Real-Time Head Tracking with a Duckie-Drone using Reinforcement Learning

Team S.A.D
Santiago Pinzon, Alex Tapley, Drew Bodmer
CS4610 Robotic Science and Systems
{pinzon.s, tapley.a, bodmer.d}@husky.neu.edu

I. Problem

In the world of robotics, machine learning currently takes second place to well-known and researched algorithms such as SLAM for tasks involving navigation and localization, due to its ability to map the environment of a robot, as well as gather information regarding where it is in reference to that environment. Other algorithms, like YOLO, are useful for real-time object detection and tracking by creating bounding boxes over certain features in the robots' environment. In addition to these algorithms, LiDAR is commonly used to get information regarding the surroundings of the robot by reading information from lasers to gather distance from objects and location.

The goal of our project was to use a camera mounted beneath a Duckie-Drone to track and follow the movements of an object below. The images taken from the birds-eye-view camera would be used as the input to a model trained using reinforcement learning inside of a simulated environment within Unity. The output of the trained model would be interfaced with ROS to maneuver the Duckie-Drone about the environment.

We composed this problem into three main subtasks - assembling the hardware, training in simulation, and interfacing the trained model with the hardware.

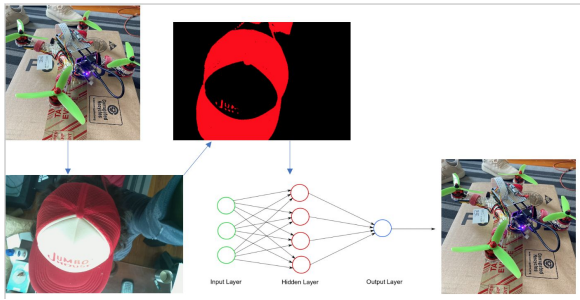

Fig. 1: System Diagram

All three subtasks are necessary for the overall ability of the final product. In assembling the hardware for the Duckie-Drone, the key component is the camera. The camera mounted on the underside of the drone is constantly taking images of the environment below it. These images from the camera are the inputs to the ROS interface. Within ROS, we have to take the image from the camera, perform some image processing on it and run it through the

trained model. Our model is trained within Unity with reinforcement learning to take in a single camera image and produce an output vector *[x, z]* where *x* is the force to move the drone in the x-direction and *z* is the force to move the drone in the z-direction. To simplify things, we are assuming the drone does not need to adjust its height in the y-direction while tracking the object. Using this vector, we are able to tell the drone the direction and magnitude in which to move in order to track and follow the object.

II. Process

1. Assembling the Hardware - Santiago:

This task was very front-loaded in terms of completing the project. Under normal circumstances it would have been the very first step completed and a collaboration between all three members, but due to the special circumstances surrounding this semester, it had to become a solo project. This process was fairly straight-forward, albeit slightly tricky given the limited access to resources. For example, the group did not have ready access to a multi-meter, making electrical debugging a non-trivial task. Thankfully, after following the instructions laid out in the Duckie-Drone manual, there were minimal issues with the drone. When plugging in the battery for the first time, smoke began to appear from the drone. Upon further examination, one of the power wires from the flight controller to the ESCs had overheated. Fortunately, this did not appear to affect the condition of the drone, and after re-soldering a connection that had popped off, the drone turned on with no issues.

I had built a drone before, and as a result I had experience working with all the different hardware on the drone. However, this was still a challenge at times, as I did not have a soldering fan or a helping hand for the matter, and often had to jerry rig set ups using electrical tape and what tools I could find around my apartment. By the end of the assembly I was fairly proficient at stripping wires with scissors, and making tiny solder joints one handedly. The part that caused me the most trouble was soldering the pi hat. As I had to resort to holding it down with a wrench and some tape the pins hardened partially crooked. This ultimately did not prove to be as much of an issue as I had originally thought, but it is a fix I would have liked to make. Another unfortunate casualty of circumstances was the fact that I had not grabbed enough zip-ties to create the feet for the drone, and as a result it would sit on the battery, which did not provide a stable or even base for the drone. With the main assembly completed, after flashing both the Raspberry Pi and the Flight Controller with the appropriate software, it was time to connect to the drone for the first time.

Fig. 2: The assembled drone

This part proved to be fairly difficult due to vague documentation. After finding the correct password to connect to the drone's network, we finally had access to the Pi via SSH. This worked well enough, but it had the issue that the computer connected to the drone had no wifi connection. A secondary reason for connecting the drone to wifi was that the PiDrone package that came with the Linux distro was outdated, and the camera would not work unless it was updated. Up until this point we had been accessing the Pi headlessly, but in order to connect the drone to wifi, we resorted to using a TV as a monitor. For some reason, the Linux distro used in the Duckie-Drone manual has the wifi disabled automatically. The PiDrone package came with a script to connect the drone to a wifi network, but in our experience, this script never quite worked. After reading through several forum posts, the solution appeared to be turning off wifi power management. Once the drone was connected to wifi, we were able to pull the updated PiDrone.

The package included an HTML file that when run provides an interface through which the drone can be controlled. Our aim was to have the drone be controlled autonomously, but this interface was useful for debugging the camera. In order to provide Drew and Alex with data to work on, I had originally written a quick python script which focused the camera and took a photo. I ran this script several times to get images of myself from above using the drone. These images, although not enough to train a neural network, would help Alex and Drew get an idea of the data which the camera would be feeding the network. For example, we learned that the camera outputted images at a resolution of 720x480p, and from the interface we knew that the speed of the images updating was fairly slowly.

In order to interact with the drone itself I had to learn a fair bit about ROS itself. The drone operates with a set up of ROS nodes that all communicate with one another and allow the different systems to interact and communicate. I had personally never worked with ROS before, and as a result the commands were not particularly intuitive. I spent a fair bit of time confused as for why I was unable to switch windows within ROS, until I came to the realization that I was hitting the apostrophe key instead of the tilde key. After this epiphany I was able to see how the different nodes were interacting with one another. The IR and Camera nodes would take in data from the real world and pass along the processed data

to the flight controller, with a few intermediate steps. At first the **CameraStream(CHANGE)** node was not working and did not publish images to the interface. As I mentioned above, after consulting with another group, I came to two realizations. Firstly, the Pi_drone package had to be updated and secondly, the drone had to be in position control mode in order for it to publish images to the interface.

2. Training the Model - Alex:

This specific subtask focuses on the creation and training of the reinforcement learning model to be used by the drone's ROS interface to track and follow an object. This subtask can be split up into three smaller subtasks - designing the environment in which to train the drone, writing the code to allow the drone to interact with the environment, and the actual training of the model to produce meaningful output.

a. Designing the Environment

The first step towards creating the trained model involved designing the environment in which the drone would train. Unity 2019.3.10 was used to create the environment, due to its physics engine and compatibility with Unity ML-Agents - a package created by Unity specifically for training reinforcement learning models in simulation. Since reinforcement learning is computationally expensive, the environment needed to be as simple as possible to allow for quicker training. Our main goal was to have the Duckie-Drone follow and track an object, so our environment consisted of a single floor and 4 walls. However, an issue arose when constructing the walls in which the directional overhead light cast dark shadows into the environment. These shadows would eventually cover the object we are following, making it impossible to make out the color of the object. Also, the color difference in the floor with/without the shadow would make training more difficult. Reinforcement learning agents do not respond well to change in environments, so the shadow floor vs. non-shadow floor would cause longer training times. To combat this, the shadows were turned off in the environment and the walls were made invisible. This allowed for no shadows, while also keeping the drone within the confines of the environment.

After the environment was created, the agent (Duckie-Drone) and the target (object to follow) were added. The Duckie-Drone was represented by a flattened green cube and the object to follow was represented by a red cylinder. A camera was attached to the bottom of the drone in order to capture visual observations from the environment below. The drone was then placed a decent distance above the target.

One issue that arose over time with the environment was that the overall space was too large. This meant that for most of the training episode, the drone would not be seeing the target in the camera frame at all. This hurts training because the model does not get a lot of rewards from the environment. Since it doesn't get a lot of rewards, the model cannot determine when a move is "good" vs. "bad". In order to fix this issue, the environment was shrunk down

to half the size. By reducing the size of the environment, it was more likely that the drone would see the target, resulting in better training.

Another issue that came up originally was that a capsule was being used instead of a cylinder for the agent's shape. A capsule is a cylinder with a half-sphere on top. This half sphere reacted with the directional light in the environment to create a glare on top of the target. This glare changes the color of the target during training depending on the target's location in the environment. To fix this, the capsule was changed to a cylinder so that the color of the target was consistently (255,0,0). This change made it easier for the agent to recognize what was the target.

In the beginning, a single environment was used to train the agent. However, five more environments were eventually added in order to increase training efficiency. This allowed for six agents to be trained in parallel as opposed to only one. By having six agents running at once, the model is supplied with a lot more [observation, action, new observation, reward] pairs to train on.

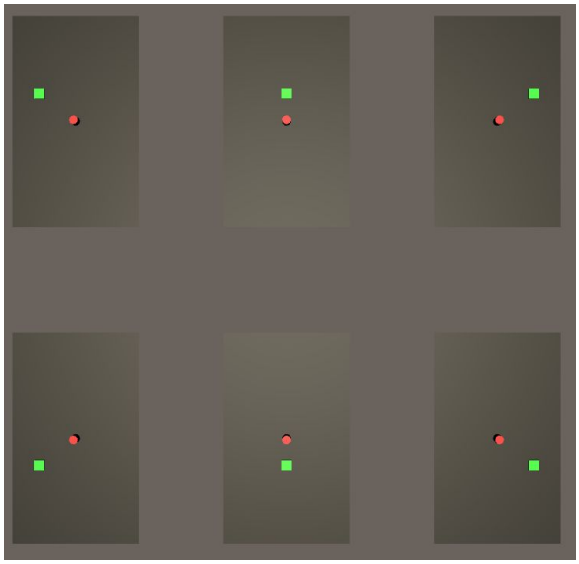

Fig. 3: Training Environment

b. Writing the Code

After the environment, agent, and target were created, code had to be written to interface between all three. The Unity ML-Agents package provides this functionality with their 'Agent' class. Unity allows the user to attach scripts to objects within the environment. By attaching an 'Agent' class script to the agent in the environment, the user is able to control what happens at the start of each episode, when the agent collects a reward, how the agent can move, what happens at each time step, etc. The six agents in our environment all have the same script attached and work within their own "local" environment.

At the start of each episode, the agent and target are placed in the center of the environment with the drone being placed 13.5 units above the target. This starting position was determined by the heights of the agent/target as well as the field of view of the camera attached to the agent. At startup, a random location within the

environment is chosen. The target is then moved towards that location, giving the appearance of the target "walking". Once it reaches the specified position, a new random location is determined and the target moves towards that one. This is done consistently until 1000 time steps are reached and the episode is over. Once the episode is over, the process is restarted.

Throughout this process, the Unity ML-Agents backend is running the images from the camera through our model. In our case, the model spits out an array of two values *[x, z]* where *x* is the amount of force used on the agent in the x-direction and *z* is the amount of force used on the agent in the z-direction. Once these forces are applied, the distance between the agent and target is measured. If the distance is less than a specified amount, a reward of +1 is given. This is the only reward given in the environment.

Initially, the model was operating in a discrete action space. However, this was changed to a continuous action space in order to more accurately describe how the drone could move. This, of course, affects training, but made the motion of the drone more apt to follow an object that was not moving in the same discrete space.

Along with the change of action space, the dimensions of the camera were changed as well. As given by the hardware of the drone, the images we are receiving are 720x480p. These images are far too large to train efficiently on. Since we only care about the color of the target, this value was shrunk down to 180x120p as an input to the model. However, during training, this was eventually bumped up to 360x240p to showcase a larger target to the model.

The reward of the system was changed a lot as well. In the beginning, the distance between the drone and agent did not take into account the height difference, which of course gave a net reward of zero for all of the training. This distance value was then changed to < 13.1 to accommodate the height. However, after training for a while, the agent was not getting rewards often enough, resulting in worse training. To fix this, the distance in which the agent receives a reward was changed to < 13.2.

c. Training the Model

To train the model, Unity ML-Agents 0.15.1 was used. The Unity ML-Agents package comes with a few different reinforcement learning algorithms implemented in Tensorflow for the user to train with. The main algorithm we used for our training was their proximal policy optimization algorithm (PPO). PPO is an actor-critic architecture with a special add-on module for gradient clipping. This gradient clipping helps to avoid large jumps in the policy, resulting in a smaller chance of the policy finding a sub-optimal solution.

Multiple trials were run with different hyperparameters. In all cases, the algorithm used included a convolutional neural network (CNN) with 256 nodes for our visual observation. One trial included using the default parameters given by Unity ML-Agents for most of their training examples. This includes two fully-connected dense layers with 128 hidden nodes, a batch size of 1024, a buffer size of 10240, and a

beta value of 5.0e-3 (among others). In another trial, there was only one dense layer with 256 hidden nodes, a batch size of 64, a buffer size of 2024, and a beta value of 1.0e-2. In addition, an extra module - Intrinsic Curiosity Module (ICM) - was added. ICM is used in environments where the reward is sparse. The agent compares what it is expecting to see to what it actually sees and gets a reward based on the difference. This rewards the agent for exploring things that it is not used to, resulting in a lot more exploration by the agent.

Many issues arose during training - specifically linked to the ML-Agents package. For some reason, whenever a change was made to the Unity project, ML-Agents would not be able to connect the environment to the trainer, making training impossible. To fix this issue, the entire ML-Agents repo would need to be deleted from my laptop and recloned, along with creating a brand new python virtual environment with the dependencies. This issue was brought to the attention of the Unity team in Issue #3804 and eventually moved over to the Unity Forum. As of the time of writing this, the only way to fix the issue whenever a line of code is changed is to reclone the repository. This was a big issue for training, because, in our case, direction plays a huge factor in how the agent should be moving. However, Unity ML-Agents does not support frame-stacking at this point in time. Since I could not get into the code to add it, we could not use frame-stacking, which severely limited our training.

Another issue that arose was that old models could not be loaded properly. The hard-hitting example for this was when one model stopped training at the 'max_steps' value of 5.0e6. In order to train for a longer amount of time, I had to change the value in 'max_steps'. But when the value was changed, I had to re-clone the repository. This created an issue, because then things did not line up between the new model I wanted to run and the old model I wanted to load. The only workaround to this was to retrain another model and lose all progress I had made with the old. Unfortunately, this meant another day of training.

The largest issue with training was the time. Training a model takes an enormous amount of time (days). If that model does not work, then all that time is wasted. If a hyperparameter is not correct, or a change needs to be made to the environment/reward function, then you need to restart all the training over again. Since a lot of reinforcement learning is trying new hyperparameters, there is a lot of trial and error. This was a big issue in getting a working final model. Not to mention, all training had to be done on a 2015 MacBook Pro, which is definitely not the best choice for running machine learning algorithms.

The hyperparameters that showed the most promise were the ones associated with the classic PPO without ICM. The 'max_steps' was set to 1.0e7, since it took around 5.0e5 steps to get meaningful data. ICM could have worked but the training was taking longer than expected. Entropy was still rising after 5.0e5 steps, meaning the agent was still testing random paths during

training. Classic PPO showed entropy falling around that same time with extrinsic rewards rising quickly.

3.Interfacing within ROS - Drew:

This task was arguably the least work-intensive of the three tasks, but was the best way we could come up with to split up the project. We needed a way to feed the camera image into the model and the model result back into the drone flight controller. This involved creating a ROS node that could subscribe to the camera feed ROS topic and publish to the drone's pose ROS topic. The node also needed to load our model and correctly process the image. Since I didn't have access to the drone, I wanted to run the code on a personal machine to test it without needing to upload it to the drone every time. I tried running it two different ways:

a. Windows ROS installation:

There were immediate issues with python and ROS versions, as well as library and dependency issues. I went through the ROS tutorials on nodes and topics, and how to install ROS and setup a workspace and build packages. After completing the tutorial, I attempted to build the drone package. ROS was unable to recognize the pidrone package, and there were failures with build dependencies and file paths. Debugging led to linux-centric forums, and answers for Windows users were sparse.

b. ROS Development Studio

Since most of the information I could find online used a bash shell, I decided it would be easier to use a Linux virtual machine instead. I used the ROS Development Studio because it provided a free online linux VM with an IDE. I managed to run basic scripts from the drone such as the ekf_localization.launch, but was unable to run start_pidrone_code.sh without errors. After several hours of debugging I decided it wasn't worth the effort to try to run the drone code on a machine.

The script to pull images from the drone camera, process them to pull out the target, and send instructions to the drone is split into two parts. hat-tracking.py is the code to run the full process, and process-image.py is just for importing the model and image processing, with all drone communications removed. It was split up in this way to allow for testing of the script functionality without communicating with the drone. To get an image from the camera, I used a subscriber that pulls from /picamera/image_raw, a ROS topic that the camera node publishes to. The image was then converted to a CV image for processing by OpenCV. To give the drone commands from the model, I used /desired/Twist, which takes in two vectors - an angular and linear vector. Since we are only doing 2D motion, the linear x and z values were the only ones needed.

One aspect of reinforcement learning that makes sim-to-real so difficult, is that real life images look nothing like simulated training images. To try and bridge this gap, we decided to try and make the real-life image look as close as possible to the simulated images. This included using an upper and lower bound to determine which

color “red” we wanted our target to have. Using these bounds, we could pull out any “red” in the image and make it the same color “red” as used in training. To accomplish this, I used openCV to create a mask of the image and overlaid it with the original image. I then maxed out the red channels in the masked image to create a fully-red section that would be usable for the model. The blue and green channels were zeroed out in order to make the rest of the background black.

In order to use the model, we had to load the graph used in training and apply the trained weights to it. During training, the image was resized to ½ the original dimensions in order to increase training time. To make sure the input image was the same size, OpenCV was used to resize the image from the camera. The model returns a 2x1 vector of x and z position corrections, which can be published to the /desired/twist topic, to move the drone’s position to the desired position - above the target.

III. Results

Since our project involved training a drone in simulation and moving it to the actual hardware, our results can be divided as such.

As stated in section II, we ran multiple trials with different additional modules and hyperparameters to judge which would give us optimal results. The main difference in trials was the use of ICM. In the graphs below, you can see the results of the ICM trial. This trial ran for a little over a day and got to around ~700,000 steps.

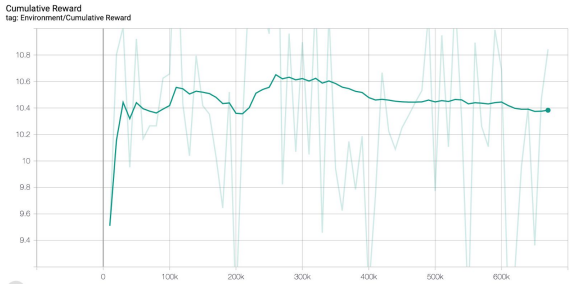

Fig. 3: ICM Trial Reward

Figure 3 above shows the cumulative reward graph from training with ICM. The reward gained corresponds directly with how well the drone followed the target. The drone gets a reward of 1 whenever it is a certain distance from the target. As shown in this graph, the reward converged around 10.5 pretty early on and did not move much from there. To determine why this is happening, we can look at the entropy graph.

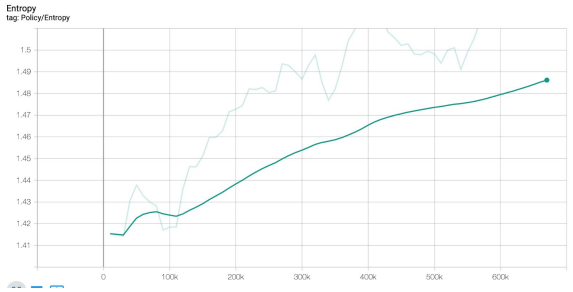

Fig. 4: ICM Trial Entropy

Entropy relates to the amount of randomness in the model’s actions. A higher entropy means a higher chance the agent will take a random move instead of following the model’s advice. This allows the agent to ‘break free’ and explore situations it may not usually find itself in. Over the course of training, we expect entropy to rise and then fall towards the end. Once entropy falls, the reward tends to

become more stable. This graph shows entropy rising slowly at the time the training was stopped. This tells us that the model was not done training at that time. However, the slow slope also shows us that training is taking longer than expected.

One more graph we can look at is the curiosity reward.

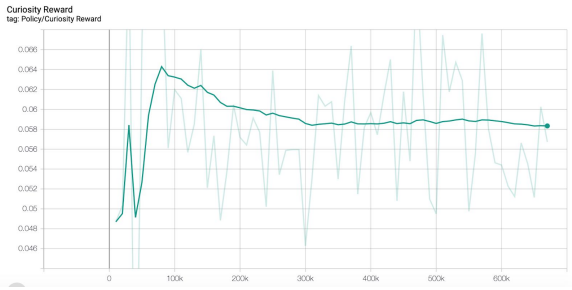

Fig. 5: ICM Trial Curiosity Reward

Curiosity reward is only calculated when ICM is used as an additional module. This corresponds to how 'surprised' the agent is to see a certain observation. The more surprised it is, the higher the reward. In ICM, the curiosity reward should rise and fall - same as entropy. However, in this case, the reward stayed stagnant, meaning the agent was not seeing anything new really. This stagnation tells us that curiosity may not be very useful in this case.

Our final trial did not use ICM. Instead, it used the default hyperparameters used by Unity ML-Agents. This model was run for over 1.8 million steps, lasting around 3 days of training time.

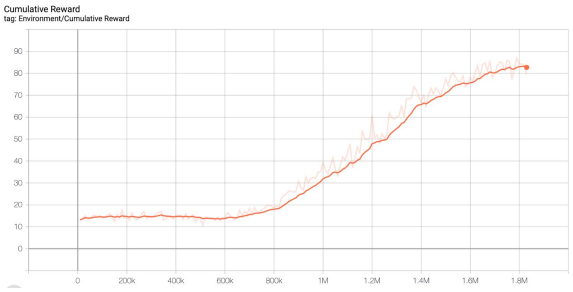

Fig. 6: Final Reward

The above graph shows the cumulative reward gained by our final model. This model stayed stagnant for around ~600,000 steps, but slowly increased from there. Upon convergence at the top, reward per episode was about 82 - much higher than previous trials. The reason for the stagnation in the beginning can be seen in the entropy graph.

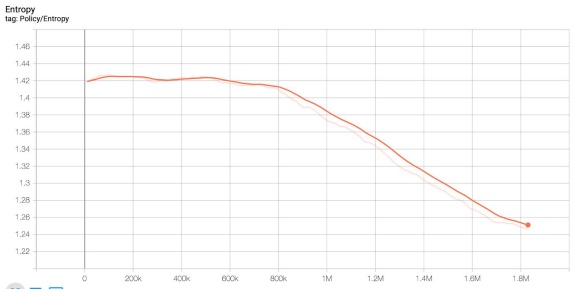

Fig. 7: Final Entropy

In the beginning, entropy was high and not moving much. However, around ~600,000 steps (the same time our reward increases) we can see entropy begin to fall. Towards the end, there is still some randomness in the model, but not nearly as much as the initial amount.

We can also look at the value loss of the model to see why the reward did what it did. The value loss is a metric used by the model to calculate the new weights

associated with the nodes. A high loss means there is more training involved.

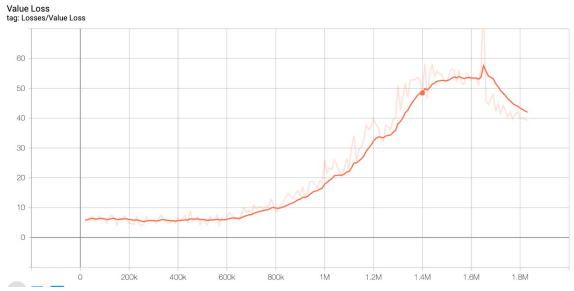

Fig. 7: Final Value Loss

Similar to the previous graphs, the value loss was stagnant in the beginning, meaning there weren't large steps in training. However, once the loss got larger, the agent began to learn more. When the loss decreases, it tells us the model has passed it's peak of learning, and is now learning less and less (since it has pretty much reached peak performance).

At the end of training, we are able to take our trained model and watch it perform in our simulated environment. A video showcasing this is available inside the GitHub repo within section IV. In the video, you can see two different panes. On the left, you can see a view from above the drone of the entire environment. On the right, you can see what images are getting passed into the model from the camera beneath the drone. In watching the right, you can see the target stays relatively centered beneath the drone. The hardest part for the drone is to know when the target will change directions. Since this is not known, you can see the target drift away from center upon a direction change. This is because the drone continues to follow the previous path until it notices the target has moved. Once it sees the target taking a different trajectory, it is able to correct itself and move above it once again.

When it came time to run the model on the drone, there was a lot of anticipation. We had written a ros-node that would take photos from the camera, and pass them to our model, then take the results from the model and pass them to the flight controller node again. We loaded up the node onto the drone, tried running it, and it immediately failed. The model we had written used TensorFlow for its machine learning networks, and the line importing it was throwing an error. We then spend the next several hours attempting to solve the error.

At first we tried making sure the drone was connected to the internet. Then we tried installing Tensorflow with pip. This one revealed another issue; pip could not find a suitable version of TensorFlow. We tried just about everything we could think of or find, but nothing could get TensorFlow to install. Drew even recreated the environment in a virtual machine to see if he could debug the issue. He managed to solve the problem on his machine but the issue remained on the drone. We found that the drone was running Python 2.7.12 while Tensorflow had not been compatible with Python 2 since version 1.2. As a result, we tried installing TensorFlow 1.2, but the error persisted. Fortunately, Alex found something online that talked about checking what version of python we were running - 32 and 64-bit. Upon further investigation, we found that the processor on the Pi was a 32-bit processor. We were physically unable to run

the models we had spent so much time and effort creating and training. Short of restarting from scratch, there was nothing we could do to progress further with the drone.

In relation to the image processing, the results from processing an image from the drone are shown in the image below:

The first image shows the original image of the hat overhead, as seen by the camera underneath the drone. The second image shows the processed image that would be given to the model. The image has been sized down by 1/2 and the red channel has been maxed out while all other channels have been zeroed.

IV. Resources

The source code relating to this project can be found on our GitHub repo https://github.com/atapley/SADrone. This repo holds our code for ROS, as well as the information from our trained models.

Inside of the '/Images' folder, you can find all related images and videos for our project. '/Bug_Errors' holds images used in the GitHub issue for Unity not running. hat_masked.png and input_hat.png are the images shown in this document relating to the output image from the camera and the output image from our ROS color masking code. The '/Tracker-<X>' folders hold the images from this document's results section relating to the Tensorboard graphs for two separate runs - Tracker-5 is the ICM trial and Tracker-8 is the final trial we ran without ICM. '/HeadTracker_Model' is a screen recording of our model running in a simulated environment.

'/ROS' holds the code used in the interface between the model and the hardware of the drone itself. '/Training_Model' has the code and outputs for the Unity model. '/models' holds the actual information relating to the final model and '/summaries' has the Tensorboard information for the corresponding graphs. '/Unity_Env_Files' holds all of the Unity files necessary for recreation of the environment and model.

To run and see the Tensorboard graphs related to the ICM and final trial, you have to open up a terminal and 'cd' into the '/Training_Model' folder. From here, you will run '**tensorboard --logdir=summaries --port=6006**'. When you go to

'localhost:6006' in a browser, you will see the Tensorboard page with all of the information from our two most important runs.

1. Related Work/Applications

DJI. Film Like a Pro: DJI Drone "ActiveTrack" https://store.dji.com/guides/film-like-a-pro-with-activetrack/

- The DJI ActiveTrack software is what inspired our idea in the first place. It allows a user to have their drone follow them autonomously.

D. Pathak. Curiosity-driven Exploration by Self-supervised Prediction. https://arxiv.org/pdf/1705.05363

- This work is the basis for ICM, one of the modules we used in training. While we did not use this for the final trial, it was used in a few.

J. Schulman. Proximal Policy Optimization Algorithms. https://arxiv.org/abs/1707.06347

- This work is the basis of our reinforcement learning model. PPO was used as the primary training algorithm.

S. James. Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks. https://arxiv.org/abs/1812.07252

- This work provided the grounds for our color masking in the ROS code. By making real-life look more like the simulation, the agent is able to cross the sim-to-real gap easier.

2. Main Resources

For learning how to work with the drone we primarily relied on two tutorials published by DuckieDrone. The first tutorial tutorial covered the assembly of the drone and was followed incredibly thoroughly (https://docs.duckietown.org/daffy/opmanual_sky/out/build.html). The second tutorial dealt with programming and interacting with the software on the drone and as a result, as used more as a reference than a guide (https://docs.duckietown.org/daffy/doc-sky/out/index.html). These tutorials occasionally skipped over potentially important information, but overall were an incredibly useful resource to have. Without the first tutorial, interfacing with the drone would have taken much, much longer.

For the *Training the Model* subtask, the biggest resource was Unity ML-Agents documentation, as well as their examples. Unity's blog posts and forum was helpful as well with fixing bugs or finding new ways to interact with the environment. Unity ML-Agents had a great doc for learning to create new environments and have an agent interact with it.

Youtube tutorials were used in order to find out how to load the Tensorflow graph into ROS. Surprisingly, there was not much documentation around this, so trial and error was necessary.

Websites such as https://www.pyimagesearch.com/2014/08/04/opencv-python-color-detection/ were useful in learning OpenCV for applying color

masks to images and detecting specific colors in images using Python and OpenCV.

For the *interfacing* subtask, the ROS wiki site was the primary resource, as it provided extensive tutorials on how to build ROS nodes, how to connect them and how to build workspaces. The OpenCV docs were also used to understand HSV images and how to work with masking.

3. Existing Packages/Code

For the *Training the Model* subtask, Unity ML-Agents 0.15.1 was the key package. This had all of the backend Tensorflow algorithms as well as the code to interface our environment with our training. Tensorflow 2.0.1 was also the main dependency used with the Unity ML-Agents package. Python 3.6.4 was the python used in the virtual environment for training. The Unity ML-Agents package has a lot of documentation around it, and I found (aside from the installation issues I had) their documentation was very helpful in creating the environments and being able to train your own models.

For the *interfacing* subtask, the primary packages were OpenCV and rospy. Rospy allowed for creating the ROS node easily in Python. OpenCV included all of the image-processing functionality needed for creating the input to the model.

V. Reflection

Santiago - This project was very challenging for several reasons, the least of which being my lack of materials. Personally, the hardest part was our inability to meet as a group. With each group member in different cities and states, it was physically impossible to meet. This resulted in each partner asking the other two to share what resources and knowledge they had via text or even email. Drew nor Alex could test their code without the drone, which I had, and as such if I was otherwise preoccupied with one of my other classes, progress would halt. Despite this we did our best to cooperate when we could, with Drew and Alex asking me to run various commands on the drone while I reported back on what had happened, making changes as necessary.

Ideally, I would have handled this project differently. The transition to online classes was unavoidable, but put a strain on my time that I was not used to. This meant that I was unable to dedicate as much time to this project as I would have liked. The physical assembly of the drone took me a week of work, and figuring out how to interact with the drone easily took another week or two. This meant that by the time that I had a working drone for Alex and Drew to start working with, three weeks had progressed, and we had even less time to work on the code. Under ideal circumstances, assembling the drone would have been an afternoon or two, and interacting with it would have taken a few hours at most. That way, the majority of our time could have been spent working on the code itself.

I have personally learned a large amount from this project, primarily related to working with ROS and machine learning. These were topics I was aware of going into the project, but had never worked with personally. ROS will ultimately prove to be

useful in other projects as I am currently working on a project to design and create a robotic shark through my club, Northeastern Robotics. Additionally, I ultimately want to work in robotics and my next co-op deals with writing code for autonomous applications, so it is important to gain as much experience as I can in an environment such as a class project.

Alex - Given the circumstances of the semester, we did not get to work on our first choice project, which was to have a drone revolve around a person. While similar, that project would not have included machine learning. As I have experience with machine learning already, I wanted to branch out and try something a little different. However, given the fact that we were all separated and could not work together, our project had to take a different turn. Getting to work with that aspect, and also being able to introduce multiple drones would be a cool extension off of this project.

My advice prior to starting this would be to start earlier. Training a machine learning model takes a lot of time, and there is not a lot of room for error when you get towards the deadlines. Since we had a lot of failed trials, getting an optimal working model so close to the deadline was a struggle.

Like I said before, I already had machine learning experience coming into this project, so I wanted to work on something a little more different. Unfortunately, this could not be the case given the semester. Due to this, the project was less fulfilling than I would have hoped it to be. However, I did learn more about Unity and their ML-Agents package, since the last version I used was 0.4.0b. It also gave an interesting view on how projects like this can work when all members are separated, which may be useful in the future.

In terms of future robotics projects to work on, my next co-op is in the field of machine learning for UAVs. This project gave me insight into what I may be working on and some issues that I can expect to show up.

Drew - I was really excited for this project at the beginning of the semester when it was introduced. Unfortunately, of course, the switch to online classes completely changed the scope and goals of the project. I was the least experienced person on the team, and I had been hoping to work on all aspects as a learning experience. Because we were forced to split off pieces of the project to work on, it made sense for my partners to take pieces that they were more familiar with. Since interfacing the model with the drone requires both the model and the drone, I had a strangely disconnected task for a good portion of the project and resorted to learning as much about ROS as I could.

I think it would have been extremely beneficial, regardless of the special circumstances, to start the project much earlier in the semester. I think that it would be useful to spread out the project to allow for more flexibility. I learned a ton (probably more than necessary) about ROS using Ubuntu and openCV. I think despite all of the issues that arose, I am really happy with how much I learned working on this

project. I think in particular something I would tell myself is to be careful at the start to look at dependencies and make sure everything is set up correctly, otherwise you think there is an issue with your code when it is really an incorrectly imported library. I saw somewhere while working on the project that “Robotics is a decathlon” and after this project I couldn't agree more. The amount of different pieces that must work together, from soldered wires to machine learning is pretty insane.