QuadCtrl

Quadcopter Controller based on Deep Reinforcement Learning

Alessandro Maggio
Serban Cristian Tudosie

Artificial Intelligence in Industry

Alma Mater Studiorum - Università di Bologna

March 18, 2022

Problem Overview

The goal of this work is to control a quadcopter drone in a simulated physical environment to achieve simple tasks based on a model-free approach.

Usual way:

- Usually domain knowledge is required to achieve good results.
- Control Theory combined with Physics insights are necessary.
- Very case-dependent and not generalizable.

Our way:

- General approach, easy to extend for other use cases
- No Physics or Control Theory knowledge required
- Hybridable with traditional techniques

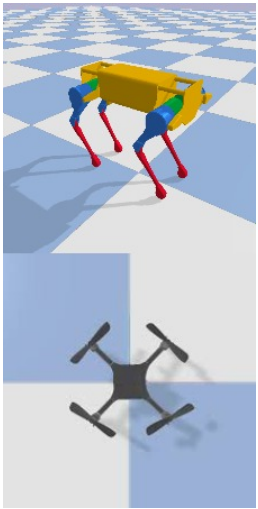
Index

A Deep Reinforcement Learning Approach

Among all possible alternatives in DRL, we have chosen a **model-free** one to embrace the most generalizable technique.

This allows a faster transition to other models, such as having a helicopter instead of a quadcopter.

Even changing the problem frame into an autonomous vehicle would not require any substantial rework of the project.



A Deep Reinforcement Learning Approach

We have decided to produce two types of controllers for our quadcopter:

- **Direct drone control**: The Neural Network is employed both at training and testing time, directly controlling the engines rotation speed. In real world experiments, the NN should be deployed on the on-board system.

- **Hybrid approach**: for the Hovering task, the Neural Network is trained to tune a PID controller which is able to achieve the tasks. At testing time only the PID has to be runned on the embedded device, so computationally much cheaper.

Tasks Description

We designed two different tasks for this project:

Hovering Task

- Drone starts from a certain point in 3D space and the goal is to remain still and stabilized in it.
- It must be robust to both white noise and selective noise (wind simulation).

Movement Task

- Drone starts from a random altitude and it must reach the target position
- Has to keep the full control before and after the goal is reached.
- Robust to noise as in Hovering.

Our DRL Implementation

- No free lunch! -
Adopting a DRL technique forces to take many design decisions. The most relevant choices in our case have been:

- Picking an algorithm (on-policy vs off-policy)
- Build the reward functions (very time consuming ⚠)
- Network architecture: depth, width, optimizer etc.
- Hyperparameter tuning: training duration, episode length, warmup time

Our DRL Implementation - Algorithm Choice

After trying A2C, SAC and TD3 then we decided to opt out for Proximal Policy Optimization (PPO), an on-policy algorithm.

Why on-policy?

- We work in continuous 3D space
- Big exploration space $\rightarrow$ on-policy works better
- Learning the policy while performing actions $\rightarrow$ helps convergence

Why PPO?

- State of the art
- Easier to tune
- Robust to small perturbations of hyperparameters

Our DRL Implementation - Build the reward functions

A crucial point in RL is handcrafting a reward function which represents in the best way what we are aiming for. All the coefficient and parameters have been found empirically.

Hovering reward function:

- Biased on keeping the right altitude
- Combines square and absolute distance
- Identical for both direct and hybrid approach

```
def _reward_standard(self):
    state = self._getDroneStateVector(0)
    dist_z = 0.9 * (abs(1 - state[2]))
    dist_x = 0.1 * ((0 - state[0]) ** 2)
    dist_y = 0.1 * ((0 - state[1]) ** 2)
    r = - dist_z - dist_x - dist_y
    return r
```

Our DRL Implementation - Build the reward functions

Moving reward functions:

- Two functions: warmup and fine-tuning
- The former is biased to respect the altitude
- The latter helps focusing on the x-y position

```
def _reward_w_penalty(self):
    state = self._getDroneStateVector(0)

    penalty_z = 0
    if (1.4 - state[2]) < 0:
        penalty_z = np.abs(1.4 - state[2]) / 4

    dist_x = ((0 - state[0]) ** 2 - penalty_z)
    dist_y = (0 - state[1]) ** 2
    dist_z = (1.2 - state[2]) ** 2

    r = -(0.9 * dist_z + 0.05 * dist_x + 0.05 * dist_y)
    return r
```

Our DRL Implementation - Network architecture

The PPO algorithm is based on the Actor-Critic method. The two networks share the first layers whose input is the current state. Starting with two Denses [128, 128] it splits in:

Actor Head:

- Select the actions $\rightarrow$ vector output
- Denses [128, 256]
- training duration, episode length, warmup time

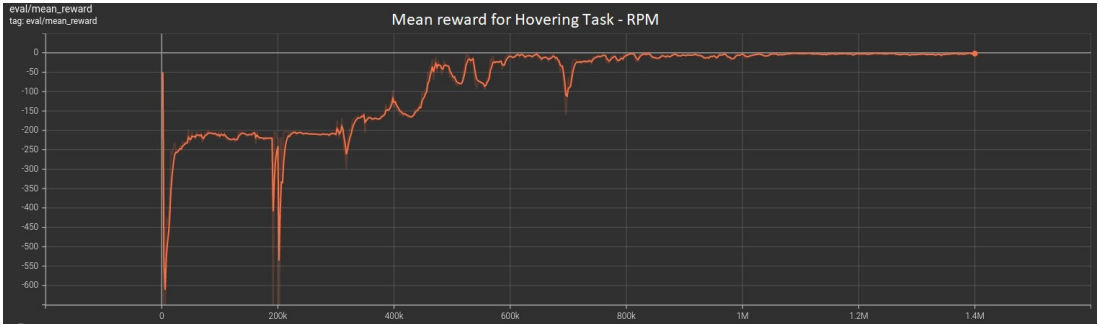
Critic Head:

- Evaluate the actions $\rightarrow$ scalar output
- Dense [256]
- training duration, episode length, warmup time

Trade-off between complexity and model capacity.

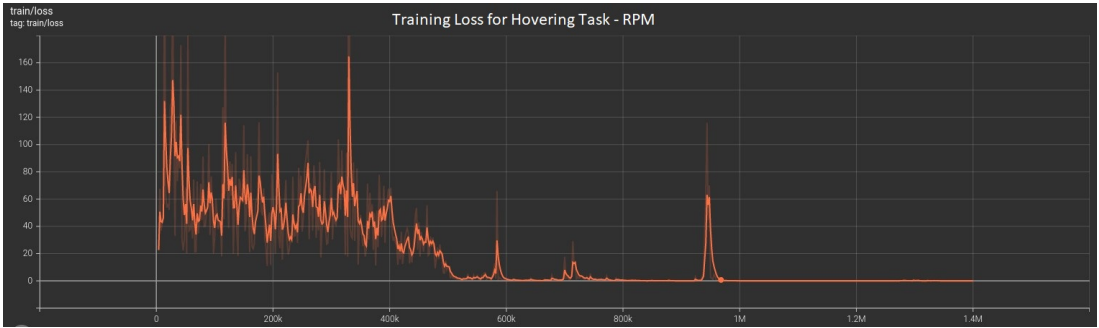
Training

The training phase has a variable duration depending on the task. The performances have been evaluated with **Tensorboard**, observing the mean reward and the loss trend.



Training

The training phase has a variable duration depending on the task. The performances have been evaluated with **Tensorboard**, observing the mean reward and the loss trend.



Testing

In order to reproduce a more realistic environment, we have introduced two kind of **noise at testing time**: a white noise and an impulsive perturbation. The white noise affects the entire drone state (position, roll/pitch/yaw, angular velocity, linear velocity). The impulsive perturbation is applied for a short time but represents an heavy external push to the drone along the x and z axes.

- White noise 1: mean $= 0$, std $= 0.03$.
- Impulsive noise 1: mean $= 0.2$, std $= 0.05$, 1 second.
- White noise 2: mean $= 0$, std $= 0.08$.
- Impulsive noise 2: mean $= 0.3$, std $= 0.1$, 1 second.
- Impulsive noise 3: mean $= 0.6$, std $= 0.1$, 1 second.

Testing

Together with a visual feedback of the simulation, we adopted a quantitative report of the experiments thanks to the plots of the state evolution during the entire run.

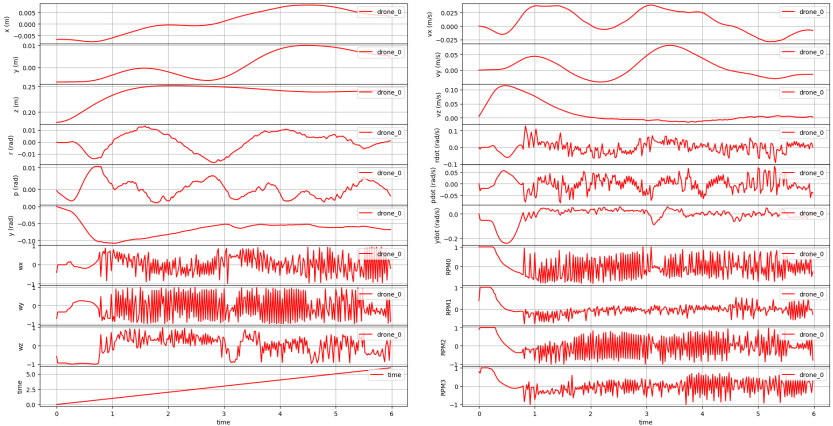


Figure: Six seconds simulation plot for the Moving task without noise.

Results

The following tab summarizes the outcomes our main experiments.
We have also tried to initialize the quadcopter with **non-zero** roll and pitch. In this case, PID responded quickly also for large tilts, while the direct RPM control supports only small angles.

	No Noise	White Noise 1	Impulsive Noise 1	White Noise 2	Impulsive Noise 2	Impulsive Noise 3
Hovering PID	✔	✔	✔	✔	✔	✔
Hovering RPM	✔	✔*	✔	✘	✔*	✔*
Movement RPM	✔	✔	✔	✘	✔	✔*

Figure: (*) some difficulties in stabilization

Conclusions

Our conclusions:

- As we can observe, the model is robust to limited perturbations in time or intensity but the full-DRL controlled drone does not compensate a persistent high magnitude noise.
- The PID Controller (trained with PPO) is clearly the winner, also in tilt experiments.
- This result is intuitive since a **hybrid approach between AI and the discipline related to the problem domain**, Control Theory in our case, is quite always the best choice.
- Thanks to the great Deep Learning power of overcoming the human limits in finding a good tuning, we demonstrated that it is possible to build a drone controller even in a model-free context (and in just one hour training!).

And now some clips...

QuadCtrl

Thanks for your attention