How Well Can a Drone Play Frogger?

Christopher Klammer
Master's of Computer Vision Student

***Abstract*—This project focuses on the journey of teaching a drone to take off and navigate to a goal using Deep Reinforcement Learning agents. The goal is to explore a plethora of methods in practice, play around with modelling a reward function, and making incremental improvements. By the end of this effort, the goal is to have a semi-functioning agent to navigate from point A to point B while having a plethora of future work to explore!**

I. Introduction

Autonomous navigation is one of the more direct applications of reinforcement learning to the world of robotics. Autonomous navigation, in this context, is purely the task of a robot incrementally taking actions towards a location or destination in an environment without human intervention. In this project, I look to explore the autonomous navigation problem for a drone and ascertain which method(s) work best for the task at hand. In this case, the task will start as learning to fly, evolve into learning to avoid obstacles, and always focus on attempting to reach the goal as unscathed as possible.

II. Environment and the Task

The environment used is the gym-pybullet-drones environment. This is a Pybullet-based gym environment that is able to work with single and multi-agent reinforcement learning with quad-copters. The drone model used in this experiment was a small form factor C2F drone. In previous work, we analyzed the take off task. However, since that was fairly easy, we transition to to a full navigation task. The starting point is at the origin and the table is 5 meters forward, 1 meter higher, and 1 meter to the right. Now, in the first part, the environment is setup to attempt to fly from one table to another table. In part two of the modifications, we will add some additional obstacles. Namely, six hovering drones and a large (static) soccer ball. Our goal by the end is to start building a solution that can ”walk before it runs” and somewhat navigate without obstacles and see how the drone reacts when obstacles and a camera are added.

A. The Drone

The action space of the drone is continuous and is described by:

1) F_{thrust}: The thrust along the drone's z-axis
2) T_x: The torque around the x-axis
3) T_y: The torque around the y-axis
4) T_z: The torque around the z-axis

More formally:

$F_{thrust} = \sum_{i=1}^{4} F_i$
$T_x = L * (F_2 - F4)$
$T_y = L * (F_3 - F_1)$
$T_z = \sum_{i=1}^{4} M_i$

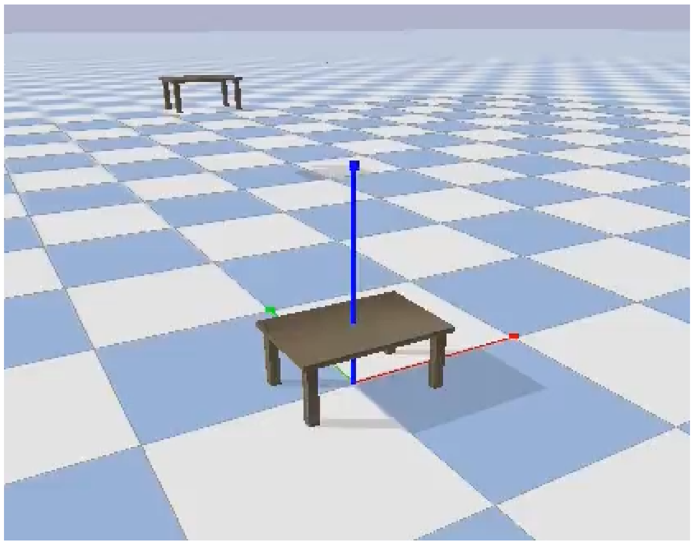
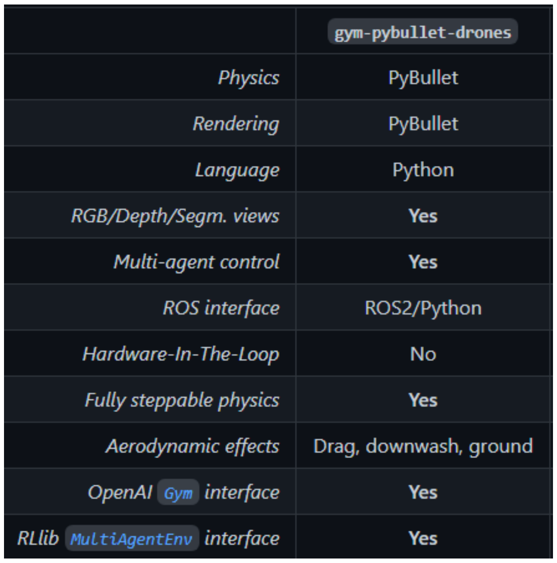


Fig. 1. The environment without obstacles



	gym-pybullet-drones
Physics	PyBullet
Rendering	PyBullet
Language	Python
RGB/Depth/Segm. views	**Yes**
Multi-agent control	**Yes**
ROS interface	ROS2/Python
Hardware-In-The-Loop	No
Fully steppable physics	**Yes**
Aerodynamic effects	Drag, downwash, ground
OpenAI `Gym` *interface*	**Yes**
RLlib `MultiAgentEnv` *interface*	**Yes**

Fig. 2. Support of Gym Pybullet Drones

B. Environmental Setup

Within the environment, there will be three different sections:

1) The Start
2) The Abyss
3) The Goal

The start begins on a table before the abyss, the drone with

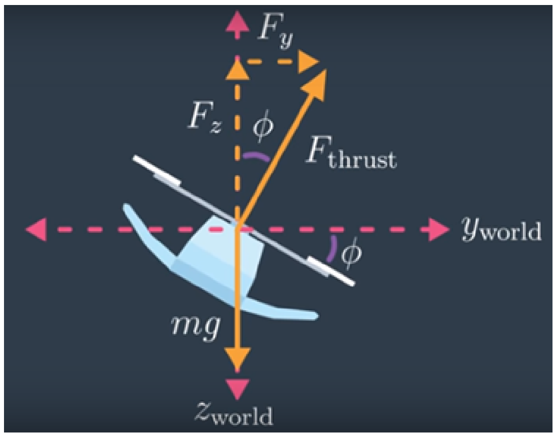


Fig. 3. Diagram of the Forces in the World and Body Frames

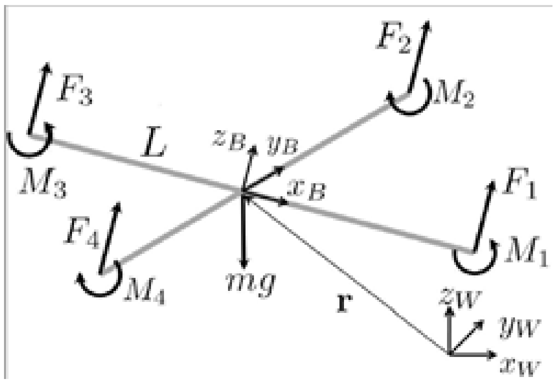


Fig. 4. Forces and Moments on a Drone

the learning agent will be randomly placed somewhere on the starting platform before taking off towards the goal.

The abyss will either be empty or will consist of a few dynamic obstacles (drones) on a somewhat fixed path. They will be programmed to hover with some variance and might have some unpredictable elements to them. Additionally, there is one static obstacle in the form of a large soccer ball.The idea is to build a sense of progression when evaluating agent performance. I think this will also gauge how difficult the problem is. For example, it may be possible to have the drone navigate static obstacles but way too challenging to handle the dynamic obstacles. The failure or success in this project lies in analyzing the trade-offs and viability of certain algorithms in certain situations and understanding why rather than always flying the perfect path.

The goal will be a similarly sized table after the abyss, arriving near the platform would be considered a success. Landing a drone perfectly on a platform could likely be considered an extension of this work. The focus will be more on the obstacle avoidance and navigation, acting as a baseline for future work.

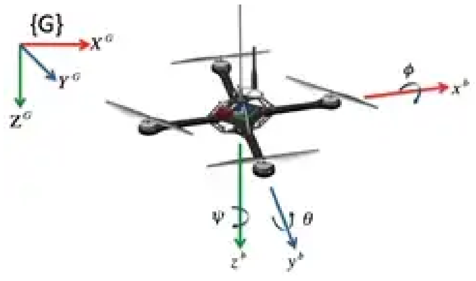


Fig. 5. Orientation of the Drone

III. The Reward Function

A. Hovering and Altitude

The reward function for taking off will involve a higher reward when the drone is close to a hovering state at the desired altitude.

The reward will include a higher reward when the drone is closer to the desired altitude:

$$R_{hover} = k_1 * exp(-(z_{drone} - z_{goal})^2) \quad (1)$$

Here k_1 is a constant that helps decide how much to weight the reward of the hovering. In this work, k_1 was set to 2.

B. Stability

The idea here is to have the drone slowly gain stability first when the drone is learning to navigate. Early on, the drone will be crashing very often. Just being able to stabilize in the beginning will increase the reward and hopefully offer some passive stability. It is chosen that the norm of the angular velocity of the drone would decide how stable the drone is. That is, the lower the norm of angular velocity vector, the higher the reward will be. Meanwhile, a high value for the angular velocity norm will yield negative rewards.

$$c = \left\| \begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \right\| \quad (2)$$

- $\dot{\psi}$: Change in Yaw
- $\dot{\theta}$: Change in Pitch
- $\dot{\phi}$: Change in Roll

$$R_{rotation} = -k_4 * exp(-c^2) \quad (3)$$

C. ***Navigation***

This is the main goal of the work, we wish to try to get as close to the goal as possible without taking too long of a path. Here we discuss the terms of the reward including:

- Distance from the Goal
- Penalty for crashing or going too high
- Peanlty for wandering too much

1) Distance from Goal: Here, we want to incentives in the form of an exponentially increasing reward for when the drone approaches the goal. I chose to make this exponentially increasing because precision is more important the closer the drone gets to the goal.

$$dist_{goal} = \left\| \begin{bmatrix} x_{drone} \\ y_{drone} \end{bmatrix} - \begin{bmatrix} x_{goal} \\ y_{goal} \end{bmatrix} \right\| \tag{4}$$

$$R_{goal} = k_2 * exp(-dist_{goal}^2) \tag{5}$$

In this work, we chose $k_2 = 10$ as getting to the goal is extremely important.

2) Wandering Penalty: There is a constant negative reward for each timestep travelling towards the goal. This will hopefully enforce a straighter path.

$$R_{wander} = -k_5 \tag{6}$$

Where k_5 was 0.1 in the work.

3) Crashing Penalty: Crashing is the worst possible thing that can happen to the drone. In the real world, UAVs cost thousands or even tens of thousands of dollars. Therefore, a large penalty for crashing is in order. In this work, we define crashing as: colliding with an object, going too high in the environment (assume there's a ceiling), and getting too low in altitude. Overall, this should help the drone be more conservative and stay in the air longer.

$$R_{crash} = -k_3 \tag{7}$$

Where k_3 was 10 in the work

IV. Modifications

A. Rewarding Heading Angle

Up to this point, the drone makes its way to the goal, however, it really does not seem to do so in a direct way. At this point, it seems like a prudent idea to add a term to our reward function for when the drone goes off course from the destination. Doing so may result in a more direct path to the goal. Specifically, we can mathematically formulate this by looking at which direction the drone is heading and comparing that to the direction the goal is in.

This is calculated as:

$$G_{direction} = pos_{goal} - pos_{drone} \tag{8}$$

$$\theta_{error} = arccos(\frac{v \cdot G_{direction}}{\|v\| * \|G_{direction}\|}) \tag{9}$$

$$\begin{aligned} R_{heading} &= min(\frac{1}{2\theta_{error}}, 2) && \text{if } \theta_{error} < \tfrac{\pi}{8} \\ R_{heading} &= -2\theta_{error} && \text{else} \end{aligned}$$

$$R_{heading} = k_6 * R_{heading} \tag{10}$$

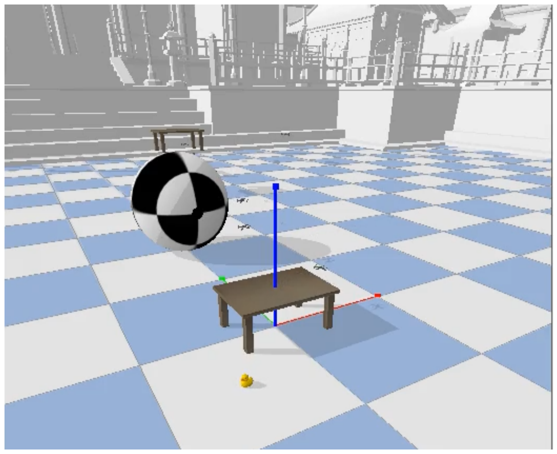


Fig. 6. Environment with the Soccer Ball and Drones Added

We only give positive rewards if the drone is pointing in the direction of the goal with an error angle of maximum $\frac{\pi}{8}$. Anything else will yield a negative reward. To see the results of this modification, reference the "Analysis" section.

B. Avoiding Obstacles and Adding RGB Camera

In this section, we have now added obstacles to the environment. There are drones lined up in a straight line blocking the goal. Now, the goal is to navigate to the goal while avoiding the obstacles. As stated before, we choose to have 6 drones that are equidistant between the starting point and the goal. These drones are hovering with a given force to counteract gravity. However, simulations are imperfect and they occasionally veer off in different directions. Lastly, a large static soccer ball is added to completely block part of the bath for the drone.

For this efort, I experimented with an RGB camera in addition to the drone kinematics to represent the state. The hope is that the neural nets start to recognize certain patterns in the pixels before collisions and start to influence the actions. I expect this to act as a baseline for navigating with obstacles, not to perform near optimally. In this part, we choose some of our best performing models (with and without modifications) and try it with the new state space.

V. Results & Analysis

A. Random Agent

Here, a random agent is used to measure trivial performance. That is, when acting randomly in the environment, how good is the drone behaving and getting rewarded. We will use this performance to compare against our other methods.

1) Video of a Random Agent: **Part 1 of: https://www.youtube.com/watch?v=DG5u-0viQxk**

2) Performance of a Random Agent: The takeoff task was conducted with a random agent. Each colored line represents a different trajectory. Each trajectory is played out until:

- The drone crashes
- The number of timesteps have elapsed (10 seconds)

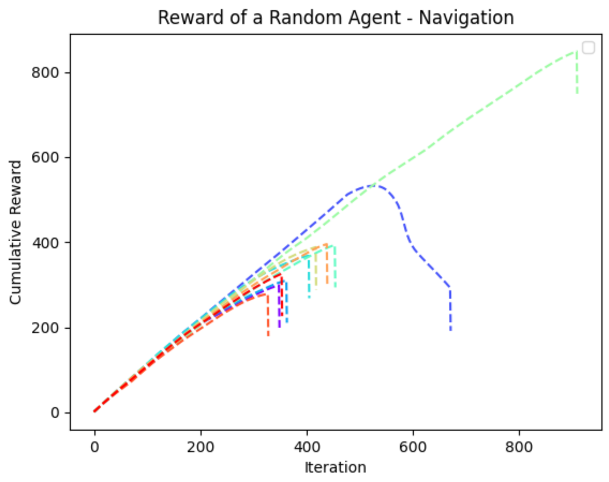


Fig. 7. Random Agent Performance for the Navigation Task

B. *PPO Agent (On-Policy)*

1) Video of the PPO Agent after 1e6 Training Steps: **Part 2 of: https://www.youtube.com/watch?v=DG5u-0viQxk**

PPO stands for Proximal Policy Optimization. This method is centered around using advantage estimation in order to improve the policy based on data generated by rolling out only the current policy. PPO calculates both an actor and a critic. The actor is trained according to the advantage of taking a current state and action. This method calculates the advantage by using the actor and the critic in order to find how much better than average actions are. For this, a separate critic network is trained to be able to estimate the value function of our state space, will be trained on the value function estimate compared to the rewards to go for a current state in the trajectory. Overall, this method is one of the state of the art methods today and performed well on our task.

2) Parameters Used::

- train steps: 1e6
- learning rate: 3e-4
- n steps: 2048
- batch size: 64
- n epochs: 10
- gamma 0.99
- gae lambda: 0.95
- clip range: 0.2
- clip range vf: None
- normalize advantage: True
- ent coef: 0.0
- vf coeff: 0.5
- max grad norm: 0.5

3) Training: PPO while training sees a strong increase in reward all the way through the iterations. The dots are the episodic rewards at each iteration and the line attempts to follow the trend of the episodic rewards to a best fit line. At the end, there is a general convergence for the policy and starts to approach 5000 for its episodic rewards which is the best out of the bunch.

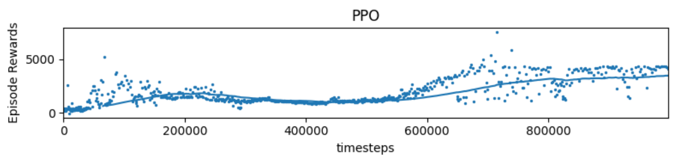


Fig. 8. Training Rewards during 1e6 Training Steps

4) Performance: The PPO agent does pretty well with taking off and staying at a modest altitude. However, it definitely goes much higher than necessary before going forward than veering off to the right. At this point, it seems as if the agent is really focusing on not crashing and starting off higher alleviates the risk of crashing. However, there is much improvement and would prefer the agent reaches or speeds past the goal in addition to not crashing. Still, this is a big improvement over the random agent. One thing that might

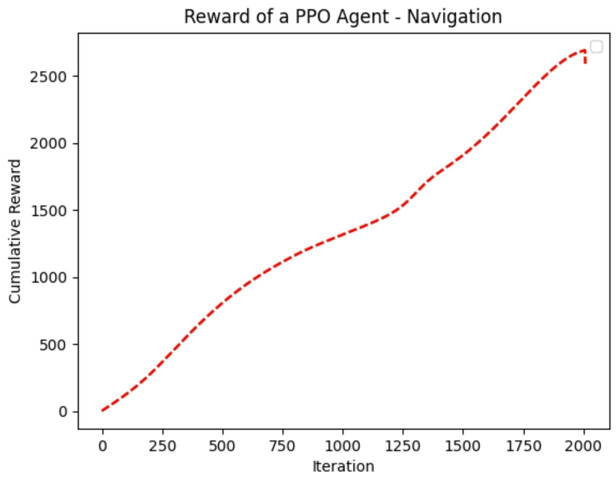


Fig. 9. Navigation Performance after 1e6 Training Steps

help this behavior is making crashing even more costly than it is -10 is not a big penalty once you have a cumulative reward of over 2500.

C. SAC Agent (Off-Policy)

SAC stands for Soft Actor Critic. SAC uses double Q-learning and utilizes a replay buffer. SAC is very similar to TD3 except that SAC learns a stochastic policy policy and there is no target policy smoothing. Another novel thing SAC brings to the table is that it includes entropy in its cost function. This method looks to maximize expect return but also looks to maximize entropy to a certain extent to encourage exploration. This generally helps learning in the long run but may cause it to suffer early on.

1) Video of the SAC Agent: **Part 3 of: https://www.youtube.com/watch?v=DG5u-0viQxk**

2) Parameters Used::

- learning rate: 3e-4
- buffer size 1000000
- learning starts 100
- batch size 256
- tau 0.005
- gamma 0.99
- train freq: = 1
- gradient steps: int = 1
- action noise = None
- ent coef: ”auto”
- target update interval: 1
- target entropy: ”auto”
- use sde: False
- sde sample freq: -1
- use sde at warmup False,

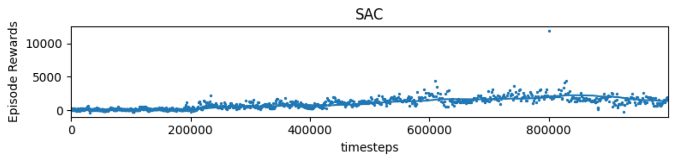


Fig. 10. Training Performance of the SAC algorithm over 1e6 trainsteps

3) Training of the Agent: For training, there was a consistent increase in episodic rewards. The episodic rewards started at 0 and ended at around 2.5 to 3 thousand. Overall, it starts to learn the task but needs some more tuning to move forward.

4) Performance of the SAC Agent: The SAC algorithm struggles more than the PPO algorithm. It hits a 2000 cumulative reward when using the best model during training which is inferior to its PPO counterpart. Also, this is visible because it veers to he right even more than the PPO algorithm. Again, the takeoff is not a problem and is fairly similar to PPO. However, SAC is certainly far from converging and could take a straighter path to make the navigation must simpler. Going forward, I feel some sort of modification must be made to the reward function in order to increase performance consistently. I discuss some later in the work, however, one short term fix would be to increase the stability weight or increase the weight of crashing to be more conservative.

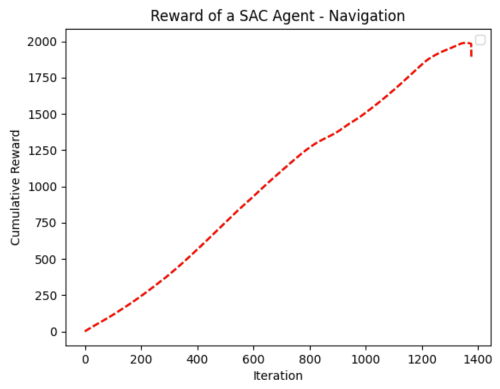


Fig. 11. SAC Evaluation Performance after 1e6 trainsteps

D. *Model-Based RL with On-Policy Data*

Here, we attempted to use the work from Homework 4 in order to test how a model-based RL agent would do in the current environment. This particular algorithm uses model-based RL techniques while utilizing on-policy data. We do this by sampling actions with either a random or CEM based strategy using the current policy, then, the first action is used to continue the trajectory. For this method, we are learning a dynamics model to maximize the return and use that dynamics model in order to sample more optimal actions. Int this work, we looked to utilize the last part of the homework to receive the best change of optimal performance

1) Parameters Used::

- Number of Iterations: 15
- Batch Size: 5000
- Batch Size Initial: 5000
- Train Batch Size: 512
- Eval Batch Size: 400
- Number of Layers: 2
- MPC Horizon: 15
- Ensemble Size: 5
- Size: 250
- Episode Length: 500
- Agent Train Steps per Iter: 1500
- MPC Action Sampling Strategy: CEM
- MPC Action Number of Sequences: 1000
- CEM Iterations: 4
- CEM Number of Elite: 8
- CEM Alpha: 1.0
- Learning Rate 1e-3
- Add SL Noise: True

2) Video of Results: **https://www.youtube.com/watch?v=XB2SSQaMKco**

3) Performance of the Model-Based Agent: Note here that we are using average reward instead of cumulative reward over one iteration. In our experiment, we reached a peak average evaluation return of around 500. Meanwhile, the random agent has a cumulative reward that hits 500 but its average reward is much closer to 200. Thus, we can say, at least, the model starts to learn something and does much better than average. Looking at the evaluation returns, it really doesn't seem to do much better as the iterations run on. There are a few spikes in the graph, however, it likely did not have enough iterations to learn a very good dynamics model in such a complicated 3D environment. Furthermore, from the video, it is clear the agent definitely has issues converging and stay up in the air for a long time. Unlike the previous methods, this method is looking to stay pretty low in the sky. By the end, the drone starts to get pretty close to the table but still definitely does much worse, from a rewards setting, compared to the online and offline methods.

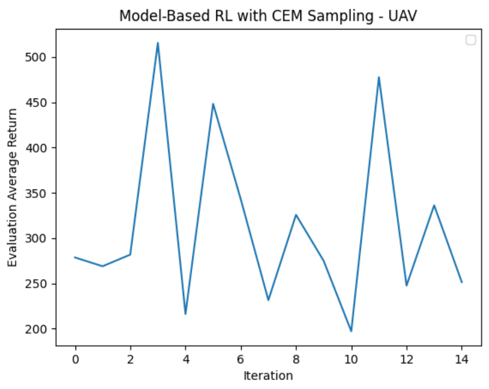


Fig. 12. Average Evaluation Return after each Iteration for the MB Agent

E. Results of Modification 1 - Penalization for Heading Angle

1) Video of Modification 1: **Part 4 of: https://www.youtube.com/watch?v=DG5u-0viQxk**

2) Results and Analysis: As stated before, we had decent but not amazing performance because the UAV was not taking a straight or optimal path to the goal. Here, we show the effect of changing the reward function (mentioned above) affected the results of the task at hand. We train our modification on the PPO algorithm because it is the quickest to train and also is the best performing thus far.

Below, we can see that our agent performs better than the random agent. However, we also see that this reward function is much more punishing. The variance of cumulative reward is much higher which means that our rewards are much more sensitive to the direction the UAV heads.

With this modification, the new agent receives very positive reward then falls off half way through the trajectory. This is because our agent takes a straight line right to the goal but overshoots the goal. However, the UAV has greatly benefited from this reward function change because it wastes less time exploring non-optimal trajectories and routes. However, some care should be taken in the future because right now the UAV has no capability to land on the other table, just get to the other side of the abyss. However, the episode lengths are capped in training and in evaluation, they are not, just until the drone crashes so that is why the results look this way. However, realistically, this would pose as a safety concern and would likely result, at best, with a broken drone. Therefore, more work needs to be done before even attempting to transfer the algorithm to the real world.

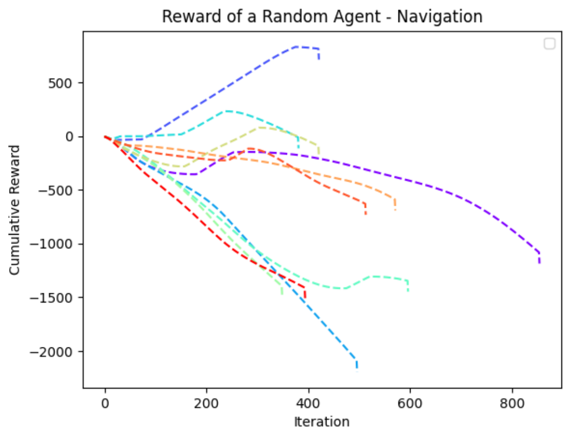


Fig. 13. Random Action Performance with the New Reward Function

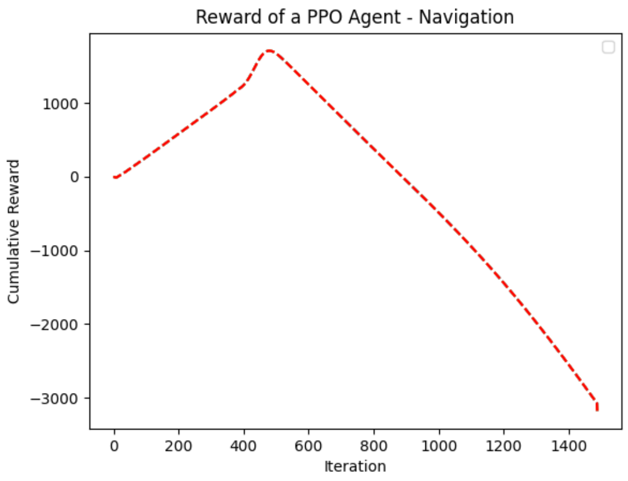


Fig. 14. Evaluation Rewards after Training

F. Results of Modification 2 - Adding Obstacles and RGB Inputs

1) Video of Modification 2: **https://youtu.be/-WqwEsbRQZ4**

2) Results and Analysis: In this modification, we made our task much harder, seeing if we can still reach the goal if we add lots of obstacles to the environment. Meanwhile, we allow the RL agents to use multiple streams of input data (a RGB camera and kinematic information). Overall, we immediately see that the task is much harder. As we can see from the figure below, the random agent performs horribly and receives almost exclusively negative rewards, always crashing as a result.

When training, we trained a PPO algorithm with the same parameters as mention above, only adding the RGB input. This, in turn, helped build some concept of obstacle avoidance and is better than random agent performance. Looking at the rewards after training, there is a small blip of positive performance before the drone loses controls and crashes. We can also see this where the drone starts to move towards the goal before swerving to the right and crashing. When adding the improved reward function from modification 1, we still dramatic improvement. Even though the reward plot doesn't show it, the drone actually moves forward then starts to roll to the left to dodge the ball but misses. I believe with more iterations, it would find a way to avoid the ball and get further. Overall, I believe that the RGB input is helping but is not learning enough meaningful items for decision making.

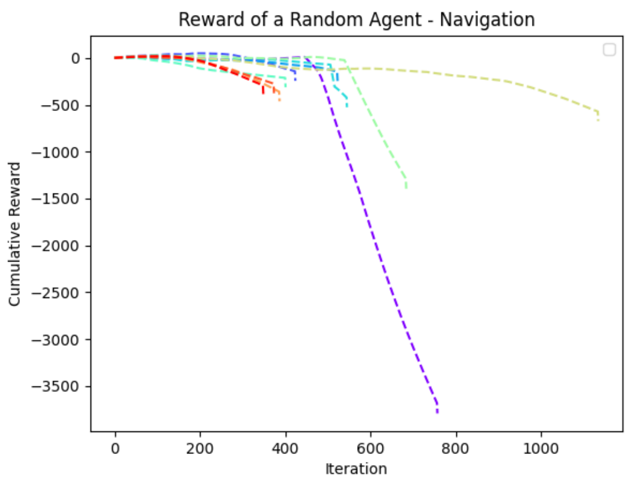


Fig. 15. Random Agent Rewards with Obstacles

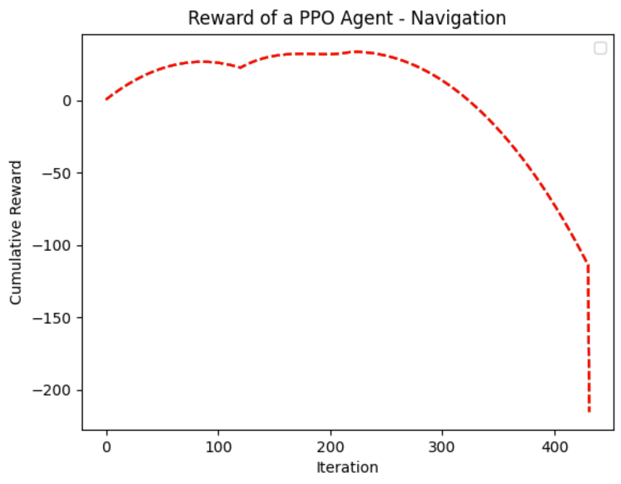


Fig. 16. Result of the best model when training on 1e6 trainsteps with kinematic and RGB state information

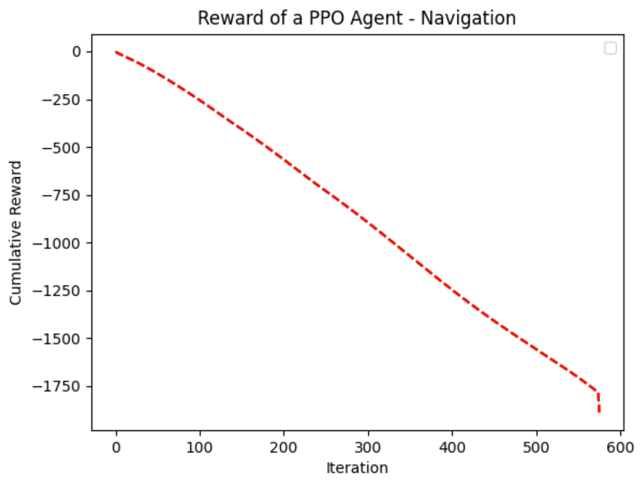


Fig. 17. Result of the best model when training RGB+kinematic state information using the reward function from modification 1

VI. Conclusions

In this work, we started from the ground up to assess different DRL agents capability to perform the UAV navigation task from point A to B. We started with a no obstacle environment and we saw reasonable results of being able to takeoff, stabilize, and navigate the environment towards the goal which was heavily improved with modification 1. We saw that the SAC and the PPO agents both performed well and were able to get close to completing the task with only 1e6 training iterations. Meanwhile, training a model-based agent with on policy data did not yield great results but still performed better than random. Subsequently, performance dramatically increased with modification 1 which added a term to penalize the drone heading away from the goal. Quickly, this task became increasingly simplified. Next, we purposed to add obstacles and other hovering drones to attempt to make this challenge more difficult. We saw this quickly with the random agent having abysmal results. However, once using the altered reward function from modification 1 and the RGB inputs, we could see agent start to attempt to avoid the obstacles and action perform some learning. This work is far complete and has so many interesting directions to go in future work.

VII. Future Work

In future work, we'd first like to address the agent's ability to land at or near the target. Once we finally "solved" the task of navigating to the goal, we were still left with the drone moving past the goal without slowing down. It is possible more iterations could solve this issue but I believe limiting the drone's maximum speed and tuning the reward function is critical for this improvement.

It would also be prudent to revisit the model-based approaches and see if we can learn a dynamics model with the reward function from modification 1. I believe with this simplified environment, the rewards are more straightforward and a stricter possible state space is implicitly created. Specifically, trying the iLQR method while also rerunning the model-based experiment with the homework implementation would be interesting to try.

For obstacle avoidance, reducing the speed would also help. The drone is moving so fast that any attempts to actuate the motors to avoid the obstacles is far too late. Slowing down the speed of the drone will allow the drone more time to turn and go around static and dynamic obstacles. Additionally, adding depth camera and segmentation mask information I think would heavily improve the performance of obstacle avoidance. These representations are definitely more direct than a simple RGB input is. Thus, I believe doing some experiments with these different inputs would yield a much better result and allow the drone to avoid obstacles and get much closer to the goal. Ultimately, there are a few options here, first we would like to try the raw inputs by themselves as a baseline. Then, there could either be some fusion or intermediary information/engineering that could help utilize these inputs to improve performance.

A. Link to the Videos

Link to the Videos: https://www.youtube.com/@roboklamz1602