College of Engineering

ELEC 491 – Electrical Engineering Design Project

Final Report

Autonomous Drone SLAM, Positioning and Real-time Object Detection

Fall 2021

Table of Contents

I. Abstract

In the 21st Century, the usage of Unmanned Aerial Vehicles has increased exponentially. Unmanned Aerial Vehicles are now being used in the fields of construction, military, architecture, aerial photography, agriculture, forestry and such. Though these technologies have been around for a while now, manufacturers and researchers are embedding different features like autonomous flight and deep learning algorithms in an attempt to broaden the range of applications drones can be used in. Our project addresses the embedding of four main features into our custom built drone namely, autonomous flight, Simultaneous Localization and Mapping (SLAM), deep learning based object detection algorithm for car parking space occupancy detection, and a platform independent user interface for the ease of interactivity with the developed software. The steep growth of the population has resulted in a decrease in the empty parking spaces. This results in traffic congestion, excess consumption of fuel and the frustration of the drivers. Current solutions for the detection of empty car parking spaces usually require the installment of sensors or a surveillance camera. These solutions are not feasible since the current methods fail to cover large complex areas which change relatively fast in time. Our project offers a generalizable solution for the detection and classification of car parking spaces in complex or large environments using drones equipped with object detection algorithms and autonomous flight scripts. Our project is primarily targeting the parking lot operators and the shopping malls with huge parking space capacities.

II. Introduction

The amount of private vehicles owned increases as the human population continues to grow. In some of the cities, this increase in vehicles results in insufficient parking spaces. The search for parking spaces leads to waste of resources such as time and money. Moreover, the population due to extended fuel consumption during the search for empty parking spaces raises environmental concerns. A generalizable and cheap solution to finding empty parking spaces does not exist. All the existing solutions are either expensive and complicated to set up and repair or not customizable enough to be used in different circumstances. Our project offers a generalizable solution which could be used to detect empty car parking spaces in large and complex environments. We proposed an autonomous drone which could perform simultaneous localization and mapping (SLAM) and deep learning based object detection for car parking space occupancy detection which is controllable via a platform independent user interface that we have developed. Our solution satisfies the software requirements of the following. A platform independent user interface which can be used to interact with the functionalities of the drone. For this back end services and a complementary front end user interface are developed for the ease of user interactivity with the drone. During the flight of the drone, sensory data could be collected in the form of a rosbag file which could then be processed by Kimera's implementation of SLAM algorithm to generate the three dimensional map of its course of flight. Drone could perform deep learning based object detection which detects empty and occupied car parking spaces using the live video stream acquired from the camera mounted on it. The annotated live stream of the object detector can be simultaneously observed from the same user interface that was built to control the autonomous flight of the drone. The common solutions to the autonomous flight control, SLAM and car parking space occupancy detection regulate around the following methodologies. To begin with, the autonomous flight control is usually implemented using ardupilot and pixhawk autopilot hardware designs which run on px4 software. These solutions are open sourced and they provide low cost and high end autopilot systems for hobbyists and professional developers. We decided to use holybro's pixhawk 4 autopilot flight controller equipped with px4 on our drone. Having this allowed us to abstract the details of the hardware's that are available on the drone such as sensors and motors. Using this we were able to easily calibrate our drone and not have to worry about updates to the drivers of the hardware which could have resulted in serious

amounts of time spent fixing related problems. For interacting with the flight controller we chose Robot Operating System (ROS). There were other solutions available which were also compatible with our choice of flight controller such as DroneKit but these did not support simulations. With ROS, we were able to use Gazebo simulation software to model our drone virtually and test it safely in various environments. Moreover, ROS allowed us to code using python and c++ which we were very fond of. Furthermore, for the SLAM requirement of our drone we decided to use Kimera's implementation. Kimera is a c++ library developed and open sourced by MIT. It was our choice of library since it supported ROS which we also used for the development of the autonomous flight control, and it was supported by px4 which made collecting the required sensory data much easier. Regarding the available methods for the implementation of SLAM algorithms there were two major approaches. One relied on hard coded conventional algorithms to build the mesh and the other relied on deep learning to figure out the three dimensional structure of the environment. Though conventional implementations of SLAM are currently the state-of-the-art solutions which yield better results over the deep learning based approaches, a mix between the two extremist approaches yields certain gains and has the potential of being the future [1]. It is in this sense that Kimera's modular approach that can be used to leverage the benefits of this mixed approach caught our attention and made it stand out from the rest of the popular alternative libraries[2]. When it comes to the task of finding empty car parking spaces there are a lot of existing solutions utilized by shopping malls and other businesses with parking spaces, to make finding empty parking spaces easier. These solutions often include sensors for each parking space or a counter to show how many cars in total are in the parking lot at the moment and how many empty spaces are left. Counter solution is cheaper but lacks precision and does not indicate where the empty spaces are. On the other hand sensors located at each parking space can give more precise information to people, but comes at a significant cost of integrating many sensors. These sensors do often malfunction or struggle detecting cars therefore lowering the system's accuracy. Computer vision based solutions are easier to deploy, more robust and more customized. They leverage the fact that many places have CCTV cameras already installed for surveillance purposes and their streams can be integrated with a computer vision software over the Real Time Streaming Protocol (RTSP). There are various different approaches when it comes to implementing computer vision for car occupancy detection

problems. Easiest solution is to draw bounding boxes for each parking space in a stationary camera's field of view and then crop these boxes to feed them into a simple car, no-car image classification model. For each image, it compares the intersection of the bounding boxes for both classes to deduce whether car parking slots are empty or occupied [3]. This approach works quite well but it requires the user to set up the bounding boxes of parking spaces for every camera. Even though this would be a one time only setup, it prevents the system from being used in non-stationary cameras such as drones. Also, parked cars can obstruct the car slots lines which could cause the model to not detect the car parking spaces. To tackle these issues, there have been some attempts to train models to directly recognize occupied and empty parking slots. They detect the parking spaces with object detectors and then use a separate image classification model to decide if the parking space is empty or not [4]. This system requires two models and therefore more computing resources to run. Also, the papers that implemented this approach did not give test results on data that is different from the training data. They used different images from the same camera in their testing and training data. Since, the location of the parking spaces do not change, it is not clear if the object detector just memorized the parking spaces or actually learned something generalizable. To address these drawbacks another approach is to directly train object detectors to detect and classify empty/full parking spaces. Similar to this approach, YOLOv3 which is another deep learning based computer vision architecture was trained on the PKLot dataset [6] and achieved promising results [5]. We are inspired by the performance of VGG16 on CNRPark dataset [7]. We proposed to use YOLOv5 object detectors trained using PKLot dataset [6]. Out of all the YOLO family models YOLOv5 stood out for its accuracy/performance tradeoff which EfficientDet algorithms lacked a lot given we used the Jetson family of boards on the drone namely, nano.

III. System Design

With the rising popularity of unmanned aerial vehicles, implementation of drones for various industrial solutions has gained momentum. These drones are getting equipped with various sensors and specialized algorithms to achieve their tasks. Autonomous flight control, simultaneous localization and mapping (SLAM) and object detection are among the most commonly implemented algorithms on drones. Our project makes use of these aforementioned methods and on top of these builds a platform independent user interface for a user friendly user experience. For the development of the autonomous flight control system we used pixhawk 4 flight controller board which runs on px4 software. This flight controller abstracted the low level requirements of our design and saved us a lot of time by simplifying the process. It was with the use of this that we were able to not interact with the drivers of the hardware that was mounted on the drone such as GPS, barometer, inertial measurement unit (IMU) and motors using low level codes. Px4 software uses uORB messaging to communicate with the hardware that it is linked with. We needed a way to interact with this software to be able to read in sensory data coming from the drone and initiate the corresponding action that is proper with the desired outcome. For this we used Robot Operating System (ROS). Px4 had official support for ROS therefore we went with it. ROS had a package named MAVROS which we used in order to communicate with the flight controller on our drone using the code that we developed. Px4 uses uORB messaging to communicate with the hardware but it accepts MAVLINK messaging protocol to communicate with external software. Using MAVROS package, we were able to bridge the gap between our python code written using ROS which gets translated to MAVLINK messages which could be interpreted by the px4 software on the pixhawk 4 flight controller. Similarly, px4 could read sensory data from hardware in the uORB message format and convert it to MAVLINK messages which could then be read in python using the MAVROS package of ROS. This way we were able to read sensory data readings on the drone and interact with the hardware components such as motors to initiate actions simultaneously. ROS comes with different versions depending on the operating system so we had to choose one. Since the Kimera library which we were using for the SLAM implementation and the drone's on board computer Jetson Nano's OS was running on Ubuntu 18.04 (Bionic Beaver), we decided to choose ROS melodic which supported Ubuntu 18.04. ROS supported coding in python and c++ languages and we chose to develop in python for its

ease of use. Using python with MAVROS, we developed our custom flight scripts which performed certain tasks and required its task specific values as a command line argument. Afterwards, we needed a way to call these custom flight scripts with the desired outputs according to the users desires. For taking these parameters from the user such as which flight mode to use and with which parameters to perform that custom flight mode, we developed a platform independent user interface. Using this interface the user configures its desired task and submits it. Upon submission, the front-end parses the input from the user, embeds it in a JSON object that will be sent in the body of the corresponding request to the back end service. The back end service parses the incoming information in the body of the request, formats it so that the custom flight scripts will be compatible with it and then calls the custom flights scripts with the corresponding command line arguments. The flight scripts which are called from the back-end service runs on the Jetson nano which is our choice of on board computer. The Jetson nano is set up with ROS melodic therefore our python scripts run without a problem on the board. The scripts initiate an action using ROS, which then using the MAVROS package gets converted to MAVLINK messages. Since the Jetson nano is connected to the pixhawk 4 flight controller it receives these converted MAVLINK messages and acts accordingly by using uORB messages to interact with the hardware that it is attached to. To sum up, user asks for an action using the front-end web interface which then makes a HTTP request to the corresponding back-end services. The back-end services call the custom flight scripts with the corresponding parameters on the Jetson nano board which is mounted on the drone. Using ROS's MAVROS package the desired outcome is produced by the pixhawk 4 flight controller. For the development of the user interface we have decided to develop a web browser based user interface. This way we had a user interface which works platform independent. In other words, regardless of the operating system, the choice of web browser or the type of the device, the user would most likely be able to use our user interface to interact with our product. For the front-end development we decided to use the Angular 2+ framework. We benefited from Angular's component based design support. This way we were able to reuse our code and reduce the redundancy in the code written by being able to convert relevant elements of the user interface into related components. Moreover, Angular has a built-in typescript support. Typescript is a programming language that gets compiled into javascript so that it can run on web browsers. Unlike javascript, typescript supports object oriented programming and adds

static typing support. Having object oriented programming support helped us with implementing design patterns and it is much more like java language which we are more accustomed to coding in. We have leveraged the benefits of using popular javascript libraries in many places but two of them were essential namely, angular material and bootstrap 5. We have used Angular material for the component designs. In other words, we did not go through the process of designing and implementing user interface elements but instead, we have used angular materials components. These components were good looking and user friendly so they met our expectations. Furthermore, since web browsers are supported by many different devices with drastically different screen sizes, we needed our user interface to adapt its design to the screen size of the device dynamically. For this purpose instead of using media queries in css in a conventional way, we utilized bootstrap 5 library. Bootstrap 5 is the fifth release of the bootstrap framework which is open sourced by Twitter. Bootstrap is very famous in the industry and has many built in functionalities that makes designing responsive (adaptive to various screen sizes) web pages much easier.

For the development of the back-end services we used both Java Spring Boot framework and python's Flask framework. Java Spring Boot was used to develop the back end services that respond to the front-end web user interface's autonomous flight requests. Furthermore, Flask was used for simultaneously live streaming the annotated frames of the live video footage coming from the drone's on board camera which got processed by our car parking space occupancy detection model (YOLOv5). We decided to use the Java Spring Boot framework for its object oriented programming paradigm support. We developed our code using the Model View Controller (MVC) design pattern. Since our user interface allowed for multiple devices to connect simultaneously, we needed to handle concurrency in our code. Java Spring Boot's Bean concepts helped us a lot during the handling of concurrent objects and service calls. On the other hand, we had to handle concurrency using multithreading libraries of python on our own when we developed object detector live stream back-end service using python's Flask library. We used Flask for its easy to set up nature and its compatibility with python coding language. The object detector that we were using was already implemented in python. Therefore, implementing the live streaming back end service was much easier to work with when it was coded using the same coding language as the object detector was developed in.

For the object detection model for detecting car parking space occupancy as empty or occupied we used YOLOv5. We trained our models using a YOLOv5 model which is pre-trained on MS COCO dataset. We customized this model to detect empty and occupied car parking spaces. As a training approach, instead of training our model from scratch, we performed transfer learning on the pre-trained model to leverage the benefits of transfer learning such as reduced training time and improved accuracy. For the hardware requirements of training the models, we used Google Cloud services to create Deep Learning Virtual Machine and Google Collaboratory with Nvidia Tesla V100 and Tesla K80 GPU's respectively. We experimented with hyperparameters by training for small epochs and observed candidate models that we can continue with. We tried data augmentation using RoboFlow's services to add variation to the dataset. The Jetson Nano board on the drone had torch and torchvision versions that were not compatible with the Nvidia driver and CUDA version available with the JetPack SDK that Jetson Nano used. This led our object detector to run on CPU and not leverage the GPU power available on the Jetson nano board. To solve this issue, we used Docker. Docker is a popular OS level virtualization technology that eases portability of software among different devices. We developed our own docker images by modifying the Nvidia L4T ML docker image to meet our needs.

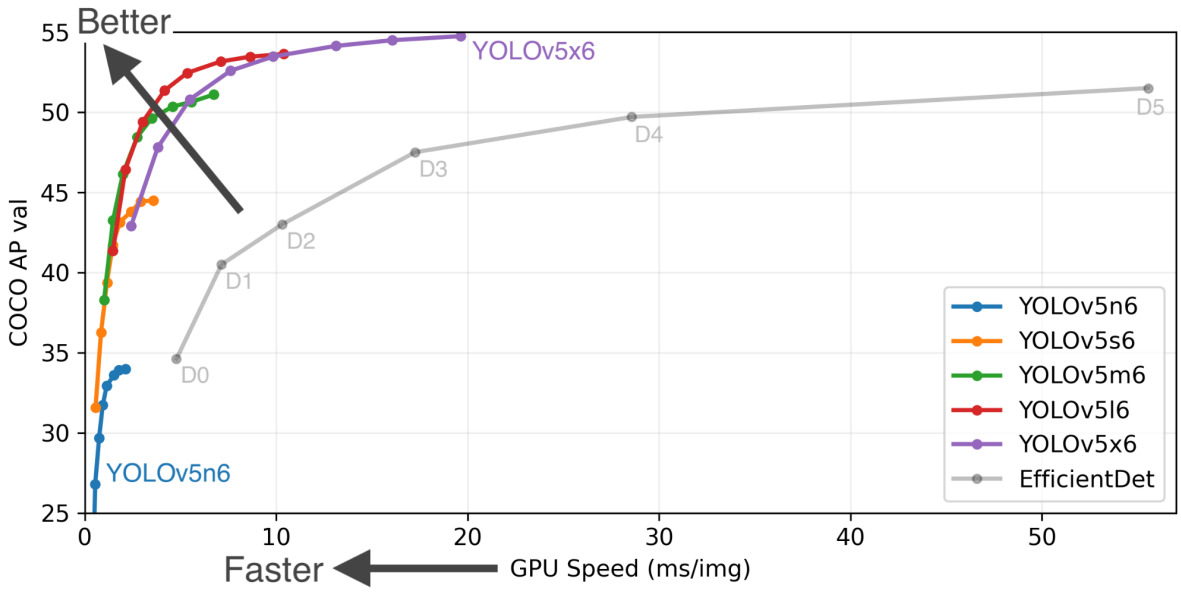

Figure 1: YOLOv5 AP val vs GPU Speed Graph [8]

For developing the simultaneous localization and mapping (SLAM) implementation we used Kimera which is a popular method around the autonomous movement and SLAM community. Kimera is a c++ library that handles the volumetric mapping of an environment using sensory and visual data. Simply put, it takes images and sensory data, connects the dots between different images to form meshes and uses the sensory data and Inertial Movement Unit (IMU) data to approximate the volumetric definition of the structure [2]. It takes the data from ROS. In other words, both the px4 software on the pixhawk 4 flight controller, the ROS and the Kimera communicate with each other.

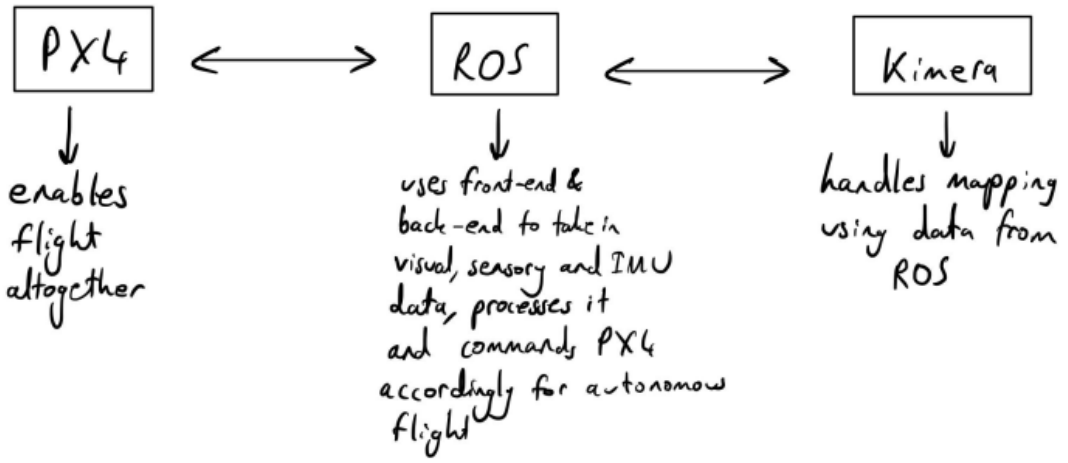

Figure 2: Relationship between Px4, ROS and Kimera

Finally, Jetson nano's firewall was modified using the ubuntu firewall (ufw) apt package. The corresponding ports for the ssh, user interface, back-end services and object detector live stream service were opened. Also, .bashrc file was modified with custom .sh files to start these services upon booting the jetson nano automatically when the drone is powered.

IV. Analysis and Results

Software was successfully developed to perform autonomous flight via the use of our custom flight scripts written in python coding language. These codes were integrated into the user interface that was developed. The integration was tested on both Gazebo simulations using IRIS drone model and both in real life test flights. During the tests we have observed that the use of the user interface brought about the corresponding actions from the drone successfully whether it be as simple as arming (propellers turn) the drone on demand or as complicated as following a pre-set flight path by the user. For the user interface, both the front-end and the back-end code was tested for handling multiple clients simultaneously and they passed the tests. Moreover, the front-end was tested for compatibility with different types of devices with different screen sizes. During our tests we have seen that our user interface worked from chrome, firefox, opera, safari in Laptops and Desktops running with Ubuntu, Windows and Mac OS. Moreover, Iphone pro 11 max's safari and Samsung Galaxy Ultra S21's browser's worked well too. During these we have witnessed that the layout of the user interface changed dynamically and adapted a user friendly layout in each case.

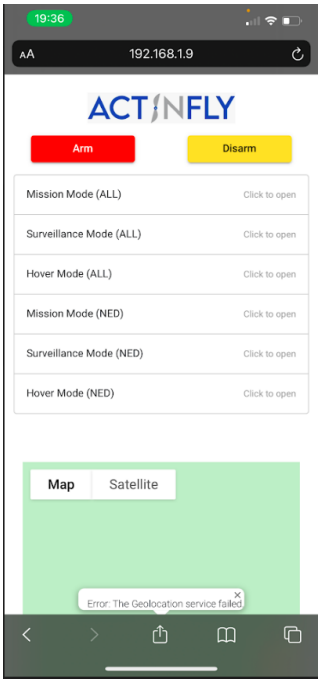

Figure 3: User interface's view from Iphone 11 Pro Max screen

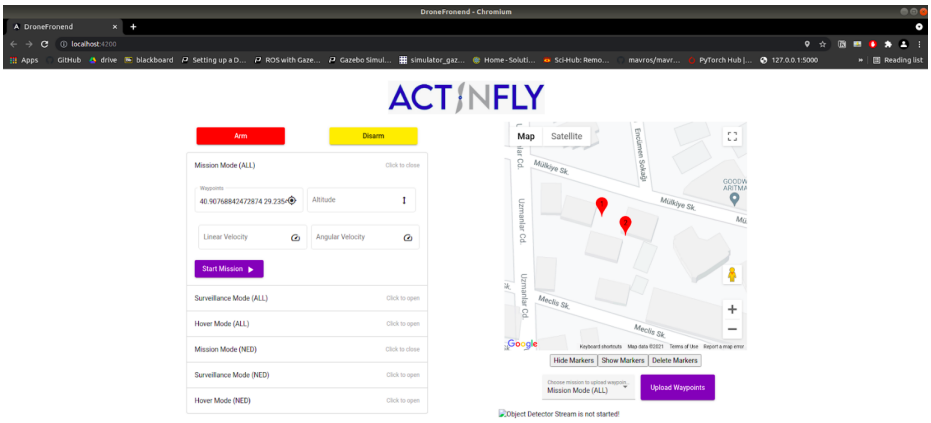

Figure 4: User interface's view from 23'' Desktop Monitor running on Ubuntu 18.04

Furthermore, the object detector is successfully implemented to work with the camera mounted on the drone and stream the frames annotated by the object detector (YOLOv5) in real time on the same user interface that we have developed for controlling the drone. The streaming is also tested for handling multiple users and it withheld our tests successfully. We have trained our custom object detector on the PKLot dataset using transfer learning to detect car parking spaces as empty and occupied parking spaces. By using transfer learning on a model which was already pre-trained on the MS COCO dataset we managed to yield higher accuracy in shorter training times due to the already learned lower features being leveraged. Due to the limitations in the PKLot dataset which we have used to train, we could not get very accurate and generalizable solutions for our task. To counteract the datasets limitations we experimented with various data augmentation techniques. Even though these helped us with increasing our accuracy scores, it could not get us to a feasible solution performance. The table below summarizes the general Mean average precision scores we got during some of our different experiments with the data augmentation types. Some of the techniques we used are horizontal flipping, cropping, rotation by 15 or 90 degrees.

Results of Augmentions YOLOv5

Dataset + model	Mean average precision @ 0.5	Mean average precision @ 0.95
PKLot		
Resize + Tile + All *(epoch: 200, batch:32)*	*54.119*	*1.787*
Resize + Tile + All yolov5m *(epoch: 200, batch:16)*	*36.864*	*13.986*
Tile + All *(epoch: 200, batch:32)*	*20.941*	*98.922*
Tile + All- crop - 90 rotation *(epoch: 200, batch:32)*	*35.664*	*19.176*
Tile + Rotation 15 + Sheer *(epoch: 100, batch:32)*	*40.287*	*23.082*

Figure 5: Yielded training results for various data augmentation experiments

We used the mean average precision metric to evaluate the performance of our model. Mean average precision reflects a good summary of metrics since it takes into account precision and recall at different confidence scores and bounding box overlaps. Specifically, we used mean average precision (MAP) at 50 and 95 percent overlap. The overlap is measured by comparing the bounding boxes from the prediction and label. We tried two versions fo YOLOv5, namely, small and medium. Performance improvements we could gain from the more complex medium model were not enough to justify the amount of computing power used. YOLOv5 medium was too slow to train compared to YOLOv5 small. Medium model got 0.72298 mAP@95 and 0.97096 mAP@0.5 in 4 epochs while the small model got 0ç.5401 mAP@95 and 0.9696 mAP@0.5 in 4 epochs. However the small model managed to train for 12 epochs in the same amount of time that the medium model took 4 epochs. Ath the end of the 12 epochs small model got 0.8314 mAP@95 and 0.98782 mAP@0.5 therefore outperforming the medium model. It is possible to get better performing models with YOLOv5 medium but YOLOv5 small is a lot more efficient. Given our limited resource in terms of GPU access we decided to continue training with YOLOv5 small models. Also, the fact that small models yielded higher fps values on the Jetson Nano board which we used to perform object detection was also another factor that affected our decision.

Figure 6: (YOLOv5m (batch size:8, trained for 10 epochs) detection outputs on a test image)

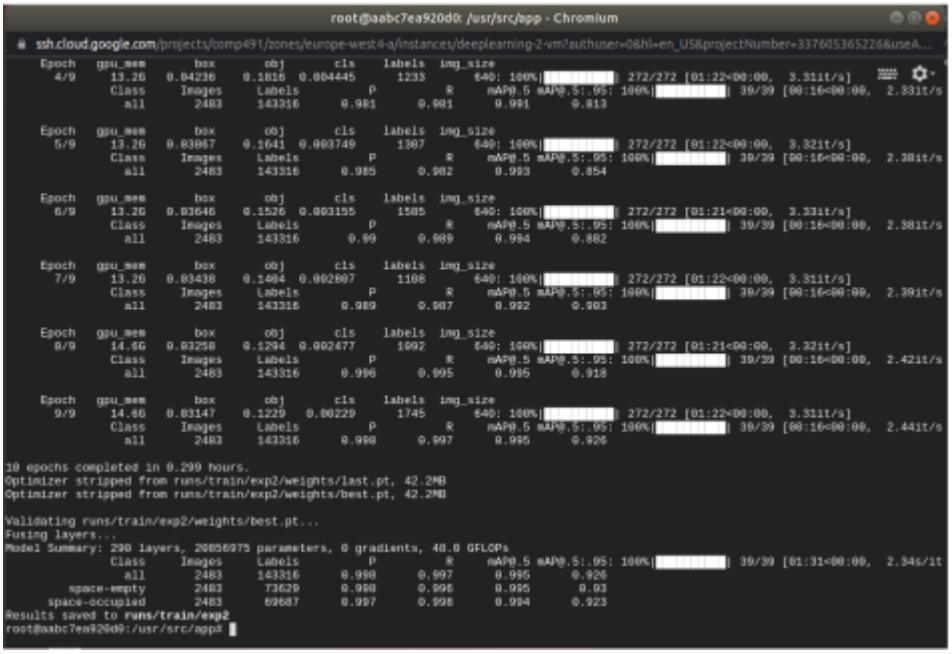

Figure 7: Google Colab Deep Learning VM instances shell screenshot that shows the YOLOv5m's training performance(batch size:8, trained for 10 epochs)

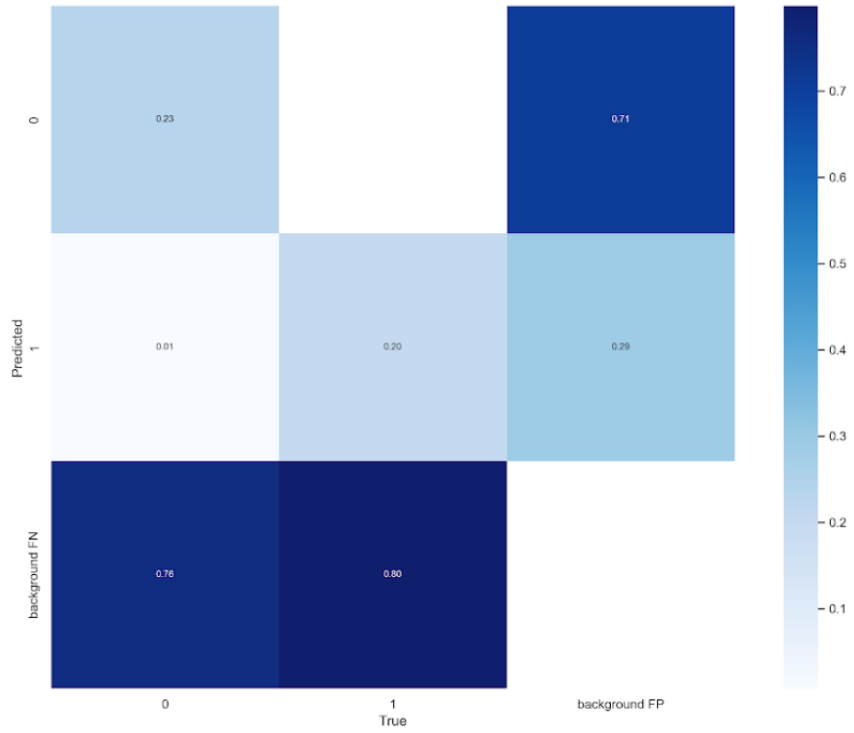

Figure 8: Confusion matrix for YOLOv5s with batch size of 32 trained for 200 epochs (rotation and shear applied to PKLot dataset)

Finally, the SLAM model was implemented as promised. The three dimensional reconstruction of the environment can be performed using the Kimera library and the sensors present on the drone. On the other hand, SLAM's integration to the user interface could not be achieved due to the time limitations of the project. The realsense camera which is used for SLAM also has its own bugs and quirks. For instance, if the camera got obstructed by a close object after that point on the SLAM no longer works as it should. The lighting is very crucial for the SLAM's performance too. The Jetson Nano's GPU is limiting the performance of the SLAM model and is definitely not enough for running the SLAM and object detection simultaneously in high performance.

V. Conclusion

Our project deliverables are met to a great extent. To begin with, we have successfully implemented autonomous flight using our custom flight scripts. We have demonstrated that these scripts can be used to bring out the desired action from the hardware on the drone which works harmoniously with each other. Moreover, user interface and back-end services were developed in a platform independent and responsive way which could be reached through any other device that is connected to the same local network as the Jetson Nano board mounted on the drone. This user interface is successfully integrated with the custom flight scripts to ease the use of our systems. Both the user interface and the custom flight scripts competence in completing its task was demonstrated both in Gazebo software in the loop (SITL) simulations with the IRIS drone model, and in the real world test flights. Custom object detector was also developed for the project. The object detector successfully reads frames of the video stream coming from the camera mounted on the drone and performs inference on those. Moreover, these annotated frames are streamed to the user interface in real time. Therefore, the annotated video stream of the live camera feed coming from the drone can be easily watched from the same user interface that was built for controlling the drones autonomous flight capabilities. Object detector code was built using YOLOv5 in a way to by only changing the weight checkpoint (pt) on the object detector's folder, one can use the same interface built with another custom YOLOv5 object detector that could perform possibly any other arbitrary task. This was intentionally built to support this functionality since the same autonomous flight code and the user interface for both commanding the drone and watching the object detector's live stream can be used to perform different tasks. For instance, by just changing the weights in our code one can use the same setup and drone to perform industrial risk inspection using the camera's feed. Our project currently could switch between the weights of an object detector trained on the MS COCO dataset and PKLot dataset. The one trained on the PKLot dataset was our main object detector goal. It is trained to detect empty and occupied car parking spaces. This classifier does not work very well and could be improved in the future. Due to the limitations of the dataset (PKLot) that we used for training, the object detector could not learn to generalize very well with good Mean average precision values. In the future as an improvement one could gather their own larger and more diverse labeled dataset on the

car parking space occupancy detection and use it to train better models. Furthermore, more experiments and different models could be trained to see performance improvements. We could not experiment further by training longer epochs due to our limited access to GPU resources. For instance, Google Collaboratory usually timed out after training YOLOV5s for 10 epochs. As a better alternative, Google Cloud credits were bought to train on Tesla V100. After some time Tesla V100 GPU's could not be used to train since Google Cloud kept giving error due to that resource not being available in any region that we have tried of. We had to keep on training using Tesla K80 and on our local machines Nvidia GPU's which were more suitable for gaming than they were for training deep learning models. .Also, Jetson Nano had performance limitations which restricted us to the use of small and medium sized YOLO family models and did not permit us to use large YOLO models or EfficientDet architectures. Though we have researched better alternatives to Jetson Nano such as Jetson Xavier and Jetson TX2 we did not proceed to buy them due to lack of time and the fluctuations in the Turkish currency. We figured a way to use our drone to perform simultaneous localization and mapping (SLAM). For furthering this experience, one can find a way to embed this functionality to the user interface using some libraries such as rviz. Moreover, the realsense camera that we used had its flaws such as glitching when an object got too close to the camera. Better alternatives to the realsense could be also seeked. At the time of the development of the SLAM model, Kimera library which we have used did not have any official documentation. In the future upon the arrival of better resources to learn Kimera, our code could be optimized and compressed to further the performance of the SLAM model. Additionally, for the compatibility of our code issue, we used docker for containerizing our drone's object detector. This could be also done for the front-end and the back-end services to improve compatibility of these code repositories using similar customized docker containers. Finally, higher end drones of DJI company supports ROS development. We would suggest generalizing our software by supporting DJI drones since our development has also made use of ROS and its packages.

VI. References

[1] C. Chen, B. Wang, C. X. Lu, N. Trigoni, and A. Markham, “A survey on Deep Learning for localization and mapping: Towards the age of Spatial Machine Intelligence,” arXiv.org, 29-Jun-2020. [Online]. Available: https://arxiv.org/abs/2006.12567. [Accessed: 11-Oct-2021].

[2] A. Rosinol, M. Abate, Y. Chang and L. Carlone, "Kimera: an Open-Source Library for Real-Time Metric-Semantic Localization and Mapping," 2020 IEEE International Conference on Robotics and Automation (ICRA), 2020, pp. 1689-1696, doi: 10.1109/ICRA40945.2020.9196885.

[3] G. Amato, F. Carrara, F. Falchi, C. Gennaro, and C. Vairo, “Car parking occupancy detection using Smart Camera Networks and deep learning,” 2016 IEEE Symposium on Computers and Communication (ISCC), 2016.

[4] V. Visualbuffer, “Visualbuffer/Parkingslot: Automated parking occupancy detection,” GitHub, 11-Aug-2020. [Online]. Available: https://github.com/visualbuffer/parkingslot. [Accessed: 30-Dec-2021].

[5] “Asian Institute of Technology.” [Online]. Available: http://ise.ait.ac.th/wpcontent/uploads/sites/57/2020/12/Car-Parking-Occupancy-Detection-using-YOLOv3.pdf. [Accessed: 22-Nov-2021].

[6] Almeida, P., Oliveira, L. S., Silva Jr, E., Britto Jr, A., Koerich, A., PKLot – A robust dataset for parking lot classification, Expert Systems with Applications, 42(11):4937-4949, 2015.

[7] G. Amato, F. Carrara, F. Falchi, C. Gennaro, C. Meghini, and C. Vairo, “Deep learning for decentralized parking lot occupancy detection,” Expert Systems with Applications, vol. 72, pp. 327–334, 2017.

[8] Ultralytics. (n.d.). *Ultralytics/yolov5: Yolov5 🚀 in PyTorch > ONNX > CoreML > TFLite*. GitHub. Retrieved January 18, 2022, from https://github.com/ultralytics/yolov5

VII. Appendix

- Project's main code repository on GitHub: https://github.com/cangozpinar/Autonomous-Drone-with-SLAM-and-ComputerVision
- Customized Docker image used for object detector code: https://hub.docker.com/r/cgozpinar18/drone_object_detector
- Customized Docker image used for training YOLOv5 and Scaled YOLOv4 models: https://hub.docker.com/r/cgozpinar18/scaled_yolov4

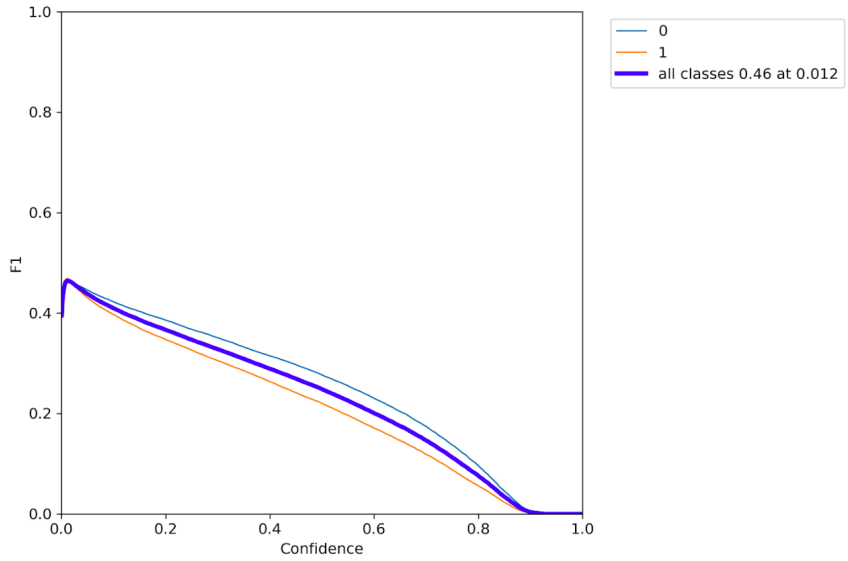

Figure 9: f1 curve for the training experiment(YOLOv5m, batch size: 16, epochs: 200, image size: 640)

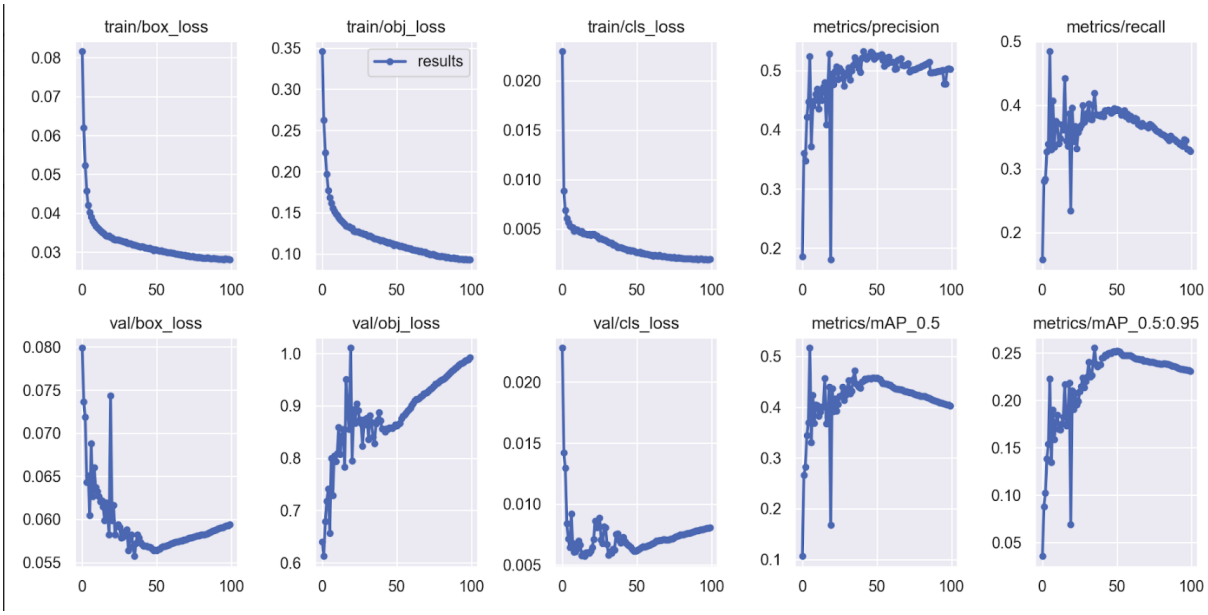

Figure 10: YOLOv5s results (YOLOv5s, batch size: 32, epochs: 200, data augmentation: resize, rotation)

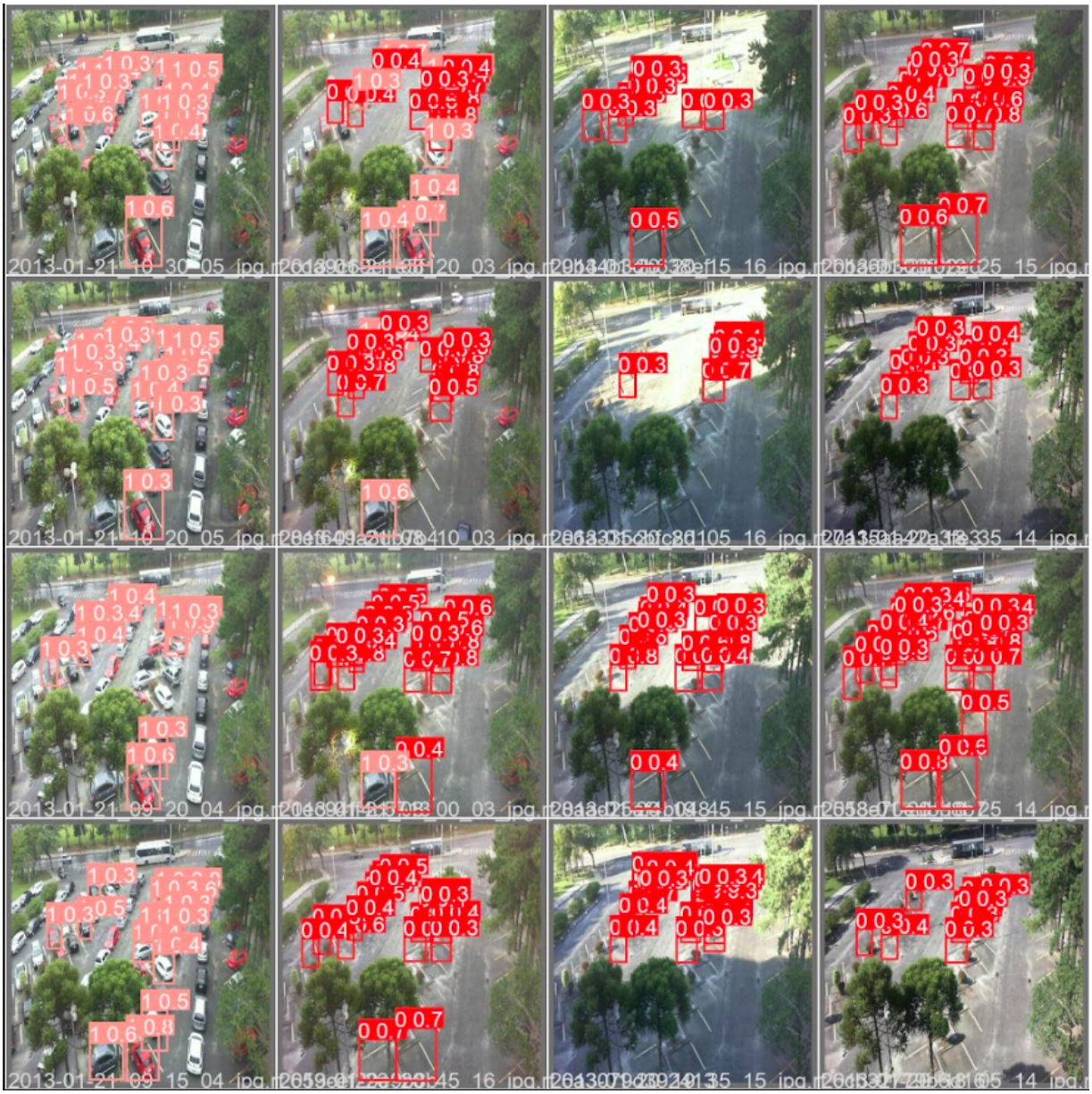

Figure 10: YOLOv5s inference results on unseen images(pink detections: occupied car parking space, red detections: empty car parking space)