Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Ingvild Christoffersen Fleisje

Reward Shaping and Performance Analysis of Proximal Policy Optimization for Quadrotor Navigation in Environments of Varying Complexity

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Dr. Kostas Alexis
Co-supervisor: Mihir Kulkarni
June 2023

NTNU
Norwegian University of
Science and Technology

Ingvild Christoffersen Fleisje

Reward Shaping and Performance Analysis of Proximal Policy Optimization for Quadrotor Navigation in Environments of Varying Complexity

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Dr. Kostas Alexis
Co-supervisor: Mihir Kulkarni
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4900

Cybernetics and robotics, Master Thesis

Reward Shaping and Performance Analysis of Proximal Policy Optimization for Quadrotor Navigation in Environments of Varying Complexity

Ingvild Christoffersen Fleisje

26 June, 2023

Supervisor: Prof. Kostas Alexis
Cosupervisor: Mihir Kulkarni

Preface

This master thesis was written as part of a research project for the Autonomous Robots Lab (ARL) at the Department of Engineering Cybernetics Norwegian University of Science and Technology (NTNU), somewhat building on the work of earlier students and ARL workers. The lab aims to develop robotic systems able to perform complex and useful tasks in any environment. The thesis was motivated by the lab's initiative to explore learning-based methods, combined with or replacing the traditional control framework.

Reinforcement learning is not a part of the cybernetics curriculum, so an elaborate theory section is thus included in this thesis. I have tried my best to narrow this broad topic down, finding the important methods and understanding their differences. The comparable limited theory on Variational Autoencoders is partly taken from or inspired by earlier unpublished work in a specialization project I performed during the fall of 2022. These sections will be marked by (*).

Long training times and shared computational resources have made it difficult to perform as many experiments as desired. Furthermore, the simulation environment has demanded very specific dependencies with Python module versions, and unforeseen errors have occurred due to this and creating delays.

Acknowledgments

I would like to acknowledge my supervisor Prof. Dr. Kostas Alexis for his guidance and expertise in the field of quadrotor control and reinforcement learning. Also, a great thanks to my cosupervisor and Ph.D. candidate Mihir Kulkarni for all the technical assistance and helpful discussions. In addition, thanks to Gunnar Aske for providing access to the necessary computational resources to conduct the experiments. Finally, I thank friends and family for engaging words and support throughout my student career leading up to this final thesis.

Abstract

Developing autonomous aerial vehicles capable of executing fast and agile maneuvers in complex environments has been challenging. Traditional control relies on several modules that help sense, map, plan, and control the robot, each module adding delays and uncertainties that accumulate through the control system [1]. Their need for a system model makes them infeasible in uncertain or dynamic environments. The recent development of learning-based methods combined with deep neural networks, known as deep reinforcement learning (DRL), has allowed traditionally separate modules for motion control, perception, and prediction to be combined into a single model, capable of learning control policies directly from camera inputs [2]. The Proximal Policy Optimization (PPO) [3] algorithm has been a popular choice in DRL-based control tasks due to its state-of-the-art performance on several reinforcement learning benchmarks.

This thesis presents DRL-based controllers for autonomous quadrotor navigation in free and obstacle-filled simulator environments using the Proximal Policy Optimization (PPO) algorithm. Given the quadrotor's observations, including depth data for obstacle avoidance, the DRL controller output control commands to guide the quadrotor toward a three-dimensional target position. To minimize navigation delays, a variational autoencoder (VAE) is employed to compress the depth data into a low-dimensional representation.

The study investigated reward shaping and evaluation in the context of quadrotor navigation. The results shed light on the importance of carefully crafting reward functions, considering exploration strategies, and adjusting hyperparameters based on environmental complexity. The findings highlight the potential of combining reward terms to enhance convergence properties and the role of exploration in discovering optimal policies. However, limitations stemming from single test runs and challenges related to handling downward trajectories indicate areas for further investigation and improvement. This research contributes to the understanding of reward function design in reinforcement learning for quadrotor navigation and sets the stage for future advancements in this field.

Sammendrag

Utviklingen av autonome luftfartøy som kan utføre raske og smidige manøvrer i komplekse omgivelser har vært utfordrende. Tradisjonell kontroll er avhengig av flere moduler som oppfatter, kartlegger, planlegger og styrer roboten, hvor hver modul legger til forsinkelser og usikkerheter som akkumuleres gjennom kontrollsystemet [1]. Behovet for en systemmodell gjør tradisjonelle metoder uegnet i usikre eller dynamiske omgivelser. Nylig utvikling av læringsbaserte metoder kombinert med dype nevrale nettverk, kjent som Deep Reinforcement Learning (DRL), har gjort det mulig å kombinere tradisjonelt separate moduler for bevegelseskontroll, persepsjon og prediksjon i en enkelt modell, som er i stand til å lære kontrollstrategier direkte fra kameradata [2]. Algoritmen Proximal Policy Optimization (PPO) [3] har blitt et populært valg i DRL-baserte kontrolloppgaver på grunn av sin fremragende ytelse på flere benchmarks.

Denne masteravhandlingen presenterer DRL-baserte kontrollere for autonom navigasjon av en drone i både frie og hindringsfyllte omgivelser ved hjelp av PPO-algoritmen. Gitt dronens observasjoner, som inkluderer dybdedata for unnamanøvrering av hindringer, genererer DRL-kontrolleren styrekommandoer som veileder quadrotoren mot en målposisjon i tre dimensjoner. For å minimere navigasjonsforsinkelser benyttes en variational autoencoder (VAE) for å komprimere dybdedataene til en lavdimensjonal representasjon.

Studien undersøkte design av og evaluerte resultatene av forskjellige belønningsfunksjoner for navigasjonsproblemet. Resultatene belyser betydningen av å nøye utforme belønningsfunksjoner, vurdere utforskningsstrategier og justere hyperparametere basert på omgivelsenes kompleksitet. Funnene viser potensialet ved å kombinere ulike former for belønning for å forbedre konvergensegenskapene og rollen utforskning spiller i oppdagelsen av gode kontrollstrategier. Begrensninger i antallet utførte eksperimenter og utfordringer med å håndtere nedgående navigasjonsbaner utgjør et grunnlag for videre undersøkelse og forbedring. Resultatene bidrar til forståelse av designet av belønningsfunksjoner i reinforcement learning for dronenavigasjon i komplekse omgivelser og legger grunnlaget for fremtidige fremskritt på dette feltet.

Contents

Figures

Chapter 1

Introduction

1.1 Background and Motivation

A primary goal in the field of Artificial Intelligence (AI) is to develop fully autonomous agents that can interact with their environments and learn optimal behaviors through trial and error [4]. Achieving this goal has traditionally been challenging, limited by memory capacity and computational complexity. Reinforcement Learning (RL) has emerged as an important framework for addressing these challenges, particularly when combined with deep neural networks. RL has found applications in various domains, including robotics, autonomous driving, game theory, and economics [4, 5].

Unmanned aerial vehicles (UAVs) have gained significant attention due to their versatility and potential applications in numerous industries, such as surveillance, search and rescue, and package delivery. However, navigating UAVs safely and efficiently through complex, obstacle-filled environments remains a primary challenge. The traditional navigation frameworks often rely on predefined models or expert knowledge, which may not be adaptable to dynamic and unpredictable scenarios [1]. Moreover, classical motion planning and trajectory generation operate as pipeline processes, where each module introduces errors that accumulate and cause delays, hindering fast flight.

Deep reinforcement learning (DRL), combining RL with powerful function approximations such as neural networks (NNs), has overcome many shortcomings of traditional navigation [1, 6]. Replacing or integrating DRL-based modules into the traditional navigation framework has allowed us to learn control policies directly from camera inputs [4], enabling UAVs to learn how to autonomously adapt their navigation strategies based on environmental feedback without the need for a map [6–8].

Unlike other machine learning classifications, reinforcement learning relies on a reward

signal for providing feedback to the agent [2]. Reward functions guide the learning process by providing feedback and shaping the agent's behavior. Therefore, careful reward shaping plays an important role in training robots to learn desired behaviors.

The reinforcement learning algorithm Proximal Policy Optimization (PPO) [3] has been successful in a wide range of reinforcement learning tasks, achieving state-of-the-art performance on several benchmarks. Its comparative simplicity, robustness, and effectiveness make them a particularly popular choice in control tasks.

Motivated by the limitations of traditional control methods and the powerful potential of deep reinforcement learning, this thesis implemented a learning-based control system for goal-based quadrotor navigation in environments of varying complexity. From experiments conducted with different reward functions in a simulator environment, this thesis provides insight into the impact of reward function design on the performance of PPO for the navigation task. Additionally, it investigates the trade-offs between safety, efficiency, and robustness for different obstacle densities.

1.2 Problem Statement

This thesis aims to use proximal policy optimization (PPO) to solve a map-free navigation task in free and cluttered environments. The quadrotor is assumed to be equipped with sensors measuring the position, orientation, linear velocities, and angular velocities of the quadrotor and with a depth camera for the obstacle avoidance task. This creates a two-folded problem in (1) designing a learning-based agent for discovering an optimal control policy that, given the observed measurements, yields control commands that safely guide the quadrotor to its target position and (2) designing a depth compression module for learning a suitable low-dimensional representation of the perception data to lower the input dimension and total system delay.

Specifically, (1) given the quadrotor states, i.e., measurements, the control policy should output the desired orientation and/or angular velocities to be used in traditional low-level controllers to produce thrust and torque commands for guiding the quadrotor to its target. (2) Given a depth image, the depth learning network should produce a lower dimensional vector that captures the important features of the data.

Furthermore, the thesis involved designing the reward function of PPO to yield safe and efficient point-to-point navigation in different environments while respecting the quadrotor's physical constraints. The main objectives were to:

1. Investigate how different reward functions affect the performance of the PPO al-

gorithm for different complexity environments.
2. Analyze limitations in performance and the role of exploration in different-complexity environments.

1.3 Scope

We are dealing with a large and complex system, so diving into every part is unfeasible. This thesis will present the proposed DRL-based controller and depth learning network in detail while giving little attention to the low-level controllers. Changing and tuning the network structures (VAE and AC) was out of the scope of this thesis, yet it will be commented on.

PPO should be easy to manually tune in the sense that one knows exactly what each parameter represents. However, after a few failed attempts that yielded poorer results than the recommendation found in the literature, this was found to be too time demanding and considered out of scope for this thesis. The hyperparameters of PPO were thereby chosen based on recommended values.

The focus was reward function shaping, focusing on changing the design of the position reward and exploration term in the reward function. However, due to limited time and computational resources, the conducted experiments had to be very selective and limited to only one run per reward shape in each environment.

1.4 Outline

The outline of this thesis will be as follows:

- **Chapter 1: Introduction -** motivates the thesis, along with the objectives, scope, and outline.
- **Chapter 2: Reinforcement Learning -** introduces the Reinforcement Learning framework and its traditional tabular methods.
- **Chapter 3: Deep Reinforcement Learning -** extends traditional RL framework to solve continuous problems by the use of deep neural networks.
- **Chapter 4: Proximal Policy Optimization -** provides details on the PPO algorithm and the theories it builds upon.
- **Chapter 5: Related Work -** presents some of the main developments in learning-based control and papers where similar problems have been studied.
- **Chapter 6: Method -** gives the problem formulation and presents the network designs, implementation, and experimental setup.

- **Chapter 7: Results -** presents and comments the results.
- **Chapter 8: Discussion -** discusses the results.
- **Chapter 9: Conclusion -** summarizes the main findings.

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) is a machine learning framework for developing intelligent agents that can learn to make decisions and take actions based on feedback from their environment [5]. RL is inspired by how humans and animals learn from trial-and-error experiences to adapt to their surroundings. In RL, an agent interacts with an environment by taking actions, receiving feedback as rewards or penalties, and updating its behavior accordingly. This chapter introduces the RL framework, some of its traditional tabular methods, and their advantages and drawbacks.

2.1 Introduction

Reinforcement learning is, together with unsupervised and supervised learning, a classification of Machine learning. While unsupervised and supervised learning focus on analyzing data to identify patterns and make predictions, respectively, RL is distinct in its emphasis on goal-directed learning from interaction [5]. RL typically involves sequential decision-making where feedback is given through a reward signal, not as supervised labels [2]. Thus, designing the reward function is an important aspect of the success of a reinforcement learning algorithm.

Reinforcement learning is a framework for training and optimizing experience-driven agents that learn incrementally through trial and error until they achieve optimal behavior [4]. A learning agent must be able to sense the state of the environment, take actions that affect the state, and have a goal related to the environment [5]. Through interacting with the environment, the agent can learn to take actions that maximize its cumulative reward over time, leading to improved performance in achieving its goal.

2.2 Reinforcement Learning Problem

The reinforcement learning problem is "the problem of learning from interaction to achieve a goal" [5]. The decision-maker, or *agent*, interacts with its *environment* by observing it and taking actions based on its observations. Each chosen action results in a change in the environment and provides the agent with a reward. The agent wants to maximize its rewards over time.

A reinforcement learning system consists of an agent, an environment, a policy, a reward signal, a value function, and, optionally, a model of the environment [5].

2.2.1 Policy

The agent behaves or chooses actions according to the *policy* it follows [2, 5]. A policy π is a mapping from states to their corresponding available actions. Policies can be deterministic or stochastic. Deterministic policies map states to actions directly $A_t = \pi(S_t)$, while stochastic policies represent probabilities of choosing each possible action from a given state, i.e., the probability $\pi(A_t = a|S_t = s)$. The agent may change its policy as it gathers experience.

2.2.2 Reward and Return

The *reward signal* defines the goal of a reinforcement learning problem by specifying what good and bad actions for the agent are [5]. At each time step, the agent receives a numerical reward $r_t \in \mathbb{R}$, indicating the immediate desirability of the current state or state-action pair. The reward signals can generally be considered stochastic functions of both the environment's state and the actions taken.

RL aims to maximize the total reward received in the long run [5]. In practice, the goal is to maximize the *expected return*. The return G_t is a function of the sequence of future rewards and is popularly given as the discounted accumulated reward [2, 5]:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

Where $\gamma \in [0, 1]$ is a discount factor determining how much we value more immediate versus distant rewards.

2.2.3 Value Function

The *value function* indicates what behavior is good in the long run [5]. It predicts the expected, accumulative, discounted future reward by considering the states likely to follow and the rewards available in those states [2, 5]. The most important component of almost all reinforcement learning algorithms is a method for efficiently estimating values [5].

Value functions can measure how well each state, or state-action pair, is [2]. These two functions are referred to as the state-value function (2.2) and the action-value function or Q-function (2.3), respectively.

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \tag{2.2}$$

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{2.3}$$

with G_t as given in (2.1).

While the reward signal is the primary basis for altering the policy, an efficient value estimation method is essential for making and evaluating decisions [5]. As we mainly care about long-time rewards, action choices are made based on which next state yields the highest value estimations rather than the highest reward. The environment directly gives rewards. However, values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime, making policy learning a complex task.

2.2.4 Model

The *model* of the environment is an optional component of some RL systems [5]. It mimics the behavior of the environment and allows inferences to be made about how the environment will behave in response to the agent's actions. Models are used for planning, which involves deciding on a course of action by considering possible future situations before they are experienced. Reinforcement learning systems that use models and planning are called model-based methods, while simpler model-free methods are explicitly trial-and-error learners.

2.2.5 Exploration Vs. Exploitation

In reinforcement learning, an agent must balance exploring uncertain policies with exploiting the current best policy [2]. Exploration involves taking a random action to gain

more knowledge about the environment, while exploitation means choosing the action that seems to give the highest future reward.

One simple approach is the ϵ-greedy algorithm that selects a random action (explores) with probability ϵ and a greedy action (exploits) with probability $1-\epsilon$ [2]. $\epsilon \in (0,1)$ is usually a small value close to zero s.t. the agent will exploit its knowledge of the environment most of the time. One can improve this method by incrementally decreasing epsilon as the agent learns about its environment to gradually reduce the likelihood of exploration.

2.3 Finite Markov Decision Processes

This section covers finite Markov decision processes as a tool to formalize sequential decision-making and mathematically formulate the reinforcement problem.

2.3.1 Agent-Environment Interaction

The agent's goal is to learn a policy (control strategy) π that maximizes the expected return G_t (cumulative, discounted reward) [4]. Every interaction with the environment yields information, which the agent uses to update its knowledge to reach its goal.

Figure 2.1 illustrates how an RL agent interacts with its environment over time. At each time step t, the agent observes its environment as being in a state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states [2, 5]. For the given state, the policy returns an action $A_t \in \mathcal{A}$ from the set of actions available from the current state $\mathcal{A}(s)$ [4, 5]. Finally, the environment transitions to the next state S_{t+1}, and the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$. This continues until some termination criterion is met; for instance, convergence to optimal behavior or a maximum number of time steps reached.

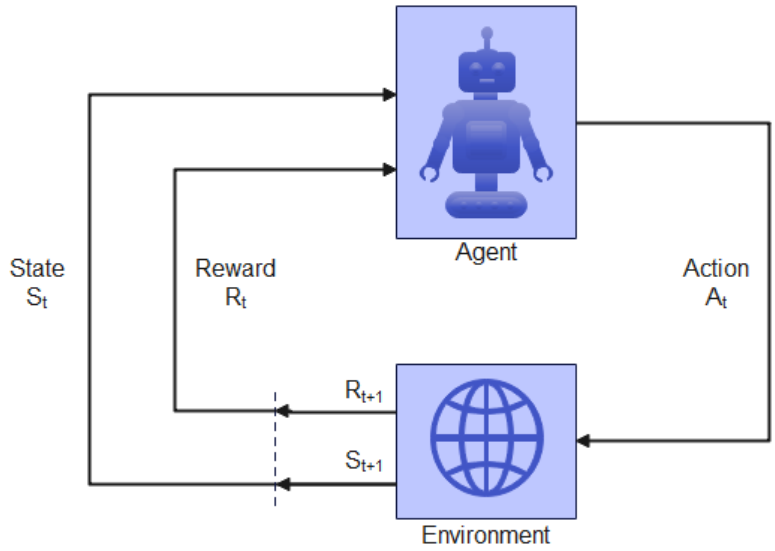


Figure 2.1: Agent-environment interface - The agent senses its environments state at time t, S_t, and takes action A_t. This results in a change in the environment state and provides the agent with a numerical reward R_t.

2.3.2 Markov Decision Processes and Markov Property

Reinforcement learning adopts the formal framework of Markov decision processes (MDP) to specify the relationship between a learning agent and its environment regarding states, actions, and rewards [5]. An RL problem can be formulated as a Markov decision process if it satisfies the *Markov property*, i.e., that the future only depends on the current state and action [2, 4]. A state signal with this property holds all necessary information about the past agent-environment interaction to understand the current state of the environment and that which might affect the future [5].

An MDP is defined as the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ [2], where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{P}(S_{t+1}|S_t, A_t)$ is the state transition probability distribution, $\mathcal{R}(S_t, A_t, S_{t+1}$ is the reward function, and $\gamma \in [0,1]$ is the discount factor.

2.3.3 Finite Markov Decision Processes

Many machine learning problems can be modeled as a finite Markov decision process [5]. Finite MDPs are defined by their finite state and action spaces and given by the dynamics of the environment at one time step:

$$p(s', r|s, a) \doteq \Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{2.4}$$

These one-step dynamics are the probability of each possible next state s' and reward r pair given any current state s and action a.

From the dynamics (2.4), we can calculate the state-transition probabilities (2.5), and the expected rewards for state-action pairs (2.6) and state-action-next-state triplets (2.7) [5].

$$p(s'|s,a) = \Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r\in\mathcal{R}} p(s', r|s, a) \tag{2.5}$$

$$r(s,a) = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a\right] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a) \tag{2.6}$$

$$r(s,a,s') = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'\right] = \frac{\sum_{r\in\mathcal{R}} r p(s', r|s, a)}{p(s', r|s, a)} \tag{2.7}$$

2.3.4 Recursive Property of Value Functions, and the Bellman Equation

Value functions are recursive, and for any policy π and any state s we have [5]:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t = s\right] && (2.8)\\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] && (2.9)\\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s\right] && (2.10)\\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s\right]\right] && (2.11)\\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] && (2.12)
\end{aligned}$$

The last equation (2.12) is known as the Bellman equation for v_π and works as a basis in many methods that compute, approximate, or learn v_π [5]. The Bellman equation expresses the relation between the value of a state and its successors. It weights all the possible actions by their occurrence probability and averages over them. The equation states that the value of the initial state is equal to the expected discounted value of the next state plus the anticipated reward obtained during the transition.

2.3.5 Optimal Policies and Value Functions

The aim of RL resembles that of an optimal control problem [4]. The agent's goal is to learn an optimal policy π_* that returns an action for each given state that will maximize the expected (discounted) return. In finite MDPs, we can define an optimal policy (2.13) as any policy that is better than or equal to all other policies [4, 5]. A policy is optimal, i.e. $\pi \geq \pi'$, if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all states $s \in \mathcal{S}$.

$$\pi^* = \arg\max_\pi \mathbb{E}[G|\pi] \tag{2.13}$$

An optimal policy has a corresponding optimal state-value function v_* (2.14) and optimal action-value function q_* (2.15).

$$v_*(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \tag{2.14}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s) \tag{2.15}$$

The optimal q-function can be written in terms of the optimal value function:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{2.16}$$

2.3.6 The Bellman Optimality Equation

The Bellman optimality equation describes the optimal value of a state or action in a Markov decision process (MDP). It provides a way to compute the maximum expected cumulative reward an agent can obtain by following the optimal policy. The Bellman optimality equation for v_* is given by [5]:

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}|S_t = s, A_t = a] \tag{2.17}$$

$$= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_*(s')\big] \tag{2.18}$$

The Bellman equation is a system of equations, one for each state. It is not dependent on a specific policy as, under an optimal policy, the value of a state is equal to the expected return for the best action available from that state [5]. For finite MDPs, equation (2.17) has a unique solution independent of the policy.

The Bellman optimality equation for q_* is given by:

$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\Big] \tag{2.19}$$

$$= \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big] \tag{2.20}$$

If the dynamics of the environment are known, the Bellman equation system can be solved with a nonlinear equation-solving method [5]. When the optimal value function is computed, it is quite straightforward to determine an optimal policy, as the actions that appear best after a one-step search will be optimal. This becomes even easier when we have q_* because, in this case, the agent can look for an action that maximizes $q_*(s,a)$ without doing the one-step search. Using the q-function, the dynamics of the environment can be completely unknown.

Solving an RL problem using explicit computation of the Bellman optimality equation requires the assumptions of; (1) fully known environment dynamics, (2) sufficient computational resources to compute a solution, and (3) the Markov property [5]. In practice, all these assumptions are rarely true simultaneously. Therefore, many methods aim to solve the Bellman optimality equation approximately.

2.3.7 Episodic Vs. Continuing Tasks

An RL problem can be continuous or episodic depending on the task [2]. When the agent–environment interaction naturally breaks down into a sequence of separate episodes, we have an episodic RL problem [5]. The sequence of states, actions, and rewards in an episode form a trajectory of the policy [4]. Often, there is no need to distinguish between different episodes, meaning we can consider only one episode to state something about them all [5].

The state is reset after every T-th time step, creating episodes of length T.

When the agent–environment interaction is a continuing task without natural divisions, we call the task continuing [5]. In the case of continuous tasks, the final time step $T = \infty$, but the sum of rewards will be finite as long as $\gamma < 1$.

The return has several different definitions depending upon the nature of the task and whether one wishes to discount delayed reward. The undiscounted formulation is appropriate for episodic tasks, while the discounted formulation is appropriate for continuing tasks.

It is possible to obtain a common notation of return for episodic and continuous tasks [5]. This is done by introducing the *absorbing state* that marks the end of an episode, only

transitions to itself, and generates zero rewards.

The return, valid for both episodic and continuous RL problems, can be given as in Equation 2.1 or alternatively as:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{2.21}$$

2.3.8 Off-Policy and On-Policy Methods

A classification of RL methods is whether they are on-policy or off-policy [5]. On-policy methods focus on evaluating or improving the decision-making policy used by the agent. In contrast, off-policy methods aim to evaluate or improve a different policy than the one generating data samples. In on-policy methods, the target policy aligns with the agent's current policy, driving continuous exploration and improvement. Conversely, off-policy methods involve an initial exploration phase followed by learning a deterministic optimal policy that may be unrelated to the policy followed by the agent.

Off-Policy Methods

Off-policy methods reuse previously collected data by training a Q-function that satisfies Bellman's optimality equations [9]. These methods can leverage data obtained at any time, irrespective of the exploration strategy used during data collection. While off-policy methods offer improved data efficiency, they often face challenges related to optimization stability [10]. Although satisfying Bellman's equations may not guarantee optimal policy performance [9], off-policy algorithms can still exhibit impressive empirical performance and sample efficiency. The lack of theoretical guarantees can introduce potential instability in practical applications.

On-Policy Methods

On-policy methods directly optimize the policy's performance objective without relying on past data [9]. On-policy algorithms update the policy using only the most recent data collected, ensuring a more stable learning process [9]. However, the use of data only once makes on-policy methods tend to be less sample efficient [2]. Stability is prioritized in on-policy approaches, often at the expense of sample efficiency. Various methods have been developed to help enhance sample efficiency in on-policy methods, such as Vanilla Policy

Gradient (VPG), Trust Region Policy Optimization (TRPO), and Proximal Policy Optimization (PPO).

2.4 Tabular Solution Methods

Tabular RL methods solve the RL problem directly by explicitly modeling the state, action, and reward dynamics while representing and storing the values or policies for each state-action pair in a tabular format. These methods can be divided into dynamic programming (DP), Monte Carlo (MC) methods, and temporal-difference (TD) learning, where the latter is a combination of the first two [5].

2.4.1 Dynamic Programming

Dynamic programming (DP) is a method for solving optimal control problems represented by finite MDPs in RL [2, 5]. With a complete knowledge of the MDP to solve, including a perfect model, DP can be used to compute optimal policies and/or value functions. Two popular DP methods are policy iteration and value iteration. Both policy and value iteration bootstraps its current estimate using previously calculated values to increase sample efficiency and capture long-term trajectory information [11].

DP algorithms are obtained by turning Bellman equations into update rules for improving approximations of the desired value functions [5]. The Bellman equation gives recursive decomposition, and the value function stores and reuses sub-solutions, s.t. MDPs satisfy the properties needed to use DP.

Policy iteration

Policy iteration (PI) is a method used in reinforcement learning to find an optimal policy for a given Markov decision process [2, 12]. PI searches for the optimal policy by iterating through many policies to find the one with the highest return [11].

Policy iteration consists of two steps - policy evaluation and policy improvement - being repeated until convergence [2, 11]. In policy evaluation, the value function is estimated and updated according to the following update rule [5]:

$$v_{k+1}(s) = \mathbb{E}_\pi [R_{t+1} + \gamma v_k(S_{t+1}|S_t = s] \tag{2.22}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \big[r + \gamma v_k(s') \big] \tag{2.23}$$

In policy improvement, actions are selected greedily based on the current value function to generate a better policy [2, 5]. Policy iteration will ultimately achieve convergence towards an optimal policy and value function through this process. The policy update is given by:

$$\pi'(s) = \arg\max_a q_\pi(s,a) \tag{2.24}$$
$$= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(s_{t+1}|s_t = s, a_t = a] \tag{2.25}$$
$$= \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.26}$$

The iterative nature and exact convergence of policy iteration make it computationally expensive and, therefore, rarely used in practice [5, 11].

Value iteration

Value iteration can be seen as policy iteration where the policy evaluation terminates after a single update of each state [5, 11]. It aims to find the optimal policy by determining the optimal value functions and extracting the policy based on the highest values associated with each state. However, this extraction process requires a perfect system model. If a model is available, the state transition probabilities can be determined, and actions with the highest probabilities can be selected to transition to states with high values. Without a model, the focus shifts to identifying $Q(s,a)$ instead of extracting the optimal policy.

The policy improvement update is the same as for policy iteration in equation (2.24) [5]. The policy evaluation step updates the value (2.27) and action-value (2.29) functions according to:

$$v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(s_{t+1}|s_t = s, a_t = a] \tag{2.27}$$
$$= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{2.28}$$

$$q_{k+1}(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a_{t+1}} q_k(s_{t+1}, a_{t+1})] \tag{2.29}$$

Advantages and drawbacks of DP methods

Dynamic programming (DP) provides exact solutions to optimal policies but is not widely used in practice [5, 11]. They suffer from "the curse of dimensionality," implying that the computational demands increase exponentially with the number of state variables. Furthermore, they require a perfect model of the environment, which is often intractable, and their updates are often biased due to bootstrapping [11].

2.4.2 Monte Carlo Methods

Monte Carlo (MC) methods are a class of reinforcement learning algorithms that estimate value functions and find optimal policies through sample episodes [5]. Unlike dynamic programming, we do not need complete knowledge of the environment as MC methods learn from experience obtained by sampling sequences of states, actions, and rewards from actual or simulated environment interactions. The convergence towards optimal solutions is achieved through general policy iteration (GPI) using the sampled experience.

MC methods find the optimal policy by sampling multiple state-action-reward sequences and computing average returns [11]. The average returns are updated after each trajectory and as more samples are collected, the estimated values converge to the true values; $v_k(x) \to v_\pi(x)$ for all $x \in X$.

MC methods depend on exploration as the system model is unavailable to the agent [11]. Through exploration, the agent discovers the value functions, including state transition probabilities and rewards, for each action in every state, ultimately leading to the discovery of the optimal policy. Typically, exploration is initiated by starting in a random state at the beginning of each episode.

The nature of the MC updates makes these methods most suitable for episodic tasks [11]. In episodic tasks, the value functions can be naturally updated after each terminal state.

The way value functions and corresponding policies interact to attain optimality is similar to DP. However, instead of computing the value functions from knowledge of the MDP, MC methods learn value functions from sample returns [5].

There are two prediction problems in Monte Carlo methods; state-value prediction and action-value prediction.

State-value prediction

The first prediction problem aims to learn the state-value function for a given policy [5]. Monte Carlo methods estimate the expected return by averaging the returns observed in each occurrence of a state. The average should converge to the expected value as more and more returns are observed.

Action-value prediction

When a model is unavailable, state values alone are insufficient to determine a policy [5]. In this case, we estimate action values rather than state values, known as action-value prediction. Action-value prediction involves estimating $q_\pi(s, a)$, the expected return for starting in state s, taking action a, and following policy π. The Monte Carlo methods for action values are similar to those for state values, but they focus on occurrences of state-action pairs rather than states. As for state-value prediction, convergence is achieved as the number of visits to state-action pairs increases.

Advantages and drawbacks of MC methods

Monte Carlo (MC) methods can achieve optimal behavior without prior knowledge of the environment's dynamics [11], making them effective in scenarios where explicit modeling is challenging [5]. They can be used with simulation or sample models, allowing for easy generation of sample episodes even when constructing explicit transition probability models is difficult. Additionally, MC methods can focus on specific subsets of states, efficiently evaluating regions of interest without evaluating the entire state set. Finally, MC methods are less affected by violations of the Markov property since they do not rely on value estimates of successor states, avoiding the issues associated with bootstrapping.

A challenge in MC methods arises when many state-action pairs are never visited [5]. Continual exploration can address this problem by starting episodes in various state-action pairs with nonzero probabilities, but it may not always be a reliable solution. Alternative approaches, such as stochastic policies with a nonzero probability of selecting all actions in each state, are commonly employed to ensure exploration in practice.

2.4.3 Temporal-Difference Learning

Temporal-difference (TD) learning is an efficient method for solving the Bellman optimality equations to estimate optimal value functions [11]. It combines the ideas of dynamic

programming (DP) and Monte Carlo (MC) methods, meaning it learns directly from experience as in MC, and estimates are updated at each timestep based on other learned estimates as for DP [5].

In contrast to Monte Carlo methods that only update their value estimate at the end of an episode, TD methods update the value function estimate V at each timestep according to [5]:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{2.30}$$

$$\leftarrow V(s_t) + \alpha[\underbrace{R_{t+1} + \gamma V(s_{t+1}) - V(s_t)}_{\text{TD-Error } \delta_t}] \tag{2.31}$$

where δ_t is the TD error.

2.4.4 Summary Statistics of Tabular Methods

Table 2.1 gives an overview of some of the characteristics of DP, MC, and TD methods.

	DP	MC	TD
Model	Yes	No	No
Bias	High	Low	High
Variance	Low	High	Low
Computational cost	High	Medium	Low
Bootstrapping	Yes	No	Yes

Table 2.1: Comparison of DP, MC, and TD methods

Chapter 3

Deep Reinforcement Learning

This chapter will explain the power of combining reinforcement learning with deep neural networks, creating the deep reinforcement learning (DRL) framework. The focus will be on methods that learn stochastic policies and, optionally, value functions.

3.1 Background and Motivation

The rise of deep learning (DL) has significantly impacted many areas of machine learning, including reinforcement learning [4]. Deep neural networks have enabled RL methods to solve earlier intractable problems and have extended the applications of RL to complex robotic tasks [2]. It has been an important step towards building autonomous systems with a good visual understanding and producing control policies directly from sensor data [1].

3.1.1 Limitations of Traditional Reinforcement Learning

Originally, RL was proposed to solve MDPs, which are discrete representations of the optimal control problem [11]. Many of the traditional RL methods, including the ones presented in section 2.4, only work in environments with a relatively small, finite set of discrete states and actions [5]. In control applications, the states and actions are often multi-dimensional and continuous, such that discretizing the action space would be infeasible due to exponential growth in computational complexity and memory requirements for each degree of freedom [13]. Furthermore, a too-rough discretization of the action space might result in poor and unsmooth behavior, making it difficult to perform complex maneuvers and properly respect the physical constraints of the controlled agent.

3.1.2 Extending RL to Continuous Control Problems

A solution to the "curse of dimensionality" problem described above is deep reinforcement learning, where we approximate the policy and/or value functions through function approximation methods [5, 13]. The key issue of function approximation is how to generalize from past experiences with similar states to the current state, allowing for intelligent decision-making even when encountering new and unfamiliar states. Functional approximators attempt to solve the generalization problem by creating an approximation to some desired function from examples generated by the said function in order to save computational resources [5]. Some generalizing function approximators used in RL have been neural networks, decision trees, and the value-function approach [14], where neural networks are the most common choice.

To overcome the limitations of traditional reinforcement learning, many recent methods aim to discover a suitable parameterization of the policy and/or value function through the use of neural networks (NNs) [13]. These methods include Deep Q-Networks (DQN) [15], Trust Region Policy Optimization (TRPO) [12], Proximal Policy Optimization (PPO) [3], and the Deep Deterministic Policy Gradient (DDPG) algorithm [13]. These techniques leverage the representational power of neural networks to approximate the policy and/or value function, enabling more efficient learning in environments with continuous state and action spaces.

3.2 Policy Approximation Methods

Policy approximation methods aim to represent the policy using a parameterized function. The primary goal of policy approximation methods is to optimize the policy parameters to maximize the expected cumulative reward. This optimization is typically achieved through iterative updates, adjusting the policy parameters based on the observed rewards and feedback from the environment.

3.2.1 Parameterized Policy

The conventional method of approximating and using a value function to derive a policy has been proven theoretically intractable [14]. Rather than using a value function to estimate the expected return of each state or action, policy approximation methods learn a parameterized policy π_θ (3.1) that maps states or observations to actions [5]. The deterministic policy in value-based methods is replaced with a stochastic policy describing

the probability that action a is taken at time t given an environment in state s and policy parameter vector $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. Instead of the policy returning a single action directly, actions are sampled from the probability distribution $a_t \sim \pi_{\boldsymbol{\theta}}$.

$$\pi(a|s, \boldsymbol{\theta}) = P(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}) \tag{3.1}$$

We often represent the policy as a neural network where the state is given as input, and the network outputs the probabilities of selecting each action [14]. The policy parameters $\boldsymbol{\theta}$ correspond to the network weights, and the policy can be tuned and improved directly through its parameters. Sometimes the value function is used to learn the policy parameter vector but is not needed for action selection as it is in value-based methods [5]. In this case, the weight vector of the value function is denoted $\boldsymbol{w} \in \mathbb{R}^d$.

3.2.2 Kullback-Leibler Divergence

The Kullback-Leibler (KL) divergence measures the difference between two logarithmic probability distributions [16]. The KL divergence is always non-negative and is equal to zero if and only if P and Q are the same distribution in the case of discrete variables or equal "almost everywhere" in the case of continuous variables. Thus, it can be interpreted in a way as the distance between these distributions.

$$D_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{P(y)}\right] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{3.2}$$

Minimizing the KL divergence is equivalent to minimizing the cross entropy between distributions [16]. In policy approximation methods, reducing the KL divergence between the current policy distribution and a target distribution is beneficial for updating the policy toward an optimal state. This is commonly achieved through gradient-based optimization algorithms that adjust the policy parameters in the direction that reduces the KL divergence. By providing a measure of the disparity between the current and desired distributions, the KL divergence guides the policy updates to converge to an optimal policy.

3.2.3 Policy Gradients Methods

Policy gradient methods, presented by Sutton et al. [14], provide a popular solution to the policy approximation problem, where the policy (and possibly value function) parameters are updated using the gradient of some performance measure.

In policy gradient methods, the stochastic policy parameters $\boldsymbol{\theta}$ are updated through an approximate gradient of some objective $J(\boldsymbol{\theta})$ [5, 14]:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{3.3}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^d$ is a stochastic estimate of the gradient of $J(\boldsymbol{\theta})$ w.r.t. the current policy parameter $\boldsymbol{\theta}_t$. The learning rate $\alpha \in (0, 1]$ decides the impact the computed gradient has on the new policy parameters. The update rule provides a unique weight update for each element in $\boldsymbol{\theta}$ and guarantees convergence to a locally optimal policy.

The performance gradient with respect to the policy parameter for any MDP can be estimated by results of the policy gradient theorem:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s p(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \tag{3.4}$$

$$= \mathbb{E}_{\pi, s_t \sim p(s)} \left[\sum_a Q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \right] \tag{3.5}$$

where $p(s)$ is the on-policy distribution under π, and the proportionality is a constant equal to the average episode length for the episodic case and equality for the continuous case [5]. The right-hand side is the sum of states weighted by how often they occur under the target policy. Not knowing the environment model, i.e., the probabilities $p(s)$, is not a problem as the gradient does not depend on these. We only need an explicitly defined policy that is differentiable w.r.t. $\boldsymbol{\theta}$.

Advantages and limitations of policy gradient methods

Policy gradient methods have many advantages. In addition to handling continuous action spaces and model-free tasks, they can "learn appropriate levels of exploration and approach deterministic policies asymptotically" [5]. However, policy gradient methods tend to have poor sample efficiency. If many samples are used in the gradient update, this can improve learning efficiency but result in poor generalization. Using a limited set of samples could lead to better generalization but will increase training times.

3.2.4 Actor-Critic Methods

Actor-Critic (AC) methods combine the benefits of policy-based and value-based methods [5]. They learn approximations to policy and value functions, thus extending the idea of

policy parameterization in policy gradient methods to also parameterize the value or action-value function. The learned policy $\pi(a|s,\boldsymbol{\theta})$ selects actions and is referred to as the *actor*, while the estimated value function $\hat{V}_\pi(s,\boldsymbol{w})$ criticizes the chosen actions and is known as the *critic*. Here, $\boldsymbol{\theta}$ are the actor parameters, and $\boldsymbol{w}$ are the critic parameters. Learning is always on-policy, so the critic must learn about and critique whatever policy is currently being followed by the actor.

The critic is either a value or state-value function that evaluates if the selected action resulted in better or worse results than expected [5]. This can be measured by the TD error for value or action-value function:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V(S_t) \tag{3.6}$$

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \tag{3.7}$$

If the TD error is positive, it suggests increasing the probability of selecting action A_t in the future. If the TD error is negative, it suggests decreasing the probability.

Figure 3.1 shows the actor-critic architecture.

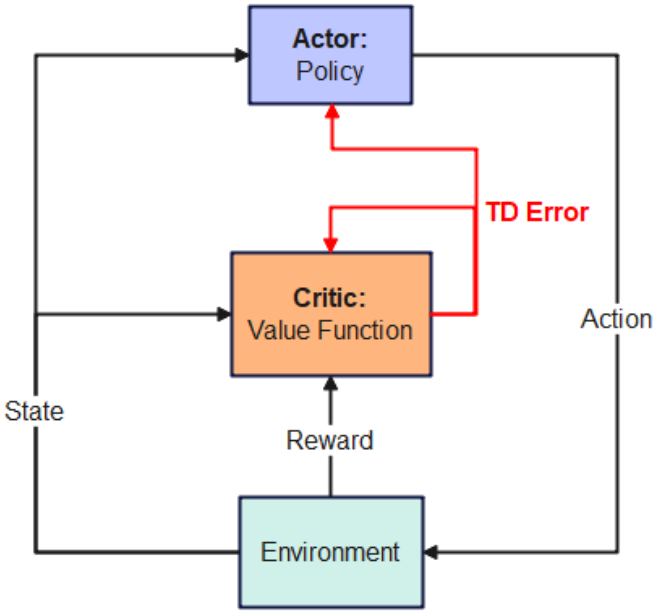


Figure 3.1: Actor-Critic Architecture - The architecture of actor-critic methods shows the policy as the actor selecting actions. The environment provides a reward for the action taken and the next state. The value function, or critic, calculates the TD error, which drives both actor and critic learning.

Initially, the actor/policy parameters $\boldsymbol{\theta}$ and critic/value weights $\boldsymbol{w}$ are initialized (often randomly). For each episode, the state is initialized. While we have not reached the end of an episode or some other termination criteria, the agent interacts with its environment by applying actions sampled from the policy distribution and observing the next state and reward. After an observation, the TD error δ is calculated and used to update the network weights $\boldsymbol{w}$ and $\boldsymbol{\theta}$.

Advantages

AC methods are computationally efficient and can explicitly learn stochastic policies [5]. If they are gradient-based, they also tend to have great convergence properties.

Chapter 4

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a family of reinforcement learning algorithms introduced in 2017 by researchers at OpenAI [3]. These methods are effective for a wide range of reinforcement learning tasks and have achieved state-of-the-art performance on several benchmarks. Their relative simplicity, robustness, and effectiveness make them a popular choice in many RL applications.

This chapter will cover some of the theories PPO builds upon, present the PPO algorithm and discuss its usefulness and shortcomings.

4.1 Motivation and Preliminaries

Various reinforcement learning approaches using neural network function approximators have been proposed in recent years [3]. These include deep Q-learning, standard policy gradient, and trust region methods. However, deep Q-learning struggles with many basic continuous control tasks, while the standard policy gradient methods have data efficiency and robustness issues. Trust region policy optimization (TRPO) is complex and incompatible with architectures that use noise or parameter sharing.

4.1.1 Policy Gradient Methods and Advantage Function

Policy gradient methods aim to minimize the policy loss given by [3]:

$$L^{\mathrm{PG}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\log \pi_{\boldsymbol{\theta}}(a_t|s_t) \hat{A}_t \right] \tag{4.1}$$

where $\hat{\mathbb{E}}_t$ denotes the empirical expectation over the timesteps. The stochastic policy $\pi_{\boldsymbol{\theta}}$ is implemented as a neural network that takes the environment state as input and outputs

probabilities for each possible action as output. The advantage function, $\hat{A}_t$ (4.2), estimates the relative value of the action selected at time t [12]. In other words, it tells us how well the action did compared to an expectation or baseline estimate, i.e., the value function.

$$\hat{A}(s,a) = \underbrace{Q_\pi(s,a)}_{\text{discounted return}} - \underbrace{V_{\boldsymbol{\theta}}(s)}_{\text{estimate}} \tag{4.2}$$

where $a_t \sim \pi_{\boldsymbol{\theta}}(a_t, s_t), s_{t+1} \sim P(s_{t+1}|s_t, a_t)$, meaning actions are sampled from the current stochastic policy and states from the stochastic state transition distribution.

In policy gradient methods, an estimator for the policy gradient is calculated [3]. A common gradient estimator is obtained by differentiating the objective in equation (4.1):

$$\hat{g} = \hat{\mathbb{E}}_t \left[\Delta_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{4.3}$$

Performing gradient descent steps in batches of sampled experience will often lead to destructively large policy updates, leading to instability and poor performance [9, 12]. Larger batches provide a more accurate estimate of the gradient [5]. However, they could lead to biased updates and poor generalization. Contrarily, small batch sizes can offer a regularizing effect due to the variance they add to the gradient. Training could, in this case, require a small learning rate to maintain stability, and the runtime will be longer as it takes more steps to observe the training data.

4.1.2 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) is a policy optimization method developed by Schulman et al. in 2015 [12]. It updates policies by taking the largest step possible to improve performance while constraining how far the new policy can be from the old [9]. Thus, it overcomes the large update challenge of policy gradient methods, helping ensure efficient and stable policy updates.

TRPO aims to maximize a "surrogate" objective (4.4). The objective is equal to the conservative policy iteration objective L^{CPI} (4.5) with an additional constraint on the KL-divergence between the new and old stochastic policy distributions [3]. The constraint defines a trust region of radius δ in which we allow the updated policy to deviate from the old policy.

$$\max_\theta \hat{\mathbb{E}} \left[L^{\text{CPI}} \right] \text{ s.t. } \hat{\mathbb{E}}_t \left[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \leq \delta \tag{4.4}$$

$$L^{\text{CPI}} = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}} \left[r_t(\theta) \hat{A}_t \right] \tag{4.5}$$

In the above expressions, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ it the probability ratio between the new and old policy, and $\text{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta)$ denotes the KL divergence between the new and old policy [3]. The other terms are as defined in the previous section, i.e., section 4.1.1.

TRPO can improve its stochastic policy directly by tweaking the policy parameters $\boldsymbol{\theta}$. The theoretical policy parameter update is defined as follows [9]:

$$\theta_{k+1} = \arg\max_\theta L(\theta) \text{ s.t. } \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \tag{4.6}$$

The implementation of TRPO uses Taylor approximations to simplify the complex theoretical expressions of the objective and update rule [9, 12].

Trust Region Policy Optimization is data efficient, provides stable learning, and performs well on several benchmarks. However, it is based on advanced theory making it complicated to understand and hard to implement. This motivated the search for a simpler method, keeping the performance, stability, and sample efficiency of TRPO, which resulted in Proximal Policy Optimization.

4.2 Proximal Policy Optimization

The main goal of PPO, similar to TRPO, is to determine how to take the largest possible improvement step in a policy based on available data without causing a decline in performance [9]. PPO shares the advantages of TRPO without relying on complicated second-order approaches. In addition, PPO supports parameter sharing, i.e., a joint architecture for the policy and value function, is easier to implement and has been observed to perform similarly or better than TRPO in practice [3].

This section covers the two main versions of PPO, PPO-Penalty, and PPO-Clip. The latter is the most popular and overall best-performing of the two and is therefore given more focus than the first.

4.2.1 PPO-Penalty

PPO-Penalty is an algorithm that addresses the KL-constrained update problem similar to TRPO [3, 9]. However, instead of enforcing a hard constraint on the KL-divergence, it incorporates a penalty term directly into the objective function. The additional term penalizes

policy updates that deviate too far from the previous policy, helping control the magnitude of policy updates. By imposing constraints on the policy updates, PPO-Penalty offers a more explicit mechanism for maintaining stability and preventing large policy deviations compared to TRPO.

KL-penalized objective

The objective in PPO-Penalty is referred to as the KL-penalized objective and given as follows:

$$L^{\text{KLPEN}}(\theta) = \hat{\mathbb{E}}_t \left[r_t(\theta)\hat{A}_t - \beta \text{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta) \right] \tag{4.7}$$

where β is a hyperparameter called penalty coefficient, and the remaining terms are defined as in the previous sections, i.e., section 4.1.1 and 4.1.2.

The penalty term is the second term inside the expectation of the PPO-Penalty objective (4.7). Integrating the penalty term into the objective function eliminates the need for manual tuning of the constraint thresholds, which can be difficult in practice [9]. The algorithm dynamically adjusts the penalty coefficient for the next policy according to the following rule [3]:

$$\begin{aligned} &\beta \leftarrow \beta/2 \text{ if } d < d_{\text{target}}/1.5 \\ &\beta \leftarrow 2\beta \text{ if } d > 1.5 d_{\text{target}} \end{aligned} \tag{4.8}$$

where we compute $d = \hat{\mathbb{E}}_t[\text{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta)]$ and d_{target} is some target value of the KL divergence. This means we halve the value of the penalty coefficient if the estimated KL divergence is below a threshold and double it if it is above some other threshold. The thresholds are set in relation to the target KL divergence. The above values are chosen based on experimentation, but the algorithm is not very sensitive to them, nor is it to the initial value of β [3]. The adaptive nature of the algorithm ensures greater flexibility and robustness during the policy training process.

Policy update

The policy parameter in PPO-Penalty is updated by solving an unconstrained optimization problem:

$$\theta_{k+1} = \arg\max_\theta L^{\text{KLPEN}}(\theta) - \beta_k \text{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta) \tag{4.9}$$

The penalty coefficient changes between iterations to approximately enforce the TRPO constraint on the KL divergence.

Algorithm

The PPO-Penalty algorithm is presented in 1 [17].

Algorithm 1 Proximal Policy Optimization with Adaptive KL Penalty (PPO-Penalty)

Initialize policy π_θ with parameters θ_0
Initialize value function V_ϕ with parameters ϕ_0
Set target KL divergence d_{target}
Initialize KL penalty coefficient β_0
for $k = 0, 1, 2, \ldots$ **do**
    Collect trajectories $\mathcal{D}_k$ running the current policy π_θ in the environment
    Compute rewards $\hat{R}_t$
    Compute advantage estimates $\hat{A}_t$ based on the current value function V_{ϕ_k}
    Update policy parameters by maximizing the surrogate objective with adaptive KL penalty:

$$\theta \leftarrow \arg\max_\theta L^{\text{KLPEN}}(\theta) - \beta_k \text{KL}(\pi_{\theta_{\text{old}}}, \pi_\theta)$$

    by taking K steps of minibatch SGD with Adam optimization
    Calculate d using the updated policy
    if $d \geq 1.5 d_{\text{target}}$ **then**
        $\beta \leftarrow 2\beta_k$
    end if
    if $d \leq d_{\text{target}}/1.5$ **then**
        $\beta \leftarrow \beta_k/2$
    end if
end for

4.2.2 PPO-Clip

PPO-Clip aims to balance exploration and exploitation while ensuring stable policy updates. Unlike traditional methods that use a KL-divergence term, PPO-Clip utilizes a clipping technique in its objective function [3]. Clipping constrains the policy update to staying within a certain range, preventing large policy updates that can cause instability. It also improves sample efficiency by mainly exploring promising regions of the policy space. Clipped sur-

rogate objectives also act as a form of regularization, promoting smoother updates and better generalization to unseen states. Finally, they strike a balance between exploration and exploitation, allowing effective exploration without deviating too far from the successful actions of the learned policy.

Clipped surrogate objective

The objective function of PPO-Clip is called the clipped surrogate objective L^{CLIP} (4.10) [3, 18]. It yields a lower, or pessimistic, bound for the unconstrained TRPO objective L^{CPI} (4.5) and includes a penalty for having too large policy updates [3].

$$L^{\text{CLIP}}(\boldsymbol{\theta}) = \hat{E}_t\left[\min(r_t(\boldsymbol{\theta}))\hat{A}_t, \text{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{4.10}$$

In the above expression, $\hat{E}_t$ is the empirical expectation over the timesteps. The probability ratio $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)}$ denotes the rate in which the new policy differs from the old [3]. $\hat{A}_t$ we recognize as the estimated advantage at time t and ϵ is a clipping hyperparameter (usually 0.1 or 0.2).

Clipping

The clipping function, as illustrated in figure 4.1, cuts the probability ratio r between $1-\epsilon$ and $1+\epsilon$ depending on the sign and magnitude of the advantage function. If the advantage is positive, the objective will improve if the likelihood $\pi_{\boldsymbol{\theta}}(a|s)$ increases. The minimum term in the objective limits the amount we can increase the policy update to $(1+\epsilon)$ times the old policy. On the other hand, if the advantage is negative, the objective will improve if the likelihood $\pi_{\boldsymbol{\theta}}(a|s)$ decreases. As before, the minimum term in the objective limits the amount we can decrease the policy update to $(1-\epsilon)$ times the old policy.

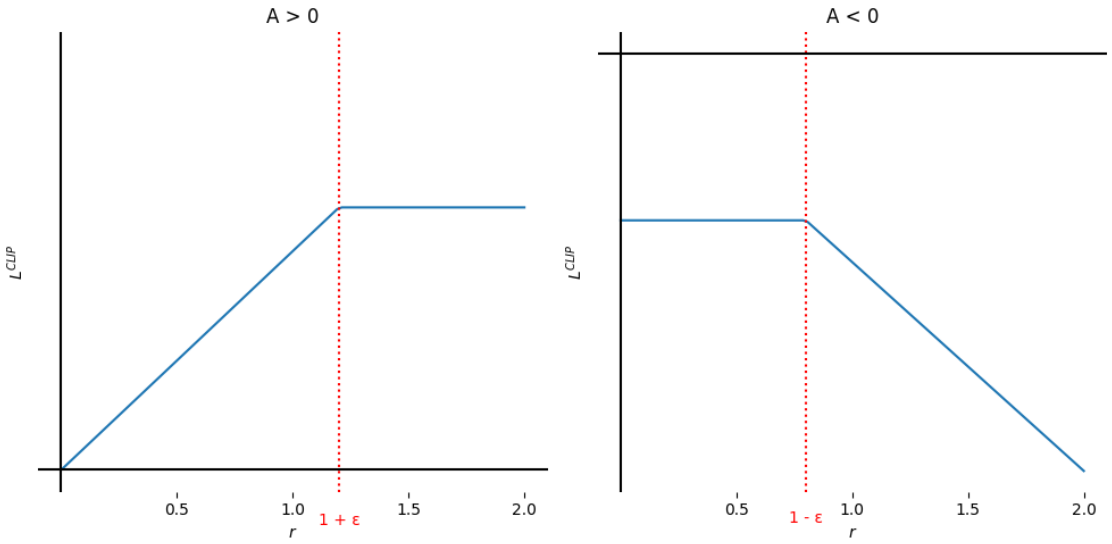


Figure 4.1: Clipping function in PPO-Clip - The plots show a single time step of the objective function L^{CLIP} as a function of the probability ratio r. (1) The left figure shows the case of a positive advantage, where we are currently at $r = 1$, and the new objective term cannot increase more than to the value corresponding to where $r = 1 + \epsilon$. (2) The right plot shows the case of a negative advantage, where we are currently at $r = 1$, and the new objective term cannot decrease more than to the value corresponding to where $r = 1 - \epsilon$. Recreation from original paper [3].

Actor update

The policy update in PPO-Clip is as follows [9]:

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} E_{s,a\sim\pi_{\boldsymbol{\theta}_k}}[L^{\text{CLIP}}(\boldsymbol{\theta})] \tag{4.11}$$

Maximizing the objective is typically done through multiple steps of minibatch SGD [9].

Critic update

The critic wishes to minimize the difference between the approximated value $\hat{V}_\pi(s, \boldsymbol{w})$ and the target value $V_\pi(s)$. The error can be calculated as the mean square value error [5]:

$$\overline{\text{VE}}(\boldsymbol{w}) = \mathbb{E}\left[(\hat{V}_\pi(s, \boldsymbol{w}) - V_\pi(s))^2\right] \tag{4.12}$$

Or equivalently, for the action-value function:

$$\overline{\text{VE}}(\boldsymbol{w}) = \mathbb{E}\left[(\hat{Q}_\pi(s, a, \boldsymbol{w}) - Q_\pi(s, a))^2\right] \tag{4.13}$$

This error can be minimized through stochastic gradient descent w.r.t. the parameters $\boldsymbol{w}$:

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\delta_t \nabla_{\boldsymbol{w}} Q_\pi(s, a, \boldsymbol{w}) \tag{4.14}$$

where δ_t is the TD error.

Algorithm

Finally, we present the PPO-Clip algorithm as documented by SpinningUP at OpenAI [9].

Algorithm 2 Proximal Policy Optimization with Clipped Surrogate Objective (PPO-Clip)

Initialize policy $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}_0$
Initialize value function $V_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}_0$
Set clip parameter ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect trajectories $\mathcal{D}_K = \{\tau_i\}$ by running policy $\pi(\boldsymbol{\theta}_k)$ in the environment
  Compute rewards $\hat{R}_t$
  Compute advantage estimates $\hat{A}_t$ based on the current value function $V_{\boldsymbol{\phi}_k}$
  for several epochs **do**
    Update policy parameters by maximizing the clipped surrogate objective:

$$\boldsymbol{\theta} \leftarrow \arg\max_{\boldsymbol{\theta}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\mathbb{E}_t\left[\min\left(r_t(\boldsymbol{\theta})\hat{A}_t, \text{clip}\left(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right]\right)$$

    typically using SGD with Adam
    Update value function parameters by minimizing the following loss function:

$$\boldsymbol{\phi} \leftarrow \arg\min_{\boldsymbol{\phi}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(V_{\boldsymbol{\phi}}(s_t) - \hat{R}_t\right)^2$$

    typically with some gradient descent algorithm
  end for
end for

Parameter sharing

PPO allows for neural network architectures that share parameters between the policy and value function [9]. If this is the case, a value function error term should be included in the

objective [3]. The value function error term, $L_t^{\text{VF}}(\boldsymbol{\theta}) = (V_{\boldsymbol{\theta}}(s_t) - V_t^{\text{target}})^2$, is the squared-error loss between the current value function and the target value function where the target value function can be computed as $V_t^{\text{target}} = R_t + \gamma V_{\boldsymbol{\theta}}(s_{t+1})$ Furthermore, an entropy bonus S can be added to ensure sufficient exploration. Combining the previous objective with the additional terms gives the new objective:

$$L_t^{\text{CLIP + VF + S}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\boldsymbol{\theta}) - c_1 L_t^{\text{VF}}(\boldsymbol{\theta}) + c_2 S[\pi_{\boldsymbol{\theta}}](s_t) \right] \tag{4.15}$$

with coefficients c_1, c_2.

4.2.3 PPO, actor-critic

An actor-critic version of PPO is given in algorithm 3 [3].

Algorithm 3 Proximal Policy Optimization, Actor-Critic

for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt $\boldsymbol{\theta}$, with K epochs and minibatch size $M \leq NT$
  $\boldsymbol{\theta}_{\text{old}} \leftarrow \boldsymbol{\theta}$
end for

The actor-critic structure of PPO is illustrated in detail in figure 4.2.

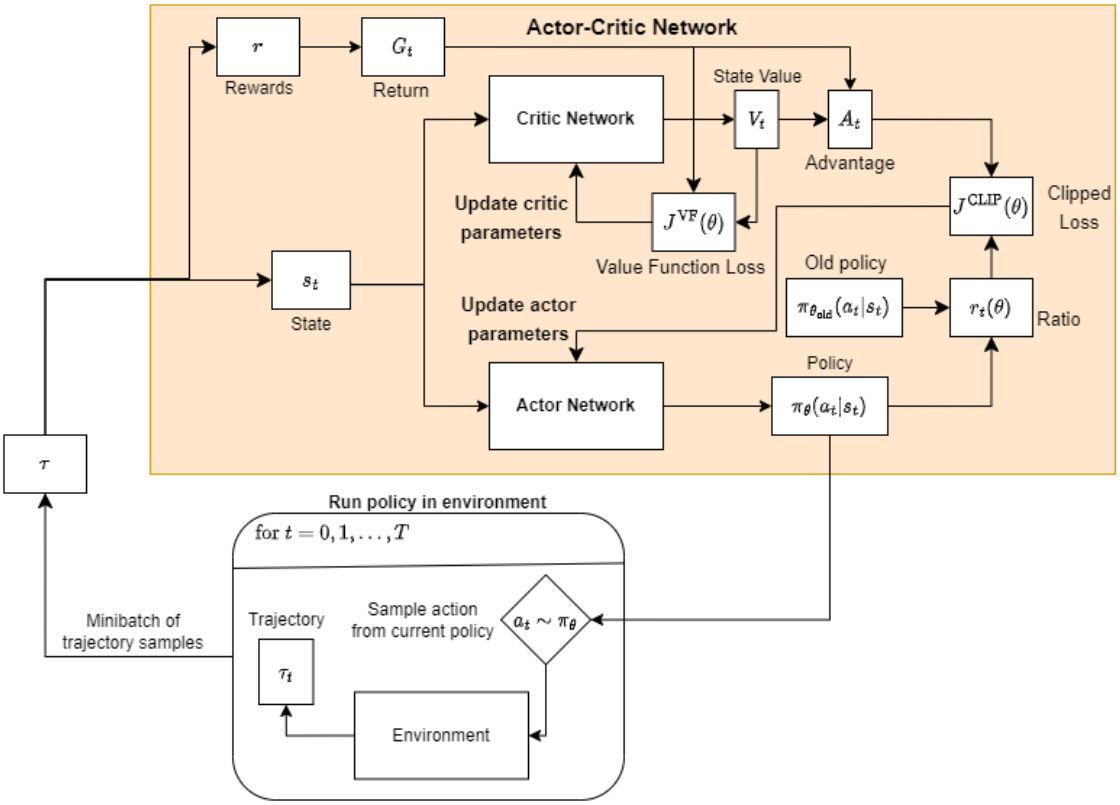


Figure 4.2: Actor-Critic Structure of PPO - As PPO is on-policy, we run the current policy in the environment for T timesteps to collect sample trajectories τ. This allows us to calculate the return G from the observed rewards, and with the current value estimate V_t, we can calculate the advantage estimates A_t and the value function loss. The actor network gives the updated policy, which can be compared to the old policy to calculate the ratio $r_t(\theta)$. The clipped surrogate objective is calculated from the ratio and the advantage estimates. For each epoch, PPO updates the policy parameters by minimizing the clipped surrogate loss and updates the value function parameters by minimizing the value function loss.

4.2.4 Exploration Vs. Exploitation in PPO

PPO balances between exploration and exploitation like other on-policy methods, i.e., by sampling actions based on its most recent stochastic policy [9]. The exploitation-exploration trade-off is controlled by an entropy coefficient, which is a hyperparameter that decides the level of uncertainty or randomness in the actions selected by the policy by weighting the entropy of the policy distribution in the objective function. A higher entropy coefficient will encourage exploration, and a lower entropy coefficient will favor exploitation. Finding a suitable entropy coefficient is often done by experimenting and tuning and depends on the task and its complexity. If the coefficient is set too high, the policy may become overly

exploratory and less focused on exploiting the currently learned policy. Conversely, if the coefficient is set too low, the policy may become too deterministic and may not explore new actions effectively. During training, the policy tends to become less random to exploit its accumulated knowledge. However, this can sometimes trap the policy in local optima.

4.2.5 Generalized Advantage Estimator

The generalized advantage estimator (GAE) GAE(γ, λ) estimates the advantage function in PPO and is defined as the exponentially-weighted average of these k-step estimators [19]:

$$\hat{A}^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{4.16}$$

where V is an approximate value function. $\delta_t^V = r_t + \gamma(V(s_{t+1} - V(s_t))$ is the TD residual of V with discount factor γ and smoothing parameter λ.

Chapter 5

Related Work

One of the first deep RL methods was deep Q-learning (DQL) [15], using deep Q-networks (DQNs) to handle problems with high-dimensional state spaces. In 2015, DQNs were extended to continuous control tasks with Deep Deterministic Policy Gradients (DDPG) [20] able to cope with large or continuous action spaces. These developments eventually led to DRL-based control methods substituting traditional control frameworks in robotic tasks, including quadrotor navigation.

DRL control has made it possible to learn complex control directly from perception data and simple sensor measurements. DroNet, a prediction network for quadrotor navigation presented by Loquercio et al. [8], successfully used RGB images combined with angle measurements to predict steering angles and collision probabilities. Choi et al. [6] developed a method for navigating complex real-world environments only using a limited FOV depth camera. In 2021, Hoeller et al. [21] presented a learning-based method for map-less navigation with a quadrupedal robot in cluttered environments. Their trained control policy used a depth representation, learned by a variational autoencoder network, and trajectory information to output control commands to guide the robot safely to its target location. Their result showed very rare collisions with obstacles within the camera's field of view. Song et al. [22] used Proximal Policy Optimization (PPO) for the challenging task of autonomous drone racing through gates. They achieved high success and fast flying despite the high dimensional state and action space and the complexity of the required maneuvers.

Finally, the DRL method studied in this thesis will study the performance of PPO in a quadrotor navigation task through cluttered 3D environments. It is building on the works of Nitschke [23] and current works at the ARL lab at NTNU. Where Nitschke used curriculum learning for gradually improving the learned policy [24], this thesis will focus on learning starting in environments of different static complexities.

Chapter 6

Method

This chapter provides some background on quadrotor control, the experimental design, implementation, and setup developed to provide insight into the research questions presented in section 1.2.

6.1 Quadrotor Control

This section provides a brief look at traditional control methods and their limitations. Then we see how deep reinforcement learning can be used as an alternative approach, along with its advantages and disadvantages compared to the traditional framework. Finally, the quadrotor dynamics used in this thesis are presented.

6.1.1 Traditional Control

Navigation is essential for all mobile robots, including unmanned aerial vehicles [1]. The navigation problem is to find a path between a starting and target position in a two-dimensional (2D) or three-dimensional (3D) environment while avoiding obstacles. The traditional navigation framework combines several modules that handle different sub-tasks - such as perceiving and mapping the environment, localizing the robot in the environment, planning the path, and controlling the motion of the robot. Simultaneous Localization and Mapping (SLAM) is one such method that "constructs a map of its environment and uses a localization algorithm to determine the current position of the robot" [1] and uses a path planning and controller module to guide the robot to its target.

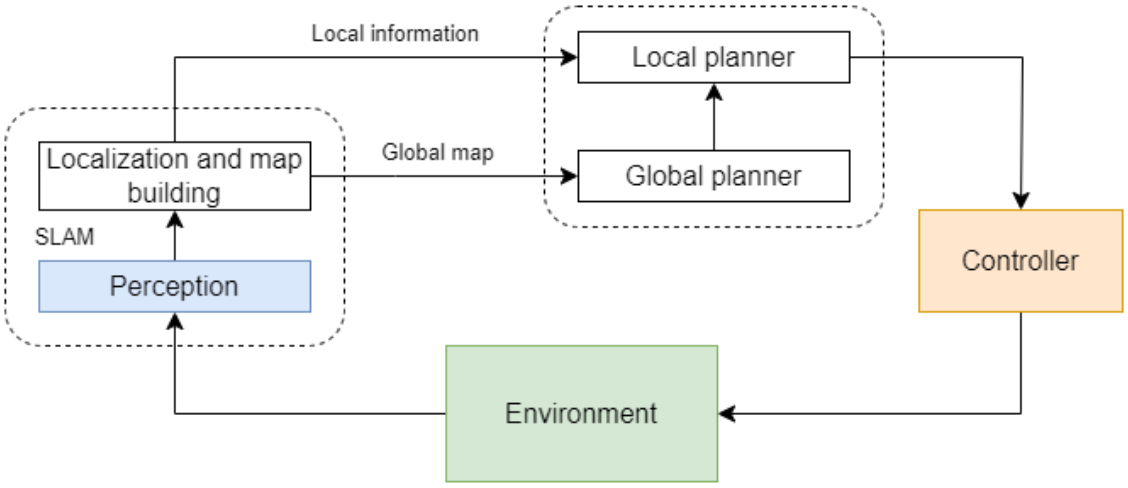


Figure 6.1: Traditional Navigation Framework - recreated from [1].

Traditional control methods are based on well-developed mathematical models and control theory principles, providing a solid foundation for analysis and design [1]. They often come with theoretical stability and performance guarantees, and offer a clear understanding of how the control decisions are made based on the underlying mathematical models.

Despite its interpretability, well-established theory, and stability and performance guarantees, traditional navigation has several limitations. Each part of the traditional navigation framework offers a careful design challenge, and "their integration often leads to large computational errors" that "gradually accumulate along the pipeline from mapping, to positioning, to the path planning algorithm" [1]. In practice, this often leads to poor performance. Additionally, traditional methods rely on access to a "high-precision global map that is very sensitive to sensor noise, resulting in limitations in the ability to manage an unknown or dynamic environment" [1]. Finally, they often require manually tuning the control parameters, which can be time-consuming and challenging.

6.1.2 Deep Reinforcement Learning Control

With the powerful representation capabilities of deep neural networks, Deep Reinforcement Learning (DRL) is a promising alternative to traditional control methods [5, 16]. DRL allows for learning navigation strategies directly from raw sensor inputs [1, 2], eliminating the need for manually designing the controller. DRL has been successfully applied in robotics, facilitating real-time control without delay and accumulated error issues present in the traditional control framework.

When using Deep RL in control applications, the agent represents the controller; the environment is the controlled system (plant), and the actions yield the control signal [5]. The DRL task is formulated as a Markov Decision Process, where the objective is to discover "the optimal policy for guiding a robot to its target position through interaction with the environment" so as to maximize the expected rewards [1].

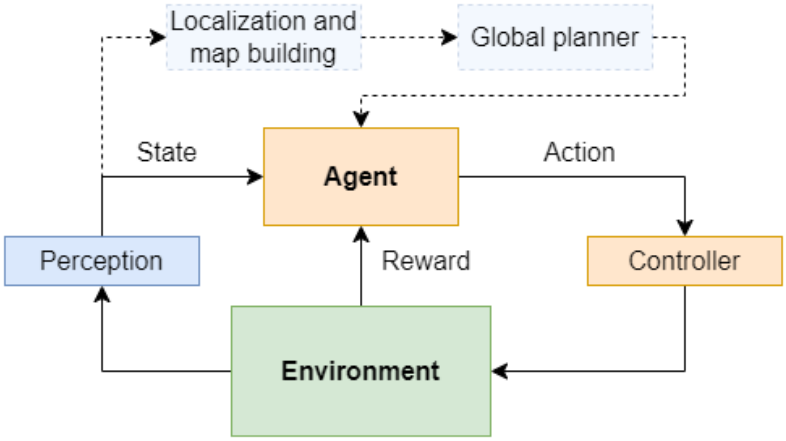


Figure 6.2: Deep Reinforcement Learning-Based Navigation Framework - recreated from [1].

Succeeding in the DRL task involves adequately defining the MDP, including the state space, action space, and reward function.

State space

The state space should capture relevant information about the quadrotor's environment and internal state [5]. Generally, anything the agent cannot change arbitrarily is considered part of its environment, i.e., a part of the state vector. This can include its position, velocity, orientation, sensor measurements (e.g., camera images, LiDAR data), and any other contextual information necessary for navigation and collision avoidance.

Action space

The action space must represent the quadrotor's available maneuvers or control inputs. These actions can be low-level controls, such as the forces applied to the quadrotor propellers and adjustments in velocity and orientation [5]. Typically, achieving the desired motion in a mobile robot involves Proportional-Integral-Derivative (PID) controllers or other low-level motion controllers to generate motion control instructions. These controllers are

responsible for translating discrete moving actions and continuous velocity commands into appropriate control signals that effectively control the robot's motion [1].

Transition dynamics

The transition dynamics should describe how the environment evolves based on the drone's actions [5]. This can include the physics of the drone's motion, the effects of control inputs, and the dynamics of obstacles in the environment.

Reward function

The reward function plays a crucial role in training the RL agent to complete a task successfully. However, in navigation tasks, the reward system often consists of sparse rewards, where positive or negative rewards are only provided upon reaching the target or colliding with obstacles [1]. This sparsity can slow or hinder the agent's learning and convergence.

To improve training efficiency, one can use reward-shaping methods that provide denser and more intermediate feedback to the agent [1]. We can provide positive rewards for making progress toward the goal, add rewards for moving in the right direction, or penalize each time step for encouraging the robot to move faster toward the target. A term that penalizes proximity to obstacles could be added to promote safer navigation. This can be done by defining a function that calculates the distance between the agent and the obstacles and assigns a penalty based on the proximity. The penalty should increase as the agent gets closer to the obstacles. To encourage the agent to follow smooth trajectories, a term that rewards the agent for maintaining a consistent velocity or penalizes abrupt changes in position or velocity can be added. This can be achieved by calculating the difference between the current and previous positions or velocities and assigning a reward or penalty based on the magnitude of the difference.

Constraints

Constraining the RL-controlled drone's movement to be within the field of view of its camera sensor help ensure safety, efficiency, and precise control. An algorithm that uses sensor data in its state information from the environment will learn to associate specific sensor readings with specific actions. Without constraints on the directions of movement, the drone can collide without having sensed the collision object, leading to inefficient and poor learning and a higher risk of failure when applied to a real-world scenario.

Advantages and disadvantages of DRL-based control

DRL-based navigation offers advantages such as mapless navigation, strong learning capability, and reduced reliance on sensor accuracy [1]. DRL-based navigation can replace or integrate with traditional navigation frameworks, assuming roles such as localization, map building, and local path planning.

Despite its many promising capabilities, there are some drawbacks to consider when using deep reinforcement learning [1]. DRL control often requires a large amount of data and exploration to learn effective policies, resulting in sample inefficiency. The complexity of DRL models can make them challenging to interpret, limiting the understanding of the reasoning behind their decisions. In complex environments, DRL agents may encounter local optima, necessitating additional measures to ensure robust navigation. Their learned policies may exhibit unexpected or undesirable behavior, which could be critical when safety and stable flight is important.

Simulation vs. real environment

The purpose of developing DRL-based control is to apply it in the real world. However, "training in a physical environment is very time-consuming and dangerous" [1], so typically, agents are trained in a simulation environment before transferring this learning to the physical environment. Simulators play a key role in training robots improving both the safety and iteration speed in the learning process [25]. They provide a virtual environment where robots can be trained and tested without the risks associated with real-world interactions.

Training and testing

Training and testing reinforcement learning (RL) policy optimization algorithms in an obstacle-free environment allow the agent to learn basic skills and movements without the added complexity and difficulty of avoiding obstacles. It provides easier debugging and tuning of the RL algorithm and can build a good foundation before introducing obstacles. Thereafter, the setup can be tested in the obstacle environment after suitable adaptions.

Another option is curriculum learning [24], where the obstacle density gradually increases during training. This could yield more efficient training and a higher success rate at a higher obstacle-density level.

6.1.3 Quadrotor Dynamics

The quadrotor dynamics and attitude controller used in this project are based on the full six degrees of freedom quadrotor UAV dynamic model presented by Goodarzi et al. in 2011 [26]. They present a nonlinear controller that solves an output attitude tracking problem with exponential stability of the tracking error equilibrium.

Figure 6.3 shows the quadrotor model. The position of the quadrotor $\boldsymbol{x} \in \mathbb{R}^3$ is given as the relative 3D coordinate of the center of mass with respect to the inertial reference frame $\{\vec{e_1}, \vec{e_2}, \vec{e_3}\}$. The body-fixed frame $\{\vec{b_1}, \vec{b_2}, \vec{b_3}\}$ has its origin at the quadrotor center of mass, with $\vec{b_1}, \vec{b_2}$ defined on the plane of the four rotors. Each of the four propellers generates a thrust force $f_{1,2,3,4}$ in the $-\vec{b_3}$ direction.

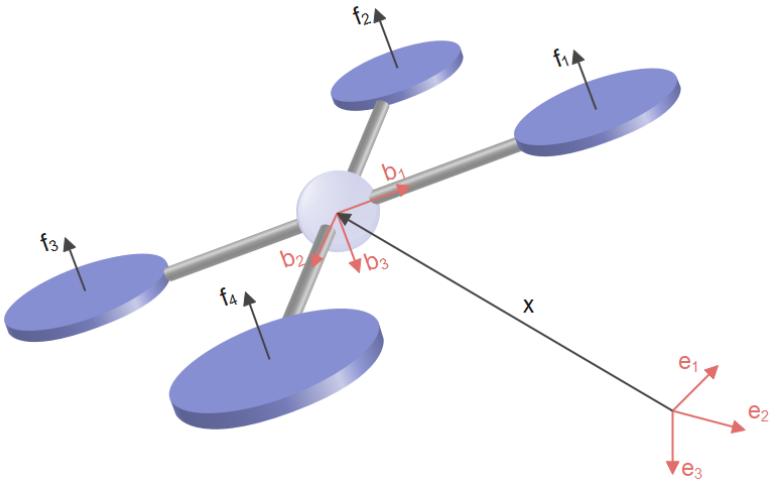


Figure 6.3: Quadrotor model - The quadrotor is represented by its relative body-frame position and orientation in the inertia frame.

The equations of motion of the quadrotor are given by:

$$\dot{\boldsymbol{x}} = \boldsymbol{v} \tag{6.1}$$

$$m\dot{\boldsymbol{v}} = mge_3 - f\boldsymbol{R}e_3 \tag{6.2}$$

$$\dot{\boldsymbol{R}} = \boldsymbol{R}\hat{\Omega} \tag{6.3}$$

$$\boldsymbol{J}\dot{\boldsymbol{\Omega}} + \boldsymbol{\Omega} \times \boldsymbol{J}\boldsymbol{\Omega} = M \tag{6.4}$$

where $\boldsymbol{v} \in \mathbb{R}^3$ is the linear velocity, $m \in \mathbb{R}$ the quadrotor mass, $g \in \mathbb{R}$ the gravitational acceleration, $f = \sum_{i=0}^{4} f_i \in \mathbb{R}$ the total thrust force (assumed identical to the torque), $\boldsymbol{R} \in \mathrm{SO}(3)$ the rotation matrix from body-fixed to the inertial frame, $\boldsymbol{\Omega} \in \mathbb{R}^3$ is the angular

velocity in the body-fixed frame, $\boldsymbol{J} \in \mathbb{R}^{3\times3}$ is the inertia matrix w.r.t. the body frame, and $M \in \mathbb{R}^3$ is the total moment vector in the body-fixed frame.

The sum of forces in the e_3 direction is given by the gravitational force mg and by the thrust vector $f\boldsymbol{R}$ in the $-e_3$ direction, giving the second dynamics equation (6.2). The two last motion equations are the attitude dynamics (6.3), 6.4 from which the attitude controller next will be derived.

The hat map in the third equation, $\widehat{\cdot}: \mathbb{R}^3 \to \mathfrak{so}(3)$, is defined as follows: for any vector $\mathbf{v} = (v_1, v_2, v_3) \in \mathbb{R}^3$, we have

$$\widehat{\mathbf{v}} = \begin{pmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{pmatrix},$$

where $\mathfrak{so}(3)$ denotes the space of 3×3 skew-symmetric matrices.

The quadrotor has three translational and three rotational degrees of freedom. As it is impossible to achieve asymptotic output tracking of attitude and position simultaneously [26], one must choose between controlling attitude or position. Here we use an attitude controller, i.e., aiming to control the rotation matrix or, equivalently, the quaternion vector of the quadrotor.

Attitude controller

We assume a smooth attitude command $\boldsymbol{R}_d(t) \in \mathfrak{so}(3)$ satisfying the kinematics:

$$\dot{\boldsymbol{R}}_d = \boldsymbol{R}_d \widehat{\boldsymbol{\Omega}}_d \tag{6.5}$$

where $\boldsymbol{R}_d, \boldsymbol{\Omega}_d$ denotes the desired attitude and angular velocity, respectively. For a given tracking command $(\boldsymbol{R}_d, \boldsymbol{\Omega}_d)$, and the current attitude and angular velocity $(\boldsymbol{R}, \boldsymbol{\Omega})$, the attitude tracking error is defined by:

$$e_R = \frac{1}{2}(\boldsymbol{R}_d^T \boldsymbol{R} - \boldsymbol{R}^T \boldsymbol{R}_d)^\vee \tag{6.6}$$

where the vee map $\vee : \mathfrak{so}(3) \to \mathbb{R}^3$ is defined as the inverse of the hat map.

The angular velocity command is given by:

$$\hat{\boldsymbol{\Omega}}_d = \boldsymbol{R}_d^T \dot{\boldsymbol{R}}_d \tag{6.7}$$

The angular velocity tracking error is given by:

$$e_\Omega = \boldsymbol{\Omega} - \boldsymbol{R}^T \boldsymbol{R}_d \boldsymbol{\Omega}_d \tag{6.8}$$

The moment vector (6.9) describes the nonlinear attitude controller:

$$\boldsymbol{M} = -k_R \boldsymbol{e}_R - k_\Omega e_\Omega + \boldsymbol{\Omega} \times \boldsymbol{J}\boldsymbol{\Omega} - \boldsymbol{J}(\hat{\boldsymbol{\Omega}}\boldsymbol{R}^T \boldsymbol{R}_d \boldsymbol{\Omega}_d - \boldsymbol{R}^T \boldsymbol{R}_d \dot{\boldsymbol{\Omega}}_d) \tag{6.9}$$

where k_R, k_Ω are positive constants.

The body-frame moment vector $\boldsymbol{M} \in \mathbb{R}^3$ and the thrust magnitude f can be used to calculate the individual propeller thrusts f_1, f_2, f_3, f_4 from the relation in equation (6.10).

$$\begin{bmatrix} f \\ M_1 \\ M_2 \\ M_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -d & 0 & d \\ d & 0 & -d & 0 \\ -c_{\tau f} & c_{\tau f} & -c_{\tau f} & c_{\tau f} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \tag{6.10}$$

6.2 Problem Formulation

This thesis aims to use proximal policy optimization to solve a map-free navigation task in free and cluttered environments. Due to the limitations of traditional control (section 6.1.2) and traditional reinforcement learning (section 3.1), the task is to design a DRL-based controller without discretizing the state nor action space. Furthermore, the quadrotor is assumed to be equipped with a depth camera for perception and sensors for measuring the state of the quadrotor. This creates a two-folded problem in (1) designing a learning-based agent for discovering an optimal control policy that, given the observed measurements, yields control commands that safely guide the quadrotor to its target position and (2) designing a depth compression module for learning a suitable low-dimensional representation of the perception data to lower the input dimension and total system delay.

Specifically, given position, orientation, linear velocities, and angular velocity measurements of the quadrotor, the control policy should output the desired orientation and/or angular velocities to be used in traditional low-level controllers to produce thrust and torque commands for guiding the quadrotor to its target.

The reinforcement learning task was to learn the parameters $\boldsymbol{\theta}$ of a stochastic policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$ from which sampled actions $\boldsymbol{a} \sim \pi_{\boldsymbol{\theta}}$ would result in the quadrotor state $\boldsymbol{s}$ converging to a target state $\boldsymbol{s}^*$ within the end of an episode T; i.e., $\boldsymbol{s} \to \boldsymbol{s}^*$ as $t \to T$. The target state is a goal position, i.e., a 3D point $\boldsymbol{x}^* = \begin{bmatrix} x & y & z \end{bmatrix}^T \in \mathbb{R}^3$. As it is infeasible to reach an

exact position, the goal is defined as a region within a certain radius of the goal. To account for the difference in task complexity, the target reaching radius is $1.0m$ in the cluttered environment and $0.5m$ in the free environment. Figure 6.4 shows an example target position as a sphere indicating the area where the drone is successful if it reaches within.

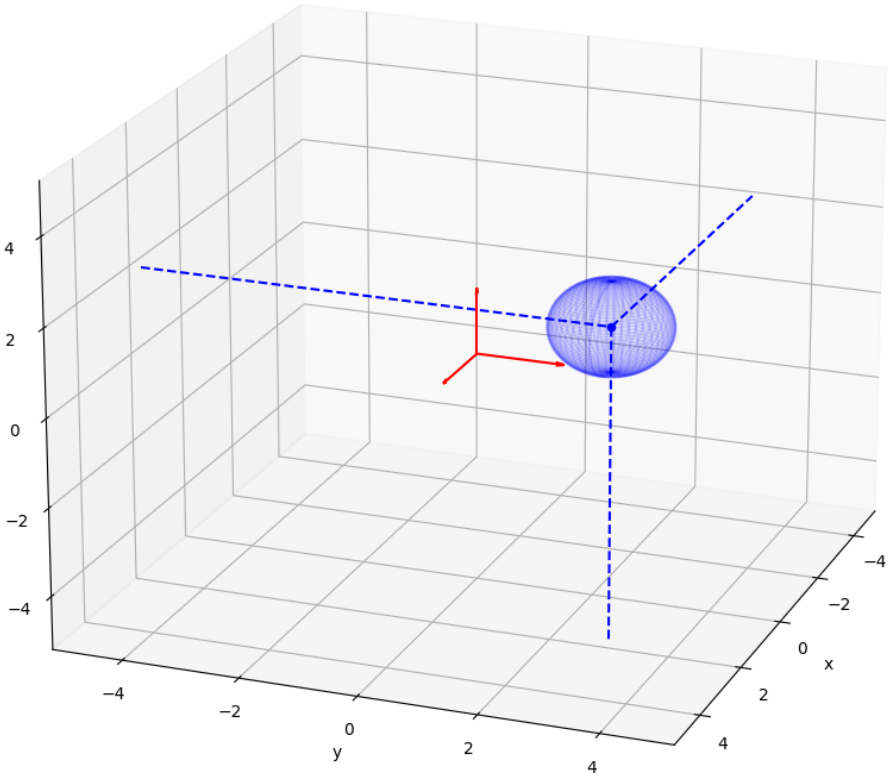


Figure 6.4: Example target - target reaching radius is shown as a sphere surrounding the target position. The origin axes are indicated in red.

The navigation problem is formulated somewhat differently in the free and the obstacle-filled environments. The following subsections cover the relevant definitions and highlight their differences.

6.2.1 Free Environment

In the free environment, the goal was to reach a given target position as efficiently as possible. The state and action vectors in the free environment are given as follows:

$$\boldsymbol{s} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{q} \\ \boldsymbol{v} \\ \boldsymbol{\Omega} \end{bmatrix} \in \mathbb{R}^{13}, \boldsymbol{a} = \begin{bmatrix} \boldsymbol{R}_d \\ f_d \end{bmatrix} \in \mathbb{R}^4 \tag{6.11}$$

For the state vector, we denote $\boldsymbol{x} = \begin{bmatrix} x & y & z \end{bmatrix}^T \in \mathbb{R}^3$ as the relative 3D position as defined in figure 6.3, $\boldsymbol{q} = \begin{bmatrix} \eta & \epsilon_1 & \epsilon_2 & \epsilon_3 \end{bmatrix}^T \in \mathbb{R}^4$ as the quaternion vector describing the orientation of the quadrotor with respect to the world frame, $\boldsymbol{v} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T \in \mathbb{R}^3$ as the linear velocity vector, and $\boldsymbol{\Omega} = \begin{bmatrix} \omega_\rho & \omega_\theta & \omega_\phi \end{bmatrix}^T \in \mathbb{R}^3$ as the angular velocity vector.

The action or control input consists of the desired orientation $\boldsymbol{R}_d = \begin{bmatrix} \rho_d & \theta_d & \phi_d \end{bmatrix}^T \in \mathbb{R}^3$ represented by the desired roll, pitch, and yaw angles $\rho_d, \theta_d, \phi_d \in \mathbb{R}$, and the desired total thrust force f_d.

The target position is defined as the fixed point $\boldsymbol{x}^* = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \in \mathbb{R}^3$, while the initial position is randomly placed within the environment bounds.

Figure 6.5 shows an overview of the control problem. Given low-level controllers (outlined in blue), this thesis aims to design the module for policy learning (orange RL-agent module). The errors computed in the low-level controllers use additional state information to compare the actual orientation to the desired orientation produced by the RL agent.

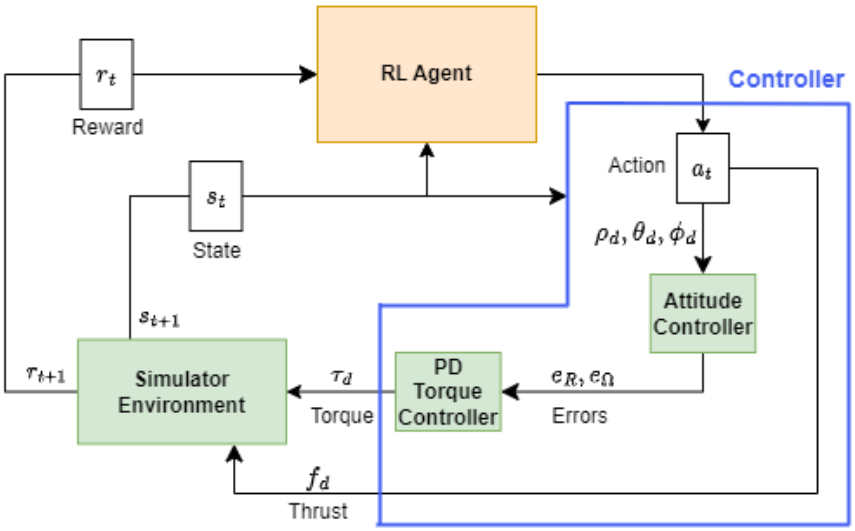


Figure 6.5: Problem overview in the obstacle-free environment - Given the quadrotor states s_t, and rewards r_t from a simulation, the RL module should output the desired control actions a_t. The desired angular velocities are fed through an attitude and torque controller to produce the desired torque command τ_d. Finally, the desired thrust f_d and torque commands are applied to the quadrotor in the simulator resulting in a state transition and a reward.

6.2.2 Obstacle-Filled Environment

The state and action vectors in the obstacle-filled environment are given as follows:

$$\boldsymbol{s} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{q} \\ \boldsymbol{v} \\ \boldsymbol{\Omega} \\ \boldsymbol{z} \end{bmatrix} \in \mathbb{R}^{141}, \boldsymbol{a} = \begin{bmatrix} v_d \\ \theta_d \\ \dot{\phi}_d \end{bmatrix} \in \mathbb{R}^3 \tag{6.12}$$

As for the free environment, $\boldsymbol{x} \in \mathbb{R}^3$ is the relative 3D position as defined in figure 6.3, $\boldsymbol{q} \in \mathbb{R}^4$ is the quaternion vector, $\boldsymbol{v} \in \mathbb{R}^3$ is the linear velocity vector, and $\boldsymbol{\Omega} \in \mathbb{R}^3$ is the angular velocity vector. The critical alteration from the free environment state vector was the addition of the depth image representation, or latent code, $\boldsymbol{z} \in \mathbb{R}^{128}$. This will help the algorithm associate different depth data inputs to different rewards, allowing the agent to learn collision avoidance.

The action in the obstacle-filled environment is represented by a vector consisting of commands controlling velocity and orientation, where v_d is the desired linear velocity, θ_d is the desired pitch angle, and $\dot{\phi}_d$ is the desired yaw rate. The linear velocity v_d is defined in

the direction of the pitch angle from the latest orientation. The pitch angle θ is constrained to be within $-\frac{\pi}{5} \leq \theta \geq \frac{\pi}{5}$, representing the minimum and maximum velocity angles such that the quadrotor only flies in directions within its field of view.

The target and initial position are randomly placed within the environment's bounds.

The obstacle-avoidance task required some type of environment perception; this thesis uses a depth camera. To use depth data in the reinforcement learning module without causing large delays or complexity, the depth image must be compressed to a lower dimension. However, there needs to be a balance between a high compression rate and the ability to capture enough information about the environment to be successful in obstacle avoidance.

Figure 6.6 shows an overview of the navigation problem. Given low-level controllers (outlined in blue), this thesis aims to design the module for policy learning (orange RL-agent module) and for depth representation learning (blue depth compression module). The errors computed in the low-level controllers use additional state information to compare the actual velocities and orientations to their desired values produced by the RL agent.

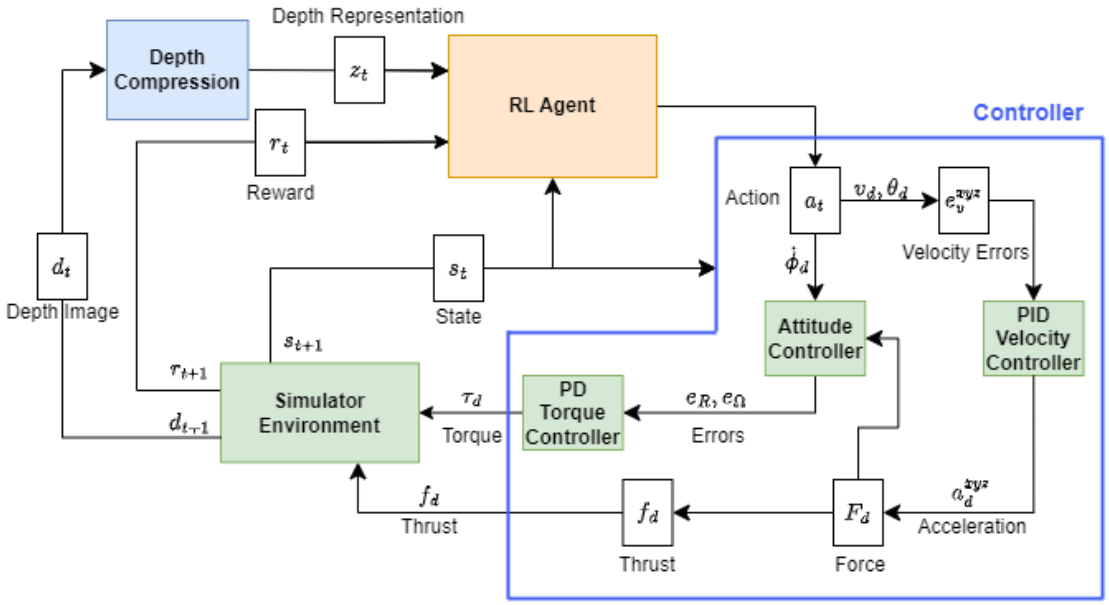


Figure 6.6: Problem overview in the obstacle-filled environment - For an input depth image d_t, the depth compression module should give a suitable depth representation z_t. The RL agent, or control policy, should produce the desired control actions a_t given the rewards r_t, quadrotor observations s_t, and depth representation from the simulator environment. The desired velocity and pitch angle are fed through a PID controller to produce desired forces F_d. These forces can be used to calculate the desired thrust forces f_d^{1234} commands to apply to the quadrotor propellers. In addition, they are used in the attitude controller, along with the desired yaw rate $\dot{\phi}_d$ to give the attitude and angular velocity errors $e_{\mathbb{R}}, e_\Omega$. These errors are fed through the torque controller to yield the desired torques τ_d. Finally, the desired thrust and torque commands are applied to the quadrotor in the simulator resulting in a state transition, a new depth image, and a reward.

6.3 Proposed Solution

Here the proposed approach for solving the navigation tasks specified in the above section is presented. The proposed solution consists of (1) using representation learning to lower the dimension of the depth camera and (2) using reinforcement learning to discover a control policy that leads the quadrotor safely and efficiently to its target. For policy learning, we propose the optimization algorithm PPO with the actor-critic structure presented in figure 4.2. For learning a representation of the depth camera data, we propose using an encoder network trained through a Variational Autoencoder (VAE). For the free environment, perception is not necessary as it is known to have no obstacles, so here we only focus on policy learning. The overall structures for the two solutions are shown in figures 6.7 and 6.8.

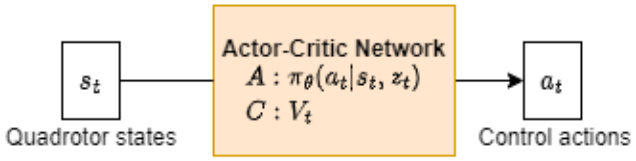


Figure 6.7: Solution overview in the obstacle-free environment - Given the quadrotor observations, or states, s_t the policy network should output the desired control actions a_t.

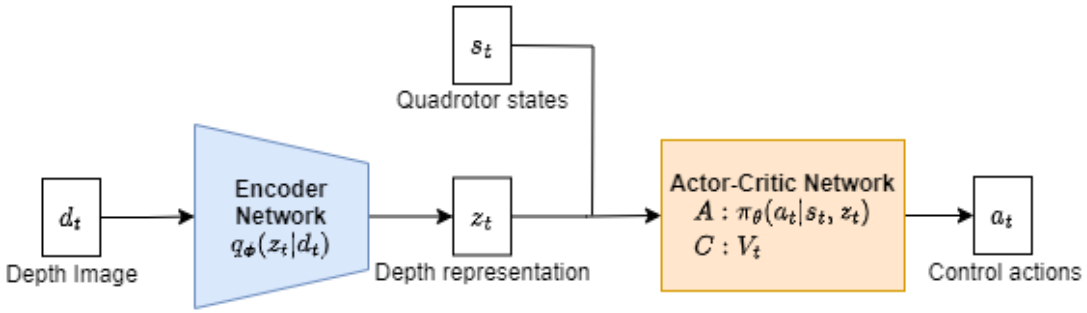


Figure 6.8: Solution overview in the obstacle-filled environment - For an input depth image d_t, the encoder network should give a suitable representation z_t. The policy network should output the desired control actions a_t given the quadrotor observations s_t and depth representation from the encoder. The desired actions are fed through a PID controller to produce thrust commands to apply to the quadrotor.

6.3.1 Policy Learning

Here follows the motivation behind the design choices made for the policy learning task. This includes the choice of optimization algorithm and the reward function and actor-critic network designs of the free and obstacle-filled environments.

Optimization algorithm

Choosing a suitable policy optimization algorithm to train the RL agent is important. This involves deciding between model-based and model-free algorithms, on-policy and off-policy algorithms, or between value-based and policy-based methods. For the presented navigation task, proximal policy optimization seemed a promising choice. As presented in chapter 4, PPO is among the state-of-the-art RL methods for control applications. For the quadrotor navigation task, PPO allows the agent to learn directly from interaction with the environment without requiring a model of the environment dynamics. Furthermore, it supports continuous action spaces, which is necessary to ensure precise and smooth control in the obstacle avoidance task. Additionally, the clipped surrogate objective of PPO helps stabilize

and control the policy updates. This allows the agent to explore new actions while leveraging knowledge gained from previous iterations, leading to more reliable and controlled behavior in complex environments. Finally, PPO allows for parallel training of the agents, which is a great time saver when learning complex tasks like obstacle avoidance.

Finding a good policy requires the RL algorithm to properly and efficiently explore the state-action space. As we saw in section 4.2.4, PPO has some natural exploration from the stochasticity of its policy. However, it uses an entropy coefficient to introduce additional uncertainty into the action sampling in order to keep the policy from getting too deterministic.

The performance of PPO is highly dependent on the reward function used during training. The reward functions were designed based on the general motivation presented in section 6.1.2 to encourage reliable, efficient, and safe navigation. To motivate the agent to reach its target, rewards are given based on the quadrotor's distance to the target d_{target} (6.13). This is the Euclidean distance between the agent's current and goal positions. The reward function should yield a higher reward the smaller this distance is, so some suitable candidates for position reward functions include inverse square and exponential decay.

$$d_{\text{target}} = \sqrt{(x - x_{\text{target}})^2 + (y - y_{\text{target}})^2 + (z - z_{\text{target}})^2} \tag{6.13}$$

Reward function design in the free environment

In the free environment, agents are rewarded based on closeness to the target d_{target} and on certain desirable behaviors while penalized for kills. Success is defined as the quadrotor reaching its target area, i.e., within a radius of $0.5m$ outside its goal position. An agent going out of bounds, i.e., $d_{\text{target}} > 14.5$, is killed and counted as a failure. Agents are also killed if they fail to reach the target by the end of an episode.

The total reward R_{free} 6.14 consists of a position reward term R_p and four behavioral reward terms. The position reward will be defined to encourage the agent to minimize the distance to its target. However, this is the focus of the reward-shaping experiments and will be presented in detail in section 6.5. Here we go through the common reward function setup of the behavioral terms in the free environment. The behaviors we wish to see when the agent closes in on the target are low speeds or stationarity, minimal action inputs, staying upright, and no spinning.

$$R_{\text{free}} = \begin{cases} -5 & \text{if } d_{\text{target}} > 14.5 \\ R_p + R_p(R_v + R_a + R_s + R_u) & \text{otherwise} \end{cases} \tag{6.14}$$

where R_v is the velocity reward, R_a the action input reward, R_u the uprightness reward, and R_s the spinning reward. Multiplying the sum of these terms by the position reward will ensure these behaviors have stronger effects the closer the agent is to the target.

The velocity term is given by:

$$R_v = \frac{K_v}{\beta_v + \boldsymbol{v}^2} \tag{6.15}$$

From (6.15), we see the smaller velocity, the higher reward. However, the position reward multiplication ensures that this only applies close to the target, so it does not particularly limit the speed during most of the navigation process. Having a smaller velocity when reaching the target will help avoid overshooting the target and encourage the agent to stay put.

The action reward term R_a (6.16) encourages small action inputs; thrust f, roll ρ, and pitch θ. Like before, the effects are only significant when the agent is close to the target.

$$R_a = \frac{K_f}{\beta_f + f^2} + \frac{K_\rho}{\beta_\rho + \rho^2} + \frac{K_\theta}{\beta_\theta + \theta^2} \tag{6.16}$$

The spinning term is given by:

$$R_s = \frac{K_s}{\beta_s + \dot{\phi}^2} \tag{6.17}$$

where $\dot{\phi}$ is the yaw rate (spinning) around the up axis.

The reward for uprightness is given as:

$$R_u = \frac{K_u}{\beta_u + \lambda^2} \tag{6.18}$$

where $\lambda = \left| 1 - \frac{\epsilon_2}{\sqrt{1-\eta^2}} \right|$ is the tiltage defined through parts of the quaternion vector.

Table 6.1 gives the common reward parameters used in the free environment. These are found through experimentation.

Velocity	$K_v = 1.0$	$\beta_v = 0.2$
Action	$K_f = \beta_f = 1.0$	$K_\rho = \beta_\rho = K_\theta = \beta_\theta = 0.5$
Spinning	$K_s = 0.4$	$\beta_s = 0.2$
Uprightness	$K_u = 1.0$	$\beta_u = 0.5$

Table 6.1: Common reward function terms for the free environment

Reward function design in the cluttered environment

In the obstacle-filled environment, agents are rewarded based on closeness to the target d_{target} and on certain behaviors. Success is defined as the quadrotor reaching its target area, i.e., within a radius of $1.0m$ outside its goal position, without bumping into obstacles. A collision happens if the robot is within a certain radius of an obstacle, i.e., if the distance to the closest obstacle is less than 0.2, $d_{\text{obst}} \leq 0.2$. If the robot has not crashed but has not yet reached its target by the end of the episode, the attempt is classified as a timeout.

The total reward R_{obst} 6.19 consists of a position reward term R_p and three behavioral reward terms. As before, the position reward will be defined to encourage the agent to minimize the distance to its target, and its design will be presented in detail in section 6.5. Here we go through the common reward function setup of the behavioral terms.

The reward function was designed to encourage safe and stable behavior not only at the target but also during flight, which we overlooked in the free environment. To minimize the risk of collisions and unsafe behavior, we wish to encourage the agent to go in the direction of the target, have a small difference in action input from one step to the next, and avoid fast spinning around its own axis.

$$R = \begin{cases} -6 & \text{if } d_{\text{obst}} \leq 0.2 \\ R_p + R_{\text{adp}} + R_\phi + R_h & \text{otherwise} \end{cases} \tag{6.19}$$

where R_{adp} is the action difference penalty, R_ϕ is the yaw reward, and R_h the direction or heading reward.

The action difference penalty R_{adp} (6.20) yields a negative reward if the difference in action input is large. It is meant to encourage safety and stability by preventing large jumps in action input, which could cause fast spinning of the propellers and high-frequency switches in input.

$$R_{\text{adp}} = -K_{\text{adp}}\left(|f_{t-1} - f_t|^2 + K_\rho|\rho_{t-1} - \rho_t|^2 + |\theta_{t-1} - \theta_t|^2\right) \tag{6.20}$$

where $f_{t-1}, \rho_{t-1}, \theta_{t-1}$ are the thrust, roll, and pitch from the previous timestep.

The yaw reward R_ϕ is given as follows:

$$R_\phi = K_\phi \cdot \exp\left(\frac{-\phi^2}{\beta_\rho}\right) \tag{6.21}$$

The direction reward is given by:

$$R_h = K_h \cdot \exp\left(\frac{-\epsilon_v^2}{\beta_h}\right) \tag{6.22}$$

where ϵ_v is the velocity error.

Table 6.2 gives the common reward parameters used in the obstacle-filled environment. These are found through experiments.

Action Difference	$K_{\text{adp}} = 0.2$	$K_\rho = 0.5$
Yaw	$K_\phi = 0.02$	$\beta_\rho = 0.05$
Direction	$K_h = 0.05$	$\beta_h = 0.2$

Table 6.2: Common reward function terms in the obstacle-filled environment

Actor-critic network design

As we know from the theory section on proximal policy optimization, its supports parameter sharing. Therefore, we can combine the actor-critic network into a common network that takes the state vector $\boldsymbol{s}_t$ as input. The actor-network head outputs the stochastic policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t)$ represented by the means $\boldsymbol{\mu}_t$ and standard deviations $\boldsymbol{\sigma}_t$ of the action probability distribution. The distribution is chosen to be Gaussian. The critic network head gives the value estimate V_t. Due to the difference in complexity of the navigation task in obstacle-free and obstacle-filled environments, the two network structures were designed differently.

The actor-critic network structure for the obstacle-free navigation is shown in figure 6.9. The network is a fully connected neural network, also called multi-layer perceptron (MLP), with three hidden layers with 256 nodes each. It uses rectified linear unit activation (ReLU) (6.23), which is recommended for most feedforward neural networks [16].

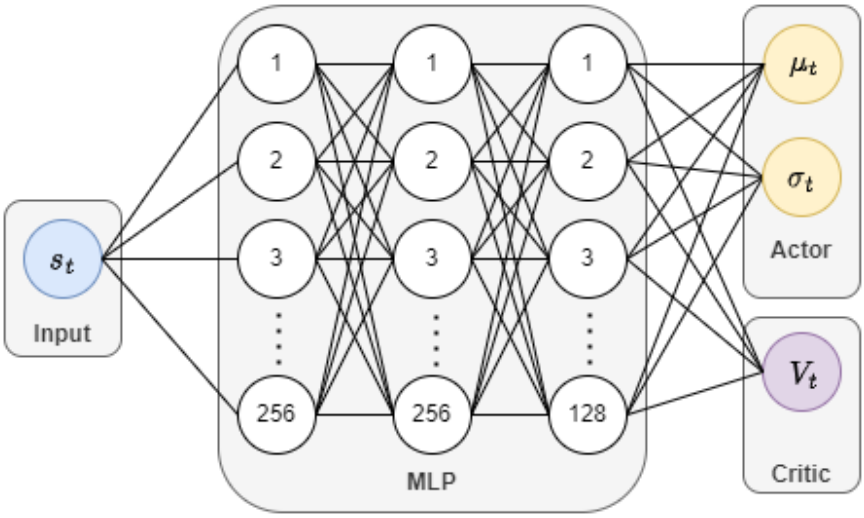


Figure 6.9: Actor-Critic Network Structure in the free environment - The network input is the observation vector $\boldsymbol{s}_t$ of quadrotor states. The outputs of the network are the estimated value function V_t and the means μ_t and standard deviations σ_t of the policy distribution $\pi_{\boldsymbol{\theta}}$ from which actions $\boldsymbol{a}_t$ are sampled.

$$g_{\text{ReLU}}(x) = \max(0, x) \tag{6.23}$$

For the obstacle-filled environment, the actor-critic network structure is given in figure 6.10. Complex environments demand a more elaborate network structure, so the AC network is an MLP with four hidden layers, with 512, 512, 256, and 64 nodes, respectively. It uses exponential logarithmic unit (ELU) activation (6.24), an alternative to ReLU, shown to improve model accuracy, generalization, and learning speed for deeper neural networks [27].

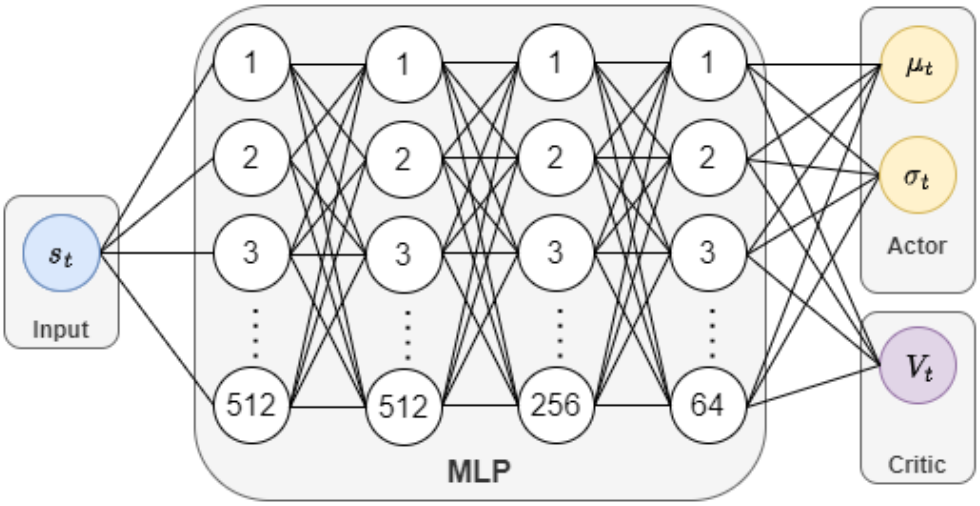


Figure 6.10: Actor-Critic Network Structure in the obstacle-filled environment - The network input is the observation vector $\boldsymbol{s}_t$ that consists of the quadrotor states and the depth image representation. The outputs of the network are the estimated value function V_t and the means μ_t and standard deviations σ_t of the policy distribution $\pi_{\boldsymbol{\theta}}$ from which actions $\boldsymbol{a}_t$ are sampled.

$$g(x)_{\text{ELU}} = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases} \tag{6.24}$$

Adam optimization

The optimization technique used in this project is Adaptive Moment Estimation (Adam). Adam is, according to the authors Kingma and Ba, "a simple and computationally efficient algorithm for gradient-based optimization of stochastic objective functions" [28]. Due to its superior performance and many advantages, it is a popular alternative to stochastic gradient descent for updating network parameters in deep neural networks. These advantages include managing sparse gradients and non-stationary objectives, simple implementation, small memory requirements, and efficiency for large datasets and/or parameter spaces.

6.3.2 Depth Representation Learning

The proposed solution for learning depth representation is an encoder network trained through a variational autoencoder (VAE). The VAE learns to compress the original input in a way that keeps important features of the original image and attempts to reconstruct the original input from the compression. If the VAE shows good reconstruction abilities,

the lower dimensional code will be a reasonable representation of the original image. The network should not be too complex to avoid delays in the depth encoding, yet able to represent the main features of the depth data.

Variational autoencoders(*)

A VAE consists of two interconnected probabilistic models: an encoder and a decoder [29]. The encoder, also known as the recognition model, approximates the posterior distribution of latent variables given the input data. The decoder, or generative model, uses this approximation to update its parameters through an "expectation maximization" learning process. The VAE trains these two networks simultaneously to produce a lower-dimensioned representation, or latent space representation, of the original input data (encoding) and then attempt to reconstruct the original data from the reduced representation (decoding) [30]. The structure of a VAE is illustrated in figure 6.11.

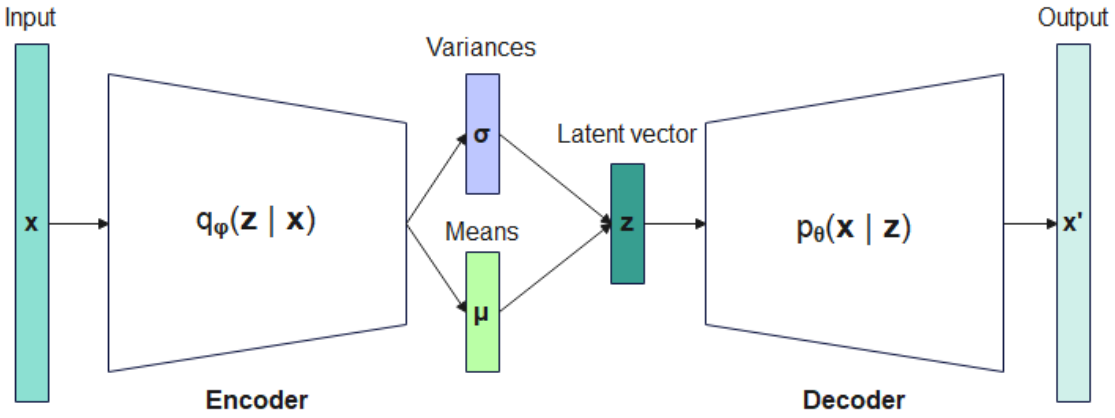


Figure 6.11: Variational Autoencoder Architecture - The encoder takes the data $\boldsymbol{x}$ as input and learns an approximation $q_\phi(z|x)$ of the intractable true posterior $p_\theta(z|x)$. Then encoder produces a prior $p_\theta(z)$ over the latent space variables $\boldsymbol{z}$, represented by means $\boldsymbol{\mu}$ and variances $\boldsymbol{\sigma}$. We get the latent vector $\boldsymbol{z}$ by sampling from these latent distributions. From $\boldsymbol{z}$, the decoder network learns a model to generate new data $\boldsymbol{x}'$ as similar as possible to the original data $\boldsymbol{x}$.

An encoder network is used to compress the depth image data to a dimension suitable to be used as parts of the state vector. The main motivation for doing this is to lower the complexity requirement of the policy network, together with the time and computation complexity of training. The encoder is trained to produce a probability distribution over the compressed data, or latent vector, from which the original data could have been generated [30]. The probabilistic encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ is also called a recognition or inference model. It

produces a distribution over the possible latent variables $\boldsymbol{z}$ that could have generated a datapoint $\boldsymbol{x}$. Thus, the encoder learns a lower-dimensional representation that preserves the key features of the original input data.

The decoder network samples the latent vector $\boldsymbol{z}$ from the means $\boldsymbol{\sigma}$ and variances $\boldsymbol{\mu}$ of the multivariate Gaussian produced by the encoder. The probabilistic decoder $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ produces a distribution over the possible corresponding $\boldsymbol{x}$ that could have been generated from a code $\boldsymbol{z}$.

VAE objective(*)

The optimization objective of the variational autoencoder is the *evidence lower bound* (ELBO) (6.25) [30]. ELBO maximization simultaneously improves (1) the recognition model by approximately minimizing the KL divergence of the true and approximate posterior, and (2) approximately minimizes the marginal likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ [29, 30].

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x}) = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) - D_{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})] \leq \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \tag{6.25}$$

From (6.25), we have that the difference between $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ and $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\boldsymbol{x})$ is the KL divergence. The KL-divergence is non-negative; thus, the ELBO is at most equal to the desired log-probability [16, 29]. They are equal if the distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ equals $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$.

VAE network for the navigation task

This project aims to achieve obstacle avoidance in environments with a high density of obstacles. Therefore, it proposes a latent vector $\boldsymbol{z}$ of dimension 128 to properly represent the depth data of its environment. The somewhat large size might cause delays in the total system, but it might be essential in the most cluttered environments. The VAE architecture is shown in figure 6.12.

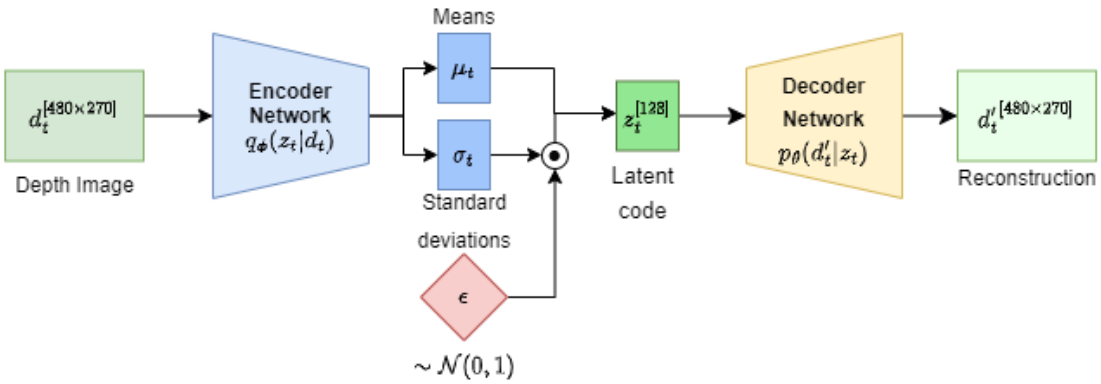


Figure 6.12: Variational Autoencoder Network Architecture - The original depth image d_t is input to the encoder network, producing the means μ_t and standard deviations σ_t of the approximate posterior distribution q_ϕ. Using the parameterization trick, adding uncertainty with the variable ϵ sampled from a Gaussian distribution, we can sample the latent code z_t from μ_t and σ_t. The decoder network learns to produce the reconstruction d'_t given the latent representation.

The encoder network architecture was inspired by DroNet, a prediction network for quadrotor navigation presented by Loquercio et al. [8]. DroNet successfully used RGB images combined with angle measurements to predict steering angles and collision probabilities, making it a promising starting point for deriving the encoder architecture of this project's task. The quadrotor training will be done in a simulator that produces error-free depth images in controlled environments with predictable obstacle shapes. This makes the need for a complex network structure lower compared to that of DroNet using real-world RGB images. Thus, the encoder network in this project ended up being a somewhat condensed version of DroNet.

The encoder structure design is shown in figure 6.13. It combines residual blocks with convolutional and fully connected layers. The special structure of residual networks has been shown to help generalization in shallow and deep networks [31] and help tackle the vanishing/exploding gradient problems that can appear in deep networks [8]. The convolutional layers are used due to their ability to capture features in data efficiently. The convolution operation can be seen as a filter, capable of extracting different types of features from the input data [16]. In convolutional neural networks (CNNs), the shallow layers often have simple filters to detect features like edges or corners. As we move deeper into the network, we typically find more complicated filters that detect shapes, patterns, or even full objects. This makes them useful in tasks involving image processing and/or object detection.

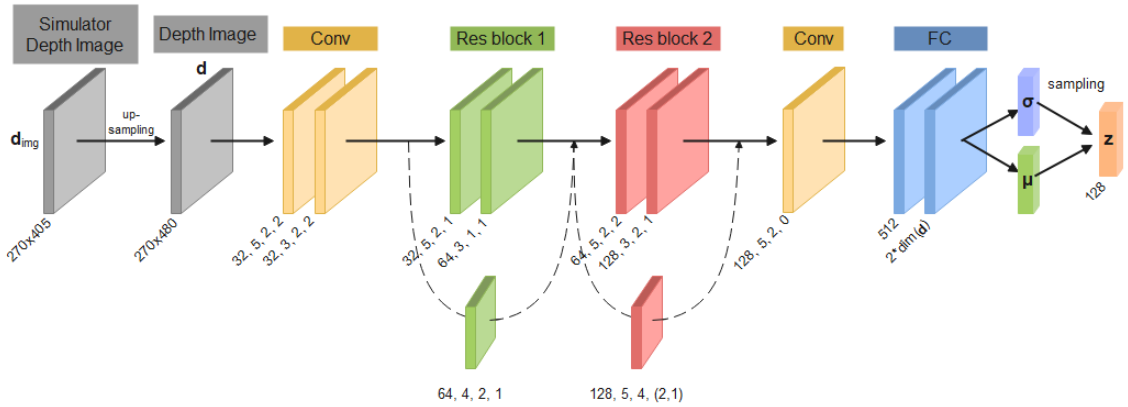


Figure 6.13: Encoder network structure - the arrows indicate the output from one layer being sent as input to the next, while the thin boxes represent a network layer. The numbers below each layer indicate its output dimension, kernel size, stride, and padding. Jump connections are indicated with dashed lines.

Primarily, the image from the simulation camera $\boldsymbol{d}$ is upsampled to match the input of the encoder, i.e., (270, 480). Then the upsampled version is input to the first two convolutional layers. Thereby follows the residual layers, structured as illustrated in figure 6.14. The output from these blocks is sent through a final convolutional layer, and a two-layered fully connected network produces the means and standard deviations used to sample the latent code $\boldsymbol{z}$.

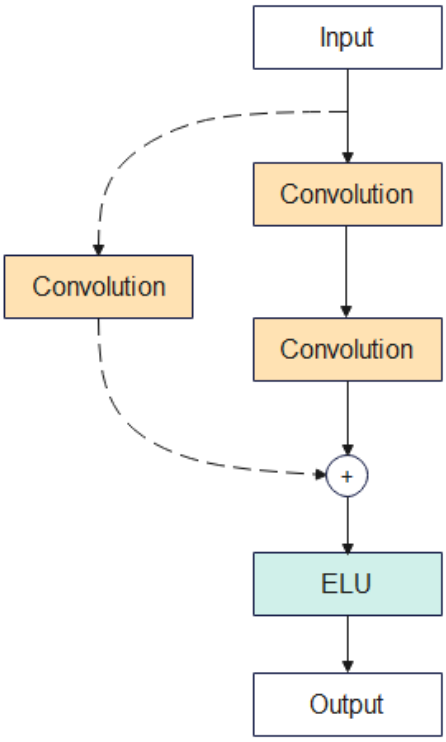


Figure 6.14: Residual block structure - the layer input is sent through two convolutional layers and added together with a downsampled version of the original layer input and sent through the ELU activation to produce the output.

The network structure of the decoder is shown in figure 6.15.

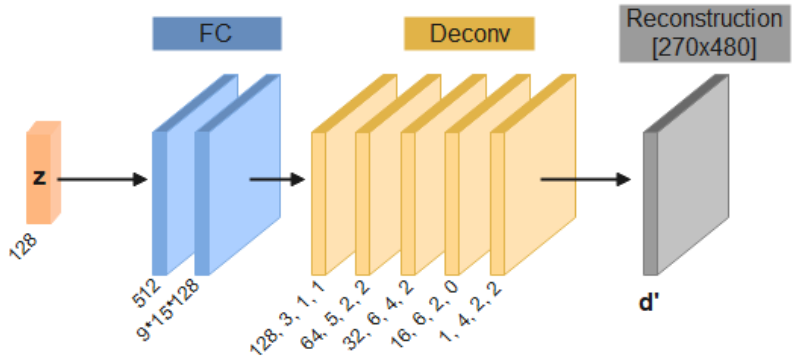


Figure 6.15: Decoder network - the number of output channels, kernel size, stride, and padding of each deconvolutional layer is indicated. The output size is indicated for the fully connected (FC) layers.

The decoder network samples the latent vector $\boldsymbol{z}$ from the means $\boldsymbol{\sigma}$ and variances $\boldsymbol{\mu}$ of the multivariate Gaussian produced by the encoder. Two fully connected layers (FC),

followed by a deconvolutional network performing transpose convolutions, produce the reconstruction $\boldsymbol{d}'$.

VAE Training

The VAE is trained by the Auto-Encoding Variational Bayes algorithm [30], as seen in 4. To bridge the simulation-to-reality gap, the VAE was trained using images from a simulation environment and real-world environments. The dataset consisted of depth images generated from simulator environments with different obstacle types and a selection of depth images from the NYU-depth V2 dataset [32], taken by a Microsoft Kinect in real-world indoor environments. Figure 6.16 shows a representative selection of example depth images, along with the reconstructions produced by the VAE in figure 6.17.

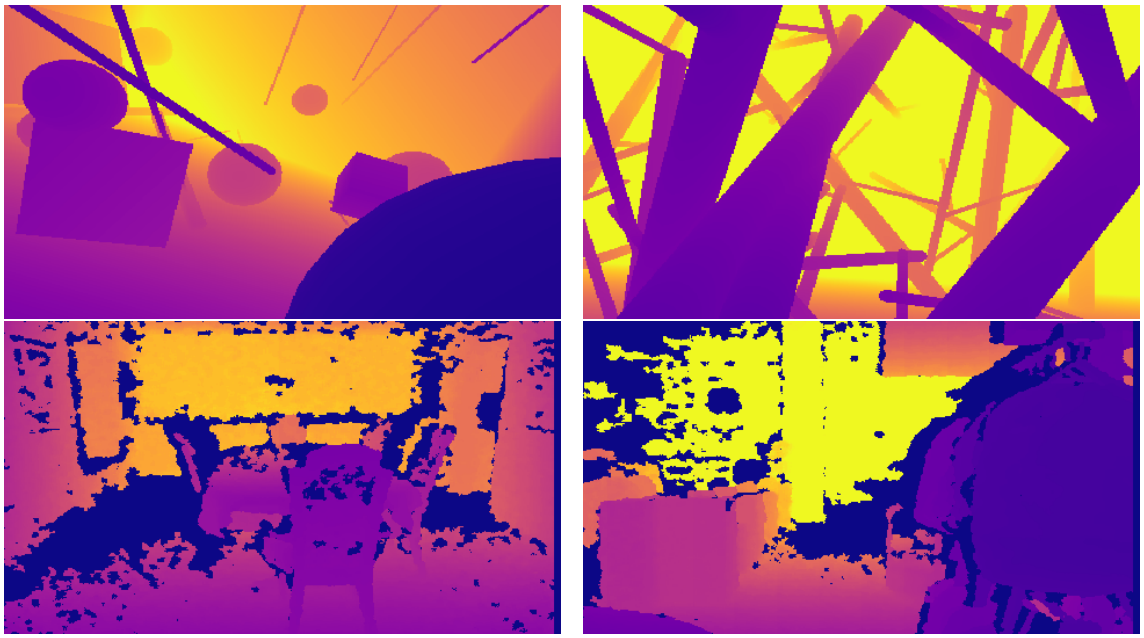


Figure 6.16: Example depth images - The top images show geometric and thin obstacles from a simulator camera. The bottom images are taken from the NYU-depth V2, where missing depths are set to zero and thus displayed as the darkest color of the depth map.

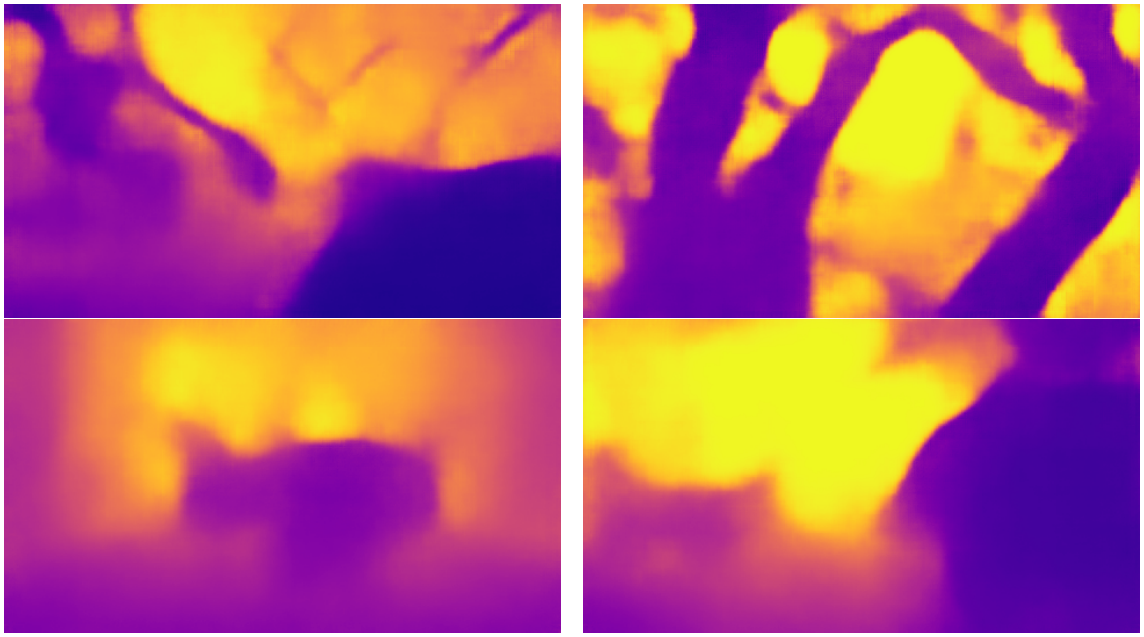


Figure 6.17: Example reconstructed depth images - The top images show geometric and thin obstacles from a simulator camera. The bottom images are taken from the NYU-depth V2.

6.3.3 Final Solution Summary

Figure 6.18 and 6.19 summarize the solutions to be implemented in the following section. Their explanations are simplified as they show everything happening at a single timestep. In reality, the system collects a minibatch of trajectories per parameter update, as explained in section 4.2.3. Furthermore, this process was done for multiple agents simultaneously.

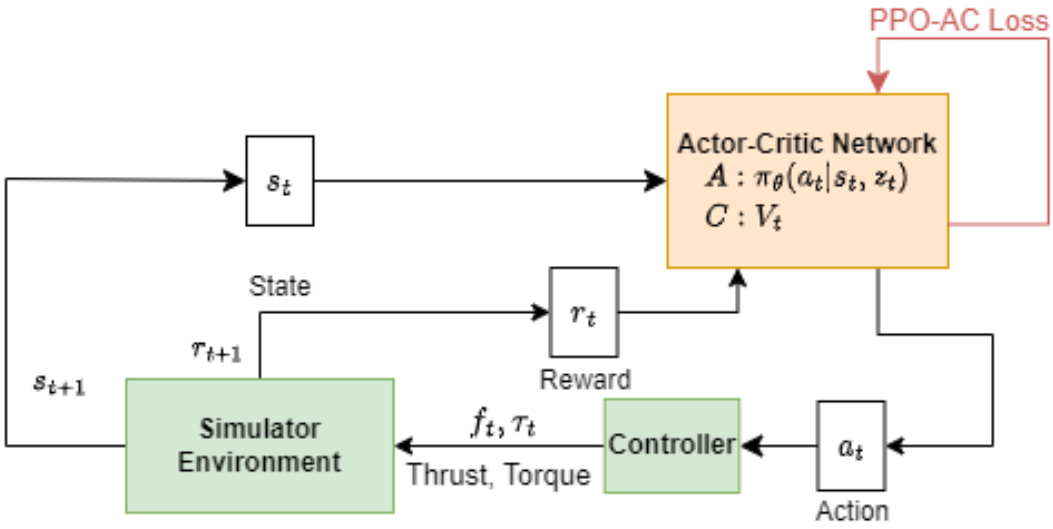


Figure 6.18: Solution overview in the obstacle-free environment - the simulation environment provides state observations s_t and rewards r_t to the PPO actor-critic network that outputs the desired action commands a_t and value estimates V_t. The AC network parameters are updated through the PPO-clip objective loss. The action commands are fed through a low-level controller to produce thrust f_t and torque τ_t commands to be applied to the quadrotor agent, which transitions to a new state s_{t+1} and calculates the corresponding reward r_{t+1}.

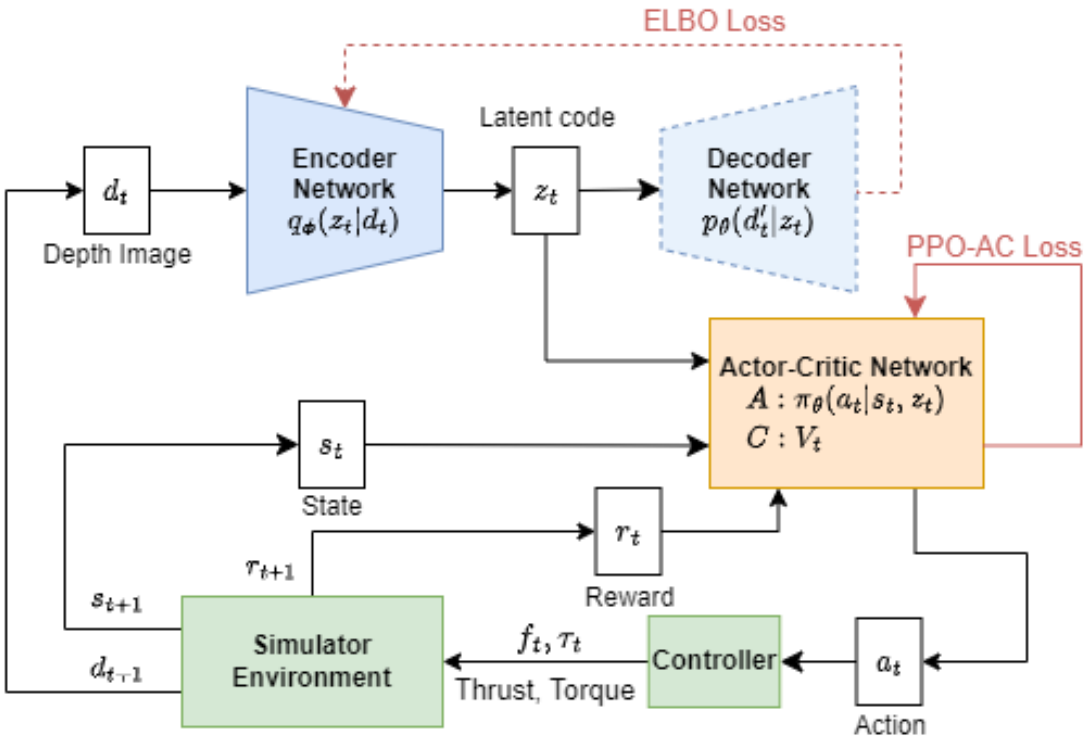


Figure 6.19: Solution overview in the obstacle-filled environment - the Isaac Gym environment provides state observations s_t and rewards r_t. Additionally, the simulator camera provides a depth image d_t, compressed into latent code z_t through the encoder network. The encoder network was trained through a VAE network by minimizing the ELBO loss. The quadrotor states and latent code serve as input to the PPO actor-critic network that outputs the desired action commands a_t and value estimates V_t. The AC network parameters are updated through the PPO-clip objective loss. The action commands are sent through the low-level controllers to produce the thrust f_t and torque τ_t to be applied to the quadrotor agent, which transitions to a new state s_{t+1} and calculates the corresponding reward r_{t+1}.

6.4 Implementation

This section presents the implementation of the navigation task. This includes the environmental setup in Isaac Gym, how the networks were implemented through PyTorch, and introduces the reinforcement learning framework RL Games.

6.4.1 Quadrotor Navigation in Isaac Gym

The simulation environment used for conducting the experiments is NVIDIA Isaac Gym [25]. They provide a learning platform designed for training control policies directly on

GPUs, enabling parallel simulations and faster training times. They also offer a simple API that allows the creation and setup of environments with agents and objects and loading data from commonly used URDF and MJCF file formats, making it convenient to import pre-existing models into the simulation. Furthermore, it comes with a basic Proximal Policy Optimization (PPO) implementation and an RL task system that can be substituted as the user desires.

To solve the quadrotor navigation task, the RL agent, environment, and state and action spaces were adapted from the existing standard tasks provided by Isaac Gym.

Agent

The *actor* or agent in Isaac Gym is an entity of rigid bodies connected via joints [25]. The quadrotor agent was created from a URDF model file and is displayed in figure 6.20.

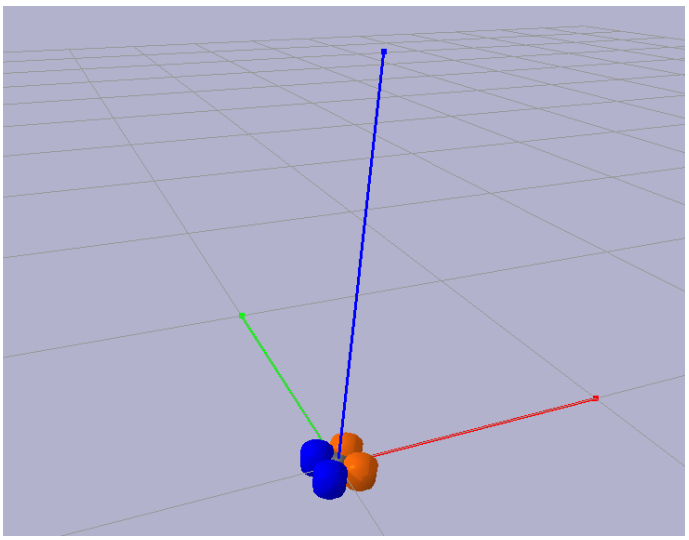


Figure 6.20: Quadrotor Model in the Obstacle-filled Environment - The quadrotor agent is modeled as a rigid body consisting of four connected disks that represent the individual propellers.

Environment

The free environment is an empty indefinite 3D environment where the agent can move freely. Figure 6.21 show the Isaac Gym free environment for $N = 16384$ parallel agents.

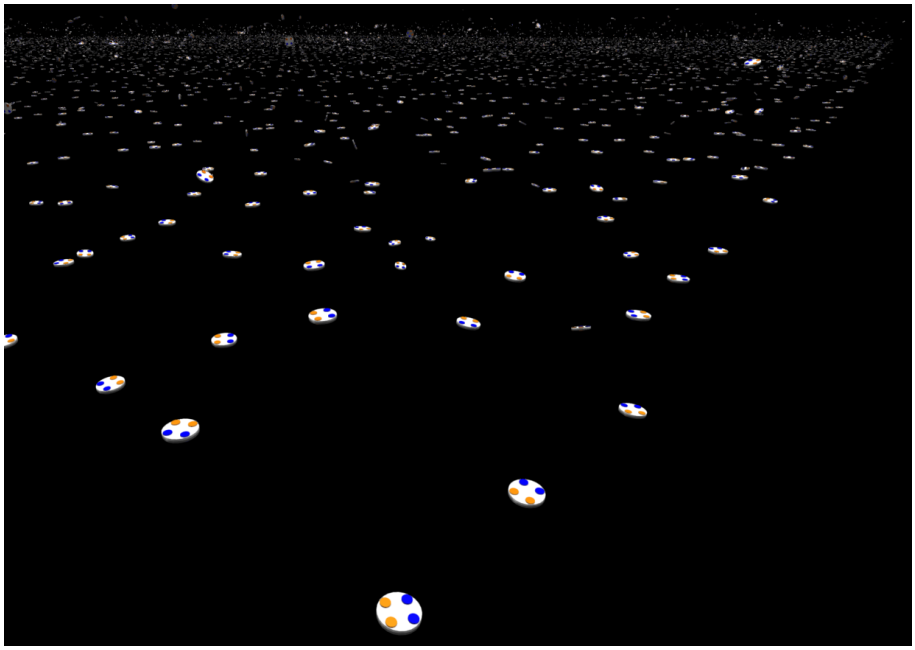


Figure 6.21: Free environment in Isaac Gym - Each quadrotor agent is modeled as a disk with four smaller colored disks representing the propellers.

The obstacle-filled environment is defined as a fixed-size box in 3D space. It trains $N = 128$ parallel agents, as shown in figure 6.22.

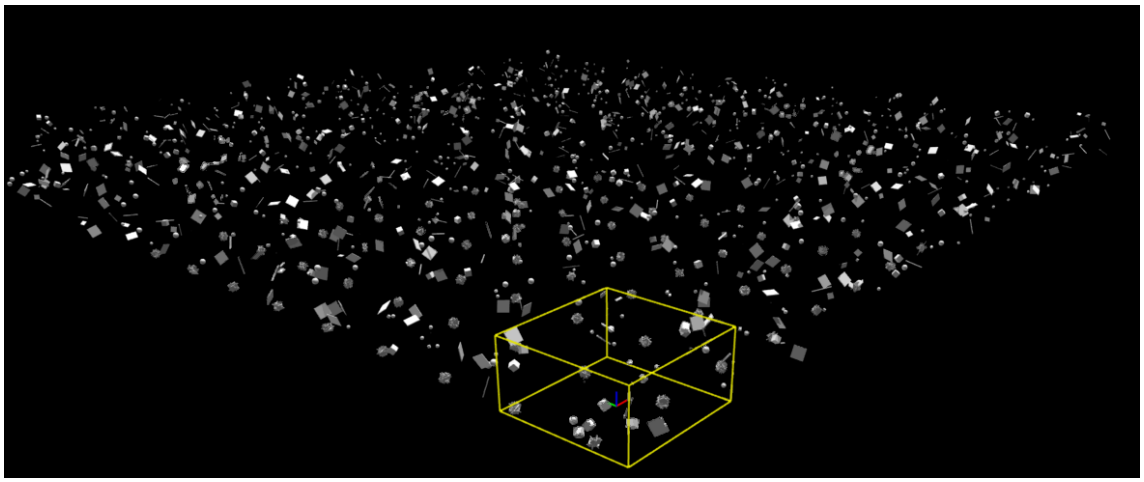


Figure 6.22: Obstacle-filled environments in Isaac Gym - for $N = 128$ parallel agents.

The obstacles are simple 3D geometric shapes, i.e., different-sized cuboids and spheres. The cuboids are generated with a tendency to align cubes, planes, and thin poles. The objects are placed randomly, and the placement changes for each run and is different for

each environment, creating a variety of training samples.

Four environments with different obstacle density levels are chosen and later used in the experiments. The environments will be referred to as E10, E20, E30, and E50, the numbers representing the number of obstacles present in each environment. Screenshots from example environments with the four different obstacle densities are shown in figure 6.23.

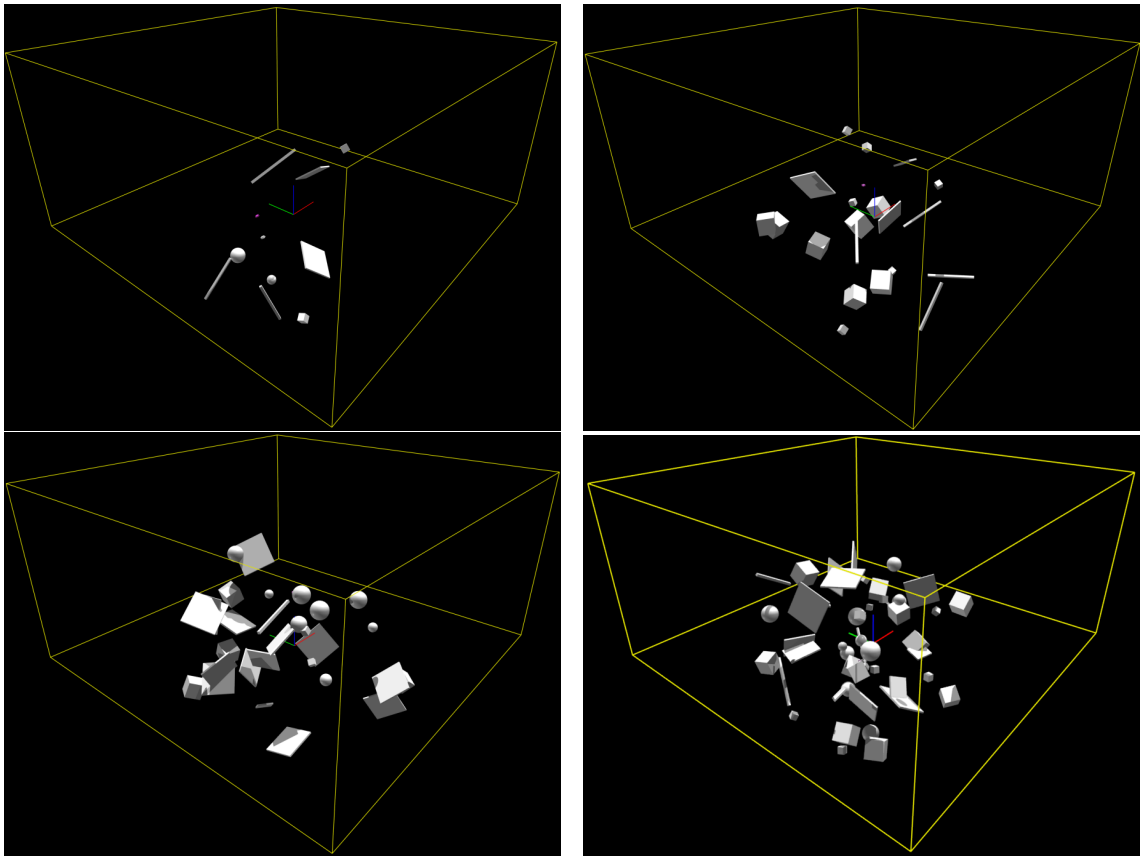


Figure 6.23: Example environments with different obstacle densities - (1) Top left: E10, (2) Top right: E20, (3) Bottom left: E30, (4) Bottom right: E50.

States and Targets

An actor's positions, rotations, and velocities can all be obtained via the Isaac Gym API. The states for all parallel environments are given in a single tensor, called the root state tensor, which is obtained from the Isaac Gym environment by a call to the Isaac Gym function `aquire_root_state_tensor`. This tensor also holds the target positions and obstacle states of each environment. The target is defined as a sphere as illustrated in figure 6.24; however,

it will not yield a collision with the quadrotor.

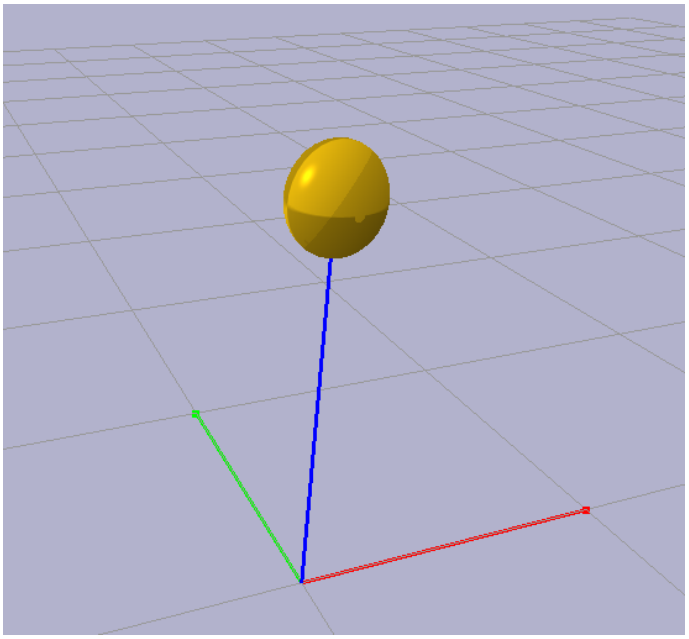


Figure 6.24: Target - The target region is defined as a sphere of radius equal to the target reaching radius, which is 0.5 in the free environment and 1.0 in the obstacle-filled environment. The inertia frame is indicated by i_1 in red, i_2 in green, and i_3 in blue.

Simulator camera

The depth image is of size 270×405 and compressed to a vector of size 128 by the encoder network described in section 6.3.2.

Velocity controller

A PID controller with $\boldsymbol{K}_P = \begin{bmatrix} 4.0 & 4.0 & 4.0 \end{bmatrix}$, $\boldsymbol{K}_I = \begin{bmatrix} 0.3 & 0.3 & 0.3 \end{bmatrix}$, and $\boldsymbol{K}_D = \begin{bmatrix} 0.1 & 0.1 & 0.1 \end{bmatrix}$ controls the quadrotor velocities in the obstacle-filled environment.

6.4.2 Python API

Isaac Gym provides a Python API, and the *RL Games* [33] library was used to implement the navigation task. The quadrotor agent was defined in Python as a class instance of type `Quadrotor` for the free environment or `QuadrotorWithObstacles` for the obstacle-filled environment. The two classes inherit functions from a general task class called `VecTask` that provide basic functions for initialization, memory allocation, and creating the environment.

The two classes defined specifically for the quadrotor task define two additional functions `pre_physics_step` and `post_physics_step`. Given the actions as input, the first function applies the actions to the environment by setting torques, thrusts, and position targets. The second function computes the rewards and new observations and handles resets where a collision or final timestep has occurred. RL Games also provides an actor-critic implementation of PPO that calculates the PPO-clipp loss and updates the policy and value network parameters. A class called `ActorCritic` defines the actor-critic network and handles the action sampling.

It is also worth mentioning that RL Games clip their actions and observations by normalizing them. For stochastic values, this means giving them a zero mean and standard variation of 1. Deterministic values are scaled to be between -1 and 1 or 0 and 1.

Configurations

There are a few different types of configuration files. One defines the type of RL algorithm used along with its network and hyperparameter settings. Another describes the agent, i.e., the `Quadrotor` or `QuadrotorWithObstacles` objects. This is where simulation and environment parameters can be set.

Python Modules

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. In the transformations between Euler angles, quaternions, and rotation matrices, the module `Pytorch3d` [34] was used.

TensorFlow(*)

TensorFlow is an end-to-end open-source platform developed by Abadi et al. for creating and training machine learning models [35]. TensorFlow provides many useful building blocks for creating machine learning models, including fully-connected layer blocks, convolutional layer blocks, activation functions, and deconvolutional layer blocks. The system networks were implemented by combining these TensorFlow blocks.

TensorBoard was developed as a visualization tool for understanding the machine learning model behavior and the structure of the computation graphs [35]. This project used this tool to visualize and compare the performance metrics for different reward functions used for RL agent training.

6.5 Experiment

Several experiments were completed to provide insight into and answer the research questions of the thesis. This section presents the setup of the experiments performed, including the hyperparameters and the position reward design. Recall that the common behavioral reward setups were presented in section 6.3.1.

6.5.1 Obstacle-Free Environment

PPO with three different choices of position reward shapes was tested in the free environment.

Common hyperparameter settings

Table 6.3 shows the used hyperparameter values that were held constant during the presented experiments in the free environment. If altered in some experiments, the value is shown in parenthesis and stated in the relevant experiment. The hyperparameters were chosen based on recommended or commonly used values.

Hyperparameter	Value	Description
γ	0.99	Discount factor
τ	0.95	Generalized Advantage Estimation (GAE) coefficient
ϵ	0.2	Clipping coefficient
bounds_loss_coefficient	0.0001	Bounds loss coefficient
critic_coefficient	2	Critic coefficient
entropy_coefficient	0.0	Entropy coefficient
α	$1e^{-3}$	Learning rate (adaptive)
$\delta_{\max}$	0.016	Threshold for KL-divergence
M	32768	Mini-batch size
N	16384	Number of environments (parallel agents)
T	250	Maximum number of steps in an episode
max_epochs	5000	Maximum number of epochs

Table 6.3: Common hyperparameters for the free environment

Experiment F.1 - Comparing rewards

This experiment aims to compare the efficiency and performance potential of different reward functions. The position reward shapes chosen for the test were negative quadratic (6.26), inverse square (6.27), and exponential decay (6.28) with d_{target} used as the input variable.

$$R_p = R_{\text{q}} = 10 - 0.4d_{\text{target}}^2 \tag{6.26}$$

$$R_p = R_{\text{is}} = \frac{10}{1 + d_{\text{target}}^2} \tag{6.27}$$

$$R_p = R_{\text{exp}} = 10e^{-0.4d_{\text{target}}} \tag{6.28}$$

The quadratic function forms a downward-opening parabola, yielding a rapid increase in function value as the input variable moves closer to zero. The exponential and inverse square function has a steep increase near zero and a gentler decline as the input variable moves away from zero. As seen in figure 6.25, each reward function was designed to have the same peak reward value of 10. Furthermore, they all have the same kill penalty of -5, and all other settings and parameters were kept constant during the experiment.

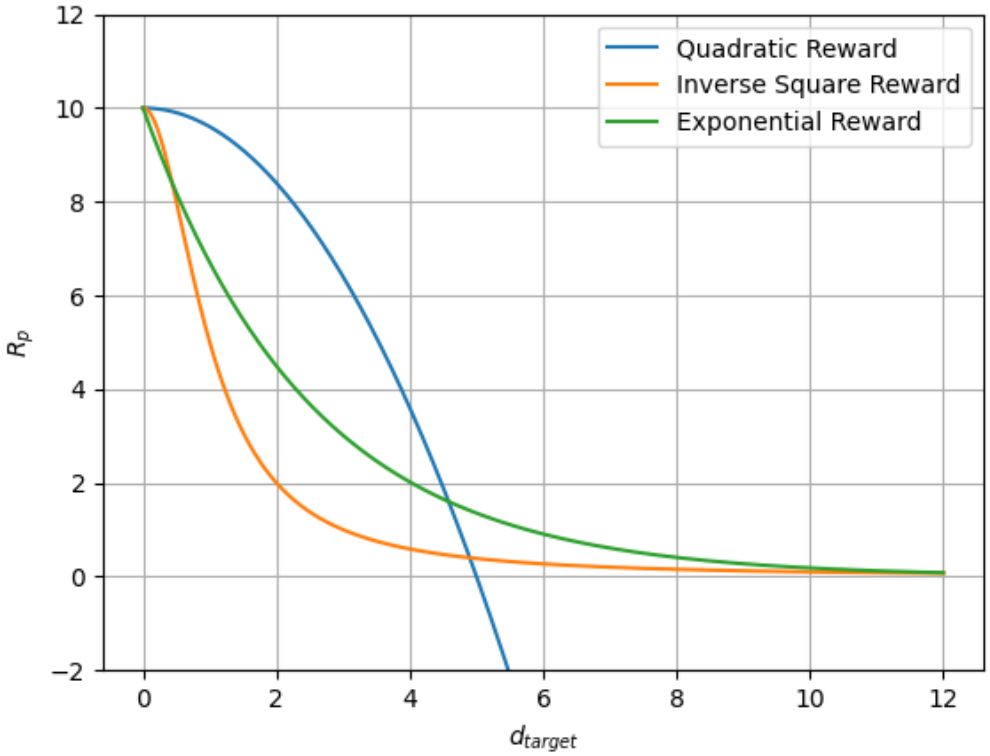


Figure 6.25: Reward functions in the free environment - comparison of the quadratic, inverse square, and exponential reward functions.

6.5.2 Obstacle-Filled Environment

PPO with various reward functions was tested in different environmental complexities in the obstacle-filled environment.

Common hyperparameter settings

Table 6.4 shows the used hyperparameter values that were held constant during the presented experiments in the obstacle-filled environment. They were chosen based on common choices and recommendations from other experiments.

Hyperparameter	Value	Description
γ	0.99	Discount factor
τ	0.95	Generalized Advantage Estimation (GAE) coefficient
ϵ	0.2	Clipping coefficient
bounds_loss_coefficient	0.0001	Bounds loss coefficient
critic_coefficient	2	Critic coefficient
entropy_coefficient	0 (0.01)	Entropy coefficient
α	$3e^{-4}$	Learning rate (adaptive)
$\delta_{\max}$	0.016 (0.02)	Threshold for KL-divergence
M	256	Minibatch size
N	128	Number of environments (parallel agents)
T	180	Maximum number of steps in an episode
max_epochs	15000	Maximum number of epochs

Table 6.4: Common hyperparameters for the obstacle-filled environment

Experiment 1 - Baseline with inverse square reward (PPO-Baseline)

This experiment uses the best-performing position reward, aka the inverse square reward, from the obstacle-free case to form a baseline for the obstacle-filled environment.

The first position reward function R_{is} (6.29) is a single inverse square term that ensures a high reward increment close to the target.

$$R_p = R_{\text{is}} = \frac{10}{1 + d_{\text{target}}^2} \tag{6.29}$$

Experiment 2 - Mixed reward (PPO-Mix)

This experiment investigates the effects of mixing different position reward terms into a single reward function R_{mix}. This allows fine-tuning the shape of the reward in a way that is not possible by only using one type of term.

The position reward function (6.30) is a sum of two inverse square terms, an exponential and a linear term. The three first terms ensure a high reward close to the target, with the main increment happening around 1-2 meters from the target. The linear term ensures a somewhat denser reward further away from the target that might help pull it in the direction of the target in a larger part of its environment. However, not too high a reward, so it won't be too pruned to exploit this reward instead of exploring other behaviors.

$$R_p = R_{\text{mix}} = \frac{0.8}{0.1 + d_{\text{target}}^2} + \frac{2}{1 + d_{\text{target}}^2} + 3e^{-0.4d_{\text{target}}} + \frac{20 - d_{\text{target}}}{20} \tag{6.30}$$

Experiment 3 - Sparse rewards (PPO-Sparse)

The reward function R_{sparse} (6.31) follows some of the same reasoning as R_{mix} being a sum of two inverse square terms, an exponential, a linear, and a constant term. Additionally, the KL threshold was increased from 0.016 to 0.02. Increasing the threshold could lead to larger policy updates and faster learning, which could compensate for sparser rewards.

$$R_p = R_{\text{sparse}} = \frac{0.3}{0.1 + d_{\text{target}}^2} + \frac{5}{1 + d_{\text{target}}^2} + 0.5e^{-0.1d_{\text{target}}} + \frac{8 - d_{\text{target}}}{8} \tag{6.31}$$

Experiment 4 - Enhanced exploration (PPO-Explore)

One of the research questions this thesis aimed to answer was whether increased exploration would positively influence the performance of the PPO algorithm in obstacle-filled environments. This experiment provides insight into this by encouraging more exploration and comparing the results to the second experiment.

To see if the performance could be improved with increased exploration, the same reward function used in the second experiment, i.e., the mixed reward (6.30), was tested again with an entropy coefficient of 0.01 instead of 0.0. As mentioned earlier, the entropy coefficient controls the amount of randomness in choosing actions, thus enhancing exploration. The experiment was conducted in the simplest and the two most complex environments, i.e., E10, E30, and E50.

$$R_p = R_{\text{explore}} = \frac{0.8}{0.1 + d_{\text{target}}^2} + \frac{2}{1 + d_{\text{target}}^2} + 3e^{-0.4d_{\text{target}}} + \frac{20 - d_{\text{target}}}{20} \tag{6.32}$$

Position Reward Comparison

Figure 6.26 shows the inverse square, mixed, and sparse reward functions together for comparison.

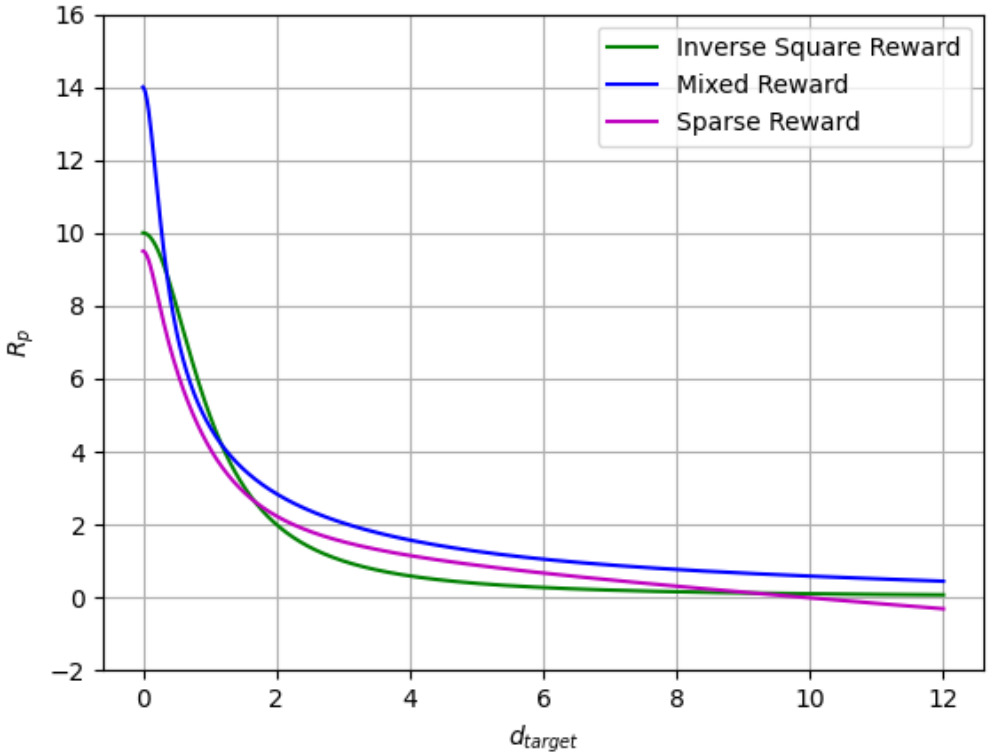


Figure 6.26: Comparison of reward functions - as functions of the target distance.

Chapter 7

Results

7.1 Obstacle-Free Environment

In this section, we will evaluate the performance of the PPO algorithm on a quadrotor in an obstacle-free environment. The performance will be evaluated based on the rate of successful attempts the agent achieves during training, along with the efficiency and stability of the training process. We will also look at the target distances and rewards to understand the learned quadrotor behavior better. Due to the simplicity of the task, one could expect close to 100% success by the end of the training for a reasonable reward function design.

The following experimental results correspond to the experimental setups given in section 6.5.1.

Experiment F.1 - Comparing Rewards

Running PPO with the three reward functions defined in section 6.5.1, we get the success rates shown in figure 7.1. The kill rate is simply the complement of the success rate, found by subtracting the success rate from 1, so we do not plot it explicitly.

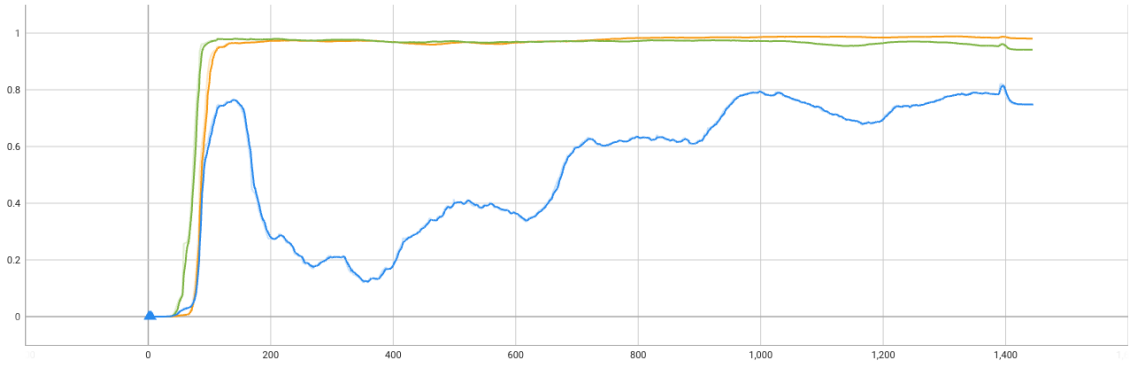


Figure 7.1: Experiment F.1: Comparing success rates of different reward functions - In all plots, we see the quadratic reward (blue), inverse square reward (orange), and exponential reward (green) plotted over the number of training iterations.

The plots show that the exponential and inverse square reward performs quite similarly. The exponential provides a somewhat denser reward further away from the target, which could be why it learns slightly faster than the inverse square. However, the inverse square ultimately reaches a higher success of 98.8%. Likely, the small number of unsuccessful agents is likely due to the stochasticity of the policy, causing some randomness in the action choices.

PPO with the quadratic reward function yields the poorest performance and slowest convergence. Based on this experiment alone, it is hard to answer why the performance has an early peak and then decays before rising again. However, the comparably worse performance compared to the other two rewards is likely due to the suboptimality of the quadratic function shape in an RL learning perspective. We investigate this further by looking at the absolute distance to the target in figure 7.2 and 7.3.

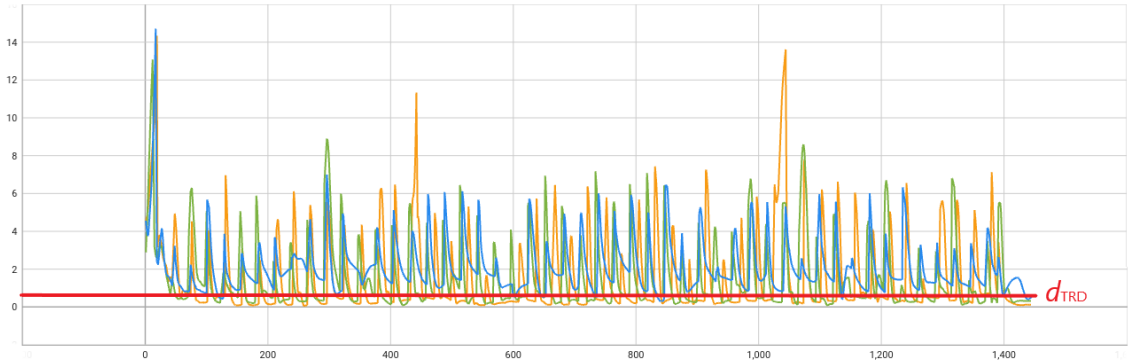


Figure 7.2: Experiment F.1: Target distance plots of different reward functions - The plot shows the target distance for a single environment over all the training episodes. Each peak represents the beginning of an episode. The quadratic reward is shown in blue, the inverse square reward in orange, and the exponential reward in green. The target reaching distance d_{TRD} is displayed as a red line.

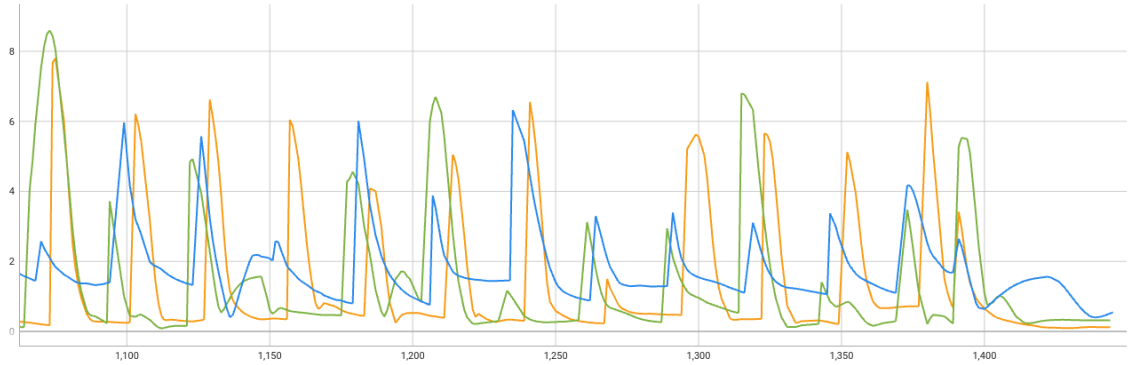


Figure 7.3: Experiment F.1: Target distance plots of different reward functions (zoomed) - The plot shows the target distance for a single environment over some of the final training episodes. Each peak represents the beginning of an episode. The quadratic reward is shown in blue, the inverse square reward in orange, and the exponential reward in green.

In figure 7.2, we see the target distance plotted over all the training episodes from environment zero. A zoomed version showing the final episodes are provided in figure 7.3. The comparison shows that the target distance, except for a few initial episodes, converges exponentially towards a value close to zero for the inverse square and exponential rewards. The quadratic reward also converges, but to higher values, often outside the target reaching radius. This behavior can be explained by the shape of the quadratic function. Around the target, i.e., where $d_{\text{target}} = 0$, the quadratic function is quite flat. This low increment in reward and resulting small gradients make the weight/parameter updates of PPO minuscule, making the agent care little whether it is exactly at the target or somewhere close to

it on the flat region of the quadratic reward surface.

From figure 7.3, we can also deduct something about the flying speed of the quadrotor. The inverse square and exponential rewards have somewhat steeper convergence curves, meaning they fly faster than the agent trained with quadratic rewards, further demonstrating their better performance. We see that the speed is higher further away from the target, which makes sense as the reward function rewards lower velocities close to the target to avoid the quadrotor overshooting it.

Figure 7.4 shows quite similar training rewards for the three runs. First, this might be surprising as the quadratic reward function gave significantly lower success rates during the whole training process. However, let's look back at how the rewards are given only based on closeness to the goal and no additions for actually reaching the goal. This makes perfect sense as the agent receives a high reward even when converging only close to it. The initially negative reward for the quadratic reward can be explained by looking at the plot in figure 6.25. It shows that the quadratic function is negative for $d_{\text{target}} > 5$, which means it will yield negative rewards not only for kills but also for timeouts in a radius larger than 5 from the target. The final decrease in the rewards can be explained by the fact that not all agents finish their episodes at the same time. Some are reset early due to kills, resulting in fewer agents left in training to yield rewards.

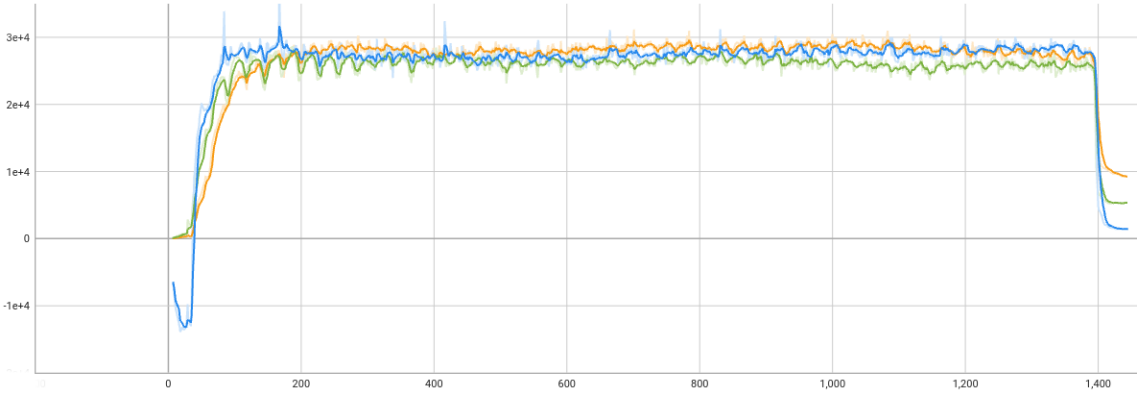


Figure 7.4: Experiment F.1: Rewards - The plot shows the rewards, where the quadratic reward is shown in blue, the inverse square reward in orange, and the exponential reward in green.

7.2 Obstacle-Filled Environment

The performance of the PPO algorithm in obstacle-filled environments will be evaluated based on the success, crash, and timeout percentages the agent achieves for each obstacle

density level. When relevant, we will also have a look at the rewards and losses.

The ideal scenario is to have a high success percentage in combination with a low crash percentage. This means the quadrotor manages to reach its target within the time limit in the main part of its attempts with minimal fatal attempts. In the cases where success is low, we study the number of crashes vs. timeout and possibly the rewards for trying to deduct something about the quadrotor behavior. If the crash rate is low and the timeout rate high, this could indicate that the quadrotor has almost reached its target if rewards are also high. Contrarily, if the rewards are small, this could indicate that the policy is caught in a local optimum where it has learned not to crash but has become overly careful. In cases where the crash percentage is high, this suggests either that the environment is too complex for the algorithm to yield safe navigation or that the policy never found any optima and has diverged.

The following experimental results correspond to the experimental setups from section 6.5.2. In this section, the results are presented, and any unexpected behaviors are commented on to be discussed in chapter 8.

7.2.1 Experiment 1 - Baseline with Inverse Square Reward (PPO-Baseline)

Figure 7.5 shows the success, timeout, and crash statistics of the first experimental setup from the three different obstacle-filled environments.

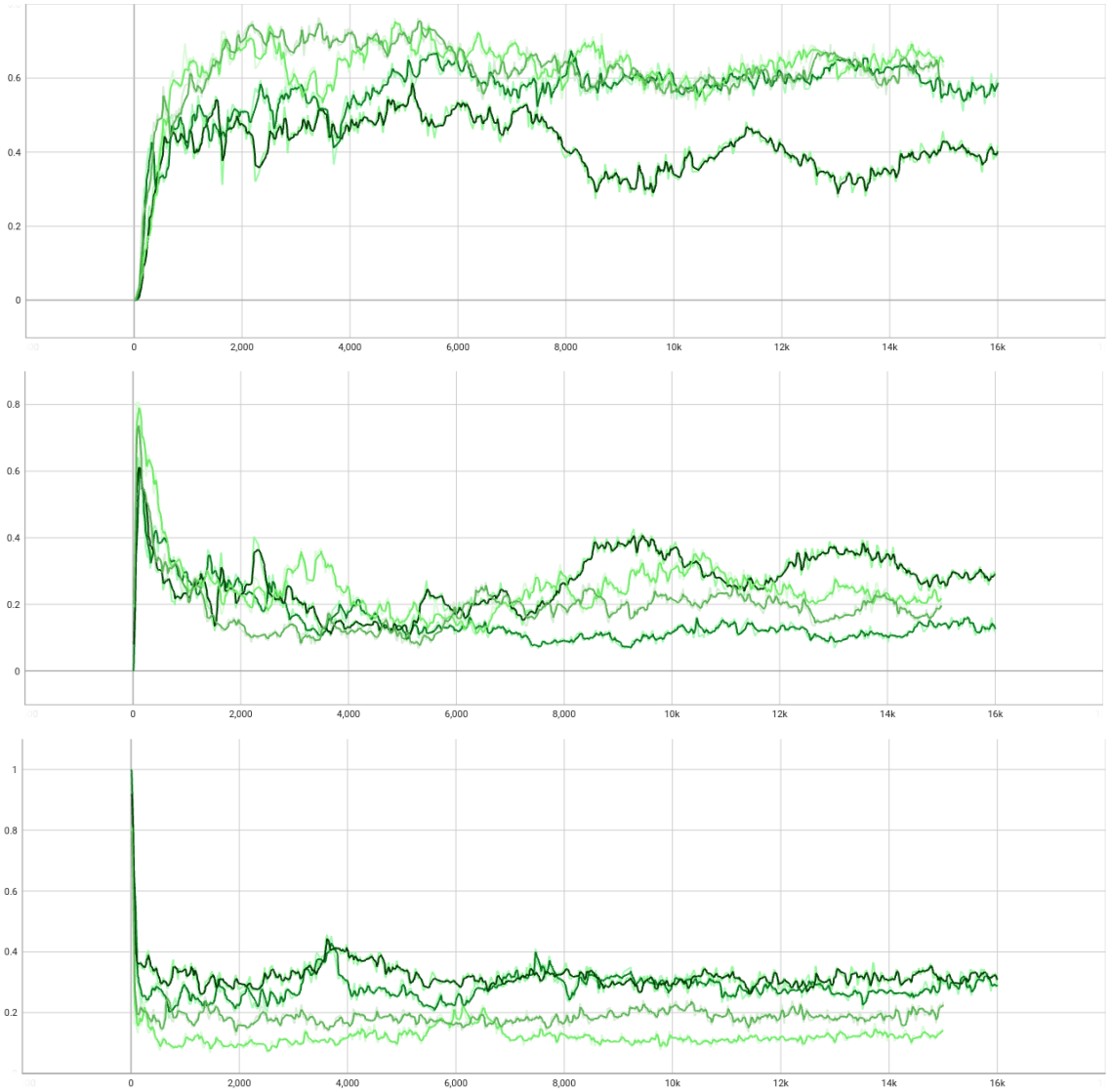


Figure 7.5: PPO-Baseline: Performance rates for three obstacle-density levels - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show E10, E20, E30, and E50 with darker green indicating higher obstacle density.

As expected, we see that the fewer obstacles we have, the less the agent crashes. One could expect this to be reflected in the rate of successful runs as well. However, a curious result is that PPO-Baseline for the three simplest environments seems to converge to approximately the same success rate.

Figure 7.6 shows the rewards for the baseline experiment. We see that the higher crash rates in the E30 environment have led to a lower total reward, while the E10 and E20 envir-

onment has little difference, even though E10 has better performance and crash statistics than E20.

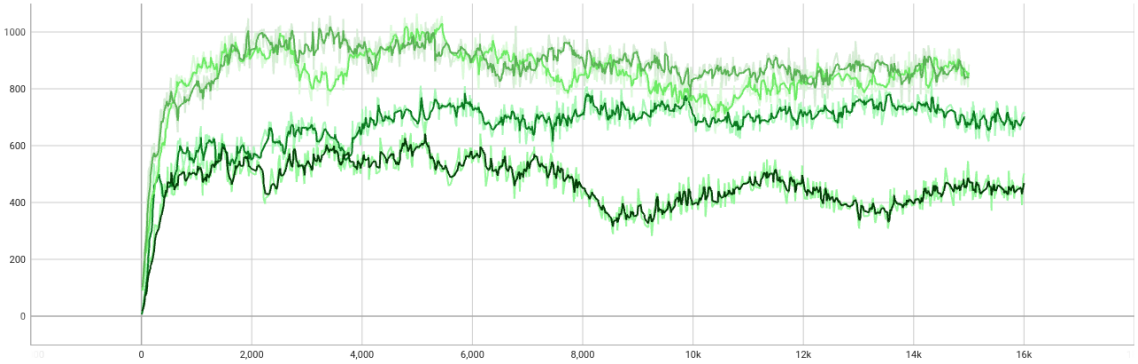


Figure 7.6: PPO-Baseline: Reward - The plot shows the rewards the agent receives over time for E10, E20, E30, and E50. As before, darker green indicates higher obstacle density.

7.2.2 Experiment 2 - Mixed Reward (PPO-Mix)

Figure 7.7 shows the success, timeout, and crash statistics of the second experimental setup from the four different obstacle-filled environments.

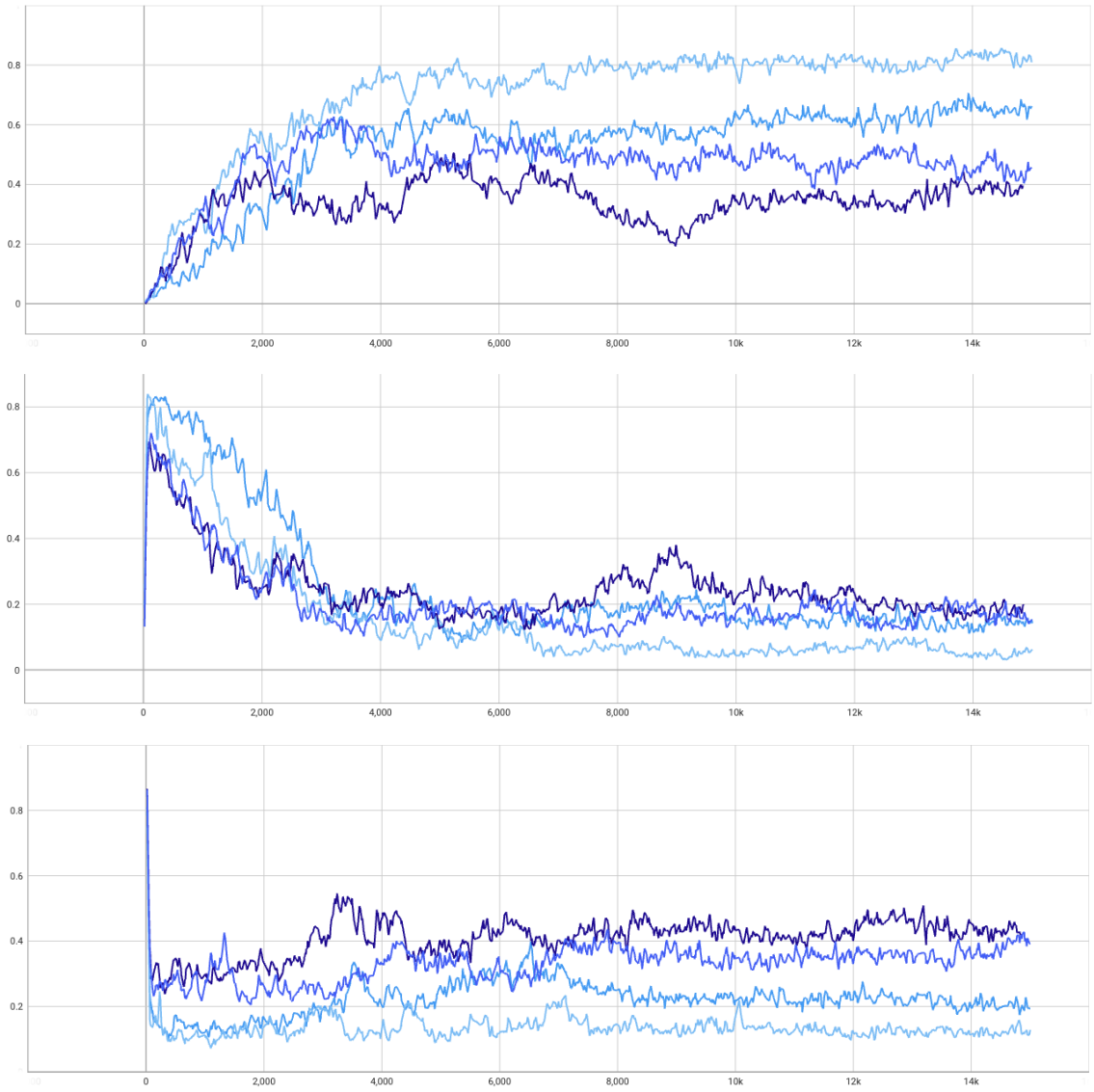


Figure 7.7: PPO-Mix: Performance rates for all four obstacle-density levels - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show E10, E20, E30, and E50, where darker blue indicates higher obstacle density.

The plots show a clear division in performance based on environmental complexity in all performance metrics. The three simplest environments show a steady increase in success rates which indicates good abilities to control policy updates in an advantageous direction at most training steps. For the highest complexity environment E50, the agent seems to struggle more with its policy updates.

Figure 7.8 shows the rewards achieved for PPO-Mix. As expected, the simplest envir-

onment's high success and low crash rates give the most reward. All environments show a quite similar initial learning rate. The E10 and E20 experiments show gradual increments throughout training, while in the more complex environments, we see more limited improvements over training.

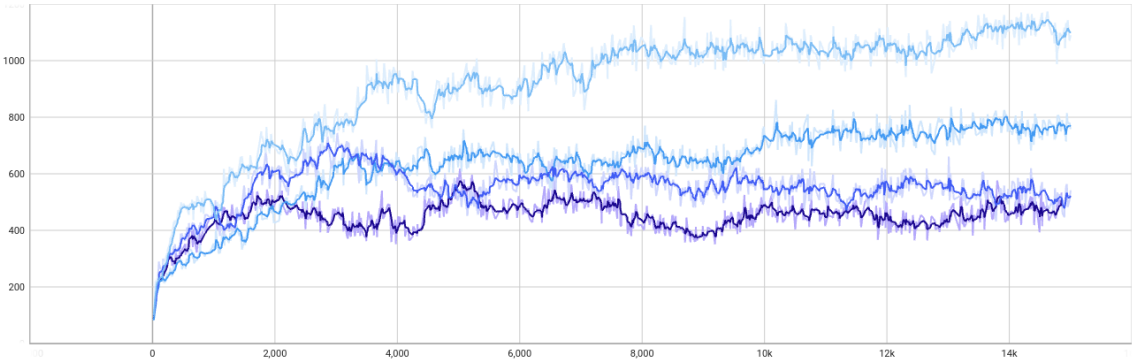


Figure 7.8: PPO-Mix: Reward - The plot shows the rewards the agent receives over time for E10, E20, E30, and E50. As before, darker blue indicates higher obstacle density.

7.2.3 Experiment 3 - Sparse Rewards (PPO-Sparse)

Figure 7.9 shows the success, timeout, and crash statistics of the third experimental setup from three different obstacle-filled environments.

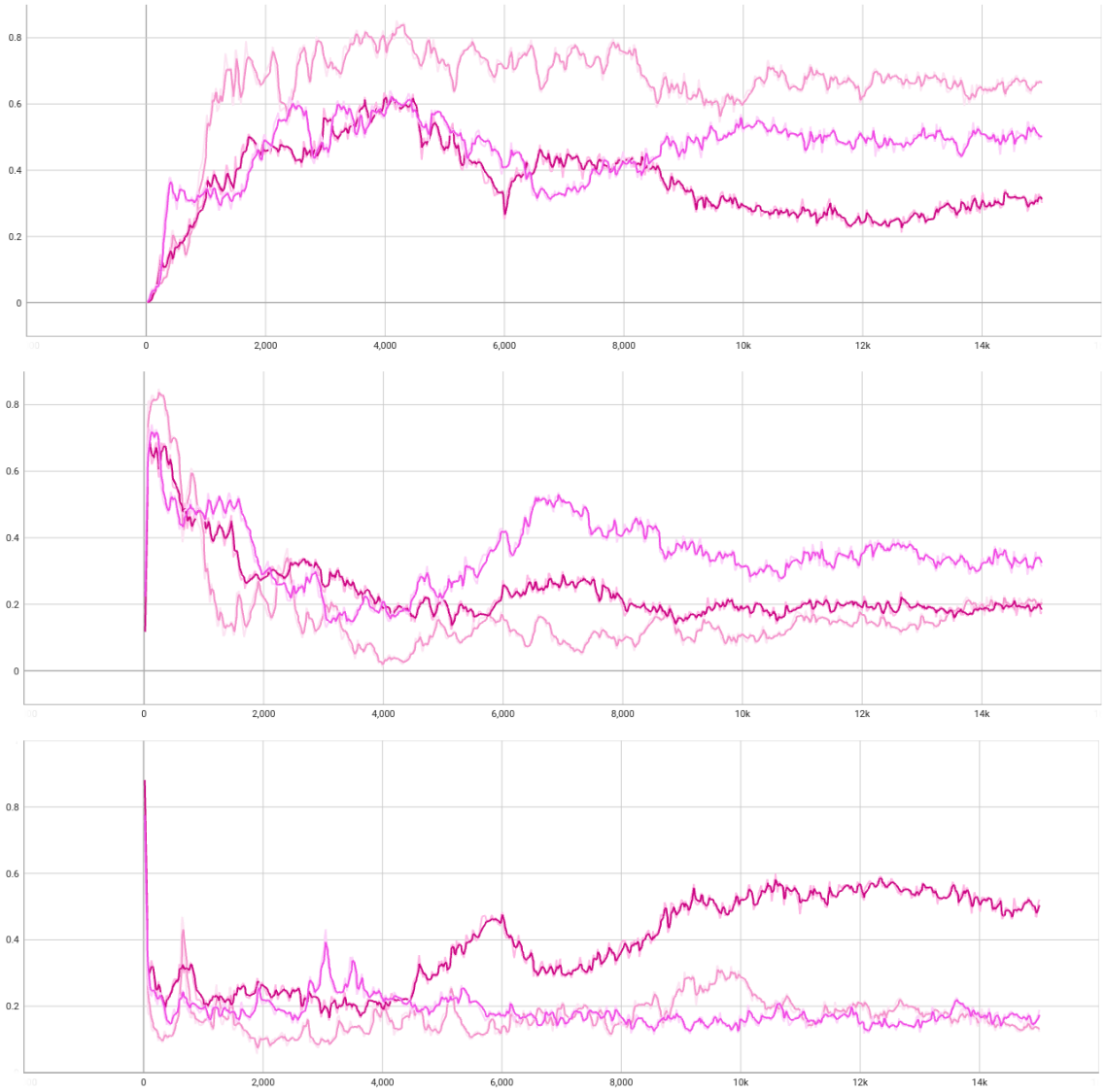


Figure 7.9: PPO-Sparse: Performance rates for E10, E20, and E30 - The topmost plot shows the success rates, the middle plot shows the timeout rates and the bottom plot shows the crash rates. The result of PPO-Baseline is shown in green, PPO-Mix is in blue, and PPO-Sparse is in pink. As before, a darker plot color indicates a higher obstacle density level.

From the plots, we see that all experiments find their optimal relatively early in training. This could be due to the increased KL threshold, leading to larger policy updates and faster learning. In simpler environments, the agent manages to stay at approximately the same optimum. In contrast, in the E30 environment, the policy drifts away from its initial optimum, never managing to find a better behavior. The larger policy updates could have led to instability in the learning process, making large policy updates that the agent is not

able to recover from.

7.2.4 Experiment 4 - Enhanced Exploration (PPO-Explore)

Figure 7.10 shows the success, timeout, and crash statistics of the fourth experimental setup from three different obstacle-filled environments; E10, E30, and E50.

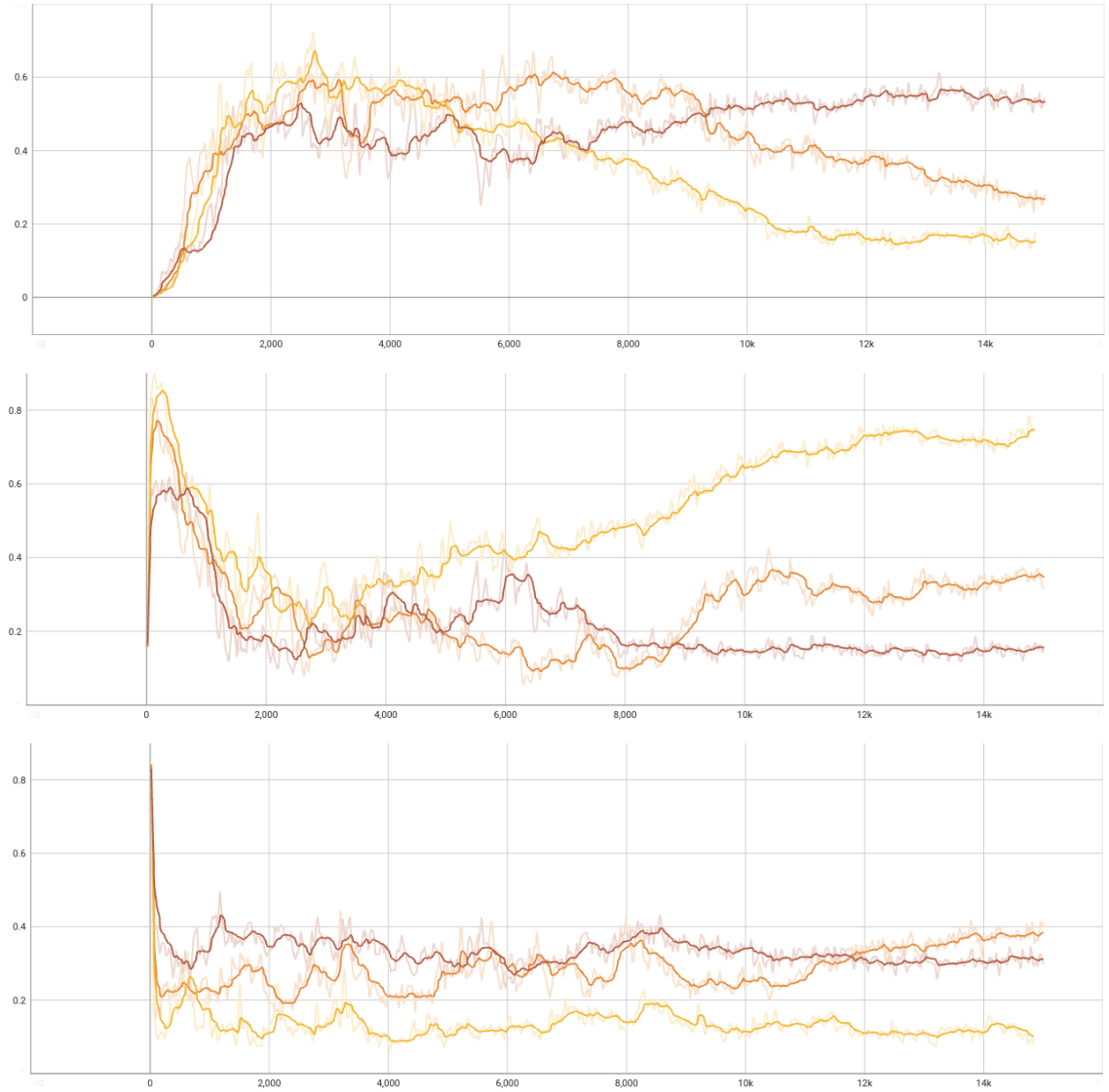


Figure 7.10: PPO-Explore: Performance rates for three obstacle-density levels - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. In all plots, E10 is shown in yellow, E30 in orange, and E50 in red.

As before, the crash percentage is higher the more cluttered the environment, but with enhanced exploration, we see that the maximum success rates achieved over the course of training are more or less the same for all environments. The overall trend shows earlier performance peaks for simpler environments, however also a larger constant drift away from the initial discovered local optimum.

Figure 7.11 show the rewards of PPO-Explore. As earlier, we see a clear correlation between success rate and rewards.

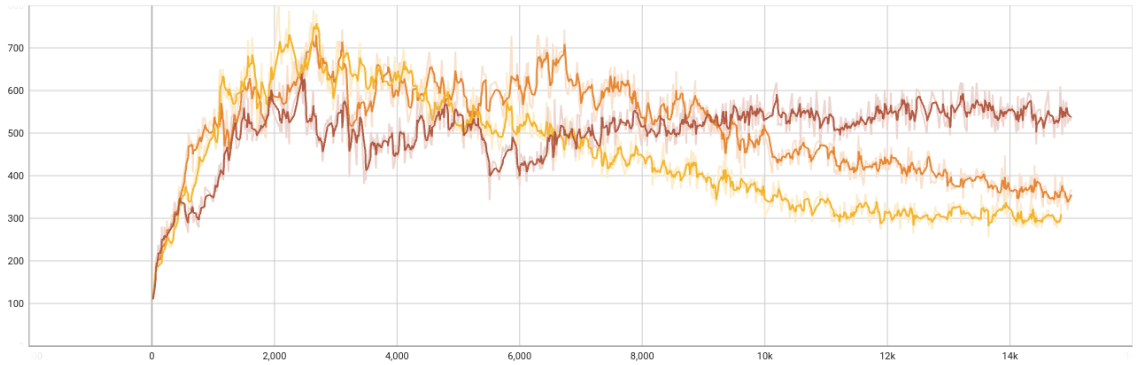


Figure 7.11: PPO-Explore: Rewards - The plot shows the rewards the agent receives over time for E10 (yellow), E30 (orange), and E50 (red).

Chapter 8

Discussion

This section summarizes and discusses the overall findings from chapter 7, identifies possible bottlenecks in performance, and suggests improvements for future work.

8.1 Free Environment

The experiments performed in the free environment show that the reward function design significantly affects PPO's learning efficiency. Both the exponential and the inverse square reward functions showed fast learning and exponentially fast target convergence in close to all attempts. However, the inverse square achieved the best performance with a success of 98.8% and slightly better target convergence. The quadratic reward also showed convergence abilities but was a more unreliable choice. The quadratic shape is difficult to design to provide a strong enough inclination and gradients around the target without yielding too much inclination and negative rewards further away from the target.

Overall, the simplicity of the task only seems to demand a decaying and differentiable function to ensure learning. However, a strong inclination around the target value was necessary to guarantee accurate target convergence. This makes the inverse square and exponential decay functions good reward function design choices.

8.2 Obstacle-Filled Environment

8.2.1 Comparing Performance for each Environmental Complexity

E10

Figure 8.1 shows the performance of PPO-Baseline, PPO-Mix, PPO-Explore, and PPO-Sparse in the environment with 10 obstacles. From the plots, we see that both PPO-Mix and PPO-Sparse outperform the baseline. PPO-Mix achieves the best and most stable performance in the simplest environment. Yet, PPO-Sparse achieves approximately the same maximum success and minimum crash rates at a much earlier training iteration. The increased entropy of PPO-Explore seems to find an initially promising policy. Still, it ends up diverging, probably due to too much uncertainty in the action sampling resulting in poor exploitation of the accumulated knowledge.

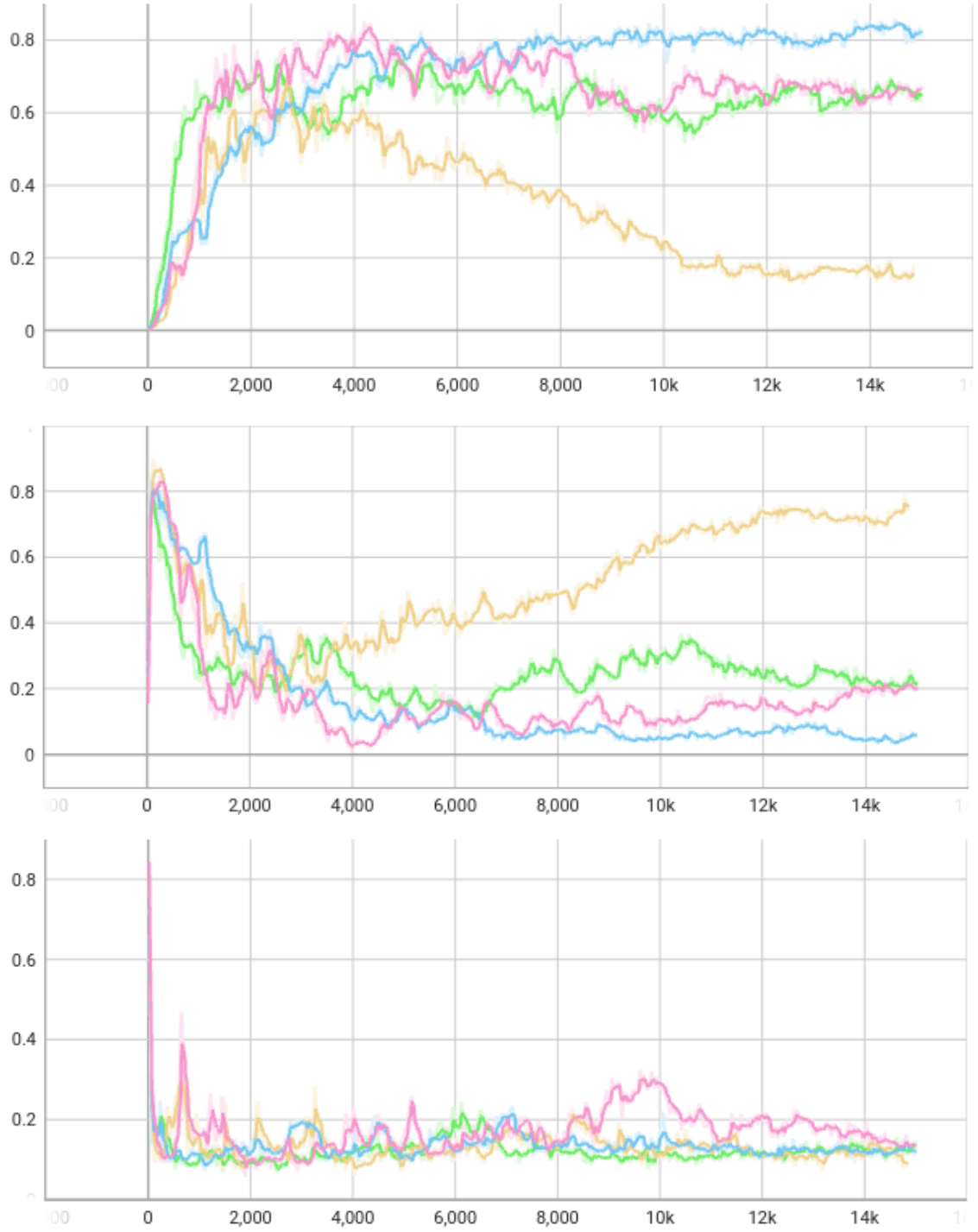


Figure 8.1: E10: performance rates for all experiments - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show PPO-Baseline in green, PPO-Mix in blue, PPO-Explore in orange, and PPO-Sparse in pink.

The maximum success and minimum crash statistics for environment E10 are summarized in table 8.1.

	Min crash %	Max success %
PPO-Baseline	7 (2520)	75 (2499)
PPO-Mix	7 (1365)	86 (14497)
PPO-Explore	7 (4041)	72 (2709)
PPO-Sparse	6 (1900)	85 (4193)

Table 8.1: Training results in E10 - training iteration indicated in parenthesis.

E20

Figure 8.2 shows the performance of PPO-Baseline, PPO-Mix, and PPO-Sparse in the environment with 20 obstacles. From the plots, we see that none of the other reward function designs outperforms the baseline. PPO-Mix yields better performance than PPO-Sparse and converges as it did in the simplest environment. In this experiment, PPO-Sparse shows some tendency to drift away from the current optimal policy during training. However, the policy updates get it back on track eventually.

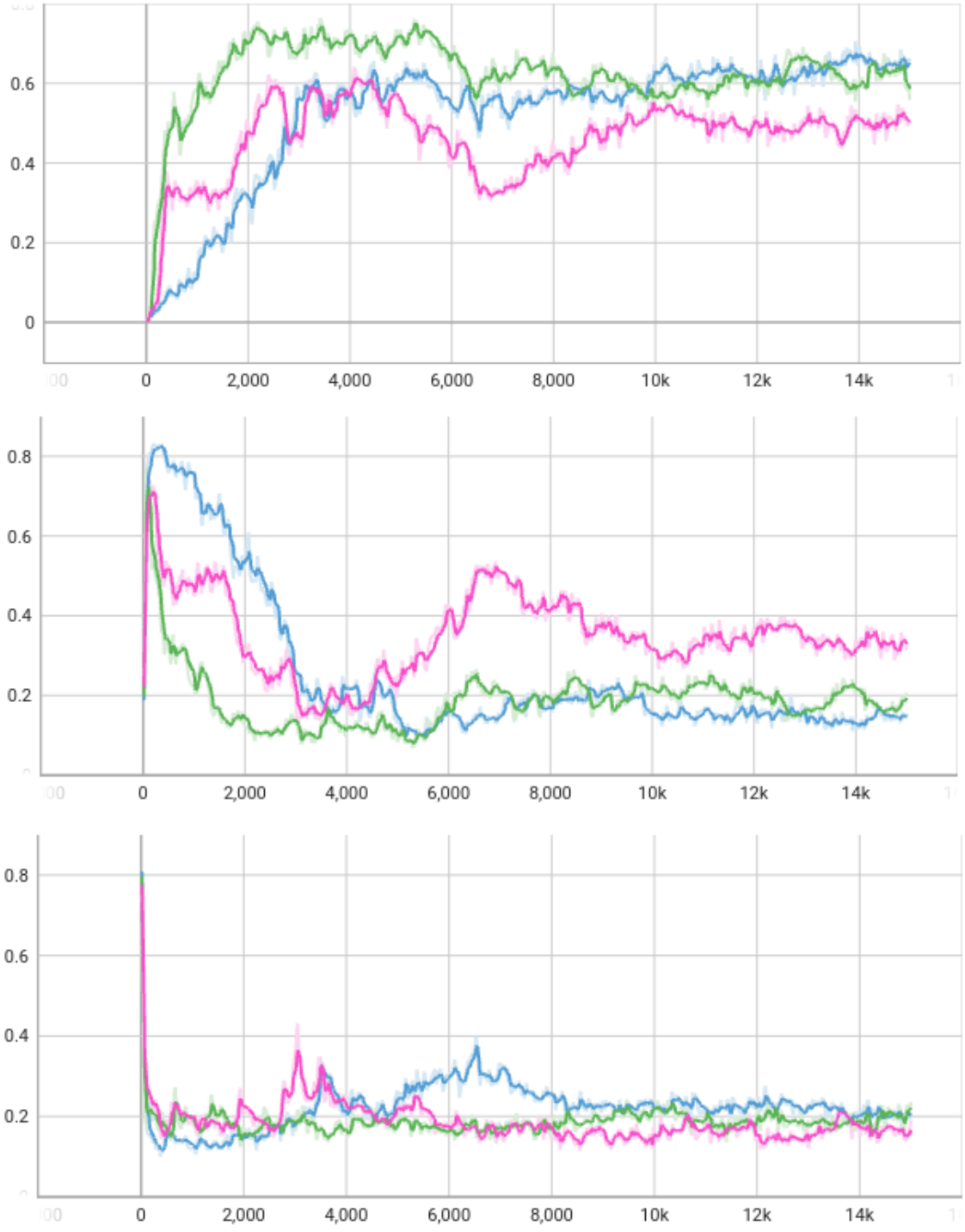


Figure 8.2: E20: performance rates for all experiments - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show PPO-Baseline in green, PPO-Mix in blue, and PPO-Sparse in pink.

The maximum success and minimum crash statistics for environment E20 are summarized in table 8.2.

	Min crash %	Max success %
PPO-Baseline	12 (1937)	76 (3419)
PPO-Mix	11 (367)	71 (13931)
PPO-Sparse	11 (8549)	64 (4085)

Table 8.2: Training results in E20 - training iteration indicated in parenthesis.

E30

Figure 8.3 shows the performance of PPO-Baseline, PPO-Mix, PPO-Explore, and PPO-Sparse in the environment with 30 obstacles. As for the E20 environment, none of the other reward function designs manages to beat the baseline in the rate of successful attempts. PPO-Explore and PPO-Sparse manage somewhat lower crash rates, however. As for the E10 and E20 experiments, PPO-Sparse reaches an early peak in performance at around training iteration 4000. Thereafter, it does not manage to stay at this optimum. As mentioned in section 7, this could be due to the increased KL threshold. This could make it more favorable to exploration in the initial training stages, resulting in a higher likelihood of actions moving the agent away from the current policy and optimum.

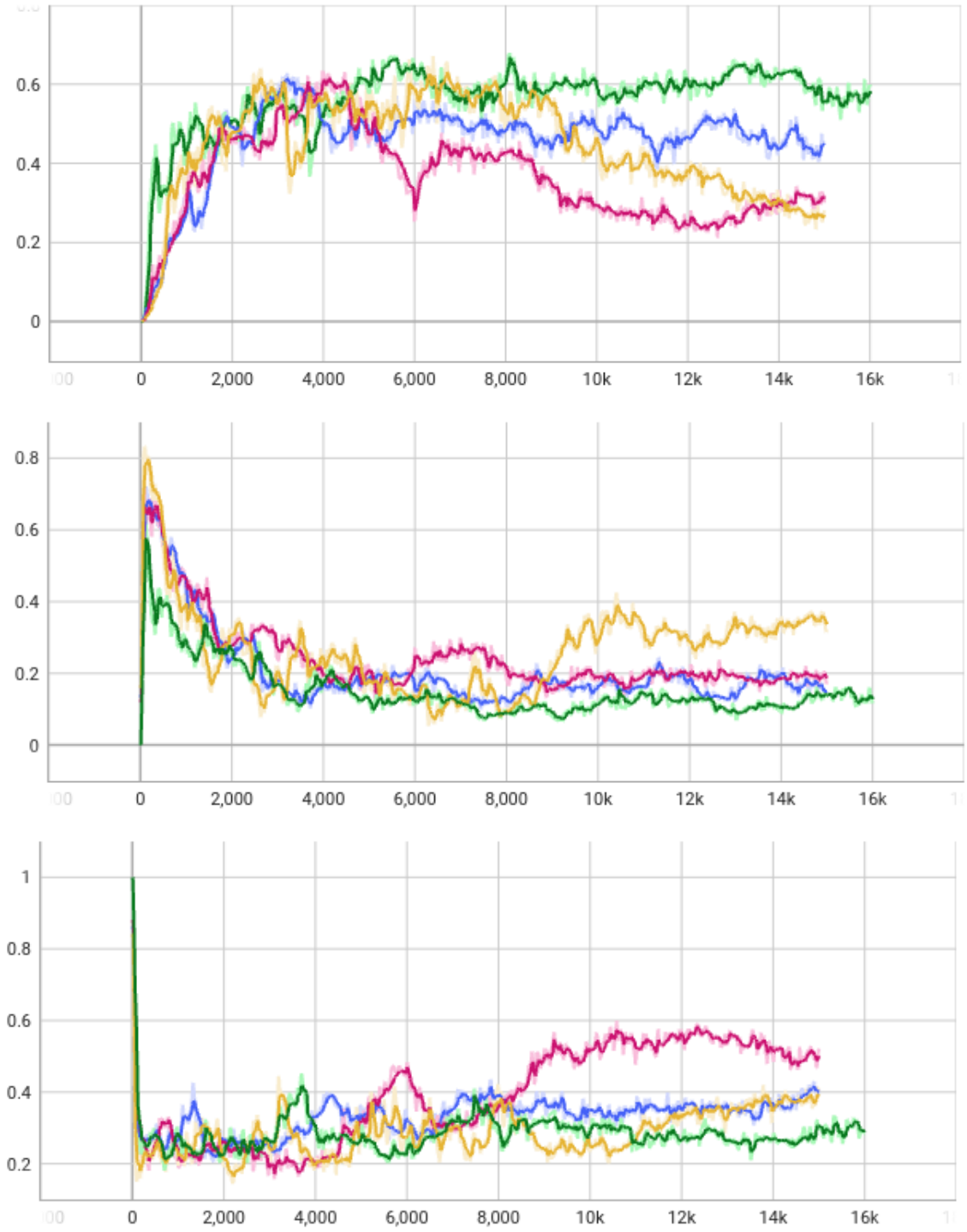


Figure 8.3: E30: performance rates for all experiments - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show PPO-Baseline in green, PPO-Mix in blue, PPO-Explore in orange, and PPO-Sparse in pink.

The maximum success and minimum crash statistics for environment E30 are summarized in table 8.3.

	Min crash %	Max success %
PPO-Baseline	18 (699)	68 (8079)
PPO-Mix	21 (1776)	62 (3174)
PPO-Explore	14 (2195)	67 (6407)
PPO-Sparse	16 (3104)	63 (3982)

Table 8.3: Training results in E30 - training iteration indicated in parenthesis.

E50

Figure 8.4 shows the performance of PPO-Baseline, PPO-Mix, and PPO-Explore in the environment with 50 obstacles. The results show that PPO-Explore achieves the best performance in the most complex environment, only just outperforming the baseline. Over the training run, PPO-Explore shows better convergence properties.

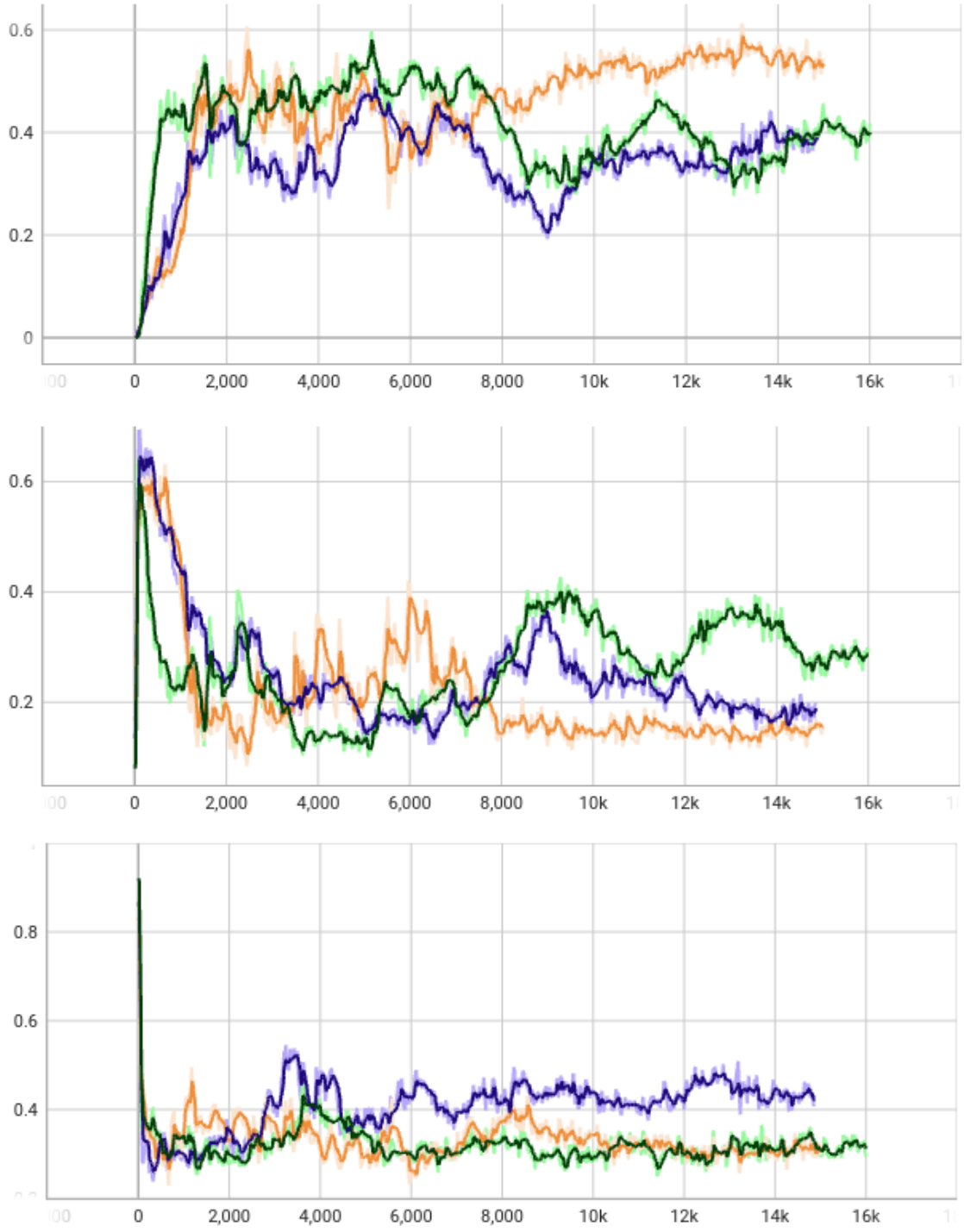


Figure 8.4: E50: performance rates for all experiments - The topmost plot shows the success rates, the middle plot shows the timeout rates, and the bottom plot shows the crash rates. The plots show PPO-Baseline in green, PPO-Mix in blue, and PPO-Explore in orange.

The maximum success and minimum crash statistics for environment E50 are summarized in table 8.4.

	Min crash %	Max success %
PPO-Baseline	25 (11410)	59 (5141)
PPO-Mix	24 (320)	50 (5226)
PPO-Explore	23 (5965)	61 (13211)

Table 8.4: Training results in E50 - training iteration indicated in parenthesis.

8.2.2 Success Vs. Crash

The maximum success and minimum crash statistics for each experiment in each environment are summarized in table 8.1, 8.2, 8.3, and 8.4. The results show different reward functions perform better or worse depending on the complexity of the environment. PPO-mix was best for E10 with 86% success vs. 11% crash. PPO-baseline performed best in E20 and E30 with 76% and 68% success and 14% and 23% crash, respectively. In the highest complexity environment, E50, PPO-explore was the best with 61% success and 26% crash. If there exists a reward function that is generalizable to all environmental complexities, it was not found in this experiment.

As expected, we see a clear overall correlation between obstacle density level and minimum crash percentage. All reward functions show more or less the same abilities when it comes to collision avoidance for a given complexity level. Overall, there seems to be a hard limit to how small the collision rate can become for the current setup, which highly depends on the obstacle density. They all converge quite fast to small crash rates, indicating that obstacle avoidance is the motivation for learning initially. This is reasonable due to sparse rewards far away from the target and high penalties for crashing.

8.2.3 Training Losses

Now, we take a brief look at each of the training losses for the experimental results chapter 7 and discuss whether the results are as expected or if not, provide theories on why.

Critic loss

The overall results from the experiments show convergence of the critic loss, shown for all experiments in figure 8.5. This suggests that the value function approximated by the

critic network has successfully learned to estimate the expected future rewards or returns associated with different states or state-action pairs.

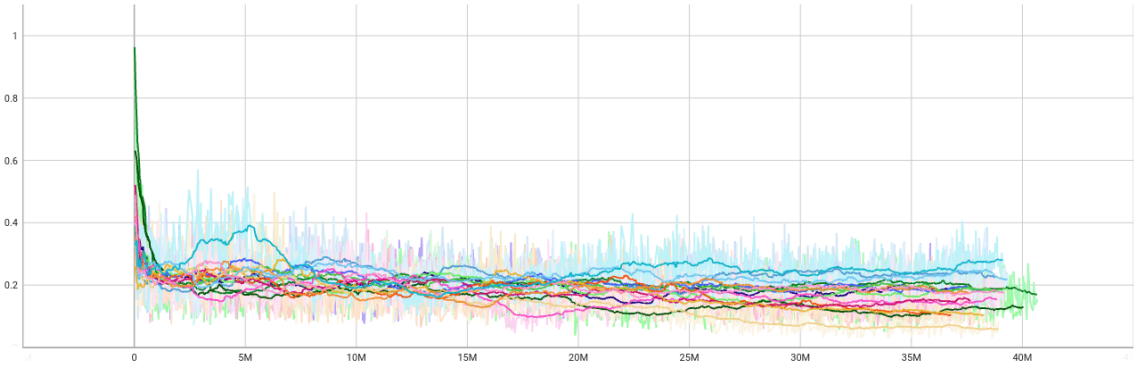


Figure 8.5: Critic loss in all experiments - smoothed for easier comparison.

Actor loss

The results show large oscillations in the actor loss, as shown in figure 8.6, which could indicate some instability in the training process. This instability could have several causes, for instance, high learning rates, improper network weight initialization, or an improper exploration-exploitation trade-off. As the learning rate is adaptive, and network weights are initialized randomly such that the probability of this causing the oscillations in all experiments is quite low, the best guess is that the algorithm could be struggling to find the right exploration-exploitation balance. This could make the policy network alternate between exploration and exploitation phases, causing the actor loss to oscillate.

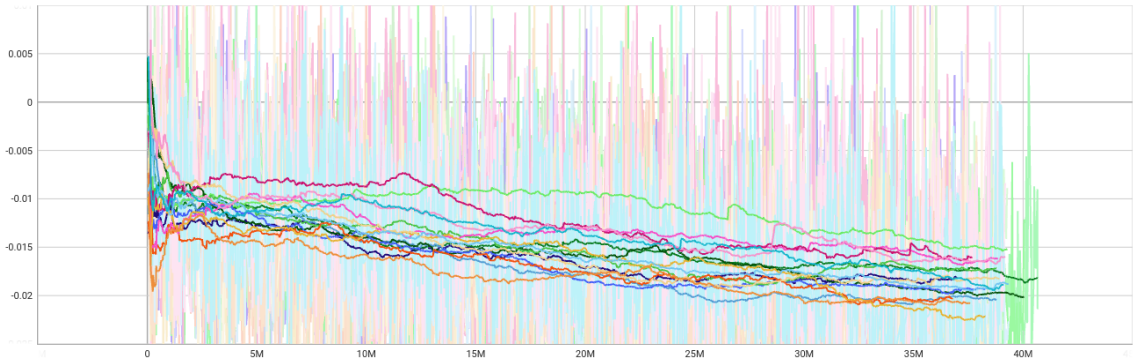


Figure 8.6: Actor loss in all experiments - smoothed for easier comparison.

Entropy loss

The entropy loss decays steadily, slowly reducing the tendency to explore and thus increasing the exploitation of almost all experiments. The exception is PPO-Explore, which has an initial increase in entropy loss before it somewhat decays over the training iterations. The fact that it never reaches zero or negative values indicates that the learned policy stays quite stochastic and does not exploit its environment to the same extent as the policies learned by the other PPO variants. Thus, a smaller yet non-zero entropy coefficient could be key to accomplishing the right exploration-exploitation trade-off in complex environments.

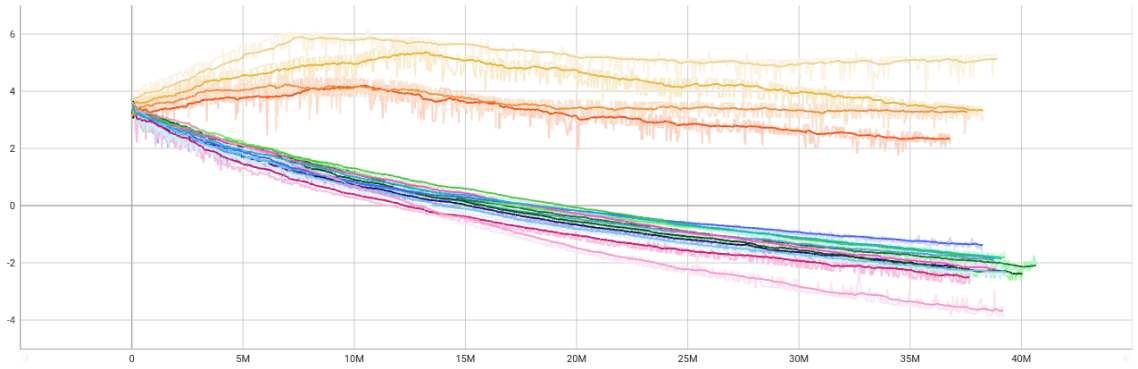


Figure 8.7: Entropy loss in all experiments - smoothed for easier comparison.

Bounds loss

In figure 8.8, we plot the bounds losses for each environmental complexity. The bounds losses are expected to rise initially as the agent explores its environment. We see this is the case for all experiments. After a while, they should decrease - indicating that the agent has become more proficient in staying within the desired bounds. Ideally, the bounds loss should converge to a low value, indicating that the agent consistently operates within or close to the specified bounds.

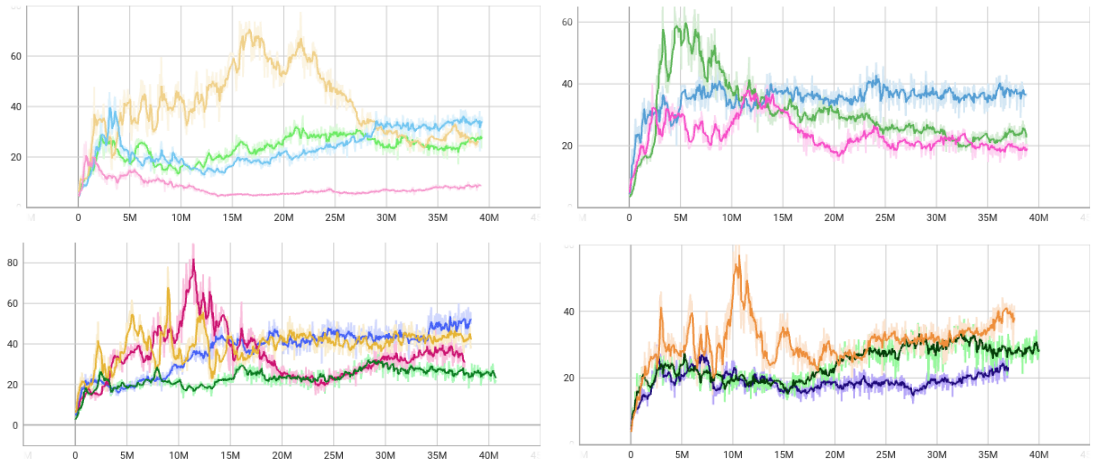


Figure 8.8: Bounds losses for four obstacle-density levels - (1) The top left plot shows E10, (2) the top right plot shows E20, (3) the bottom left plot shows E30, and (4) the bottom right plot shows E50. The plots show PPO-Baseline in green, PPO-Mix in blue, PPO-Explore in orange, and PPO-Sparse in pink.

The bounds losses are a bit curious as they keep rising, which could mean the size of the policy updates increases over training. A higher bounds loss could mean we allow for more exploration and can help avoid getting trapped in local optima, but it may also introduce more instability.

8.2.4 Behavior Analysis

Now we take a look at some quadrotor trajectories in the simulation environment to investigate quadrotor behavior. The figures will show a free-world optimal trajectory, a green line from starting position to the target ignoring obstacles, and the quadrotor trajectory in a color range from yellow, indicating high speed, to red, indicating low speed.

Starting with some good trajectories, figure 8.9 shows some trajectories close to optimal trajectories.

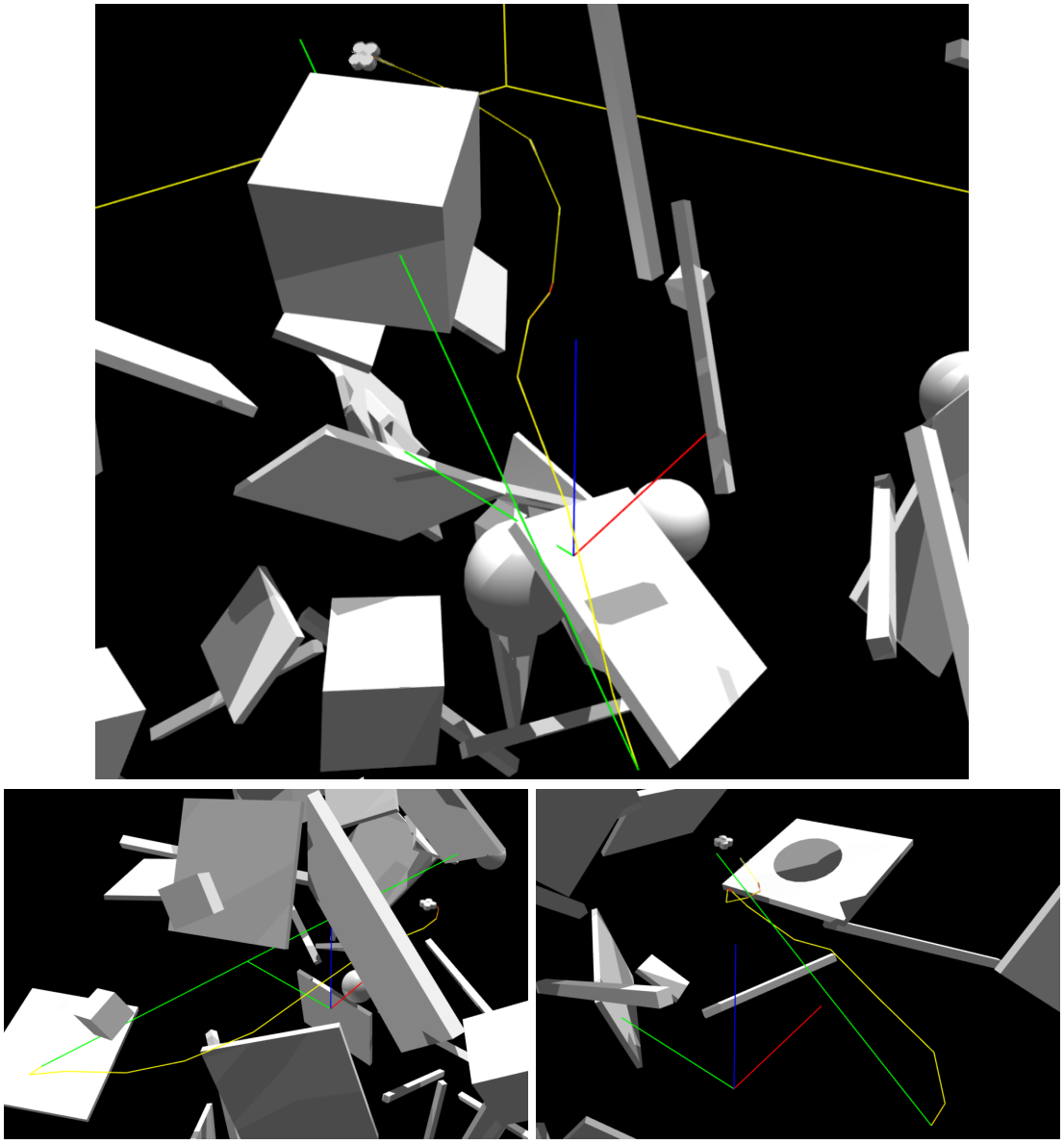


Figure 8.9: Trajectories showing close to optimal behavior - taken from training in E50.

The trajectories show that the learned policy has the ability to produce reasonable trajectories to reach a target 3D position in the highest complexity environment. However, far from all trajectories was this successful. The agent tends to be passive if the target path is mostly downward. In these situations, it spins slowly around its own up axis and tends to drift upwards and away from its target, as seen in figure 8.10. This could partly be explained by the field of view constraining the agent to only move in the directions visible through its simulator camera, making it impossible to go directly down. The direction re-

ward was meant to compensate a bit for this as it encourages the agents to point its camera toward the target. However, there is a limit to how much pitch the quadrotor is allowed to have to respect known quadrotor constraints.

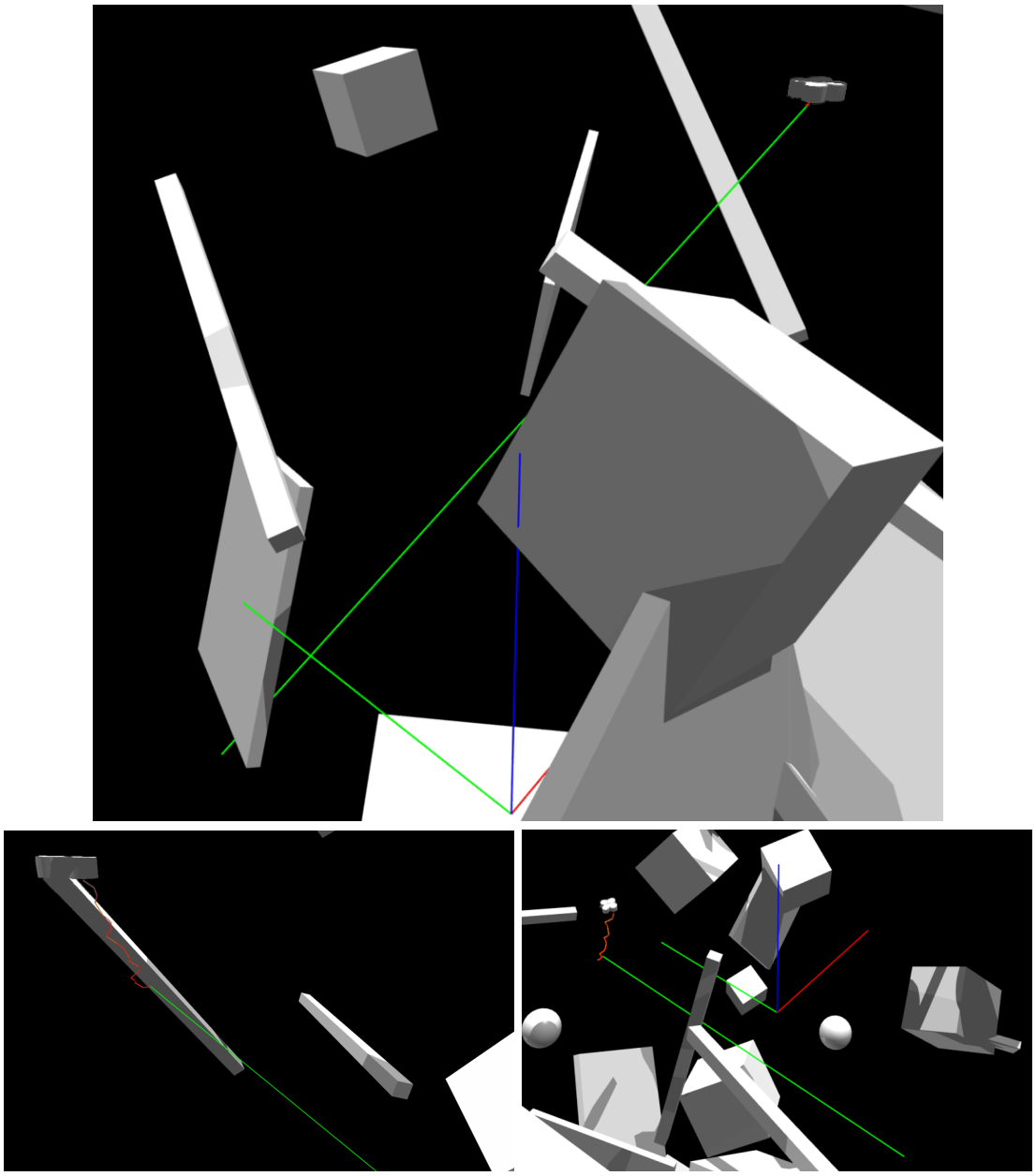


Figure 8.10: Trajectories showing passive and spinning behavior - taken from training in E50.

8.2.5 Exploration

PPO-Clip is known to mainly explore promising regions of the policy space [3]. In environments with more obstacles, these could be more difficult to find; thus, enhancing exploration can increase the likelihood of finding good trajectories. We saw that increasing the entropy coefficient (PPO-Explore), and thus the policy's tendency to explore, resulted in more promising results in the higher complexity environments. However, in simpler environments, increasing the entropy coefficient of PPO will likely result in suboptimal behaviors as it will make the policy less certain about the optimal action to take in a given state. This was also seen in the experiments, as PPO-Explore performed comparatively worse in the low-complexity environments.

8.2.6 Possible Bottlenecks

Reward function design

The main focus of this thesis has been position reward shaping. This has resulted in little attention on other aspects of reward shaping. As suggested in [1], one could add negative rewards in the vicinity of obstacles, penalizing the agent for getting close to them. This could work as a repulsive force, encouraging faster learning and safer navigation. In our model-free task, where obstacles are randomized in each environment for each episode, this could have been implemented as a close-depth penalty in the reward function.

That all crashes produce the same negative reward, without separating between an immediate crash and a crash occurring just before reaching the target could have led to inefficient training and poor policy updates. The agent will not learn anything from the initially good actions taken in the late-crash trajectory and actually lower the probabilities of these actions in its update. The chosen collision definition could therefore be a factor slowing down learning. From the various experiments, we see a fast decrease in the collision rates compared to timeouts indicating the agents highly prioritize avoiding collisions. This is especially a trend at the beginning of training, and the learned passivity could serve as a desolate local optimum which PPO could struggle to exit from, slowing down or hindering the learning process.

A proposed improvement is adding the crash penalty term to the total reward such that the policy learning could benefit from all examples. We will advise setting the crash penalty slightly higher than done in these experiments to ensure the total reward for a collision is still negative no matter where the crash happens. Adding a timestep parameter to the state vector and penalizing the agent based on higher timestep values could encourage faster

flight. However, based on the behavioral analysis, this is seemingly not the main cause of timeouts or slow flying speeds in the conducted experiments.

According to Zhu et al. [1], manually designed reward functions have a tendency to overfit the specific task and, therefore, lack universality. Secondly, they state that an unsuitable reward could lead to improper learning guidance. Furthermore, "designing a robust and suitable reward function greatly depends on the designer's experience and intuition" [36]. A possible solution to these problems is automatic reward-shaping techniques. Chiang et al. [37] proposed the AutoRL algorithm, which searches for the best reward function and the neural network architecture. Their findings show AutoRL to generalize well to new environments and to be a helpful tool in complex navigation tasks. Another option presented by Zhang et al. [36] employs a general reward function based on a matching network (MN). The MN-based reward function gains experience by pre-training on trajectories from similar navigation tasks, giving higher training speed in new tasks.

Networks

The PPO actor-critic network design choice used in the DRL-based controllers could have limited the performance or generalization abilities of PPO in the navigation task. Further work should investigate if the selected network designs are capable of capturing the complex relationships and dependencies present in the quadrotor navigation task. This could be done by experimenting with different network sizes (layer widths and number of layers), activations, and network types.

Hyperparameters

The hyperparameters of PPO were chosen based on recommended values without much consideration of the specific task. Not specifically tuning these to fit the navigation task could have led to a limitation in achievable performance. PPO should be easy to manually tune in the sense that one knows exactly what each parameter represents. However, after a few failed attempts that yielded poorer results than the recommendation found in the literature, this was found to be too time demanding and considered out of scope for this thesis. A suggestion to improve the recommended or manually tuned parameters is to initially run a method for learning the optimal hyperparameters and use this as a starting point.

Effect of random initialization and only single experiments

The choice of initialization plays a crucial role in determining the quality of solutions obtained in Proximal Policy Optimization (PPO). If the initial parameters are unfortunate, the algorithm may struggle to find a good local optimum that is close to the global one, especially in complex environments with many obstacles. In our experiments, we were constrained by limited time and computational resources, so we could only run the experiments once, starting from randomly initialized network weights. This single initialization introduces a source of experimental error and raises questions about the validity of the presented results and comparisons. Depending on the initialization, we could observe better or worse performance, making our measurements potentially non-representative of the true performance across different difficulty levels.

Learning can be slowed down in simple environments with sparse obstacles because the agent encounters obstacles less frequently during its trajectory toward the target. To illustrate this, Figure 8.11 compares two different initialization seeds in a simple E5 environment with only five obstacles.

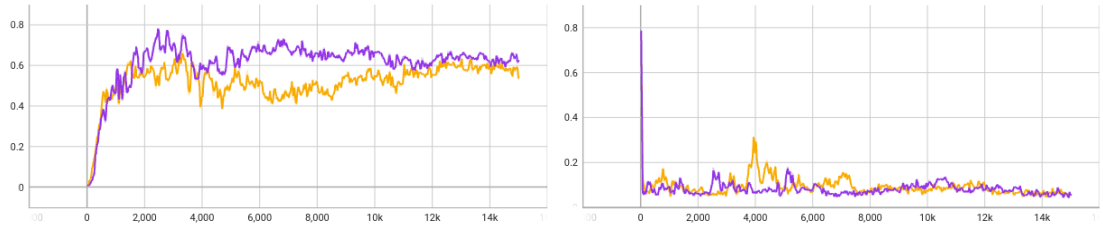


Figure 8.11: Experimenting with two different initialization seeds - the plots show success and crash rates for two training runs in a simple E5 environment.

By examining the plots, we observe a 10% difference in maximum success between the two seeds. This performance discrepancy suggests that the training process is sensitive to randomnesses, such as the random initialization of network parameters or the randomized placement of obstacles in environments with sparse obstacles. Enhanced exploration might have helped overcome the randomness issue in the higher-complexity environments by encouraging the agent to explore more of the state-action space.

The conducted experiments suggest that the degree of exploration should be adjusted according to the environment's complexity. To further investigate this, I propose running the experiments multiple times with different initialization seeds and comparing experiments with different levels of exploration using the same seed.

Ideally, it would have been beneficial to repeat the experiments multiple times with dif-

ferent seeds and average the results. Alternatively, using the same seed in each experiment would ensure a more accurate comparison but might limit the algorithm's performance potential due to a suboptimal initialization choice. Another approach could involve implementing a method for optimizing the initial network parameters to ease the sensitivity to initialization.

8.2.7 Training Times

Training a quadrotor to perform a complex task requires more computational resources and is significantly more time-demanding than learning a simple task. The free environment agents train in approximately half an hour (converging long before), while the obstacle-filled environments take around 16.5 hours to finish training. The free environment's smaller state space and general simplicity allow us to train 128 times as many agents in parallel compared to the obstacle environment.

Chapter 9

Conclusion

This master thesis explored the design and evaluation of reward functions for quadrotor navigation in complex environments using the Proximal Policy Optimization (PPO) algorithm. The experiments revealed that a reward function combining different reward terms, resulting in an overall denser reward function with a stronger inclination near the target and a smaller inclination further from the target, could lead to improved convergence properties of the policy in PPO. However, the level of environmental complexity played a crucial role, emphasizing the importance of carefully tuning hyperparameters to achieve optimal performance. Interestingly, increasing exploration proved beneficial in the most cluttered environments, leading to the discovery of more promising policies. However, in simpler environments, increased exploration resulted in poor policy updates.

It is important to acknowledge the limitations of this study, primarily the reliance on a single test run in each environment per experiment. This limited approach hinders confident conclusions as PPO could become stuck in local optima, making the obtained results dependent on factors like initialization. Running multiple experiments would have provided higher confidence and a more robust comparative evaluation. However, due to computational resource limitations and time constraints, running a larger number of experiments was not feasible.

The results highlight the dependency of the quadrotor's navigation abilities on the complexity of the environment. In obstacle-filled environments, a higher complexity leads to a higher occurrence of crashes, with minimal variation observed across various reward designs. The findings also suggest the existence of limitations in handling downward-facing trajectories, contributing to a significant number of timeouts. This challenge requires further investigation in future work.

While the position reward design showed promising results in facilitating learning,

there appears to be a bottleneck in the control system that limits the performance of PPO for the navigation task. Potential areas for improvement include addressing collisions and timeouts more effectively, refining hyperparameter choices, and optimizing network designs.

In conclusion, this thesis contributes to the understanding of reward function design and its impact on the performance of PPO in quadrotor navigation. The findings emphasize the need for tailored reward functions, exploration strategies, and hyperparameter settings based on the complexity of the environment. Future work should focus on running multiple experiments, investigating the handling of downward trajectories, and optimizing other control system components further to enhance the performance of PPO for the navigation task.

Bibliography

[1] K. Zhu and T. Zhang, 'Deep reinforcement learning based mobile robot navigation: A review,' *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, 2021. DOI: 10.26599/TST.2021.9010012. [Online]. Available: https://www.sciopen.com/article/10.26599/TST.2021.9010012.

[2] Y. Li, *Deep reinforcement learning*, 2018. DOI: 10.48550/ARXIV.1810.06339. [Online]. Available: https://arxiv.org/abs/1810.06339.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, *Proximal policy optimization algorithms*, 2017. DOI: 10.48550/ARXIV.1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347.

[4] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, 'Deep reinforcement learning: A brief survey,' *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017. DOI: 10.1109/msp.2017.2743240. [Online]. Available: https://doi.org/10.1109%5C%2Fmsp.2017.2743240.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction (2nd ed.)* The MIT Press, 2018.

[6] J. Choi, K. Park, M. Kim and S. Seok, 'Deep reinforcement learning of navigation in a complex and crowded environment with a limited field of view,' in *2019 International Conference on Robotics and Automation (ICRA)*, Montreal, QC, Canada: IEEE Press, 2019, pp. 5993–6000. DOI: 10.1109/ICRA.2019.8793979. [Online]. Available: https://doi.org/10.1109/ICRA.2019.8793979.

[7] L. Busoniu, D. Ernst, B. De Schutter and R. Babuska, 'Approximate reinforcement learning: An overview,' *Proceedings 2011 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-11)*, Oct. 2013. DOI: 10.1109/ADPRL.2011.5967353.

[8] C. R. d.-B. Antonio Loquercio Ana I. Maqueda and D. Scaramuzza, *Dronet: Learning to fly by driving*, 2018. DOI: IEEERoboticsandAutomationLetters(RA-L). [Online]. Available: https://rpg.ifi.uzh.ch/docs/RAL18_Loquercio.pdf.

[9] J. Achiam, *Spinning up documentation*, 2020. [Online]. Available: https://spinningup.openai.com/_/downloads/en/latest/pdf/.

[10] O. Nachum, M. Norouzi, K. Xu and D. Schuurmans, *Trust-pcl: An off-policy trust region method for continuous control*, 2018. arXiv: 1707.01891 [cs.AI].

[11] R. Nian, J. Liu and B. Huang, 'A review on reinforcement learning: Introduction and applications in industrial process control,' *Computers Chemical Engineering*, vol. 139, p. 106 886, 2020, ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2020.106886. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098135420300557.

[12] J. Schulman, S. Levine, P. Moritz, M. I. Jordan and P. Abbeel, *Trust region policy optimization*, 2015. DOI: 10.48550/ARXIV.1502.05477. [Online]. Available: https://arxiv.org/abs/1502.05477.

[13] T. P. Lillicrap *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509.02971 [cs.LG].

[14] R. S. Sutton, D. McAllester, S. Singh and Y. Mansour, 'Policy gradient methods for reinforcement learning with function approximation,' in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen and K. Müller, Eds., vol. 12, MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

[15] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning,' *Nature*, vol. 518, pp. 529–33, Feb. 2015. DOI: 10.1038/nature14236.

[16] I. J. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[17] J. Achiam, *Advanced policy gradient methods*, http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf, Accessed: 2023–04-06, 2017.

[18] G. Brockman *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[19] J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, 2018. arXiv: 1506.02438 [cs.LG].

[20] T. P. Lillicrap *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509.02971 [cs.LG].

[21] D. Hoeller, L. Wellhausen, F. Farshidian and M. Hutter, *Learning a state representation and navigation in cluttered and dynamic environments*, 2021. arXiv: 2103.04351 [cs.RO].

[22] Y. Song, M. Steinweg, E. Kaufmann and D. Scaramuzza, *Autonomous drone racing with deep reinforcement learning*, 2021. arXiv: 2103.08624 [cs.RO].

[23] P. Nitschke, *Reinforcement learning for fast, map-free navigation in cluttered environments using aerial robots*, 2022.

[24] N. Rudin, D. Hoeller, P. Reist and M. Hutter, *Learning to walk in minutes using massively parallel deep reinforcement learning*, 2022. arXiv: 2109.11978 [cs.RO].

[25] V. Makoviychuk *et al.*, *Isaac gym: High performance gpu-based physics simulation for robot learning*, 2021. arXiv: 2108.10470 [cs.RO].

[26] F. A. Goodarzi, D. Lee and T. Lee, 'Geometric adaptive tracking control of a quadrotor unmanned aerial vehicle on SE(3) for agile maneuvers,' *Journal of Dynamic Systems, Measurement, and Control*, vol. 137, no. 9, 2015. DOI: 10.1115/1.4030419. [Online]. Available: https://doi.org/10.1115%5C%2F1.4030419.

[27] D.-A. Clevert, T. Unterthiner and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, 2015. DOI: 10.48550/ARXIV.1511.07289. [Online]. Available: https://arxiv.org/abs/1511.07289.

[28] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[29] D. P. Kingma and M. Welling, 'An introduction to variational autoencoders,' *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. DOI: 10.1561/2200000056. [Online]. Available: https://doi.org/10.1561%5C%2F2200000056.

[30] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2013. DOI: 10.48550/ARXIV.1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114.

[31] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition,' in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[32] P. K. Nathan Silberman Derek Hoiem and R. Fergus, 'Indoor segmentation and support inference from rgbd images,' in *ECCV*, 2012.

[33] D. Makoviichuk and V. Makoviychuk, *Rl-games: A high-performance framework for reinforcement learning*, https://github.com/Denys88/rl_games, May 2021.

[34] N. Ravi *et al.*, 'Accelerating 3d deep learning with pytorch3d,' *arXiv:2007.08501*, 2020.

[35] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[36] Q. Zhang, M. Zhu, L. Zou and M. L. Y. Zhang, *Learning reward function with matching network for mapless navigation*, 2020. DOI: 10.3390/s20133664. eprint: 32629934.

[37] H.-T. L. Chiang, A. Faust, M. Fiser and A. Francis, *Learning navigation behaviors end-to-end with autorl*, 2019. arXiv: 1809.10124 [cs.RO].

Appendix A

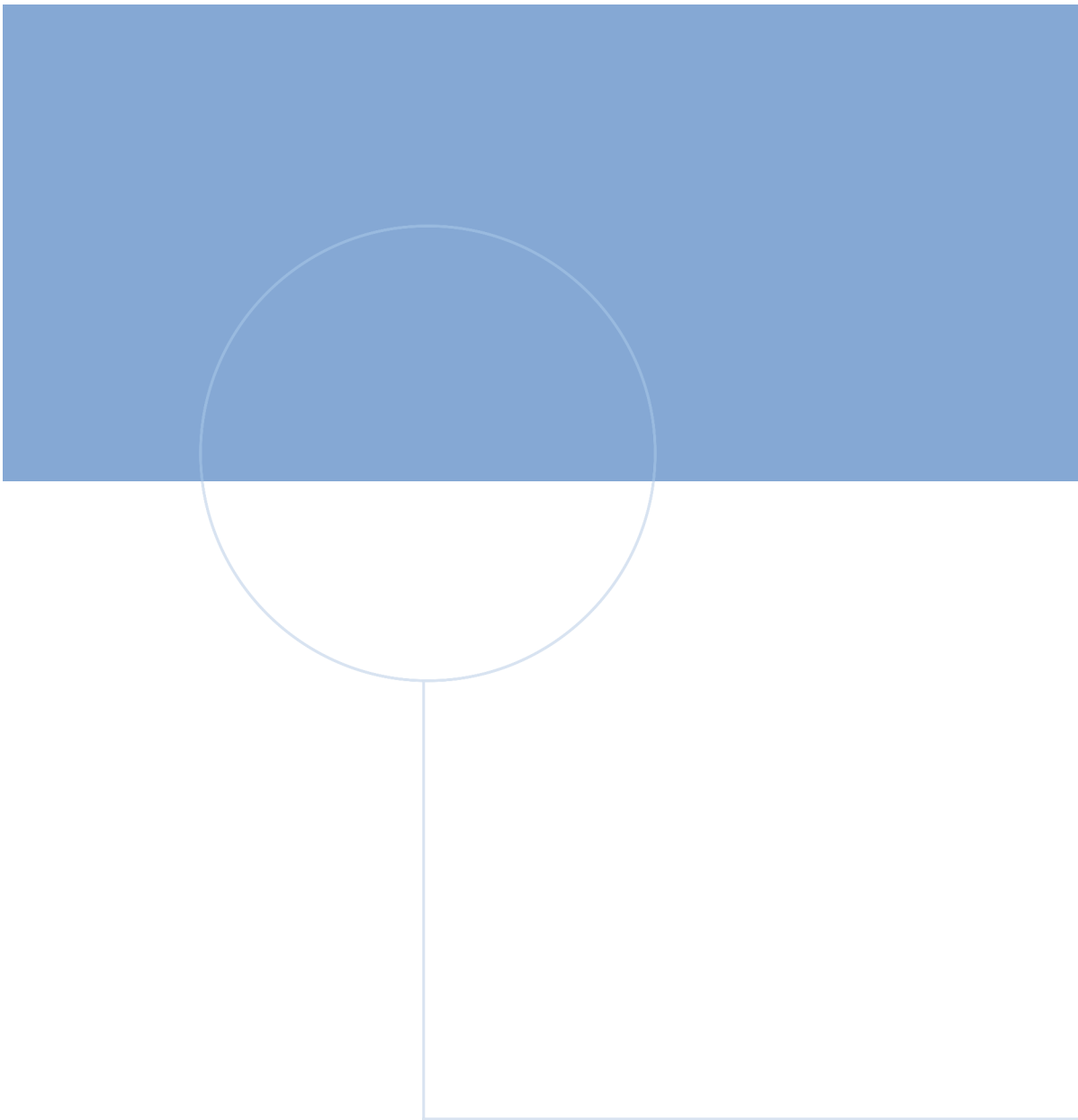
Algorithms

A.1 Auto-Encoding Variational Bayes

Implementation is from the original paper [30] and the noise distribution is chosen to be Gaussian.

Algorithm 4 Auto-Encoding Variational Bayes (AEVB)

1: Initialize parameters $\boldsymbol{\theta}, \boldsymbol{\phi}$
2: **repeat**
3: Draw N random data points from the dataset to produce $\boldsymbol{X}^N$
4: Collect random samples $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ from noise distribution
5: Calculate gradients of the minibatch estimator $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}^M, \boldsymbol{\epsilon})$
6: Update parameters $\boldsymbol{\theta}, \boldsymbol{\phi} \leftarrow \boldsymbol{g}$ using the gradient
7: **until** parameters convergence



NTNU

Norwegian University of
Science and Technology