Project Report: Reinforcement Learning-Based Quadcopter Stabilization Using CoppeliaSim

Nir Manor, Gil Gur Arieh

May 2024

Link to project on GitHub

1 Introduction

This work is the final project for the Advanced Topics in Robotics course. The primary objective of this project is to develop a reinforcement learning (RL) model to control a quadcopter flight into a fixed goal and stabilize it, without utilizing any control model or having any prior knowledge of the drone physics. The project utilizes the CoppeliaSim simulation environment to test and train the RL agent.

2 Background

2.1 Reinforcement Learning Overview

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions by performing actions in an environment to maximize cumulative reward. The key components of RL include:

- **Agent**: The learner or decision-maker.
- **Environment**: The external system the agent interacts with.
- **State (s)**: A representation of the environment at a specific time.
- **Action (a)**: a possible move the agent can take.
- **Reward (r)**: Immediate feedback from the environment based on the agent's action.
- **Policy (π)**: A strategy that maps states to actions.
- **Value Function (V)**: Estimates the expected reward for a given state.

- **Q-Function (Q)**: Estimates the expected reward for a given state-action pair.

The RL process involves the agent observing the current state, selecting an action based on its policy, receiving a reward, and transitioning to a new state. This cycle continues, and the agent learns to improve its policy to maximize the cumulative reward over time.

2.2 RL algorithms

2.2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a popular RL algorithm that balances ease of implementation and performance. PPO belongs to the class of policy gradient methods, which directly optimize the policy.

Key Features of PPO:

- **Clipped Objective Function**: PPO optimizes a clipped surrogate objective function, which helps to maintain the stability of the training process by preventing large updates to the policy.
- **Trust Region**: PPO ensures that the new policy is not too far from the old policy by using a trust region constraint.
- **Advantage Estimation**: PPO uses Generalized Advantage Estimation (GAE) to reduce variance in policy updates and improve learning efficiency.

PPO Algorithm Steps:

1. **Initialize Policy and Value Networks**: Start with random policy and value networks.
2. **Collect Trajectories**: Interact with the environment using the current policy to collect a set of trajectories.
3. **Compute Advantage Estimates**: Calculate advantage estimates for each state-action pair using the collected trajectories.
4. **Optimize the Policy**: Update the policy network by maximizing the clipped surrogate objective function.
5. **Update the Value Function**: Update the value network by minimizing the prediction error of the value function.
6. **Repeat**: Continue collecting new trajectories and updating the networks until convergence.

2.2.2 Actor-Critic Methods

Actor-Critic methods are a type of RL algorithm that use two separate neural networks:

1. **Actor**: The policy network (actor) selects actions based on the current state.
2. **Critic**: The value network (critic) evaluates the actions by estimating the value function.

How Actor-Critic Works:

1. **Initialize Networks**: Start with random actor and critic networks.
2. **Select Action**: The actor selects an action based on the current state.
3. **Observe Reward and New State**: The environment returns a reward and a new state.
4. **Evaluate Action**: The critic evaluates the action by estimating the value function.
5. **Compute Advantage**: Calculate the advantage, which is the difference between the observed reward and the expected value.
6. **Update Actor**: The actor is updated using the advantage to improve the policy.
7. **Update Critic**: The critic is updated to better estimate the value function.
8. **Repeat**: Continue interacting with the environment, updating the actor and critic until convergence.

3 Project Components

The model for this project has three major components:

1. **CoppeliaSim scene** - contains the physical drone model (from the existing mobile robot models). The control scheme in the Lua childscript was removed, and thrust values are taken from the API signals and sent directly to the "handle propellers" function.
2. **Gymnasium (Gym) environment** - contains the RL environment modelling. The Gym environment sends actions to the CoppeliaSim scene and reads the object positions and orientations which compose the states. The environment then calculates the immediate reward for each state and action pair.

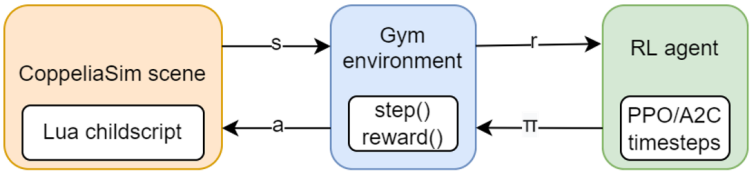

Figure 1: Visualization of the code structure for the project

3. **RL Algorithm** - The learning algorithm defines the policy which determines which actions will be sent to the environment. The algorithm collects the rewards and updates the policy in accordance with the algorithm.

The code structure is demonstrated in figure 1. More details on the code components in the following sections.

3.1 CoppeliaSim Scene

The CoppeliaSim scene contains the physical drone model, which was adapted from an existing mobile robot model. The Lua childscript was modified to remove the control scheme, allowing thrust values to be taken directly from API signals and sent to the "handle propellers" function, which converts the thrust values of each propeller ($k_i\omega_i^2$) to the force and the torque applied at the propeller. Full details of the Lua script are available in the accompanying CoppeliaSim file. During the trial and error process between simulations, the CoppeliaSim scene was simplified by removing the air particles (which served as visualization as well as a control factor in the original drone model) and by removing the floor. The CoppeliaSim scene is shown in figure 2. More on CoppeliaSim-Python synchronization in following sections.

3.2 Gym Environment

For this project, a custom "Gymnasium" (formerly "Gym") environment was constructed. Gymnasium environments are a common, open tool for developers and researchers to construct Markovian environments, best suited for MDP and Reinforcement Learning problem solving. The file containing the Gym Environment is `DroneEnv.py`. The environment components are described in the subsections below.

3.2.1 State Space

Each state is defined by a 16×1 vector with the following information, taken from the CoppeliaSim simulation:

- Drone orientation in quaternion form [4].
- Drone cartesian position in global coordinates [3].

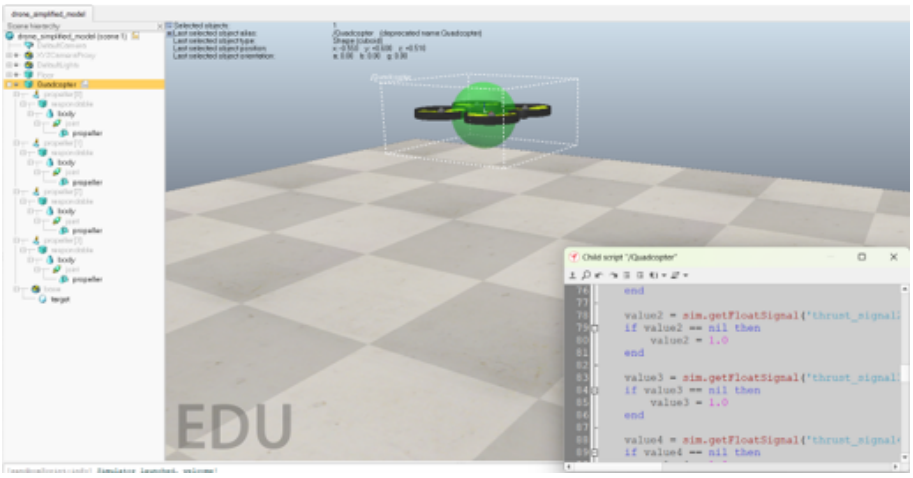

Figure 2: The CoppeliaSim scene

- Drone Linear velocity in global coordinates [3].
- Drone Angular velocity in Drone coordinates (alpha, beta, gamma) [3].
- Target position in global coordinates [3].

3.2.2 Action space

Each action sent to the CoppeliaSim scene is a 4×1 vector representing the propeller thrust values. The range of thrust values needed to fully control a quadcopter should be up to 2.5 times its overall weight. For our quadcopter, the center of mass is 0.12 kg and each propeller link is 0.1 kg (totaling 4 links). The maximum thrust for each motor can be calculated as follows:

$$\text{Total weight} = 0.12 \text{ kg} + 4 \times 0.1 \text{ kg} = 0.52 \text{ kg} \tag{1}$$

$$\text{Maximum total thrust} = 0.52 \text{ kg} \times 2.5 \times 9.81 \text{ m/s}^2 = 12.755 \text{ N} \tag{2}$$

$$\text{Maximum thrust per motor} = \frac{12.755 \text{ N}}{4} = 3.18825 \text{ N} \tag{3}$$

Therefore, the action space is $[0, 3.18825]^4$.

3.2.3 Reward calculation

The reward shape was chosen to have the following form:

$$r(s) = c_0 - c_1 \left(|\alpha| + |\beta|\right) - c_2 \left|z_{drone} - z_{target}\right| - c_3 \sqrt{(x_{drone} - x_{target})^2 + (y_{drone} - y_{target})^2}$$

An additional constant reward value (c_T) is subtracted if the state is terminal.

3.2.4 Environment step

The `env.step(action)` method is a required method in any Gymnasium environment. Its input is an action within the action space, and its outputs are the subsequent state, immediate reward, terminated (boolean, true if the new state is terminal) and truncated (boolean, positive if the maximum step count is reached).

Send action to CoppeliaSim The action input is separated into four thrust values which are sent to CoppeliaSim as float signals. The synchronization trigger is then sent, which allows CoppeliaSim to run a timestep with the sent values.

Get new state Positions, angles and velocities are read from CoppeliaSim using the appropriate API methods. Step counter is then increased by 1, and `truncated` is set true if the maximum step level is reached.

Terminal state check The new state parameters are checked to see if any of the termination conditions apply. Termination conditions that were used and varied during the simulation trial-and-error improvements are:

- Horizontal distance (projection of the euclidean distance in the x-y plane) is larger than a cutoff value.
- Vertical distance (in z values) is larger than a cutoff value.
- Ground collision (center of mass is below a cutoff distance from the floor. This condition was discontinued when the floor was removed from the model).
- Angle too big (if roll or pitch angles are too large so that the drone could hardly stabilize itself, if at all).

Reward The new state and the termination condition are sent to the reward calculating method, which returns the immediate state reward.

3.2.5 Environment Reset

The `env.step()` method is a required method in any Gymnasium environment. The method resets the environment and outputs a state.

Stop the simulation The method starts with stopping the running simulation and waiting a fixed amount of time to make sure CoppeliaSim completes stopping. The synchronization is also reset (synchronization will be further discussed in the following sections).

Setting initial positions and orientation Introduced in one of the improvements between simulation sequences, this part sets the drone and target initial positions to random values within a pre-defined box around the original initial positions, and sets the drone pitch and roll angles to slightly deviate from the upright initial position. The values are set to CoppeliaSim through ordinary API functions such as `setObjectPosition`. The state is then read from CoppeliaSim in the same was it is read in the `step` method, to make sure the values have indeed been sent and to make sure the state is represented in the same way as it is in `step`.

3.3 RL framework

The Reinforcement Learning code is present in `rl_main.py`. A2C and PPO agents were taken "off the shelf" using `stable_baselines3`. Both algorithms involve an actor and a critic neural network, which were chosen to have two hidden layers with 128 neurons at each layer (The layers initially had 64 neurons each, but the number was increased as one of the improvements introduced during the process). The number of training timesteps was changed between simulations. A discount factor was included so that the model accounted for the one-time terminal state reward.

4 Major Challenges

4.1 Python and CoppeliaSim Integration and Synchronization

4.1.1 Operation Modes for Interacting with CoppeliaSim

CoppeliaSim offers different operation modes to handle communication between Python and the simulator. The two primary modes used in this project are:

simx_opmode_oneshot:

- **Purpose**: Non-blocking updates where immediate execution is required without waiting for a response.
- **Usage**: Suitable for setting multiple thrust signals in quick succession.
- **Example**:

```
sim.simxSetFloatSignal(self.client_ID, 'thrust_signal1',
    thrust[0], sim.simx_opmode_oneshot)
```

simx_opmode_blocking:

- **Purpose**: Blocking calls to ensure accurate state retrieval.

- **Usage**: Waits for the command to be executed and the result to be returned before proceeding. This mode ensures that the state information received is up-to-date.
- **Example**:

```
_, drone_quat = sim.simxGetObjectQuaternion(client_ID,
    heli, -1, sim.simx_opmode_blocking)
```

4.1.2 Synchronization Challenges and Solutions

Challenges:

- Ensuring that the Python code and CoppeliaSim simulation are perfectly synchronized was challenging. Inconsistent synchronization could lead to actions being applied based on outdated states, causing instability in the quadcopter's behavior.
- Properly handling the simulation reset to ensure the quadcopter starts from the correct initial state each episode.

Solutions:

Synchronous Mode:

- **Description**: Enabling CoppeliaSim's synchronous mode ensures the simulation only progresses when triggered by the Python code.
- **Implementation**:

```
sim.simxSynchronous(self.client_ID, True)
sim.simxStartSimulation(self.client_ID, sim.
    simx_opmode_oneshot)
```

Synchronous Trigger:

- **Description**: The `simxSynchronousTrigger` command is used in each step to manually advance the simulation by one step, ensuring precise control over the simulation timing.
- **Implementation**:

```
sim.simxSynchronousTrigger(self.client_ID)
```

Ensuring Command Execution:

- **Description**: The `simxGetPingTime` function ensures that the commands sent to CoppeliaSim have been processed. This function sends a ping command to the server and waits for a reply, ensuring that all previously sent commands have been executed.
- **Implementation**:

```
sim.simxGetPingTime(self.client_ID)
```

Pause and Resume Communication:

- **Description**: The `simxPauseCommunication` command ensures that multiple thrust signals are sent in one communication packet, reducing the risk of partial updates.
- **Implementation**:

```
sim.simxPauseCommunication(self.client_ID, True)
sim.simxSetFloatSignal(self.client_ID, 'thrust_signal1',
    thrust[0], sim.simx_opmode_oneshot)
sim.simxSetFloatSignal(self.client_ID, 'thrust_signal2',
    thrust[1], sim.simx_opmode_oneshot)
sim.simxSetFloatSignal(self.client_ID, 'thrust_signal3',
    thrust[2], sim.simx_opmode_oneshot)
sim.simxSetFloatSignal(self.client_ID, 'thrust_signal4',
    thrust[3], sim.simx_opmode_oneshot)
sim.simxPauseCommunication(self.client_ID, False)
```

Delay for Stabilization:

- **Description**: Introducing a short delay after resetting the simulation allows the quadcopter to stabilize before starting a new episode.
- **Implementation**:

```
time.sleep(0.01)  % Sleep for 0.01 seconds to let the
    simulation stabilize
```

4.1.3 Adjusting Simulation Parameters

Description: To ensure the simulation runs correctly and in sync with the reinforcement learning process, we adjusted the Simulation Time Step and Dynamics Time Step in CoppeliaSim to be equal to each other, setting both to 10 ms (0.01 seconds).

4.2 Reward Shaping, Termination and Truncation

Choosing the right values for the reward constants and intensifying or releasing the termination criteria have been major challenges during the process of working on this project, and we consider them to still be open. We recall the reward is of the form:

$$r(s) = c_0 - c_1\left(|\alpha| + |\beta|\right) - c_2\,|z_{drone} - z_{target}| - c_3\sqrt{(x_{drone} - x_{target})^2 + (y_{drone} - y_{target})^2}$$

Constant positive reward The c_0 term was introduced in one of the simulation stages. Our experience showed that having non-positive immediate reward might "encourage" the learning agent to prefer terminating episodes to avoid accumulating negative reward rather than trying to stay in the air and minimize the absolute value of the negative rewards. This term makes sure that the immediate reward at any non-terminal step is always positive.

Pitch and Roll angles penalty The c_1 term gives negative rewards for high angular deviations from the upright drone position. While it can be claimed that this term is redundant (knowing that a drone must stay upright to stay in a fixed position), we found using this term practical, especially in early stages of the simulation sequence.

Vertical and Horizontal distance from target Different values were tested for the vertical and horizontal distance from target reward. In some multi-stage simulation sequences, the horizontal and vertical constants were set to zero during some training stages in order to focus on the drone stabilizing, and were later introduced either individually or together.

Maximum angle termination condition In all simulation settings, a maximum pitch and roll value was set so that if the drone reaches it, the state becomes terminal. The values were tested between 15 and 85 degrees, with a lower value ensuring drone stabilization but shortening simulation time, preventing the drone from obtaining low-reward experience.

Reached ground termination condition Earlier versions of the CoppeliaSim scene included a floor, on which the drone was able to crash. A termination condition on the drone height (center of mass z-value) was added to prevent simulation time being spent with the drone on the floor.

Too far from target termination conditions Separate z and x-y distance values were tested, creating various cylindrical spaces around the target in which the drone could be.

Truncation condition Knowing that each CoppeliaSim time step is 0.01 seconds, we used truncation times corresponding with 2-3 seconds (200-300 timesteps). In practice, the drone was not able to stabilize itself for simulations as long (termination conditions were met before truncation condition), and truncation was discontinued (technically, set to 2000 timesteps = 20 seconds).

4.3 RL Algorithm and Modelling

Certain model decisions were taken for the RL agent as well. We consider most of these choices as open challenges.

Learning Algorithm Both A2C and PPO were used in simulations in this project. While A2C is more familiar to us, simpler to implement and is generally faster, PPO is more sample-efficient and stable.

Neural Network Size Initially, actor and critic networks were each chosen to have two hidden layers with 64 neurons each, as was default in the code implementation. After some online consultation, we decided to widen the networks to have 128 neurons in each hidden layer, to better fit the action and state space sizes.

Discount Factor A discount factor value of $\gamma = 0.95$ was chosen. Given that the immediate step reward (in absolute value) is roughly 0-3, This value of the discount factor ensured that a one-time negative reward of 100 was "felt" about 70 steps before termination and a reward of 10 about 20 steps.

Timesteps When consulting with ChatGPT, given our model size, algorithm and state-action space size, 1 million steps were recommended. On our standard computers, a 100,000-step simulation takes roughly 9 hours, which allows for 2 such simulations each human day. This has proved to be the biggest challenge of this project, as simulations were cut off due to computer crashes, power-outs and other reasons. After every successfully-run simulation, the resulted model was observed and modifications to the Environment and model were introduced. Changes in the environment made it illogical to continue the simulation in a sequence, so training was started over. The largest amount of timesteps we achieved in a simulation sequence (our latest) is 350,000.

5 Results

All the resulted models and figures of mean reward during evaluation steps are saved in the `models` folder. The first digit in the model name is the simulation sequence and the second digit is the stage within the sequence. Notes on changes to the code (and sometimes the code itself) are also saved in the folders.

Sequence 0 tried both three-stage and two-stage training. In three-stage training, the initial state takes into account only rewards relating to angular deviations from the upright position, then introduces vertical distance from the target and then horizontal. This resulted in the drone converging to a policy with which it descends to the ground slowly enough (though not turning the propellers completely off) so that it stays upright until truncation. Two-stage training starts with both stabilization and angular penalty, trying to prevent that.

Sequence 1 utilized a total reward function, and introduced the positive constant to each reward value.

Sequence 2 introduced both the physical scaling of the action space as well as the randomized initialization of the target position and drone position and orientation. This sequence resulted in the drone completely shutting off 2 of its 4 propellers, using only 2 nonzero thrust values for control.

Sequence 3 eliminated all termination criteria and reduced the truncation condition back to 200 steps, after seeing that nearly every episode ends in termination before the drone obtained much episodic experience. During this sequence we also switched from A2C to PPO.

6 Conclusion and Future Work

This course project combined our newly-acquired knowledge of utilizing CoppeliaSim scenes to simulate robotics with our interest and curiosity with Reinforcement Learning in robotics. With some challenges overcome and others in the way of being overcome, the most significant setback during the work on this project was the long simulation time.

After the discussion in class, it is suggested to use multiple drones in the simulation and running multiple agents at each timesteps, in order to increase the information gained in each timestep. Running the simulation on a stronger computer (multi-core computer, for example) is also advised.

Shorter simulation time could help solve the other open challenges, namely reward shaping and the choice of model parameters.