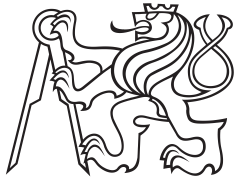
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Position Control of a Quadrotor with a Hanging Load Using Deep Reinforcement Learning

Tomáš Tichý

Supervisor: Ing. Teymur Azayev, Ph.D.
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Tichý Tomáš** Personal ID number: **474617**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Position Control of a Quadrotor with a Hanging Load Using Deep Reinforcement Learning

Master’s thesis title in Czech:

Pozi ní ízení kvadkoptéry s visící zát ží pomocí hlubokého posilovaného u ení

Guidelines:

The task looks at end to end control of drones with a hanging load using Reinforcement learning (RL). The observations should be the drone attitude and available velocities, as well as the target waypoint (or several waypoints in sequence). The outputs are the individual continuous motor actuations.
This approach is traditionally approached using an inner stabilization loop and an outer control loop that deals with transport delay. One of the difficulties of this task is the unobserved angle of the pendulum with respect to the drone. We implement and analyze approaches that explicitely estimate the unobserved variables using a Recurrent Neural Network (RNN) and feeding it as input to the control agent, as well as implementing the agent as the RNN, making the estimation implicit and resulting in end-to-end control.
The task also examines the robustness of adaptation to changing drone parameters and pendulum weight using a state of the art approach called Rapid Motor Adaptation (RMA), that does online parameter estimation and control simultaneously.
This diploma thesis is a follow up work on a prepared software simulation project done previously.
The deliverables of the thesis are as follows:

- Analyze the possibilities of training a positional control agent using RL with and without the pendulum angles as input and quantify how well the angles can be estimated from the state action history using an RNN.
- Train a positional control agent using RL on a environment with non-changing parameters, as well as one where the parameters of the drone and pendulum change every episode. An explicit adaptation algorithm such as the RMA [1] approach or similar can be used for adaptation.
- Attempt to train a pure RNN agent that can implicitly perform adaptation end to end on the parameter-changing environment and compare performance with the explicit approach that uses the RMA algorithm.
- Optionally compare the RL agent on a single environment (or small set) against a model predictive control or signal shaping approach.
- Optionally run the RL algorithm on a real small drone with a realsense t265 camera (or similar) providing localization and demonstrate basic positional control.

Bibliography / sources:

[1] Kumar, Ashish et al. “RMA: Rapid Motor Adaptation for Legged Robots.” ArXiv abs/2107.04034 (2021): n. pag.
[2] Song, Yunlong, Mats Steinweg, Elia Kaufmann and Davide Scaramuzza. “Autonomous Drone Racing with Deep Reinforcement Learning.” 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2021): 1205-1212.
[3] Busek, Jaroslav, Mat j Ku e, Martin Hromcik and Tomáš Vyhlídal. “Control Design With Inverse Feedback Shaper for Quadcopter With Suspended Load.” Volume 3: Modeling and Validation; Multi-Agent and Networked Systems; Path Planning and Motion Control; Tracking Control Systems; Unmanned Aerial Vehicles (UAVs) and Application; Unmanned Ground and Aerial Vehicles; Vibration in Mechanical Systems; Vibrations and Control of Systems; Vibrations: Mode (2018): n. pag.

© VUT v Praze, Design: VUT v Praze, VIC

Name and workplace of master's thesis supervisor:

Ing. Teymur Azayev, Ph.D.   Vision for Robotics and Autonomous Systems FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **25.01.2023**   Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Teymur Azayev, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt   Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank my parents and significant other for supporting me throughout my studies. I am also very grateful to my supervisor, without whom this project would not be possible.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 26. May 2023

....
signature

Abstract

This thesis focuses on employing deep reinforcement learning to control the position of a quadcopter carrying a hanging load. The task is approached by using proximal policy optimization to directly optimize a control policy represented as a neural network using gradient ascent on the accumulated reward. The system is both trained and evaluated using the MuJoCo physics simulator. The resulting policy is capable of successfully stabilizing the quadcopter and following the reference setpoint position. This is demonstrated by using full state information and partial state information without knowledge of the suspended load state. The latter task is tackled by using a state estimator network trained via supervised learning on the pendulum states. Additionally, the thesis investigates the ability to adapt to changing model parameters, to help mitigate the Sim-to-Real gap.

Keywords: Proximal policy optimization (PPO), Deep reinforcement learning, Hanging load, Quadcopter control, MuJoCo simulator

Supervisor: Ing. Teymur Azayev, Ph.D.
AmpX,
Klimentska 1216,
Praha 1

Abstrakt

Tato práce se zaměřuje na využití hlubokého posilovaného učení pro poziční řízení kvadrokoptéry s visící zátěží. Tento úkol je řešen pomocí metody proximal policy optimization, kde se přímo optimalizuje řidící policy reprezentovaná neuronovou sítí pomocí gradientního stoupání na kumulativní odměnu. Systém je trénován a testován pomocí fyzikálního simulátoru MuJoCo. Výsledná policy je schopna úspěšně stabilizovat kvadrokoptéru a sledovat referenční cílovou polohu. Toto je demonstrováno jak při použití úplných informací o stavu dronu, tak i při použití pouze částečných informací bez znalosti stavu zavěšené zátěže. Řízení při neúplné znalosti stavu je řešeno pomocí neuronové sítě pro odhad stavů, která je trénovaná učením s učitelem na datech o stavu kyvadla. Práce také zkoumá schopnost přizpůsobit se změnám parametrů modelu, s cílem zmírnit problém rozdílu mezi simulací a reálným světem.

Klíčová slova: Proximal policy optimization (PPO), Hluboké posilované učení, Visící zátěž, Řízení kvadrokoptéry, MuJoCo simulátor

Překlad názvu: Poziční řízení kvadkoptéry s visící zátěží pomocí hlubokého posilovaného učení

Contents

1 Introduction 1
1.1 Context 1
1.2 Goals 1

2 Theory 3
2.1 Reinforcement learning 3
2.1.1 Markov decision process 3
2.1.2 Policy gradient 5
2.1.3 Policy gradient estimation 6
2.1.4 Generalized advantage estimation 8
2.1.5 Proximal policy optimization 9
2.2 Policy architecture 11
2.2.1 Multi-layer perceptron 12
2.2.2 Long short-term memory 13
2.2.3 Convolutional network 14
2.3 Drone model 14
2.3.1 Suspended load 16
2.3.2 Roll, pitch and yaw 17
2.4 Drone control framed as an MDP 18

3 Implementation 19
3.1 Physics simulator 19
3.1.1 MJCF 19
3.1.2 Environment model 20
3.2 Training framework 22
3.2.1 VecEnv 22
3.2.2 PPO training 23
3.2.3 Policy configuration 24
3.3 BaseDroneEnv 25
3.3.1 Drone parameter generation 26
3.3.2 Initial state distribution 27
3.3.3 Environment interaction 27

4 Experiments 29
4.1 Reward parametrization 29
4.1.1 Distance based reward 29
4.1.2 Energy based reward 30
4.2 Observations 31
4.2.1 Full state observation 31
4.2.2 Full state observation with drone parameters 32
4.3 Position control 32
4.3.1 Neural network architecture 32
4.3.2 Training and environment configuration 33
4.3.3 Results 34
4.4 Pendulum state estimation 41
4.4.1 Neural network architecture 42
4.4.2 Training and environment configuration 43
4.4.3 Results 44
4.5 Model parameter adaptation 53
4.5.1 Neural network architecture 53
4.5.2 Training and environment configuration 56
4.5.3 Results 57
4.6 Discussion 61
4.6.1 Position control 61
4.6.2 Pendulum state estimation 62
4.6.3 Model parameter adaptation 62

5 Conclusion 63
5.1 Summary 63
5.2 Future work 64

A Bibliography 65

B Code structure 67

Figures

2.1 Markov decision process 4
2.2 $Beta(\alpha, \beta)$ distribution probability density function 12
2.3 LSTM computation graph 13
2.4 Ilustration of the quadcopter model 15
2.5 Ilustration of the quadcopter model with hanging load 17

3.1 Drone model assembled from primitive shapes 21
3.2 Mutltiple drones inside a single simulation instance 21

4.1 Neural network architecture 33
4.2 Control MLP training with distance-effort based reward 35
4.3 Control MLP with distance-effort based reward trajectories 36
4.4 Control MLP with distance-effort based reward position 36
4.5 Control MLP with distance-effort based reward angles 37
4.6 Control MLP training with distance effort reward 37
4.7 Control MLP with energy based reward trajectories 38
4.8 Control MLP with energy based reward position 39
4.9 Control MLP with energy based reward angles 39
4.10 Dynamic setpoint tracking trajectories 40
4.11 Dynamic setpoint tracking position and heading 41
4.12 Dynamic setpoint tracking angles 41
4.13 Policy with pendulum state estimator 42
4.14 LSTM estimator network architecture 43
4.15 CNN estimator network architecture 43
4.16 Control MLP with unknown pendulum state training 44
4.17 Control MLP with unknown pendulum state trajectories 45
4.18 Control MLP with unknown pendulum state errors 45
4.19 Control MLP with unknown pendulum state angles 46
4.20 LSTM estimator policy training 47
4.21 LSTM estimator training 47
4.22 LSTM estimator policy trajectories 48
4.23 LSTM estimator policy position errors 48
4.24 LSTM estimator policy angles . 49
4.25 LSTM estimator angle estimates 49
4.26 CNN estimator policy training 50
4.27 CNN estimator training 50
4.28 CNN estimator policy trajectories 51
4.29 CNN estimator policy position errors 51
4.30 CNN estimator policy angles . . 52
4.31 CNN estimator angle estimates 52
4.32 LSTM adaptation network computation graph 54
4.33 RMA computation graph 55
4.34 RMA environment factor encoder computation graph 56
4.35 RMA adaptation module computation graph 56
4.36 LSTM adaptation training 57
4.37 LSTM adaptation trajectories . 58
4.38 LSTM adaptation position errors 58
4.39 LSTM adaptation angles 59
4.40 RMA policy training 59
4.41 RMA adaptation module training 60
4.42 RMA trajectories 60
4.43 RMA position errors 61
4.44 RMA angles 61

B.1 Code structure 67

Tables

3.1 PPO training parameters 23
3.2 BaseDroneEnv configuration parameters 25
3.3 Drone parameter distribution settings 26
3.4 Drone initial state distribution settings 27

4.1 PPO training configuration 34
4.2 BaseDroneEnv training configuration 34
4.3 Estimator training configuration 44
4.4 Performance comparison rewards 53
4.5 State estimation accuracy comparison 53
4.6 Adaptation training configuration 56
4.7 Adaptation training configuration 57

Chapter 1

Introduction

1.1 Context

Quadcopters, also known as drones, have emerged as a transformative technology with significant importance in today's world. Their versatility, agility, and remote-controlled capabilities make them invaluable in diverse fields such as aerial photography, surveillance, search and rescue operations, environmental monitoring, and delivery services. These unmanned aerial vehicles offer enhanced efficiency, cost-effectiveness, and accessibility, revolutionizing industries and enabling novel applications that were previously unattainable.

The capability of drones to transport loads through aerial means is of significant importance due to its wide range of practical applications. This feature allows drones to transport supplies, deliver packages, and undertake tasks such as aerial construction, infrastructure maintenance, or emergency aid delivery in challenging or inaccessible environments. However, successfully executing this task poses significant technical challenges, including maintaining stability and ensuring precise control. Overcoming these complexities necessitates the development of advanced engineering solutions and sophisticated control algorithms.

One promising approach to designing control algorithms for drones is through the utilization of deep reinforcement learning [3], [7], [4]. These methods build upon the remarkable advancements in deep learning achieved in recent years, particularly in areas such as image recognition and natural language processing. By leveraging the principles of deep reinforcement learning, drones can learn to make decisions and adapt their behavior based on interactions with their environment, paving the way for efficient, adaptive, and autonomous systems.

1.2 Goals

The goals of this thesis are threefold. Firstly, to leverage deep reinforcement learning (RL) techniques in developing a positional controller for a quadcopter. This involves designing and implementing a RL-based control algorithm that enables the drone to autonomously navigate and maintain desired positions.

Secondly, to evaluate the effectiveness of a recurrent neural network (RNN) in estimating the states of a pendulum attached to the drone. This experiment aims to assess the RNN's capability to accurately capture and predict the pendulum's dynamics. Lastly, to train a neural network policy that can adapt and adjust to changing drone parameters, allowing the drone to adaptively respond to variations in its physical characteristics. These three goals inherently necessitate the development and implementation of a realistic simulation environment to facilitate RL training, integrating it with a suitable reinforcement learning framework, and establishing the neural network architecture. The simulation environment will serve as a crucial tool for training and evaluating the quadcopter's control algorithms, providing a realistic virtual platform to simulate real-world scenarios and challenges.

Chapter 2

Theory

2.1 Reinforcement learning

This section includes a brief introduction to reinforcement learning using policy gradient methods together with some interesting results.

2.1.1 Markov decision process

The problem of drone control can be modelled as a Markov Decision Process (MDP), which is formally a 4-tuple

$$(S, A, P, r)\,, \tag{2.1}$$

where S is the set of states, A is the set of actions, P is the transition function $P(s_{t+1}|s_t, a_t)$ which encodes the probability of a next state $s_{t+1} \in S$ given the current state $s_t \in S$ and an action $a_t \in A$ and finally r is the reward function which associates a real value with each state transition $r_t = r(s_{t+1}, s_t, a_t) \in \mathbb{R}$. An illustration of this model can be seen in figure 2.1. The objective in an MDP is then to find a (in general stochastic) policy $\pi(a|s)$ in order to maximize the expected return of a trajectory

$$J_\pi = \mathbb{E}_\tau \left[R(\tau) \right]\,, \tag{2.2}$$

where the expectation is taken over trajectories $\tau = (s_0, a_0, s_1, a_1, ...)$. The trajectories are generated as follows. The initial state s_0 is drawn from some probability distribution $\rho_0 : S \to \mathbb{R}^+$ on the state space S. The actions and states that follow are distributed according to the policy $a_t \sim \pi(a_t|s_t)$ and the transition function $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$. The function $R(\tau)$ is commonly defined as a sum of the rewards collected over the trajectory

$$R(\tau) = \sum_{t=0}^{T} r_t\,, \tag{2.3}$$

where T is the time horizon. If the states s_t of the MDP are not directly observable, the model can be extended to a Partially Observable Markov Decision Process (POMDP), formally a 6-tuple

$$(S, A, P, r, \Omega, O)\,, \tag{2.4}$$

where S, A, P, r are the same as in (2.1), Ω is the observation space and $O(o_t|s_t)$ is the observation model and encodes the probability of observing $o_t \in \Omega$ given the state s_t at timestep t. This introduces a new problem of estimating the unknown state s_t.

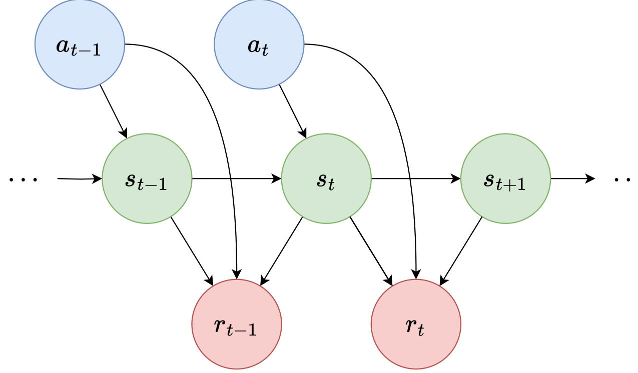


Figure 2.1: Markov decision process

Solution

While traditional methods like value iteration, policy iteration, Q-learning, and Monte Carlo methods have been effective in solving small-scale Markov Decision Processes (MDPs) [15], they face significant challenges when applied to complex control problems with continuous action and state spaces, such as drone control. The continuous nature of the action and state spaces necessitates the use of different approaches.

Traditional approaches often rely on model-based control design techniques. These methods involve analyzing simplified models of the system dynamics or employing Model Predictive Control (MPC) optimization to find suitable control strategies. These traditional approaches may struggle to capture the full complexity and dynamics of real-world systems, especially when facing uncertainties and nonlinearities. Furthermore, optimization process involved in MPC requires solving complex mathematical programs repeatedly, which can be time-consuming and limit its applicability in systems that require real-time control.

Deep reinforcement learning (RL) emerges as a promising alternative in this context, offering a powerful framework to tackle complex control problems with continuous action and state spaces. In the realm of RL, there exist numerous approaches and algorithms, each with its own strengths and limitations. For the purpose of this thesis, the focus will be on the policy gradient method, which performs a gradient-based direct policy search. The policy gradient approach has gained significant attention and demonstrated remarkable success in training agents to perform tasks in high-dimensional and continuous environments [11] [2].

2.1.2 Policy gradient

The policy gradient method focuses on directly optimizing a policy $\pi_\theta(a_t|s_t)$ parametrized by θ (see section 2.2 on how the policy is constructed). The optimization process iteratively improves the policy using gradient ascent steps

$$\theta_{k+1} = \theta_k + l_r \cdot \nabla_\theta J_\pi \,, \tag{2.5}$$

where J_π is the expected return (2.2) of the underlying MDP, l_r is the learning rate and θ_k, θ_{k+1} denote the parameters at the current and next timestep respectively. Note that more sophisticated gradient ascent schemes (such as Adam [8]) are typically used nowadays and (2.5) only serves to give an idea about the approach.

The main problem, that needs to be adressed is the objective gradient $\nabla_\theta J_\pi$ evaluation. The following analysis based on [1], [13] serves as an introduction to how this problem is typically solved. The objective optimized in an MDP is

$$J_\pi = \mathbb{E}_\tau \left[R(\tau) \right] = \int_\tau P(\tau|\theta) R(\tau) \,, \tag{2.6}$$

where

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \tag{2.7}$$

is the probability of the trajectory τ conditioned on the policy parameters θ. The gradient w.r.t. θ is then

$$\begin{aligned} \nabla_\theta J_\pi = \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) = \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) = \\ = \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) \,. \end{aligned} \tag{2.8}$$

Now, the gradient of the logarithm can be simplified using (2.7) as

$$\nabla_\theta \log P(\tau|\theta) = \nabla_\theta \log \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \,, \tag{2.9}$$

since the transition and the initial state probabilities do not depend on θ. Substituting this into (2.8) gives us the policy gradient

$$\nabla_\theta J_\pi = \int_\tau P(\tau|\theta) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \,. \tag{2.10}$$

Since computing the integral in (2.10) is intractable, we need to approximate the distribution $P(\tau|\theta)$ by samples

$$P(\tau|\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \delta(\tau - \tau_i) \,, \tag{2.11}$$

where $\delta(\cdot)$ is the Dirac delta distribution and $\{\tau_i\}_{i=1}^N$ are the sampled trajectories generated as $\tau_i \sim P(\tau|\theta)$. Plugging this into (2.10) gives us an estimate for the policy gradient

$$\nabla_\theta J_\pi \approx \hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau_i) \,. \tag{2.12}$$

This is already quite useful result in that it enables us to learn a neural network policy in order to maximize the cummulative reward. All we need to do is to sample a batch of trajectories $\{\tau_i\}_{i=1}^N$, compute the returns $R(\tau_i)$ and use (2.12) to perform gradient ascent with respect to θ. However, there are some practical issues with this approach which will be discussed further.

2.1.3 Policy gradient estimation

One of the main problems with the policy gradient estimate (2.12) lies in its high variance when using the Return (2.3). Lucklily, there is a way to reduce this variance and still keep the estimate unbiased. We can decompose $R(\tau)$ in (2.10) as

$$R(\tau) = Q(s_{t:T}, a_{t:T-1}) - b(s_{0:t}, a_{0:t-1}) \,, \tag{2.13}$$

where $b(s_{0:t}, a_{0:t-1})$ depends on $s_{0:t} = (s_0, ..., s_t)$ and $a_{0:t-1} = (a_0, ..., a_{t-1})$, i.e. states and actions from the trajectory before a_t and $Q(s_{t:T}, a_{t:T-1})$ depends on $s_{t:T} = (s_t, ..., s_T)$ and $a_{t:T-1} = (a_t, ..., a_{T-1})$, i.e. states and actions from the trajectory that come after a_t. Then the policy gradient reads

$$\begin{aligned} &\int_\tau P(\tau|\theta) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) = \\ &= \int_\tau P(\tau|\theta) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) [Q(s_{t:T}, a_{t:T-1}) - b(s_{0:t}, a_{0:t-1})] \end{aligned} \tag{2.14}$$

We can split the integral (2.14) and focus on the second part. The sum can be put in front of the integral and then the sum is over terms

$$\int_\tau P(\tau|\theta) \nabla_\theta \log \pi_\theta(a_t|s_t) b(s_{0:t}, a_{0:t-1}) \,. \tag{2.15}$$

Let us omit θ and label the parts of the trajectory before and after a_t as $\tau^- := (s_{0:t}, a_{0:t-1})$ and $\tau^+ := (s_{t+1:T}, a_{t+1:T-1})$ respectively to simplify notation. Then we get

$$\begin{aligned} &\int_\tau P(\tau) \nabla \log \pi(a_t|s_t) b(s_{0:t}, a_{0:t-1}) = \\ &= \int_{\tau^-} \int_{a_t} \int_{\tau^+} P(\tau^-) \pi(a_t|s_t) P(\tau^+|s_t, a_t) \nabla \log \pi(a_t|s_t) b(\tau^-) = \\ &= \int_{\tau^-} P(\tau^-) b(\tau^-) \int_{a_t} \pi(a_t|s_t) \nabla \log \pi(a_t|s_t) \int_{\tau^+} P(\tau^+|s_t, a_t) = \\ &= \int_{\tau^-} P(\tau^-) b(\tau^-) \int_{a_t} \pi(a_t|s_t) \nabla \log \pi(a_t|s_t) = \\ &= \int_{\tau^-} P(\tau^-) b(\tau^-) \cdot 0 = 0 \,, \end{aligned} \tag{2.16}$$

where we used the facts that the integral of a probability distribution is equal to 1 and

$$\int_{a_t} \pi(a_t|s_t) \nabla \log \pi(a_t|s_t) = \int_{a_t} \nabla \pi(a_t|s_t) = \nabla \int_{a_t} \pi(a_t|s_t) = \nabla 1 = 0\,. \quad (2.17)$$

Therefore $b(s_{0:t}, a_{0:t-1})$ has no effect on the expected value of the policy gradient. Now, let us take a look at the first part of the integral (2.14), while again ignoring the sum for simplicity

$$\begin{aligned}
&\int_{\tau} P(\tau) \nabla \log \pi(a_t|s_t) Q(s_{t:T}, a_{t:T-1}) = \\
&= \int_{\tau^-} \int_{a_t} \int_{\tau^+} P(\tau^-) \pi(a_t|s_t) P(\tau^+|s_t, a_t) \nabla \log \pi(a_t|s_t) Q(s_{t:T}, a_{t:T-1}) = \\
&= \int_{\tau^-} P(\tau^-) \int_{a_t} \nabla \pi(a_t|s_t) \int_{\tau^+} P(\tau^+|s_t, a_t) Q(s_{t:T}, a_{t:T-1})\,.
\end{aligned} \quad (2.18)$$

This gives us an important insight: We can change the return $R(\tau)$ used for policy gradient estimation to $\Psi_t = Q'(s_{t:T}, a_{t:T-1}) - b'(s_{0:t}, a_{0:t-1})$, where b' is arbitrary and Q' must satisfy

$$\int_{\tau^+} P(\tau^+|s_t, a_t) Q'(s_{t:T}, a_{t:T-1}) = \int_{\tau^+} P(\tau^+|s_t, a_t) Q(s_{t:T}, a_{t:T-1})\,. \quad (2.19)$$

Using the return (2.3), we get that

$$Q(s_{t:T}, a_{t:T-1}) = \sum_{t'=t}^{T} r_{t'}\,. \quad (2.20)$$

Some popular choices for Ψ_t then include

1. $\sum_{t=0}^{T} r_t$: total reward of the trajectory
2. $\sum_{t'=t}^{T} r_{t'}$: reward following action a_t
3. $\sum_{t'=t}^{T} r_{t'} - b(s_t)$: baselined version of previous formula
4. $Q^{\pi}(s_t, a_t)$: state-value function
5. $A^{\pi}(s_t, a_t)$: advantage function
6. $r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$: Temporal difference residual

where

$$\begin{aligned}
Q^{\pi}(s_t, a_t) &= \int_{\tau^+} P(\tau^+|s_t, a_t) \sum_{t'=t}^{T} r_{t'}\,, \\
A^{\pi}(s_t, a_t) &= Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)\,, \\
V^{\pi}(s_t) &= \int_{a_t} \pi(a_t|s_t) \int_{\tau^+} P(\tau^+|s_t, a_t) \sum_{t'=t}^{T} r_{t'}\,.
\end{aligned} \quad (2.21)$$

As it turns out, using the advantage function $A^\pi(s_t, a_t)$ yields the lowest variance of the gradient estimate in practice [13]. This then leads to the policy gradient having the form

$$\nabla_\theta J_\pi = \int_\tau P(\tau|\theta) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t, a_t) \,. \tag{2.22}$$

The only problem is, that the true advantage function $A^\pi(s_t, a_t)$ is unknown and needs to be estimated. A commonly used approach to advantage estimation is discussed in section 2.1.4.

2.1.4 Generalized advantage estimation

It is possible to further decrease the variance of the gradient estimate, while introducing bias. Here we assume infinite time horizon for the return computation, i.e.

$$R(\tau) = \sum_{t=0}^{\infty} r_t \,. \tag{2.23}$$

A simple way to decrease the variance is then to use a discounted form of the return (2.23)

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \,, \tag{2.24}$$

where $\gamma \in [0, 1]$ is the discount factor, which downweights reward far into the future. The TD residual

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \,, \tag{2.25}$$

where $V(\cdot)$ is an approximation to the true value function $V^\pi(\cdot)$ (2.21), can be considered an estimate of the advantage of the action a_t for the discounted return case. The term $\gamma V(s_{t+1})$ is the source of bias of this advantage estimate. Consider taking the sum of k of these TD residuals (2.25)

$$\begin{aligned} \hat{A}_t^{(1)} &= \delta_t = -V(s_t) + r_t + \gamma V(s_{t+1}) \\ \hat{A}_t^{(2)} &= \delta_t + \gamma \delta_{t+1} = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\ \hat{A}_t^{(k)} &= \sum_{l=0}^{k-1} \gamma^l \delta_{t+l} = -V(s_t) + \sum_{l=0}^{k-1} \gamma^l r_{t+l} + \gamma^k V(s_{t+k}) \end{aligned} \tag{2.26}$$

which again yields biased estimators of the advantage function, but the bias is getting smaller with larger k as the biasing term $\gamma^k V(s_{t+k})$ is getting increasingly discounted. The sum of the rewards $\sum_{l=0}^{k-1} \gamma^l r_{t+l}$, on the other hand, is a source of large variance of this estimate and is getting more pronounced with larger k. The generalized advantage estimate is defined as an exponentially weighted average of these k-step estimators

$$\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots) = \ldots = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \,. \tag{2.27}$$

For $\lambda = 0$, we get $\hat{A}_t = \delta_t$, which has low variance, but high bias. For $\lambda = 1$, we get $\hat{A}_t = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}$, which has high variance and low bias. The parameter $\lambda \in [0, 1]$ then controls the tradeoff between bias and variance of the advantage estimate.

Value function estimation

We discussed how to estimate the Advantage function using trajectory samples and the approximate value function $V(s_t)$. This approximate value function itself is commonly represented as a neural network $V(s_t) = V_{\theta'}(s_t)$, parametrized by θ'. The simplest approach to training this value function is by minimizing the regression loss

$$L^V(\theta') = \hat{\mathbb{E}}_t \left(V_{\theta'}(s_t) - \hat{V}_t \right)^2 , \tag{2.28}$$

where $\hat{V}_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l}$ is the discounted sum of rewards collected on a trajectory from state s_t onward. $\hat{\mathbb{E}}_t$ denotes the empirical average taken over a trajectory batch $\{\tau_i\}_{i=1}^N$. This regression loss can again be minimized via gradient descent performed on the network parameters θ'. It is common to have the value network $V_{\theta'}(s_t)$ share some parameters with the policy $\pi_\theta(a_t|s_t)$.

2.1.5 Proximal policy optimization

Using the policy gradient as is can lead to unstable behaviour during training. One of the approaches to combat this behaviour is called Proximal Policy Optimization[14] (PPO). The estimate of the policy gradient (2.22) using advantage estimates $A_\pi(s_t, a_t) \approx \hat{A}_t$ is

$$\hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t , \tag{2.29}$$

which is proportional to the gradient of the objective

$$L^{\mathrm{PG}}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right] , \tag{2.30}$$

where $\hat{\mathbb{E}}_t$ designates the empirical average over a trajectory batch $\{\tau_i\}_{i=1}^N$. The gradient estimate (2.29) can equivalently be obtained as the gradient of

$$L'(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{2.31}$$

taken at $\theta = \theta_{\mathrm{old}}$, which are the parameters of the policy used for the trajectory sampling. The PPO approach replaces this objective with the surrogate objective

$$L^{\mathrm{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\hat{A}_t,\, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] , \tag{2.32}$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

and $\epsilon > 0$ is a hyperparameter. If for a given sample, the advantage is positive, the per-sample objective reduces to

$$\min\left(r_t(\theta),\ (1+\epsilon)\right)\hat{A}_t \tag{2.33}$$

which will have a zero gradient whenever $r_t(\theta) \geq (1+\epsilon)$. On the other hand, if the advantage is negative, the per-sample objective reduces to

$$\max\left(r_t(\theta),\ (1-\epsilon)\right)\hat{A}_t\,, \tag{2.34}$$

which will have a zero gradient for $r_t(\theta) \leq (1-\epsilon)$. This means that the objective is allowed to improve only to some extend given by the hyperparameter ϵ. This adjustment to the objective also allows us to perform multiple gradient ascent steps using the same trajectory samples and thus improve the sample efficiency. A second approach mentioned in the PPO paper is to add a term penalizing the Kullback–Leibler (KL) divergence between the old and the new policy

$$L^{KL} = D_{KL}(\pi_{\theta_{old}}(a_t|s_t) \| \pi_\theta(a_t|s_t)) \tag{2.35}$$

to the original objective (2.31). The KL divergence between probability distributions $p(x)$ and $q(x)$ is defined as

$$D_{KL}(p\|q) = \int_x p(x)\log\left(\frac{p(x)}{q(x)}\right)\,. \tag{2.36}$$

The first approach is however used more commonly. The objectives (2.32), (2.28) can be combined in a single objective

$$L^{PPO}(\theta,\theta') = c_v \cdot L^V(\theta') - L^{\text{CLIP}}(\theta)\,, \tag{2.37}$$

where c_v is the value loss coefficient. The objective (2.37) can than be minimized using the procedure 1 (adapted from [1])

Algorithm 1 PPO

1: Input: initial policy parameters θ_0, initial value function parameters θ'_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}_{i=1}^N$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{V}_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l}$.
5: Estimate advantages, $\hat{A}_t$ based on the current value function $V_{\theta'_k}$.
6: Update the policy and value weights by minimizing the PPO loss:

$$\theta_{k+1}, \theta'_{k+1} = \arg\max_{\theta,\theta'} L^{PPO}(\theta,\theta'),$$

typically via stochastic gradient ascent with Adam.
7: **end for**

2.2 Policy architecture

The policy gradient methods 2.1.2 aim at optimizing a stochastic policy $\pi_\theta(a_t|s_t)$ directly using on-policy trajectories (sampled using the policy itself). This procedure relies on two features of the policy: the policy mapping being stochastic and the ability to compute the gradient $\nabla_\theta \pi_\theta(a_t|s_t)$ for any sampled action a_t. A neural network is designed to be a differentiable general function approximator and therefore the ability to compute the gradient is not a problem. However, a neural network is also designed to be deterministic and cannot be used to represent a probability distribution directly. Therefore something needs to be built on top of it, in order to constuct the stochastic policy. In particular, we need some extra computational module, that turns the network outputs into a probability distribution while preserving the differentiability.

For a discrete action space $A = \{a^1, ..., a^m\}$, we can simply construct a neural network mapping to a vector of matching size $n_\theta : S \to \mathbb{R}^m$ and use a softmax layer[1] on top of it. We can then interpret the softmax output as a probability distribution over the discrete action space. The gradient is then straightforward to obtain. For a continuous action space e.g. $A = \mathbb{R}$, we can discretize it by picking some representative actions and use the approach for discrete action spaces. However, it may be difficult to choose the discretization and it may lead to suboptimal results. A different approach is to construct a neural network that maps to the a set of parameters $\kappa = n_\theta(s_t)$ of a parametric probability distribution $p_\kappa(a_t)$ over the action space. We only need to be able to compute the gradient of a given sample with respect to the parametrization. The normal distribution $\mathcal{N}(\mu, \sigma)$ parametrized by $\kappa = [\mu, \sigma]$ can be used as an example. We can first obtain a sample a'_t from a standard normal distribution $\mathcal{N}(0, 1)$ (zero mean, unit variance) and then translate and scale it using the network outputs $[\mu(\theta), \sigma(\theta)]$ i.e.

$$\begin{aligned} a_t &= \mu(\theta) + \sigma(\theta) a'_t \,, \\ a'_t &\sim \mathcal{N}(0, 1) \,. \end{aligned} \tag{2.38}$$

The sample action a_t obtained like this is then differentiable with respect to the parameters $\kappa = [\mu(\theta), \sigma(\theta)]$ and therefore also with respect to the network parameters θ. Another distribution better suited for bounded action spaces is the $Beta(\alpha, \beta)$ distribution with probability density function (PDF)

$$p(x) = c \cdot x^{\alpha-1}(1-x)^{\beta-1} \qquad \text{for } x \in [0, 1] \,, \tag{2.39}$$

where

$$c = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx$$

is the normalization constant and $\kappa = [\alpha, \beta]$, where $\alpha, \beta > 0$ are the distribution parameters. The shape of this PDF is shown on figure 2.2 for

[1] $\text{softmax(x)}_i = \dfrac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ $i = 1, ..., n$ for the input $x \in \mathbb{R}^n$.

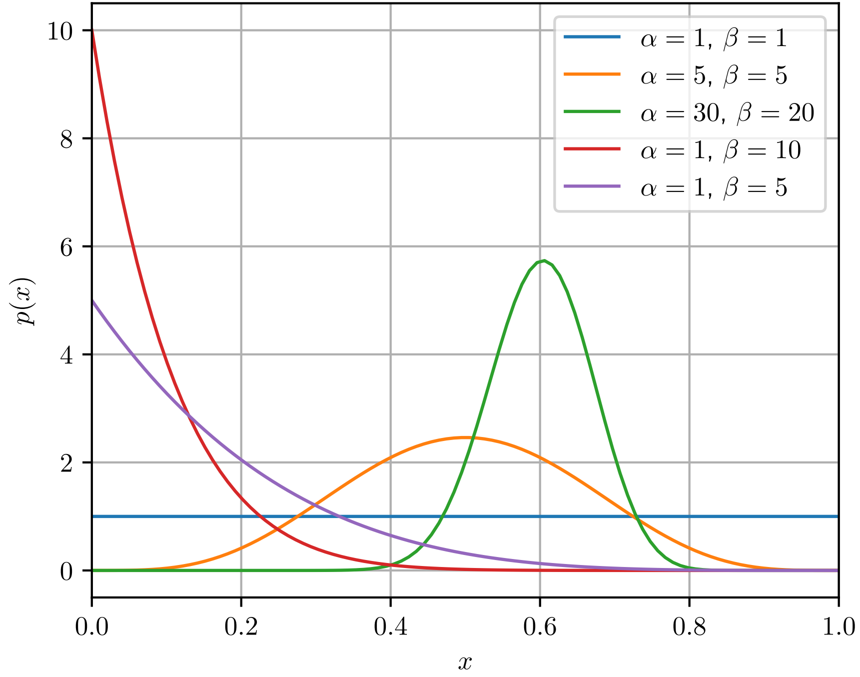


Figure 2.2: $Beta(\alpha, \beta)$ distribution probability density function

different sets of parameters α, β. This has the advantage of being defined on an interval $[0, 1]$ in contrast to the normal distribution, which has infinite support. How to perform the differentiable sampling with *Beta* distribution is however beyond the scope of this thesis.

2.2.1 Multi-layer perceptron

The simplest kind of neural network is a multilayer perceptron (MLP), which is made up by k fully connected layers $\{l_i\}_{i=1}^{k}$. Each fully connected layer performs a computation

$$l_i(x) = f(W_i x + b_i)\,, \tag{2.40}$$

where $x \in \mathbb{R}^n$ is the layer input vector, $W_i \in \mathbb{R}^{m \times n}$ is a weight matrix, $b_i \in \mathbb{R}^m$ is a bias vector and $f(\cdot)$ is a non-linear activation function. A common choice for the non-linearity is $f(\cdot) = \tanh(\cdot)$ when dealing with RL policies. The output of the network is then obtained by chaining the linear layers together

$$n_\theta(x) = (l_k \circ l_{k-1} \circ \ldots \circ l_2 \circ l_1)(x)\,, \tag{2.41}$$

where $\theta = \{W_1, b_1, \ldots, W_k, b_k\}$ denotes weights and biases of the individual layers, i.e. the learnable parameters of the network.

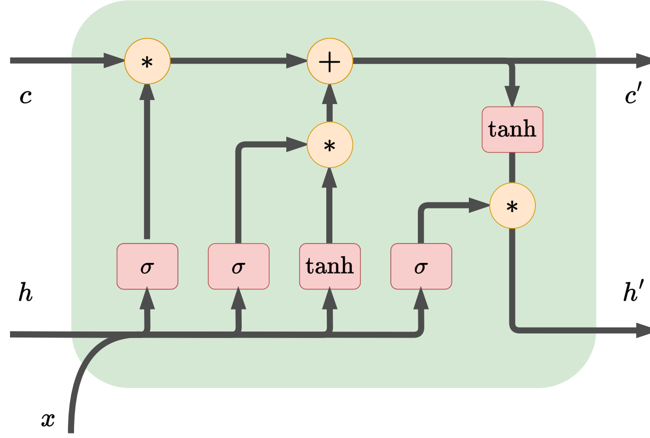


Figure 2.3: LSTM computation graph

2.2.2 Long short-term memory

A recurrent neural network (RNN) is a special type of a neural network designed to work with sequential data, i.e. the input to the network is a sequence of vectors $(x_t)_{t=0}^{T}$ indexed by time. The network than processes the whole sequence step by step while also retaining an internal state, which acts as a memory. The long short-term memory (LSTM) architecture is a type of an RNN that suppresses the vanishing gradient problem when compared to a naive implementation. It can be described by the set of equations

$$\begin{aligned} i &= \sigma(W_{ii}x + W_{hi}h + b_i) \\ f &= \sigma(W_{if}x + W_{hf}h + b_f) \\ g &= \tanh(W_{ig}x + W_{hg}h + b_g) \\ o &= \sigma(W_{io}x + W_{ho}h + b_o) \\ c' &= f * c + i * g \\ h' &= o * \tanh(c') \end{aligned} \tag{2.42}$$

where x is the current input, c and h are the cell state and hidden state at the current timestep and c' and h' are the cell state and hidden state at the next timestep. The $\sigma(\cdot)$ denotes the sigmoid[2] function and $*$ denotes the Hadamard (element-wise) product. The computational graph of the LSTM cell is depicted in figure 2.3. To put it simply, the network is presented the current input x as well as previous cell state c and hidden state h at each timestep. These are then processed by a set of fully connected layers and combined together to produce the new cell state c' as well as the new hidden state h', which is used as the output of the cell.

The use of an RNN is justified when dealing with a partially observable MDP, where the true state s_t of the system is not observable. The RNN can then be used to either estimate the true current state s_t from a sequence

[2]The sigmoid function is defined as $\sigma(x) = \dfrac{e^x}{1+e^x}$.

of past observations $o_{t-T:t}$, or to map the observations to the distribution parameters directly.

2.2.3 Convolutional network

Another neural network architecture that is commonly used with sequential data is a convolutional neural network (CNN). This architecture makes use of a one dimensional convolutional layer, that based on an input sequence $(x_i)_{i=1}^{I}$ of vectors $x_i \in \mathbb{R}^n$ outputs another sequence $(y_j)_{j=1}^{J}$ of vectors $y_j \in \mathbb{R}^m$, where

$$y_j = f(b + \sum_{k=0}^{K} W_k x_{j \cdot s+k}) \,. \tag{2.43}$$

The $W = [W_0, ..., W_K]$, where $W_k \in \mathbb{R}^{m \times n}$ are the weights of the layer and $b \in \mathbb{R}^m$ is the bias. The $f(\cdot)$ denotes again a non-linear activation function. The output sequence lengh is determined by the input sequence length, the receptive field K and the stride s as $J = \frac{I-K}{s} + 1$. The convolutional layer can be thought of as a fully connected layer with sparse and periodic weight structure.

2.3 Drone model

A simple quadcopter model consists of a rigid body connected to four motors with propellers positioned typically in the "x" configuration as depicted on figure 2.4. Let the coordinate system of the rigid body be denoted as $\mathcal{B}$ and the world coordinate system as $\mathcal{W}$. The four propellers are oriented such that their rotation axes are parallel to the drone's b_z axis, which is also the direction of the forces they exert. The dynamics of the motors can be approximated by a first order system

$$\frac{d}{dt}\omega_m = \frac{\omega_d - \omega_m}{\tau} \,, \tag{2.44}$$

where ω_m $[\mathrm{rad\,s^{-1}}]$ is the motor angular velocity, ω_d $[\mathrm{rad\,s^{-1}}]$ is the desired angular velocity and τ [s] is the motor time constant. The magnitude of the force exerted by a propeller (thrust) F_m [N] can be approximated as

$$F_m \approx k \cdot \omega_m^2 \,, \tag{2.45}$$

where k $[\mathrm{N\,s^2\,rad^{-2}}]$ is the thrust force coefficient. Along with the thrust, the motor also exerts a torque T_m [N m] on the system modelled as

$$T_m \approx c \cdot F_m \,, \tag{2.46}$$

where c [m] is the torque proportionality constant. Note that the direction of the torque T_m is always opposite to the direction of rotation of the given motor. The motor rotation directions are oriented in an alternating fashion

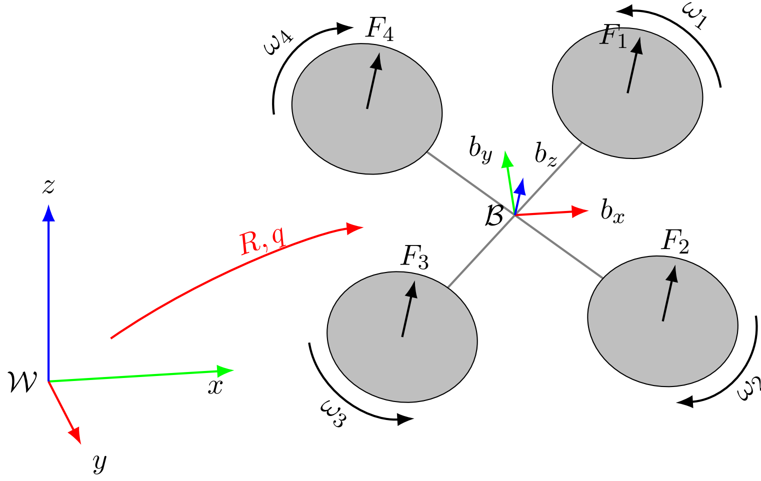


Figure 2.4: Ilustration of the quadcopter model

as depicted in figure 2.4. This gives rise to the following forces being exerted on the body frame

$$\begin{bmatrix} F \\ T_x \\ T_y \\ T_z \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ l_d/\sqrt{2} & -l_d/\sqrt{2} & -l_d/\sqrt{2} & l_d/\sqrt{2} \\ -l_d/\sqrt{2} & -l_d/\sqrt{2} & l_d/\sqrt{2} & l_d/\sqrt{2} \\ -c & c & -c & c \end{bmatrix}}_{A} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix}, \tag{2.47}$$

where l_d [m] is the length of the drone arms, F [N] is the total thrust, T_x, T_y, T_z [N m] are the torques along the body's b_x, b_y and b_z axes respectively and F_1, F_2, F_3, F_4 [N] are the motor thrusts. The matrix A is reffered to as the mixing matrix, or allocation matrix and maps the motor thrusts to the forces and torques exerted on the body frame $\mathcal{B}$.

The rotational dynamics can then be obtained using the Euler's equation of motion

$$T = J\dot{\omega} + \omega \times J\omega\,, \tag{2.48}$$

where $T = [T_x, T_y, T_z]^\top$ is the extrinsic torque vector exerted by the propellers, $J \in \mathbb{R}^{3\times3}$ [kg m^2] is the tensor of inertia of the drone body, $\omega \in \mathbb{R}^3$ [rad s^{-1}] is the intrinsic angular velocity vector of the drone and $\dot{\omega} = \frac{d\omega}{dt}$ is the intrinsic angular acceleration of the drone. The orientation of a drone with respect to the world frame $\mathcal{W}$ can be described by a rotation matrix $R \in SO(3)$[3]. This rotation matrix evolves as

$$\dot{R} = R[\omega]_\times\,, \tag{2.49}$$

where $[\omega]_\times$ is a skew symmetric matrix[4]. The translational dynamics are

[3] A matrix $R \in \mathbb{R}^{3\times3} \in SO(3)$ if it satisfies $R^\top R = I$ and $\det(R) = 1$.

[4] The matrix $[a]_\times$ constructed from the vector $a \in \mathbb{R}^3$, such that $[a]_\times b = a \times b$ for any vector $b \in \mathbb{R}^3$.

obtained from the Newton's second law of motion as

$$\ddot{q}^{\mathcal{W}} = \frac{1}{m_d} R \begin{bmatrix} 0 \\ 0 \\ F \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}, \tag{2.50}$$

where $q^{\mathcal{W}} = [x, y, z]^\top$ [m] is the position vector of the drone in the world frame, m_d [kg] is the mass of the drone, F [N] is the total thrust of the motors from (2.47) and g [m s^{-2}] is the gravitational acceleration.

This approximate model of the quadcopter is fully described by the following set of parameters

$$p = \{\tau, k, c, l_d, J, m_d, g\}, \tag{2.51}$$

the following dynamic state variables

$$s = \{q^{\mathcal{W}}, \dot{q}^{\mathcal{W}}, R, \omega, \omega_1, \omega_2, \omega_3, \omega_4\} \tag{2.52}$$

and the controllable inputs of this model are the four desired motor angular velocities

$$a = \{\omega_{d1}, \omega_{d2}, \omega_{d3}, \omega_{d4}\}. \tag{2.53}$$

Note that the use of the rotation matrix R in the state formulation (2.52) does not yield a minimal representation of the drone state. This is because a rotation matrix R has nine entries, whereas any rotation in 3D space has only 3 degrees of freedom (Euler's rotation theorem). Alternative rotation representations include the axis-angle, Euler angles (or Tait–Bryan angles) or a unit quaternion representation, each of which has its advantages and disadvantages.

2.3.1 Suspended load

The suspended load (pendulum) below the drone can be modelled as a rigid rod of length l_p [m] with a weight of mass m_p [kg] at the end of it. The rod is attached to the drone at the origin of its coordinate frame $\mathcal{B}$ via a rotational joint with 2 degrees of freedom. The addition of this suspended load leads to a quite elaborate analysis [6], which is beyond the scope of this work. What is important, is that it introduces two additional parameters l_p and m_p to the dynamical model as well as two extra coordinates $\psi' \in [-\frac{\pi}{2}, \frac{\pi}{2}]^2$ that describe the orientation of the frame $\mathcal{P} = \{p_x, p_y, p_z\}$ rigidly connected to the pendulum weight as shown in figure 2.5. The rotational joint can be though of as two succesive joints, each rotating about a single axis. The first joint connected to the drone frame rotates about the drone's b_x axis and the second joint rotates about rotated b_y axis. To put it clearly, given the two joint angles $\psi' = [\psi'_1, \psi'_2]$ of the first and second rotational joint respectively, the rotation matrix describing orientation of the pendulum frame $\mathcal{P}$ can be written as

$$R_p = R_x(\psi'_1) R_y(\psi'_2), \tag{2.54}$$

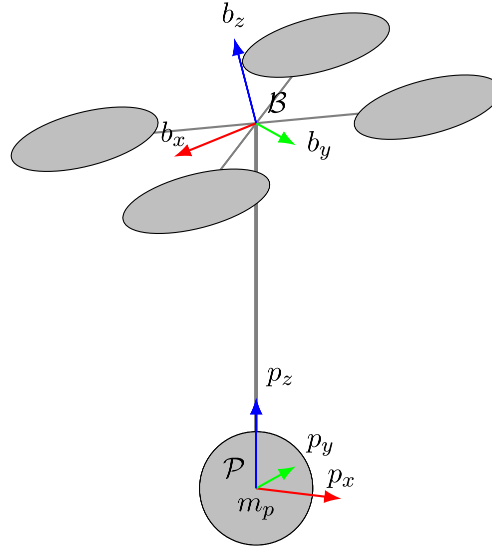


Figure 2.5: Ilustration of the quadcopter model with hanging load

where $R_x(\cdot)$, $R_y(\cdot)$ are defined as in (2.59). The end of the pendulum in the drone's frame can be obtained as

$$q_p^{\mathcal{B}} = R_p \begin{bmatrix} 0 \\ 0 \\ -l_p \end{bmatrix} . \tag{2.55}$$

The parameter set then becomes

$$p = \{\tau, k, c, l_d, J, m_d, g, l_p, m_p\} , \tag{2.56}$$

and the state vector becomes

$$s = \{q^{\mathcal{W}}, \dot{q}^{\mathcal{W}}, R, \omega, \psi', \dot{\psi}', \omega_1, \omega_2, \omega_3, \omega_4\} , \tag{2.57}$$

where ψ' are the pendulum angles and $\dot{\psi}' = \frac{d\psi'}{dt}$ is the rate of change of those angles. This introduces a problem in the form of two extra degrees of freedom we have to account for in control design and no extra actuators. Also, it is often the case, that the pendulum state cannot be directly measured and state estimation of the pendulum angles ψ' and angular velocity $\dot{\psi}'$ is needed.

2.3.2 Roll, pitch and yaw

There are numerous references to so called roll, pitch and yaw angles throughout this thesis. These are a set of Tait–Bryan angles describing the orientation of a rigid body in 3D space. The precise definition of what these three angles represent is however not unanimous in the literature. This is a common source of misunderstanging and one of the disadvantages of this representation. This is why it is necessary to state how the angles are used to construct some unambiguous rotation representation such as a rotation matrix. Given the three

angles α, β, γ (roll, pitch and yaw respectively) describing the orientation of a rigid body $\mathcal{B}$ in the world frame $\mathcal{W}$, the rotation matrix is constructed as

$$R = R_z(\gamma) R_y(\beta) R_x(\alpha)\,, \tag{2.58}$$

where

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix},$$

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}, \tag{2.59}$$

i.e. we first rotate about the world x axis by the roll angle α, then we rotate about the world y axis by the angle pitch angle β and finally about the z world axis by the yaw angle γ. Given that the body frame $\mathcal{B}$ is centered at the origin of the world frame $\mathcal{W}$, the transformation of a vector q from $\mathcal{B}$ to $\mathcal{W}$ is then

$$q^{\mathcal{W}} = R q^{\mathcal{B}}\,. \tag{2.60}$$

2.4 Drone control framed as an MDP

When dealing with controlling a quadcopter as modelled in section 2.3, we typically use a digital computer, which operates in discrete timesteps. Therefore we can consider a discretized dynamical model of the drone instead of the continuous time one. This leads to the drone model having the form of a general dicrete non-linear system

$$s_{t+1} = f(s_t, a_t)\,, \tag{2.61}$$

where $s_t \in S$ is the state of the drone at timestep t and $a_t \in A$ is the action at timestep t. The f function here symbolically represents the discretized version of the continuous dynamical model discussed in section 2.3. The discretized dynamics can be obtained by integrating the continuous dynamical model in time between timesteps t and $t+1$. Instead of doing this analytically, which may often be intractable, this task is left to a physics simulation software, which integrates the continuous dynamics through numerical methods. Since the dynamical model assumed is deterministic, the transition function is formally

$$P(s_{t+1} | s_t, a_t) = \delta(s_{t+1} - f(s_t, a_t))\,, \tag{2.62}$$

where $\delta(\cdot)$ is the Dirac delta function. Now all that is needed for the complete MDP formulation is the reward function

$$r(s_{t-1}, s_t, a_t)\,. \tag{2.63}$$

The form of the reward function depends on the desired behaviour of the system and is discussed in the experimental section of this thesis 4.1.

Chapter 3

Implementation

3.1 Physics simulator

In order to optimize the policy using the PPO algorithm (1), it is impractical to collect the trajectory samples with a real quadcopter. Besides crashing the quadcopter whenever the randomly initialized policy makes a bad choice, the speed at which state-action pairs can be sampled is severely limited. Instead, we can use a physics simulator to compute the trajectories of a model of the real system. The simulator used for RL training in this thesis is based on the MuJoCo [16] (Multi-Joint dynamics with Contact) library, which is a C/C++ library that is capable of fast and accurate physical model simulation as well as scene rendering. It provides a Python API, which enables seamless usage with commonly used deep learning frameworks such as PyTorch[12]. The use of a simulated model unfortunately introduces a so called Sim-to-Real gap, i.e. a discrepancy between the model we use for training and the real world system, that we want to control. This gap can be mitigated by using a more accurate model for simulation, or training the policy to be robust to model mismatch. Both of these directions need to be considered in the simulated environment development.

3.1.1 MJCF

In order to define the simulated model for MuJoCo, a user can use the MJCF scene description language, which is derived from the XML format. As such, it is human-readable and thus quite easy to understand and use. This has no impact on performance though, since the MJCF is compiled into more efficient data structures prior to the actual simulation. The basic building blocks of a MJCF model are bodies, geoms, joints and actuators. Bodies correspond to actual physical bodies used in the simulation. Each body consists of possibly multiple geoms, which give them their shape and mass used for the body dynamics and collision computation. Bodies can be attached via joints to constrain their relative movement, e.g. with a hinge joint, which enables only rotational motion around its predefined axis. The actuators are then used to impose some forces, velocities or even positions onto parts of the bodies. The MJCF model is also used to define lighting for visualization and various

additional physical simulation parameters such as air density, joint damping and more.

3.1.2 Environment model

In order to train a policy that is robust to model mismatch, it is desirable to generate not one single simulation model, but multiple with small perturbations in the model parameters. Luckily, the authors of MuJoCo also provide the dm_control [17] library, which makes this task a breeze. The mjcf module from dm_control provides an object oriented programming interface for MJCF model generation, which enables conveniently constructing the simulated model programatically in Python.

The MJCF model is generated with possibly multiple quadcopters inside a single environment. This allows collecting many trajectories in parallel inside a single simulation instance, increasing the efficiency of the system. The MJCF model object is then converted to an XML file for use with the MuJoCo simulator. The MJCF model generation can be configured using parameters such as the frequency of the underlying physical simulation, or a list of the drone model parameters.

Simulated drone model

Every simulated quadcopter model is parametrized by a set of 6 parameters p^i, which consists of the following values

$$\begin{aligned} &m_d - \text{Total mass of the drone [kg]} \\ &l_d - \text{Length of the drone arms [m]} \\ &m_p - \text{Mass of the pendulum weight [kg]} \\ &l_p - \text{Length of the pendulum [m]} \\ &F_m - \text{Maximal force exerted by a propeller [N]} \\ &\tau - \text{Time constant of the drone motors [s]} \end{aligned} \tag{3.1}$$

This set of parameters (3.1) is a little different to the parameter set discussed in (2.56). First of all, F_m replaced the thrust proportionality constant k and torque proportionality constant c is assumed to be fixed for simplicity. The gravitational constant g is assumed to be non-chaning, and the tensor of inertia J is computed by MuJoCo from the underlying drone geometry, which can be affected only by the arm length l_d. This is why g and J are ommited from the configurable parameters.

The parameters can be different for each drone inside the simulation to allow training robust, or adaptive policies. Each drone inside the simulation is assembled according to the parameter set using primitive shapes that capture the general structure of a typical quadcopter as discussed in section 2.3. This includes a main body, four arms, motors and propellers. The total drone mass m_d is distributed across these body parts. Besides the geometry, there are also four actuators aligned with the four propellers, which can be controlled to exert forces on the drone body. A pendulum consisting of a rod of length

l_p and a cube-shaped weight of mass m_p is then connected to the main body from below via two 1 DOF rotational joints rotated 90 degrees along the b_z axis relative to each other. The first one, connected to the drone's frame rotates along the b_x axis and the second one rotates about the rotated b_y axis.

A screenshot of the drone model inside the simulated environment can be seen in figure 3.1. The figure 3.2 depicts the simulation environment with multiple drones used for training. The mutual collisions between the drones have been disabled and therefore each drone model is simulated independently of the others. The white ball in the figure 3.2 is a visualization of the reference state and the red, green and blue cylinders sticking out of it correspond to the orientation of its x, y, z axes respectively.

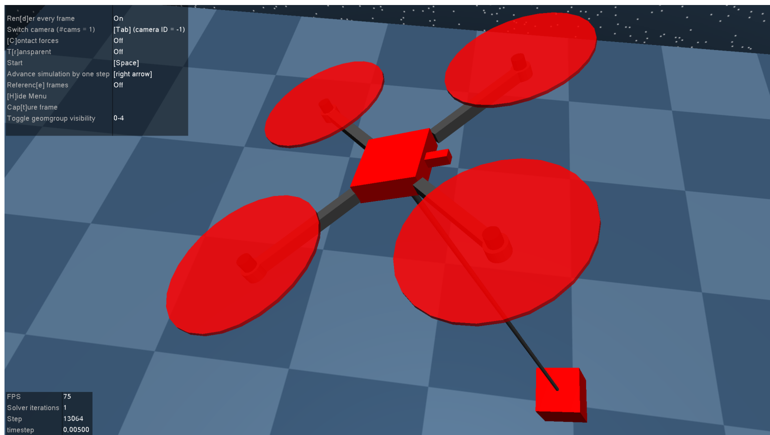


Figure 3.1: Drone model assembled from primitive shapes

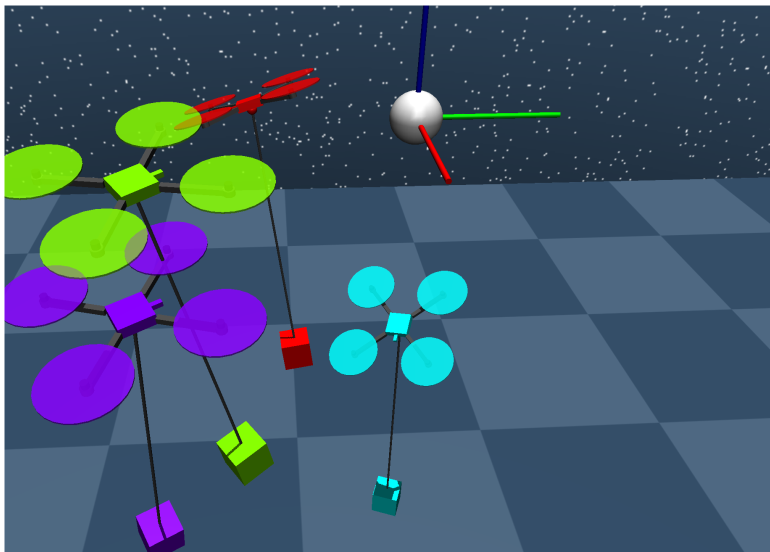


Figure 3.2: Mutltiple drones inside a single simulation instance

Actuator model

The MuJoCo simulator does not support the drone propeller model mentioned in (2.3) out of the box. Therefore a simpler motor model is used, ignoring the quadratic dependence of the torque and thrust produced on the angular speed of the motor and replacing it by a linear relationship. The first order system response will be preserved to model the delayed motor response. The implemented actuator model is then

$$
\begin{aligned}
\frac{d}{dt}act(t) &= \frac{ctrl - act(t)}{\tau} \\
F(t) &= F_m \cdot act(t) \\
T(t) &= ori \cdot \frac{F_m}{100} \cdot act(t)
\end{aligned}
\tag{3.2}
$$

where $ctrl$ is the control signal, $act(t)$ is the actuation state, and $F(t)$ and $T(t)$ are the force and torque exterted by the actuator respectively. The $ori \in \{-1, 1\}$ denotes the propeller rotation orientation and only affects the orientation of the torque exterted by the actuator. Note, that $acc(t)$, $F(t)$ and $T(t)$ are functions of a continuous time variable denoted by t. This is because although we control and observe the simulation in discrete timesteps, the underlying dynamical model MuJoCo uses for simulation considers time to be continuous. This allows the simulated states to evolve continuously between the dicrete observation and control steps.

3.2 Training framework

RLlib [10] is an open-source library for reinforcement learning providing implementations of various RL algorithms. The use of such library allows us to focus less on the implementation of the underlying training algorithms and more on the simulation environment, rewards, policy model and hyper-parameter tuning specific to our problem. The main advantage of using RLlib over other available frameworks is the support for multiprocessing, vectorized environments and recurrent neural networks. The RLlib provides a versatile API that allows the user to work at arbitrary level of abstraction – from specifying only the environment and leaving the rest to RLlib to tuning every part of the training pipeline. RLlib supports two popular deep learning frameworks PyTorch and Tensorflow, that can be used to define the structure of the trained policy.

3.2.1 VecEnv

To utilize the Rllib's vectorized environment capabilities, an environment class has to inherit from the VecEnv class, which tells Rllib, that there are potentially many simulated agents in the environment. The user also has to specify the number of agent in the simulation and the action and observation spaces used for interaction. Then the vector_step() and vector_reset() functions

that allow Rllib to interact with the environment have to be implemented. This differs a bit from the standard gym interface that is commonly used in RL, but the idea is the same. The function vector_step() takes in a list of actions for all the agents in the simulation, performs a simulation step on all of them and returns the observations, rewards and terminations information. The vector_reset() functions is used to reset the agents to initial states. The use of this vectorized environment enables using a single MuJoCo simulation instance with multiple simulated agents and thus sample multiple trajectories in parallel with high efficiency.

3.2.2 PPO training

To train a policy on the environment, the PPOConfig class is first instantiated. This class holds all the relevant configuration parameters used during training. The most important parameters are shown in table 3.1. The environment class env passed to PPOConfig has to implement the standard gym interface, or the VecEnv interface discussed previously.

Parameter	Description
lambda	λ parameter used for the generalized advantage estimation (2.27)
gamma	γ parameter used for the generalized advantage estimation (2.27)
lr	learning rate used for gradient ascent
clip_param	ϵ used in the PPO objective (2.32)
vf_loss_coeff	c_v used in the PPO objective (2.37)
train_batch_size	number of agent experience timesteps used in learning epoch
sgd_minibatch_size	batch size used for gradient ascent
num_sgd_iter	number of gradient ascent iterations per epoch
rollout_fragment_length	collected experience timestep count
num_rollout_workers	number of environment instances used during training
env	the environment class
env_config	environment configuration
model	model configuration dictionary

Table 3.1: PPO training parameters

Training the policy

After specifying the relevant parameters through the PPOConfig, the PPO training can be started. In the process, the policy and environments are instantiated and made ready for the training. The policy is constructed based on the model parameter of the PPOConfig (viz. 3.2.3). A total of num_rollout_workers environments are used, where each one uses separate operating system process. This enables collecting trajectories from all the

environments in parallel, increasing training speed when compared to single process training. Each environment can be configured the env_config, which is passes to the environment constructor.

Each training step performs the training procedure described in 1. First, rollout_fragment_length timesteps are sampled from each agent in the environments. This is repeated until at least train_batch_size timesteps have been collected in total. Then the trajectories are split into training batches, where each one is train_batch_size timesteps long. Each training batch is then used for gradient descent over minibatches of size sgd_minibatch_size. The gradient descent over the whole training batch is repeated num_sgd_iter times. The loss is minimized with steps of stochastic gradient descent (SGD) using the PyTorch's Adam[8] implementation. The only differences to the discussed PPO algorithm 1 is the possible splitting into training batches. RLlib also adds an extra term to the PPO loss, which penalizes the KL divergence (2.35) between the old and updated policy.

3.2.3 Policy configuration

The model configuration dictionary passed to the PPOConfig defines the policy used in training. One can use the predefined models provided by RLlib and use the model configuration to specify the type, depth, width, etc. of the neural network used. To gain full control of the used policy, however, it is desirable to construct a custom neural network model from scratch. Note that the neural network and the action distribution are separate objects in RLlib. The custom_model_config then enables passing extra arguments to the model constructor for further customizability.

Custom neural network

The custom models can be defined using the PyTorch framework. The constructor of the model class is given the observation and action space specification, the expected number of outputs and a custom_model_config dictionary. The constructor is supposed to initialize all the layers of the network and their weights as usual with PyTorch framework. Apart from that, the constructor can also configure its view_requirements attribute, which serves to inform RLlib about the information the model needs for inference. The default view_requirements attribute is set to only include the last observation seen by the agent. But this can be configured to also include e.g. the previous action used by the agent, or a history of previous observations and actions in the given episode.

The forward function is supposed to take in the batched data and compute the distribution parameters that are then used to sample the actions. If using the GAE (viz. section 2.1.4), the value function has to also be implemented to provide the value estimates. It is also possible to define a custom loss function, which can be used to add for example some supervised loss or weight decay to the objective.

Custom action distribution

As mentioned in 2.2, the policy consists of the neural network and the action probability distribution that is parametrized by the network outputs. In the RLlib framework, the probability distribution is set by default be the normal distribution parametrized by the mean and variance. The normal distribution has however infinite support, which means we have to clip the sampled actions when using a bounded action space. This can potentially lead to problems with exploration of the actions space. A solution would be to use a truncated normal distribution constrained to the bounded action space, but this is currently not implemented in RLlib, or PyTorch. Another solution is to use the Beta distribution (viz. (2.39)), which has a bounded support by definition and is also already implemented in PyTorch.

3.3 BaseDroneEnv

The main interface for working with the MuJoCo simulation is the BaseDroneEnv class, which provides a gym-like interface to the environment discussed in section 3.1.2. This class is based on the Gymnasium [5] library's MujocoEnv class, which had to be modified to work with the RLlib's VecEnv 3.2.1 interface. It handles the environment creation, randomized drone parameter generation, setting initial drone states and agent-environment interaction. The environment can be configured using parameters shown in table 3.2.

Parameter	Description
num_drones	number of simulated drones
frequency	frequency of the underlying physical simulation
skip_steps	number of simulation steps between environment interactions
reference	the reference state given by 3D position and heading
start_pos	the starting state of the drones given by a 3D position and heading
max_distance	maximum distance between the drone and the reference before the episode is truncated
max_steps	the maximum number of timesteps in an episode before truncation
state_difficulty	used for scaling size of the initial state distribution
param_difficulty	used for scaling size of the drone parameter distributions
regen_env_at_steps	number of timesteps before the drone parameters are regenerated

Table 3.2: BaseDroneEnv configuration parameters

The goal that drives the development of this environment is to learn a neural

network policy that can control a quadcopter, such that it reaches some static setpoint state. While a neural network is a general function approximator, its expressivity depends on the number of learnable parameters. With more parameters, however, more data as well as time is required for training. It is therefore key to constrain the state space used for training in order to get reasonable results. This is why the initial position is sampled from a small region around the starting position given by start_pos (viz. section 3.3.2). Additionally, the episode is truncated if the drone gets too far away from the reference position, which is controlled by the max_distance parameter. Furthermore, as the drone gets closer to the static setpoint position, the trajectories start to look more alike and carry less information for the policy to improve. If we were to use uninterrupted episodes, these uninformative trajectories would dominate in the dataset used for the policy training. In order to avoid this problem, the episodes are truncated after max_steps steps. The maximum number of steps in an episode should be set such that the drone has enough time to reach the setpoint from any initial position and stabilize reasonably well.

3.3.1 Drone parameter generation

During simulation initialization, every drone parameter $p_j^i \in p^i$, $j = 1, 2, ..., 6$ in the set (3.1) is drawn from a uniform distribution[1]

$$p_j^i \sim \mathcal{U}_{[c_j - w_j, c_j + w_j]}\,, \tag{3.3}$$

where c_j and w_j are the center and (half-) width of the uniform distribution respectively. The c_j and w_j are set by default to the values shown in table 3.3. The width parameters are additionally scaled be the param_difficulty value, to enable easy configuration of the drone parameter ranges.

Parameter	Center c_j	Width w_j	Units
m_d	1	0.1	[kg]
l_d	0.17	0.02	[m]
m_p	0.3	0.05	[kg]
l_p	1.2	0.2	[m]
F_m	7	1	[N]
τ	0.01	0.0025	[s]

Table 3.3: Drone parameter distribution settings

This is repeated for every drone inside the simulation. The generated parameters $\{p^1, ..., p^n\}$ are then used to generate the simulated environment model with n simulated quadcopters. It is also possible to change these drone parameters during training, if enabled. In such case, the MuJoCo simulation

[1] The uniform distribution $\mathcal{U}_{[a,b]}$ has probability density $p(x) = \begin{cases} \frac{1}{b-a} & x \in [a,b] \\ 0 & \text{otherwise} \end{cases}$

is shut down and reinitialized with new drone parameters, while copying the final states from the previous simulation instance into the new one.

3.3.2 Initial state distribution

At start of each episode, the quadcopter state is initialized randomly. The complete state of a drone is represented as

$$s = \{q, \dot{q}, \psi, \omega, \psi', \dot{\psi}', act\}\,, \tag{3.4}$$

where $q \in \mathbb{R}^3$ is the position vector, $\dot{q} = \frac{dq}{dt}$ is the velocity vector, $\psi = [\alpha, \beta, \gamma]$ are the roll, pitch and yaw angles of the drone respectively, $\omega \in \mathbb{R}^3$ is the intrinsic angular velocity of the drone body, $\psi' \in [-\frac{\pi}{2}, \frac{\pi}{2}]^2$ are the pendulum swing angles, $\dot{\psi}' = \frac{d\psi'}{dt}$ is the rate of change of the pendulum swing angles and $act \in [0,1]^4$ is the actuation state. The position and velocity are described in the global frame $\mathcal{W}$, where as the angular velocity ω is in the local frame $\mathcal{B}$. The initial drone state s_0 is computed according to the initial state distribution parameters 3.4 in the following way.

Parameter	Description	Value
q_m	max position deviation	2 [m]
v_m	max velocity magnitude	4 $[\mathrm{m\,s^{-1}}]$
ω_m	angular velocity magnitude	2 $[\mathrm{rad\,s^{-1}}]$
da_m	drone angle magnitude	1.6 [rad]
pa_m	pendulum angle magnitude	1 [rad]
pv_m	pendulum angular velocity	1 $[\mathrm{rad\,s^{-1}}]$

Table 3.4: Drone initial state distribution settings

The position q is sampled from a sphere with radius q_m centered on the starting position. The velocity $\dot{q}$ and angular velocity ω vectors are sampled uniformly from 3D spheres of radii v_m and ω_m respectively centered at the origin. The yaw angle is sampled uniformly on the interval $[0, 2\pi]$. The roll and pitch angles of the drone are both sampled uniformly from the interval $[-da_m, da_m]$. Similarly, the pendulum states ψ', $\dot{\psi}'$ are sampled from the intervals $[-pa_m, pa_m]$ and $[-pv_m, pv_m]$ respectively. The actuation is simply set to all zeros. All the distribution parameters 3.4 are further scaled by the state_difficulty value for easy configuration of the distribution size.

3.3.3 Environment interaction

To interact with the simulated environment, the underlying MuJoCo simulation can be stepped by calling the vector_step() function. This function accepts a list of actions $\{a_t^1, ..., a_t^n\}$, where n is the number of simulated drones and t is the current simulation timestep. It then performs a simulation step and returns a list of observations $\{o_{t+1}^1, ..., o_{t+1}^n\}$, a list of collected rewards

$\{r_t^1, ..., r_t^n\}$ and information about episode termination. If enabled, the scene is also rendered to a visualization window. Every action a is represented as

$$a \in [0,1]^4 \,, \tag{3.5}$$

or in other words, the action space $A = [0,1]^4$ is the four-dimensional unit cube and represents the desired motor settings for the four drone propellers. These are internally transformed to the control signals *ctrl* for each motor by the affine mapping $f(x) = 0.1 + 0.9 \cdot x$. This introduces bias to the control signal, so that we do not operate the drone motors at too low throttle, which could lead to the motor stalling in the real world. Each observation o is by default constructed as

$$o = \{q, \psi, \dot{q}, \omega, \psi', \dot{\psi}', \ddot{q}^{\mathcal{B}}, act, q^*, h^*, p\} \,, \tag{3.6}$$

which apart from the state variables (3.4) also includes the acceleration $\ddot{q}^{\mathcal{B}} = \frac{dq^{\mathcal{B}}}{dt}$ expressed in the local frame, the reference position $q^* \in \mathbb{R}^3$, reference heading $h^* \in [0, 2\pi]$ and the drone model parameters p (3.1). The reference and heading are necessary to inform the policy of the desired state. The parameters will become important later, when dealing with adaptive control and the acceleration is added to help with pendulum state estimation.

Chapter 4

Experiments

The three main tasks tackled in this thesis are:

1. Training a policy to control the position of a quadcopter with suspended pendulum.
2. Training a pendulum state estimator network.
3. Training a policy that is able to adapt to changing drone parameters.

All of these tasks have a common goal, which is to minimize the distance of a quadcopter from some setpoint state. A static reference setpoint is considered and is given by a reference position q^* and heading h^*, i.e. the desired yaw angle.

4.1 Reward parametrization

4.1.1 Distance based reward

The simplest kind of reward function that captures the objective of distance minimization is

$$r(s, q^*, h^*) = -\|q - q^*\| - c_h \cdot |h - h^*| \,, \tag{4.1}$$

where c_h determines the ratio of position and heading error in the reward. Since the actuation force of the drone motors is limited, the policy has to make some compromise between position and heading error minimization. The idea behind the coefficient c_h is to tune the relative importance of these two errors and steer the policy closer to desired behavior. However, the reward (4.1) is always negative which can be a problem. The value function used for advantage estimation is trained to regress the value estimate

$$\hat{V}_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l} \,, \tag{4.2}$$

which in this case is a sum of negative terms. The issue is that this value estimate can get larger, if the episode is simply truncated early. This can

steer the policy to learn behaviours that lead to early episode truncation. In our case, the episode is truncated if the drone gets too far from the reference position, and it can therefore happen, that the policy will simply learn to reach the truncation boundary as fast as possible. To circumvent this, a positive constant can be added to the reward function (4.1)

$$r_{\text{dist}}(s, q^*, h^*) = c - \|q - q^*\| - c_h \cdot |h - h^*| \,, \tag{4.3}$$

where $c > 0$ is chosen such that the reward is positive for the relevant part of the state space observed by the policy. The c constant can be interpreted as a reward for staying alive in the simulation.

The reward (4.3) is the simplest kind of reward that captures the objective of minimizing the distance from the setpoint state. However, this does not mean that this reward will yield optimal results. The PPO algorithm tries to optimize the policy using on-policy data in some neighbourhood of the policy space. This means that not only is the dataset biased by the used policy, but so is the policy update itself. The PPO algorithm can therefore get stuck in local maxima of the cummulative reward objective. The reward may then be adjusted to penalize undesired behaviours and help guide the policy training.

Effort penalization

In practice it is common to prefer policies that appart from minimizing the distance from a setpoint try to also conserve energy. These are opposing objectives as using less energy leads to less agressive control and therefore slower movements towards the reference setpoint. The effort penalization can easily be added to the reward (4.3) as

$$r_{\text{effort}}(s, q^*, h^*) = r_{\text{dist}}(s, q^*, h^*) - c_e \cdot \| \max(act - act_{\text{hover}}, 0)\| \,, \tag{4.4}$$

where act is the current actuation state, c_e is the effort coefficient and

$$act_{\text{hover}} = \frac{g(m_d + m_p)}{4F_m} \tag{4.5}$$

is the actuation state per motor required to hover the drone in steady state. The form of this effort penalty ensures that we do not penalize energy exerted to make the drone simply hover, which could slow down the policy training.

4.1.2 Energy based reward

An alternate reward is based on the observation, that while using rewards (4.3), (4.4) achieves low setpoint position error, the drone behaviour is rather chaotic. In particular, the policy performs aggresive movements, pumping energy into the pendulum, which then hinders its performance. Sometimes, this leads to destabilization and subsequent inability to recover. This reward is given as

$$r_{\text{energy}}(s, q^*, h^*) = r_{\text{effort}}(s, q^*, h^*) - c_d \cdot E_d - c_a \cdot (|\alpha| + |\beta|) \,, \tag{4.6}$$

where E_d is the scaled mechanical energy deviation of the pendulum, c_d is the energy deviation coefficient, α and β are the roll and pitch angles of the drone respectively and c_a is the angle coefficient. The roll and pitch angle penalization is added to incentivise the policy to stabilize the drone further. The idea of the E_d term is to penalize the total mechanical energy of the pendulum. However, the potential energy grows with height and so we cannot use it in the reward directly for obvious reasons. Instead, we take the mechanical energy and subtract a baseline corresponding to the lowest atainable energy in steady state, which is equal to the potential energy of the pendulum when hanging straight down. This is how we obtain the energy deviation

$$\Delta E = \frac{1}{2} m_p \|v_p\|^2 + m_p g z_p - m_p g z_{p0} = \frac{1}{2} m_p \|v_p\|^2 + m_p g z_d \,, \tag{4.7}$$

where v_p is the velocity of the pendulum in the world frame, z_p is the current height of the pendulum, z_{p0} is the minimum atainable height of the pendulum given the drone height, $z_d = z_p - z_{p0}$ is their difference, m_p is the mass of the pendulum weight and g is the gravitational acceleration. The energy deviation (4.7) corresponds to the extra energy of the pendulum when compared to the steady state of the drone-pendulum system. The deviation is further divided by m_p to obtain the scaled deviation

$$E_d = \frac{\Delta E}{m_p} = \frac{1}{2} \|v_p\|^2 + g z_d \,, \tag{4.8}$$

which is independent of the pendulum mass, but still proportional to ΔE. For a single drone model, this does not change anything, since we further scale the term by the coeffficient c_d. For multiple drone models, this rescaling causes the reward to penalize large angle deviations and angular velocities equally for all weight masses. The quantities v_p and z_d can be obtained from the drone state using (2.55), (2.60).

4.2 Observations

The following section discusses the observation vectors used for the various experiments in this chapter.

4.2.1 Full state observation

While training the policy to control the quadcopter, the full state information will be used as input to the policy. The observation vector is then

$$o = \{q^{*\mathcal{B}}, \alpha, \beta, h_{err}, \dot{q}^{\mathcal{B}}, \omega, \psi', \dot{\psi}', \ddot{q}^{\mathcal{B}}, act\} \,, \tag{4.9}$$

where $q^{*\mathcal{B}} \in \mathbb{R}^3$ is the position of the reference setpoint in the drone's frame, α and β are the roll and pitch angles respectively, $h_{err} \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is the signed heading error, $\dot{q}^{\mathcal{B}} \in \mathbb{R}^3$ is the drone's velocity in the local frame, $\omega \in \mathbb{R}^3$

is the drone's intrinsic angular velocity, $\psi' \in [-\frac{\pi}{2}, \frac{\pi}{2}]^2$ and $\dot{\psi}' \in \mathbb{R}^2$ are the angles and angular velocities of the pendulum joint, $\ddot{q}^{\mathcal{B}} \in \mathbb{R}^3$ is the drone's acceleration vector in local frame and $act \in [0,1]^4$ is the actuation state. The position, velocity and heading are obtained as

$$\begin{aligned} q^{*\mathcal{B}} &= R^\top (q^* - q) \\ \dot{q}^{\mathcal{B}} &= R^\top \dot{q} \\ h_{err} &= ((h^* - \gamma + \pi) \mod 2\pi) - \pi \end{aligned} \tag{4.10}$$

where $q^* \in \mathbb{R}^3$ is the reference setpoint position, $q \in \mathbb{R}^3$ is the drone's position in the world frame, R is the rotation matrix describing the orientation of the drone with respect to the world frame (viz. section 2.3.2), $\dot{q} \in \mathbb{R}^3$ is the drone's velocity in the world frame, h^* is the reference heading and γ is the drone's yaw angle.

This observation (4.9) effectively removes rendundant information corresponding to the translation of the drone-reference system and also the rotation about the world's z axis. The choice to represent the translation and velocity vectors in the local frame $\mathcal{B}$ is rather natural. The acceleration $\ddot{q}^{\mathcal{B}}$ vector is fully dependent on the rest of the drone's state and is thus redundant. It is however included in the observation as it is also used for pendulum state estimation, where the acceleration vector can provide useful information for the policy.

4.2.2 Full state observation with drone parameters

The observation (4.9) is well suited for a single drone model. When dealing with changing drone parameters however, it no longer provides enough information to fully describe the drone's state. This is why the observation is extended by the drone model parameters (3.1), which yields the observation vector

$$o = \{q^{*\mathcal{B}}, \alpha, \beta, h_{err}, \dot{q}^{\mathcal{B}}, \omega, \psi', \dot{\psi}', \ddot{q}^{\mathcal{B}}, act, p\}, \tag{4.11}$$

where p are the drone model parameters and the rest is the same as in (4.9).

4.3 Position control

In the first task, the goal is to train a neural network policy $\pi_\theta(a_t|s_t)$, that can be used to control a quadcopter such that a distance from a reference setpoint is minimized. Full state knowledge is assumed as the goal of this section is to simply explore the capabilities of the RL policy and different reward parametrizations. The observation vector used in this task (here labeled s_t) is the full drone state one as described in (4.9).

4.3.1 Neural network architecture

The policy used for control in this task consists of a neural network $n_\theta(s_t)$, which maps the state vector s_t to a distribution parameter vector $\kappa \in \mathbb{R}^8$.

This parameter vector is then mapped to

$$[\alpha, \beta] = \log(\exp(\text{clip}(\kappa, -50, 50)) + 1) + 1 \,, \tag{4.12}$$

where $\alpha = [\alpha_1, ..., \alpha_4]$ and $\beta = [\beta_1, ..., \beta_4]$ are parameters used for the action distributions $Beta_i(\alpha_i, \beta_i)$ (viz. (2.39)). The clip function is defined as

$$\text{clip}(x, a, b) = \begin{cases} a & \text{for } x < a \\ b & \text{for } x > b \\ x & \text{for } x \in [a, b] \end{cases} \tag{4.13}$$

Note that these distribution parameters are constructed such that $\alpha_i, \beta_i > 1$. During training, the action for i-th motor is sampled from the corresponding $Beta_i(\alpha_i, \beta_i)$ distribution. During evaluation, the action for each motor is instead obtained as a mean of the corresponding $Beta_i(\alpha_i, \beta_i)$ distribution given by

$$\mu_i = \frac{\alpha_i}{\alpha_i + \beta_i} \,. \tag{4.14}$$

The neural network $n_\theta(s_t)$ is constructed as an MLP (2.2.1) as illustrated on figure 4.1. It consists of 3 sets of layer l_s, l_a and l_v. The shared layers l_s are made up by two tanh activated fully connected layers with output sizes $[256, 128]$ respectively, which process the state s_t into the features f, which are then fed into the value layers l_v and action layers l_a. The action layers l_a are made up by a single linear layer (without activation function) and outputs the 8 parameters κ used for action sampling. The value layers l_v are then two fully connected layers with sizes $[128, 128]$ with tanh activation and a linear layer with a single output - the value function estimate $V(s_t)$.

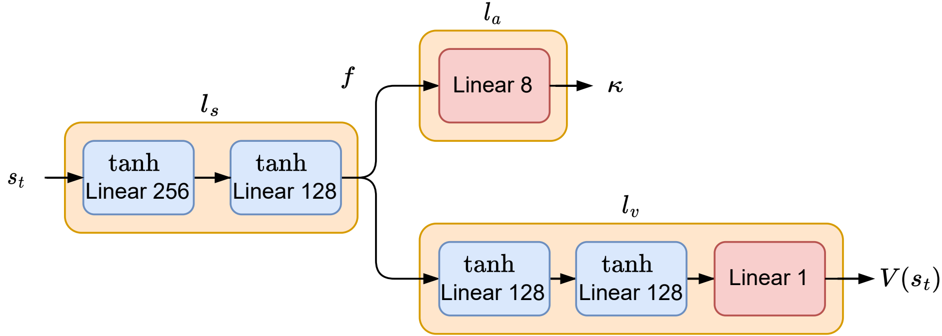


Figure 4.1: Neural network architecture

4.3.2 Training and environment configuration

The parameters of the PPO training (table 3.1) are set as shown in table 4.1. Additionaly, a learning rate schedule is used, such that the initial learning rate changes to 0.001 over the course of the training and than remains constant.

Parameter	Value
lambda	0.96
gamma	0.985
lr	0.01
clip_param	0.2
vf_loss_coeff	1
train_batch_size	131072
sgd_minibatch_size	32768
num_sgd_iter	10
rollout_fragment_length	256
num_rollout_workers	8

Table 4.1: PPO training configuration

The environment parameters (table 3.2) are set as shown in table 4.2.

Parameter	Value
num_drones	64
frequency	200
skip_steps	2
reference	$[0, 0, 10, 0]$
start_pos	$[0, 0, 10, 0]$
max_distance	4
max_steps	1024
state_difficulty	0.5
param_difficulty	0

Table 4.2: BaseDroneEnv training configuration

The drone model is considered to be non-changing and thus the param_difficulty of the environment is set to 0. This makes is so that the model parameters of all the drones in the environment are set to the center values described in table 3.3. The state_difficulty is set to 0.5, as this leads to reasonable results in reasonable amount of time. Starting position start_pos is set to be the same as the reference, corresponding to position 10 meters above the origin and 0 heading angle.

4.3.3 Results

Distance reward with effort penalization

The following results were obtained by using the distance-based reward (4.4) with coefficients $c = 3$, $c_h = 0.4$ and $c_e = 0.8$ for the policy training. The reward (4.4) was further scaled down by a factor of 100, because of the underlying training implementation not being reward scale invariant. The policy was trained for 200 epochs, until the mean episode reward stopped

improving at a value of 27.16. The figure 4.2 shows the progression of mean episode reward and length as well as the changing learning rate.

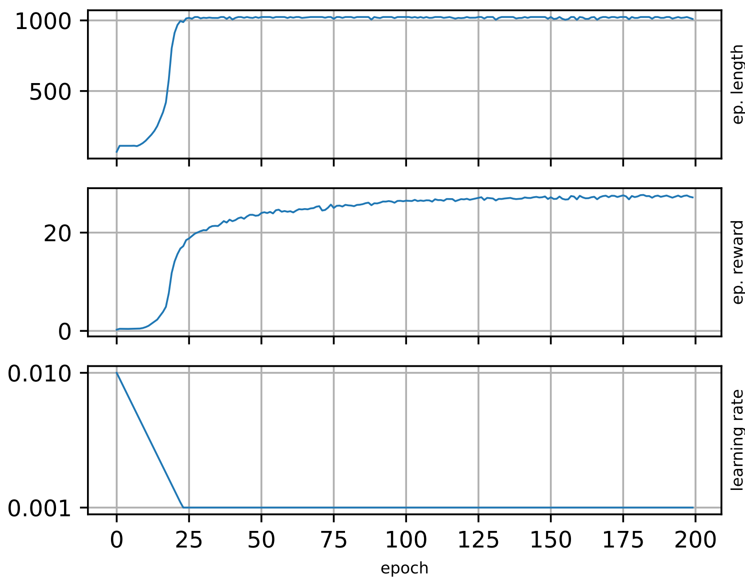


Figure 4.2: Control MLP training with distance-effort based reward

The figure 4.3 shows ten example trajectories starting in randomly sampled states around the coordinates $q_0 = [-2, 0, 10]$. The state distribution follows the settings used for training. The reference setpoint is set to $q^* = [0, 0, 10]$ and $h^* = 0$. The figures 4.4 and 4.5 show the corresponding quadcopter coordinate and angle evolution of the drone trajectories respectively.

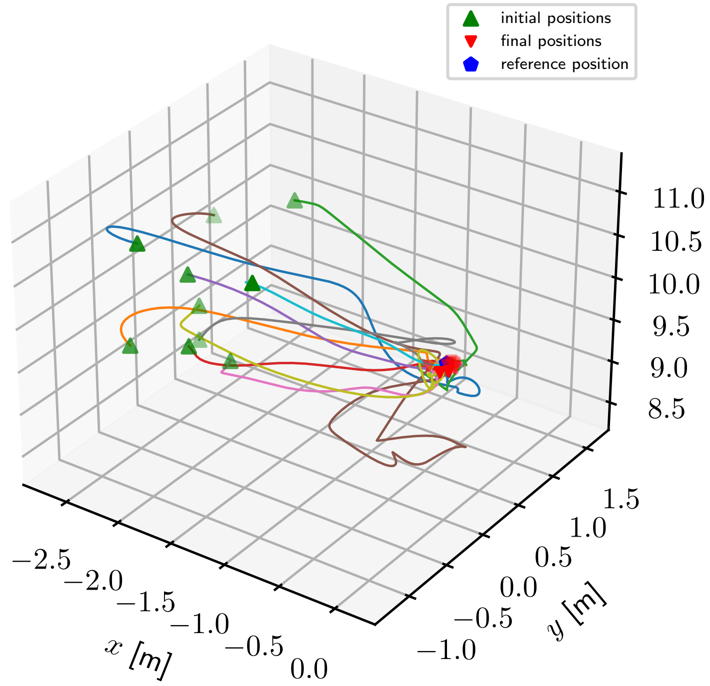


Figure 4.3: Control MLP with distance-effort based reward trajectories

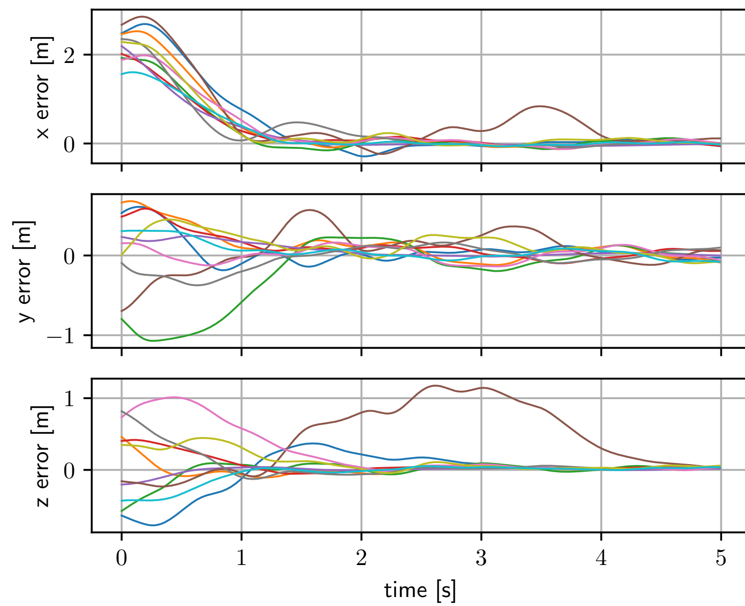


Figure 4.4: Control MLP with distance-effort based reward position

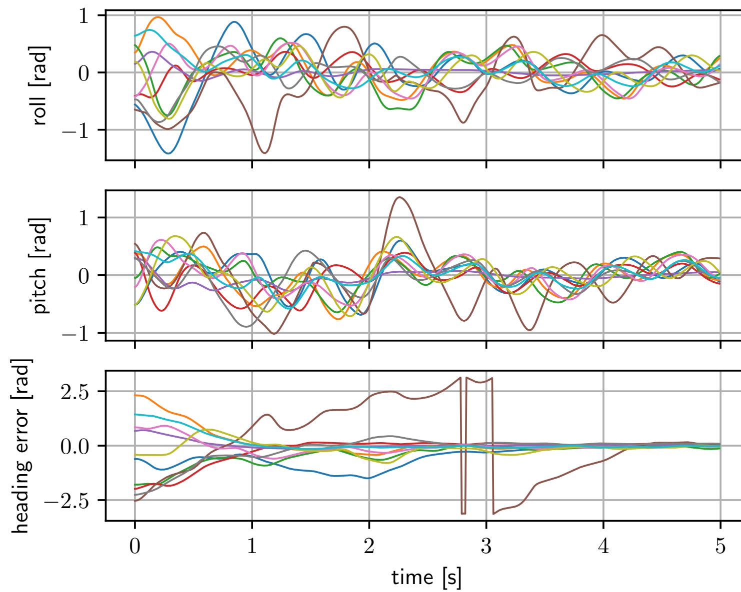


Figure 4.5: Control MLP with distance-effort based reward angles

Energy based reward

This experiment uses the energy based reward (4.6) with coefficients $c = 7$, $c_h = 0.4$, $c_e = 0.1$, $c_d = 0.1$, $c_a = 0.05$ for training the policy. This reward is again scaled down by a factor of 100 due to the underlying implementation. The policy is again trained for 200 epochs, until the mean episode reward stopped improving at the value 78.74. The figure 4.6 shows the progression of mean episode reward and length, as well as the learning rate progression.

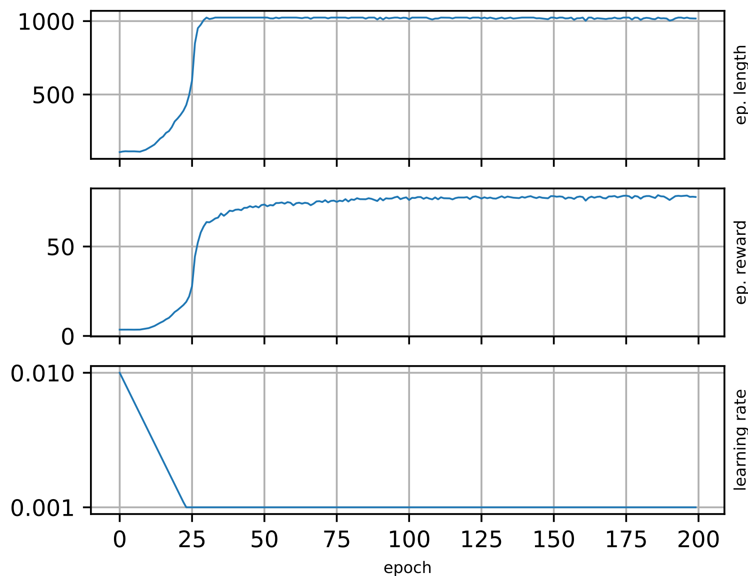


Figure 4.6: Control MLP training with distance effort reward

The figure 4.7 shows ten example trajectories generated in the same way as in 4.3.3. The starting states are in fact identical to the ones used in 4.3. The figures 4.8 and 4.9 show the corresponding quadcopter coordinates and angles.

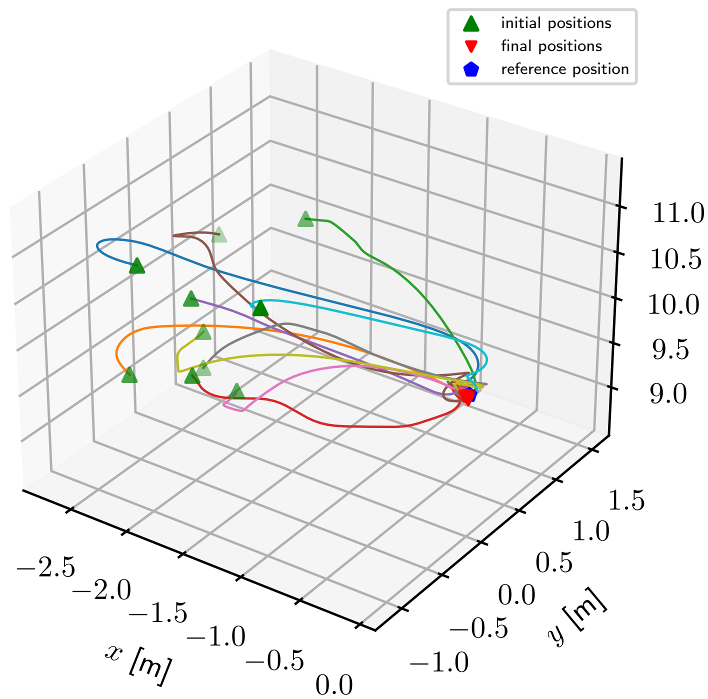


Figure 4.7: Control MLP with energy based reward trajectories

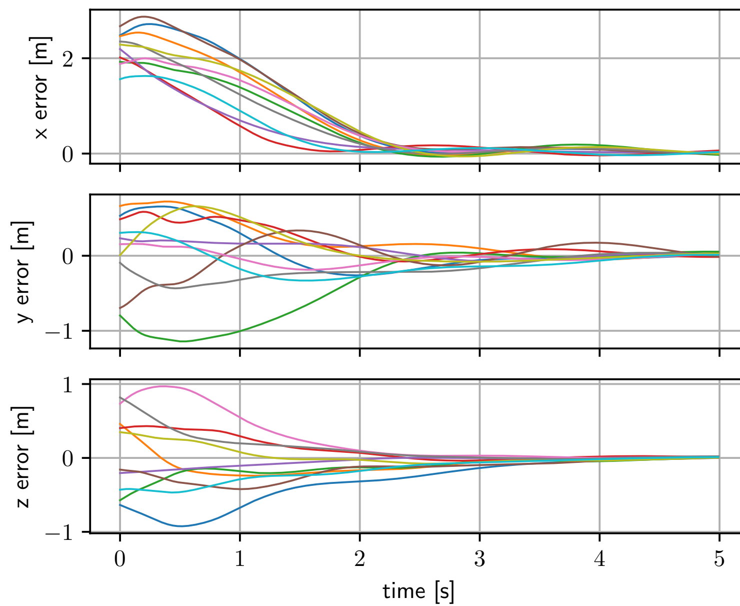


Figure 4.8: Control MLP with energy based reward position

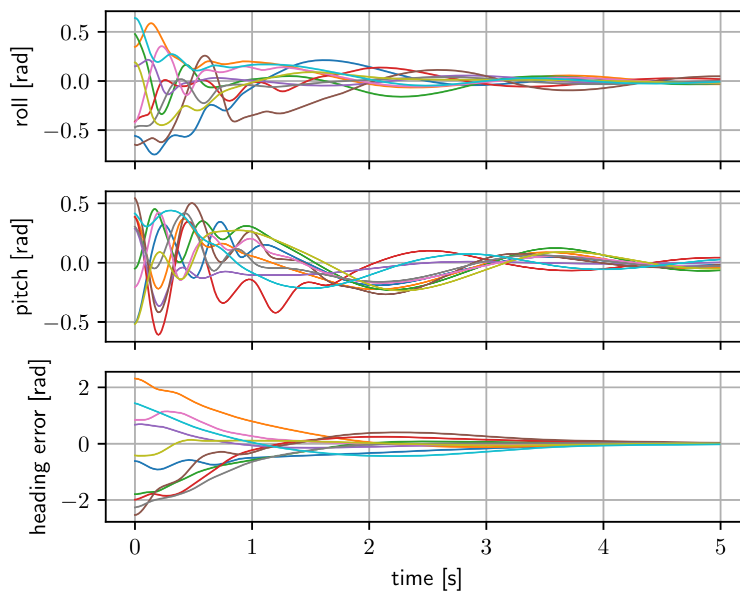


Figure 4.9: Control MLP with energy based reward angles

Trajectory tracking

The performance of the learned policy is also evaluated on trajectory tracking. The policy is designed with a static reference in mind, but trajectory tracking can be achieved by simply moving the static reference setpoint between timesteps. The policy does only have access to a single point on the trajectory and has no information about the future evolution of the trajectory. Despite

this, it is quite interesting to see how the policy copes with dynamic setpoint tracking. The trajectory consists of a spiral to show response to the setpoint moving in all three dimensions. The heading is also varied to point in the direction of movement. The resulting quadcopter trajectories are shown in figure 4.10 and correspond to the policies trained using energy-based and distance-based reward from previous section. Figure 4.11 then shows the evolution of the reference setpoint as well as the drone positions and heading angle for both trajectories. Figure 4.12 then shows the evolution of the roll and pitch angles in the trajectories.

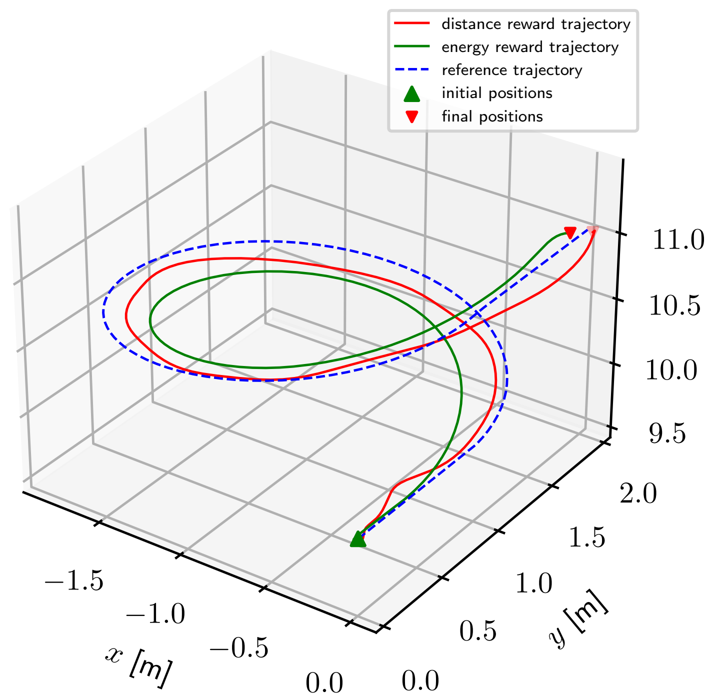


Figure 4.10: Dynamic setpoint tracking trajectories

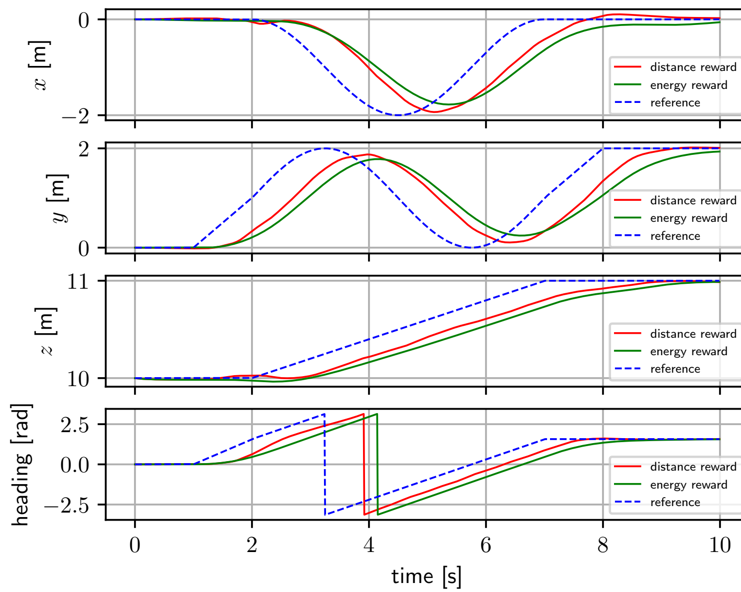


Figure 4.11: Dynamic setpoint tracking position and heading

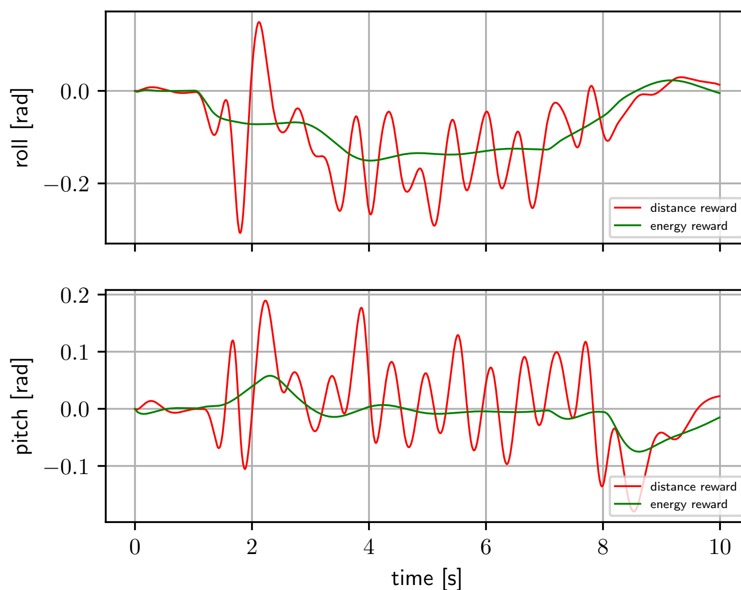


Figure 4.12: Dynamic setpoint tracking angles

4.4 Pendulum state estimation

In the second task, the goal is to show whether the pendulum states can be estimated using a state estimation network. Here, it is necessary to differentiate between a state of the system s_t and the observation o_t. Just as in the previous task, the full state observation (4.9) is used for training. Only

this time, the observed state vector s_t is decomposed into the observation vector o_t and the pendulum state $[\psi', \dot{\psi}']$. The policy can be constructed just as in the previous task, only now it operates on the state estimate $\hat{s}_t$ instead of the ground truth state s_t. This state estimate $\hat{s}_t$ is constructed by concatenating the observation vector o_t with the pendulum state estimate $[\psi', \dot{\psi}']_{\text{est}}$. The goal here is then to compute $[\psi', \dot{\psi}']_{\text{est}}$ using only the known observations.

4.4.1 Neural network architecture

The general structure of the used policy is depicted in figure 4.13. The policy $\pi(a_t|\hat{s}_t)$ is the same as used in the position control task 4.3.1, only now the state estimate $\hat{s}_t$ replaces the state s_t. The policy does however not have direct access to the pendulum state $[\psi', \dot{\psi}']$. It is instead replaced with the estimated pendulum state $[\psi', \dot{\psi}']_{\text{est}}$, which is computed by the state estimator module $\eta(o_{t-T:t}, a_{t-T:t-1})$ based on the observation and action histories $o_{t-T:t}$, $a_{t-T:t-1}$. The state estimator network $\eta(o_{t-T:t}, a_{t-T:t-1})$ architecture is discussed in sections 4.4.1 and 4.4.1. Regardless of the underlying architecture, the state estimator networks are trained using the supervised loss

$$L_{\text{est}} = \|[\psi', \dot{\psi}']_{\text{est}} - [\psi', \dot{\psi}']\|^2 \,, \tag{4.15}$$

which leverages the knowledge of the ground truth pendulum states.

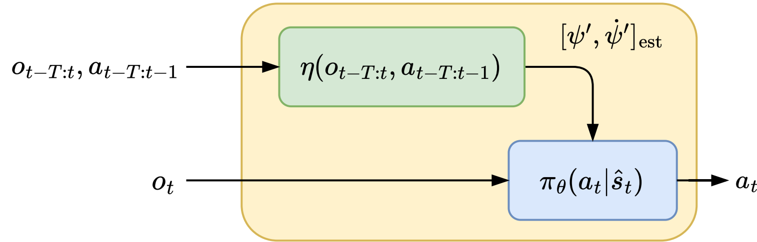


Figure 4.13: Policy with pendulum state estimator

LSTM state estimator

The first estimator module architecture is based around the LSTM 2.2.2 block. The computation graph is depicted in figure 4.14. The input to this RNN at each timestep t is the previous observation and action o_{t-1} , a_{t-1} paired with the current observation o_t. These are passed through two fully connected tanh activated layers with output of sizes 32. The output of these fully connected layers then feeds into the LSTM block of size 32 along with its previous hidden state h_{t-1} and cell state c_{t-1}. The output of the LSTM block feeds into a final linear layer of size 4, which outputs the state estimates $[\psi', \dot{\psi}']_{\text{est}} \in \mathbb{R}^4$.

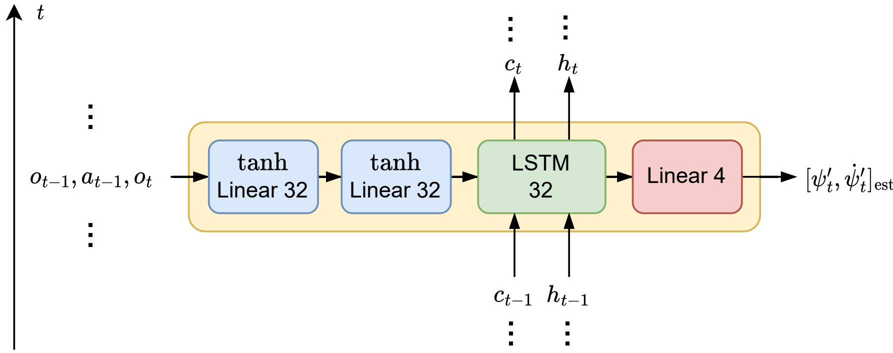


Figure 4.14: LSTM estimator network architecture

CNN state estimator

The second state estimator architecture is based on the 1D convolution 2.2.3. The computation graph of the network is depicted in figure 4.15. The input to the network at timestep t is a sequence of action-observation pairs $(a_{l-1}, o_l)_{l=t-T}^{t}$. The time window size is chosen to be $T = 64$. This sequence is first passed through a tanh activated linear layer of output size 32, and is then fed through the two time-wise convolutional layers. The convolutional layers are of sizes $(32, 5, 2)$ and $(16, 5, 1)$, where the numbers correspond to output feature dimension, kernel size and stride respectively. The output of the second convolution is then flattened into a single feature vector, which is then passed through another linear layer of size 128 with tanh activation and finally linearly mapped to the state estimates $[\psi', \dot{\psi}']_{\text{est}} \in \mathbb{R}^4$.

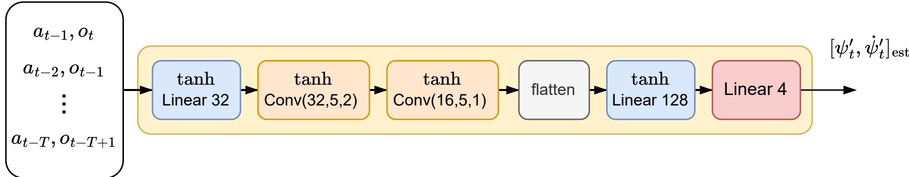


Figure 4.15: CNN estimator network architecture

4.4.2 Training and environment configuration

Since the pendulum state estimation is supposed to be used in order to control the quadcopter just as in the previous task 4.3.1, there is no need to change the training parameters. Therefore the same setup from 4.3.2 is used for the pendulum state estimation as well. The policy is first trained in the same way as in previous task 4.3.1 using ground truth pendulum state $[\psi', \dot{\psi}']$.

After the policy training is finished, the ground thruth state is replaced by the estimate $[\psi', \dot{\psi}']_{\text{est}} = \eta(o_{t-T:t}, a_{t-T:t-1})$, the weights of the policy are fixed and only the estimator is trained with on-policy data, i.e. while unrolling trajectories using the policy $\pi(a_t|\hat{s}_t)$. Some training parameters are here

changed according to the table 4.3. All the other parameters of the training and of the policy are reused from 4.3.2. Most notably, the learning rate is made smaller and the rollout length has been increased to span the maximum episode length of 1024. The number of used quadcopters in the simulation has been decreased to 16 as to keep the train batch size the same. A learning rate schedule is used, that slowly brings the learning rate from initial 0.001 down to 0.0001.

Parameter	Value
lr	0.001
rollout_fragment_length	1024
num_drones	16

Table 4.3: Estimator training configuration

4.4.3 Results

Policy without state estimation

In order to showcase the impact of the unknown pendulum angles, the simple policy 4.3.1 is first trained in the same way as in 4.3.3. The pendulum states are however removed from the observation vector (4.9). The policy is trained for 200 epochs and the progression of the mean episode reward, length and learning rate is shown in figure 4.16. The reward has stopped improving after reaching a value of 77.7.

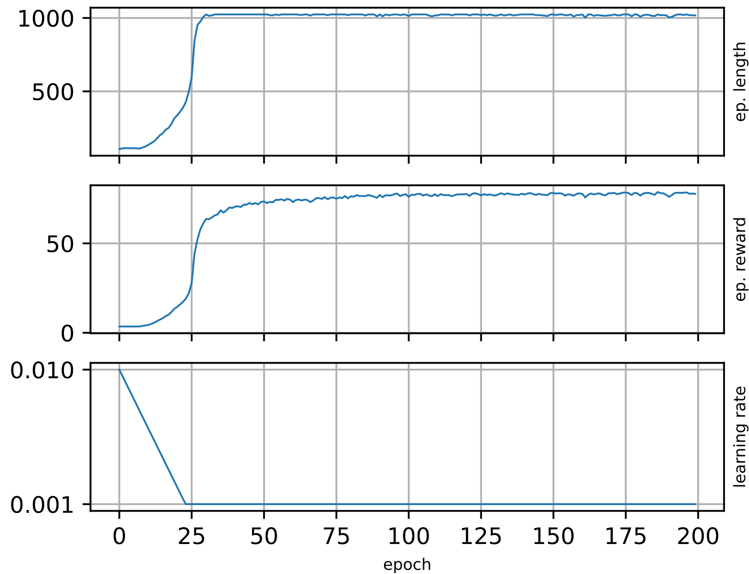


Figure 4.16: Control MLP with unknown pendulum state training

The figure 4.17 shows ten example trajectories beggining in initial states sampled in the same way as in 4.3. The figures 4.18 and 4.19 show the evolution

of quadcopter coordinates and angles from the trajectories respectively.

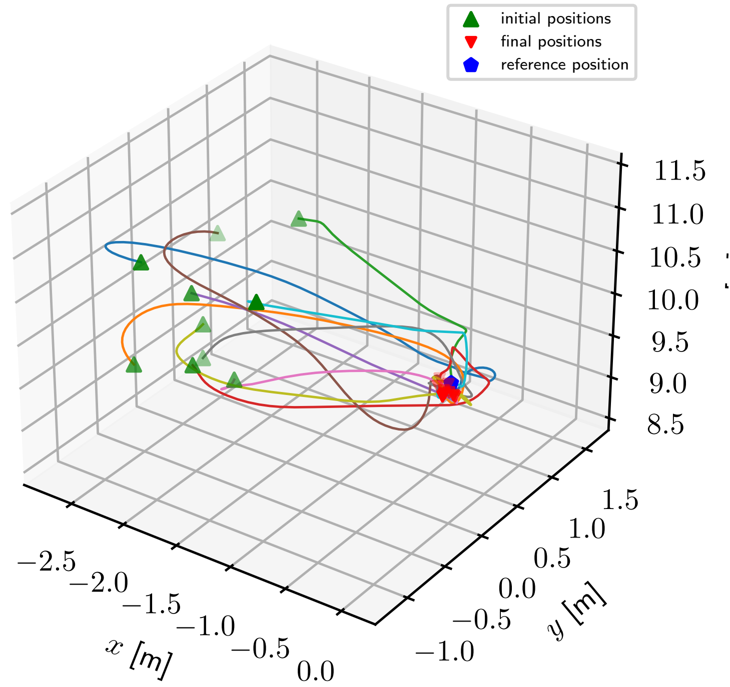


Figure 4.17: Control MLP with unknown pendulum state trajectories

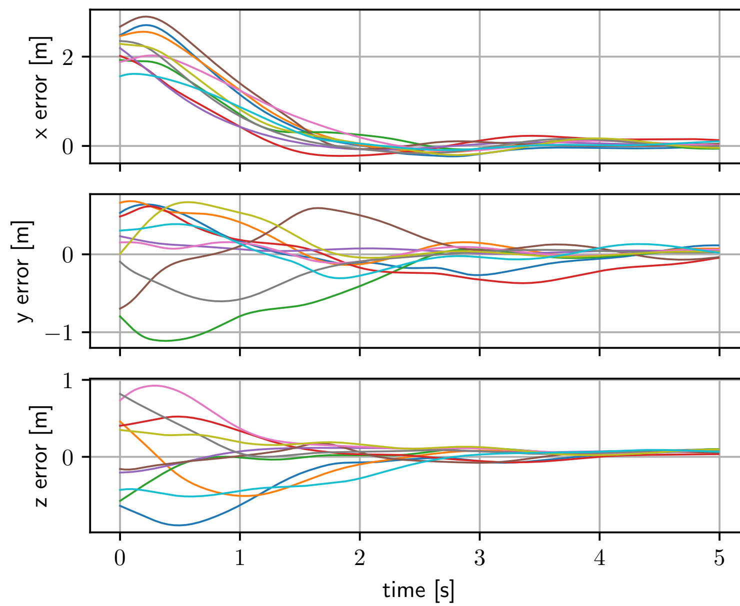


Figure 4.18: Control MLP with unknown pendulum state errors

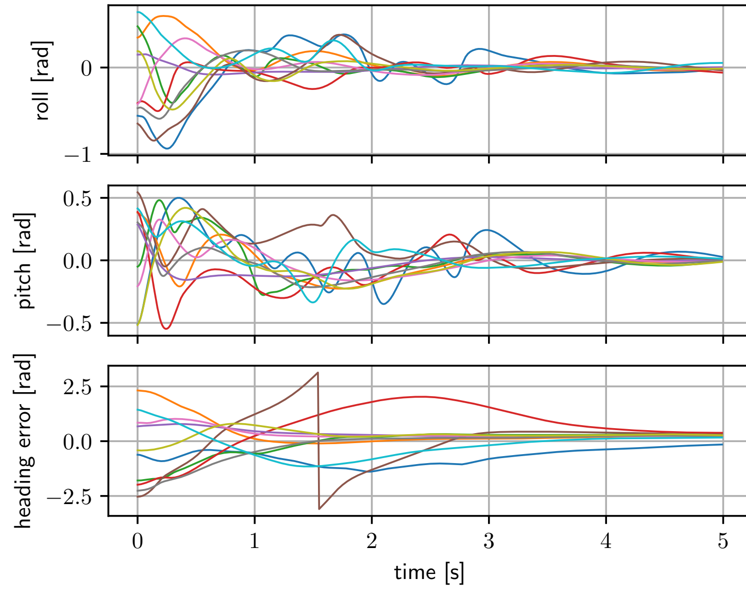


Figure 4.19: Control MLP with unknown pendulum state angles

LSTM state estimator policy

Even though the policy from 4.3.1 is used, it is trained from scratch for completeness. The energy based reward (4.6) with coefficients $c = 7$, $c_h = 0.4$, $c_e = 0.1$, $c_d = 0.1$, $c_a = 0.05$ is used for training the policy. This is again scaled down by 100. The figure 4.20 shows the progression of the mean episode length, mean episode reward and learning rate. After convergence of the mean episode reward at a value of 78.61, the policy weights are frozen and the estimator module is trained using the MSE loss to predict the pendulum states. The LSTM estimator network training took 100 epochs and its progress is depicted in the figure 4.21. The final MSE loss stopped improving after reaching $4.9 \cdot 10^{-3}$.

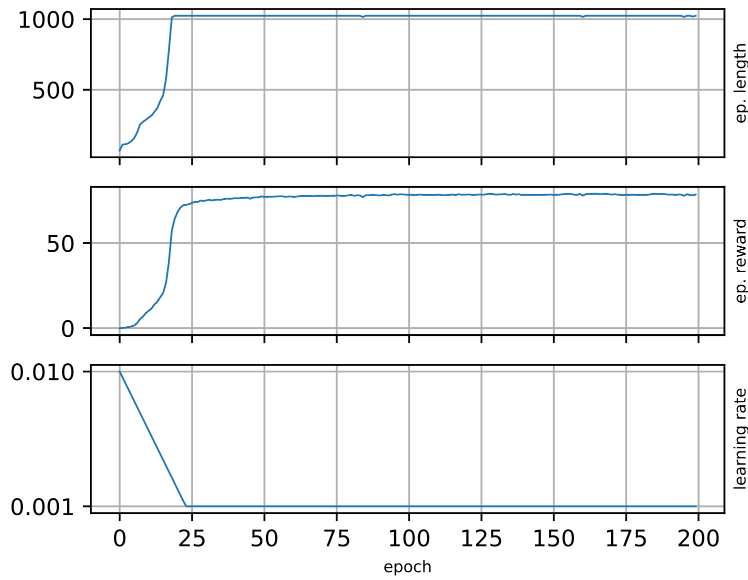


Figure 4.20: LSTM estimator policy training

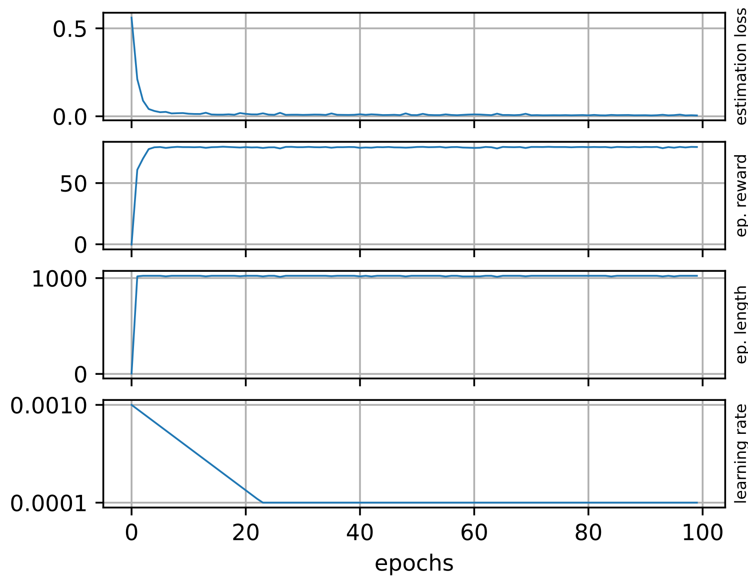


Figure 4.21: LSTM estimator training

The figure 4.22 shows ten example trajectories beggining in initial states sampled in the same way as in 4.3. The trajectories were obtained using the trained policy and the LSTM state estimator network. The figures 4.23 and 4.24 show the evolution of quadcopter coordinates and angles from the trajectories respectively. The figure 4.25 then shows the ground truth pendulum angles corresponding to one of the trajectories $\psi' = [\psi'_1, \psi'_2]$ and pendulum angular rates $\dot{\psi}' = [\dot{\psi}'_1, \dot{\psi}'_2]$ together with their estimates.

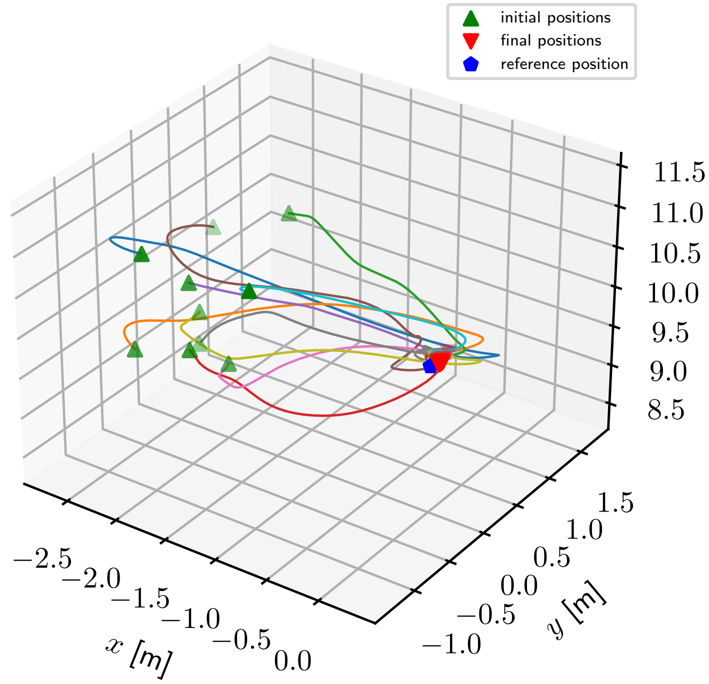


Figure 4.22: LSTM estimator policy trajectories

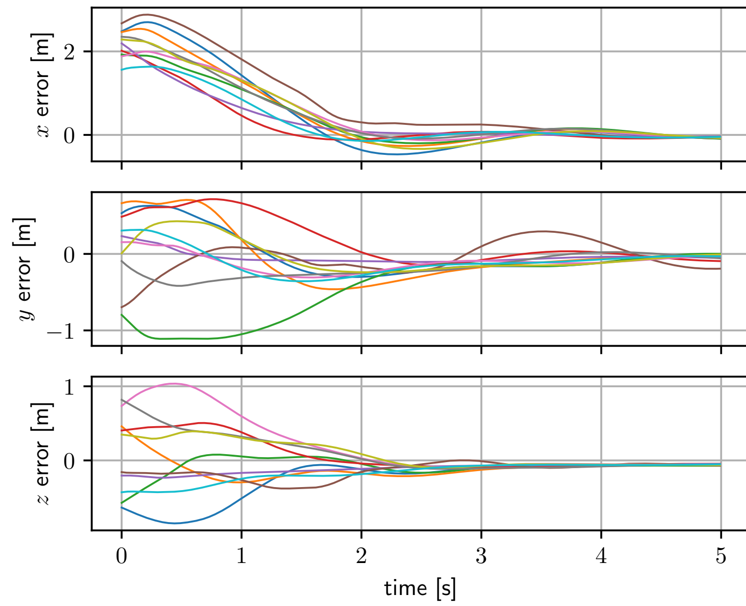


Figure 4.23: LSTM estimator policy position errors

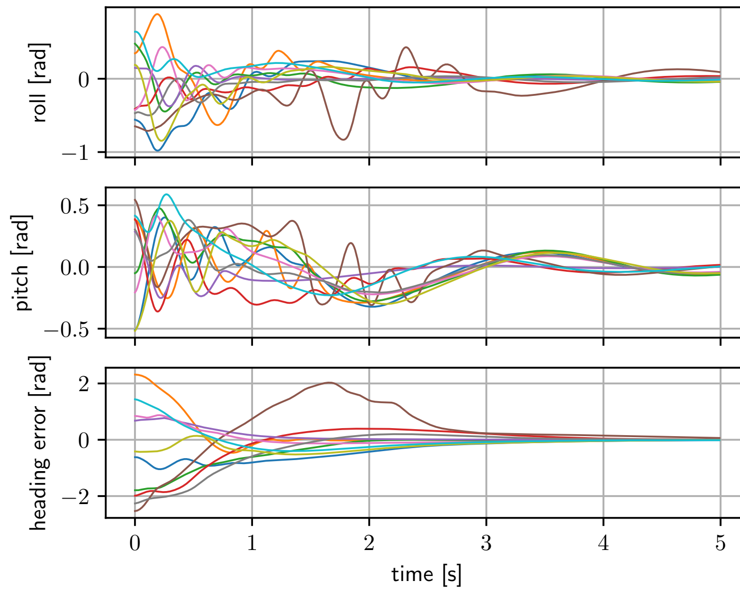


Figure 4.24: LSTM estimator policy angles

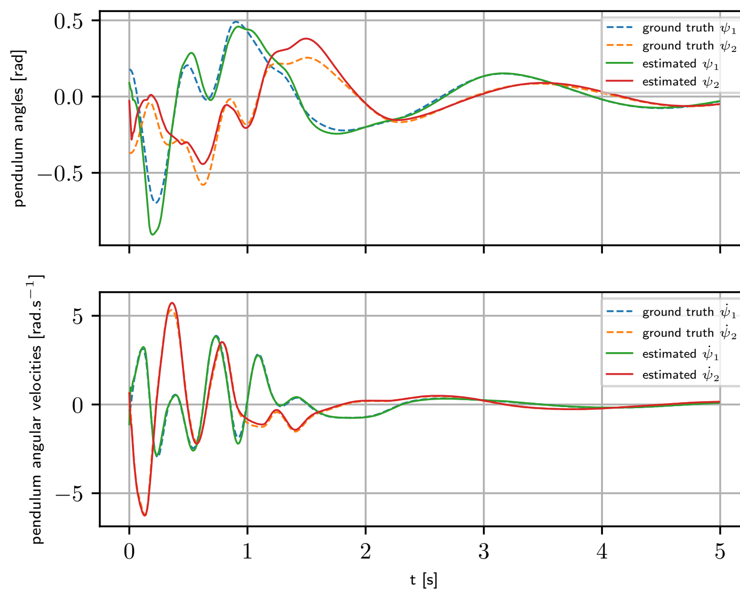


Figure 4.25: LSTM estimator angle estimates

CNN state estimator

The policy is trained in exactly the same way as in 4.4.3 and the mean episode reward, length and learning rate progress can be seen in figure 4.26. The reward stopped improving after reaching a value of 78.47. The state estimator network, which is again trained in the same way as in 4.4.3, has its training progress depicted in the figure 4.27. The final training MSE loss is $1.1 \cdot 10^{-2}$.

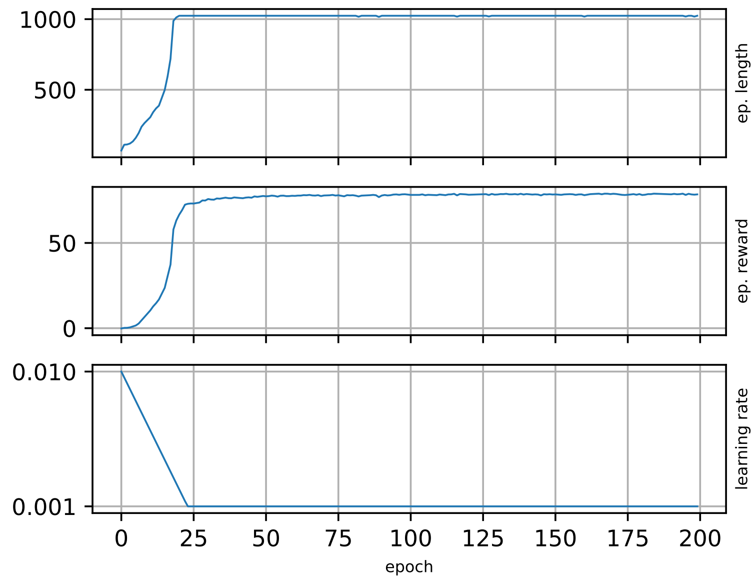


Figure 4.26: CNN estimator policy training

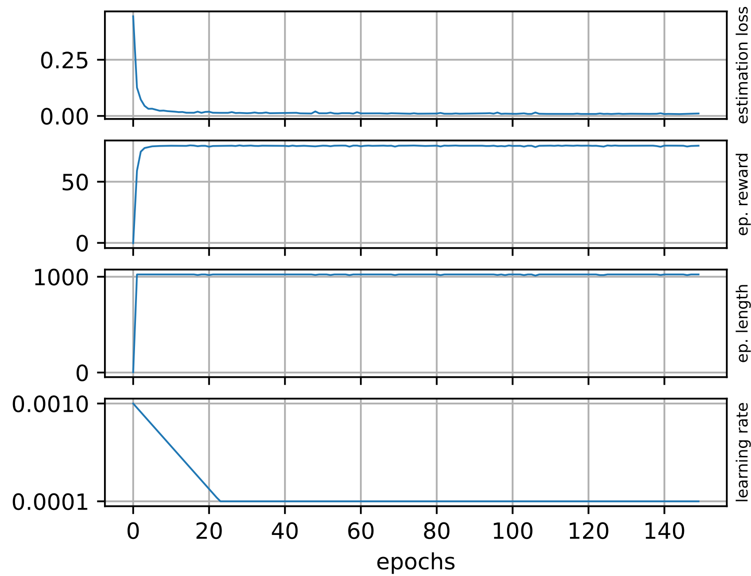


Figure 4.27: CNN estimator training

The figure 4.28 again shows the ten example trajectories, where the initial states and reference are the same as in 4.22. The figures 4.29 and 4.30 show the evolution of coordinates and angles from the trajectories respectively. The figure 4.31 then shows the ground truth pendulum angles corresponding to one of the trajectories $\psi' = [\psi'_1, \psi'_2]$ and pendulum angular rates $\dot{\psi}' = [\dot{\psi}'_1, \dot{\psi}'_2]$ together with their estimates.

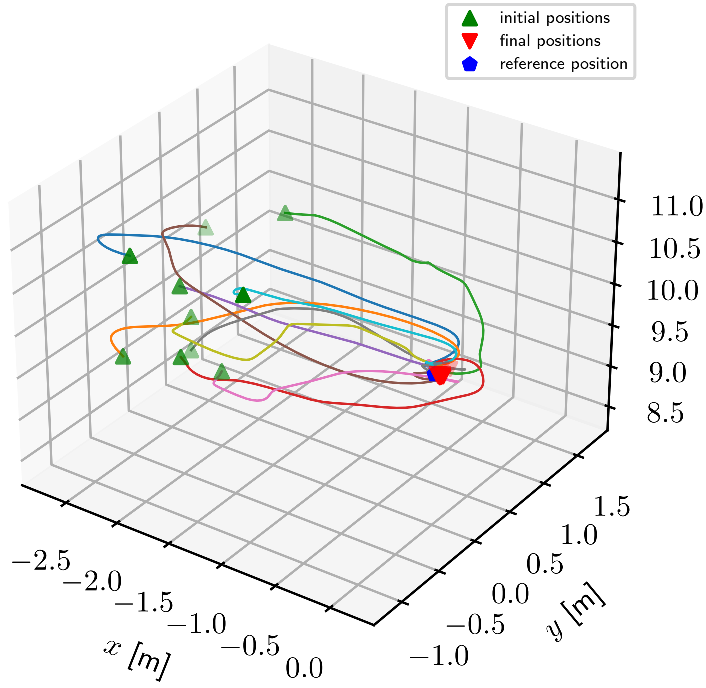


Figure 4.28: CNN estimator policy trajectories

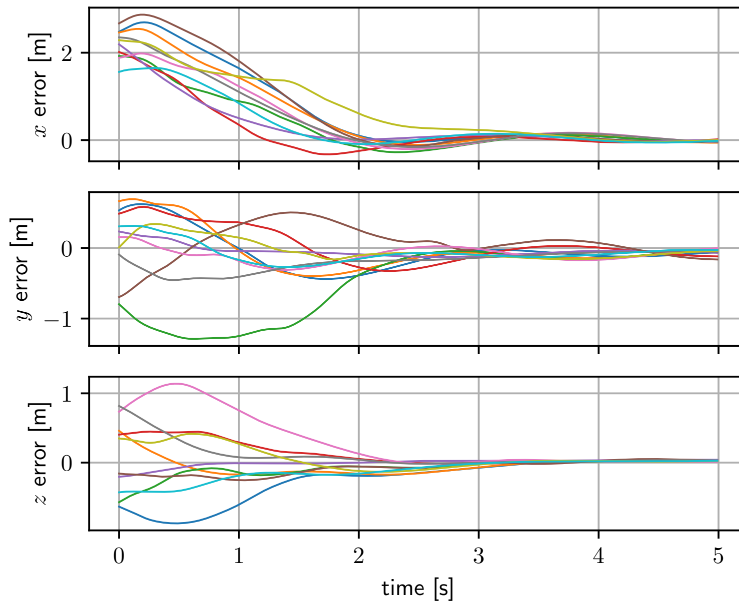


Figure 4.29: CNN estimator policy position errors

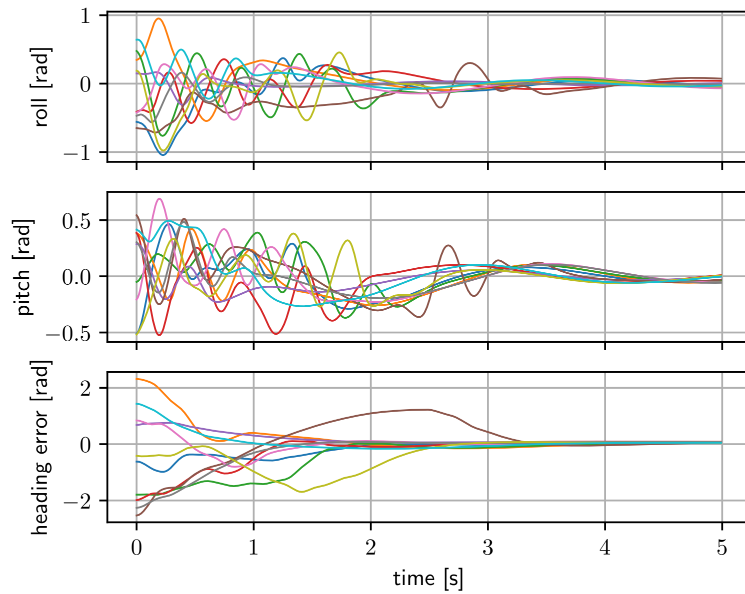


Figure 4.30: CNN estimator policy angles

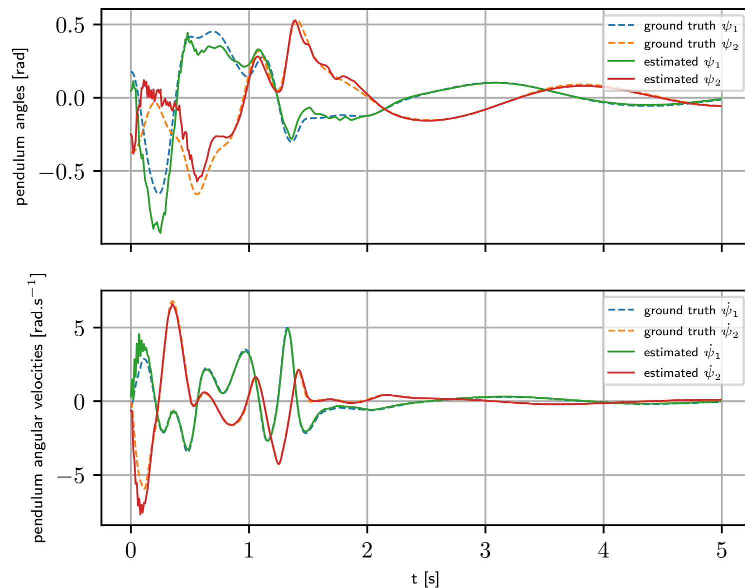


Figure 4.31: CNN estimator angle estimates

Performance comparison

This experiment compares the performace of a control policy that has full state information and three policies that do not have access to the pendulum states. The comparison criterium is the average accumulated reward per episode. The compared policies are the three trained in section 4.4.3 and the energy based one from section 4.3.3, which all use the same energy-based

reward for training. A total of 1024 trajectories are unrolled using each policy while sampling the initial state around the reference setpoint in the same way as during training 4.3.2. Every trajectory lasts for 5 seconds. The resulting average rewards for all four policies are shown in table 4.4. The full state and partial state columns refer to the policy discussed in 4.3.1 trained with full state information and without pendulum state in the observation respectively.

Policy	Full state	Partial state	LSTM estimator	CNN estimator
Reward	37.45	36.62	37.31	37.29

Table 4.4: Performance comparison rewards

State estimation comparison

To compare the pendulum estimation quality, the following experiment was performed. A total of 1024 drone trajectories were unrolled using the trained policy with estimator from a randomly sampled position around the reference setpoint using the training parameters. The trajectories were terminated after one second to avoid saturating the data with pendulum states near the steady state, which would not yield very informative measure of the performance. The first 64 timesteps of these trajectories are ignored, as the CNN estimator network does not work very well without enough samples. The error in estimated state is evaluated using mean squared error (MSE) for all four components separately and is compared for both the CNN-based and LSTM-based state estimator in table 4.5.

Network	ψ'_1 **MSE**	ψ'_2 **MSE**	$\dot{\psi}'_1$ **MSE**	$\dot{\psi}'_2$ **MSE**
LSTM estimator	$1.0 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$8.4 \cdot 10^{-2}$	$6.2 \cdot 10^{-2}$
CNN estimator	$7.8 \cdot 10^{-3}$	$1.0 \cdot 10^{-2}$	$1.1 \cdot 10^{-1}$	$7.1 \cdot 10^{-2}$

Table 4.5: State estimation accuracy comparison

4.5 Model parameter adaptation

In the third task, the goal is to adapt to changing model parameters.

4.5.1 Neural network architecture

One of the main issues with deploying a RL based policy trained using a simulator in the real world is the mismatch between the simulated model and the real system. One way to combat this is to simply train the policy as in 4.3.1, while varying the drone model parameters. This then leads to the policy being more robust to model mismatch, but may also cause the policy to be too conservative. This is because the policy is not given any information about the drone model dynamics and thus is forced to select an action that is likely to work for any model from the training set. Another

solution would be to provide the drone model parameters to the policy, which would then enable the policy to adapt its decision to the model. These parameters are however known precisely only for the simulated model, and are troublesome to acquire for the real one. Luckily there are ways to get around this problem. One approach is to use an RNN network to construct the policy. The hope is that the RNN will be able to adapt to the changing drone behaviour implicitly. Another approach employed in this thesis is to use a neural network architecture, that is first trained with the known model parameters in the observation vector and then an adaptation neural network is trained to predict these parameters based on a state action history.

LSTM adaptation network

The RNN architecture used for implicit model adaptation is depicted in figure 4.32. This network is based around the original control policy used in 4.3.1. The main difference is that a parallel recurrent network branch has been added to the shared layers l_s used in 4.3.1. This reccurent branch consists of two tanh activated fully connected layers with sizes 256 and 64 respectively. The output of these two FC layers is fed into the LSTM 2.2.2 layer of size 64. The output of this LSTM block is concatenated with the output of the feed forward branch from the shared layers l_s into a feature vector f. The rest of the network is the same as in 4.3.1. The LSTM is of course also fed with its previous cell and hidden states c_t, h_t, but this is ommited from the graph 4.32 for clarity.

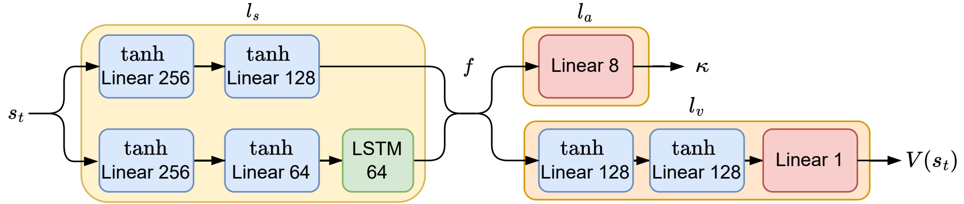


Figure 4.32: LSTM adaptation network computation graph

Rapid motor adaption paper

The second approach to make the policy robust to changing model parameters used in this thesis is adapted from the paper RMA: Rapid Motor Adaption for Legged Robots[9] by Kumar et al. The policy used in the paper consists of a base policy

$$\pi(a_t|s_t, a_{t-1}, e_t)\,, \tag{4.16}$$

which outputs the distribution over the action space as usual. The inputs are the current state s_t, the previous action a_{t-1} and the model parameter encoding e_t. The model parameter encoding is obtained via a Environment Factor Encoder

$$e_t = \mu(p_t)\,, \tag{4.17}$$

which is itself another neural network mapping the ground truth model parameters p_t to the encoding e_t. This policy is trained using the PPO algorithm 2.1.5 to maximize the cummulative reward. Once converged, the policy is supposed to be able to adapt to the changing model parameters p_t. This is the first phase of the training. The second phase replaces $\mu(p_t)$ with an adaptation module

$$\hat{e}_t = \phi(s_{t-T:t-1}, a_{t-T:t-1})\,, \tag{4.18}$$

which aims at estimating the parameter encoding $\hat{e}_t$ based on the state and action histories $s_{t-T:t-1}$, $a_{t-T:t-1}$. The adaptation module in the paper is constructed by chaining an MLP and a 1D (time-wise) convolutional neural network. The output of the adaptation module $\hat{e}$ is used as an input to the base policy $\pi(a_t|s_t, a_{t-1}, \hat{e}_t)$ instead of the ground truth e_t, so that the adaptation module is trained on-policy. The weights of μ and π are fixed as they are assumed to be already optimized. The loss

$$MSE(\hat{e}_t, z_t) = \|\hat{e}_t - e_t\|^2 \tag{4.19}$$

is then minimized, while using the known encoding $e_t = \mu(p_t)$. At deployment, only ϕ and π are used to adapt to unknown and possibly changing model parameters p_t. The computational graph of both training phases can be seen in figure 4.33.

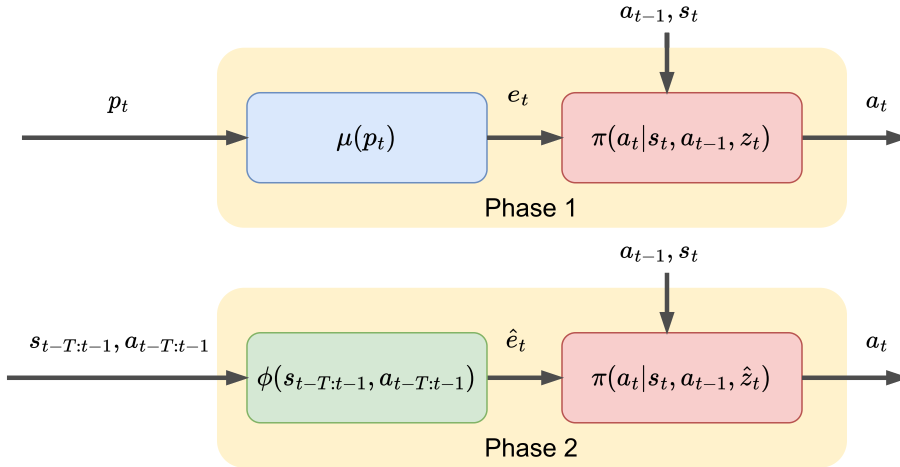


Figure 4.33: RMA computation graph

RMA model

The policy is not given the previous action a_{t-1} here, since the actuation state is already present in the state vector s_t. The policy $\pi(a_t|s_t, e_t)$ is then constructed in the same way as in 4.3.1, the only difference being, the input dimension is expanded by the parameter embedding size, which is chosen to be 8. The environment factor encoder $\mu(p_t)$ is modelled as a two-layer

MLP depicted in figure 4.34, that consists of a tanh activated FC layer of size 32 followed by a simple linear layer, which outputs the parameter encoding $e_t \in \mathbb{R}^8$.

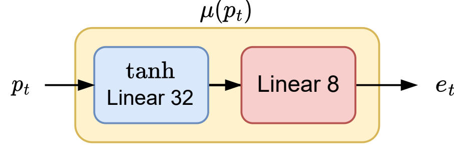


Figure 4.34: RMA environment factor encoder computation graph

The adaptation module computation graph is depicted in 4.35. It is given the state and action histories $s_{t-T+1:t}$, $a_{t-T:t-1}$, which are then processed in the following manner. The module first passes every state-action pair $[s_t, a_{t-1}]$ of the state action history through two tanh FC layers of sizes 32 and 32. The results are passed through two tanh activated 1D convolutional layers with parameters $(32, 5, 2)$ and $(16, 5, 1)$, where the parameters correspond to the output feature dimension, kernel receptive field and stride respectively. The output of these convolutions is then flatten and passed through another tanh FC layer of output size 32 and finally linearly mapped to the parameter embedding estimate $\hat{e}_t \in \mathbb{R}^8$.

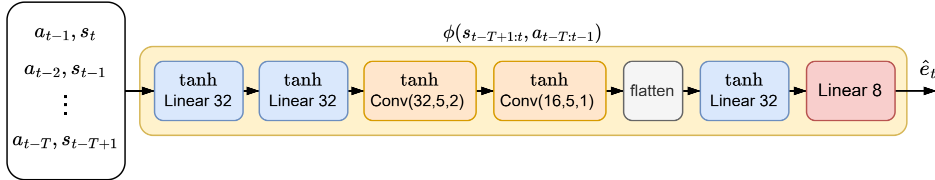


Figure 4.35: RMA adaptation module computation graph

4.5.2 Training and environment configuration

Most of the training and environment parameters are again set as in 4.3.2. The only exception are the parameters shown in table 4.6. This makes it so that the drone model parameters are regenerated periodically every 256 timesteps as discussed in 3.3.1. The reduced state difficulty compared to 4.3.2 is set so that the policy converges faster.

Parameter	Value
param_difficulty	2
state_difficulty	0.3
regen_env_at_steps	256

Table 4.6: Adaptation training configuration

For the adaptation module training, further adjustments are made as shown in table 4.7. A learning schedule is used, that starts at learning rate 0.001 and progresses to the final value of 0.0001.

Parameter	Value
regen_env_at_steps	256
num_drones	32
max_steps	512
lr	0.001
num_sgd_iter	2
sgd_minibatch_size	8192

Table 4.7: Adaptation training configuration

4.5.3 Results

LSTM policy

The LSTM policy has been trained for 200 epochs and the mean episode length, reward and learning rate progression can be seen in figure 4.36. The final training episode reward was 79.35.

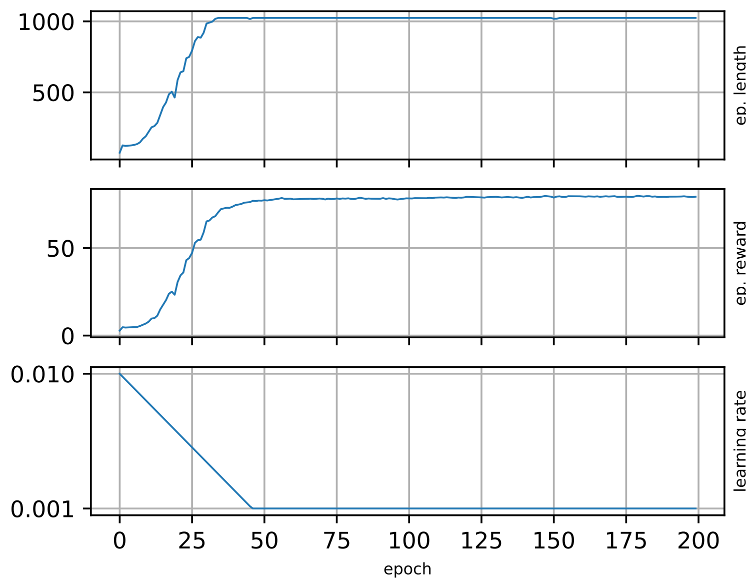


Figure 4.36: LSTM adaptation training

The figure 4.37 shows ten example trajectories beggining in initial states sampled using the training configuration around coordinates $[-1, 0, 10]$, with the setpoint position being $[0, 0, 10]$. The drone model parameters are also randomly generated using the training configuration for each quadcopter inside the simulation as discussed in 3.3.1. The figures 4.38 and 4.39 show the evolution of quadcopter coordinates and angles from the trajectories respectively.

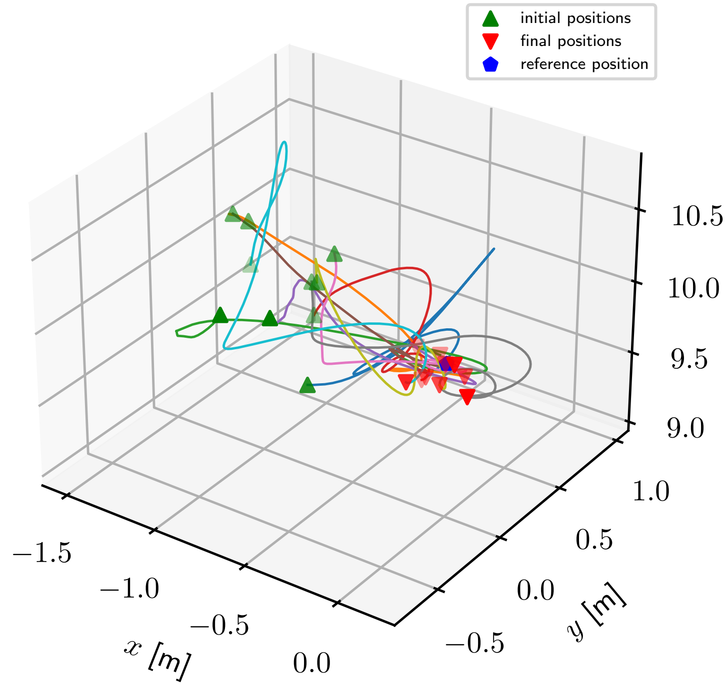


Figure 4.37: LSTM adaptation trajectories

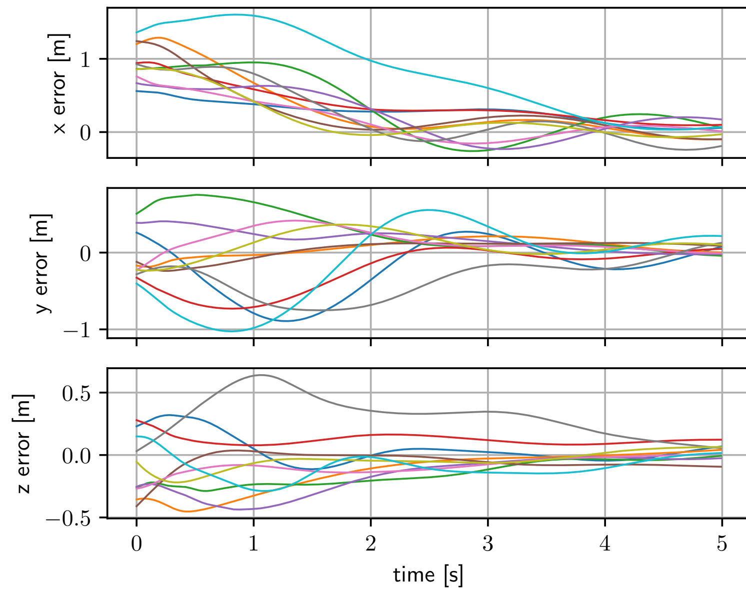


Figure 4.38: LSTM adaptation position errors

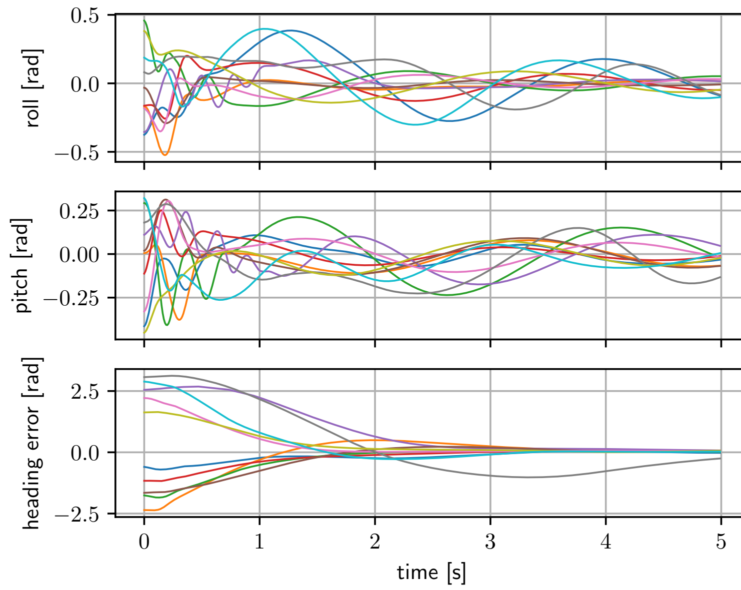


Figure 4.39: LSTM adaptation angles

RMA based model

The RMA policy was trained for 200 epochs and its training progress can be seen in the figure 4.40. The trained policy reached mean episode reward of 79.88. After the policy has been trained, its weights were fixed and the adaptation module training proceeded. The adaptation module was trained using MSE loss for 50 epochs and the corresponding training progress is shown in the figure 4.41.

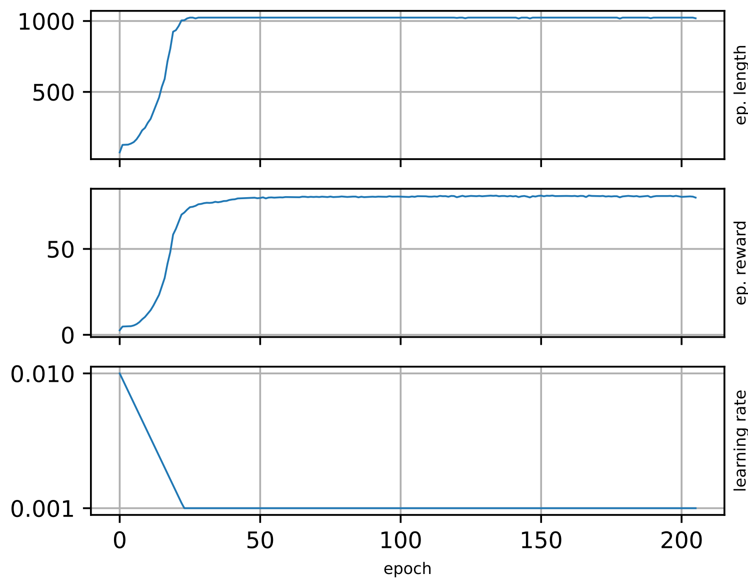


Figure 4.40: RMA policy training

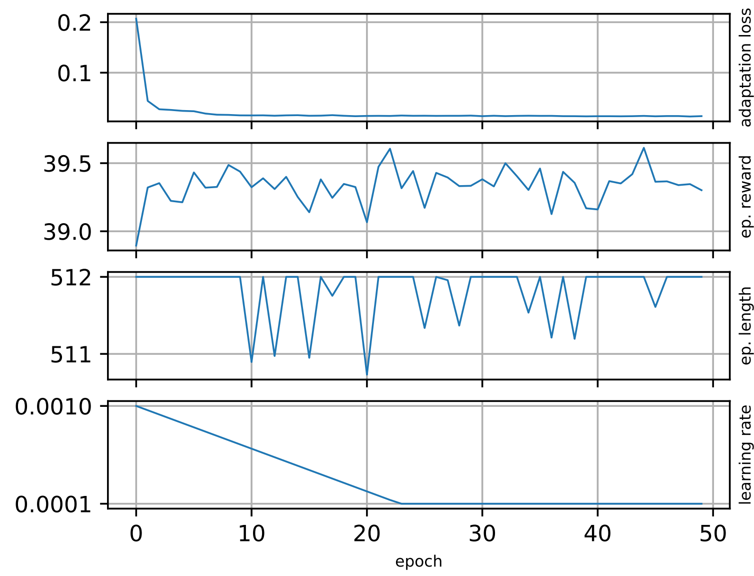


Figure 4.41: RMA adaptation module training

The figure 4.42 shows ten example trajectories beggining in initial states sampled in the same way as in 4.37. The figures 4.43 and 4.44 show the evolution of drone coordinate errors and angles from the trajectories respectively.

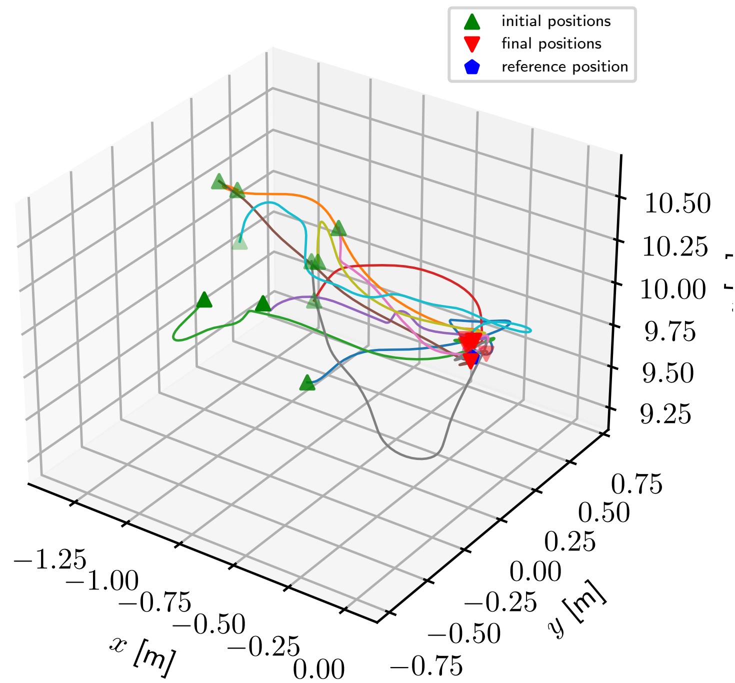


Figure 4.42: RMA trajectories

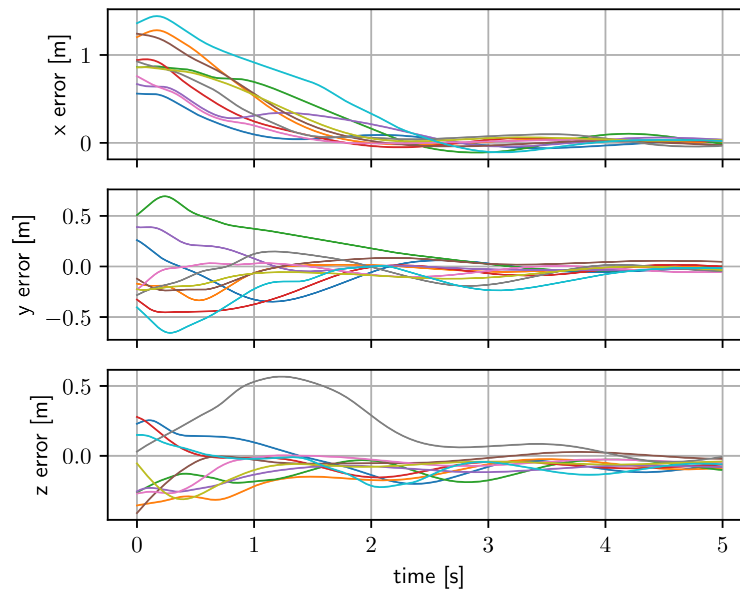


Figure 4.43: RMA position errors

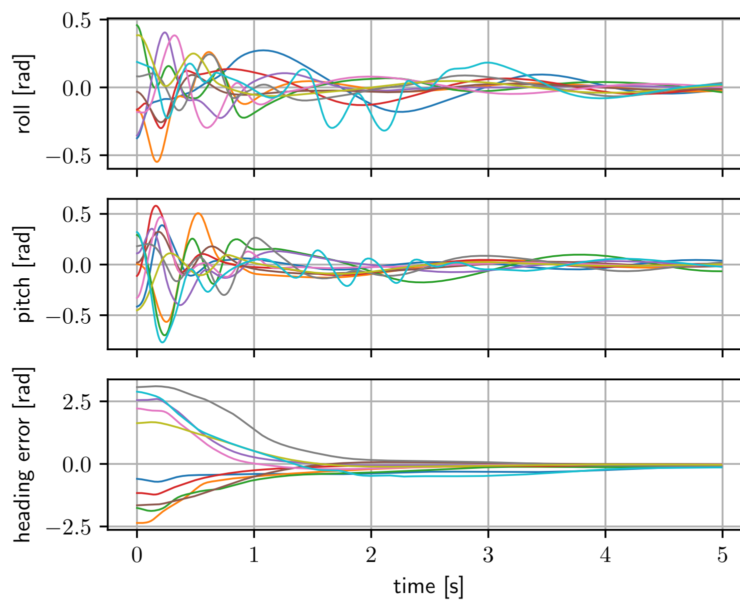


Figure 4.44: RMA angles

4.6 Discussion

4.6.1 Position control

The energy-based reward has shown to be more suitable for policy training as the obtained policy results 4.3.3 seem to be in better alignment with

what we want the policy to do when compared to the simpler distance based reward results 4.3.3. In particular, the position errors with the energy based reward 4.8 show better convergence that when using the distance reward 4.4. The trajectories 4.3 and angles 4.5 with distance reward also show that the quadcopter behaviour is more chaotic than when using the energy based reward 4.7, 4.9. This is due to the policy having better handling of the suspended load.

From the trajectory tracking experiment 4.3.3, the distance based reward seems to achieve smaller position error 4.10, 4.11 than the energy based reward. This comes however at a cost of more chaotic movements 4.12, which can be undesirable when handling the suspended load. This is why the energy-based reward is used in the rest of the experiments.

4.6.2 Pendulum state estimation

The results of training a policy with unknown pendulum state 4.4.3 have shown worse performance than the full state policy 4.3.3. This is evident from the lower training reward of 77.7 as compared to 78.74 and also the position errors, which are larger for the unknown pendulum state 4.18 as compared to 4.8. The training rewards achieved with the LSTM and CNN estimator networks are 78.61 and 78.47 respectively, which suggests that the performance of the policy is improved by including the state estimator network and is much closer to the full state policy. This can also be seen in the final position errors of the state estimator policies 4.23, 4.29 as compared to the policy without pendulum information 4.18. The state estimator networks show decent pendulum state estimation accuracy 4.25, 4.31. The performance of the angular velocity estimation seems better than that of pendulum angles.

The evaluation of the four policies 4.4 shows again, that the average reward improves when using the state estimator network as compared to the policy without full state information. The LSTM and CNN estimator networks show very similar performance achieving rewards 37.31 adn 37.29 on the evaluation respectively. This is still not as good as the full state policy, which achieved 37.45, but still much closer to its performance compared to partial state information policy, which achieved reward 36.62.

The state estimator networks are further compared in table 4.5. These results show that the LSTM estimator network has in general improved state estimation accuracy as compared to the CNN estimator network.

4.6.3 Model parameter adaptation

When it comes to controlling a quadcopter with varying model parameters, the LSTM policy trained in 4.5.3 showed poor performance when compared to the non-varying model policy 4.3.3. The final position errors 4.38 are much worse compared to 4.8. The explicit adaptation approach using the RMA model 4.5.3 performed much better in terms of the position errors 4.43. The RMA model does still not match the non-varying model performance 4.3.3 though, which is expected as this is a much harder task.

Chapter 5
Conclusion

5.1 Summary

In this thesis, the use of deep reinforcement learning techniques for positional control of a quadcopter with a hanging load was explored. Though there are many approaches in the field of reinforcement learning, this thesis focuses on employing gradient-based direct policy search technique known as proximal policy optimization (PPO). This technique works by directly optimizing a stochastic policy, which maps the current state (or observations) to a distribution over the action space. The policy is used to sample trajectories using a simulated model of the original system, which are then used to reinforce actions that lead to higher accumulated rewards.

The first step of the implementation included defining a simplified model of the quadcopter and utilizing the MuJoCo simulator for the physical simulation of the drone dynamics with a suspended load. The simulated quadcopter model is assembled from primitive geometrical shapes, four force actuators to simulate the propellers, and a pendulum connected via two perpendicular rotational joints. The implemented simulation environment enables spawning multiple drone models inside a single simulation instance with configurable model parameters. The training itself is performed using the RLlib library, which includes basic implementations of many popular reinforcement learning algorithms, such as PPO.

The implemented instruments are used to train policies tackling three tasks. In the first task, full state information is provided to a multi-layer perceptron based policy and the atainable performance is evaluated. The ability of the policy to stabilize and control a quadcopter is tested using two reward parametrizations. Training with both rewards lead to the policy being able to stabilize the drone successfully from various initial states as well as follow the reference setpoint state. The first reward is simply based on penalizing the distance from a reference state and the exerted effort. The second reward, based on penalizing the energy of the pendulum, has been found to handle the hanging load better than the distance-based reward and is used in the rest of the experiments.

In the second task, the policy has only access to partial state information which excludes the pendulum state. This naturally leads to decreased control

performance. This is solved by augmenting the policy with a state estimator network that is trained via supervised learning on the pendulum states. The two trained estimator networks based on the 1D convolution and long short-term memory architectures respectively are shown to estimate the pendulum states with decent accuracy, improving the control performance.

The third task then focuses on controlling quadcopters with varying model parameters such as pendulum weight, length or the maximum propeller thrust. This task is first tackled by a recurrent neural network policy trained with PPO, which exhibits weak control performance. The second approach relies on first training the policy with known model parameters and then training an adaptation module, that estimates the model parameter embedding using available trajectory data. This second approach is shown to be able to adapt to the unobserved model parameters far better than the simple recurrent neural network policy.

5.2 Future work

When dealing with reinforcement learning-based policies trained in simulated environments, a significant challenge arises due to the mismatch between the simulated and real systems. The used simulated model is by far not perfect as it lacks aerodynamic effects, assumes rigid bodies only and most notably, does not simulate sensor noise. This is intentional, as these fenomena would make training the control policy far more difficult. However, in order to be able to deploy the learned policies in the real world, these problems need to be adressed.

Appendix A

Bibliography

[1] Josh Achiam. Spinningup. `https://spinningup.openai.com/en/latest/spinningup/`, 2020. [Online; accessed 10-May-2023].

[2] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula, 2020.

[3] Aditya Deshpande, Rumit Kumar, Ali Minai, and Manish Kumar. Developmental reinforcement learning of control policy of a quadcopter uav with thrust vectoring rotors. 10 2020.

[4] Aditya M. Deshpande, Ali A. Minai, and Manish Kumar. Robust deep reinforcement learning for quadcopter control. *IFAC-PapersOnLine*, 54(20):90–95, 2021. Modeling, Estimation and Control Conference MECC 2021.

[5] Farama Foundation. Gymnasium. `https://github.com/Farama-Foundation/Gymnasium`, 2017. [Online; accessed 10-May-2023].

[6] Maria Eusebia Guerrero-Sánchez, Rogelio Lozano, Pedro Castillo Garcia, Omar Hernandez Gonzalez, Carlos Daniel Garcia Beltran, and Guillermo Valencia-Palomo. Nonlinear Control Strategies for a UAV Carrying a Load with Swing Attenuation. *Applied Mathematical Modelling*, 91:709–722, 2021.

[7] Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, oct 2017.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[9] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots, 2021.

[10] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning, 2018.

[11] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation, 2019.

[12] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[13] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[16] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[17] Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dmcontrol: Software and tasks for continuous control. *Software Impacts*, 6:100022, 2020.

Appendix B

Code structure

The attached code is organized as shown in figure B.1. The files in the environments directory include the definitions of the simulation, drone model, rewards, and observations used during training. The files in the models directory implement the PyTorch policy models. The README.md file contains brief instructions for installing required libraries and running the policy training in the simulation environment.

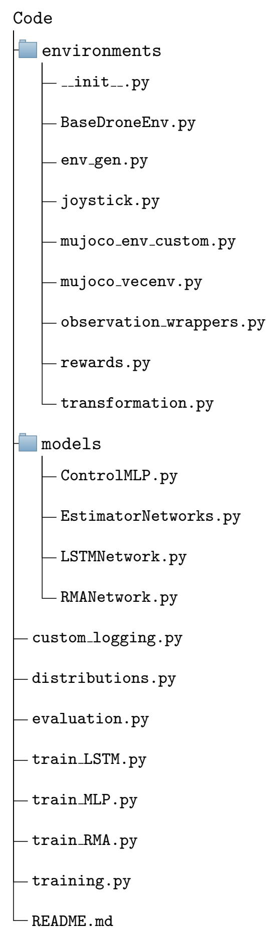


Figure B.1: Code structure