Autonomous Search With AI

Final Design Document

Group 28

Spring 2021

Team Members:

Patrick Bauer (Team Lead, Simulation Environment, Navigation)

Pablo Trivino (Object Detection, Robot Sensors)

Mark Pedroso (Object Detection, AI Training)

Noah Avizemer (Simulation Environment, Mapping)

Nathanael Cassagnol (Navigation)

Sponsor:

Lockheed Martin Missiles and Fire Control Applied Research

Scott Nelson, Joseph Rivera

1. Executive Summary

This project started as an attempt to make a swarm of 5 flying drones with artificial intelligence and advanced swarming algorithms map an environment and locate a target. Through the many pitfalls and hurdles of this semester these goals have changed to fit our current capabilities and the capabilities of the software we are using. This project is sponsored by Lockheed Martin and using their industry-level advice and insight it now consists of a singular ground based robot that searches a mapped environment for its target. It looks for the target using the same artificial intelligence.

From a high-level perspective, this project is all simulated from a single code repository. By running a series of commands in an Ubuntu 20.04 machine you can start the simulation, instantiate the turtlebot and its target, begin the pathfinding algorithms, and begin the artificial intelligence target detection. The turtlebot will then proceed to search the environment until it either finds its target and homes in on it (success) or it completes its search loop and ends the simulation (failure). This turtlebot is also completely autonomous as once the project is started it will run to completion without any human input.

Several different algorithms and methodologies have been incorporated into this final project. The turtlebot is doing several things simultaneously while it completes its goal. It begins by taking in the search bounds of its environment and turning those into a series of evenly spaced waypoints to navigate to. Then it takes in a basic map of the environment and runs an A* algorithm to find the quickest path to the first of these. The turtlebot uses its lidar and ultrasonic sensors while it is navigating to perform obstacle avoidance and to run SLAM. Mapping the environment for new information to update the path while localizing the robot against the map to stay on track.

Then, even as it is navigating to its waypoints, performing obstacle avoidance, and localizing itself it also constantly looks for the target object using a camera feed run through the YOLO V3 Neural Network. If this network IDs a target with a high enough probability the turtlebot will pause its waypoint navigation and jump into another algorithm to home in on the target. If the ID probability for the target remains high and the turtlebot locates the target close to it with it’s lidar sensors the simulation is suspended as a success.

2. Identification of the project and its significance, motivation

Personal Motivations

Patrick Bauer

I was motivated to work on this project because the most enjoyable projects to work on for me are the ones that are on the edge of innovation. I believe that the combination of both robots and AI will affect the future of humanity in a positive way. The technology that we are working on can evolve to help defend our nation from outside threats, help provide healthcare to patients from a distance in a timely manner and provide access to places humans can not reach in an effort to aid in search and rescue.

With respect to drones, I used drones to fly them for my personal business to take pictures of properties for sale. The technology used to keep them stable in the air, send a live video feed to a remote screen, and access areas that one normally could not, fascinated me and I believe that drones can be used for a wide variety of purposes.

Artificial Intelligence can be seen in two different lights. One viewpoint is that artificial intelligence is disrupting the economy by taking peoples' jobs. The other viewpoint is that artificial intelligence is opening up new opportunities that otherwise would never have been thought of. I believe that in the right hands, AI can greatly improve everyone's quality of life.

From the list of projects available to work on, this one sparked my interest the most. Although most other projects have unique ideas, the solution for them is either a mobile application or a website (web app). I have experience creating several of each but no experience creating robot simulations using artificial intelligence and want to broaden my skill set. The software development skills, project management skills, and teamworking skills developed from working on this project will help expand my computer science knowledge. I believe that working on this project will allow me to be a better software developer throughout the rest of my career.

Lastly, the most attention grabbing aspect of this project was that it was sponsored by a major government contractor, Lockheed Martin. Other projects were sponsored by individuals, other students, or small businesses. Being able to work alongside an industry leader in simulations I think is an amazing opportunity.

Noah Avizemer

Out of all of the subfields of computer science, artificial intelligence seems to be one of the most talked-about and an interesting field to people. It also happens to be something that I have not had any experience with. One of the reasons that I chose this topic is to learn about artificial intelligence, and hopefully gather experience and knowledge about this topic and be able to apply it towards other projects. Artificial intelligence is an area of computer science that is becoming more and more important. In the present, knowledge in this area is valuable and sought after; in the future it will be even more so.

A big part of this project includes researching the different algorithms and technologies that will be used to make it work. I think that this is appealing because I feel like I am making something new, whereas, many of the other projects seem to be standard applications that are not that interesting. Choosing a project that I am personally interested in was important to me because I know that I will work harder on it. Furthermore, the fact that this project is sponsored by Lockheed Martin was a huge plus for me. I think one of the most important things for me to experience before graduation is to gain experience working with a company in the industry. This project gives me the ability to receive requirements from the sponsor and work to deliver this large project with my group over a long period of time; this is important since I have no industry experience as of now, and it is extremely beneficial to have some kind of experience working with people in the technology industry before going to work there.

Also, the fact that this project deals with robot simulation was interesting to me since I have not worked with anything like this before and I think that autonomous robots are an important technology and hold a large significance now and even more so in the future. This project is the beginning of something much larger; the simulation that we develop in this project may eventually be used as a basis to implement this technology

in the real world. I think the thought of this being implemented in real life and actually having an impact on the world is a big motivation for me to work on this project. Not only does this project have a military application, but the technology may be repurposed for things such as search and rescue operations. Overall I think that this project will allow me to gain valuable skills and experience in research and software development. These skills and experience that I will gain from working on this project will be beneficial to my knowledge of computer science, the technology industry, and my eventual career search after I have graduated.

Nathanael Cassagnol

My personal interest in this project arises from several factors. For the past 6 years I have been expanding my education in computer science and working towards this point in my life. This project was my first choice out of the presented options for Senior Design because I feel it best encapsulates and expands on my knowledge and interests. I hope to use this project to expand on my algorithm understanding, my practical skills, and even my professional development.

My research and interests have recently been focused on simulated swarming systems; this is what immediately caught my eye about this project. As soon as I heard of the requirements for the project, I had algorithms in mind that I was already researching and searching for applications for. This project gives me an opportunity to not just work on these but to also learn of new algorithms for practical applications in this field.

Another important aspect of this project to me is its physicality. The majority of the projects I do nowadays are entirely virtual and programming problems or applications, such as a video game environment, a web application, or even a backend algorithm. This project appeals to me because, while the end result remains a simulated swarm of robots, I still need to consider a variety of real-world complications and stipulations that I normally disregard. This project was the most physical one in my opinion, and that gives me a huge boost in the skills I need and the considerations I need to make to develop applications for real world hardware.

The fact that the project is so carefully simulated also is the reason that Professor Heinrich and our sponsors at Lockheed Martin have mentioned that there is the

possibility of it being used next year as the base of a physical version of the robots for the Computer Engineering program's Senior Design class. I personally take great pride and motivation in the fact that our project is an exciting blend of Artificial Intelligence, Robotics, and Computer Algorithms.

Finally, my last motivation for choosing this project was the fact it was one of the projects with a sponsor. Not only was the project sponsored but it was also sponsored by an important company in the field, Lockheed Martin. Through our contact with Lockheed Martin, I hope to get a glimpse of how the work gets done in a corporate project scenario like this, and to gain insight and advice of the in-depth algorithms used in actual production-level projects.

Mark Pedroso

Most classes in basic undergrad computer science just focus on obviously teaching the fundamentals of computer science and theory. Most of the knowledge I have acquired can relate to working backend tasks, and most of the knowledge that I have acquired from personal projects can relate to front-end tasks. The one area of computer science that I feel like I have never touched until this semester has been AI.

I have taken data analytics courses under my fintech minor and have taken high-level math courses under my mathematics minor, so I feel like a natural next step is to try and learn AI/machine learning. When reading up on AI it seems to combine both of my passions of programming and math so naturally, I wanted to pick an AI project when I got to senior design.

One of the best ways to learn a topic is to start a project that pertains to it. I did this to learn the front end by creating a personal website and it was way more insightful than just reading books and watching YouTube courses. I also will be joining Qualtrics, potentially under a team that focuses on data analytics and machine learning. Even though we will probably not use object detection algorithms ever, I will still learn a lot about fundamental AI techniques and best practices doing this project.

Not only will this project give me a greater understanding of AI, but it will also impact the future of how the military uses drones for search and destroy purposes. Also, aside

from acquiring knowledge about a subject field I have little to no experience in, I just found the idea of this project fascinating.

Starting computer science as a freshman I always knew somewhat of what I was going to be doing in the future in a general sense. Either working on websites (front end or backend), or making some type of app or script for a company. I never really considered or thought of AI because the topic was a little unfamiliar to me and not in my standard definition of computer science.As I progressed through my years of universityI learned more about different avenues I could take with my degree such as data analytics or AI. It seemed like the new domain or up and coming field that companies were putting a lot of resources in because of the powerful/insightful results that AI could lead to. I knew even if I did not do this for my job that it was definitely a good skill to have under my belt because of the high demand.

Pablo Trivino

I am personally interested in this project because of the opportunities it provides both academically and professionally. This project is sponsored by Lockheed Martin, an internationally recognized company that has experience implementing and designing useful and efficient software solutions to their problems. Therefore, our sponsor will advise us throughout this project, giving us an excellent opportunity to gain insight on the development lifecycle of an industry-level project.

The goal of this project is to create a robot that can find and eliminate a predetermined target in a simulated suburban environment. To achieve this, we must generate a simulation of a suburban environment, create and train an object-detection model, and implement pathfinding and searching algorithms. I am particularly interested in the machine learning aspect of this project, as it is a field in which I would like to pursue a career. This project is an opportunity to familiarize myself with various machine-learning technologies, such as Keras and TensorFlow, as well as a chance to investigate different object-detection algorithms, such as YOLO, RetinaNet, and Faster RCNN. It is also a chance to gain experience gathering data, and creating and fine-tuning models. At the same time, we will also be using Gazebo and ROS, which

are both widely used, industry-grade software development tools, to test the effectiveness of our object-detection models and pathfinding algorithms.

Over the course of this project, I hope to learn about how to efficiently use TensorFlow and Keras to implement an object-detection algorithm, how to preprocess and sanitize input images before they are analyzed by a convolutional neural network, and how to change the parameters of the model to improve its overall accuracy. As mentioned before, we will also be using ROS and Gazebo. I hope to learn to manage robots using ROS, a meta-operating system, and to use Gazebo, a program that allows a user to create a simulated indoor or outdoor environment. ROS and Gazebo are commonly used in conjunction with each other to test robots, which is what we will be doing to test our robots.

Professionally, this project also offers a chance to gain insight into the development process of an actual industry project. Unlike other academic coursework, this project aims to solve a real-world problem, as evidenced by the Lockheed Martin sponsorship. This is also an experience to see the development process of a project, including the research, implementations, development, and presentation phases.

Broader Impacts

The elephant in the room of course is that our project could be easily adapted to develop a weapons system. Now that the turtlebot has been developed to search a dynamic area and find a target it is trivial to change the weights of the target detection AI and have it search for any target. It is therefore extremely important for us to consider beforehand the negative implications our technology could have. However, it is also important to consider that now that we have made this lightweight search system this technology can also be reused for beneficial purposes.

It is possible for the outcome of the project to be reused as a search and rescue robot in impoverished areas due to its low cost and flexible design. The reason for this is because the result is a turtlebot that transverses a dynamic area and locates a target using AI. Since this can be done without putting humans at risk and the AI can be retrained to look for different classes this will prove to be a useful tool when disaster

strikes in locations where it is inadvisable or impossible to stage a large scale search and rescue endeavor.

Legal, Ethical, and Privacy Issues

This project does not have any privacy liabilities since no user data is stored for any period of time other than information on what the target in question is. Any information on the target that may be stored will just be a photo of the target robot and does not contain any protected human data. Therefore, this project does not need to take into consideration any laws or regulations that protect user data such as HIPAA or COPPA.

From a legal standpoint, it is worth considering ITAR or International Traffic in Arms Regulations. ITAR restricts the export of defense and military technologies, which this project will be simulating as weaponized robots. However, in the case of this project which will not be simulating firearms on the robots ITAR does not come into effect. That is because ITAR only applies to items on the United States Munitions List.

Even if this project came under the scope of ITAR members of this group will only be sharing the technology between ourselves, Lockheed Martin, and Professor Heinrich, all of which fall under the definition of U.S. Persons. This means that ITAR regulations allow us to view and share the code, what is not allowed is sharing of the project to entities that are not U.S. Persons. Heavy fines and other consequences will arise if the information and materials of the project are shared outside the U.S. If either Lockheed Martin or any of the groupmates do want to share the project, it will be under export control legislation and they must get authorization from the Department of the State.

Certain ethical issues arise in this project that will have to be considered by all involved as we move forward. These issues mainly revolve around the fact that the final goal of this project is to develop a simulated arms and weapons system. Furthermore, we are attempting to develop a fully autonomous, robot-based arms system which requires particular ethical caution and responsibility. This being said, our ethical responsibilities as designers making weapons are understood by the team and mitigated by two facts. Firstly, by the fact that first we will be working with simulated hardware and it will be the responsibility of Lockheed Martin or the next group to work on this to bring it into

the real world. And secondly, in our computer vision algorithms we will be specifically training the target-finding algorithm to look for a certain simulated robotic target and not for human targets.

From a legal standpoint, there are federal laws that need to be adhered to when operating robots in public areas. Since our project is concerned with the simulation of these robots instead of actually using them in the real world, we kept these laws in mind when it came to design. We were not required to obtain the appropriate licenses to use the robots because we did not need to. If future teams expand on this project by implementing it in the real world, it will be up to them to follow all the related laws and regulations.

When it comes to privacy issues, as developers of this simulation, we are not actually navigating robots through peoples' property, we are only simulating them. We do recognize that if this project is implemented in the real world, the robots may violate privacy laws when they search residential or commercial areas. It is up to the teams that implement these robots in the real world to obtain the appropriate permissions to avoid any repercussions.

3. Technical objectives, goals, specifications, and requirements

The Autonomous Search With AI project has several requirements that must be met to complete its goal of searching and disabling a predetermined target in a simulated urban area. The requirements are separated into three categories: robots, simulation, and AI. Each requirement is listed below, along with relevant specifications.

Simulation Environment Specifications

The simulation environment and algorithms shall all be functional within Gazebo and utilize the Robot Operating System (ROS) robotic middleware.

The environment shall emulate a realistic suburban neighborhood or similar. There should be at least two streets with four rows containing buildings and houses. Several other obstacles can be added to ensure robustness of the pathfinding algorithms and image recognition AI. The environment should be around 400ft by 400ft.

The simulation shall contain a target to be searched for by the robot. The target should be a Gazebo ROS bb8 robot randomly hidden within the environment to test the robustness of the pathfinding algorithms and image recognition AI.

These requirements must be met to ensure that the robot can locate and disable the target in an urban environment. Gazebo will be the software used for the simulation.

- Create a simulated environment using Gazebo
    - Modify the environment to simulate a realistic urban area.
    - Add static obstacles such as buildings, houses, and trees.
- Create a robot in the simulated environment
    - Implement and apply the pertinent code we develop to the robot
- Create the target in the simulated environment
    - Have the target spawn at a random location within the urban area
- Simulate constraints on resources such as the following:

- Time constraints
- Fuel constraints
- Range constraints

Robot Specifications

The robot shall be designed in such a way so that individual components can be interchanged with functionality remaining the same. Example: if technology improves, a lower resolution camera can be swapped out with a higher resolution camera.

The robot shall be fully autonomous (no human input required) while completing the mission. The robot shall also respond to human input and override all autonomous algorithms when human input is provided.

The robot shall publish their sensor status, position, orientation, and whether it found the target to a ROS topic. The robot shall subscribe to instructions from a ROS topic.

The robot's processor shall receive information from onboard sensors and make a decision on what direction to turn or move based on an algorithm.

The robot's processor shall search for a specified target using a trained robot vision AI. The onboard camera will supply the robot with images or a live video feed.

The robot shall be GPS denied. They should track individual positions based on a local algorithm such as SLAM.

These requirements must be met to ensure that the robot can effectively and autonomously search the environment and provide information to the user. ROS will be used as the framework for our robot's software, as it is open-sourced, reliable, and used in the industry.

- Have the robot be fully autonomous, meaning it can carry out all its functions without any human input.
- Implement a pathfinding algorithm that the robot will use to efficiently traverse and navigate the environment.

- Locate and disable a specified target by efficiently searching the environment, even under constrained resources.
- Provide collision-detection for the robot, preventing it from crashing into obstacles in the environment. This includes dynamic obstacles such as people and cars.
- Utilize the meta-operating system ROS for:
    - Hardware abstraction
    - Implementing functionality
    - Conducting unit tests
- Implement a method of communication for the robot, allowing it to send information between topics and python code.
- Utilize LIDAR to:
    - Detect objects at range of 1-500 meters
    - Detect movement
    - Sensing the dimensions of objects in the environment

Image Recognition Specifications

The artificial intelligence onboard the robot's processor shall implement a trained image recognition or video recognition model to search for a specified target. The AI should be able to identify the target relatively quickly.

The AI will be able to identify the objects it detects as either the target or not the target. We will be using open-source tools such as NumPy, TensorFlow, and Python to create and train the AI.

- Able to identify the target when it is detected by the robot.
- Able to label other objects the robot detects as not the target.
- Able to quickly perform its calculations within a few seconds. The robot will have to label several objects so it is important that the AI can quickly label them as the target or not the target.

The robot will have to be able to detect a target while scanning through an urban environment. This would rely on some object detection algorithm that can quickly process the surrounding area and be able to determine if the target is visible. There are

many algorithms out there for image detection but when it comes to detecting an image from a video, that is when some algorithms fall short. Since the robot will be constantly moving and processing the environment, we need an algorithm that can quickly process the video stream with decent accuracy. Object detection typically uses deep learning techniques that use convolutional neural networks. Neural networks can have many different types of architectures and those architectures are defined by the type of layers that are implemented and how they are connected to each other.

One important requirement is that the robot must be able to efficiently search an urban environment for the target. To evaluate the effectiveness of our algorithms, it is beneficial to create either at least three different maps or a randomly generated world in our simulation, each with varying sizes, building placements, and clutter. This would allow us to test several features of the project. By randomly generating clutter and dynamic obstacles, we would test the object-collision detection algorithms of our robot, and by placing the robot and target in random locations we would test the pathfinding and searching algorithms. This way, we can avoid overfitting our AI model, ensuring it performs efficiently and accurately in different environments.

Another requirement is that our robot must be able to identify a target, which will be accomplished with AI. This will be a classification and computer vision problem where the AI must label objects it detects as either the target or not the target. A supervised learning approach is necessary, so we will develop an efficient way to gather the data necessary for training.

There are several computer vision models that meet these requirements, these will be discussed in a later section. These computer vision models all aim to be able to identify to what class a detected object belongs. We also investigated if we wanted to implement our own version of these models or select a premade model and adjust it to our needs.

User Interface Specifications

The simulation shall have an interface which allows for a user to start, terminate, and monitor the ongoing mission. Information about the status of the robot should be displayed for the user.

Target Specifications

In the simulation, our world is set up to include an urban environment with two roads and multiple structures and buildings located throughout it. The robot recommended to us by our sponsors at Lockheed Martin was the Burger TurtleBot3, but we decided on using a bb8 model as the target. The robot will then search through this environment to find and identify the target.

Figure: bb8

Initial Ideas and Member Assignment

Patrick Bauer

From a high-level perspective, there should be a main command and control console that receives information from the robot at regular intervals (the robot's position, orientation, direction headed, the status of sensors, and whether the robot has found the specified target), processes the information and decides what the robot should do, and sends instruction information back to the robot (What area to search next, whether the mission was successful or not).

Nathanael Cassagnol

My initial idea when hearing the project proposal was the BOIDs algorithm. Named after "bird-like systems," this algorithm was developed by Craig Reynolds in 1986. The use of this algorithm is that it works with dynamic situations well and will not be disturbed by the loss or miscommunication of a single robot. This algorithm is used to generate flocking behavior from a series of individual agents, in this case the robots. One downside of this algorithm is it works better for large amounts of agents when we want a lightweight flock of only 4 to 5 agents.

For this reason my main focus throughout the project would be researching and implementing the pathfinding methods for the robot swarm.

Following my interest into robot cooperation and the BOID algorithm it was decided that my focus on the project would be into robot pathfinding and collision avoidance. This area became crucial once the sponsor decided on working in a GPS denied simulation. Without a stable GPS position from each robot, suddenly coordinating them and mapping the environment becomes significantly more difficult.

Noah Avizemer

Our group has not yet established how certain tasks of the robot swarm will be completed by each robot in the swarm. We could just have all the robots be able to complete all of the needed tasks like pathfinding, destroying, and relaying information to the server, but I believe that it may be more efficient to split up these tasks. An idea to consider is that each robot in the swarm is assigned a specific role and completes tasks that are relevant to specific tasks assigned by its role. For example, we can have certain robots in the swarm whose only job is finding the target; all other robots would just follow these robots. There can be a command robot with the ability to relay information to the rest of the robots, communicate with the server, and keep track of information sent to it by the other robots. There can also be robots in the swarm whose sole task is to destroy the target; these robots can be equipped with weapons to accomplish this task.

We do not necessarily have to go all or nothing with assigning roles to robots. There can be some robots with multiple roles and some with a single role. We need to be able to figure out which allocation of roles would be the most efficient in completing the task. Perhaps this can change based on the environment, the number of targets, or any other variables that would go into the simulation.

Mark Pedroso

So far, we have established that the robots will search an area to find a preset target. But what if the target moves and gets out of the vision of the robots once the target has been found? We should implement a way for the robots to continue searching the area of the last place the target was located if the target manages to get out of view. Another idea would be a retreat feature. If we suddenly want the robots to return back to the initial point of release for any reason we can simply call them back and they will stop their searching.

Pablo Trivinio

One important requirement is that the robots must be able to efficiently search a suburban environment for the target. To evaluate the effectiveness of our algorithms, it might be beneficial to create either at least three different maps or a randomly generated world in our simulation, each with varying sizes, building placements, and clutter. This would test the robots' ability to successfully navigate different environments and its ability to find the target. We would also be ensuring the robots can manage different obstacle placements and degrees of clutter. However, this would be implemented much later in the project, if at all, as it is not necessary for the robot to work.

Another requirement that our robots must meet is to be able to identify a target, which will be accomplished with machine learning. This will be a classification and computer vision problem where the AI must label objects it detects as either the target or not the target. A supervised learning approach will probably be necessary, so we will likely need to develop an efficient way to gather the data necessary for training. Also, because we will be dealing with computer vision models, labeling the data with ground-truth boxes will be an important part of the data pre-processing tasks. We will have to find precise software to accurately label the targets within each of the input images. Ideally, it would be nice if this process could be automated, as there will be thousands of input images. However, because the bounding boxes must be precise, it might be necessary to manually hand-label and annotate every image individually.

4. Research and Investigations

Physical Systems Research

Disabling Target

Once the target is located by the robot, the problem arises of disabling the target. Several factors apply here, in simulation and in final applications. Factors such as collateral damage, effectiveness, cost, and practicality.

Starting with the naïve approach to target elimination the simplest solution would be to equip the robot with conventional weaponry. While attaching a small caliber gun to the

robot would be simple, we largely ruled against this approach. This approach for disabling targets is not preferred because in terms of effectiveness, the accuracy and potency of firearms are both low against a moving target. This approach is also largely irresponsible, our team did not wish to be responsible for creating mobile weapons stations that were completely autonomous. Also, the ricochet and missed shots present a danger to bystanders, the robot itself, and to the surrounding area. These reasons are also why physical interception is also not well advised.

Another method of disabling targets that has proven successful is nets [16]. In Tokyo the police department is currently utilizing drones with large nets under them to intercept terrorist and malicious drones. The nets, hoisted by large police drones, work by tangling the target robots' rotors, allowing them to be carried back to the officer for forensics. In our application this method would have several benefits. These nets avoid all the complications of the weapons method, they would not present any danger to the surrounding area or bystanders. Unfortunately, we cannot utilize this method because the target we will be testing with is ground based, and therefore largely immune to this method of interception.

To reiterate, we have been looking for a way to disable a ground-based robot that minimizes damage to the surrounding area or civilians and bystanders. This leaves us relying on some energy weapon configuration. We then focused on two different methods, radio frequencies and microwave bursts.

Radio frequency jamming is a currently implemented method of robot disabling. By sending a powerful wave of radio frequencies you can disrupt the target robot's communications, instructions, and global navigation satellite system information stream. In commercial robots this will lead to an immediate cease in activities followed by a shutdown. Meanwhile the operator stops receiving data such as the video feed from the target robot and is unable to send the robot new instructions. While this method is effective against commercial robots, when you take into account that in the scenario we are working in has our own robot completely autonomous and non-reliant on global navigation satellite system information, we can assume the target is also autonomous as well, and therefore immune to radio frequency bursts. These bursts and any kind of radio jamming technology are also heavily monitored and regulated by the FCC which would make physical testing of the robot prohibitive.

This leaves a higher energy beam as the only option for our use case. Microwave disabling has been tested in Japan [17] already and this method was recommended to us by our sponsor as well. Microwave bursts are more effective than radio frequency bursts for our purposes because of their higher energies. Along with affecting communications they also damage the target's internal systems and most importantly their delicate sensors. Without their sensors functional the target robot is effectively neutralized as threats.

Pathfinding Research

When researching pathfinding for this project we needed to create a robust pathfinding system that worked with no external information in an unfamiliar environment. This led us to several problems. Nathanael initially wanted to pursue a Bird Like Algorithm approach to the problem but unfortunately these stipulations from our sponsor made this untenable, requiring more complex algorithms to be needed.

To solve this issue, we split the problem into three parts, firstly basic collision avoidance on the robot, secondly the mapping of the environment from the robot, and thirdly the actual local position of the robot during runtime. The last two problems require significant hardware and computations to solve, but once solved enable the robot to be placed in the environment with only their initial orientation known and subsequently map and track their way through a dynamic environment utilizing internal sensors.

Collision Avoidance

Collision avoidance is the simplest aspect of this pathfinding approach. In simple terms what we want for collision avoidance is if the robot is given a heading in a dynamic environment and an obstacle is placed or arrives in this heading it is the responsibility of the robot to change heading to escape this hazard.

The robot will accomplish this by simply iteratively checking through sensors whether the path is obscured. If the path is obscured the robot will change path. This process

repeats until the robot reaches its destination. The sensor used in this project will be lidar, which is a method for measuring distances by illuminating the target with laser light and measuring the reflection with a sensor. Inside of ROS the data captured from the sensor is returned as a point cloud with a range of thirty meters. The lidar system was chosen over alternatives due to recommendations by our sponsor at Lockheed Martin.

Another potential sensor for this application is ultrasonic sensors. Ultrasonic sensors work similarly to lidar but with sound waves instead of light waves. One upside to using ultrasonic sensors is they are much less expensive than lidar. Although that does not concern us too much in this context of only simulating equipment, lidar can cost more than 10,000$ [7]. However, the downside to ultrasonic sensors is they have a much shorter range than lidar in practice.

Local Relative Position

The second issue to handle in navigation of the robot is a way to calculate its position. Because the environment is GPS denied without using internal sensors the robot would have no way of determining where their search area is for the target. To solve this problem, we will give the robot a set starting configuration, in relation to the search area, and the robot will then be using internal sensors to update upon that initial local position. All this information will be shared with the main python code so the configuration of the robot relative to the map is always known.

Upon finishing our research two methods of developing this internal position definition were identified: Visual Odometry and Inertial Measurement. Comparing and contrasting what our research has identified about these two methods will allow our team to decide upon the preferred one for this project. However, there were practical issues to consider such as runtime or accuracy, so we kept both in mind when it came time for initial testing of the system.

Visual Odometry is the process of determining the local position and orientation of the robot by analyzing iterative camera images. That being said, in practice the robot's hardware would be a forward facing camera. Then machine learning algorithms would analyze the footage from the camera every timestep and generate an updated

predicted local position. These algorithms are already available for use and would not have to be trained by our team.

The benefits of using this algorithm is that we can use the forward facing camera for multiple purposes, the photos taken by it can be searched for the target while also deciding the robot's position and even mapping the environment using optical-flow. This method also has precision advantages over other methods because it calculates based on the exterior state of the environment and not the lossy or inaccurate internal states of the robot.

The second method to consider is the more conventional Inertial Measurement. For Inertial Measurement, the sensors we would need are an accelerometer, gyroscope, and magnetic field detector. Working in unison these three sensors would detect the robot's headings, orientation, speed, and acceleration. By adding this information every timestep to the last position of the robot, where the initial position is known, it is possible to perform dead reckoning navigation, and know where the robot is locally.

Both of these approaches have the same issue in practice, which I will describe as drift. As the robot continues to travel, errors in its predicted location keep adding up, throwing off the path in regard to the actual location or offset.

This issue can be fixed in several ways by tweaking the algorithms used and the team will have to see how significant the problem is in practice and how to approach it once the robot is being tested.

One amenity to the problem currently under consideration is if the robot knew the distance between itself and local objects, it would be possible for the main server to triangulate the robot based on the distances and the predicted position, correcting drift as it happens. This distance data would be useful for the problem the local position map seeks to solve.

Finding a sensor that gives distance in ROS has been a challenge, but it is theoretically possible using RFID sensors and transmitters on the robot. The RFID sensor receiving the signal can then determine the distance between the robot and obstacles by measuring the strength of the signal. This data can then be sent to the main server to be averaged and applied.

Mapping the Environment

The final issue run across by the pathfinding suite is that of mapping the searched environment. The use of mapping the environment is to identify potential hazards for the robot and to determine whether the area has been searched already. Mapping an unknown environment while simultaneously locating your position in it is a difficult algorithm to compute. Luckily, we are not the first to encounter this problem. Through advice from our sponsor at Lockheed Martin and Professor Heinrich we have been pointed in the direction of the SLAM class of algorithms.

SLAM

Simultaneous localization and mapping, or SLAM is a series of algorithms that maps an unknown area while finding the agent's location in it at the same time. This problem is, on paper, seemingly a chicken-or-the-egg problem. How can you map an environment if you do not know where you are, and how can you know where you are in the environment if it has not been mapped? There are solutions however rising from probability theory and the rise of powerful sensors such as lidar.

This problem and corresponding solution are of course exactly what we need for our current application. We want to put a robot in the environment, and have it simultaneously map the area to find the target without having information on its location, and localize itself and its compatriots inside this map without GPS information. The main issue with this algorithm is it is computationally difficult which leads to issues in real time applications.

By generating a map with features (landmarks) and computing the robot’s path through it SLAM algorithms can calculate the approximate location and map of the area. The map is then updated by taking a combination of the range to the landmarks and inputs from the robot such as its velocity and heading. This is all probability calculations which are increasingly difficult to perform in real time.

There is an entire family of SLAM algorithms. Some which we have considered include Visual-Slam, which performs the SLAM functions taking input from visual cameras. Collaborative Slam, where several agents work in unison to map a shared environment.

And Fast-Slam, which is a variety of SLAM that focuses on defining a set number of landmarks rather than a point swarm or more high-definition model.

Fast-Slam

Fast-Slam is a newer derivative of SLAM, as stated before it runs significantly better because it lowers the amount of information about the exterior environment it needs to store. Fast-Slam also speeds up calculations by using a “predefined” path through the map to perform calculations. Once the path (location) is set the calculations for mapping the environment become more manageable. The runtime can be further optimized using computation trees. Because Fast-Slam only cares about large landmarks instead of defining a point cloud it is also more robust to noise in the calculations [10]. It will be important to see if a limited number of landmarks is a feasible approach in the urban environment the robot will be searching.

Through our current research I recommend the Hector slam coding library for initial explorations into the feasibility of SLAM. I recommend this because it is well defined and documented in ROS, with tutorials showing how to simply enable it with simulated lidar sensors. This code library is also currently maintained and updated, and it can be used under a permissive free software license. Finally, what has caught my attention regarding this library is the impressiveness of the posted results and examples.

Navigation Waypoints and the A* Algorithm

While using Fast-Slam to map the entire area is possible it runs into issues of memory complexity and time involvement. A faster method of searching the environment would be to assume that the environment was already mapped for all non-moving objects (buildings, parked vehicles, trees). Once you have a mapped environment you can make sure you quickly and exhaustively have searched it.

So by giving our robot a starting map we perform the following steps. First we generate an (x,y) grid surrounding our robot. The bounds of this grid are given to the robot at runtime. That means that the robot can be given any dynamic rectangular search area at runtime. It also means that this search area can be further divided into multiple areas in order to split the searching among multiple robots.

In order to navigate from the robot's starting point (0,0) to any other point on the map we require an algorithm that takes the costmap and returns the shortest path to the target while avoiding obstacles. The algorithm we ghost for this is A* (pronounced A-Star). The robot also uses its lidar sensors and the existing costmap to continually locallize itself on the map and keep from drifting off its path.

A* was originally created in 1968 in order to provide pathfinding to an autonomous robot, and it is still the quickest and most exhaustive pathfinding algorithm in use today. A* works by expanding a tree of possible paths through the costmap until it finds and returns the one with the shortest cost.

Homing in on the Target

After the Artificial Intelligence locates a target we still need to make our way to the target. Although the robot now has a view of the target, we are still unaware of what obstacles are on the way to the target. We also do not know the exact (x,y) location of the target, so we can not simple run A* to head to it. Our homing algorithm to make the way to the target assumes the AI returns us a bounding box of where on the camera feed the target appeared, and a probability that what is on the screen is actually the target we are looking for.

With this data the robot can carefully and autonomously make its way to the target. Firstly it checks the percentage of it actually being a target, we do not want to head to a false positive. Then it will make its way towards the target using the bounding box. If the bounding box for the target is on the left side of the screen go forwards and left, if it is right go forwards and right. This is continued until the target is in front of the robot. If it turns out to be a false negative the robot simply resumes visiting waypoints.

Artificial Intelligence Target Detection

The goal of the artificial intelligence portion of this project is to allow the robot to understand and identify the objects that it sees with its cameras. Using a computer vision algorithm, our machine learning model will be trained to accurately identify its target, which will be a bb8 unit. Because the robot will use the computer vision algorithm to effectively see the world, it is important to choose and implement an algorithm that generates rapid and accurate classifications.

The architecture for most computer vision algorithms tends to be fairly intricate, as there are several steps in both predicting the possible locations of objects and classifying the objects within those locations. Therefore, a substantial amount of research on the different steps of computer vision is necessary to understand which model would most closely fit our needs. By the end of the project, the model should be tuned and refined to have the best possible degree of accuracy and precision.

Neural Networks

The most common algorithm structure in Artificial currently is Neural Networks. Neural Networks are a popular solution for solving a large variety of problems in artificial intelligence such as categorization and image detection. The concept of a neural network is that of a directed graph with the neurons being the nodes and the weighted edges leading to an answer once data is inputted. Neural networks are trained by having the weights on their edges tweaked slowly to a minimum of loss on their answers.

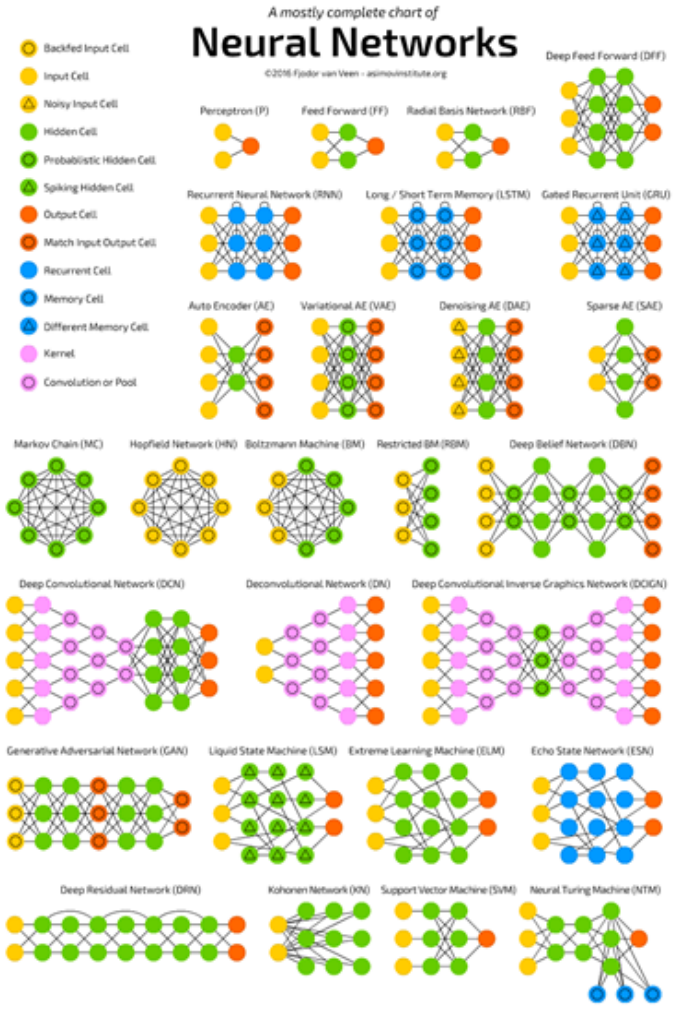


Figure: Common Neural Network Architectures [15]

Before getting into convolutional neural networks, let us discuss the other two common neural networks for deep learning and why we will not be needing them for object detection. The first one is the Artificial Neural Network (ANN).

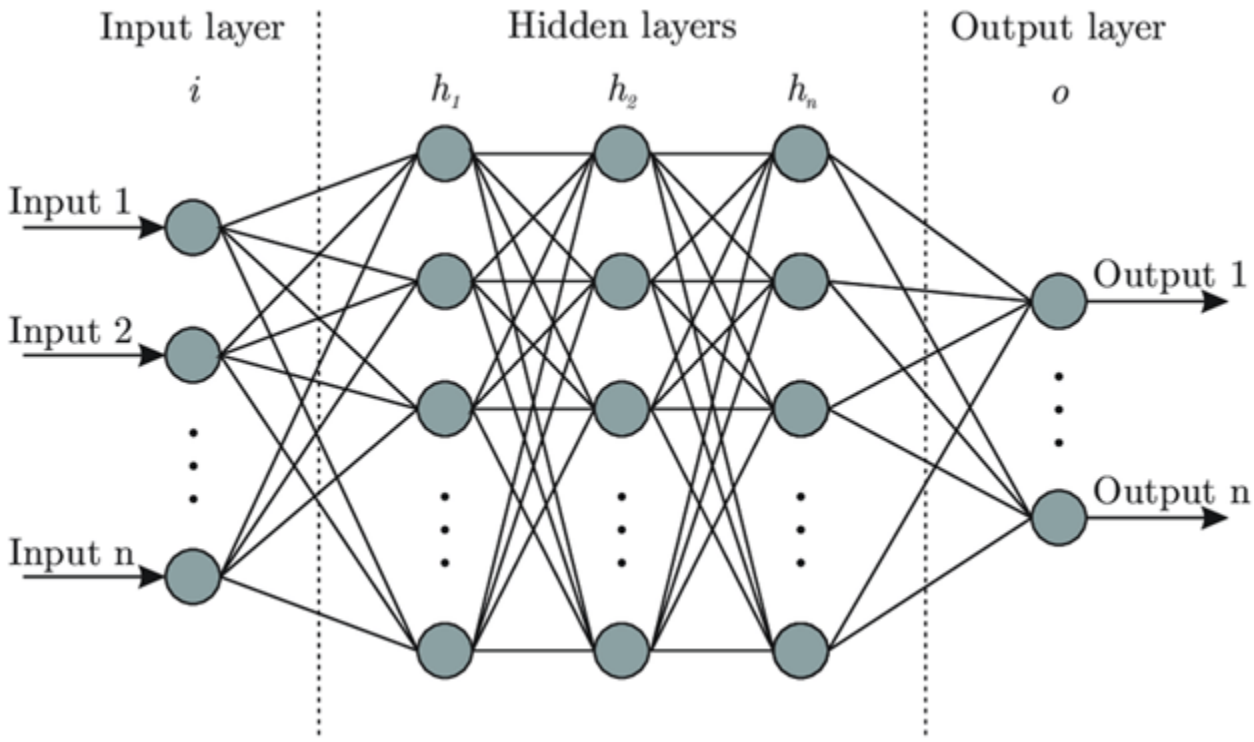


Figure: Components of an ANN [2]

These are typically the ones referred to when one just says neural network. Each node would be a logistic regression and inputs are only processed forward which makes this a feed forward neural network. As one can tell from the picture, there are 3 layers in an

ANN. The first layer receives the inputs then the hidden layer processes the inputs then the outer layer is the results. Now, the reason why an ANN would not be best for us to use for target detection is that when dealing with image classification, an ANN converts a 2d image to a 1d vector before beginning training. This causes two problems:

- The number of trainable parameters increases immensely as the image size gets larger
- ANN loses spatial features of the image aka the way the pixels are arranged in an image

The other neural networks common in deep learning that we will not be using are Recurrent Neural Networks.

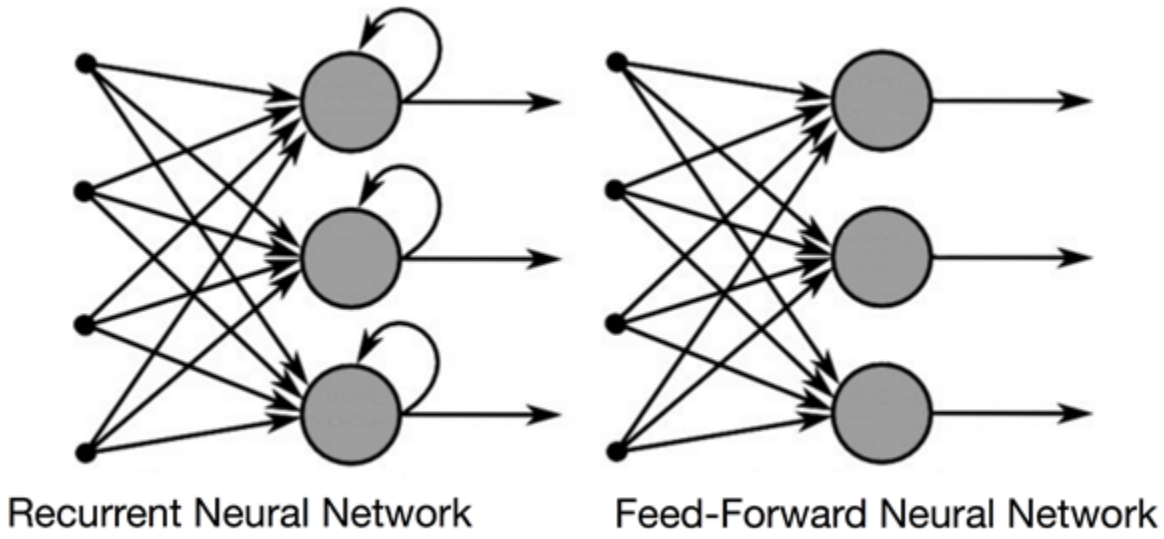


Figure: RNN vs FFNN [23]

RNN

Now an RNN is exactly the same as an ANN but with one difference. There is a recurrent connection to the hidden state. This loop ensures that data presented after the initial input data is also captured which makes this a great neural net to use on time

series, audio data, or computer vision. Now for target detection this method of RNN is combined with a CNN to make a CRNN. This method actually has extreme accuracy because the inputs are initially processed by the CNN and then their outputs are given to the RNN. Unfortunately, CRNNs are typically slow because of the amount of accuracy that they produce. Something like a video camera identifying open parking spots would use this but an actively navigating robot constantly seeking new information in an urban environment would want something quicker than a CRNN most likely. Of course, we will test most of these different methods.

Now this paper mentioned convolutional neural networks (CNN) previously, but now a brief description as to what they are will be given.

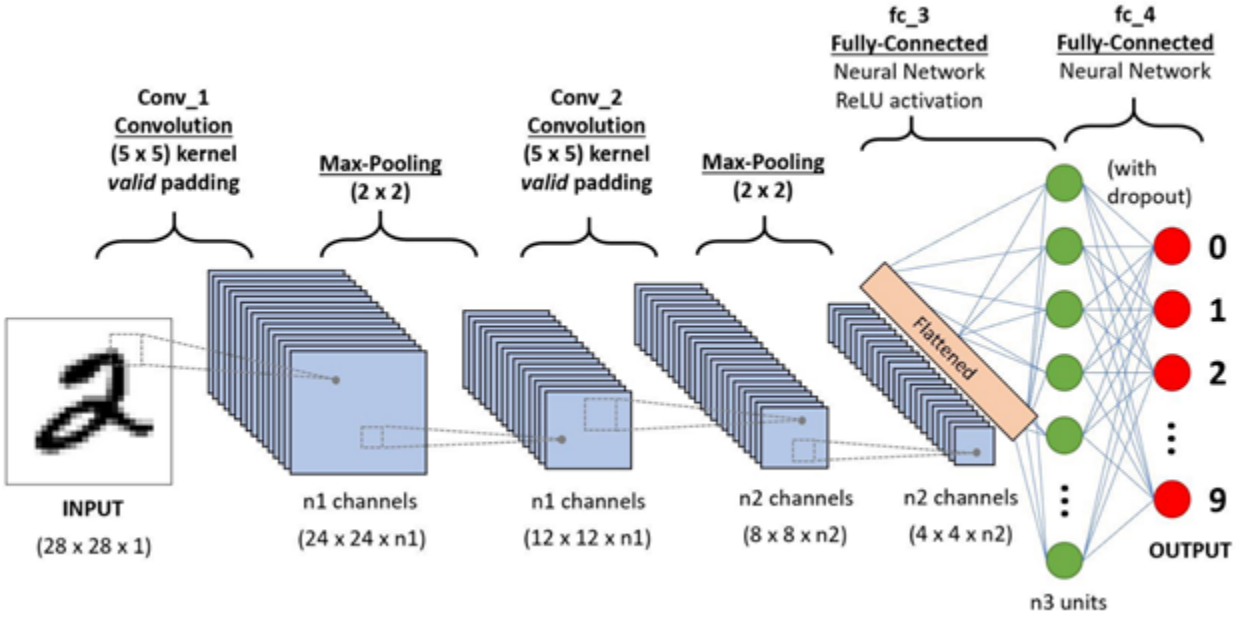
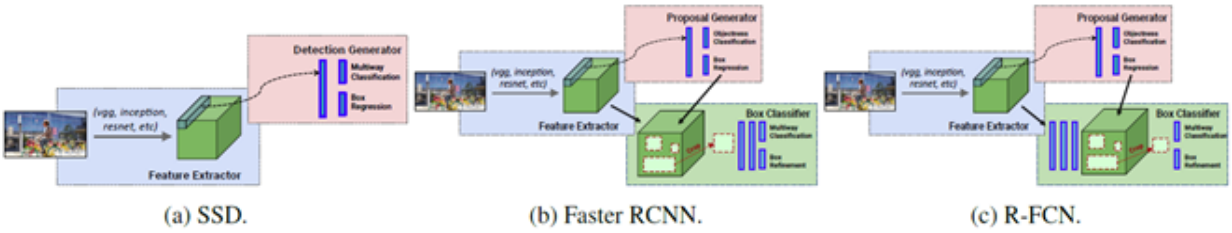


Figure: Components/layers of a CNN [18]

The main difference that makes CNNs so powerful is that they are spatial, which means that instead of neurons being connected to neurons in the previous layer, they are only connected to neurons close to it with all having the same weight. The simplifications of the connections in the network mean that the network upholds the spatial aspect of the dataset. This means that if the CNN detects a car in the top left of the image, it knows that it is in the top left of the image and not everywhere in the image. In the

convolutional layer of the neural net, the building blocks are filters also known as kernels. These filters extract relevant features from the input using convolution operation. CNNs are mostly used with image recognition but when combined with RNNs as mentioned earlier, they can be extremely accurate at detecting objects in videos, but the only downside is how costly the operations are as mentioned before. This brings in the idea if we could run an algorithm that only uses one neural net like a CNN without having to rely on the RNN to feed in the data. This is called single shot detection.

SSD or Single Shot Detection is what comes to mind when speed is more of a factor. Two shot detection is what comes to mind when accuracy is more of a factor. The main difference is that a single shot detection makes a fixed number of predictions on the grid while a two-shot detection makes a proposal. By proposal we mean that first the neural net will detect a region that might have the object we want and then a second neural net will see if the object we are searching for is in this region.



(a) SSD. (b) Faster RCNN. (c) R-FCN.

Figure: SSD vs Faster RCNN vs R-FCN [24]

Let us not get into the specifics of RCNN and R-FCN right now, but we wanted to show this picture to demonstrate the difference of stages between single shot and two shot detection. So, a faster RCNN is faster than a normal RCNN because it avoids duplicate computations, but the speed still falls short because the run time depends on the number of regions proposed by the RPN. Now, the picture below is an example of single shot detection which we will be spending the most time on because we believe it will yield the best results in regard to the robot's object detection.

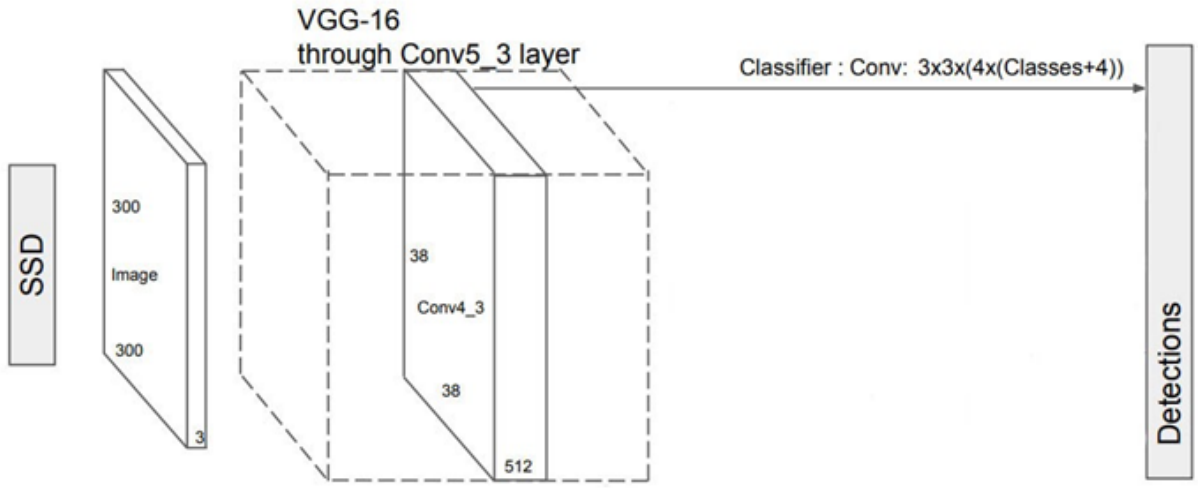


Figure: SSD components [4]

Single shot detection is composed of two parts:

- Extracting feature maps
- Applying convolution filters

So, for single shot detection to extract feature maps, it uses VGG16. VGG16 is a convolutional neural network architecture that won ILSVR/ImageNet competition in 2014. As far as vision architecture goes, it still holds weight as one of the best. Previously with image detection, it was pretty common to have a large number of hyperparameters, but VGG16 uses 3 by 3 convolution layers with a stride 1 and always had the same padding and maxpool layer of 2 by 2 filter if stride 2. VGG16 has two fully connected layers with a SoftMax for output. The 16 in VGG16 refers to the number of layers it has that have weights. This network is massive with roughly 138 million parameters.

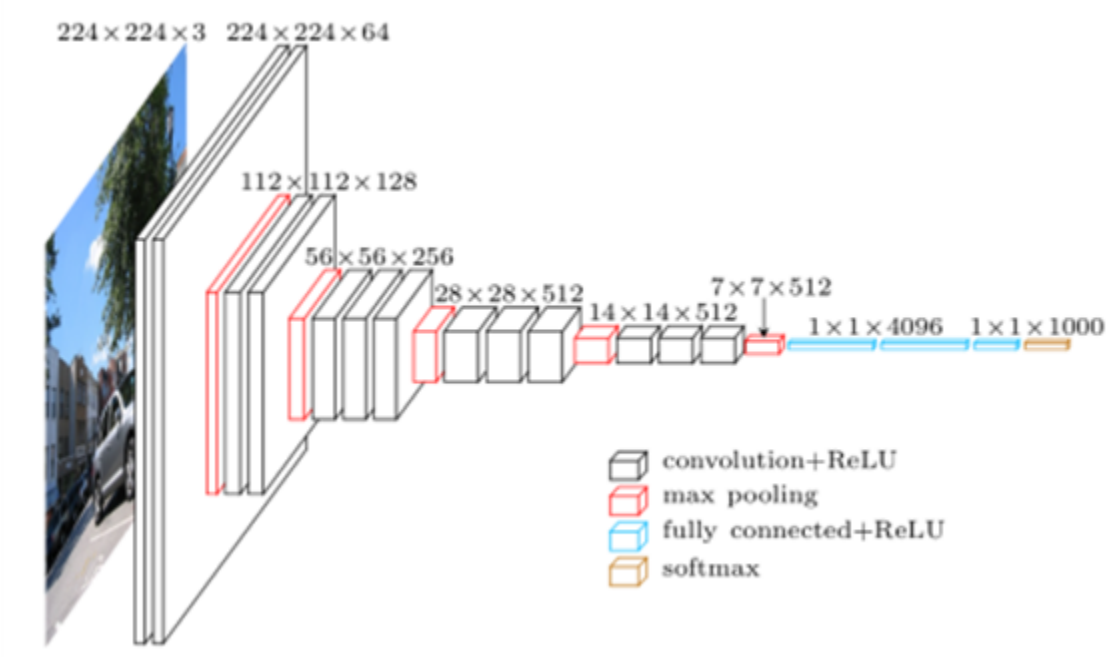


Figure: Dimensions of full SSD [13]

Now that we have established what the role of VGG16 is in single shot detection we can continue. So, after the SSD uses VGG16 to extract feature maps it then detects objects using the Conv4_3 which is a layer from the VGG16 network. The conv4_3 makes 4 predictions per cell regardless of the depth of the feature maps and each prediction is composed of a boundary box.

As stated before, single shot detection does not use a neural net that deals with region proposal, hence single shot. It avoids this step by computing both the location and class scores using small convolution filters. Once the feature maps are extracted, a 3 by 3 convolution filter gets applied for each cell to make predictions. These filter results are just like the regular CNN filters. Each filter then outputs a certain number of channels with scores for each class plus one boundary box.

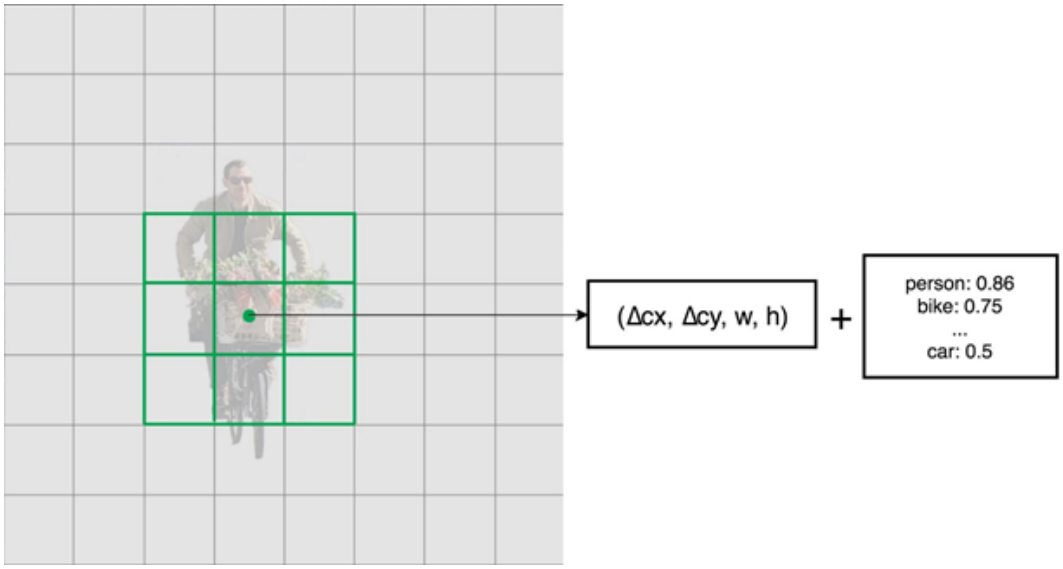


Figure: Graph representation of boundary box [4]

This would be for one layer, but single shot detection actually uses multiple layers because there can be other objects as well. A CNN reduces the spatial dimension gradually and the SSD uses lower resolution layers to detect larger scale objects. Example on next page.

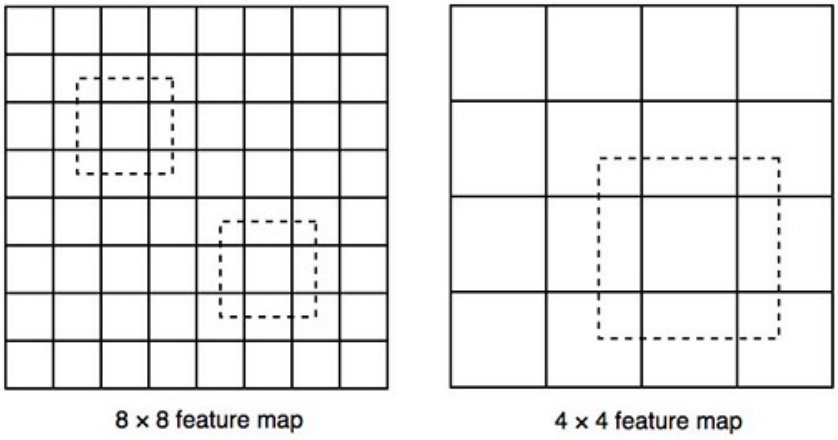


Figure: 8x8 feature map vs 4x4 feature map [4]

This makes sense because if the object is larger than there is no need for smaller pixels to detect it or a smaller grid.

As mentioned before, SSD uses VGG16 when implemented but on top of that it adds six more convolution layers. Five of the six will aid in object detection and three of the five make six predictions instead of 4. So, if we total this, the SSD makes 8,372 predictions because of the six layers. Here is an image showing these extra layers.

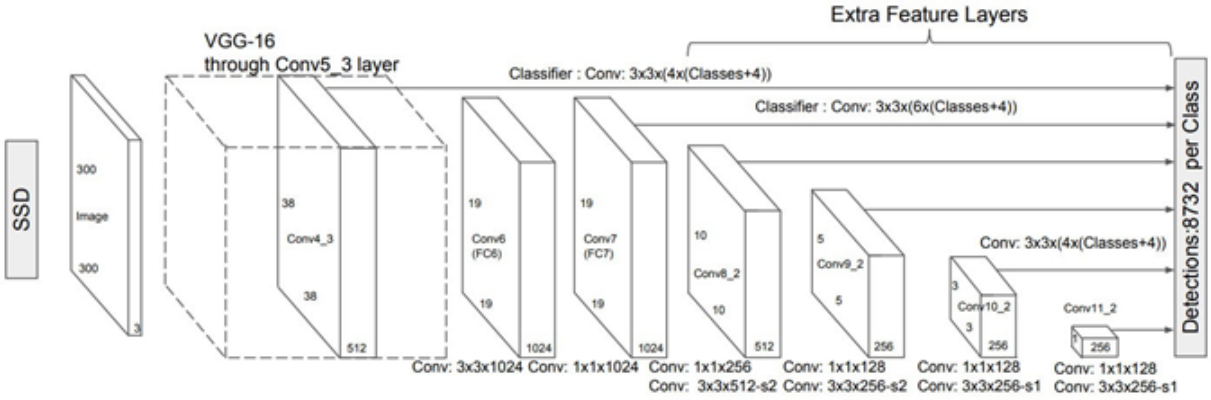


Figure: SSD components including VGG-16 layer [8]

With the addition of those multiple layers, accuracy increases by a lot. Here is an image showing the accuracy with different numbers of feature map layers.

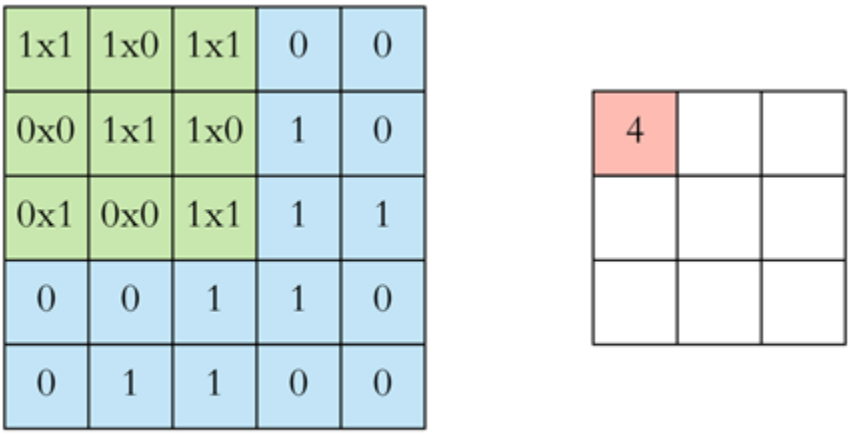
Prediction source layers from:						mAP use boundary boxes?		# Boxes
38 × 38	19 × 19	10 × 10	5 × 5	3 × 3	1 × 1	Yes	No	
✔	✔	✔	✔	✔	✔	74.3	63.4	8732
✔	✔	✔				70.7	69.2	9864
	✔					62.4	64.0	8664

Figure: Feature map comparison [4]

Convolutional Neural Networks

Convolutional neural networks (CNNs), as opposed to feed-forward networks, examine data through a window, preserving the spatial relationships of inputs as it gathers semantic data. For this reason, CNNs are popular and useful in image-classification models.

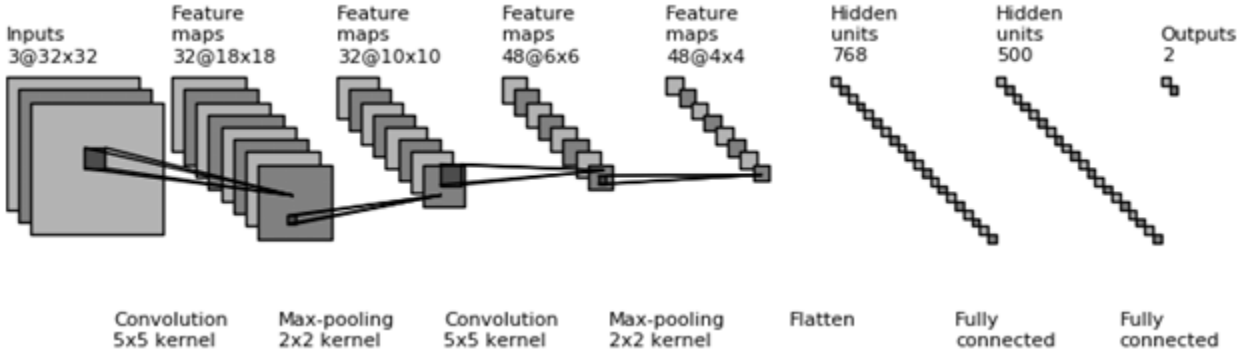
A CNN views an image through different windows, or filters, which are represented as a matrix. These filters, which are applied to subsections of the input image, aim to describe certain characteristics of that image. For example, a filter applied on an image of a square might attempt to identify vertical lines, and another filter might try to identify corners or horizontal lines. Illustrated below is a basic example of a filter (green) applied to an input image (blue). The filter will then be applied to every section of the input data, a process known as convolving, and the dot product of each pass is stored in a feature map. Strides define how many pixels the filter should move between passes. If stride is one, then the squares shift one column to the right.



Input x Filter Feature Map

The result of these filters generates a feature map that describes where in the image the characteristic from the filter was present, thereby maintaining the spatial data of the inputs. Further filtering can be applied to the results of the previous filters, creating increasingly specific information about the image. Between these layers, or

convolutions, down sampling is used to reduce the size of the resulting feature maps. Below are some traditional examples of convolutional neural networks.



The image above shows the general process of a CNN, in which a filter is applied to the input data, generating multiple feature maps. Another set of filters is then applied to the resulting feature maps, and so on until a final output is created. Down sampling is also applied here to lessen the number of outputs. The feature maps in the initial layers likely describe more general information about the image, and the final feature maps describe more specific characteristics of the image. A comprehensible interactive visualization can be found at https://www.cs.ryerson.ca/~aharley/vis/conv/ , created by Adam W. Harley.

TensorFlow

TensorFlow is a fundamental open-source platform used to design machine learning models and neural networks. The scalability, simplicity, and comprehensiveness make it a popular choice in the industry. Benefits of using TensorFlow, according to their website, include:

- Data pre-processing
- Data ingestion
- Model evaluation
- Visualization
- Collection of machine learning tools

Our sponsor has also recommended that we familiarize ourselves with TensorFlow, as it will give us experience designing machine learning models and is compatible with ROS, which we will use to manage our robot.

Downloading TensorFlow is a simple process, and a complete guide can be found in https://www.tensorflow.org/install/pip#ubuntu . Installation instructions for Windows, Mac, and Ubuntu can be found here. Because our ROS runs in an Ubuntu virtual machine, it is fortunate that TensorFlow can be easily installed in Ubuntu. Installing TensorFlow is as simple as running a few command line arguments.

Keras

Another useful library to consider is Keras, which is a neural network library that simplifies designing machine learning models with high-level architecture. Jupyter Notebook and Google Colab can both be used with Keras and TensorFlow to provide a basic example of an image classifier. From the Jupyter notebook, found at https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb , the following functionalities are apparent:

- Loading in data.
- Choosing activation function. In many cases, the activation function will be RELU, with a softmax function applied at the end.
- Applying a softmax function to the data
- Setting up the layers of the model with the Sequential function found in Keras's library
- Evaluating the model's accuracy using the metrics to provide loss and accuracy information over the course of the epochs.

Keras and TensorFlow are available in Python, the programming language we will likely use to design our model. NumPy is another useful library that can be used in conjunction with Keras and TensorFlow to speed up the calculations of our model.

TensorFlow – Building a model

TensorFlow is designed to quickly and easily build models for machine learning using its various functions and libraries. By using some of the tools that Keras provides, layers can be easily added to a machine learning model. The model can also be evaluated with Keras and visualized with *matplotlib* to assess its accuracy. In the following example, TensorFlow and Keras are used to detect if a hand is forming a rock, paper, or scissor gesture. The code is written in Python on a Jupyter Notebook.

The first step is to import all the necessary classes, several of which come from TensorFlow. These import statements are necessary to use a feature extractor later and to import datasets from a large database that contains several preprocessed images, as well as to build the model.

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
from tensorflow.keras import layers

import numpy as np
import matplotlib.pyplot as plt
```

In particular, the most important imports are tensorflow, tensorflow_hub, and layers, which come from tensorflow.keras. TensorFlowHub provides access to transfer learning, which means that pretrained architectures created by professionals can be used for feature extraction. The layers import is necessary to build the model later.

The next step is to load the input images. In this example, the code will receive images from TensorFlow datasets, which were previously imported. The load function accesses the dataset and returns a split as specified by the user. For example, in the case of the ‘rock_paper_scissor’ dataset, the images come split into a training and testing group, as seen on the TensorFlow Datasets documentation.

```
(trainset, testset), info = tfds.load(
    name  = 'rock_paper_scissors',
    split = ['train', 'test'],
    as_supervised = True,
    with_info = True
)
```

```
print(info.features['label'].num_classes)
num_classes = info.features['label'].num_classes
```

```
3
```

The ‘with_info’ argument returns information about the dataset, such as the number of examples and classes, and the ‘as_supervised’ argument returns an input-label tuple. Before continuing, it is necessary to preprocess the data by formatting it to the appropriate size. In this case, the feature extractor that will be used requires that the images be 224 by 224 pixels. The following code applies the formatting to each image and generates batch sizes of 32 for all the data. In the case of the training set, it also randomly shuffles the input images.

```
image_size = 224
num_examples = info.splits['train'].num_examples

def formatinput(image, label):
    image = tf.image.resize(image, (image_size, image_size))/255.0
    return image, label

trainset = trainset.shuffle(num_examples).map(formatinput).batch(32).prefetch(1)
testset  = testset.map(format_image).batch(32).prefetch(1)
```

The next step is to acquire the feature extractor. A feature extractor analyses the input for patterns using the convolutional neural networks. The TensorFlow Hub website has an extensive list of pretrained feature extractors that are available. Because these architectures have been designed and implemented by experts of machine learning, it is much more beneficial to use one of these premade feature extractors, as opposed to designing one’s own.

This example uses ResNet-50 for its feature detection. ResNet-50, as described in this document, is used by RetinaNet in conjunction with a Feature Pyramid Network for object detection. The TensorFlow Hub documentation provides an example on how to download and use ResNet-50, which is similar to the code below.

```
URL = 'https://tfhub.dev/tensorflow/resnet_50/feature_vector/1'
feature_detector = hub.KerasLayer(URL, trainable = False)

model = tf.keras.Sequential([
    feature_detector,
    layers.Dense(num_classes, activation = 'softmax')
])
```

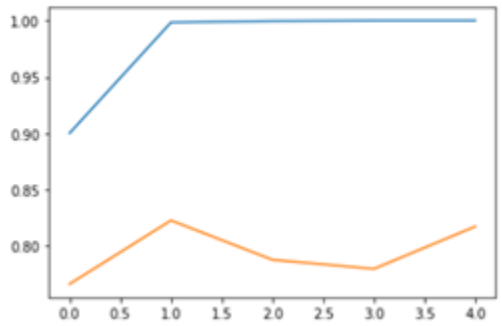
Here, the feature detector is downloaded from the TensorFlow Hub URL, and the 'trainiable' parameter is set to false, as we do not want to alter the pretrained weights. Next, the model is built using the 'Sequential' function. In this model, the feature detector detects patterns, and a final fully-connected, or 'Dense', layer is applied to the model. This last layer separates the inputs into a specified number of classes, which is three in this case (rock, paper, or scissor).

Now that the model has been built, it is time to compile and run it using the 'compile' and 'fit' methods. In the compile method, the optimizer and loss functions are specified. In the fit method, the number of epochs, the input stream, and the validation data are specified. The 'verbose' argument determines how much information is printed after each epoch.

```
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy'])
history = model.fit(trainset, epochs=5, validation_data=testset, verbose=1)
```

After some time, the training is complete. To visualize the outputs, matplotlib, which was imported earlier, provides a plotting method that will be useful to represent the data. Using the metrics from the model, the accuracies of both the training and validation set can be plotted. This is helpful for detecting overfitting and underfitting, as well as for tuning parameters.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.show()
```



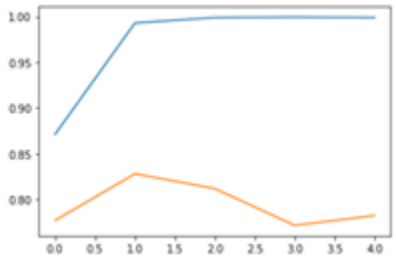
The blue line represents the accuracy of the training data, and the yellow line represents the accuracy of the validation data. After the first epoch, the model appears to have reached near-perfect accuracy for the training set. In fact, upon checking the exact numbers, the training accuracy for epochs 2 and 3 is 99%, and epochs 4 and 5 achieved an accuracy of 100%. On the other hand, the validation accuracy appears to not have improved past the first epoch, and is significantly worse than the training accuracy. This is likely indicative of overfitting. Luckily, there are a few ways to address this. Dropout can be introduced to prevent overfitting, or an early stop might also be beneficial. In this case, however, adding dropout improves the accuracy to 82%, but makes the validation accuracy much more stable.

Changing the feature detector is also fairly easy. Changing the detector from ResNet-50 can be done by changing the URL and changing the formatting of the input images as specified by the documentation. For example, in the following code the feature detector is changed from ResNet-50 to MobileNet, which is another classifier described in this document.

```
# URL = 'https://tfhub.dev/tensorflow/resnet_50/feature_vector/1'
URL = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4'
feature_detector = hub.KerasLayer(URL, trainable = False)

model = tf.keras.Sequential([
    feature_detector,
    layers.Dropout(.2),
    layers.Dense(num_classes, activation = 'softmax')
])
```

The URL is easily changed to the MobileNet by copying it from the website. The documentation states that the input images are 224 by 224 pixels, just like in ReNet-50. By plotting the results in matplotlib, the model shows signs of overfitting.



Therefore, stopping the model after one or two epochs would be beneficial in this example.

Setting Up a Deep Learning Machine

Before selecting a computer vision model, it is necessary to set up a machine for deep learning. Installing the CUDA will be greatly beneficial to us, as it provides ideal deep learning environments and optimizations. The benefits and importance of GPUs and CUDA are highlighted below.

GPU

Because of the massive amounts of calculations required for deep learning, running an object detector on a CPU is not optimal. Fortunately, we can use our computer's Graphics Processing Unit (GPUs) to greatly hasten calculations using parallel processing. Most experts recommend NVIDIA drives for deep learning, as they are designed with deep learning in mind. Adrian Rosebrock, author of *Deep Learning for Computer Vision with Python*, recommends buying or renting a GPU, or using a cloud service such as Amazon or Google with time-based charges. However, he also notes most GPUs will work effectively if they are tweaked a little. On Windows 10, we can check what GPU we have by

1. Opening Task Manager by right clicking the taskbar
2. Selecting the "Performance" tab in the Task Manager
3. Selecting GPU

For example, my GPU is indeed from NVIDIA (GTX 1070), which makes it compatible with CUDA. For training, we will use the free GPU from Google Colab, which will allow any member to train the model, irregardless of their GPU. It also bypasses any CUDA and cuDNN installations and dependency issues. Once training is complete, the weights produced by Google Colab should be downloaded and moved to a local machine. Here, installing CUDA and cuDNN is necessary to increase the frames per second (FPS) of the object detector. Therefore, the local machine should have an NVIDIA GPU.

CUDA

CUDA (Compute Unified Device Architecture) is a parallel processing platform that is commonly used in conjunction with NVIDIA GPUs for deep learning. CUDA allows users to utilize their GPUs to significantly speed up computationally intensive calculations.

A developer using CUDA can use a kernel function, which is marked with the _global_ keyword, to simultaneously run thousands of threads at the same time. CUDA organizes these threads into grids, and these threads within each grid can cooperate with each other. To install CUDA:

1. Verify the GPU is CUDA compatible
    a. Details on how to check the GPU are above
2. Check if the GPU is NVIDIA
3. Download the CUDA toolkit
    a. http://developer.nvidia.com/cuda-downloads
    b. Choose preferred installation
4. Execute the CUDA installer and follow the prompts
5. Verify the installation
    a. Run provided sample programs to ensure the hardware and CUDA software are properly communicating.

CUDA Installation – Ubuntu

Because most of the project will be developed in a virtual machine with Ubuntu 20.04, it is important to download and set up the correct version of CUDA. Fortunately, CUDA can be mostly installed by running commands from the command line. Simply open up the terminal by pressing *Ctrl-Alt-T* or by manually clicking the terminal icon on the computer. Next, run the following command.

```
$ sudo apt update
$ sudo apt install nvidia-cuda-toolkit
```

It is recommended that the command be run in the home directory, which can be accessed from anywhere by running *cd ~*. Also, using *sudo* runs the program as the

root user, which gives the user access to all the permissions. For this step, inputting the computer's password will likely be necessary before proceeding.

After running the above command, verify that the installation has been correctly installed by running the command *nvcc –version*, which displays the version of CUDA. The output should be something similar to this:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Sun_Jul_28_19:07:16_PDT_2019
Cuda compilation tools, release 10.1, V10.1.243
```

This command shows the version of CUDA that is being used. For darknet usage, the version of CUDA should be at least 10. If CUDA was not properly installed, then running the terminal will not recognize *nvcc* as a valid command. Finally, verify that CUDA code can be run on the machine by compiling and testing a simple script. Sample scripts can be found on the NVIDIA website. For instance, to compile and run NVIDIA's sample script called *saxpy*, the code can be copied or downloaded from their website into the machine with CUDA. This code is written in C, but is saved as a *saxpy.cu* instead of *saxpy.c*, the usual format for code written in C. Now that some sample code has been acquired to test the installation of CUDA, it is time to compile and run it. Compiling this particular piece of code is similar to running code written in C, only that *nvcc* replaces *gcc*.

```
nvcc -o saxpy saxpy.cu
```

```
% ./saxpy
```

If CUDA is properly installed, the code will compile and run and produce an output that is formatted as

Max Error: followed by a percentage. A look at this code reveals that it is a variant of the SAXPY function, which takes two arrays and a scalar and multiplies everything in the first aray by the scalar then adds the result to each corresponding element in the second array. This version of the code, however, runs extremely quickly thanks to CUDA.

A tutorial for using CUDA can be found at https://github.com/llSourcell/An_Introduction_to_GPU_Programming/blob/master/add.cu . The code covers some basic examples that outline the fundamentals of CUDA and explain its usage.

RetinaNet

RetinaNet is a one-stage object detection model that works well with dense and small-scale objects. RetinaNet is widely considered to be an effective model because of its speed and accuracy. It has an open-source implementation and is compatible with ROS, the software that we will use as a meta-operating system for our drones.

RetinaNet consists of three major components:

1. The backbone architecture (ResNet)
    a. Bottom-up pathway
    b. Top-down pathway + Lateral connections
2. The object classification subnetwork
3. The bounding box regression subnetwork

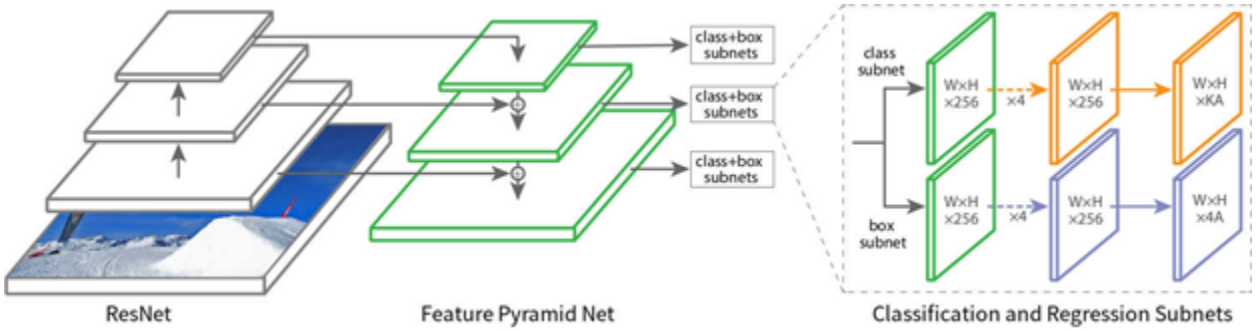


Figure: This image illustrates the different sections of RetinaNet and how they interact.

RetinaNet Backbone

The backbone architecture consists of a Feature Pyramid Network, which in turn consists of layers of feature maps. The deeper the layer, the lower the resolution of the feature map, resulting in the pyramid structure depicted above. The bottom-up pathway creates the Feature Pyramid Network. The top-down pathway, the pyramid in

green, up-samples neighboring feature maps and merges them to create a new feature map. This is repeated until every feature map created in the bottom-up has a new corresponding feature map.

Feature Pyramid Networks are designed to surpass the normal shortcomings of a Feature Pyramid. As the layers of a normal pyramid become deeper, the information in each feature map contains less spatial data but more semantic data. Furthermore, previous Feature Pyramid architectures were computationally expensive.

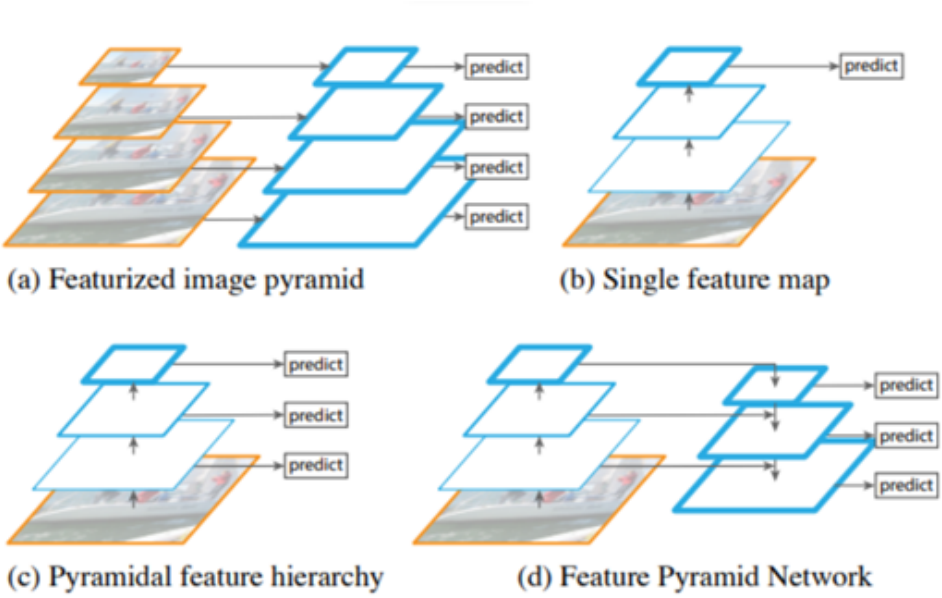


Figure: The above image describes four pyramid architectures.

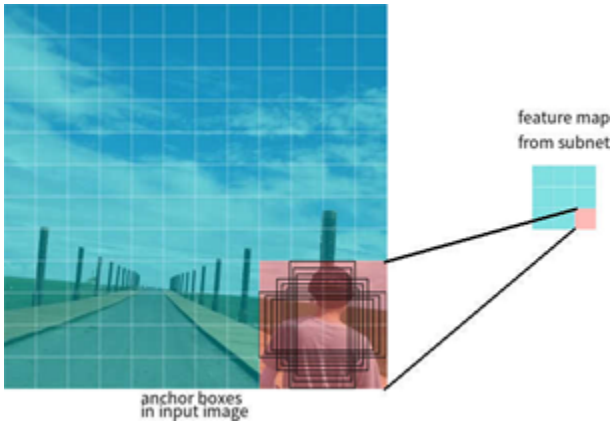
a. Images are subsampled, reducing their size. This creates a pyramid structure.
b. Features are extracted from each level (blue pyramid)
    i. Computationally expensive
    ii. Uses the top feature map
    iii. Faster but less precise
c. CNNs create feature maps that decrease in size after each pass. (c) represents the output of a CNN architecture, which is commonly used in Single shot detectors.
    i. Does not reuse feature maps from different layers.
d. The FPN forms connections between the layers to provide more information.
    i. Deeper layers are smaller, thus containing less spatial data

ii. Initial layers contain less specific semantic data

iii. FPN forms connections between layers to provide both semantic and spatial data.

RetinaNet Object Classification Subnetwork

The object classification subnetwork decides the probability of an object being present inside of the anchor boxes. If the dimensions of a feature map created by the FPN are D, then RetinaNet creates D x D anchor boxes, each of which predicts if an object from C number of classes is present, meaning the resulting feature map has C x (D x D) channels.



The object classification model is attached to each layer of the FPN and uses RELU activation functions and sigmoid functions to generate a prediction. The resulting feature map has a shape of (W, H, C x (D x D)), where W is the width of the input image and H is the height of the input image.

RetinaNet Regression Subnetwork

Similarly to the object classification subnetwork, the regression subnetwork is attached to each layer of the FPN, but results in a (W, H, 4 (D x D)) feature map. The regression

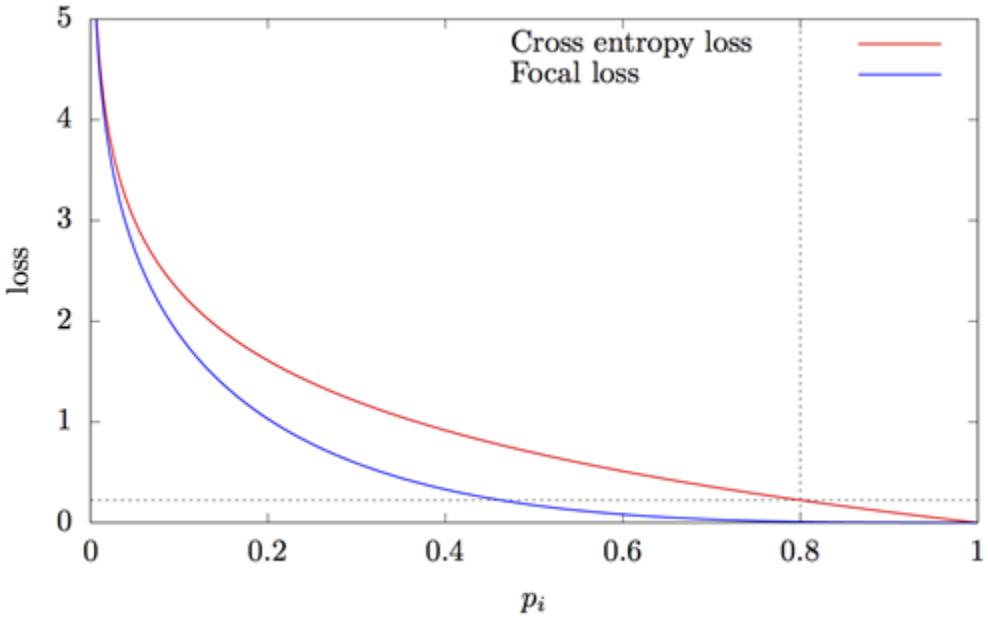
subnetwork is responsible for detecting the size and shapes of the objects found in the anchor boxes and determining the offset between the anchor boxes and ground truth boxes, which are the hand-labeled bounding boxes annotated before training.

Because of the regression subnetwork, all input images must be annotated with ground-truth boxes, which are the bounding boxes that the regression network will reference for its loss function. These ground-truth boxes accurately and neatly enclose the target object, and are usually represented by a set of coordinates. The input format for a ground-truth box might range from a set of four pairs that describe the X and Y coordinates of the corners of the ground-truth box, or one pair of X and Y coordinates that represent the center of the ground truth box, along with a width and height value for the box. Images can have multiple ground-truth boxes, and having accurate annotations for the input images is imperative to the success of the model. Therefore, as described in the Gathering Data sections, it is important to accurately label these with precise software.

RetinaNet Focal Loss

Instead of using a cross-entropy loss function, RetinaNet uses focal loss. Focal loss emphasizes the contributions of fixing incorrect labels and decreases the contribution of easy examples, which improves the overall accuracy of the model.

In 2017, researcher Tsung-Yi Lin and his colleagues came up with the idea to modify the cross-entropy loss function to improve its accuracy. The result improved the accuracy of models such as RetinaNet, causing them to incorporate Dr. Lin's modified version of cross-entropy into their architecture. The idea behind the model can be illustrated by an example. Consider a prediction with a confidence score of 80%. Although one can be fairly certain that this prediction is correct, the original cross-entropy loss function will still count the 20% uncertainty. However, if the loss that the 20% of uncertainty contributes is scaled down, then this prediction, which is likely correct, will contribute less to the overall loss score. However, in the case of a difficult prediction, such as one that only yields a 60% confidence score, the loss score will be more heavily impacted on account of the scale factor. This method "encourages" the model to improve its accuracy in the more difficult examples.



The difference between focal loss and cross-entropy loss is visualized in the above graph, where the blue line represents the focal loss and the red line represents the cross-entropy loss. The X-axis represents the confidence score of a prediction and the Y-axis represents the loss attributed to that prediction. As the graph displays, a score of confidence score of 80% using focal loss contributes virtually nothing to the loss. However, on the opposite end of the graph, extremely low certainty scores using focal loss contribute as much loss as regular cross-entropy.

Data Preparation for RetinaNet

As previously mentioned, RetinaNet is a single shot detector that is both accurate and fast. Detailed below is the process of preparing data before it is analyzed by the model. This step is required not only in RetinaNet but also in most computer vision models.

Gathering Data

Before building the model, we must gather the input data. In our case, we want the model to be able to identify a BB8 model quickly and accurately in our simulation of a suburban environment. Therefore, our input data for our RetinaNet implementation should consist of several images of the turtlebot. To provide useful information to the model, the images should show the turtlebot from various angles, distances, and locations. It might also be useful to place distractors near the BB8 to improve our model's accuracy, or rather precision, as the cost of a false positive would likely be high. The distractors placed should be relevant to the setting in which the simulation occurs, meaning they should be objects that are typically found in an urban or suburban environment.

Splitting and Labeling the Data

After creating the dataset of BB8 images, the next step is to split the data into training, testing, and validation sets. From the complete set, 70% should go to training, 15% to testing, and 15% to validation. If we choose to exclude a validation set then 80% should go to training and 20% should go to testing. The training set should then be divided up into batches, which contain examples from the complete training set chosen at random.

The images in the training datasets should then be labeled using a tool named labelImg (label image), which can be installed from the command line in Ubuntu. With labelImg, we use bounding boxes to label the BB8s in each image. After labeling every image in

the batch, a text file will be generated for each image. These files will be used later in the model.

After data preparation is complete, we must then define the classes. These classes are what our model will use to classify objects. In the RetinaNet tutorial, for example, the classes included airplanes, cars, and automobiles, among other things. In our instance, however, we might only need a binary or even single classification system, as we are only concerned if an object is or is not a BB8.

To summarize the steps of data preparation, they are:

1. Gather images of the BB8 unit
2. Split the dataset into training, testing, and perhaps validation
    a. The training set should account for 80% of all data if no validation, and 70% if there is validation
    b. The testing set should account for 20% if no validation, and 15% if there is validation
    c. If there is validation, use 15% of the entire data
3. Create multiple batches from the dataset

    ·Batches consist of examples from the training data chosen at random
4. Label the images in the batch
    a. Use labelImg to create label boxes
    b. We might want to look into a quicker way of labeling the training images, as we could potentially have thousands of input images.
5. LabelImg will generate an xml file for each image
6. Define the classes
    a. In our case, we might only need one or two classes.

Labeling Data

Like most computer vision algorithms, YOLOv3 relies on bounding boxes to properly train the model. To conduct supervised learning, input images should be pre-labeled and annotated with bounding boxes that encompass the object that the model is being trained to detect. Because the model will use these bounding boxes to search for an object, it is important that they be accurately and properly generated. However, there is no simple way to quickly generate these boxes on input images. One possible solution is to create a script that will automatically draw these boxes. The script relies on linear interpolation, which is explained below.

Linear Interpolation

Linear interpolation is a mathematical method of connecting data by using polynomials. For example, if you have two values A and B, linear interpolation estimates the value of a third point between A and B, which can be accomplished by drawing a line between A and B and taking the value at that third point. On a two-dimensional plane, linear interpolation always results in a straight line.

Calculating the line between points A and B can be done using this formula:

$$y = y_1 + (x - x_1) * (y_2 - y_1) / (x_2 - x_1)$$

This formula can be used to find the slope and intercept of the line, which gives the slope-intercept form of $y = mx + b$.

Consider the example of (5,10) and (3,6):

$$y = 10 + (x - 5) * (6 - 10)/(3 - 5)$$

$$y = 10 + x(-5) * (-4)/(-2) = 2x$$

Using this information, we know that the value 4 gives a y-value of 8.

Although the mathematics behind linear interpolation is fairly basic, it is a powerful tool that can be useful for generating bounding boxes. With some pieces of information about the pixel placement, a bounding box can be generated with minimal information.

For example, using an image's pixel arrangement, we can get the row and column of the top-right corner and bottom-left corner of the desired output box and conduct linear interpolation to generate a bounding box with the following dimensions:

height=|*top-bottom*|

width=|*right-left*|

Conversely, the top-left corner and bottom-right corner of the desired box can be used to generate the same bounding box. Another strategy that might be useful is to select a pixel as the center point of the bounding box and specify the height and width of the desired result.

Automating Labeling

Because the computer vision model will require thousands of input images, finding a way to automate the bounding box process might be useful. With the information on linear interpolation from above, a Python script can be created to accomplish automation.

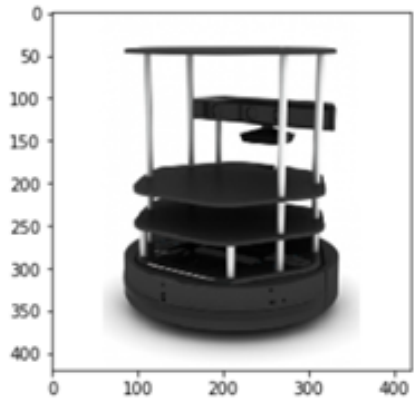
The first step, as always, is to import the necessary libraries. In this case, *cv2* will be required to import and draw on the image, and *pyplot* from *matplotlib* will be necessary to display the image.

```
import cv2
import matplotlib.pyplot as plt
```

The next step is to import the image. To do so, the *imread()* function provided by cv2 will be useful. The function requires that the path of the desired input image be specified. In this case, the path is hardcoded into the function call, but, if this script were to generate bounding boxes for thousands of data, another function would be necessary to change the path to the correct script. Naming the pictures sequentially (picture00001, picture00002…) would be the ideal way to handle inputting the data, as the image paths can be easily specified.

```
image_path = "turtlebot.jpg"
image = cv2.imread(image_path)
```

The image in this script is a sample image of a turtlebot found online. By using the print function from *pyplot*, the picture is displayed on a graph such that the height and width of the pixels can be approximated.



As seen above, this image is approximately 400 by 400 pixels. Using the information on the x-axis and y-axis, the bounding box can be approximated. As discussed above, the coordinates of two opposing corners of the desired result is all the information necessary to generate a bounding box. In this case, the top-left corner is approximately located at pixel (70, 35) and the bottom-right corner is approximately located at pixel (350, 390). The rest of the bounding box's pixels will be generated with linear interpolation using the *rectangle* function from *cv2*. The documentation for *cv2* states that the *rectangle* function takes as its arguments the image, the corners, the desired color of the bounding box, and the desired thickness of the bounding box.

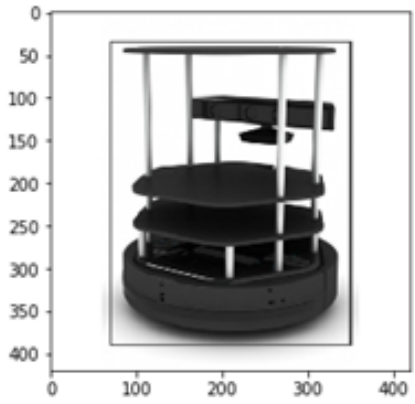
```
top_left     = (70, 35)
bottom_right = (350, 390)

color = (80, 80 , 80) # Gray

thickness = 2

image = cv2.rectangle(image,
                      top_left,
                      bottom_right,
                      color, thickness)
```

Finally, the resulting bounding box can be printed using pyplot's *imshow()* function.



Indeed, as expected, a gray bounding box has been generated. The bounding box is accurate, and successfully enclosed the entire turtlebot. However, there are some issues that must be worked out to apply this script to the entire dataset, namely the positioning of the box. In the input datasets, the positioning and proximity of the turtlebot are bound to vary. If the bounding box is always in the center, then the results will likely be inaccurate, as there is no guarantee that the turtlebot will be positioned in the center image. Furthermore, a turtlebot that is in the center but far away will also provide poor information, as the bounding box will have too much empty space.

To remedy this, there might be two possible solutions. The first solution is to ensure that the turtlebot's location is relatively static, ensuring that both the positioning and proximity of the turtlebot in the input image will allow it to fit comfortably within a predetermined bounding box. However, this might be difficult to achieve, and might also negatively impact the model's ability to make predictions from various angles. The second solution is to take the pictures in a predetermined manner such that predicting where the bounding box will go can be achieved while also attempting to provide input images taken from different angles.

As our sponsor suggested, a way to gather input images is to set up a drone with a camera above the bb8 unit. The drone will then begin to circle the drone while continuously taking pictures, providing a sizable number of input images. Initially, the drone's altitude should be at around five feet above the drone and should continue to increase until it reaches 15 feet. This process can then be repeated in different locations, providing background noise that the model will learn to ignore. If the pictures are consistent with their placement of the turtlebot, which they should be as they are

being taken by a drone that moves precisely in the air, the script can choose where it chooses the top-left and bottom-right pixel coordinates accordingly.

Unfortunately, automatic labeling was out of the scope of the project. Although it is theoretically possible to automatically label an image dataset, the process of setting up such a system proved to be more complex and time-consuming than manually gathering and labeling the data. Furthermore, the precision and tightness of the bounding boxes created from manual annotations would be superior to those produced by an automatic script.

Manually Labeling Data

There are several different software options that can be used to manually label this data, but the main one used in this project will be labelImg.

To install labelImg, the repository should first be cloned from GitHub. Like all GitHub repositories, labelImg can be easily downloaded using git by locating the green button labeled ‘Copy’ in the repository and clicking the clipboard icon. Running the command “git clone [URL]” will download all the content from the GitHub.

After the GitHub repository is installed, setting up labelImg is still necessary. The instructions in the GitHub repository to set up labelImg vary depending on one’s operating system, software, and preferences. Assuming Anaconda is downloaded on a Windows OS, the following command should be run after downloading the repository after entering the labelImg directory, which should have been created after being cloned from GitHub.

```
conda install pyqt=5
conda install -c anaconda lxml
pyrcc5 -o libs/resources.py resources.qrc
python labelImg.py
python lablImg.py [IMAGE_PATH] [PRE-DEFINED CLASS FILE]
```

With these commands done, labelImg is almost ready to be used. But, before that, two more steps are necessary. As per the usage instructions, labelImg takes in two parameters when run. The first parameter is a directory with all the images that will be annotated and labeled, and the second parameter is a text file with a text file containing a list of the classes. (Using the script without parameters still works, but having the two parameters simplifies searching for the classes and images).

In this example, a folder with three images of the bb8 unit has been set up. These will be the input images in this case.

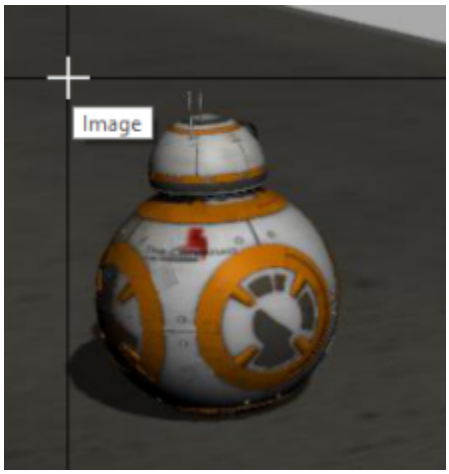
As mentioned before, a text file should be created with a list of classes (e.g. cars, people, trees, cats, or dogs). In this case, however, the only class will be “bb8.” Now that both the folder containing the input images and the class text file have been created, the python script can be run as follows:

```
(base) C:\Users\pablo\Desktop\GitHub Repos\labelImg>python labelImg.py images classes.txt
```

Here, the two arguments are present: images and classes.txt, where “images” is a folder with input images and “classes.txt” is the text file containing the list of classes. Also, the above command should be run in the Anaconda command line if the Anaconda installation and set up instructions were followed, as shown previously.

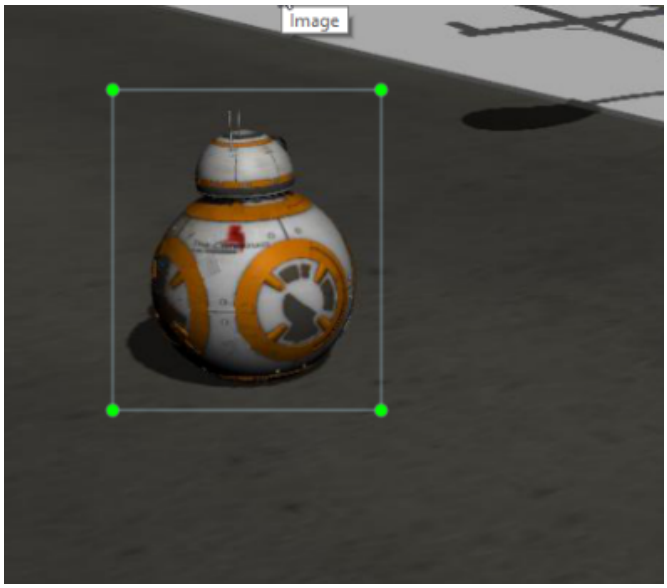
A new program called “labelImg” should now be open with an image from the input folder. There is a list of commands on the left side of the screen, each of which provides a helpful function to facilitate the labeling process. Before beginning annotations, however, the button on the left labeled “PascalVOC” should be clicked once, changing it to “YOLO.”

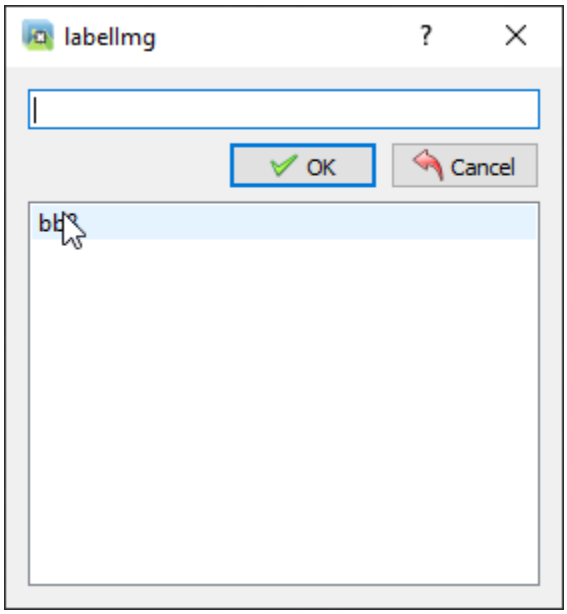
The other features of the tools section include buttons to zoom in and out of the picture, “Next” and “Previous” buttons to navigate the selected images folder, buttons to change image folders if necessary, buttons to change the save folder, and, of course, an option to create a box. To create a new bounding box, firstly press the “Create RectBox'' button. This will create two black lines, one of which is horizontal and the other vertical. The black lines intersect wherever the cursor is on the screen, and are designed to help users accurately create their desired bounding box. The cursor, or the intersection of these two black lines, should be placed just above or below of the target and just to its left or right. In other words, the cursor should be placed in a corner of the bounding box that will be generated. The black lines aid the user by providing a visual representation of where the horizontal and vertical sides will be. With these lines, the user can ensure that the starting corner of the box is appropriately located so that it can encompass the entire target by placing the cursor such that there is a small amount of room between the target and the lines. The bounding box should tightly surround the target, but leave sufficient space so that the edges and corners of the target are still within the bounding box. This is the biggest advantage of manually labeling the data. By using labelImg, it is easy to ensure the quality of the annotations and bounding boxes.



As seen here, in this case, the cursor is placed in the top-left corner of the future bounding box. With the two black lines, it is easy to ensure that the entire target will be encompassed within the bounding box. With the top-left corner of the box location, simply clicking and dragging the cursor to the opposite corner, in this case the bottom-right of the BB8 model, will generate the rest of the image. Like last time, helpful lines will appear to ensure that the BB8 is being appropriately labeled.

Remember to ensure that the entire model is being captured within the bounding box. It is acceptable to leave a small amount of extra space within the bounding box, as it is more important for the entire BB8 to be within the bounding box, including its edges, than to have a tight bounding box.





After comfortably enclosing the target within a bounding box, a pop-up will appear. This pop-up shows a list of possible classes, which are accessed from the classes.txt input file from earlier. In this case, only one class exists. Finish by selecting the class, clicking the OK button, and saving the image. The image will be saved as a text file that

contains the corners of each bounding box within the image. If there were more than one target in this image, a bounding box for each target should be generated. Furthermore, if the user believes that this image will be difficult for training, whether it be because of positioning, proximity, background noise, or something else, the image can be labeled as ‘difficult’ by checking a box in the top right corner.

To continue navigating the folder with the images, simply click the “Next” button and continue labeling. Although this method produces high-quality bounding boxes, it is comparatively slow to the scripting method.

Because there is only one class, we used 1344 images in total. These images were hand-labeled using labelImg and took approximately three or four hours to gather and label. From those images, 80% were used for training and 20% were used for testing.

Evaluating the Model

Evaluating the model’s accuracy is as important as training it. Because our project aims to disable a predetermined target, it is incredibly important that the robot has a high degree of accuracy, as the consequences of an incorrect classification could be severe. Fortunately, there are different methods of visualizing the areas where the model’s performance struggles to generate accurate results. These visualizations provide useful insights into the model’s architecture, machinations, and flow. Moreover, there are different strategies that we should be conscious of that can improve the model’s accuracy. As shown in the YOLO section of this document, YOLOv3 is extremely accurate, precise, and quick when compared to other models, meaning that YOLOv3’s classifier, which is called darknet, handles and implements many paradigms that effectively reduce error.

In one of the previous examples of this document, TensorFlow was used to implement a feature vector using ResNet with TensorFlowHub. In that example, the metrics are specified at the time that the model is compiled.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

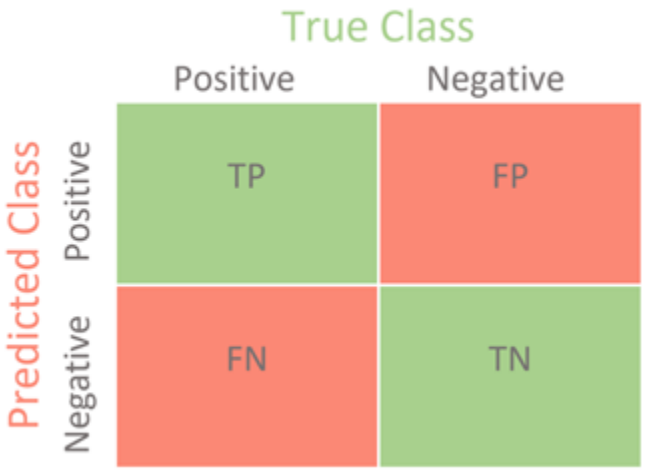
As seen above, setting the metrics argument equal to [‘accuracy’] generates lots of useful information about the data, specifically its loss, validation loss, overall accuracy, and validation accuracy. These metrics are useful for detecting overfitting and underfitting, as well as displaying the model’s behavior as it develops over the course of the epochs. The information from metrics can be accessed after the model is trained using model.fit.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.show()
```

This snippet of code, taken from the same example, shows how the metrics information specified above can be displayed in a graph. In this case, plt refers to matplotlib.pyplot and history refers to the result of model.fit(..). A more detailed evaluation of the specific results of this particular model can be found in the TensorFlow section. These metrics are incredibly useful and should always be included, as the graphs can clearly visualize the model’s tendencies. For example, the metrics can be used to detect overfitting. The user might compare the graphs for loss and validation loss as well as the graphs for accuracy and validation accuracy to decide if an early stop would be the appropriate course of action to mitigate error in the model.

Confusion Matrix

A confusion matrix is a clear and concise visualization of the data, and can be reliably used to interpret the tendencies and overall performance of a computer vision model. A confusion matrix is a two-by-two matrix, and each cell within that two-by-two matrix displays a different value that indicates some piece of information about the model that is being evaluated. Calculating the values for each cell is fairly simple, and metrics such as accuracy and precision, which measure two different things, can be acquired from the results.



Above is the format of a confusion matrix. The confusion matrix consists of four grids, each of which shows how the model performed relative to the actual score. The confusion matrix consists of the following:

- The True Positives, which indicates how many correct positive predictions the model gave.
- The False Positives, which indicate how many incorrect positive predictions the model gave.
- The False Negatives, which indicates how many incorrect negative predictions the model gave.
- The True Negative, which indicates how many correct negative predictions the model gave.

In the case of this project, a positive prediction refers to the model identifying an object as a BB8 unit. Likewise, a negative prediction refers to the model identifying something as anything that is not the BB8 unit. A confusion matrix can demonstrate if the model has a tendency to generate false positives or false negatives, which facilitate the tuning process to improve the overall score of the model. The true negatives, however, are relatively irrelevant. This is because it is difficult to determine what qualifies as a true negative. For example, if a true negative is counted each time the model correctly ascertained that no target is present, then the true negative rate would be colossal. Therefore, if a confusion matrix were to be generated, the true negative section would be omitted.

Using the numbers from each cell, four ratios can be calculated: accuracy, recall, precision, and the F1 score.

Accuracy measures how many correctness of the model compared to all predictions. That is, accuracy is determined by adding the true positives and true negatives and dividing them by the total. However, accuracy can be misleading depending on the data. For example, a model that always predicts positive will have a high accuracy if the majority of real-life cases are positive. Accuracy works best when the real-world distributions of positives and negatives is relatively symmetrical. However, because it relies on the true negative score, it is impossible to measure for an object detector.

A = (TP + TN)/(TP+TN+FP+FN)

Precision measures how many of the model's positive predictions were actually correct. This is calculated by taking the value for True Positives and dividing it by the sum of True Positives and False Positives, the total number of positives.

P = TP / (TP + FP)

Recall measures how many positive cases the model captured. This is calculated similarly to the precision rate. To calculate recall, divide the True Positives by the sum of the True Positives and False Negatives, which is the sum of real positives.

R = TP / (TP + FN)

The F1 Score is another metric that evaluates the overall performance of the model. However, because recall, accuracy, and precision might display misleading information, the F1 Score aims to balance out the results of the previous metrics. The F1 Score is calculated in the following way:

F1 = 2 * (Precision * Recall) / (Precision + Recall).

Recall and Precision, however, can both provide useful information depending on the context of the project and nature of the dataset. Recall should be examined closely when the cost of a False Negative is high, and Precision should be examined closely when the cost of a False Positive is high. In the case of this project, the model incorrectly labeling an object it sees as a bb8 unit will cause the robot to navigate

towards it. Therefore, precision should be prioritized as the cost of a False Positive is high.

At the end of the project, our model performed extremely well. Our testing dataset contained 268 images with 268 target objects, of which 267 were correctly identified. So, we had 267 true positives, 1 false negative, and 2 false positives. Our metric scores were as follows:

Recall: 99.6%

F1-Score: 99.7%

Mean-Average Precision (MAP): 99.98%

Using RetinaNet

There exist several general-purpose implementations of RetinaNet on the internet. Therefore, we should decide if we want to build a model from scratch or adjust a premade implementation. One such implementation can be found at https://github.com/jaspereb/Retinanet-Tutorial , and another at https://github.com/fizyr/keras-retinanet . Building a model from scratch and selecting a premade model both have their benefits, so we should carefully consider our needs and expectations.

Building RetinaNet

Building a model from scratch can be an informative experience, though a bit more involved and time-consuming. In fact, our sponsor recommended that we create our own model if we want to gain experience with machine learning. A Python implementation can be accomplished using TensorFlow, NumPy, Keras, and matplotlib. Several tutorials exist, although they can be rather complicated to understand. As previously described, TensorFlow and Keras provide many useful functions that would facilitate this process. For example, built-in activation functions and loss functions are

available in TensorFlow. Because of the massive amounts of calculations, NumPy is essential to speed up the process.

After preparing the data and defining the classes, we would have to create the backbone of our model by generating a FPN, make the necessary connections to generate new feature maps, and create the object classification and regression model. Afterwards, we would have to evaluate the model's precision, accuracy, recall, and F1 score.

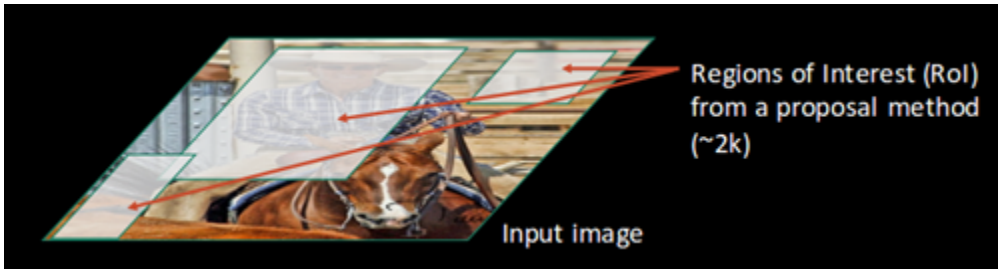
Premade RetinaNet

Premade implementations of RetinaNet can also be found on GitHub and TensorFlow. Because the code for these models is already made, selecting a premade implementation might save us time. We would, however, miss out on a learning experience. If we choose a premade model, we must still adjust it to fit our needs more closely. Namely, we must change the classes defined in the model to reflect our project, and perhaps even change other aspects of the code.

A third possibility is to combine these two options by using a pretrained CNN. In this case, only the convolutional neural network is premade, and our responsibility would be to implement the other aspects of the model, such as the data preparation and evaluation. This option would save us time while also giving us hands-on experience with computer vision.

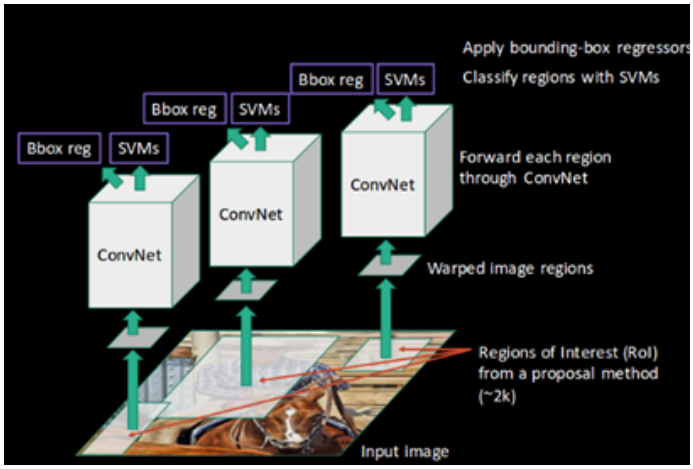
RCNN

RCNN stands for Region-based Convolutional Neural Network (RCNN). In certain regards, RCNNs are similar to RetinaNet. Both, for example, require an object classifier and regression. RCNNs use selective searches to extract boxes from images, which will be checked for objects. The selective search proposes regions of interest based on patterns it identifies based on color, enclosure, texture, and scales.



The number of proposal regions usually numbers around 2 thousand. As seen in the above picture, regions of interest are based on patterns in an input image. The CNNs will then identify the objects within these regions of interest.

These steps followed by an RCNN are highlighted in the following image, which uses the same input image as the one used in the example for regions of interest.



The steps are the following:

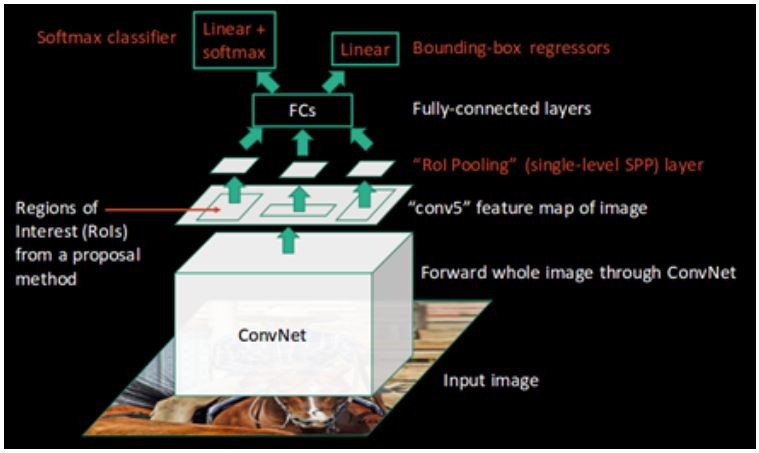
1. Generate regions of interest from an input image
    a. Selective search
2. Regions are passed to the CNN
    a. Before passing the regions, they must be reshaped to match the input size of the CNN
3. A Support Vector Machine (SVM) is used to divide regions into different classes.
4. Bounding box regression is used to generate bounding boxes for unidentified regions.

RCNN, however, has certain limitations that make them computationally expensive. Namely, extracting two thousand regions of interest for each image and running a CNN on each region is a slow process. In fact, it takes RCNN almost a minute to make predictions, making it nearly unusable for large data sets. This is where Fast RCNN and Faster RCNN come in, which aim to fix some of the shortcomings of RCNN.

Fast RCNN

Before understanding Faster RCNN, it is useful to understand Fast RCNN. Processing two thousand regions of interest per image is an incredibly slow process, especially if the number of images is large. Datasets might include hundreds, thousands, or even millions of input images, meaning RCNNs must process 2000 * N regions. Fast RCNN addresses this by running the CNN once per image and sharing the output across the two thousand regions.

In Fast RCNN, the input image is fed to the CNN, generating a feature map. The regions of interest are then extracted from this feature map, after which they are reshaped into a specific size and inputted to a fully connected network. Continuing with the same example, the same image as before would undergo the following process.



The steps are as follows:

1. The image is fed into a CNN
    a. This a feature map from which regions of interest are extracted
2. Apply “RoI Poolng” to every region of interest
    a. This step makes every region of interest the same size
3. The resized regions are passed into a fully connected network
    a. The FCN classifies objects within the regions using softmax regression
    b. The FCN uses linear regression to generate bounding boxes

As we can see, this fixes some of the issues RCNN had. Fast RCNNs do not run CNNs over N * 2000 regions, significantly decreasing runtime. Moreover, the fully connected network applies the classification and bounding box regression simultaneously, further decreasing runtime.

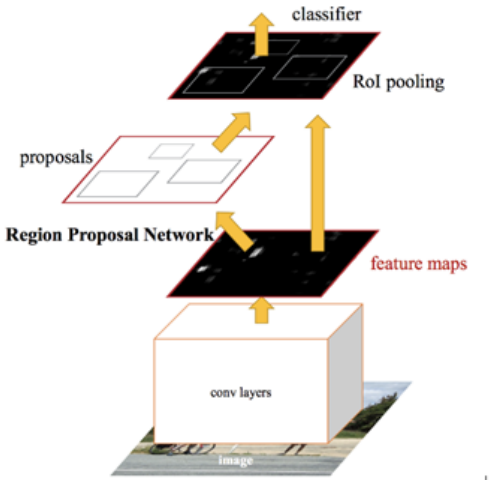
With its improvements, Fast RCNN is faster than RCNN. However, it still suffers from certain limitations. Extracting two thousand regions of interest is still a long process that slows down the model. Fast RCNNs take about two seconds to make predictions. Although this is much better than RCNN, it is still slow when the data set is large.

Faster RCNN

Faster RCNN is a modified version of Fast RCNN. While Fast RCNN uses a selective search method to find its regions of interest, Faster RCNN uses a Region Proposal

Classifier (RPN). The input to an RPN is a feature map, and its output is a set of object proposals with an associated objectness score. RPNs work by using a sliding window to generate anchor boxes. Each anchor box has two responsibilities: to predict if there is an object within the anchor box and to adjust the box to fit the object.

Faster RCNN makes use of similar steps as Fast RCNN, but also varies in its implementation. Overall, these variations and differences allow Faster RCNN to surpass Fast RCNN in terms of both speed and accuracy, making it an excellent choice for a model. In the following image, the steps of the Faster RCNN model are displayed.



As with before, the same image is being used as the input.

1. Pass the image into the CNN
    a. This returns feature maps
2. Apply RPN to the feature maps
    a. This returns object proposals with their respective scores.
3. Use RoI Pooling to make the proposals the same size
4. Pass the results to a FCN
    a. Softmax layer classifies objects
    b. Regression layer predicts boxes
    c. Both run at the same time

As the name implies, Faster RCNN is faster than both RCNN and Fast RCNN. With RCNN, the selective search method located two thousand regions of interest per image and had three networks to generate its results. This resulted in extremely long

predictions. Fast RCNN fixed this by applying the CNN to the entire image and then extracted the regions of interest. A fully connected network then simultaneously ran the classification and regression networks. Faster RCNN further improves upon Fast RCNN by using an RPN. This RPN is much faster than the selective search and decreases the overall runtime. In fact, predictions with Faster RCNN now take only 0.2 seconds.

MobileNetV2

Because we are using a turtlebot3 robot, maximizing accuracy and speed is important. MobileNetV2 is a convolutional neural network architecture designed specifically for mobile devices. Therefore, if we choose to implement Faster RCNN, we would use MobileNetV2 as the CNN.

With Keras, MobileNetV2 can easily be implemented:

```
tf.keras.applications.MobileNetV2(
    input_shape=None,
    alpha=1.0,
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
    **kwargs
)
```

https://keras.io/api/applications/mobilenet/ describes the input arguments of MobileNetV2.

As seen above, there are a total of ten possible arguments necessary to implement MobileNetV2. The first argument is the *input_shape* tuple, which should only be specified if the argument *include_top* is set to False. The documentation states that the input images should have a width of 244, a height of 244, and 3 input channels. In convolutional neural networks, channels are a way to represent the red, green, and blue values of colored images. A grayscale image, for example, would have only a single input channel. Channels allow the convolutional neural network to gain information not

only from the spatial and semantic information on the input images, but also from their coloration. So, because the MobileNetv2 implements this manner of convolutions, it is important to specify that the channels will be three. The expected input tuple should have the form (*width, height, channels*) or the form (*channels, width, height*). In most cases, it is simpler to go with the former formating of (244, 244, 3). Because of this, a function should be declared that resizes all the input images into the expected input size.

Looking back at the code from the TensorFlow implementation of MobileNet using transfer learning from TensorFlowHub, the function is fairly simple using TensorFlow's resize utility function.

The second argument is alpha,which is a floating-point number between 0 and 1. This argument controls the width of the neural network. As explained by the documentation, if the alpha is greater than 1, then the number of filters in each layer decreases. Conversely, if alpha is greater than 1, then the number of filters in each layer decreases. *Alpha's* default value of 1 preserves the number of filters in each layer as described in the MobileNetV2 paper.

The third argument is *include_top*, and dictates whether or not a fully-connected layer at the end should be added to the model. By default, this is true. Because *include_top* defaults to true, *input_shape*, by default should be set to None. However, one might want to remove the fully-connected layer at the top if they wish to change the number of classes and keep only the feature extraction that the previous layers provide.

The fourth argument is *weights*, which should be set either to *None*, "imagenet", as seen in the picture, or to a path to a custom weights file.

Skipping ahead to the sixth argument, the *pooling* argument controls how the results from the filters will be pooled.

- *None* indicates that the result from the filter should keep its shape, which will be the same size as the previous convolutional block.
- *Avg* indicates that average pooling should be used. This method sees the pooling layer conserve the average of a specified window after convolution.
- *Max* indicates that max pooling should be used. As described previously, only the maximum value of a specified window is kept after the pooling layer.

The seventh argument, *classes*, specifies the number of classes that the images should be classified into. According to the documentation, this argument should only be specified if the *weights* argument is set to *None* and the *include_top* argument is set to true.

Finally, the eighth and final argument is the *classifier_activation* argument. This argument is optional if *include_top* is set to false. When using a pretrained top layer, the value of the *classifier_activation* argument should be set either to *None* or to *softmax*, which is an activation function that modifies a series of vectors so that they add up to 1.

The YOLO (You Only Look Once) Algorithm

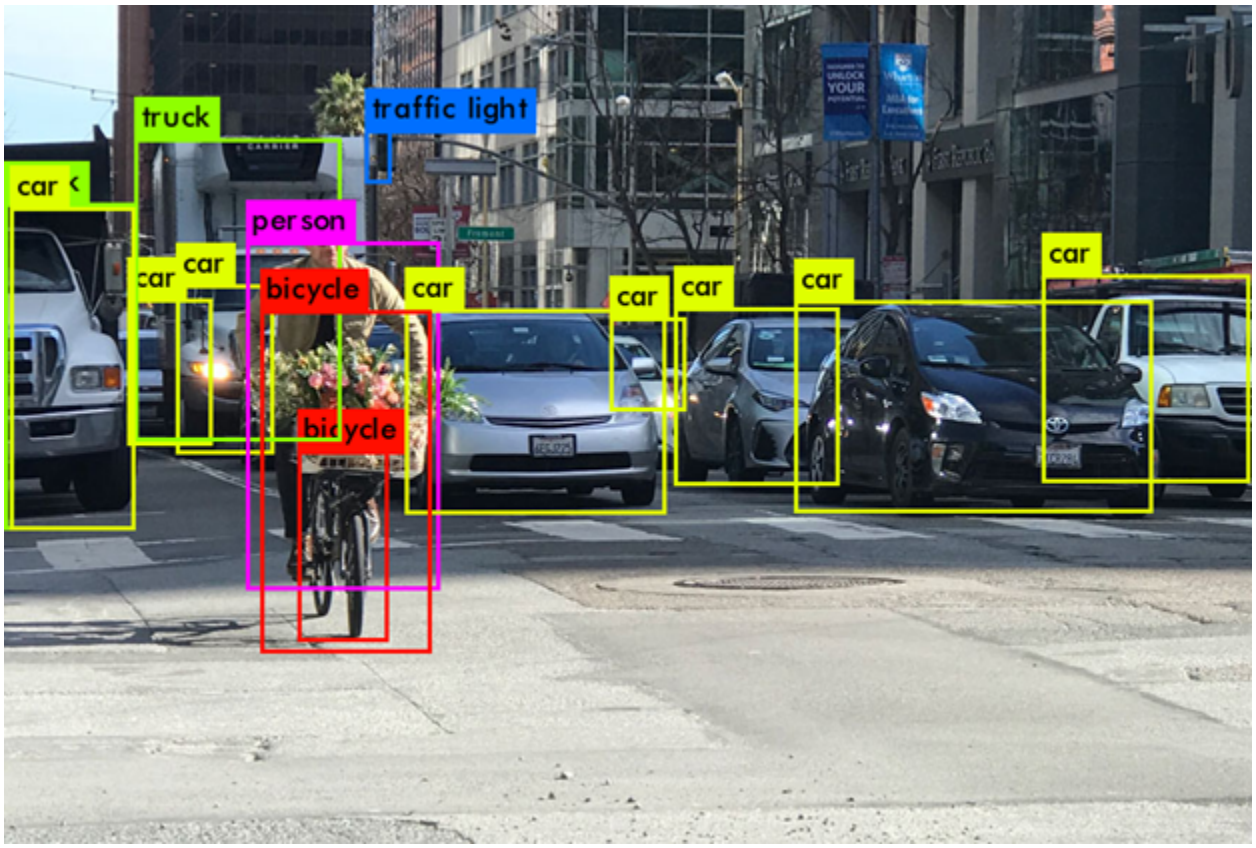


Figure: YOLO object prediction result image [4]

Now finally let us end with discussing the YOLO object detection algorithm. The reason we ended with this was because to understand the YOLO architecture we first had to research many of the principles discussed earlier. YOLO is arguably one of the most effective object detection algorithms used today, created in 2015 by Joseph Redmon.

To start, YOLO became popular because unlike many other algorithms out there for object detection, YOLO only has to look once, hence the name, which drastically increases the runtime while still keeping high accuracy and allows it to be used in real time.

This cite gives a great overview on how YOLO works:

“With YOLO, a single CNN simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This model has a number of benefits over other object detection methods:

- YOLO is extremely fast
- YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance.
- YOLO learns generalizable representations of objects so that when trained on natural images and tested on artwork, the algorithm outperforms other top detection methods.” *[22]*

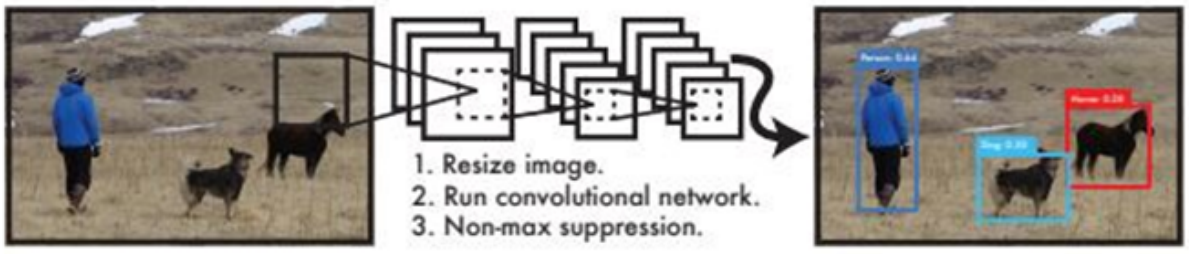


Figure: Generalized steps of YOLO [14]

Now, there have been 4 versions of YOLO developed already.

- YOLOv1 in 2015 by Joseph Redmon et al
- YOLOv2 in 2016 by Joseph Redmon et al

- YOLOv3 in 2017 by Joseph Redmon et al
- YOLOv4 in 2020 by Alexey Bochkovskiy et al

We will be looking into YOLOv4 for the purpose of our robot to use because it is the latest model obviously. There is a YOLOv5 but there has not been enough testing for it to be accepted by the community yet making it more controversial for being a true upgrade from version 4.

The main benefit to YOLOv4 is that it beats the existing methods by a large margin in areas of detection performance and speed. When designing YOLOv4 the team really tried to put an emphasis on optimizing neural networks detectors for parallel computations.

This is how the architecture is broken down on a high level.

- Backbone: CSPDarknet53
- Neck: Spatial Pyramid Pooling additional module, PANet path-aggregation
- HEAD: YOLOv3

So to go through this list. A backbone is usually pre trained on ImageNet so that it is used as a feature extractor which gives you a feature map representation of the input.

ImageNet is an image database according to the WordNet hierarchy (only nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images

CSPDarknet53 is a unique backbone that augments the learning capacity of CNN, the spatial pyramid pooling section is attached overhead CSPDarknet53 for improving the receptive field and distinguishing the highly important context features.

The PANet is then deployed by the method for parameter aggregation for distinctive detector levels.

Now one might be thinking, how is YOLOv4 better than any of the previous YOLOs?

- A huge focus from the developers was that this algorithm could be ran on a 1080 Ti or 2080 Ti GPU to train fast and accurate object detectors
- The methods covering CNB (Cross-iteration batch normalization, PAN (Path aggregation network), that are greater skilled and applicable for single GPU training

- The consequences of Bag-of-Freebies and Bag-of-Specials object detection procedures all the while detector training was confirmed

Just for that last list to make more sense, a Bag-of-Freebies is a set of methods that only change the training strategy or only increase the training cost. A Bag-of-Specials essentially is getting something of value discounted. So in the case of YOLOv4, this would be only increasing the inference cost by a small amount but significantly improving the accuracy because of that. So basically barely affecting runtime while drastically improving accuracy.

Just to show YOLOv4 compared to other models here is a graph of YOLOv4 over the Microsoft COCO dataset compared to other models.

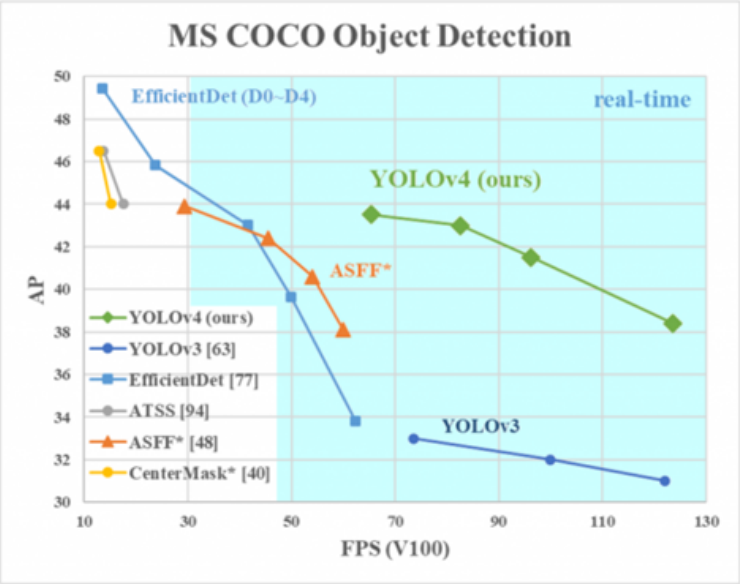


Figure: Trade off graph of multiple object detectors [20]

Looking strictly at the FPS axis YOLOv4 is double the speed of EfficientDet, Also when looking at YOLOv3 the AP has increased by around 10% and the FPS has increased by about 12%.

Now, this paper previously mentioned the bag of freebies and bag of specials but now lets list what they contain in YOLOv4.

- Bag of Freebies for backbone
    - CutMix/Mosaic data augmentation

    - DropBlock regularization
    - Class label smoothing
- Bag of Freebies for detector
    - CIoU-loss
    - CmBN
    - DropBlock regularization
    - Mosaic data augmentation
    - Self-Adversarial Training
    - Eliminate grid sensitivity
    - Using multiple anchors for single ground trust
    - Cosine annealing scheduler
    - Optimal hyper-parameters
    - Random training shapes
- Bag of Specials for backbone
    - Mish activation
    - Cross-stage partial connections (CSP)
    - Multi-input weighted residual connections (MiWRC)
- Bag of Specials for detector
    - Mish activation
    - SPP-block
    - SAM-block
    - PAN path aggregation block
    - DIoU-NMS

In conclusion, the main takeaway from YOLOv4 is that the speed and accuracy ratio is almost unbeatable and can be trained on conventional GPUs which will make it a great algorithm to use for our robot under simulation on our laptops for target detection.

Google Collab Setup Instructions For Yolov4

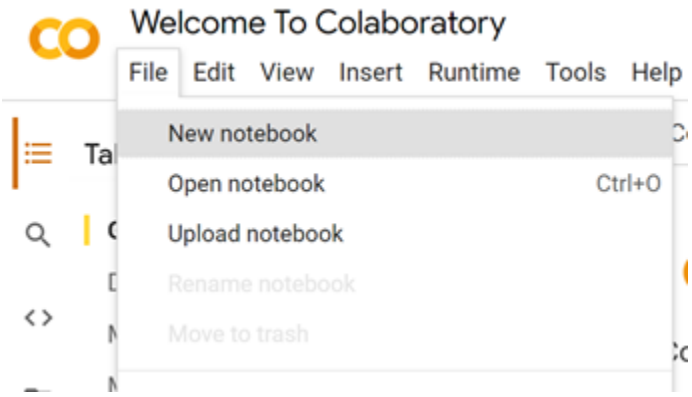
So one of the main reasons YOLOv3 was chosen as the object detection method was because of how its intent was to be able to run on a standard GPU. We will be looking into google colab for implementing/training our YOLO neural net. Here are the steps to get started

Step 1: Use existing Gmail account or create one

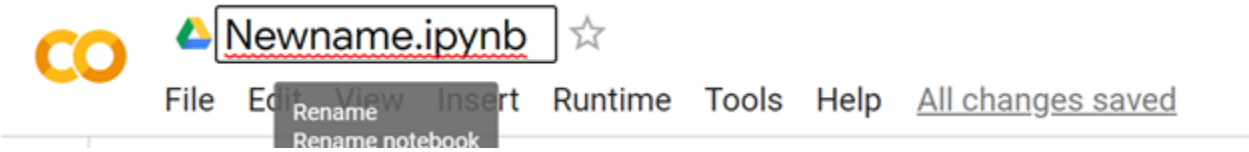
- A Gmail account is required to run code on google colab
- Login into google colab through this link https://colab.research.google.com
- If you do not have a Gmail account please set one up through this link https://accounts.google.com/signup?hl=en
- Once you login there will be a “Welcome To Colaboratory” file to read through the basics

Step 2: Create a new notebook to use

- Go to file then select create new notebook



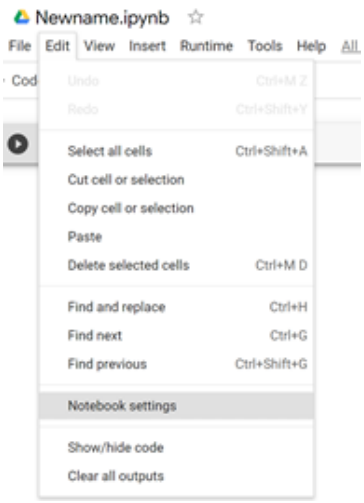
- Now rename the notebook (at the top) to whatever you like



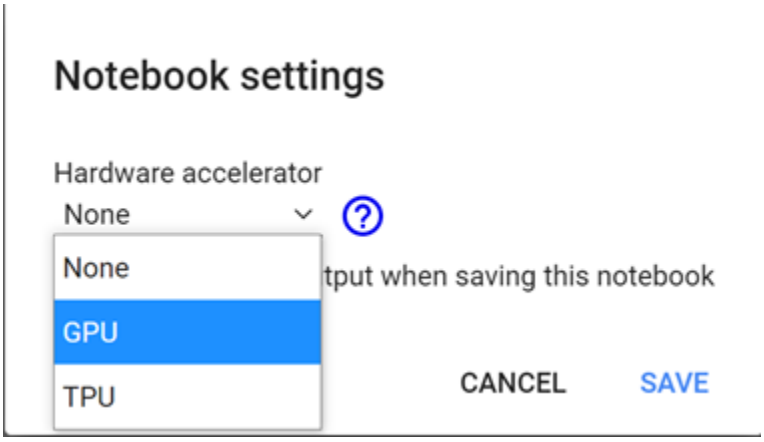
Step 3: Enable your GPU in the notebook

This step is really important because we are trying to reduce the training runtime as much as possible.

- Click on edit Notebook settings



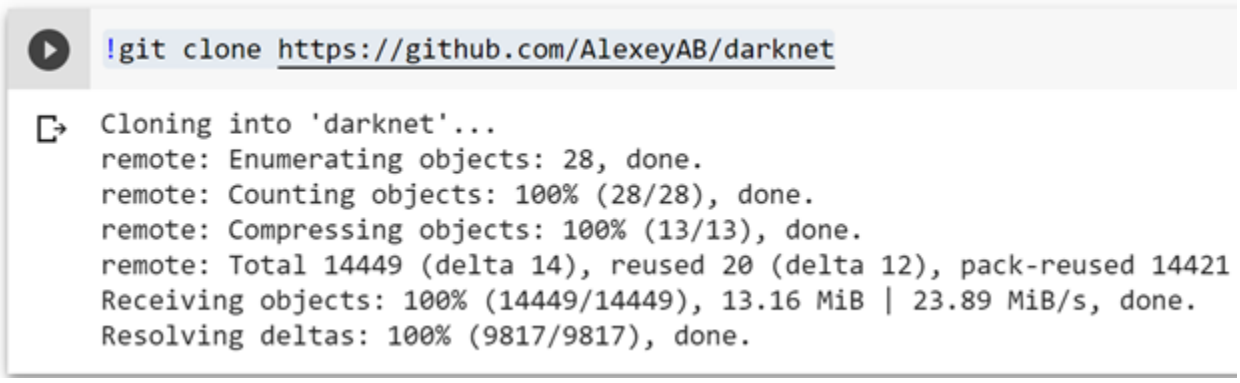
- Then select your GPU from the dropdown



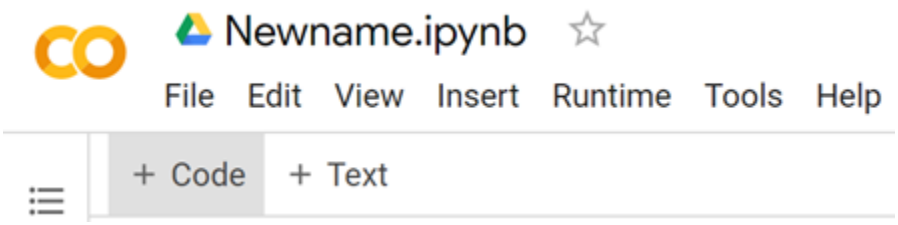
Step 4: Cloning and building Darknet

- Type this in the run cell then press run

```
!git clone https://github.com/AlexeyAB/darknet
```



- Now add a new cell for the next bit of code we will add by clicking on the +Code



- Copy/paste these commands and run them in the cell to allow Makefile to have the GPU and OpenCV enabled

```
cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
```

```
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
```

```
/content/darknet
```

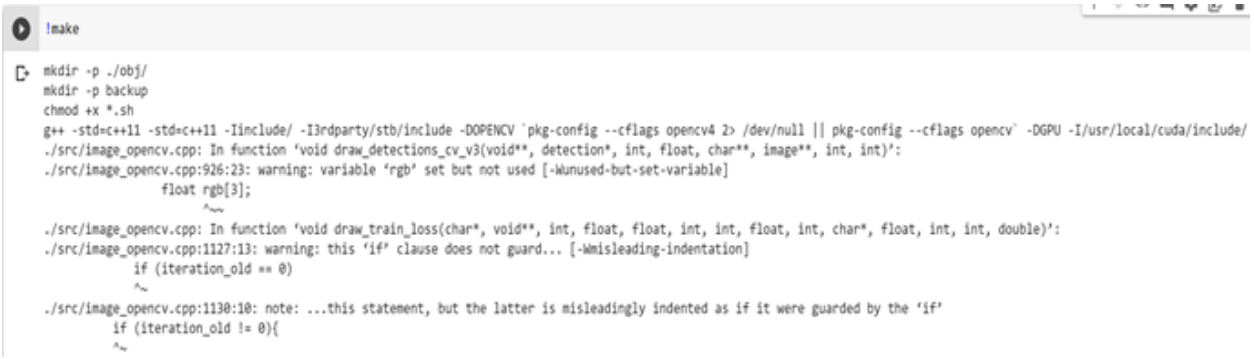
- Verify Cuda by copying the following code into a new cell

```
!/usr/local/cuda/bin/nvcc -version
```

```
!/usr/local/cuda/bin/nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Sun_Jul_28_19:07:16_PDT_2019
Cuda compilation tools, release 10.1, V10.1.243
```

- Next, in a new cell run the !make command to start building darknet. The beginning should look like this



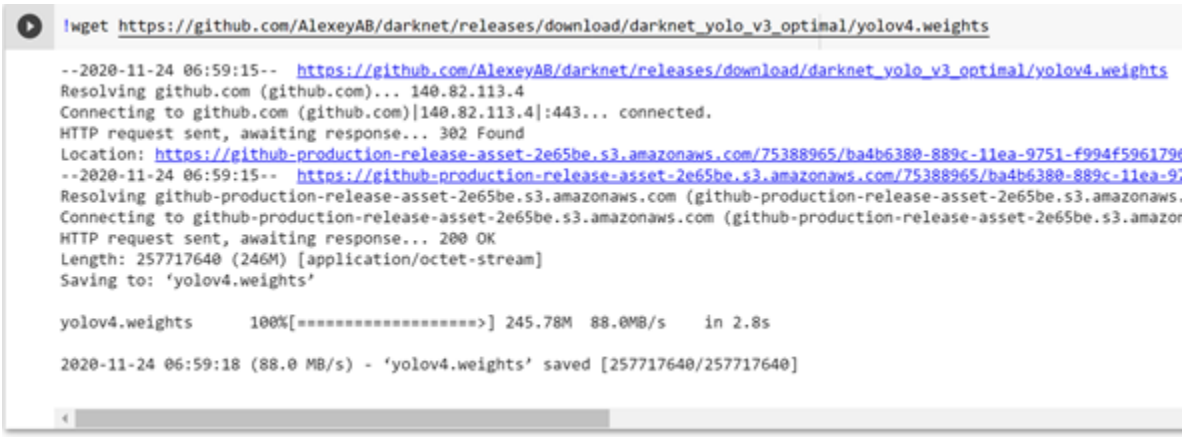
```
!make

mkdir -p ./obj/
mkdir -p backup
chmod +x *.sh
g++ -std=c++11 -std=c++11 -Iinclude/ -I3rdparty/stb/include -DOPENCV `pkg-config --cflags opencv4 2> /dev/null || pkg-config --cflags opencv` -DGPU -I/usr/local/cuda/include/
./src/image_opencv.cpp: In function 'void draw_detections_cv_v3(void**, detection*, int, float, char**, image**, int, int)':
./src/image_opencv.cpp:926:23: warning: variable 'rgb' set but not used [-Wunused-but-set-variable]
                float rgb[3];
                      ^~~
./src/image_opencv.cpp: In function 'void draw_train_loss(char*, void**, int, float, float, int, int, float, int, char*, float, int, int, double)':
./src/image_opencv.cpp:1127:13: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
            if (iteration_old == 0)
            ^~
./src/image_opencv.cpp:1130:10: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'if'
         if (iteration_old != 0){
         ^~
```

Step 5: Download pre-trained YOLOv3 weights

- This Yolo has already been trained with 80 classes we can predict
- These 80 classes come from the COCO dataset, which is a popular dataset used for testing
- Create a new cell and run the command

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov3.weight
```



Step 6: Run detections

- Create standard function to upload images

```
[4]  def upload():
       from google.colab import files
       uploaded = files.upload()
       for name, data in uploaded.items():
         with open(name, 'wb') as f:
           f.write(data)
           print ('saved file', name)
```

```
upload()
```

- Select the image you wish to test to make sure the model is working

```
upload()
```

-

- Make sure the photo has been uploaded by showing it

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('YOLO.jpg')
imgplot = plt.imshow(img)
plt.axis("off")
plt.show()
```

- Next is the actual step of running the yolo model on the test picture. This is the command format
-

"!./darknet detector test <path to .data file> <path to config> <path to weights> <path to image>"

```
!./darknet detector test cfg/coco.data cfg/yolov4.cfg yolov4.weights YOLO.jpg
```

```
 CUDA-version: 10010 (10010), cuDNN: 7.6.5, CUDNN_HALF=1, GPU count: 1
 CUDNN_HALF=1
 OpenCV version: 3.2.0
 0 : compute_capability = 750, cudnn_half = 1, GPU: Tesla T4
net.optimized_memory = 0
mini_batch = 1, batch = 8, time_steps = 1, train = 0
   layer   filters  size/strd(dil)      input                output
   0 conv     32       3 x 3/ 1    608 x 608 x   3 ->  608 x 608 x  32 0.639 BF
   1 conv     64       3 x 3/ 2    608 x 608 x  32 ->  304 x 304 x  64 3.407 BF
   2 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   3 route  1                                     ->  304 x 304 x  64
   4 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
```

-

- Now print predictions stored in the predictions.jpg file from darknet to see if the pretrained model is working

Using YOLOv3 with custom data

Step 1: Gathering/Labeling Dataset

- There are two ways to do this. Either grab some pre-prepped data from Google or manually grab and label the data collected. We will be doing the latter since there are no pre-prepped BB8 datasets.
- We have already discussed on how to gather and label the dataset previously so make sure to follow those steps to have the dataset ready

Step 2: Move dataset to colab

- Copy over both datasets (obj.zip and test.zip) into the root directory of colab.

```
!cp /mydrive/yolov3/obj.zip ../
!cp /mydrive/yolov3/test.zip ../
```

- Unzip both zip files so that they are now in the /darknet/data folder

```
!unzip ../obj.zip -d data/
!unzip ../test.zip -d data/
```

Step 3: Setting up config file

- Next go to content/darknet/cfg/yolov4-custom.cfg

- We want to now upload this file to our drive to be able to make the edits we want

```
!cp cfg/yolov3-custom.cfg /mydrive/yolov3/yolov3-custom.cfg
```

- Now we can go to and edit the file making the changes we want such as batch size, subdivisions, epochs, etc. This will be expanded upon later during testing.

- Now upload the file back when changes have been made

```
!cp /mydrive/yolov3/yolov3-custom.cfg ./cfg
```

- Now create a new file called obj.names where for **EACH** line there will be **ONE** class name in the **SAME ORDER** as the classes.txt file from the data generation step. Also make sure you have no spaces in your class names
- Next, create a new file called obj.data file and fill it in with the correct data, in our case we would have one class that is ‘bb8 that we would detect. This file is where we save the weights of our model while we train it. Make sure to create a backup folder in google drive

```
classes = 1
train = data/train.txt
valid = data/test.txt
names = data/obj.names
backup = /mydrive/yolov4/backup
```

- Now upload both files back to your Yolo directory

```
!cp /mydrive/yolov3/obj.names ./data
!cp /mydrive/yolov3/obj.data ./data
```

- The last part of step 3 will be to add our train.txt and test.txt files that hold the paths to all the training images and validation images. This can either be done

by a python script of manually adding each images path to the train.txt and test.txt

Step 4: Download pre-trained weights

This next step is important because it is not NECESSARY, but it will definitely increase the accuracy of the model while decreasing the training time. Training with pretrained weights is a common practice for these reasons.

- First run this command:

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137
```

Figure: Expected output.

Step 5: Train your custom model

- Now it is time to train the model that has been prepped. This will take hours potentially so either run it while working on other projects on the computer or find an idle script to paste into your javascript component in inspector view on your browser since colab kicks idle users off after roughly 90 mins
- Run the following command filling in the necessary parts to train

```
!./darknet detector train <path to obj.data> <path to custom config> yolov4.conv.137 -dont_show -map
```

- So in our case it would be:

```
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg yolov4.conv.137 -dont_show -map
```

- After the training we can print the average loss vs iterations. Loss should be under 2 to be considered accurate

```
imShow('chart.png')
```

- A helpful side note would be that if the idle script for some reason does not work and colab gets disconnected you can rerun from your last save point in the backup folder with the most recent saved train weights

```
!./darknet detector train data/obj.data cfg/yolov3-obj.cfg /mydrive/yolov3/backup/yolov3-obj_last.weights -dont_show
```

Step 6: Getting more statistics out of your model

- Run the following command on the saved weights file of your training to get more statistics such as mean average precision, average IoU, detection time, etc

```
!./darknet detector map data/obj.data cfg/yolov3-obj.cfg
/mydrive/yolov4/backup/yolov3-obj_1000.weights
```

Step 7: Finally running the custom trained model

- First step would be to reconfigure the cfg file for testing by changing batch size and subdivisions, but we will get into the specifics on that later.
- Once that is done, upload an image you want to test your yolo model on and then run this command. Do not forget to set the threshold value to whatever is deemed best

```
!./darknet detector test data/obj.dat cfg/yolov4-obj.cfg
/mydrive/yolov3/backup/yolov3-obj_last.weights /mydrive/images/car2.jpg -thresh
0.3 imShow('predictions.jpg')
```

Step 8: Running the custom model on a video

- This step actually is one of the easier ones, first upload the video to test with the newly trained model
- Run this command to start detecting and saving the video in the google drive as well

```
!./darknet detector demo cfg/obj.data cfg/yolov3-obj.cfg
mydrive/yolov3/backup/yolov3-obj_last.weights -dont_show
/mydrive/videos/test.mp4 -i 0 -out_filename /mydrive/videos/results.avi
```

Running Yolo in ROS

Step 1: Launch Linux VM

- To run yolo with ROS we need to use Linux so first step is to open the VM

Step 2: Install CUDA

CUDA is a library that allows programs to take advantage of the computer in use's GPU. We do not need CUDA to run yolo, but it will drastically increase the speed with CUDA

- All we need to do to install CUDA is run this command

```
sudo apt install nvidia-cuda-toolkit
```

Step 3: Clone Darknet/YOLO

- First we need to change directories to inside gazebos source
- Then we need to run the command to clone the Darknet repo

```
cd ~/catkin_ws/src
```

```
git clone -recursive https://github.com/leggedrobotics/darknet_ros.git
```

Step 4: Build Darknet

- Now it is time to build darknet by running the following command (this is only for first time, next time just do catkin build)

```
catkin build -DCMAKE_BUILD_TYPE=Release
```

- On version 18.04 this command may have to be ran instead

```
Catkin build -DCMAKE_BUILD_TYPE=Release
-DCMAKE_C_COMPILER=/usr/bin/gcc-6
```

Step 5: Configure Darknet/YOLO

Now since we are doing custom object detection the prebuilt YOLO's will not have the class of turtlebot and the custom cfg/weights we made with colab so we need to paste those in from our repo

- First let us copy our obj.cfg into darknet under the relative path:

```
your_path/darknet_ros/darknet_ros/yolo_network_config/cfg
```

- Now copy the desired weights to this path:

```
your_path/darknet_ros/darknet_ros/yolo_network_config/weights
```

- Next, we have to create a new YAML file to use since once again this is custom object detection
- This is what the prebuilt yolo YAML looks like

```
yolo_model:

  config_file:
    name: yolov2-tiny.cfg
  weight_file:
    name: yolov2-tiny.weights
  threshold:
    value: 0.3
  detection_classes:
    names:
      - person
      - bicycle
      - car
      - motorbike
      - aeroplane
      - bus
      - train
      - truck
```

- We would change the config file and weight file to the custom ones that we uploaded
- We would also change the detection_classes to just 1 class of turtle bot
- Once done editing changes make sure to modify the line in darknet_ros.launch with this:

Step 6: Begin training

- Now let us begin the training by running this command

```
./darknet detector train cfg/obj.data cfg/obj.cfg darknet19_448.conv.23
```

- This could take a couple hours/days but remember that backups are only created after every 100 and 1000 iterations so make sure to restart training after it stops until satisfied
- Run this command to start where it left off in training

```
./darknet detector train cfg/obj.data cfg/obj.data cfg/obj.cfg obj_1000.weights
```

Step 7: Using Darknet/YOLO

- In the yaml file under darknet_ros/darknet_ros/config, change the image topic from /camera/rgb/image_raw to this

```
/webcam/image_raw
```

- The file darket_ros.launch will launch the darknet/yolo ros node. This file is found under darket_ros/darknet_ros/launch
- In that file choose which version of yolo you trained with. Pick the custom yaml that was created previously

```
<arg name=”network_param_file”
default=”$(find darknet_ros)/config/yolov2-tiny.yaml”/>
```

Now launch ROS to begin testing with the robot.

Why we chose YOLOv3

Even though YOLOv4 is superior in most ways to YOLOv3, we ended up using v3 instead. This reason was mainly due to the availability of resources for implementing YOLOv3 with ros and gazebo. In the future, once YOLOv4 has more active ros repo implementations, it will be the obvious choice. Even with this set back, YOLOv3 still outperforms many other object detections algorithms and gives us an extremely good fps to precision ratio. Averaged around 30 fps with a mAP score of 99.98%.

5. Explicit Design Summary

The core design idea is to use a document to informally describe the design of the software and hardware. Below is a detailed design section describing details of how the different parts of the project are connected to each other.

Simulated Hardware Block Diagrams

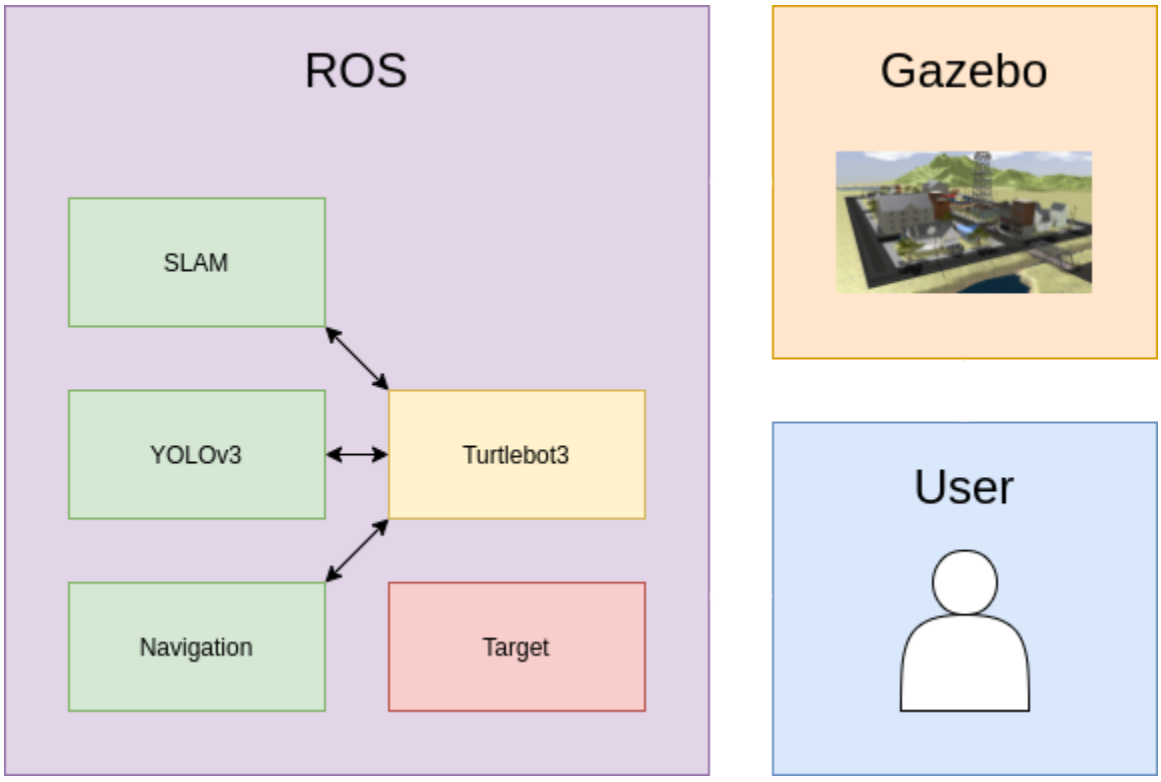


Figure: The user interacts with Gazebo and ROS, getting and setting information. ROS sends the simulation information to Gazebo for the user to view and simulates the actual robots and code running on them.

Software Block Diagrams

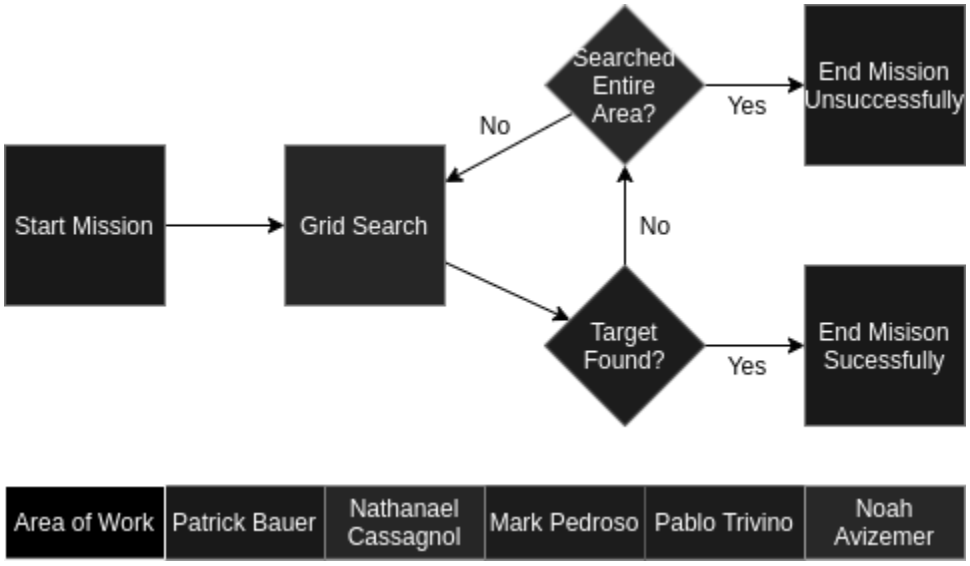


Figure: A High-level diagram of the stages the mission progresses through.

Navigation

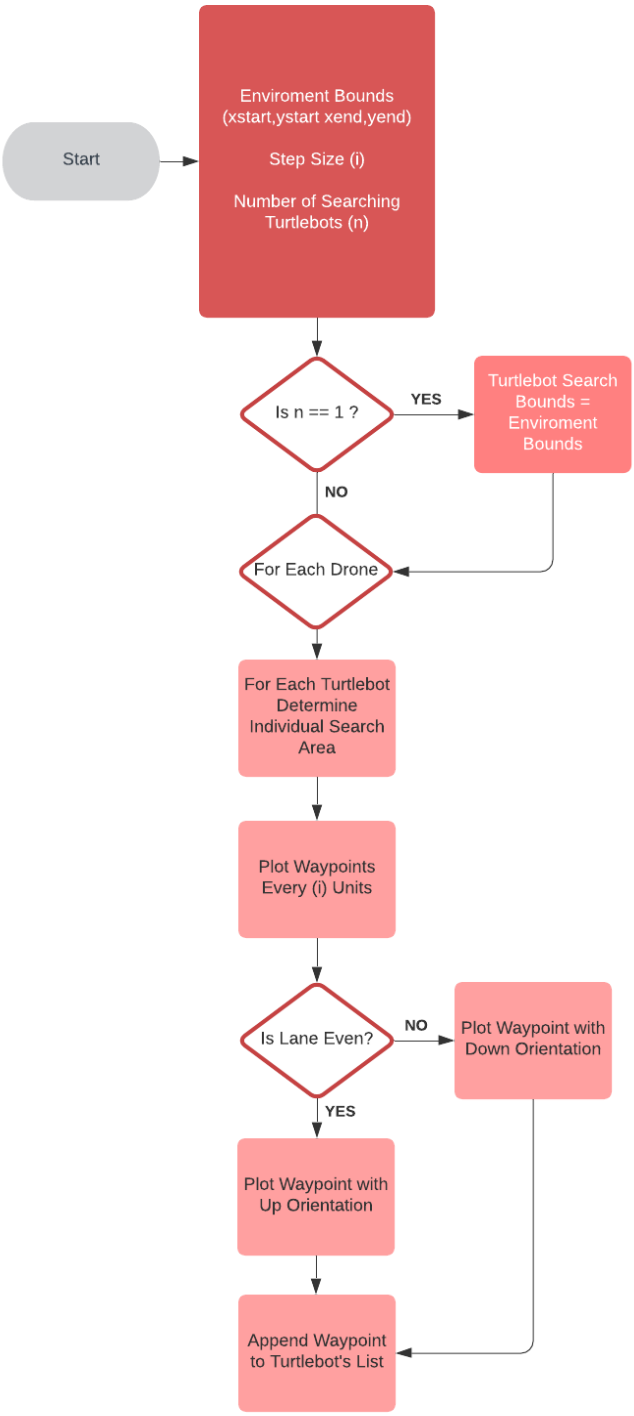


Figure: The Block Diagram for the Waypoint Creation Algorithm

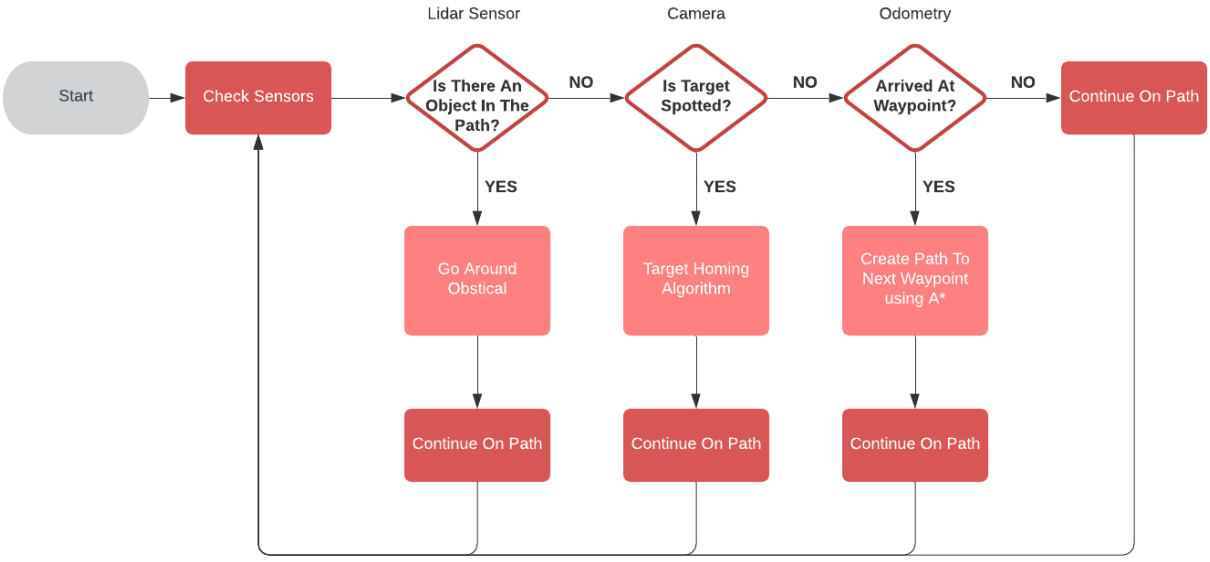


Figure: A High-Level diagram of the navigation workcycle

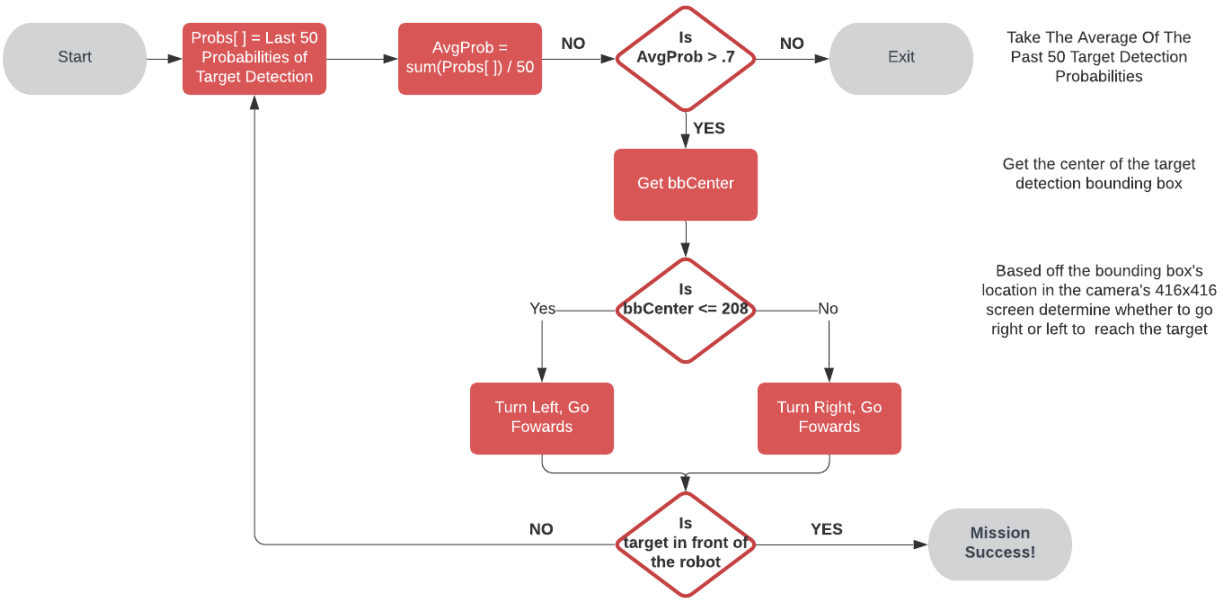


Figure: A Block Diagram of the Target Homing Algorithm

Entity Relationship Diagrams

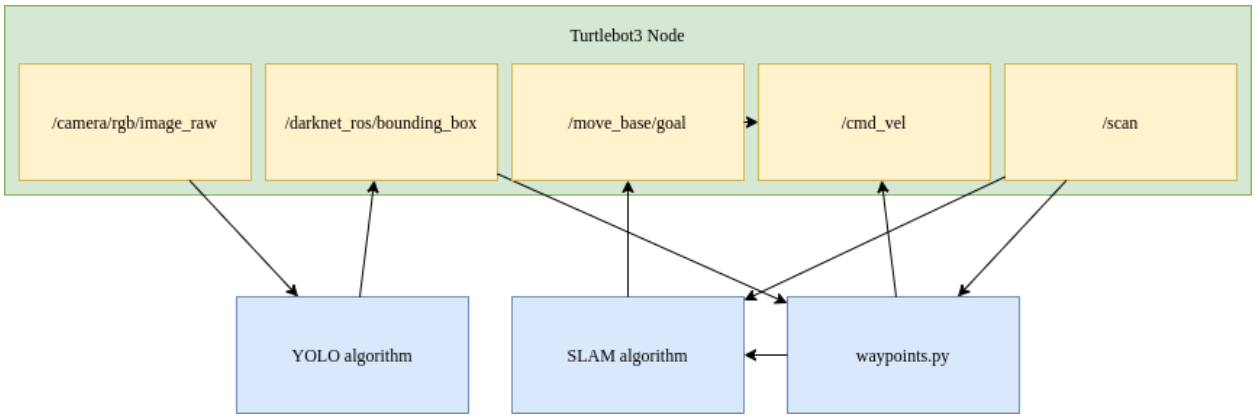


Figure: An entity-relationship diagram that shows the software interacting with the ROS node turtlebot3 node and topics.

6. Detailed Design Content

Simulation Environment

The entirety of this project is carried out as a simulation. It is necessary to have a simulation environment that consists of an urban layout with the target autonomously navigating around it; our robot must be able to navigate this environment to find the target and eliminate it within certain constraints provided to us. Simulators allow for the modeling of as much as or as little of reality as desired. Simulated sensors are able to be ideal devices, or can be simulated to include levels of error or other faults. Also, automated testing of algorithms requires simulated robots, since during the training process the algorithms need to be able to understand the consequences of their actions.Though our project does not require us to physically create the robot and have it work in real life, simulations like these are an important beginning step to that process. If in the future this project will be taken up by another team that must create the physical robot, our testing in the simulation must show that the robot is able to navigate and carry out the necessary tasks without crashing. The cost of crashing a real robot is high, especially for small budgeted senior design teams.

Command and Control Graph

Processor Node - This node is published and subscribed to a local communication topic. It receives information about the robot's status and whether a target was found. It then makes a decision based on our algorithm or allows for user input on what to do next. The processor node is also published and subscribed to the input/output data topic. This is where data is sent to and received from the user.

User Interface Node - This node represents the user interface of the Command and Control laptop. The user sees information on a screen and has access to controls to provide input to control the robot's actions. It is published and subscribed to the Input/Output Data topic.

Communication Antenna Node- This node represents an antenna on the Command and Control laptop. It is published and subscribed to the local communication topic as

well as the long range communication topic. Information about the robot's status is received from the long range communication topic and is sent to the processor through the local communication topic. Instructions for the robot are received from the local communication topic and sent out to the robot via the long range communication topic.

Robot Graph

Communication Antenna Node - This node represents an antenna on the robot. It is published and subscribed to the robot data topic as well as the long range communication topic. Information about the robot's status is received from the robot data topic and sent out to the control via the long range communication topic. Instructions for the robot are received from the long range communication topic and passed on to the robot's processor via the robot data topic.

Processor Node - This node represents an onboard processor on the robot. It is published and subscribed to the robot data topic. It sends information about its status and receives instructions on what to do. It is published to the movement instructions topic. After processing instructions or following the pathfinding algorithm, it sends information on what direction to move. The processor node is subscribed to the video feed topic. It receives data from the onboard camera. It processes the data and searches for a specified target and makes a decision based on the input. Lastly, the processor node is subscribed to the distance feed topic. It receives distance data that is processed to determine whether the robot is close to colliding with an object.

Navigate Node - This node represents the movement functionality of the robot. It is subscribed to the movement instructions topic. It receives information on what direction to move. With this information, the robot will adjust in speed to move in the specified direction.

Camera Node - This node represents the forward facing camera on the robot. It is published to the video feed topic and sends a continuous live feed to the processor.

LIDAR Node - This node represents a forward facing LIDAR sensor on the robot. It is published to the distance feed topic and sends a continuous stream of distance data collected to the processor.

7. Software and Tools Used

Gazebo

Our simulator of choice is Gazebo. This software is open-sourced, which is important for us since we are working with a zero dollar budget. It is also widely used in the industry and recommended to us by our sponsor at Lockheed Martin. Gazebo is able to simulate robotic vehicles in various 3D environments. We are able to build an environment that we wish to test the robot swarm in and accurately simulate the robot's actions. Gazebo includes a physics engine to apply real forces to the simulations and supports various sensors and controls for the robot that are necessary for the simulation. Another important reason Gazebo was chosen is its easy integration with ROS (Robotic Operating System), the interface for the robots in the simulation.

Gazebo is a robot simulation tool that enables developers to rapidly test algorithms, design autonomous robots, and train AI systems using realistic scenarios. It also offers the capability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Provided with a robust physics engine, high-quality graphics, and user-friendly graphical interfaces, as a team, we decided that Gazebo was the best open-source robot simulator on the market. The integration that Gazebo offers with ROS was also a major factor in our decision.

ROS (Robotic Operating System)

ROS is a robotics middleware that is currently maintained by the Open Source Robotics Foundation. It consists of the infrastructure, tools, and capabilities for programming robots. It has a large community that makes it easy to share your own work and try other people's work through ROS packages. ROS is also open-sourced, making it perfect for those working on a limited budget.

The Robot Operating System (ROS) is a framework for creating robot software. It is a collection of libraries and tools that simplifies the creation of complex robotic systems across a wide variety of robot platforms. The main features of ROS include infrastructure, tools, capabilities, and ecosystem.

Advantages	Disadvantages
Provides lots of infrastructure, tools, and capabilities Easy to implement packages and algorithms Large community Free, open-source, BSD license.	Approaching maturity, but still changing, lots of APIs change. Security and scalability are not primary concerns. The only OS supported is Ubuntu Linux. (It can be installed on other operating systems but may not work correctly)

Robot Communication Within ROS.

In the real world, robots use physical components to scan and interact with the environment. For example, a camera can be used to get a picture or a live feed of the surrounding areas and an antenna to send radio waves with collected data and other useful information. Since we are running a simulation in Gazebo and not using physical hardware that produces radio waves to send information, we need to simulate the communication between the robot and the command and control terminal another way. ROS is really useful because it provides a framework for this simulated communication: nodes and topics.

ROS Nodes

A node is a process that performs computation. Nodes can be combined into a graph and they communicate with each other by streaming topics. A robot, such as a turtlebot in our case, is a collection of nodes where each node represents one component of the robot. One node controls the LIDAR, one node controls the video input, one node controls the pathfinding, and one node represents the processor.

There are many benefits to using nodes in ROS. Instead of creating a monolithic system, nodes take the divide and conquer approach. From an overall system view, this is great because if one node crashes, the other nodes can continue to operate. This lets each of the team members to work on their component of the project without Implementation details are also hidden and the nodes only expose an API to the rest of the system (graph). This allows for other implementations of the node to be interchanged as well as using different programming languages.

ROS Nodes in Python

ROS nodes refer to executables that are connected to the ROS network. For this project, the language of choice to write our own ROS nodes is Python. To write a ROS node in Python, every file must begin with the following:

```
#!/usr/bin/env python
```

This is to ensure that the script is executed as a Python script. Following this, the imports for the libraries that are used may be included. For all ROS nodes, the rospy library must be imported:

```
Import rospy
```

Rospy is a Python client library for ROS. The rospy client allows users to interface with ROS topics, services, and parameters. The goal of rospy is to favor implementation speed over runtime performance; this allows for algorithms to be quickly prototyped and tested in ROS.

ROS Topics

ROS Topics are named buses over which nodes exchange messages. Topics have publish/subscribe functionality which allows for the separation of information creation and consumption. Nodes do not need to know which other nodes they are communicating with. Instead, they just subscribe to the relevant topic when they need data and publish to the relevant topic when they produce data. Topics allow for unidirectional, streaming communication.

In our project, we used topics to allow for our robot to communicate with other nodes and the user. We will also use nodes and topics to allow for information to be passed between the individual components of a robot. A common data frame will be created for information to be published/subscribed for the local robot topic as well as the long range communication topic.

ROS Messages

All data that is published by a node and received by other nodes is wrapped in a ROS message format. ROS messages are defined by the type of the message and the data format. For example, the ROS package named std_msgs has the type String which consists of characters. Other packages allow for the sending of pictures, robot navigation, robot sensors, and other data types.

ROS Namespaces

All resources in the ROS Computation Graph, such as: nodes, parameters, topics and services, are provided with a hierarchical naming structure by Graph Resource Names. These names are useful and critical in larger and more complicated systems composed in ROS; therefore, it is important to understand how these work.

Some examples of possible names in ROS include:

- / (the global namespace)
- ucf/robot/name
- /wg/node1

These names are important in providing encapsulation. In ROS, every resource is defined in a namespace; this may be shared with many other resources. Resources are able to create resources in their own namespace and access resources within or above their own namespace. Generally, connections between resources in distinct namespaces are achieved by integration code above each namespace. The encapsulation provided by namespaces is crucial in isolating different portions of the system; this means that resources in their own namespace are not grabbing the wrong resources or interacting with resources that are outside of its namespace without explicitly being set up to do so.

Resources do not need to be aware of what namespace they are located in; names are resolved relatively. When programming, nodes that are part of the same system and work together can be written as if they are located in the top level namespace. When these nodes are integrated into the system at large, they can be pushed down into a namespace that defines processes of these nodes. To create a valid name, they must meet the following criteria:

- The first character is an alpha character (A-Z, a-z), tilde(~), or forward slash (/)
- Subsequent characters are alphanumeric, underscores, or forward slashes
- Base names cannot have forward slashes or tildes in them

Resolving Names

In ROS, there are four types of Graph Resource Names: base, relative, global, and private. These have the following syntax:

- base
- relative/name
- /global/name

- ~private/name

Names resolution is done relative to the node's namespace by default. For example, the node /robot/node1 has the namespace /robot; therefore, the name node2 will resolve to /robot/node2. Base names are defined as names with no namespace qualifiers. These are a subclass of relative names and follow the same resolution rules as them. Base names are most often used when initializing the node name. If a name starts with "/", then it is a global name. Global names are fully resolved; however, they limit code portability since it cannot be pushed down to a lower namespace. If a name starts with "~", then it is private; this converts the node's name into a namespace. Passing parameters to a specific node are easily done using private names.

Examples of name resolution:

Node	**Relative**	**Global**	**Private**
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar -> /wg/node3/foo/bar

Below is a screenshot of the namespaces for the topics in the turtlebot portion of the project:

```
noah@noah-VirtualBox:~/catkin_ws$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/robot1/cmd_vel
/robot1/imu
/robot1/joint_states
/robot1/odom
/robot1/scan
/robot2/cmd_vel
/robot2/imu
/robot2/joint_states
/robot2/odom
/robot2/scan
/robot3/cmd_vel
/robot3/imu
/robot3/joint_states
/robot3/odom
/robot3/scan
/robot4/cmd_vel
/robot4/imu
/robot4/joint_states
/robot4/odom
/robot4/scan
/rosout
/rosout_agg
/tf
```

Figure: ROS topic list

As seen in the screenshot, there are namespaces for each robot; these are called /robot1, /robot2. Each of these namespaces correspond to a single robot. Since the

robot needs to navigate around independently, the scanner topics and the velocity topics must fall under the namespace for each individual robot. This allows us to set the velocity for the robots individually and obtain sensor data from the robots individually. This topic list also includes global namespaces such as /clock. We want every resource to access the same clock and be on the same time; therefore, the clock should be a global namespace.

ROS Workspace

The official build system of ROS is called Catkin. All packages that are built using ROS are built in a catkin workspace. A build system generates targets from raw source code that is able to be used by an end user. The targets that are generated can be in multiple forms, including: executables, generated scripts, exported interfaces, or any non static code. The source code is organized into packages; these packages consist of at least one target when built.

ROS is composed of a large collection of independent packages that depend on each other, use various different programming languages, tools, and other conventions. This leads to the build process targets to be completely different for another target. Catkin attempts to make building and running ROS code easier by simplifying the process with certain tools and conventions. Overall, building packages is a huge part of utilizing ROS.

Inside of the package for this project, there is the following directory structure:

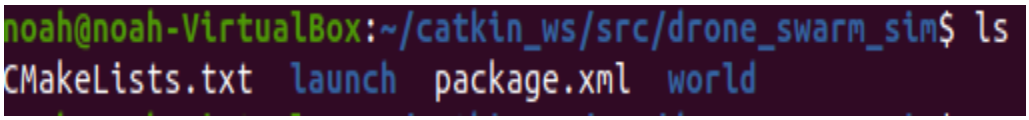


Figure: Package directory structure

As seen in the figure above, our package is broken down into two subdirectories: launch, and world. Splitting up these files allows for greater organization in building a large project like this. The launch directory contains all the launch files required to launch the world and spawn all the models in it. These launch files also can launch executables that will automatically run when the launch is activated. There also is a directory for worlds. This “world” directory includes the world file that is used for this project.

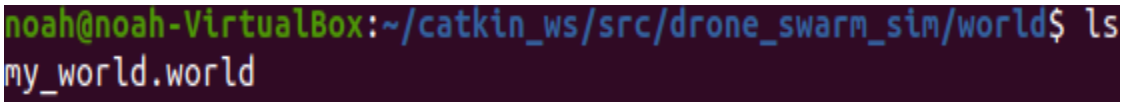


Figure: Screenshot of world directory

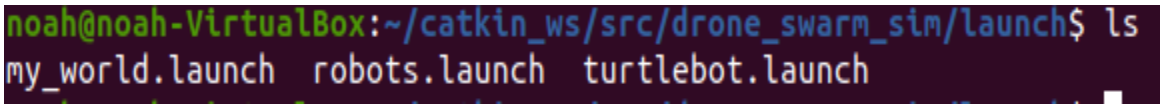


Figure: Screenshot of launch directory

The image below shows a portion of the contents of the world file. World files in Gazebo are formatted as XML files. These world files contain descriptions of the world to be simulated by Gazebo, this includes the layout of robots, sensors, objects, etc.

```
noah@noah-VirtualBox:~/catkin_ws/src/drone_swarm_sim/world$ cat my_world.world
<sdf version='1.7'>
  <world name='default'>
    <scene>
      <ambient>0.4 0.4 0.4 1</ambient>
      <background>0.7 0.7 0.7 1</background>
      <shadows>1</shadows>
    </scene>
    <light name='sun' type='directional'>
      <cast_shadows>1</cast_shadows>
      <pose>0 0 10 0 -0 0</pose>
      <diffuse>0.8 0.8 0.8 1</diffuse>
      <specular>0.2 0.2 0.2 1</specular>
      <attenuation>
        <range>1000</range>
        <constant>0.9</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <direction>-0.5 0.1 -0.9</direction>
      <spot>
        <inner_angle>0</inner_angle>
        <outer_angle>0</outer_angle>
        <falloff>0</falloff>
      </spot>
    </light>
    <model name='ground_plane'>
      <static>1</static>
      <link name='link'>
        <collision name='collision'>
```

Figure: Screenshot of portion of world file

ROS Launch Files

Launch files in ROS are a way to easily start a master and multiple nodes. Along with this, parameters may be set and other initialization requirements may be met. To run launch files in ROS, one must use the **roslaunch** command. If no master is set, roslaunch will also start roscore. When using the **roslaunch** command, the package that the launch files may be specified, or the file path to the launch file is specified. The syntax to run the **roslaunch** command is as follows:

```
roslaunch package_name launch_file
roslaunch ~/.../.../launch_file
```

Launch files are formatted using a certain XML format; these files have a .launch extension. The common convention for organization is to place the launch files in a “Launch” directory in the workspace; however, launch files may still be placed anywhere inside of a package directory. Launch files must have their contains contained within a pair of launch tags, like so:

```
<launch>
(content of launch file)
</launch>
```

To start a node in a launch file, <node> tags are used. The package, type, and name are arguments that must be set when using <node> tags. The package, or pkg, argument refers to the package that is associated with the node that will be launched. The type argument refers to a node executable file (in our case, the Python executables). The name argument assigns a name to the node that is being launched; overwriting the name of the node with the name argument is possible, since the name given in this argument takes priority over the name assigned in the code. There are also a few other optional parameters that are useful for certain scenarios. A few examples of commonly used optional parameters are:

- **Respawn/Required**
    - Even though these arguments are not required, it is common practice to have one or the other set. If required is set to true (Required=true), then all the nodes associated with a launch file will be shut down if this particular node goes down. If the respawn argument is set to true (Respawn=true), then the node will be restarted if closed for any reason.
- **ns**
    - The ns argument is used to launch a node inside of a namespace. Setting this parameter is most useful when launching multiple instances of the same node.
- **arg**
    - The arg argument is used to set local variables inside of the launch file. This tag is used in the following way: <arg name=”...” value=”...”/>

The syntax for <node> tags is as follows:

```
<node pkg="..." type="..." name="..."/>
```

For organizational purposes, this project is split into multiple launch files. This includes files called: “start.launch”, “start_bb8.launch”, “turtlebot3_navigation.launch”, and lastly “yolo_v3_custom.launch”. The contents and explanations for each launch files is shown below:

```
<?xml version="1.0"?>

<launch>
    <!-- Start World -->
    <include file="$(find autonomous_turtlebot3_swarm)/launch/start_world.launch">
        <arg name="world_name" value="$(find autonomous_turtlebot3_swarm)/worlds/urban.world"/>
    </include>

    <!-- Start Searching Turtlebot -->
    <param name="robot_description" command="$(find xacro)/xacro --inorder $(find autonomous_turtlebot3_swarm)/urdf/turtlebot3_waffle.urdf.xacro" />


    <param name="tf_prefix" value="robot1"/>
    <include file="$(find autonomous_turtlebot3_swarm)/launch/start_turtlebot3.launch">
        <arg name="init_pose" value="-x 0.0 -y 0.0 -z 0.0" />
        <arg name="robot_name" value="robot1" />
    </include>

</launch>
```

Figure: Shows the contents of the “start.launch” file.

The “start.launch” file is the driver launch file; in other words, when launching the package for this project, this is the first file to launch. This launch file is responsible for launching the world file, setting some global parameters, and calling other launch files that will be needed. As seen in the screenshot above, this launch file includes the “start_turtlebot3.launch” file, driving the launch of that file. This file deals with spawning our turtlebot into the world. Its contents are shown below:

```
<launch>
    <arg name="robot_name"/>
    <arg name="init_pose"/>

    <node
        name="spawn_minibot_model"
        pkg="gazebo_ros"
        type="spawn_model"
        args="$(arg init_pose) -urdf -param /robot_description -model $(arg robot_name)"
        respawn="false"
        output="screen"
    />

    <node
        pkg="robot_state_publisher"
        type="state_publisher"
        name="robot_state_publisher"
        output="screen"
    />

</launch>
```

Figure: Shows the contents of the “start_turtlebot3.launch” file.

At the beginning of the “start_turtlebot3.launch” file, there are two arguments: robot_name and init_pose. Robot_name defines the namespace for the robot that is being spawned and init_pose defines the initial position that the robot will be spawned at. Below that are two nodes that are run, “spawn_minibot_model” and “robot_state_publisher”. The “spawn_minibot_model” node is the part of the launch file that deals with spawning the model of the robot into the simulation. The “robot_state_publisher” node publishes the 3D poses of the robot using a kinematic tree model of the robot.

Next, is a picture showing the contents of the “start_bb8.launch” file:

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <group ns="bb8">
        <param name="tf_prefix" value="bb8"/>
        <param name="robot_description" command="$(find xacro)/xacro '$(find bb_8_description)/urdf/bb_8.xacro'"/>
        <rosparam file="$(find bb_8_gazebo)/config/bb_8_description_control.yaml" command="load"/>

        <arg name="roll" default="0"/>
        <arg name="pitch" default="0"/>
        <arg name="yaw" default="0" />

        <node name="robot_state_publisher_bb_8" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
            <param name="publish_frequency" type="double" value="5.0" />
            <remap from="/joint_states" to="/bb_8/joint_states" />
        </node>


        <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
            output="screen" args="--namespace=/bb_8
                                  joint_state_controller
                                  direction_joint_position_controller
                                  head_joint_position_controller
                                  --shutdown-timeout 3">
        </node>

        <node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
            args="-urdf -param robot_description -model bb_8 -x -15.0 -y -15.0 -z 0.5"
        />
    </group>
</launch>
```

Figure: Shows the contents of the “start_bb8.launch” file.

This launch file is responsible for spawning in the target model into the simulation. It contains multiple arguments that deal with the initialization of the model into the world, such as roll, pitch, and yaw. It also contains the nodes that deal with actually spawning the model into the world.

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find autonomous_turtlebot3_swarm)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>

  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find turtlebot3_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation.rviz"/>
  </group>
</launch>
```

Figure: Shows the contents of the “turtlebot3_navigation.launch” file.

Above is the “turtlebot3_navigation.launch” file; this file launches all the nodes necessary for navigation through the simulation. This file launches the map, the localization, move_base, and rviz nodes. The map provides an occupancy grid of open space and obstacles in the world. This allows movement goals to be set that take optimal paths around obstacles. The localization is launched by starting the amcl node; this allows the robot to know where it is inside the environment. Next, move_base is launched. This allows navigation goals to be given to the robot and also contains everything necessary for path planning. Lastly, rviz is launched; this allows us to see what the robot sees as it makes its way through the world.

```
<?xml version="1.0" encoding="utf-8"?>

<launch>

  <!-- Use YOLOv3 -->
  <arg name="network_param_file" default="$(find darknet_ros)/config/yolov3_custom.yaml"/>
  <arg name="image" default="camera/rgb/image_raw" />


  <!-- Include main launch file -->
  <include file="$(find darknet_ros)/launch/darknet_ros.launch">
    <arg name="network_param_file" value="$(arg network_param_file)"/>
    <arg name="image" value="$(arg image)" />
  </include>

</launch>
```

Figure: Shows the contents of the “yolo_v3_custom.launch” file.

This last launch file shown above is the “yolo_v3_custom.launch” file. The starts up the object detector to work on the turtlebot’s camera. This launch file takes in arguments for our custom parameters that work for detecting our target in the world, as well as launches darknet with those parameters.

ROS Node Communication

There are two main node graphs connected by a topic for communication within the Autonomous Search With AI simulation. The first graph represents the command and control laptop. The second graph represents the turtlebot.

Rviz

Rviz, or ROS visualization, is a 3D visualization tool for robots, sensors, and algorithms. It allows the user to see the robot's perception of its world, real or simulated. In other words, Rviz allows you to see what the robot thinks it sees. This is useful in visualizing the state of the robot. Knowing what the robot perceives is useful in creating a map for the robot to follow to navigate throughout the world; this is the main purpose that Rviz will be used for in this project. As said by Morgan Quigley, who is an original developer of ROS, "rviz shows you what the robot thinks is happening, while Gazebo shows you what is really happening".

Python

Since we decided on using ROS as our interface to program the robot, we were left with two options for programming languages to use: Python and C++. C++ is widely used in robotics due to its interaction with low level hardware and faster runtime; however, it is also more difficult to write and is less readable than Python. Python is simple to read and write, and also gives us access to various libraries, such as Tensorflow, that are useful for machine learning. This, along with most of the group being knowledgeable in Python led us to decide that Python is the best suited programming language for this project.

Tensorflow

Tensorflow is an open-sourced machine learning and computation library for Python. It includes machine learning and deep learning algorithms that are made easily usable. Developers are able to create dataflow graphs, describing how data moves through a graph. Each node in the graph is represented by a mathematical operation and each edge of the graph is a tensor (multidimensional array of data). Tensorflow can be used to train and run neural networks for things such as image recognition; this is how the robot will be able to detect the target. Tensorflow is the machine learning library

recommended by the sponsor; therefore, it was decided to be the best choice for this project.

JIRA

From a management perspective, our team used Jira to implement the agile sprint methodology. We originally started the project using Trello, but later switched to Jira as a recommendation from our sponsor. Jira feels more professional and is designed to help teams manage work. We used it to track bugs, issues, and tasks for each member throughout the project. The software has built in functionality for scrum. It allowed us to keep track of tickets in the back log, assign tickets, log work, create bi-weekly sprints, and most importantly, allowed us to communicate our progress to our sponsor at Lockheed Martin.

8. Environment Setup

The following instructions ensured that as a team we were using the same build environment while designing, implementing, and testing the Autonomous Search With AI. It is also provided for any future users who wish to run the simulation for themselves or for users who wish to expand on this project.

Installing & Setting up Ubuntu

Step 1: Install Oracle VM VirtualBox

Step 2: Click the Blue New button to create a new VM on your machine.

Step 3: Download the Ubuntu 20.04.1 ISO and mount it in a virtual optical drive.

Step 4: Start the VM in VirtualBox.

Step 5: Navigate the menus to Devices > Optical Drives > Choose a disk file, and select the Ubuntu 20.04.1 ISO.

Step 6: Power off the VM and restart it.

Step 7: Follow the on-screen instructions and select the default settings to install Ubuntu. (On the 4th page of the installation, select the check box “Install third-party software for graphics and Wi-Fi hardware and additional media formats)

Step 8: Allow for any updates to install

If not using a VM, make sure to use Ubuntu version 20.04.1 to ensure compatibility.

Note: The better the computer’s graphics, RAM, and Storage, the faster the simulation will run. A computer with a dedicated graphics card and at least 8 GB of ram is recommended.

Setting up the Gazebo environment

Run the following commands in a linux terminal to install and then launch gazebo

```
sudo apt install git
sudo apt install curl
curl -sSL http://get.gazebosim.org | sh
gazebo
```

Setting up ROS

Make sure you have a locale that supports UTF-8. Check this in the terminal.

```
locale
```

If the locale is not UTF-8, then run the following commands to set the locale to UTF-8:

```
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

Add the ROS apt repositories to your system.

```
sudo apt update && sudo apt install curl gnupg2 lsb-release
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Set up keys:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Update your apt repository caches after setting up the repositories.Then install the desktop version of ROS.

```
sudo apt update
sudo apt install ros-noetic-desktop-full
```

Next, the environment has to be set up. Set up your environment by sourcing the setup.bash file

```
source /opt/ros/noetic/setup.bash
```

For a better user experience, also install argcomplete to allow ROS 2 command line tools to allow for autocompletion.

```
sudo apt install -y python3-pip
pip3 install -U argcomplete
```

Next, to make a workspace in ROS, catkin needs to be installed:

```
sudo apt-get install ros-noetic-catkin
source /opt/ros/noetic/setup.bash
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
Source devel/setup.bash
```

ROS is now installed and set up!

Getting the Project Simulation

Navigate to the following directory and run the following commands to get the Autonomous Search With AI simulation:

```
cd ~/catkin_ws/src
git clone https://github.com/patrick1bauer/autonomous_search_with_ai
```

Adding a Robot to the Simulation

Launching Several Robots at Once

Launching several robots at once allows us to do many fancy things with ROS., such as using an AI robot swarm to find a target. Adding one robot is simple, but adding multiple robots depends on the simulation. The robot model had to be prepared to launch multiple instances within a single simulation.

Download and Install the robot model

The model of the robot used for this project is the waffle turtlebot3 and a bb8 unit

Launching a Single Robot

Step 1: Launch the Gazebo simulation world

Step 2: Use the already provided command in the terminal to launch the simulation:

```
roslaunch autonomous_search_with_ai start.launch
```

Figure: A single robot spawned into an empty world. Note: The location of where the robot will spawn is defined in the launch file

Launching Multiple Robots

The problem with using the same command to spawn a second (or more) robot is that the second robot will have access to the same topics as the first robot. There will be no differentiation between the two robots. Commands sent to a specific topic will affect both robots at the same time. Each robot should have their own separate topics so they can be controlled independently and given different tasks. This problem was solved by using ROS namespaces. To use namespaces, the spawning command had to be prepared differently. A new package and a new launch file was created that spawns multiple robots. Below are the steps used to achieve this.

Step 1: Navigate to the following directory:

```
~/simulation_ws/src
```

Step 2: Create a new package for launching multiple robots

```
catkin_create_pkg many_robots rospy
```

Step 3: Create and enter a new launch directory within the new many_robots directory

```
cd many_robots
mkdir launch
cd launch
```

Step 4: Create a start.launch file

```
touch start.launch
```

Step 5: Save the start.launch file as an XML file.

Now, the spawn_quadrotor.launch file has to be included in the start.launch file where the spawn_quadrotor.launch file is used once for each individual robot needed. Each robot also has a set number of parameters that need to be specified for them to launch properly. The following XML snippet shows the code required for each robot where ”robot1” and the y value should be different for each robot:

```
<group ns=”robot1”>
    <include file=”$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.launch”>
        <arg name=”name” value=”robot1” />
        <arg name=”tf_prefix” value=”robot1” />
        <arg name=”y” value=”-1.0” />
    </include>
</group>
```

The following commands can be used to launch multiple robots using the new .launch file created:

```
roslaunch many_robots start.launch
```

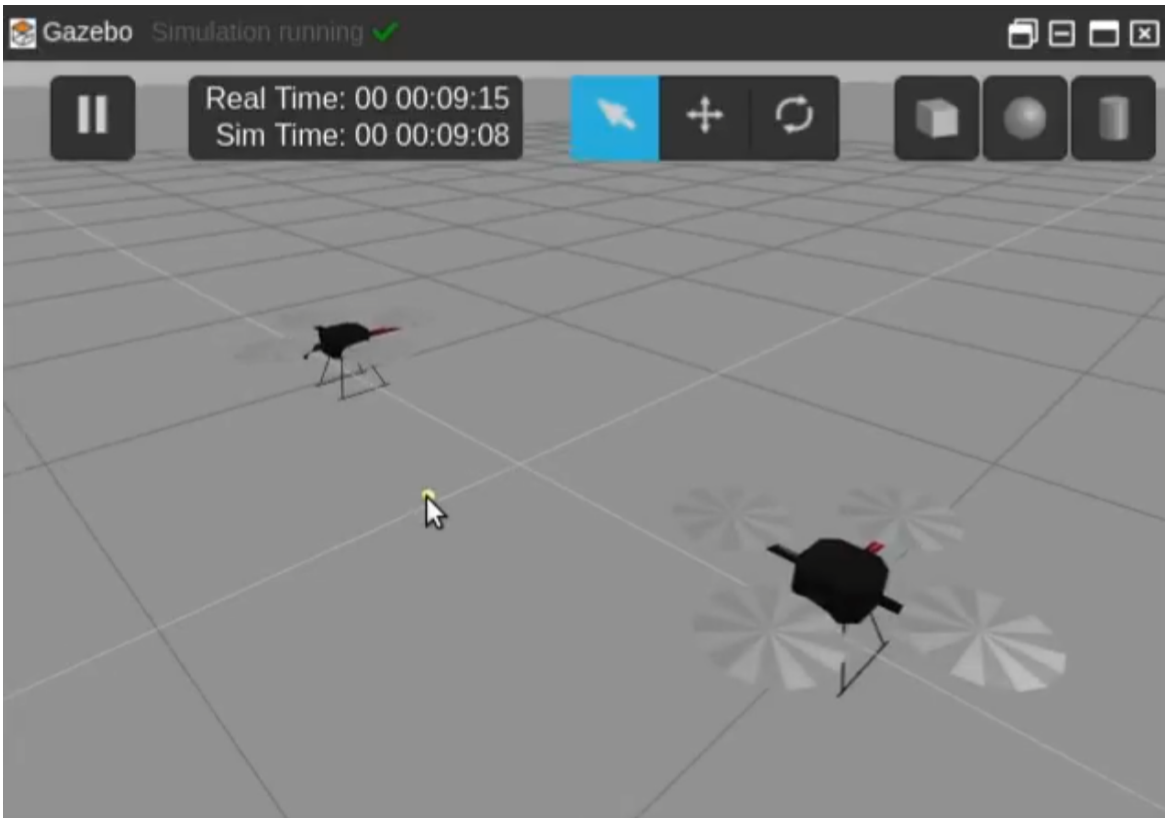


Figure: Multiple drones spawned each grouped into their own ROS namespace.

Controlling the robots using ROS

After starting one or more robots, one can use the topics to control the robot(s) as desired.

To view all the ROS topics associated with a robot, use the following command:

```
rostopic list
```

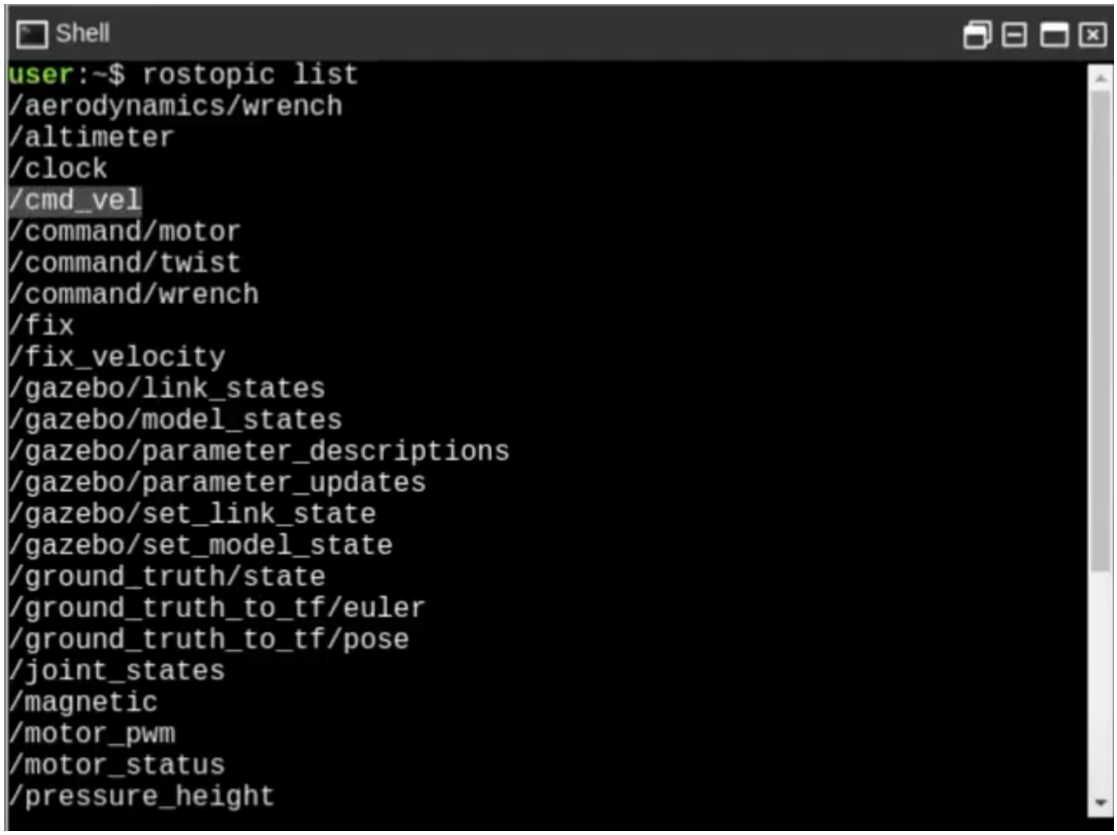


Figure: A list of the ROS topics after spawning a single robot. The highlighted topic (/cmd_vel) will be used to provide speed to the robot so it can move around.

As an example, the following command can be used to give the robot a linear z velocity of 1 (this makes the robot move up at a constant speed):

```
rostopic pub -r 10 /cmd_vel geometry_msg/Twist '{linear: {x: 0,y: 0.0,z: 1.0}, angular: {x: 0.0,y: 0.0, z: 0.0}}'
```

This same command can be used to rotate the robot by changing the angular coordinates.

Adding a Camera to the Robot

Adding the Camera Code

```
<joint name="camera_joint" type="fixed">
  <axis xyz="0 1 0" />
  <origin xyz="${camera_link} 0 ${height3 - axel_offset*2}" rpy="0 0 0"/>
  <parent link="link3"/>
  <child link="camera_link"/>
</joint>

<!-- Camera -->
<link name="camera_link">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
  <box size="${camera_link} ${camera_link} ${camera_link}"/>
    </geometry>
  </collision>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
  <box size="${camera_link} ${camera_link} ${camera_link}"/>
    </geometry>
    <material name="red"/>
  </visual>

  <inertial>
    <mass value="1e-5" />
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
  </inertial>
</link>
```

The first elements of this block are an extra link and joint added to the URDF file that represents the camera. This attaches the camera to the robot. Next, a Gazebo plugin was added to give the camera functionality. The camera publishes the image/feed to a ROS message. Listed below are several properties of the camera plugin used:

```
<gazebo reference="camera_link">
```

This is the same as the name of the link in the URDF code.

```
<sensor type="camera" name="camera1">
```

Definition of the camera sensor.

```
<update_rate>30.0</update_rate>
```

The maximum number of camera images taken per second within Gazebo.

```
<horizontal_fov>1.3962634</horizontal_fov>
<image>
  <width>800</width>
  <height>800</height>
  <format>R8G8B8</format>
</image>
<clip>
  <near>0.02</near>
  <far>300</far>
</clip>
```

These properties can be changed to match a physical camera's hardware.

```
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
```

This is where the gazebo_ros/gazebo_ros_camera.ccp file is linked to.

```
<cameraName>rrbot/camera1</cameraName>
<imageTopicName>image_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
```

These define the ROS topic the camera will publish the images to.

Starting the Camera

After creating the robot.xacro and robot.gazebo files, the following commands can be used in separate terminals to launch both Rviz and Gazebo.

```
roslaunch robot_sim my_world.launch
```

```
roslaunch robot_description robot_rviz.launch
```

Environment Layout

The robot is tasked with navigating through an urban environment to find and destroy the target.

The requirements for the environment as given by our sponsor, Lockheed Martin:

- Two streets, urban environment
- Must contain buildings and other structures
- 400 ft x 400 ft (subject to change)
- Other clutter (to attempt to throw off the image detection)

Figure: Screenshot of the world that we are working with.

Environment Construction

Worlds in Gazebo are saved as world description files. These files contain all the elements of a simulation, such as: robots, sensors, and static objects. The Gazebo

server (gzserver) reads these files to generate and populate a world. Gazebo comes with multiple example worlds, found within the /worlds directory of the Gazebo resource path. There are also built in launch files that load some of these world files in the gazebo_ros package.

It is possible to load prebuilt worlds that other people have made, so for our purposes, we have found a city world that satisfies the requirements laid out in the previous section. The world that we use for our simulation will mostly be made with simple models. Any other complex models will be obtained either by inserting models that come with Gazebo, or possibly obtaining community made models; this is how road and building models will be obtained.

Another consideration is that we have multiple worlds for our robot to search through. This can be useful because it is important to know that the algorithms that the robot uses are trained for general purposes, and are not trained specifically to find and eliminate targets on a single world. Having multiple worlds that contain different models and paths can be useful in this regard.

Adding structures in Gazebo can be done by doing the following:

For simple structures, you can find these at the toolbar at the top. To insert these structures, simply drag them down and place them where they are needed.

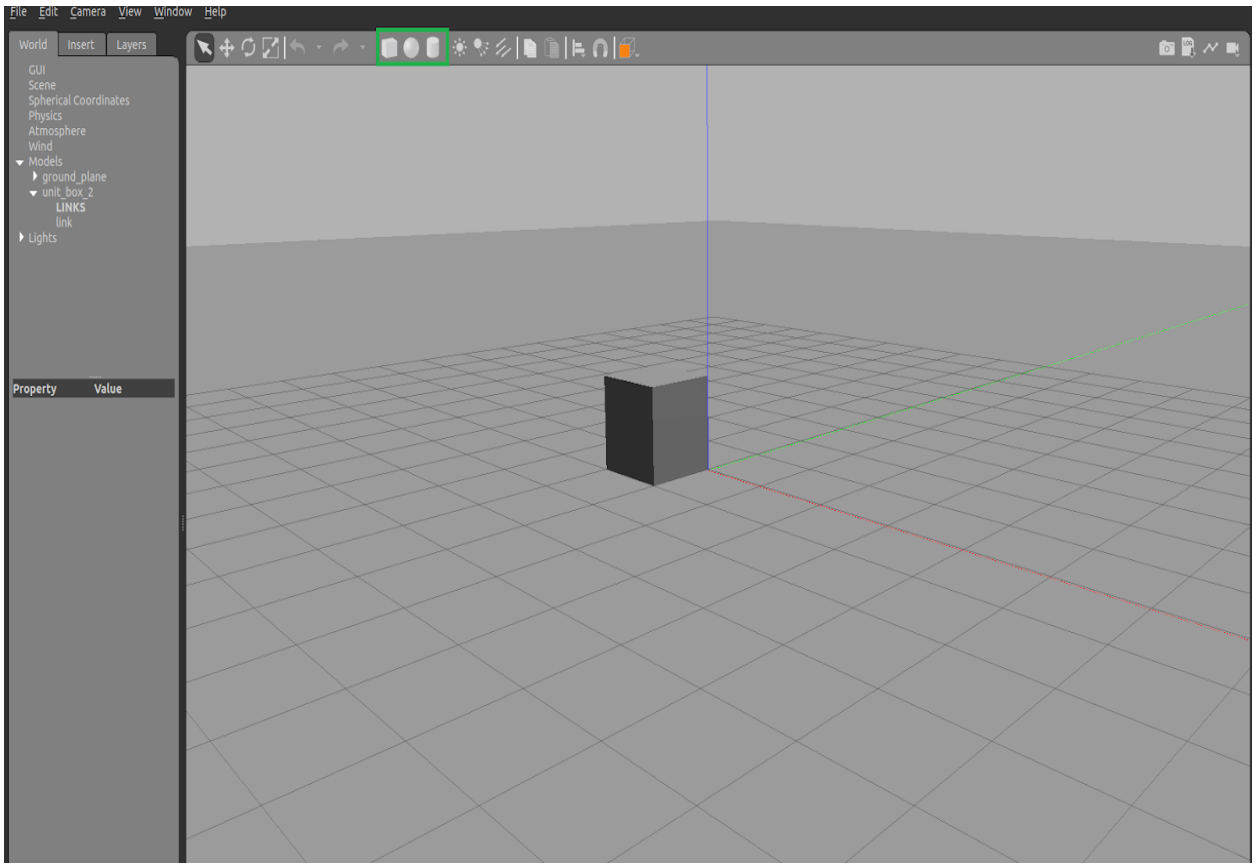


Figure: Select simple structures from at the top, indicated by the green rectangle

To add more complex structures, go to the left side and click insert. After a little while loading, more complex models will be available for use in Gazebo. These models can be clicked and dragged onto the world for use, just like the simple models. For even more specialized models, it is possible to obtain these via the Gazebo community, or made from scratch in a 3D modeling program.

The following image shows a closer look at the world:

Figure: Zoomed in screenshot of the world

When the world is launched, a script will execute that will cause the bb8 model to be spawned into the world. Also there are models of obstacles; these are included to try to throw off our image detection. A consideration is to not only detect the bb8, but also detect non targets such as people. In real life this is especially important because it is important that innocent people are not mistaken for a target.

BB8 Unit

The model for the target that was chosen is bb8. The bb8 can be easily customized in various ways depending on which situation it is needed to be in. The mechanical parts can be reconstructed and sensors can be added.

Figure: bb8 created in Gazebo

TurtleBot3 Navigation

ROS Navigation Stack

The navigation stack takes in information from the odometry and sensor streams and sends velocity commands to a mobile base as output. To use the navigation stack on a robot, it must meet the following requirements:

- The robot is running ROS
- There is a tf transform tree in place
- Sensor data is published using the correct ROS message types

The robot must also meet the following hardware requirements:

- Differential drive and holonomic wheeled robot
    - The mobile base is controlled by sending velocity commands in the form of x velocity, y velocity, theta velocity
- Planar laser mounted on the mobile base
    - Used for map building and localization
- Nearly square or circular in size
    - It may have trouble with arbitrary sized robots

Our robot of choice, Turtlebot3, meets all of the necessary requirements that have been mentioned above. Use of the navigation stack is optimal for navigating the turtlebot around the map.

Creating the Map

Robot navigation is done through maps in ROS. To get the turtlebots to navigate through the world autonomously we first must: create a map for the robots to follow, save the map, and finally use the map.

Once Gazebo is launched and everything is running, there will be the navigation.rviz window and the standard Gazebo window, looking like the following:

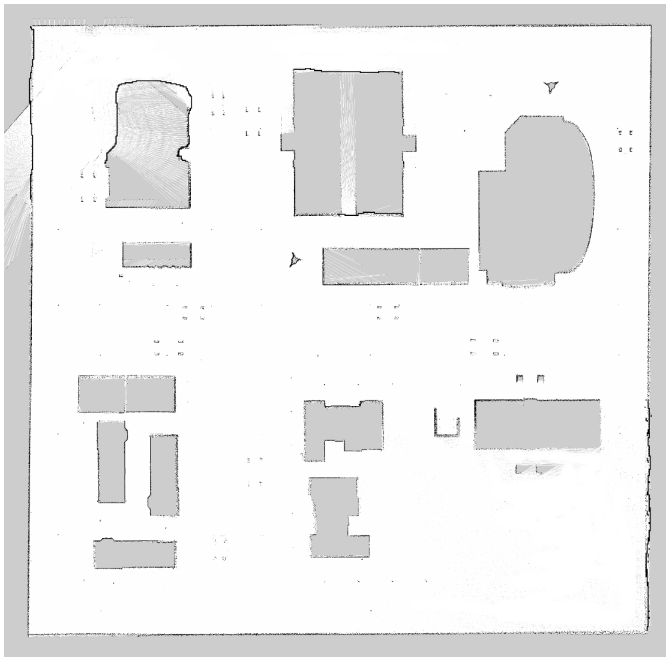


Figure: The world map displayed.

As the turtlebot is moved around the world, it will begin to fill up the map. This can be seen on the navigation.rviz screen, where you can see the grid begin to change color as the turtlebot gains vision of it, filling up the map.

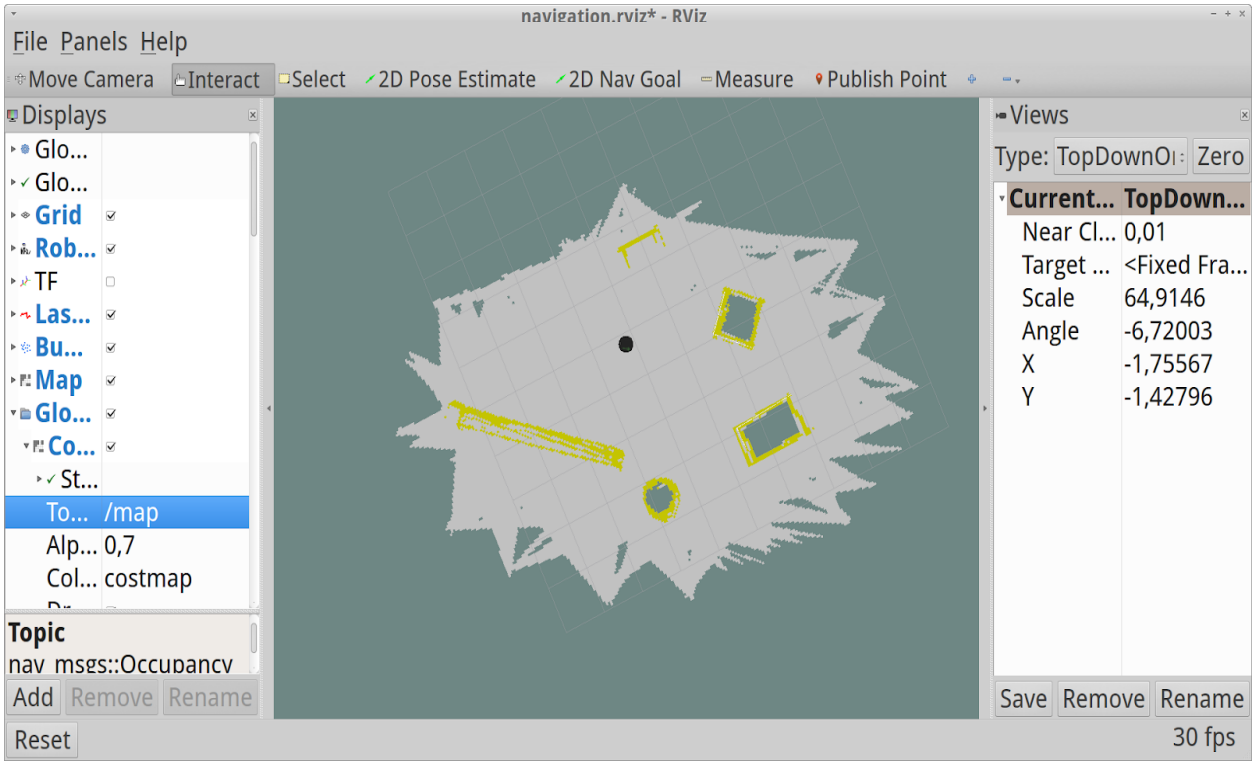


Figure: As the turtlebot is moved through the world, it begins to fill the picture in.

When the turtlebot has sufficiently navigated throughout the world, seeing all obstacles and paths, the map is created.

The turtlebot can be manually moved throughout the world by running the teleoperation node. This command allows the user to control the robot to perform SLAM operations manually. This can be useful to make sure that every corner of the environment is measured, and that nothing in the world is not being taken into account in the map that is being created. The controls for this are as follows:

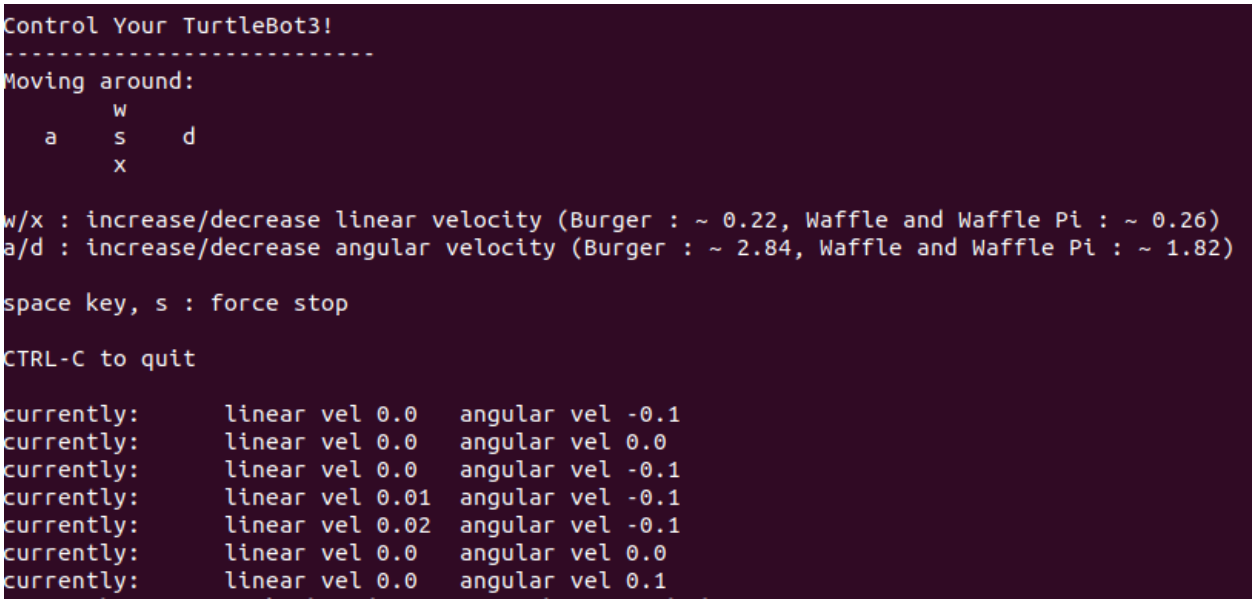


Figure: Shows the launch of the teleoperation node and the controls.

However, we found that manually mapping the environment is very tedious and time consuming; therefore, we decided to map out the environment autonomously. Thus, we launch a file that explores unknown areas of the map until there is no space left to explore. This allowed us to easily map any environment.

Testing and Running the Map

To ensure that the map is created successfully, it is important to first test the map before running it with the navigation.

Start by launching Gazebo:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Launch the navigation demo:

```
roslaunch turtlebot3_gazebo amcl_demo.launch
map_file:=/home/<user_name>/turtlebot3_custom_maps/tutorial.yaml
```

Launch Rviz:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

If the Rviz looks like the following picture, then the map has been realized successfully. The objects and obstacles should be highlighted and the rest of the movable space should be filled in as shown:

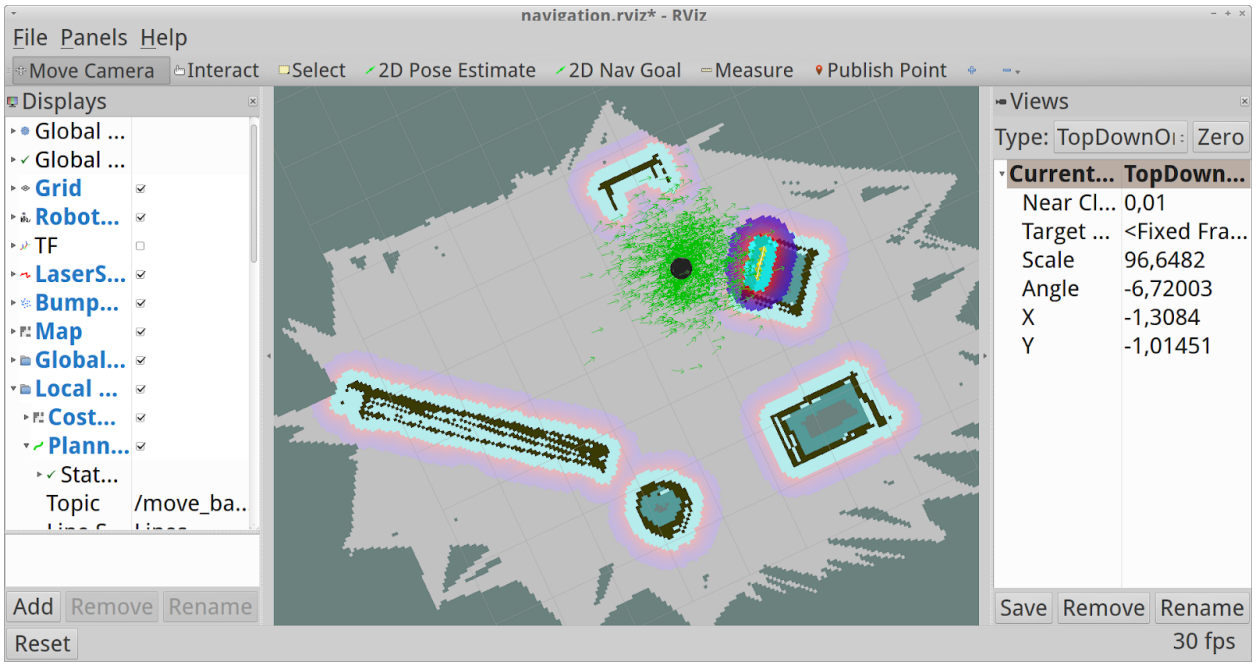


Figure: Shows a fully realized map

To direct the turtlebot to a specific location, click the 2D Nav Goal button and then click on the map where you want the turtlebot to go. Drag in the direction that you want the turtlebot to be pointing once it reaches its destination.

SLAM Algorithm

SLAM (Simultaneous Localization and Mapping) is the pathfinding technique used by the turtlebot. A map is drawn by estimating the current location in an arbitrary space. When creating the map above, SLAM is used. SLAM allows robots to map out an

unknown environment; this map information is used for pathfinding and obstacle avoidance, among other things.

Many times, the map that is initially created is not the optimal map for the robots to use. In the Gmapping package (where the SLAM algorithms are from) there are also parameters and tools that help with tuning the map that was created. Certain parameters can be changed to adjust for certain environments, making the map more efficient.

The slam_gmapping node takes in sensor data and builds the map. This process can be tweaked with these parameters.

Examples of tuning parameters that can be changed to impact performance are:

- **maxUrange**
    - Sets the maximum usable range of the lidar sensor
- **minimumScore**
    - Minimum score for considering the outcome of the scan matching
    - This can be used to avoid jumping pose estimates
- **linearUpdate**
    - Whenever the robot translates, it runs a scan process
    - Value set as distance translated
- **angularUpdate**
    - Whenever the robot rotates, it runs a scan process
    - Value set as rotation amount
- **Map_update_interval**
    - The amount of time (in seconds) between map updates
    - If set to be low, the map is updated more frequently but requires a greater computational load
- **maxRange**
    - The maximum range of the sensor
    - Regions with no obstacles appear as free space in the map
- **Occ_thresh**
    - Threshold on occupancy values

- Cells with greater occupancy are considered occupied

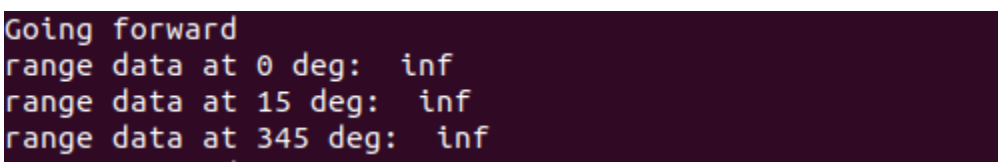
The most common map to use is the two-dimensional Occupancy Grid Map (OGM). This map shows the free space, which is where the robot can move. It shows occupied space, which is where the robot cannot move. Also, it shows gray space, which is an area unknown to the robot.

Target Wandering

A simple script was made that uses the laser sensors on the robots to detect obstacles in its path. If there is an obstacle in a range that meets a certain threshold, the robot will rotate and try to move forward in another direction. In doing this, we were able to avoid using SLAM and having to map out the world to have the robot wander. The script is embedded into a launch file so that it automatically executes when the simulation is started; thus, allowing the robots to immediately begin dispersing themselves throughout the world. This approach works better because no location data is needed from the robots; they just need to be dispersed throughout the world.

Using the laser scanner data, the robot is able to detect objects around it. As shown in the following screenshots, the robot is constantly scanning, receiving range data for objects in its path. To make a decision on whether to turn or not, range data is gathered from three different data points around the turtlebot: 0 degrees, 15 degrees, and 345 degrees. This is to ensure the turtlebot's surroundings are taken into account when going forward or making a turn. The following data was retrieved with a threshold of 4 meters, 2 meters, 2 meters for 0 degrees, 15 degrees, and 345 degrees respectively.

As shown below, the scanner does not detect any objects in the robot's path, therefore there is an infinite range the robot can travel before it must make a turn to avoid an obstacle. In reality, this just means that there is a large range, not an infinite range, since the scanner can only scan objects from so far away.



In the image below, we can see that an object has been scanned to be 3.4 meters at 0 degrees in front of the robot. Since this is below the threshold of 4 meters used in this demonstration, the robot must now turn to avoid collision with the obstacle.

```
Turning
range data at 0 deg:   3.4026010036468506
range data at 15 deg:  inf
range data at 345 deg:  inf
```

```
    def __init__(self):
        #initialize node
        rospy.init_node('turtlebot_wander')

        rospy.loginfo("CTRL + C to stop Turtlebot")
        rospy.on_shutdown(self.shutdown)

        # Publishes velocity to turtlebot
        self.cmd_vel = rospy.Publisher('robot/cmd_vel', Twist, queue_size=10)
        # Retrieves scanner data
        self.sub = rospy.Subscriber('robot/scan', LaserScan, self.laserNavigation)
        # Infinite loop
        rospy.spin()
```

Figure: shows initialization

In the code above, the initialization method is shown. This method initializes the ROS node, sets the publisher variable, sets the subscription variable, and then loops. The cmd_vel variable publishes the velocity data of the turtlebot to the cmd_vel topic for the robot. Whenever a change is made to the velocity, it must be published to the cmd_vel topic for the velocity to actually change for the robot. The sub variable subscribes to the scan topic for the robot. The scan topic receives data from the laser scanner and calls the laserNavigation method to make decisions based on the data received from the laser scanner.

The laserNavigation method receives the data from the laser scanner on the turtlebot; from this data, the method determines how to move the robot to avoid obstacles. For

our purposes, we have three thresholds set: one for 0 degrees, 15 degrees, and 345 degrees. If the laser scanner detects an obstacle that has a distance away from the robot that is within the thresholds, the robot move will be updated to avoid this obstacle. In this iteration of the code the following values were set for the threshold values:

0 Degrees: 4 meters

15 Degrees: 2 meters

345 Degrees: 2 meters

The meaning of this is that if an object 0 degrees in front of the robot is 4 meters away, then that object will be considered an obstacle and the robot must move to avoid it. This also holds true for objects 15 degrees and 345 degrees in front of the robot that are within 2 meters away. The reason for these thresholds being lower than the 0 degree threshold is because the 0 degree threshold indicates objects directly in front of the robot, sitting in its direct path. Objects that are 15 degrees or 345 degrees in front of the robot are not as much of a concern since they do not lie directly in the robot's path; however, these objects still need to be considered when making a decision on where to turn the robot.

As seen in the code snippet below, if there is no object in any of the thresholds, the robot's velocity is set to 0.5 m/s and its angular velocity is set to 0. Therefore, the robot continues to move forward and does not have to rotate to avoid an obstacle. If there is range data that falls within the threshold, the robot will stop moving, rotate, and then check if there are any new objects that are in its path. If there are no objects detected within the threshold again, the robot stops turning and begins to move forward again.

```
def laserNavigation(self, data):
        print ('range data at 0 deg:  {}'.format(data.ranges[0]))
        print ('range data at 15 deg:  {}'.format(data.ranges[15]))
        print ('range data at 345 deg:  {}'.format(data.ranges[345]))

        # Thresholds subject to change, we can play around with these numbers
        thr1 = 4
        thr2 = 2
        thr3 = 2
        move = Twist()

        # No object is scanned within range threshold, turtlebot moves forward
        if data.ranges[0]>thr1 and data.ranges[15]>thr2 and data.ranges[345]>thr3:
                move.linear.x = 0.5
                move.angular.z = 0.0
                print("Going forward")
        # Object is scanned within range threshold, turtlebot rotates and trys to move forward
        else:
                print("Turning")
                move.linear.x = 0.0
                move.angular.z = 1
                # After rotating, checks again if there is obstacle
                if data.ranges[0]>thr1 and data.ranges[15]>thr2 and data.ranges[345]>thr3:
                        move.linear.x = 0.5
                        move.angular.z = 0.0

        self.cmd_vel.publish(move)
```

Figure: Code snippet showing the robot wandering

Target Approach

Once the robot has discovered the target, the robot will pause the grid search algorithm and begin to navigate towards the detected object. When the turtlebot is close enough to have a >99% probability for detection, the simulation is complete and the target has been found and identified.

Once the target is identified, the robot will communicate to the user that a target has been found successfully.

Removing a Model in ROS

For our ROS implementation of this takedown method, we ran into the problem of ROS having no simulated microwave burst devices available to us. Therefore, for the final project we will be simulating the takedown by microwave burst by calculating the time the robot has a confirmed view of the target.

Once the command and control server has recognized a confirmed ID of the target from the robot for the last 50 detections, the bb8 will be “identified”

9. Building the Simulation

Build, Prototype, Test, and Evaluation Plan

The entire project will be built and tested in a simulation using Gazebo and ROS in a Virtual Machine. For the robot to be functional, the pathfinding, navigation, and computer vision algorithms must be completed, along with the simulated suburban environment in which the robot will attempt to locate and disable its target. With these components developed, the next step is to integrate them in ROS and Gazebo so that our simulated robot may use YOLOv3 to correctly view the world and SLAM to swiftly and accurately navigate the environment. ROS, which is the operating system for the robot, works well with Gazebo, and can be used to efficiently manage the different components of the robot. After the navigation and computer vision aspects of the robot are appropriately implemented, the models of the robot will behave according to our algorithms.

The project can be evaluated in several places. The computer vision aspect of the project will be evaluated using the built-in tools of TensorFlow, which detail both the loss and validation loss, as well as the accuracy and validation accuracy. Other Python libraries can generate a clean visualization of the model’s accuracy over epochs, which can be used to detect overfitting and underfitting. To ensure that the model is being assessed fairly, approximately 15% of the input images should be dedicated to a validation dataset, and another 15% of the input images will be used for the testing dataset. Validation data tests the model's accuracy after each epoch, thereby showing its periodic progress. Test data provides an unbiased final evaluation of the model. A confusion matrix can then be created to assess the accuracy, precision, recall, and F1 Score of our model.

Once the robot and all its components have been implemented into a working Gazebo simulation of a suburban environment, the pathfinding, collision detection, and real-time object detection can be tested and evaluated. Gazebo and ROS combined create an effective visualization of the robot. A bb8 robot will be placed within the simulated environment for the turtlebot to locate. In this stage, it is important to ensure that the robot avoids obstacles within the environment. Furthermore, the ability of the robot to effectively search for the target will also be evaluated. Measuring the speed

and accuracy of the robot's detection is likewise important to the success of the project, as the robot relies on our computer vision architecture to see the world. Depending on the performance of the robot, its various components can be tweaked and modified to improve its accuracy. It might therefore be worth testing the robot on maps with varying layouts and obstacles to evaluate how they perform in new environments. The bb8 will also be placed randomly around the simulation to provide a greater challenge for the robot.

Related Work & Consultants

Related work

A similar project was done in collaboration with students from the University of Pisa and the University of Naples in Italy. The project that these people worked on was in relation to swarm coordination of mini-UAVs for target searching. They highlight the great potential of Unmanned Aerial Vehicles (UAVs) in searching through unstructured environments. The tradeoffs in performance are explored with the use of these UAVs; on one hand, they are small, lightweight, agile, and easy to incorporate many types of sensors to navigate through an unstructured environment[1]. However, these UAVs have limited computing power, limited endurance, and imperfect sensors. The team explores the idea of using these UAVs in a swarm to capitalize on all the benefits and minimize the downsides to using UAVs. The focus of this project is the use of these UAV swarms in search and rescue contexts; this differs from the goal of our project, since we are more focused on using a ground based robot to find a target that is randomly placed in the environment. The researchers find that while their initial setup of the UAV swarm is effective in accomplishing their goal of target search, there is still a lot of work to be done with parameter optimization to make the overall mechanism tuned to the scenarios. They end off their findings with a call for further research into parameter optimization strategies to make UAV swarms more effective in their tasks.

Another similar project was completed at the University of North Dakota, by Mitch Campion, Prakash Ranganathan, and Saleh Faruque. They explore the ability of drone swarms to distribute tasks and coordinate the operation of drones independent of an operator. Their main point of research looks to propose a drone swarm architecture that utilizes cellular mobile network infrastructure to increase drone autonomy and reliability. They find that using cellular mobile network infrastructure alleviates many of the limiting factors for drone swarms, such as: range of communication, networking challenges, and size-weight-and-power considerations[9].

Consultants

In this project, we consulted different individuals that helped provide information and advice on implementing and developing the robot, AI, and simulation. Joseph Rivera, our advisor and connection to the Lockheed Martin Fire and Missile Control Applied Research, helped coordinate our team by holding bi-weekly meetings to check our progress. These meetings helped encourage accountability and progress within our project, and Joseph Rivera also provided helpful insights, particularly in the computer vision section of the project. Joseph, who is highly experienced training convolutional neural networks, advised us on the following concepts and technologies, which ultimately helped facilitate our research and implementation of the computer vision algorithms:

- Computer Vision Models
    - Faster RCNN
    - YOLO
    - Single Shot Detectors
    - RetinaNet
    - MobileNetv2
- Using JIRA to implement Agile methodologies
- Approaching the bb8
- Preprocessing the input data for the computer vision model
    - Labeling and gathering data

For the Gazebo and ROS aspects of this project, we consulted Dr. Gita Sukthankar, a professor at the University of Central Florida who specializes in Robotics. After contacting Dr. Sukthankar, she directed us to a helpful series of YouTube tutorials that introduced us to developing in the ROS and Gazebo environments. Dr. Sukthankar also provided us with her lecture notes, which were convenient and contained useful information about creating simulations, integrating ROS with Gazebo, and working with robots.

10. Administrative Content

Budget & Financing

The budget that we have decided on for this project is zero dollars. All the technologies and services that will be used to complete this project are either open-sourced or offer a free tier that satisfies all of our needs. ROS and Gazebo, which are the tools that will be used to create and run the robot simulations, are both open-sourced, requiring no licensing fees to use.

Version control will be done using the free tier of GitHub, as the tools gained from the paid tiers of GitHub will likely not be necessary for this project. Any 3D-modeling or graphics needed for this project will be created using Blender, a free and open-source computer graphics software. TensorFlow and NumPy are two open-source libraries for Python that we will be using to program the AI for the robot; these both require no cost.

Task management will be done using Trello using the free tier. Trello allows us to keep track of tasks that need to be completed and that have already been completed. The command and control center will be hosted on google cloud VMs which will be paid for by student credits.

Finally, all team communication will be done using Discord, Skype, or Zoom; all of these offer a free tier that allows us to communicate with each other free of cost. Overall, it is estimated that no funds will be necessary to complete this project.

Software and tools used:

Software / Tool	Cost
ROS Course Access	$50.00 x 3
ROS Data Increase	$5.00 x2
Robotic Operating System	$0.00, Open Source Software
Gazebo	$0.00, Open Source Software
Python	$0.00
Tensorflow	$0.00
Google Collab	$0.00
Github	$0.00
Discord	$0.00
Jira	$0.00
Total:	$160.00

Figure: Infographic of estimated costs

Milestone Chart

Fall 2020

8-24-2020	Classes Start - Team formation
9-18-2020	Sponsor Meeting, Discussion of goals
9-25-2020	Collaborate on initial project proposal
9-30-2020	Generate project timeline for sponsor approval
10-2-2020	Receive sponsor approval on project timeline and scope
10-5-2020	Research Begins Nathanael: Pathfinding Mark: Computer Vision Pedro: Computer Vision Noah: Ros/Gazebo setup, target elimination Patrick: Ros/Gazebo setup, robot communication
10-30-2020	Completion of research
10-30-2020	Begin creation of Gazebo environment
11-27-2020	Complete Ros/Gazebo orientation
12-11-2020	Initial Prototype Have Gazebo Environment set up Working robot LIDAR Video-Input
12-11-2020	Begin gathering training data
12-12-2020	Fall semester ends

12-14-2020	Present Semester's work to Lockheed Martin engineers.

Spring 2021

1-11-2021	Spring semester starts
1-18-2021	Complete data gathering and labeling
1-22-2021	Finalization of artificial intelligence algorithm
1-22-2021	Finalization of pathfinding algorithm definitions
2-26-2021	Completion of training of artificial intelligence algorithm
2-26-2021	Competition of pathfinding algorithms Measure time complexity against precision Which algorithm combination requires the least hardware
3-12-2021	Finish implementation of pathfinding algorithms
3-12-2021	Finish implementation of computer vision algorithms
4-9-2021	Complete unit testing
4-23-2021	Finish creating all documentation
4-26-2021	Final presentation to sponsor
4-28-2021	Spring semester ends, Graduation

11. References

1. *A.L. Alfeo, M.G.C.A. Cimino, N. De Francesco, A. Lazzeri, M. Lega, G. Vaglini, "Swarm coordination of mini-UAVs for target search using imperfect sensors", Intelligent Decision Technologies, IOS Press, Vol. 12, Issue 2, Pages 149-162, 2018*

2. *Bre, Facundo, et al. "Fig. 1. Artificial Neural Network Architecture (ANN i-h 1-h 2-h n-o). ." ResearchGate, 25 Sept. 2020, www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051.*

3. *Hardesty, Larry. "Explained: Neural Networks." MIT News, 14 Apr. 2017, news.mit.edu/2017/explained-neural-networks-deep-learning-0414.*

4. *Hui, Jonathan. "SSD Object Detection: Single Shot MultiBox Detector for Real-Time Processing." Medium, Medium, 28 Dec. 2018, medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06.*

5. *Jordan, Jeremy. "Convolutional Neural Networks." Jeremy Jordan, 26 July 2019, www.jeremyjordan.me/convolutional-neural-networks/.*

6. *"Keras Documentation: MobileNet and MobileNetV2." Keras, keras.io/api/applications/mobilenet/.*

7. *Kumar, Akshay. "How to Use a LiDAR Sensor With Robot Operating System (ROS): ROS." Maker Pro, Maker Pro, 5 Dec. 2020, maker.pro/ros/tutorial/how-to-use-a-lidar-sensor-with-robot-operating-system-ros.*

8. *Liu, Wei, et al. "SSD: Single Shot MultiBox Detector." 29 Dec. 2016.*

9. *Mitch Campion, Prakash Ranganathan, and Saleh Faruque. "A Review and Future Directions of UAV Swarm Communication Architectures." University of North Dakota, https://und.edu/research/rias/_files/docs/swarm_ieee.pdf*

10. *Montemerlo, Michael, et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem." Aaai/iaai 593598 (2002).*

11. *“RetinaNet Explained and Demystified.” NICKZENG, 12 May 2018, blog.zenggyu.com/en/post/2018-12-05/retinanet-explained-and-demystified/.*

12. *Sharma, Pulkit. “A Step-by-Step Introduction to the Basic Object Detection Algorithms (Part 1).” Analytics Vidhya, 11 Oct. 2018, www.analyticsvidhya.com/blog/2018/10/a-step-by-step-introduction-to-the-basic-object-detection-algorithms-part-1/.*

13. *Thakur, Rohit. “Step by Step VGG16 Implementation in Keras for Beginners.” Medium, Towards Data Science, 20 Aug. 2019, towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c.*

14. *Thatte, Abhijit V. “Evolution of YOLO - YOLO Version 1.” Medium, Towards Data Science, 21 May 2020, towardsdatascience.com/evolution-of-yolo-yolo-version-1-afb8af302bd2*

15. *Veen, Fjodor van. “The Neural Network Zoo.” The Asimov Institute, 27 Apr. 2019, www.asimovinstitute.org/neural-network-zoo/.*

16. *Wagstaff, Keith. “Tokyo Police to Deploy Net-Carrying Drone to Catch Rogue Drones.” NBCNews.com, NBCUniversal News Group, 11 Dec. 2015, www.nbcnews.com/tech/tech-news/tokyo-police-deploy-net-carrying-drone-catch-rogue-drones-n478596.*

17. *YOSHIKAWA, KAZUKI. “Japan Successfully Zaps Drones with Microwave Beams.” Nikkei Asia, Nikkei Asia, 31 Dec. 2019, asia.nikkei.com/Business/Technology/Japan-successfully-zaps-drones-with-microwave-beams.*

18. *“Convolutional Neural Network (CNN).” AI Wiki, docs.paperspace.com/machine-learning/wiki/convolutional-neural-network-cnn.*

19. *“ How RetinaNet Works?” ArcGIS API for Python, developers.arcgis.com/python/guide/how-retinanet-works/.*

20. *“Introduction To YOLOv4.” Analytics Steps, www.analyticssteps.com/blogs/introduction-yolov4.*

21. *“Naming resolution examples”, Ros.org, http://wiki.ros.org/Names*

22. *Science, ODSC - Open Data. “Overview of the YOLO Object Detection Algorithm.” Medium, Medium, 25 Sept. 2018, medium.com/@ODSC/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0*

23. *Published by nerdthecoder, et al. “Recurrent Neural Net.” Nerdcoder, 3 Feb. 2019, nerdthecoder.wordpress.com/2019/02/03/recurrent-neural-net/.*

24. *“The Battle of Speed vs. Accuracy: Single-Shot vs Two-Shot Detection Meta-Architecture.” Allegro AI, 20 Apr. 2020, allegro.ai/blog/the-battle-of-speed-accuracy-single-shot-vs-two-shot-detection/.*