RL approach to Multi-Rotor Failure in Quadcopters

Nikhil Badami*, Sridhar Reddy Velagala*
badaminikhil@gatech.edu, svelagala7@gatech.edu
* indicates equal contribution

***Abstract* – Quadcopters are susceptible to a variety of failures, amongst which are rotor failures. In the presence of rotor failures, the quadcopter should be able to either complete its objective or land safely. In this paper, we study the feasibility of using deep reinforcement learning to develop a controller capable of controlling a quadcopter under a variety of failure conditions. Specifically, we train policies to control a quadcopter under Single-Rotor Failure, dual opposite rotor failure, and dual adjacent rotor failure. We also compare policies found by Soft Actor-Critic and Proximal Policy Optimization to determine which algorithm is best for this task. We show that a robust controller is possible using deep RL for the Single-Rotor Failure case as well as the Dual-Rotor Failure (opposite) case. We were unable to train a controller for Dual-Rotor Failure (adjacent) and investigate why we were unable to do this.**

I. Introduction

Quadcopters are susceptible to a variety of failures during flight. Amongst these possible failure cases is the possibility of rotor failure, or the loss of power to one or more rotors. In such scenarios, we want to prioritize two things: completion of the objective if possible, or, safe landing so as to avoid damage to the quadcopter if objective completion is not possible.

In this project, we explore the feasibility of developing a model-free controller using deep reinforcement learning (RL). We investigate the ability of different algorithms to train controllers for various failure conditions. We begin by implementing the baseline in [1], which uses the Soft Actor-Critic algorithm (SAC) [6]. We then extend this work to Dual-Rotor Failures and evaluate different policies trained using both Soft Actor-Critic as well as Proximal Policy Optimization (PPO) [7].

Our key contributions are as follows. We were able to successfully implement the baseline paper, which shows that SAC can be used to train a policy to control a quadcopter under Single-Rotor Failure. We also show that deep RL can be used to train a policy to control an agent under Dual-Rotor Failure (opposite) conditions. While unsuccessful in developing a controller for Dual-Rotor Failure (adjacent), we present results and discuss the challenges in developing a controller for this scenario. Finally, we also investigate the feasibility of using PPO to train policies for these scenarios.

II. Related Work

Several previous works have established the feasibility of using reinforcement learning not only to control drones in general, i.e., no rotor failure, but also with rotor failure. [2] established the feasibility of developing a controller for a quadcopter using reinforcement learning. They trained an RL agent to control a quadcopter in flight and achieved greater performance than a traditional model-based method. [3] developed RL based controllers to make a quadcopter hover in place with Zero, Single, and Dual-Rotor Failure scenarios. [1] Extended the work of [3] and developed an RL controller capable of controlling a quadcopter with Single-Rotor Failure through hovering, landing and path following experiments.

III. Methodology

A. Environment and Training

We used the gym-pybullet-drone environment [4] to run our experiments. As the name implies, this is an environment compatible with OpenAI's popular Gym framework for training reinforcement learning agents. For the baseline and both alphas, we run three experiments. First, we train an agent to hover in place under the specified failure conditions. Second, we train an agent to navigate to a particular hover location after being initialized to a random position in a 2x2x2 cube centered around origin. Note that this experiment is different from the one defined in the baseline where the agent is specifically trained to land. We believe this will make the controller much more robust to movement in all of the 3-dimensional space. Finally, we tested the drone agent navigate a pre-defined square path.

In order to simulate rotor failure conditions, we manually set some rotor values to 0 RPM. For the baseline, we set the 2nd rotor to 0 RPM. In alpha1, we set the 0th and 2nd rotors to 0 RPM and for alpha2, we set the 1st and 2nd rotors to 0 RPM. Like in [1] we assume the presence of a failure detection system in the drone in a real world setting that would automatically switch controllers if a failure in the drone's rotors is detected. Thus, we do not focus on that aspect in this work and focus entirely on training the network with rotor failure. Each agent is trained for 5 million time steps unless specified otherwise. We used the agent trained to hover in place to pre-train the other two experiments. We implement our network using Stable Baselines [8], specifically, we use a four layer MLP with the following nodes in each layer: [512, 256, 256, 128].

B. MDP Formulation

1) Reward Functions: We experimented with three different reward forumlations. The equations are shown below.

$$r_d = -\sqrt{\bar{x_t}^2 + \bar{y_t}^2 + \bar{z_t}^2} \tag{1}$$

$$r_t = r_1 + r_2 \tag{2}$$

$$r_1 = -c_1 \tanh\left(c_2 \sqrt{\bar{x_t}^2 + \bar{y_t}^2 + \bar{z_t}^2}\right) \quad (3)$$

$$r_2 = -\sum_{i=1}^{4} \frac{|\tau_{t-1}^i - \tau_t^i|}{c_3} \quad (4)$$

Equation 1 was used as our initial reward curve when training. We eventually switched to using equation 2 because of how equation 1 behaves near the hover point. Consider the following plot.

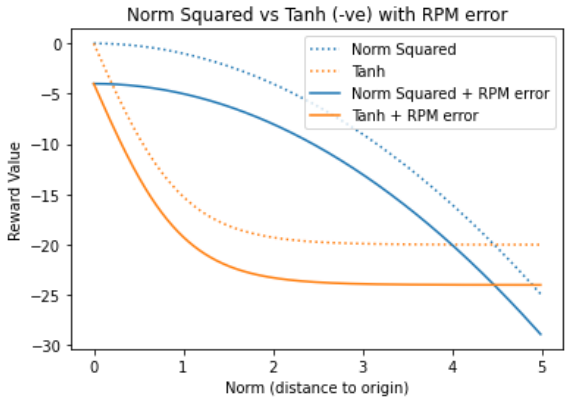

Fig. 1. Variations in Reward Functions. Equation 1 (dotted blue), Equation 3 (dotted orange), Equation 2 (solid orange)

The dotted-blue curve is equation 1 and the dotted-orange curve is equation 3. As we can see, equation 1 does not have a very steep slope as the agent approaches the desired hover point, in this case, the origin. In practice, we found that this led to the agent settling into local minima and hovering away from the hover point instead of settling at the point itself. We believe that the steeper slope of the function after applying tanh helped incentivize the agent to reach the true optimum for hovering.

We also included equation 4 in our reward to help minimize the rapid changes in RPM values sent to each rotor. This is added to the TanH reward component to make up the final reward function (Equation 2). c1, c2 and c3 are constants set to 20, 1.0 and 0.5 respectively. τ_t represents the RPM of the rotors at time t.

2) Observation Space: At each time step the agent receives the displacement of the drone from the desired hover point, the roll pitch and yaw of the drone, the ground velocity of the drone and the angular velocity of the drone. All parameters are triplets corresponding to the three dimensions.

3) Terminal Conditions: Each episode is run for 5 million time steps. There are no early stopping conditions and the episode will only end once all iterations are completed.

IV. Baseline Results

The goal of the paper is to investigate the feasibility of using different RL algorithms to train controllers to handle rotor failure in drones. In this section of the paper, all agents were trained for 5 million time steps using SAC algorithm and uses the reward function described by equation 2 unless otherwise specified. As we will see in the sections below, we were able to successfully recreate the baseline.

A. Baseline Results - Hover

We were successful in reproducing the baseline results, specifically in regards to hovering. In this experiment, the agent was initialized at the origin and then trained to hover at the origin with a single rotor disabled. The two graphs below show the reward acheived by the baseline paper and that of our agent.

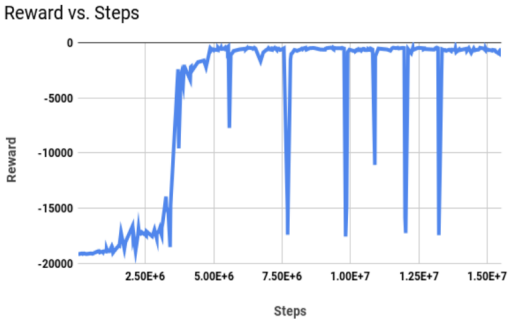

Fig. 2. Reward vs Steps plot from Baseline Paper [1] for a Single-Rotor Failure Scenario

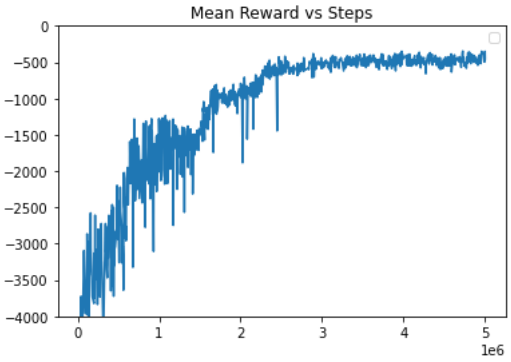

Fig. 3. Reward vs Steps plot from this Paper for a Single-Rotor Failure Scenario

As can be seen from the figures 2 and 3, our reward plot indicates our agent was able to achieve a similar reward curve as the baseline agent. There is significant noise in the reward earlier on in training which we attribute to differences in simulation. The early convergence of our agent is attributed to using the pre-trained model for this experiment and the reward scaling differences are due to the different constant we used for our reward function. To view a video of our agent, please see the "baseline-hover" video in the supplementary materials folder submitted with this paper.

B. Baseline Results - Navigate to Point and Hover

As can be seen in figure 4, the agent is able to successfully navigate from random point in a 2x2x2 space to the desired hover point (origin in this case). The initialization points and drone path are shown in red and the hover point is shown in green. The agent is successfully able to navigate and stabilize at the green point. Please see the video "baseline-navigation" in the supplementary materials folder for more information.

C. Baseline Results - Navigate Square Path

The final experiment we implemented for the baseline was having the agent navigate a square path. Figure 5 shows the baseline agent's behavior along the path.

As can be seen, the agent is able to follow the path very closely. The deviation from the path is on the order of centimeters, and can be attributed to the inherent instability of the drone's flight due to rotor failure. Please see the video "baseline-square" in the supplementary materials folder for more information.

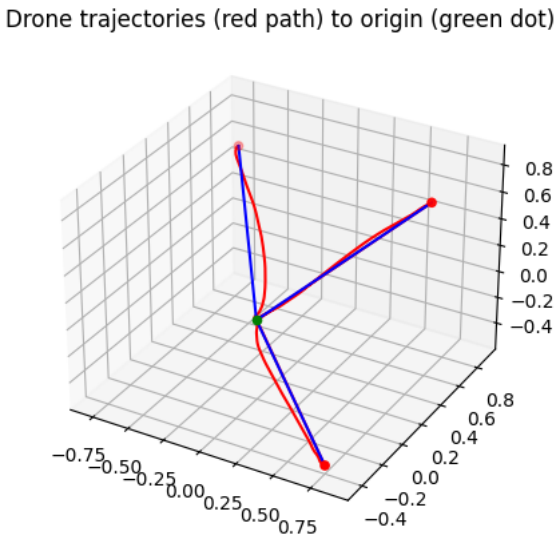

Fig. 4. Drone trajectories for the Single-Rotor Failure Scenario starting at three random points and moving to origin

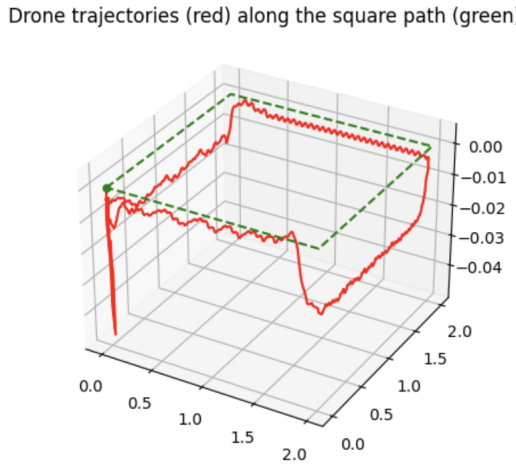

Fig. 5. Drone trajectory (Single-Rotor failure) moving along a square path

V. Discussion - Baseline Results

Figure 6 shows the normalized RPM values that are predicted by the agent's actor network. RPM 2 is the rotor that will be manually set to 0. The value of RPM 2 (2nd rotor) is set to 0 RPM after taking the output of the neural network, which is why the reported value is not constant at 0 in this plot. In our drone's configuration, RPMs 0 and 2 (or 0th and 2nd rotors) and RPMs 1 and 3 (or 1st and 3rd rotors) are opposite of each other. An important thing to notice is what happens in the earlier time steps when the drone initially tries to stabilize itself. RPMs 1 and 3 seem to have behavior that mirrors each other, indicating correct rotor functionality. RPM 0 has a much noisier initialization, indicating that it is reacting to RPM 2 being offline and is trying to adjust accordingly. While the output for RPM 2 is ultimately not used, we find it interesting nonetheless that the network has learned to predict a near 0 value for the rotor.

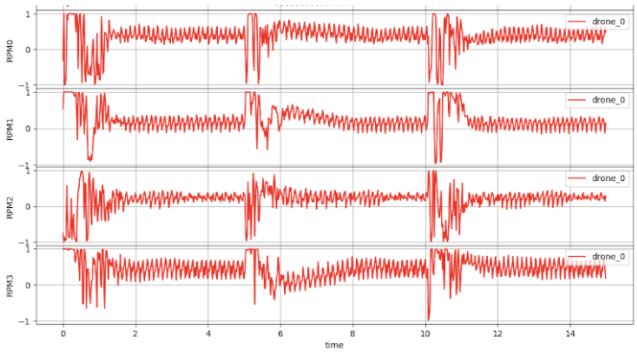

Fig. 6. Actor network's RPM output in 3 seperate episodes for the Single-Rotor Failure scenario. The y axis is the normalized RPM values with [-1, 0, +1] values mapping to [0 RPM, Hover RPM, Max RPM] of the rotors correspondingly. Note that RPM 2 (2nd rotor) output will be forced to 0 RPM to disable this rotor

VI. Alpha 1 Results

We now present the results for our first alpha, recreating the baseline experiments under Dual-Rotor Failure conditions. In alpha 1, 0th and 2nd rotors, which are opposite to each other, are set into a failure condition.

A. Alpha 1 Results - Hover

The hover experiment for alpha 1 was initially trained for 5 million time steps to be consistent with the baseline experiments. As can be seen in figure 7, however, the agent was not able to converge to a good reward in that amount of time.

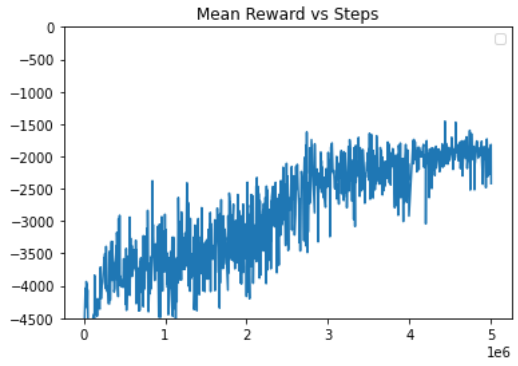

Fig. 7. Reward vs Steps plot for a Dual-Rotor Failure (opposite) Scenario for the first 5 million steps

We decided to use this pre-trained weights to train for a further 5 million time steps. As can be seen in figure 8, the agent was able to achieve much better reward with the additional training. The agent sill does not approach

comparable reward to the baseline experiments, either when compared with our implementation of the baseline or the reported results from the baseline paper. We believe that this is due to the agent settling into a local optima. While the agent learned to successfully hover in place, it often hovered slightly away from the desired hover point thought it was stable while doing this. This can best be seen in the supplementary video "alpha-1-square," where the agent successfully completes the path but does so hovering slightly above the path instead of moving directly along the line. We believe that additional training is required for the agent to converge to the global optima, in other words, to be able to hover at the desired hover point. Please see the video "alpha-1-hover" in the supplementary materials folder for more details.

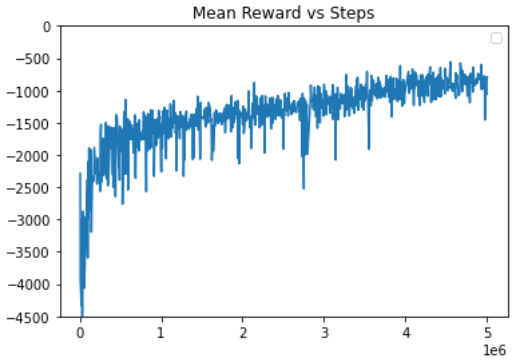

Fig. 8. Reward vs Steps plot for a Dual-Rotor Failure (opposite) Scenario for 5 to 10 million steps (due to using the pre-trained model with 5 million steps)

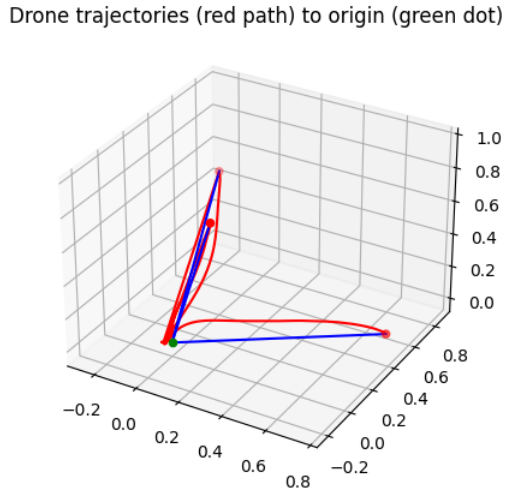

Fig. 9. Drone trajectories for the Dual-Rotor Failure (Opposite) Scenario starting at three random points and moving to origin

B. Alpha 1 Results - Navigate to Point and Hover

As can be seen in figure 9, our agent for alpha 1 is able to successfully complete the navigation experiment (with random point initialization). Please see the video "alpha-1-random-nav" for more details.

C. Alpha 1 Results - Navigate Square Path

The path the alpha 1 agent takes when completing the square path is shown in figure 10. The agent follows a path slightly above the specified path. As mentioned previously, we believe this is due to the agent settling into a local minima in the reward function instead of the global minima, or the desired hover location. We believe further training of the agent will alleviate this issue. Please see the video "alpha-1-square" for more details.

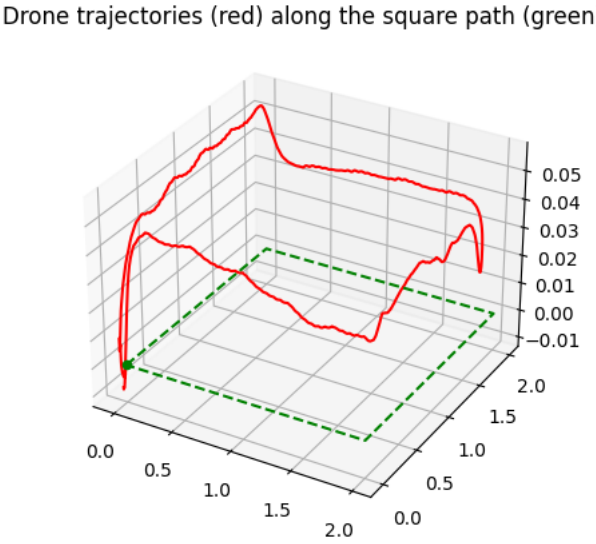

Fig. 10. Drone trajectory for the Dual-Rotor Failure (Opposite) Scenario following a predefined square path

D. Discussion - Alpha 1 Results

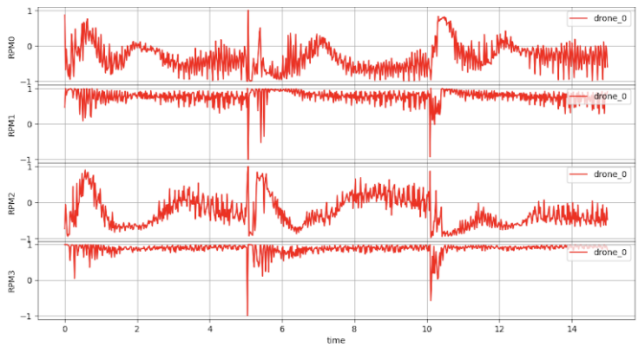

Fig. 11. Actor network's RPM output in 3 separate episodes for the Dual-Rotor Failure (Opposite) scenario. The y axis is the normalized RPM values with [-1, 0, +1] values mapping to [0 RPM, Hover RPM, Max RPM] of the rotors correspondingly. Note that RPMs 0 and 2 (0th and 2nd rotors) will be forced to 0 RPM to disable these rotors

Figure 11 shows the normalized RPM values that are predicted by the agent's network. Rotors in the opposite configuration (0th and 2nd) will be manually set to 0 RPM. When compared to the baseline, the normalized RPM values predicted by the agent's neural network seem to make more sense. The 1st and 3rd rotors are active and produce produce a near constant value indicating stable hovering as expected. As the 0th and 2nd rotors are set to 0 RPM, they are irrelevant to the task at hand and the value predicted for each rotor is attributed to noise.

VII. Alpha 2 Results

Our second alpha for this paper was to investigate Dual-Rotor Failure (adjacent) with the adjacent configuration. For this, the 1st and 2nd rotors are set to 0 RPM. Unfortunately, this was not successful in training an agent to fly and hover. As can be seen in figure 12, even after 5million time steps, the reward had little to no improvement. We believe that this is due to the inherent difficulty of the task. When opposite rotors are disabled, the agent can still stabilize the drone by spinning and otherwise maintain symmetrical thrust in the drone. When adjacent rotors fail, this becomes impossible. We believe that it may not be possible to solve this task using any kind of controller.

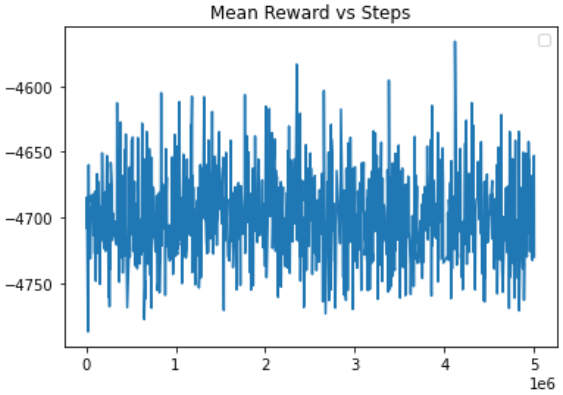

Fig. 12. Reward vs Steps plot for a Dual-Rotor Failure (adjacent) Scenario for 5 million steps

VIII. Alpha PPO

Part of our alpha was to investigate using PPO as an alternative means of training a successful agent. While PPO did have the benefit of faster training than SAC, it completed 20 million iterations in approximately 16 hours whereas SAC with 5 million iterations took approximately 1 day 7 hours, the algorithm was nonetheless unable to learn a working policy for any of the experiments. Figure 12 shows the reward plot of trying to train an agent to hover using PPO.

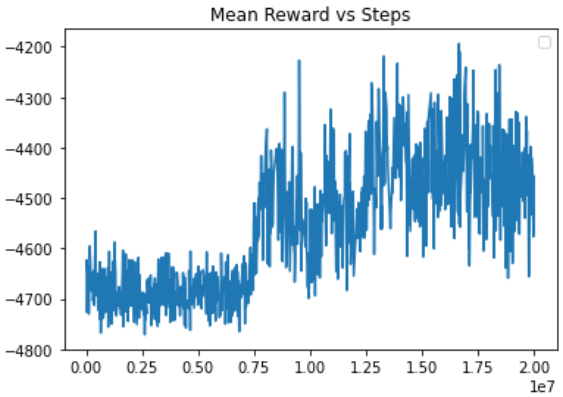

Fig. 13. Reward vs Steps plot for a Single-Rotor Failure Scenario using PPO for 20 million steps

The agent is clearly still learning as it show a jump in reward but ultimately does not learn how to hover properly. The agent, while having some maneuverability, ultimately falls for the duration of the episode. We believe that the reason PPO was not able to train a working agent in 20 million time steps is due to the sample inefficiency of the algorithm. Clearly, the agent was learning but it did not have enough iterations to learn a policy as robust as the one learned by SAC. Due to this, we decided not to purse PPO as a feasible algorithm for drone rotor failure. Please see the "PPO-agent" video for more information.

IX. Conclusion and Future Work

In this work, we investigated the feasibility of using deep reinforcement learning to train an agent to control a drone under the conditions of single and multi rotor failure. We successfully reimplemented our baseline, [1], and successfully extended the work to Dual-Rotor Failure. We also investigated different algorithms for training the agent and found that SAC is the best algorithm. While PPO could potentially achieve comparable results, it requires significantly more iterations to do so, while SAC can achieve a robust agent in less than 5 million time steps.

Future work should investigate extending this work to the real world. This work can be viewed as a proof of concept showing the feasibility of using deep RL for this type of problem. The next logical step is trying to test this approach on a real drone and bridging the sim to real gap.

X. References

[1] P. Sharma, P. Poddar and P. Sujit, “A Model-free Deep Reinforcement Learning Approach To Maneuver A Quadrotor Despite Single Rotor Failure” arXiv:2109.10488v1, 2021

[2] J. Hwangbo, I. Sa, R. Siegwart and M. Hutter, “Control of a Quadrotor with Reinforcement Learning” arXiv:1707.05110v1, 2017

[3] R. Arasanipalai, A. Agrawal and D. Ghose, “Mid-Flight Propeller Failure Detection and Control of Propeller-deficient Quadcopter using Reinforcement Learning”, arXiv:2002.11564v2, 2020

[4] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok and A. Shoellig, “Learning to Fly – a Gym Environment with Pybullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control,” arXiv:2103.-2142v3, 2021

[5] E.Liang, R. Liaw, P. Mortiz, R. Nishihara, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan and I. Stoica, “RLlib: Abstractions for Distributed Reinforcement Learning,” arXiv:1712.09381v4, 2018

[6] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel and S. Levine, “Soft Actor-Critic Algorithms and Applications”, arXiv:1812.05905v2, 2019

[7] J. Shulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, "Proximal Policy Optimization Algorithms", arXiv:1707.06347, 2017

[8] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O, Klimov, A. Nichol, M. Plappert, A. Radford, J. Shulman, S. Sidor, Y. Wu, "Stable Baselines", https://github.com/hill-a/stable-baselines, 2018