POLYTECHNIC UNIVERSITY OF TURIN

Master's Degree in Mechanical Engineering

Master's Degree Thesis

Agile Drone Path Planning Based on Reinforcement Learning Algorithms

Supervisors

Prof. Giorgio GUGLIERI

Francesco MARINO

Candidate

Afshin ZEINADDINI MEYMAND

October 2023

Summary

Over the last decade, autonomous drone systems have significantly increased in various industries such as surveying, search and rescue, and last-mile delivery. These systems rely on various algorithms for trajectory planning, which are designed to navigate in different environments. However, most of the algorithms developed for trajectory planning are dedicated to static environments, where all objects other than the autonomous vehicle remain fixed during the system's operation.

One of the significant challenges that arise when working with autonomous drone systems is the dynamic nature of the environment. When the environment is not entirely static, and other objects such as the goal object are moving, it requires the implementation of different algorithms for various tasks such as object detection, state estimation, and trajectory planning. These algorithms must be able to accurately detect and track the moving objects, estimate their state, and plan a trajectory that avoids collisions while still reaching the goal object. This is a complex task that requires advanced techniques and algorithms.

Several solutions are available for state estimation of moving objects in dynamic environments, one of which is using Visual-Inertial Odometry (VIO) cameras. VIO cameras are specialized cameras that are specifically designed for state estimation tasks by providing precise and accurate 3D tracking data. They work by using multiple cameras to capture images of markers placed on the object of interest. These markers are small, highly reflective, and typically placed in a known pattern on the object. The VIO system uses advanced algorithms and image processing techniques to track the markers in the camera images, even in challenging conditions such as low light or fast motion. The system compares the images captured by each camera and calculates the distance between the markers by using the principle of triangulation, thus providing a precise and accurate 3D position and orientation of the object. Additionally, VIO cameras also use an Inertial Measurement Unit (IMU) which is a device that measures linear and angular accelerations, and magnetic fields. The IMU sensor works in conjunction with the cameras to provide additional information about the object's movement. The IMU sensor measures the angular velocity and the linear acceleration of the object, and then the data is fused with the visual data obtained from the cameras to improve the estimate of the object's

state. Overall, the use of VIO cameras in autonomous drone systems provides a robust and efficient solution for state estimation of moving objects in dynamic environments, as it is able to track the object's position and orientation in real-time, even under challenging conditions.

The use of Visual Odometry sensors for state estimation in model predictive control for trajectory planning of autonomous drones in dynamic environments can be challenging, as two main problems need to be addressed. These problems are:

- Handling the continuous action spaces: The action space for controlling the drone's trajectory is continuous and high-dimensional, making it challenging to find the optimal policy using traditional techniques.
- Dealing with uncertainty and non-stationary environments: The environment in which the drone operates is dynamic and uncertain, making it challenging to predict the system's future state and plan a trajectory that avoids collisions by VIO systems.

One solution to these problems is to use Kalman Filters for state estimation, by assuming that computer vision algorithms can extract the position of the gate centers without knowing anything about the dynamic model of the gates. Kalman Filters are a powerful tool for state estimation in dynamic systems, as they can handle non-linear systems and estimate the state of the system even in the presence of uncertainty and noise. It is widely used in various fields such as control systems, navigation, and robotics. Once the position of the gate centers is estimated using Kalman Filters, the trajectory of the drone for passing through that center can be derived. This trajectory can then be fed to different Advanced Actor-Critic Reinforcement Learning Algorithms such as Deep Deterministic Policy Gradient(DDPG), Soft-Actor Critic (SAC), and Proximal Policy Optimization (PPO) to derive the best policy. These algorithms are well-suited for trajectory planning tasks in dynamic environments as they can handle continuous action spaces, uncertainty, and non-stationary environments. RL algorithms are used to learn the optimal policy for an agent to make decisions based on the system's state. RL algorithms use a trial-and-error approach to learn the optimal policy by exploring different actions and receiving feedback in the form of rewards. When Kalman filtering and RL algorithms are combined, the Kalman filter can provide accurate and precise state estimates, while the RL algorithm can learn the optimal policy for the agent to make decisions based on these estimates. Combining these two techniques can lead to improved performance in tasks such as trajectory planning, control, and decision-making. One of the key challenges in RL is defining the reward function, which can be different for each task and environment. The mixed Kalman Filter and RL algorithms address this challenge by defining the reward function based on the distance of the drone's current position with respect to the predicted position of the center of the gate.

The built model of the KFRL algorithm is a continuous state-action environment, which means that the state of the system and the actions taken by the agent are continuous variables. In this model, the agent can take any action in a continuous range of values, rather than only a discrete set of actions. This allows for more flexibility and precision in controlling the drone's trajectory.

Acknowledgements

I would like to thank my advisor, Francesco Marino, for his consistent feedback sessions, guidance, and patience. I also extend my gratitude to Prof. Giorgio Guglieri for allowing me to pursue my passion and conduct research in Path Planning and Reinforcement Learning.

I'd like to express my appreciation to my friends, Ahmad Moori, Mohammad Andayesh, and Masoud Arabbeiki, for their practical and emotional support. Lastly, I want to express deep appreciation for my parents and brothers for believing in me, which kept my motivation high throughout my master's degree journey.

Afshin

Table of Contents

List of Tables

List of Figures

Acronyms

RL
Reinforcement Learning

DRL
Deep Reinforcement Learning

DQN
Deep Q-Network

DDQN
Double Deep Q-Network

D3QN
Duelling Double Deep Q-Network

PER
Prioritized Experience Replay Buffer

DDPG
Deep Deterministic Policy Gradient

TD3
Twin Delayed Deep Deterministic Policy Gradient

SAC
Soft Actor Critic

PPO
Proximal Policy Optimization

SIMD

Single Instruction Multi Data

VIO

Visual-Inertial Odometry

Chapter 1

Introduction

1.1 Introduction

Interest in autonomous vehicles covering aerial, terrestrial, and underwater areas has dramatically increased due to their prospective ability to transform many industries. The development of reliable path-planning methods is crucial for task optimization and the enhancement of the versatility and efficiency of these vehicles. Autonomous drones show substantial potential, able to conduct intricate tasks across a wide array of environments. However, the constantly changing nature of these environments introduces distinct obstacles, making path planning for drones an intriguing field of study. Autonomous drones are utilized in various duties, from surveillance and delivery services to search and rescue missions. To carry out these tasks effectively, they require path planning optimization considering the immediate environment and the inherent dynamism of real-world settings. Several path planning methodologies have been investigated, such as minimum snap path planning [1, 2] and minimum time path planning [3, 4]. Despite their efficiency, these strategies are predominantly designed for static environments. Conversely, drones often operate in dynamic environments where objects and conditions can shift unpredictably. As a result, path planning for dynamic environments is inherently more intricate and demanding. To enable effective path planning and navigation, estimating the status of dynamic objects is made even more challenging by the cluttered nature of these environments. Existing research has started addressing these challenges. Path planning and state estimation are central to the operations of autonomous drones. Path planning involves determining the optimal route a drone should take to accomplish its task, which can greatly differ depending on the task and environment specifics. A common method involves defining specific waypoints the drone must reach and then determining the most efficient route. State estimation complements path planning by supplying vital information about

the current and future states of the environment. This information facilitates informed decisions about the drone's path. State estimation techniques can be employed to identify significant features in the environment, such as the center of goal objects. Predicting the future location of these objects, especially in dynamic environments, continues to be a field-wide challenge. In this study, we put forth a novel methodology to tackle the challenges of state estimation and path planning in dynamic environments. Our strategy harnesses the capabilities of Kalman filter algorithms for estimating and forecasting the states of dynamic objects in the environment. More specifically, we use these algorithms to estimate and forecast the positions of waypoints, which are defined as the centers of gates in our context. The Kalman filter facilitates estimation of the current state of these waypoints, and their future states over a variable time horizon, contingent on the drone's distance from each gate. To complement state estimation, we also suggest the employment of advanced actor-critic reinforcement learning (RL) algorithms for optimal path planning. We implement three RL algorithms in a continuous action-state environment to discern the optimal path through the predicted waypoints. The path's optimality is defined based on a reward function provided to the RL algorithms. The combination of state estimation and path planning techniques is intended to significantly enhance the performance of autonomous drones in dynamic environments. In this paper, our primary objective is to explore the effectiveness of a combined approach that uses the Kalman filter to predict future positions of gates over changing time horizons and different Reinforcement Learning (RL) algorithms for optimal path planning in dynamic environments. We aim to assess and compare the performance of three specific RL algorithms under varying environmental conditions, with a particular emphasis on dynamic settings.

1.2 Related Work

The landscape of available training environments includes several three-dimensional reinforcement learning simulation environments, such as Gym [5] and AirSim [6], each with their challenges. For instance, despite AirSim's realistic and detailed nature, it demands a high-performance system due to its computational intensity. On the other hand, Gym, which is less computationally intensive, lacks environments specifically dedicated to drone path planning, with some environments supporting only discrete action spaces. Our research aims to surmount these hurdles by formulating a customized continuous action-state environment using the Python Pygame library. This tailor-made environment reduces the computational demand compared to AirSim and is also precisely suited to our specific task. We have integrated a field-of-view feature commonly absent in Pygame environments but vital for authentic drone perception to enhance its realism. Various studies have

explored autonomous drone path planning, utilizing the Potential Field Method (PFM), Rapidly exploring Random Tree (RRT), and the Voronoi diagram (VD). Each of these techniques presents its limitations. For example, the Potential Field Method is susceptible to local minima, causing a deadlock where the drone gets trapped in a position surrounded by obstacles [7, 8]. RRT, although efficient in navigating high-dimensional spaces, may need help finding the optimal path, and its performance can be compromised in densely cluttered environments [9, 10]. Voronoi Diagrams, while ensuring a safe distance from obstacles, do not necessarily yield the shortest or most efficient path, and their generation can be computationally intense, particularly in complex or dynamic environments [11, 12]. There has also been considerable effort in applying reinforcement learning to path planning [13, 14, 15, 16, 17]. The Song Et al. study didn't test the algorithm's performance in dynamic environments. It depended on Vicon cameras for gate state prediction, which may not be universally applicable in real-world contexts [15]. Another study applied three reinforcement learning algorithms in a static environment [13]. Some other researchers have conducted various studies on Model Predictive Control algorithms to address challenges in the field of path planning. [18, 19, 20].

Furthermore, there is ongoing research on dynamic environments aimed at predicting the desired dynamic goal to navigate towards it. However, the challenge lies in the fact that the implemented approach requires Visual-Inertial Odometry (VIO) cameras for state estimation of the dynamic gates and predicting their future positions. [21].

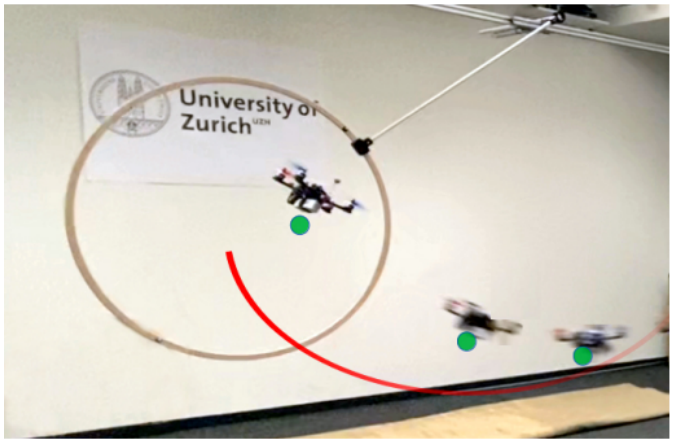

Figure 1.1: Policy Search Model Predictive Control proposed by Perception Group at University of Zurich [21]

Our research strives to bridge these gaps by leveraging Kalman Filters for state estimation and prediction [22]. We aim to design a reinforcement learning environment that meets specific needs, such as supporting a continuous action state

environment, providing specific state elements, and defining the agent's reward function based on the Kalman filter prediction. Additionally, we incorporate the field of view of real cameras as boundary conditions for the environment to enhance the realism and practicality of our approach.

1.3 Methodology

We employed a comprehensive methodology with a 3D simulation environment to address the issues related to dynamic environments as well as continuous action space environments. This environment was used to emulate the dynamic conditions that drones may encounter in real-world scenarios. Firstly, some algorithms are implemented to evaluate the results in discrete action space environments, and in the next step the environments and actions considered to be a continuous one. This 3D simulation allowed us to explore all possible outcomes under various scenarios and conditions, thereby providing a robust and extensive platform for the training and testing of our autonomous drone systems. These conditions include the different assumptions for initial position of the drone and gates, velocity and acceleration of the gates, as well as the field of view of the drone. Our methodology also involved the use and comparison of different reinforcement learning (RL) algorithms: Deep Deterministic Policy Gradient (DDPG), Dueling Double DQN, Proximal Policy Optimization (PPO), and Prioritized Experience Replay algorithms for discrete action state environments as well as Deep Deterministic Policy Gradient, Soft-Actor Critic (SAC), and Proximal Policy Optimization for continuous action state environments. These algorithms were trained to focus on Path planning and state estimation, two critical factors for successful drone operation in dynamic environments. The algorithms were evaluated on the same scenarios within the 3D simulation to ensure a fair comparison. The performance of each algorithm was then analyzed and compared, with particular attention paid to their respective ability to navigate successfully in the dynamic simulated scenarios with continuous action state environments. Moreover, the Kalman Filter, a well-known technique for state estimation, was integrated with the RL techniques. The goal was to enhance the drones' decision-making process and path planning. Combining these two techniques, we aimed to create an autonomous drone system that could adapt and respond effectively to non-static environments.

Chapter 2

Deep Reinforcement Learning and Kalman Filtering Algorithms

2.1 Deep Reinforcement Learning

2.1.1 Introduction to Deep Reinforcement Learning

The difference between deep learning and reinforcement learning is that deep learning is learning from a training set and then applying that learning to a new data set, while reinforcement learning is dynamic learning by adjusting actions based on feedback to maximize a reward. Reinforcement learning is dynamic learning with a trial-and-error method to maximize the outcome, while deep reinforcement learning is learning from existing knowledge and applying it to a new data set.

Feedback in Reinforcement Learning

Deep Reinforcement Learning is a complex sequential decision-making problem under uncertainty. The kinds of feedback in DRL are as follows:

- **Sequential:** The opposite of sequential feedback is one-shot feedback. Decisions don't have long-term consequences in problems that deal with one-shot feedback, such as supervised learning. One of the main challenges of sequential feedback is that your agents can receive delayed information, like in a chess game. Delayed feedback makes it tricky to interpret the source of the feedback.
- **Evaluative:** The crux of evaluative feedback is that the goodness of the feedback is only relative because the environment is uncertain. While not

having access to the model of the environment, we must explore to gather new information or improve on our current information. Eventually, the exploration-exploitation trade-off arises. The opposite of evaluative feedback is supervised feedback. In a classification problem, your model receives supervision; during learning, your model is given the correct labels for each of the samples provided.

- **Sampled**: in Deep Reinforcement Learning, agents need to generalize using the gathered feedback and make intelligent decisions based on that generalization. The opposite of sampled feedback is exhaustive feedback. To exhaustively sample environments means agents have access to all possible samples.

Function Approximation in Reinforcement Learning

Motivations for the use of function approximation to solve reinforcement learning problems

- **High Dimensionality of State and Action Space**: The main drawback of tabular reinforcement learning is that using a table to represent value functions is no longer practical in complex problems. Environments can have high-dimensional state spaces, meaning that the number of variables that comprise a single state is vast.

- **Continuous State and Action Space**: Environments can additionally have continuous variables, meaning that a variable can take on an infinite number of values. To clarify, state and action spaces can be high dimensional with discrete variables, they can be low dimensional with continuous variables, and so on.

Advantages when using function approximation Function approximation can make our algorithms more efficient and more complex relationships can be discovered with a non-linear function approximator, such as a neural network.

Value Functions

Using neural networks to approximate value functions can be done in many ways. There are many different value functions we could approximate, as:

- The state-value function $v(s)$: It helps you know how much expected total discounted reward you can obtain from state s and using policy π thereafter. But, to determine which action to take with a V-function, you also need the MDP of the environment so that you can do a one-step look-ahead and take into account all possible next states after selecting each action.

- The action-value function $q(s, a)$: If we had the values of state-action pairs, we could differentiate the actions that would lead us to either gain information, in the case of an exploratory action, or maximize the expected return, in the case of a greedy action. Therefore, it can be used in control problems.
- The action-advantage function $a(s, a)$: it helps us differentiate between values of different actions and lets us easily see how much better than average an action is.

Neural Network Architecture

We can have two different Neural Network Architecture, and the choice depends on the complexity of the state, action and environment in which the agent is training.

- State-action-in-value-out architecture: This architecture is designed to take both the state and action as inputs and output a value, which usually represents the expected reward or the quality of the action given the state. It's ideal for situations where the relationship between state-action pairs and the corresponding value is crucial to be explicitly modeled.
- State-in-values-out architecture: Suited for scenarios where the action space is vast or continuous, this architecture processes the state as its input and returns a value for each potential action or a distribution over continuous actions. This design proves invaluable in environments where evaluating every individual state-action pair would be computationally impractical due to the massive number of possible actions.

Objective Function for Optimization

$$L_i(\theta_i) = \mathbb{E}_{s,a}[(q^*(s, a) - Q(s, a; \theta_i))^2] \tag{2.1}$$

The equation 2.1 reports an ideal objective in value-based deep reinforcement learning would be to minimize the loss with respect to the optimal action-value function q^*. We want to have an estimate of q^*, Q, that tracks exactly that optimal function. If we had access to the optimal action-value function, we'd use that, but if we had access to sampling the optimal action-value function, we could then minimize the loss between the approximate and optimal action-value functions. Furthermore, the optimal action-value function would be as follows:

$$q^*(s, a) = \max_{\pi} \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a], \forall s \in S, \forall a \in A(s) \tag{2.2}$$

Based on the above function, an optimal action-value function is a policy that gives the maximum expected return for each action in each state. Since we do not

have the optimal policy, we can not sample the optimal Q-values, and as a result, we do not have the optimal action-value function q_*. Instead, we must alternate between evaluating a policy (by sampling actions from it), and improving it (using an exploration strategy, such as epsilon-greedy)

Targets for Policy Evaluation

We can use different targets to estimate the action-value function of a policy π.

- Monte Carlo targets (MC): Use all rewards found in a trajectory from a start state to the terminal state.
- Temporal Difference targets (TD): Use the value of the next state as an estimate of all rewards to go.
- N-step targets: $N - step$ is like TD, but instead of bootstrapping after one step, you use n steps.
- lambda targets: Lambda target mixes in an exponentially decaying fashion all $n - step$ targets into one.

Learning Strategy

- On-Policy: if we were to use the on-policy target, the target would approximate the behavioral policy; the policy-generating behavior and the policy being learned would be the same.
- Off-Policy: In off-policy target we always approximate the greedy policy, even if the policy-generating behavior isn't totally greedy.

Notice that both on-policy and off-policy targets estimate an action-value function. **The Q-learning target as an off-policy TD target** In practice, an online Q-learning target would look something like the following:

$$y_i^{Q-learning} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_i) \tag{2.3}$$

The bottom line is we use the experienced reward and the next state to form the target. We can plug in a more general form of this Q-learning target as in the following:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(r + \gamma \max_a Q(s^{'}, a^{'}; \theta_i) - Q(s, a; \theta_i))^2] \tag{2.4}$$

But it's basically the same. We're using the expectation of experience tuples to minimize the loss. When differentiating through this equation, you must notice

the gradient doesn't involve the target. The gradient must only go through the predicted value.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[(r + \gamma \max_a Q(s', a'; \theta_i) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)] \tag{2.5}$$

Optimization Method

Gradient descent is a stable optimization method given a couple of assumptions: data must be independent and identically distributed (IID), and targets must be stationary. In reinforcement learning, however, we cannot ensure any of these assumptions hold, so choosing a robust optimization method to minimize the loss function can often make the difference between convergence and divergence. Some optimization methods are introduced in the following:

- **Gradient Decent**
  Based on the dataset which is given for the optimization, the gradient descent algorithm can be defined as follows:
  - Batch Gradient Decent: The batch gradient descent algorithm takes the entire dataset at once, calculates the gradient of the given dataset, and steps toward this gradient a little bit at a time. Then, it repeats this cycle until convergence. When you have a considerable dataset with millions of samples, batch gradient descent is too slow to be practical. Moreover, we don't even have a dataset in reinforcement learning in advance, so batch gradient descent isn't a practical method for our purpose either. Batch gradient descent goes smoothly toward the target because it uses the entire dataset at once, so a lower variance is expected.
  - Mini Batch Gradient Decent: In mini-batch gradient descent, we use only a fraction of the data at a time. We process a mini-batch of samples to find its loss, then backpropagate to compute the gradient of this loss, and then adjust the weights of the network to make the network better at predicting the values of that mini-batch. With mini-batch gradient descent, you can control the size of the mini-batches, which allows the processing of large datasets. In mini-batch gradient descent we use a uniformly sampled mini-batch. This results in noisier updates, but also faster processing of the data.
  - Stochastic Gradient Decent [23, 24]: At the other extreme, you can set the mini-batch size to a single sample per step. In this case, you're using an algorithm called stochastic gradient descent. With stochastic gradient descent, in every iteration we step through only one sample. This makes it a noisy algorithm. It wouldn't be surprising to see several steps taking us further away from the target, and later back toward the target.

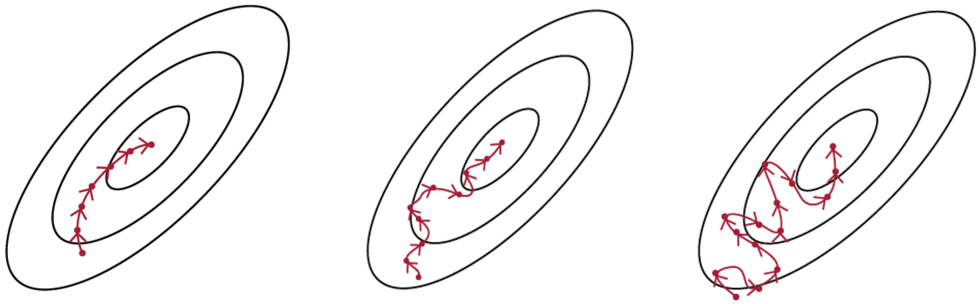

Figure 2.1: Batch Gradient Decent, Mini Batch Gradient Decent, Stochastic Gradient Decent

- **Gradient Decent with Momentum**
  An improved gradient descent algorithm is called gradient descent with momentum, or momentum for short [25]. This method is a mini-batch gradient descent algorithm that updates the network's weights in the direction of the moving average of the gradients, instead of the gradient itself.

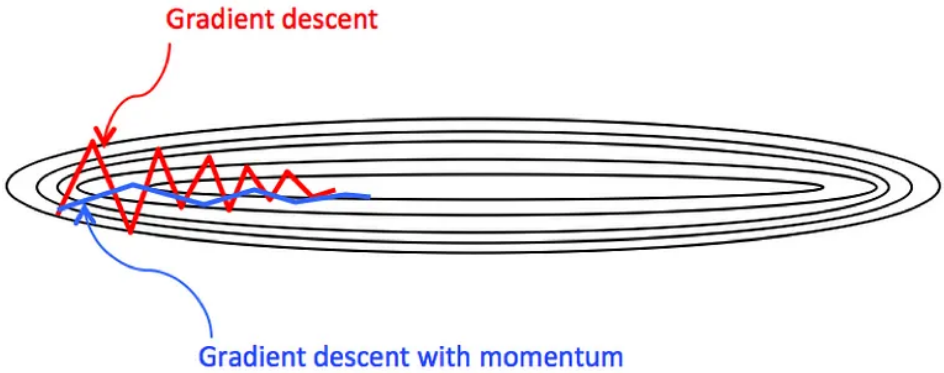

Figure 2.2: Gradient Decent Vs Gradient Decent with Momentum

- **Root Mean Square Propagation (RMSprop)**
  Root mean square propagation (RMSprop) is an alternative to using momentum. RMSprop and momentum do the same thing of dampening the oscillations and moving more directly toward the goal, but they do so differently. While momentum takes steps in the direction of the moving average of the gradients, RMSprop takes the safer bet of scaling the gradient in proportion to a moving average of the magnitude of gradients. It reduces oscillations by merely scaling the gradient in proportion to the square root of the moving

average of the square of the gradients or, more simply put, in proportion to the average magnitude of recent gradients.

- **Adaptive Moment Estimation (Adam) [26]**
  Adam is a combination of RMSprop and momentum. The Adam method steps in the direction of the velocity of the gradients, as in momentum. But, it scales updates in proportion to the moving average of the magnitude of the gradients, as in RMSprop. These properties make Adam as an optimization method a bit more aggressive than RMSprop, yet not as aggressive as momentum.

2.1.2 Value-Based Deep Reinforcement Learning Algorithms

This section will introduce some value-based algorithms based on their improvements.

Deep Q-Network

Deep Q-Network (DQN) is an algorithm for training a deep neural network to play a game or control a system through reinforcement learning. It was introduced by Google DeepMind in a 2015 paper titled "Human-level control through deep reinforcement learning" [27, 28].
The DQN algorithm combines Q-learning, a standard reinforcement learning algorithm, with a deep neural network as a function approximator. Q-learning is a model-free algorithm that allows an agent to learn the optimal action to take in a given state by updating an estimate of the maximum expected future reward for each action. The neural network is used to approximate the Q-function, which gives the expected future reward for each action in a given state.
The DQN algorithm uses experience replay, where the agent stores a history of its experiences in a buffer and samples from this buffer to update the network weights [29]. This helps to decorrelate the experiences and stabilize the learning process. Experience replay consists of a data structure, often referred to as a replay buffer or a replay memory, that holds experience samples for several steps, allowing the sampling of mini-batches from a broad set of past experiences. Having a replay buffer allows the agent two critical things. First, the training process can use a more diverse mini-batch for performing updates. Second, the agent no longer has to fit the model to the same small mini-batch for multiple iterations. The algorithm also uses a target network, which is a copy of the primary network that is used to compute the target Q-values for the primary network's updates. This helps to reduce the variance in the updates and improve the stability of the learning process.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim u(D)} \Big[(r + \gamma \max_a Q(s', a'; \theta^-) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \Big] \tag{2.6}$$

A target network is a previous instance of the neural network that we freeze for a number of steps. The gradient update now has time to catch up to the target, which is much more stable when frozen. This adds stability to the updates. we're now obtaining the experiences we use for training by sampling uniformly at random the replay buffer D, instead of using the online experiences.
It's important to note that in practice, we don't have two "networks," but instead, we have two instances of the neural network weights. We use the same model architecture and frequently update the weights of the target network to match the weights of the online network, which is the network we optimize at every step. Using target networks prevents the training process from spiraling around because we're fixing the targets for multiple time steps, thus allowing the online network weights to move consistently toward the targets before an update changes the optimization problem and a new one is set. By using target networks, we stabilize training, but we also slow down learning because you're no longer training on up-to-date values.
Another way you can lessen the non-stationarity issue, to some degree, is to use larger networks. More powerful networks make subtle differences between states more likely to be detected. Larger networks reduce the aliasing of state-action pairs; the more powerful the network, the lower the aliasing; the lower the aliasing, the less apparent correlation between consecutive samples. And all of this can make target values and current estimates look more independent of each other.
But, a more powerful neural network takes longer to train. It needs not only more data (interaction time) but also more compute (processing time). Using a target network is a more robust approach to mitigating the non-stationary problem

Double DQN

Q-learning tends to overestimate action-value functions. Our DQN agent is no different; we're using the same off-policy TD target, after all, with that max operator. The crux of the problem is simple: We're taking the max of estimated values. Estimated values are often off-center, some higher than the true values, some lower, but the bottom line is that they're off. The problem is that we're always taking the max of these values, so we have a preference for higher values, even if they aren't correct. Our algorithms show a positive bias, and performance suffers.

$$\begin{aligned}\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim u(D)}[(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta^-); \theta^-) \\ - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)]\end{aligned} \tag{2.7}$$

Double Deep Q-Network (Double DQN) is an extension of the Deep Q-Network (DQN) algorithm for training deep neural networks to play games or control systems through reinforcement learning. It was introduced by Google DeepMind in a 2015 paper titled "Deep Reinforcement Learning with Double Q-learning" [30].
In the DQN algorithm, the Q-values for each action in a given state are approximated using a deep neural network. The Q-values are updated using the Bellman equation, which is a recursive relationship that describes the relationship between the expected future reward for an action and the immediate reward and the expected future reward for the next action. In the Bellman equation, the expected future reward for the next action is typically computed using the same neural network that is used to approximate the Q-values.
However, this can lead to an overestimation of the expected future reward, which can cause the Q-values to be biased and the learning process to be unstable. The Double DQN algorithm addresses this issue by using two separate neural networks to approximate the Q-values and to compute the expected future reward for the next action. The primary network is used to select the action, and the target network is used to compute the expected future reward. This helps to reduce the overestimation of the expected future reward and improve the stability of the learning process.
Like the DQN algorithm, the Double DQN algorithm also uses experience replay and a target network to stabilize the learning process. It has been successfully applied to a variety of tasks and has been shown to improve the performance of the DQN algorithm in some cases.

Dueling DDQN

Dueling Double Deep Q-Network (Dueling Double DQN) is an extension of the Double Deep Q-Network (Double DQN) algorithm for training deep neural networks to play games or control systems through reinforcement learning. It was introduced by Google DeepMind in a 2016 paper titled "Dueling Network Architectures for Deep Reinforcement Learning". [31].
Like the Double DQN algorithm, the Dueling Double DQN algorithm uses two neural networks: a primary network and a target network. The primary network is used to approximate the Q-values for each action in a given state and to select the action to take. The target network is used to compute the expected future reward for the next action.
The Dueling Double DQN algorithm introduces a new architecture for the primary

network that separates the network into two streams: one for the state value function and one for the action advantage function. The state value function gives the expected return for a given state, and the action advantage function gives the advantage of taking a particular action over the average of all actions in a given state. The Q-values are then computed as the sum of the state value function and the action advantage function for each action.
This architecture has several benefits. First, it allows the network to learn the relative importance of the state value function and the action advantage function separately, which can improve the learning process. Second, it allows the network to learn the optimal action to take in a given state without having to learn the Q-values for all actions in that state, which can reduce the complexity of the learning process. Finally, it allows the network to generalize better to new states and actions, which can improve the performance of the algorithm.
Like the Double DQN algorithm, the Dueling Double DQN algorithm also uses experience replay and a target network to stabilize the learning process. It has been successfully applied to various tasks and has been shown to improve the performance of the Double DQN algorithm in some cases.

$$Q_(s, a; \theta, \alpha, \beta) = [V(s; \theta, \beta)] + [A(s, a; \theta, \alpha)] \tag{2.8}$$

The Q-function is parameterized by theta, alpha, and beta. Theta represents the weights of the shared layers, alpha the weights of the action-advantage function stream, and beta the weights of the state-value function stream.

$$Q_(s, a; \theta, \alpha, \beta) = [V(s; \theta, \beta)] + \left([A(s, a; \theta, \alpha)] - \frac{1}{|A|} \sum_{a'} A(s, a^{'}; \theta, a) \right) \tag{2.9}$$

But because we cannot uniquely recover the Q from V and A , we use the above equation in practice. This removes one degree of freedom from the Q-function. The action-advantage and state-value functions lose their true meaning by doing this. But in practice, they're off-centered by a constant and are now more stable when optimizing.

Prioritized Experience Replay Buffer (PER)

Prioritized Experience Replay (PER) is a technique for improving the efficiency of the experience replay process in reinforcement learning. It was introduced by Google DeepMind in a 2016 paper titled "Prioritized Experience Replay" [32].
In reinforcement learning, experience replay is a technique where the agent stores a history of its experiences in a buffer and samples from this buffer to update

the network weights. This helps to decorrelate the experiences and stabilize the learning process. However, traditional experience replay uniformly samples from the buffer, which can be inefficient because it may sample trivial or unimportant experiences more frequently than important or rare experiences.
To address this issue, the PER algorithm introduces a priority value for each experience in the buffer. The priority value reflects the importance of the experience, with higher values indicating more important experiences. The algorithm then samples from the buffer using a distribution that is proportional to the priority values, which allows it to sample more important experiences more frequently.
The priority values are updated using the temporal difference error, which measures the difference between the expected and actual return for an action. Experiences with larger temporal difference errors are assigned higher priority values because they are more likely to be important for learning.
The PER algorithm has been shown to improve the efficiency of the learning process by allowing the agent to learn more from important experiences and to learn more quickly. It has been used in combination with a variety of reinforcement learning algorithms, including Deep Q-Networks (DQN) and Double Deep Q-Networks (Double DQN).

$$|\delta_i| = \underbrace{\underbrace{|\underbrace{r + \gamma Q\left(s^{'}, \arg\max_{a^{'}} Q(s^{'}, a^{'}; \theta_i, \alpha_i, \beta_i\right); \theta^{-}, \alpha^{-}, \beta^{-}}_{\text{Duelling DDQN target}} - Q(s, a; \theta_i, \alpha_i, \beta_i)|}_{\text{Duelling DDQN error}}}_{\text{Absolute Duelling DDQN error}} \tag{2.10}$$

2.1.3 Policy-Based Deep Reinforcement Learning Algorithms

Pure policy-gradient methods constitute a class of reinforcement learning algorithms that focus on directly optimizing stochastic policies, eschewing the use of value functions. Unlike other methods such as Q-learning or Actor-Critic, which leverage value functions as a means of estimating the expected cumulative reward from a given state or state-action pair, these methods directly optimize the policy to guide the agent's interactions with the environment. Stochastic policies, central to pure policy-gradient methods, define a probability distribution over actions for each state, promoting exploration in the action space and enabling the agent to discover superior actions over time. As a result, pure policy-gradient methods offer a direct and efficient approach to learning optimal strategies for agents interacting with complex environments, bypassing the need for intermediate value estimations [33].

While accurately determining the values of states in reinforcement learning can be a complex task, employing a rough approximation can prove beneficial for reducing the variance of the policy-gradient objective. By leveraging approximate state values, one can effectively mitigate the fluctuations in the policy-gradient estimates, thereby enhancing the stability and convergence of the learning process. Although pure policy-gradient methods typically focus on direct policy optimization without utilizing value functions, incorporating approximate state values can help strike a balance between exploration and exploitation in the action space, ultimately leading to more efficient and robust policy learning in challenging environments.

Incorporating a value function as a baseline or for calculating advantages can significantly reduce the variance of the targets utilized in policy-gradient updates, ultimately facilitating more efficient and stable learning. By employing a value function, the algorithm can differentiate between the intrinsic value of a state and the advantage of taking specific actions, which allows for more precise policy updates. Consequently, this reduction in variance often translates to accelerated learning, as the agent can converge more rapidly to an optimal policy. Thus, while pure policy-gradient methods focus on direct policy optimization, integrating value functions in the learning process can enhance the overall performance and robustness of the reinforcement learning algorithm.

In addressing continuous environments, researchers often investigate methods that learn deterministic policies [34], which are characterized by their consistency in returning the same action, deemed optimal, when presented with an identical state. Deterministic policies offer a distinct advantage in continuous environments, as they facilitate more efficient exploration of the action space and provide a clear action selection mechanism. By focusing on the optimal action for each state, these methods aim to learn a concise and effective policy that guides the agent's interactions with the environment, enabling it to navigate complex and continuous scenarios with improved performance and reliability.

Deep Deterministic Policy Gradient (DDPG)

In this section, the focus is on the Deep Deterministic Policy Gradient ($DDPG$) algorithm, an approach that can be viewed as an extension of DQN for continuous action spaces. DDPG incorporates key techniques found in DQN, such as using a replay buffer for off-policy training of an action-value function and employing target networks for training stabilization. However, DDPG goes beyond DQN by training an approximate deterministic policy that represents the optimal action. As a result, DDPG is classified as a deterministic policy-gradient method specifically designed for continuous action spaces, offering an efficient and robust solution for tackling complex environments that demand continuous decision-making [35].

DDPG can be visualized as an algorithm with a similar architecture to DQN,

sharing some key aspects in the training process. Both DDPG and DQN involve the agent collecting experiences online and storing them in a replay buffer. During each step, the agent retrieves a mini-batch from the buffer, typically sampled uniformly at random, and uses it to calculate a bootstrapped Temporal Difference (TD) target and train a Q-function.

However, there are notable differences between DQN and DDPG. While DQN determines the greedy action using an $argmax$ operation on the target Q-function, DDPG employs a target deterministic policy function trained to approximate the greedy action. This means that, in DDPG, the best action in the next state is directly approximated using a policy function, rather than relying on the $argmax$ of the Q-function as in DQN.

The use of a deterministic policy function in DDPG has several advantages:

- It bypasses the need for an $argmax$ operation, which can be computationally expensive in continuous action spaces.
- The policy function provides a more direct approximation of the optimal action, potentially leading to faster convergence.
- The deterministic policy function can help reduce the exploration-exploitation trade-off, as it directly targets the optimal action for each state.

This approach can be advantageous in continuous action spaces, as it eliminates the need for $argmax$ calculations and offers more direct action approximations, potentially leading to improved learning efficiency and performance.

The DDPG value function objective is as follows:

$$\begin{aligned} L_i(\theta_i) = \mathbb{E}_{(s_{KF},a,r,s'_{KF})\sim u(D)} \Big[\Big(r + \gamma Q\left(s'_{KF}, \mu(s'_{KF};\phi');\theta^-\right) \\ -Q\left(s_{KF}, a;\theta_i\right)\Big)^2\Big] \end{aligned} \tag{2.11}$$

It means that in DDPG, similar to DQN, a mini-batch is sampled from the replay buffer during the training process. However, DDPG deviates from DQN by learning a deterministic policy, denoted as mu, instead of relying on the $argmax$ operation over the Q-function. The policy mu is trained to approximate the deterministic greedy action for the given state, thereby streamlining the action selection process and improving the efficiency of learning in continuous action spaces.

To implement DDPG effectively, the introduction of a policy network is necessary. The goal of this network is to provide the optimal action for a given state, and it must be differentiable with respect to the action. This requirement makes it essential for the action to be continuous, ensuring efficient gradient-based learning. The optimization objective is straightforward, as it involves using the expected Q-value derived from the policy network, mu. The agent's task is to identify

the action that maximizes this value. It is important to note that, in practice, optimization techniques usually focus on minimizing the negative of this objective, as most optimization algorithms are designed for minimization problems.

Therefore, the DDPG's deterministic policy objective:

$$J_i(\phi_i) = \mathbb{E}_{(s_{KF})\sim u(D)} \left[Q(s_{KF}, \mu(s_{KF}; \phi); \theta) \right] \tag{2.12}$$

In our case, target networks are not used, and instead, the online networks are employed for both the policy (action selection) and the value function (action evaluation). This approach simplifies the architecture and training process. Moreover, since a mini-batch of states is sampled for training the value function, these same states can be efficiently utilized for training the policy network as well. This simultaneous use of state samples for both networks ensures consistency in learning and promotes computational efficiency during the training process.

In DDPG, deterministic greedy policies are trained with the goal of taking in a state and returning the optimal action for that state. However, in an untrained policy, the actions generated might not be accurate, even though they are deterministic. As previously discussed, agents need to strike a balance between exploiting existing knowledge and exploring new possibilities. Due to the deterministic nature of the DDPG agent's policy, on-policy exploration is not feasible, as it would result in the agent being "stubborn" and consistently selecting the same actions.

To address this issue, DDPG employs off-policy exploration, injecting Gaussian noise into the actions selected by the policy. This allows the agent to explore new actions and learn from different experiences, which may be crucial for the agent's overall performance. Consequently, in DDPG, the agent explores by adding external noise to the actions, utilizing off-policy exploration strategies to efficiently learn and adapt to its environment.

Twin-Delayed Deep Deterministic Policy Gradient Algorithm (TD3)

This algorithm is introduced by Scott Fujimoto et al. in 2018 [36]. There are some differences between TD3 and DDPG algorithms which can be considered as the improvements to the DDPG algorithm. These differences are as follows:

- It has double learning technique as DDQN instead of just one learning network:

  In TD3, a particular kind of Q-function with two separate networks is used which leads to two different estimations of the state-action pairs.

$$J_i(\theta_i^a) = \mathbb{E}_{(s_{KF}, a, r, s'_{KF})\sim u(D)} \left[(TWIN^{target} - Q(s, a; \theta_i^a))^2 \right] \tag{2.13}$$

$$J_i(\theta_i^b) = \mathbb{E}_{(s_{KF},a,r,s'_{KF})\sim u(D)}\left[(TWIN^{target} - Q(s,a;\theta_i^b))^2\right] \tag{2.14}$$

The twin network loss would be the some of Mean Square Errors of each networks.

$$TWIN^{target} = r + \gamma \min_n Q\left(s'_{KF}, \mu(s', \phi^-); \theta^{n,-}\right) \tag{2.15}$$

And the target is calculated as the minimum between the two networks. Also, consider the target network is used for both the policy and value networks

- It adds noise to both the action passed to the environment and to the target actions which leads to more robustness to approximation error:

  To improve the exploration through the environment, a Gaussian noise is injected to actions used for the environment as well as to the actions used to calculate the targets. By this technique, the network would be more generalized over similar actions.

  $$clamp(x, l, h) = max(min(x, h), l)$$

  This is the definition of the clamp function which clips a value x between a low l and a high h.

  So, we have:

  $$a^{',smooth} = clamp(\mu(s', \phi^-) + clamp(\epsilon, \epsilon - l, \epsilon - h), a - l, a - h) \tag{2.16}$$

  The clipped Gaussian noise ϵ is introduced and added to the action to smooth that. Firstly, the ϵ is sampled and then clamped to be between a preset min and max for ϵ. Then, the clipped Gaussian noise is added to the action and the action is clamped to be between the min and max allowable action according to the specific environment. The smoothed action is used to calculate the $TD3^{target}$ as follows:

  $$TD3^{target} = r + \gamma \min_n Q(s', a^{',smooth}; \theta^{n,-}) \tag{2.17}$$

- It delays updates to the policy and target networks which leads the online Q-function updates more frequently:

  This delay is advantageous when the online Q-function undergoes abrupt changes early in the training. By pausing the policy's updates for several value function updates, it enables the value function to stabilize and produce more accurate values before influencing the policy. The suggested delay for the policy and target networks is every other update to the online Q-function.

  The other thing that you may notice in the policy updates is that we must use one of the networks of the online value model for getting the estimated Q-value for the action coming from the policy. In TD3, we use one of the two networks, but the same network every time.

Soft Actor-Critic Algorithm (SAC)

Off-policy refers to the learning approach in which the algorithm leverages experiences generated by a behavior policy that is distinct from the policy being optimized. In the context of DDPG and TD3, this is achieved by employing a replay buffer. A replay buffer is a memory structure that stores experiences (comprising state, action, reward, and next state) gathered from multiple previous policies. The replay buffer enables the algorithms to learn from a diverse set of experiences, which can lead to more stable and efficient learning [37].

To facilitate exploration, both DDPG and TD3 incorporate off-policy exploration strategies. In particular, they use Gaussian noise injection. Gaussian noise is a type of random noise that follows a Gaussian distribution. By adding this noise to the action vectors that interact with the environment, the algorithms encourage the exploration of various states and actions. This strategy helps the agent to discover novel and potentially better solutions, as opposed to merely exploiting the current knowledge.

In summary, DDPG and TD3 are off-policy reinforcement learning algorithms that train deterministic policies. They make use of a replay buffer to store and learn from diverse experiences generated by different behavior policies. To promote exploration, they apply Gaussian noise injection to the action vectors that are sent to the environment. These techniques contribute to the effectiveness and stability of the learning process.

The on-policy agents utilize stochastic policies, which inherently incorporate a degree of randomness. This randomness plays a crucial role in exploration, as it allows the agents to discover new possibilities and experiences.

In order to further encourage the element of randomness within these stochastic policies, an additional component called the entropy term is introduced into the loss function. By incorporating this entropy term, we can ensure that the agents

maintain a healthy balance between exploration and exploitation, which is essential for effective learning and decision-making. This approach helps prevent the agents from becoming overly focused on a single strategy or getting stuck in a local optimum, thus promoting more diverse and well-rounded learning experiences.

In this section, we delve into an algorithm known as the Soft Actor-Critic (SAC), which uniquely combines elements from two distinct paradigms. The SAC algorithm shares similarities with off-policy methods like DDPG (Deep Deterministic Policy Gradient) and TD3 (Twin Delayed Deep Deterministic policy gradient), as it operates in an off-policy manner. However, unlike these algorithms that utilize deterministic policies, SAC employs a stochastic policy.

By integrating the best of both worlds, the Soft Actor-Critic algorithm is able to reap the benefits of off-policy learning, such as efficient and stable updates from a replay buffer, while also harnessing the exploration advantages provided by stochastic policies. This fusion results in an algorithm that can achieve robust and effective learning, while maintaining a strong ability to explore and adapt to a wide range of environments and situations.

Incorporating entropy into the Bellman equations is a key aspect of the Soft Actor-Critic algorithm. This unique feature involves integrating the entropy of the stochastic policy directly into the value function, which the agent aims to maximize. The simultaneous maximization of both expected total reward and expected total entropy inherently promotes diverse behaviors while ensuring the agent continues to maximize the expected return.

By factoring in the entropy, the SAC algorithm encourages the agent to explore a wide range of possible actions and strategies, rather than simply focusing on a narrow set of options. This approach fosters greater adaptability and learning, as the agent is exposed to a broader spectrum of experiences and can fine-tune its decision-making process accordingly. As a result, the agent's ability to effectively navigate complex and dynamic environments is significantly enhanced, ultimately leading to more robust and reliable performance.

In SAC, we define the action-value function as follows:

$$\begin{aligned} q_\pi(s_{KF}, a) = \mathbb{E}_{r, s'_{KF} \sim P(s_{KF}, a), a' \sim \pi(s'_{KF})} \Big[r + \gamma \Big(q_\pi(s'_{KF}, a') \\ + \alpha H \left(\pi(.|s'_{KF}) \right) \Big) \Big] \end{aligned} \tag{2.18}$$

The expectation is over the reward, next state, and next action. We're going to add up the reward and the discounted value of the next state-action pair. However, we add the entropy of the policy at the next state. Alpha tunes the importance we give to the entropy term.

$$\boldsymbol{SAC}^{target} = r + \gamma \left[\min_n Q(s'_{KF}, \hat{a}'; \theta^{n,-}) - \alpha log \pi(\hat{a}' | s'_{KF}; \phi) \right] \tag{2.19}$$

$$J_{\pi}(\phi) = \mathbb{E}_{s_{KF} \sim u(D), \hat{a} \sim \pi} \left[\min_{n} Q\left(s_{KF}, \hat{a}; \theta^{n}\right) - \alpha log \pi(\hat{a} | s_{KF}; \phi) \right] \tag{2.20}$$

$$J(\alpha) = \mathbb{E}_{s_{KF} \sim u(D), \hat{a} \sim \pi} \left[\alpha \left(H + log \pi(\hat{a} | s_{KF}; \phi) \right) \right] \tag{2.21}$$

Proximal Policy Optimization (PPO)

The Proximal Policy Optimization (PPO) [38] algorithm represents a notable advancement in the field of reinforcement learning, specifically targeting on-policy methods. PPO's major contribution is the introduction of a surrogate objective function, which is a fundamental departure from traditional on-policy approaches such as the Advantage Actor-Critic (A2C). The surrogate objective function enables the algorithm to take multiple gradient steps using the same mini-batch of experiences, whereas other on-policy methods like A2C necessitate the immediate disposal of experience samples after a single optimization step [39].

The core concept behind PPO's surrogate objective function is that it allows for multiple policy updates using the same set of experiences. This is achieved by comparing the new policy to the old policy using a likelihood ratio, where the ratio represents the probability of taking an action under the new policy divided by the probability of taking that action under the old policy. The objective function aims to maximize the expected return while keeping the policy updates within a certain trust region, meaning the new policy should not deviate significantly from the old policy.

PPO's clipped objective function plays a crucial role in controlling the magnitude of policy updates. The clipping mechanism prevents the policy from undergoing drastic changes after each optimization step, thereby ensuring that updates are conservative and controlled. This approach has two major benefits.

First, it addresses the problem of performance collapse, a prevalent issue in on-policy policy gradient methods. Performance collapse occurs when the policy updates are too aggressive, causing the algorithm to lose the progress made in previous iterations. By constraining policy updates, PPO's clipped objective function reduces the risk of performance collapse, resulting in a more stable learning process.

Second, the conservative optimization approach allows for the reuse of mini-batches of experiences in multiple optimization steps. This is a significant advantage over other on-policy methods, as it leads to improved sample efficiency. The ability to reuse experiences means that the algorithm can extract more information from each sample, reducing the overall number of samples required for effective learning.

In conclusion, the Proximal Policy Optimization algorithm presents a significant improvement over traditional on-policy methods in reinforcement learning. Its

innovative surrogate objective function allows for multiple gradient steps using the same mini-batch of experiences, and the clipped objective function ensures conservative policy updates. These features contribute to mitigating performance collapse and enhancing sample efficiency, making PPO a powerful and reliable choice for many reinforcement learning problems.

Proximal Policy Optimization (PPO) can be thought of as an enhancement to the Advantage Actor-Critic (A2C) algorithm. It is crucial to understand that PPO should not be confused with being an improvement to Soft Actor-Critic (SAC). While PPO shares similarities with other reinforcement learning algorithms such as Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic (TD3), and SAC, it has a different lineage and purpose.

PPO and A2C both utilize the actor-critic architecture, where an actor is responsible for making decisions based on the current policy, and a critic estimates the value function to help update the policy. The key distinction between PPO and A2C lies in the way PPO handles policy updates, allowing for multiple gradient steps using the same mini-batch of experiences, resulting in improved sample efficiency and stability.

On the other hand, TD3 and SAC are both offshoots of DDPG, an algorithm that combines ideas from deep learning and reinforcement learning to handle continuous control problems. TD3 is a direct improvement over DDPG, addressing issues such as overestimation bias and instability. SAC, although developed concurrently with TD3, shares several features with it. The second version of the SAC paper even incorporated some of the ideas from TD3. However, it is important to note that SAC is not a direct improvement to TD3; rather, it is a distinct algorithm with a focus on maximizing the entropy of the policy, enabling better exploration.

In summary, Proximal Policy Optimization is an enhancement of the Advantage Actor-Critic algorithm, sharing the same actor-critic architecture and building upon the A2C framework. While PPO shares certain similarities with other reinforcement learning algorithms such as DDPG, TD3, and SAC, it is crucial to recognize that PPO is specifically an improvement to A2C, and not to the other algorithms.

$$\begin{aligned} J\left(\phi, \phi^{-}\right) = \mathbb{E}_{(s_{KF}, a, A^{GAE}) \sim u(D(\phi^{-}))} \Bigg\{ \min \Bigg[& \frac{\pi(a|s_{KF}; \phi)}{\pi(a|s_{KF}; \phi^{-})} A^{GAE}, \\ & clamp\left(\frac{\pi(a|s_{KF}; \phi)}{\pi(a|s_{KF}; \phi^{-})}, 1-\epsilon, 1+\epsilon \right) A^{GAE} \Bigg] \Bigg\} \end{aligned} \tag{2.22}$$

$$\begin{aligned} L\left(\phi, \phi^{-}\right) = \mathbb{E}_{(s_{KF}, a, G, V) \sim u(D(\theta^{-}))} \{ & \max \left[G - V(s_{KF}; \theta), \right. \\ & \left. G - (V + clamp\left(V(s_{KF}; \theta) - V, -\delta, \delta\right)) \right] \} \end{aligned} \tag{2.23}$$

2.2 Introduction to Kalman Filtering Algorithms

A Kalman Filter is able to update and estimate of an evolving state. The goal of the Kalman Filter is to take a probabilistic estimate of an state and update it in real time using two steps, prediction and correction. In the following some extended of Kalman Filter algorithm are introduced in details.

2.2.1 Standard Kalman Filter (SKF)

Introduction

The Kalman Filter requires the following motion and measurement models:

- Motion Model:

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{2.24}$$

  in which u_{k-1} is an external signal that affects the evolution of our system state (e.g. the acceleration in three dimensions to change the speed of a drone).

- Linear Measurement Model:

$$y_k = H_k x_k + v_k \tag{2.25}$$

and the following noise properties are considered for this model:

- Measurement noise:

$$v_k \sim N(0, R_k)$$

- Process or motion noise:

$$w_k \sim N(0, Qk))$$

  which governs how certain we are that our linear dynamic system is actually correct and how uncertain we are about the effects of our control inputs.

The Kalman Filter is a recursive least square estimator that also includes a motion model. The following steps should be taken to find the estimates of the state we are focusing on:

- Prediction:

$$\check{x}_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} \tag{2.26}$$

$$\check{P}_k = F_{k-1}\hat{P}_{k-1}F_{k-1}^T + Q_{k-1} \tag{2.27}$$

- Update the Optimal Gain:

$$K_k = \check{P}_k H_k^T (H_k \check{P}_k H_k^T + R_k)^{-1} \tag{2.28}$$

- Correction:

$$\hat{x}_k = \check{x}_k + K_k(y_k - H_k \check{x}_k) \tag{2.29}$$

$$\check{P}_k = (1 - K_k H_k)\check{P}_k \tag{2.30}$$

in which,

- $y_k - H_k \check{x}_k$ is the innovation matrix
- $\check{P}_k$ is the corrected state covariance matrix
- $\check{x}_k$ is the prediction given motion model at time k
- $\hat{x}_k$ is the corrected prediction give measurement at time k

A smaller state covariance means we are more certain about the drone's position after we incorporate the position measurement and the measurement noise variance is quite small.

Bias in State Estimation

We say an estimator or a filter is unbiased if it produces an *average* error of zero at a particular time step k over many trials. So, we drive our system like a drone for k time steps and record the following data to get the estimation errors and repeat this process.

- Process and Motion:

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1}$$

- Measurement:

$$y_k = H_k x_k + v_k$$

where in 2.3, the P_k is the true position of the drone in 3D environment and $E[\hat{P}_k]$ is the mean of the estimated position values. To compute the bias for the Kalman Filter we have the error dynamics as:

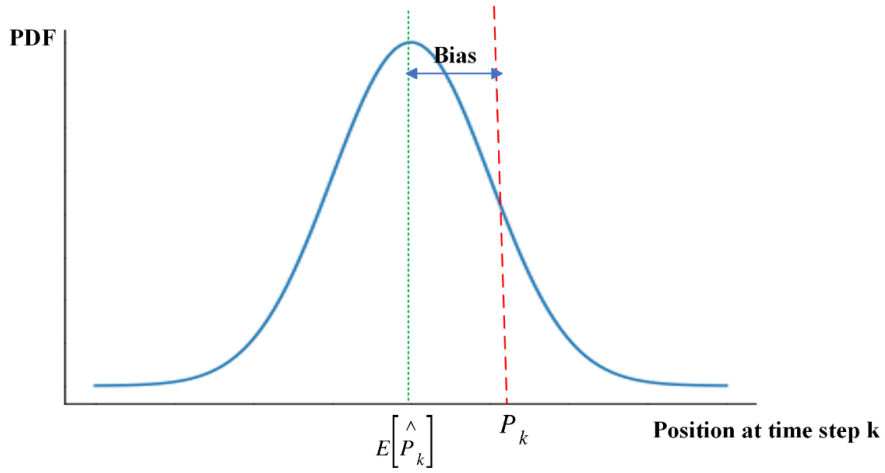

Figure 2.3: Definition of Bias on Probability Distribution Function

- Predicted State Error:

$$\check{e}_k = \check{x}_k - x_k$$

- Corrected Estimate Error:

$$\hat{e} = \hat{x}_k - x_k$$

By using the Kalman Filter equations, we can derive:

- Predicted State Error:

$$\check{e}_k = F_{k-1}\check{e}_{k-1} - w_k$$

- Corrected Estimate Error:

$$\hat{e} = (1 - K_k H_k)\check{e}_k + K_k v_k$$

The expectation values of these errors is equal to zero.For this to be true, we need to ensure that our initial state estimate is unbiased and our noise is white, uncorrelated, and zero mean. So, for the Kalman Filter for all k steps:

- Expectation of the Predicted State Error:

$$E[\check{e}_k] = E[F_{k-1}\check{e}_{k-1} - w_k] = F_{k-1}E[\check{e}_{k-1}] - E[w_k] = 0$$

- Expectation of the Corrected Estimate Error:

$$E[\hat{e}] = E[(1 - K_k H_k)\check{e}_k + K_k v_k] = (1 - K_k H_k)E[\check{e}_k] + K_k E[v_k] = 0$$

This does not mean that the error on a give trial will be zero, but that with enough trials, our expected error is zero.

Consistency in State Estimation

By consistency we mean that for all time steps k, the filter covariance $\hat{P}_k$ matches the expected value of the square of our error. Therefore, a filter is consistent if for all k we have:

$$E[\hat{e}_k^2] = E[(\hat{P}_k - P_k)^2] = \hat{P}_k \tag{2.31}$$

The empirical variance of our estimate should match the variance reported by the filter. This means that our filter is neither overconfident, nor underconfident in the estimate it has produced. A filter that is over confident, and hence inconsistent, will report a covariance that is optimistic. Therefore, the filter will essentially place too much emphasis on its own estimate will be less sensitive to future measurement updates which may provide critical information. So long as our initial estimate is consistent and we have white zero mean noise, then all estimates will be consistent.

$$\left.\begin{array}{r} E[\hat{e}_0\hat{e}_0^T] = \check{P}_0, E[v] = 0 \\ E[w] = 0, \text{White noise} \end{array}\right\} \Rightarrow E[\check{e}_k\check{e}_k^T] = \check{P}_k, E[\hat{e}_k\hat{e}_k^T] = \hat{P}_k \tag{2.32}$$

We have shown that given our linear formulation, and zero mean white noise, the Kalman Filter is unbiased. We can also say that the filter is consistent if:

$$E[\hat{e}_k] = 0$$

$$E[\hat{e}_k\hat{e}_k^T] = \hat{P}_k$$

In general, if we have white, uncorrelated zero mean noise, the Kalman Filter is the best(Lowest Variance) unbiased estimator that uses only a linear combination of measurements. Therefore, the Kalman Filter is the best linear unbiased estimator.

2.2.2 Extended Kalman Filter (EKF)

Introduction

The Extended Kalman Filter uses the first-order linearization to turn a nonlinear problem into a linear one. This algorithm talks about the role of the Jacobian matrices and how to compute them. Actually, Linear systems do not exist in reality. So, there is a question about how we can adapt the Kalman Filter algorithm to nonlinear discrete-time systems?

We have:

$$x_k = f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.33}$$

$$y_k = h_k(x_k, v_k) \tag{2.34}$$

in which the x_k is the dynamic of the system and y_k is the measurement model.

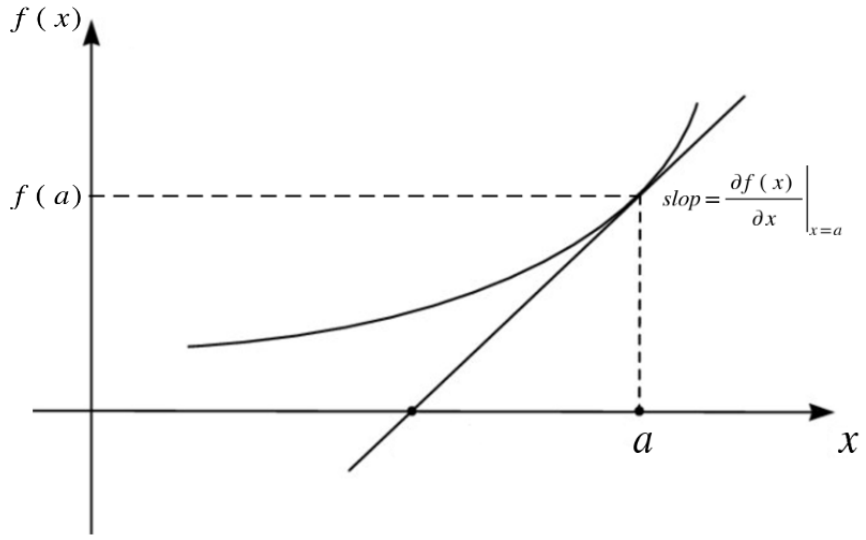

Figure 2.4: Nonlinear system and the meaning of first order linearization

To linearize a nonlinear system, firstly a point should be chosen like point a, and the approximation of the nonlinear function by a tangent line at that point should be derived. So, the EKF can be defined as the Linearized Kalman Filter.

Suppose a nonlinear $f(x)$ function is given. Mathematically, the linear approximation is computed using a first-order Ttaylor expansion as:

$$f(x) \approx f(a) + \frac{\partial f(x)}{\partial x}\Big|_{x=a}(x-a) + \frac{1}{2!}\frac{\partial^2 f(x)}{\partial x^2}\Big|_{x=a}(x-a)^2 + \frac{1}{3!}\frac{\partial^3 f(x)}{\partial x^3}\Big|_{x=a}(x-a)^3 + \ldots \tag{2.35}$$

and the first-order Taylor expansion would be as:

$$f(x) \approx f(a) + \frac{\partial f(x)}{\partial x}\Big|_{x=a}(x-a)$$

Algorithm

The procedure to derive the equations of EKF is going to be explained in details in the following.

- Linearize Motion Model:
  For the EKF, the operating point is chosen as the most recent state estimated point with known input and zero noise. Therefore, we have linearized motion model for EKF as follows:

$$\begin{aligned} x_k &= f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}) \approx f_{k-1}(\hat{x}_{k-1}, u_{k-1}, 0) \\ &+ \frac{\partial f_{k-1}}{\partial x_{k-1}}\Big|_{(\hat{x}_{k-1}, u_{k-1}, 0)} (x_{k-1} - \hat{x}_{k-1}) \\ &+ \frac{\partial f_{k-1}}{\partial w_{k-1}}\Big|_{(\hat{x}_{k-1}, u_{k-1}, 0)} w_{k-1} \end{aligned} \tag{2.36}$$

  the two last terms are named as:

$$F_{k-1} = \frac{\partial f_{k-1}}{\partial x_{k-1}}\Big|_{(\hat{x}_{k-1}, u_{k-1}, 0)}$$

$$L_{k-1} = \frac{\partial f_{k-1}}{\partial w_{k-1}}\Big|_{(\hat{x}_{k-1}, u_{k-1}, 0)}$$

- Linearize Measurement Model:
  the linearized measurement model will be as the following:

$$\begin{aligned} y_k &= h_k(x_k, v_k) \approx h_k(\check{x}_k, 0) \\ &+ \frac{\partial h_k}{\partial x_k}\Big|_{(\check{x}_k, 0)} (x_k - \check{x}_k) + \frac{\partial h_k}{\partial v_k}\Big|_{(\check{x}_k, 0)} v_k \end{aligned} \tag{2.37}$$

  and the two last terms are named as:

$$H_k = \frac{\partial h_k}{\partial x_k}\Big|_{(\check{x}_k, 0)}$$

$$M_k = \frac{\partial h_k}{\partial v_k}\Big|_{(\check{x}_k, 0)}$$

- Computing the Jacobian Matrices:
  Now, we have linear system in state-space and the $F_{k-1}, L_{k-1}, H_k, M_k$ are called the Jacobian matrices of the system. A Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function.

$$\frac{\partial f}{\partial x} = [\frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_n}] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.38}$$

Consider each column of the Jacobian matrix contains the derivatives of the function outputs with respect to a given input. Intuitively, the Jacobian matrix tells us how fast each output of the function is changing along each input dimensions.

- Write the final equations:
  With the linearized models and Jacobians, we can now use the Kalman Filter equations which derived as:

$$x_k = f_{k-1}(x_{k-1}, u_{k-1}, 0) + F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + L_{k-1}w_{k-1}$$

$$y_k = h_k(\check{x}_k,0) + H_k(x_k - \check{x}_k) + M_k v_k$$

  – Prediction:

$$\check{x}_k = f_{k-1}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.39}$$

$$\check{P}_k = F_{k-1}\hat{P}_{k-1}F_{k-1}^T + L_{k-1}Q_{k-1}L_{k-1}^T \tag{2.40}$$

  – Optimal Gain:

$$K_k = \check{P}_k H_k^T (H_k \check{P}_k H_k^T + M_k R_k M_k^T)^{-1} \tag{2.41}$$

  – Correction:

$$\hat{x}_k = \check{x}_k - K_k(y_k - h_k(\check{x}_k,0)) \tag{2.42}$$

$$\hat{P}_k = (1 - K_k H_k)\check{P}_k \tag{2.43}$$

  in which:
  $\check{x}_k$ is the prediction given motion model at time k
  $\hat{x}_k$ is the corrected prediction give measurement at time k

  Consider the motion model is linearized about the previous state estimate and the measurement model is linearized about the predicted state.

2.2.3 Single-Instruction Multi-data Kalman Filter

The Kalman filter is a mathematical algorithm that provides an efficient computational solution to estimate the state of a process in a way that minimizes the mean of the squared error. It is used in a wide range of engineering [40] and data analysis applications to filter out noise and provide accurate data about the state of a system over time.

The Kalman filter becomes even more powerful when integrated with the SIMD concept. The SIMD (Single Instruction, Multiple Data) Kalman filter leverages the power of modern processors to perform multiple operations simultaneously. By packing several data items into one register and operating on all of them simultaneously, the SIMD Kalman filter significantly increases the data processing speed. This approach is particularly beneficial in scenarios where operations can be naturally parallelized, such as in high-energy physics experiments where large volumes of data need to be processed quickly. Using the SIMD Kalman filter, these operations can be performed much faster, leading to more efficient data processing and analysis.

The SIMD (Single Instruction, Multiple Data) Kalman filter works by leveraging the power of modern processors to perform multiple operations simultaneously. The SIMD Kalman filter can process multiple data streams against a single instruction stream, particularly beneficial in scenarios where operations can be naturally parallelized. This makes it especially useful in high-energy physics experiments, where large volumes of data need to be processed quickly. The SIMD (Single Instruction, Multiple Data) Kalman filter operates in several steps to leverage the power of modern processors for efficient data processing. Initialization: The algorithm begins with an initial approximation of the system's state. This approximation is represented as a vector, and its covariance matrix is set to a large positive number, indicating a high level of uncertainty. Prediction: The algorithm predicts the state of the system at the next time step. This prediction is based on the current state and the system's dynamics, represented by the prediction matrix. The prediction step produces an estimated state vector and its associated covariance matrix, which represents the uncertainty of the prediction. Process Noise: This step accounts for the probabilistic deviations in the system's state due to process noise. The estimated state vector and its covariance matrix are updated to reflect this noise. Filtration: The state vector is updated with the new measurement to get the optimal estimate of the system's state and its covariance matrix. The Kalman gain matrix, which determines how much weight to give to the new measurement based on its uncertainty, is calculated and applied. The total deviation of the obtained estimation from the measurements is also calculated. The filter's complete mathematical formulations are reported in [41].

In the context of this paper, the SIMD form of the Kalman filter has been used to filter the information about the position of the gates. This did allow us to obtain, after a small number of observations, an accurate estimate of the gate's movement. The downstream algorithm later used this information to estimate the position of the gate at a given moment in time.

In the study discussed in this paper, the SIMD version of the Kalman filter has been employed to filter the data related to the position of the gates. This application enabled us to acquire a precise estimate of the gate's motion after a limited number

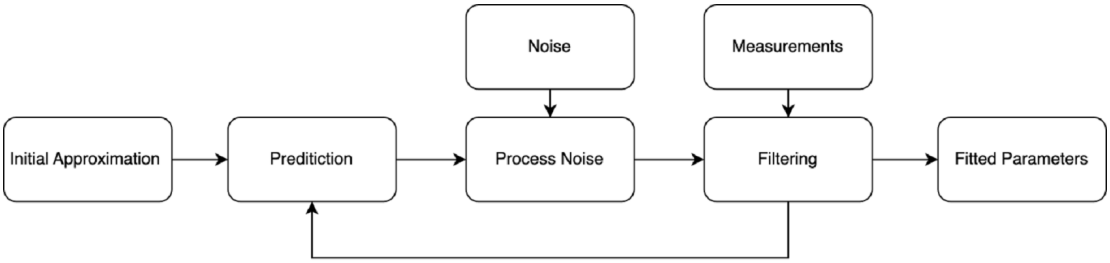

Figure 2.5: Kalman Filtering Block Diagram

of observations. The downstream algorithm subsequently utilized this information to approximate the gate's position at a specific time point. In the prediction step of the Kalman filter, an estimate of the new position of the gate is made based on the previous state. This step incorporates the state transition model and the control input, if any, to predict the current state of the gate. This prediction forms the prior estimate for the state of the system. The observations from the environment then come into play during the update step. These observations are used to adjust the prediction based on what was measured, which can be particularly beneficial when there is uncertainty or noise in the system. In this case, the measurements pertain to the gate's movement. The difference between the prediction from the prior state and the observation from the environment, also known as the innovation or residual, is then used to update the state estimate. This updated estimate is a weighted average of the prior estimate and the current measurement, with more weight given to estimates with more certainty. Through several iterations, the above process yields an accurate estimate of the gate's movement, effectively tracking the gate's position over time. This ability to use observations to predict and correct the position estimate makes the Kalman filter an excellent tool for dealing with the dynamics of moving objects like the gate in our study.

Chapter 3

Simulation Environment and Experimental Setup

3.1 Introduction

3.2 Two Dimensional Drone Environment

In the design of this environment, a specific object-oriented programming paradigm was utilized to model the relationships and interactions between various elements in the system. The primary focus was to create a modular and scalable architecture that would allow for easy modification and extension.

To achieve this, a class hierarchy was established, allowing for the separation of concerns and encapsulation of data. The classes were structured in such a way that common functionality and properties were inherited from a base class, and specialized behavior was implemented in derived classes. This structure facilitated code reusability and maintainability.

Additionally, to store and manage the data associated with each object, appropriate attributes were defined within the classes. These attributes allowed for the proper representation of the object's state and properties while maintaining data integrity and minimizing redundancy. Access to these attributes was controlled through carefully designed methods, ensuring that external entities could only interact with the object's data in a safe and consistent manner.

To enable seamless communication and collaboration between objects, well-defined interfaces, and methods were employed. This approach allowed for the efficient exchange of data and execution of tasks, promoting a high degree of cohesion and reducing coupling between objects. This modular design facilitated the ease of testing and debugging the software, ensuring that each component functioned correctly and efficiently within the larger system.

Overall, the careful design and implementation of classes, attributes, and methods within the software contributed to a robust and efficient solution that met the project's requirements and provided a solid foundation for future development and improvement.

The environment is a 2D drone game where the drone navigates through two gates. The environment is created as a class called *DroneGateGame*, which initializes various attributes like the drone's position, gate positions, field of view (fov), and dimensions of the environment (w and h). There are eight scenarios in which the drone and gates have different behaviors.

- Fixed starting point of drone and static gates with discrete state-action space
- Random starting point of drone and static gates with discrete state-action space

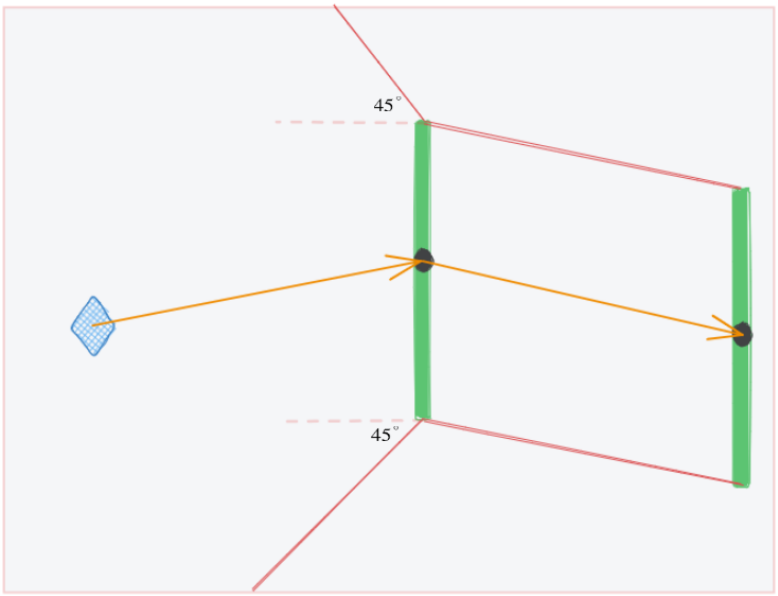

Figure 3.1: Sample of different Starting Points (Orange points) and path of gates at different time steps

3.2.1 State Representation

The state of the drone is represented by its relative position and velocity with respect to the center of each gate in the environment plan. The state space in the cases of having static gates is two-dimensional (x, y), and in the cases of having dynamic gates is three-dimensional $(x, y, \dot{y})$ in which the relative velocity of the gate with respect to the drone along y direction is included.

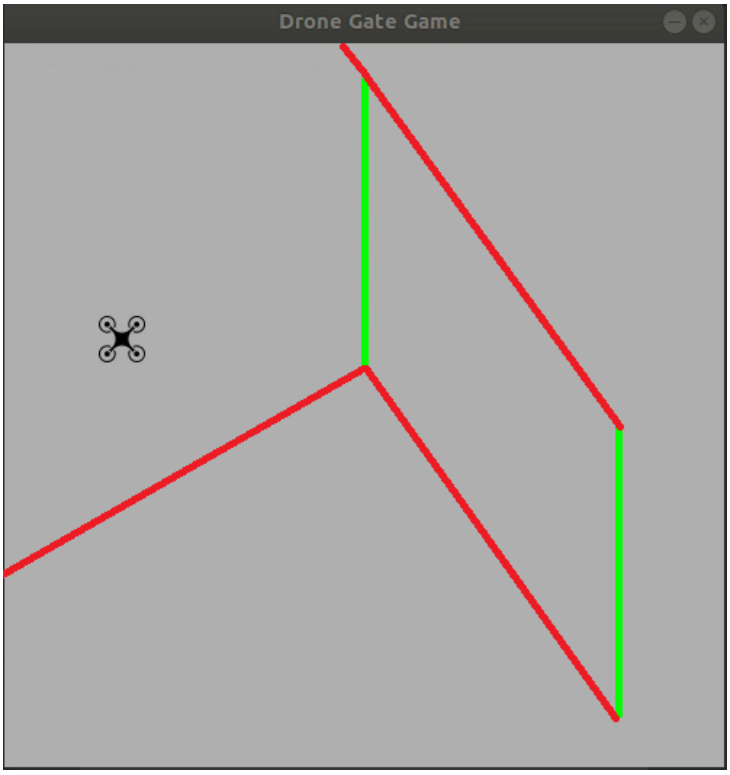

Figure 3.2: Sample of different Starting Points (Orange points) and path of gates at different time steps

3.2.2 Action Space

The action space in the environment with discrete state-action space consists of four possible actions: moving up, down, left, or right. The actions are defined by the integer values 0, 1, 2, and 3, representing $right$, $left$, $down$, and up, respectively. While in the case of continuous action-state space, the action space consists of one action which is the angle by which the drone should move alongside by a unit magnitude.

3.2.3 Reward Function

The reward function is designed to encourage the drone to move closer to the gates. It takes into account the magnitude of the distance between the drone and the center of the gate before and after taking a step. If the drone gets closer to the gate center, the reward will be positive; otherwise, it will be negative. Furthermore, if the drone hits the boundaries, which include both the environment's boundaries and locations where the gate centers are out of the camera's field of view, the

reward will be a significantly negative value at that step.

3.3 Three Dimensional Drone Environment

The environment is a 3D drone game where the drone navigates through two gates. The environment is created as a class called DroneGateGame, which initializes various attributes like the drone's position, gate positions, field of view (fov), and dimensions of the environment (w and h). There are eight scenarios in which the drone and gates have different behaviors.

- Fixed starting point of drone and static gates with continuous state-action space
- Fixed starting point of drone and dynamic gates with continuous state-action space
- Random starting point of drone and static gates with continuous state-action space
- Random starting point of drone and dynamic gates with continuous state-action space

3.3.1 State Representation

In a three-dimensional environment, the investigation is focused on continuous action states. Similar to two-dimensional environments, various behaviors are taken into account for gates and different initial conditions.

In the case of static gates, the state is represented as (x, y, z), where x, y, and z denote the relative positions of the drone concerning the gate in the respective x, y, and z directions. This representation provides sufficient information for navigating the drone in a 3D environment with fixed gate locations.

However, when considering an environment with dynamic gates, the state representation must be expanded to include additional information about the relative velocities in the y and z directions. In this case, the state is represented as $(x, y, z, \dot{y}, \dot{z})$. The additional components, $\dot{y}$ and $\dot{z}$, account for the relative velocity in the y and z directions, respectively, enabling the drone to react to moving gates more effectively.

By considering both static and dynamic gates in a three-dimensional environment, the analysis becomes more comprehensive and adaptable to different scenarios. This approach allows for a more robust understanding of drone navigation and control, which can be beneficial for applications such as drone racing, search and rescue missions, and environmental monitoring, where the ability to maneuver through complex environments is crucial.

3.3.2 Action Space

The action space for both static and dynamic gates encompasses the magnitude of movement in the x, y, and z directions. Each of these values falls within the range of $[-1, 1]$, representing the displacement in each respective direction. By normalizing the sum of all the movements, the magnitude of the total displacement at each step is equal to one unit.

In this action space, the drone can move in any direction within the three-dimensional environment by combining the appropriate magnitudes for each axis. This flexible approach allows the drone to navigate and adapt to various scenarios, whether it involves static or dynamic gates. By using a normalized range for the magnitudes, the system ensures consistency and comparability across different movement directions and environments.

This action space representation is well-suited for applications in drone navigation and control, as it enables the system to respond effectively to a wide range of situations. The normalization of the movement magnitudes ensures a consistent and comparable scale for evaluating the drone's performance across different scenarios.

3.3.3 Reward Function

The reward function is designed to encourage the drone to move closer to the center of the gate at each time step. The reward is calculated based on the amount of displacement at each step multiplied by the sign of the displacement. In other words, the relative position concerning the center of the gate before and after taking a step is considered. A higher positive reward is granted when the drone moves closer to the center of the gate, while a more significant negative reward is given when the drone moves further away from the center. Moreover, if the drone hits the boundaries, which include both the environment's boundaries and locations where the gate centers are out of the camera's field of view, the reward will be a substantially negative value at that step. The formula for the reward function is as follows:

$$Reward(t) = \begin{cases} -30 & \text{if Collision to Boundaries} \\ +30 & \text{if Reach Latest Gate Center} \\ d_{t-1}^{rel}(s_{t-1}, a_{t-1}) - d_t^{rel}(s_t, a_t) & \text{Otherwise} \end{cases}$$

in which,

- $Reward(t)$ represents the reward at time t.
- $d_t^{rel}(s, a)$ is the relative distance at time t given the state s and action a.

- s_t and a_t are the state and action at time t, respectively.
- s_{t-1} and a_{t-1} are the state and action at the previous time step, $t-1$.

By utilizing this reward function, the drone's navigation system is encouraged to optimize its path toward the center of the gate, thereby improving its overall performance in various environments. This reward structure is particularly useful in applications such as drone racing, search and rescue missions, and environmental monitoring, where precise navigation and control are essential.

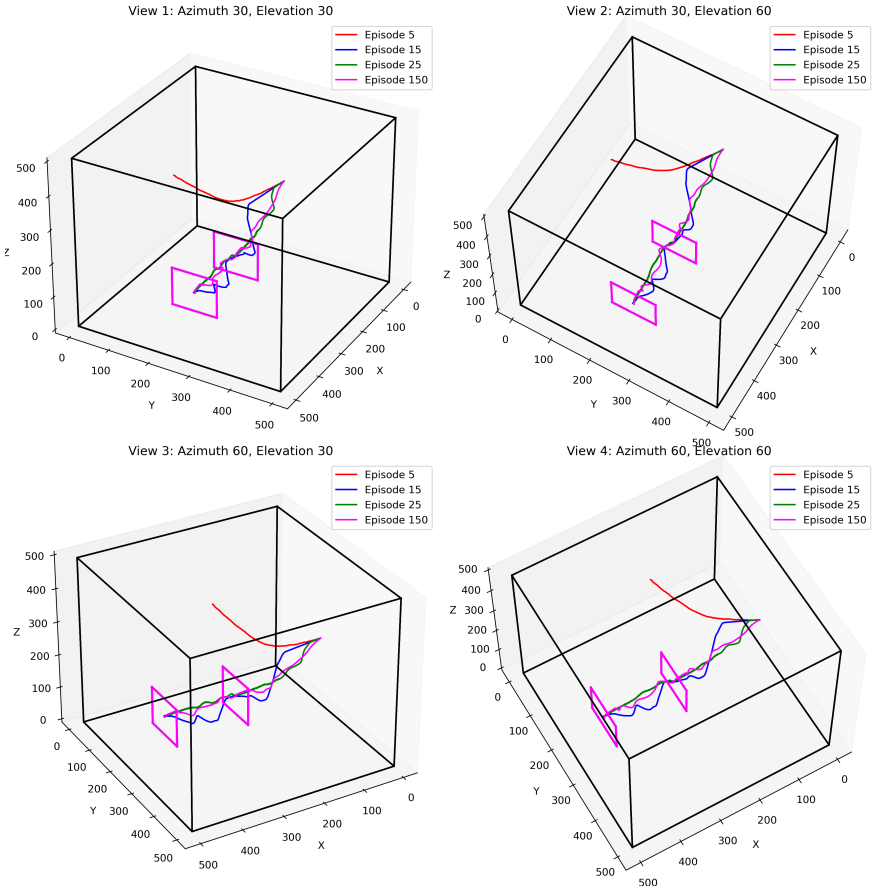

Figure 3.3: Sample of Movement of the Drone in 3D environment at different training episode in Fixed Starting Points

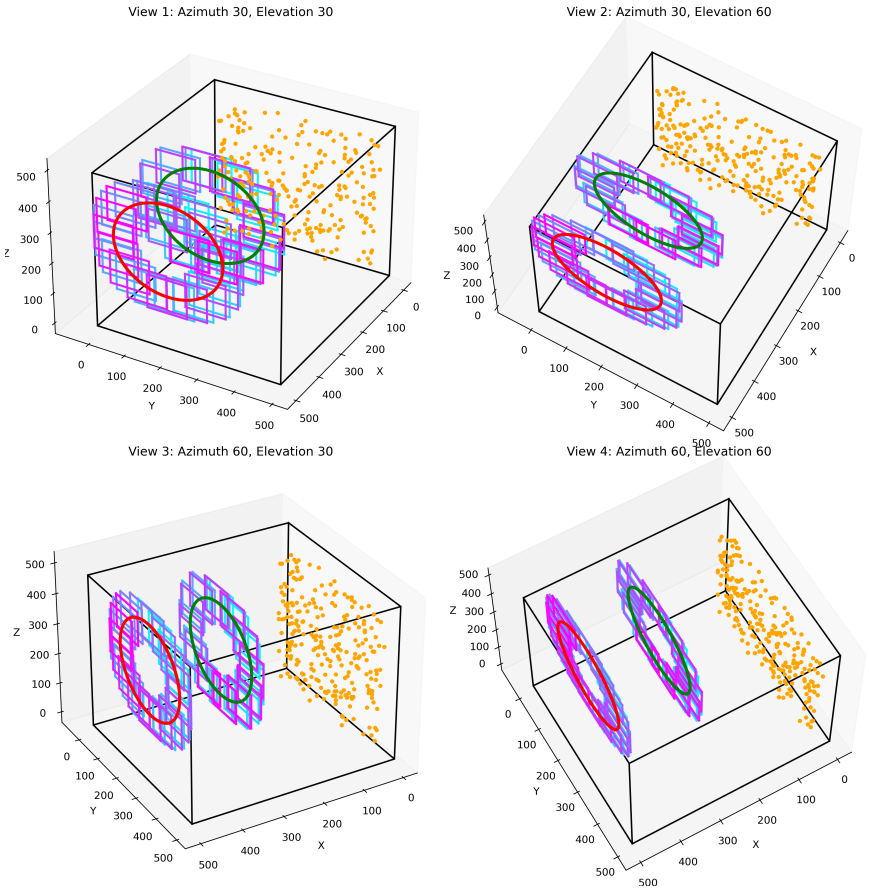

Figure 3.4: Sample of different Starting Points (Orange points) and path of gates at different time steps

Chapter 4

A Kalman Filter Reinforcement Learning Approach for Path Planning

4.1 Kalman Filter for State Estimation

The Kalman Filter Algorithm employed in this environment is the Standard Kalman Filter, which assumes the problem to be not overly nonlinear. As we are dealing with a simple inverted pendulum, and the characteristics of the pendulum change at each episode, the system can be considered relatively simple. The advantage of using a standard Kalman filter lies in its ability to provide a good trade-off between the accuracy of estimations and the complexity of the model from a computational perspective.

The Kalman Filter consists of two steps: prediction and update. The input for the prediction step is the time at which the drone is expected to reach the center of the gate. This time can be approximated using the average step time and the distance to the center of the gate. The average step time is dependent on the computational load of the algorithm as well as the processing power of the system. In other words, the average step time represents the average duration it takes for the agent to make a step. By considering these two parameters, the time to reach the gate center can be determined, and the outputs of the prediction step, which include the relative positions and relative velocities of the gate with respect to the drone along the x, y, and z directions, can be calculated.

Moreover, the inputs of the update step are the relative positions of the gate with respect to the drone (x, y, z), and the outputs are the estimations of the gate's future position concerning the drone at the time calculated in the previous step.

It is important to note that the states of the drone for Reinforcement Learning algorithms are precisely the outputs of the update step at each time step.

By incorporating the Standard Kalman Filter into the drone's navigation system, it can more accurately estimate its position and velocity relative to the gate, leading to improved performance in various environments.

4.2 Reinforcement Learning Algorithms with Kalman Filter Integration 2D

In this section, different Reinforcement Learning algorithms are trained to evaluate and compare the performance of each algorithms for two different scenarios. The first scenario includes all the initial conditions to be known more specifically the starting point of the drone, whereas in the second scenario, the initial starting point of the drone is selected randomly at each episode. So, the second scenario will consist of more challenges and it will be more generalized rather than the first one.

4.2.1 Deep Deterministic Policy Gradient

Training and Hyperparameters

In all the following Training and Hyperparameters sections, most of the parameters are considered to be constant to have a better comparison among the nature of the algorithms by themselves. As you will see, the ***gamma*** coefficient which is the discount factor that adjusts the rewards importance over time in all the algorithms is considered to be 0.99. Also, the ***nS*** factor for the environments including static and dynamic gates is 2 and 3, respectively. And ***nA*** factors in 2D discrete action state environments is 4 which are the movement directions that are Up, $Down$, $Right$, and $Left$.

Moreover, the complexity of the networks are determined by *hidden_dims* factors and because of that relative advanced neural network, the *policy_optimizer_lr* and *value_optimizer_lr* factors are investigated to be small values. The point should be noticed here is that *policy_optimizer_lr* factor considered lower than *value_optimizer_lr* value to have less intervention of the disturbances and noises to the policy network.

Performance Evaluation

Because of the simplicity of the 2D environments as well as the discret action state environments, the training episodes to get the highest reward at each episode is not too much.

Listing 4.1: Parameter Settings for DDPG Algorithm in 2D Environment

```
gamma: 0.99
policy_model_fn = lambda nS, bounds: FCDP(nS, bounds, hidden_dims
    =(256, 256))
policy_max_grad_norm = float('inf')
policy_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
     lr=lr)
policy_optimizer_lr = 0.00003
value_model_fn = lambda nS, nA: FCQV(nS, nA, hidden_dims=(256,
    256))
value_max_grad_norm = float('inf')
value_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
    lr=lr)
value_optimizer_lr = 0.0003
training_strategy_fn = lambda bounds: NormalNoiseStrategy(bounds,
    exploration_noise_ratio=0.1)
replay_buffer_fn = lambda: ReplayBuffer(max_size=100000,
    batch_size=512)
n_warmup_batches = 1
update_target_every_steps = 1
tau = 0.001
```

As shown in the Episode score graph and in the plotted trajectories, the agent learns faster to reach the center of the gates in the first scenario which is the fixed starting point. That is the reason why the trajectories for most of the episodes in the case of having static gates are similar to each other. But, as the randomness to the environment increases, the learning would be more challenging, the trajectories would be different, and more episodes is needed learn the desired policy.

4.2.2 Dueling Double DQN

Training and Hyperparameters

In this algorithm, the ***nS*** and ***nA*** parameters considered same as the previous algorithm (DDPG), but the network is more complicated by increasing the neurons at each hidden layer. Also the ***lr*** parameter is considered to be changed based on an Exponential rate which leads to have more stable learning in most of the cases.

Performance Evaluation

As you can see, this algorithm is not too much dependent on the environment variables and the required episodes to learn the policy for the agent in two scenarios are almost the same. But as you can see, on the up right plot corresponding to

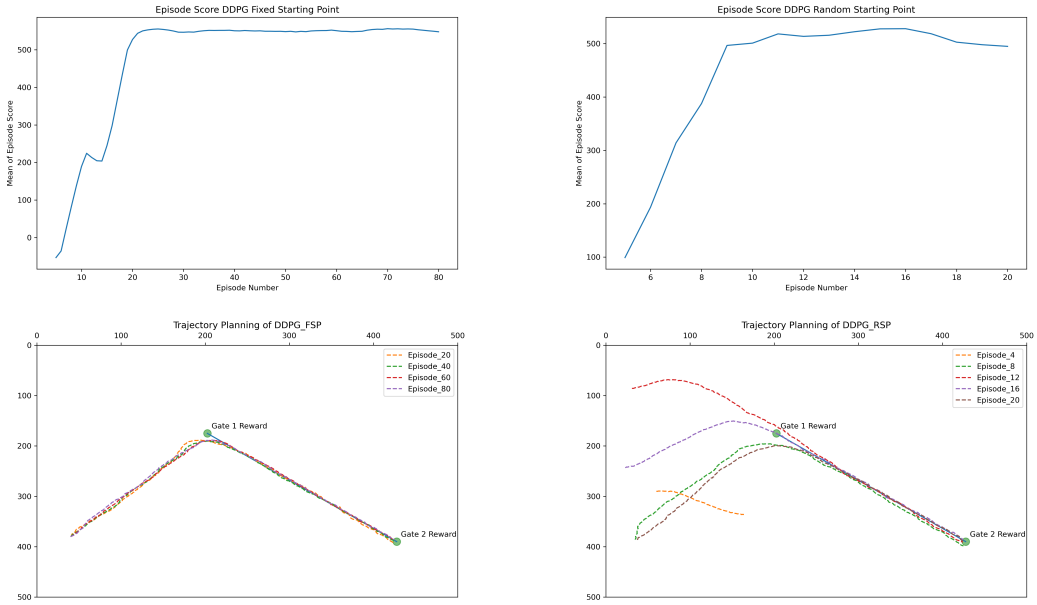

Figure 4.1: Episode Scores and Trajectories of DDPG Algorithm with Fixed and Random Starting Points

Listing 4.2: Parameter Settings for D3QN Algorithm in 2D Environment

```
gamma: 0.995
value_model_fn = lambda nS, nA: FCDuelingQ(nS, nA, hidden_dims
    =(512,512))
value_optimizer_fn = lambda net, lr: optim.RMSprop(net.parameters
    (), lr=lr)
value_optimizer_lr = lambda: LR_ExpStrategy(init_lr= 0.0005,
    min_lr=0.0002, decay_steps=100000)
max_gradient_norm = float('inf')
training_strategy_fn = lambda: EGreedyExpStrategy(init_epsilon
    =1.0,
min_epsilon=0.05,decay_steps=100000)
replay_buffer_fn = lambda: ReplayBuffer(max_size=80000, batch_size
    =512)
n_warmup_batches = 1
update_target_every_steps = 1
tau = 0.01
```

Episode Score for Random Starting Point, the reward is oscillating in range of [450,550] which caused by the length of the path the agent should take. This fact

is admitted by its lower trajectory that shows the path is taken almost perfectly in different episodes.

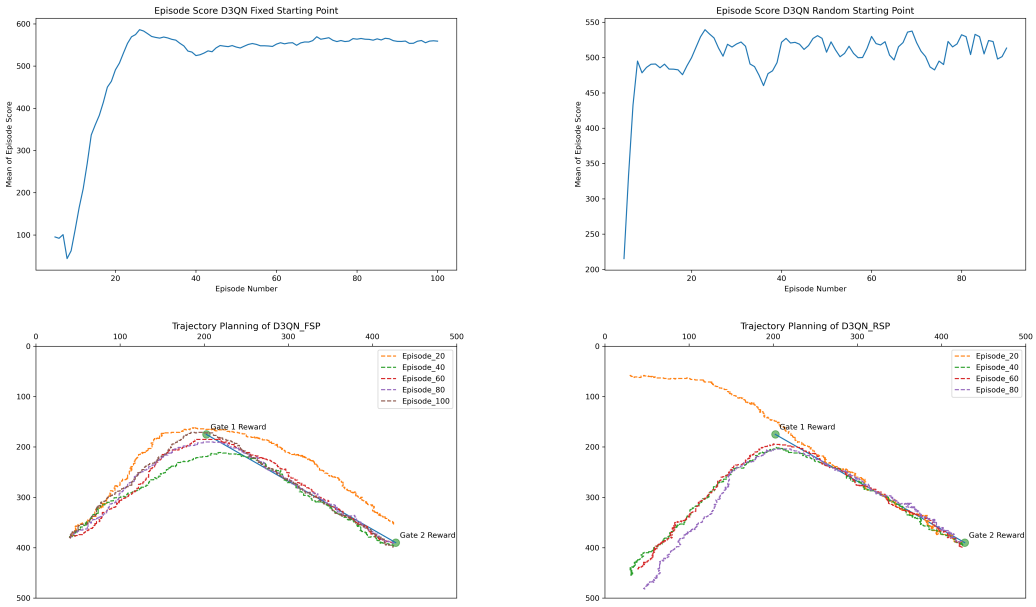

Figure 4.2: Episode Scores and Trajectories of D3QN Algorithm with Fixed and Random Starting Points

4.2.3 Prioritized Experience Replay

Training and Hyperparameters

As the previous algorithms, most of the parameters are the same as those ones. in $PrioritizedReplayBuffer$, the $rank\_based$ hyperparameter is considered to be False. If it is set to $True$, it mean the priority order is based on the following equation:

$$p_i = \frac{1}{rank(i)}$$

Otherwise, if that hyperparameter is set to $False$, the prioritization would be proportional which is the absolute TD error plus a small constant epsilon to avoid zero priorities as the following equations:

$$p_i = |\delta_i + \epsilon|$$

. Moreover, the *alpha* coefficient is used in the following formula to get the probabilities from priorities:

$$p_i = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

. Finally, the *beta* hyperparameter is implemented in the following equation to get the importance=sampling weights using the probabilities.

$$w_i = (NP(i))^{-\beta}$$

in the above equation, the importance-sampling based is calculated by multiplying each probability by the number of samples in the replay buffer and then powered by $-\beta$. and then the weights are scaled down to have the largest weight be equal to one.

$$w_i = \frac{w_i}{max_j(w_i)}$$

Listing 4.3: Parameter Settings for PER Algorithm in 2D Environment

```
gamma: 0.99
value_model_fn = lambda nS, nA: FCQ(nS, nA, hidden_dims=(512, 512)
    )
value_optimizer_fn = lambda net, lr: optim.RMSprop(net.parameters
    (), lr=lr)
value_optimizer_lr = lambda: LR_ExpStrategy(init_lr= 0.0005,
    min_lr=0.0002, decay_steps=100000)
max_gradient_norm = float('inf')
training_strategy_fn = lambda: EGreedyExpStrategy(init_epsilon
    =1.0,
min_epsilon=0.1,
decay_steps=150000)
replay_buffer_fn = lambda: PrioritizedReplayBuffer(
max_samples=80000, batch_size=6000, rank_based=False,
alpha=0.6, beta0=0.1, beta_rate=0.99997)
n_warmup_batches = 1
update_target_every_steps = 1
tau = 0.01
```

Performance Evaluation

As is evident from Figure 4.3, the algorithm successfully reaches the desired results in the Fixed Starting Point case. However, in the case of having a Random Starting Point, it exhibits instability. It appears that this algorithm has the poorest performance among the mentioned algorithms in 2D discrete action environments.

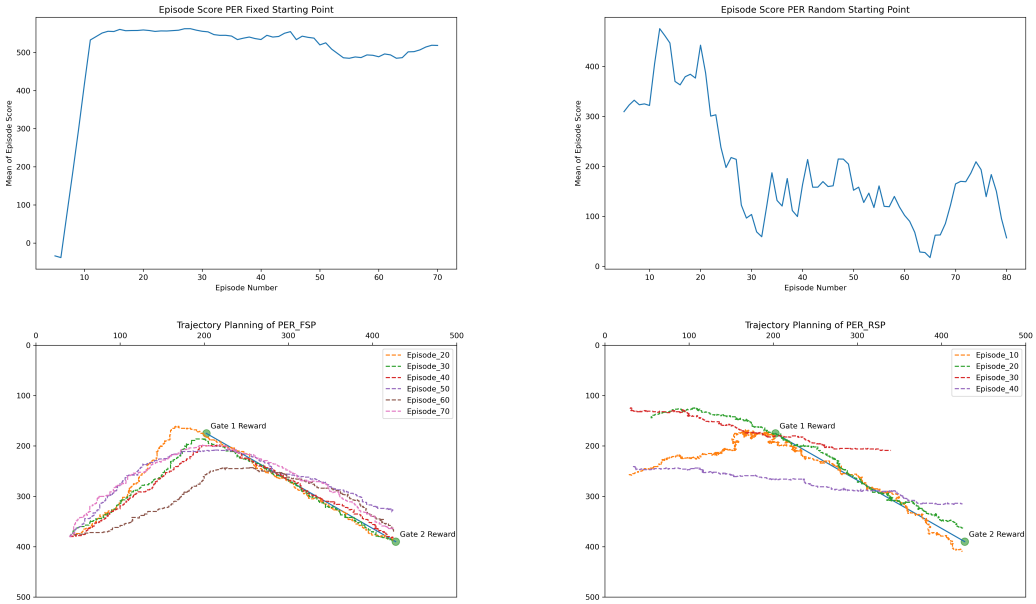

Figure 4.3: Episode Scores and Trajectories of PER Algorithm with Fixed and Random Starting Points

4.2.4 Proximal Policy Optimization

Training and Hyperparameters

In the following, the hyperparameters of the PPO algorithm are defined. As it is evident, the PPO algorithm has more hyperparameters than the other discussed algorithms. This leads to the PPO algorithm being more of a hyperparameter tuning problem.

Two hyperparameters *policy_stopping_kl* and *value_stopping_mse* are the conditions which determine the policy and value optimization networks to be done or ignored. Also, the *policy_optimization_epochs* and *value_optimization_epochs* are the hyperparameters which determine for how many time the policy and value networks being optimized. Moreover the ***sd***, ***g***, ***t***, ***me***, and ***mes*** hyperparametrs are the state dimension, gamma factor, tau factor, maximum episodes, and maximum steps at each epiode, respectively. And finally, the *policy_sample_ratio* and *value_sample_ratio* are the ones determine the size of the batch whcih are going to be used for policy and value functions at each *policy_optimization_epochs* and *value_optimization_epochs*, respectively.

Listing 4.4: Parameter Settings for PPO Algorithm in 2D Environment

```
gamma: 0.99
policy_model_fn = lambda nS, nA: FCCA(nS, nA, hidden_dims=(256, 256))
policy_model_max_grad_norm = float('inf')
policy_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(), lr=lr)
policy_optimizer_lr = 0.0003
policy_optimization_epochs = 80
policy_sample_ratio = 0.8
policy_clip_range = 0.1
policy_stopping_kl = 0.02
value_model_fn = lambda nS: FCV(nS, hidden_dims=(256, 256))
value_model_max_grad_norm = float('inf')
value_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(), lr=lr)
value_optimizer_lr = 0.0005
value_optimization_epochs = 80
value_sample_ratio = 0.8
value_clip_range = float('inf')
value_stopping_mse = 25
episode_buffer_fn = lambda sd, g, t, me, mes: EpisodeBuffer(sd, g, t, me, mes)
max_buffer_episodes = 16
max_buffer_episode_steps = 1000
entropy_loss_weight = 0.01
tau = 0.01
```

Performance Evaluation

As is evident in the Episode Reward plots and the corresponding trajectories, the algorithm reaches the desired reward in Fixed Starting Points scenarios. However, it takes more time and more episodes to reach an optimal policy in Random Starting Point scenarios. It should also be considered that through hyperparameter tuning, it might achieve better performance under the same conditions.

4.3 Reinforcement Learning Algorithms with Kalman Filter Integration, 3D

In this section, three well-known Reinforcement Learning (RL) algorithms are investigated and analyzed in the context of the environment discussed in the previous section. Each algorithm is examined in detail, focusing on its suitability and performance within the specified environment, taking into account the unique

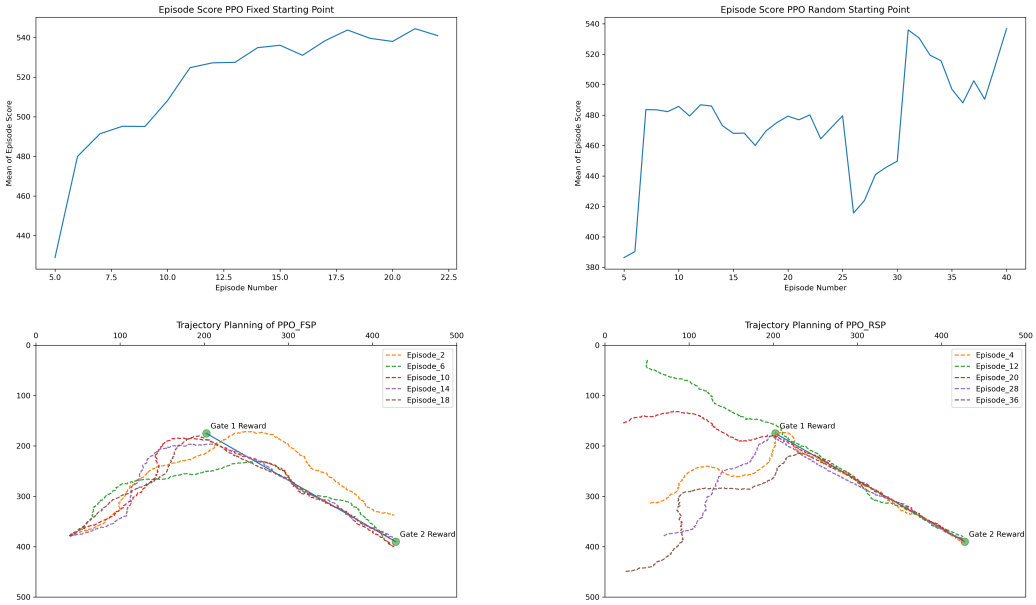

Figure 4.4: Episode Scores and Trajectories of PPO Algorithm with Fixed and Random Starting Points

characteristics and challenges presented by the drone navigation task.

4.3.1 Deep Deterministic Policy Gradient

Algorithm Integration

It is a deterministic policy algorithm and so it uses off-policy exploration strategy. For example, the Gaussian noise is added to the action-selection process that enhances the exploratory of deterministic policies.

Training and Hyperparameters

In the following Training parameters, the *FCQV* and *FCDP* are the Q-function network and policy network, respectively. The output of *FCQV* network is just a single value that represents the value of the state-action pair. Also consider the activation function of the last layer of the *FCDP* network is considered as *tanh* to have an action between $[-1,1]$. In the training of the agent with this algorithm *NormalNoiseStrategy* function is considered to add noise to the action which leads to increase the exploration of the agent in the environment. Then the action is clipped to be in the desired action range and finally, the noise ratio is changed

Algorithm 1 Deep Deterministic Policy Gradient (DDPG)

Initialize actor network π and critic network Q with random weights θ^{π}, θ^{Q}
Initialize target networks π' and Q' with weights $\theta^{\pi'} \leftarrow \theta^{\pi}$, $\theta^{Q'} \leftarrow \theta^{Q}$
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 $a_t = \pi(s_t|\theta^{\pi}) + N_t$ 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 $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^{Q}))^2$
        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^{\pi}} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^{\pi}} \pi(s|\theta^{\pi})|_{s_i}$$

        Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\pi'} \leftarrow \tau\theta^{\pi} + (1-\tau)\theta^{\pi'}$$

    end for
end for

based on the designed schedule inside the function. This schedule can be constant, linear, or exponential.

Listing 4.5: Parameter Settings for DDPG Algorithm in 3D Environment

```
gamma: 0.99
policy_model_fn = lambda nS, bounds: FCDP(nS, bounds, hidden_dims
    =(128, 128))
policy_max_grad_norm = float('inf')
policy_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
     lr=lr)
policy_optimizer_lr = 0.0003
value_model_fn = lambda nS, nA: FCQV(nS, nA, hidden_dims=(128,
    128))
value_max_grad_norm = float('inf')
value_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
    lr=lr)
value_optimizer_lr = 0.0005
training_strategy_fn = lambda bounds: NormalNoiseStrategy(bounds,
    exploration_noise_ratio=0.1)
replay_buffer_fn = lambda: ReplayBuffer(max_size=1000000,
    batch_size=256)
n_warmup_batches = 5
update_target_every_steps = 1
tau = 0.001
```

Performance Evaluation

As shown in Figure 4.5, the algorithm is not as stable and consistent as the other two algorithms (SAC and PPO) for a 3D continuous action state environment. It can be observed that as the randomness of the environment increases, the stability and consistency of the algorithm also increase, and the agent can eventually reach the desired policy after several iterations.

4.3.2 Soft Actor-Critic

Algorithm Integration

This algorithm is known as entropy-maximization algorithm that includes maximizing a mixed objective of the value function and policy entropy which leads to getting the most reward with the most diverse policy. It is an off-policy algorithm in which the agent can reuse the experiences to enhance the policies.

Algorithm 2 Soft Actor-Critic (SAC)

Initialize actor network π, critics Q_1 and Q_2 with random weights θ^{π}, θ^{Q1}, θ^{Q2}
Initialize target critic networks Q_1' and Q_2' with weights $\theta^{Q1'} \leftarrow \theta^{Q1}$, $\theta^{Q2'} \leftarrow \theta^{Q2}$
Initialize temperature α and target entropy $\bar{H}$
Initialize replay buffer R
for episode = 1, M **do**
    Receive initial observation state s_1
    for t = 1, T **do**
        Select action $a_t \sim \pi(s_t|\theta^{\pi})$
        Execute action a_t and observe reward r_t and 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
        Compute target $y_i = r_i + \gamma \min_{j\in\{1,2\}} Q_j'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})) - \alpha \log \pi(a_{i+1}|s_{i+1})$
        Update critics by minimizing the loss:

$$L_{Q1} = \frac{1}{N}\sum_i (y_i - Q_1(s_i, a_i|\theta^{Q1}))^2$$
$$L_{Q2} = \frac{1}{N}\sum_i (y_i - Q_2(s_i, a_i|\theta^{Q2}))^2$$

        Update actor policy by maximizing:

$$J_{\pi} = \mathbb{E}_{s\sim D, a\sim\pi}[Q_1(s,a) - \alpha \log \pi(a|s)]$$

        Adjust temperature α to maintain entropy near target:

$$\alpha \leftarrow \alpha - \mathrm{lr}_{\alpha}(\bar{H} - \mathbb{E}_{a\sim\pi}[-\log \pi(a|s)])$$

        Update the target critics:

$$\theta^{Q1'} \leftarrow \tau\theta^{Q1} + (1-\tau)\theta^{Q1'}$$
$$\theta^{Q2'} \leftarrow \tau\theta^{Q2} + (1-\tau)\theta^{Q2'}$$

    end for
end for

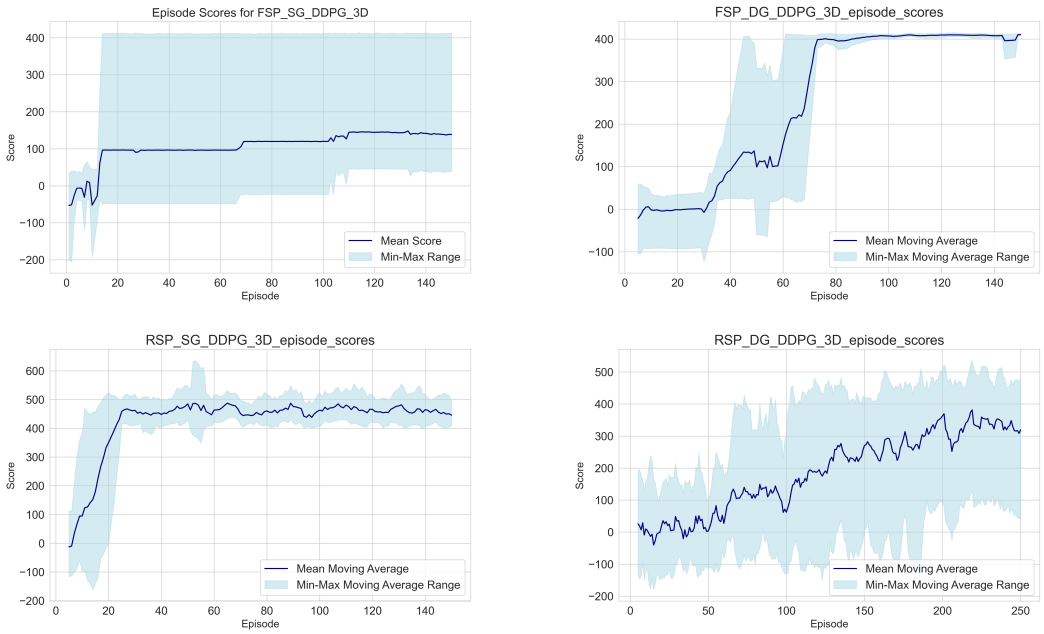

Figure 4.5: Episode Scores and Trajectories of PER Algorithm with Fixed Starting Points, and Fixed Vs. Dynamic Gates

Training and Hyperparameters

In SAC algorithm, the *FCGP* is used as a Fully Connected Gaussian Policy network. Because of the nature of the algorithm, the hidden layers in this network are connected to two separate networks to which represent the mean of actions and logarithmic standard deviation. And also consider in the Optimization function, two separate networks are used to get the minimum Q-Value between these models.

Performance Evaluation

As shown in Figure 4.7, the algorithm outperforms DDPG. It is more stable and consistent compared to the DDPG algorithm. However, in cases where random starting points are employed, the shaded area is wider than that of the PPO algorithm, which will be discussed in the next section. This indicates that, on average, the model performs well, but in some seeds, the agent deviates significantly from the optimal average policy.

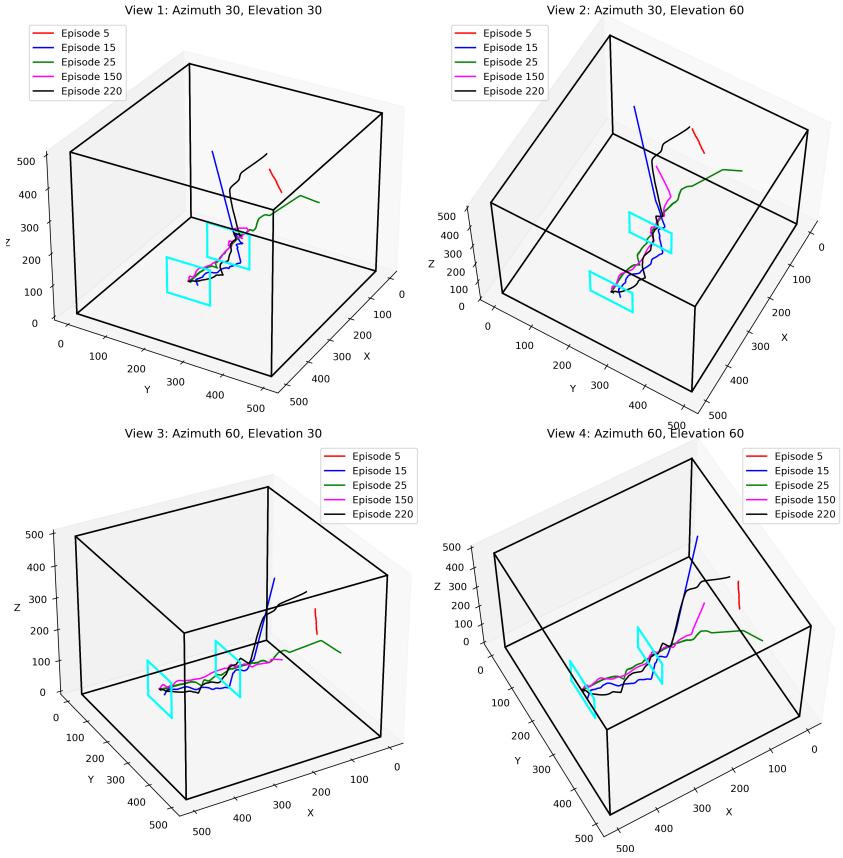

Figure 4.6: Trajectory obtained with the algorithm DDPG in a dynamic environment for increasing episodes

4.3.3 Proximal Policy Optimization

Algorithm Integration

This is an on-policy algorithm despite the previous algorithms which were off-policy algorithms. Also, this algorithm is more conservative than the previous ones because of clipped objectives strategy it has in optimization process.

Training and Hyperparameters

In the optimization of PPO algorithm, the loss is calculated as the negative of the minimum of the objectives.

Algorithm 3 Proximal Policy Optimization

Initialize policy parameters θ, value parameters ϕ, and empty buffer $\mathcal{B}$.
while training not over **do**
    for $i = 1, \ldots,$ max_buffer_episodes **do**
        Generate episode i using policy π_θ and add to buffer $\mathcal{B}$
    end for
    for $k = 1, \ldots,$ policy_optimization_epochs **do**
        Sample minibatch from $\mathcal{B}$
        Calculate surrogate objective $L_t^{CLIP+\mathrm{VF}}(\theta)$ with respect to old policy $\pi_{\theta_{\mathrm{old}}}$
        Update θ by maximizing $L_t^{CLIP+\mathrm{VF}}(\theta)$
        Calculate policy divergence $KL[\pi_{\theta_{\mathrm{old}}}|\pi_\theta]$
        if $KL >$ policy_stopping_kl **then**
            Break
        end if
    end for
    for $k = 1, \ldots,$ value_optimization_epochs **do**
        Sample minibatch from $\mathcal{B}$
        Calculate value loss $L_t^{VF}(\phi)$
        Update ϕ by minimizing $L_t^{VF}(\phi)$
        Calculate mean squared error (MSE) between value estimates
        if MSE $>$ value_stopping_mse **then**
            Break
        end if
    end for
    Empty buffer $\mathcal{B}$
end while

Listing 4.6: Parameter Settings for SAC Algorithm in 3D Environment

```
gamma: 0.99
policy_model_fn = lambda nS, bounds: FCGP(nS, bounds, hidden_dims
    =(128, 128))
policy_max_grad_norm = float('inf')
policy_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
     lr=lr)
policy_optimizer_lr = 0.0003
value_model_fn = lambda nS, nA: FCQSA(nS, nA, hidden_dims=(128,
    128))
value_max_grad_norm = float('inf')
value_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
    lr=lr)
value_optimizer_lr = 0.0005
replay_buffer_fn = lambda: ReplayBuffer(max_size=1000000,
    batch_size=256)
n_warmup_batches = 5
update_target_every_steps = 1
tau = 0.001
```

Figure 4.7: Episode Scores and Trajectories of SAC Algorithm with Fixed Starting Points, and Fixed Vs. Dynamic Gates

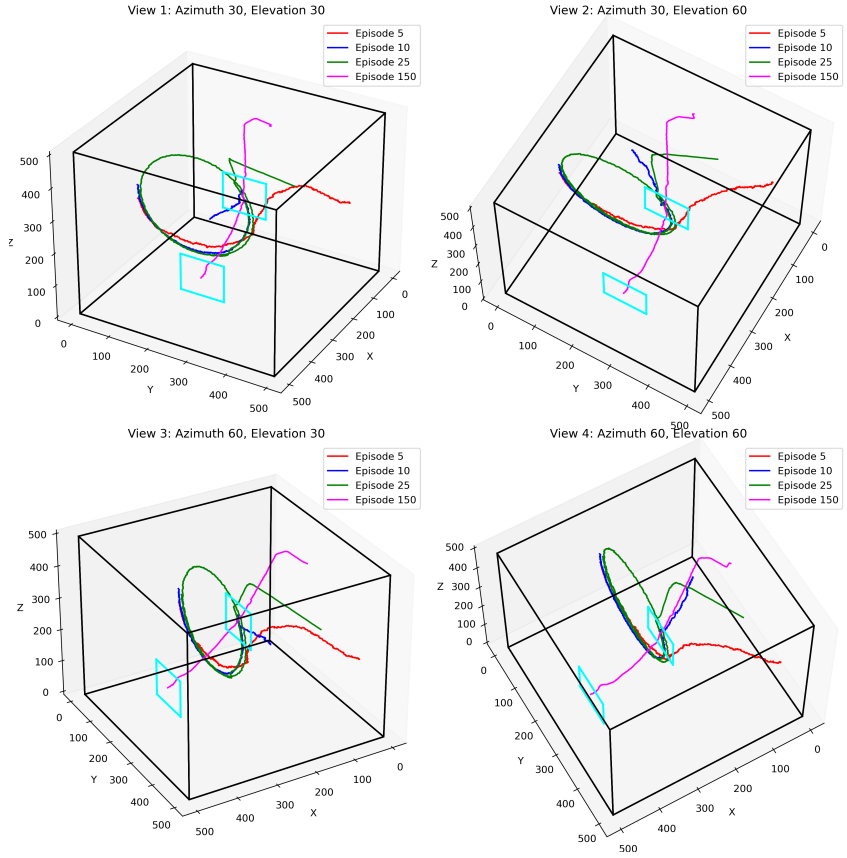

Figure 4.8: Trajectory obtained with the algorithm SAC in a dynamic environment for increasing episodes

Performance Evaluation

As shown in Figure 4.9, the algorithm performs significantly better than the two previous ones. It is more stable and consistent when fixed starting points are studied. Additionally, in cases with random starting points, the agent eventually reaches the desired policy, and the shaded area in the static gates is narrower than that of the SAC algorithm. Moreover, in the case of random starting points and dynamic gates, in the final episodes, the shaded area is narrower than the SAC algorithm's, indicating that in the final episodes, the agent performs close to its optimal average policy.

Listing 4.7: Parameter Settings for PPO Algorithm in 3D Environment

```
gamma: 0.99
policy_model_fn = lambda nS, nA: FCCA(nS, nA, hidden_dims
    =(128,128))
policy_model_max_grad_norm = 0.5
policy_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
     lr=lr)
policy_optimizer_lr = 0.0003
policy_optimization_epochs = 10
policy_sample_ratio = 0.9
policy_clip_range = 0.1
policy_stopping_kl = 0.02
value_model_fn = lambda nS: FCV(nS, hidden_dims=(128,128))
value_model_max_grad_norm = 0.5
value_optimizer_fn = lambda net, lr: optim.Adam(net.parameters(),
    lr=lr)
value_optimizer_lr = 0.0005
value_optimization_epochs = 10
value_sample_ratio = 0.9
value_clip_range = 0.1
value_stopping_mse = 25
episode_buffer_fn = lambda sd, g, t, me, mes, seed: EpisodeBuffer(
    sd, g, t, me, mes, seed)
max_buffer_episodes = 6
max_buffer_episode_steps = 1000
entropy_loss_weight = 0.015
tau = 0.97
policy_lr_scheduler_step_size= 30
policy_lr_scheduler_gamma= 0.96
value_lr_scheduler_step_size= 30
value_lr_scheduler_gamma= 0.96
```

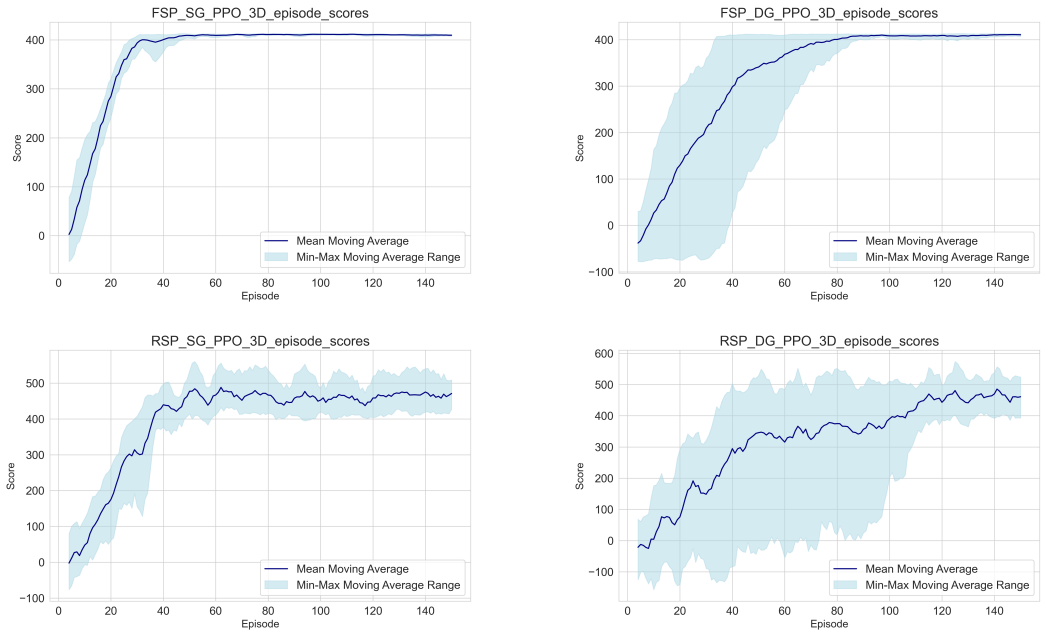

Figure 4.9: Episode Scores and Trajectories of PPO Algorithm with Fixed Starting Points, and Fixed Vs. Dynamic Gates

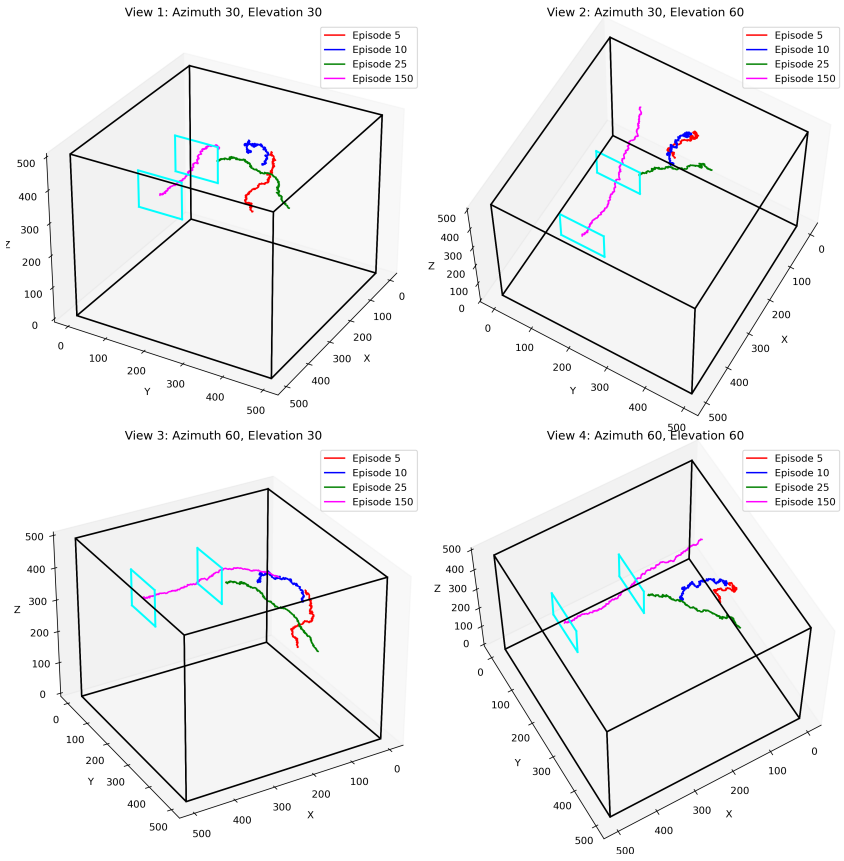

Figure 4.10: Trajectory obtained with the algorithm PPO in a dynamic environment for increasing episodes

Chapter 5

Results

5.1 Experiment Results and Analysis

5.1.1 Performance Metrics

In our research, we employ three main criteria to evaluate the performance of the reinforcement learning algorithms: cumulative score per episode, the count of episodes in which the agent accomplished the objective, and the duration per episode (episode elapsed). These criteria can be further divided into five metrics defined as follows:

- Cumulative Score Per Episode: The cumulative score per episode directly measures the agent's performance in its assigned task. Superior scores suggest that the agent optimizes its actions to fulfill its objective.

- Count of Episodes to Accomplish the Objective: This criterion assesses the agent's learning proficiency. An agent that reliably accomplishes the objective in fewer episodes proves to be quicker in resolving the task and adapting to the environment.

- Duration per Episode (episode elapsed): By evaluating the duration spent on each episode, we can gauge the agent's computational efficiency. This becomes crucial when comparing algorithms that garner similar scores, as computational efficiency could be the deciding factor in algorithm choice.

- Average Reward per Action: The average reward per action offers a measure of the quality of the decisions made by the agent. An increased average reward per action indicates that the agent is making superior decisions, leading to higher rewards for each action. This becomes particularly noteworthy in scenarios where the agent has a limit on the number of actions it can execute

in an episode or when the objective is to garner the highest reward in the least possible time or number of steps. This metric balances the speed (number of steps) and the quality (reward) of the actions, preventing strategies that excessively prioritize one aspect at the detriment of the other.

- Success Rate: This represents the ratio of episodes where the agent accomplishes its objective. A higher success rate indicates that the agent can more reliably complete its task.

Finally, by multiplying Success Rate and Average Rewards together, you get a new metric that can be interpreted as follows:

- High Average Reward per Action, High Success Rate: If the product is high, it suggests that the algorithm is both proficient (it receives a high reward per action) and successful (it frequently accomplishes its goal). This is the optimal situation.
- Low Average Reward per Action, High Success Rate: It might imply that the algorithm frequently accomplishes its goal but could be more proficient (it receives a low reward per action).
- High Average Reward per Action, Low Success Rate: Conversely, a moderate product might suggest that the algorithm is proficient (it receives a high reward per action) but does not frequently accomplish its goal.
- Low Average Reward per Action, Low Success Rate: If the product is low, it suggests that the algorithm needs to be proficient and successful. This would imply that the algorithm's performance is subpar.

These metrics offer a comprehensive perspective of the effectiveness and efficiency of the reinforcement learning algorithms under study.

5.1.2 Results and Discussion

In reviewing the outcomes of our experiments, several key findings were discernable. The performance of the reinforcement learning algorithms was primarily evaluated through metrics: cumulative score per episode, the number of successful episodes, and the episode duration. The cumulative score per episode served as a crucial indicator of the overall efficiency of the algorithms, offering insights into the ability of the agent to accumulate rewards over the course of an episode. The count of successful episodes measured the agent's reliability in achieving the objectives. Lastly, the duration per episode or episode elapsed, allowed us to gauge the speed of the agent's learning process, with shorter durations reflecting faster convergence to optimal behaviors. These criteria collectively provided a

comprehensive understanding of the algorithms' capabilities in various aspects of reinforcement learning.

Table 5.1: Summary of the results obtained with the fixed starting point (FSP) environment

Algorithm	Environment	Total Episodes	Episodes Goal Reached	Average Elapsed Time	Average Reward per Step	Success Rate (%)
FSP_SG_DDPG	Static	1000	**432**	6,768	**0,778**	0,432
FSP_SG_SAC		1000	990	**9,223**	0,709	**0,990**
FSP_SG_PPO		984	**886**	**0,640**	**0,795**	0,900
FSP_DG_DDPG	Dynamic	1000	**770**	10,801	**0,997**	0,770
FSP_DG_SAC		1000	944	**20,647**	0,694	**0,944**
FSP_DG_PPO		984	802	**1,986**	**0,743**	0,815

In 5.1 a comprehensive summary of the results attained using the three selected algorithms is offered, all originating from a consistent starting point across all episodes. The Soft Actor-Critic (SAC) algorithm stands out for its rapid convergence to high rewards with a minimal episode count in both static (SG) and dynamic (DG) environments. However, despite this swift convergence, the SAC manifests some instability, even at high episode counts. On the other hand, the Proximal Policy Optimization (PPO) algorithm demonstrates a slower convergence rate but eventually reaches similar end-of-training rewards as SAC and exhibits less uncertainty in resolving the environment. The slower convergence of PPO is attributed to its inherent design that encourages more conservative policy updates. This approach, while leading to a slower learning rate, also contributes to the algorithm's superior stability, as it avoids drastic policy changes that could potentially disrupt the learning process and result in increased uncertainty.

Table 5.2: Summary of the results obtained with the random starting point (RSP) environment

Algorithm	Environment	Total Episodes	Episodes Goal Reached	Average Elapsed Time	Average Reward per Step	Success Rate (%)
RSP_SG_DDPG	Static	1000	941	5,475	**1,038**	0,941
RSP_SG_SAC		1000	994	9,688	0,752	**0,994**
RSP_SG_PPO		984	**822**	**0,721**	0,805	0,835
RSP_DG_DDPG	Dynamic	1000	**414**	14,652	0,743	0,414
RSP_DG_SAC		1000	759	34,967	0,542	**0,759**
RSP_DG_PPO		984	726	**2,327**	**0,746**	0,738

The results of 5.2 provide a detailed overview of the outcomes achieved using the three selected algorithms, each initiated from a random starting point defined at the beginning of each episode. The Soft Actor-Critic (SAC) algorithm is noteworthy due to its speedy convergence toward high rewards, requiring only a low number of episodes in both static (SG) and dynamic (DG) settings. Conversely, the Proximal Policy Optimization (PPO) algorithm displays a slower convergence rate but eventually attains comparable end-of-training rewards to SAC and shows less volatility when navigating the environment.

5.1.3 Effects of Gate Movement Speed

In summarizing our research efforts, we scrutinized the performance of various reinforcement learning algorithms under the diverse dynamism of our environment. Specifically, we focused on the gate speeds' variability effect on the reward function. For the sake of this experiment, we parametrized the velocity, setting it to a low limit of $0.45 rad/s$, which denoted the starting point, scaling up to a high end of $0.90 rad/s$. Our observations exposed those algorithms, such as DDPG (Deep Deterministic Policy Gradient) and SAC (Soft Actor-Critic), which appeared to be considerably swayed by the varying speeds of the gates. Consequently, a broad spectrum was evident in the range of rewards during the different trials, indicating high variability and inconsistency in the outcomes.

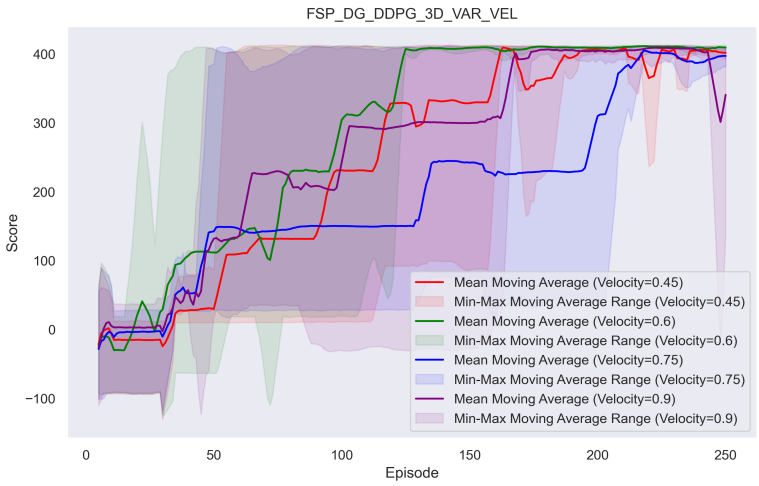

Figure 5.1: Trajectory Planning of DDPG Algorithm with Random Starting Point

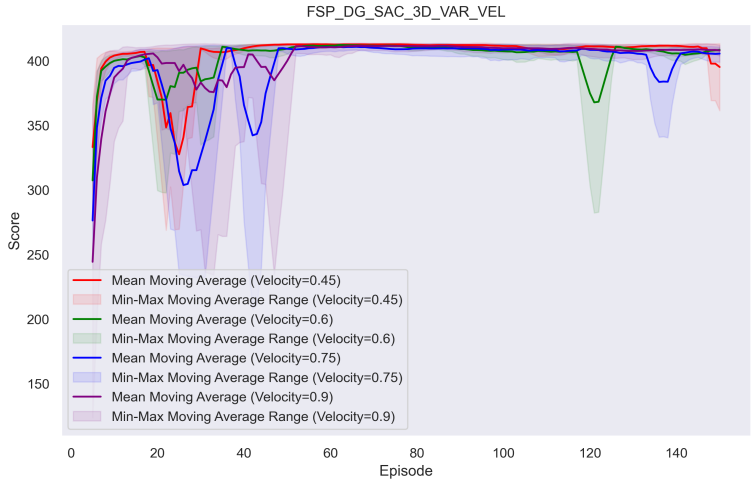

Figure 5.2: Trajectory Planning of DDPG Algorithm with Random Starting Point

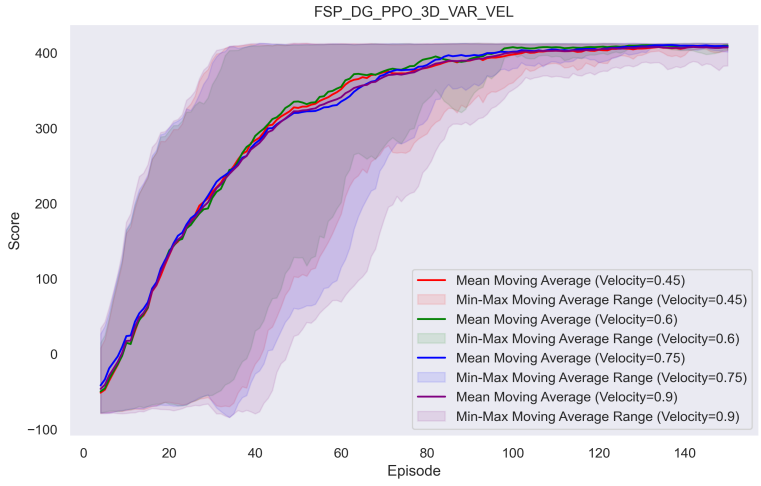

Figure 5.3: Trajectory Planning of DDPG Algorithm with Random Starting Point

Chapter 6

Conclusions and Future Work

Our research findings offer valuable insights into the performance of Deep Deterministic Policy Gradient (DDPG), Soft Actor-Critic (SAC), and Proximal Policy Optimization (PPO) algorithms in intricate environments. Dynamic environments pose significant challenges for robotics systems, primarily due to their unpredictability and high adaptability required for successful operation. These challenges stem from such environments constantly changing, with the robot needing to react and adapt in real-time to achieve its objective. In particular, DDPG's deterministic policy approach showed limitations in these settings. While beneficial in directly approximating the optimal action and possibly hastening convergence, DDPG's exploration strategy could benefit from further refinement to enhance performance in more complex, evolving scenarios.

On the other hand, SAC displayed rapid learning, attributed to its unique integration of a stochastic policy and an entropy component within its objective function. By continually seeking to balance reward maximization and action diversification, SAC skillfully negotiates the exploration-exploitation trade-off, fostering speedy learning in complex environments. While not the fastest to learn, PPO demonstrated robust and steady performance after adequately approximating the optimal policy and value functions. PPO's stability is derived from its clipped objective function, which regulates policy updates to avoid drastic changes that could lead to performance volatility. We also leveraged the Kalman filter to accurately predict the gate's position in space, effectively managing the dynamic nature of the moving gate. The filter uses an iterative process of prediction and correction based on previous states and current measurements, demonstrating its prowess in handling dynamic motion tracking. However, these findings are based on simulations, and as we move forward, we need to consider the next

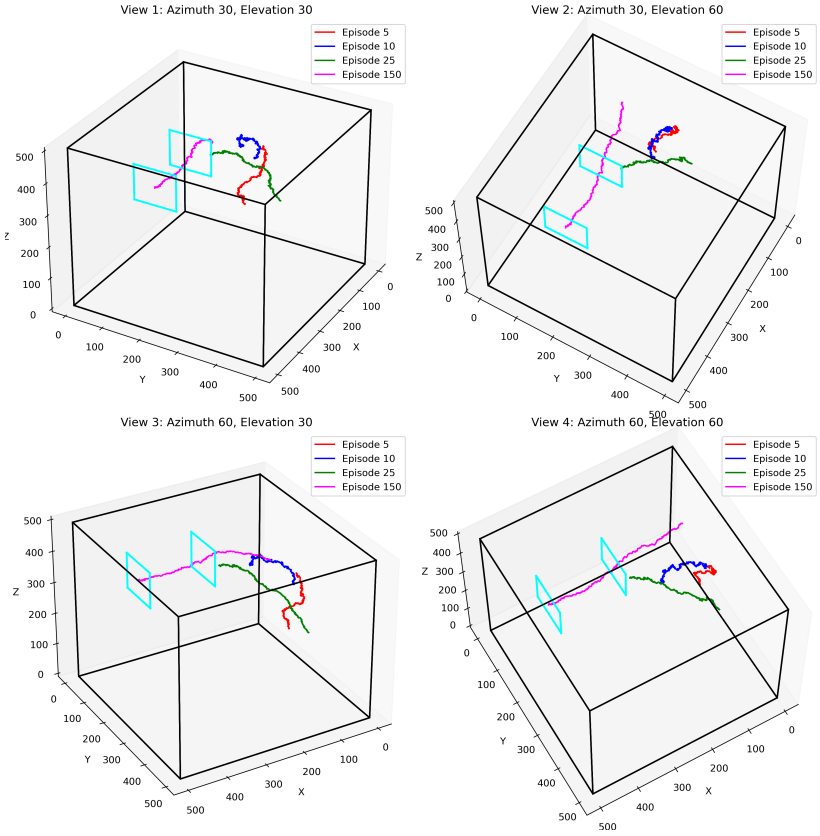

Figure 6.1: Trajectory obtained with the algorithm PPO in a dynamic environment for increasing episodes

steps: transitioning from simulation to hardware testing. Despite the fidelity of our simulations, real-world scenarios can introduce unforeseen variables and challenges that need to be accounted for in the simulation environment. Issues such as sensor noise, mechanical failures, and real-world physics discrepancies may arise and can significantly impact the performance of the robotics systems and the algorithms controlling them. To bridge this 'reality gap', future research will focus on hardware-in-the-loop testing, where the algorithms will be integrated with physical systems in controlled environments. These tests will provide valuable data on how these algorithms perform under real-world conditions and how they can be further optimized. Additionally, we anticipate refining our models to handle environmental changes better, incorporating more advanced perception capabilities, and increasing the robustness of the systems. We aim to enhance our robotics systems' practical applicability and performance in dynamic environments by carefully iterating our designs and algorithms based on this new data.

As shown in 6.2, the *DDPG* is the most dependent on the environment condition,

while the PPO is the most stable one from the average reward per step point of view. Also, the *SAC* and *PPO* have higher success rate than the *DDPG*.

Future work will involve improving the corner detection algorithms for the gates in a simulation environment, identifying the center of the gates corresponding to those corners, and integrating the proposed algorithm into this project, including the gate's center detection. As a continuation of the simulation environment, all the algorithms will be implemented in a real environment using a drone.

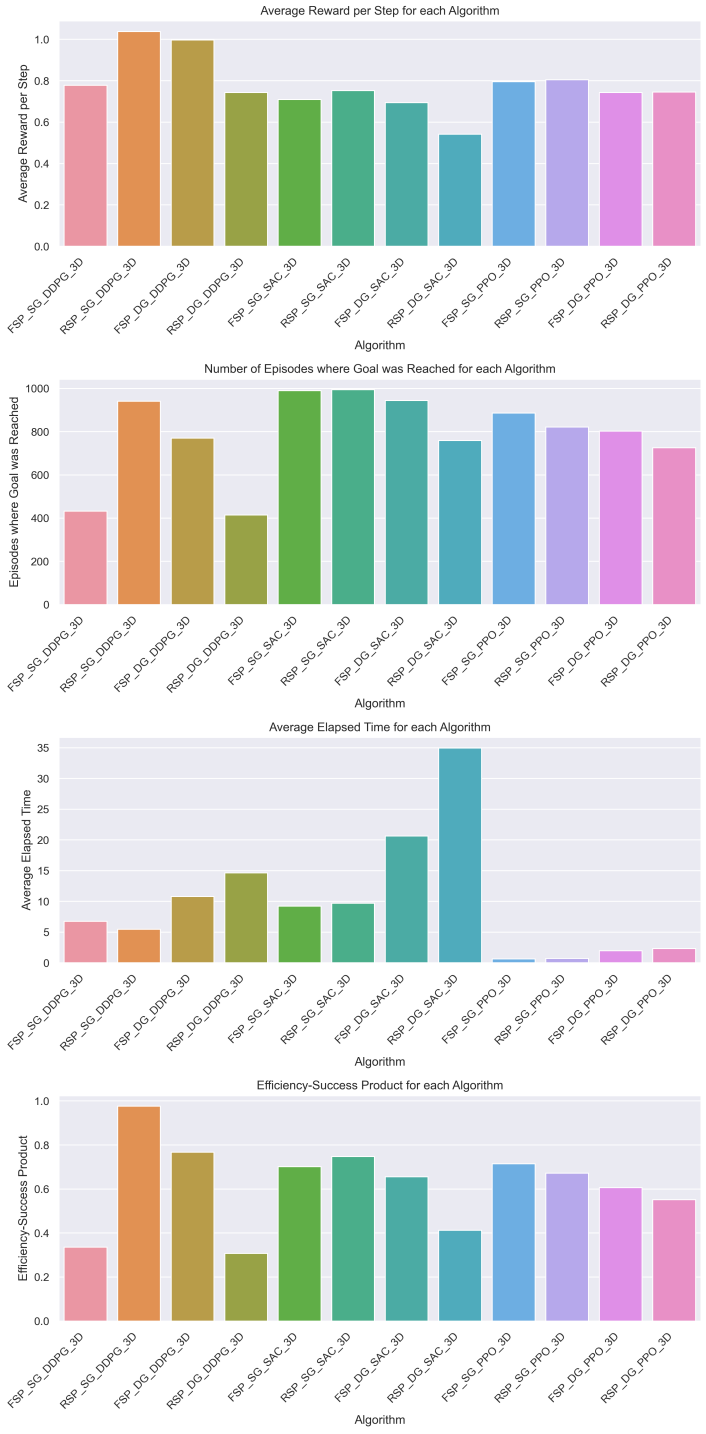

Figure 6.2: Summary of the Performances for different algorithms metrics

Bibliography

[1] Daniel Mellinger and Vijay R. Kumar. «Minimum snap trajectory generation and control for quadrotors». In: *2011 IEEE International Conference on Robotics and Automation* (2011), pp. 2520–2525. URL: https://api.semanticscholar.org/CorpusID:18169351 (cit. on p. 1).

[2] Muhammad Awais Arshad, Jamal Ahmed, and Hyochoong Bang. «Quadrotor Path Planning and Polynomial Trajectory Generation Using Quadratic Programming for Indoor Environments». In: *Drones* 7.2 (2023). ISSN: 2504-446X. DOI: 10.3390/drones7020122. URL: https://www.mdpi.com/2504-446X/7/2/122 (cit. on p. 1).

[3] Robert Penicka and Davide Scaramuzza. «Minimum-Time Quadrotor Waypoint Flight in Cluttered Environments». In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 5719–5726. DOI: 10.1109/LRA.2022.3154013 (cit. on p. 1).

[4] Robert Penicka, Yunlong Song, Elia Kaufmann, and Davide Scaramuzza. «Learning Minimum-Time Flight in Cluttered Environments». In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 7209–7216. DOI: 10.1109/LRA.2022.3181755 (cit. on p. 1).

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. «OpenAI Gym». In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540 (cit. on p. 2).

[6] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. «AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles». In: *CoRR* abs/1705.05065 (2017). arXiv: 1705.05065. URL: http://arxiv.org/abs/1705.05065 (cit. on p. 2).

[7] Yoram Koren and Johann Borenstein. «Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation». In: vol. 2. May 1991, 1398–1404 vol.2. DOI: 10.1109/ROBOT.1991.131810 (cit. on p. 3).

[8] Tharindu Weerakoon, Kazuo Ishii, and Amir Ali Forough Nassiraei. «An Artificial Potential Field Based Mobile Robot Navigation Method To Prevent From Deadlock». In: *Journal of Artificial Intelligence and Soft Computing Research* 5.3 (2015), pp. 189–203. DOI: doi:10.1515/jaiscr-2015-0028. URL: https://doi.org/10.1515/jaiscr-2015-0028 (cit. on p. 3).

[9] Steven M. LaValle. «Rapidly-exploring random trees : a new tool for path planning». In: *The annual research report* (1998). URL: https://api.semanticscholar.org/CorpusID:14744621 (cit. on p. 3).

[10] Iram Noreen, Amna Khan, and Zulfiqar Habib. «Optimal Path Planning using RRT* based Approaches: A Survey and Future Directions». In: *International Journal of Advanced Computer Science and Applications* 7.11 (2016). DOI: 10.14569/IJACSA.2016.071114. URL: http://dx.doi.org/10.14569/IJACSA.2016.071114 (cit. on p. 3).

[11] Priyadarshi Bhattacharya and Marina L. Gavrilova. «Roadmap-Based Path Planning - Using the Voronoi Diagram for a Clearance-Based Shortest Path». In: *IEEE Robotics & Automation Magazine* 15 (2008). URL: https://api.semanticscholar.org/CorpusID:1684741 (cit. on p. 3).

[12] Mitsuhiro Ozaki, Jagannath Aryal, and Paul Fox-Hughes. «Dynamic Wildfire Navigation System». In: *ISPRS International Journal of Geo-Information* 8.4 (2019). ISSN: 2220-9964. DOI: 10.3390/ijgi8040194. URL: https://www.mdpi.com/2220-9964/8/4/194 (cit. on p. 3).

[13] Myoung Hoon Lee and Jun Moon. *Deep Reinforcement Learning-based UAV Navigation and Control: A Soft Actor-Critic with Hindsight Experience Replay Approach*. 2021. arXiv: 2106.01016 [eess.SY] (cit. on p. 3).

[14] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. «Planning and Decision-Making for Autonomous Vehicles». In: *Annual Review of Control, Robotics, and Autonomous Systems* 1.1 (2018), pp. 187–210. DOI: 10.1146/annurev-control-060117-105157. eprint: https://doi.org/10.1146/annurev-control-060117-105157. URL: https://doi.org/10.1146/annurev-control-060117-105157 (cit. on p. 3).

[15] Yunlong Song, Mats Steinweg, Elia Kaufmann, and Davide Scaramuzza. «Autonomous Drone Racing with Deep Reinforcement Learning». In: *CoRR* abs/2103.08624 (2021). arXiv: 2103.08624. URL: https://arxiv.org/abs/2103.08624 (cit. on p. 3).

[16] Philipp Foehn, Angel Romero, and Davide Scaramuzza. «Time-Optimal Planning for Quadrotor Waypoint Flight». In: *CoRR* abs/2108.04537 (2021). arXiv: 2108.04537. URL: https://arxiv.org/abs/2108.04537 (cit. on p. 3).

[17] Antonio Loquercio, Elia Kaufmann, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. «Deep Drone Racing: From Simulation to Reality with Domain Randomization». In: *CoRR* abs/1905.09727 (2019). arXiv: 1905.09727. URL: http://arxiv.org/abs/1905.09727 (cit. on p. 3).

[18] Angel Romero, Sihao Sun, Philipp Foehn, and Davide Scaramuzza. «Model Predictive Contouring Control for Near-Time-Optimal Quadrotor Flight». In: *CoRR* abs/2108.13205 (2021). arXiv: 2108.13205. URL: https://arxiv.org/abs/2108.13205 (cit. on p. 3).

[19] Tim Salzmann, Elia Kaufmann, Jon Arrizabalaga, Marco Pavone, Davide Scaramuzza, and Markus Ryll. «Real-Time Neural MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms». In: *IEEE Robotics and Automation Letters* 8.4 (2023), pp. 2397–2404. DOI: 10.1109/LRA.2023.3246839 (cit. on p. 3).

[20] Sihao Sun, Angel Romero, Philipp Foehn, Elia Kaufmann, and Davide Scaramuzza. «A Comparative Study of Nonlinear MPC and Differential-Flatness-Based Control for Quadrotor Agile Flight». In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3357–3373. DOI: 10.1109/TRO.2022.3177279 (cit. on p. 3).

[21] Yunlong Song and Davide Scaramuzza. «Policy Search for Model Predictive Control with Application to Agile Drone Flight». In: *CoRR* abs/2112.03850 (2021). arXiv: 2112.03850. URL: https://arxiv.org/abs/2112.03850 (cit. on p. 3).

[22] R. E. Kalman. «A New Approach to Linear Filtering and Prediction Problems». In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552. eprint: https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/82/1/35/5518977/35_1.pdf. URL: https://doi.org/10.1115/1.3662552 (cit. on p. 3).

[23] Léon Bottou. «Stochastic Gradient Learning in Neural Networks». In: 1991. URL: https://api.semanticscholar.org/CorpusID:12410481 (cit. on p. 9).

[24] L Eon Bottou. «Online Learning and Stochastic Approximations». In: 1998. URL: https://api.semanticscholar.org/CorpusID:2101184 (cit. on p. 9).

[25] Ning Qian. «On the momentum term in gradient descent learning algorithms». In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166 (cit. on p. 10).

[26] Diederik P. Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980 (cit. on p. 11).

[27] *Playing Atari with Deep Reinforcement Learning*. 2013 (cit. on p. 11).

[28] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *Nature* 518 (2015), pp. 529–533. URL: https://api.semanticscholar.org/CorpusID:205242740 (cit. on p. 11).

[29] Long-Ji Lin. «Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching». In: *Mach. Learn.* 8.3–4 (May 1992), pp. 293–321. ISSN: 0885-6125. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699 (cit. on p. 11).

[30] Hado van Hasselt, Arthur Guez, and David Silver. «Deep Reinforcement Learning with Double Q-learning». In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461 (cit. on p. 13).

[31] Ziyu Wang, Nando de Freitas, and Marc Lanctot. «Dueling Network Architectures for Deep Reinforcement Learning». In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581 (cit. on p. 13).

[32] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. «Prioritized Experience Replay». In: (Nov. 2015) (cit. on p. 14).

[33] Ronald J. Williams. «Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning». In: *Machine Learning* 8 (2004), pp. 229–256. URL: https://api.semanticscholar.org/CorpusID:19115634 (cit. on p. 15).

[34] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. «Deterministic Policy Gradient Algorithms». In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. URL: https://proceedings.mlr.press/v32/silver14.html (cit. on p. 16).

[35] 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». In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1509.02971 (cit. on p. 16).

[36] Scott Fujimoto, Herke van Hoof, and David Meger. «Addressing Function Approximation Error in Actor-Critic Methods». In: *CoRR* abs/1802.09477 (2018). arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477 (cit. on p. 18).

[37] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. «Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor». In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290 (cit. on p. 20).

[38] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. «Proximal Policy Optimization Algorithms». In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347 (cit. on p. 22).

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. «Proximal Policy Optimization Algorithms». In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347 (cit. on p. 22).

[40] Shengyong Chen. «Kalman Filter for Robot Vision: A Survey». In: *IEEE Transactions on Industrial Electronics* 59 (2012), pp. 4409–4420 (cit. on p. 30).

[41] Sergey Gorbunov, Udo Kebschull, Ivan Kisel, Volker Lindenstruth, and Walter F. J. Müller. «Fast SIMDized Kalman filter based track fit». In: *Comput. Phys. Commun.* 178.5 (2008), pp. 374–383. DOI: 10.1016/j.cpc.2007.10.001. URL: https://doi.org/10.1016/j.cpc.2007.10.001 (cit. on p. 31).