Reinforcement Learning Based Quadcopter Controller

Fang-I Hsiao (fihsiao)

Cheng-Min Chiang (cmchiang)

Alvin Hou (alvinhou)

Abstract

The goal of our work is to explore the application of Reinforcement Learning (RL) to autonomous control systems. Specifically, we are interested in building an RL-based control system for quadcopters. We implement several RL-algorithms, including A2C and PPO, and evaluate these agents in a simulated environment. The agents are able to stabilize the quadcopter and moved to the target position after training. Finally, we compare our results with classical PID controllers.

1 Introduction

In recent years, Reinforcement Learning (RL) has been applied successfully to a wide range of areas, including robotics [3], chess games [13], and video games [4]. In this work, we explore how to apply reinforcement learning techniques to build a quadcopter controller. A quadcopter is an autonomous helicopter that is lifted by the thrust from four motors. These motors also produce torques that cause the quadcopter to rotate. Unlike planes, quadcopters are inherently unstable, so the controller needs to act fast based on the observations from sensors, which typically include accelerometer and gyroscope.

We choose to use deep reinforcement learning algorithms as our controllers. Deep reinforcement learning algorithms are RL algorithms that use deep neural networks as a function approximator. Most of the RL algorithms will need to go through a training phase, in which the agent performs poorly. It could be a problem if we were using a real quadcopter since the drone could be damaged if the agent loses control during the training phase. To reduce the cost and difficulty, we built a virtual environment to simulate a flying quadcopter in 3-dimensional space. We try to make our simulator as realistic as possible. For each time step, the motion and the rotation of the quadcopter are calculated and updated based on physical laws. Moreover, to stimulate the uncertainties a quadcopter will encounter in real-world scenarios, we added different noises, including sensor errors, motor noises, and wind. The goal for the controller is to stabilize and navigate the quadcopter to a particular position.

We implement two deep RL algorithms: A2C [9] and PPO [11]. In contrast to Deep Q-Network [8], a well known deep RL algorithm extended from Q-learning, A2C and PPO directly optimize the policy instead of learning the action value. This is more suitable for our task because the action space of the task is continuous, which Deep Q-learning can not easily deal with.

2 Related Work

Quadcopter controllers are usually implemented using proportional–integral–derivative (PID) controllers [10]. PID controller performs well because the physical dynamics of a quadcopter is well known, so fine-tuning the model and parameters such as the gain in the stabilizing task is simple. PID controller will enjoy benefits such as fast convergence and easily interpreted output. However, it is much more difficult for PID controllers to generalize to new or complicated settings, such as new environments or complex goals.

Several prior works are using RL algorithms in autonomous control. In Hwangbo et al's [2] work, they trained a controller with policy and value networks using natural gradient descent. To stabilize the learning process during the initial phase, they paired the RL controller with a low gain PD controller. In our work, we aim to train the deep RL agent without any aid from other controller.

3 Environment

The observations our agent received are 15-dimensional vectors in $\mathbb{R}^{15}$ which include the position, velocity, acceleration, angular displacement, and angular velocity. Each of them is a 3-dimensional vector representing the value in x, y, and z-axis. This is in accordance with normal sensors equipped in a quadcopter.

The controller controls the quadcopter by setting the thrust a_1, a_2, a_3, a_4 of the four motors with each a_i being a real value between 0 and 1. $a_i = 1$ means that the i-th motor is operating in full power, while $a_i = 0$ means that the motor is completely idle. The action space is thus $[0, 1]^4$.

The goal of the controller is to navigate the quadcopter to a fix target point. We initialize the position of the quadcopter randomly so the goal is different each time. The reward function we choose is

$$R_t = \max\big(0, 1 - \|\boldsymbol{x} - \boldsymbol{x}_{\text{goal}}\|\big) - C_\theta\|\boldsymbol{\theta}\| - C_\omega\|\boldsymbol{\omega}\| \tag{1}$$

Where $\boldsymbol{x}, \boldsymbol{\theta}, \boldsymbol{\omega}$ are the position, angular displacement and angular velocity of the drone, and $\boldsymbol{x}_{\text{goal}}$ is the position of the goal. The first term rewards the agent when the drone is close to the target, and the other are penalizing terms to ensure that the quadcopter is not spinning and points in the right direction. C_θ and C_ω are constants that are intentionally kept small so staying close to the target will be the main goal.

The challenging part in stabilizing the quadcopter is that its rotational inertia is much less than its motion inertia (mass). The quadcopter needs a considerable amount of thrust to stay airborne, while a slight difference between the power of motors can produce a huge angular acceleration to the quadcopter, causing the drone to lose control.

We also add several noises to resemble the real world better.

- We add random Gaussian noise to the sensor to represent sensor error.
- We also add random noise to the thrust of the motors.
- To simulate the effect of wind, we add $v_{\text{wind}} \cdot \Delta t$ to the position of the drone for each timestamp, where the velocity of the wind v_{wind} changes gradually over time.

We build our environment to be compatible with OpenAI gym API [1] so that we could easily switch to simpler environments in OpenAI gym to test our algorithm. For debugging and visualization purposes, we also modify a web interface that could visualize the flying drone in an interactive 3D scene. This enables us to observe how the drone flies in terms of the velocity, acceleration, and the path it chooses.

4 Methods

One of the challenges in this problem is that the action space is continuous, and RL algorithms such as Deep Q-learning can only handle discrete action space. Fortunately, several RL algorithms could be modified to work in continuous action space, such as Deep DPG (DDPG) [7], Advantage Actor-Critic methods (A2C) and Proximal Policy Optimization (PPO).

We choose to implement A2C and PPO. These two algorithms are based on the policy gradient methods that directly optimize the policy by estimating the policy gradient. The policy gradient of a policy π parameterized by θ could be calculated using the following formula [14]:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q(s, a)\nabla_\theta \log \pi(a \mid s)] \tag{2}$$

Then, stochastic gradient descent is used to update the policy.

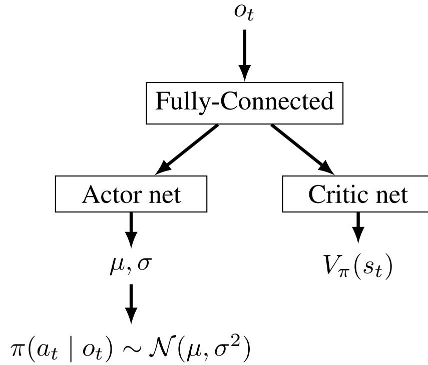


Figure 1: Our Actor-Critic network structure.

4.1 Advantage Actor-Critic

Advantage Actor-Critic (A2C) method combines the value-based method with the policy-based method. Equation (2) is known to have a high variance. In order to reduce the variance, Advantage Actor-Critic methods introduce a "baseline" $V(s)$ and replace $Q(s,a)$ with the advantage function $A(s,a)$ defined by $A(s,a) = Q(s,a) - V(s)$. In addition to the policy network π_θ (called "Actor") that is typically approximated by a neural network, it uses another network (called "Critic") to learn the value function $V(s)$ which estimates the expected reward for each state. The resulting policy gradient formula is:

$$\nabla J(\theta) = \mathbb{E}_\pi[\nabla_\theta \log \pi(a \mid s) A(s,a)]$$

To deal with continuous action spaces, we change the actor network to output the mean and variance of a Gaussian distribution instead of the probability of each action as in discrete action spaces. The network structure we used is shown in figure 1.

4.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a state-of-the-art reinforcement learning algorithm based on policy gradient methods. Standard policy gradient methods are on-policy, which means after updating the policy once, they need to sample again to evaluate the new policy. This is inefficient for most problems because interacting with the environment is time-consuming. By contrast, PPO allows multiple gradient updates using different batch of the sampled data. Therefore, it is more data-efficient, and the training speed is faster.

However, if we perform multiple updates, training will become unstable because the policy may change a lot at one step. To stabilize the training process, PPO adds a constraint during policy updates. It requires the new policy to not differ too much from the old policy. This approach is similar to TRPO [12], while PPO implements this constraint in a simpler way. Precisely, they both maximize the following surrogate objective function instead of the original objective function we described above:

$$L(\theta) = \mathbb{E}[r(\theta)A(s,a)], \quad \text{where } r(\theta) = \frac{\pi_{\theta_{new}}(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{3}$$

TRPO prevents the new and old policies from diverging too much by enforcing the KL-divergence between these two distributions to be small. This constraint is complicated and hard to be implemented, so instead, PPO enforces the probability ratio between the new and old policies $r(\theta)$ to be within $[1-\varepsilon, 1+\varepsilon]$ by clipping it. As a result, PPO maximizes a new objective function which takes the minimum between the original and clipped objective functions, and thus prevents the policy from moving toward a new policy whose parameters result in high $r(\theta)$:

$$L(\theta) = \mathbb{E}[\min(r(\theta)A(s,a), \text{clip}(r(\theta), 1-\varepsilon, 1+\varepsilon)A(s,a))] \tag{4}$$

5 Results

We train our A2C and PPO agent with number of training steps between 100000 to 200000 and test the performance every a certain amount of steps. The training time ranges from half an hour to an hour running on a computer with a 6 core i5-9400F CPU and a Nvidia 2070 GPU. Figure 2 plots the reward over the number of training steps. From the graph, we can see that the performance of both A2C and PPO agent improve over time. For this problem, we consider the agent to solve the problem when it receives an average reward greater than 200 per episode. The agent should be able to stay close to the target indefinitely, or at least for a long time if it achieves this average reward, though it may slightly drift around the target. PPO performs significantly better than A2C with a final reward 500 v.s 300, and also learns much faster.

The behavior of the quadcopter at each step is depicted in Figure 4 for PPO method. The orange trail represents the path the quadcopter flies. In the first 24000 steps (a), the quadcopter flies randomly and receives little to none reward. At around 60000 to 72000 steps (c), we can see that the quadcopter tries to hover around the origin, although not perfectly stable. At step 120000 (d), our quadcopter could maintain its position at the origin quite well with a moderate variance. After 300000 steps (e), the quadcopter navigates straight to the origin and remains still at the same spot. This indicates that our agent successfully learned to stay as close to the target as possible, while in the same time prevent itself from spinning around.

While both PPO and A2C are able to stabilize the drone, they performed worse than a carefully tuned PID controller. Figure 3 shows the comparison between the PPO and the PID controller. Sub-figure (a) plots the value of the drone's orientation θ_x over time, and the PID controller gives a much smoother control. In sub-figure (b) we can also see that the PID controller gives smoother outputs, while the PPO agent seems to learn to abuse alternating the motors between full and zero power, and use the duty cycle to control the lifting force.

From our experiments, we also found that the reward function (Equation (1)) plays a significant role in the trained policy. The parameters C_θ and C_ω must be carefully tuned so that the agent will learn the desired behavior. For example, if C_θ and C_ω are set too low, while the trained agent can still keep the drone staying at the target point, it will also make the drone rotate at an absurdly high speed. If these parameters are set too high, the agent simply give up. It chooses to shut down the motor completely and let the drone free-fall to avoid any angular motion.

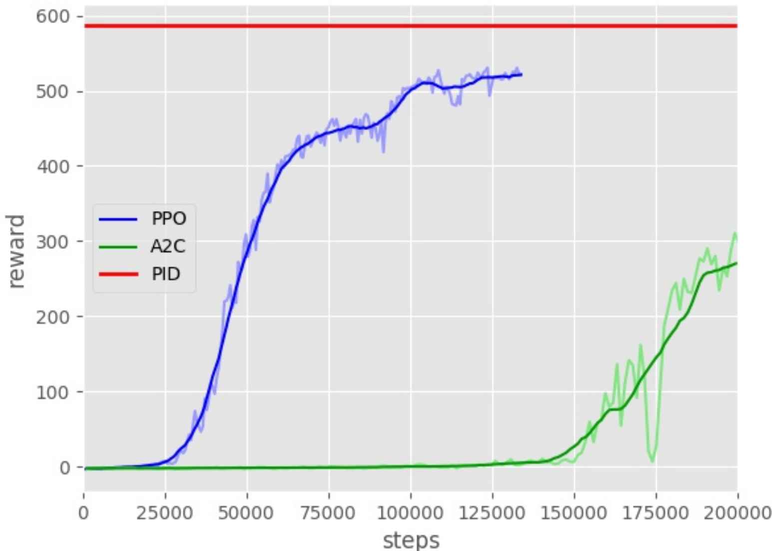
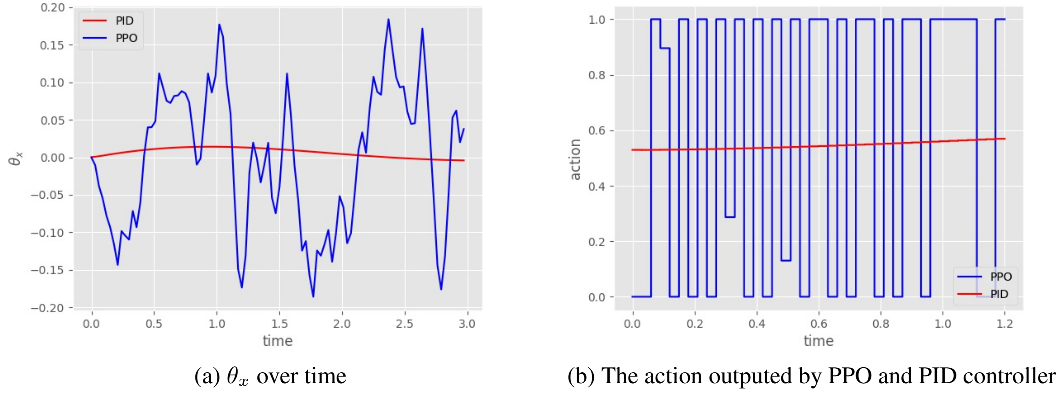
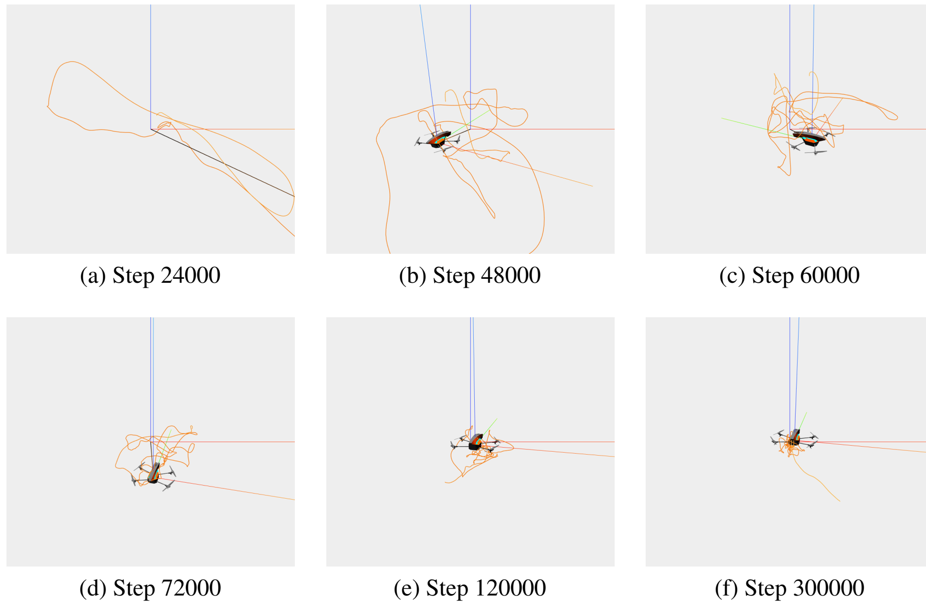


Figure 2: Reward over trained steps



(a) θ_x over time
(b) The action outputed by PPO and PID controller

Figure 3: Comparison between PPO and PID controller



(a) Step 24000
(b) Step 48000
(c) Step 60000
(d) Step 72000
(e) Step 120000
(f) Step 300000

Figure 4: Behavior of the quadcopter for each time step for PPO

6 Future work

There are some extensions that could be made in the future. First, compare to the output of our PPO agent, the action output by the PID controller is more desired. One solution may be adding an additional term in the reward function (equation (2)) to penalize the behavior of rapidly changing the thrust. We may also need to change the normal distribution in the actor net to other distribution that will encourage the sampled output to stay in the middle of the action space. Currently we observe that after training, the mean μ becomes too large or small so that the sampled action is often 1 or 0.

Moreover, we assume that there is no delay in changing the power of the motors. However, to better resemble the real-world, a delay should be added. This modification will cause the state transition function depends on previous states and actions, and we might need to add short term memory units such as LSTMs into our neural networks.

Finally, we believe that the biggest advantage of RL algorithms over classical PID controllers is that RL algorithms have the flexibilities to learn a wide range of goals (such as performing a flip or

avoiding obstacles) autonomously, while PID controllers have to be tuned by a human for each new objective. This advantage is not shown in this work, and one possible future work is to extend the goal to more complex ones.

7 Conclusion

In this paper, we build a quadcopter simulator that can simulate a flying drone based on the equation of motion and rotation. We successfully train two RL algorithms, A2C and PPO, with the environment. The resulting controllers are capable of stabilizing and navigating a quadcopter to the target. Both A2C and PPO models can perform the task well after trained with enough time steps. We also observe that PPO performs much better and learns faster compared to A2C. Still, a PID controller gives a better result and more stable control. However, a PID controller requires human modeling and tuning while RL-based controllers do not require to have previous knowledge of the physics or dynamics of the quadcopter.

8 Contribution

1. Fang-I Hsiao
   - Worked on the implementation of PPO and tested it in our environment.
2. Cheng-Min Chiang
   - Modified the simulation environment and made it compatible to OpenAI gym API.
   - Modified and fine-tuned the PPO and A2C agents and trained it with our environment.
3. Alvin Hou
   - Modified the web drone environment to visualize and capture the quadcopter's navigation results.
   - Implemented and tested DQN on Breakout-v0 as an experimental result.

9 Acknowledgement

Part of the simulator, visualizer, and PID controller code originates from the work of `leomao` and `step5` on github [6] [5].

References

[1] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[2] Jemin Hwangbo et al. "Control of a Quadrotor With Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 2.4 (Oct. 2017), pp. 2096–2103. ISSN: 2377-3774. DOI: `10.1109/lra.2017.2720851`. URL: `http://dx.doi.org/10.1109/LRA.2017.2720851`.

[3] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement Learning in Robotics: A Survey". In: *Int. J. Rob. Res.* 32.11 (Sept. 2013), pp. 1238–1274. ISSN: 0278-3649. DOI: `10.1177/0278364913495721`. URL: `http://dx.doi.org/10.1177/0278364913495721`.

[4] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement Learning in Robotics: A Survey". In: *Int. J. Rob. Res.* 32.11 (Sept. 2013), pp. 1238–1274. ISSN: 0278-3649. DOI: `10.1177/0278364913495721`. URL: `http://dx.doi.org/10.1177/0278364913495721`.

[5] leomao and step5. *AlphaPiAr*. `https://github.com/AlphaLambdaMuPi/AlphaPiAr`. 2015.

[6] leomao and step5. *WebDrone*. `https://github.com/AlphaLambdaMuPi/WebDrone`. 2015.

[7] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[8] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[9] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. 2016, pp. 1928–1937.

[10] Viswanadhapalli Praveen, S Pillai, et al. "Modeling and simulation of quadcopter using PID controller". In: ().

[11] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[12] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. 2015, pp. 1889–1897.

[13] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–503. URL: `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`.

[14] Richard S Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12*. Ed. by S. A. Solla, T. K. Leen, and K. Müller. MIT Press, 2000, pp. 1057–1063. URL: `http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf`.