interact with each other. This interconnectivity has led to a rising number of new and exciting applications in many areas. An interesting use of IoT is in the industrial level, where the use of devices like robotic arms and sensors is rapidly increasing to build more efficient, autonomous and fault tolerant systems. These systems are getting more advanced with several metrics being considered to evaluate their performance. This is why it's important to find algorithms that is most suited for a specific industrial scenario, considering how much an algorithm can affect those metrics.

1.2 Project's Goals

The goal of this project is to implement an autonomous system using the Dobot Magician robotic arm [1] and a camera module to emulate an industrial scenario on a smaller scale. This system is going to perform a pick and place operation on cubes, by picking and sorting them according to their color. To achieve this, we used 3 different methods for detection and color matching, along

with solutions to make its truly autonomous with the end goal being to compare them on their performance and resources used.

1.3 System Setup

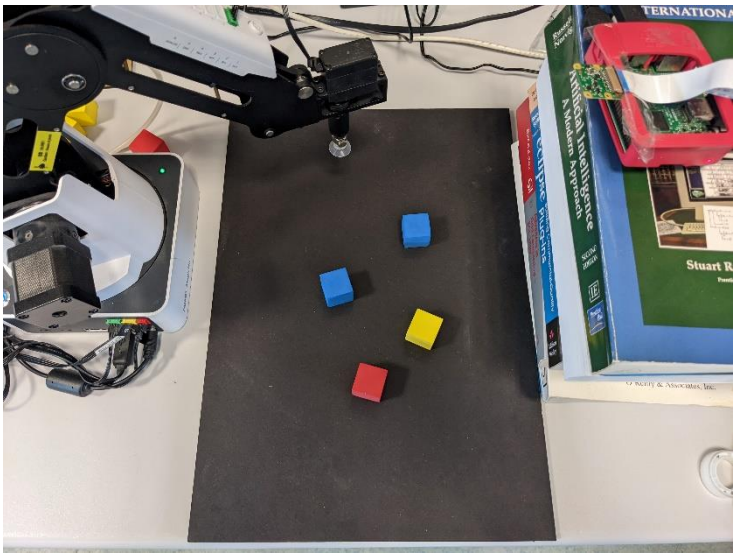


Figure 1.1 Setup used

In this project we used the setup shown in Figure 1.1, which consists of a raspberry pi equipped with a picamera module, and a Dobot Magician. The raspberry pi has been placed on top of some books to fully capture the working area, and the Dobot Magician close by to pick up and sort the detected cubes. For collecting all the necessary metrics for the experiments, an already setup monitoring stack has been utilized. The latter monitoring system, uses Netdata [2], a widely-known open-source tool that captures real-time metrics, like CPU utilization, network traffic, disk / memory usage, etc. In addition, for storing and querying the collected data, Prometheus [3] was utilized as the exported source. Prometheus is also an open-source framework, used as an efficient time series database. At last, for Prometheus to be able to know the services it needs to retrieve data from, Consul [4] was included for service discovery purposes

1.4 Contribution

Based on the considerations of a similar industrial system, we emulated a smaller scale version of that system, and found how much different methods can affect it based on the most important metrics such as the accuracy and resources used. In addition, we measured how much these metrics change in an already implemented system, when parameters like workload and speed of operation are altered. Taking our results in consideration can help other people when deciding to build their own autonomous system even on a larger scale.

Finally, by making some changes in a public library used to control the robot we helped its future users to implement their solutions easier.

1.5 Chapter Outline

In the first chapter we introduce the motivation and the general idea behind this project, mentioning the goal and our contribution to it. The second chapter focuses more on the technologies used, including the hardware the software that were utilized. The third chapter goes more in depth in the implementation of this project, describing the required tasks, how we approached, the problems and the solutions we came up for every step. In the fourth chapter we describe the methodology and experiments run along with the results of those experiments and finally the fifth chapter is about the conclusion we got from the experiments.

Chapter 2

Background

2.1 Technologies Used

This project was implemented in a Raspberry Pi 3B using the Raspberry Pi Buster OS and a picamera. All of the implementation and library extensions were done using Python 3.7.3.
In this project we implemented the multiple methods to detect the desired objects and relay that information to the robot, for it to pick them and place them according to their color. After that, we run multiple experiments and used a monitor stack to record the necessary metrics and come to some conclusions about the methods used.

2.2 Dobot Magician

2.2.1 Introduction to Dobot Magician

Dobot magician is a multi-functional desktop robotic arm for practical training education, like the application we are developing in the scope of this desertion. It has the possibility to be used

in various different applications by changing its end effectors. These include the gripper, the suction cup, a pen holder, a laser, and a 3D printer. With these end effectors, the robotic arm can be used for drawing and writing, laser engraving, picking and placing objects and 3D printing. In addition, Dobot provides numerous kits to extend the possibilities of the robot, like a conveyer belt kit and a sliding rail which can be used to simulate some industrial work on a smaller scale.

Figure 2.1 Dobot magician with the conveyer belt kit

Control software is available on multiple platforms, including Linux and Windows computers, smaller devices like Arduinos and Raspberry Pis [6] and even through an iOS application. A useful application to get started with the robotic arm is Dobot Studio [7], which includes a graphical interface through which you can control it. This can be either done by simple scripts or using some more approachable methods like Blockly [8], a tool for creating programming applications using visual building blocks, mouse controls and motion controls.

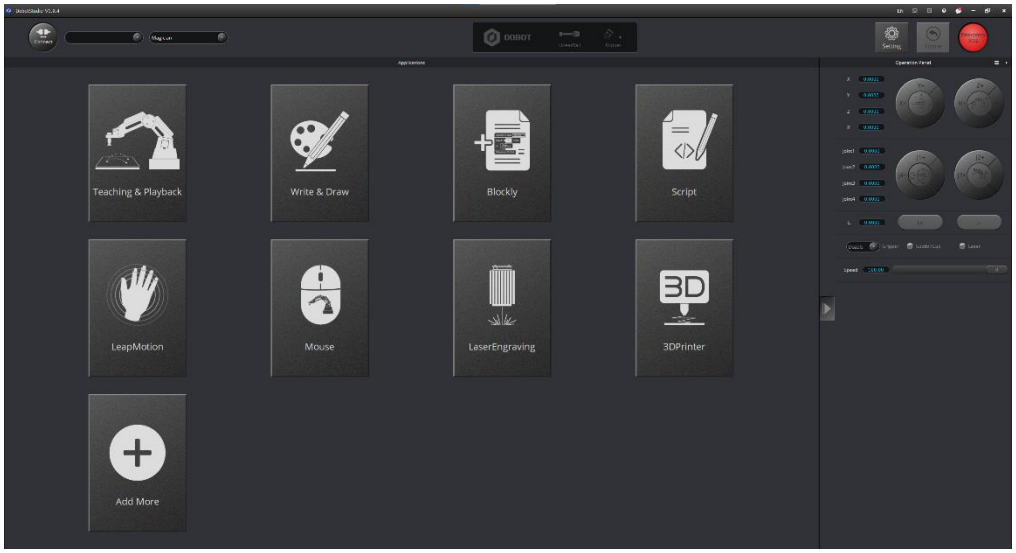


Figure 2.2 Dobot Studio initial screen

2.2.2 Dobot API

To communicate with the robot through Python we are going to use the pydobot library [5], which utilizes the Dobot Communication Protocol [19]. The official API although had great documentation, required complex instructions to install to the raspberry and more code to achieve the same result. In contrast, pydobot requires no other installation other than itself as it communicates directly using the serial port and encapsulates most of our desired operations into simple-to- use functions.

More specifically the library provides these 4 useful commands:

- pose(), to get the coordinates of the robot
- move_to (x, y, z, wait), to move the robot to the destination point with the ability to specify if we want to wait until the process executes before continuing
- speed (velocity, acceleration), to set the speed at which the robot moves
- suck (enable), to enable the suction cup

A feature that wasn't implemented by the library is the homing operation. The homing operation guarantees that the coordinates of the robot remain consistent every time it's rebooted, so it is essential for developing a consistent and easier to use system.

To achieve this operation, we created a fork of the library to implement it ourselves. After studying the source code, we figured out that every function works by sending a message to the robot consisting of an id corresponding to a command available in the communication protocol and a control value based on the type of the command. So, by looking at the official documentation of the communication protocol we implemented the home operation using 31 for the id, which corresponds to the SetHOMECmd command and 0x03 for the control value.

```
def home(self,wait=False):
    msg = Message()
    msg.id = CommunicationProtocolIDs.SET_HOME_CMD
    msg.ctrl = ControlValues.THREE
    return self._send_message(msg)
```

Figure 2.3 Code implementation of the home operation

2.3 Coral USB Accelerator

The Coral USB Accelerator [9] is a tensor processing unit developed by Google. More specifically it's an application-specific integrated circuit which is used to accelerate machine learning inferencing in a small form factor. It is very useful for situations where machine learning capabilities are needed in an edge device which doesn't have the resources to run inference models.

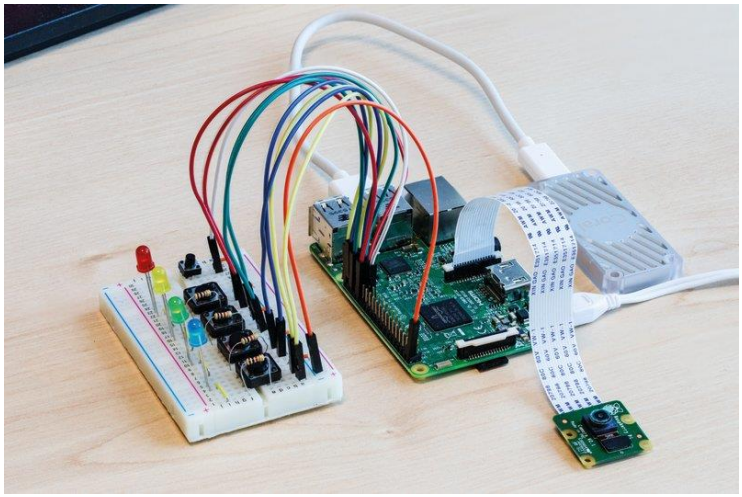


Figure 2.4 A raspberry pi with the picamera module and a TPU https://magpi.raspberrypi.com/articles/teachable-machine-coral-usb-accelerator

This image shows a very common setup where a raspberry pi is used with the picamera module. The pi can be mounted on a moving device and use the accelerator to process the image on itself, something that would not be possible due to its limited resources or without compromising the latency by sending the camera feed to a remote device. It has the ability to run 4 trillion operations per second (4 TOPS), using 2 watts of power [10], which also makes it suitable to run with low power devices.

The TPU (Tensor Processing Unit) is currently only compatible with TensorFlow Lite models [11] although you cannot build TensorFlow Lite models directly. Instead, you must build a regular TensorFlow model and convert it, as it is shown in the image below.

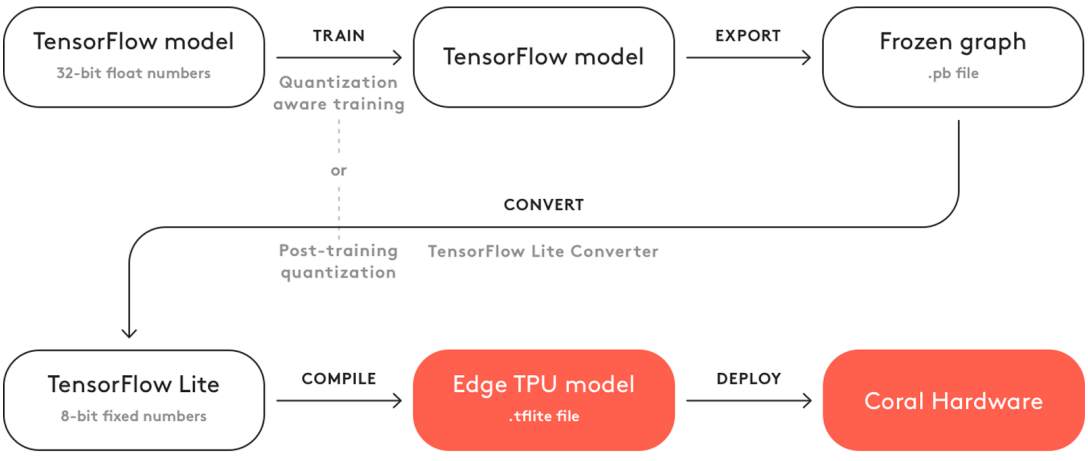


Figure 2.5 Flowchart of creating or converting a TensorFlow model for the TPU https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

Alternatively, you can use a technique called transfer learning, which allows you to take a compatible pre-trained model and with sufficient training data change that model to work for your goal.

The coral API [12] is available in both Python and C++ with methods to run the inferences, distribute the workload on multiple TPUs and retrain TensorFlow models.

2.4 EfficentNet

EfficientNet [13] is the convolutional neural network architecture of the model that we used to detect our objects in real time. It was developed by two engineers from, with its main selling features being high accuracy and speed while reducing parameter size and FLOPS. The theory behind the architecture, is to instead of arbitrary scaling the networks dimensions, like most conventional architectures use, it uniformly scales each dimension with a fixed set of scaling coefficients

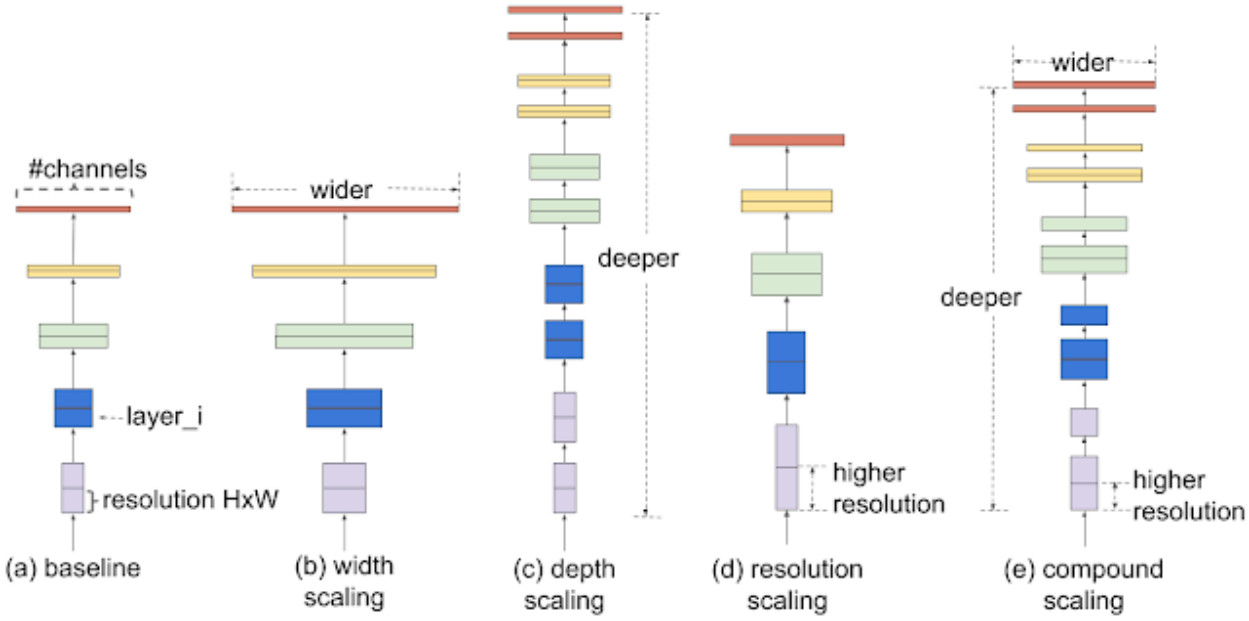


Figure 2.6 Scaling logic behind the EfficientNet architecture
https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html

This makes it a great choice for mobile devices like the TPU where computational power is not abundant, without compromising on accuracy or speed.

2.5 Prometheus

Prometheus is an open-source monitoring and alerting system. It’s an easy to setup method to for recording metrics from a specific endpoint on certain intervals. The difference between Prometheus and other monitoring systems is that it “pulls” the data instead of pushing them. In a traditional monitoring system, the edge device has to send the data itself which can increase the load. Instead, Prometheus pulls the data over HTTP in batches, so the system only has to convert its metrics when requested instead of sending them as they’re created. In addition, this provides an easy way to tell when there is a failure on the monitored device since the request will fail making it a great solution in industrial environments where there are many edge devices and quickly detecting a hardware failure is important.

Chapter 3

Implementation

3.1 System Overview

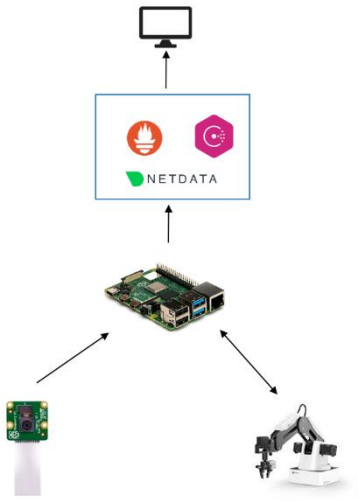


Figure 3.1 System Overview

Before talking about the implementation let's have an overview of our architecture. A picamera module is connected to the raspberry capturing video footage of our workplace. This footage is analyzed on the raspberry pi which then sends the required information to the robotic arm which is also connected to the raspberry. The monitoring stack captures information from the raspberry,

including resources used from the raspberry and the robotic arm, and sends it to our administration station from which we can graph the results.

3.2 System Calibration

The first thing we implemented is the hand-eye calibration. Having a reliable method to translate the coordinates from one coordinate system to another, while avoiding complicated methods, is essential.

These are the steps we followed:

1. Execute the homing operation. The homing operation guarantees that the coordinates of the robot will stay the same every time it turns on. If the operation is not performed, then the system will have to be calibrated every time, since the coordinates will be based on the initial position of the robot and the translation will not be valid for a different coordinate system.
2. Find each cube's center in an image. The points act as a reference that will be matched with the coordinates of the robot. Any object can be used for this step, but for a simpler calibration process we found that using easily recognizable shapes like a square or circle grid can help, since there are many algorithms available that can detect them. Alternatively, we could use the actual objects that we will detect if we have a high accuracy algorithm, or even manually write down the coordinates.
3. Move the robot hand to the points detected in the previous step.
4. Use the *estimateAffine2D* function from OpenCV [14] to calculate the transformation matrix. With 2 sets of points, this function can create a matrix which we can be utilized to translate any point from the image to the corresponding robot hand coordinate. To achieve this, we can use the following equation:

$$\begin{bmatrix} RobotX \\ RobotY \end{bmatrix} = \begin{bmatrix} \alpha 1 & \alpha 2 & \alpha 3 \\ \alpha 4 & \alpha 5 & \alpha 6 \end{bmatrix} \times \begin{bmatrix} ImageX \\ ImageY \\ 1 \end{bmatrix}$$

Although simple, this method requires had the disadvantage of needing quite a lot of manual work from the user, especially if there are many calibration points to improve its accuracy.

In addition, the algorithm occasionally struggles to accurately transform the points that are far from the calibration objects.

Figure 3.2 Example of calibration image

For example, as figure 3.1 depicts, 6 cubes were used for calibration. When running the operation, cubes the that will be inside the blue area will have their coordinates correctly translated, however cubes that will be in the red area may not produce satisfactory results. This can be improved by spreading the calibration points or adding more of them to cover the whole working area; however, this will also increase the manual work needed for the calibration.

Finally, the latter process must be repeated in case of camera or robot relocation.

3.3 Color Matching

For the robot to sort the cubes, a color detection method is needed. A simple solution would be to find the RGB value of the detection point and group them together. The problem with this approach is that the RGB values can greatly vary when the cubes are exposed to different lighting conditions.

To solve this problem, we can use the webcolors [15] library which can match the RGB value to a real-life color. Although the library will return a ValueError in case there isn't an exact match, we can manually calculate the closest color using this formula:

$$d(p, q) = \sqrt{(p - q)^2}.$$

Where p is our target color and q is every color available in the library. From here, the minimum value is the closest color

Another issue with this method is that the calculated color is often not one of the four desirable colors but a variation of it. For example, instead of "blue" the output could be "light blue" or "sky blue". To combat this, we search the 4 desired colors in the output string ("green" in "olive green", or "red" in "bright red") and run a few tests to collect the most common interpretations that don't explicitly include the 4 colors in their name ("pink" is red, "khaki" is "yellow").

Figure 3.3 Example of color matching

Figure 3.2 shows an example of color matching. As we can see, the matched colors are variations of our target colors with a prefix before. Even the same two green cubes are matched with a slightly different color.

3.4 Cube detection

After some research we came up with the following 3 different methods to detect the cubes:

- Find the outline of the cubes using the *findContours* function of OpenCV

- Match a template image of a cube to an actual cube from the camera frame using the *matchTemplate* function of OpenCV
- Use the Coral USB Accelerator to detect the cubes using a machine learning model

Since the algorithms often can't detect every cube at once, we run them again when (i)the robot has already picked the detected ones, (ii) until there are none left, or (iii) the algorithm can't find the rest.

3.4.1 Contour detection

The first method that we tested was contour detection. This method works by finding continuous curves with similar characteristics, thus detecting the formed shapes.

The function works on binary images, so the first step is to convert the image to binary and apply the morphological open operation to remove some of the noise present. Then we can use the *findContours* function to find the outlines.

This method often detects unwanted small shapes formed by the reflection of the light or large shapes formed by objects in the background. To solve this issue, we also calculate the area of the shape and keep only those within the acceptable range of a cube, which we find after running a few tests.

Having detected the cubes, we can calculate the center of the contour to find the pickup point. Upon finding the pickup point, we can then calculate its color using the method described above.

The biggest drawback of this method is that it is highly sensitive to light since too much or too little light exposure can hide the objects in the binary image. In a controlled environment this may not be an issue, since lighting conditions can be calibrated for optimal detection and then stay the same, but in a place where natural light is involved the performance of this method can drastically change through at the day. So using a method like this, in a real-life application may not be even possible since we do not control the light all the time.

Another situation where this method can struggle is when two cubes are too close to each other, resulting their contours blending together and being considered as one shape. This can lead to wrong calculation of pick-up points, or to completely discard them due to the area threshold.

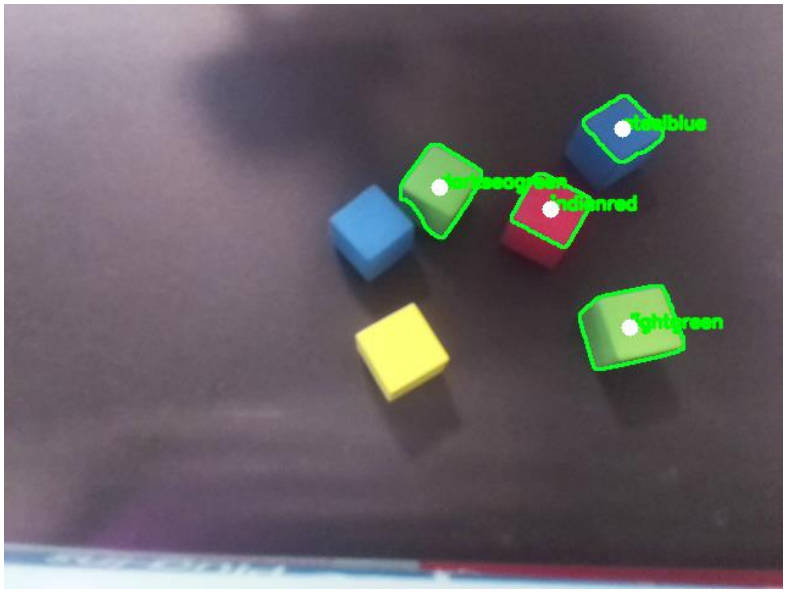


Figure 3.4 Example of the contour detection method

Here we can see an example of the method in use. As we can see, the cubes that are in a shadier part of the area are detected without problems, however, the yellow and blue cubes that are closer to the light were not detected.

3.4.2 Template Matching

Template matching is a useful technique for finding a small image inside a larger image, which is exactly our goal since we are trying to find a cube in our working area. To find the location of the target, the algorithm slides the template over the target image and compares the template and patch of input image under the template image.

OpenCV has several comparison methods from which “TM_CCOEFF” gave the best results.

After using the *templateMatch* function, we capture an image where every pixel denotes how much does the neighbor of that pixel match with the template. The maximum value represents the top left corner of the matched region from which we can calculate the pickup point and then the color at that location.

A big advantage of this method in regard to the previous one, is that it is much less light sensitive. Since the algorithm returns how much the template matches with every chunk of the

image, by adjusting the acceptable threshold, the highest match should always be a cube even if part of it is obscured by light.

Unfortunately, because we only use the max value from the result, that means that the method returns just one cube for every frame. As a result, we must run the algorithm every time for each cube, which makes it more computationally expensive.

There is a method for multiple object matching, where we find all the point that are above a certain threshold, but it creates a lot of false positives and brings the overall accuracy down.

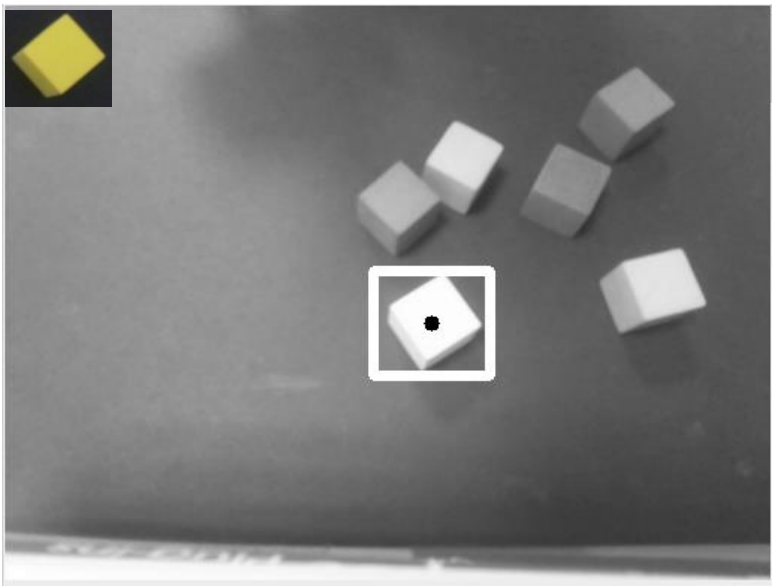


Figure 3.5 Example of the template matching method along with the template image used

Figure 3.4 shows the template used and matched cube. Since the method works with grayscale images, light conditions don't play such a big role like in the previous, so it has no trouble detecting the cube that had more direct light on it.

3.4.3 Coral USB Accelerator

A common issue with using machine learning models for image recognition is finding a model that was already trained for your specific need. Most models are trained to detect common objects, so the more specific your items are, the harder is finding a model that can identify them.

For our implementation, we couldn't find a model which can identify cubes. The main reason is that for most geometric shapes, using machine learning is considered overkill, since using image processing methods like the aforementioned are simpler and can get satisfactory results.

Nonetheless, it's still an interesting idea which we could compare to the previous methods. So instead of finding a pre-trained model we decided to create our own.

The largest difficulty of creating a new detection model is finding or creating a large enough dataset with the correct annotations, which is the class of the object and its coordinates in the image.

To make this process easier we decided to use synthetic data. The idea behind synthetic data is to create a 3D model of the target object in a modeling software, such as Blender or Maya [15][16], and then render as many images as we want using different parameters. This technique can let us create large datasets where each image has a different combination of lighting, background, camera angle and object texture, thus having a big variety of training data which is essential when creating a machine learning model.

To create our synthesized dataset, we used "*zpy*" which is a combination of a python library and a Blender add-on with which we can easily set our parameters and automatically get a dataset with correct annotations in the coco-json format.

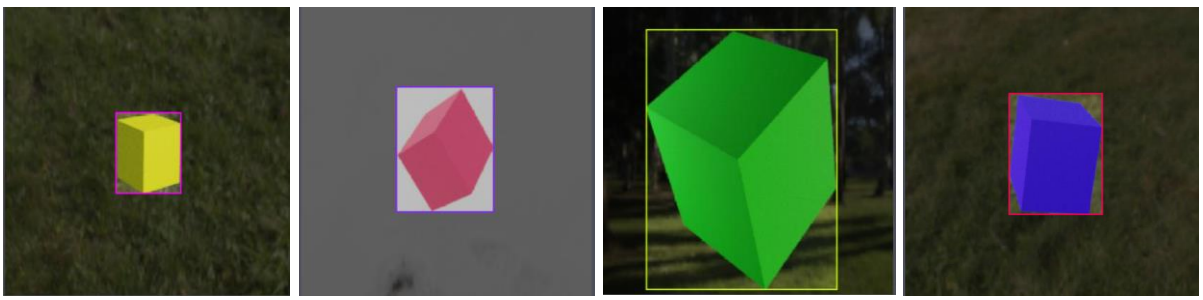


Figure 3.6 Examples of synthetic data with their bounding boxes

Figure 3.5 shows some of the images generated by zpy [17] along with the generated annotations. As we can see every image is taken from a different angle and depth, has a different backround and lighting with the bounding boxes generated automatically.

Having created our dataset, we can now use it to train our model. Instead of training a new model from scratch, we used a technique called "*fine-tuning*" on an already pre-trained model optimized to run on the TPU. Fine-tuning is a process where instead of retraining the whole model, which can take a lot of time, you unfreeze the last few layers of the model, where the final classification occurs. This results in faster training time and can be done with a smaller dataset without compromising the accuracy of the model.

The final dataset consists of a total 1600 images, 400 for each cube color with 70% allocated for training, 20% for validation and 10% for training, with the accuracy of the model being around 96%.

This model, in addition of having good results, has the additional benefit of not relying on the color detection method defined above, which can be affected by the light. Instead, the model has 4 classes, one for which color, so the detection and the color classification is done at the same time and is quite impervious to lighting conditions since the dataset was trained with a variety of light settings.

One issue is that the inference doesn't detect all cubes simultaneously, which doesn't seem to be due to its inability, because when running the inference again after the moving the detected ones the rest are also detected. A possible explanation is that when there are many cubes on the working area, the confidence for some cubes slightly drops and puts them below the threshold, which is set to 92% to avoid some false positives that were happening during testing.

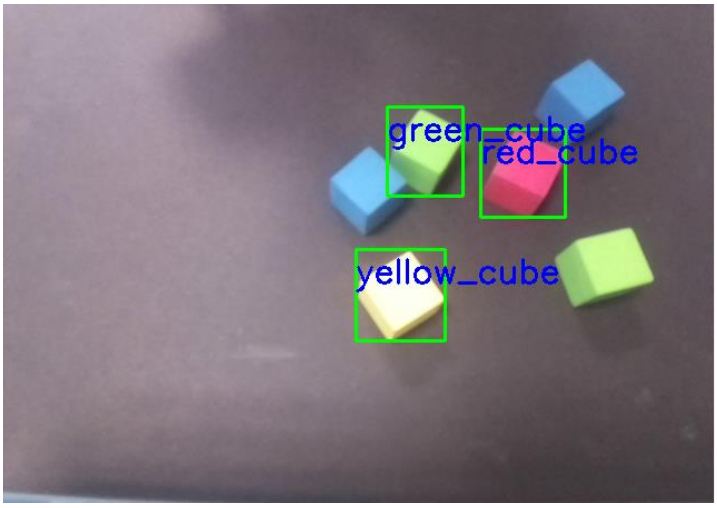


Figure 3.7 Example of the TPU method

Figure 3.6 shows that only three of the 6 cubes were detected from the first inference, so another one had to be run for the rest. Nonetheless, we can see that the method had no trouble detecting the yellow cube which had even more direct light on it compared to the image used for contour method, and colors of the cubes were correctly identified

Chapter 4

Methodology and Experiments

4.1 Methodology

There are two major steps when someone wants to create an algorithm, as the ones we created. The first one is to find and implement different methods that achieve the same result and then compare them based on certain criteria to find the best one.

To implement the pick and place operation, two things must be considered. How will the robot hand know where the cubes are relative to the camera, and the actual detection methods that will find the locations of the cubes.

The first part is known as the hand-eye calibration problem, where given a set of points from one system, in our case the camera, a transformation matrix is used to calculate the corresponding set of points of the robot system. There are various methods to achieve this, each with its own

degree of implementation difficulty, and the best one is based on the setup used and the acceptable error. In our situation, since there is just one robot, one static camera and the size of the cubes can allow a relatively large margin of error, thus a more simplistic approach can be used that will still give acceptable results.

The second part can be a little more complex due to the number of possible techniques that can be used. Possible solutions can range from simple image processing methods and up to more sophisticated machine learning algorithms. Each method has its own advantages and disadvantages and the best one is determined only by the desired goal of the application. In this study, we tried to use 3 different methods and compare them so we could get a variety of results.

Since the operation has an additional function of sorting the cubes based on their color, a method for finding the color of each cube must also be implemented.

Having implemented the algorithm, the next step is to set up the experiments and run them for each method. The experiments are set up using 2 variables, the number of cubes that need to be detected and the speed of the robotic arm. The algorithms will be run on a raspberry pi which can collect performance metrics, such as CPU utilization, memory, network traffic, and energy consumption, using a monitoring stack. Results will be based on 3 major criteria.

- Accuracy. This includes not only if the robot can successfully pick up the cubes but also the correct assignment of the cube to its respective color.
- Resources used. The main metrics we care about are power usage, CPU utilization and temperature to see which methodology is the most efficient.
- Time needed to complete. This includes the time needed to load the necessary data in the memory, calculate the transformation matrix, process the image, and move the robot until all cubes are picked up.

4.2 Experiments

The main 3 criteria with which we will be judging the algorithms are accuracy, power consumption of the raspberry and their accuracy.

Additionally, we also recorded these following metrics: disk reads, network usage, pi temperature, CPU usage, memory usage and the power draw of the robotic arm. Some of which

are corelated to the previous 3, to help interpret the results and perhaps find some unexpected patterns.

To measure the accuracy, we wrote down the results, specifying all additional info such as the number of successful pick-ups and reasons for failure.

4.2.1 Experiment Setup

To keep the experiments fair, we tried to keep the setup the same for each method. We used 12 cubes, 3 for each color, spread evenly across the working area and marked their positions so we can have them in the same spot for the next experiments.

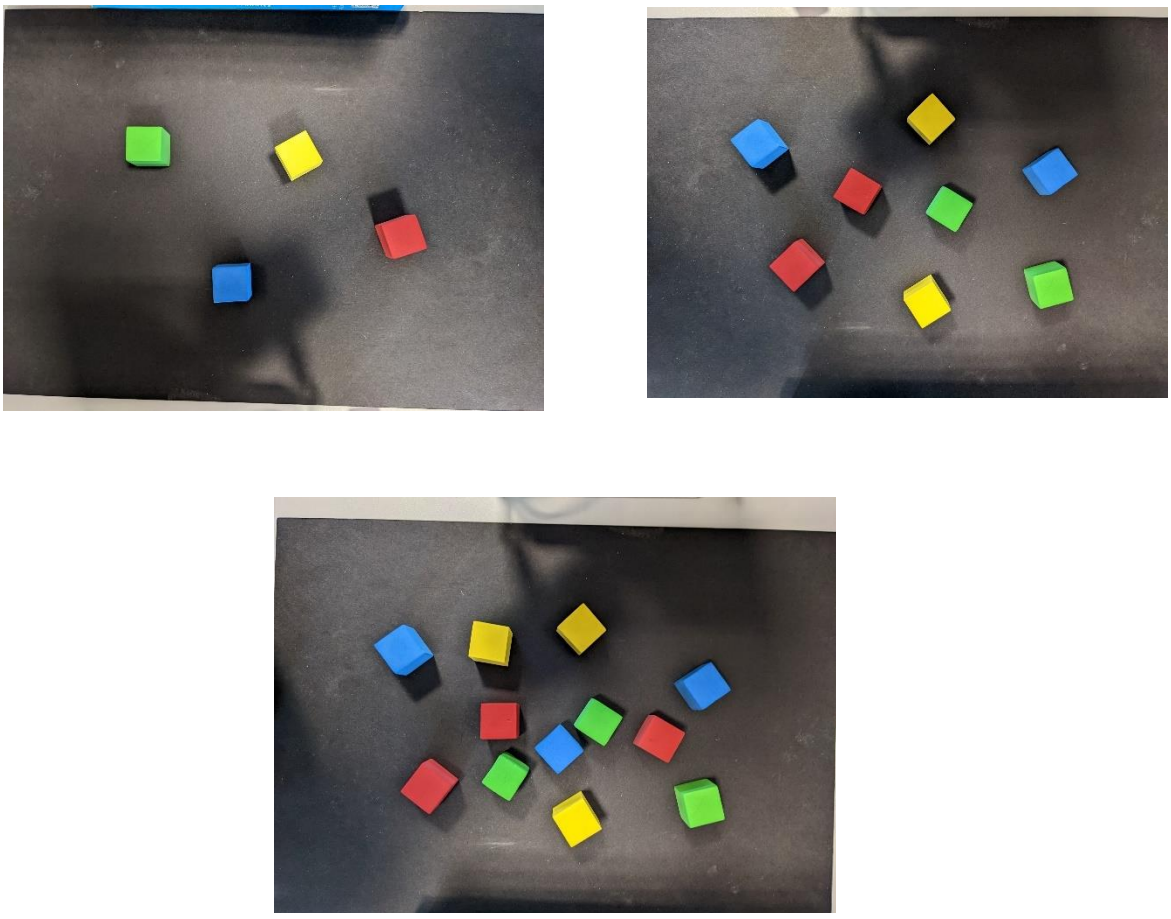


Figure 4.1 Placement of each cube for the experiments

The placement of the cubes was based on the maximum reach of the robot.

The speed and acceleration of the robotic hand also stayed the same with both at 200.

In addition, we also tested how the results vary when changing the experiment parameters. Since the TPU method was the most consistent we tried changing the number of cubes to 8 and 4 to see how much the performance changes relatively to the cubes present, and the speed of the robotic arm to 100/100 and 300/300 to see if there is any difference in its accuracy

4.2.2 Method Accuracy

First let's compare the accuracy of each method. Again, we run all experiments with the 12 cubes at the same position and robotic arm speed. In addition, in those trials we, run the methods repeatedly until all cubes were picked up, or at least attempted to, or the methods couldn't detect any more.

Method	Successful Pickups	Reason of failure
Contour Detection	10	Couldn't detect one red and one blue cube
Template Matching	9	Robot coordinates were slightly incorrect
TPU	11	Robot coordinate was slightly incorrect

Table 4.1 Number of successful pickups and reasons of failure for each method

As we can see from table 4.1, all 3 algorithms had pretty good results. The only one who struggled with detecting the cubes was the contour detection method.

As expected, the first method had difficulties detecting the cubes that weren't in good lighting conditions. Specifically, from this and other test experiments, we found that it had the most trouble detecting red and blue cubes probably because these colors are darker than the yellow and green cubes, so even with small shadows the were too dark for the algorithms to detect.

The other two methods had no trouble detecting the cubes. For both, the issue was the image-robot coordinate translation.

Although the TPU mistake is in the acceptable error range, the template matching method does seem to result in worse robot coordinates. The probable cause is that the center of the cube that is calculated from the detected area, is slightly lower than the actual center of the cube. So, combining that with the small error from the translation operation in some edge cases, results in more frequent failed pickups.

A possible solution was to manually facture in this error when moving the robot, but by doing that we create more errors for cases where the translation was correct.

4.2.3 Method Execution Time

Now let's see how much time each method needed to complete. Since the speed of the robot and the number of cubes is the same, the most important factors are how fast the algorithms are and how many times they had to be run to complete.

Even though the number of successful pickups is different for each method, the second and third methods still had the same number of attempts, so the only difference we need to consider is for the first one, which completely missed two cubes

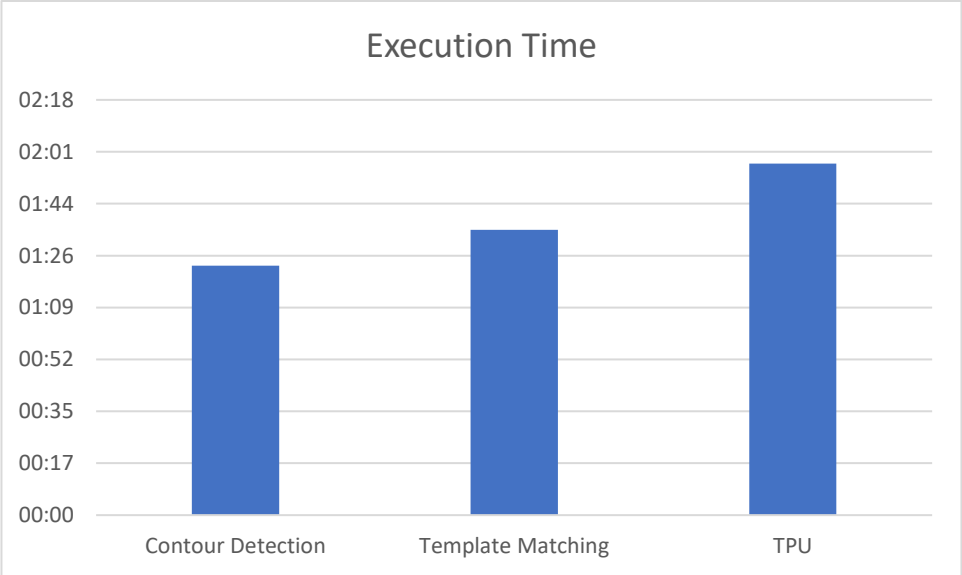


Figure 4.2 Execution time of each method

From this graph we can see that the TPU method is slower by a large margin. The main reason is because the model doesn't detect all 12 cubes at the same time, so the inference must be run

multiple times. For this scenario specifically, the model detected 5 cubes the first time, 4 the second and 2 the third, and one more for the last one.

The other 2 methods are a little bit closer. Even though the second method needed a little more time, it also detected and attempted to pick up 2 additional cubes. If we want a way to compare them more directly, we can calculate the average time to pick up each cube by taking the execution time and dividing it by the total number of cubes detected. So, for contour detection the average time is 83 seconds/ 10cubes = 8.3 seconds/cube and for template matching it's 95 seconds/12 cubes = 7.9 seconds/cube. For comparison, the TPU method needs 9.75 seconds/cube.

4.2.4 Method Resources used

Finally, we are going to compare the resources used by each method starting with the power needed.

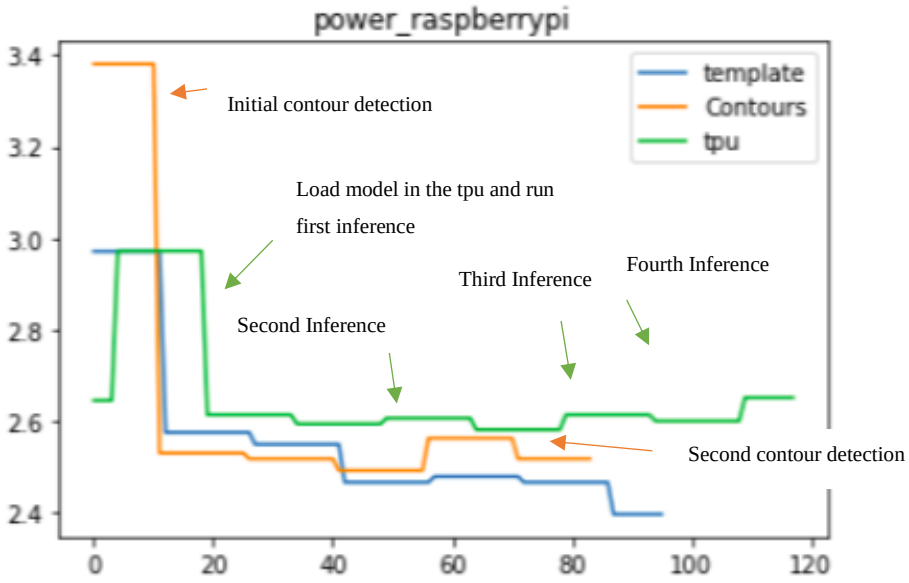


Figure 4.3 Power draw of each method

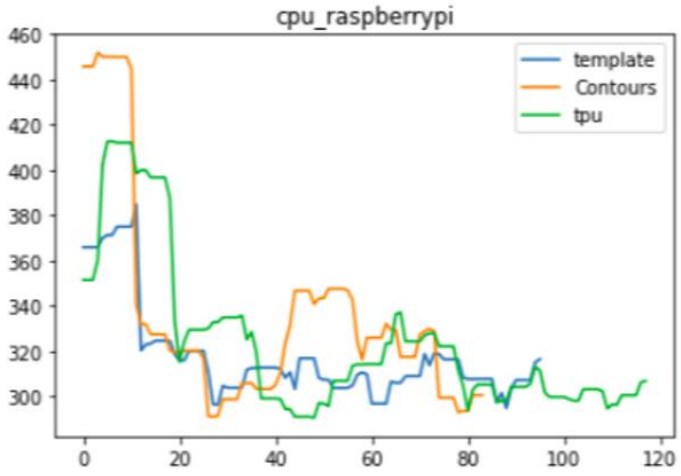
From figure 4.2 we can see that all 3 methods follow a similar pattern. They have an initial big power draw from the initial procedures, which include loading the necessary information into memory, calculating the transformation matrix, and doing the first image processing. Then, the

power drops while waiting for the robot to pick up the detected cubes and increases by a little bit every time an algorithm is used. This is obvious with the contour and the TPU method where the first one needed 2 uses so there are 2 distinct power spikes, and the second one needed 4 inferences so there are 4 distinct power spikes.

The only one who had unexpected results is the template matching method. Although it also has that initial big power draw and some additional power spikes, they do not match the 12 times the method was used. On the contrary, the power used seems to continuously decrease.

Comparing the overall power needed, the contour method is clearly more resource heavy than the others. Not only it has the biggest initial power draw, but by comparing the power spikes, which should indicate the power only the algorithms need, we can see that it also uses the most for a single use.

Next, we are going to see the CPU usage and the temperature of the raspberry. These metrics are directly related to power consumption so what we are looking for is to confirm the previous results by comparing them to these.



Fiaure 4.4 CPU usaae of each method

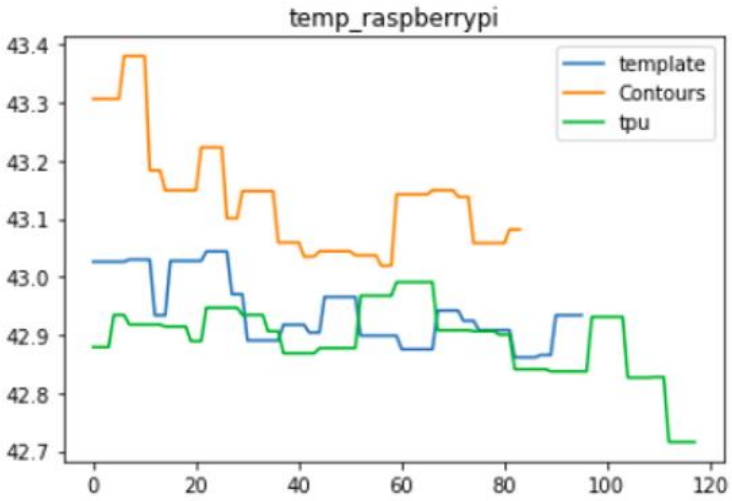


Figure 4.5 Temperature of the raspberry during each method's execution

As we can see, the results are in line with what we expected. The CPU usage mirrors the power consumption graph with an initial big spike in the beginning and then some spikes for the contour and TPU methods. Again, the contour method is the most resource heavy, with the other two being more similar to each other, although we would have expected the TPU method to have lower CPU utilization since the processing is happening at the TPU and not on the raspberry itself.

This also reflects on the temperature, with the contour method resulting in higher temperatures although the difference between the methods is not significant.

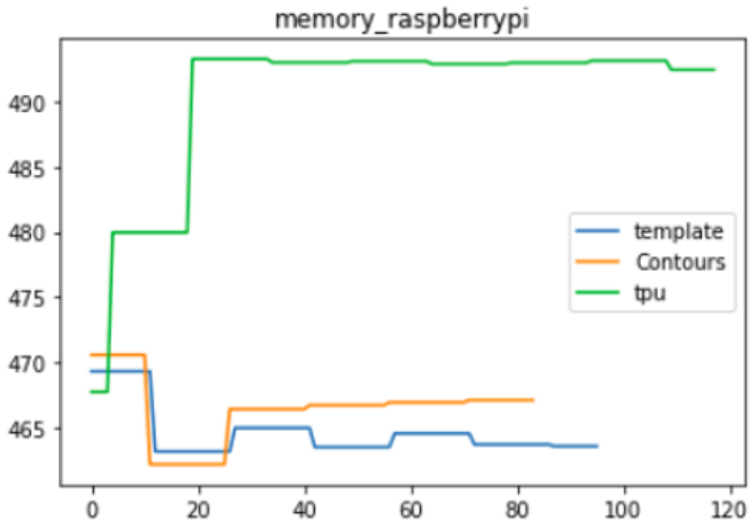


Figure 4.6 Memory usage of each method

Finally, memory usage also is close to what we expected. The contour and template matching methods are very similar since they both just need to hold the image that is being processed. The TPU method needs to additionally load the prediction model, so the memory usage goes up.

4.2.5 Changing parameters

4.3

Now that we have compared each method, let's see how other parameters affect the performance of the system. The two things we are going to adjust and test is the speed and acceleration of the robotic arm and the number of cubes picked up. For these experiments, we used the TPU method since from what we saw in the previous part it is the most accurate.

In total we ran 9 experiments, one for each combination of speed (100/100,200/200,300/300) and number of cubes (12,8,4). None of the unsuccessful attempts were caused by the detection method

4.3.1 Accuracy

	100/100	200/200	300/300
12	12	11	10
8	8	8	8
4	4	4	4

Table 4.2 Accuracy results for the 9 combinations of parameters

From this table we can see that the speed and acceleration of the robotic hand can affect the accuracy of the system by a small amount. This seems to only happen with 12 cubes, probably because they are more spread out, so the combination of the small calibration error and the small inaccuracy from the higher speeds result in more failed pickups.

4.3.2 Execution Time

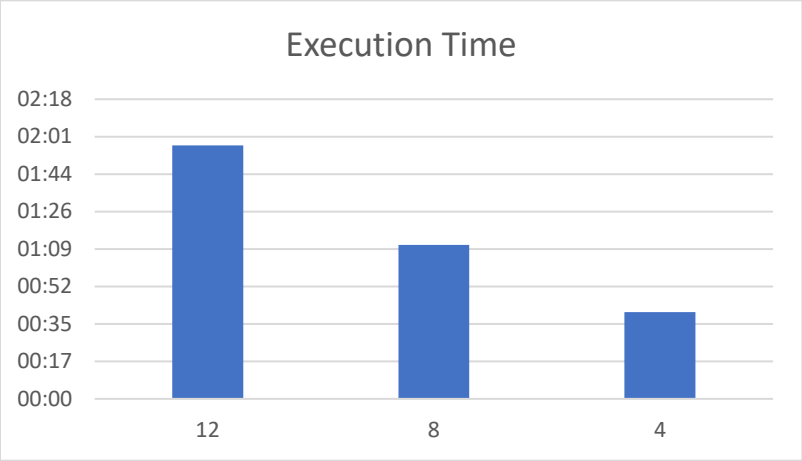


Figure 4.7 Execution time for different number of cubes

Of course, with less cubes the execution time is going to be lower. What we are looking for here, is there is a significant change in inference time when it's trying to detect more cubes. To approximate this, we can again use the average time per cube metric which we used in the previous part.

So, after calculations, their times are 10s/cube for 12, 9s/cube for 8 and 10s/cube. Based on these results there doesn't seem to be a significant impact.

4.3.3 Resources

Finally, let's see how much power is needed for different numbers of cubes.

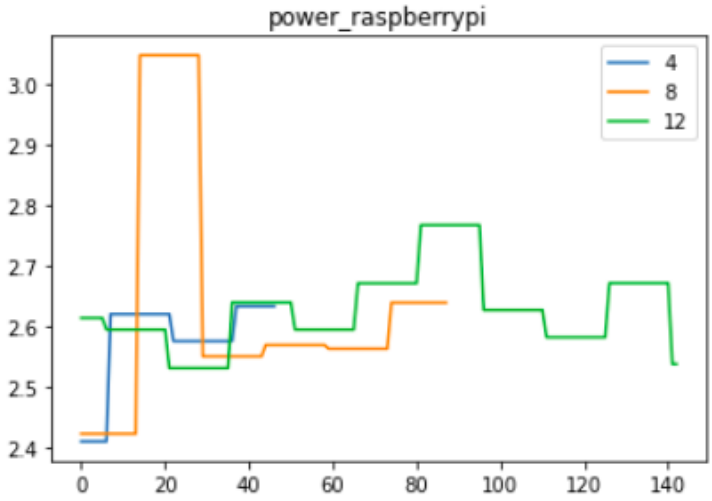


Figure 4.8 Power draw for different number of cubes

To get a better understanding of these results we have to factor the number of inferences needed to detect all the cubes. For 12 cubes, 4 inferences were run, for 8 cubes 2 inferences were run and for 4 just one was run. So, the power spikes correctly match the expected results.

In terms of overall power, all three experiments had similar results with the only anomaly being the initial power spike in the 8 cubes experiment.

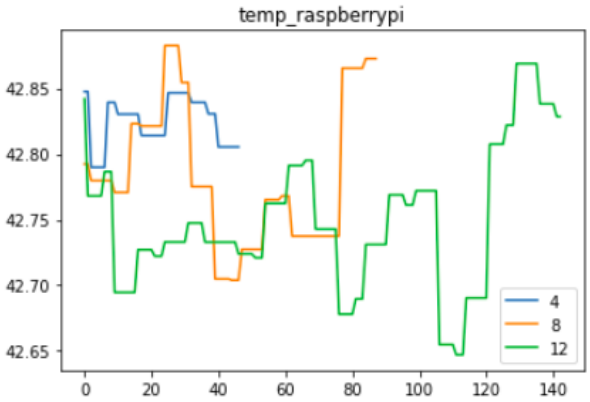


Figure 4.10 Temperature of the raspberry during the

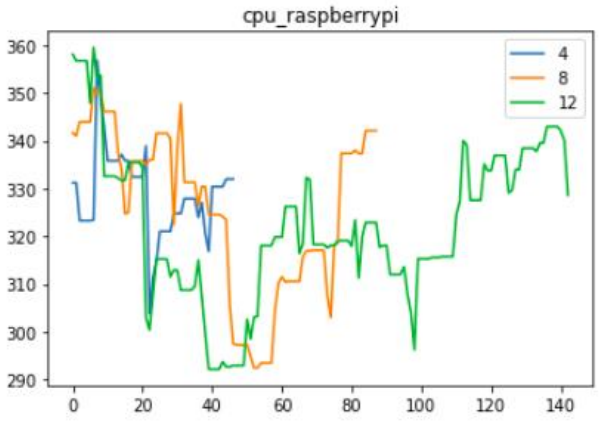


Figure 4.9 CPU utilization during the experiments

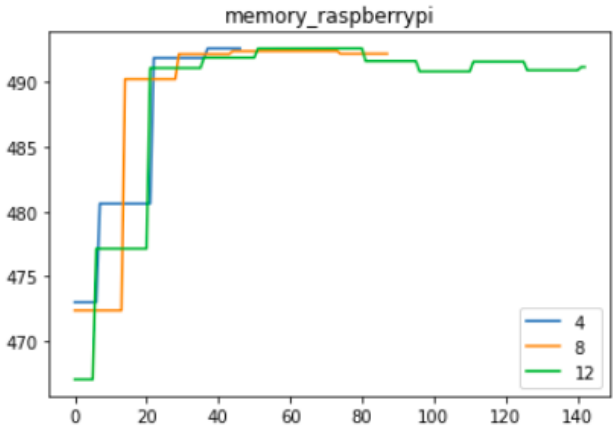


Figure 4.11 Memory usage during the experiments

Figures 4.8,4.9, and 4.10 show that temperature, CPU utilization and memory usage are not affected much by the number of cubes.

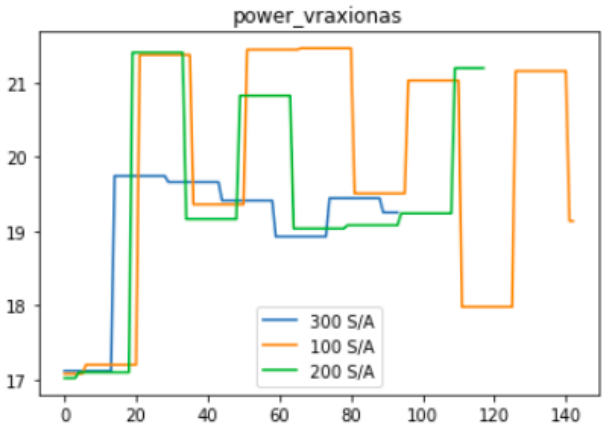


Figure 4.12 Power usage of the robot arm during the experiments

Figure 4.11 also shows that there isn't a big difference in power usage when changing the speed and acceleration of the robot.

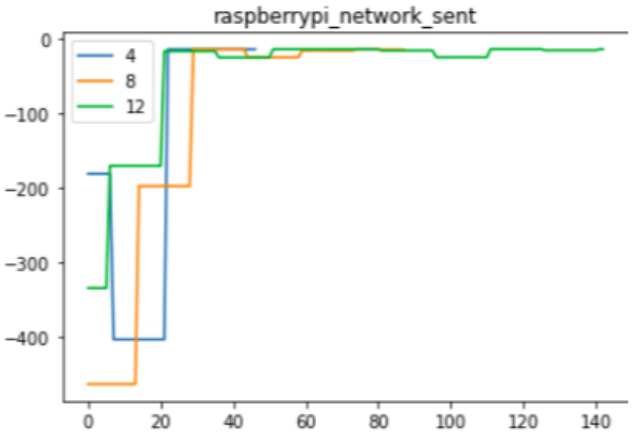


Figure 4.13 Network sent during the experiments

Finally, network sent, which is the data transferred between the raspberry and the TPU also doesn’t change much for a different number of cubes.

Chapter 5

Conclusions

5.1 Conclusions

Summarizing, in this thesis we created a system using a robotic arm and a camera to detect, pickup and sort cubes based on their color. The main challenges were to find different methods to detect the position of the cubes and identify their colors and then find a way relay those positions in the coordinate system of the robotic arm. After these challenges were solved, we used a monitor stack and run multiple experiments to see how each method and parameter of our systems affects its performance.

From the previous experiments we can end up with some conclusions about the various parts of our system.

Starting with our calibration method, it seems to be working good enough in most situations. However, it has a noticeable error margin, which can result in unsuccessful pickups especially when paired with methods that don't exactly detect the center of the cube. This means that for situations where high accuracy is desired, this method is not suitable.

The color detection method also produces very satisfactory results. During the experiments there wasn't a situation where a detected cube was sorted into the wrong color. Even so, this was after a lot of trial and error to figure out some common outputs, and no harsh lighting conditions. During our testing if there was a direct lighting source above the cubes, the method couldn't correctly identify the color because the detection point was mostly white from the light. A possibly better method would be to calculate the color based on the whole detection region instead of just one point.

For the 3 detection methods we got some conclusive results. The TPU method was the most accurate by far, not only be correctly detecting the cubes but also identifying their color. Following it was the template matching method, which even though it correctly detected the cubes the detection region was slightly off resulting in some unsuccessful pickups. The least accurate method was the contour detection which mostly struggled due to lighting conditions, something that was more apparent with darker colored cubes.

In terms of speed, the template matching algorithm has the best results, closely followed by the contour detection method and last is the TPU method which was significantly slower than the previous two.

For overall resources used, contour detection was the most resource heavy, by requiring the most power, with the other two methods being more similar with a slightly lower power consumption.

We also got some results about the performance of the system when changing the speed of the robotic arm and the number of cubes.

The speed of the robotic arm does play a role in the accuracy of the system, although by a small amount and with a combination of other factors such as the accuracy of the method used. The power draw of the robot also doesn't fluctuate much for different speed and acceleration.

Changing the number of cubes also slightly affected the accuracy of the system, although this probably is a result of the error from the calibration method, since more cubes are spread out in the working area.
Regarding the other metrics, no significant change was detected in resources used or in execution time relative to the number of cubes.

5.2 Future Work

Taking into consideration our conclusions, there are some improvements and changes we could make to improve our system.

The first obvious change is the calibration method. A technique more commonly used in the industrial field, is to calibrate the camera itself, by finding its internal parameters like the focal length and the lens distortion parameters. This method not only can provide better accuracy results but is also independent of the camera's position. This means that once calibrated, the

camera can be easily relocated without needing calibration again, thus making it also able to be mounted on the robot for applications where a moving robot is required.

For the detection methods, there are some improvements we could make for each method like adjusting the detection area for template matching and making contour detection less susceptible to light. In addition, we can use new methods like feature matching or create models based on different architectures like YOLO or Single Shot Detector.

Finally, we can add new features to our system. For example, we can use the conveyer belt kit provided by DOBOT which would really put to test the accuracy and speed of each method.

References

[1] DOBOT | Robotics Solution Provider for STEAM Education, Industry & Businesses. [online] Available at: https://www.dobot.cc/

[2] Netdata | Monitoring and troubleshooting transformed [online] Available at: https://www.netdata.cloud/

[3] Prometheus | Monitoring system & time series database [online] Available at: *https://prometheus.io*

[4] Consul | Consul by HashiCorp [online] | Available at: https://www.consul.io/

[5] Pydobot GitHub repository [online] | Available at https://github.com/luismesas/pydobot

[6] Dobot product overview [online] | Available at: *https://www.dobot.cc/dobot-magician/product-overview.html*

[7] Dobot Studio [online] | Available at *https://www.dobot.cc/downloadcenter.html*

[8] Blockly Dobot documentation [online] | Available at *https://www.dobot.cc/online/help/dobot-m1/79.html*

[9] Coral USB Accelerator [online] | Available at *https://coral.ai/products/accelerator/*

[10] USB Accelerator datasheet [online] | Available at *https://coral.ai/docs/accelerator/datasheet/*

[11] Coral USB Accelerator Compatibility [online] | | Available at: *https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview*

[12] Coral API [online] | | Available at: *https://coral.ai/docs/reference/py/*

[13] EfficientNet Overview [online] || Available at: *https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html*

[14] OpenCV [online] | | Available at: *https://opencv.org/*

[15] Webcolor library [online] | Available at: *https://pypi.org/project/webcolors/*

[16] Blender [online] | Available at: *https://www.blender.org/*

[17] Maya [online] | Available at: *https://www.autodesk.com/products/maya/overview*

[18] ZPY [online] | Available at: *https://github.com/ZumoLabs/zpy*

[19] Dobot Communication Protocol [online] | Available at: *https://download.dobot.cc/product-manual/dobot-magician/pdf/en/Dobot-Communication-Protocol-V1.1.5.pdf*