Studienarbeit STUD-512

Deep Reinforcement Learning-based Control for the Quadcopter with Model Indications

von
Pala Ahmed

Betreuer: Prof. Dr.-Ing. Prof. E.h. P. Eberhard
Wei Luo, M.Sc.

Universität Stuttgart
Institut für Technische und Numerische Mechanik
Prof. Dr.-Ing. Prof. E.h. P. Eberhard

Mai 2021

Contents

Chapter 1

Introduction

The usage of automated robots in the industry has risen drastically in the last decades, where robots can solve complex tasks in static and dynamic environments. Unmanned aerial vehicles (UAVs) have thereby gained more and more attention due to their broad applicability and agility. A typical form of an UAV is the quadcopter, a multicopter with four equally distanced rotors, which have also grown significant popularity in scientific studies. Recent popular research topics, for instance, deal with the cooperated application of multiple drones [CarneyEtAl21] or with the utilization of machine learning methods for navigation and obstacle avoidance [BlukisEtAl18].

For applying the UAV, one of the critical challenges is to find a proper control strategy to ensure a safe, responsive flight. Currently, there are numerous methods for controlling a quadcopter, ranging from relatively simple methods like PID controllers with linearization assumptions, more advanced control methods like model-predictive controllers (MPCs), and up to machine learning based approaches. In general, they can be grouped into three categories: linear control methods, nonlinear control methods, and intelligent control strategies [KimGadsdenWilkerson20]. This work focuses on the third category, on subjects in the application of machine learning-based methods on the control of quadcopters.

A powerful discipline in the field of machine learning is deep reinforcement learning, which combines methods of traditional reinforcement learning with deep learning methods using neural networks. It has been successfully applied in a wide range of applications where the deep reinforcement learning agent is able to learn a strategy through interaction with its environment and makes decisions based on its gained experience. The strength in deep reinforcement learning lies in the capability to learn a control strategy solely through raw input data and in the wide applicability. Recent impressive results have shown that a deep re-

inforcement learning agent can even surpass a professional human player in a strategy board game, where the search space for the best next move is too numerous to be determined through exhaustive search in a reasonable time window [SilverEtAl16].

Although many deep reinforcement learning approaches have often been first tested on game-like tasks, they have proven to be as well adaptable for controlling robots. For instance, agents have been trained on deep reinforcement learning to control a robot arm in the simulation on pick-and-place tasks and have been successfully transitioned to the real robot [FranceschettiEtAl20]. Related approaches have trained an agent to fully control the motion of the robot with a high degree of freedom, which also showed impressive results on the real robot [HwangboEtAl19]. In the application on quadcopters, deep reinforcement learning has been applied on high-level control tasks, e.g. to avoid obstacles and change the trajectory [BlukisEtAl18], and also to the lower-level direct control of the actuators [HwangboEtAl17]. Due to the fact that the number of actuators of the quadcopter is smaller than the number of the degrees of freedom, traditional control strategies are not directly applicable, and thus make the design of a robust controller more complex. Deep reinforcement learning-based controllers, in contrast, do not require a pre-defined controller structure. The control strategy is generally learned through approximation, where the neural network maps the sensory inputs to specific control outputs.

In this work, such a deep reinforcement learning-based controller is designed for a specific quadcopter. Hereby, the controller is supposed to control the quadcopter by directly controlling the four actuators. The training is resolved in two stages, where the agent is trained on two differently designed tasks. In the first stage, the agent is trained based on an imitation learning method to control and fly the quadcopter to a set target point. The agent hereby learns basic control behaviors from an interactive expert, which is resolved through a model-based controller. In the second training stage, the control strategy of the agent gets further improved in a training setup based on a policy-gradient method, where the agent is trained to compensate for relatively large disturbances and enhance its overall robustness.

The following chapters describe how the stated problems are addressed in this work. Chapter 2 introduces the theoretical background of classical reinforcement learning and related topics. Thereafter, Chapter 3 discusses the implementation of the deep reinforcement methods and the setup of the agent and the expert. Chapter 4 introduces the simulation framework and describes the two-staged training task. The results of the training are also presented and the performance of the agent is validated in different scenarios, where the agent is compared with the expert. Finally, Chapter 5 provides a brief reflection of the results.

Chapter 2

Fundamentals

This chapter discusses the theoretical background for further understanding of the subject of the thesis. Section 2.1 starts with the description of the dynamic model of the quadcopter and proposes a control method for small-angle conditions. Thereafter, the basic ideas of reinforcement learning are introduced in Section 2.2 containing an example for a tabular method for solving parts of the problem. Section 2.3 gives an insight into neural networks and their learning process.

2.1 Quadcopter

The quadcopter, also called quadrotor or drone, is a typical design for a Unmanned Aerial Vehicle (UAV). It is propelled by two oppositely rotating pairs of rotors, which are placed in equal distance to the center of mass, as shown in Fig. 2.1. By having four actuators and six degrees of freedom, it represents an under-actuated system, which increases the complexity in controller designs [EmranNajjaran18]. Therefore, the quadcopter is often assumed to operate near the hovering state with small roll, pitch, and yaw angles, which allows taking advantage of linearization assumptions for the controller design [PowersMellingerKumar15].

2.1.1 Dynamic Model

A convenient way to model the kinematics of a quadcopter is provided by using $Z - X - Y$ Euler angles and two reference frames [PowersMellingerKumar15]. Figure 2.1 presents the coordinate systems of the world frame and the quadcopter body frame as a rigid body system. The initial frame I is defined by the axis x, y and z, with z pointing upwards. The body frame B is attached to the center of

mass with x_B pointing in the forward direction and z_B being parallel to z, when the quadcopter is in an ideal hovering state.

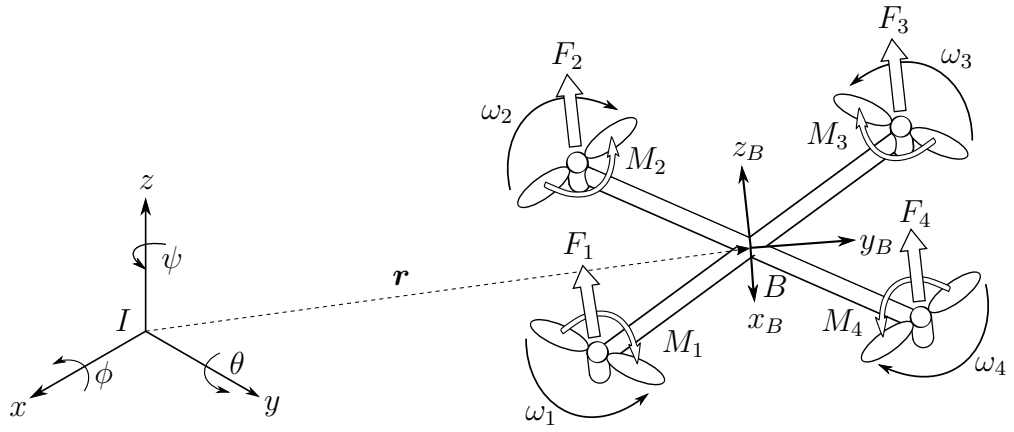


Figure 2.1: The schematic and coordinate system of a quadcopter. The quadcopter consists of four equidistant rotors, each two rotating with ω_i in the opposite direction. The body frame is located at the center of mass with x_B pointing at the front and z_B pointing upwards in an ideal hovering state. This graphic is modified from the source [Luukkonen11].

To describe the frame B, a series of rotations is required. The associated rotation matrix is given by

$$ {}^{I}\boldsymbol{R}^{B} = \begin{bmatrix} c_\psi c_\theta - s_\phi s_\psi s_\theta & -c_\phi s_\psi & c_\psi s_\theta + c_\theta s_\phi s_\psi \\ c_\theta s_\psi + c_\psi s_\phi s_\theta & c_\phi c_\psi & s_\psi s_\theta - c_\psi c_\theta s_\phi \\ -c_\phi s_\theta & s_\phi & c_\phi c_\theta \end{bmatrix}, \tag{2.1} $$

where $s_{(\cdot)}$ and $c_{(\cdot)}$ denote $\sin(\cdot)$ and $\cos(\cdot)$, respectively.

The components of the angular velocity in the body frame are denoted by ω_{x_B}, ω_{y_B}, and ω_{z_B} and depend directly on the derivatives of the roll, pitch, and yaw angles by the following relation [Mellinger12]:

$$ \begin{bmatrix} \omega_{x_B} \\ \omega_{y_B} \\ \omega_{z_B} \end{bmatrix} = \begin{bmatrix} c_\theta & 0 & -c_\phi s_\theta \\ 0 & 1 & s_\phi \\ s_\theta & 0 & c_\phi s_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}. \tag{2.2} $$

The position vector from frame I to frame B is denoted by $\boldsymbol{r}$. The acting forces on the system are the gravitational force in negative z-direction and the thrusts F_i from each rotor in the positive z_B-direction. The motion equations for the center mass are derived from the Newton-Euler equations, given by

$$ m \cdot \ddot{\boldsymbol{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + {}^{I}\boldsymbol{R}^{B} \begin{bmatrix} 0 \\ 0 \\ \sum_i F_i \end{bmatrix}, \tag{2.3} $$

with

$$F_i = k_F \omega_i^2, \;\; i = 1, 2, 3, 4. \tag{2.4}$$

The forces F_i are modeling the perpendicular thrusts at each rotor, where the thrust coefficient k_F can be determined by experimentation [PowersMellingerKumar15]. Furthermore, the rotors produce a moment given by

$$M_i = k_M \omega_i^2, \;\; i = 1, 2, 3, 4, \tag{2.5}$$

which, in total, are mutualy canceled by the opposite rotating rotor pairs at equal rotor speeds. The moment coefficient k_M can also be determined by experimentation.

Analogously to Eq. 2.3, the motion equations for the angular accelerations are given by:

$$\boldsymbol{I} \begin{bmatrix} \dot{\omega}_{x_B} \\ \dot{\omega}_{y_B} \\ \dot{\omega}_{z_B} \end{bmatrix} = \begin{bmatrix} \frac{L}{\sqrt{2}}(F_2 - F_4) \\ \frac{L}{\sqrt{2}}(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} \omega_{x_B} \\ \omega_{y_B} \\ \omega_{z_B} \end{bmatrix} \times \boldsymbol{I} \begin{bmatrix} \omega_{x_B} \\ \omega_{y_B} \\ \omega_{z_B} \end{bmatrix}, \tag{2.6}$$

where $\boldsymbol{I}$ denotes the inertia matrix and L represents the distance of two opposing rotors. $\boldsymbol{I}$ is assumed to be a diagonal matrix [Mellinger12].

2.1.2 Control for Small-Angle Conditions

By assuming small angles, the linearization of the motion equations is taken into the controller design. It allows decoupling the attitude controller and position controller of the quadcopter, as proposed by [Mellinger12]. Thereby, a nested control structure is designed, as illustrated in Fig. 2.2. A benefit of this design is that the attitude controller, which is responsible for controlling the roll, pitch, and yaw angles, can be run at a higher frequency to enhance the stability of the system [Mellinger12].

The quadcopter is directly controlled through the rotor speeds ω_i, with $i = 1, 2, 3, 4$, and the thereby generated forces and moments according to Eqs. 2.4 and 2.5, respectively. The desired control force $u_{1,\text{des}}$ and the control moments $u_{2,\text{des}}, u_{3,\text{des}}$ and $u_{4,\text{des}}$, are given by

$$\boldsymbol{u}_{\text{des}} = \begin{bmatrix} u_{1,\text{des}} \\ u_{2,\text{des}} \\ u_{3,\text{des}} \\ u_{4,\text{des}} \end{bmatrix} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & k_F \frac{L}{\sqrt{2}} & 0 & -k_F \frac{L}{\sqrt{2}} \\ -k_F \frac{L}{\sqrt{2}} & 0 & k_F \frac{L}{\sqrt{2}} & 0 \\ k_M & -k_M & k_M & -k_M \end{bmatrix} \begin{bmatrix} \omega_{1,\text{des}}^2 \\ \omega_{2,\text{des}}^2 \\ \omega_{3,\text{des}}^2 \\ \omega_{4,\text{des}}^2 \end{bmatrix}. \tag{2.7}$$

This formulation of the control vector $\boldsymbol{u}_{\text{des}}$ is derived from Eq. 2.6 and the assumption that the products of the inertia components are small, as well as the

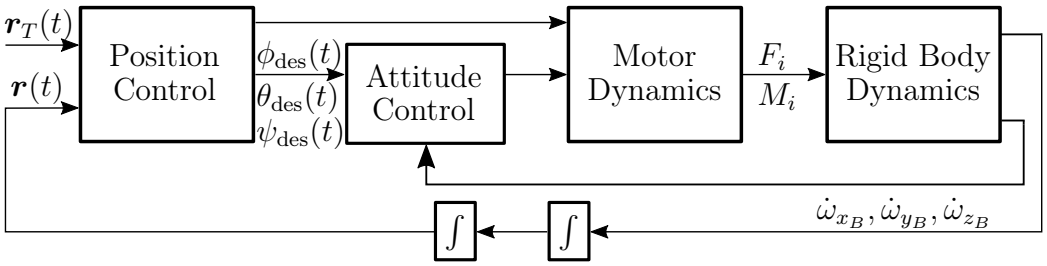


Figure 2.2: A nested PID-control loop and the signal pipeline. The position controller evaluates the position error and provides the input for the attitude controller [Mellinger12].

component of the angular velocity of the yaw rotation ω_{z_B}, which leads to the following control rule for the attitude controller [Mellinger12]:

$$\begin{aligned} u_{2,\text{des}} &= k_{p,\phi}(\phi_{\text{des}} - \phi) + k_{d,\phi}(\omega_{x_B,\text{des}} - \omega_{x_B}) \\ u_{3,\text{des}} &= k_{p,\theta}(\theta_{\text{des}} - \theta) + k_{d,\theta}(\omega_{y_B,\text{des}} - \omega_{y_B}) \\ u_{4,des} &= k_{p,\psi}(\psi_{\text{des}} - \psi) + k_{d,\psi}(\omega_{z_B,\text{des}} - \omega_{z_B}). \end{aligned} \tag{2.8}$$

The gain vectors $k_{p,(\cdot)}$ and $k_{d,(\cdot)}$ denote the proportional and derivative gains for the respective controlled angles.

For the position controller, the pitch and roll angles are used to control the x and y position, u_4 to control the yaw angle and u_1 to control along the z-direction [Mellinger12]. The desired accelerations $\ddot{r}_{i,\text{des}}$ are calculated from the position error $e_j = r_{j,T} - r_j$, $j = 1, 2, 3$, with

$$\ddot{r}_{j,\text{des}} = \ddot{r}_{j,T} + k_{d,j}(\dot{r}_{j,T} - \dot{r}_j) + k_{p,j}(r_{j,T} - r_j) + k_{i,j} \int (r_{j,T} - r_j), \tag{2.9}$$

where $\boldsymbol{r}_T$ indicates the pursued trajectory and $k_{i,(\cdot)}$ denote the respective integral gains of the PID feedback. In the hovering state, it is obvious that $\dot{\boldsymbol{r}}_T = \ddot{\boldsymbol{r}}_T = \mathbf{0}$. The linearization and invertation of Eq. 2.9, as described by [Mellinger12], leads to the following relation for the desired roll and pitch angles for the attitude controller, as well as $u_{1,des}$:

$$\begin{aligned} \phi_{\text{des}} &= 1/g(\ddot{r}_{1,\text{des}} \sin\psi_T - \ddot{r}_{2,\text{des}} \cos\psi_T) \\ \theta_{\text{des}} &= 1/g(\ddot{r}_{1,\text{des}} \cos\psi_T + \ddot{r}_{2,\text{des}} \sin\psi_T) \\ \ddot{u}_{1,\text{des}} &= m \cdot \ddot{r}_{3,\text{des}}. \end{aligned} \tag{2.10}$$

Hereby, ψ_T denotes the tracked yaw angle with $\psi_T = \psi_0$ at hovering state.

2.2 Reinforcement Learning

Reinforcement learning is a machine learning method where a so-called agent primarily learns by interacting with its environment. In general, the agent is an instance that makes decisions based on its observations and its experience. Unlike other types of machine learning, the agent has to explore the environment to generate its training data and learns by trial-and-error.

In a given state $\boldsymbol{s}_t \in \mathcal{S}$ at each time step $t = 0, 1, 2, ...$, the agent acts according to its previous experience with one possible action $\boldsymbol{a}_t \in \mathcal{A}(\boldsymbol{s})$. As a consequence, the agent transitions into the state $\boldsymbol{s}_{t+1}$ where it receives a reward signal $r_{t+1} \in \mathcal{R}$ from the environment for being in this particular state. The agent itself is not told what action to take but rather learns the quality of the taken actions by the received reward signals. Accordingly, the agent tries to learn the best possible action in a given state to maximize the rewards, classifying it as an optimal control problem. The idea of a learning agent interacting with the environment to figure out the state-action relation is the basic idea of reinforcement learning and is illustrated in Fig. 2.3.

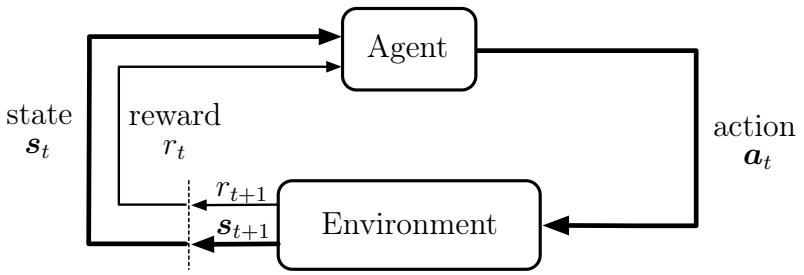


Figure 2.3: An illustration of the agent-environment interaction. At each time step t, the agent selects a possible action $\boldsymbol{a}_t \in \mathcal{A}(\boldsymbol{s})$ for the current state representation $\boldsymbol{s}_t \in \mathcal{S}$. As a result, the agent transitions into state $\boldsymbol{s}_{t+1}$ and the environment feeds back an associated reward signal r_{t+1} [SuttonBarto18].

2.2.1 Markov Decision Process

The Markov Decision Process is the mathematical idealized formulation of the reinforcement learning problem [SuttonBarto18]. It is a sequential decision making problem where the behaviour of the environment and the current state at time step t does not depend on the history of actions $\boldsymbol{a}_0, \boldsymbol{a}_1, ..., \boldsymbol{a}_{t-1} \in \mathcal{A}(\boldsymbol{s})$ taken before by the agent, according to the Markov Assumption. It consists of a five-tuple $\langle \mathcal{S}, \mathcal{A}, R, P, \rho_0 \rangle$ with:

- $\mathcal{S}$ being the set of possible states,
- $\mathcal{A}$ being the set of possible actions with $\mathcal{A}(\boldsymbol{s})$ being the available actions in state $\boldsymbol{s}$,
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ being the reward function with $r_{t+1} = R(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})$,
- $P : \mathcal{S} \times \mathcal{A} \to [0, 1]$ being the transition probability, e.g., $P(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ of transitioning into state $\boldsymbol{s}' \in \mathcal{S}$ for a given state $\boldsymbol{s}$ and an action $\boldsymbol{a}$,
- and $\boldsymbol{\rho}_0$ being the distribution of the initial state.

The Markov Decision Process is considered finite, when $\mathcal{S}$ and $\mathcal{A}$ have a finite amount of elements. Figure 2.4 illustrates the sequence of a Markov Decision Process.

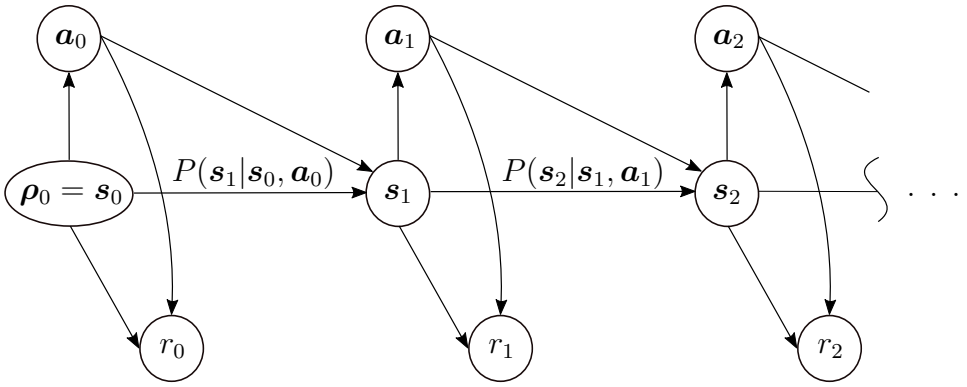


Figure 2.4: The Markov Decision Process illustrated as a graph with nodes and edges. From the initial state $\boldsymbol{s}_0$, which has the probability distribution $\boldsymbol{\rho}_0$, the action $\boldsymbol{a}_0$ is taken and the reward r_0 is received for being in this state. Through the action $\boldsymbol{a}_0$, the state $\boldsymbol{s}_0$ transitions into state $\boldsymbol{s}_1$ with the probability of $P(\boldsymbol{s}_1|\boldsymbol{s}_0, \boldsymbol{a}_0)$. This process continues finitely or infinitely, depending on the number of states and actions or, respectively, the time horizon.

2.2.2 Policy

The policy of the agent is a rule to decide which action to take given a certain state $\boldsymbol{s}_t$. Hence, optimizing the policy is equal to optimizing the agent's behavior.

In general, a policy can be either deterministic or stochastic. In the deterministic case, the policy denoted by $\pi(\boldsymbol{s}_t)$ directly maps a specific state $\boldsymbol{s}_t \in \mathcal{S}$ to one

specific action $\boldsymbol{a}_t \in \mathcal{A}(\boldsymbol{s})$. On the contrary, a stochastic policy $\pi(\cdot|\boldsymbol{s}_t)$ models a probability distribution over all possible actions $\boldsymbol{a}_t$. The action is then sampled from that probability distribution $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$. In this work, a stochastic policy with a Gaussian distribution is utilized due to an infinite amount of state-action combinations, which will be further discussed in Chapter 3.

Besides, policies are usually formulated with computable functions that depend on a set of parameters $\boldsymbol{\theta}$ that can be adjusted to optimize policy. Thus, stochastic policies are usually denoted by $\pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s}_t)$. In deep reinforcement learning, which will be discussed in Section 3.1, those functions are usually approximated with neural networks.

2.2.3 Reward and Return

The agent's goal in reinforcement learning is to maximize the cumulated rewards over a trajectory $\tau = (\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, \boldsymbol{a}_1, ...)$ instead of the immediate higher reward. Similar to other optimization approaches, greedily taking the higher immediate reward could lead to a suboptimal overall result. In the context of reinforcement learning, a trajectory describes the sequence of visited states and taken actions by an agent.

Formally, the objective to be maximized is the expected return, where the return G_t is a function of the reward sequence over a trajectory $\tau(\boldsymbol{s}, \boldsymbol{a})$. Dense reward functions, which continuously feedback a reward signal depending on the specified metric, are typically set negative. Sparse rewards, on the contrary, are either positive or negative and are only received when a specific condition is reached. Furthermore, the reward for the terminal state is always $r_T = 0$.

The simplest expression of the expected return G_t is the sum of the future rewards $\sum_{k=0}^{T} r_{t+k+1}$. Since a reinforcement learning task can be an ongoing process with large or upto infinitely state-action sequences with $T = \infty$, a discount factor $\gamma \in (0, 1)$ is considered to ensure the convergence of G_t. Thereby, the expected return is given by

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.11}$$

2.2.4 Value and Advantage Functions

In order to evaluate a certain state or a certain state-action pair, the value function $v_\pi(\boldsymbol{s})$ is introduced. It is a prediction of the expected discounted return of a

state-action pair for a policy $\pi(\cdot|\boldsymbol{s})$. The value funcion $v_\pi(\boldsymbol{s})$ is defined by

$$v_\pi(\boldsymbol{s}) = \mathbb{E}_\pi[G_t|\boldsymbol{s}_t = \boldsymbol{s}] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|\boldsymbol{s}_t = \boldsymbol{s}\right], \text{ for all } \boldsymbol{s} \in \mathcal{S}. \tag{2.12}$$

Equation 2.12 can also be formulated recursively by the Bellman Equation:

$$\begin{aligned} v_\pi(\boldsymbol{s}) &= \mathbb{E}_\pi[G_t|\boldsymbol{s}_t = \boldsymbol{s}] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma^k G_{t+1}|\boldsymbol{s}_t = \boldsymbol{s}] \\ &= \sum_a \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'}\sum_r P(\boldsymbol{s}', r|\boldsymbol{s}, \boldsymbol{a})\big[r + \gamma\mathbb{E}_\pi[G_{t+1}|\boldsymbol{s}_{t+1} = \boldsymbol{s}']\big] \\ &= \sum_a \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}', r} P(\boldsymbol{s}', r|\boldsymbol{s}, \boldsymbol{a})\big[r + \gamma v_\pi(\boldsymbol{s}')\big] \end{aligned} \tag{2.13}$$

Hereby, the value of state $\boldsymbol{s}$ only depends on its successor state $\boldsymbol{s}'$ and the transition probability $P(\boldsymbol{s}', r|\boldsymbol{s}, \boldsymbol{a})$. This relationship simplifies the computation, approximation, and learning of $v_\pi(\boldsymbol{s})$ and is applied in many reinforcement learning methods.

To evaluate the selected action $\boldsymbol{a}$, the action-value function $Q_\pi(\boldsymbol{s}, \boldsymbol{a})$ is introduced given the state $\boldsymbol{s}$ and the policy $\pi(\boldsymbol{a}|\boldsymbol{s})$, which is defined by

$$\begin{aligned} Q_\pi(\boldsymbol{a}, \boldsymbol{s}) &= \mathbb{E}_\pi[G_t|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}\right], \\ &\quad \text{for all } \boldsymbol{s} \in \mathcal{S} \text{ and } \boldsymbol{a} \in \mathcal{A}(\boldsymbol{s}). \end{aligned} \tag{2.14}$$

An optimal policy denoted by π^* will maximize the expected return. A policy π is considered better or equal to a policy π', when the value function $v_\pi(\boldsymbol{s}) \geq v_{\pi'}(\boldsymbol{s})$. It implies that the policy with the highest expected return is consequently the optimal policy with the highest action-value. Therefore, the optimal value function $v^*_\pi(\boldsymbol{s})$ and the optimal action-value function $Q^*_\pi(\boldsymbol{s}, \boldsymbol{a})$ are given by

$$v^*(\boldsymbol{s}) = \max_\pi v_\pi(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q^*(\boldsymbol{a}, \boldsymbol{s}) \tag{2.15}$$

and

$$\begin{aligned} Q^*(\boldsymbol{a}, \boldsymbol{s}) &= \max_\pi Q_\pi(\boldsymbol{a}, \boldsymbol{s}) \\ &= \mathbb{E}_{\pi^*}[r_{t+1} + \gamma v^*(\boldsymbol{s}')|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a}]. \end{aligned} \tag{2.16}$$

Hwoever, it is not always necessary to describe how good an action is in an absolute sense. It is sufficient to know how much better an action is over the other actions on average. The advantage function $A_\pi(\boldsymbol{s}, \boldsymbol{a})$ in Eq. 2.17 describes this condition for a specific action $\boldsymbol{a}$ in state $\boldsymbol{s}$ over a randomly selected action of $\pi(\cdot|\boldsymbol{s})$:

$$A_\pi(\boldsymbol{s}, \boldsymbol{a}) = Q_\pi(\boldsymbol{s}, \boldsymbol{a}) - v_\pi(\boldsymbol{s}) \tag{2.17}$$

2.2.5 Temporal-Difference Learning

Classic reinforcement learning problems are solved with tabular methods like dynamic programming, Monte Carlo methods, and temporal-difference (TD) learning [SuttonBarto18]. They refer to problems where action and state spaces are small enough to be represented in a tabular setting. Also, they aim to find a solution for the Bellman Equation in Eq. 2.13 iteratively. TD learning, for instance, fuses ideas of dynamic programming and Monte Carlo methods. It comes to work in many deep reinforcement learning methods and thus will be discussed here in more detail.

TD usually is used to approximate the value function. Thereby, it does not need a model of the environment to predict the next values and instead learns from raw experience. In continuous tasks, the actual expected return is not available online since future rewards are not available at the current time step. Thus, the value function is updated through the TD error considering the immediate reward with

$$v(\boldsymbol{s}) \leftarrow v(\boldsymbol{s}) + \alpha[r_{t+1} + \gamma v(\boldsymbol{s}') - v(\boldsymbol{s})], \tag{2.18}$$

where α denotes the learning rate and γ the discount factor. The target for the TD update is $r_{t+1} + \gamma v(\boldsymbol{s}')$, whereas $r_{t+1} + \gamma v(\boldsymbol{s}') - v(\boldsymbol{s})$ is named the TD error. This special case is called a one-step TD, or TD(0), of the more general TD(λ). Algorithm 1 in Appendix A.1 presents the pseudo-code for the tabular TD(0) learning. A significant advantage of this method is that the value can be updated online and does not require a finished episode.

2.3 Neural Networks

Neural networks are inspired by neurons and neuronal structures in animal brains. The basic idea comes from the work of Hubel and Wiesel, whose experiments showed that neurons are organized in hierarchical layers of cells [HubelWiesel59]. Building on this idea, one of the first mathematical models of a neuronal network was formulated by Fukushima, named the Neocognitron [Fukushima80]. Recent algorithmic innovations, the exponential growth in computational power, and large amounts of labeled data-sets enabled the success of today's neural networks, like the classification approach with the ImageNet data-set in 2012 [KrizhevskySutskeverHinton12].

The mathematical model is inspired by the biological neuron. The input of a neuron is the sum of the weighted signals, $\sum_i w_i x_i$ and the bias b_i, coming from the connected preceding neurons, where w_i denotes the weight and x_i denotes the signal output of the i-th preceding neuron. An activation function models the

excitation of the neuron when a certain threshold is reached, as further described in more detail in Section 2.3.2. The idea behind the learning mechanism is that the weights of the neurons are learnable, as is the case with biological neurons [CS231n21].

2.3.1 Multiple-Layer Feedforward Networks

State-of-the-art neural networks have a rich diversity in shapes and architectures. In the context of this thesis, the multiple-layer feedforward architecture is the most relevant, also called deep feedforward network or multiple-layer perceptron (MLP) in the literature. This kind of architecture suits as a class of universal approximators [HornikStinchcombeWhite89].

Figure 2.5 describes the structure of a feedforward network. It consists of one input layer, a variable number of hidden layers, and one output layer, all hierarchically and densely connected. The signals pass each layer sequentially from the input layer up to the output layer to map the desired output.

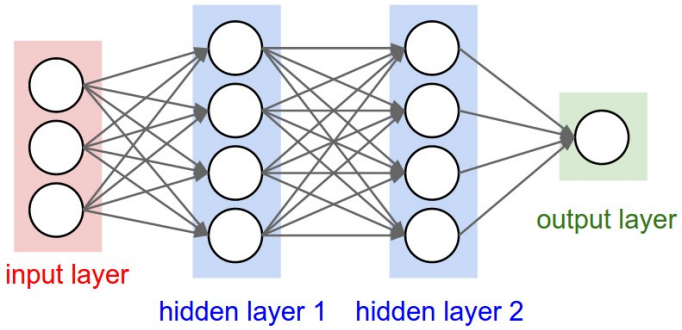


Figure 2.5: An abstract representation of a multiple-layer feedforward network. Each node represents a neuron, and each edge a one-directional connection from one neuron to the other. A multiple-layer feedforward network consists of one input layer, one output layer, and a variable amount of hidden layers [CS231n21].

The feedforward network approximates a specific function f^* by mapping a given input x into an output $y = f(x, \boldsymbol{\theta})$ and learning the parameters $\boldsymbol{\theta}$. In a neural network, the mapping function $f(\cdot, \boldsymbol{\theta})$ is a composition of several functions $f^{(i)}$, depending on the number of layers and connected neurons, defined by

$$f(\cdot, \boldsymbol{\theta}) = f^{(i)}(f^{(i-1)}(...f^{(2)}(f^{(1)}(\cdot, \theta^{(1)}), \theta^{(2)})...), \theta^{(i)}). \tag{2.19}$$

The parameter set $\boldsymbol{\theta}$ is typically a vector of all weights w_i and biases b_i of each neuron, which are inkrementally tuned and optimized until $f(x, \boldsymbol{\theta}) \approx f^*$ [Ma19, BruntonKutz19].

2.3.2 Activation Functions

The mentioned function $f(\cdot)$ is a certain activation function or transfer function. In general, the activation function performs a specific mathematical operation on an input. Common activation functions for feedforward networks in control tasks are the Tanh and the leaky ReLU function:

- The Tanh function is a hyperbolic tangens activation function with an output range of $[-1, 1]$. It is a zero-centered function that is saturated both in the negative and positive value range. The Tanh function is given by

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1. \tag{2.20}$$

- The Leaky ReLU function is a recitified linear unit (ReLU) function with a negative slope for negative inputs, defined by

$$LeakyReLU(x) = \begin{cases} f(x) = \alpha x, x < 0 \\ f(x) = x, x \geq 0 \end{cases}, \tag{2.21}$$

  with α usually being 0.01. Therefore, the function ranges from $-\infty$ to ∞. Compared with the normal ReLU function, the leaky ReLU function does not exterminate negative inputs immediately, which helps the model to the training data properly [Szandała21].

2.3.3 Learning Porcess

The learning process of a neural network can be generally split into four phases.

1. **Forward pass**. First, the input gets passed forward through the sequence of neurons in each layer. Thereby, the feedforward network estimates the desired output y^* with $y = f(x, \boldsymbol{\theta})$.

2. **Loss function calculation**. In the next step, the estimated output y is compared to the corresponding desired output y^* with a suitable method to measure the deviation. The loss function $L(\boldsymbol{\theta})$ highly depends on the task. Two common methods are:

   - the mean-squared error loss:

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i}^{n} (y_i - y_i^*)^2, \tag{2.22}$$

     where n is the number of training examples,

- and the logarithmic loss:

$$L(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i}^{n} \log p(y_i^*). \tag{2.23}$$

Here, $p(y_i)$ denotes the prediction probability of the desired output y_i^* over the probability distribution from which y_i is sampled [WangEtAl20].

3. **Backpropagation**. Through the evaluation of the loss function $L(\boldsymbol{\theta})$, the parameters of the neural network $\boldsymbol{\theta}$ can be adjusted to maximize the accuracy of future predictions. One efficient possibility is given with the Backpropagation Algorithm introduced by [RumelhartHintonWilliams86].

   The goal of the backpropagation is to measure how changing specific weights and biases of the neural network will affect the loss $L(\boldsymbol{\theta})$. Therefore, the global gradient $\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ is computed by recursed application of the chain rule with respect to the parameters $\boldsymbol{\theta}$ [CS231n21]. Figure 2.6 illustrates this process. The local gradient is computed and back-propagated to the preceding neuron through each layer and for each output neuron. Then, the local gradient is compound into the global gradient and used to update the network in the update step [GoodfellowBengioCourville16].

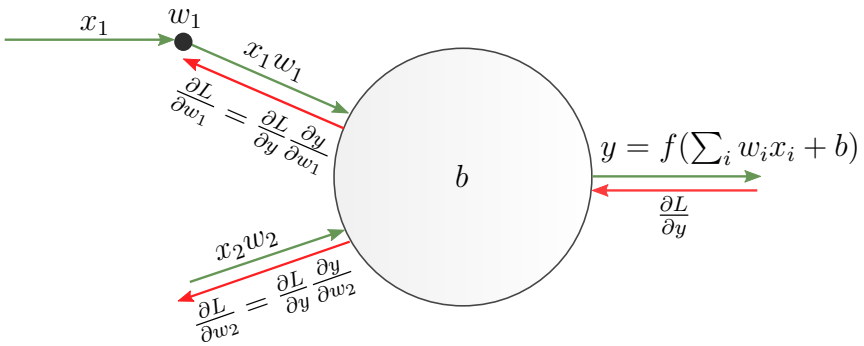


Figure 2.6: The forward pass (green) and the backpropagation (red) in a neuron. The neuron receives two inputs x_1 and x_2 and responds with the output y. The partial derivatives of an output y are computed through the reversed chain rule to determine the local gradients.

4. **Update**. In the final step, the weights and biases of the network are updated, typically through some form of gradient descent. The stochastic gradient descent, for example, basically performs one parameter update at

a time based on the global gradient of the loss function over one training example, given by

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, y_i y_i^*), \tag{2.24}$$

with $\alpha \in (0, 1]$ being the learning rate. The parameters $\boldsymbol{\theta}_{k+1}$ are the updated set of parameters of the neural network after one update at step k, with $k = 0, 1, 2, ..., K$.

In contrast, mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent by splitting the training data set into mini-batches, which can lead to more stable convergence through generalization [Ruder16].

2.3.4 Regularization

Despite their high capabilities, large-sized neural networks can be prone to overfitting. A model is overfitted when slight changes in the inputs have a high impact on the accuracy of the outputs. Hence, regulating the weights of the neurons is an effective method to generalize without decreasing the size of the network or increasing the amount of training data [Nielsen15].

One of these methods is the L2-regularization, which is also used in this work. It is probably the most common method to regulate the weights of a neural network. It uses the squared distance of every weight w_i and multiplies it by a strength λ. The L2-regularization loss is given by

$$L_{\mathrm{L2}}(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i}^{n} \lambda w_i^2, \tag{2.25}$$

and is added to the overall loss function as an additional term. It penalizes high weight values and causes the neural network to maintain small weights. The idea behind it is that small, but many inputs at each neuron are more desirable than large but fewer inputs [CS231n21].

Chapter 3

Methods and Setup

This chapter specifies which methods are used to address the deep reinforcement learning (DRL) problem and how they are implemented, based on the theoretical background of the preceding chapter. In this work, a DRL-based controller is trained in two stages with a hybrid approach to develop a control strategy for the quadcopter, as shown in Fig. 3.1. In the first training stage, the controller is trained based on imitation learning to imitate the control strategy of a model-based expert controller. Based on the resulting pre-trained neural network model, the DRL-based controller is then further trained and optimized in a more challenging task with a policy-gradient method to explore a more robust control strategy and enhance its performance and reliability.

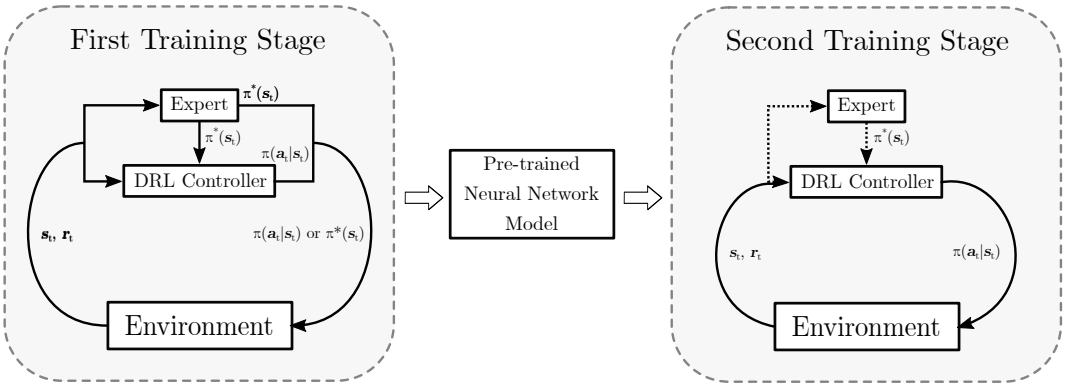


Figure 3.1: The macro-structure of the hybrid method in this work to train the DRL-based controller. In the first training stage, the DRL-controller is trained based on imitation learning to imitate the control strategy of the expert. Thereafter, the pre-trained model is used to further improve the DRL-based controller with a pure DRL method and additional action suggestions from the expert.

Section 3.1 introduces the DRL methods used for the described hybrid approach. For the first training stage with imitation learning, a model-based expert controller is set up, which is described in Section 3.2. Finally, Section 3.3 describes the setup of the trained DRL-based controller, which is referred to as the agent, since it is the standard term used in reinforcement learning. The agent is set up as a neural network receiving an observation from the environment and sending an action feedback. The neural network is then trained to minimize the learning objective, depending on the specific learning method.

3.1 Deep Reinforcement Learning

Reinforcement learning provides a robust framework on which an optimal control strategy can be investigated without directly requiring a robot model. In real-world reinforcement learning applications, state spaces can get exceptionally large and unmanageable. The number of possibilities at each time step would overrun the feasibility of tabular methods of classic reinforcement learning. Calculating the optimal policy with tabular methods would also exceed the current memory and hardware capabilities, and, most importantly, it would be disproportionately time-consuming. Therefore, deep reinforcement learning uses neural networks to approximate a real-time capable solution with neural networks as function approximators.

To address the problem statement of this thesis, two DRL methods are used in a hybrid approach to find an optimal control strategy: a policy-gradient method and an imitation learning method. Policy-gradient methods are a class of methods that aim to improve a policy by interacting with the environment and stating an objective that correlates with its performance. In DRL, the success of the training process strongly depends on the problem formulation. The agent is not trained on direct instructions on how to achieve the goal but on reward signals that indicate a desirable behavior. This leaves an abundance of options open with which a viable solution can be found. However, this indirect guidance can be detrimental to the success of the control task since an ill-stated DRL problem with too many degrees of freedom and too few restrictions could hinder the agent finding a solution at all. Therefore, in DRL, the success of an agent by learning through trial-and-error is not automatically guaranteed.

Since the quadcopter is an under-actuated system moving along the three-dimensional space, the control task is even more difficult for methods that solely rely on trial-and-error. The DRL agent might not find a proper control strategy if the task is formulated such that the rewards are too ambiguous to lead to the desired goal. To encourage stable and fast convergence, an agent is pre-trained with an imitation learning method in this work, where it learns a basic control

behavior from a model-based expert controller through instructions and interventions into the control task. Consequently, the agent tries to adopt the expert's behavior and approximate the explored states to suggested actions.

However, using imitation learning exclusively can lead to a relatively unstable control behavior since the trained agent might fail to generalize. Therefore, the agent is trained in a second stage based on the pre-trained neural network model with a policy-gradient method, where the expert still gives action suggestions that are beneficial in some occasions, but the agent explores and decides itself which action to take. Unlike imitation learning methods, policy-gradient methods generate a more robust control behavior but are much more sample inefficient. By combining both methods in a two-staged training setup, the benefits of both methods can be exploited and a robust control strategy faster and more efficiently developed. Training a DRL-based controller based on a pre-trained neural network model and combining pure DRL methods with imitation learning has proven to perform well in several DRL approaches [GoecksEtAl20, NagabandiEtAl17].

3.1.1 Policy-Gradient Methods

The primary aim of policy-gradient methods is to directly learn a parametrised policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$ that selects the actions without requiring a value function. Policy gradient methods use a performance measure $J(\boldsymbol{\theta})$ to measure the quality of the policy. The optimal parameters $\boldsymbol{\theta}^*$ are given when $J(\boldsymbol{\theta})$ is maximized. Therefore, policy-gradientt methods perform a gradient ascent in $J(\boldsymbol{\theta})$ instead of the gradient descent of Eq. 2.24, given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{3.1}$$

where α denotes the learning rate and $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ the policy-gradient.

In contrast, value-based methods like deep Q-networks (DQN) aim to learn the Q-function (action-value function, see Eq. 2.14) for each possible action [MnihEtAl15]. Therefore, the quality of the action selected by the policy is not estimated directly. Minor changes in the action-value function can lead to fluctuations in the policy during the learning process, whereas policy-gradient methods continuously and smoothly update the policy. Another advantage of policy-gradient methods over value-based methods is the fact that they can handle larger continuous state-spaces [SuttonBarto18]. Policy-gradient methods, however, are prone to get stuck in local minima and are sample inefficient since they are model-free.

The policy-gradient is the gradient of the expected return from trajectories sam-

pled by the policy. In the simplest form, it can be expressed by

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[G_t] \\ &= \int_{\tau} \nabla_{\boldsymbol{\theta}} P(\tau|\boldsymbol{\theta}) G_t \\ &= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) G_t \end{aligned} \tag{3.2}$$

with $\mathcal{D}$ being the set of state-action pairs. Most of the policy-gradient methods derive directly from this relation.

Equation 3.2 applies the log-likelihood of an action instead of the action probability because of the stochastic policy. Using the log-likelihood of an action probability instead of the actual probability has several computational advantages since likelihoods can become very small and close to zero. Thereby, the gradient of a specific transition probability can be alternatively expressed by

$$\nabla_{\boldsymbol{\theta}} P(\tau|\boldsymbol{\theta}) = P(\tau|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log P(\tau|\boldsymbol{\theta}). \tag{3.3}$$

Actor-Critic Architecture

Despite the general distinction between value-based and policy-gradient methods, actor-critic methods are policy-gradient methods that take the value function into account. An actor-critic architecture comprises an actor, which is basically the policy $\pi_{\boldsymbol{\theta}}$, and the critic, which estimates the on-policy value function v_{π}, as illustrated in Figure 3.2.

At each time step, the actor selects an action according to its parameters $\boldsymbol{\theta}$. Then, the critic evaluates the actions based on the reward of the environment, usually by a TD error (see Section 2.2.5). Both actor and critic will then be updated based on the critic's evaluation. Likewise, the critic also has its own set of parameters, usually implemented through a deep neural network.

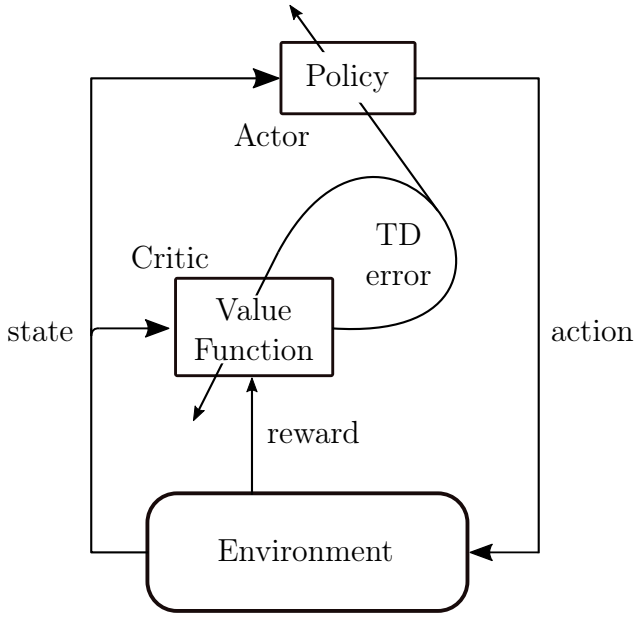


Figure 3.2: The actor-critic architecture. The actor (policy) selects an action that changes the state, and the critic (value function) evaluates the action through the TD error with the received reward and the new state. The resulting TD error is then used to update both actor, and critic [SuttonBarto18].

Proximal Policy Optimization

Proximal policy optimization is a popular state-of-the-art policy-gradient method with an actor-critic architecture. It builds on approaches of other algorithms in this class, most strongly on the concept of the trust-region and KL-divergence introduced in the trust-region policy optimization (TRPO) method [SchulmanEtAl15]. At the same time, PPO appears to be less complex in the implementation and the tuning, compared to TRPO and other algorithms like deep deterministic policy gradient [SilverEtAl14], and asynchronous methods [MnihEtAl16].

Many gradient descent methods rely on the line search method in order to find the optimal configuration. In line search, the steepest direction is picked and then followed by a fixed step size, which can lead to a suboptimal convergence behavior. Trust-region methods, instead, set a specific region with a maximum step size to ensure that the new policies are not too far away from the old policies. In PPO, the maximum step size is either regularized with a KL-divergence penalty or by clipping the loss function. In this thesis, the clipping method is used and further

discussed.

Like TRPO, PPO does not directly have a cost function that quantizes the performance of the policy. Instead, it tries to maximize a surrogate loss, which estimates the expected value for the new policy $\pi_{\boldsymbol{\theta}}$ by sampling from the old policy $\pi_{\boldsymbol{\theta}_{old}}$. The basic surrogate loss is given by

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau\ \pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{a}|\boldsymbol{s})}A(\boldsymbol{s},\boldsymbol{a})\right], \tag{3.4}$$

where $\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{a}|\boldsymbol{s})}$ denotes the ratio $r(\boldsymbol{\theta})$ of the new policy to the old policy, and $A(\boldsymbol{s},\boldsymbol{a})$ denotes the advantage of the taken action. Because the policy is stochastic, the log-likelihood is used alternatively to determine the policy ratio. Thereby, the policy ratio is instead defined by

$$r(\boldsymbol{\theta}) = \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}) - \log \pi_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{a}|\boldsymbol{s}). \tag{3.5}$$

Without constraints, this loss function leads to instability and divergence through too large policy-gradient steps. The PPO method provides a second method besides the standard KL-divergence constraint method to overcome this issue. Thereby, the surrogate objective is clipped by a factor ϵ, expressed by

$$J_{\text{PPO-clip}}(\boldsymbol{\theta}) = \min\left(\left(\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}) - \log \pi_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{a}|\boldsymbol{s})\right)\hat{A}(\boldsymbol{s},\boldsymbol{a}), g(\epsilon, \hat{A}(\boldsymbol{s},\boldsymbol{a}))\right), \tag{3.6}$$

where

$$g(\epsilon, \hat{A}(\boldsymbol{s},\boldsymbol{a})) = \begin{cases}(1+\epsilon)\hat{A}(\boldsymbol{s},\boldsymbol{a}), \hat{A}(\boldsymbol{s},\boldsymbol{a}) \geq 0\\ (1-\epsilon)\hat{A}(\boldsymbol{s},\boldsymbol{a}), \hat{A}(\boldsymbol{s},\boldsymbol{a}) < 0\end{cases} \tag{3.7}$$

denotes the clipping function for the upper and lower limit and $\hat{A}(\boldsymbol{s},\boldsymbol{a})$ denotes the advantage function estimate. This surrogate objective is computed and averaged over all time steps t. The advantage function $\hat{A}(\boldsymbol{s},\boldsymbol{a})$ is estimated by the critic. The critic, in turn, has to be trained to estimate the correct value function on which the advantage function is computed. Therefore, a clipped value loss is additionally given to the overall loss function to train the critic, given by

$$\begin{aligned} J_{v_{\boldsymbol{\pi}}\text{-clip}}(\boldsymbol{\phi}) = \max\Bigg(&\left\|v_{\boldsymbol{\pi}}(\boldsymbol{s}) - \hat{G}\right\|^2,\\ &\left\|v_{\boldsymbol{\pi},\text{old}}(\boldsymbol{s}) + \text{clip}\Big(v_{\boldsymbol{\pi}}(\boldsymbol{s}) - v_{\boldsymbol{\pi},\text{old}}(\boldsymbol{s}), -\epsilon, \epsilon\Big) - \hat{G}\right\|^2\Bigg), \end{aligned} \tag{3.8}$$

where $\hat{G}$ denotes an estimate of the expected return. In the standard implementation of the PPO method, both the return and the advantage function are estimated with the Generalized Advantage Estimation method, explained in the following subsection [EngstromEtAl20, IlyasEtAl18]. The pseudo-code for the clipped version of the PPO method is shown in Algorithm 2 in Appendix A.4.

General Advantage Estimation

During the training phase with the PPO method, the exact advantage function $A(\boldsymbol{s}, \boldsymbol{a})$ is usually not given. Therefore, an estimated advantage function $\hat{A}(\boldsymbol{s}, \boldsymbol{a})$ is computed with the Generalized Advantage Estimation (GAE) method [SchulmanEtAl18]. GAE estimates the expected return of a state-action pair by computing the Temporal Difference TD(λ). It uses a discounted sum of the TD residual:

$$\delta_t = r_t + \gamma v_\pi(\boldsymbol{s}_{t+1}) - v_\pi(\boldsymbol{s}_t), \tag{3.9}$$

where γ denotes the discount factor. The advantage estimate for time step t is thereby given by

$$\hat{A}_t(\boldsymbol{s}, \boldsymbol{a}) = \sum_{l=0}^{T} (\gamma\lambda)^l \delta_{t+l}, \tag{3.10}$$

with the decay rate λ. The TD residual δ is also used to approximate the value loss for the critic in the described PPO method. The estimated return is then given by

$$\hat{G}_t = \sum_{l=0}^{T} (\gamma\lambda)^l \delta_{t+l} + v_\pi(\boldsymbol{s}_t). \tag{3.11}$$

3.1.2 Imitation Learning Methods

Another technique in the field of DRL is imitation learning. In imitation learning, the agent learns a policy from an expert, where the agent tries to imitate the expert's behavior and strategy. Strictly speaking, imitation learning is a supervised learning method and thus distinguishes itself from the classical reinforcement learning definition. However, it is associated with reinforcement learning since some of its methods still presuppose an interaction with the environment and, furthermore, are based on fundamental assumptions made in traditional reinforcement learning.

Some imitation learning methods try to learn a policy through expert demonstrations and suggestions, like behavioral cloning and data aggregation. Another type of methods like inverse reinforcement learning and generative adversarial imitation learning [HoErmon16] instead aim to approximate the reward function of the environment. In regards to this thesis, the focus is set on the data aggregation method, which is used to pre-train an agent based on a model-based expert controller.

Behavioral Cloning

Behavioral cloning is a basic method in imitation learning and tries to learn the expert's strategy through supervised learning. The general term expert can refer to a human or another instance like another agent or a programmed controller demonstrating a control task. The agent tries to learn at each time step the link between the current state and the control signal from the expert through evaluating the expert's chosen action. Therefore, the learning objective is the difference between the approximated action from the agent for a given state and the optimal action from the expert.

This kind of imitation learning method is related to policy-gradient methods since both aim to optimize the policy directly. In policy-gradient methods, stochastic policies are usually used in practice to ensure exploration [SuttonBarto18]. By using a stochastic policy $\pi_\theta(\cdot|\boldsymbol{s})$, the negative log-likelihood of the expert action is used to determine the action loss, given by

$$L_{\mathrm{BC}}(\boldsymbol{\theta}) = -\frac{1}{T}\sum_{t=0}^{T} \log p(\pi^*(\boldsymbol{s}_t)|\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)), \tag{3.12}$$

where $\pi^*(\boldsymbol{s}_t)$ denotes the expert action at time step t and $p(\pi^*(\boldsymbol{s}_t)|\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t))$ the prediction probability of $\pi^*(\boldsymbol{s}_t)$ in state $\boldsymbol{s}_t$ given policy $\pi_\theta(\cdot|\boldsymbol{s})$. This loss function can be interpreted as the logarithmic of the likelihood of the agent choosing the expert's action.

This method can train an agent completely offline through recorded expert trajectories and works quite well for small state spaces. For tasks with higher complexity, the agent cannot compensate disturbances and uncertainties due to a lack of generalization. Small changes in the action approximation can cause the agent to end up in a non-encountered state. There, it cannot fit the situation to a specific action and cannot recover to get on track again. Therefore, the complexity and the capability of the agent are relatively limited [Sammut10].

Data Aggregation

Data Aggregation tries to tackle the issues given in Behavioral Cloning by installing an interactive expert during the exploring and learning process [RossGordonBagnell11]. By training on expert demonstrations, the agent encounters only optimal trajectories of the expert. The trajectories are optimal in a sense that they are considered optimal to the agent regarding the cost function. However, if the agent deviates from an optimal trajectory, it ends up in an undiscovered state where it did not learn a suitable action from the expert. This issue is resolved in the data aggregation method by constraining the expert to intervene in the control process instead of taking over complete control.

Algorithm 3 in Appendix A.5 explains the interaction between the agent and the expert in a pseudo-code, as implemented in this thesis. Likewise, Fig. 3.3 illustrates the following described process. At each time step t, an action from the agent $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$ and from the expert $\pi^*(\boldsymbol{s}_t)$ are suggested for the current state $\boldsymbol{s}_t$. With the probability of $\beta \in [0, 1]$, the expert action is chosen to control the quadcopter, otherwise, the agent's action is chosen with the probability of $(1 - \beta)$. The agent is then trained on the expert actions to each visited state with the behavioral cloning objective of Eq. 3.12.

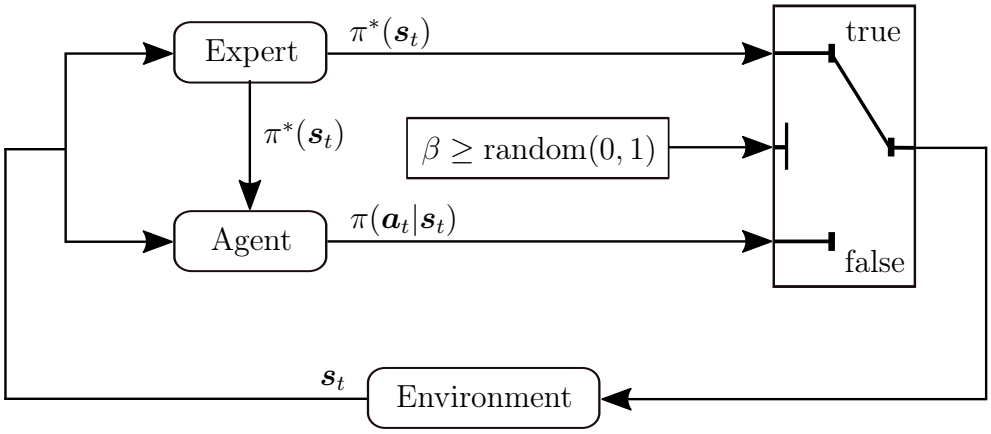


Figure 3.3: The schematic of the Data Aggregation algorithm as implemented in this work. At each time step t, the environment provides the current state s_t to the expert and the agent. Both expert and agent propose an action based on the current state. With the probability of β, the expert's action $\pi^*(\boldsymbol{s}_t)$ is chosen, otherwise the the agent's action $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$. The agent is then trained based on the expert's action.

Since the expert does not only demonstrate by taking over the complete control of the quadcopter, the agent is able to experience non-optimal states through its mistakes. The expert then suggests actions on which the agent can learn to recover and get back on track. The probability coefficient β determines how often the expert intervenes to support the agent in the control tasks.

3.2 Expert Setup

The expert is used for the imitation learning task to provide instructions on which the DRL-based controller learns a basic control strategy. Thereby, the exert is running parallel to the agent providing continuous action feedback on the visited states. In this thesis, the expert is implemented through a PID controller as detailed in Section 2.1.2. It is, however, also possible to use a higher-performing expert like a Model-Predictive controller (MPC) [MerabtiBouchachiBelarbi15].

3.2.1 PID Controller Setup

The attitude controller is tuned based on the dynamic model in Section 2.1.1, as proposed by [Mellinger12]. Consider the equation of motion for rotation of the $\boldsymbol{x}$-axis, Eq. 2.6, with

$$I_{xx}\ddot{\phi} + k_{d,\phi}\dot{\phi} + k_{p,\phi}\phi = 0, \tag{3.13}$$

where $k_{d,\phi}$ denotes the derivative gain and $k_{p,\phi}\phi$ the proportional gain of ϕ and $\dot{\phi}$. It is a second order system, which can be compared to

$$\ddot{\phi} + 2\xi\dot{\omega_n}\phi + \omega_n^2\phi = 0, \tag{3.14}$$

with ξ denoting the damping factor and ω_n representing the natural frequency. Therefore, the control rule for the angular gains can be set according to

$$\begin{aligned} k_{d,\phi} &= 2\xi\omega_n \\ k_{p,\phi} &= \omega_n^2. \end{aligned} \tag{3.15}$$

ω_n is not experimentally determined in the context of the thesis and instead adopted from [Mellinger12], with $\omega_n \approx 9\frac{\text{rad}}{\text{s}}$. $\xi \approx 1$ was also adpoted to design the controller close to critical damping. The gains for the angles ϕ and θ were both set according to Eq. 3.15.

For the Position Controller, $k_{p,\boldsymbol{r}}$, $k_{i,\boldsymbol{r}}$ and $k_{d,\boldsymbol{r}}$ were tuned experimentally by trial-and-error. The position controller is running at the same frequency as the attitude controller, contrary to the recommendation of [Mellinger12]. Instead, the desired angles ϕ_{des} and θ_{des} were both penalized with a factor of $\frac{1}{4}$ to ensure smoother rotations but nearly equal linear accelerations. A list of all control gains is given in Table A.1 in Appendix A.2.

Moreover, the optimal control gains depend on the distance to the goal point and the current state of the quadcopter. There are several methods in order to achieve a faster trajectory, like gain scheduling and adaptive gains [ÅströmEtAl93]. Using one of these methods, however, leads to a more aggressive behavior of the controller. Therefore, the gains were adjusted once to the specific learning task to maintain a smoother trajectory.

3.3 Agent Setup

DRL-based controllers based on neural networks are applicable to a wide variety of tasks. They have proven, for instance, to perform enormously well in systems with high degrees of freedom, like in humanoid robots and human-form robot hands [OpenAIEtAl19, MeloMáximo19].

In this thesis, a DRL-based controller is trained to control all four rotors directly and is implemented through a multi-layer feedforward network, as introduced in Section 2.3.1. The neural network is supposed to approximate and select an action for a given state based on the observation it receives. The action output of the neural network is then considered the agent's policy or the control strategy, which is primarily the optimization objective. Since the controller is designed based on DRL, the standard term agent is used synonymously to refer to the DRL-based controller.

3.3.1 Observation Space

The observation space is a representation of the state in the environment, on which the agent can determine an action. In particular, the observation space is a state vector that serves as the input for the neural network.

State Representation

The observation space is represented by a vector that consists of 18 features in total, containing the following components:

- the current position of the quadcopter in the initial frame $\boldsymbol{r}$,
- the vector representation of the rotation matrix in the body frame ${}^{B}\boldsymbol{R}^{I}$,
- the linear velocity in the body frame $\boldsymbol{v}_B$,
- and the angular velocity of the quadcopter in the body frame $\boldsymbol{\omega}_B$.

Alternatively, the orientation of the quadcopter can be described through quaternions or Euler angles. They both require fewer entries in the state vector than the rotation matrix and thus reduce the number of parameters in the neural network to be tuned. However, both have a considerable disadvantage when it comes to the distinctiveness of particular states. Quaternions, for instance, describe the same rotation with both positive and negative signs, with $-q = q$. This relationship can lead to confusion and wrong pose interpretation of the agent. Likewise,

the Euler angles representation suffers from discontinuity when a full rotation is accomplished, where $2\pi = 0$. Therefore, the rotation matrix is used to represent the orientation component to map the actual orientation correctly.

Normalization

Normalizing inputs and outputs is a standard method to speed up the training and raise the estimation accuracy of neural networks [SolaSevilla97]. By limiting the magnitude of both inputs and outputs, the neural network can classify the situation more effectively. As suggested in the work of [HwangboEtAl17], the state is scaled to roughly follow a Gaussian distribution. In the same way, the policy is also scaled to follow nearly a Gaussian distribution. In continuous robotic tasks, such as the operation of the quadcopter, it is, however, difficult to approximate the distribution of the individual state components for the scaling.

The normalization is resolved with a vector for the expected mean of the observation μ, which is assumed to be zero for each component, and a vector for the standard deviation σ. The neural network input will then be scaled down with the following relation:

$$s_t^{\text{in}} = \frac{s_t - \mu}{\sigma}. \tag{3.16}$$

In tasks where some state vector components have clear boundaries, the standard deviation can be easily approximated through the expected average value of the component. For example, in the training task in Section 4.2.1, a target position is randomly generated within a distance of 10 m. The average expected value for each coordinate is 0, since the sampled coordinates range from $[-10, 10]$ m and follow a Gaussian distribution. Therefore, the x-, y-, and z-coordinates are expected to have the same standard deviation of $\sigma_i = \frac{10}{\sqrt{3}}$, with $i = 1, 2, 3$. By sampling a representative amount of targets and computing the standard deviations for each coordinate with

$$\sigma_i = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \tag{3.17}$$

this expected standard deviation can be observed.

For the linear and angular velocity with no clear boundaries, the standard deviation is experimentally determined by running a representative amount of training rollouts and computing the variance with the parallel algorithm, introduced by [ChanGolubLeVeque79]. This method computes the mean value of a state component online over the sets of collected states. Hereby, after each episode, the average value of a state component is computed over all sets of states, which

then is used to update the standard deviation. This method is used to approximate the standard deviation for the linear and angular velocities. The scaling components for the approximated standard deviation $\boldsymbol{\sigma}$ is given in Table A.2 in Appendix A.3.

3.3.2 Action Space

The DRL agent is supposed to directly control the generated rotor thrusts through the rotor velocities according to Eq. 2.4. Therefore, the neural network output size is equal to the number of rotors, where each output resembles the control signal for a respective rotor. The agent does not have a model of the quadcopter and cannot compute the state transition. The control strategy is instead approximated.

In the quadcopter setup, the rotor thrust is limited to a factor of $[0.5, 1.5]$ of the thrust necessary for hovering. Like the observation space, the action space is also normalized to simplify the training process. Since the boundary of the action space is known, the action space is scaled down to the interval of $[-1, 1]$. After that, the scaled control action is scaled up again with:

$$u_t = (0.5 \cdot u_t^{\text{out}} + 1) * u_{\text{hover}}. \tag{3.18}$$

3.3.3 Neural Network Architecture

Two neural networks are set up in an actor-critic architecture for the training task. Both networks take the same state vector as input. The actor, representing the DRL-based controller and the agent's policy, outputs an action vector, while the critic, on the other hand, outputs a value estimate based on the expert's actions.

For both actor and critic, two hidden layers are set up, each composed of 64 neurons. Since neural networks are pretty versatile in applications, these number of neurons and hidden layers are selected based on networks that have shown success in similar tasks [HwangboEtAl17]. Moreover, a higher number of neurons and hidden layers also increases the number of parameters to be tuned. Variations in the number of neurons were also tested, but the best results regarding time efficiency and the resulting benefit were achieved with 64 neurons and two hidden layers. The Tanh function is chosen as the activation function for the hidden layers since they performed better than the leaky ReLU function in experimentation.

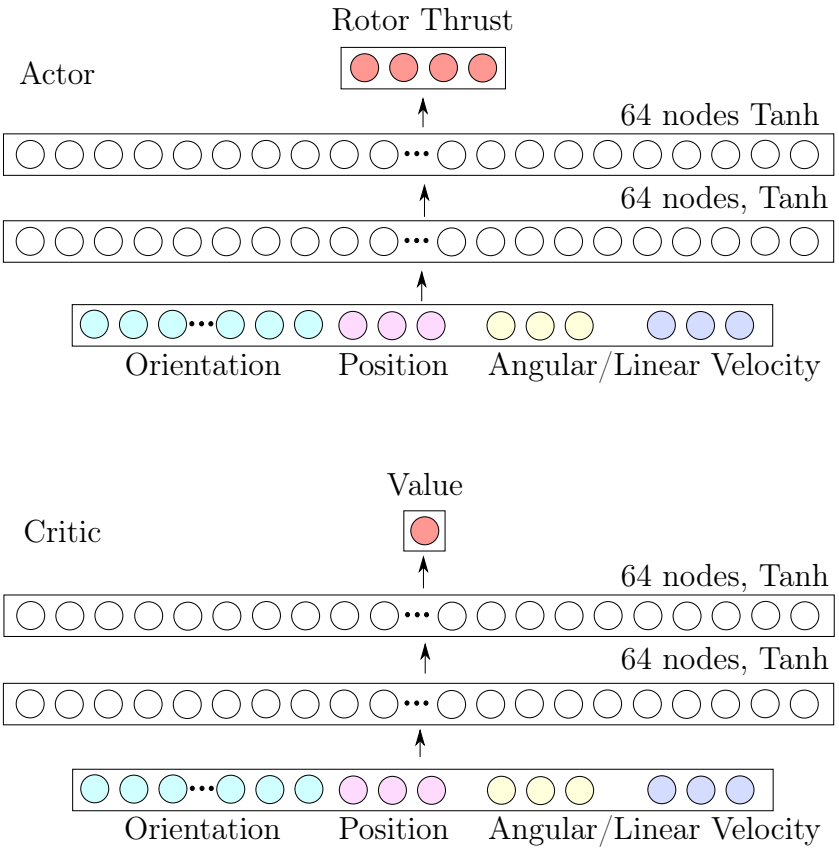


Figure 3.4: The two neural networks in an actor-critic architecture, as set up for the training. Both the actor and the critic take in the orientation, position, velocity, and angular velocity as input and have two hidden layers with 64 neurons and the Tanh activation function. The actor outputs the desired rotor thrusts while the critic outputs a value estimate. The actor network hereby represents the DRL-based controller [HwangboEtAl17].

Chapter 4

Simulation and Evaluation

The last chapter discussed the methods used to approach the deep reinforcement learning (DRL) problem. This chapter specifies the task environment and the training procedure on which the DRL-based controller is trained. Section 4.1 introduces the simulation framework and the setup of the quadcopter environment. In Section 4.2, the training setup of the two-staged training of the DRL-based controller is specified, and the results of the respective training stages are presented. In Section 4.3, the DRL-based controller is evaluated and compared to the expert in different scenarios.

4.1 Simulation Setup

The whole training of the DRL-based controller is simulated with Raisim, a simulation engine for dynamic and robotic simulations [HwangboLeeHutter18]. It is a multi-body physics engine specialized in simulations of rigid-body systems in general. It is comparable to common simulation engines like MuJoCo, and PyBullet [TodorovErezTassa12, CoumansEtAl13]. It is closed-source, but a free academic license is provided.

To layout the training process efficiently, the gym submodule of Raisim is used for the training with DRL. A gym is a toolkit for developing and comparing DRL algorithms. It simplifies setting up a loop of recurrent episodic tasks, automating the training, and monitoring the learning process. Furthermore, it enables parallelization of the training through multiple environments, making the training highly time-efficient. A comparable commonly used gym is the OpenAI-gym [BrockmanEtAl16].

Figure 4.1 illustrates the macro-structure of the simulation framework. The DRL algorithms in this work are written in the Python programming language since

most of the machine learning frameworks are available for Python. Since Python is known to operate slower than other lower-level programming languages like C and C++, Raisim-gym supports vectorized environments and multiprocessing for the physics simulation, programmed in C++, enabling fast running of multiple parallel simulated environments to speed up the training process. The communication between the DRL algorithm and the environments takes place through a Python wrapper interface. At each time step, the trained agent receives a batch of state representations for each environment and sends back a batch of actions for each state of each environment. The transition to the following states is then simulated for each environment separately.

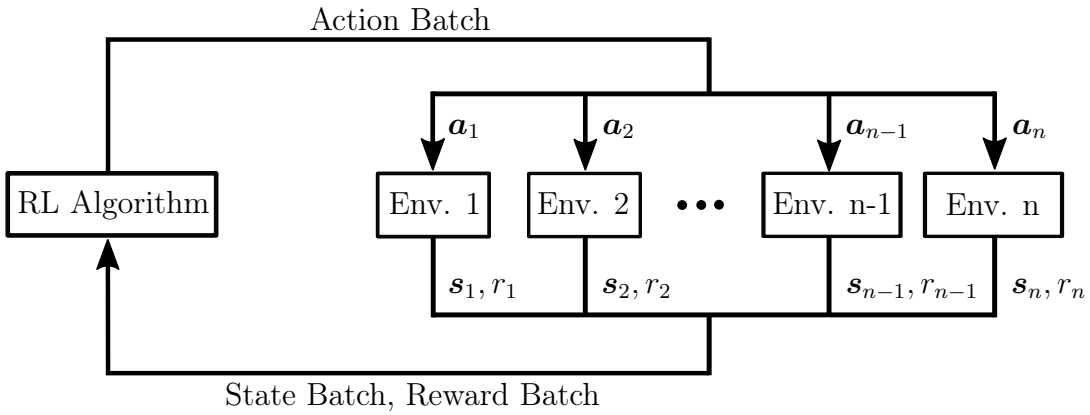


Figure 4.1: A schematic of the simulation framework. The reinforcement learning algorithm is implemented separately from the environment and the robot model. At each time step t, the reinforcement learning algorithm receives a batch of states and rewards from each n parallel environments and sends a batch of actions to take back for each environment.

As discussed in Section 3.3, the DRL-based controller is modeled and trained with a multi-layer feedforward neural network. A Pytorch framework is used to model the neural network and perform the training [PaszkeEtAl19]. It allows to build custom neural networks and provides various optimization methods. It is also used to speed up the neural network training by delegating parts of the processing to the GPU. Throughout the training process, the progress of the DRL-based controller is tracked with Tensorboard [AbadiEtAl15].

4.1.1 Quadcopter Environment

In this thesis, a quadcopter model from the ITM is implemented. In the previous work, the quadcopter is modeled and simulated in Gazebo using the ROS frame-

work [ITM, Stanford Artificial Intelligence Laboratory et al., KoenigHoward04]. Unlike Raisim, Gazebo is not specialized in DRL tasks and does not simulate multiple environments in parallel to batch the data for the training process. Another advantage of Raisim is the possibility to simulate without a visualization, which also speeds up the training process by using less computational resources.

The quadcopter model is a rigid-body system with a base body and four rotors. Each rotor is connected to the base through a joint with one degree of freedom. Also, each rotor has a mass and inertia, which are negligible compared to the base. The quadcopter cannot directly generate thrust through the rotation of the rotors in the simulation. Therefore, the acting forces and moments are transformed and applied at the base w.r.t the base coordinate system according to Eqs. 2.3 and 2.6. The aerodynamical drags and gyroscopic effects of the rotors are both assumed to be negligible. For the visualization, the rotor velocities are set to an average speed of 700 rpm and still produce small gyroscopic moments.

The quadcopter environment itself only contains the quadcopter model and an infinitely wide ground, as shown in Fig. 4.2. The discretization steps are set to 0.0025 s for the simulation and 0.01 s for the control loop in all tasks. The total amount of time steps per episode differ depending on the task.

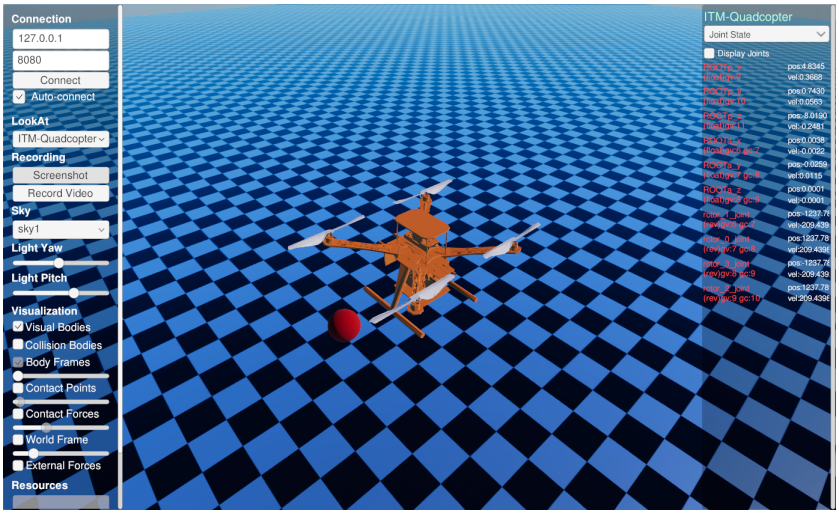


Figure 4.2: The visualization of the quadcopter (orange) in the simulation environment. The red dot resembles the goal point, and the checked surface is the ground. The rotors (grey) can rotate independently.

Further Setup in the Simulation

Simulations, in general, cannot precisely depict all real-world properties due to discretization errors and model uncertainties. Modeling real-world conditions is only possible to a certain extend. It is critical in DRL to set up the simulation such that the policy can be transitioned to the real world safely. Deep reinforcement learning-based controllers are well known to suffer from a so-called reality gap, where the controller fails to generalize on the real-world task [ZhaoQueraltaWesterlund20]. Recent studies have nonetheless proven that a successful transition can be accomplished in practice [HwangboEtAl19]. Therefore, an additional setup is taken in taken into account to lower the gap to the real-world environment.

Motor model. The exact experimentally determined model of the motor rotation delay is not available for this quadcopter. According to [PowersMellingerKumar15], the rotor speed delay can be linearly modeled with

$$\dot{\omega}_i = k_m(\omega_i^{\text{des}} - \omega_i), \tag{4.1}$$

where k_m denotes the motor gain. For a quadcopter of the size of the ITM quadcopter, $k_m = 20$ can be assumed. Equation 4.1 is applied step-wise to the actual rotor speed with the simulation time step of 0.0025 s. Therefore, the rotor speed is increasing in four steps until the next control iteration starts. Furthermore, a delay of one control time step of 0.01 s is applied to consider a controller delay. Also, the rotor speed limits were set to $[0.5, 1.5]$ times the rotor speed needed for hovering due to safety reasons.

Sensor Noise. The quadcopter consists of multiple sensors through which the state can be estimated. The state estimation is usually a different domain and provides a state estimate for the controller. The controller then computes a control strategy based on the state estimate. Due to sensor noise and estimation errors, there is still a some amount of uncertainty in the real-world application. These uncertainties were not measured in the context of this thesis. However, a Gaussian noise of 5 % is considered to the state representation of the environment to make the DRL-based controller more robust.

4.2 Training Setup and Results

The training aims to design a DRL-based controller for a quadcopter that fully control the quadcopter through the thrusts generated by the rotors. The main

requirements on the controller are that it can fly to any target point while compensating for disturbances. Therefore, a two-staged training is set up to train the controller in order to fulfill these conditions. Since a hybrid method based on DRL is used to train a controller, the term agent is used synonymously to refer to the DRL-based controller.

In the first stage, an agent is pre-trained in a target tracking task with the data aggregation method based on the expert's suggestions to imitate the control behavior of the expert. Thereafter, the agent is further trained in a second stage based on the pre-trained neural network model with the proximal policy optimization (PPO) method and additional expert action suggestions. Thereby, the agent has to solve a more challenging task where it has to recover from large disturbances. Thereby, the agent is supposed to learn to generalize and cope with more critical situations. By training in these two stages, a DRL-based controller with a more robust control strategy can be achieved much more efficiently than in a training setup solely with data aggregation or PPO.

4.2.1 Imitation Learning Setup

The training with imitation learning aims to initialize the policy and the neural network model of an agent based on a model-based expert controller. The expert provides continuous action suggestions for each encountered state on which the learner develops a policy to imitate the expert's actions. In this thesis, a fine-tuned PID controller is utilized as the expert controller, as described in Section 3.2.

Training Task Setup

The purpose of the task is to train an arbitrary stochastic policy $\pi_\theta(\cdot|\boldsymbol{s})$ to track randomly generated target points by following the instructions of the expert. At each episode and in each parallel environment, the quadcopter spawns at the initial position $\boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m in an ideal hovering state with $\boldsymbol{q}_0 = [1, 0, 0, 0]^\intercal$, $\boldsymbol{v}_0 = [0, 0, 0]^\intercal$ $\frac{m}{s}$, and $\boldsymbol{\omega}_0 = [0, 0, 0]^\intercal$ $\frac{rad}{s}$. In each environment, a target point with a distance of 10 m in an arbitrary direction relative to the quadcopter is randomly generated, which is to be tracked. The quadcopter is thereby spawned 15 m above the ground to provide enough tolerance to the ground for the agent.

At each time step, the expert's action is chosen with a probability of β, otherwise, the agent's proposed action is chosen, according to the data aggregation algorithm in Section 3.1.2. The action selection happens independently for each environment with the same probability. Both trajectories are then aggregated, and the agent is trained on the action deviance to the expert.

The engagement of the expert, given by the probability β, is set at an initial value of 0.5 and then linearly decreased by 0.001 until $\beta = 0.3$. In this way, the agent receives more support at the beginning, where the agent's actions are highly random. By reducing the help of the expert, the agent is engaged to act more and more on its own.

Loss Function

The overall loss function for the training with imitation learning is composed of several loss terms. The main loss term is the negative log-likelihood of the expert action over the agent's policy distribution, according to behavioral cloning loss stated in Eq. 3.12. An L2-regularization loss, as stated in Eq. 2.25, and an entropy loss are additionally considered in the overall loss function. The entropy describes the randomness of an action chosen by the policy, which is given by

$$H(\boldsymbol{\theta}) = - \sum_{\boldsymbol{a} \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s})} \pi_\theta(\boldsymbol{a}|\boldsymbol{s}) \log \pi_\theta(\boldsymbol{a}|\boldsymbol{s}). \tag{4.2}$$

The entropy directly relates to the unpredictability of the actions chosen by the policy. It is often used to constrain an agent into exploring longer before the actions become too deterministic. Generally, the longer the agent is trained, the more confident it becomes in its action selections. Therefore, the variance of the chosen actions decreases until the actions become nearly deterministic. With the L2-regularization loss, the neural network should remain more balanced in the weight distribution.

Since an actor-critic architecture is used, the critic is also pre-trained through an additional weighted value function loss in the same manner as in Eq. 3.8. In total, the overall loss function is given by

$$J(\boldsymbol{\theta}) = L_{\mathrm{BC}} + c_{\mathrm{L2}} L_{\mathrm{L2}} + c_H H(\boldsymbol{\theta}) + c_{v_\pi} J_{v_{\boldsymbol{\pi}}\text{-clip}}(\boldsymbol{\phi}), \tag{4.3}$$

where $c_{(\cdot)}$ denotes the respective coefficient for the individual partial loss function. The losses are determined and averaged over a mini-batch of state-action pairs. The respective coefficients $c_{(\cdot)}$ of the overall loss function are listed Table A.4 in Appendix A.5.

4.2.2 Reinforcement Learning Setup

The implemented expert is suitable for non-acrobatic flights with small angles and smooth trajectories. Real-world tasks, however, involve uncertain conditions like wind turbulences, estimation errors, and model errors. Therefore, a second more challenging task is set up where the agent is trained on the PPO method

based on the pre-trained neural network model of the agent trained in the first stage to generalize more on the control task.

Training Task Setup

The second training task aims to enhance the stability of the agent and is inspired by the approach of [HwangboEtAl17]. Up to this point, the agent has learned to approach a target point, where the training consisted of avoiding unstable situations. To make the agent's policy more robust against disturbances, the agent experiences a high initial disturbance in a state recovery task, where it has to figure out a control strategy through trial-and-error and approach the initial position.

At the beginning of each episode, the agent is spawned at the initial position $\boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m with random orientation $\boldsymbol{q}_{rand}$, random linear velocity $\boldsymbol{v}_{rand}$ and random angular velocity $\boldsymbol{\omega}_{rand}$. The goal is to recover from heavy random disturbances before hitting the ground and to fly back to the initial position $\boldsymbol{r}_0$. The random velocity is sampled from a Gaussian distribution with a standard deviation of 2 $\frac{m}{s}$. Likewise, the angular velocity is sampled from a Gaussian distribution where the standard deviation is 4 $\frac{rad}{s}$.

Loss Function

In this training setup, the agent is trained with the clipped version of the PPO method, as described in Section 3.1.1. The main surrogate loss function, according to Eq. 3.6, is computed to determine the performance of the policy at each episode.

The L2-regularization loss and the entropy loss are considered additionally to the PPO loss function together with a weighted behavioral cloning loss of the expert's actions, which makes it a hybrid method. The behavioral cloning loss is supposed to encourage the agent to approach the target point when the received reward signals are too ambiguous to find the target correctly. The overall composed loss function, on which both the actor and the critic are trained, is thereby given by:

$$J(\boldsymbol{\theta}) = J_{\text{PPO-clip}} + c_{v_\pi} J_{v_{\boldsymbol{\pi}}\text{-clip}}(\boldsymbol{\phi}) + c_{\text{L2}} L_{\text{L2}} + c_H H(\boldsymbol{\theta}) + c_{\text{BC}} L_{\text{BC}}, \tag{4.4}$$

where the individual parameters $c_{(.)}$ denote the respective weights. Table A.3 in Appendix A.4 lists all coefficients and parameters implemented for this task.

Reward Shaping

The reward function is critical for the success of the reinforcement learning task. The reward is given when the agent takes an action and transitions into a new state. It is generally difficult to assess how to shape the reward optimally. Therefore, different reward functions were tried, and the most relevant are listed as follows. In general, the reward also refers to negative rewards or penalties. The reward signals are either received continuously after each control loop or when a specific condition is reached. Table A.3 in Appendix A.4 lists the values of all following reward coefficients as implemented in this work.

Position reward. The main objective of both tasks is to reach a certain destination state, depending on the specific task. The most relevant information about the destination is the position $\boldsymbol{r}_{goal}$, which is the main target. The position reward is therefore a feedback of how close the agent is to its target point, defined by

$$r_t(\boldsymbol{r}) = -c_r \cdot ||\boldsymbol{r}_{\text{target}} - \boldsymbol{r}_t||_2. \tag{4.5}$$

The coefficient c_r denotes the position reward coefficient.

Orientation reward. The orientation reward is responsible for preventing a tilt of the z_B-axis of the quadcopter w.r.t the the z-axis of the initial frame. Penalizing the tilt encourages the agent to pursue an orientation that is more stable for flight. The orientation reward is given by

$$r_t(\phi, \theta) = -c_{\phi,\theta} \cdot \arccos({}^B R^I_{33}) = -c_{\phi,\theta} \cdot \arccos(\cos\phi \cos\theta). \tag{4.6}$$

The orientation reward coefficient is set nearly as high as the position reward coefficient to encourage a stable behavior.

Angular velocity reward. Another way to stabilize the quadcopter tilt is to restrict the angular velocity of the quadcopter. Considering that the agent initially does not know how to stabilize from high rotations, the angular velocities might get relatively high. By penalizing high angular velocities, the quadcopter is encouraged to act against the rotation. The angular velocity reward is given by

$$r_t(\boldsymbol{\omega}) = -c_\omega \cdot ||\boldsymbol{\omega}_B||_2. \tag{4.7}$$

The reward coefficient c_ω is set much lower than the other presented rewards since a high angular velocity is desirable in some cases like fast maneuvering.

Collision reward. If the agent collides with an obstacle, it will receive a penalty proportionally much larger than other rewards. In this thesis, the only obstacle is the ground plane. Therefore, to make the agent avoid the ground, a high negative reward is applied when the agent collides with it. Also, by colliding with the ground, the agent has reached a terminal state, and the training episode

is over. The collision reward is defined by:

$$r_t = \begin{cases} -R_{\text{col}}, \text{ if collision with ground} \\ 0, \text{ else} \end{cases} . \tag{4.8}$$

Distance reward. A similar penalty, as set in the collision reward, is also applied when the agent exceeds a certain distance from the desired destination. This should motivate the agent to stay in the bounds where it might find a trajectory with a higher reward easier. Likewise, by reaching the set distance, the training episode is over and the agent has reached its terminal state. The distance reward is defined by

$$r_t(\boldsymbol{r}, d) = \begin{cases} -R_d, \, ||\boldsymbol{r}_t||_2 > d \\ 0, \text{ else} \end{cases} . \tag{4.9}$$

4.2.3 Results

In both stages, the training takes place in 240 independent environments in parallel, where each environment generates a random target point or random initial state at each episode, depending on the task. An episode of training, therefore, generates 240 trajectory samples on which the agent is trained. A trajectory consists of 1500 state-action pairs since the total simulation time of an episode is set to 15.0 s with a control time step of 0.01 s. After each finished episode, the agent is trained in a number of epochs n_{epochs} on the generated training data. For each epoch, the 240 training samples are split into a number of mini-batches $n_{\text{mini-batches}}$ and chronologically prepared to train the agent. An average loss is computed for each mini-batch, based on the respective loss function and the current parameters of the agent. Thereafter, the agent's parameters are updated based on the average loss with the Adaptive Moment Estimation (ADAM) method [KingmaBa14]. The agent is therefore updated $n_{\text{epochs}} \cdot n_{\text{mini-batches}}$ times on the training samples before it generates new training samples in a new episode. The specific training parameters are listed in the Tables A.3 and A.4 in the Appendices A.4 and A.5.

The training results for both agents are presented in the following sections. To differentiate the agents of both stages, the agent of the imitation learning task is referred to as IL-agent and the agent of the reinforcement learning task as RL-agent. The training progress of the respective agents is also compared to the training progress of a reference agent trained in the same task solely with PPO. The comparison occurs with the same task setup where the respective agent controls the quadcopter without the expert intervening. The parameters of the best-performing version of the IL-agent are chosen for the RL-agent to be further trained in the second stage. After the second stage, the overall best

performing version of the IL-agent is chosen and validated in different scenarios in Section 4.3.

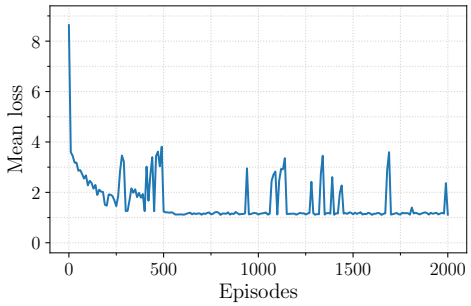
First Training Stage

The IL-agent was trained on 240 random target points simultaneously at each episode. In this setup, a target point is considered to be reached when the quadcopter is within a specified tolerance. The PID controller, being the expert, takes about 7.75 s to reach the target point within a tolerance of 0.1 m. Therefore, the simulation time is set to 15.0 s to provide enough time for the IL-agent to explore the environment. The average simulation speed for this setup was about 19.7 s per episode for all 240 environments. In total, it took about 11 h to simulate 2000 episodes. The IL-agent was trained on 480,000 random target points in total.

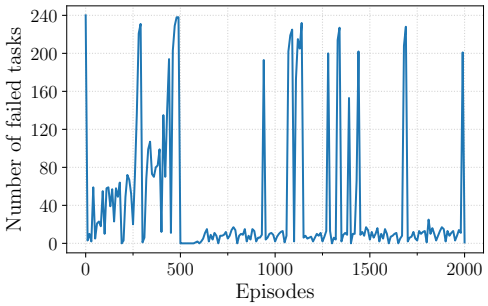
The success of the training with the data aggregation method was relatively unstable and, in some configurations, entirely arbitrary. Besides the regular hyperparameters, a highly crucial training parameter was the action selection probability β. Many combinations in the parameter β have been tested, like a constant β or β decreasing from a range of 1.0 to 0.0 and vice versa. A high initial β, where the expert mainly demonstrates the control task, reduced the learning success of the IL-agent. In the same way, starting from an initial β of 0.0 and increase gradually lead to a slow learning process. The best results were achieved for β in a range of $[0.3, 0.5]$. Thereby, the best and most stable policy could be trained with a setup, where β initially started at 0.5 and decreased with a rate of 0.001 per episode until a value of 0.3.

The learning rate had to be set relatively low with 0.00005 since the IL-agent became too fast too confident in the action selection. The variance in the policy probability distribution decreased too quickly, and the IL-agent stopped making significant progress and started overfitting the current targets. This behavior could not be regularized through the entropy loss. Fig. 4.3 presents the training progress and compares the IL-agent to an agent trained in the same task with the PPO algorithm.

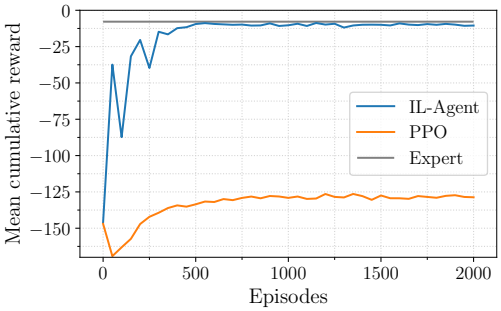
The IL-agent made relatively fast progress and reached its best performance at Episode 550. After that, more training did not result in a better control strategy. In some episodes, the IL-agent seemed to unlearn how to control the quadcopter due to overfitting on the current target points of the episode, which lead to the recurrent high peaks in Figs. 4.3 (a) and (b). Compared to the PPO-agent, the IL-agent managed to make significant progress and reach a success rate of 100 % in episode 550. Thereby, the IL-agent managed to reach an average total reward of -8.23, nearly as high as the average reward of -7.84 the expert receives for



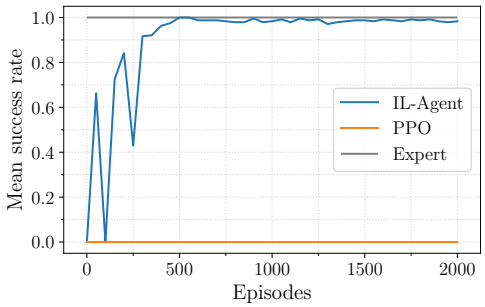
(a) Mean overall loss



(b) Number of failures during the training process



(c) Mean cumulative rewards



(d) Mean success rate in the target tracking task

Figure 4.3: The training progress of the IL-agent is shown in (a) and (b). The performance of the IL-agent in the first training task is compared in (c) and (d) to the expert and a reference agent, which is trained only with PPO. In (b) and (d), a task is considered successful when the quadcopter at least does not collide with the ground.

its control strategy. However, the PPO-agent could not learn a successful control strategy at all since the task setup is not fit for the learning method. A possible reason could be the high percentage of failures where the PPO-agent cannot clearly interpret the reward signals in this task.

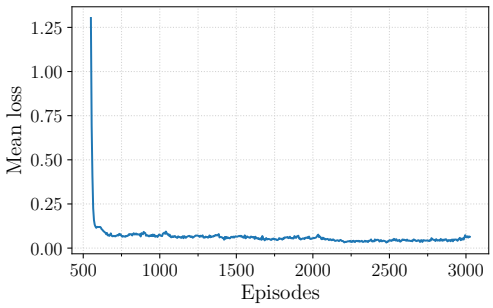
Second Training Stage

The second training stage was performed based on the pre-trained neural network model of episode 550 since this version of the IL-agent was the most reliable and achieved the highest reward. The RL-agent was trained for another 2450 episodes with the setup described in Section 4.2.2. The simulation took about 13,30 s on average for each episode with a total simulation time of 9.05 h for all 2450 episodes, where the RL-agent was trained on 588,000 state recovery tasks in total.

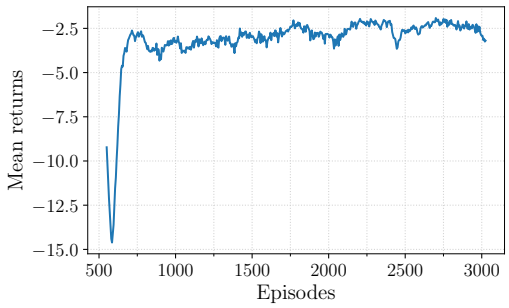
The training results are presented in Fig. 4.4. The RL-agent quickly learned to stabilize the quadcopter and recover from the initial chaotic movements, as indicated by the mean loss and mean return. From there, the improvements were slowly and steadily. By comparing the RL-agent with the expert and a PPO-agent, solely trained on PPO in the same training setup with 3000 episodes, the RL-agent outperforms both in the average success rate and in the average reward. In the second training task, the RL-agent is clearly ahead of the expert and manages to achieve better results than the PPO-agent. Through the second training stage, the RL-agent also improved on the target tracking task, as shown in Figs. 4.4 (e) and (f). Thereby, the RL-agent could reach a higher reward than the expert for the first time. The best overall RL-agent was achieved after 2800 episodes which received an average reward of -7.76 in the first training task, compared to the average reward of -7.84 of the expert. In the state recovery task, the RL-agent achieved an average success rate of 88.75 %, while the expert only reached an average success rate of 29.48%. A task is considered successful, when the agent or the expert manages to recover without colliding with the ground.

The PPO-agent performed relatively similarly to the RL-agent, with a best average success rate of 82.5 % in the state recovery task. However, it could not reach an average return as high as the RL-agent since after recovering from the disturbances, the PPO-agent could not find the destination and circled the target point. Also, it did not manage to generalize enough to perform the target tracking task reliably. Still, it could achieve much better results in the target tracking task compared to the PPO-agent trained in the first stage, even though it was trained on the stage recovery task. In the state recovery task, the PPO-agent could learn much easier a control strategy, most likely due to the task properties. The PPO-agent could probably determine or somehow get an idea of the desired goal since the quadcopter is spawned at the position with the highest reward.

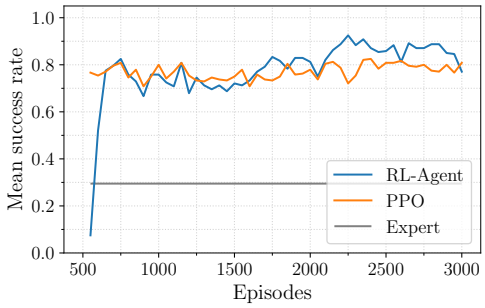
Therefore, the search space to actions leading to higher rewards is much smaller in the state recovery task compared to the target tracking task.



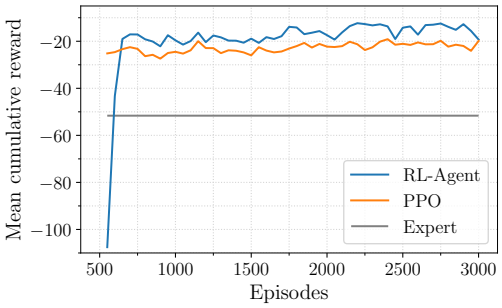
(a) Mean overall loss



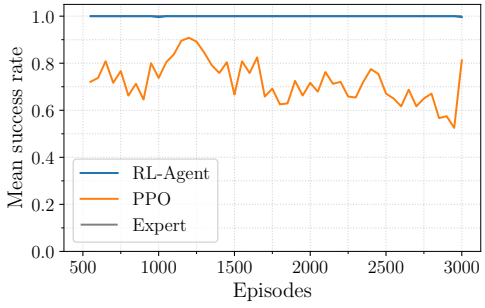
(b) Mean returns from GAE



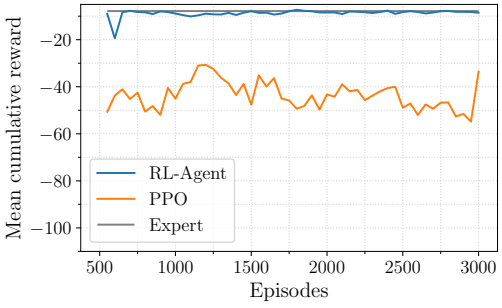
(c) Mean success rates in state recovery task



(d) Mean cumulative rewards in state recovery task



(e) Mean success rates in target tracking task



(f) Mean cumulative rewards in target tracking task

Figure 4.4: The training progress of the RL-agent as displayed in Figures (a) and (b). Figures (c) and (d) show the performance of the agent in the second training task compared to the expert and a PPO-agent trained only with PPO on the state recovery task. Figures (e) and (f) show a comparison of the RL-agent to the expert and the same PPO-agent on the target tracking task.

4.3 Validation

In this section, the best agents of both stages are quantitatively validated in their respective training tasks. Since the RL-agent has an overall better performance compared to the IL-agent, the RL-agent is further compared to the expert in different tasks. Hereby, other scenarios are simulated that represent possible real-world situations to examine how the RL-agent performs in general tasks.

4.3.1 Target Tracking

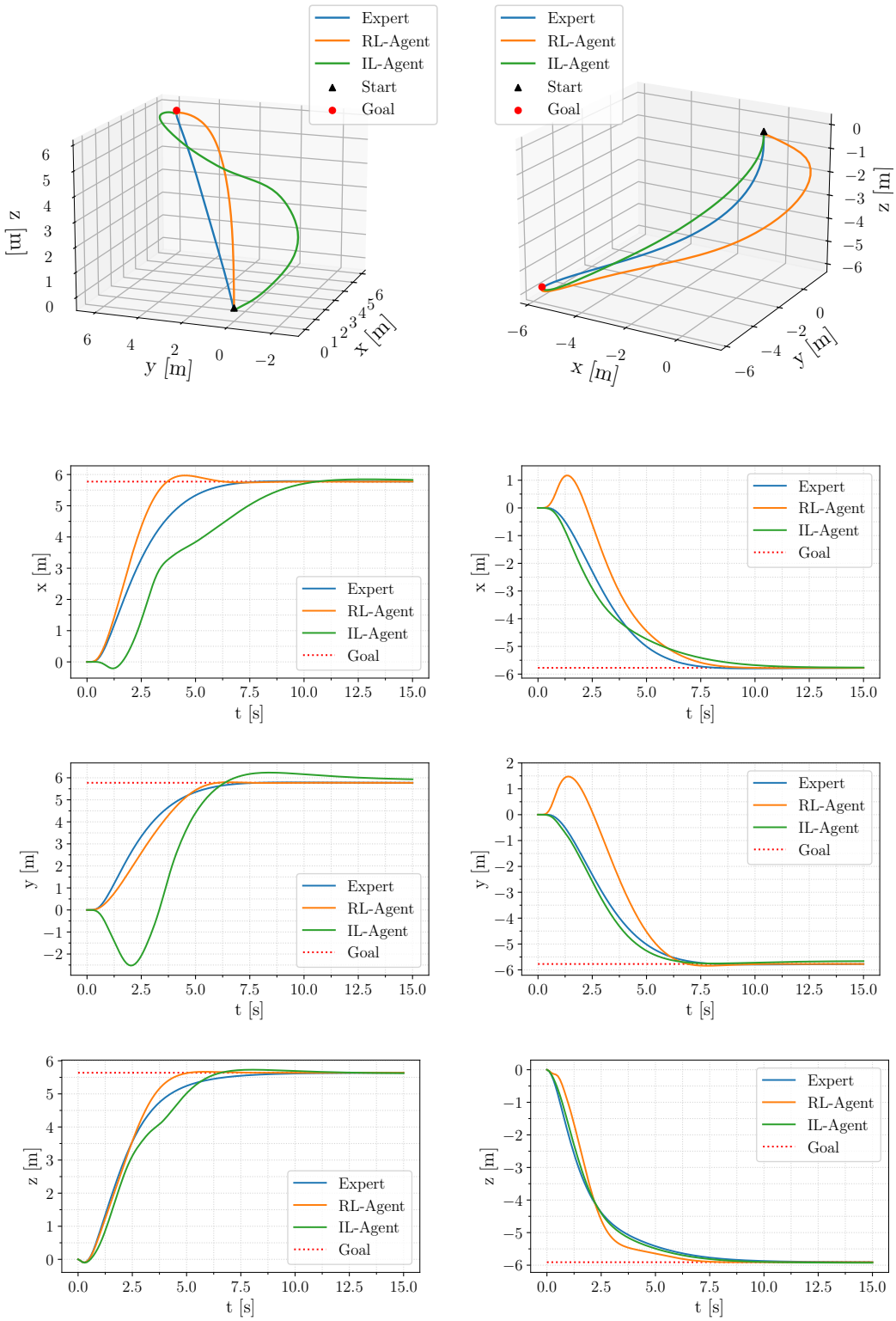
The respective agents are quantitatively and qualitatively evaluated in the random target tracking task, as described in Section 4.2.1. In addition to the presented results in Section 4.2.3, the average duration and the average path lengths are measured to get a more detailed idea of the performances of the agents compared to the expert.

IL-Agent

In direct comparison to the expert, the IL-agent is overall slower and takes a larger route to find the target point. The duration and the length of the route depend on the target's position and do not directly follow a pattern. Figure 4.5 shows two trajectories for two exemplary target points $\boldsymbol{r}^{(1)}_{target} = [10, 10, 10]^\intercal \cdot \frac{1}{\sqrt{3}}$ m and $\boldsymbol{r}^{(2)}_{target} = [-10, -10, -10]^\intercal \cdot \frac{1}{\sqrt{3}}$ m, both having a distance of 10 m from the initial position. For $\boldsymbol{r}^{(1)}_{target}$, the IL-agent could not reach the target point within a tolerance of 0.1 m due to an offset. However, when expanding the tolerance to 0.2 m, the IL-agent takes about 10.03 s to reach the destination, while the expert needs 6.54 s with the same criterion. For a tolerance of 1 m, the duration decreases to 5.76 s for the IL-agent and 4.57 s for the expert. Obviously, the IL-agent needs nearly half of the time to approach the last 1 m. With a total path length of 15.84 m, the IL-agent also takes a larger route than the expert with 10.15 m, which is also observable in Fig. 4.5 (a).

For the target point in the opposite direction $\boldsymbol{r}^{(2)}_{target}$, the expert could reach the goal much faster with 8,71 s with a tolerance of 0.2 m and a path length of 10.88 m. In comparison, the expert needs about 7.13 s and, surprisingly, 11.15 m to reach the destination.

By and large, the IL-agent is, as expected, slower compared to the expert since it learned from the expert. To get a better idea of the general performance of the IL-agent, Fig. 4.6 demonstrates the average duration of a target tracking task with the tolerances of 0.2 m up to 1 m. The average path length to a random

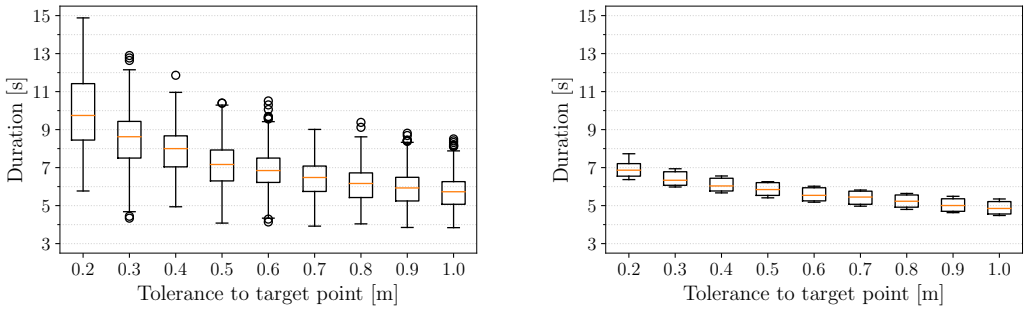


(a) Trajectories to target point $\boldsymbol{r}_{target}^{(1)}$

(b) Trajectories to target point $\boldsymbol{r}_{target}^{(2)}$

Figure 4.5: The trajectories of the expert and the agents for target points $\boldsymbol{r}_{target}^{(1)}$ (a) and $\boldsymbol{r}_{target}^{(2)}$ (b).

target point is about 11.76 m for the IL-agent and 10.49 m for the expert. When the IL-agent is supposed to track a more distant target of 15 m distance, the average success rate decreases to 62.19 %. For targets within the distance of 20 m, the average success rate drops to 14.48 %.



(a) Task durations of the IL-agent

(b) Task durations of the expert

Figure 4.6: The durations from the initial state to a random target point of the IL-agent (a) and the expert (b) sampled from over 240 random targets per tolerance step.

RL-Agent

In the direct speed comparison, the RL-agent is still slightly slower in reaching the destination compared to the expert. While the expert needs 6.54 s on average to reach a goal with a tolerance of 0.2 m, it takes 10.67 s for the RL-agent on average, which is an improvement over the IL-agent. The RL-agent needs 5.97 s on average to reach the target point with a tolerance of 1 m, compared to the 4.57 s of the expert. Therefore the last 1 m to the target point is approached by the RL-agent relatively damped, as is the case for the IL-agent. The average route length to the target point decreased to 10.61 m, which is still longer than the expert's path length of 10.15 m.

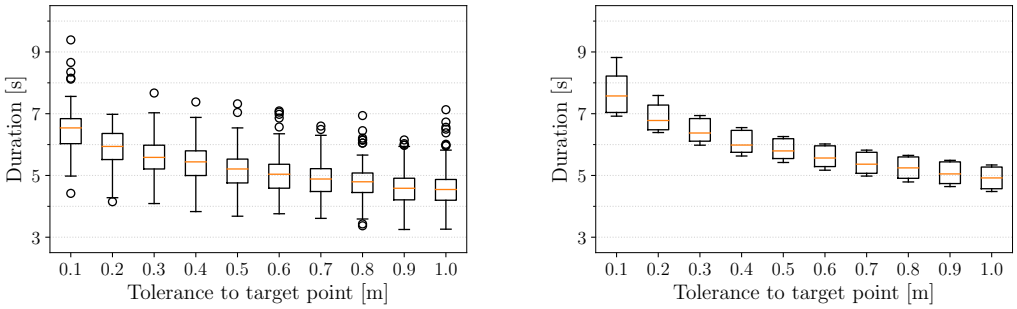
The difference in the speed and the offset of the RL-agent can, however, be compensated through the normalization scaling of the observation, which is described in Section 3.3.1. The agent receives a normalized representation of the state in the environment, given by Eq. 3.16. The offset to the target point can be compensated by adding the average position offset w.r.t. the target points to the position component of the mean $\boldsymbol{\mu}$, with $\mu_1 = -0.04249$, $\mu_2 = 0.03032$ and $\mu_3 = 0.01967$. by applying these values for $\boldsymbol{\mu}$, the agent approaches the target point directly and has no more significant offset.

By adjusting the position component of the standard deviation σ with a scaling

factor α, the distance to the projected target point can be stretched and compressed in the view of the agent. Therefore, when the agent is near the actual target point, a stretching of the target point can be beneficial regarding the speed with $\alpha < 1$. However, applying $\alpha < 1$ leads to unstable behavior for longer distances since an $\alpha < 1$ projects the target point to a more distant position out of the scope where the agent was trained on. The agent can reliably track target points with a distance higher than 10 m only to a certain extend. Therefore, the factor α is parameterized dependent on the actual distance of the target, given by

$$\alpha = 0.5 * (1 + \frac{||\boldsymbol{r} - \boldsymbol{r_{target}}||_2}{10}). \tag{4.10}$$

The closer the agent gets to the actual target, the lower α gets and vice versa. By applying Eq. 4.10, the speed of the RL-agent in reaching the target improves drastically while also enhancing the stability for more distance target points. With this scaling factor, the RL-agent now outperforms the expert PID controller in the average speed to the target point. Fig. 4.7 presents the comparison of the average durations for reaching a target point with several tolerance steps. The agent is now able to reach a target point within a tolerance of 0.1 m in 6.53 s on average, while it takes the expert 7.75 s for the same criterion.



(a) Task durations of the RL-agent

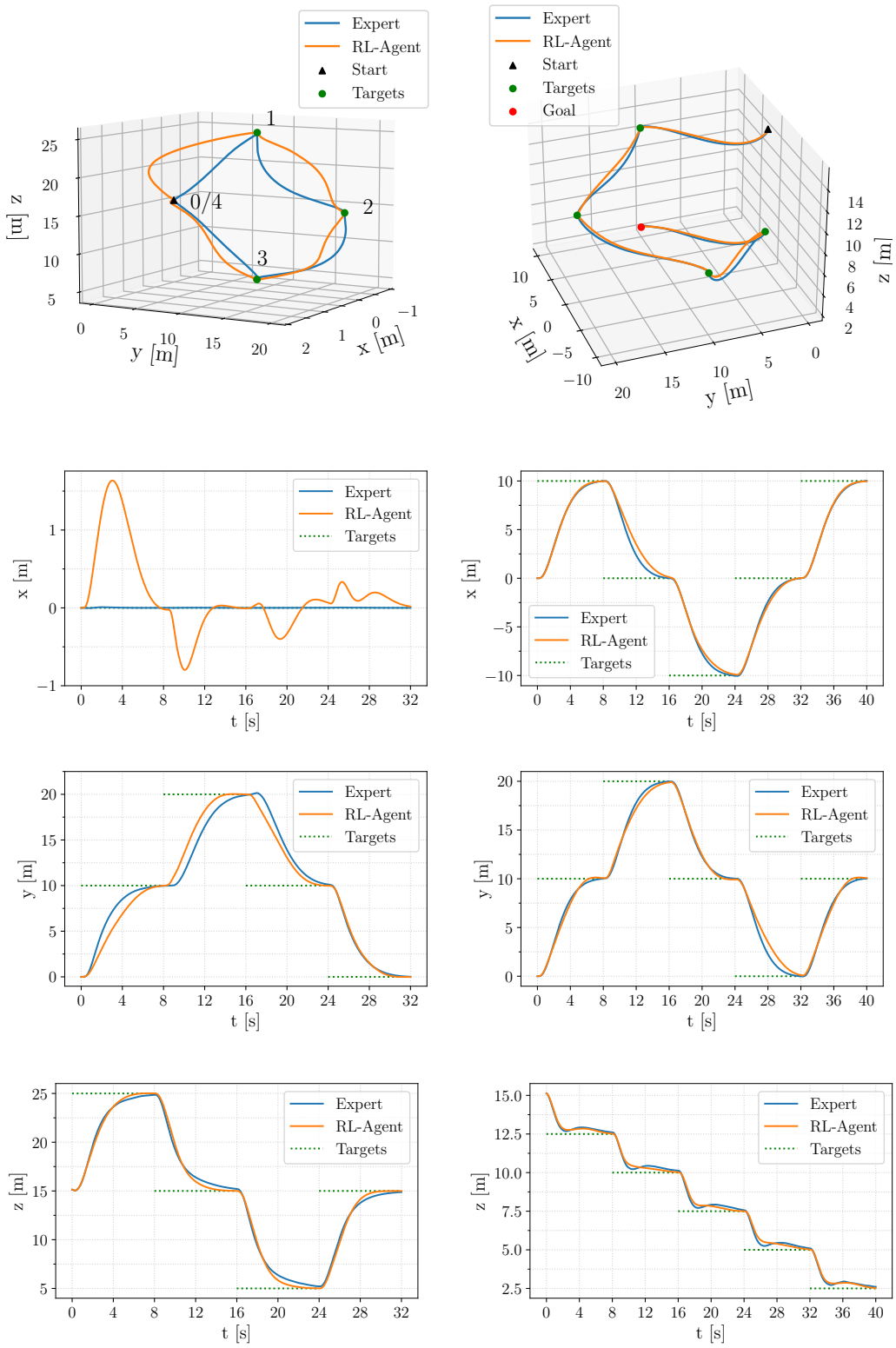
(b) Task durations of the expert

Figure 4.7: The durations from the initial state to a random target point of the RL-agent (a) and the expert (b), sampled from over 240 random targets per tolerance step.

Multiple Targets Tracking

To investigate the behavior of the agent in more complex tasks, two waypoint tracking tasks with several targets are set up and presented in Fig. 4.8.

In the first task, the agent has to track four points ordered in the z-y-plane with $\boldsymbol{r}^{(1)}_{plane} = [0, 10, 25]^\intercal$ m, $\boldsymbol{r}^{(2)}_{plane} = [0, 20, 15]^\intercal$ m, $\boldsymbol{r}^{(3)}_{plane} = [0, 10, 5]^\intercal$ m, and



(a) Trajectories to target points $\boldsymbol{r}_{plane}$

(b) Trajectories to target point $\boldsymbol{r}_{spiral}$

Figure 4.8: The agent and the expert in two multiple target point tracking tasks. In (a), the four targets are given in the z-y-plane, while in (b), five targets are placed in the 3D space in a order resembling a downward spiral.

$\boldsymbol{r}_{plane}^{(4)} = \boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m. Compared to the expert, the agent is slightly faster in reaching the targets, but oscillates relatively large in the x-direction, while the expert straightly approaches the target.

In the second task, the quadcopter has to track five points in the 3D-space ordered in a downward spiral, with $\boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m, $\boldsymbol{r}_{spiral}^{(1)} = [10, 10, 12.5]^\intercal$ m, $\boldsymbol{r}_{spiral}^{(2)} = [0, 20, 10]^\intercal$ m, $\boldsymbol{r}_{spiral}^{(3)} = [-10, 10, 7.5]^\intercal$ m, $\boldsymbol{r}_{spiral}^{(4)} = [0, 0, 5]^\intercal$ m, and $\boldsymbol{r}_{spiral}^{(5)} = [10, 10, 2.5]^\intercal$ m. Hereby, the RL-agent behaves quite similar to the expert and reaches the targets again slightly faster without any oscillations.

4.3.2 Recovering from Disturbances

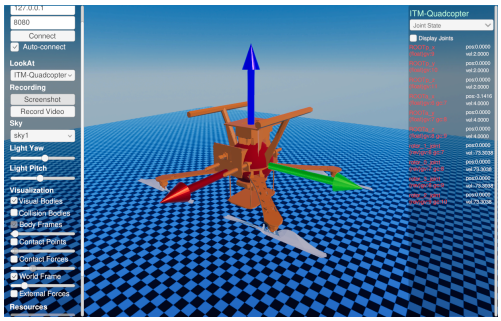
In this section, the RL-agent is evaluated in more critical situations against the expert. In real-world applications of the quadcopter, it is more likely to experience disturbances that result in unstable and critical conditions. The RL-agent is tested in the following situations to evaluate its capabilities compared to the RL-agent.

Recovering from Unstable States

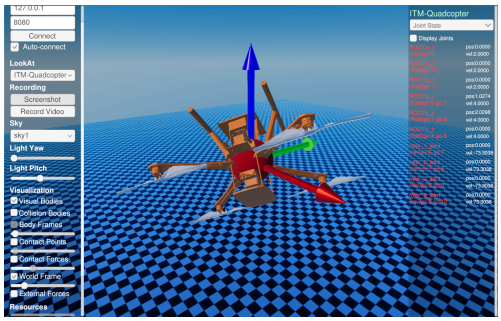
The RL-agent and expert are both tested in the random state recovery task setup of the second training stage to quantize their stability. Hereby, an average success rate is computed over four runs with 240 samples. In direct comparison, the average success rate of the RL-agent is 88.25 %, while the average success rate of the expert is only 29.48 %.

To illustrate the trajectories of the RL-agent and the expert, two exemplary tasks with defined initial states are performed and plotted on Fig. 4.9. In initial state $\boldsymbol{s}_{init}^{(1)}$ with the initial position $\boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m, the quadcopter is spawned upside down with an orientation of $\boldsymbol{q}_{init}^{(1)} = [0, -1, 0, 0]^\intercal \frac{\text{m}}{\text{s}}$, a velocity of $\boldsymbol{v}_{init}^{(1)} = [2, 2, 0]^\intercal \frac{\text{m}}{\text{s}}$, and an angular velocity of $\boldsymbol{\omega}_{init}^{(1)} = [4, 4, 4]^\intercal \frac{\text{rad}}{\text{s}}$. The agent manages to recover from the disturbance after about 1.4 s and fly back to the initial position $\boldsymbol{r}_0$ while the expert collides with the ground after 1.8 s. A sequence of figures showing the motion of the RL-agent and the expert in this task is given in Appendix A.6.

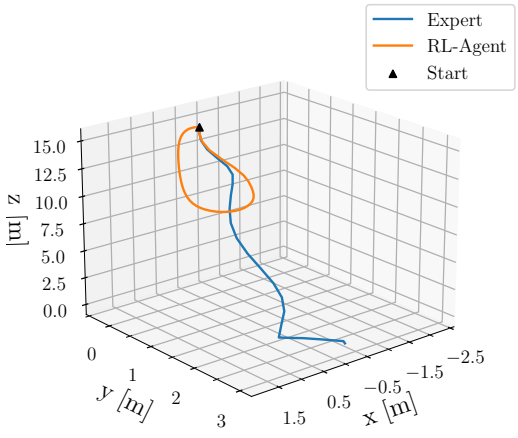
In initial state $\boldsymbol{s}_{init}^{(2)}$ with initial position $\boldsymbol{r}_0$, the quadcopter is spawned with a relatively arbitary orientation of $\boldsymbol{q}_{init}^{(2)} = [0.42, 0.41, 0.81, 0]^\intercal$, which corresponds to a rotation of about $[152, 43, -115]^\intercal$ ° in Euler angles. The velocity is set to $\boldsymbol{v}_{init}^{(1)} = [2, 2, -2]^\intercal \frac{\text{m}}{\text{s}}$ and the angular velocity to $\boldsymbol{\omega}_{init}^{(1)} = [4, 4, 4]^\intercal \frac{\text{rad}}{\text{s}}$ again. In this scenario, the expert could also recover and fly back to the initial position $\boldsymbol{r}_0$. The agent takes a smoother trajectory and gets in to a stable state more quickly.

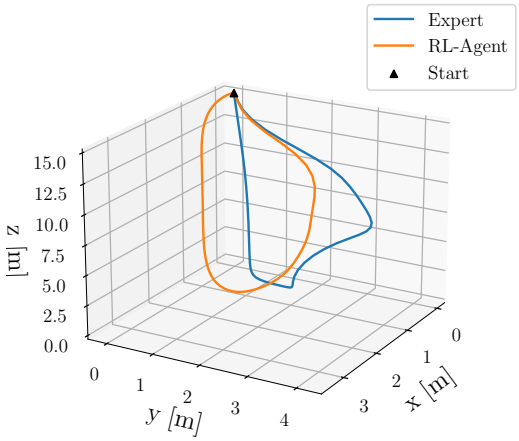


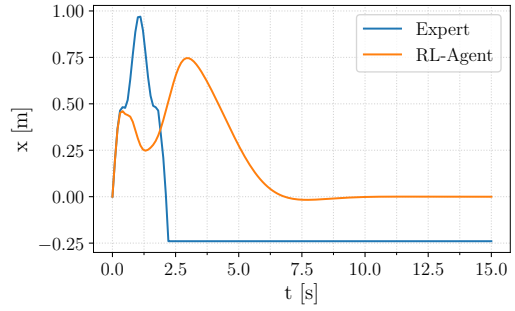
(a) Quadcopter pose for $\boldsymbol{s}_{init}^{(1)}$

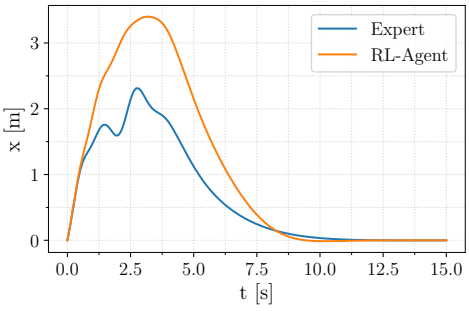


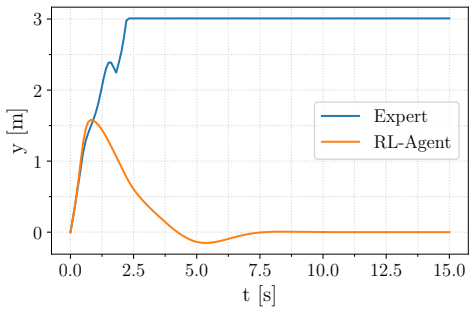
(b) Quadcopter pose for $\boldsymbol{s}_{init}^{(2)}$

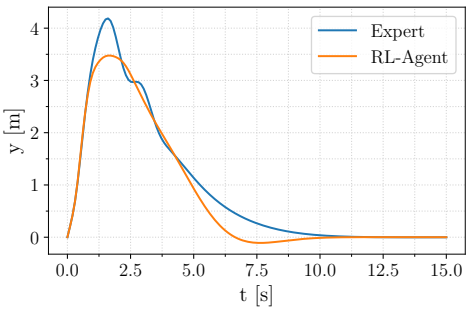


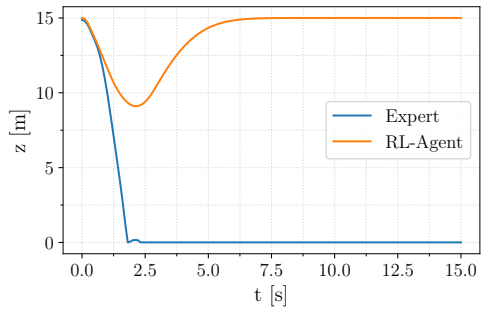




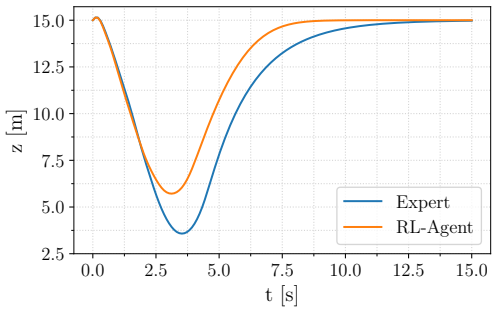








(c) Trajectories for $\boldsymbol{s}_{init}^{(1)}$



(d) Trajectories for $\boldsymbol{s}_{init}^{(2)}$

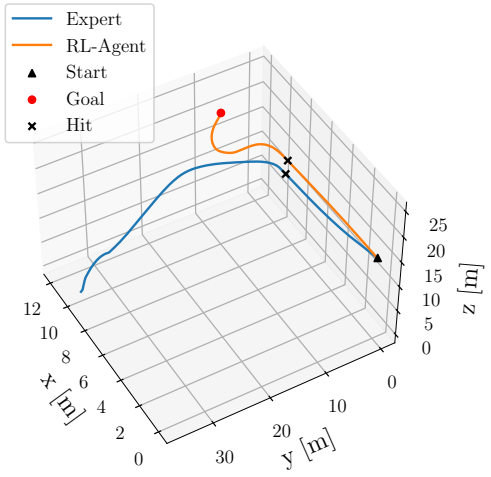
Figure 4.9: The trajectories of the expert and the RL-agent for the initial states $\boldsymbol{s}_{init}^{(1)}$ in (c) and $\boldsymbol{s}_{init}^{(2)}$ in (d). The respective poses of the quadcopter are shown in (a) and (b).

Recovering from Side Hits

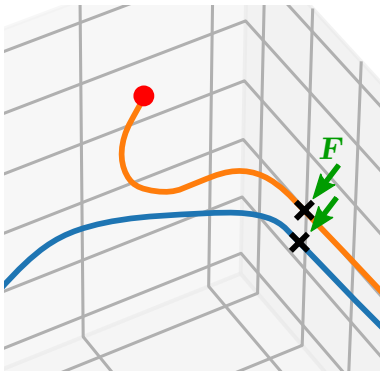
In real-world applications, the quadcopter can encounter side forces, e.g., through irregular gusts of wind, objects or obstacles. To investigate the behavior of both RL-agent and expert, a hit from the side is simulated through a resulting force and torque, which is presented in Fig. 4.10. The target is the position $\boldsymbol{r}_{target} = [10, 10, 25]^\intercal$ m and the quadcopter is spawned in the initial position $\boldsymbol{r}_0 = [0, 0, 15]^\intercal$ m. A side hit is applied 2.5 s after the beginning of the movement with a force $\boldsymbol{F} = [-20, -20, 0]^\intercal$ N and a torque $\boldsymbol{M} = [-5, -5, -5]^\intercal$ Nm. As shown in Fig. 4.10, the RL-agent manages to recover from the side hit easily while the PID controller totally loses control and does not recover.

Compensating Constant Drags

Another possible scenario to investigate is a drag force, for example, given by an additional mass or constant wind resistance. Hereby, a continual drag force is applied on the quadcopter with $\boldsymbol{F} = [-3, -3, -3]^\intercal$ N, which, generously rounded, corresponds to a weight force as high as the weight of the quadcopter. Both the RL-agent and the expert failed to compensate for the drag at the last 1.8-2.3 m, as shown in Fig. 4.11. The RL-agent did not encounter such conditions during the training and could not learn from such a situation.

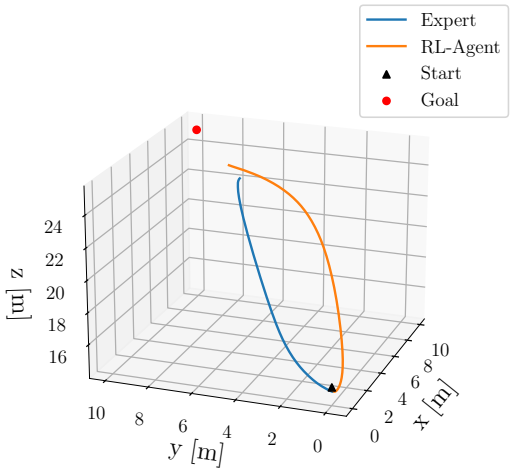


(a) Trajectories after a side hit

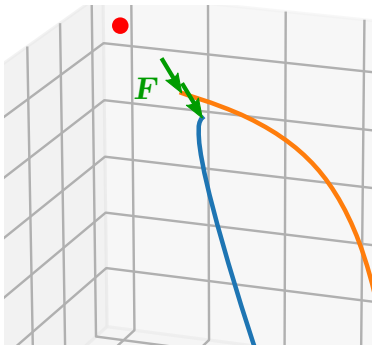


(b) Cropped plot of (a) with the direction of the forces

Figure 4.10: The RL-agent and the expert both recieve a side hit after 2.5 s while flying to the target point at $\boldsymbol{r}_{target} = [10, 10, 10]^\intercal$ m. (a) shows the trajectory of both RL-agent and expert, while (b) is cropped into the plot to visualize the direction of the force $\boldsymbol{F} = [-20, -20, 0]^\intercal$ N.



(a) Trajectories with a constant drag



(b) Cropped plot (a) with the direction of the forces

Figure 4.11: The expert and the RL-agent approaching the same point $\boldsymbol{r}_{target} = [10, 10, 10]^\intercal$ m in (a) with a continuously acting drag force $F = [-3, -3, -3]^\intercal$ N, visualized in (b).

Chapter 5

Conclusion

In this thesis, a controller has been designed and trained based on deep reinforcement learning (DRL), which can control and fly a quadcopter to given target points while compensating disturbances. The DRL-based controller, also named DRL agent, learned to directly control all four rotors of the quadcopter. Thereby, the agent was modeled with a neural network to approximate an optimal control strategy and trained in two stages with two different tasks in a hybrid method.

In the first training stage, the agent has been trained with imitation learning based on the data aggregation method. Thereby, it learned to track target points with the help of an interactive expert by imitating its control behavior. The expert was implemented through a PID-controller for small-angle conditions. With the occasional interventions and continuous suggestions of the expert for each state, the agent could learn a control strategy much faster than a reference agent, which was trained on pure DRL with the Proximal Policy Optimization (PPO) method in the same training task. The performance of the agent, however, was still limited to the maximum performance of the PID-controller since the expert is the optimum that the agent can theoretically reach. Due to a lack of generalization and accuracy in the navigation, the agent was not able to fully match the behavior of the expert.

Furthermore, the second stage was designed to fill the performance gap with a training task based on deep reinforcement learning. A more challenging training task was performed based on the pre-trained agent model to enhance the stability of the controller, where the quadcopter was spawned with a random initial state, and the agent had to recover from a relatively high initial disturbance. The agent was thereby trained with the PPO method, a policy-gradient method with an on-policy value function to predict the expected return and thereby evaluate the agent's actions. The reward was shaped such that it encouraged the agent to primarily pursue a stable state, avoid a collision with the ground, and then

decrease the distance to the actual target point. Additional support from the expert through action suggestions was considered in the training process to prevent an offset to the approached target point and speed up the training process. By and large, the agent was able to outperform the expert and a reference agent, trained with PPO in the same task in the second stage. The trained agent could fulfill the state recovery task in over 88 % of the times, which is reasonably high compared to the success rate of over 29 % of the expert. The reference agent achieved a relatively high success rate of over 82%, but did not reach the overall performance of the agent. In other scenarios, the agent could recover much easier from disturbances like side hits compared to the expert. In the target point tracking task, the agent was also able to reach the goal state about 15 % faster.

Despite the promising learning results, there is still room for improvement for the resulting DRL-based controller. The trajectory could still be optimized since they occasionally contain small oscillations perpendicular to the position vector from the current position to the target point. Like the expert, it also cannot compensate for continuous external forces like a mass attached to it or constant air drag on its own since it did not experience such conditions during the training. An agent could be initially trained on a higher-performing expert like a model-predictive controller to obtain a more optimal initial policy in future work. Also, a value-based deep reinforcement learning method could be applied in the second stage to compare the performance of the resulting agents.

Appendices

A.1 Temporal-Difference Algorithm

Algorithm 1 Tabular TD(0) for estimating v_π [SuttonBarto18].

Input: the policy π to be evaluated
Output: the value function v_π
$v(\boldsymbol{s}) = 0$, for all $\boldsymbol{s} \in S$, except $v(\boldsymbol{s}_{terminal}) = 0$ /* *Initialize* */
for *each episode* ***do***
    $\boldsymbol{s} = 0$ /* *Initialize* */
    for *each step t of episode, s is not terminal* ***do***
        $\boldsymbol{a}_t \leftarrow$ *action given by* π *for* $\boldsymbol{s}_t$
        take action $\boldsymbol{a}_t$*, observe* r*,* $\boldsymbol{s}_{t+1}$
        $v(\boldsymbol{s}_t) \leftarrow v(\boldsymbol{s}_t) + \alpha[r_{\boldsymbol{s}_{t+1}} + \gamma v(\boldsymbol{s}_{t+1}) - v(\boldsymbol{s}_t)]$
        $s \leftarrow$ *new state* $\boldsymbol{s}_{t+1}$
    end for
end for

A.2 Expert PID Controller Gains

Table A.1: PID controller gains of the expert

Parameter description	**Parameter**	**Value**
Position controller		
Proportional gain	$k_{p,x} = k_{p,y} = k_{p,z}$	1.5
Integral gain	$k_{i,x} = k_{i,y} = k_{i,z}$	50
Derivative gain	$k_{d,x} = k_{d,y} = k_{d,z}$	4.1
Attitude controller		
Proportional gain, roll angle	$k_{p,\phi}$	0.54165
Proportional gain, pitch angle	$k_{p,\theta}$	0.81810
Proportional gain, yaw angle	$k_{p,\psi}$	0.80676
Derivative gain, roll angle	$k_{d,\phi}$	0.12037
Derivative gain, pitch angle	$k_{d,\theta}$	0.18180
Derivative gain, yaw angle	$k_{d,\psi}$	0.17928

A.3 Observation Scaling Components

Table A.2: Scaling components of the observation state vetor

Parameter description	**Parameter**	**Value**
Position scaling component	$\boldsymbol{\sigma}_r$	$\frac{10}{\sqrt{3}}$
Orientation scaling component	$\boldsymbol{\sigma}_R$	$\sqrt{3}$
Velocity scaling component	$\boldsymbol{\sigma}_v$	5
Angular velocity scaling component	$\boldsymbol{\sigma}_\omega$	5

The scaling components of the linear and angular velocity are determined experimentally and rounded up to an average value since an exact value is not needed. The position and the orientation scaling components are, on the contrary, determined based on the expected average value.

A.4 Proximal Policy Optimization Implementation

Algorithm 2 Proximal Policy Optimization in the clipped version [MnihEtAl16].

1: **Input**: initial parameters $\boldsymbol{\theta}_0$ of policy $\pi_{\boldsymbol{\theta}_0}$, initial parameters $\boldsymbol{\phi}_0$ of value function $v_{\boldsymbol{\phi}_0}$
2: **Output**: optimal policy $\pi_{\boldsymbol{\theta}_K}$ and optimal value function $v_{\boldsymbol{\phi}_K}$
3: **for** $k = 0, 1, 2, ...$ **do**
4: run $\pi_{\boldsymbol{\theta}_k}$, collect trajectories $\mathcal{D}_k = \{\tau_i\}$ for all time steps $t = 0, 1, 2, ..., T$
5: compute rewards-to-go $\hat{R}^{(t)}$
6: compute advantage estimates $\hat{A}^{(t)}_{\pi_{\theta_k}}$ based on $v_{\boldsymbol{\phi}_k}$ and $\hat{R}^{(t)}$
7: updade the policy by maximizing the PPO-Clip objective:

$$\boldsymbol{\theta}_{k+1} \leftarrow arg\max_{\boldsymbol{\theta}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} J^{(t)}_{\text{PPO-clip}}(\boldsymbol{\theta_k})$$

8: fit value function by regression on mean-squared-error:

$$\boldsymbol{\phi}_{k+1} \leftarrow arg\min_{\boldsymbol{\phi}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} J^{(t)}_{v_{\boldsymbol{\pi}},k\text{-clip}}(\boldsymbol{\phi_k})$$

9: **end for**

The pseudo-code of the Proximal Policy Optimization (PPO) algorithm for a batch size of 1, as implemented in this thesis. The objective functions $J_{\text{PPO-clip}}(\boldsymbol{\theta})$ and $J_{v_{\boldsymbol{\pi}}\text{-clip}}(\boldsymbol{\phi})$ are detailed in Eqs. 3.6 and 3.8. The parameters for the training setup of Section 4.2.2 are listed in Table A.3. The parameters are either denoted with the respective symbol or with the respective parameter name in the script files. The reward coefficients are applied according to Secton 4.2.2.

Table A.3: Parameter setup for the training with PPO

Parameter description	**Parameter**	**Value**
Environment and training parameters		
Number of environments	num_envs	240
Time steps per environment per episode	n_steps	1500
Number of mini-batches per epoch	num_mini_batches	3
Number of epochs per episode	num_learning_epochs	6
PPO-specific parameters		
Discount factor	γ	0.998
GAE decay rate	λ	0.95
Learning rate	α	0.00075
Maximum gradient clipping value	max_grad_norm	0.5
PPO clipping parameter	clip_param	0.2
Loss coefficients		
Value loss coefficient	c_{v_π}	0.5
Entropy loss coefficient	c_H	0.0001
L2-regularization loss coefficient	c_{L2}	0.0001
Behavioral cloning loss coefficient	c_{BC}	0.005
Reward coefficients		
Position reward coefficient	c_r	0.003
Orientation reward coefficient	$c_{\phi,\theta}$	0.002
Angular velocity coefficient	c_ω	0.0005
Collision reward coefficient	R_{col}	15
Distance reward loss coefficient	R_d	15

A.5 Data Aggregation Implementation

Algorithm 3 Implementation example of Data Aggregation

1: **Input**: initial parameters $\boldsymbol{\theta}_0$ of policy $\pi_{\boldsymbol{\theta}_0}$, expert π^*
2: **for** $k = 0, 1, 2, ..., K$ **do**
3: For every time step $t = 0, 1, 2, ..., T$ and for probability $\beta_k \in [0, 1]$, let selected action be

$$\pi_k = \begin{cases} \pi^*, \textbf{ if } \beta_k \geq \text{random}(0,1) \\ \pi_{\boldsymbol{\theta}_k}, \textbf{ if } \beta_k < \text{random}(0,1) \end{cases}$$

4: run π_k, collect trajectories $\mathcal{D}_k = \{\tau_k(\boldsymbol{s}, \pi_k)\}$
5: get dataset $\mathcal{D}_k^* = \{\tau_k(\boldsymbol{s}, \pi^*)\}$ of visited states by π_k and actions given by π^*
6: aggregate datasets $\mathcal{D}_k \leftarrow \mathcal{D}_k \cup \mathcal{D}_k^*$
7: update policy on $\mathcal{D}_k$, e.g. through mean-suqared error:

$$\boldsymbol{\theta}_{k+1} \leftarrow arg\min_{\boldsymbol{\theta}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} -\log \pi^*(\boldsymbol{s}_t)$$

8: adjust expert action selection probability with

$$\beta_{k+1} \leftarrow \max(\beta_k - \beta_{\text{decay}}, \beta_{\text{target}})$$

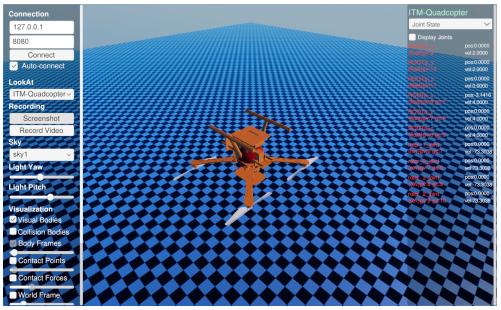
9: **end for**

The Data Aggregation algorithm, as implemented in this thesis. The original algorithm is introduced by [RossGordonBagnell11]. Table A.4 lists all parameters used for the training setup described in Section 4.2.1. The reward function setup and the value loss described in Sections 4.2.2 and 4.2.2 were used together with the respective coefficients in Table A.3 to pre-train the critic neural network.

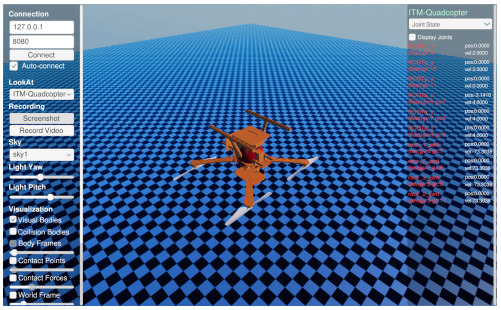
Table A.4: Parmeter setup for the training with Data Aggregation

Parameter description	**Parameter**	**Value**
Environment and training parameters		
Number of environments	num_envs	240
Time steps per environment per episode	n_steps	1500
Number of mini-batches per epoch	num_mini_batches	8
Number of epochs per episode	num_learning_epochs	12
Data Aggregation specific parameters		
Initial action selection probability	β_{init}	0.5
Target action selection probability	β_{target}	0.3
Action selection probability decay rate	β_{decay}	0.001
Learning rate	α	0.00005
Loss coefficients		
Value loss coefficient	c_{v_π}	0.5
Entropy loss coefficient	c_H	0.0005
L2-regularization loss coefficient	c_{L2}	0.0001

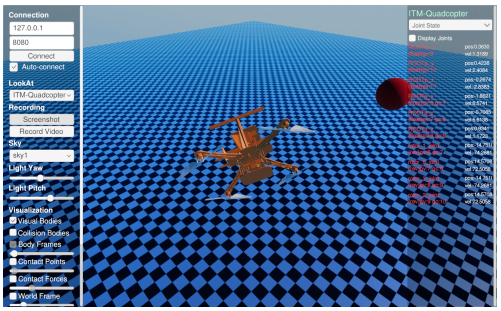
A.6 Motion Sequence of a State Recovery Task



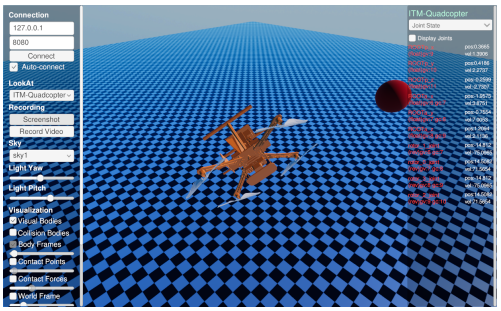
(a) RL-Agent, $t = 0$ s



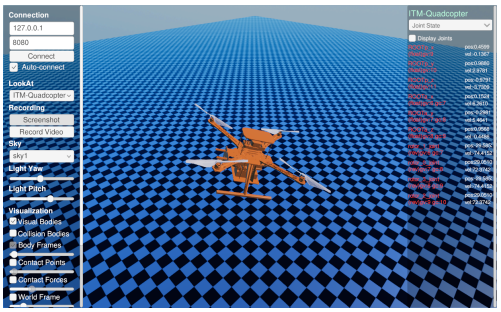
(b) Expert, $t = 0$ s



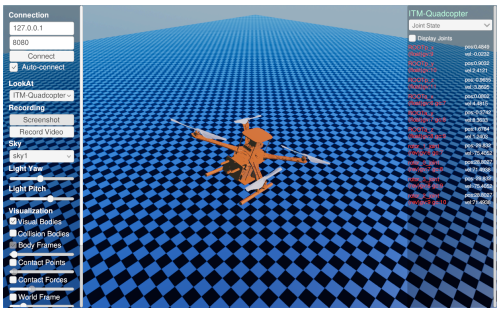
(c) RL-Agent, $t = 0.2$ s



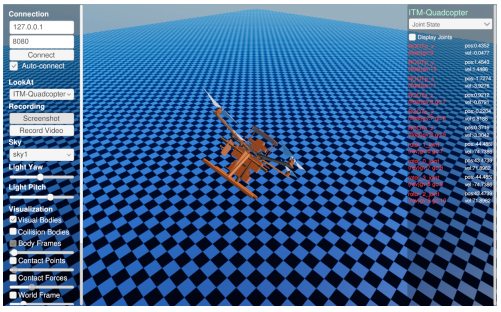
(d) Expert, $t = 0.2$ s



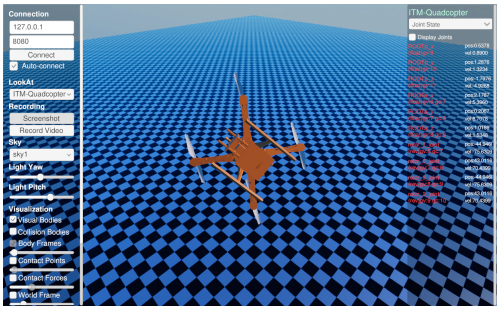
(e) RL-Agent, $t = 0.4$ s



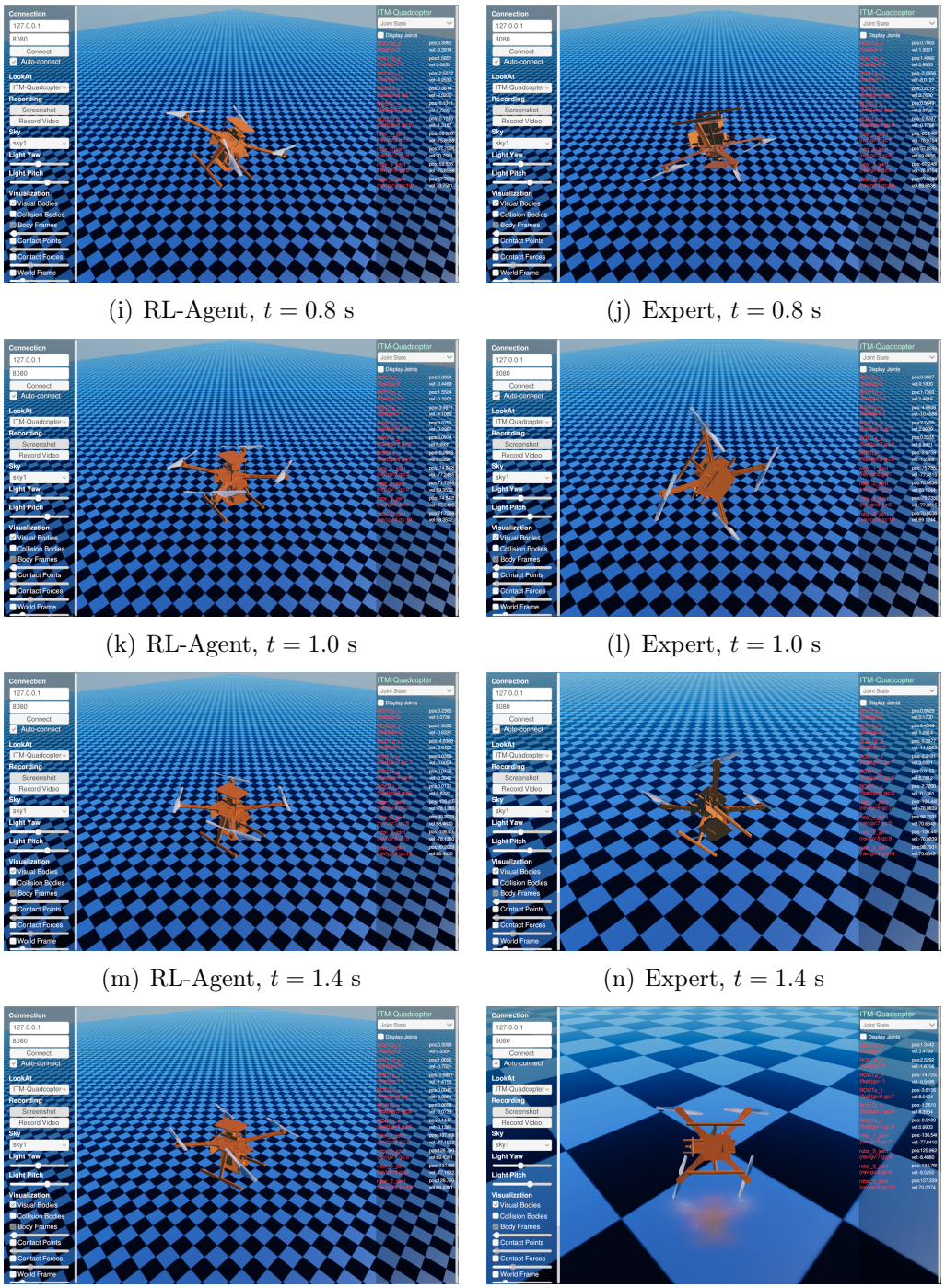
(f) Expert, $t = 0.4$ s



(g) RL-Agent, $t = 0.6$ s



(h) Expert, $t = 0.6$ s



(i) RL-Agent, $t = 0.8$ s (j) Expert, $t = 0.8$ s

(k) RL-Agent, $t = 1.0$ s (l) Expert, $t = 1.0$ s

(m) RL-Agent, $t = 1.4$ s (n) Expert, $t = 1.4$ s

(o) RL-Agent, $t = 1.8$ s (p) Expert, $t = 1.8$ s

Figure A.1: Motion sequences of the RL-agent (left column) and the expert (right column) in the state recovery task of Section 4.3.2, for initial state $\boldsymbol{s}_{init}^{(1)}$ with orientation $\boldsymbol{q}_{init}^{(1)} = [0, -1, 0, 0]^\intercal$ $\frac{\text{m}}{\text{s}}$, velocity $\boldsymbol{v}_{init}^{(1)} = [2, 2, 0]^\intercal$ $\frac{\text{m}}{\text{s}}$, and angular velocity $\boldsymbol{\omega}_{init}^{(1)} = [4, 4, 4]^\intercal$ $\frac{\text{rad}}{\text{s}}$.

A.7 Inhalt der CD-ROM

Die beigelegte CD-ROM enthält in der obersten Dateistruktur die Einträge

- **stud_512.pdf**: das PDF-File zur Studienarbeit STUD–512.
- **STUD_512/**: ein Verzeichnis mit den TEX-Dateien des in LaTeX verfassten Berichtes zur Studienarbeit STUD–512 sowie alle dazugehörigen Grafiken als *.eps und *.svg Dateien.
- **DATA/**: ein Verzeichnis mit den für diese Arbeit relevanten Daten, Hilfsprogrammen, Skripts und Simulationsumgebungen.

Zusätzliche Informationen stehen in den readme.txt-Dateien der jeweiligen Verzeichnisse zur Verfügung.

Bibliography

[AbadiEtAl15] Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2015.

[BlukisEtAl18] Blukis, V.; Brukhim, N.; Bennett, A.; Knepper, R.A.; Artzi, Y.: Following High-level Navigation Instructions on a Simulated Quadcopter with Imitation Learning. pp. 1–11, 2018.

[BrockmanEtAl16] Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W.: OpenAI Gym. CoRR, pp. 1–4, 2016.

[BruntonKutz19] Brunton, S.L.; Kutz, J.N.: Neural Networks and Deep Learning. Cambridge University Press, 2019.

[CarneyEtAl21] Carney, R.; Chyba, M.; Gray, C.; Wilkens, G.; Shanbrom, C.: Multi-Agent Systems for Quadcopters. Journal of Geometric Mechanics, pp. 1–28, 2021.

[ChanGolubLeVeque79] Chan, T.; Golub, G.; LeVeque, R.: Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. Tech. rep., 1979.

[CoumansEtAl13] Coumans, E.; et al.: Bullet physics library. Open source: bulletphysics.org, Vol. 15, No. 49, pp. 1–5, 2013.

[CS231n21] CS231n: Convolutional Neural Networks for Visual Recognition, https://cs231n.github.io, 2021.

[EmranNajjaran18] Emran, B.; Najjaran, H.: A Review of Quadrotor: An Underactuated Mechanical System. Annual Reviews in Control, Vol. 46, pp. 165–180, 2018.

[EngstromEtAl20] Engstrom, L.; Ilyas, A.; Santurkar, S.; Tsipras, D.; Janoos, F.; Rudolph, L.; Madry, A.: Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. CoRR, pp. 1–14, 2020.

[FranceschettiEtAl20] Franceschetti, A.; Tosello, E.; Castaman, N.; Ghidoni, S.: Robotic Arm Control and Task Training through Deep Reinforcement Learning. pp. 1–8, 2020.

[Fukushima80] Fukushima, K.: Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. Biological Cybernetics, pp. 193–202, 1980.

[GoecksEtAl20] Goecks, V.G.; Gremillion, G.M.; Lawhern, V.J.; Valasek, J.; Waytowich, N.R.: In Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Dense and Sparse Reward Environments. p. 465–473, Auckland New Zealand: International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[GoodfellowBengioCourville16] Goodfellow, I.; Bengio, Y.; Courville, A.: Deep Learning. MIT Press, 2016. Http://www.deeplearningbook.org.

[HoErmon16] Ho, J.; Ermon, S.: Generative Adversarial Imitation Learning. In Lee, D.; Sugiyama, M.; Luxburg, U.; Guyon, I.; Garnett, R. (Eds.): Advances in Neural Information Processing Systems, Vol. 29, pp. 1–9, Barcelona, Spain, 2016.

[HornikStinchcombeWhite89] Hornik, K.; Stinchcombe, M.; White, H.: Multilayer Feedforward Networks are Universal Approximators. Neural Networks, Vol. 2, No. 5, pp. 359–366, 1989.

[HubelWiesel59] Hubel, D.H.; Wiesel, T.N.: Receptive Fields of Single Neurones in the Cat's Striate Cortex. Journal of Physiology, pp. 574–591, 1959.

[HwangboEtAl17] Hwangbo, J.; Sa, I.; Siegwart, R.; Hutter, M.: Control of a Quadrotor with Reinforcement Learning. IEEE Robotics and Automation Letters, Vol. 2, No. 4, pp. 2096–2103, 2017.

[HwangboEtAl19] Hwangbo, J.; Lee, J.; Dosovitskiy, A.; Bellicoso, D.; Tsounis, V.; Koltun, V.; Hutter, M.: Learning Agile and Dynamic Motor Skills for Legged Robots. Science Robotics, Vol. 4, No. 26, pp. 1–20, 2019.

[HwangboLeeHutter18] Hwangbo, J.; Lee, J.; Hutter, M.: Per-Contact Iteration Method for Solving Contact Dynamics. IEEE Robotics and Automation Letters, Vol. 3, No. 2, pp. 895–902, 2018.

[IlyasEtAl18] Ilyas, A.; Engstrom, L.; Santurkar, S.; Tsipras, D.; Janoos, F.; Rudolph, L.; Madry, A.: A Closer Look at Deep Policy Gradients. CoRR, Vol. abs/1811.02553, pp. 1–27, 2018.

[ITM] ITM: Model Predictive Tracking Control of a Quadrotor in an Indoor Environment. Master's thesis. MSC-279.

[KimGadsdenWilkerson20] Kim, J.; Gadsden, S.A.; Wilkerson, S.A.: A Comprehensive Survey of Control Strategies for Autonomous Quadrotors. Canadian Journal of Electrical and Computer Engineering, Vol. 43, No. 1, pp. 3–16, 2020.

[KingmaBa14] Kingma, D.P.; Ba, J.: Adam: A Method for Stochastic Optimization. CoRR, pp. 1–15, 2014.

[KoenigHoward04] Koenig, N.; Howard, A.: Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2149–2154, Sendai, Japan, 2004.

[KrizhevskySutskeverHinton12] Krizhevsky, A.; Sutskever, I.; Hinton, G.E.: In ImageNet Classification with Deep Convolutional Neural Networks: Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc., 2012.

[Luukkonen11] Luukkonen, T.: Modelling and Control of Quadcopter. Tech. rep., Aalto University, School of Science, 2011.

[Ma19] Ma, Z.: The Function Representation of Artificial Neural Network. CoRR, pp. 1–25, 2019.

[Mellinger12] Mellinger, D.W.: Trajectory Generation and Control for Quadrotors. Ph.D. thesis, University of Pennsylvania, 2012.

[MeloMáximo19] Melo, L.C.; Máximo, M.R.O.A.: Learning Humanoid Robot Running Skills through Proximal Policy Optimization. In 2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE), pp. 37–42, Rio Grande, Brazil, 2019.

[MerabtiBouchachiBelarbi15] Merabti, H.; Bouchachi, I.; Belarbi, K.: Nonlinear model predictive control of quadcopter. In 2015 16th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA), pp. 208–211, 2015.

[MnihEtAl15] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; Hassabis, D.: Human-Level Control Through Deep Reinforcement Learning. Nature, Vol. 518, No. 7540, pp. 529–533, 2015.

[MnihEtAl16] Mnih, V.; Badia, A.P.; Mirza, M.; Graves, A.; Lillicrap, T.P.; Harley, T.; Silver, D.; Kavukcuoglu, K.: Asynchronous Methods for Deep Reinforcement Learning. CoRR, pp. 1–19, 2016.

[NagabandiEtAl17] Nagabandi, A.; Kahn, G.; Fearing, R.S.; Levine, S.: Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. CoRR, pp. 1–10, 2017.

[Nielsen15] Nielsen, M.: Neural Networks and Deep Learning. Determination Press, 2015.

[OpenAIEtAl19] OpenAI; Akkaya, I.; Andrychowicz, M.; Chociej, M.; Litwin, M.; McGrew, B.; Petron, A.; Paino, A.; Plappert, M.; Powell, G.; Ribas, R.; Schneider, J.; Tezak, N.; Tworek, J.; Welinder, P.; Weng, L.; Yuan, Q.; Zaremba, W.; Zhang, L.: Solving Rubik's Cube with a Robot Hand. CoRR, Vol. abs/1910.07113, pp. 1–51, 2019.

[PaszkeEtAl19] Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; Chintala, S.: In PyTorch: An Imperative Style, High-Performance Deep Learning Library: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc., 2019.

[PowersMellingerKumar15] Powers, C.; Mellinger, D.; Kumar, V.: Quadrotor Kinematics and Dynamics. Springer Netherlands, 2015.

[RossGordonBagnell11] Ross, S.; Gordon, G.; Bagnell, D.: A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In Gordon, G.; Dunson, D.; Dudík, M. (Eds.): Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, Vol. 15 of Proceedings of Machine Learning Research, pp. 627–635, Fort Lauderdale, USA, 2011.

[Ruder16] Ruder, S.: An Overview of Gradient Descent Optimization Algorithms. CoRR, pp. 1–14, 2016.

[RumelhartHintonWilliams86] Rumelhart, D.E.; Hinton, G.E.; Williams, R.J.: Learning Representations by Back-Propagating Errors. Nature, Vol. 323, No. 6088, pp. 533–536, 1986.

[Sammut10] Sammut, C.: Behavioral Cloning. Boston, MA: Springer US, 2010.

[SchulmanEtAl15] Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.; Moritz, P.: Trust Region Policy Optimization. In Bach, F.; Blei, D. (Eds.): Proceedings of the 32nd International Conference on Machine Learning, Vol. 37, pp. 1889–1897, Lille, France, 2015.

[SchulmanEtAl18] Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; Abbeel, P.: High-Dimensional Continuous Control Using Generalized Advantage Estimation. arXiv:1506.02438 [cs], pp. 1–14, 2018. ArXiv: 1506.02438.

[SilverEtAl14] Silver, D.; Lever, G.; Heess, N.; Degris, T.; Wierstra, D.; Riedmiller, M.: Deterministic Policy Gradient Algorithms. In Proceedings of the 31st International Conference on Machine Learning, Vol. 32, pp. 387–395, Bejing, China: PMLR, 2014.

[SilverEtAl16] Silver, D.; Huang, A.; Maddison, C.J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature, Vol. 529, pp. 484–503, 2016.

[SolaSevilla97] Sola, J.; Sevilla, J.: Importance of input data normalization for the application of neural networks to complex industrial problems. IEEE Transactions on Nuclear Science, Vol. 44, No. 3, pp. 1464–1468, 1997.

[Stanford Artificial Intelligence Laboratory et al.] Stanford Artificial Intelligence Laboratory et al.: Robotic Operating System.

[SuttonBarto18] Sutton, R.S.; Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, 2018.

[Szandała21] Szandała, T.: Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. Singapore: Springer Singapore, 2021.

[TodorovErezTassa12] Todorov, E.; Erez, T.; Tassa, Y.: MuJoCo: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference

on Intelligent Robots and Systems, pp. 5026–5033, Vilamoura-Algarve, Portugal, 2012.

[WangEtAl20] Wang, Q.; Ma, Y.; Zhao, K.; Tian, Y.: A Comprehensive Survey of Loss Functions in Machine Learning. Annals of Data Science, pp. 1–26, 2020.

[ZhaoQueraltaWesterlund20] Zhao, W.; Queralta, J.P.; Westerlund, T.: Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: A Survey. CoRR, pp. 1–8, 2020.

[ÅströmEtAl93] Åström, K.; Hägglund, T.; Hang, C.; Ho, W.: Automatic Tuning and Adaptation for PID Controllers - A Survey. Control Engineering Practice, Vol. 1, No. 4, pp. 699–714, 1993.

Erklärung

Hiermit versichere ich, dass

- ich die vorliegende Arbeit selbständig verfasst habe,
- ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe,
- ich die eingereichte Arbeit weder vollständig noch in Teilen bereits veröffentlicht habe,
- das elektronische Exemplar mit den anderen Exemplaren übereinstimmt.

Datum	Unterschrift