CS394R Final Project

Alejandro Moreno, Arnav Iyer

May 2022

Project Video: https://www.youtube.com/watch?v=81Ts0d0QkdI
Project Code: https://github.com/alemoreno991/RL$_{f}inal_{p}roject/tree/spinningup/gym_{m}ujoco$

1 Introduction

Reinforcement learning is a growing field that has impact in many disciplines. In particular, this project studies how RL can be beneficial in the robotics domain. Therefore, the main idea is to train an agent to fly a drone. Specifically, the goal is to make a drone move towards a desired point in space and the hover there. This has been done in the past [4] [6], but we wanted to investigate the reward function and the use of different algorithms, which seems to not have been done thoroughly in previous work.

The project was initially to use RL to automatically tune a PID controller for the drone's attitude. For that, a quadcopter dynamics simulator developed by ourselves was going to act as the environment. But, we had to rethought the project because our own drone simulator was too slow to make training feasible. So, after doing some research a MuJoCo-based physics engine with GymAI API was found. This environment gave us a baseline to train an agent fast and reliably. Not to mention the possibility of render the training process. Our goal was now to find and implement an RL algorithm that we could use to train the drone. Since the problem at hand deals with continuous state and action space we did some research and found modified implementations [1] of SAC[5] and DDPG[7], and realized that SAC worked the best. However, the learning process not only took long time but also, given the nature of the implementation of the RL algorithms, it was not possible to save the model. In this sense, the testing process was too slow since it was needed to train the agents every time. After having these issues, we switched to using OpenAI Spinning Up's [2] baseline implementations of VPG, DDPG, PPO, TD3, and SAC to find which algorithm would work the best. SAC and TD3 both converged to find a policy that was able to stabilize the drone until the arbitrary end of an episode (3000 steps), but VPG, PPO and DDPG were not able to converge. Instead, they either leveled off or increased very gradually, achieving returns 2 or 3 orders of magnitude smaller than SAC and TD3 within similar time-frames.

For hyperparameter tuning, we used the parameters that the authors specified in the paper, and we trained until the average return of an episode in an epoch was around 19,000. This number is around the maximum achievable return in one of our episodes.

Once we had an agent capable of stabilizing and navigating the drone, we wanted to see if we could use a combination of these two agents to build a local planner for drone navigation. If we keep moving our target point, will the drone agent be able to follow the target point and navigate along a predefined trajectory?

2 Motivation

Control theory is a well known and studied field of engineering. The methods and strategies developed during the last century came a long way to become what we know today as Classical, Modern, Robust, Optimal, Adaptive, etc. All of this methods are based on the assumption that a model of the plant is available to the designer. Moreover, this methods are based on linear systems, hence, are not able to handle non-linearities very well. This fact is what motivates this work because a drone is a highly non-linear system when one wants to operate it in its whole flight envelope.

The linear control systems enumerated before are very good options when one wants to operate close to a working point. In our case, they would do very well if properly tuned to keep the drone in quasi-static condition (hover-like).

Nevertheless, they would perform poorly if that same controller were to be used to design an acrobatic or a racing drone.

As explained in [3] and [9], two papers that talk about reinforcement learning being used to solve classical control problems, there are three reasons behind using RL in control settings:

- No model needed.
- No expert knowledge needed.
- Naturally handles non-linearities.

The list presented above is a major improvement our project could achieve over the control methods previously mentioned. The model unknowns are more often than not very hard to figure out, both economically and time-wise. Even if the model is perfectely known, the synthesis of the controller could in some cases like PID be very time consuming to fine tune. Nevertheless, in case one had a good model of the plant classical control theory most likely result in a very appealing option, it wouldn't account for changes in the operating point. Let's say the drone is at rest (hovering) and it suddenly wants to race to a goal. The aerodynamic effects and the non-linearities of the kinematics would then kick in and the plant's model would differ very much from the one used to tune the controller in a classical sense. Here is were RL could potentially achieve better performance by being able to handle the full flight envelop and maintain stability and performance throughout.

3 Theory

A quadcopter is potentially a very challenging system to control if one would like to operate it in different environments. For example, the wind or the drag force the vehicle is going to undergo could play a major role when designing a traditional control strategy. In this project a model-free reinforcement learning algorithm is used. However, it is interesting to show some of the dynamics that one would have to consider if a different approach was used.

3.1 Kinematic model

A kinematic model allows one to know the position, velocity and attitude of a body with respect to their temporal dependence to acceleration and angular acceleration.

- Define a frame of reference
- Choose a parametrization for the angular representation

The state vector is defined as the 18 element vector described in Table 1.

Table 1: State vector.

State variable	Description
$r_x^{\boldsymbol{I}}$	Position - X in inertial frame
$r_y^{\boldsymbol{I}}$	Position - Y in inertial frame
$r_z^{\boldsymbol{I}}$	Position - Z in inertial frame
$v_x^{\boldsymbol{I}}$	Velocity - X in inertial frame
$v_y^{\boldsymbol{I}}$	Velocity - X in inertial frame
$v_z^{\boldsymbol{I}}$	Velocity - X in inertial frame
R_{BI}	Direction-cosine matrix that rotates from inertial to body frame
ω_x^B	*roll* angular rate, expressed with respect to B.
ω_y^B	*pitch* angular rate, expressed with respect to B.
ω_z^B	*yaw* angular rate, expressed with respect to B.

The input vector is defined as the 6 element vector described in Table 2.

Table 2: Kinematic inputs.

Input	Description
F_x	Force - X expressed in body frame
F_y	Force - Y expressed in body frame
F_z	Force - Z expressed in body frame
N_x^B	Torque - X expressed in body frame
N_y^B	Torque - Y expressed in body frame
N_z^B	Torque - Z expressed in body frame

The kinematic model can be represented as Equation 1.

$$\begin{cases} m\dot{\boldsymbol{a}}_{\mathrm{I}} = m\boldsymbol{v}_{\mathrm{I}} \\ \dot{\boldsymbol{v}}_{\mathrm{I}} = -\boldsymbol{g} + R_{\mathrm{BI}}^T \sum_{i=1}^4 \boldsymbol{a}_i^{\mathrm{B}} \\ \dot{R}_{\mathrm{BI}} = -[\boldsymbol{\omega}_{\mathrm{B}}\times]R_{\mathrm{BI}} \\ \dot{\boldsymbol{\omega}}_{\mathrm{B}} = J^{-1}\left(\boldsymbol{N}_{\mathrm{B}} - [\boldsymbol{\omega}_{\mathrm{B}}\times]J\boldsymbol{\omega}_{\mathrm{B}}\right) \quad \text{where} \quad \boldsymbol{N}_{\mathrm{B}} = \sum_{i=1}^4 \boldsymbol{N}_i + \boldsymbol{r}_i \times \boldsymbol{F}_i \end{cases} \tag{1}$$

One should be aware that this is just a model and the real system could be much more complicated. For instance, the mass and inertia matrix of the drone could change over time. In addition, the structure could be flexible to a certain degree.

3.2 Dynamic model

A dynamic model establishes how the forces and moments impact the system.

- **Gravity:** it is not constant and modelling it accurately is not at all trivial.
- **Aerodynamics:** drag and lift forces are complex (to say the least) to accurately represent. In addition, wind can radically change the impact of these forces and torques. Furthermore, any asymmetry in the structure will also generate torques which makes the model even more difficult to get.
- **Propulsion:** the motors have their own dynamic which one would need to consider
- **Other effects:** interaction between the magnetic field of the motor and metallic pieces with the earth's magnetic field (or industrial electric fields).

As one can guess at this point, representing the real forces and moments is unfeasible. Even the best models will contain errors and mismatches. However, one could think that, even though it is hard to an accurate representation of reality, it is still possible (this is what aeronautical engineers did and still do).

3.3 Reinforcement learning

In this project, the idea is to use a model-free approach that allows the designer to become independent of the complexities and issues related to modelling the dynamics and kinematics of a drone.

State: is the one described in the kinematic section. Note that there is no need to identify the time dependency before hand. Only observation of this continuous state vector is needed.

Actions: the input to each of the 4 motors is going to be considered, where the propellers are allowed to spin in both directions (generating positive and negative thrust). The action space is, therefore, continuous and bounded.

Reward: a quadratic combination of weighted error between the current state and the desired state ($-x_e^T Q x_e$), and a quadratic combination of weighted input effort ($-u^T R u$) is considered. In addition, we add a reward for each time step the agent stays alive, penalties and bonuses for ending the episode and achieving a certain goal, and a penalty for straying too far from the straight line between the start and the goal.

3.3.1 Our Reward Function

Our reward function is adapted from [4] Quadrotor environment. However, we added a few more terms and tuned the values of their constants. Here is a detailed summary into what exactly is penalized and rewarded in our reward function. Bolded additions are novel to our method, and scale factors used are in parentheses.

Penalties:

- Distance to Goal (-5).
- Orientation Error (Norm of euler angles) (-0.02).
- Magnitude of linear and angular velocities (-0.01, -0,001).
- Magnitude of taken action (-0.0025).
- Flat penalty for leaving environment bounds (-15).
- Distance from line between start and goal (-5).
- Change in action from previously taken action (-3).

Bonuses:

- Flat Reward for staying alive (5)
- Flat bonus to reach the goal (15).
- Velocity towards goal (0.1).

4 Related Work

The project started by using a quadrotor simulator developed in another course. It is a high fidelity dynamic simulator developed in python that offers great flexibility to modify and understand the physics behind a drone. However, The simulator was not very fast, and training an RL agent, while technically possible, would require too much time. Therefore, after a literature search, we found a MuJoCo-based quadcopter simulator [4]. Due to the nature of this simulator, our action space was the 4-dimensional vector of voltages to each motor. The work that provided the simulator environment also came with a sample reward function that took into consideration multiple things - (list these things). Many of these things were elements of the reward function we discussed earlier. Although we kept the basic elements of this reward function, during training we had to add more to it and tune the reward function to work better for our purposes. Elements of the reward function we took from the prior work were the penalties relating to distance from goal and velocity magnitudes, and a reward for staying alive. We added a reward term that adds negative reward proportional to the drone's distance from the straight line between the start and goal.

4.1 Drones and Reinforcement Learning

There is lots of previous work in training drones to fly with RL, but many, like [6] and [10] use advanced techniques like Model-predictive control and sophisticated reward functions to achieve state-of-the-art results. The goal of our project is to investigate more simpler techniques, such as constructing and tuning a basic reward function, and comparing training performance of different state-of-the-art RL algorithms.

Our domain is one with a continuous state and action space. Because of this, The Soft Actor-Critic [5], (Twin Delayed) Deep Deterministic Policy Gradient [7] [**td3**], and Proximal Policy Optimization algorithms [8] were all options we though would be worthwhile to test. The following section is an overview of some of these algorithms. Note that PPO has been omitted because during testing, we found that it performed extremely poorly compared to the others.

4.2 Deep Deterministic Policy Gradient

DDPG is an off-policy algorithm for continuous action spaces that learns an action-value function with bellman updates, and simultaneously learns a policy from the action-value function. There are two networks, the actor and the critic. The critic learns the action-value function Q through a modified bellman update that makes use of replay buffers, to stabilize learning and use data more efficiently, and a target policy network. A target network is one that is essentially an average of the previous states of a network that is used as the training target, i.e. What we subtract from Q in the loss function. Because this term depends on Q itself, keeping another network that is an average of the policy network's previous states stabilizes the minimization. The actor's update is much simpler. It is simply a gradient ascent with respect to the actor's parameters that maximizes expected values of the Q-function, using the critic.

Below is an outline of DDPG's learning process after it has stored enough experiences in the replay buffer. Networks and target networks for the actor and the critic are initialized to equal each other.

For each necessary update:

1. Sample N transitions $(s, a, r, s', d), ...$ from the replay buffer.
2. For each of the transitions, compute target values using target networks.

$$t = r + \gamma(1 - \mathbb{1}_d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$$

3. Update Q with gradient descent proportional to average diff with target.

$$\nabla_\phi \frac{1}{N} \sum_{\forall(s,a,r,s',d)} (Q_\phi(s, a) - t)^2$$

4. Update policy with gradient descent proportional to sum of action values of states.

$$\nabla_\theta \frac{1}{N} \sum_{\forall s} Q_\phi(s, \mu_\theta(s))$$

5. Update target networks (Spinning Up uses polyak averages).

4.3 Twin Delayed DDPG

Twin Delayed DDPG, or T3D, addresses weaknesses of the original DDPG algorithm. Because DDPG uses a learned Q-function to learn the policy, the learned policy μ_θ will take advantage of errors in the learned Q-function $_\phi$. TD3 is mostly similar to DDPG, with the exception of three differences that vastly increase its robustness:

- Sometimes, Q_ϕ overestimates q-values, which affects policy learning. TD3 learns two Q-functions simultaneously and uses the smaller of the two values in target computation.
- TD3 updates μ_θ and $\mu_{\theta targ}$ less frequently than $_\phi$ and $Q_{\phi_{targ}}$. Paper authors recommend updating $_\phi$ twice before updating μ_θ.
- In order to minimize the chance that the policy learns to take advantage of an incorrectly-learned Q-function, gaussian noise is added to $Q_{\phi_{targ}}$.

4.4 Soft Actor-Critic

Like DDPG and TD3, Soft Actor-Critic, or SAC, learns a policy in an actor-critic model and is off-policy. Like TD3, it even learns two Q-value functions for the same reasons. Unlike the others, however, it learns a stochastic policy as opposed to a deterministic policy.

SAC's key feature is called entropy regularization. Entropy as defined in RL, is the weighted sum over action probabilities with weight equal to the negative log of the action probability. In a discrete setting, $H(\pi(|s_t)) =$

$-\sum_{a \in A} \pi(a|s_t) \log \pi(a|s_t)$. There is a similar analog for continuous action spaces. Entropy Regularization involves adding $\alpha H(\pi(|s'))$, the entropy of a transition, to $R(s, a, s')$, the reward of a transition, in the formulation of V^π and Q^π.

In the pseudocode below, the log of the action probability is subtracted from $Q_{\phi_{targ}}$ when calculating the target and $_\phi$ in the policy gradient update. This rewards higher entropy.

For each necessary update:

1. Sample N transitions $(s, a, r, s', d), ...$ from the replay buffer.
2. For each of the transitions, draw a' from $\pi_\theta(|s')$ and calculate targets.

$$t = r + \gamma(1 - \mathbb{1}_d)(\min_{i=1,2} Q_{\phi_{targ}}(s', a') - \alpha \log \pi_\theta(a'|s'))$$

3. Update both Q-functions, for $i = 1, 2$

$$\nabla_{\phi_i} \frac{1}{N} \sum_{\forall (s,a,r,s',d)} (Q_\phi(s, a) - t)^2$$

4. Sample a_θ from $\pi_\theta(|s)$ and update policy.

$$\nabla_\theta \frac{1}{N} \sum_{\forall s} (\min_{i=1,2} Q_{\phi_{targ}}(s', a_\theta(s)) - \alpha \log \pi_\theta(a_\theta(s)|s))$$

5. Update target networks (Spinning Up uses polyak averages).

5 Results

The procedure followed was to train the agents in different conditions and then test its performance across a variety of conditions, that is various initial states. Table 3 shows exactly what cases were contemplated. Testing conditions are elaborated on in the following section.

Table 3: Benchmark strategy to compare different training settings and initial testing conditions.

	Initial testing condition		
Training condition	Vanilla	Moderate	Extreme
Vanilla	X	-	-
Moderate	X	X	X
Extreme	-	-	X

5.1 Training stage

We trained the drone to hover in place (vanilla), to navigate to a goal point from a randomized starting state (moderate), and the same navigation tasks but harder initial conditions (extreme). These problem settings can be summarized in Table 4.

The training was carried out through SpinningUp's interface, where we trained until the average return per epsiode leveled off. Each epoch in training consists of 6000 steps, and each epsiode has a maximum of 2000 steps. This means that epochs contain less episodes as the agent becomes better at the task. All hyperparameters were taken from the relevant algorithm's papers.

Table 4: Training conditions.

Training condition	**Configuration** Random start	Initial State
Vanilla	No	null
Moderate	Yes	$er_i < 1.2$ $\lvert v_i \rvert < 0.5$ $\lvert \phi \rvert, \lvert \theta \rvert, \lvert \psi \rvert < \frac{\pi}{3}$ $\lvert \omega_i \rvert < \frac{\pi}{10}$
Extreme	Yes	$er_i < 1.2$ $\lvert v_i \rvert < 2.0$ $\lvert \phi \rvert, \lvert \theta \rvert, \lvert \psi \rvert < \frac{\pi}{2}$ $\lvert \omega_i \rvert < \frac{\pi}{3}$

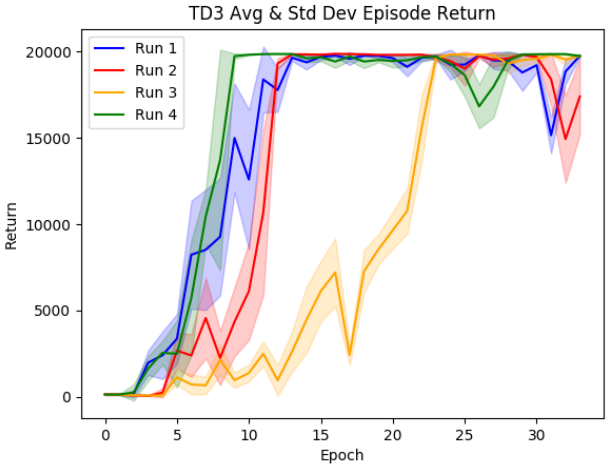
In addition to the strategy previously exposed, the training process was carried out by executing 5 runs of approximately[1] 50 epochs, where an epoch represents 1000 episodes or 5000 steps.

The first algorithms used to tackle the problem were VPG, DDPG and PPO. However, the cumulative reward was nowhere near what was expected. The aforementioned can be appreciated in Figure 1. As to why our training failed here, we are still a bit confused. Aside from what is shown here, we tried training VPG and PPO for a much longer time, and they still did not perform up to par.

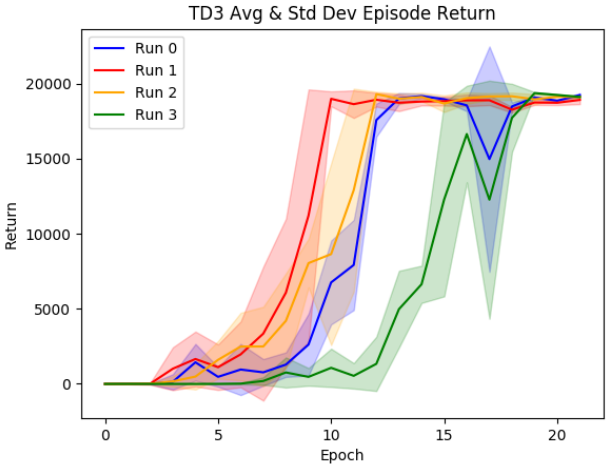
[1]Some algorithms required less training

Figure 1: VPG, DDPG and PPO cumulative return for the vanilla training case. Given that 5 runs were executed, the plot reflects the average and standard deviation corresponding to the learning process of the 5 agents.

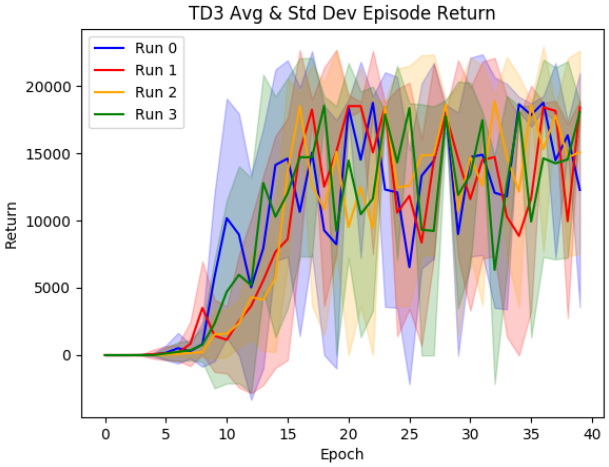
We also trained on TD3 and SAC. In this case, the situation changed dramatically. Within the first 20 epochs both algorithms were performing quite well, with more consistent results for TD3. Perhaps TD3 is more robust than SAC during training because it is trying to learn a deterministic policy rather than a stochastic policy, and entropy regularization encourages too much exploration. Figure 3 and Figure 2 reveal this.



(a) Drone Hovering

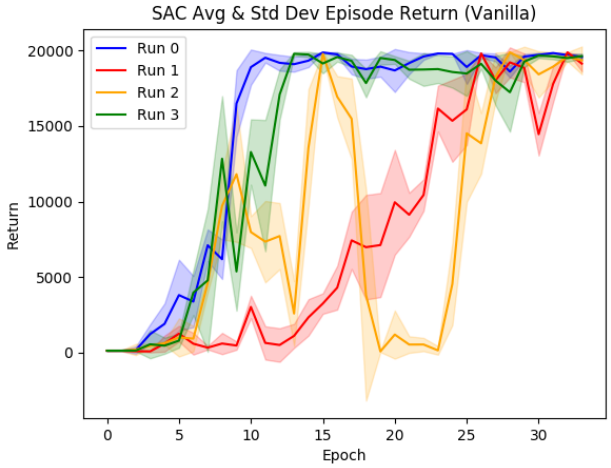


(b) Drone Navigation from A to B

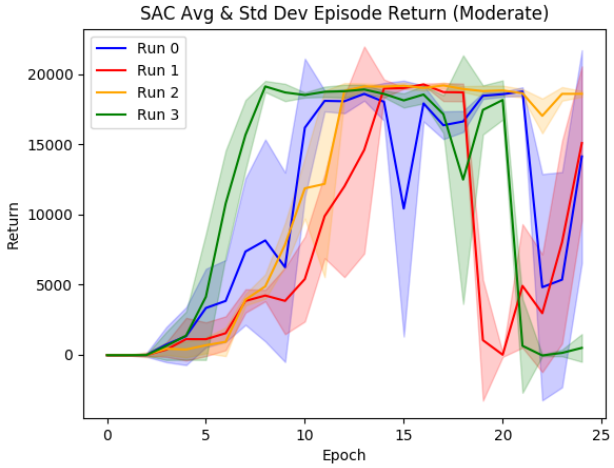


(c) Drone Navigation with harder initial conditions

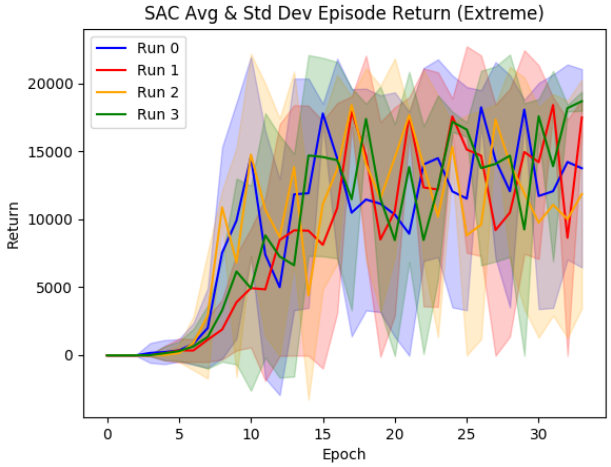
Figure 2: TD3 Average return per episode. We can see that TD3 performs much better than VPG, PPO, and DDPG, and it achieves much higher returns.



(a) Drone Hovering



(b) Drone Navigation from A to B



(c) Drone Navigation with harder initial conditions

Figure 3: SAC Average return per episode. SAC does seem to converge to an optimal, but it clearly is not as stable as TD3. This may be due to SAC's entropy regularization.

5.2 Testing the agents

The trained agents were tested in a controlled test-bed. That is, selecting a testing scenario (vanilla, moderate, extreme) and running the policy the agents learnt in training. Table 5 shows the scenarios used. It is possible to appreciate that each scenario consists of 4 different tests, were the initial state vector is modified.

Table 5: Testing conditions.

Training condition	Position [m]	Euler angle [deg]	Velocity [m/s]	ω [rad/s]
Vanilla	$[-.6, .8, 3.5]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$
	$[0, 0, 3.0]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$	$[-.01, .005, .002]^T$
	$[0, 0, 3.0]^T$	$[5, -2, 0]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$
	$[0, 0, 3.0]^T$	$[0, 0, 0]^T$	$[.1, .05, .01]^T$	$[0, 0, 0]^T$
Moderate	$[-.6, .8, 3.5]^T$	$[-30, 25, 0]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$
	$[.5, -.4, 3.7]^T$	$[0, 0, 0]^T$	$[.3, -.24, .1]^T$	$[0, 0, 0]^T$
	$[.7, -.4, 2.5]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$	$[.15, -.2, .1]^T$
	$[.5, -.4, 2.7]^T$	$[-20, 30, 0]^T$	$[0, -.05, .09]^T$	$[-.05, .15, -.08]^T$
Extreme	$[-.6, .8, 3.5]^T$	$[-80, 85, 180]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$
	$[.5, -.4, 3.7]^T$	$[0, 0, 0]^T$	$[1.8, -1.6, -1.4]^T$	$[0, 0, 0]^T$
	$[.7, -.4, 2.5]^T$	$[0, 0, 0]^T$	$[0, 0, 0]^T$	$[1.35, -1.5, 1.1]^T$
	$[.5, -.4, 3.0]^T$	$[60, 85, 0]^T$	$[1.3, -1.2, -.9]^T$	$[-.45, .35, .18]^T$

We generated graphs for some of these test cases, and we walk through them below.

5.2.1 Example - Moderate Agents

In the following two figures, we show that the agents we trained to travel from their initial state to a goal state are able to hover in place (Figure 4) and also travel from one state to another (Figure 5). The graphs show only 400 steps but that is just a visualization decision we adopted to be able to show the transition from initial condition to hover. Longer episodes were studied and the agents reach a steady state no matter how long the simulation is.

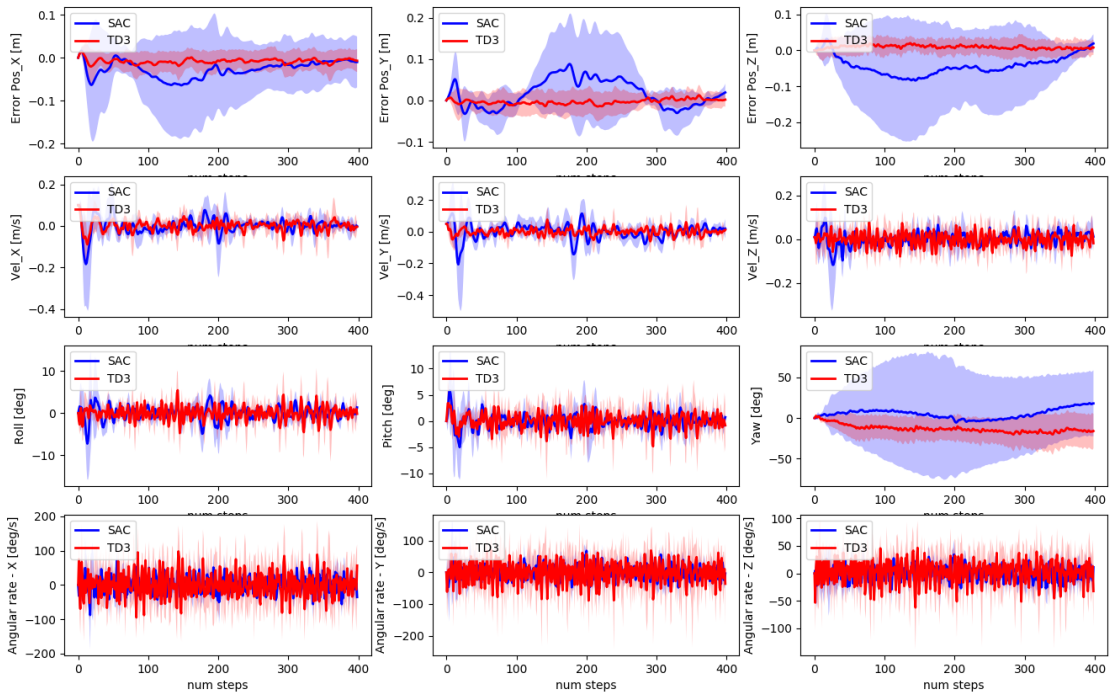


Figure 4: Moderate training and fourth test case of the moderate initialization. Given that 5 runs were executed, the plot reflects the average and standard deviation corresponding to the testing process on the 5 agents.

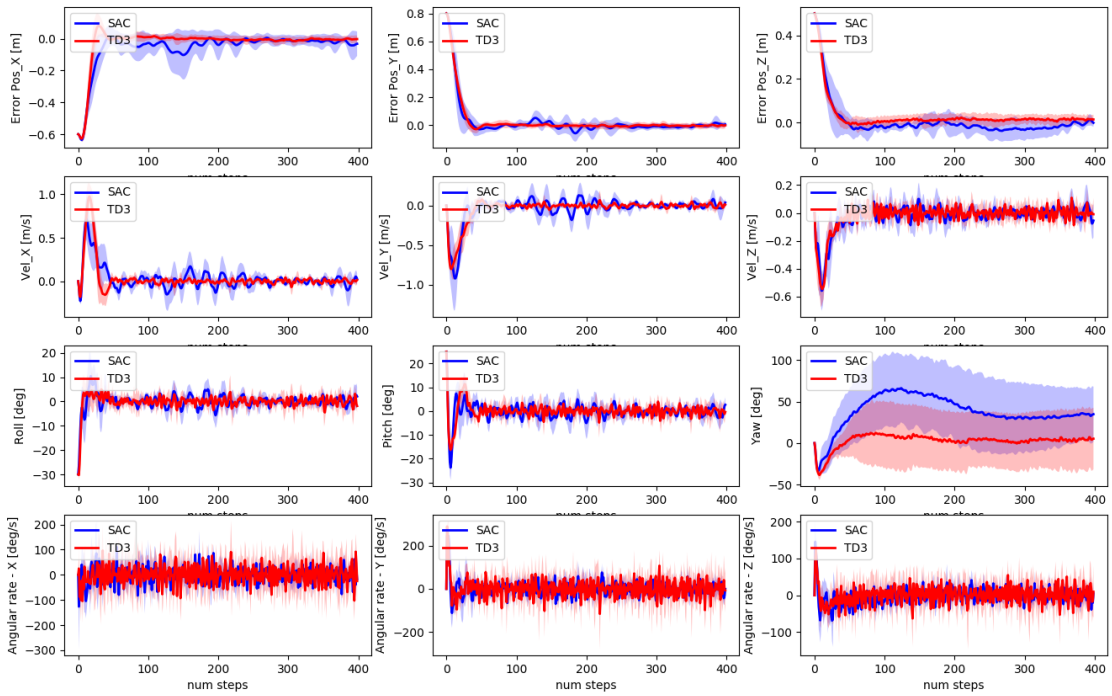


Figure 5: Moderate training and first test case of the moderate initialization. Given that 5 runs were executed, the plot reflects the average and standard deviation corresponding to the testing process on the 5 agents.

It is interesting to note that in the third test of the moderate scenario (Figure 6 one can appreciate the difference in performance between TD3 and SAC. Here, at least one of the drones trained with SAC flies out of the bounding box represented by an allowed position error and therefore the episode ends. This further illustrates that TD3 was more robust when evaluating our trained agents.

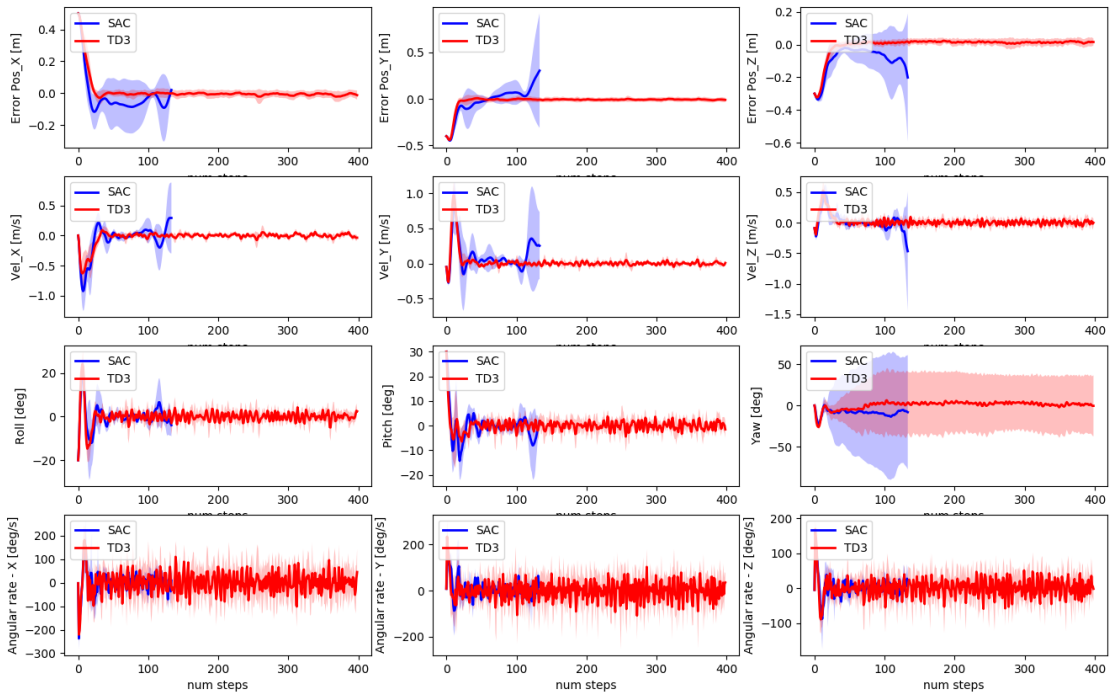


Figure 6: Moderate training and fourth test case of the moderate initialization. Given that 5 runs were executed, the plot reflects the average and standard deviation corresponding to the testing process on the 5 agents.

5.2.2 Extreme training

Handling the extreme case is something that we wanted to test because it would mean that the agent is very likely to recover from aggressive maneuvers as we are testing here. But it would also mean that acrobatic drones could in fact be implemented through proper training. This fact is not minor, since traditional control methods would not be able to handle those cases. Therefore, our results not only come from a model-free learning process but also beats some traditional methods in what they can handle.

Figure 7 indeed shows that the agents were able to learn how to recover from very challenging initial conditions and at the same time move to a certain location. Once again, we can see that TD3 was more robust than SAC, and its agents had less variation in the policy they learned.

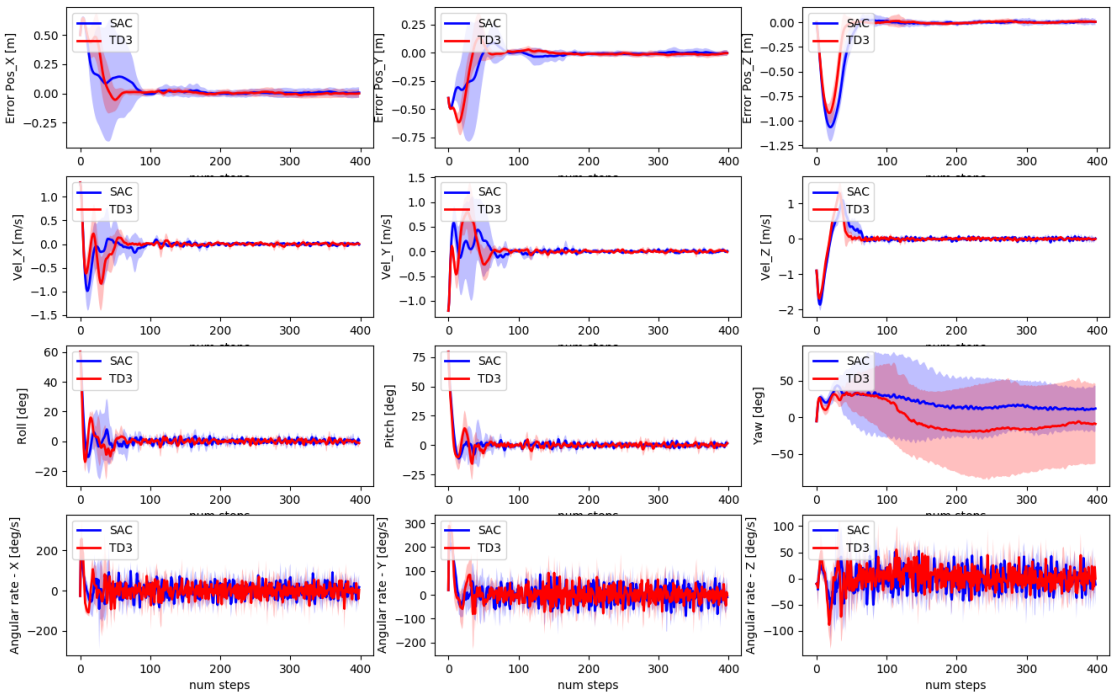


Figure 7: Extreme training and fourth test case of the extreme initialization. Given that 5 runs were executed, the plot reflects the average and standard deviation corresponding to the testing process on the 5 agents.

5.3 Following a Trajectory

After training the drone, we found that the TD3 agent learned the most robust policy. We wanted to see how accurately our trained agent could follow a set of trajectories, especially after we added reward for staying close to the straight line between start and goal. Below, we can see a graph illustrating the top-down view of our drone as it flies along a square trajectory. For 3 out of 4 sides of the square, we can see it follows the trajectory accurately. However, for some reason, it seems the agent never had a chance to learn a better policy for navigating to the left.

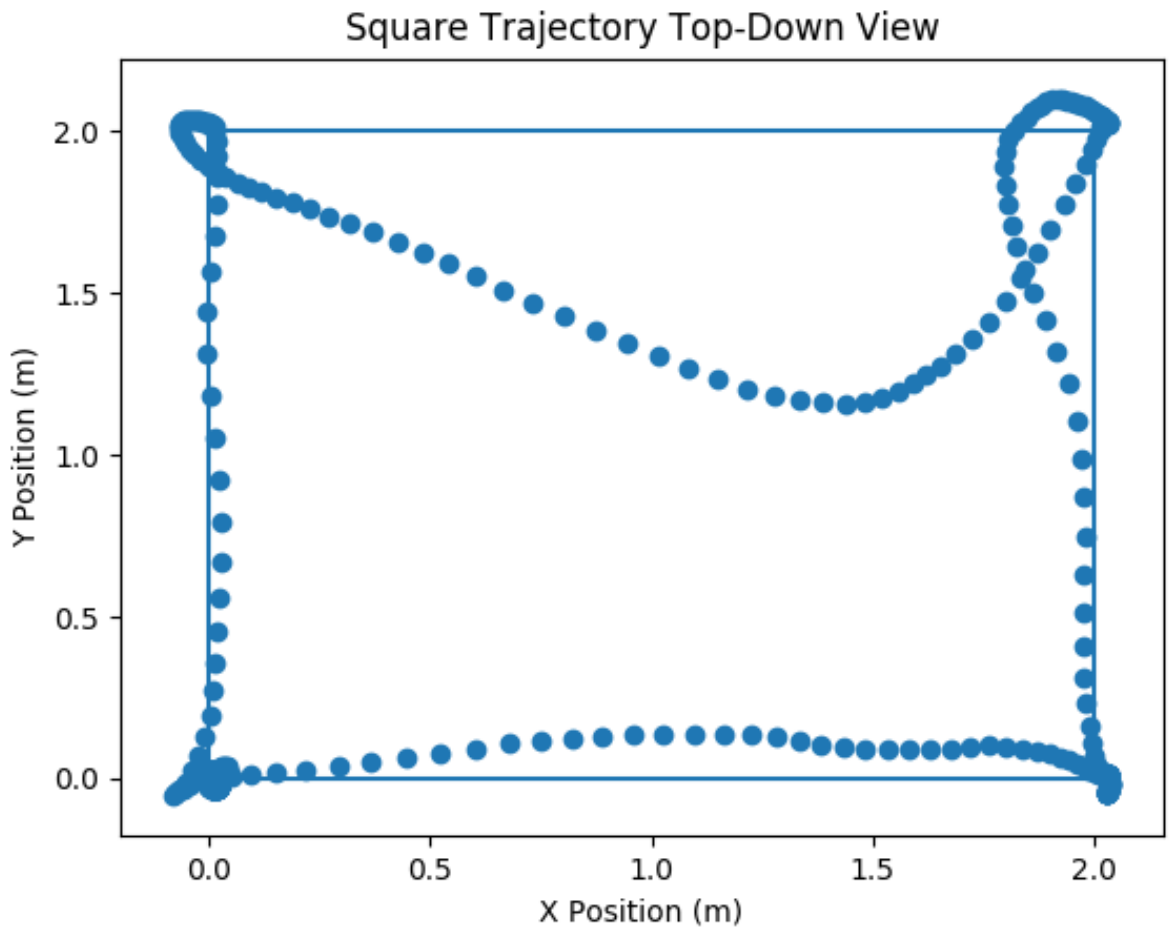


Figure 8: A top-down plot of the drone's position following a 2m square at a fixed height. The drone gets off course on the top edge of the square, but follows the trajectory relatively well for the other edges.

Trying to make a finer trajectory with more intermediate goals, we see that there is a consistent erroneous behavior. The drone is always veering to the left before arriving at its goal. Adding different incentives to our reward function somewhat helped follow the lines better, but it still is not very precise.

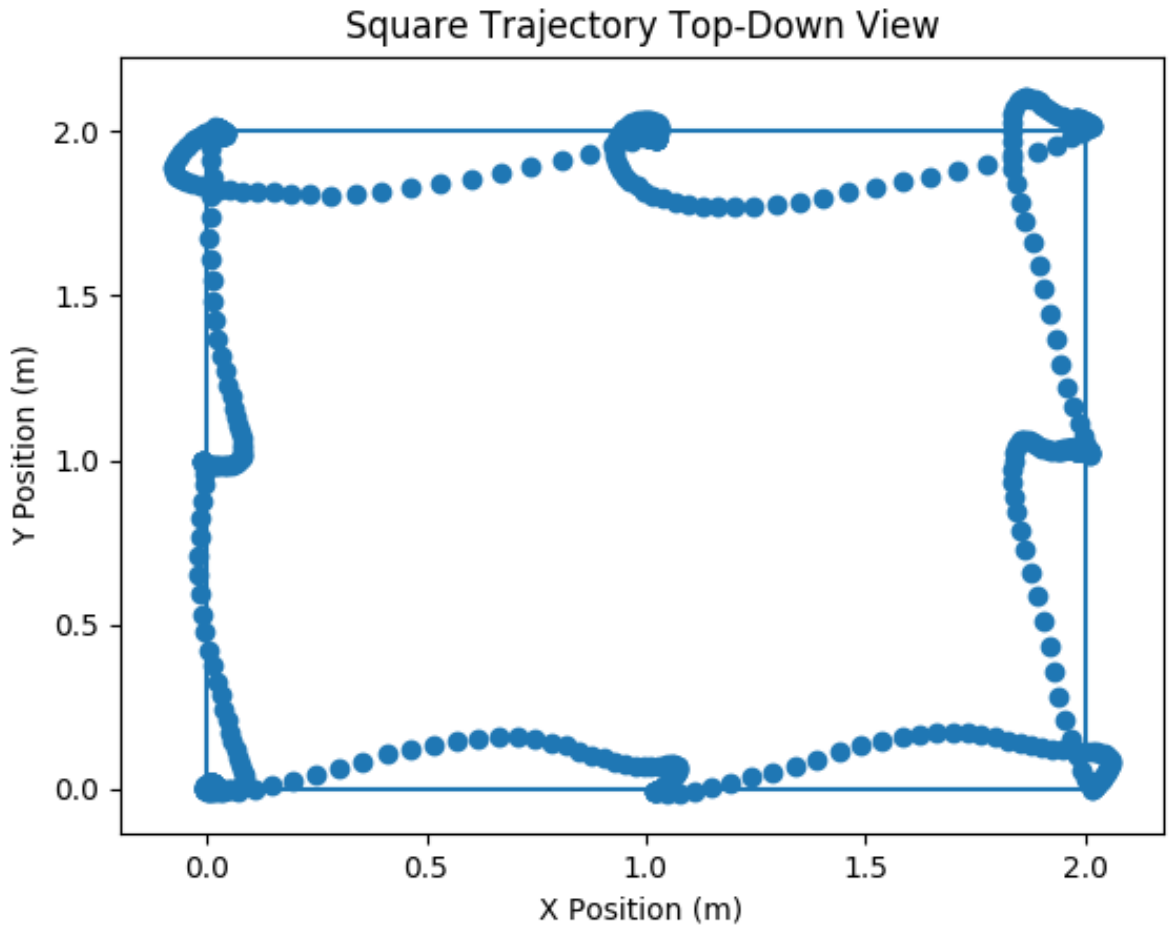


Figure 9: A top-down plot of the drone's position following a 2m square trajectory at a fixed height, with target points more frequently placed along the trajectory. There is a consistent swerve to the left before reaching each target point.

6 Discussion

First of all, it is worth mentioning that the project was very time consuming due to the various complications that arose and the training times required. Nevertheless, it was possible to solve all the issues and verify that it is indeed possible to train an agent to stabilize a drone even from extreme initial conditions.

Something worth mentioning is that the steady state (hover) achieved is not quite perfect. It is possible to observe a vibration-like behavior (high frequency oscillation of ω and the corresponding euler angles) when the drone is hovering. Hence, a possibility would be to modify the reward function to avoid or attenuate it. Currently, we tried simply penalizing the magnitude of our velocities, angular and linear, but this didn't have an effect on the oscillations. Were we using a PID controller, the Integral term would be able to smooth out the control signal over time, but because of the Markovian nature of RL, we were wary of trying to make our penalties functions of many previous states.

With regards to the algorithm results, we believe that TD3 was able to converge faster than SAC because TD3 doesn't make use of entropy regularization. To control a drone (either to hover or to navigate to a point), there is usually one optimal control signal, and after learning for a while, exploration can be curtailed. Because SAC was trying to learn a stochastic policy, it takes more time to converge to an optimal solution. TD3, on the other hand, quickly finds and exploits good policies.

References

[1] URL: https://github.com/babyapple/tidy-rl.

[2] Joshua Achiam. "Spinning Up in Deep Reinforcement Learning". In: (2018).

[3] Lucian Buşoniu et al. "Reinforcement learning for control: Performance, stability, and deep approximators". In: *Annual Reviews in Control* 46 (2018), pp. 8–28. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2018.09.005. URL: https://www.sciencedirect.com/science/article/pii/S1367578818301184.

[4] Aditya M Deshpande et al. "Developmental reinforcement learning of control policy of a quadcopter UAV with thrust vectoring rotors". In: *Dynamic Systems and Control Conference*. Vol. 84287. American Society of Mechanical Engineers. 2020, V002T36A011.

[5] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. DOI: 10.48550/ARXIV.1801.01290. URL: https://arxiv.org/abs/1801.01290.

[6] Jemin Hwangbo et al. "Control of a Quadrotor With Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 2.4 (Oct. 2017), pp. 2096–2103. DOI: 10.1109/lra.2017.2720851. URL: https://doi.org/10.1109%2Flra.2017.2720851.

[7] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[8] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[9] Chen Tessler, Yonathan Efroni, and Shie Mannor. "Action Robust Reinforcement Learning and Applications in Continuous Control". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 6215–6224. URL: https://proceedings.mlr.press/v97/tessler19a.html.

[10] Tianhao Zhang et al. "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search". In: *CoRR* abs/1509.06791 (2015). arXiv: 1509.06791. URL: http://arxiv.org/abs/1509.06791.