NTNU
Kunnskap for en bedre verden

Reward Shaping in Quadcopter Control Using Deep Deterministic Policy Gradients

Eilef Olsen Osvik

Autumn 2020

PROJECT THESIS
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor: Kostas Alexis

Preface

During the autumn semester 2020 at NTNU we have carried out an extensive literature study on Deep Reinforcement Learning methods, and implemented one of the algorithms in a simulator. While the thesis is written in solitaire, the project has been in collaboration with Martin Aalby Svalesen and is in complement with his project thesis "Robustness in Deep Reinforcement Learning for Quadrotor Control". This project is a continuation of the work done by the Autonomous Robots Lab [1] in Nevada, US, which emerged from the DARPA subterranean challenge in 2019 [2]. Their goal is to navigate and map in GNSS-denied and geometrically constricted spaces using Unmanned Aerial Vehicles (UAVs) with efficiency and robustness. The master thesis to be conducted next semester will build on this by further combining learning based methods with sensory inputs like vision and LiDAR. For now, we are focused on learning methods for control of an UAV using continuous state space reinforcement learning.

Preliminaries for this project is a brief understanding of the basic principles behind Machine Learning and control theory.

Trondheim, 2020-12-28

Eilef Olsen Osvik

Executive Summary

In this work, the topic of how the components and values of the reward function in a reinforcement learning problem can impact the dynamics of a quadrotor is discussed. The particular problem to be solved for the quadrotor, is waypoint navigation.

Firstly, the necessary information and methods regarding reinforcement learning for continuous state and action space are accounted for. Then, the problem to be solved is characterized along with the metrics relevant for success or failure. The results are then presented.

The conclusion of this work is that there are many ways of solving the problem in an efficient and robust manner. The solution of hand crafting specific reward shapes is much more resource consuming while showing less potential than many other solutions. These other proposed solutions will be accounted for in the section of future works at the end of the report.

Contents

Chapter 1

Introduction

1.1 Motivation

The topic of control for unmanned aerial vehicles (UAVs) in confined environments has been extensively researched in recent years. The ability to be agile, have a quick response, and act remotely makes highly functioning drones a very applicable solution for many problems. Recent advances in hardware technologies, such as enhanced processing units and prolonged battery life have made drone systems more durable and available for real-world problems. An improvement in this field can have significant impact in a wide variety of fields, for instance search-and-rescue[3], military operations[4], fishing [5], firefighting [6][7], maintenance [8] and countless more [9].

The topic discussed in this project is essentially quadrotor waypoint navigation. It is important to note that this objective has been solved, quite well and numerous times, by traditional control methods like the Linear-quadratic regulator (LQR) and Model-Predictive Control (MPC) [10][11][12]. As these methods have been around for a long time, there exists extensive literature on increasing efficiency, stability and other aspects of the methods. The downside, as described in [13], is the computational complexity of constrained optimization. The paper also shows promising results of a learning based method, showing similar results with a fraction of the computational cost.

This is the primary motivation for this project. Here we propose solving the problem using reinforcement Learning (RL) and neural networks. Recent breakthroughs in RL for continuous control have made reinforcement learning in robotics more applicable[14] [15]. Flying insects with small neurological brains and short life span are able to learn and perform controlled flight and obstacle avoidance without solving the constrained optimization problem in an MPC. This motivates the existence of a less computationally heavy learning based method for solving the problem. Further motivation is the recent advancement of the RL chess bot, AlphaGo [16] which has been observed to outperform all chess bots and grand masters in chess. Alphago uses similar methods as will be discussed in this project. This encourages the idea of RL being able to solve complex systems with superior performance compared with human experts.

The problem to be solved in this project thesis is a part of a larger goal. During the spring semester of 2021 a Collision resilient aerial navigation algorithm for unmanned aerial vehicles in GNSS-denied and geometrically constrained environments is to be developed, including environment sensing abilities.

Developing a basis for navigation of UAVs using RL methods is therefore a crucial step in this process, and will be explored thoroughly in this project thesis.

1.2 Background Information

This section will explain and elaborate on the necessary knowledge and methods to understand the project. To discuss the topic thoroughly we will look at the structure of a basic reinforcement learning problem, and how it relates to control theory and dynamic programming. This direction is then extended into general reinforcement learning and the difficulties as well as the tools for implementing it in real-world robotics. Most of the background information on RL in this chapter is from the highly acclaimed textbook "Reinforcement Learning - An Introduction" [17].

1.2.1 Reinforcement Learning

Reinforcement Learning (RL) is a set of methods based on the interaction between an environment and an agent. The agent has to be able to sense the state of the system and affect it by some action, and it is inherently attempting to learn from which actions it receives the largest rewards in accordance to some given objectives. The interactions between the agent and the environment is a repeated sequence of steps between the actor and environment . These sequences can be described in terms of their associated time steps t as illustrated in 1.1 and described in the list below.

- For a time step t, the agent senses some state, s_t, the environment is in.
- The agent calculates some action, a_t, and applies it to the environment.
- The environment transitions to a new state, s_{t+1}, and calculates a scalar reward, r_t, based on the tuple (s_t, a_t, s_{t+1}).
- The agent senses the new state s_{t+1} and the process is repeated.

The overall idea of RL is to train the agent to apply the actions that will maximize the reward received for any state. By receiving the maximum reward for each state the agent is perceived to be optimal. To describe this further we will elaborate on the mathematical framework that encapsulates these principles and allows for the dynamic described above. The Markov Decision Process framework is a useful mechanism to model this type of environment-agent interaction and is fundamental for the RL methods.

Markov Decision Process (MDP)

A finite Markov Decision Process is defined by a set of states S, $\{s_1, ..., s_n\}$ and a set of actions A, $\{a_1, ..., a_k\}$. The size of the state space is of some size $|S| = n$ as is the action space by some other size $|A| = k$. These are often denoted as *state-action pairs* (s, a), connecting the particular action a applied at a certain state s. If the agent is in a state $s \subseteq S$ and applies an action $a \subseteq A$ it is transitioned to another state $s' \subseteq S$

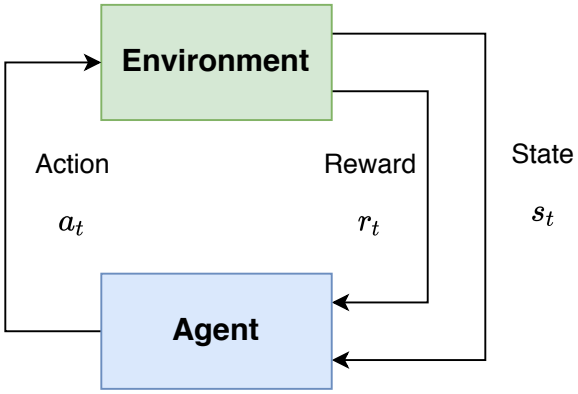


Figure 1.1: An agent receiving a state $s_t$from the environment. It then applies an action a_t to the environment and receives a reward r_t.

and a scalar reward is calculated by the *reward function* $R(s,a,s')$. Generally it is said that the reward function describes implicitly the goal of the learning[18], as the agent is supposed to maximize the future rewards. By attributing large rewards for desired behaviour and low rewards for unwanted behaviour, the behaviour of the agent can be trained. There are two different notations when mathematically describing the relationship between states, actions and next states. In the list above and figure 1.1, is the time variant notation described as (s_t, a_t, s_{t+1}). In this paragraph is the time invariant notation of (s,a,s'). These notations are convenient to interchange for deriving the methods later.

The probability of a system transitioning from one state s to the next state s' can be described as a probability written as $T(s'|s,a)$, where T is the *transition function*. It is a deterministic function mapping the three arguments to a probability $T : S \times A \times S \rightarrow [0,1]$. Each element in the transition matrix is a scalar in the range $[0,1]$. The sum of all actions in a given state are 1 as to fulfill the statistical properties.

An agents' policy is the way it chooses the particular action to be applied based on the current state. It can be expressed as a mathematical mapping between the state space and the action space: $\pi : S \rightarrow A$. Where the function π is referred to as the policy of the agent.

The MDP is explicitly defined by the set of actions A, set of states S, the transition matrix P and the reward function R. The transition matrix and reward function is referred to as the dynamic of the system.

Keeping record of past actions and states would be infeasible for complex models with long run time. Therefore a state must inhibit all information of what the agent has done in the past that is relevant for future actions. This is called the *Markov Property* and is a crucial principle for this project. [17].

Solving MDPs

Solving MDPs can be done by a multitude of methods. In its simplest form we have the mathematical model of the system, meaning the transition matrix and reward function. An agent is often interested in taking the reward that allows for high future rewards as well. We can therefore express the sum of rewards from a time step t to ∞ as th function G_t in equation 1.1.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{n=0}^{\infty} \gamma^n R_{t+n+1} \tag{1.1}$$

Here R is the reward for each time step and γ is the *discount factor* which determines how much we emphasise the future rewards [19]. γ is in the range $[0,1]$ where $\gamma = 0$ means caring only about the first reward, what is known as the greedy choice, and $\gamma = 1$ means weighing future rewards equally as the next reward. Typically γ is in the interval $[0.9, 0.99]$.

The function 1.1 has recursive properties as shown in equation 1.2.

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{1.2}$$

Since the process is stochastic, the calculations will be on the expected mean of the future returns as shown in equation1.3.

$$G_t = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...\right] = \mathbb{E}\left[\sum_{n=0}^{\infty} \gamma^n R_{t+n}\right] \tag{1.3}$$

Using 1.3 we can calculate the value of a single state when following a particular policy π. This is the expected future rewards for future state and is referred to as a value function $V(s)$ and is shown in equation 1.4.

$$V^{\pi}(s) = \underset{\pi}{\mathbb{E}}\left[\sum_{k=0}^{\infty} (\gamma^k R_{t+k+1} | s_t = s)\right], \text{ for all } s \in S \tag{1.4}$$

This is essentially a list of the different states where each particular state $v(s) = V^{\pi}(s)$ has a corresponding value of expected future rewards from that state. Further, the same method can be used to calculate the expected future rewards of the state-action pair (s,a). This is referred to as the $Q(s,a)$ function and shown in 1.5.

$$Q^{\pi}(s,a) = \underset{\pi}{\mathbb{E}}\left[\sum_{k=0}^{\infty} (\gamma^k R_{t+k+1} | s_t = s, a_t = a)\right] \tag{1.5}$$

The Q function in 1.5 is the expected future return on taking the action a_t in the state s_t and thereafter following the policy π. For the discreet problem, these can be calculated for each action in each state, forming a Q-matrix mapping each state-action pair to an expected future value. If the MDP is sampled randomly and updates the indices of the matrices described above by the functions 1.4 and 1.5. Then the statistical values in the matrices will converge to the actual sum of future rewards for each state or state-action pair as discussed in [17]. This form of random sampling to estimate the V and Q functions is referred to as *Monte-Carlo simulation*[20].

The reason for using DP to solve the MDP is that the problem easily can be split into several subproblems. This property is shown by building on 1.4 in equation: 1.6.

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=1}^{\infty} (R_t + \gamma^k R_{t+k} | s_t = s)\right] = \sum_{s'} T(s, \pi(s), s')(R(s,a,s') + \gamma V^{\pi}(s')) \tag{1.6}$$

This is referred to as the Bellman equation [21] and is crucial for solving these problems.

The Principle of Optimality Regarding RL

An important discussion in RL is the topic of optimal action. As we model our system as a series of actions, $\{a_0, ..., a_n\}$, and states, $\{s_1, ..., s_{n-1}\}$, we can categorize each application of action on a state as a subproblem of the larger problem. Meaning that for any given state, the optimal action in that state is the action that maximizes our value function in that state. In other words, a policy is deemed optimal if it takes the action that has the highest expected future rewards for each state. Comparing policies can be done with use of the value functions. This is shown in equation 1.7 where the optimal policy π^* is described as related to the highest corresponding value function. This is referred to as the optimal value function V^*.

$$\pi^* \leq \pi \quad \iff \quad V^{\pi^*} \leq V^{\pi} \quad \text{For all} \;\; s \in S \tag{1.7}$$

The principle of optimality according to Bellman is that any optimal solution to a subproblem of the whole problem is also an optimal subsolution to the problem as a whole[21]. So an optimal state-action tuple solving a subproblem is part of the larger optimal solution for the problem. Note that it is possible to have more than one optimal policy.

Similarly, we can derive an optimal Q function, Q^*, by finding the value of each state-action tuple and when following optimal policy, π^*. This is can be observed in equation 1.8.

$$Q^*(s,a) = \max_{\pi} Q_{\pi}(s,a) = \mathbb{E}\left[R_{t+1} + \gamma V^*(s_{t+1}) | \, s_t = s, a_t = a\right] \tag{1.8}$$

If the optimal Q function 1.8 is known, we can realize the corresponding optimal policy by finding the particular actions for each state that has the highest expected reward. This is referred to as the greedy policy and seen in 1.9.

$$\pi^*(s) = \underset{a}{argmax} \; Q^*(s,a) \tag{1.9}$$

Similarly, we can find the optimal value function by using the action that maximizes our Q function

$$V^*(s) = \max_{a \in A} Q^*(s,a) = \max_{a \in A} \mathbb{E}\left[R_{t+1} + \gamma V^*(s')\right] \tag{1.10}$$

Observe the usefulness of the Q function for finding the optimal policy. This usefulness has inspired many methods RL to be based on approximating the Q or V function and deriving the optimal policy from there. This is the principle behind Temporal-Difference Learning, and is very important for the methods used in chapter 2.

Temporal-Difference Learning (TDL)

TD attempts to estimate the value function directly using experience. In general for this set of methods it consists of updating the value function by some difference between expected future values and the immediate rewards for each time step. This method does not require any knowledge about the model of the system. The TD0 algorithm is the simplest for of this method. It is based on approximating the value

function as observed in equation 1.11.

$$V(s_t) \leftarrow \alpha V(s_t) + (\alpha - 1)[R_{t+1} + \gamma V^{\pi}(s_{t+1}) - V(s_t))] \tag{1.11}$$

Here, α is some step length that can be tuned. The method is to traverse the state space according to some policy π and evaluate it using the value function as expressed in 1.11. Since the calculation is based on a prior calculation of the value function, the method is called a *bootstrapping* method. This is not ideal, however it can be shown that the value function converges to the actual value function of the policy[17]. In addition this method can update the value function during simulation. Updates can be done after each time step, where prior methods had to complete the simulation[17]. TD0 can only be used to evaluate the policy and not explicitly to improve it.

Building on this principal is the Q-learning algorithm. It attempts to estimate the optimal state-action function Q^*. In this way it can determine the optimal policy π^* as seen in1.9. The updating rule of Q-learning is found in equation 1.12.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{1.12}$$

In equation 1.12, the optimal state-action function Q^* is approximated. The premise of this method is that the expected mean of the Q function converges to the actual value. This means that if all state-action pairs were visited an infinite amount of times, then $Q = Q^*$. The algorithm searches through the state-action space exploring the different state-action tuples and updates the Q values for them as seen in 1.12. Q-learning is off-policy meaning that it approximates Q^* independently of the policy being followed.

Here the behaviour of the system is approximated through sampling it without prior knowledge of the MDP and updating the Q function. As a direct result of this, the policy we learn is greatly affected by the states and actions it discovers. This means we have to be aware of how it sweeps the state-action space leading to the exploration vs exploitation problem in RL.

Exploration vs Exploitation

By choosing a sub-optimal action in a given state we can attempt to discover new and better state-action pairs and from here acquire a better policy. This is referred to as *exploration*. *Exploitation* is when the optimal action is taken. Meaning that given the current policy this action has the highest score. The exploration versus exploitation problem is inherent to RL[22] and makes it distinct from other forms of ML. In methods like supervised learning, the engineer can ensure that the data set is valid for the later application of the model. In RL the engineer has to ensure that the agent discovers enough of the state-action space in order to generate a policy valid for later applications. This makes the exploration/exploitation problem important. An agent that only exploits the knowledge by choosing the highest expected reward for that policy, referred to as the "greedy choice", is prone to sub-optimal solutions. On the other hand, if it only explores, it will not learn to stick to a single path, and rather choose all actions at random. Combining these two concepts are important for solving a problem using reinforcement learning.

In a discrete state space setting a common solution is the ϵ-greedy action selection. Here the action chosen is stochastic with two outcomes, the greedy or a random choice. The stochastic parameter $\epsilon \in$

$[0,1]$ is designed such that a greedy action is chosen with probability of $(\epsilon - 1)$ and the random action with probability ϵ. Random exploration is important to ensure that the bias of the initial state of the system is diminished [23].

1.2.2 Deep Reinforcement Learning

Problem of Continuous State and Action Space

So far the topic of reinforcement learning has been discussed in a tabular environment. To be able to use RL in robotics and realistic scenarios we have to discuss how to apply it in continuous state and action spaces. A continuous environment has, by definition, an infinite amount of states, in the same way a continuous line has an infinite amount of points along it. In a continuous state and action space, the Q matrix would be infinite as well, making it infeasible to represent the dynamics with finite computing.

There are several solutions to this issue, and in this project the use of Neural Networks (NN) to approximate the continuous state and action space will be relevant and discussed. This field of research is called Deep Reinforcement Learning (DRL).

Neural Networks

There are many different models in ML, and relevant for this project is the neural network. A NN consists of groups of neurons connected with each other in layers. The reader is assumed to have some understanding of how the neuron works. There are many types of neural networks. In this work we will discuss feed forward deep neural networks consisting of an input layer, an amount of hidden layers and an output layer. The basic premise of the neural network is to approximate some underlying function of a data set as seen in equation 1.13.

$$J(\theta) = f(x) \tag{1.13}$$

The approximation function is some function $J(\theta)$ where θ symbolises the weights of the function. The underlying function is referred to as $f(x)$. In an NN, the input is fed through the different layers. The general idea is to extract relevant information for the task. this is referred to as *forward propagation*. As the network weights often are initialized randomly, they need to be updated according to how well they did in the prediction as seen in 1.13. This is referred to as *back propagation* and utilizes the loss function observed in equation 1.14.

$$L(y, \hat{y}) = \sum_{i=0}^{n} (y - \hat{y})^2 \tag{1.14}$$

$\hat{y}$ is the prediction from the network, given some input. The actual value from the data set is y. Minimizing the loss function is the same as attempting to output the same as the underlying function $f(x)$. After calculating the loss function for some amount of data n the weights θ can be optimized by *gradient descent*. Meaning calculating the gradient of the loss function with respect to the weights as seen in equation 1.15.

$$\frac{\partial L(y,\hat{y})}{\partial \theta} \tag{1.15}$$

The gradient is a vector pointing in the direction which will maximize the loss function. As the intention is to minimize it, the update has to move some step length α in the negative direction of the gradient as observed in 1.16.

$$\theta_{new} \leftarrow \theta_{old} - \alpha\left(\frac{\partial L(y,\hat{y})}{\partial \theta}\right) \tag{1.16}$$

In this example we referred to a problem where we have labeled data. This is referred to as *supervised learning* in the machine learning domain. RL is a different set of methods where the agent has to discover unlabeled data on its own. Neural networks are here often used to approximate different parts of the problem as we will discuss further.

The term *deep learning* is contributed to neural networks with more than one hidden layers. As each layer approximates some features and propagates the information further, these networks are able to approximate complex non-linear functions. This property makes them very useful in applications like robotics.

Approximation in Action-Value Space, the Critic

A neural network has the potential to parameterize the continuous state-action space by some function $J(\theta^Q)$ and map it into a single value of expected future rewards. This is the same functionality as has been discussed of the Q function 1.5 following a specific policy[24]. This is the method proposed in the Deep Q-Networks (DQN) 2015 [25]. Q learning, as discussed earlier, approximates the optimal Q-function by simulation and updating the Q matrix continually as seen in the update rule 1.12. The same method is used [26], but the Q function is approximated by a neural network as observed in equation 1.17. Note that this method is also off-policy meaning it can use another policy to explore the state-action space like the ϵ-greedy method.

In an actor-critic method this is referred to as the critic. The weights will be denoted θ^Q. The relationship with the Q function is described in 1.17.

$$J(\theta^Q) = \mathbb{E}\left[Q^{\pi(.|\theta^Q)}(s,a)\right] \approx Q^*(s,a) \tag{1.17}$$

The method made use of a *replay buffer* to ensure stability during training. This will be discussed further in the DDPG algorithm. The DQN method is made to tackle a high dimensional state space and a low dimensional action space. This is not applicable to the problem discussed in this project thesis.

Approximation in Action Space, the Actor

A neural network can also be used to find the policy in continuous state-action space. This is referred to as the actor and its weights are denoted θ^π as seen in equation 1.18.

$$\pi(s|\theta^\pi) \tag{1.18}$$

π should be trained to approximate the mapping between the state and the action. This means that the output of the actor should be continuous, as discretising the action space is not ideal [14]. This is in part because of *the curse of dimensionality* where the amount of states and actions needed to represent a system of 6 degrees of freedom will be so large that attributing enough data to validate the network will be infeasible.

Policy Gradient Methods

Instead of attempting to learn a value function and deriving a policy from it, we derive the parameters directly. This a is set of methods called *Policy Gradient Methods* (PGM). Examples of advantages of PGMs are guaranteed convergence to at minimum locally optimal solutions, are proved to handle imperfect state representations and the optimal policy is often more compactly represented than the value function[27]. We want to update the parameters θ according to some performance measure function $J(\theta)$. This is generalized in equation 1.19.

$$\theta_{t+1} = \theta_t + \alpha\widehat{\Delta J(\theta_t)} \tag{1.19}$$

Here, $\widehat{\Delta J(\theta_t)}$ is an estimated value for the gradient of the performance measure with respect to the parameters θ. If we define the performance measure function J as $J(\theta) \doteq V_\theta(s_0))$, where the system is assumed to be episodic and initialized in some non-random state s_0, then the *Policy Gradient Theorem*[17] is defined as equation 1.20.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a Q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{1.20}$$

This gradient is proportional to the true gradient for the episodic case. For the continuing case, the gradient is exactly alike. The function $\mu(s)$ is the on-policy distribution of states under the policy π. It gives information of the chance of states occurring under the specified policy.

This is an exact solution for the update of the policy gradient. But to be able to update it using samples we have to rewrite it a bit. Considering equation 1.19. The outer sum is essentially a sum over the states weighted by their chance of occurring under the policy. This can be altered to the expectation for a specific state in a time step t, and then be randomly sampled. We are then left with equation 1.21.

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\pi}\left[\sum_a Q_\pi(s_t,a) \nabla \pi(a|s_t,\theta)\right] \tag{1.21}$$

This is referred to as the *all actions* update, as it involves a sum over all actions. This is a valid method, however, we wish to be able to update based on a single action, a_t. Similarly as the argument that contributing to equation 1.21. The sum over all actions in equation 1.21 can be replaced by the expected mean of the actions under the policy π, and then determined by sampling. To introduce the mean we have to weigh each term of the sum with the probability of action a_t being chosen, meaning dividing each term with $\pi(a_t|s_t,\theta)$. As shown in equation 1.22.

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\pi}\left[\sum_{a} \pi(a|s_t,\theta) Q_\pi(s_t,a) \frac{\nabla\pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}\right] \tag{1.22}$$

When changing the a to the sample a_t we can alter equation 1.22 into equation 1.23.

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\pi}\left[Q_\pi(s_t,a_t) \frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\right] \tag{1.23}$$

Further applying that the expected mean of $Q(s_t,a_t)$ is the sum of future rewards G_t from equation 1.1 leads to equation 1.24.

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\pi}\left[G_t \frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\right] \tag{1.24}$$

This equation lets us alter the update function described in 1.19 into the following policy parameter update1.25.

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)} \tag{1.25}$$

This update is called the REINFORCE update. Where α is some step length. This is a highly intuitive update. The parameters in policy space are updated in the direction of the policy for taking the same action a_t upon future visitations to the state s_t. The magnitude of the update is defined by the received rewards, this means that high acquired rewards will lead to a policy attempting to do more of the action. This is a Monte Carlo method as we have G_t, meaning the actual received rewards, this constrains the updates to be done after completed simulation. Also, by dividing by the probability of taking the specific action the updating advantage actions that are frequently chosen by the policy is evened out. This method assures improvement in expected return for small step lengths α. However, as a Monte Carlo method, it is prone to high variance and slow learning.

For a lower variance, and therefore a better performance, we can add some baseline to the update with the intention of distinguishing poor and good actions better. The baseline should vary with the state as some states have generally high values for their actions and should therefore have a high baseline and vice versa. With that in mind we will use some prediction of the state value, here denoted by the function $\hat{V}(s_t|\theta^V)$ with corresponding weights θ^V. This application can be observed in equation 1.26.

$$\theta_{t+1} = \theta_t + \alpha(G_t - \hat{V}(s_t,\theta^s)) \frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)} \tag{1.26}$$

To move away from the Monte Carlo solution in equation 1.26 we want to use a temporal-difference method to update the parameters with the one-step return R_t. This method is called the actor-critic method with a bootstrapping baseline. We want to use a bootstrapping technique because the bias introduced by it can be beneficial for learning[17]. The updating rule of such a general actor-critic method can be seen in equation 1.27.

$$\theta_{t+1} = \theta_t + \alpha(R_{t+a} - \gamma\hat{V}(s_{t+1},\theta^s) - \hat{V}(s_t,\theta^s)) \frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)} \tag{1.27}$$

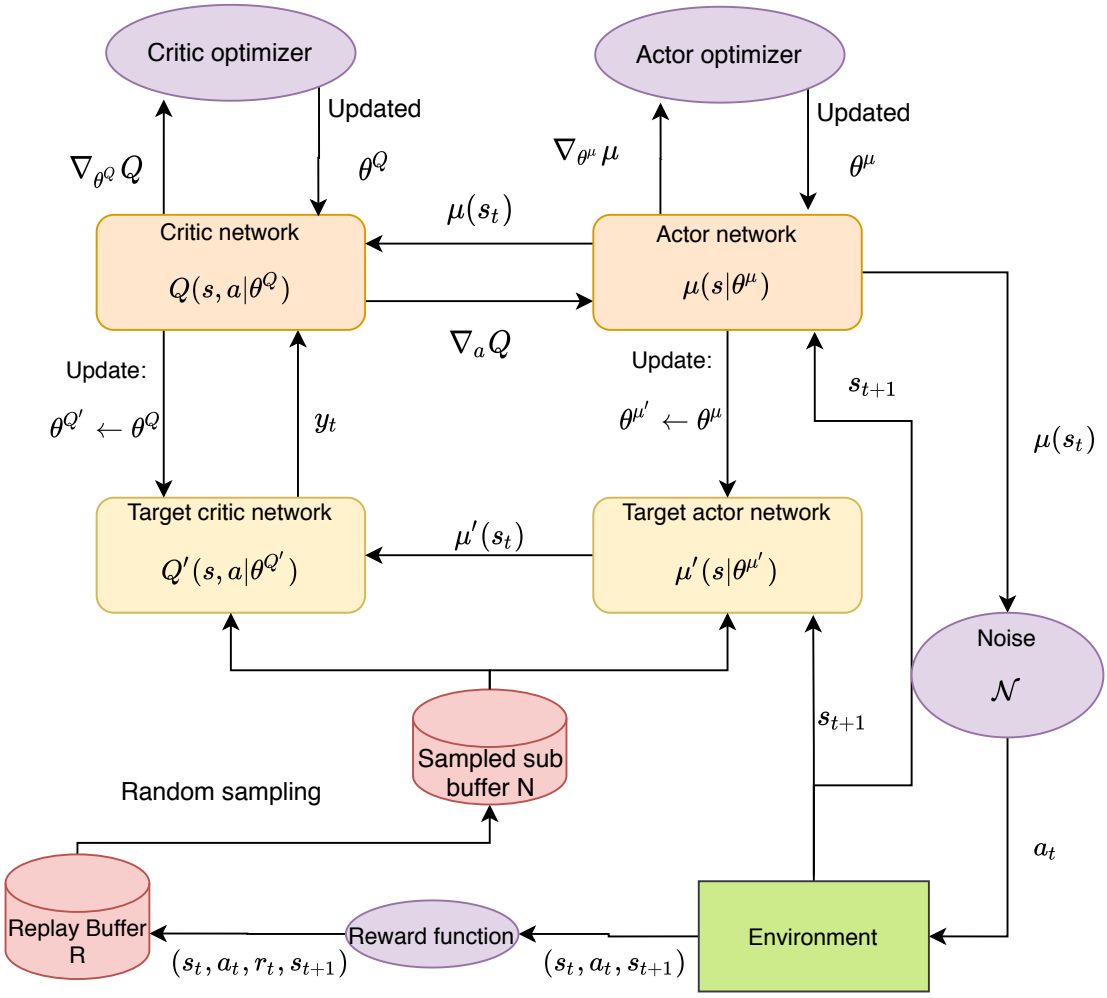


Figure 1.2: The DDPG algorithm visualized in a block diagram.

This method will be explored deeper in the actor-critic algorithm Deep Deterministic Policy Gradient.

Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm is a deep actor-critic method for reinforcement learning from 2015 [14]. The general idea of an actor-critic combination, as described in the section above, is to let the actor network predict an action given a state and the critic predict the expected future rewards of the action in the state. The information of the critic can then be used to update the weights of the actor. As non-linear function approximators are being used here, it is important to note that convergence is not guaranteed for neither the critic nor the actor networks. The different modules of the DDPG is visualized in figure 1.2.

DDPG is an off-policy algorithm, meaning it can improve its policy by using samples from older policies. This enables a distinction between an exploration policy, from here denoted $\beta(s)$, and an exploitation (greedy) policy, from here denoted $\mu(s)$. β has a stochastic element ensuring exploration, while μ is a deterministic function making calculation easier and ensuring a predictable response.

The algorithm can be viewed in 0 in appendix B as a whole. The first step of DDPG is initialization. The

actor network $(s|\theta^{\mu})$ is randomly initialized with the weights θ^{μ} and a corresponding target actor network $\mu'(s|\theta^{\mu'})$ is initialized with weights $\theta^{\mu'} = \theta^{\mu}$. The critic network, $Q(s,a|\theta^{Q})$, is randomly initialized with the weights θ^{Q} and its corresponding target critic network, $Q'(s,a|\theta^{Q'})$ with weights $\theta^{Q'} = \theta^{Q}$.

In addition, a finite size replay buffer R is initialized.This buffer is used to store information for training. When R is full, the oldest sample is deleted to make room for the new sample.

Further, a noise process N is initialized. The noise is a stochastic element to ensure exploration in a manner similar to the ϵ-greedy method. The noise used in this project is an Ornstein-Uhlenbeck process [28] the benefit of which is temporarily correlated noise, leading to temporarily correlated exploration. The system is then initialized in a given state s_1.

For each time step $t > 0$, the exploration actor $\beta(s_t) = \mu(s_t|\theta^{\mu}) + N$ selects an action a_t, which is applied to the environment. A reward r_t and a new state s_{t+1} is observed and the tuple (s_t, a_t, r_t, s_{t+1}) is stored in R.

The parameters are optimized for each time step. A random subset of size $N(s_i, a_i, r_i, s_{i+1})$ is therefore sampled uniformly of R to minimize the correlation between successive datapoints in the dataset. This is referred to as temporal autocorrelation and can lead to model bias [29].

Building on the parameterized critic function in 1.17 and the Bellman equation in **??**. The weights θ^{Q} can be optimized by minimizing the loss function L described in 1.28.

$$L(\theta^{Q}) = \frac{1}{N}\sum_{i=1}(y_i - Q(s_i, a_i|\theta^{Q}))^2 \tag{1.28}$$

Here the target networks μ' and Q' are used to calculate y_t. It is defined as the one step ahead reward as denoted in equation 1.29. This is essentially the same as the Bellman equation.

$$y_i = r_i(s_i, a_i) + \gamma Q(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{1.29}$$

The policy μ is a deterministic function and can therefore easily be calculated. The updating function can be viewed as the sum of the L2 norm between two important parts. The first part is the approximated future reward from time step t as seen in $Q(s_i, a_i|\theta^{Q})$ in equation 1.28. The second is the sum of the actual reward from the same time step r_i and the *discounted* future rewards seen from the next time step following the greedy policy 1.29. In theory both Q and y are dependent of the same weights θ^{Q}. this can lead to instability in training. This is circumvented by using the target network to calculate y. The weights θ^{Q} should converge towards the actual expected future rewards from a state-action tuple.

Considering the actor now. The actor will be optimized using the equations established by the policy gradient theorem in 1.20. This will be done with respect to the actor weights θ^{μ}. The gradient of the performance measure function $J(\theta^{\mu})$ can be seen in equation 1.30.

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_{i=1} \nabla_{\theta^{\mu}} Q(s,a|\theta^{Q})|_{s=s_i, a=\mu(s_i|\theta^{\mu})} \tag{1.30}$$

Here the actor function is differentiable and by applying the chain rule of derivation, the gradient of the policy is extracted in equation 1.31[15].

$$\nabla_{\theta^\mu} J \approx \sum_{i=1} \nabla_a Q(s,a|\theta^Q)|_{s=s_i a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i} \tag{1.31}$$

The gradient of equation 1.31 is identical to the policy gradient theorem 1.20. This gradient is then used to update the actor network weights θ^μ as shown in equation 1.19. Lastly, the target network weights are *softly* updated by some linear function of $0 < \tau \ll 1$ as described in equation 1.32 and 1.33.

$$\theta^{\mu\prime} = \tau\theta^\mu + (1-\tau)\theta^{\mu\prime} \tag{1.32}$$

$$\theta^{Q\prime} = \tau\theta^Q + (1-\tau)\theta^{Q\prime} \tag{1.33}$$

This is an important part of DDPG as sudden changes in the network parameters can lead to instability in training and often causes the network to diverge[14]. The trade off is time. As τ is a small number, the updates takes longer to converge. Instability in training means small changes in the weights can lead to vastly different behaviour. This is an important aspect of RL for robotics as unpredictive, hazardous behaviour from the robot can cause harm to people or the robot itself.

Trust Region Policy Optimization (TRPO)

The TRPO algorithm is from a paper from 2015 [30]. Like DDPG, it is a policy gradient algorithm, though it differs in many other ways.

The first part of the algorithm gathers different state-action pairs. There are two different methods here, *single path* or *vine* procedure. Single path means sampling the state-action pairs of a trajectory beginning in some initial state $s_0 \sim \rho_0$ where ρ is the state distribution. The applied actions are saved and $Q_\pi(s_t, a_t)$ is calculated by using the discounted sum of rewards generated. As it is calculated after simulation, the exact rewards connected to the state-action tuple is known and easily calculated. The *vine* procedure is based on Monte-Carlo simulation where a subset of states, called the *rollout set*, from the single path is randomly chosen and the environment and agent is reset to one of the states. Then a short trajectory simulation is performed. This is known as a *rollout*. The Q values are estimated for these datapoints.

The *advantage estimate* $A(s,a)$ is the estimated value of the selected action. It is comprised of the difference between two parts. The first part is the $Q(s,a)$. The other part of the advantage function is an approximated value function $V(s)$ mapping the state to a value of expected future rewards. This value is predicted by means of a neural network. The advantage estimate is shown in 1.34.

$$A(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{1.34}$$

If the advantage function is negative $Q_\pi(s,a) < V_\pi(s)$ then the action chosen was poorer than expected, and vice versa. The logic of TRPO is to enhance behaviour where $A > 0$ and negate behaviour where $A < 0$.

Consider the parameterized policy $\pi_\theta(s)$ with weights θ. To compare two different policies,π and $\tilde{\pi}$

the equation $\eta(\pi)$ in 1.35 is used.

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathop{\mathbb{E}}_{\pi'}\left[\sum_{t=0}^{\infty} \gamma^t A_\pi(a,s)\right] \tag{1.35}$$

A new policy has the value of the old policy plus or minus the advantages found of the states traversed in the new policy. The advantage of each state is assumed the same for the old and new policy.

The parameter $\rho_\pi(s)$ is introduced, which represents the discounted visitation frequency for arriving in a state s while following a policy π. In this way the system can be represented by state transitions instead of time, and the policy π can be explicitly stated in the equation as seen in 1.36.

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \tag{1.36}$$

When $\pi \approx \tilde{\pi}$ is a valid approximation, we can use $\rho_\pi \approx \rho_{\tilde{\pi}}$ to alter the equation. For this approximation to be valid, the new policy cannot be too far away from the old policy. By solving for $\Delta\eta$ and defining $\tilde{\pi}(a|s) = \pi(a)\frac{\tilde{\pi}(a|s)}{\pi(a|s)}$ we can rewrite 1.36 as 1.37.

$$\eta(\tilde{\pi}) - \eta(\pi) = \Delta_\eta \approx \sum_s \rho_\pi(s) \sum_a \pi(a) \frac{\tilde{\pi}(a|s)}{\pi(a|s)} A_\pi(s,a) \tag{1.37}$$

Considering the last equation, 1.37. The sums of the old policy, $\sum_s \rho_\pi(s) \sum_a \pi(a)$, and the advantage function is a scalar that needs only to be calculated once. Meaning several policy updates can be done by the same simulated trajectory. The parameter to optimize over is therefore $\frac{\tilde{\pi}(a|s)}{\pi(a|s)}$. By altering the weights of $\tilde{\pi}$ we can explore the policy space and find better policies by ensuring $\Delta\eta$ positive. The important aspect is that the approximation $\pi \approx \tilde{\pi}$ and therefore $\rho_\pi \approx \rho_{\tilde{\pi}}$ holds. The policy can therefore only be optimized within trust regions where the policy is guaranteed not to deviate too far from the old policy.

KL divergence between the two policies is used as a metric of how different a policy's actions are from another policy. It is the difference between the data distributions of π and $\tilde{\pi}$. Essentially it defines the radius of the trust region. It can be written as

$$D_{KL}(\tilde{\pi}|\pi) \leq \delta \tag{1.38}$$

Where δ is the upper bound tolerance. Using KL divergence ensures a control of the policy space, and results in generally monotonic improvement of the policy. On the other hand, DDPG optimizes by steps in parameter space. This leads to some updates having small effect on the policy and some diverging the policy. The optimization problem in TRPO is formulated in equation 1.39 subject to the constraint in 1.40.

$$\max_\theta \quad \mathbb{E}\left[\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)}\hat{A}_t\right] \tag{1.39}$$

$$\text{subject to:} \quad \mathbb{E}\left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \leq \delta \tag{1.40}$$

TRPO uses the Minorize-Maximization algorithm to iteratively maximise a lower bound function M

which approximates the actual function $\Delta\eta(\pi)$. If we denote the actual objective function to some function $J(\tilde{\pi}-\pi)$ then TRPO describes the lower bound surrogate function M as the right hand side of the inequality in 1.41.

$$J(\tilde{\pi}-\pi) \geq \mathbb{E}\left[\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)}\hat{A}_t\right] - C\sqrt{\mathbb{E}\left[KL[\pi_{\theta_{old}}(\cdot|s_t),\pi_\theta(\cdot|s_t)]\right]} \tag{1.41}$$

Here the constraint is changed to a penalization with some weight C. Finding the parameters that maximises the right side of the inequality means acquiring the policy gradient and a matrix called showing some scale of the sensitivity. We then both have a direction to update and an estimate of the sensitivity in different directions. This quality is useful what distinguishes trust-region methods from other methods.

The optimization problem is then solved by a conjugate gradient algorithm, with a line search for finding the right step length. The algorithm needs to calculate the policy distribution for each sampled step length. This can be very time consuming and is one of the behind another method for constructing the *Proximal Policy Optimization* algorithm, which is heavily derived from TRPO.

The main goals for constructing PPO are: being easy to implement and tune, and maintaining sample efficiency.

Proximal Policy Optimization (PPO)

PPO, described here [31], is a trust region method building on the method of TRPO. They are similar in many ways, therefore their differences will be discussed here.

One of the main differences is the Clipped Surrogate Objective method introduced. Consider the function $r_t(\theta) = \frac{\tilde{\pi}_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}$. Essentially, if $r(\theta) > 0$, then the new policy action is better than the action from the old policy. If $r(\theta) < 0$, then the new action was worse than the old. With this function, the PPO objective function can be written as equation 1.42.

$$L^{clip}(\theta) = \mathop{\mathbb{E}}_t\left[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{1.42}$$

The objective L^{clip} is a clipped version of the TRPO objective function. Meaning that any step deviating more than ϵ is not possible. While TRPO had to perform a line search for their policy update, which was computationally tedious, PPO solves the problem by denying the gradient update any step which the policy changes too much.

In addition, the actor and the critic network of PPO share much of the same parameters. There are similarities to finding the best action given a state and calculating the value of an action and a state. Therefore the loss function used to optimize the parameters have some different elements as shown in equation 1.43

$$L_t^{clip+vf+s}(\theta) = \mathbb{E}\left[L_t^{clip}(\theta) - c_1 L_t^{vf}(\theta) + c_2 S[\pi_0](s_t)\right] \tag{1.43}$$

Here, the L_t^{clip} is the same objective function as in 1.42. The $L_t^{vf} = (V_\theta(s_t) - V_t^{target})^2$ term is a squared-error loss of the value function to optimize parameters specifically for the critic task. $S[\pi_\theta](s_t)$ is

an entropy term. This term is positive and added to ensure exploration of the state-space. Larger entropy means a larger variation in the actions chosen. By rewarding more entropy, the policy intends to explore. The parameters c_1 and c_2 are weights. The loss function $L_t^{clip+vf+s}(\theta)$ inhibits the different functionalities of the neural network and can be used to update its parameters.

In general, PPO works really well. It outperforms TRPO with simpler code and fewer tuneable hyper parameters. The use of trust region methods is highly useful, and could be a better alternative to the DDPG algorithm.

1.3 Problem Formulation

This project focuses on how to navigate a quadcopter using the DDPG algorithm for continuous state and action spaces. The specific topic this project intends to expand on is how the structure and different components of the reward function can lead to different dynamics. In the literature of RL this is referred to as reward shaping [32]. As stated earlier, the reward function implicitly describes the goal of the agent. By rewarding desired dynamic and penalizing unwanted dynamic, we should be able to reach some policy that performs better than others with respect to the desired dynamics.

1.3.1 Goal of the Project

Consequentially, the overall goal of the project is to determine how different shapes and components of the reward functions, using the DDPG algorithm, can result in different dynamics of the drone. This goal is motivated by the notion that a more complete understanding of the impact of the reward function on the drone dynamics would make tuning easier in future works.

1.3.2 The Environment

To quantify and compare different quadcopter dynamics with desired dynamics, we have to discuss the type of environment we wish to navigate in. The overall goal for the master thesis, next semester, is for the autonomous drone to explore a confined and narrow environment with a high risk for collisions. An example of this is a mine shaft. In this project the environment is simplified to the case of trying to reach a waypoint that is only a few meters away from the drone, in a collision-free environment.

1.3.3 Desired Dynamics

Even though this environment is collision free, the dynamics of the drone should have properties that minimize the risk of collisions. This is to help with the obstacle avoidance task in the master thesis.

Firstly, the model controlling the drone should be stable. Meaning that the position of the drone converges towards a specific point in 3D space. Perturbance stability is deemed solved by an inner-loop PID controller and will not be tested for in this environment. Stability is a basic, but important premise of control theory and any model showing instability should be discarded.

The shortest path between two points is the euclidean distance between them. In an environment with a risk of collision the waypoint cannot be on the other side of some foreign object. This would make the topic of reaching waypoints a much more complex discussion. When discussing the dynamic in this project, the euclidean distance between the two points will consequently be assumed obstacle-free. This implies that any trajectory deviating from this distance will pose a potential collision hazard for the drone, and should be avoided. In general, the UAV should reach the goal point in a straight trajectory from the spawn point. Upon reaching the goal point it should hoover steadily.

The performed dynamic should also be free of oscillations. Removing oscillatory behaviour is a topic frequently addressed in control theory. This behaviour is suboptimal and can pose a hazard to the drone or personnel.

Further desired dynamics is quick responsiveness, while not overshooting the waypoint. Agility as well as damping should therefore be considered.

The desired dynamic will be evaluated using three metrics:

1. The deviation from the optimal path.
2. The plotted trajectory compared to the optimal path in 3D space.
3. The plotted trajectory for each axis and analyzing it as a first order response.

The deviation from the optimal path is represented by the root-mean-square error of the shortest distance between the euclidean distance and the actual trajectory of the model.

1.4 Related Work

As described in the motivation 1.1, traditional control theory solves the topic of quadrotor waypoint navigation quite well. We will, therefore, discuss the related work in the field combining robotics and Artificial Intelligence (AI).

Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm [33] builds on the DDPG algorithm. This methods adds functionalities with the intention of countering the overestimation approximation error in the value function. These methods are Clipped Double Q Learning, learning two Q functions and choosing the lowest estimate. Updating the critic with higher frequency than the actor and target networks. Lastly, smoothing the target actor around a small area around the target will negate spikes in the value function. It is shown to outperform other methods like DDPG and the PPO in a variety of problems.

The topic of drone control with reinforcement learning has been addressed in this paper from 2017[34]. Here they used a linear reward function and a policy optimization method with Monte-Carlo simulation. This method converged fast and performed well, slightly poorer than a baseline MPC. They were unable to converge with the DDPG algorithm with a reasonable policy.

A general proposed solution of quadrotor waypoint navigation using RL is found in this thesis from 2020 [35]. The thesis consists of solving for both the the attitude controller and position controller with a single RL controller. The goal is therefore to generate motor commands from sensory input. The proposed solution was to train the policy for two separate situations, one for getting to the waypoint and one for hoovering in the waypoint. The paper also goes into details, comparing different RL methods and using a dense reward function. Two functions were used, one penalizing position and attitude which managed to reach most waypoints,were plagued by oscillations. The other reward function had only position penalization and was less oscillatory.The paper shows that the TD3 algorithm works better for position control.

1.4.1 Reward Shaping

The importance of reward shaping has been a topic widely discussed in reinforcement learning. This paper [18] from 1999 discusses the topic as a crucial part of acquiring the optimal policy, and how RL tend

to converge slowly in complex tasks without appropriate reward shaping. They propose distance-based heuristics and dense rewards. A similar conclusion is found here [36] with a special consideration of how the reward shape can affect exploration of a large state space. Building on these papers are [37] and [38] that each propose an automatic method of designing a reward function avoiding the tedious hand crafting. The first paper consider an algorithm using knowledge of the cumulative rewards during learning to generate new reward shapes. The second paper using the domain of meta-learning to approximate a prior, which is the optimal shaping function over multiple tasks. This last paper also applied the method on the DDPG algorithm, showing promising results.

1.4.2 Imitation Learning

On the other hand, there are other solutions than optimizing the reward function that can yield better solutions. The solution proposed in [13] from 2019 shows promising results. Here they used imitation learning to train a network to predict actions matching that of an expert given the same state. The expert used was an MPC. Their proposed solution managed to imitate very similar behaviour at a fraction of the computational cost.

Following this path further is the paper [39]. It discusses the use of imitation learning, to train the actor to approximate the behaviour of an expert. After this initial learning DDPG was utilized with a sparse reward function allowing for the actor to outperform the expert. This method is utilized in several robotics environments. The paper found that a model trained by imitation learning first outperformed a trained from scratch with a more engineered reward function.

1.4.3 Inverse RL (IRL)

Combining the topics discussed above, with handcrafted/automatically induced reward function and imitation learning, is the topic of Inverse Reinforcement Learning.

Given some expert planner that can solve the RL problem with an optimal policy π^*. Implicitly, the behaviour of the expert details the reward function. Inverse RL seeks to find the reward function that explains the observed expert behaviour [40]. When the reward function is found, the policy can be calculated using traditional RL techniques. Considering [41] and [42], there are pros and cons regarding IRL. The pros seems to be generalizability, the agent reacts in a similar manner as the agent in unobserved states, this allows for safe learning in the same states. The existence of a very large set of reward functions that can lead to the expert performance is one of them. To combat this, heuristics can be used to find the most suitable reward function. This will of course lead to increased complexity. Nevertheless, inverse RL for a quadrotor system is implemented in this paper [43] and shows promising results. [44] proposes a novel method of decomposing the problem into subproblems and assigning each a local reward function. The method performs well in robotics tasks. It is reportedly not performing as well in state spaces of high dimensionality, which is an important aspect of quadrotor control.

Inverse RL is an interesting part of RL, and a good tool for keeping humans in the loop. The combination of imitation learning and IRL is often referred to as apprenticeship learning. Here the agent is first trained to imitate the expert and then trained further using the deduced reward function. This method is

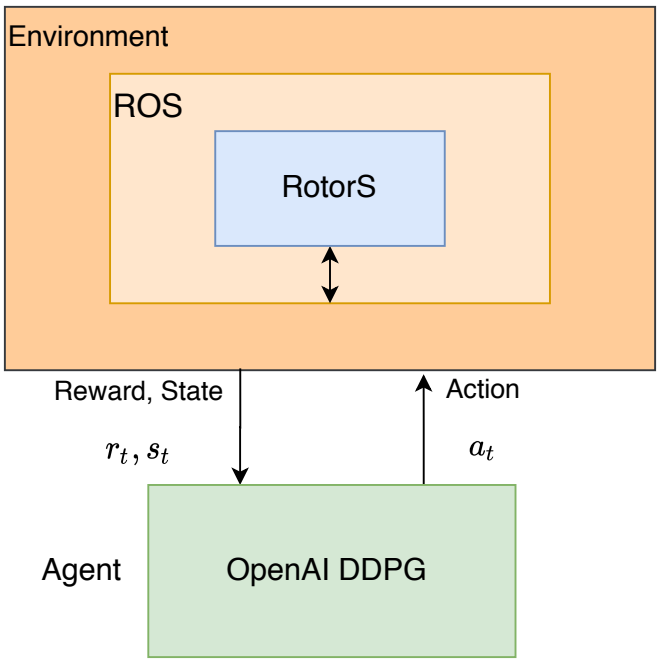


Figure 1.3: How the different software is structured. Comparable to figure 1.1.

found to be able to outperform human expert performance [45].

1.5 Experimental Setup

Several software tools were used to simulate the environment and drone, and to understand and utilize the knowledge incurred from literature. The tools are visualized in figure 1.3.

1.5.1 Robot Operating System (ROS)

For the simulation we used ROS version Melodic[46]. This is an open source framework for robot simulation software. It grants many features that are important for robotics such as message passing between modules, low-level component control and package management.

A ROS process consists of nodes and topics. It can be visualized in a graph with each sub-process as a node where the edges that connect them are different topics. The different nodes can publish or subscribe to different topics containing data. This allows for services between the node in the form of for example calculation, simulation or need to share data between other sub-processes.

The Rviz three dimensional visualization tool was utilized to render the simulation.

1.5.2 Gazebo and RotorS Simulator Environment

To run the physics simulation, Gazebo version 9.0.0[47] was used with RotorS. Gazebo is a three dimensional robot simulator program, as well as a sophisticated physics engine and rendering. Additionally,

it allows for advanced sensor and actuator simulation. This is a powerful tool allowing for safe, realistic simulation of robotic systems.

RotorS is an open-source Gazebo simulator for Micro Aerial Vehicles[48]. The simulator was developed by the Autonomous Systems Lab at the ETH university in Zürich. It allows for simulation of several multirotor models with a variety of relevant sensors and controllers. Adding sensors is an important objective of the future works for this project and therefor important aspect of the simulator. In this project, the Delta quadrotor model was used to train the models.

1.5.3 OpenAI

As means of exercising the methods and theory of RL, the framework of OpenAi Gym version 0.13[49] was used. OpenAi Gym is a toolkit specifically constructed for reinforcement learning research. The Gym has several problems that can be solved by RL methods and allows for open-source implementations of the different methods. In addition to using the Gym as an exercise for solving RL, the OpenAi Gym of the DDPG algorithm was used for the training of the models.

1.5.4 Tensorflow

In this project the open-source machine learning library Tensorflow version 2.0 [50] was used. Tensorflow is a comprehensible and uncomplicated library from Google specifically optimized for training and running neural networks models.

1.5.5 Parameters Kept Constant for the Models

The DDPG algorithm have a lot of parameters that can affect the dynamic of the drone. Here the discussion will be about some tuneable parameters we kept constant.

The network structure of the actor and critic were equal to each other and kept constant for each model. They had two hidden layers, the first with 128 units and the second with 64 units. The actor-critic network can be observed in figure 1.4. As both the actor and the critic is supposed to The topic of robustness is important when we discuss the choice of network. As the drone should be able to perform in a variety of environments and sensory uncertainties the trained model should be able to adapt to different circumstances. Models trained with deeper networks seemed to perform as well as 1.4, however it seemed to perform worse when testing the robustness of the system when for instance altering the weight of the quadrotor. This is a problem caused by overfitting as the model describes the training data with an overly complex function. This leads to wrong trajectories when tested on environments it was not specifically trained for. On the other hand, a network with less neurons performed worse as it was unable to approximate the dynamics of the system with sufficient quality. The work of Martin Aalby Svalesen titled "Robustness in Deep Reinforcement Learning for Quadrotor Control" details in depth this conundrum.

The activation function used for every layer was a rectified linear unit (ReLU). As the neuron is a linear combination of the inputs passed through an activation function. One of the important features of

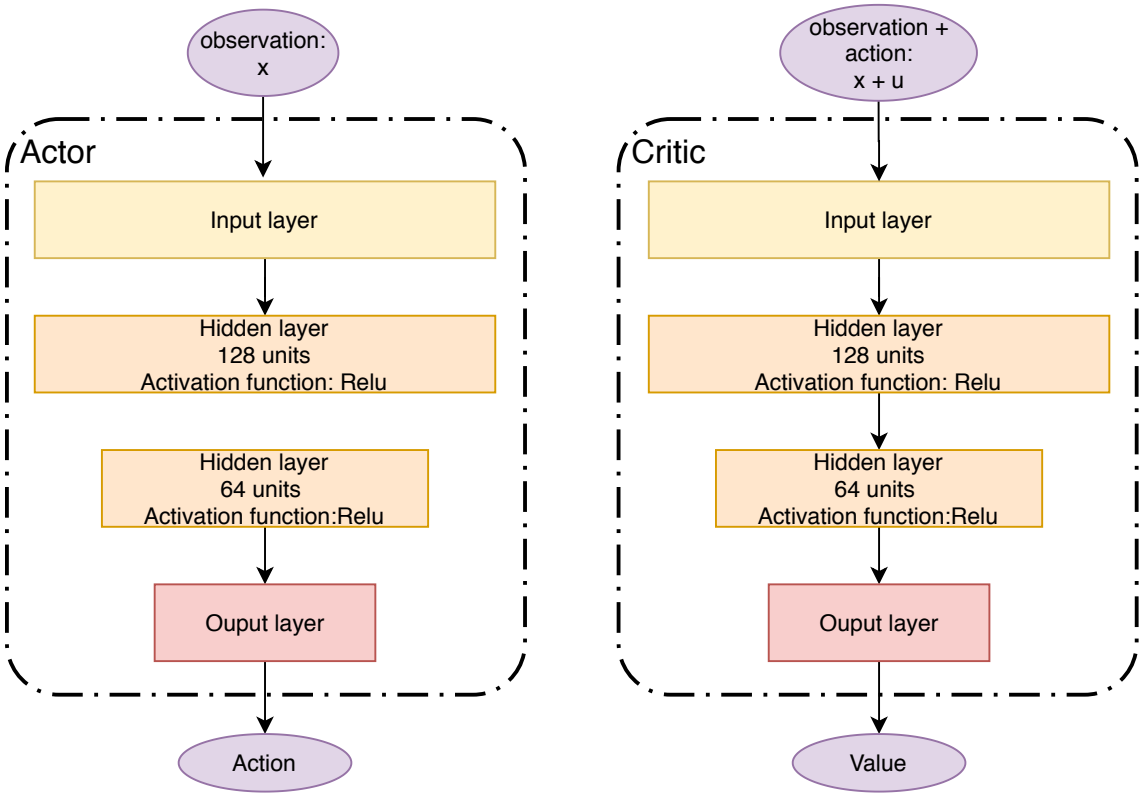


Figure 1.4: Visualization of the actor critic network. The two are identical except for the input. The observation x will vary with the reward function for the model. The action space is a vector with length 3.

the activation function is allowing for ReLU function, defined as $f(x) = max(x,0)$, allows for easier optimization than for instance the sigmoid or tanh functions[51]. This is attributed to the gradients being able to flow between the layers more easily. This makes the ReLU highly applicable in a variety of problems. It is known to approximate highly non linear functions very well. On the negative note, a problem of the ReLU is that, as it outputs zero for $x < 0$, many neurons will remain inactive. For instance a tanh activation function will not have the same problem, as it is active between $[-1,1]$. The tanh function is also relatively resilient for the *vanishing gradient* problem. For this reason it could be useful to try a tanh function, though this was not attempted in this project.

The noise process used for exploration was a zero mean Gaussian noise with standard deviation of 0.1. Considering the original DDPG paper[14], they applied the Ornstein-Uhlenbeck noise process, which has temporarily correlated noise. It is argued that in robotic systems affected by inertia, as in this example, a correlated noise will lead to a correlated exploration. Which they argue is better. This should have been investigated further but was left constant for the sake of reward shaping. The gamma parameter, referred to in section 1.2.1 as the discount factor of future rewards were 0.99.

Each model also had a collision penalty, but the drone could only collide with the ground. It was spawned at an altitude that made collision with the ground highly unlikely given the time frame. Therefore it was unconsidered for the reward shaping task.

1.6 Method

The task of this project was to alter the components and weight of the reward function. The different proposed components of the reward function was:

- Goal reward: Reward for being within the goal radius.
- Position: The difference between the goal and quadcopter.
- Input: The applied thrust, pitch and roll.
- Linear velocity: the norm of the linear velocity.
- ΔInput: The difference between the applied action of last time step and the action of this step.
- Quadratic or linear reward function

To begin, a baseline model was trained with the intention of comparing its dynamic with the other models dynamic. Its reward function was quadratic and consisted of position, input, linear velocity, Δinput and goal reward. The other models are to be discussed as different variants of this baseline.

$$reward = \begin{cases} -u^T Ru - \alpha\Delta u + \text{goal reward} & \text{When in goal sphere} \\ -u^T Ru - \alpha\Delta u - x^T Qx & \text{Otherwise} \end{cases} \tag{1.44}$$

Consisting of the different paramteres:

$$Q = \begin{pmatrix} q_{x_{pos}} & 0 & 0 & 0 & 0 & 0 \\ 0 & q_{y_{pos}} & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{z_{pos}} & 0 & 0 & 0 \\ 0 & 0 & 0 & q_{x_{vel}} & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{y_{vel}} & 0 \\ 0 & 0 & 0 & 0 & 0 & q_{z_{vel}} \end{pmatrix} \quad , \quad x = \begin{pmatrix} position \\ velocity \end{pmatrix} = \begin{pmatrix} x_{pos} \\ y_{pos} \\ z_{pos} \\ V_x \\ V_y \\ V_z \end{pmatrix} \tag{1.45}$$

$$R = \begin{pmatrix} r_{pitch} & 0 & 0 \\ 0 & r_{roll} & 0 \\ 0 & 0 & r_{thrust} \end{pmatrix} \quad , \quad u = \begin{pmatrix} u_{pitch} \\ u_{roll} \\ u_{thrust} \end{pmatrix} \quad , \quad \Delta u = ||u_t|| - ||u_{t-1}|| \tag{1.46}$$

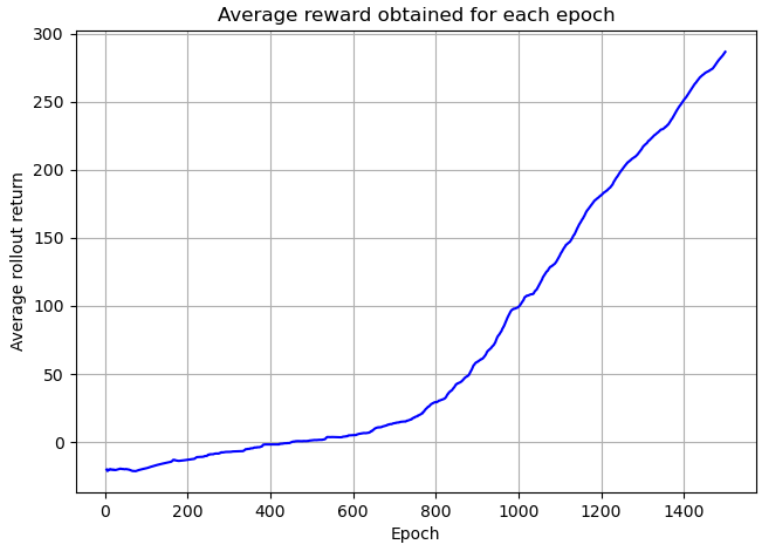
Both the Q and R are diagonal, positive definite matrices with real values. The x vector consists of the difference in position in each axis between the drone and the goal. These are the three first states. The three last are the linear velocities of the drone for each axis. The goal is static for each run and therefore the goal speed is not present.

The three actions in u are pitch, roll and thrust. Note that yaw is not part of the action, although a quadcopter easily can control the yaw. In this topic we only care about controlling the position of the drone, it is therefore deemed unnecessary for this topic. Solving for 3 dimensions with 4 parameters

leaves infinite solutions, yaw can be seen as a free variable in the context of linear algebra. Therefore it can be controlled by a decoupled attitude controller as seen in [52].

$\alpha\Delta u$ is the linear penalization of difference in input for this time step u_t and the last time step u_{t-1}. $\alpha > 0$ is a scalar weight. The individual weights of the Q and R matricies and α was found through training the model observing the dynamic of the drone through the rviz software and the plots generated. If the dynamic was unstable or failed to come significantly close to the goal the training was stopped and the model discarded.

After training several models with the same reward function for 3 million steps each dynamic was evaluated after its ability to reach and remain in the target. The best model was then chosen for further training. This model was then trained 2 million steps with a smaller learning rate for the two neural networks and smaller goal radius to converge better toward the goal point. For the training, the plot of the reward gained for each epoch provided by the Tensorboard software as pictured in 1.6 would be observed. The figure is an just an example. Here the returned reward has not converged yet and should be trained further for a better reward. It is proposed in [53] that reducing the learning rate for stochastic gradient ascent can be attributed to better convergence and so a small reduction was applied, further discussed in results. The reward function was kept constant but the parameters: learning rate and goal radius was reduced. Reduction of the goal radius was to ensure higher precision from the drone. Using this type of plot we could determine whether to train more or choose a particular epoch that performed well as the final model.



The proposed method was to go through three different reward functions to determine their dynamic. These were:

- The baseline: quadratic reward function with position, input, linear velocity, Δinput and goal reward.
- Without velocity: quadratic function with position, input, Δinput and goal reward.

- Linear function: Linear reward function with position, linear velocity, input Δinput and goal reward.

After one model of a specific reward function was trained, the reward function was altered to one of the other planned models and the process started again.

For the chosen model, ten different goal coordinates and the robot spawning point were set for the testing, as seen in the table underneath. These were identical for each of the models. The points are in three dimensional space and the unit of measure is meters.

Goal 1	Goal 2	Goal 3	Goal 4	Goal 5	Goal 6	Goal 7	Goal 8	Goal 9	Goal 10
[1,2,10]	[-2,1,9]	[3.5,1,8]	[4,1,8]	[-1,-3,8.5]	[2,1,7]	[-3,0,6]	[1,1,6]	[0,-4,9]	[2,2,10]

Drone spawn point
[0,0,8]

The root-mean-square error between the trajectory and the euclidean distance between the drone spawning point and the goal was calculated and saved for each goal point. These will be presented in the section 2.

As the training often took several hours to complete and the amount of parameters were high this process was very time and resource consuming and the amount of models were limited by this process. More reward functions were trained than in the list above. These did not perform well enough to be able to compare it with the other models here. This will be discussed further in the results of section 2.

1.7 Outline

The rest of the report is structured as follows.

Chapter 2 presents the results obtained from the different models by the metrics discussed in section 1.3.3. Here the details of the weights and shape of the cost function is elaborated on for each model.

Chapter 3 will discuss the results from chapter 2 along with their similarities and differences. Then the proposed future work will be discussed.

Chapter 2

Preliminary results and plots

Here the plots of the trajectories of the different models are to be presented along with their RMS values. We will provide three different models as discussed in section 1.6.

In the performance results for each of the models we will plot the trajectories of goal point 2, 4, 6 and 10 as these seemed to show the different dynamics of the models.

2.1 The Models

2.1.1 Model 1: Baseline

Reward Function and Training

This model was trained using the following reward function.

$$reward = \begin{cases} -u^T Ru - \alpha||\Delta u|| + \text{goal reward} & \text{When in goal sphere} \\ -u^T Ru - \alpha\Delta u - x^T Qx & \text{Otherwise} \end{cases} \tag{2.1}$$

Consisting of the different values:

$$Q = \frac{1}{250}\begin{pmatrix} 1.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 \end{pmatrix} \quad , \quad x = \begin{pmatrix} x_{pos} \\ y_{pos} \\ z_{pos} \\ V_x \\ V_y \\ V_z \end{pmatrix} \tag{2.2}$$

$$R = \frac{1}{10^6}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad , \quad u = \begin{pmatrix} u_{pitch} \\ u_{roll} \\ u_{thrust} \end{pmatrix} \quad , \quad \alpha = \frac{1}{250} \quad . \quad \Delta u = ||u_t|| - ||u_{t-1}|| \tag{2.3}$$

It was trained with a learning rate of 10^{-4} and goal radius of 0.4 meters for 3 million steps. It was then

trained further for 2 million steps with a smaller learning rate of 10^{-5} and goal radius of 0.25 meters to ensure better convergence [53]. This was trained until convergence of the neural networks.

Performance

Below is the RMS values for each of the different goal. The average score and standard deviation of the data set is shown in the table under.

Goal 1	Goal 2	Goal 3	Goal 4	Goal 5	Goal 6	Goal 7	Goal 8	Goal 9	Goal 10
0.15m	0.17m	0.23m	0.25m	0.26m	0.20m	0.19m	0.24m	0.38m	0.16m

Average	Std. Dev.
0.22m	0.067m

Here the trajectories for each of the different goal points are plotted in 3D space are presented. This is shown in figure 2.1. The trajectory of the drone is plotted next with the euclidean optimal distance between the spawning point and the goal.

The trajectories deviates a bit from the optimal path as observed in 2.1. Decomposing the trajectories seen in figure 2.1 gives us the plots in figure 2.2. These are easier to analyze. Consider the scale of the z-axis in goal 4 in figure 2.1, which makes it appear as a much poorer performance than the other goals. Considering the same goal in 2.2 makes it apparent that it performs closely to the other trajectories. The plots express the behaviour of the drone as a second order response converging to the goal point. In general this model reaches the goal radius within 2-3 seconds for most of the goal points. It shows tendencies of overshooting in goal 4 and and 6 for the y axis, in the other axis the system seemed closer to a critically dampened dynamic. There is a clear steady state error in all trajectories of a varying degree.

2.1.2 Model 2: No Velocity

Reward Function and Training

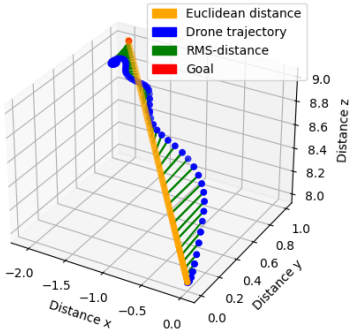
Here we removed the velocity of the baseline reward function as seen in equation 2.4.

$$reward = \begin{cases} -u^T Ru - \alpha||\Delta u|| + \text{goal reward} & \text{When in goal sphere} \\ -u^T Ru - \alpha||\Delta u|| - x^T Qx & \text{Otherwise} \end{cases} \tag{2.4}$$

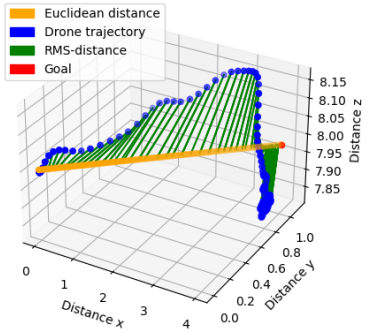
The equation consisted of the different values:

$$Q = \frac{1}{250}\begin{pmatrix} 1.2 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad , \quad x = \begin{pmatrix} x_{pos} \\ y_{pos} \\ z_{pos} \end{pmatrix} \tag{2.5}$$

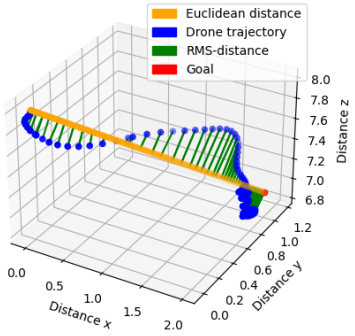
$$R = \frac{1}{10^6}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad , \quad u = \begin{pmatrix} u_{pitch} \\ u_{roll} \\ u_{thrust} \end{pmatrix} \quad , \quad \alpha = \frac{1}{250} \quad , \quad \Delta u = ||u_t|| - ||u_{t-1}|| \tag{2.6}$$



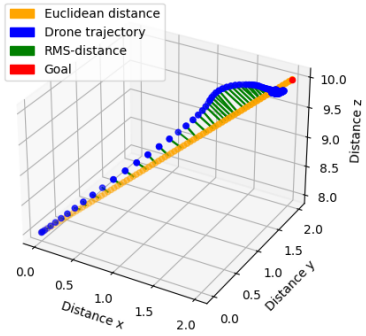
(a) Trajectory for goal 2



(b) Trajectory for goal 4

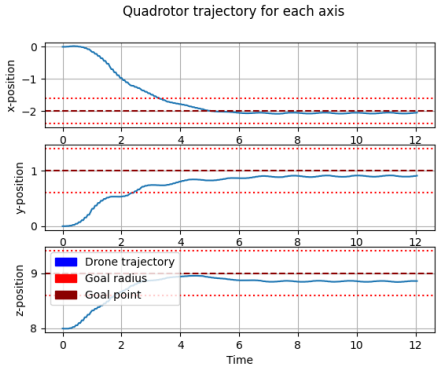


(c) Trajectory for goal 6

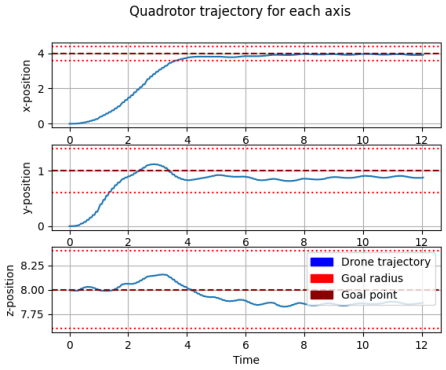


(d) Trajectory for goal 10

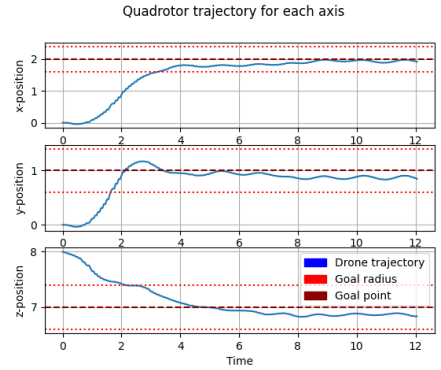
Figure 2.1: Trajectory baseline model for goal points 2, 4, 6 and 10. The blue dots are the time steps of the drone trajectory. The orange line is the optimal distance. The red dot is the goal point and the green lines are the calculated RMS distances between the trajectory and the optimal distance.



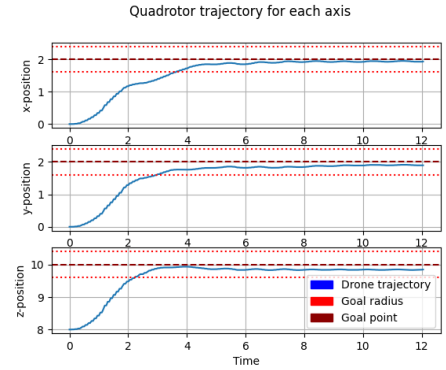
(a) Second order response for goal 2



(b) Second order response for goal 4



(c) Second order response for goal 6



(d) Second order response for goal 10

Figure 2.2: Baseline model second order response plotted for each axis for the goal points 5 and 10. The blue line is the drone trajectory. The dark red dashed line is the goal point and the lighter red dotted line is the goal point ± the goal radius.

This model was also trained for 3 million time steps with learning rate 10^{-4} and goal radius of 0.4 meters. This one converged during the initial 3 million steps further training was attempted with smaller learning rate and goal radius of 0.25 meters, but this only resulted in worse results.

Performance

Below is the RMS valued for the 10 trajectories. The average and standard deviation of the data set is in the table underneath.

Goal 1	Goal 2	Goal 3	Goal 4	Goal 5	Goal 6	Goal 7	Goal 8	Goal 9	Goal 10
0.36m	0.47m	0.35m	0.40m	0.54m	0.39m	0.27m	0.38m	0.76m	0.45m

Average	Std. Dev.
0.43m	0.135m

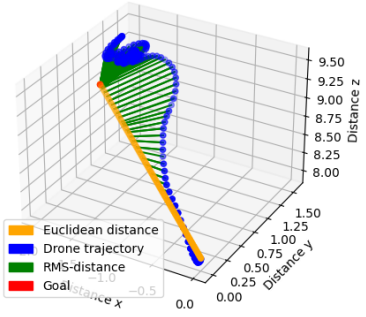
The trajectories of the no velocity model were plotted with the optimal euclidean distance in figure 2.3. These trajectory were then decomposed for each axis resulting in the plots seen in 2.4. By visual inspection it is apparent that the model is performing poorly. The deviation from the optimal path is large and hazardous. Considering the goal in 4,the model manages to deviate from the optimal path twice in a limited time frame, showing poor behaviour. This model also manages to come out of the goal on several occasions, which is not a desired dynamic trait. As observed in figure 2.4, the dynamic is characterized by oscillatory behaviour and a time constant of about 2-4 seconds. Seemingly it rises a bit slower than the baseline, though not necessarily for every goal. The standard deviation of this model is significantly higher than the baseline. This is unwanted and expresses further the suboptimality of the policy.

2.1.3 Model 3: Linear Reward function

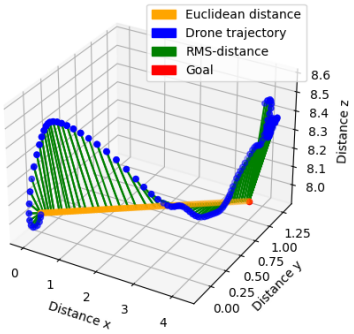
This model is based on the linear reward shape and weights are from this paper [34] which shows great potential for drone control with RL. In our model, the angular velocity was removed and the $\Delta||u||$ penalty was added. This was done in hopes of getting a more comparable dynamic with the baseline model. The method in the paper was not DDPG.

$$reward = \begin{cases} -2\cdot 10^{-4}||u|| - \alpha||\Delta u|| + \text{goal reward} & \text{When in goal sphere} \\ -4\cdot 10^{-3}||x_{pos}|| - 2\cdot 10^{-4}||u|| - \alpha||\Delta u|| - 5\cdot 10^{-4}||x_{velocity}|| & \text{Otherwise} \end{cases} \tag{2.7}$$

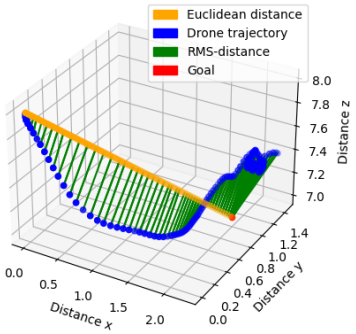
$$x_{position} = \begin{pmatrix} x_{pos} \\ y_{pos} \\ z_{pos} \end{pmatrix} \quad , \quad x_{velocity} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} \quad , \quad u = \begin{pmatrix} u_{pitch} \\ u_{roll} \\ u_{thrust} \end{pmatrix} \tag{2.8}$$



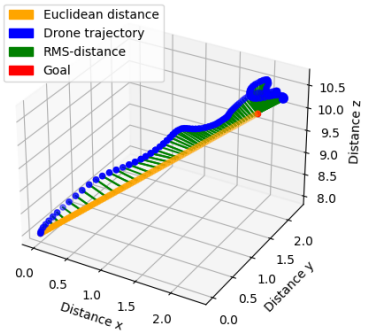
(a) Trajectory for goal 2



(b) Trajectory for goal 4

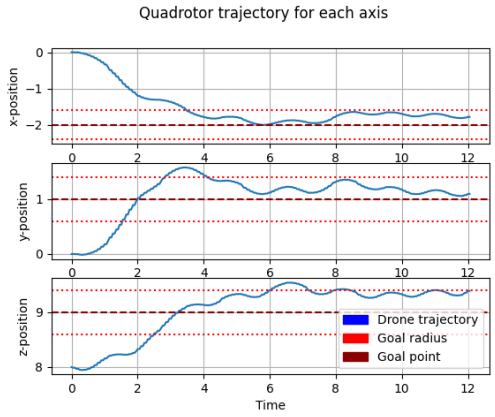


(c) Trajectory for goal 6

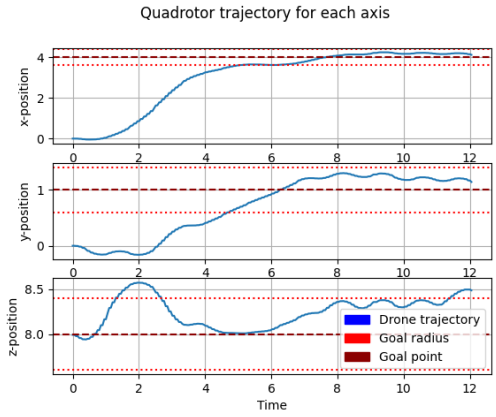


(d) Trajectory for goal 10

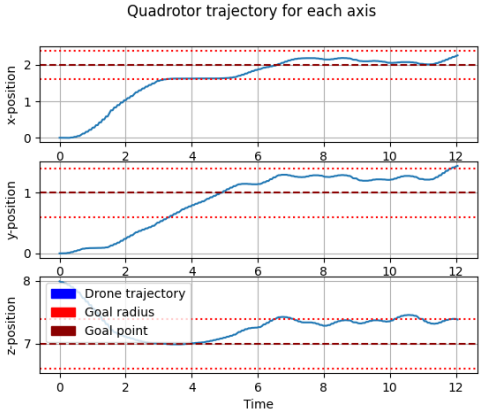
Figure 2.3: Trajectory of model without velocity penalization for goal points 2, 4, 6 and 10. The blue dots are the time steps of the drone trajectory. The orange line is the optimal distance. The red dot is the goal point and the green lines are the calculated RMS distances between the trajectory and the optimal distance



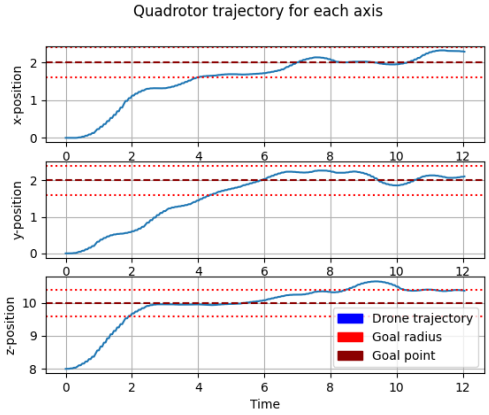
(a) Second order response for goal 2



(b) Second order response for goal 4



(c) Second order response for goal 6



(d) Second order response for goal 10

Figure 2.4: Model without velocity penalization second order response plotted for each axis for the goal points 2, 4, 6 and 10. The blue line is the drone trajectory. The dark red dashed line is the goal point and the lighter red dotted line is the goal point ± the goal radius.

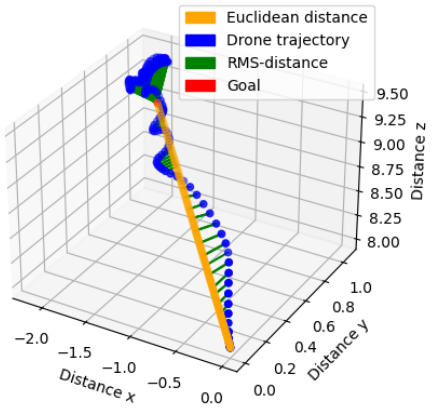
Performance

The RMS values of the linear model is in the table underneath. The average and standard deviation of the data set is shown in the lower table.

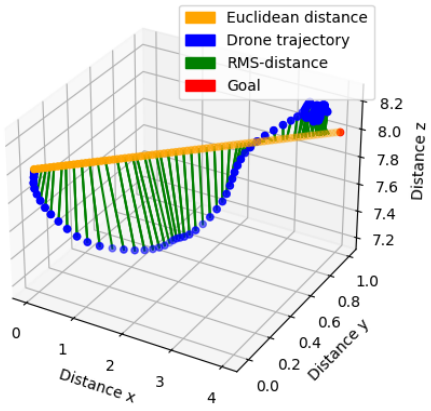
Goal 1	Goal 2	Goal 3	Goal 4	Goal 5	Goal 6	Goal 7	Goal 8	Goal 9	Goal 10
0.25m	0.29m	0.37m	0.33m	0.35m	0.27m	0.27m	0.11m	0.30m	0.33m

Average	Std. Dev.
0.28m	0.073m

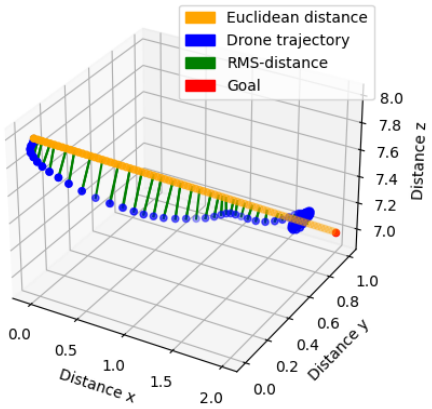
The trajectories of goal 2, 4, 6 and 10 is plotted in figure 2.5. These trajectory were then decomposed for each axis resulting in the plots 2.6. By visual inspection we observe a time constant of about 2-3 seconds. It dampens well and but overshoots a little in for example goal 2. The trajectories are closer to the optimal than the no velocity model. It seems prone to oscillations as well and with some cases of falling out of the goal zone. Still it stabilizes almost as well as the baseline model, albeit with a significant steady state error. The linear model performs with less variation than the no velocity, which is a good trait as dependability is wanted. Both the average and standard deviation of the RMS are closer to the baseline than the .



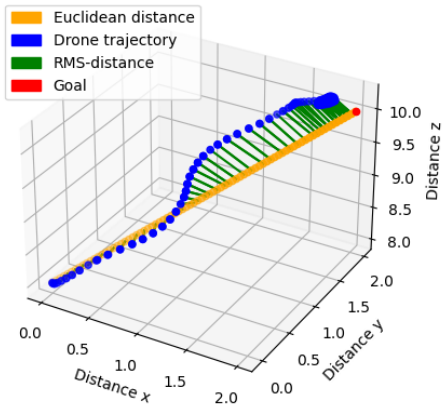
(a) Trajectory for goal 2



(b) Trajectory for goal 4

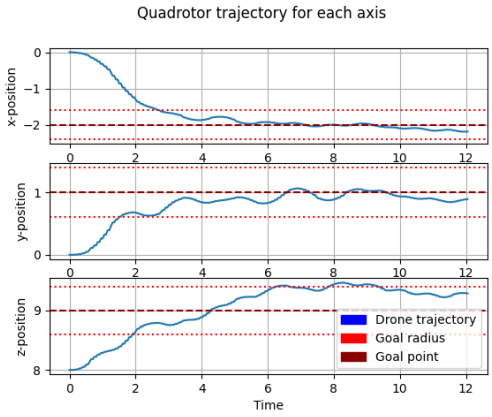


(c) Trajectory for goal 6

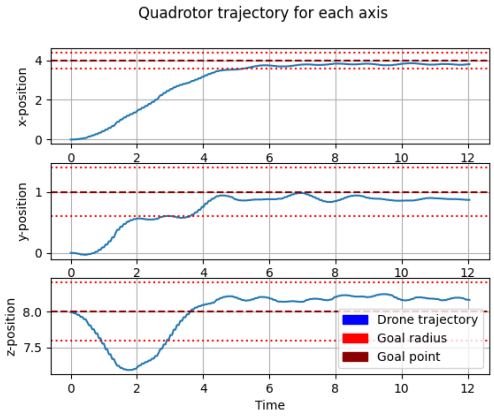


(d) Trajectory for goal 10

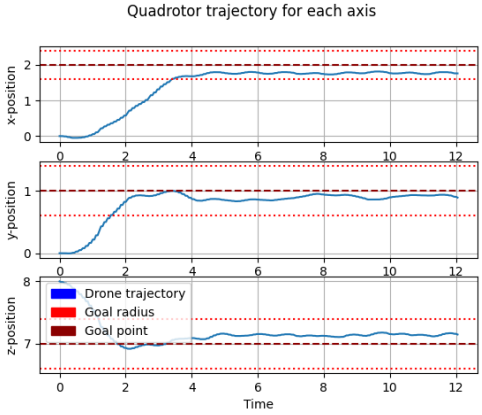
Figure 2.5: Trajectory of model with linear reward function for goal points 2, 4, 6 and 10. The blue dots are the time steps of the drone trajectory. The orange line is the optimal distance. The red dot is the goal point and the green lines are the calculated RMS distances between the trajectory and the optimal distance



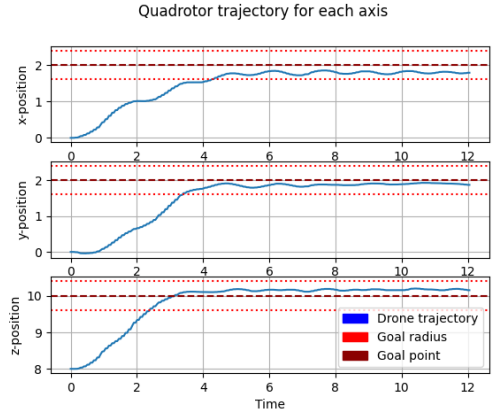
(a) Second order response for goal 2



(b) Second order response for goal 4



(c) Second order response for goal 6



(d) Second order response for goal 10

Figure 2.6: Model Linear reward function second order response plotted for each axis for the goal points 2, 4, 6 and 10. The blue line is the drone trajectory. The dark red dashed line is the goal point and the lighter red dotted line is the goal point ± the goal radius.

Chapter 3

Conclusions, Discussion, and Further Work

3.1 Summary

In this section we will discuss the different dynamics of the three models. Firstly, the overall performance of the three models will be compared with the desired dynamics objectives as discussed in section 1.3.3. Then the same performances will be compared to each other and in context of their reward functions to understand how their different reward shapes can have resulted in the particular dynamic.

Furthermore, we will discuss alternative approaches for reaching better dynamics that differ or supplement reward shaping.

3.1.1 Overall Dynamic of the Models

The dynamic of each model is unfortunately rather poor. As described in section 1.3.3, the desired dynamic of the drone is to reach the waypoint with little oscillations and deviation from the optimal path. All three models deviate from the euclidean distance by a significant amount as observed in 2.1, 2.3 and 2.5. This is unacceptable as it will use more time and resources, and risk potential collisions. From the plots 2.2, 2.4 and 2.6 oscillations are clearly observed in the trajectory, especially in the goal point, where it should hoover precisely. This is present in all three models and is an unwanted dynamic.

In addition, the time to reach the reference point of the closed-loop system is observed to be around 2-4 seconds, generally for the models. A quadrotor is notorious for its agility as discussed in the motivation 1.1. An average speed of about 1 m/s is not to be considered quick-responsive. This is an important aspect as the initial intention of a learning based method was to optimize the calculations. These calculations are on the scale of milliseconds[13]. Ensuring a quick response from a learning based solution is therefore crucial for it to compete with other controllers.

All three models manage to reach all 10 goal points. However, this is mostly because the goal point radius is 0.4m, meaning that the drone should be able to reach within almost a meter wide sphere. This type of performance in the discussed later goal of obstacle avoidance in narrow environments is impracticable. None of the models could have been utilized as they would require higher precision.

From the same plots it is also apparent that the models are prone to steady state errors. None of the trajectories observed reaches the goal point without a steady state error. This is a recurring problem in

reinforcement learning algorithms[34] and will be discussed later. This difference is probably not caused by the reward function, however it could be useful to be aware of the potential when constructing the model, if the steady state error is not entirely removed.

Thoughts on the Poor Performance

The question arises of all the trained models managed to converge to such clear suboptimal solutions to the problem. In [13] they used imitation learning to solve for quadrotor control in a physically constrained environment of much higher complexity. This paper shows that there exists points in policy parameter space that allow for an efficient mapping between state and action, which would solve the problem. In theory it should therefore be entirely feasible for such a solution here as well, but why do we not converge to it?

As discussed in the background information, RL is based on a combination of exploring the state space and exploiting the knowledge gained by exploring. Limits in time and computational power can lead to an insufficient search through the state-action space. The models from ML are only as good as the data they train on. This means that the poor performance is certainly caused by a lack of data or poor data distribution caused by exploiting suboptimal solutions. Considering the amount of time and failed models it took to train a model that worked to this degree. makes it hard to state which dynamics are related to the lack, or poor distribution, of data and which are related to the reward function.

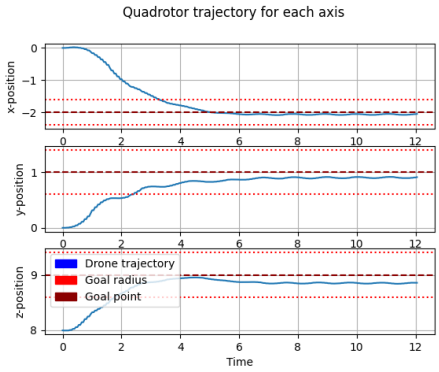
3.1.2 Difference in Dynamics

Trying to differentiate the different dynamics between the models is a difficult task. We will start by inspecting the average and standard deviations of the RMS values for each of the different models. This is observed in the table below.

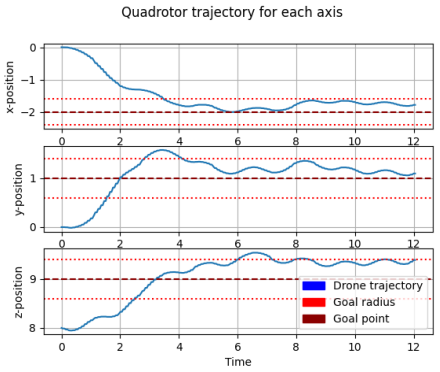
Baseline		No Velocity		Linear	
Average	Std. Dev	Average	Std. Dev.	Average	Std. Dev.
0.22	0.067	0.43	0.135	0.28	0.073

From this result it is apparent that the baseline model has a trajectory closer to the euclidean distance than the two other models, which is one of the criteria set for the desired dynamic. Supplementing on this is the trajectories in figures 2.1, 2.3 and 2.5. We observe from here a couple of characteristics. First, the no velocity model has a tendency to overshoot and especially for goal point 2 in figure 2.4 resembles an under-damped response. Second, it seems to have a slightly higher rise time than the baseline.

This would make sense regarding the reward function, as the only state we penalize is the difference in position between the robot and the goal, while the baseline also penalizes velocity. Considering just the state penalization, we can look at the no velocity model as a proportional controller of the state and the baseline as a Proportional- Derivative controller. It is well known that adding a derivative term will reduce oscillations [54]. It is viable to address the tendency of larger oscillations and overshooting of the no velocity model to the structure of the reward function.



(a) Baseline trajectory for goal 2



(b) No velocity model trajectory for goal 2

The model with the linear reward function is very similar to the baseline. It appears a bit more oscillatory and slower in its response compared to the baseline. But these characteristics are less protruding compared with the no velocity model and it is hard to gauge if the differences is attributed to the reward shape or the difficulties regarding exploration of the state-action space.

The No Velocity model also has significantly higher standard deviation than both the other models. This is highly unwanted.

In general, the baseline model can be observed to perform better for the waypoints generated in this task. The effect of not penalizing the velocity seems apparent when comparing the two models. Still, while the linear model performed worse, it is hard to conclude if the reason is the reward shape or something else. Probably, the cause is a fault in some parameters in the DDPG algorithm or the algorithm itself. Considering the paper [34] where the DDPG failed altogether in finding a decent enough policy given the training time. Other reasons will be discussed further in proposed future works.

3.2 Conclusions

In this work the current status of reinforcement learning in robotics, and in particular drone control has been elaborated on. Three quadcopter models for waypoint navigation in 3D have been trained and analyzed, focusing on the impact of their differing reward functions in their dynamic. The models have performed poorly for the tasks. This poor performance is not believed attributable to the reward function.

3.3 Discussion

The steady state error in continuous state and action space is a well documented problem as stated by this paper [55], where they implement an integral compensator for the DDPG algorithm to remove the steady state error. Slowly accumulating the steady state error is tested vastly in control theory with the PID controller and should be a valid solution to the steady-state error.

As seen in the results there are severe difficulties when engineering the reward function of a robotic problem in continuous state and action spaces. Here we have attempted to find a specific shape of the reward function to allow for efficient exploration of the vast state-action space. The shape have to allow

for both exploration and finding the best policy. This problem has here shown to be quite difficult to solve by the reward function alone. Further, There are other aspects than the reward function that can be adjusted to attempt to achieve better performance. In this discussion we will, in most part, discuss other methods of solving the problem of reinforcement learning in quadcopter waypoint navigation other than hand-crafting the reward function.

3.4 Proposed Future Work

A proposal could be to split the problem into a hoovering task and a waypoint task as shown here [35]. As these are two different dynamics they could be decoupled and trained specifically.

Imitation Learning

As discussed in related works, the topic of quadcopter control has been successfully solved by traditional control methods like the LQR and MPC before, with a downside of high computational cost in collision risked environments. As discussed in relevant works, consider the paper of a learning based approach using a graph-based planner MPC as an expert and utilizing imitation learning to train the agent [13]. Here they successfully managed to find a policy achieving efficient planning at a fraction of the computational cost of the MPC. Building on these principles, a similar approach with an adjustment is tested in [39], solving robotic tasks with DDPG. Their proposed solution is also to use an expert and imitation learning to train the networks. After this initiating training periode, the network can be trained more with a sparse reward function to further optimize the policy. An inherent problem in RL is that the agent is limited in performance to its own ability to explore the state-action space. Effectively, the solution proposed in [39] reduces the exploration space and then explores from there. This allows for finding a better policy. The paper concludes that a DDPG network trained with supervised learning outperforms the regular DDPG. Combining these two methods by using imitation learning and a sparse reward is a proposed future work that seems to be more valuable than engineering the reward function. It should be noted that this is the method used for the successful *AlphaGo* chess bot described in 1.1. By this method, we have some guarantee of the quality of the data the policy network will be exposed to. When using such a method, the MPC should run offline to ensure that the slow speed from the extensive computation is not propagated to the agent.

Inverse RL and Imitation Learning

Inverse RL as discussed in related works, could show some potential to hand-crafting the reward function. The combination of the IRL and imitation learning seems particularly interesting. Imitation learning could be used to acquire some baseline policy such that further training lets it converge easier to an even more optimal policy. Then a reward function found by an IRL method essentially representing the experts prioritizing could be used to efficiently generate data with efficiency that limits imitation learning. This could be useful for generalization and robustness of the algorithm. With only imitation learning, the agent is limited by the performance of the expert. By using the reward function of the expert, the agent

can outperform it. In this case, a trust region method should be used to ensure that the policy does not falter during further training.

Performing Rollouts

The MPC is celebrated, in part, for its ability to optimize over some finite horizon. This is one of the key differences when comparing the MPC with the DDPG algorithm. In this project, the horizon of the DDPG algorithm is of a single time step. As the method in this work has performed much worse than an MPC, extending the horizon is an interesting topic of further inspection. Performing rollouts in some way to extend the horizon for the current policy should be prioritized for future work.

General Thoughts for Future Works

In general, PPO seems to be a 'more promising algorithm than DDPG. Especially it seems like DDPG is plagued by finding sub-optimal solutions due to poor gradient updates. The general scientific community seem to have moved away from the DDPG method and towards other methods as discussed here [34]. Further work should consider using a trust region method like PPO, or some more state-of-the-art method like the TD3.

All considered, the topic of hand-crafting reward shapes seems to bear little fruit compared to other methods that can be applied to the problem. Future works should focus on either using a process to determine the reward function, for example reward shaping automation [37] or IRL, or using imitation learning to circumvent the large state-action space and with further training using RL. In this way a better solution can be found while wasting less time on engineering the reward function by hand.

Appendix A

Acronyms

UAV Unmanned Aerial Vehicle

RL Reinforcement Learning

MDP Markov Decision Process

DP Dynamic Programming

RMS Root-Mean-Square

DDPG Deep Deterministic Policy Gradient

PPO Proximal Policy Optimization

NN Neural Networks

DRL Deep Reinforcement Learning

TRPO Trust Region Policy Optimization

ROS Robot Operating System

TDL Temporal Difference Learning

DQN Deep Q-Learning

ReLU Recitified Linear Unit

IRL Inverse Reinforcement Learning

A.1 Algorithms

A.1.1 Deep Deterministic Policy Gradient

Algorithm 0: DDPG

Initialize randomly critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|'theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$ Initialize replay buffer R

for *episode = 1, M* **do**

- Initialize a random process N for action exploration;
- Receive initial observation state s_1;
- **for** *t = 1, T* **do**
  - Select action $a_t = \mu(s_t|\theta^\mu + \mathcal{N}_t$ according to the current policy and exploration noise;
  - Execute action a_t and observe reward r_t and observe new state s_{t+1};
  - Store transition (s_t, a_t, r_t, s_{t+1}) in R;
  - Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R ;
  - Set $y_i = r_i + \gamma Q'(s_{i+1}, \pi(s_{i+1})|\theta^{\mu'})|\theta^{Q'})$;
  - Update critic by minimizing the loss: $L_t(\theta^Q) = \frac{1}{N}\sum[y_i - Q(s_i, a_i|\theta^Q)]^2$;
  - Update the actor policy using the sampled policy gradient:
  - $\nabla_{\theta^\mu} \approx \frac{1}{N}\sum \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$;
  - Update the target networks:;
  - $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q\prime}$;
  - $\theta^{\mu'} \leftarrow \theta^\mu + (1-\tau)\theta^{\mu'}$;
- **end**

end

A.1.2 Trust Region Policy Optimization

Algorithm 1: TRPO

Input: initial policy parameters θ_0, initial value function parameters ϕ_0 ;

Hyperparameters: KL-divegence limit δ, backtracking coefficient α, maximum number of backtracking steps K ;

for $k = 0,1,2 \ldots$ *until convergence* **do**

Collect set of trajectories $D_k = \tau_i$ by running policy $\pi_k = \pi(\theta_k)$ in the environment ;

Compute rewards-to-go $\hat{R}_t$ Compute advantage values $\hat{A}_t$ based on the value function V_{ϕ_k};

Estimate the policy gradient as

$$\hat{g}_k = \frac{1}{|D_k|} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$$

Use the conjugate gradient algorithm to compete

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k$$

,

Where $\hat{H}_k^{-1}$ is the Hessian of the sample average KL-divergence;

Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k \hat{H}_k \hat{x}_k}} \hat{x}_k$$

,

Where $j \in 0, 1, 2, \ldots K$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint;

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k| T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left[V_\phi(s_t) - \hat{R}_t \right]^2$$

via some gradient descent algorithm

end

A.1.3 Proximal Policy Optimization

Algorithm 2: PPO

Input: initial policy parameters θ_0, initial value function parameters ϕ_0 ;

for $k = 0,1,2 \ldots$ **do**

Collect set of trajectories $D_k = \tau_i$ by running policy $\pi_k = \pi(\theta_k)$ in the environment ;

Compute rewards-to-go $\hat{R}_t$ Compute advantage values $\hat{A}_t$ based on the value function V_{ϕ_k};

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right]$$

via some stochastic gradient ascent, typically Adam;

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left[V_\phi(s_t) - \hat{R}_t \right]^2$$

via some gradient descent algorithm

end

Bibliography

[1] Autonomous robots lab. URL https://www.autonomousrobotslab.com/.

[2] Darpa subterranean challenge. URL https://www.subtchallenge.com/.

[3] Patrick Doherty and Piotr Rudol. A uav search and rescue scenario with human body detection and geolocalization. In Mehmet A. Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence*, pages 1–13, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76928-6.

[4] D. Orfanus, E. P. de Freitas, and F. Eliassen. Self-organization as a supporting paradigm for military uav relay networks. *IEEE Communications Letters*, 20(4):804–807, 2016. doi: 10.1109/LCOMM.2016.2524405.

[5] T. Ahilan, V. A. Adityan, and S. Kailash. Efficient utilization of unmanned aerial vehicle (uav) for fishing through surveillance for fishermen. *World Academy of Science, Engineering and Technology, International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering*, 9:1468–1471, 2015.

[6] K. A. Ghamry, M. A. Kamel, and Y. Zhang. Multiple uavs in forest fire fighting mission using particle swarm optimization. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1404–1409, 2017. doi: 10.1109/ICUAS.2017.7991527.

[7] Chi Yuan, Youmin Zhang, and Zhixiang Liu. A survey on technologies for automatic forest fire monitoring, detection, and fighting using unmanned aerial vehicles and remote sensing techniques. *Canadian Journal of Forest Research*, 45(7):783–792, 2015. doi: 10.1139/cjfr-2014-0347. URL https://doi.org/10.1139/cjfr-2014-0347.

[8] Sergio Bemposta Rosende, Javier Sánchez-Soriano, Carlos Quiterio Gómez Muñoz, and Javier Fernández Andrés. Remote Management Architecture of UAV Fleets for Maintenance, Surveillance, and Security Tasks in Solar Power Plants. *Energies*, 13(21):1–23, November 2020. URL https://ideas.repec.org/a/gam/jeners/v13y2020i21p5712-d438587.html.

[9] Alexander V Koldaev. Non-military uav applications. In *Aero India International Seminar-2007 Edition. Bangalore*. Citeseer, 2007.

[10] Bin Fan, Jia Sun, and Yao Yu. A lqr controller for a quadrotor: Design and experiment. pages 81–86, 11 2016. doi: 10.1109/YAC.2016.7804869.

[11] M Islam, M Okasha, and M M Idres. Dynamics and control of quadcopter using linear model predictive control approach. *IOP Conference Series: Materials Science and Engineering*, 270:012007, dec 2017. doi: 10.1088/1757-899x/270/1/012007. URL https://doi.org/10.1088%2F1757-899x%2F270%2F1%2F012007.

[12] K. Alexis, G. Nikolakopoulos, and A. Tzes. On trajectory tracking model predictive control of an unmanned quadrotor helicopter subject to aerodynamic disturbances. *Asian Journal of Control*, 16 (1):209–224, 2014. doi: https://doi.org/10.1002/asjc.587. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/asjc.587.

[13] Russell Reinhart, Tung Dang, Emily M. Hand, Christos Papachristos, and Kostas Alexis. Learning-based path planning for autonomous exploration of subterranean environments. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pages 1215–1221. IEEE, 2020. doi: 10.1109/ICRA40945.2020.9196662. URL https://doi.org/10.1109/ICRA40945.2020.9196662.

[14] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[15] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[19] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[20] Christopher Z Mooney. *Monte carlo simulation*, volume 116. Sage publications, 1997.

[21] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. ISSN 0036-8075. doi: 10.1126/science.153.3731.34. URL https://science.sciencemag.org/content/153/3731/34.

[22] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[23] Michael Wunder, Michael L Littman, and Monica Babes. Classes of multiagent q-learning dynamics with epsilon-greedy exploration. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1167–1174. Citeseer, 2010.

[24] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[27] Jan Peters and J Andrew Bagnell. Policy gradient methods. *Scholarpedia*, 5(11):3698, 2010.

[28] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930. doi: 10.1103/PhysRev.36.823. URL https://link.aps.org/doi/10.1103/PhysRev.36.823.

[29] Azad Abdulhafedh. How to detect and remove temporal autocorrelation in vehicular crash data. *Journal of Transportation Technologies*, 7:133–147, 04 2017. doi: 10.4236/jtts.2017.72010.

[30] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[32] Patrick Mannion, Sam Devlin, Karl Mason, Jim Duggan, and Enda Howley. Policy invariance under reward transformations for multi-objective reinforcement learning. *Neurocomputing*, 263:60 – 73, 2017. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2017.05.090. URL http://www.sciencedirect.com/science/article/pii/S0925231217311037. Multiobjective Reinforcement Learning: Theory and Applications.

[33] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL http://arxiv.org/abs/1802.09477.

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

[35] Tim Koning. Low level quadcopter control using reinforcement learning: Developing a self-learning drone. 2020.

[36] Adam Daniel Laud. Theory and application of rewards shaping in reinforcement learning. Text, IDEAL, 2005.

[37] Babak Badnava and Nasser Mozayani. A new potential-based reward shaping for reinforcement learning agent. *CoRR*, abs/1902.06239, 2019. URL http://arxiv.org/abs/1902.06239.

[38] Haosheng Zou, Tongzheng Ren, Dong Yan, Hang Su, and Jun Zhu. Reward shaping via meta-learning. *CoRR*, abs/1901.09330, 2019. URL http://arxiv.org/abs/1901.09330.

[39] Matej Vecerík, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin A. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *CoRR*, abs/1707.08817, 2017. URL http://arxiv.org/abs/1707.08817.

[40] Pieter Abbeel and Andrew Y. Ng. *Inverse Reinforcement Learning*, pages 554–558. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_417. URL https://doi.org/10.1007/978-0-387-30164-8_417.

[41] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *CoRR*, abs/1806.06877, 2018. URL http://arxiv.org/abs/1806.06877.

[42] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000.

[43] Seungwon Choi and Suseong Kim. Inverse reinforcement learning control for trajectory tracking of a multirotor uav. *International Journal of Control, Automation and Systems*, 15, 07 2017. doi: 10.1007/s12555-015-0483-3.

[44] Sanjay Krishnan, Animesh Garg, Richard Liaw, Brijen Thananjeyan, Lauren Miller, Florian T Pokorny, and Ken Goldberg. Swirl: A sequential windowed inverse reinforcement learning algorithm for robot tasks with delayed rewards. *The International Journal of Robotics Research*, 38(2-3):126–145, 2019.

[45] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[46] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. URL https://www.ros.org.

[47] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[48] Markus Achtelik Fadri Furrer, Michael Burri and Roland Siegwart. Rotors - a modular gazebo mav simulator framework. 2015.

[49] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

[50] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[51] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017. URL http://arxiv.org/abs/1710.05941.

[52] K. Gamagedara, M. Bisheban, E. Kaufman, and T. Lee. Geometric controls of a quadrotor uav with decoupled yaw control. In *2019 American Control Conference (ACC)*, pages 3285–3290, 2019. doi: 10.23919/ACC.2019.8815189.

[53] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[54] Carl Knospe. Pid control. *IEEE Control Systems Magazine*, 26(1):30–31, 2006.

[55] Y. Wang, J. Sun, H. He, and C. Sun. Deterministic policy gradient with integral compensator for robust quadrotor control. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(10): 3713–3725, 2020. doi: 10.1109/TSMC.2018.2884725.