AI implementation in the Fisheye camera

by

Jason Dominguez

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Robotics Engineering

by

October 2022

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.
Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

APPROVED:

Smith, Therese

Abstract

The goal of this project is to take an object detection AI and implement it onto a Raspberry Pi Zero 2W. Different AI's will each be compared to see which one will best fulfill the project requirements. Once having picked which AI is most optimized for the task, documentation on the system use will be created in order that an average user will know how to turn on the unit in order to start the object recognition. Additionally, a different set of directions will be created in order to show how to create a new model for the system to use. Other programs will be created in order to use the camera, such as a video recorder and a still image storing program.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Problem

Autonomous underwater cameras can play an important role in exploring the oceans. They allow image capture for future image analysis and object or event detection. However, a problem with the current underwater cameras is they can be power hungry as well as the storage space for images can be limited. Resulting in a the camera system being tethered to something above water to transmit images and give power which can limit it over all depth as well as be a hazard to ocean life. In order to combat the issue of power and storage Raspberry Pi 3 and 4's are being used fo recording video. This is a great step as Raspberry Pi's are compact can be lower power and can hold a lot of storage. However there are other alternatives to the pi4 that can be lower power and still do the same tasks. The Raspberry pi Zero 2W is a good alternative as it is lower power while also having enough computational power to preform some basic functions. In order to fully unitize the power on the Pi Zero as well as save storage space the goal of the project will to implement basic object detection onto a Raspberry Pi Zero 2W.

1.2 AI

Object detection with AI can be used for a number of applications as such as surveillance, robotics, or quality control. Using AI in a system allows for a smaller form factor as a human interface is no longer needed. Using AI had been known to be a struggle due to the immense computing power that used to be required in order to run. However, in recent years there has been a huge leap forward in using AI, as there are new architectures that allow for a model to run on micro computers, such as the Raspberry Pi. There are a lot easier interface tools for using AI. There are systems such as TensorFlow Model Maker that are able to build an entire model with fourteen lines of code. This allows for people to create their own models to allow use in many more applications ([KH22]).

1.3 Existing System

One AI application that is being pursed is monitoring the ocean. One such application that it is currently in use is for undersea surveillance off the coast of Greece. A system is called “NOUS”, is able to monitor the area around the shipwreck and allow for basic object detection, such as scuba divers. This system was implemented in 2020 and is something that is active and working, there are 5 cameras in the system. Each camera has a raspberry Pi 3 or 4 which is able to screen the video feed and allow for object detection ([nou]). A different system in Canada was designed in 2020 and was able to stay underwater for fourteen days on one battery charge allowing it to work for long periods of time autonomously with little interface needed ([MBC+20]).

Chapter 2

Related Work

This project has the goal of adding basic object detection to an underwater camera to allow for long term surveillance of systems or areas with minimal electrical power capabilities.

2.1 Underwater Vision

There are a couple of ways that currently exist in order to observe objects in the ocean: acoustic and optical imaging. For example, a common way for fish to be monitored is by using acoustic imaging. This allows for the fish population and different fish behavior to be monitored. This system works, as it is robotic in nature and not sight dependent, meaning lenses and cameras do not need to be constantly cleaned or maintained in order to observe the environment. However, a problem with this is that there is no color information and the identification of fish must be done solely on the size of the fish in the image. This is where cameras can play a role in underwater observation, as the item the camera is trying to observe does not have to be solid. This can be crucial for detecting things like oil leaks or damage to an undersea cable ([SHA22]).

2.2 Hardware

There a multitude of different micro computers that are available on the market. A very common and popular one is the Raspberry Pi. This is due to its small size, cost and the amount of computing power that is available. It was also used in the "NOUS" system as stated above, showing that it is capable of running AI. For our project a Raspberry Pi Zero 2W will be used. There a couple of reason for this: size and power. An important factor for the camera is that the system should be as small as possible. This will allow for it to be placed in more locations while also allowing for the system to lighter, making it more portable. Another factor is power consumption. The Raspberry Pi4 has an idle current consumption of 516mA. However, when it is under load such a running an AI model, the system will draw more than 1A. This is a lot of current when running on a battery ([Zah22]). However, the Pi Zero 2W, consumes 270mA when idling and 370mA when under a load. This lower power consumption is the biggest reason the Pi Zero was chosen as the system is intended to be used for long periods on battery power ([Zah22]).

2.3 Current AI's

There are a number of different algorithms that are able to run on a Raspberry PI, such as: "EfficientDet", "Mobilenet", "Yolo", and "Resnet". As well, there are multiple versions of each architecture, for example YOLO has v1,v2,v3,..,v7. With each version comes faster speeds or greater accuracy. We trimmed down the list of the different AI's evaluated based on advertised features that we desired. The three that were looked at in more detail were "EfficientDet", "MobileNet", and "YOLOv5". A feature important to consider when making the model is over fitting.

The base model was trained on roughly 3000 images per item. This implys a new data set should be equal to or less than that in order to prevent over fitting.

2.4 Different stage AI's

There are multiple types of AI's that each have their own advantages and disadvantages. In the world of AI's there are two different types of AI: a one stage and a two stage detector. Both single and double stage detectors are able to run object classification. The difference between the two detectors is how the AI looks at the image coming in. A one stage will make a fixed amount of predictions based on a grid. A two stage will leverage a proposal network to find different objects and then use a second network to fine tune the guess of the object ([Jor18]). The two stage is a little more accurate, but it requires more computational power in order to work. It is also slower due to the object being analyzed twice before the final output. Knowing that the final project will be run on a Raspberry Pi Zero 2W, the better approach for our project would be to use a one stage AI, mainly due to the fact the Pi Zero 2W only has 0.5gb of RAM.

2.5 New Tech

This project will entail trying to find the best preforming object detection model that is able to run on a Raspberry Pi Zero 2W in order to detect basic under water items such as fish and other species. A collection of different models will have to be built in order to see which one preforms most optimally on the Pi Zero. Another part of the project involves creating and utility programs, such as a camera AI labeling program that will help with taking new pictures of objects for training, as well as a program that is able to act as a video recorder.

Chapter 3

Overall Design

3.1 Requirements

The project has several constraints: the AI model must run on a Raspberry Pi Zero 2W due to the Pi's small size and low power consumption, for extend battery life. Another problem when using the Pi Zero 2W for AI object recognition is it contains only 512gb of RAM, which is a low amount for an AI model which usually require many times more RAM. Another requirement is the camera being used is a Sony IMX477 sensor due to compatibility with the Raspberry Pi and because the underwater camera was designed to use this sensor size.

3.2 Prove AI Reliability

In comparing a one stage to a two stage detector, the decision was made to use a one stage detector. A one stage detector model runs faster and there is a greater availability of open source one stage detectors for the Raspberry Pi versus two stage detectors. After picking the type of detector, the model architecture has to be chosen. Doing research revealed several different algorithms that can be run on a Raspberry Pi including: "EfficientDet" ,"MobileNet", "YOLO", and "Regions with Convolutional Neural Network (R-CNN)". However, due to time constraints only three of the models listed were pursued: EfficientDet, MobileNet, and YOLO.

For development and testing of the different models a Raspberry Pi 4b 8GB micro computer was used in order to make evaluating the different models a little faster, the PI 4 also allows for easy connection to an Ethernet connection, making downloading libraries easier. The first architecture tested was EfficientDet. Adafruit corporation creates a product called the “BrainCraft hat” that provides different interfaces in order to help make developing AI systems more user friendly. It has a display screen and a set of push buttons. Unfortunately, this hat does not help with the speed of the model. The hat has a tutorial provided by Adafruit that shows how to run a model on the hat with TensorFlow. However, the code that was used in the tutorial didn’t work. The problem was that the code was developed to use python 3.7 not 3.9, which is what the current test setup was configured for ([ARLW20]). Using “pyenv” the python version number was able to be changed but a new error appeared: the camera signal was not able to be fed into the program. Thinking was that there was perhaps a problem with the program itself, as that program has not been updated in a while and that the camera being used for the test set was not exactly the same camera the tutorial used, a new tutorial was tested.

The following tutorial uses PyTorch and uses a MobileNet V2 architecture([Ric]). The new tutorial uses OpenCV to capture an image from the camera, then send the image to the model. This method should prove easier than the last program as it will be able to show any errors in the program easier and point to whether it is a camera problem or a model problem. This helps debugging as any camera issues will result with an OpenCV error and any model issues will show up as an error related to PyTorch. After compiling the program and running it once, there was an error that OpenCV was not able to grab a frame from the camera. After looking into the reason, it was found that the new version of Raspberry Pi OS released in January of 2022 uses a new version of camera capture protocols called libcamera.

The problem is that current OpenCV is not compatible with libcamera causing it not to work.

In order to bypass this issue, user “Stuartofmt” came up with the idea to have libcamera-vid act as a streaming service and then OpenCV would just latch onto the url allowing OpenCV to get the video feed ([Stu22]). Once the camera signal was able to be feed into the program the model could then be utilized. After modifying the original code to include the server program, the model was indeed able to work! Unfortunately after doing some testing with the program it was found the model was pretty inaccurate mainly due to the fact the model was an image detector and not an object detector. An example was a cup: the model acted fine when the cup was empty, but when scissors or other objects were placed inside of the cup the model was much more inaccurate identifying the cup. Noticing this problem, a different type of model was used: an “object detector”. The architecture that was used was “EfficientDet-lite” which was trained on common object in context(COCO) which was built in TensorFlow. This model preformed far better and was able to pick out the different objects in the picture while also running as a faster FPS compared to that of the MobileNet v2 model.

The next model that was tried was a YOLO v5 model, this architecture worked as expected in being able to identify some basic objects ([Dol22]). However, the fps was very slow compared to the other two models. As a last test MobileNet was tested again. This time with an object detector model ([Ost22]). This model was trained on a different dataset that COCO but it seemed to have a faster FPS for some objects and was a lot more confident in the identification compared to EfficientDet model.

After seeing how the different models acted, the next step was to create a custom model. The decision was made to try training an EfficientDet model because of its simplicity to train. TensorFlow created a program called TensorFlow model maker that allows for easy transfer learning of EfficientDet models. Using model maker only requires fourteen lines of code in order to use transfer learning on the model ([Kon22]). The tutorial that was used in order to build the model was written by Khanh Khanhlvg and Nhan Ho D.([KH22]). Using the tutorial and model maker a custom model was able to created. It was trained on planes and different vehicles. The data was a mix of two different sets from Roboflow Universe ([Cha22, oS22]). This data set in total had 2442 images, however the distribution of the training and test data was not very well distributed. Another problem was that in some of the classes there were only as few as 19 pictures of the item which was insufficient for proper training. But, on objects such as planes, which consisted of most of the pictures, the model was a lot more accurate. Though the distribution was not correct, the mean average precision (mAp) of the model was really low.

Looking back over models that were tested, MobileNet was the better performing of the two.

It was selected another MobileNet model for trial. This proved to be harder than expected as there were many issues in following the tutorials. This was due to the fact a lot of the tutorials were built 4 or 5 years ago and hence many of the versions of code were not compatible any longer. After searching online, there was a tutorial created by Google that teaches how to use transfer learning for a MobileNet model ([cor]). What this tutorial did differently was that it used a Docker image that already had all software and settings correctly configured in order to ensure that the model would work. Additionally, the tutorial also came with pre-trained data. This was important as it allowed for better debugging since the data is guaranteed to be correct and the errors would be related and assignable to the Docker image itself versus the main operating system. This was helpful as there were a number or errors related to the Docker image when connecting to the internet through a proxy. As well as, data images could not be downloaded directly onto the Docker image due to network security restrictions. In order to bypass this issue, the images were copied onto the main computer OS and then copied into to the Docker image. Once all the issues with the model were resolved, there was an additional issue that arose: the model would start training but then suddenly stop. It was found that the computer training the model had insufficient resources to train the entire model, but fortunately only training the last few layers worked.

In order to compare the two models fairly, both had to be trained on a common custom data set. Knowing the camera will be placed under water, a new data set was picked containing images of fish ([fyp22]). In order to judge the accuracy of the model, cross validation was used to determine the average accuracy of the model. The dataset consisted of 1017 images grouped into five different classes. For the EfficientDet model, the images were easy fed into transfer learning as the model maker automates many of the steps for the user. However, for the MobileNet model, several custom sub-programs had to be created in order to create the csv files, text files, and tri-maps images which were all required in order to create a TFRecord file. The TFRecord would then be input to the model it was a general file that told the model the location of the images, class name, annotation and other data important for creating the model. This was implemented by a mix of both modifying code that existed on the Docker image and by writing new programs from scratch to make the required training files. Once all the files needed were generated and joined together, the models were trained. First the EfficientDet models were trained and evaluated. Then, the MobileNet models were trained and evaluated. In total there are ten models, five from each type of model. Upon testing the models there was a condition that was not originally considered: the model was trained assuming that there is a fish in every image but, this may not always be the case. Remedying this error, a new dataset was created, but this time entailing seven different class definitions: fish, healthy coral, bleached coral, marked null, dead coral, and turtles ([fyp22, The22, Hos22]). The steps for cross validation were repeated with the new data set and the results were different from the previous model. Once all the models were complete, working evaluation of the models began.

3.3 Essential Utility Programs

The camera project also needed a number of utility programs which required development. One program was a camera image labeling application. This is a program that has the goal of acquiring many quality camera images and store them in a specific object directory folder. The program also examines each image to ensure it is not too blurry before adding it to the object directory. This is important because the frame rate on the camera is slow and there tends to be many useless blurry images. Another application that was written was a digital video recorder that starts once the computer power is turned on. The video is stored on a separate SD card from the computer operating system as during the development, PI gets powered on and off regularly and can cause data corruption and to avoid filling up the SD card too quickly with video.

Chapter 4

Results and Conclusions

4.1 Test Set up

Due to limited project time and poor frame rate performance, YOLO was not tried with a custom model. We selected both the better performing EfficientDet and MobileNet for the comparison. The model testing was done on a Raspberry Pi Zero 2W running Bullseye 11 0222-04-04. There was no fan running on the CPU during the testing but a cool down period between testing the different models was given in order that each test be fair. For the results of the frames per second (FPS), CPU usage, and RAM usage parameters, the Pi OS was turned off in order to allow for as much CPU power put towards the model as possible. Each test was run for 2 minutes and both the average or the max value was gathered. During testing of the models both were converted to TFLite models and run in the same python program, with the only difference being the model used.

4.2 Results

There was not a GUI displaying images in order to evaluate maximum model performance. To calculate the average fps for each test, the total fps for the entire execution time were added together and then divided by the amount of readings that were gathered. Next, the CPU usage was collected, using the htop command

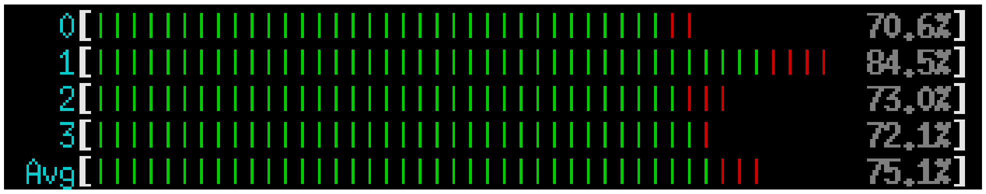
Table 4.1: FPS Results(FPS)

	Run1	Run2	Run3	Average
EfficientDet	1.1	1.08	.95	1.04
MobileNet V1	3.42	2.86	3.12	3.13

which visualized the total CPU usage as well as the usage of each CPU core on the pi. The two models had different tolls on the CPU cores. The MobileNet model used all available cores, resulting in a higher over-all CPU usage average. However, when running the EfficientDet model it was noticed that only one core at a time would get to 96% usage while the other cores were around 84% usage this is something that can be seen in figure 4.1 and figure 4.2. Because only one core had maximum usage EfficientDet showed lower average CPU usage.

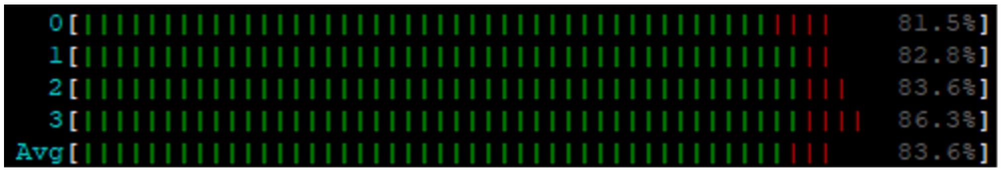
Table 4.2: CPU Usage (%)

	Run1	Run2	Run3	Average
EfficientDet	85.1	89.5	89.8	88.13
MobileNet V1	96.1	95.4	95.8	95.8



```
  0[                70.6%]
  1[                84.5%]
  2[                73.0%]
  3[                72.1%]
Avg[                75.1%]
```

Figure 4.1: CPU usage while running EfficientDet



```
  0[                81.5%]
  1[                82.8%]
  2[                83.6%]
  3[                86.3%]
Avg[                83.6%]
```

Figure 4.2: CPU usage while running MobileNet V1

The memory usage was also taken by using htop. The results are shown below. It should also be noted that without running any programs and just the terminal the Pi's memory usage is 60MB. In order compare the sizes of each model, files sizes

Table 4.3: Memory Usage (MB)

	Run1	Run2	Run3	Average
EfficientDet	149	150	153	151
MobileNet V1	142	148	148	146

were compared and recorded from Windows File Explorer.

Table 4.4: Size(KB)

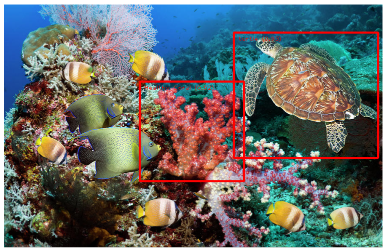
	Size
EfficientDet	7227
MobileNet V1	5456

To compare the accuracy of the models the Average Precision(AP) calculation was used. This process builds the model and then using the just built model, it is feed a set of known pre-marked images. Then the accuracy of the model in selecting the right part of the image, as well as labeling it correctly, is evaluated. A problem with basing the accuracy of the model on the AP was that there were some classes of the model that were under represented. The poor accuracy of these underrepresented classes brought down the over-all average significantly, something that can be seen in 4.4. The results for each arrangement of the data set is shown in table 4.5. The first column is where the first fifth of the data was used for validation, while the remaining four-fifths of the images were used for the training. The next column is where the second fifth of the data used for validation and remaining four-fifths used from training, and so on. The higher the AP, the more accurate the model identified objects. To see how the model acts in the real world it was run

Table 4.5: Accuracy(0-1)

	Split 0	Split 1	Split 2	Split 3	Split 4	Average
EfficientDet	.415	.415	.427	.428	.432	.4237
Mobilenet V1	.277	.255	.291	.278	.282	.2766

on a Raspberry Pi 4b this is because the Pi Zero does not have enough RAM to run


(a) EfficientDet

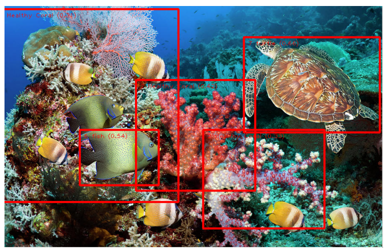
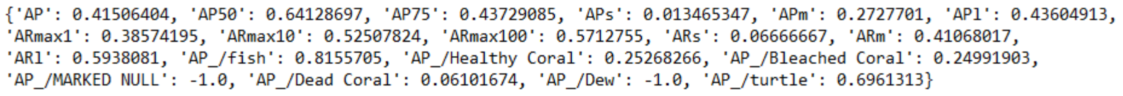

(b) MobileNet V1

Figure 4.3: Comparison on output from the models. Coral image taken by Georgette Douwma ([Dou]).



```
{'AP': 0.41506404, 'AP50': 0.64128697, 'AP75': 0.43729085, 'APs': 0.013465347, 'APm': 0.2727701, 'APl': 0.43604913,
'ARmax1': 0.38574195, 'ARmax10': 0.52507824, 'ARmax100': 0.5712755, 'ARs': 0.06666667, 'ARm': 0.41068017,
'ARl': 0.5938081, 'AP_/fish': 0.8155705, 'AP_/Healthy Coral': 0.25268266, 'AP_/Bleached Coral': 0.24991903,
'AP_/MARKED NULL': -1.0, 'AP_/Dead Coral': 0.06101674, 'AP_/Dew': -1.0, 'AP_/turtle': 0.6961313}
```

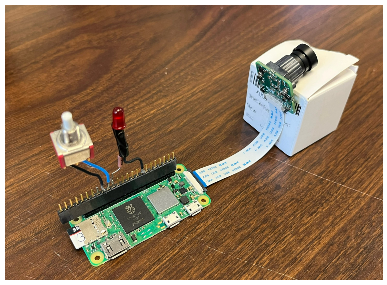
Figure 4.4: Break down of model accuracy for EfficientDet

the desktop version needed to show the image output. While running EfficientDet it was noticed that there was a some lag from what the camera was pointed at and what the model was analyzing. Keeping in mind the programs are identical except for what model was selected, this would mean the models should be having about the same amount of delay. To show the lag time between the two models the results can be seen in Table 4.6.

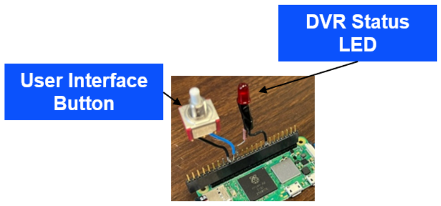
Table 4.6: Lag Time(s)

	Run1	Run2	Run3	Test4	Test5	Average
EfficientDet	2.04	1.85	1.84	1.98	1.85	1.91
MobileNet V1	1.00	0.88	1.08	0.88	1.00	0.97

In addition to evaluating the models additional programs were created in order to help ease development of the underwater Raspberry Pi. This included making a recording program that once the Pi is started up it will automatically start recording video feed of the camera and by pressing a button will save the video as well as shutdown the pi in order the battery can be unplugged. Additional hardware was created in order to have some user feed back.



(a) Side view of Pi Zero



(b) Close up View of the Hardware

Figure 4.5: Images of the Recording Hardware

4.2.1 Discussion

Looking over the results it shows that MobileNet preformed better compared to EfficientDet. There are a number of results that show this, looking at the FPS MobileNet has an average of 3.13 compared to that of 1.04 from EfficientDet. The difference in speed is a big factor as the faster the model can process images the more likely a new object can be detected. What also matters is how much the images were lagging when running the EfficientDet model. In table 4.6 it shows that there is almost twice the amount of lag compared to MobileNet. It should also be noted that during the original testing with the EfficientDet model the FPS was faster as well as there was no apparent lag. Meaning the demo model was possibly of a slightly different architecture or that the transfer learning was not done correctly.

Image speed is not the only important factor for running the camera, other factors that are important for running the camera is the hardware tolls. Looking at the CPU it can be seen that the EfficientDet was using the CPU less compared to that of its competitor. The RAM usage as well as the storage were both in favor of MobileNet as there was less RAM usage as well as the MobileNet model was 2kB less than the EfficientDet model. This could have been do to the fact that the EfficientDet model was trained on the whole model compared to the MobileNet model which was only trained on the last few layers.

The last factor is accuracy this was in favor of EfficientDet. It should be noted that it was during the testing that it can be seen on how much the data set effects the model accuracy. Looking at figure 4.4 it shows a copy of the output from an EfficientDet model that was evaluated on the underwater dataset it shows the over all accuracy while also showing the accuracy of the different classes. Whjat is significant about this set of number is ythat it shows how the data determines the overall accuracy of the model. As can be seen the fish class has .81 mAp however some of the other classes like dead coral has an accuracy of .06, this results in a lower overall average precision. That being said both models were trained on the same data set, and EfficientDet had the higher over all accuracy. What should also noted is that when running the model across multiple images EfficientDet did not identify as many objects in the image as compared to MobileNet. MobileNet preformed better in FPS, CPU usage, RAM usage as well as using less storage, which makes it the better choice for this project as it fulfills more requirements of the project compared to EfficientDet.

The additional programs also worked as expected with some added features, such as that when a USB device is plugged into the Pi while running the program, all the videos stored will be transferred over to the USB as well as deleted from the Pi's memory. This frees up space for new videos saving the need to connect the Pi to a monitor.

Chapter 5

Conclusion

The project entailed creating and evaluating different models that are currently available for the Raspberry Pi. There were four criteria for th ideal model for the project, speed, hardware usage, storage space, and accuracy. The final goal of the project is to run the models on a Raspberry Pi Zero 2W, with a Sony IMX477 camera sensor. The model architectures that were chosen to be evaluated were MobileNet, EfficientDet, and YOLO. Following the different tutorials online each model type was evaluated to see how it preforms over as well as the Raspberry Pi 4b also has more IO ports. It was found that both EfficientDet and the MobileNet seemed to run at faster FPS compared to YOLO which was running at a slower speed as well as the accuracy seemed lower resulting in it, not being further evaluated. When training both MobileNet and EfficientDet on new classes, transfer learning was used with cross validation in order to understand the over all accuracy of the model architecture. In addition to the model accuracy, the CPU usage, RAM usage and size of model was also evaluated. Once having all the data it showed that MobileNet was the better model for the project requirements as it fulfilled 3 of the four project requirements. In addition to doing model evaluation helper python programs were created to make development with underwater camera easier.

5.1 Future Work

Given more time there are a number of other improvement that can be made. One is to compare more model architectures as comparing only two is a small number compared to the amount available on the internet. Another improvement would be to create a separate controller in order to help with making the interface easier. Since this camera is meant to be used in the field it can be hard to find easy access to a monitor and keyboard and view the model output.The controller will help to have easier control, and to see the output of the model from an easy interface. A good feature for the project would be to also implement on device training. This can be helpful for adding small objects or when a computer is not near by.

5.2 Lessons Learned

Throughout the project there were a number of different things that would have been good to know, first is that the new version of Raspberry Pi OS has it own camera interface called Libby and that the most recent version of Pi OS has Picamera2 installed making the interface with the camera just as easy as before Libby. Also knowing TensorFlow is great for beginners as it has good documentation and makes building models very easy however, for anything more complex such as using transfer learning for a different model beside EfficientDet, look at other solutions online.

Bibliography

[ARLW20] Lady Ada, Kattni Rembor, and Melissa LeBlanc-Williams. Adafruit braincraft hat - easy machine learning for raspberry pi. https://learn.adafruit.com/adafruit-braincraft-hat-easy-machine-learning-for-raspberry-pi, Oct 2020.

[Cha22] Ji Changkai. Plane dataset. https://universe.roboflow.com/-alrej/plane-a5v7t, Apr 2022. visited on 2022-09-27.

[cor] Retrain an object detection model.

[Dol22] Luiz Doleron. Detecting objects with yolov5, opencv, python and c++. https://medium.com/mlearning-ai/detecting-objects-with-yolov5-opencv-python-and-c-c7cf13d1483c, Mar 2022.

[Dou] Georgette Douwma. Green sea turtle over coral reef by georgette douwma. https://photos.com/featured/green-sea-turtle-over-coral-reef-georgette-douwma.html.

[fyp22] fyp. yolov4 dataset. https://universe.roboflow.com/fyp-bekec/yolov4-38k4c, Jul 2022. visited on 2022-09-28.

[Hos22] Parvej Hosen. Turtle dataset. https://universe.roboflow.com/parvej-hosen/turtle-f9xgw, Aug 2022. visited on 2022-09-28.

[Jor18] Jeremy Jordan. An overview of object detection: One-stage methods. https://www.jeremyjordan.me/object-detection-one-stage/, Jul 2018.

[KH22] Khanh Khanhlvg and Nhan D. Ho. Khanhlvg/tflite_raspberry_pi: Tensorflow lite's raspberry pi examples. https://github.com/khanhlvg/tflite_raspberry_pi/, May 2022.

[Kon22] Poulinakis Kon. Object detection at the edge with tf lite model-maker, Sep 2022.

[MBC+20] Xavier Mouy, Morgan Black, Kieran Cox, Jessica Qualley, Callum Mireault, Stan Dosso, and Francis Juanes. Fishcam: A low-cost open source autonomous camera for aquatic research. *HardwareX*, 8:e00110, 2020.

[nou] Peristera ancient shipwreck, greece. https://nous.com.gr/naxly_project/peristeras-ancient-ship-wreck/.

[oS22] University of Siena. vehicle detection dataset dataset. https://universe.roboflow.com/university-of-siena-dls9l/vehicle-detection-dataset, Sep 2022. visited on 2022-09-27.

[Ost22] Edgar Florez Ostos. Mobilenet ssd object detection opencv 3.4.1 dnn module. https://ebenezertechs.com/mobilenet-ssd-using-opencv-3-4-1-deep-learning-module-python/, Aug 2022.

[Ric] Tristan Rice. Real time inference on raspberry pi 4 (30 fps!). https://pytorch.org/tutorials/intermediate/realtime_rpi.html.

[SHA22] Collin D. Smith, Tyson W. Hatton, and Noah S. Adams. Monitoring fish abundance and behavior, using multi-beam acoustic imaging sonar, at a selective water withdrawal structure in lake billy chinook, deschutes river, oregon, 2020. https://doi.org/10.3133/ofr20221038, 2022.

[Stu22] Stuartofmt. Pi-notes. https://github.com/stuartofmt/Pi-Notes, Jun 2022.

[The22] Thesis. Classification of corals dataset. https://universe.roboflow.com/thesis-fcq63/classification-of-corals, May 2022. visited on 2022-09-28.

[Zah22] Hammad Zahid. How much power does raspberry pi consume while operating. https://linuxhint.com/power-consumption-raspberry-pi/, Apr 2022.