République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

المدرسة الوطنية المتعددة التقنيات
قسم الآلية
École Nationale Polytechnique
Département d'Automatique

End of studies project thesis

Submitted in partial fulfillment of the requirements for the State Engineer Degree in Automation Engineering

Reinforcement and Deep Learning-based optimization of pose estimation techniques in single and multi-agent systems: application to aerial and mobile robots

Realized by :
Mr. Kobbi Islem
Mr. Benamirouche Abdelhak

Supervised by :
Pr. Tadjine Mohamed

Publicly presented and defended on the 1st of July, 2023, in front of the jury composed of :

President	Pr. Mohamed Seghir BOUCHERIT	ENP
Promoter	Pr. Mohamed TADJINE	ENP
Examiner	Dr. Messaoud CHAKIR	ENP

ENP 2023

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

المدرسة الوطنية المتعددة التقنيات
قسم الآلية
École Nationale Polytechnique
Département d'Automatique

End of studies project thesis

Submitted in partial fulfillment of the requirements for the State Engineer Degree in Automation Engineering

---

Reinforcement and Deep Learning-based optimization of pose estimation techniques in single and multi-agent systems: application to aerial and mobile robots

---

Realized by :
Mr. Kobbi Islem
Mr. Benamirouche Abdelhak

Supervised by :
Pr. Tadjine Mohamed

Publicly presented and defended on the 1st of July, 2023, in front of the jury composed of :

President	Pr. Mohamed Seghir BOUCHERIT	ENP
Promoter	Pr. Mohamed TADJINE	ENP
Examiner	Dr. Messaoud CHAKIR	ENP

ENP 2023

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

المدرسة الوطنية المتعددة التقنيات
قسم الآلية
École Nationale Polytechnique
Département d'Automatique

Mémoire de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'État en Automatique

Optimisation par Reinforcement et Deep Learning des techniques d'estimation de la pose dans les systèmes mono et multi-agents: application aux robots aériens et mobiles

Réalisé par :
Mr. Kobbi Islem
Mr. Benamirouche Abdelhak

Encadré par :
Pr. Tadjine Mohamed

Soutenu le 1er Juillet 2023, Devant le jury composé de :

Président	Pr. Mohamed Seghir BOUCHERIT	ENP
Promoteur	Pr. Mohamed TADJINE	ENP
Examinateur	Dr. Messaoud CHAKIR	ENP

ENP 2023

ملخص

في هذا العمل ، سنركز على تحسين تقنيات تقدير الوضع للروبوتات الجوية والمتحركة في كل من الأنظمة أحادية الوكيل ومتعددة الوكلاء. سيتم اقتراح مناهج جديدة تعتمد على التعلم العميق والتعلم المعزز لتعزيز الدقة والمتانة. تتضمن الأطروحة مراجعة شاملة للأدبيات ، وإدخال الأدوات البرمجية المستخدمة في البحث. سيتم تقديم طريقتين لتقدير وضعية الوكيل الفردي: مقدر QR لنسخة تكيفية من Extended Kalman Filter ونهج KalmanNet لتقدير مباشر لكسب المرشح. سيتم إثبات فعالية هذه الأساليب من خلال المحاكاة. سيتم بعد ذلك توسيع التحقيق ليشمل تقدير الوضع التعاوني في الأنظمة متعددة العوامل. كما سيتم اقتراح رواية تهدف إلى تحسين الدقة والمتانة من خلال الاستفادة من المعلومات من الوكلاء المجاورين. سيتم التحقق من صحة النتائج في بيئة محاكاة في العالم الحقيقي باستخدام ROS و Gazebo.

كلمات مفتاحية : التعليم المعزز ، التعلم العميق ، تقدير الوضع ، النظام متعدد الوكلاء.

Résumé

Dans ce travail, nous nous concentrerons sur l'optimisation des techniques d'estimation de la pose pour les robots aériens et mobiles dans les systèmes mono-agents et multi-agents. De nouvelles approches basées sur l'apprentissage profond et l'apprentissage par renforcement seront proposées pour améliorer la précision et la robustesse. La thèse comprend une revue complète de la littérature et présente les outils logiciels utilisés dans la recherche. Deux approches pour l'estimation de la pose d'un seul agent seront présentées : l'estimateur QR pour une version adaptative du filtre de Kalman étendu et l'approche KalmanNet pour une estimation directe du gain du filtre. L'efficacité de ces approches sera démontrée par des simulations. L'étude sera ensuite étendue à l'estimation collaborative de la pose dans les systèmes multi-agents. Une nouvelle approche sera également proposée afin d'améliorer la précision et la robustesse en exploitant les informations fournies par les agents voisins. Tous les résultats seront validés dans un environnement de simulation réel en utilisant ROS et Gazebo.

Mots clés : Apprentissage par Renforcement, Apprentissage Profond, Estimation de la pose, Système multi-agent.

Abstract

In this work, we will focus on optimizing pose estimation techniques for aerial and mobile robots in both single-agent and multi-agent systems. Novel approaches based on Deep Learning and Reinforcement Learning will be proposed to enhance accuracy and robustness. The thesis includes a comprehensive literature review, introducing software tools used in the research. Two approaches for single agent pose estimation, will be presented : the QR estimator for an adaptive version of the Extended Kalman Filter and the KalmanNet approach for a direct estimation of the Filter gain. The effectiveness of these approaches will be demonstrated through simulations. The investigation will then be extended to collaborative pose estimation in multi-agent systems. A novel approach will be also proposed which aims to improve accuracy and robustness by leveraging information from neighboring agents. The findings will be validated in a real-world simulation environment using ROS and Gazebo.

Keywords : Reinforcement Learning, Deep Learning, Pose estimation, Multi-agent system.

Dedication

“ To our dear parents, our families, and our friends. ”

- Abdelhak and Islem

Acknowledgments

Above all, we are grateful to Allah, the Almighty, for having given us this opportunity and granted us the ability and patience to carry out this work.

We would like to thank all those who contributed in any way to the realization of this graduation project, because on our own, we probably wouldn't have been able to do much.

We would like to thank our supervisor, Pr. TADJINE Mohamed, first of all for his continuous support throughout our final year project, for his wise guidance and advice, as well as his generous encouragement. Our gratitude extends to all the teachers in the Automation Department and at the Ecole Nationale Polytechnique in general, for all the knowledge they have imparted to us during our five years of study.

Our sincere thanks also to the jury members who honored us by evaluating our work.

Finally, we would like to express our deepest gratitude to our dear parents and respective families, may God bless them, for their continuous support, encouragement and unconditional commitment throughout the realization of this graduation project.

Contents

List of Figures

List of Abbreviations

Adam	*Adaptive Moment Estimation*
ANN	*Artificial Neural Network*
DDPG	*Deep Deterministic Policy Gradient*
DL	*Deep Learning*
DNN	*Deep Neural Network*
DRL	*Deep Reinforcement Learning*
EKF	*Extended Kalman Filter*
LSTM	*Long Short-Term Memory*
NN	*Neural Network*
PPO	*Proximal Policy Optimization*
ReLU	*Rectified Linear Unit*
RL	*Reinforcement Learning*
RMSE	*Root Mean Squared Error*
RNN	*Recurrent Neural Network*
ROS	*Robot Operating System*
SAC	*Soft Actor-Critic*
TD3	*Twin Delayed Deep Deterministic Policy Gradient*
UAV	*Unmanned Aerial Vehicles*

General Introduction

In recent years, the field of robotics has witnessed significant advancements in pose estimation techniques, which play a fundamental role in enabling robots to understand and navigate their environments. Accurate and robust pose estimation is essential for a wide range of robotic applications, including autonomous navigation, mapping and object manipulation. However, pose estimation in complex and dynamic environments remains a challenging problem due to various factors such as sensor noise and uncertainty.

In the realm of control systems and estimation, having an adaptive state estimator is crucial for accurately tracking the state of a system, particularly in dynamic and uncertain environments. The Extended Kalman Filter (EKF) has been widely employed for state estimation in nonlinear systems [7, 32, 1, 5]. However, the EKF has its limitations in terms of adaptability and performance, especially in complex scenarios. Efforts have been made to address these limitations by integrating recurrent neural networks (RNNs) into the EKF framework for state estimation. One notable approach in this regard is KalmanNet [28], which replaces the traditional EKF mechanism with RNNs. This approach proves valuable when the system is challenging to model accurately. However, for systems that are already easy to model, the application of KalmanNet may not offer significant advantages.

One of the primary challenges with the EKF lies in the fixed values for the system dynamics and measurement noise covariance matrices (Q and R). These values are typically predetermined or manually tuned, thus limiting the adaptability of the EKF in the face of changing system dynamics or measurement noise characteristics. Consequently, there is a need for an approach that overcomes these limitations and provides a more adaptable and robust state estimation algorithm.

In light of these challenges, this thesis proposes an innovative approach that incorporates Deep Reinforcement Learning (DRL) techniques into the EKF to get an adaptive state estimation algorithm. By integrating DRL, the state estimator can autonomously learn and update the values of Q and R based on observed data, environmental changes, or system failures. This adaptability enhances the performance of the EKF and enables it to better handle uncertainties and dynamic conditions in pose estimation tasks for mobile and aerial robots.

The integration of DRL techniques with the EKF algorithm holds great promise for advancing the capabilities of pose estimation in robotics. By leveraging the power of deep learning and reinforcement learning, the proposed approach can enhance the adaptability and performance of state estimation systems, enabling robots to operate more effectively and robustly in dynamic and uncertain environments.

Furthermore, this thesis extends the scope of investigation to multi-agent robotic systems, which involve the coordination of multiple robots to estimate their collective poses. this estimation is essential applications such as cooperative localization [16], tracking [17], mapping [10], path planning and collision avoidance [22]. However, the accuracy and precision of pose estimation can be compromised if individual robots fail or experience issues. In such scenarios, the development of robust methods that can adapt and provide reliable pose estimation becomes essential.

Traditional methods for collaborative pose estimation often rely on relative pose estimation where each robot exchange its relative position with its neighbors. Unfortunately, these approaches are vulnerable to errors when sensors fail or provide inaccurate measurements. Moreover, variations in the environment or unexpected robot behaviors can further degrade estimation accuracy. To address these challenges, we propose a novel approach that harnesses the power of DRL as a decision-making mechanism for selecting the most suitable robots to collaborate with, thereby enhancing overall pose estimation accuracy.

Our method introduces the concept of incorporating the distance between robots as a valuable metric to achieve more accurate pose estimation. By considering the distances between robots, we can exploit the geometric relationships within the system to refine the estimated poses. We leverage the capabilities of deep RL agent which serve as intelligent decision-makers, evaluating the precision of measurements and estimated positions of each robot in the swarm to select the most reliable collaborators for pose estimation.

To validate our findings, we implement and test the proposed techniques in a real-world simulation environment using the Robot Operating System (ROS) and Gazebo. By conducting practical experiments, we assess the performance and feasibility of our approaches and identify potential challenges and obstacles that need to be addressed in future work.

Overall, this research aims to contribute to the field of pose estimation in robotics by leveraging the potential of RL and DL techniques. The ultimate goal is to enable mobile and aerial robots to accurately perceive and understand their spatial position and orientation, facilitating a wide range of robotics applications in various domains.

Thesis organization

This thesis consists of five chapters, in addition to the general introduction and conclusion.

Chapter 1 provides a comprehensive literature review on pose estimation in robotics, highlighting the different approaches, techniques, and tools that have been developed. We will start by discussing the different approaches to pose estimation, including their strengths and limitations. Then, we will delve into the different techniques that have been proposed, such as Kalman filters and particle filters. Additionally, we will also explore the emerging area of pose estimation in multi-agent systems, which involves the coordination of multiple robots to estimate their collective poses.

Chapter 2 introduces Deep Learning and Reinforcement Learning as powerful tools to optimize and enhance the accuracy and robustness of pose estimation techniques. We will also present the software tools used in this work, highlighting their role in implementing and evaluating the proposed approaches.

Chapter 3 focuses on pose estimation for a single agent in two different applications: mobile robots and aerial robots. We will present an overview of the different approaches that have been followed including both traditional techniques such as EKF and its optimized versions using deep learning and reinforcement learning. An adaptive version of the Extended Kalman Filter (EKF) algorithm is proposed, using Reinforcement Learning to estimate the Q and R parameters of the EKF, known as QR estimator. Additionally, the KalmanNet approach is introduced which directly estimates the gain K of the EKF estimator. To demonstrate the effectiveness of these techniques, we will discuss various simulations that have been done, highlighting the strengths and limitations of each approach.

In Chapter 4, we will present the proposed approach for collaborative pose estimation in a multi-agent robotic system. The objective is to improve the accuracy and robustness of pose estimation by leveraging the information from neighboring agents. This approach combines the use of distributed EKFs and the exchange of relative pose measurements. Moreover, an RL agent is introduced to select the best measurements for each agent based on signal characteristics, providing a more efficient and accurate pose estimation process.

In Chapter 5, we will demonstrate the practical application of our findings in a real-world scenario using the Robot Operating System (ROS) with its 3D simulation environment, Gazebo. We will test the effectiveness of our solution as a component of an actual system and identify the challenges and obstacles it must overcome.

A general conclusion will summarize the work carried out, the main results obtained and the contributions made to the field of pose estimation in robotics as well as some potential perspectives and future work.

To show the relevance of the work conducted in this thesis, the main contributions are summarized in the following:

- An adaptive version of the EKF estimator based on Deep Reinforcement Learning.
- A novel approach to improve the accuracy and adaptability of collaborative pose estimation in multi-agent robotic systems.
- A comparative study between RL agents : DDPG, TD3, SAC and PPO.

Chapter 1

Pose estimation in Robotics

1.1 Introduction

In this chapter, we will review the literature on pose estimation in robotics, focusing on the different approaches, techniques, and tools that have been developed. We will start by discussing the different approaches to pose estimation, including their strengths and limitations. Then, we will delve into the different techniques that have been proposed, such as Kalman filters and particle filters. Additionally, we will also explore the emerging area of pose estimation in multi-agent systems, which involves the coordination of multiple robots to estimate their collective poses.

1.2 Pose estimation

Pose estimation is a fundamental task in robotics that involves determining the position and orientation of a robot relative to its environment, allowing the robot to move and interact with objects in its surroundings. Accurate pose estimation is essential for many robotics applications, such as navigation, mapping, and manipulation. There are various approaches to pose estimation in robotics, and they typically involve combining information from multiple sensors and sources.

In other terms, pose estimation is about estimating the position and orientation of the body frame b in the navigation frame n where

- **the body frame b**: is the coordinate frame of the moving robot and all the inertial measurements are resolved in this frame.
- **the navigation frame n**: is a local geographic frame in which we want to navigate. For most applications it is defined stationary with respect to the earth.

This problem is illustrated in Figure 1.1, where the position and orientation of the body changes from time t_1 to time t_2. [19]

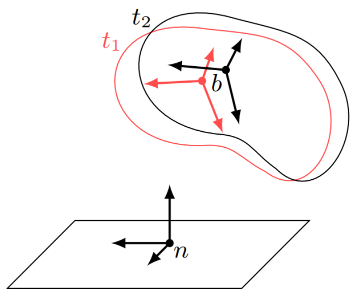

Figure 1.1: An illustration of the pose estimation problem. We want to express the position and orientation of the moving body frame b at times t_1 and t_2 with respect to the navigation frame n.

1.2.1 Vision-based pose estimation

One approach to pose estimation is to use visual sensors, such as cameras, to track the position and orientation of the robot. Visual-based pose estimation methods typically involve detecting and tracking features in the environment and using them to infer the robot's pose. These techniques can provide high accuracy and robustness, but they may be affected by lighting conditions, occlusions and other environmental factors.

1.2.2 Range-based pose estimation

Another approach to pose estimation is to use range sensors, such as lidar or sonar, to measure the distance to objects in the environment. Range-based pose estimation techniques typically involve generating a 3D map of the environment and using it to estimate the robot's pose. These techniques can provide accurate and reliable pose estimates, particularly in structured environments, but they may be limited by the range and resolution of the sensors.

1.2.3 Inertial-based pose estimation

In addition to visual and range sensors, other sources of information can be used for pose estimation in robotics, such as GPS, magnetometers and inertial sensors. These sensors can provide complementary information about the robot's position and orientation, and they can be used to improve the accuracy and robustness of the pose estimation.

1.3 Different techniques of pose estimation

One of the main challenges of pose estimation in robotics is dealing with noise, uncertainty, and other sources of error in the sensor measurements. To address this challenge, various techniques have been proposed, such as Kalman filters and particle filters. These techniques involve fusing the sensor measurements and other sources of information to estimate the robot's pose with high accuracy and robustness.

1.3.1 Kalman Filter

Kalman filter is a recursive algorithm that estimates the state of a dynamic system from a series of noisy measurements. It is widely used in robotics for pose estimation due to its simplicity and effectiveness. The Kalman filter assumes that the system dynamics can be modeled by a linear system, and the measurement noise is Gaussian distributed.

The Kalman filter addresses the general problem of trying to estimate the state x of a discrete-time controlled linear process that is governed by the linear difference equation

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}, \tag{1.1}$$

with a measurement y that is

$$y_k = Hx_k + v_k, \tag{1.2}$$

where The random variables w_k and v_k represent the process and measurement gaussian zero-mean noise, respectively.

The matrix A relates the state at the previous time step $k-1$ to the state at the current step k, in the absence of either a driving function or process noise. The matrix B relates the optional control input u to the state x. The matrix H relates the state to the measurement y_k.[36]

1.3.2 Extended Kalman Filter

The Extended Kalman Filter (EKF) is a variant of the Kalman filter that can handle nonlinear systems by linearizing the system dynamics and measurement functions around the current estimate.

Unlike the regular Kalman Filter, which tries to estimate the state of a discrete-time controlled linear process, EKF addresses the problem of estimating the state of nonlinear processes, where the state transition and/or observation models are nonlinear functions. EKF linearizes these two nonlinear functions about their mean vector and covariance matrix, similar to Taylor series.[24]

Assume the process to be estimated is:

$$x_k = f\left(x_{k-1}, u_k, w_{k-1}\right), \tag{1.3}$$

and the measurement is:

$$y_k = h\left(x_k, v_k\right), \tag{1.4}$$

where w_k and v_k represent the process and measurement noise, x_k and y_k represent the state and observation at time k respectively, f is the process model and h is the measurement model.

EKF algorithm consists of two parts of time update and measurement update. The complete set of equations is given below.[36]

EKF time update (prediction) equations:

1. Project the state ahead:

$$\hat{x}_k^- = f\left(\hat{x}_{k-1}, u_{k-1}, 0\right) \tag{1.5}$$

2. Project the error covariance ahead:

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{1.6}$$

Where A_k and W_k are the process jacobians at step k, and Q_k is the process noise covariance at step k.

EKF measurement update (correction) equations:

1. Compute the Kalman gain:

$$K_k = P_k^- H_k^T \left(H_k P_k^- H_k^T + V_k R_k V_k^T\right)^{-1} \tag{1.7}$$

2. Update estimate with measurement y_k:

$$\hat{x}_k = \hat{x}_k^- + K_k \left(y_k - h(\hat{x}_k^-, 0)\right) \tag{1.8}$$

3. Update the error covariance P_k:

$$P_k = \left(I - K_k H_k\right) P_k^- \tag{1.9}$$

Where H_k and V_k are the measurement Jacobians at step k, and R_k is the measurement noise covariance at step k.

The EKF is widely used in robotics for pose estimation, especially when using sensors such as GPS or IMUs that have nonlinear measurement models.

1.3.3 Particle Filters

Particle filters, also known as Monte Carlo filters, are a family of non-parametric Bayesian filters that can estimate the state of a dynamic system from a series of noisy measurements. Particle filters represent the probability distribution of the state using a set of particles, each of which represents a possible state of the system. The particles are propagated through time using the system dynamics, and their weights are updated based on the likelihood of the measurements.

Particle filters are particularly useful for pose estimation in robotics when the system dynamics and measurement functions are nonlinear or non-Gaussian. However, they can be computationally expensive and require a large number of particles to achieve good accuracy.

1.4 Pose estimation in multi-agent system

In multi-agent systems, where multiple robots or agents collaborate to achieve a common goal, accurate and efficient pose estimation becomes even more important. Collaborative pose estimation involves the exchange of information and coordination between multiple agents to estimate their respective poses accurately.

Pose estimation in multi-agent systems poses unique challenges compared to single-agent scenarios. These challenges include:

Limited Sensing and Communication: Agents often have limited sensing capabilities, such as range sensors or IMUs, leading to incomplete or noisy pose measurements. Moreover, communication constraints, such as bandwidth limitations or network delays can impact the exchange of pose information between agents.

Data Association and Correspondence: Establishing accurate correspondences between measurements and agents is challenging when multiple agents operate in close proximity. Ambiguities and occlusions can lead to incorrect data association, affecting the accuracy of pose estimation.

Consistency and Synchronization: Maintaining consistency and synchronization among distributed agents is crucial. Clock synchronization, alignment of coordinate frames, and handling of time delays are essential to ensure accurate fusion of pose measurements and collaborative decision-making.

Scalability and Computational Efficiency: As the number of agents increases, scalability and computational efficiency become significant concerns. Efficient algorithms and communication protocols are required to manage the increasing amount of data exchange and computation while meeting real-time constraints.

1.5 Conclusion

By thoroughly reviewing the literature, this chapter provides a strong foundation for the subsequent chapters of the thesis. It sets the stage for the exploration of Deep Learning and

Reinforcement Learning-based optimization approaches in pose estimation and the development of a novel approach for collaborative pose estimation in multi-agent systems.

This literature review establishes the importance of pose estimation in robotics and highlights the diverse range of approaches and techniques that have been developed. It serves as a valuable resource for understanding the current state of the field and identifying research gaps and opportunities for further advancement in the domain of pose estimation for robotic systems.

Chapter 2

Optimization techniques and tools

2.1 Introduction

In this chapter, we will discuss how deep learning and reinforcement learning can be used as powerful tools to optimize and improve the accuracy and robustness of pose estimation techniques. We will also highlight the software tools used in this work.

2.2 Deep Learning

In recent years, deep learning techniques have also been applied to pose estimation in robotics. These techniques have shown promising results, particularly in challenging scenarios where the traditional techniques may fail.

Machine Learning is a scientific field which studies a type of algorithms that solve problems without being explicitly programmed for them. Deep Learning is a specialization of this field, which includes Artificial Neural Networks (ANNs), a group of algorithms inspired by neurobiology with a wide range of applications, used in fields such as self-driving cars, computer vision, fault identification in electronic systems, robotics and more.

2.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are Machine Learning models whose theory has been available for years, but only recently began their widespread use thanks to the evolution of technology and the advent of powerful Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs). Artificial Neural networks consist of a number of neurons, also known as nodes or units, at the input and output of the model, and connections between them. Each set of neurons at the same level of depth is typically called a layer. If there are intermediate layers, they are called hidden layers. An ANN model with more than one hidden layer is typically considered as a Deep Neural Network (DNN). Figure 2.1 demonstrates a typical DNN architecture.

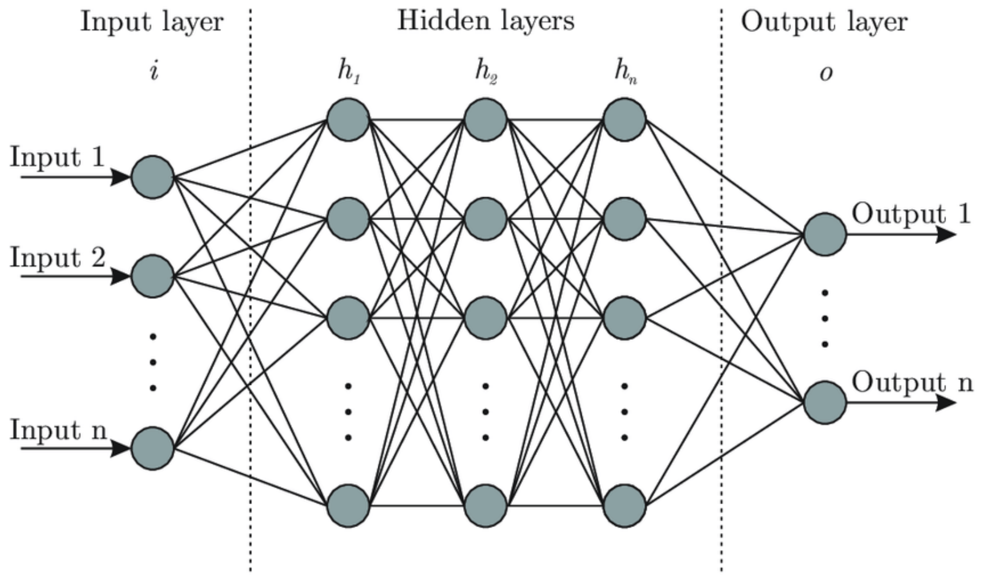

Figure 2.1: The architecture of a Deep Neural Network

Every single neuron receives inputs from other neurons via connections, except those in the first layer, which directly accept input data. The output neurons compose the final result, known as decision or prediction. The number of input neurons is the same as the number of data features and receive only a single value, while the number of output neurons is the same as the number of values to be predicted. Each connection between neurons carries a weight, while each neuron is equipped with an additional bias term. At first, the weights are randomly initialized and they update through an operation called training. Training ends up finding the appropriate weights and biases, utilizing the backpropagation algorithm, which calculates the derivative of each layer's function after every pass of the data through the network, in order to determine the changes that need to be made to the network's weights.[39]

The output of a neuron i is calculated as follows:

$$o_i = f\left(w^T u_i + b\right) \tag{2.1}$$

where w is the weight vector and b the bias of the neuron, u_i the input vector of the corresponding neuron and f the activation function.

2.2.2 Activation functions

The choice of activation functions is crucial in the training process of deep neural networks and ultimately affects their effectiveness. Activation functions are mathematical equations that are applied to the output of each neuron in a neural network. These functions determine whether a neuron should be activated or not based on the input it receives. Some activation functions also help normalize the outputs of the neurons. In DNNs, non-linear activation functions are typically used, which allow for the creation of complex non-linear mappings between the inputs and outputs. These mappings are essential for learning and modeling complex data, such as images, video, audio, and other datasets where the relationships between the inputs and the desired outputs are non-linear.

There are various types of activation functions, each with its own strengths and weaknesses. Some of the commonly used activation functions include:

2.2.2.1 Sigmoid function

The Sigmoid function Also known as logistic function maps any input value to a value between 0 and 1. It has the advantage of leading to clear divisions, as it tends to produce results near the limits of its range. Its output can be directly interpreted as a probability. The sigmoid function is given by

$$s(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

where x is the output value of the neuron. Below, we can see the plot of the sigmoid function :

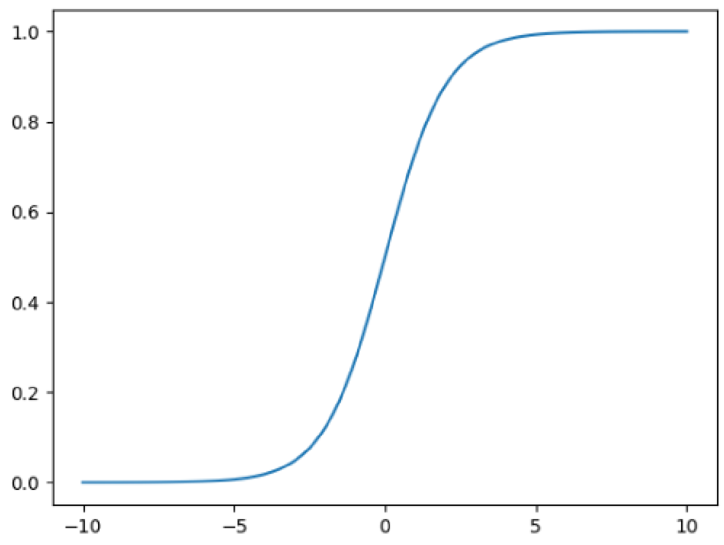

Figure 2.2: Sigmoid function plot

2.2.2.2 Tanh function

The hyperbolic tangent function (tanh) maps any input value to a value between -1 and 1. It is used in problems where the output can be negative. It is given by

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

The advantages mentioned for the sigmoid function also apply to this one, as it is a scaled and shifted version of the former:

$$tanh(x) = 2s(2x) - 1 \tag{2.4}$$

where s is the Sigmoid function. Below, we can see its plot:

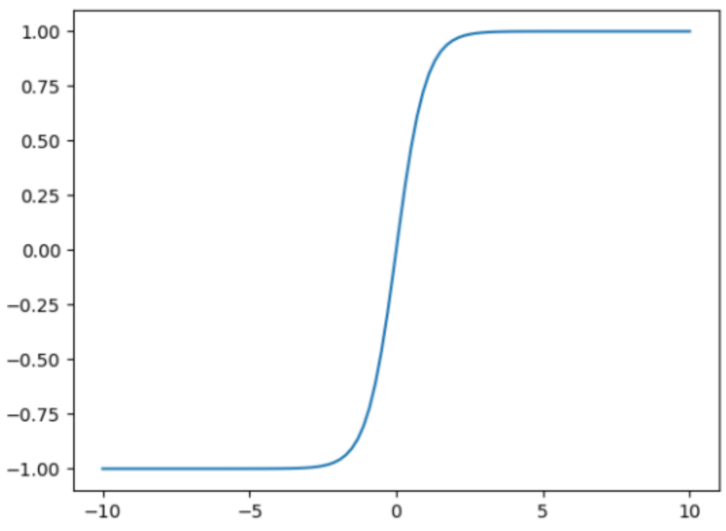

Figure 2.3: Tanh function plot

2.2.2.3 ReLU function

The ReLU (Rectified Linear Unit) function is one of the most widely used activation functions in deep learning due to its simplicity and computational efficiency. It returns the input if it is positive and returns 0 if it is negative. It is given by

$$R(x) = max(0, x) \tag{2.5}$$

The issue of this function is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. Below, we can see its plot:

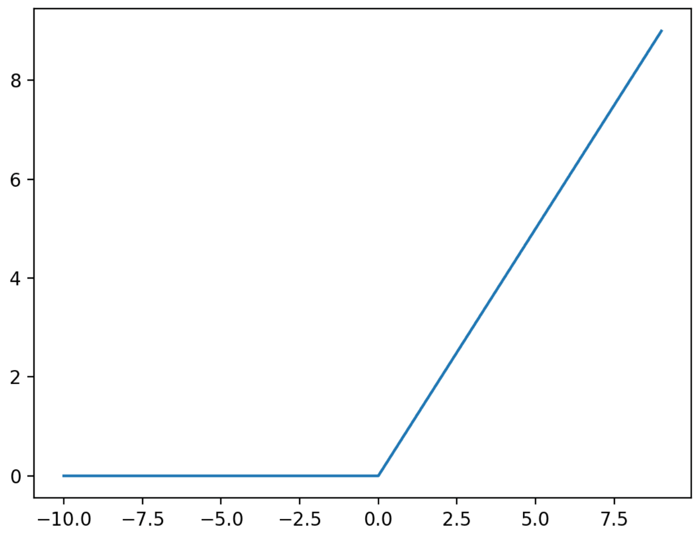

Figure 2.4: The ReLU function plot

2.2.2.4 ELU function

The Exponential Linear Unit (ELU) is an activation function for neural networks. In contrast to ReLUs, ELUs have negative values which allows them to push mean unit activations closer to zero like batch normalization but with lower computational complexity. It is given by

$$ELU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha\left(e^{x} - 1\right) & \text{if } x < 0 \end{cases} \tag{2.6}$$

where α is a real parameter. Below, we can see its plot with different values of α:

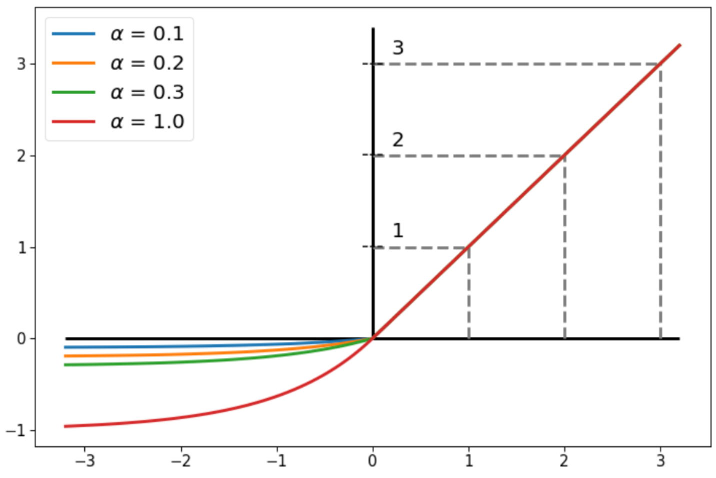

Figure 2.5: The ELU function plot with different values of α

Activation functions play a crucial role in deep learning as they determine the output of each neuron and ultimately the output of the entire neural network. The choice of activation function can impact the performance of the network, and selecting the appropriate activation function for a particular problem is an important consideration.

2.2.3 Cost functions

Deep Learning algorithms have the objective of finding the most suitable values for the optimization parameters, which are the weights w and the biases b of each layer of the network. In order to achieve this, they aim to minimize a cost function (also known as loss function). It is a measurement which determines to what extent an NN is able to predict a result correctly. More specifically, the goal is to find those parameters that minimize the cost function's output.[39]

Cost functions, also known as loss functions, are used in deep learning to measure how well a neural network is performing on a given task. The goal of training a neural network is to minimize the cost function, which means that the network is able to make accurate predictions.

There is a large variety of cost functions used in Deep Learning and each one has its own strengths and weaknesses. The most basic ones are:

2.2.3.1 Mean Squared Error

One of the most commonly used cost functions is the mean squared error (MSE). This function is used in regression problems, where the goal is to predict a continuous value. The MSE is defined as the average squared difference between the predicted value and the actual value as follows

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.7}$$

where y_i is the expected value, $\hat{y}_i$ is the predicted value, n the number of samples in dataset and i the index of sample. The MSE function penalizes large errors more heavily than small errors due to the square. Mean Absolute Error (MAE) is an alternative, which employs the norm instead.

2.2.3.2 Mean Absolute Error

Mean Absolute Error is defined as the average norm of errors between the predicted and actual values as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2.8}$$

It is important to note that MAE is not sensitive to outliers.

2.2.3.3 Root Mean Squared Error

Root Mean Squared error is the extension of MSE. It is measured as the average of square root of sum of squared differences between predictions and actual observations as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{2.9}$$

2.2.4 Backpropagation

Using the cost functions described in the previous section, the appropriate optimization parameters of a Neural Network can be found by minimizing the cost function. To achieve this, the backpropagation algorithm is used, which calculates the gradient of the cost function with respect to the bias and the weights of a network. It iterates through the layers backwards calculating the gradients for each neuron of each layer. By applying the chain rule, it is possible to make these calculations for every layer and not only the last one. [38]

Algorithm 1: Backpropagation Algorithm for Neural Networks

```
Input: Input data $X$, target data $Y$, learning rate $\alpha$, number of hidden layers $L$,
       number of nodes in each hidden layer $n_l$, activation function $g$, output function
       $f$, loss function $J$
Output: Optimized weights $w$ and biases $b$
Initialize weights $w^{(l)}$ and biases $b^{(l)}$ for each layer $l$ randomly;
for each training example $(x, y)$ do
    Forward pass
    Compute the output of the network for input $x$:    $a^{(0)} = x$;
        for $l = 1$ to $L$:  $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$;  $a^{(l)} = g(z^{(l)})$;  $y' = f(a^{(L)})$;
    Compute the error at the output layer:
        $\delta^{(L)} = \nabla_{a^{(L)}} J(y, y') \odot g'(z^{(L)})$;
    Backward pass
    Compute the error at each hidden layer:
        for $l = L - 1$ to 1:    $\delta^{(l)} = (w^{(l+1)})^T \delta^{(l+1)} \odot g'(z^{(l)})$;
    Compute the gradients for the weights and biases:
        for $l = 1$ to $L$:    $\nabla_{w^{(l)}} J = \delta^{(l)}(a^{(l-1)})^T$;    $\nabla_{b^{(l)}} J = \delta^{(l)}$;
    Update the weights and biases using the gradients and the learning rate:
        for $l = 1$ to $L$:    $w^{(l)} \leftarrow w^{(l)} - \alpha\nabla_{w^{(l)}} J$;    $b^{(l)} \leftarrow b^{(l)} - \alpha\nabla_{b^{(l)}} J$;
end
```

$\odot$ refers to element-wise multiplication.

2.2.5 Optimizers and training

Using the cost functions and the backpropagation algorithm, DNNs can be trained with an optimizer. The term training, also known as fitting, refers to finding the appropriate bias and weights for the whole network. In order to do this, they need to be updated by exploiting the gradients that the backpropagation algorithm calculates.

The most basic optimizers are described below.

2.2.5.1 Gradient Descent

Gradient Descent is the most basic but most used optimization algorithm. It is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates the new parameters by taking steps proportional to the negative of the provided gradients after every iteration as follow:

$$\theta_{new} = \theta_{old} - \alpha \nabla_{\theta_{old}} L \tag{2.10}$$

where θ is the learning parameters vector, α the learning rate and L the loss function.

The learning rate also called step size controls how quickly or slowly a Neural Network changes its learning parameters. A large α allows the model to learn faster. A smaller α may allow the model to learn a more optimal, but may take significantly longer time to train. At extremes, too large α will result in oscillations of the NN performance over the training epochs. On the other side, a NN using too small α may never converge or may get stuck on a sub-optimal solution. Therewith, α is very important for the training procedure and must be carefully chosen.

2.2.5.2 Stochastic Gradient Descent (SGD)

It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. It calculates the new parameters as follow:

$$\theta_{new} = \theta_{old} - \alpha \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.11}$$

where $\{x_i, y_i\}$ are the training examples. As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities.

2.2.5.3 Adaptive Gradient (AdaGrad)

One of the disadvantages of all the optimizers explained is that the learning rate is constant for all parameters and for each cycle. This optimizer changes the learning rate for each parameter and at every time step t. It's a type second order optimization algorithm. It works on the derivative of an error function. It achieves this by scaling the latter by the square root of the cumulative sum of squared gradients G_i for every training example, using the same update rule with SGD 2.11 as follow :

$$\theta_{new} = \theta_{old} - \frac{\alpha}{\sqrt{G_i + \epsilon}} \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.12}$$

where

$$G_i = \sum_{j=0}^{i} \nabla_{\theta_{old}} L\left(x_j, y_j\right) \left(\nabla_{\theta_{old}} L\left(x_j, y_j\right)\right)^T \tag{2.13}$$

The limitation of this optimizer is that if the neural network is deep, the learning rate becomes very small number which will cause dead neuron problem.

2.2.5.4 Root Mean Square Propagation (RMSProp)

RMS-Prop is a special version of Adagrad in which the learning rate is an Exponential Moving Average (EMA) of the gradients instead of the cumulative sum of squared gradients as follow:

$$\theta_{new} = \theta_{old} - \frac{\alpha}{\sqrt{v+\epsilon}} \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.14}$$

where

$$v_{new} = \beta\, v_{old} - (1-\beta)(\nabla_{\theta_{old}} L\left(x_i, y_i\right))^2 \tag{2.15}$$

2.2.5.5 Adaptive Moment Estimation (Adam)

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It combines RMSProp with momentum as follow :

$$\theta_{new} = \theta_{old} - \alpha \frac{\hat{m}_{old}}{\sqrt{\hat{v}_{old}+\epsilon}} \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.16}$$

where

$$\hat{v}_{new} = \frac{v_{new}}{1-\beta_2} \tag{2.17}$$

$$\hat{m}_{new} = \frac{m_{new}}{1-\beta_1} \tag{2.18}$$

$$v_{new} = \beta_2 v_{old} + (1-\beta_2) \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.19}$$

$$m_{new} = \beta_1 m_{old} + (1-\beta_1) \nabla_{\theta_{old}} L\left(x_i, y_i\right) \tag{2.20}$$

where m and v are values of the first moment which is the Mean and the second moment which is the uncentered variance of the gradients, respectively.

Adam is widely used in DNNs training. The reason for its widespread use is the fact that it tends to converge much faster than its competitors in most problems, while at the same time finding more optimal solutions in the parameters space. However, it is important to note that the choice of optimization algorithm may depend on the specific problem and the characteristics of the data being used, and other optimization algorithms may be more suitable in certain scenarios.

2.2.6 Overfitting

A well-known problem with NNs is their tendency to overfit, that is, to memorize training data instead of generalizing their knowledge to unseen data. This problem becomes larger when the number of training samples is limited. A model with a large number of parameters can describe wide range of events. Though, if the model is able to correctly predict the output for the data given as input, this doesn't mean that it will be able to do the same for new samples. The true ability of a model lies in its potential to correctly predict the labels of unseen samples, which is called generalization in DL.

To determine whether a model is overfitting, we observe its error rate on the training set during the iterations and compare it to that of an unknown set of data. In such a case, while the error rate is constantly reducing for both the training and the unknown data, at some point, it begins to grow again, only for the unknown. From that moment on, it is safe to assume that the model begins to overfit.[39]

In addition to carefully selecting hyperparameters, there are many solutions to overfitting.

2.2.6.1 Early stopping

Early stopping is a technique widely used in NNs, where the training process is stopped before the model has fully converged. This is done by observing the validation error during training, and stopping when it starts to increase. Early stopping prevents the model from overfitting to the training data. However, it may not provide the best results, since NNs often stop improving for a number of iterations, but then the validation loss starts decreasing again.

2.2.6.2 Regularization

Regularization is a technique that introduces a penalty term in the loss function of the neural network. This penalty term discourages the weights from taking on large values, which in turn prevents the model from overfitting. Two popular types of regularization are L1 regularization, which adds the absolute values of the weights to the loss function, and L2 regularization, which adds the square of the weights to the loss function. At the same time, a parameter λ is used, which defines the importance of the regularization term.

2.2.6.3 Dropout

Dropout is a regularization technique where randomly selected neurons are temporarily “turned off” in the neural network on every training iteration. This prevents the network from relying too heavily on any single neuron or feature and encourages it to learn more robust representations.

2.2.6.4 Batch normalization

Normalizing the inputs is proven to speed up and stabilize training. Batch Normalization, based on this observation, applies normalization at the level of the hidden layers. Each unit’s input is subtracted with the mean and divided by the standard deviation of all the unit’s inputs for a mini-batch of size.

2.2.7 Recurrent Neural Networks (RNN)

the most common way to process sequence data using deep learning is by employing Recurrent Neural Networks (RNNs) [35]. As the name suggests, RNNs have a recurrent aspect in their operation, which is the repeated processing and updating of their internal state. Using their internal representation, RNNs are capable of exhibiting dynamic temporal behavior and processing sequences of arbitrary length, while being able to model the long-term relationships between distant points in a sequence.

2.2.7.1 RNN architecture

Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:

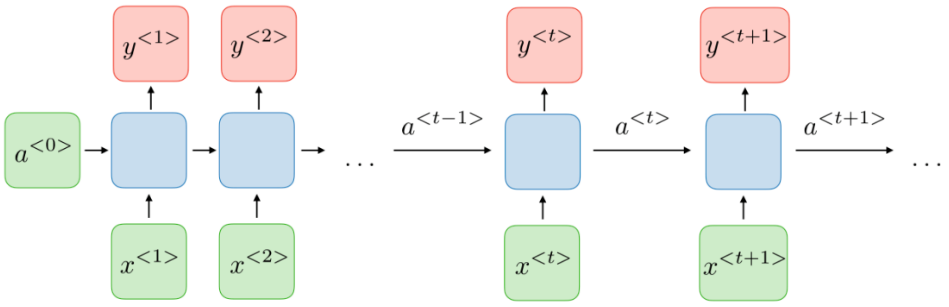

Figure 2.6: Recurrent Neural Network architecture

For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1 \left(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a \right) \tag{2.21}$$

and

$$y^{<t>} = g_2 \left(W_{ya} a^{<t>} + b_y \right) \tag{2.22}$$

where W_{ax}, W_{aa}, W_{ya}, b_a and b_y are coefficients that are shared temporally and g_1, g_2 activation functions.

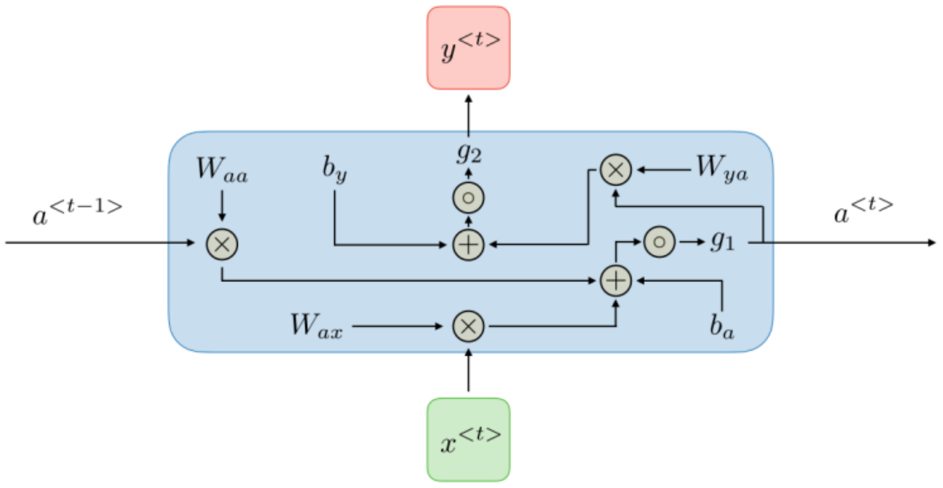

Figure 2.7: Neuron architecture in Recurrent Neural Network

The advantages of RNNs are:

- Possibility of processing input of any length
- Model size not increasing with size of input
- Computation takes into account historical information
- Weights are shared across time

There disadvantages are:

- Computation being slow
- Difficulty of accessing information from a long time ago
- Cannot consider any future input for the current state

2.2.7.2 Loss function

In the case of a recurrent neural network, the loss function L of all time steps is defined based on the loss at every time step as follows:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L(\hat{y}^{<t>}, y^{<t>}) \tag{2.23}$$

2.2.7.3 Backpropagation through time

Backpropagation is done at each point in time. At timestep T, the gradient of the loss L with respect to weight matrix W is expressed as follows:

$$\nabla_W L^{(T)} = \sum_{t=1}^{T} \nabla_W L^{(T)}|_{(t)} \tag{2.24}$$

2.2.7.4 Handling long term dependencies

The most common activation functions used in RNN modules are Sigmoid, Tanh and ReLU.

2.2.7.5 Vanishing/exploding gradient

The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers. Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU.

2.2.8 Variant RNN architectures

These are a variant network architecture of RNNs.

2.2.8.1 Bidirectional Recurrent Neural Networks (BRNN)

A Bidirectional Recurrent Neural Network (BRNN) [30] is a type of RNNs that incorporates bidirectional information flow during sequence processing tasks. Unlike traditional RNNs that process sequences in a single direction, BRNNs utilize two separate recurrent layers—one that processes the sequence in a forward direction and another that processes it in a backward direction. By combining information from both directions, BRNNs can capture contextual dependencies from past and future elements of a sequence simultaneously.

The architecture of a BRNN allows it to capture not only local dependencies within a sequence but also global dependencies that span across the entire sequence. This is particularly advantageous in tasks such as speech recognition, natural language processing, and handwriting recognition, where context from both directions is crucial for accurate predictions.

During the training process, a BRNN undergoes a forward pass, where it processes the input sequence in the forward direction, and a backward pass, where it processes the sequence in the reverse direction. The hidden states from both passes are then combined to generate the final prediction or output. This bidirectional nature enhances the network's ability to model complex dependencies and make informed predictions based on both past and future information.

2.2.8.2 Long Short-Term Memory (LSTM)

This is a popular RNN architecture, which was introduced by Sepp Hochreiter and Juergen Schmidhuber as a solution to vanishing gradient problem. In their paper [11], they work to address the problem of long-term dependencies. LSTM architecture overcomes the vanishing gradient problem and enables the modeling of long-term dependencies in sequential data. LSTMs are specifically designed to retain information over long periods, making them well-suited for tasks that require the analysis of context and memory over extended sequences.

The key innovation of LSTMs lies in their memory cell, which is capable of selectively remembering or forgetting information based on the input and the network's learned weights. The memory cell maintains an internal state that can be updated, allowing LSTMs to store relevant information for extended periods and prevent the degradation of gradients during backpropagation.

In an LSTM, the memory cell is accompanied by three gating mechanisms: the input gate, forget gate, and output gate. These gates control the flow of information within the network by selectively allowing or inhibiting the information coming from different sources. The input gate determines how much new information should be stored in the memory cell, while the forget gate regulates the amount of previously stored information to be discarded. Lastly, the output gate determines the amount of information to be outputted based on the current context.

Through these mechanisms, LSTMs can effectively capture and utilize long-term dependencies in sequential data. They have demonstrated exceptional performance in various tasks, such as speech recognition, machine translation, sentiment analysis, and time series prediction. LSTMs have significantly advanced the capabilities of recurrent neural networks and continue to be a popular choice for modeling sequential data in many fields.

2.2.8.3 Gated Recurrent Units (GRUs)

This variant is similar the LSTMs as it also works to address the short-term memory problem of RNN models. Gated Recurrent Units (GRUs) [8] are a variant of RNNs that address some limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in capturing long-range dependencies in sequential data. GRUs were introduced as a simpler alternative to LSTM networks, while still offering similar capabilities in modeling sequential information.

The key feature of GRUs is the presence of two gating mechanisms: the reset gate and the update gate. These gates control the flow of information within the network, allowing it to selectively retain or discard information as needed. The reset gate determines which parts of the previous hidden state should be forgotten, allowing the model to focus on relevant information for the current time step. Meanwhile, the update gate regulates the amount of new information to be stored in the current hidden state, allowing the model to adapt and respond to new inputs.

GRUs strike a balance between retaining long-term dependencies and managing computational complexity. They have fewer parameters than LSTMs, which makes them computationally more efficient and easier to train on smaller datasets. Despite their simplicity, GRUs have demonstrated strong performance in various sequence modeling tasks, including machine translation, speech recognition, sentiment analysis, and natural language processing.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is the third branch of machine learning, along with supervised and unsupervised learning. The RL framework allows an agent to learn through trial and error, where it learns from its experience to make better decisions in the future. By acting in the environment, the RL agent receives a reward. Its goal is to learn to choose the actions that maximize the expected cumulative reward over time. In other words, by observing the results of the actions it takes in the environment, the agent tries to learn an optimal sequence of actions to take to achieve its goal [4, 15, 25, 29, 34]. The agent interacts with an environment that is in most cases considered as a Markov Decision Process or MDP.

2.3.1 Overview and mathematical formulation

Figure 2.8 provides a schematic overview of the reinforcement learning framework. An RL agent senses the state of its environment and learns to take appropriate actions in order to achieve an optimal amount of immediate or delayed rewards. Specifically, the RL agent arrives at a sequence of different states s_k in S by performing actions a_k in A, with the selected actions leading to positive or negative rewards r_k used for learning. The sets S and A denote the sets of possible states and actions, respectively.

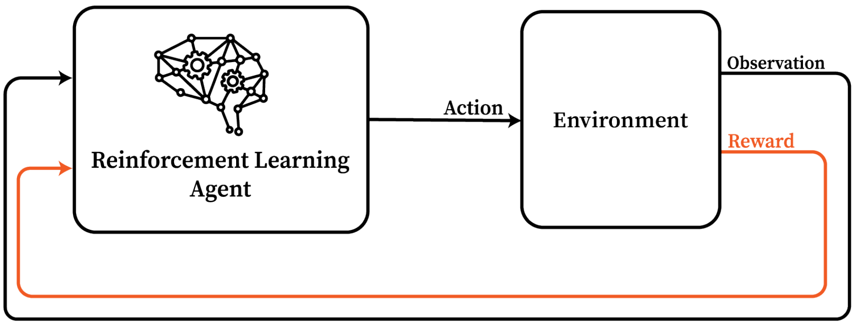

Figure 2.8: Schematic overview of the reinforcement learning framework

2.3.1.1 Markov Decision Process

The Markov Decision Process (MDP) is a mathematical framework for modeling decision problems in which outcomes are partly random and partly under the control of a decision maker. The agent, in this case, interacts with the environment over a sequence of discrete time steps.

At each time step, the agent observes the current state of the environment and takes an action. The environment then transitions to a new state, and the agent receives a reward based on the transition. The goal of the agent is to learn a policy that maximizes the expected cumulative reward over time.

A key assumption of an MDP is the Markov property, which states that the current state contains all relevant information for decision making, and that future states and rewards depend only on the current state and action, not the history of previous states and actions. As explained in [4], an MDP consists of a set of states S, a set of actions A, and a set of rewards R, along with P the probability (Pr) of transitioning from state s_k at time t_k to state s_{k+1} at time t_{k+1} given action a_k,

$$P(s', s, a) = Pr(s_{k+1} = s'|s_k = s, a_k = a). \tag{2.25}$$

and a reward function R

$$R(s', s, a) = Pr(r_{k+1}|s_{k+1} = s', s_k = s, a_k = a). \tag{2.26}$$

2.3.1.2 Policy

An RL agent recognizes its environment state s and takes an action a with a policy π that is optimized by learning to maximize future reward r. Reinforcement learning is usually formulated as an optimization problem of learning a policy $\pi(s, a)$.

$$\pi(s, a) = Pr(a = a, s = s). \tag{2.27}$$

This is the probability of taking an action in a given state to maximize the total future reward. In its simplest formulation, a policy can be a lookup table defined over discrete state and action spaces S and A, respectively. However, for most problems, this rule is too expensive to represent and learn, and pi must be represented as an approximate function parameterized by the low-dimensional vector theta: $\pi(s, a) = pi(s, a, \theta)$. This parameterized function is often called $\pi_\theta(s, a)$. Function approximation is the basis of deep reinforcement learning, and deep neural networks can be used to represent these complex functions.

2.3.1.3 Value function

Given a policy π, the value function quantifies the desirability of being in a given state:

$$V_\pi(s) = \mathbb{E}\left(\sum_k \gamma^k r_k | s_0 = s\right), \tag{2.28}$$

where $\mathbb{E}$ is the expected reward over time steps k, subject to a discount rate γ, future rewards are discounted, reflecting the economic principle that current rewards are more valuable than future rewards. Often the subscript π is omitted from the value function, in which case we refer to the value function for the best possible policy:

$$V(s) = \max_\pi \mathbb{E}\left(\sum_k \gamma^k r_k | s_0 = s\right), \tag{2.29}$$

which implies that

$$V(s) = \max_\pi \mathbb{E}(r_0 + \gamma V(s')), \tag{2.30}$$

where $s_0 = s_{k+1}$ is the next state after $s = s_k$ given action a_k, and the expectation is over actions selected from the optimal policy π. This expression, known as Bellman's equation [3], is a statement of Bellman's principle of optimality, and it is a central result that underpins modern RL. Given the value function, it is possible to extract the optimal policy as

$$\pi = argmax_{\pi}\mathbb{E}(r_0 + \gamma V(s')), \tag{2.31}$$

2.3.1.4 Quality function

The action-value function, the Quality function, or the Q-function $Q_\pi(s, a)$ [4], is defined as the maximum expected return achievable by following a particular policy π, after seeing some state s and then taking some action a. The optimal action-value function:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}(r_k | s_k = s, a_k = a, \pi) \tag{2.32}$$

corresponds with the optimal policy. The optimal action-value function follows an important identity, which is known as the Bellman equation [3]:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} Pr(s,a,s') \max_{\pi} Q^*(s',a') \tag{2.33}$$

2.3.2 Model-Based and Model-free Reinforcement Learning

2.3.2.1 Model-Based RL

Model-based RL techniques do not involve learning an optimal strategy through trial-and-error experience, so some may not consider them RL. It is possible to learn a model through trial and error, and then use that model with these techniques, which would be considered RL. For the simple cases, it is possible to learn either the optimal policy or value function through what is called policy iteration or value iteration, which are forms of dynamic programming using the Bellman equation. Dynamic programming is a powerful approach used for general optimal nonlinear control and reinforcement learning, among other things. These algorithms provide a mathematically simplified optimization framework that helps to introduce essential concepts used throughout. For more details, see [4].

2.3.2.2 Model-free RL

Model-free RL is an approach in which the agent learns to make decisions directly from experience, without constructing a model of the environment. This means that the agent learns a policy, or a mapping from states to actions, that maximizes cumulative reward without explicitly modeling the transition probabilities or rewards of the environment.

There are several types of model-free RL algorithms, such as Q-learning [9, 29], State-Action-Reward-State-Action (SARSA), and policy gradient methods. Q-learning is a popular model-free algorithm that learns an action-value function that estimates the expected cumulative reward of taking an action in a given state and then following a given policy. SARSA is another model-free algorithm that learns the expected cumulative reward of taking an action in a given state, following a given policy thereafter, and choosing the next action based on the same policy. Policy gradient methods optimize the policy directly, using gradient ascent to maximize the expected cumulative reward.

Model-free RL algorithms are useful in situations where the environment is complex, and it is difficult or impossible to construct an accurate model. They are also useful when the goal is to learn a policy that can be applied to a variety of tasks or environments without having to learn a new model for each one.

2.3.3 Deep Reinforcement Learning

Deep reinforcement learning is one of the most exciting areas in machine learning and control theory, and it is one of the most promising avenues of research toward generalized artificial intelligence. Deep learning has revolutionized our ability to represent complex functions from data, providing a number of architectures for achieving human-level performance in complex tasks such as image recognition and natural language processing. Classical reinforcement learning suffers from a representation problem because many of the relevant functions, such as the policy π, the value function V, and the quality function Q, can be highly complex functions defined over a very high-dimensional or even continuous state and action space. Thus, deep learning provides a powerful tool to improve these representations. It is possible to use deep learning in several ways to approximate the various functions used in RL, or to model the environment more generally. Typically, the central challenge is to identify and represent key features in a high-dimensional state space. For example, the policy $\pi(a, s)$ may now be approximated by :

$$\pi(s, a) \approx \pi(s, a, \theta), \tag{2.34}$$

where θ represent the weights of a neural network.

2.3.3.1 Deep Q-learning

Like the policy π, it is possible to approximate the Q function through some parameterization θ,

$$Q(s, a) \approx Q(s, a, \theta), \tag{2.35}$$

here θ represents the weights of a deep neural network. In deep Q-learning [23], the neural network takes the current state as input and outputs a Q-value for each possible action. The algorithm then selects and executes the action with the highest Q-value. The resulting experience (i.e., state, action, reward, and next state) is stored in a memory buffer that is later used to train the neural network.

During training, the neural network is optimized to minimize the difference between the predicted Q-values and the actual Q-values obtained from the experience buffer. This is done using a variant of stochastic gradient descent called Q-learning.

Deep Q-learning has been used to achieve state-of-the-art performance on a variety of challenging tasks. However, while deep Q-learning is a powerful algorithm for learning optimal policies in discrete state and action spaces, it is not directly applicable to continuous state and action spaces. The reason is that the Q-function then becomes a continuous function, and it becomes infeasible to represent and learn it with a finite number of parameters. However, there are extensions of deep Q-learning that address this limitation. One such extension is Deep Deterministic Policy Gradient and Twin Delayed Deep Deterministic Policy Gradient.

2.3.3.2 Actor-Critic methodology

Actor-Critic is a reinforcement learning method that combines the advantages of both value-based and policy-based methods. It has two components: the actor and the critic. Figure (2.9) illustrate this methodologies.

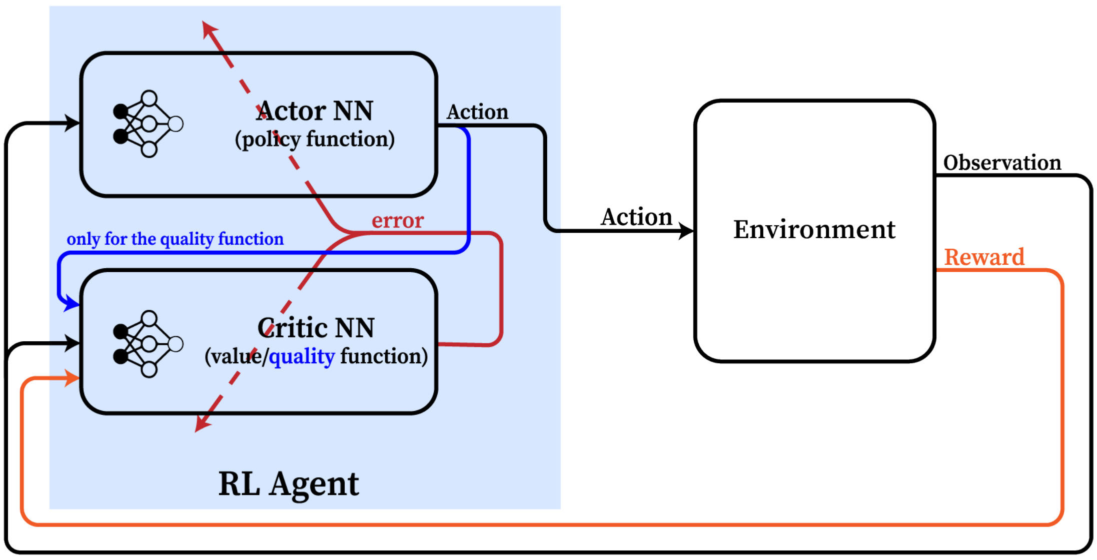

Figure 2.9: Schematic overview of the Actor-Critic Reinforcement Learning agent

The actor is responsible for selecting actions based on the current policy, which maps states to actions. The policy is represented by a neural network that takes the state as input and outputs a probability distribution over the action space. During training, the actor's parameters are updated to maximize the expected cumulative reward from following the policy.

The critic, on the other hand, is responsible for estimating the value function or expected cumulative reward obtained from a given state under the current policy. The critic is also represented by a neural network, but its output is a scalar value representing the expected cumulative reward. The parameters of the critic are updated using the temporal difference (TD) learning rule [4], which involves computing the difference between the estimated value and the actual reward obtained, and minimizing this difference.

2.3.3.3 Pretraining the Actor and Critic Networks

Pretraining the actor or critic network can improve the performance and speed up the learning process in actor-critic algorithms. This can help initialize the network weights to a better starting point and speed up the convergence of the learning process.

For the actor network, pretraining can be done by using supervised learning to learn a policy that maps states to actions before starting the reinforcement learning process. The idea is to use a dataset of expert demonstrations or simulations to train the network to imitate the expert's behavior. This pretraining step can improve the actor's ability to explore the state space and find better policies.

For the critic network, pretraining can be done by using unsupervised learning to learn a representation of the state space before starting the reinforcement learning process. The idea is to use a dataset of state observations to train the network to encode the state information into a compact and informative representation. This pretraining step can improve the critic's ability to accurately estimate the value function and reduce the number of samples needed for training.

2.3.4 RL agents

Reinforcement learning agents designed for continuous control tasks use a different approach than those designed for discrete tasks. In continuous control tasks, the agent must choose a continuous action from a continuous action space. This presents a unique challenge to the agent because the space of possible actions is infinite and continuous.

There are several types of agents that can handle a continuous state and action space, such as Deep Deterministic Policy Gradient, Twin Delayed Deep Deterministic Policy Gradient, Soft Actor-Critic and Proximal Policy Optimization. RL agents designed for continuous control tasks use a combination of neural networks, value functions, and policy optimization techniques to learn optimal policies in the continuous action space.

2.3.4.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a deep reinforcement learning algorithm specifically designed for continuous control problems. It combines two important components of reinforcement learning: Q-learning and policy gradient methods [21, 14].

The basic idea behind DDPG is to use a neural network, the actor network, to learn a deterministic policy that maps state to action. The output of the actor network is the action that the agent takes in a given state. The actor network is trained using a policy gradient algorithm that updates the weights of the network in the direction that increases the expected reward.

At the same time, DDPG uses another neural network, the critic network, to approximate the Q-value function. The critic network takes both the current state and the action taken by the actor network as inputs and outputs the Q-value of that state-action pair. The critic network is trained using a variant of the Q-learning algorithm that minimizes the mean squared error between the predicted Q-value and the actual Q-value.

DDPG uses a replay buffer to store the agent's experiences in the form of tuples (state, action, reward, next state). During training, a batch of experiences is randomly sampled from the replay buffer and used to update both the actor and critic networks. This process of sampling from the replay buffer and updating the networks is repeated many times until convergence is reached.

2.3.4.2 Twin Delayed DDPG

Twin Delayed DDPG (TD3) is a variant of DDPG that improves its performance in several ways.

A major improvement in TD3 is the use of two critic networks instead of one, which helps reduce overestimation of Q-values. The two critic networks are trained independently, and their Q-value predictions are averaged to compute the Q-value used to update the actor and target critic networks. This approach helps make the Q-value estimates more robust and less sensitive to noise in the environment [40, 14].

Another improvement in TD3 is the use of a delayed policy update mechanism, where the actor network is updated less frequently than the critic networks. This delay reduces the correlation between actor and critic updates and improves the stability of the learning process.

2.3.4.3 Soft Actor-Critic

Soft Actor-Critic (SAC) is a deep reinforcement learning algorithm designed for continuous control tasks. It is based on the actor-critical framework like DDPG and TD3, but introduces several improvements to make it more stable and effective [14].

A major improvement in SAC is the use of a maximum entropy objective. In addition to maximizing the expected reward, SAC also maximizes the entropy of the policy. The entropy of the policy represents the degree of randomness or exploration in the actions taken by the agent. By maximizing entropy, SAC encourages the agent to explore more and learn a wider variety of behaviors.

Another improvement in SAC is the use of two Q-functions instead of one as in TD3. The two Q-functions are trained independently and their estimates are averaged to reduce overestimation and improve robustness.

SAC also uses a target entropy regularization term to ensure that the policy remains sufficiently random. The target entropy is a user-defined value that represents the desired level of randomness in the policy. The entropy regularization term ensures that the policy does not become too deterministic and encourages exploration.

In addition, SAC uses a replay buffer to store experience and a delayed policy update mechanism to reduce the correlation between actor and critic updates. SAC also uses an automatic entropy tuning mechanism to adjust the weight of the entropy term during training.

2.3.4.4 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is another popular reinforcement learning algorithm designed to train policies for continuous and discrete action spaces. PPO is a type of policy gradient algorithm that iteratively updates the policy parameters to maximize the expected reward [37, 14].

The key idea behind PPO is to balance the exploration-exploitation tradeoff by limiting the size of the policy update in each iteration. Specifically, PPO uses a surrogate objective function that constrains the difference between the new and old policies. This constraint ensures that the updated policy does not deviate too far from the previous policy, thereby avoiding large policy updates that can destabilize the learning process.

PPO also uses a clipping mechanism to further constrain the policy update. The clipping mechanism limits the policy change to a small threshold, which prevents the policy from changing too much at once. This constraint helps ensure that the updated policy remains close to the old policy while still allowing for effective discovery.

In addition, PPO uses a value function to estimate the state value of the policy. The value function is trained concurrently with the policy using a variant of the actor-critical algorithm. The value function is used to compute the advantage function, which measures how much better an action is compared to the average action in a given state.

2.4 Deep and Reinforcement Learning in robotics

DL techniques offer significant advantages in handling complex data patterns and extracting meaningful features. They enable robots to learn from large datasets and make accurate predictions or classifications. DL models can capture non-linear relationships and complex dependencies, making them well-suited for a wide range of robotics applications.

RL, on the other hand, provides a mechanism for optimizing the behavior of robots through trial and error. RL agents can adapt and learn from their interactions with the environment, allowing for continuous improvement and adaptation. RL algorithms enable robots to learn effective policies and strategies, enhancing their autonomy and decision-making capabilities.

By combining DL and RL, we can address complex challenges in robotics, such as perception, control, planning and optimization. These techniques have the potential to enhance the accuracy, robustness, adaptability and efficiency of robotic systems, enabling them to operate more effectively in real-world scenarios.

2.5 Softwares

In this section, we present the different software tools that have been used in this thesis to implement and evaluate the proposed reinforcement learning and deep learning-based optimization techniques for pose estimation in robotics. The selected software tools provide a diverse range of capabilities and functionalities, enabling the development and testing of various algorithms and simulations.

2.5.1 MATLAB

MATLAB [13] is a widely used numerical computing environment that offers a comprehensive set of tools for data analysis, algorithm development, and visualization. It provides a convenient platform for implementing and experimenting with pose estimation algorithms, optimization techniques and artificial intelligence models. MATLAB's extensive libraries and functions make it suitable for prototyping and rapid development of algorithms related to pose estimation. Moreover, the availability of the Robotics System Toolbox in MATLAB simplifies the implementation of robotics-related algorithms and simulations.

2.5.2 MATLAB's Reinforcement Learning Toolbox

MATLAB's Reinforcement Learning Toolbox [14] offers a solid foundation in reinforcement learning concepts. It provides an intuitive way to define and solve reinforcement learning problems, including the formulation of states, actions, and rewards. This toolbox supports both model-free and model-based reinforcement learning approaches, enabling researchers to explore a wide range of optimization techniques.

It allows users to create custom environments tailored to the specific optimization tasks and scenarios under investigation. Researchers can define the dynamics of the environment, including system models, constraints, and interactions with the surrounding elements. By customizing the environment, it becomes possible to evaluate the performance of reinforcement learning algorithms under various conditions and assess their suitability for different optimization problems.

Also, It provides a collection of state-of-the-art reinforcement learning algorithms and agent architectures. These include popular algorithms such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Twin Delayed Deep Deterministic Policy Gradient (TD3). Researchers can select and configure the appropriate algorithm and agent architecture based on their specific optimization problem. The toolbox also supports customization and extension of existing algorithms, allowing researchers to tailor them to the unique requirements of their optimization task.

In summary, the Reinforcement Learning Toolbox in MATLAB serves as a powerful resource for researchers working on optimization techniques in various domains. Its comprehensive set of tools, algorithms, and integration capabilities facilitates the development, training, and evaluation of reinforcement learning agents, enabling the exploration of novel approaches and the advancement of optimization methods.

2.5.3 Python

Python [27] has gained immense popularity in the field of data science and machine learning. It offers a vast ecosystem of libraries and frameworks that facilitate the implementation of pose estimation algorithms, deep learning models, and optimization techniques. Libraries such as NumPy, SciPy, and OpenCV provide fundamental functionalities for numerical computations, scientific computing, and computer vision, respectively. Additionally, frameworks like TensorFlow and PyTorch enable the implementation and training of deep learning models for pose estimation tasks. Python's versatility and ease of use make it an excellent choice for implementing and integrating various components of the proposed optimization techniques.

2.5.4 Robot Operation System (ROS)

ROS [33], the Robot Operating System, is a flexible framework for developing robotic systems. It provides a collection of software libraries and tools that simplify the development and integration of robotic components. ROS offers communication mechanisms, data visualization, and hardware abstraction, making it suitable for multi-agent systems and collaborative robotics. It allows seamless communication and coordination between multiple robots, enabling the implementation of collaborative pose estimation algorithms. ROS also provides a rich set of packages for simulating robots and their environments.

2.5.5 Gazebo

Gazebo [18] is a widely used physics-based simulation environment for robotics. It offers a realistic simulation of robots and their interactions with the environment. Gazebo allows the creation of dynamic, multi-agent scenarios for testing and evaluating pose estimation algorithms. It provides accurate sensor modeling, physics simulation, and visualization capabilities, enabling the assessment of the proposed optimization techniques under various conditions. Gazebo's integration with ROS allows for seamless simulation and interaction with robotic systems developed using ROS.

By utilizing these software tools, this thesis leverages the capabilities and functionalities provided by MATLAB, Python, ROS, and Gazebo to implement, evaluate, and validate the proposed reinforcement learning and deep learning based optimization techniques for pose estimation.

These tools offer a powerful and flexible platform for the development and analysis of algorithms, facilitating the exploration and advancement of pose estimation techniques.

2.6 Conclusion

In this chapter, we have explored the potential of Deep Learning and Reinforcement Learning as powerful tools for optimizing and enhancing various aspects of robotics. By leveraging the capabilities of DL and RL, we can unlock new possibilities for improving the performance and capabilities of robotic systems.

Furthermore, the availability of user-friendly software tools has played a crucial role in implementing and evaluating DL and RL approaches in robotics. These tools provide a standardized environment and a wide range of pre-implemented algorithms, enabling researchers to focus on the specific requirements of their robotics tasks.

Chapter 3

Pose estimation for single agent

3.1 Introduction

This chapter focuses on pose estimation for single agent in two different applications: mobile robot and aerial robot. The chapter will present an overview of the different approaches that have been followed including both traditional techniques such as EKF and its optimized versions using deep learning and reinforcement learning. To demonstrate the effectiveness of these techniques, we will discuss various simulations that have been done, highlighting the strengths and limitations of each approach. Furthermore, we will provide a comprehensive analysis of the results obtained and discuss the potential for further improvements and future research directions.

3.2 Mobile robot pose estimation

Pose estimation plays a vital role in enabling mobile robots to navigate autonomously and interact with their surroundings effectively. By accurately perceiving their own position and orientation, these robots can plan optimal paths, avoid obstacles, and execute complex tasks. This capability is essential in a wide range of applications, including autonomous navigation, mapping, object manipulation, and human-robot interaction.

3.2.1 Model presentation

Consider a differential-drive mobile robot as described in previous studies [6, 26].

Figure (3.1) illustrate a schematic representation of the mobile robot in a 2D-cartesian plane and its surrounding environment, with the following notations:

V	linear velocity (m/s)
θ	orientation angle (rad)
ω	angular velocity (rad/m)
L	distance between the left and right wheels (m)
r	radius of the wheels (m)

We refer to the reference point that we know the position of in the field as the ”Anchor Point” or AP. From this AP, we calculate the distance in order to determine the position of the robot.

The system dynamics can be described by the following equations:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix} \tag{3.1}$$

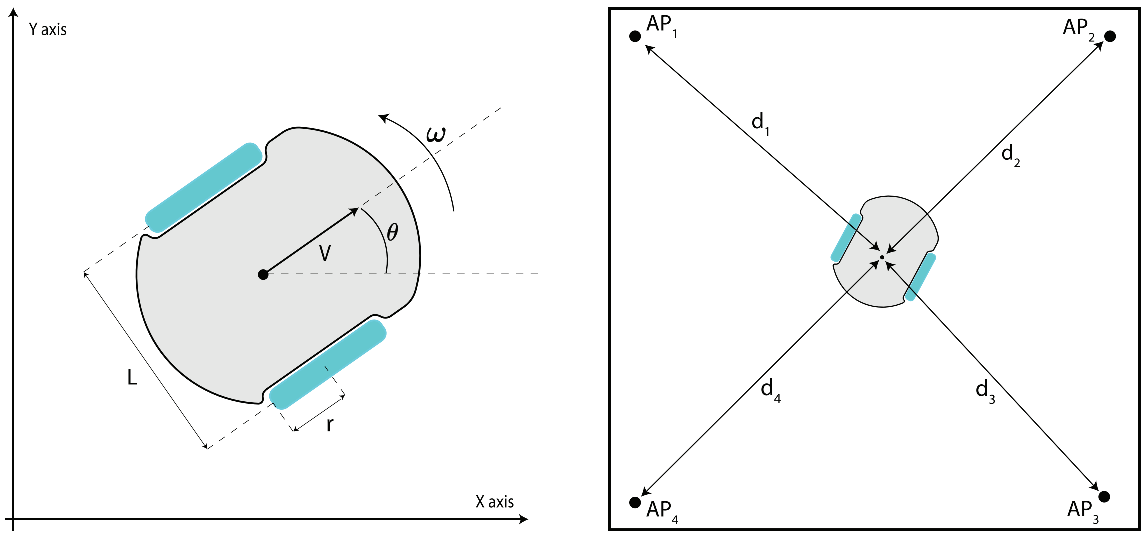

Figure 3.1: Schematic representation of the mobile robot and its environment

Figure (3.2) illustrates the forces applied to the mobile robot, with the corresponding variables defined as follows:

m	mass of the robot (Kg)
I	moment of inertia $(Kg.m^2)$
τ_l	torque of the left wheel $(N.m)$
τ_r	torque of the right wheel $(N.m)$
F_l	force of the left wheel (N)
F_r	right wheel force (N)
d_n	distance to the n^{th} anchor point (m)

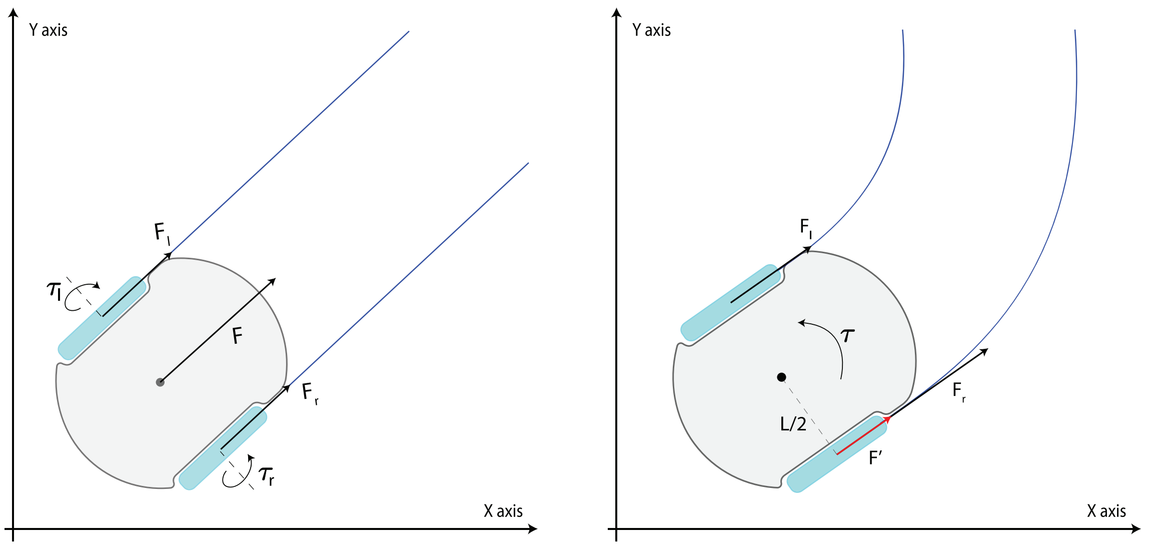

Figure 3.2: Representation of the forces applied to the robot

Applying Newton's second law, we obtain:

$$m\mathbf{a} = \sum \mathbf{F} \tag{3.2}$$
$$m\dot{V} = \mathbf{F}_l + \mathbf{F}_r \tag{3.3}$$
$$\dot{V} = \frac{(\tau_l + \tau_r)}{rm} \tag{3.4}$$

Similarly, for $\dot{\omega}$, we have:

$$I\dot{\omega} = \tau \tag{3.5}$$
$$I\dot{\omega} = \frac{L}{2}\mathbf{F'} \tag{3.6}$$
$$I\dot{\omega} = \frac{L}{2}(\mathbf{F}_l - \mathbf{F}_r) \tag{3.7}$$
$$\dot{\omega} = \frac{L}{2Ir}(\tau_l - \tau_r) \tag{3.8}$$

By combining the results from equations (3.1), (3.4) and (3.8), we obtain the following system dynamics:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \\ \dot{V} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} V\cos(\theta) \\ V\sin(\theta) \\ \omega \\ \frac{(\tau_r+\tau_l)}{rm} \\ \frac{L}{2Ir}(\tau_r - \tau_l) \end{bmatrix} \tag{3.9}$$

3.2.2 Mobile robot state-space model

The state space model can be derived from equation (3.9) as follows:

$$\dot{X} = f(X,U) = \begin{cases} X_4\cos(X_3) \\ X_4\sin(X_3) \\ X_5 \\ \frac{1}{mr}(U_1+U_2) \\ \frac{L}{2Ir}(U_1-U_2) \end{cases}, Y = h(X) = \begin{cases} X_3 \\ (X_1-P_{11})^2 + (X_2-P_{12})^2 \\ (X_1-P_{21})^2 + (X_2-P_{22})^2 \\ \vdots \\ (X_1-P_{n1})^2 + (X_2-P_{n2})^2 \end{cases} \tag{3.10}$$

Where:

State variables	X_1	X position (m)
	X_2	Y position (m)
	X_3	angle θ (rad)
	X_4	linear velocity (m/s)
	X_5	angular velocity $\dot{\theta}$ (rad/s)
Inputs	U_1	τ_r torque of the right-hand motor $(N.m)$
	U_2	τ_l torque of the left-hand motor $(N.m)$
Outputs	Y_1	orientation θ (rad)
	Y_2	square of the distance to the first anchor point positioned in P_1 (m^2)
	Y_3	square of the distance to the second anchor point positioned in P_2 (m^2)
	$\vdots$	$\vdots$
	Y_{n+1}	square of the distance to the n^{th} anchor point positioned in P_n (m^2)

3.2.3 Controllability and observability

The system described by the state space model (3.10) is indeed controllable. By manipulating the torque applied to the wheels, we can control the acceleration of the wheels, which in turn affects the speed. Equation (3.4) shows that the linear speed and acceleration of the robot can be controlled through the wheel torques. By using equation (3.1), Mathematical proof of this can be achieved through the utilization of the controllability matrix, which will be constructed employing Lie products [12].

To begin the analysis, we will determine the functions f, g_1, and g_2 :

$$\dot{X} = \begin{bmatrix} X_4\cos(X_3) \\ X_4\sin(X_3) \\ X_5 \\ (U_1+U_2)/mr \\ (U_1-U_2)(L/2Ir) \end{bmatrix} = \begin{bmatrix} X_4\cos(X_3) \\ X_4\sin(X_3) \\ X_5 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ a \\ b \end{bmatrix} U_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ a \\ -b \end{bmatrix} U_2 \tag{3.11}$$

$$\dot{X} = f(X) + g_1(X)U_1 + g_2(X)U_2 \tag{3.12}$$

Where :

$$f(X) = \begin{bmatrix} X_4\cos(X_3) \\ X_4\sin(X_3) \\ X_5 \\ 0 \\ 0 \end{bmatrix}, \quad g_1(X) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ a \\ b \end{bmatrix}, \quad g_2(X) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ a \\ -b \end{bmatrix}, \quad a = \frac{1}{mr}, \quad b = \frac{L}{2Ir}$$

Next, we proceed with the construction of the controllability matrix as follow :

$$C = \begin{bmatrix} g_1 & g_2 & adf_1g_1 & adf_1g_2 & adf_2g_1 & adf_2g_2 & adf_4g_1 & adf_4g_2 & adf_4g_1 & adf_4g_2 \end{bmatrix} \tag{3.13}$$

Where :

$$adf_1g_1 = \left[f(X)\ ,\ g_1(X)\right] = \frac{\partial g_1}{\partial X}f - \frac{\partial f}{\partial X}g_1 = \begin{bmatrix} -a\cos(X_3) \\ -a\sin(X_3) \\ -b \\ 0 \\ 0 \end{bmatrix} \tag{3.14}$$

$$adf_2g_1 = adf_1adf_1g_1\left[f(X)\ ,\ adf_1g_1\right] = \frac{\partial adf_1g_1}{\partial X}f - \frac{\partial f}{\partial X}adf_1g_1 \tag{3.15}$$

$$adf_2g_1 = \begin{bmatrix} (-X_4b + X_5a)\sin(X_3) \\ (X_4b - X_5a)\cos(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.16}$$

Similarly,

$$adf_3g_1 = \begin{bmatrix} X_5\left(X_5a - X_4b\right)\cos(X_3) \\ X_5\left(X_5a - X_4b\right)\sin(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.17}$$

$$adf_4g_1 = \begin{bmatrix} X_5^2\left(X_4b - X_5a\right)\sin(X_3) \\ X_5^2\left(X_5a - X_4b\right)\cos(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.18}$$

$$adf_1g_2 = \begin{bmatrix} -a\cos(X_3) \\ -a\sin(X_3) \\ b \\ 0 \\ 0 \end{bmatrix} \tag{3.19}$$

$$adf_2g_2 = \begin{bmatrix} (X_4b + X_5a)\sin(X_3) \\ (-X_4b - X_5a)\cos(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.20}$$

$$adf_3g_2 = \begin{bmatrix} X_5\left(X_4b + X_5a\right)\cos(X_3) \\ X_5\left(X_4b + X_5a\right)\sin(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.21}$$

$$adf_4g_2 = \begin{bmatrix} X_5^2\left(-X_4b - X_5a\right)\sin(X_3) \\ X_5^2\left(X_4b + X_5a\right)\cos(X_3) \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.22}$$

In order to test the controllability of the system based on the controllability criterion, we need to verify if the rank of the matrix C is equal to 5. To accomplish this, we can examine whether the determinants of the 5×5 submatrices are non-zero.

We begin by constructing the submatrix C_1, which consists of the columns from 1 to 5.

$$C_1 = \begin{bmatrix} 0 & 0 & -a\cos(X_3) & -a\cos(X_3) & (-X_4 b + X_5 a)\sin(X_3) \\ 0 & 0 & -a\sin(X_3) & -a\sin(X_3) & (X_4 b - X_5 a)\cos(X_3) \\ 0 & 0 & -b & b & 0 \\ a & a & 0 & 0 & 0 \\ b & -b & 0 & 0 & 0 \end{bmatrix} \tag{3.23}$$

Now we calculate the determinant of the submatrix C_1, denoted as $|C_1|$

$$|C_1| = \begin{vmatrix} 0 & 0 & -a\cos(X_3) & -a\cos(X_3) & (-X_4 b + X_5 a)\sin(X_3) \\ 0 & 0 & -a\sin(X_3) & -a\sin(X_3) & (X_4 b - X_5 a)\cos(X_3) \\ 0 & 0 & -b & b & 0 \\ a & a & 0 & 0 & 0 \\ b & -b & 0 & 0 & 0 \end{vmatrix} \tag{3.24}$$

$$|C_1| = 4a^2b^2(-X_4 b + X_5 a) \tag{3.25}$$

Here we have $|C_1| \neq 0$ for $bX_4 \neq aX_5$ and for X_5 and $X_4 \neq 0$.

Now, let's compute the determinant $|C_2|$ of the submatrix C_2 containing columns 1 to 4 and column 6.

$$|C_2| = \begin{vmatrix} 0 & 0 & -a\cos(X_3) & -a\cos(X_3) & (X_4 b + X_5 a)\sin(X_3) \\ 0 & 0 & -a\sin(X_3) & -a\sin(X_3) & (-X_4 b - X_5 a)\cos(X_3) \\ 0 & 0 & -b & b & 0 \\ a & a & 0 & 0 & 0 \\ b & -b & 0 & 0 & 0 \end{vmatrix} \tag{3.26}$$

$$|C_2| = 4a^2b^2(X_4 b + X_5 a) \tag{3.27}$$

Here, we observe that $|C_2| \neq 0$ when $bX_4 \neq -aX_5$ and both X_4 and X_5 are non-zero.

From equations (3.25) and (3.27), it can be deduced that the rank of the controllability matrix is equal to 5 when X_4 and X_5 are both non-zero. Thus, based on the controllability law, we can conclude that the system is barely controllable.

Regarding observability, to determine the exact position of the robot, a minimum of three anchor points [1] is necessary, as showed in (3.3).

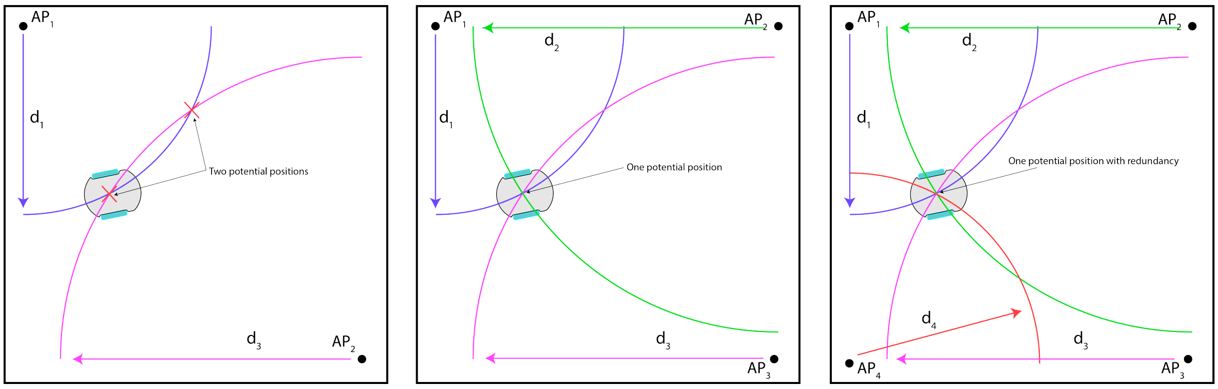

Figure 3.3: Minimum required anchor points for system observability

With three anchor points or more, and with an orientation sensor, it is possible to calculate the position and orientation of the robot. the state space will be equivalent to :

$$\dot{X} = \begin{cases} X_4\cos(X_3) \\ X_4\sin(X_3) \\ X_5 \\ (U_1+U_2)/mr \\ (U_1-U_2)(L/2Ir) \end{cases}, Y = \begin{cases} X_1 \\ X_2 \\ X_3 \end{cases} \tag{3.28}$$

However, for greater precision and accuracy, having four anchor points is ideal.

[1]refers to a fixed or stationary reference point used as the starting or zero distance for measuring distances to other points or objects

3.2.4 Mobile robot control

The controller in the system takes the current state X and the reference state X_r as inputs and generates the desired linear velocity V_d and angular velocity $\dot{\theta}_d$ needed to reach the reference state. By subtracting the current velocity and angular velocity from the desired values, we obtain the linear velocity error eV and angular velocity error $e\dot{\theta}$. These errors are then used to calculate the speed errors eV_r and eV_l for the right and left wheels.

Following the steps outlined in [26], we first calculate the error vector $e = [e_x, e_y, e_\theta]$ using the transformation matrix:

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{1r} - X_1 \\ X_{2r} - X_2 \\ X_{3r} - X_3 \end{bmatrix} \tag{3.29}$$

Next, we calculate eV and $e\dot{\theta}$:

$$\begin{cases} V_d = X_{4r}\cos(e_\theta) + K_x e_x \\ \dot{\theta}_d = X_{5r} + V_d(K_y e_y + K_\theta \cos(e_\theta)) \end{cases} \tag{3.30}$$

$$\begin{cases} eV = V_d - V = V_d - X_4 \\ e\dot{\theta} = \dot{\theta}_d - \dot{\theta} = \dot{\theta}_d - X_5 \end{cases} \tag{3.31}$$

Where K_x, K_y and K_θ are the gains of the controller, then we can calculate eV_r and eV_l the error in the speed of the left and right wheels such as:

$$\begin{cases} eV_r = \frac{2eV + Le\dot{\theta}_r}{2} \\ eV_l = \frac{2eV - Le\dot{\theta}_r}{2} \end{cases} \tag{3.32}$$

Using eV_r and eV_l from (3.32), we can then employ a PID controller to determine the control inputs U_1 and U_2:

$$\begin{cases} U_1 = K_p eV_r + K_d \frac{deV_r}{dt} + K_i \int eV_r \\ U_2 = K_p eV_l + K_d \frac{deV_l}{dt} + K_i \int eV_l \end{cases} \tag{3.33}$$

Where K_p, K_i and K_d represent the proportional, integral, and derivative gains of the PID controller, respectively.

3.2.5 Mobile robot pose estimation

In this section, we will explore the EKF for pose estimation of mobile robot. We will generalize the estimation to a full state estimation and try to improve it using deep learning and reinforcement learning in the next section.

3.2.5.1 EKF based pose estimation

We consider the state-space model of the mobile robot given by equation (3.10). The discrete-time controlled model is given by:

$$\begin{cases} X_k &= f(X_{k-1}, U_k) + \omega_k \\ Y_k &= h(X_k) + \upsilon_k \end{cases} \tag{3.34}$$

Where ω_k and υ_k are white noises with $\mathcal{N}(0, Q)$ and $\mathcal{N}(0, R)$ respectively.

In discrete-time version, the EKF algorithm is given by:

Algorithm 2: Extended Kalman Filter algorithm for pose estimation

```
Input: Initial state estimate $\hat{X}_0$, Initial covariance matrix $P_0$, number of time steps $N$,
       process model $f$, measurement model $h$, process noise $w_k$, measurement noise
       $v_k$, process jacobian $F_k$, measurement jacobian $H_k$, process noise covariance
       $Q_k$, measurement noise covariance $R_k$.
Output: Estimated State $\hat{X}_k$, estimation error $\hat{Y}_k$.
for $k = 1$ to $N$ do
    Time update (prediction)
    Predict the state: $\hat{X}_k^- = f(\hat{X}_{k-1}, U_{k-1})$
    Predict the covariance: $P_k^- = F_k P_{k-1} A_k^T + Q_{k-1}$
    Measurement update (correction)
    Update the estimation error: $\hat{Y}_k = Y_k - h(\hat{X}_k^-)$
    Compute the Kalman gain: $K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$
    Update estimate with measurement: $\hat{X}_k = \hat{X}_k^- + K_k \hat{Y}_k$
    Update the error covariance: $P_k = (I - K_k H_k)\, P_k^-$
end
```

The process jacobian F_k is given by:

$$F_k = \left.\frac{df}{dX}\right|_{\hat{X}_k, U_k} = \begin{bmatrix} 0 & 0 & -\hat{X}_4 \sin(\hat{X}_3) & \cos(\hat{X}_3) & 0 \\ 0 & 0 & \hat{X}_4 \cos(\hat{X}_3) & \sin(\hat{X}_3) & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.35}$$

The measurement jacobian H_k is given by:

$$H_k = \left.\frac{dh}{dX}\right|_{\hat{X}_k} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 2(\hat{X}_1 - P_{11}) & 2(\hat{X}_1 - P_{11}) & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 2(\hat{X}_1 - P_{n1}) & 2(\hat{X}_1 - P_{n1}) & 0 & 0 & 0 \end{bmatrix} \tag{3.36}$$

3.2.5.2 Simulations

To evaluate the EKF algorithm effectiveness, we performed four tests in different situations. Initially, we examined its performance under ideal conditions. Then, we introduced modeling errors with noise and offset to the sensors. We also experimented with three different sets of parameters for the covariance matrices Q and R. The results are shown in figures (3.4), (3.5), (3.6) and (3.7). To ensure maximum fairness in the test, we employed identical test parameters, including noise power and color, sensor offset, and the selection of the sensor that would provide the fairest results, the numerical variables for simulation parameters related to the mobile robot can be found in the ”Simulation Parameters for Mobile Robot” section of the Appendices.

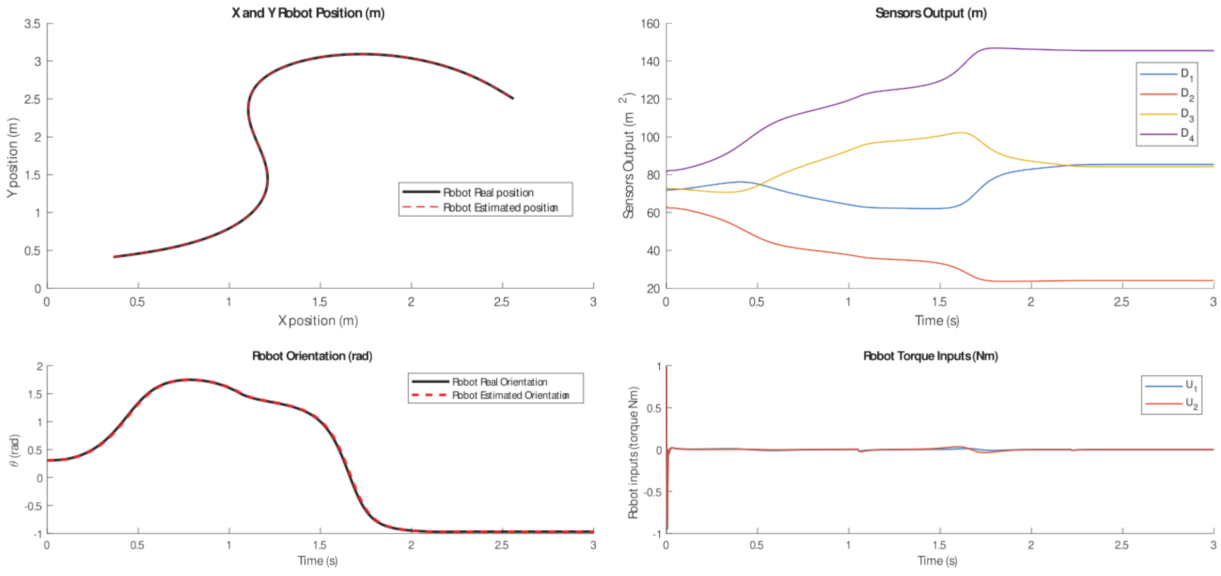

Figure 3.4: Evaluating the EKF algorithm under ideal conditions

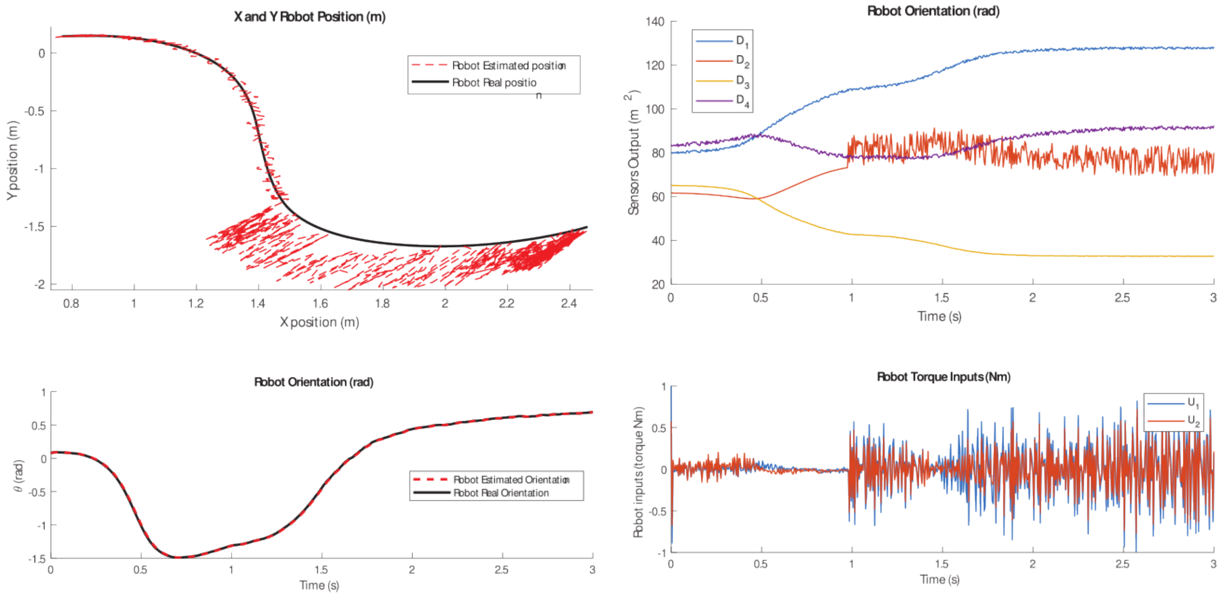

Figure 3.5: Evaluating the EKF Performance in challenging conditions with balanced covariance matrices

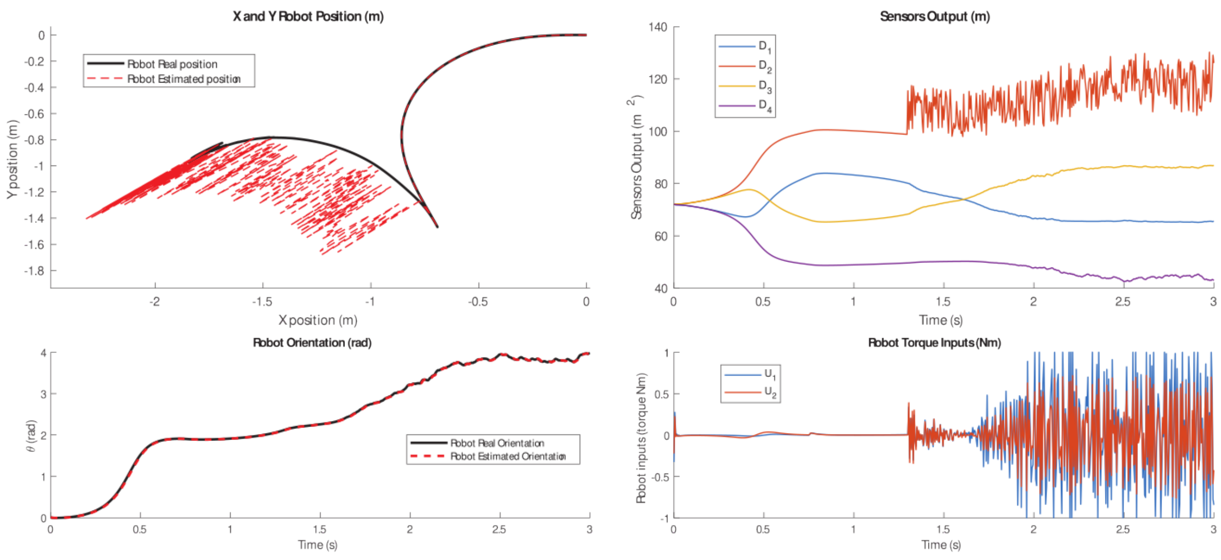

Figure 3.6: Evaluating the EKF performance in challenging conditions with higher process noise covariance than measurement noise covariance $Q > R$

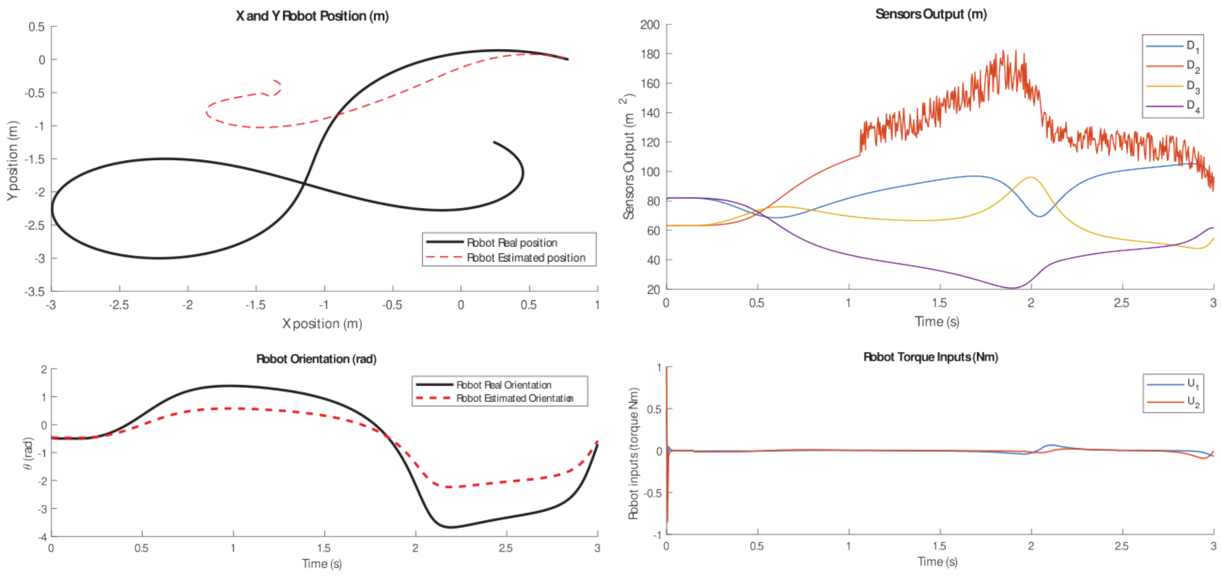

Figure 3.7: Evaluating the EKF performance in challenging conditions with lower process noise covariance than measurement noise covariance $Q < R$

3.2.5.3 Results discussion and conclusion

The test results confirm that the EKF performs well in ideal conditions, providing perfect estimations. However, in non-ideal conditions, the EKF struggles to adapt to changes. When using balanced Q and R parameters, the EKF considers both the system model and sensor measurements. However, this approach can lead to poor estimations if either the model or the sensors are inaccurate.

If we favor the use of sensors, any issues or problems with the sensors will directly impact the estimation quality. On the other hand, if we rely more on the model, the estimations will be inaccurate when the model does not accurately represent the actual system behavior.

Therefore, to effectively adapt to changes and improve estimation accuracy in real-time, it is necessary to find the right balance between utilizing sensor measurements and relying on the system model online. This requires continuous investigation and analysis to determine the optimal approach for handling these changes and ensuring robust and accurate estimations. By dynamically adjusting the weights assigned to the model and sensor measurements based on their reliability and consistency, it is possible to achieve a more adaptive and accurate estimation process. This ongoing optimization process is crucial for maintaining the EKF's performance in varying conditions and improving its ability to handle changes in the system dynamics and sensor characteristics.

3.2.6 RL based pose estimation enhancement - First approach : QR estimator

To improve the performance of the state estimation process, we will introduce optimized versions of the EKF using Deep Learning and Reinforcement Learning. We will discuss various simulations for each version and approach highlighting the strengths and limitations of each one.

In this first approach, we aim to enhance the estimator by training an RL agent to estimate the Q and R covariance matrices. To speed up the training process and validate the chosen network architecture, we first per-train the actor network using DL. This approach leverages the fact that in the simulation, we have control over the sensor noise and modeling errors, allowing us to obtain a good approximation of Q and R for training the RNN. A schematic representation of the system is illustrated in Figure (3.8).

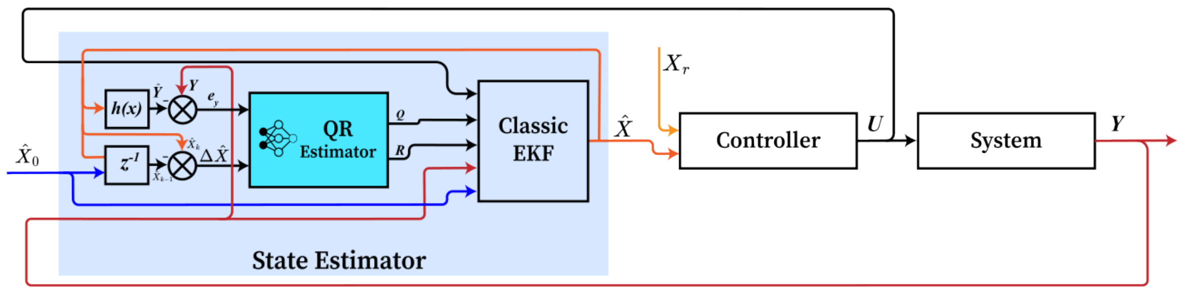

Figure 3.8: Schematic presentation of the system with the QR estimator

Where z^{-1} is a one-step delay.

3.2.6.1 Environment setup

Observation space: The observation space has a size of 10×1, which consists of the measurement error $e_Y = Y_k - \hat{Y}_k$ and the disparity between the current and previous estimated state $\Delta\hat{X} = \hat{X}_k - \hat{X}_{k-1}$.

Action space: The action space also has a size of 10×1, consisting of elements $q_{1\to5}$ and $r_{1\to5}$ of matrices Q and R, respectively. To simplify the training process, we have found that the elements of matrices Q and R follow a logarithmic scale. Therefore, we used a transformation of the form $a = -log(b)/10$, where b represents the element of one of the matrices and a is what we train the actor RNN to predict. After training is complete, we can reverse this transformation using the function$b = 10^{-10a}$ to implement the RNN in the system.

$$Q = \begin{bmatrix} q_1 & 0 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 & 0 \\ 0 & 0 & q_3 & 0 & 0 \\ 0 & 0 & 0 & q_4 & 0 \\ 0 & 0 & 0 & 0 & q_5 \end{bmatrix} \tag{3.37}$$

$$R = \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 & 0 \\ 0 & 0 & r_3 & 0 & 0 \\ 0 & 0 & 0 & r_4 & 0 \\ 0 & 0 & 0 & 0 & r_5 \end{bmatrix} \tag{3.38}$$

Reward: The reward is returned at each step and calculated as follows:

$$r = 1 - e \tag{3.39}$$

where e represents the RMSE error.

Stopping criteria: we use the "isDone" signal to determine when to end the simulation. If the error signal exceeds 10^2, the "isDone" signal is set to true indicating the end of the episode.

3.2.6.2 Pre-training the actor network with Deep Learning

Data generation: In the process of finding the best parameters for Q and R, there are typically two approaches. The first consist of conducting a comprehensive study of the system, sensors, and environment to determine the optimal parameters. The second is by trying out various parameters based on our understanding of the system and selecting the ones that gives the best results. However, in our case, as we have control over various factors such as noise characteristics, modeling errors, and more, we can create a function that takes these parameters, controls the environment accordingly and outputs acceptable Q and R parameters.

We generated data by simulating the system under different environmental characteristics, such as noise, starting points, ending points, and more. As a result, we created a dataset of 10,000 simulations, each with distinct environmental conditions. It includes the inputs (e_y and ΔX_p) as sequences, along with the corresponding output of acceptable Q and R parameters. This dataset is suitable for training the actor network to estimate the appropriate Q and R parameters.

Actor Network architecture: Given the nature of the data, it is evident that an Recurrent Neural Network is required as the actor network. It was very challenging to find an architecture that strikes a balance between simplicity for training and complexity to achieve optimal results. We tried different architectures, including the one proposed in [28]. However, we found that this architecture was computationally intensive and difficult to train, especially, considering that we would continue training with RL which is already computationally demanding.

After many tests, we find that the best architecture for our case is:

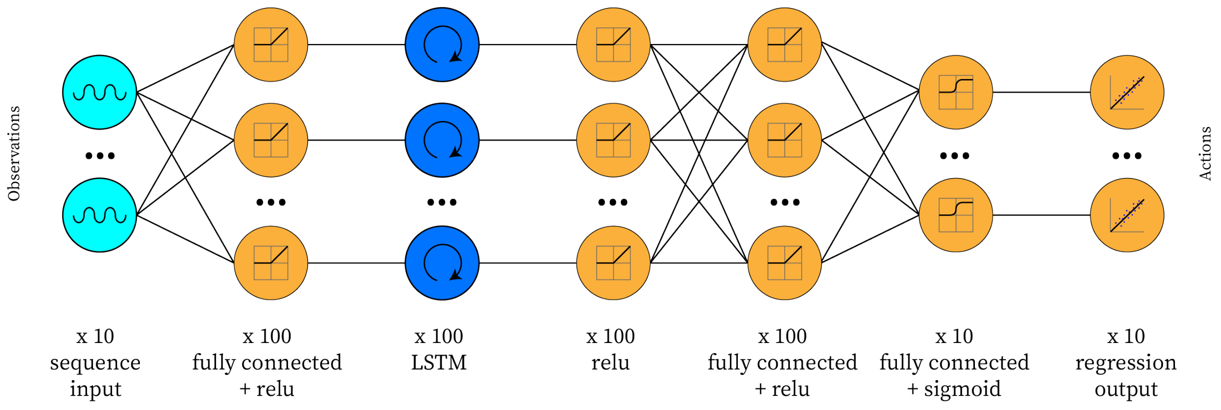

Figure 3.9: QR estimator Actor Network architecture

Actor network training: we trained the actor network with the following settings:

Optimizer	Adam	Mini-batch size	4096
Initial learning rate	0.01	Shuffle	every-epoch
Learning rate drop factor	0.5	Sequence length	longest
Max epochs	300	Validation frequency	50

Figure (3.10) represents the learning curve of actor network training.

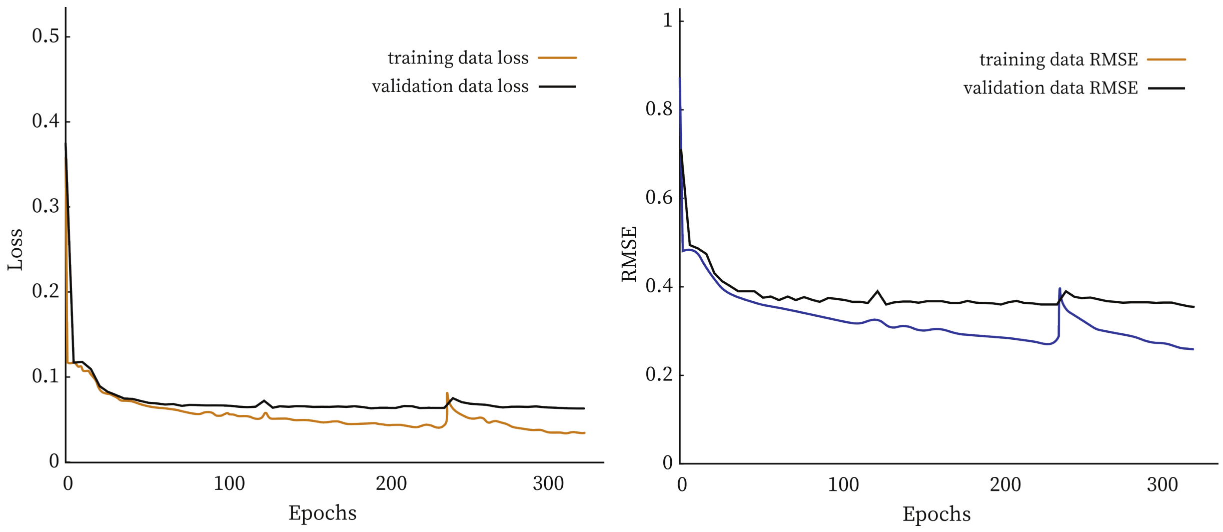

Figure 3.10: Learning curve of the actor network training (QR estimator)

Actor network training results: After training the actor network for 320 epochs, we achieved a loss of 0.05 on the training data and 0.09 on the validation data. Additionally, the root mean square error (RMSE) was measured to be 0.3 for the training data and 0.4 for the validation data. These results indicate that the network has achieved an acceptable level of accuracy in estimating the Q and R parameters.

After testing the pre-trained actor network, we get the following results:

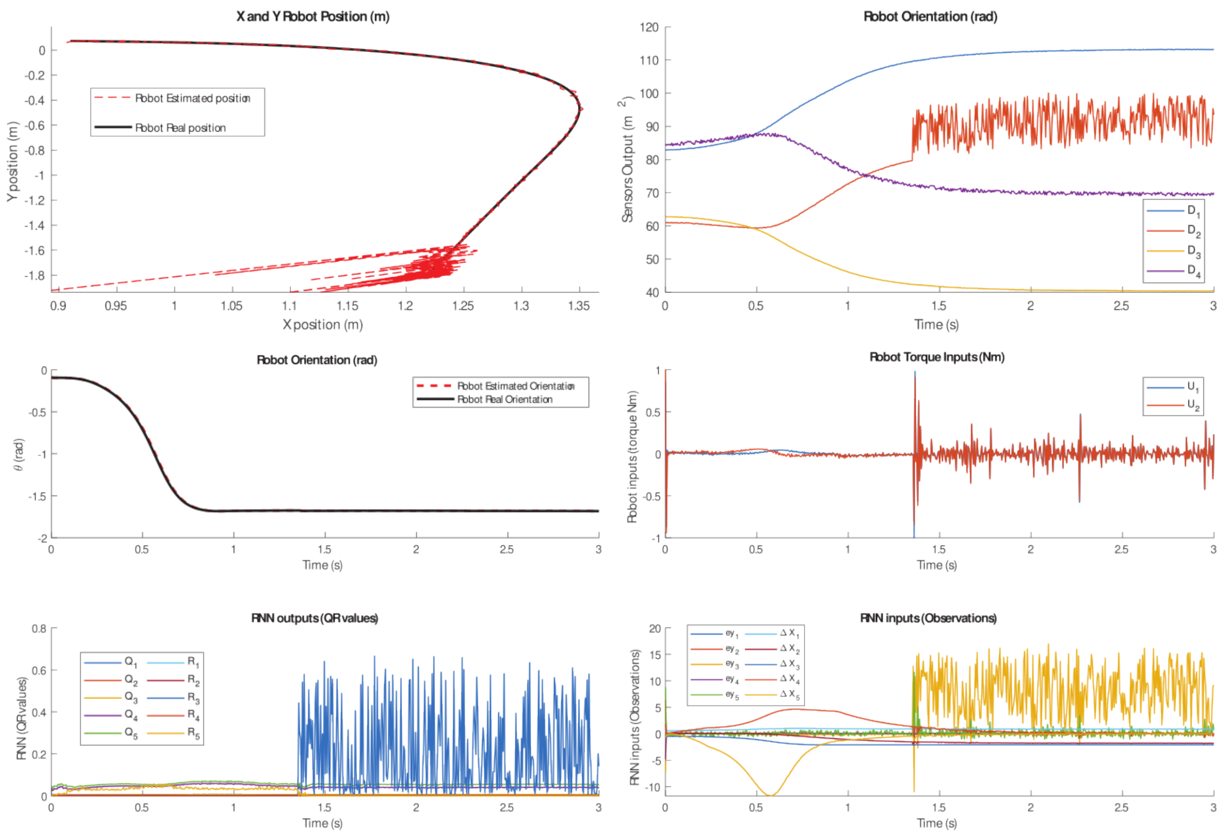

Figure 3.11: Testing the pre-trained actor network (QR estimator)

Analyzing the results:

The training of the actor network was stopped at 320 epochs to prevent overfitting, as observed by the decreasing loss and RMSE on the training data while the validation results remained constant. Despite this, the training can still be considered successful based on the achieved low loss and RMSE values.

The test results confirm the effectiveness of the trained model, as the error remains consistently below 0.3 throughout the test. Notably, when the sensors start to fail, the RNN quickly adapts to the new conditions and corrects the estimation.

These results indicate that there is potential for further improvement through the integration of RL techniques. By leveraging RL, the agent can learn to adapt even more effectively to changing conditions and enhance the estimation accuracy.

3.2.6.3 DDPG agent

Agent architecture: The DDPG agent consists of an actor network and a critic network. The actor network takes the observation as input and outputs the corresponding action. In our case, we used the pre-trained RNN as the actor network.

On the other hand, the critic network takes both the state (observation) and the action as inputs and outputs a scalar value that represents the quality or value of the State/Action pairs. For the critic network, we used the following architecture:

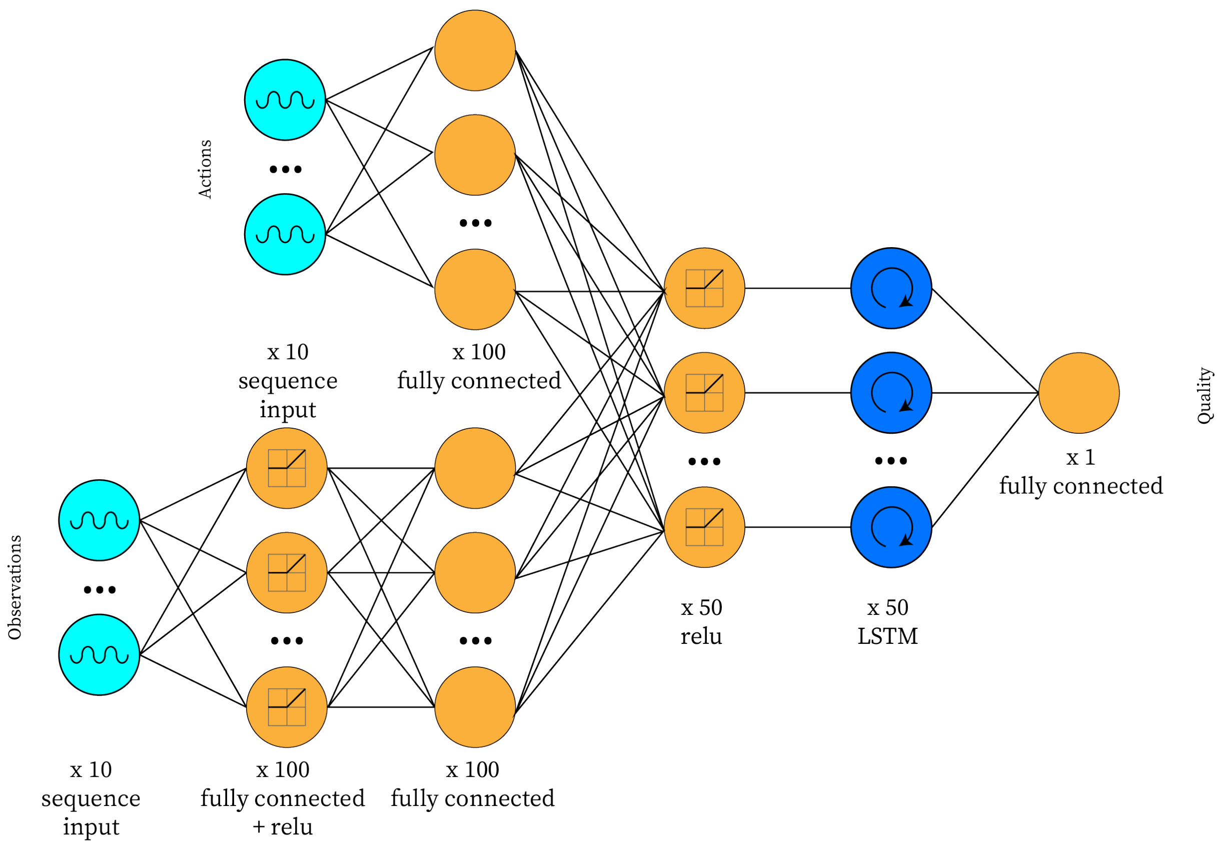

Figure 3.12: DDPG Critic Network architecture for QR estimator

DDPG agent training process: We trained the DDPG agent with the following parameters:

For critic network

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	0.1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	0.1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16

For the exploration, DDPG agents uses an exploration noise to discover new areas of the environment. Since the actor is pre-trained and we don't want to lose the performance, we set the noise variance (the exploration rate) to 0.01 with a Decay Rate of 10^{-5}.

DDPG agent training results: After 10 hours of training, the agent successfully completed 550 episodes, achieving an average reward of 355 out of 400 over the last 100 episodes.

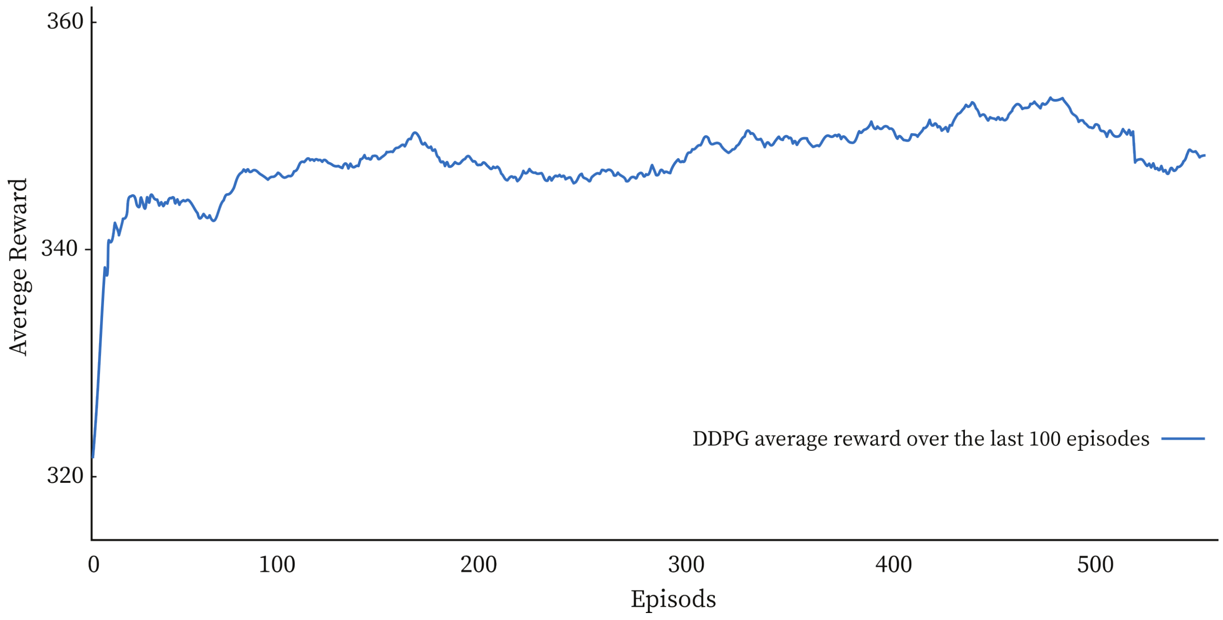

Figure 3.13: Training Progress of the DDPG agent (QR estimator)

Testing the trained DDPG agent: The implementation and utilization of the optimized policy of the DDPG agent gives the result shown in the figure (3.14) bellow:

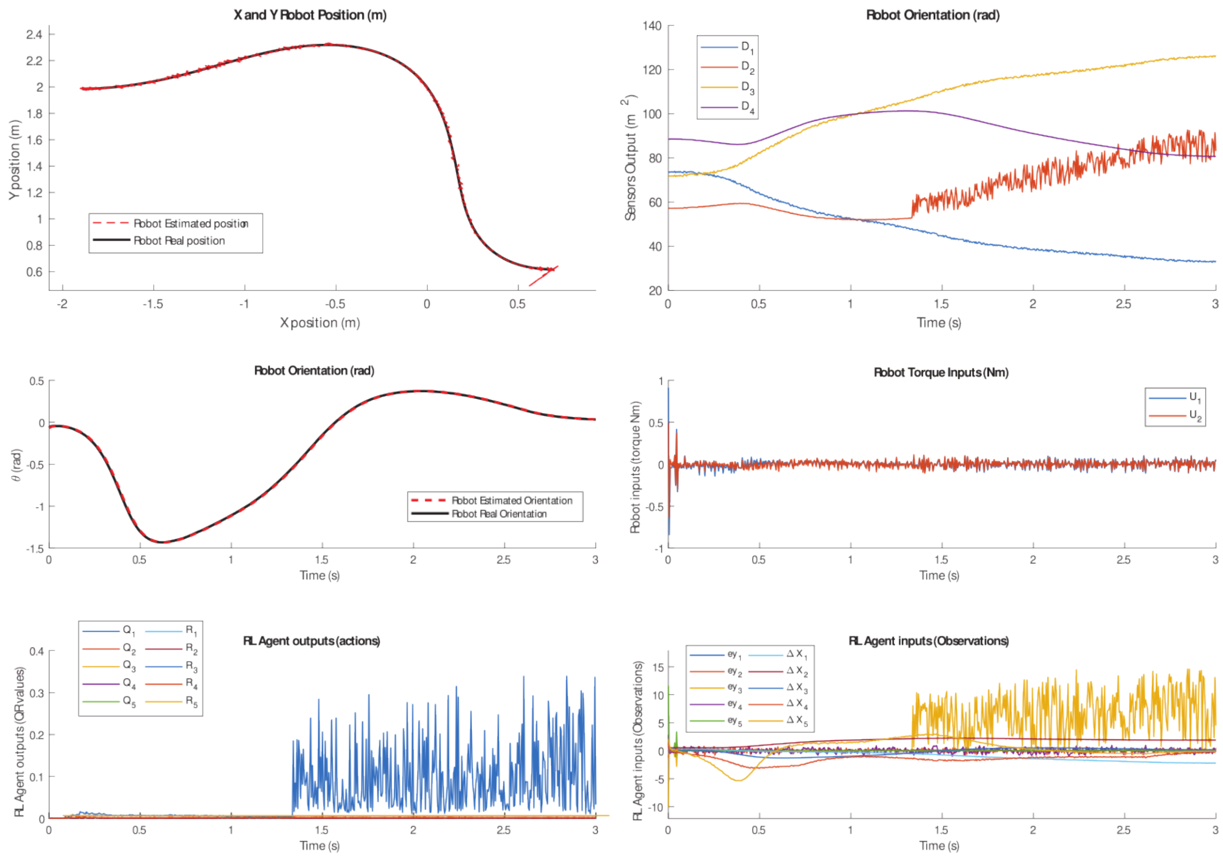

Figure 3.14: Testing the trained DDPG agent (QR estimator)

Analyzing the results:

The training progress of the DDPG agent indicates that it exhibits a slow learning nature, as evident from the fact that it took more than 10 hours to complete only 550 episodes. However, despite this slow progress, the agent showed improvement in its performance. It is worth noting that during the training, there was an error peak when the sensor failed, but the error remained below 0.1 throughout the testing phase. This suggests that with further training, the agent's performance could be further improved, leading to even better results.

3.2.6.4 TD3 agent

Agent architecture: The TD3 agent is similar to the DDPG agent. The main difference is that it incorporates two critic networks instead of just one. The actor network remains the same as in DDPG and the critic networks also use the same architecture as in DDPG. Ideally, it is recommended to use two different architectures for the critic networks in TD3. However, due to computational limitations, the same architecture was used for both critic networks in this case. Despite this limitation, the TD3 agent can still provide improvements over the DDPG agent in terms of stability and performance.

TD3 agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	32
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

TD3 agent training results: After 8 hours of training, the agent successfully completed 1350 episodes, achieving an average reward of 357 out of 400 over the last 100 episodes.

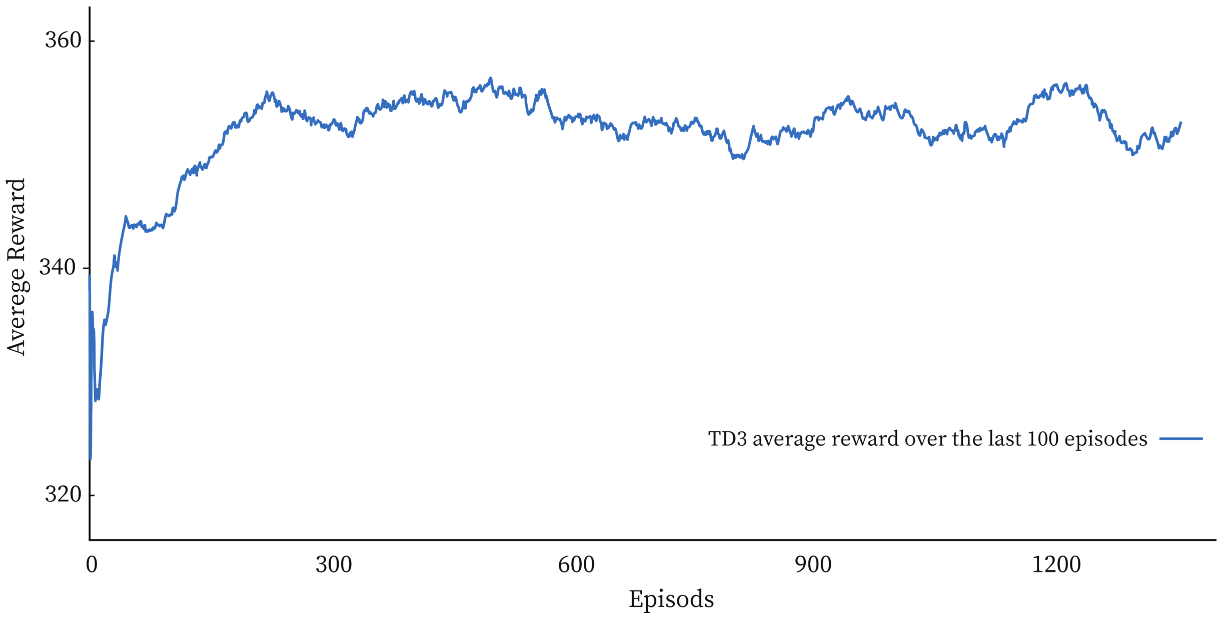

Figure 3.15: training progress of the TD3 Agent (QR estimator)

Testing the trained TD3 agent: The implementation and utilization of the optimized policy of the TD3 agent gives the result shown in the figure (3.16) bellow:

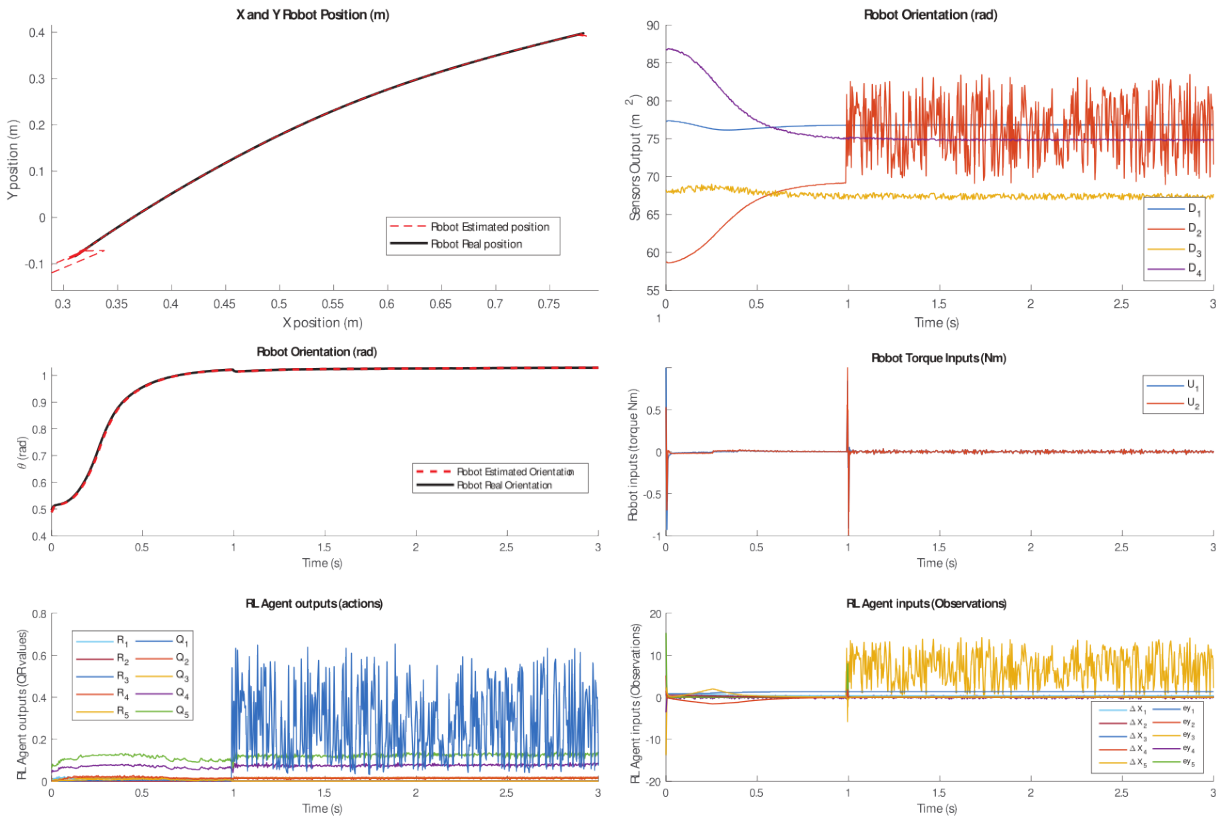

Figure 3.16: Testing the trained TD3 agent (QR estimator)

Analyzing the results:

The results obtained from the TD3 agent demonstrate its superior performance compared to the previous DDPG agent. With less than 8 hours of training, the TD3 agent completed a remarkable 1350 episodes, allowing it to encounter and adapt to various environmental conditions more efficiently and rapidly.

During testing, the TD3 agent exhibited exceptional error control, with the error signal remaining below 0.07 throughout the test, even when the sensor failed. This showcases the agent's remarkable ability to adapt quickly to changing conditions and maintain accurate estimations.

3.2.6.5 SAC agent

Agent architecture: The SAC agent architecture uses the same critic architecture as the TD3 agent, but for the actor, it uses a different architecture than the previews agents. It takes the state or observation as input and produces two outputs: the action mean and variance.

To use our pre-trained actor network, we need to modify it to include an additional output, which represents the variance. The modified architecture can be seen as follow:

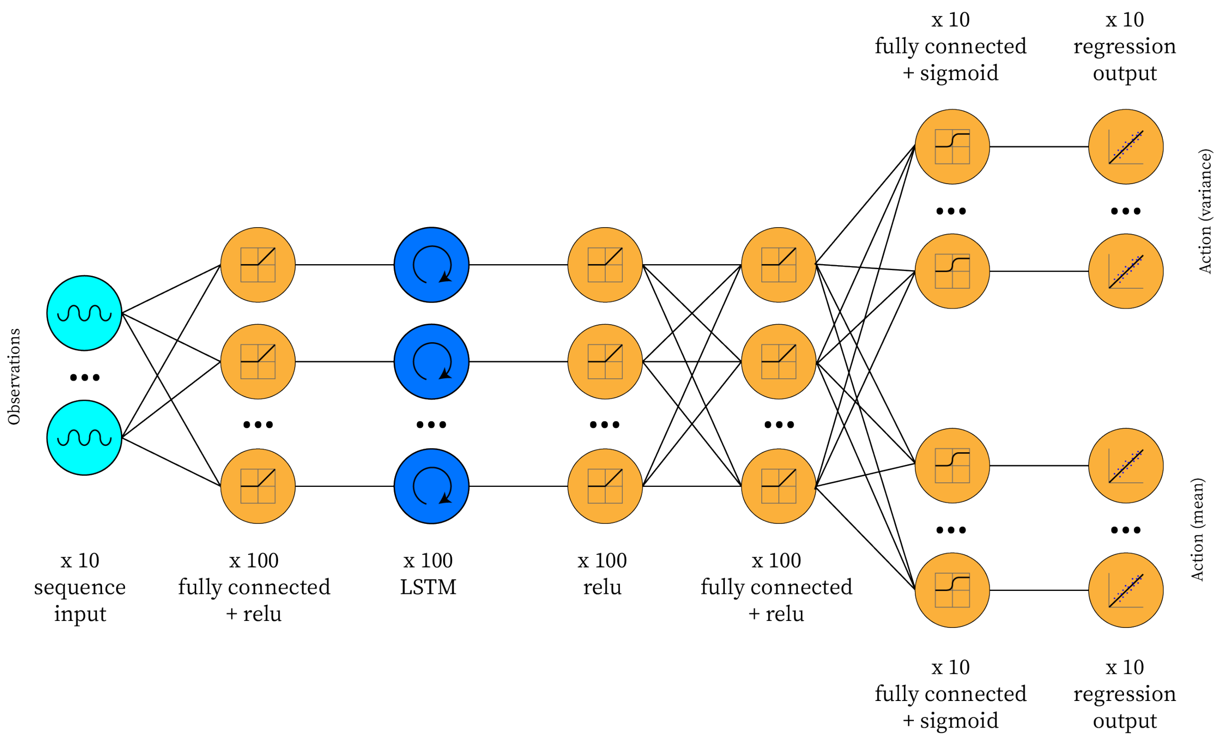

Figure 3.17: SAC Actor Network architecture for QR estimator

SAC agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-6}
Gradient Threshold	0.01

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Discount Factor	1

SAC agent training results: After 8 hours of training, the agent successfully completed 650 episodes, achieving an average reward of 355 out of 400 over the last 100 episodes.

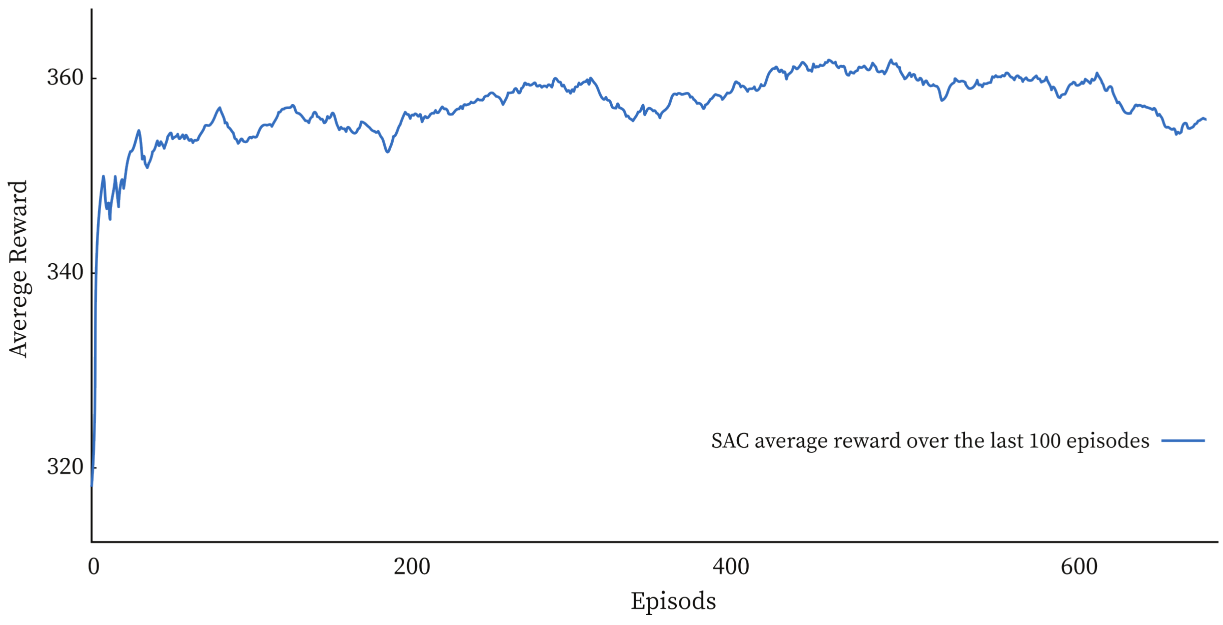

Figure 3.18: Training progress of the SAC agent (QR estimator)

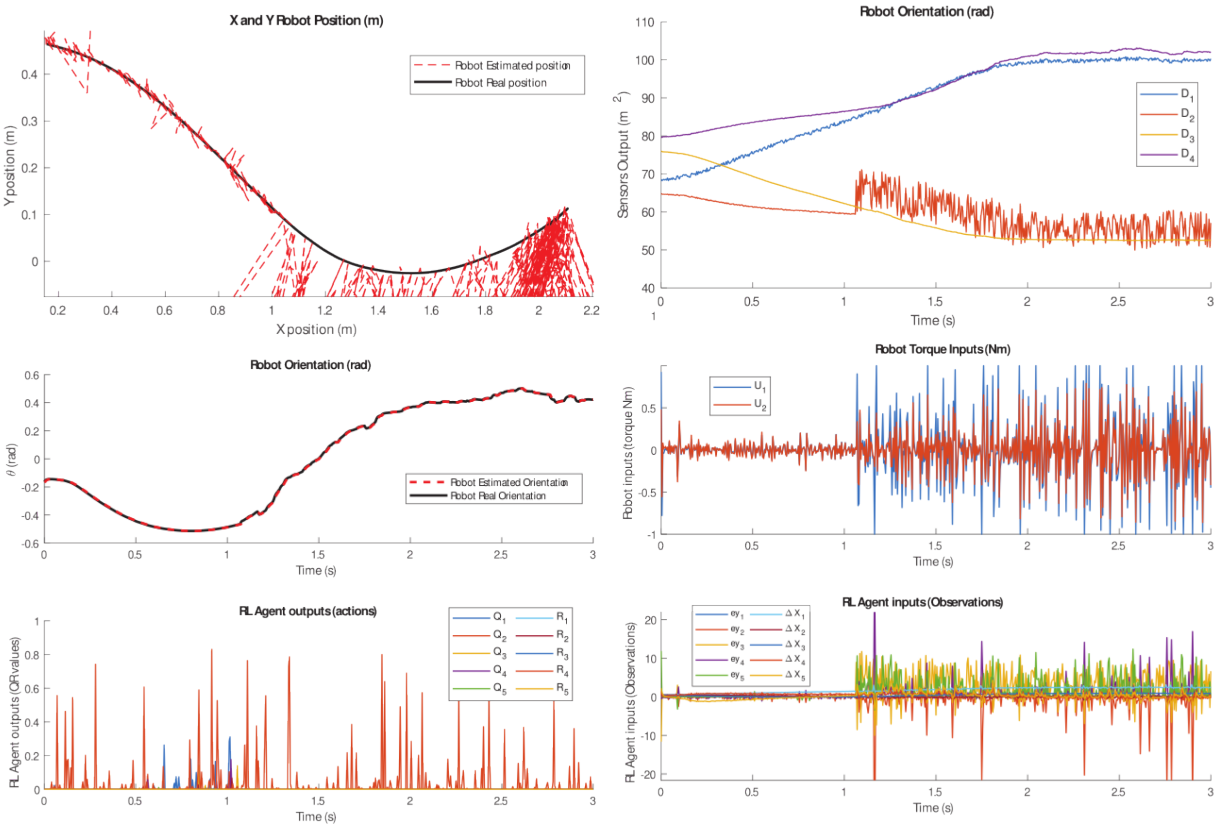

Figure 3.19: Testing the trained SAC agent (QR estimator)

Testing the trained SAC agent:

Analyzing the results: The contradictory results between the training progress and the test performance of the SAC agent can be attributed to several factors. Firstly, the SAC agent's exploration strategy, which involves exploring different actions more randomly, can lead to a more diverse training experience. This can result in a higher average reward during training but may not necessarily translate to better performance in the specific test scenario.

Additionally, the test environment may have different characteristics or challenges compared to the training environment. If the test environment is harder or presents different dynamics, it can lead to a lower performance of the SAC agent. This highlights the importance of evaluating the agent's performance in multiple test scenarios to get a comprehensive understanding of its capabilities.

3.2.6.6 PPO agent

Agent architecture: The PPO agent uses the same actor architecture as the SAC agent. This architecture is designed to output the action mean and variance based on the state or observation input. However, the critic architecture in PPO differs from SAC.

In PPO, the critic network is a value function estimator. It takes the state as input and outputs a scalar value representing the estimated value of that state. The critic architecture used in the PPO agent is shown in Figure (3.20).

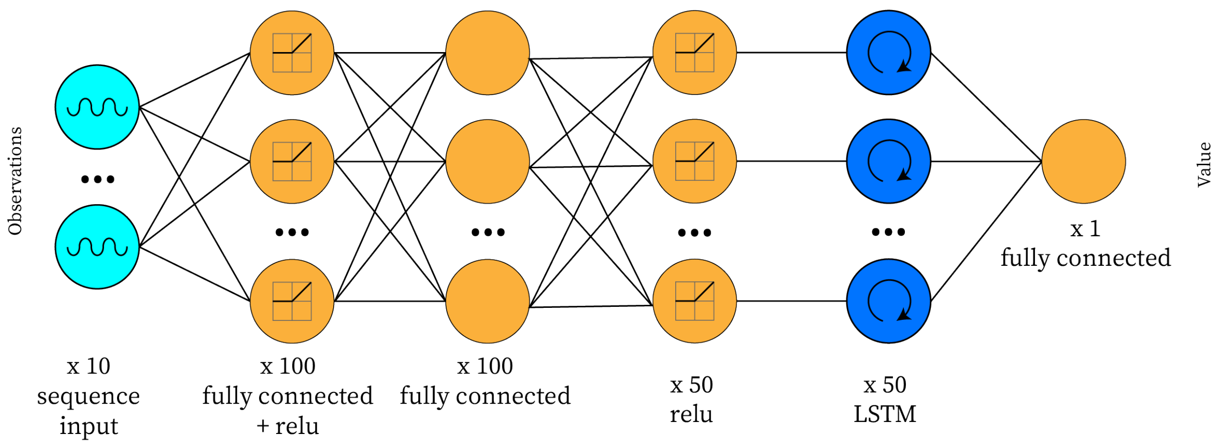

Figure 3.20: PPO Critic Network architecture for QR-Estimator

PPO agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Horizon	600
Entropy Loss Weight	0.01
NumEpoch	3
Advantage Estimate Method	gae
Discount Factor	1

PPO agent training results: After less than 7 hours of training, the agent successfully completed 10,000 episodes, achieving an average reward of 360 out of 400 over the last 100 episodes.

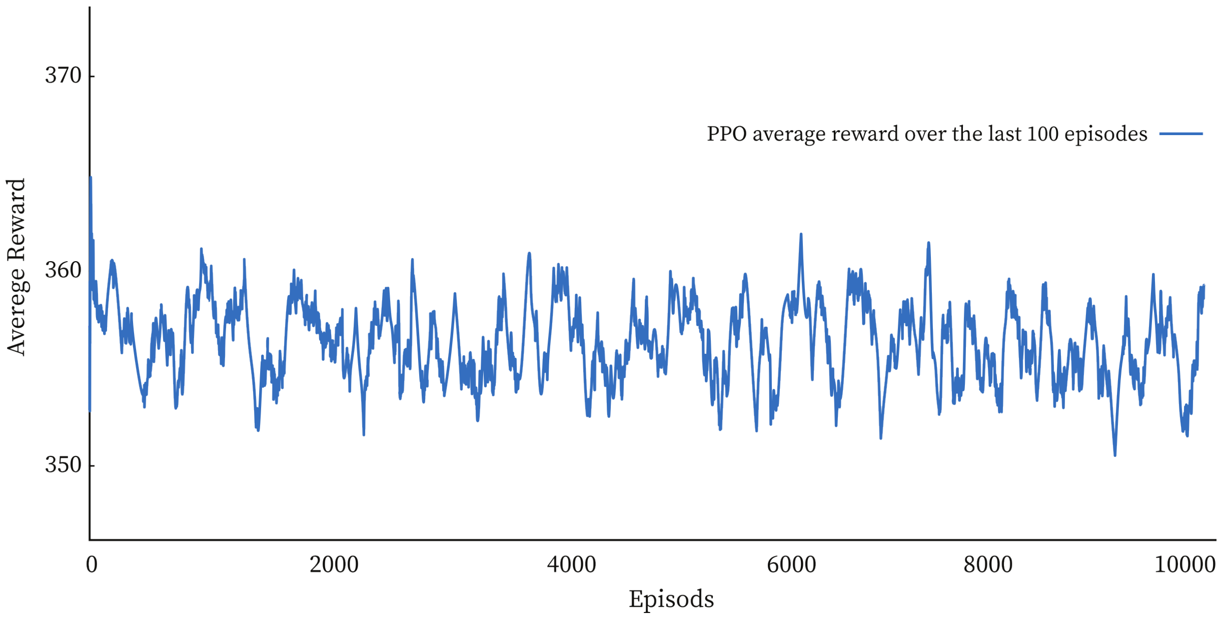

Figure 3.21: Training progress of the PPO agent (QR estimator)

Testing the trained PPO agent:

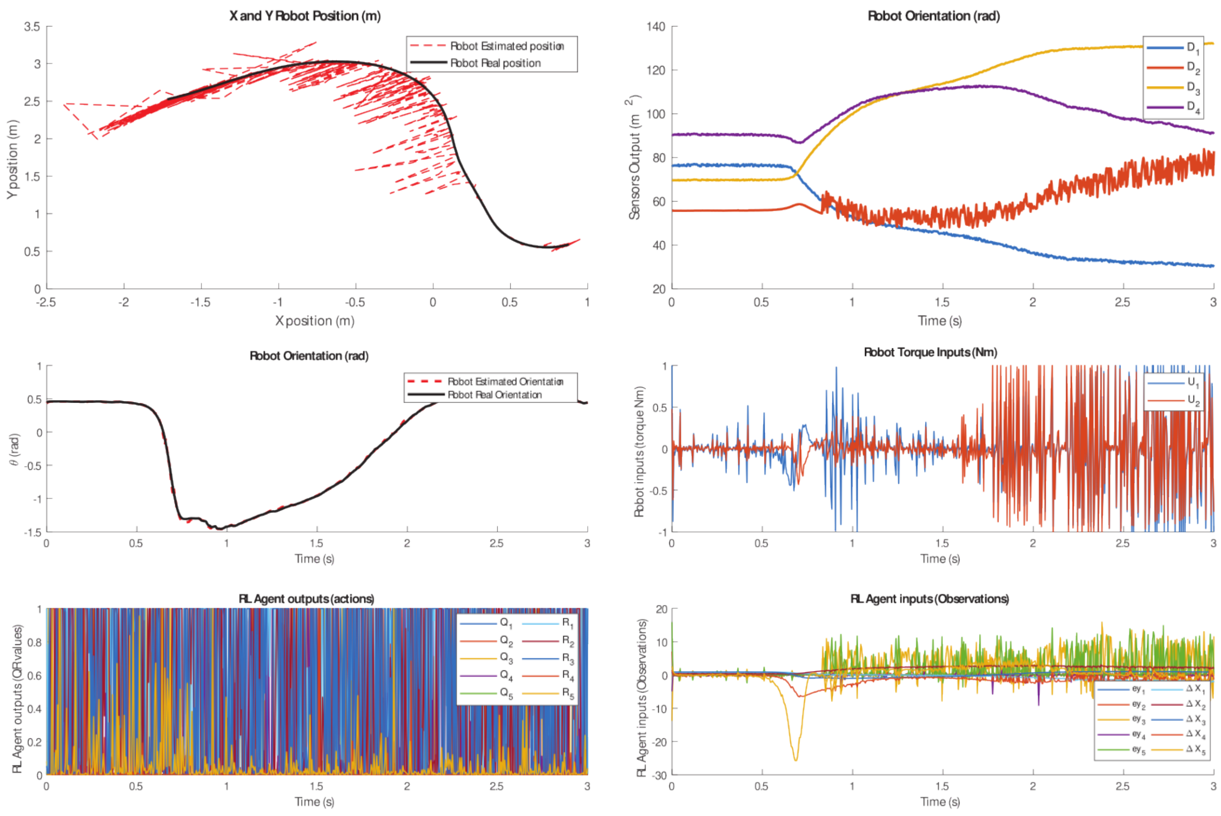

Figure 3.22: Testing the trained PPO agent (QR estimator)

Analyzing the results:

The PPO agent indeed demonstrates remarkable speed in terms of the number of episodes completed within a given training time. However, in this particular task, the training progress and performance improvement of the PPO agent appear to be limited. The average reward throughout the training remains within a relatively consistent interval, suggesting that the PPO agent may not be the most suitable choice for this specific problem. It is important to consider that different agents have their strengths and weaknesses, and selecting the most appropriate algorithm depends on the specific requirements and characteristics of the problem at hand.

3.2.6.7 Performance comparison

To compare the performance of different agents, we did 200 simulations under various conditions. Each agent was tasked with estimating the pose of the same robot in each simulation. The evaluation metric used was the integral of the square root error, computed as:

$$\int_{t_0}^{t_f} \sqrt{(X - \hat{X})^2 + (Y - \hat{Y})^2 + (\theta - \hat{\theta})^2}\, dt \tag{3.40}$$

where X, Y, θ represent the true pose of the robot, and $\hat{X}, \hat{Y}, \hat{\theta}$ represent the estimated pose.

The average sum of the square root error over the 200 simulations was calculated for each agent. The results of this comparison are presented in Figure (3.23), allowing for a comparison of the performances of the different agents.

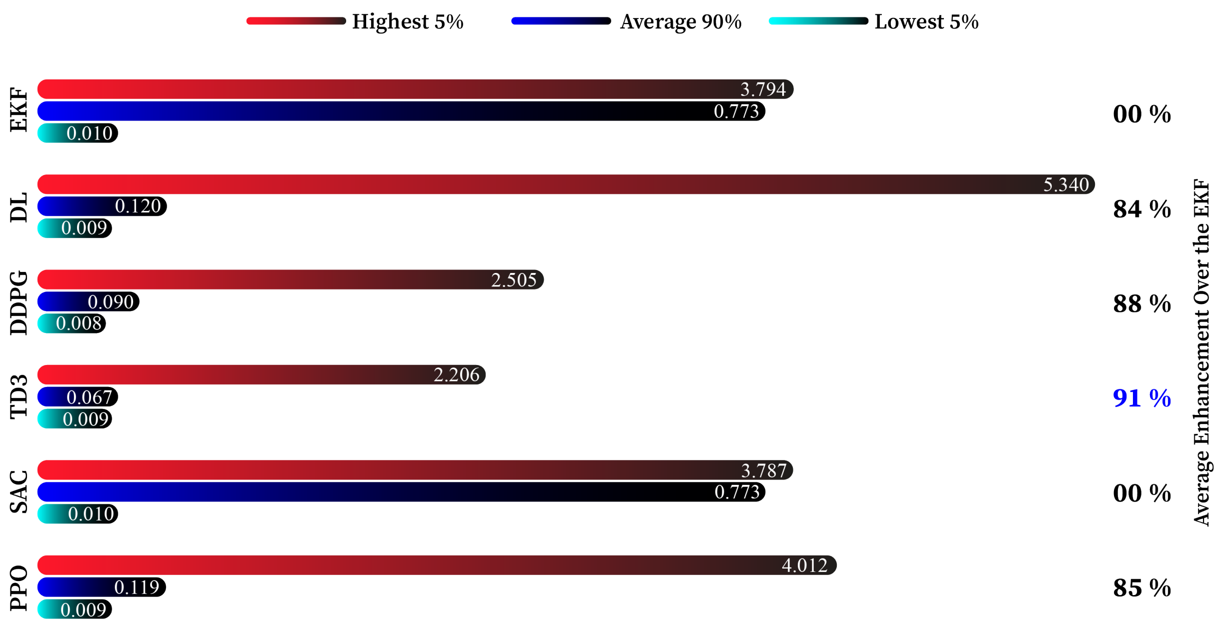

Figure 3.23: Comparison between the average error of 200 tests of the standard EKF, DL-EKF, DDPG agent, TD3 agent, SAC agent and PPO agent (QR estimator)

Analyzing the results:

The results of the test further confirm the earlier findings. The classic EKF (control) demonstrated an average error of 0.773 across the 100 tests. Implementing the trained RNN with DL (DL-EKF) resulted in an average error of 0.12, representing a significant improvement of 84% over the classic EKF.

Among the RL agents, the SAC agent performed the worst with an average error of 0.773, even worse than the DL-EKF and with no improvement over the classic EKF. The PPO agent achieved an average error of 0.119, showing a 85% improvement. While The DDPG agent performed even better with an average error of 0.09, indicating an improvement of nearly 88% compared to the classic EKF.

The best agent was the TD3 agent, which achieved an average error of 0.067, representing an impressive improvement of over 91% compared to the classic EKF. These results highlight the superior performance of RL-based agents, particularly the TD3 and the DDPG agent, in terms of estimation accuracy and adaptability in non-ideal conditions.

When examining the lowest 5% and highest 5% of performance, all agents showed comparable outcomes in ideal conditions (lowest 5%), with a maximum error of 0.01 observed in the SAC agent. However, when examining the highest 5% performers, there are differences in the results between the agents. In this regard, the SAC and PPO algorithms showed comparable or worse results compared to the EKF. Conversely, the DDPG and TD3 algorithms significantly improved performance by approximately 42% in the most challenging scenarios.

3.2.6.8 Discussion and conclusion

The earlier results demonstrate the potential of this approach, even with a relatively small RNN architecture. Despite being less complex than the recommended architecture, we achieved a significant improvement of nearly 91% over the classic EKF.

One of the main advantages of this approach is its high adaptability to changes in the environment, sensor reliability, or even the robot itself. It is worth noting that in all the tests, the model parameters used by the observer were significantly different from the actual system parameters, highlighting the robustness of the system.

Furthermore, the training process involved random environments where the modeling errors and sensor failures were randomized in each episode. If the training was more focused on addressing specific types of problems, the results could potentially be even better. Overall, these findings highlight the effectiveness and potential of using RL-based approaches for state estimation tasks.

3.2.7 RL based pose estimation enhancement - Second approach : KalmanNet

To enhance the state estimator's performance, we propose replacing the EKF's updating mechanism with a Recurrent Neural Network known as KalmanNet [28]. KalmanNet employs Deep Learning to train an RNN that generates the Kalman gain. This approach is particularly useful for handling complex dynamics or situations where accurate modeling is challenging. Our plan is to train KalmanNet using Deep Learning in diverse environments. Subsequently, we aim to further enhance its performance by integrating it into a Reinforcement Learning (RL) agent as an actor and continue training. A schematic representation of the system is illustrated if Figure (3.24).

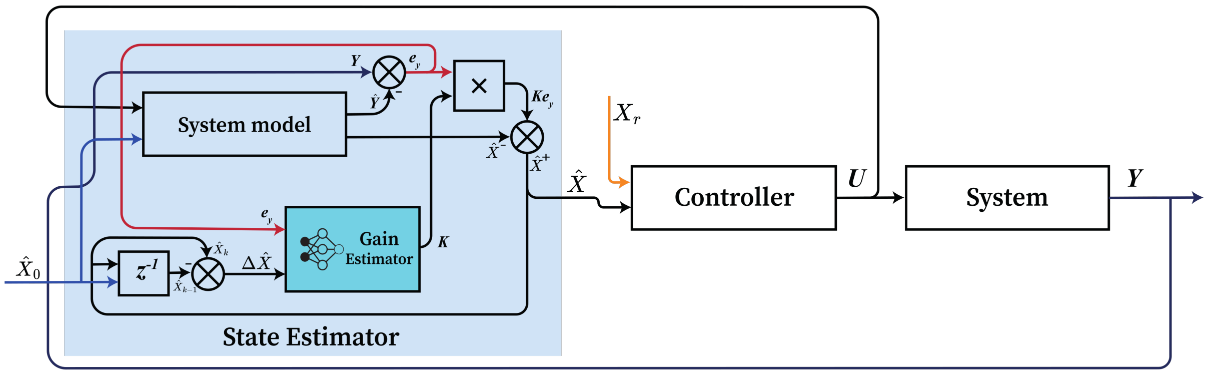

Figure 3.24: Schematic presentation of the system with KalmanNet

Where z^{-1} is a one-step delay.

3.2.7.1 Environment setup

Observation space: The size of the observation space is 10×1, consisting of the error $e_Y = Y_k - \hat{Y}_k$ and the difference between the new and previous state estimates $\Delta\hat{X} = \hat{X}_k - \hat{X}_{k-1}$.

Action space: The size of the action space is 9×1, consisting of the non-zero elements of the matrix $K(5 \times 5)$ given by:

$$K = \begin{bmatrix} 0 & K_1 & K_2 & K_3 & K_4 \\ 0 & K_5 & K_6 & K_7 & K_8 \\ K_9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.41}$$

Reward: The reward is returned at each step and calculated as follows:

$$r = 1 - \frac{e}{10} \tag{3.42}$$

where e represents the RMSE error.

Stopping criteria: we use the ”isDone” signal to determine when to end the simulation. If the error signal exceeds 10^2, the ”isDone” signal is set to true indicating the end of the episode.

Actor network: We adopted the same actor network architecture as in the first approach, which was initially recommended for this application by [28]. The only modification we made was adjusting the number of neurons in the hidden layers to 100, instead of the recommended value of $m^2 + n^2$, where m is the input size and n is the output size. The architecture of the actor network is presented in Figure (3.25).

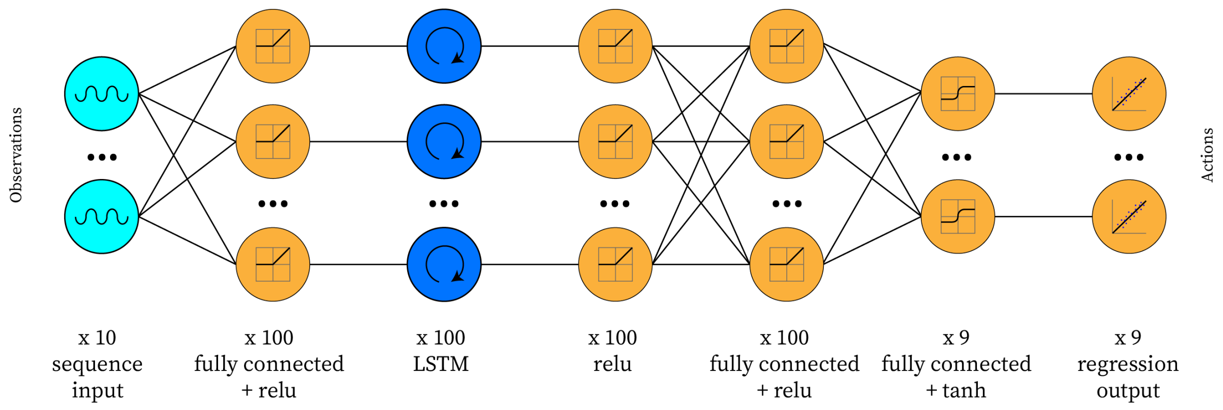

Figure 3.25: KalmanNet Actor Network architecture

Where z^{-1} is a one-step delay.

3.2.7.2 Pre-training the KalmanNet with Deep Learning

Data generation: We generated training data using the same setup as before, but we have changed the output to the non-zero elements of the Kalman gain. We simulated 10,000 runs, each lasting 3 seconds with a time step of 0.005. This resulted in a total of 6,000,000 training data points.

Actor network training: We trained the actor network with the following settings:

Optimize	Adam	Mini-batch size	4096
Initial learning rate	0.01	Shuffle	every-epoch
Learning rate drop factor	0.5	Sequence length	longest
Max epochs	300	Validation frequency	50

Training results: Figure (3.26) represents the learning curve of actor network training. After 115 minutes of training (300 epochs), the training and validation loss reached 0.07. The RMSE was 0.35 for the training data and 0.38 for the validation data.

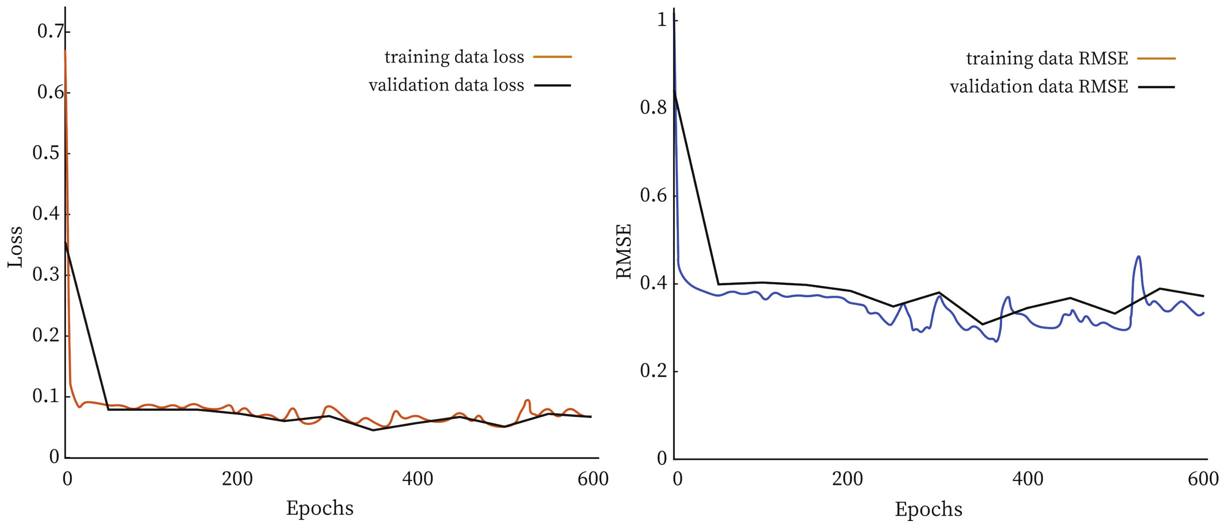

Figure 3.26: Learning curve of the KalmanNet pre-training

Testing the pre-trained KalmanNet: We implemented the trained Actor Network of KalmanNet and obtained the results shown in Figure (3.27).

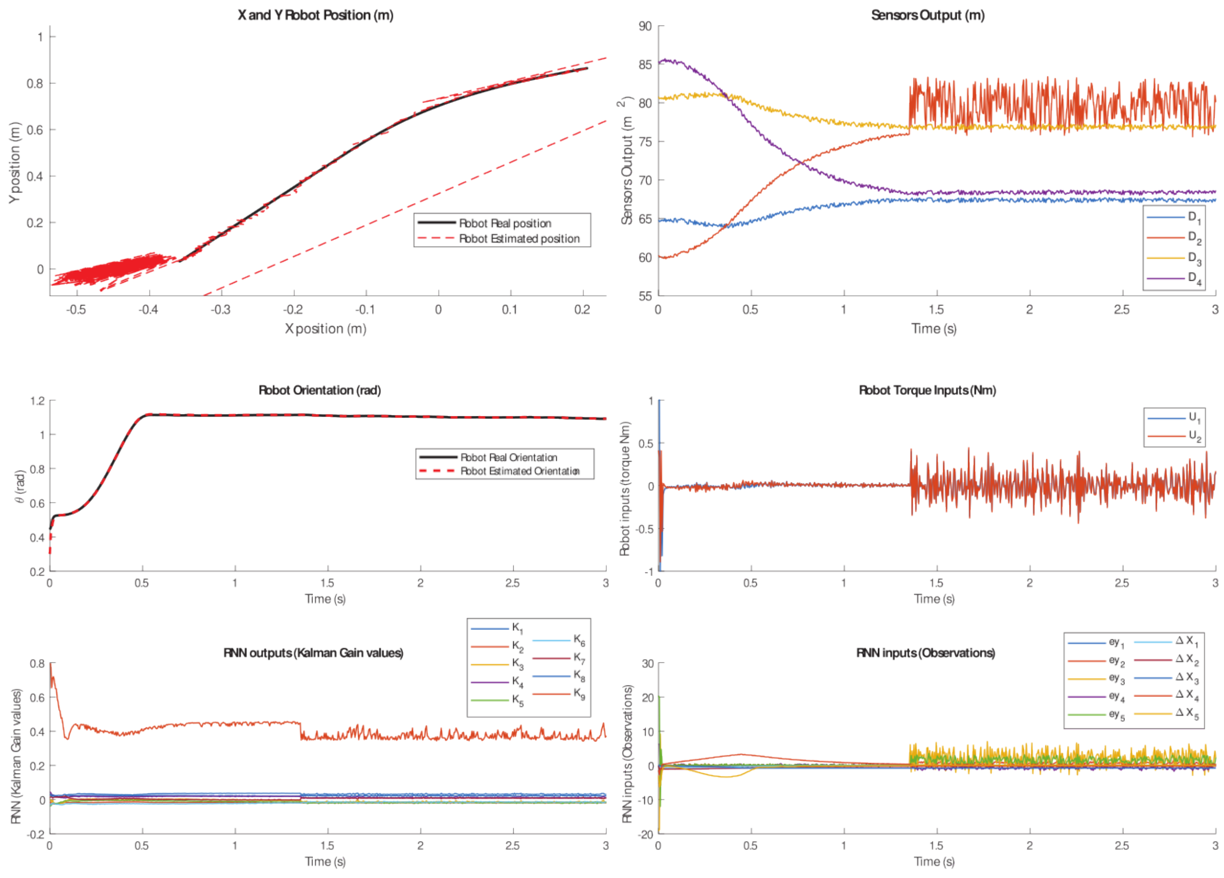

Figure 3.27: Testing the pre-trained KalmanNet

Analysis of training progress and test results:

Upon analyzing the training progress and the results of the test, it can be observed that the RNN trained with DL performs well in the initial stages of the test under normal conditions, even with high modeling errors. However, when a sensor failure occurs, the system fails to adapt and correct the estimation. This limitation could be attributed to the simplicity of the RNN architecture being used.

To address this limitation and potentially improve performance, the next step could be to implement this RNN within Reinforcement Learning agents. RL offers a framework for learning optimal actions in dynamic environments through trial and error. By integrating the RNN into an RL agent, it may be possible to enhance the system's ability to adapt and handle sensor failures more effectively. This approach can leverage the feedback and reward signals from the environment to guide the learning process and improve overall performance.

3.2.7.3 DDPG agent

Agent architecture: Similar to the first approach, we used the pre-trained actor network with the critic network presented in Figure (3.28).

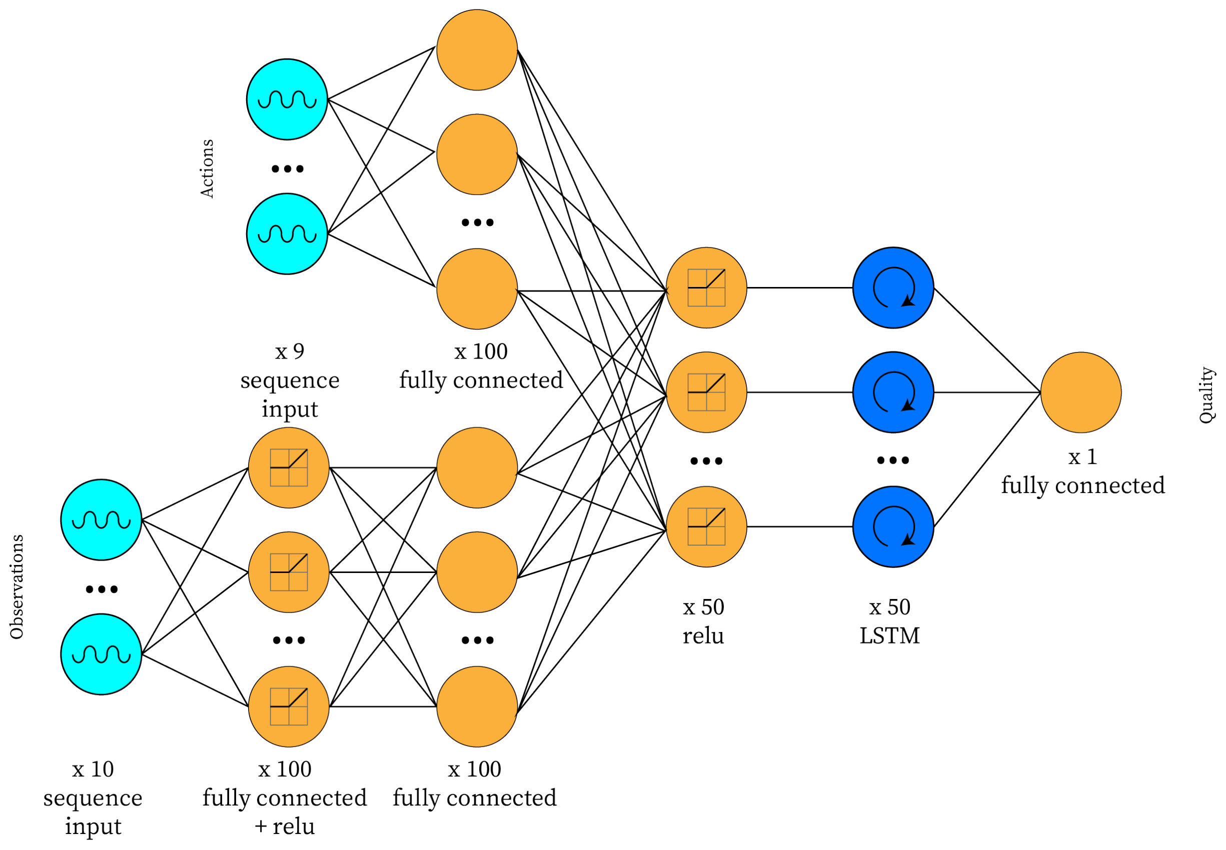

Figure 3.28: DDPG Critic Network architecture for KalmanNet

DDPG agent training process: We trained the DDPG agent with these parameters:

For critic network

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	0.1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	0.1

Agent parameters	
Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Noise Variance	0.01
Variance Decay Rate	10^{-5}

DDPG agent training results: After 100 episode of training, the agent achieved an average reward of 380 out of 400 using the step reward formula $1 - e/10$ and 200 out of 400 using the same reward formula as the first approach.

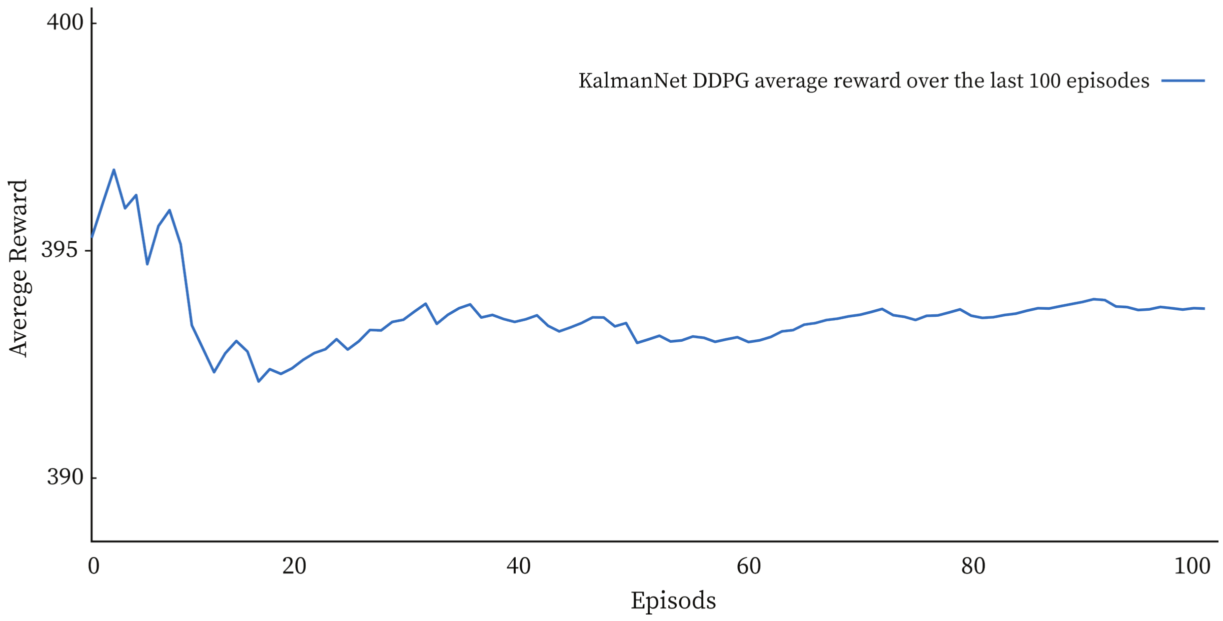

Figure 3.29: Training progress of the DDPG agent (KalmanNet)

Testing the trained DDPG agent: By implementing and using the optimized policy of the DDPG agent, we obtained the result shown in the figure (3.30) bellow:

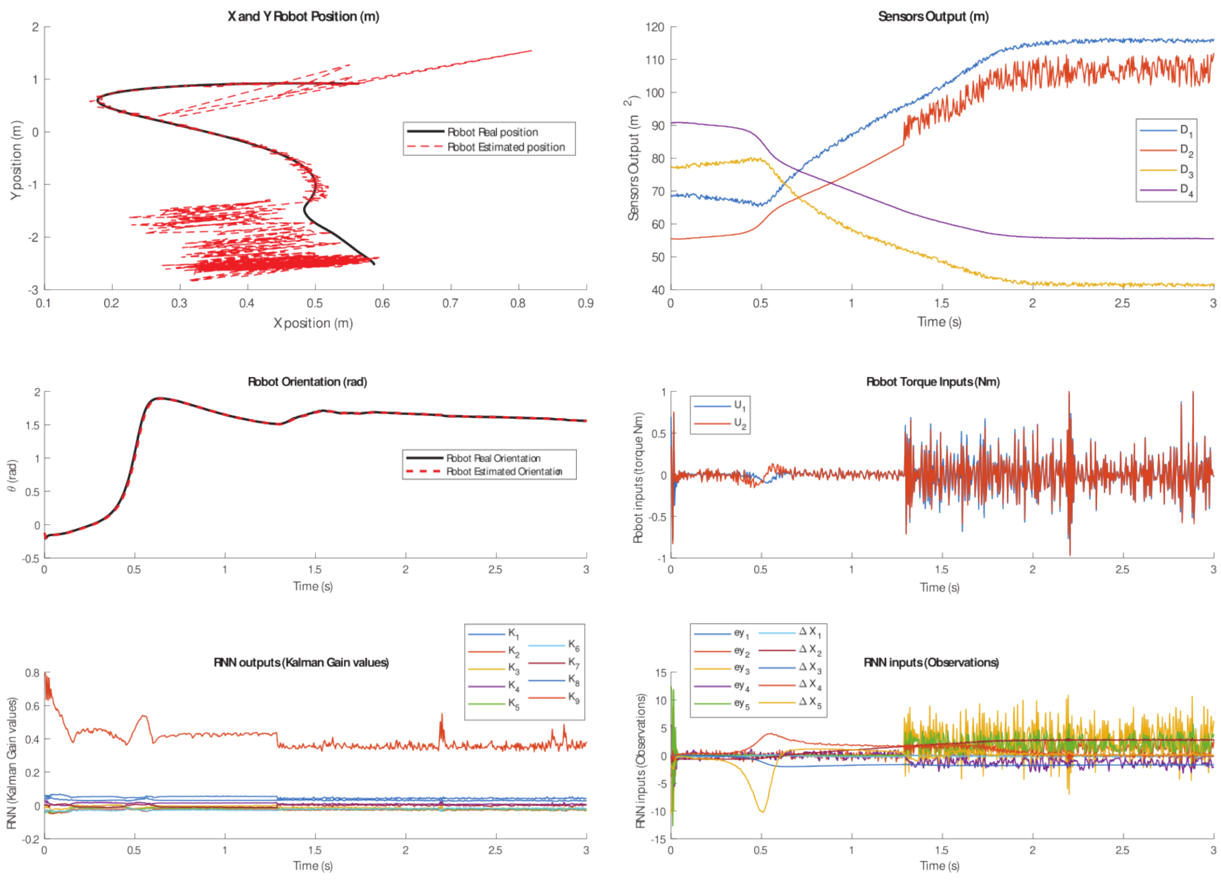

Figure 3.30: Testing the trained DDPG Agent (KalmanNet)

Analyzing the results:

Based on the training progress and the test results, it appears that the performance did not improve with the implementation of the DDPG agent. This outcome suggests that the chosen approach may not be suitable for addressing the limitations observed in the earlier analysis.

3.2.7.4 TD3 agent

Agent architecture: In this approach using the TD3 agent, we used the same actor and critic networks as in the DDPG agent, with the only difference being the choice of agent type.

TD3 agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	32
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

TD3 agent training results: After 100 episode of training, the agent achieved an average reward of 396 out of 400 (or 360/400 with the reward formula $r = 1 - e$).

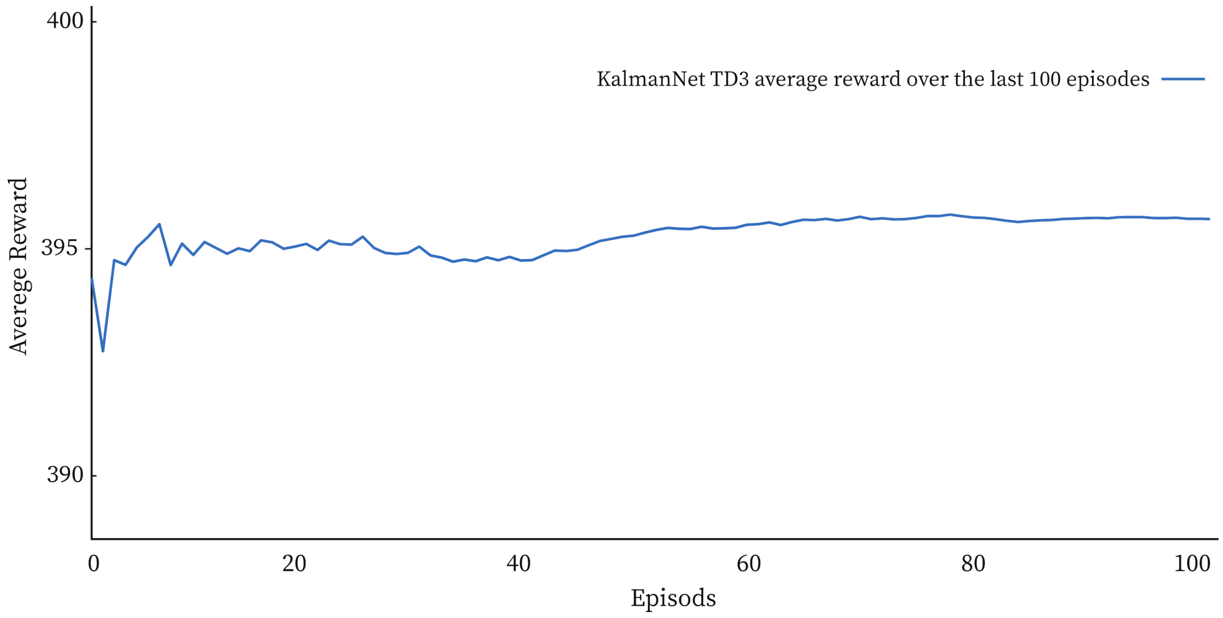

Figure 3.31: training progress of the TD3 agent (KalmanNet)

Testing the trained TD3 agent: By implementing and using the optimized policy of the TD3 agent, we obtained the result shown in the figure (3.32) bellow:

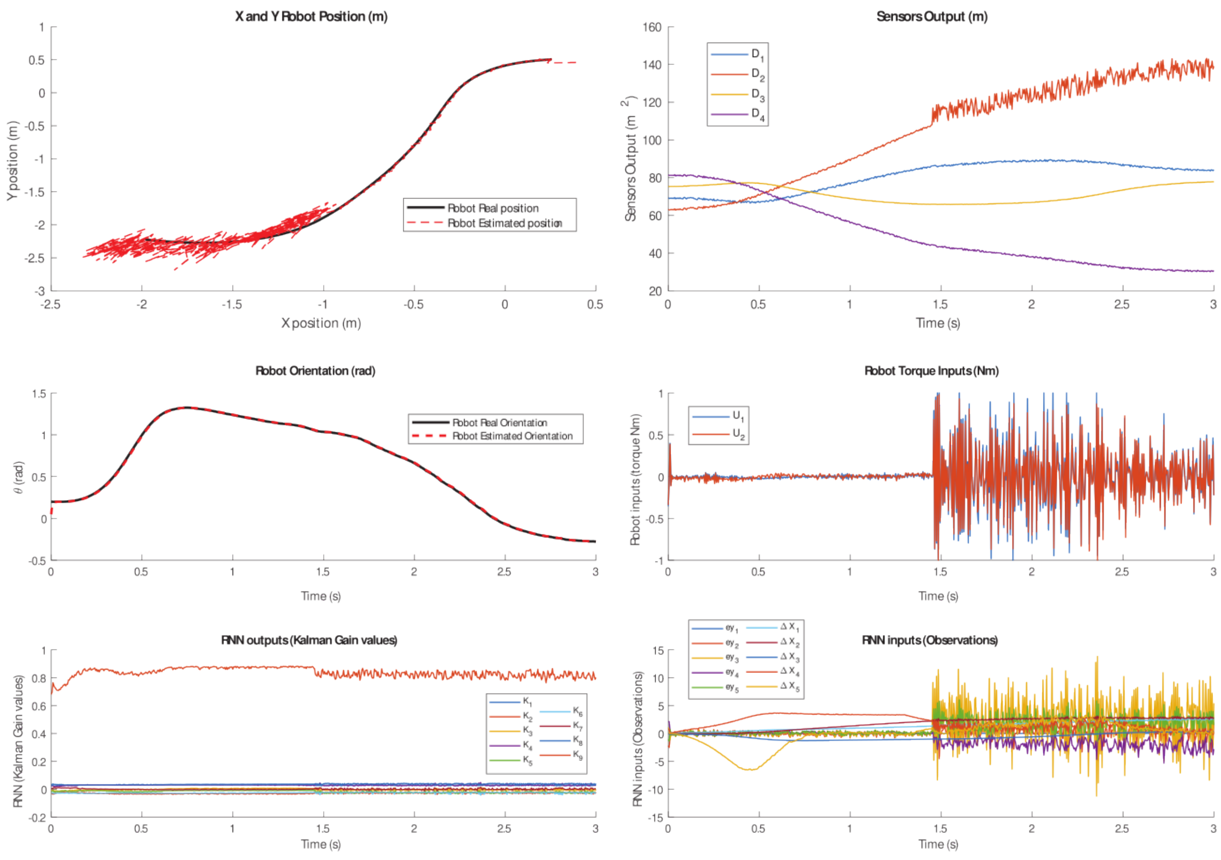

Figure 3.32: Testing the trained TD3 agent (KalmanNet)

Analyzing the results:

Based on the training progress and the test results, it appears that the performance did not change significantly with the TD3 agent compared to the previous approaches. The average reward remained the same, indicating that the agent's performance did not improve. The test results further confirm this observation.

3.2.7.5 SAC agent

Agent architecture: Similar to the first approach, in this approach, we utilized the TD3 critic with a modified version of the trained actor network. The details of the new actor network can be found in the following Figure (3.33).

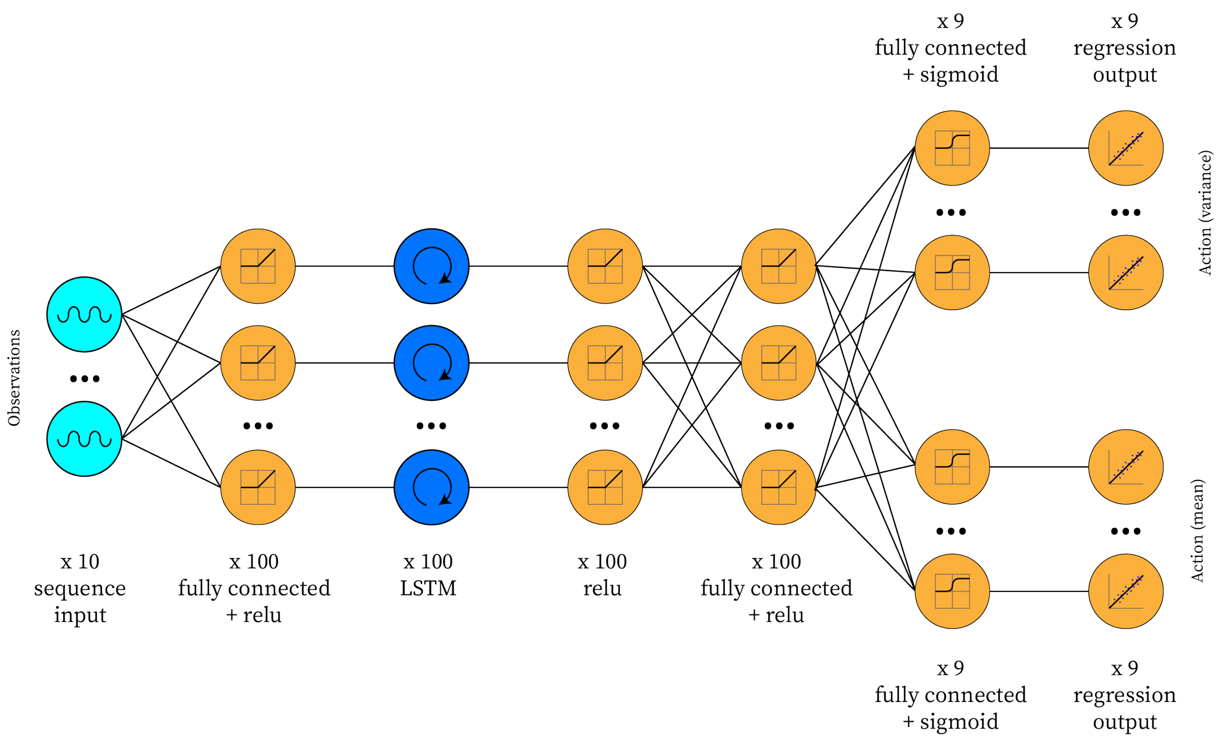

Figure 3.33: SAC Actor Network architecture for KalmanNet

SAC agent training process: We trained the SAC agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-6}
Gradient Threshold	0.01

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Discount Factor	1

SAC agent training results: After 100 episodes of training, the agent achieved an average reward of 395 out of 400 (or 350 out of 400 with $r = 1-e$).

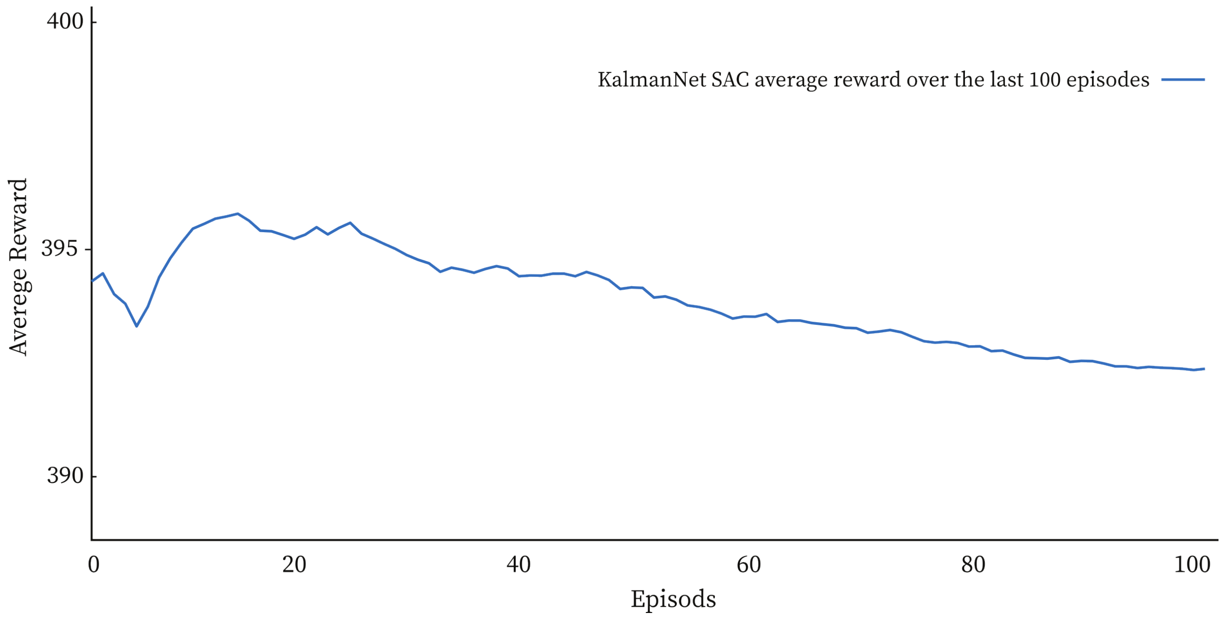

Figure 3.34: Training progress of the SAC agent (KalmanNet)

Testing the trained SAC agent:

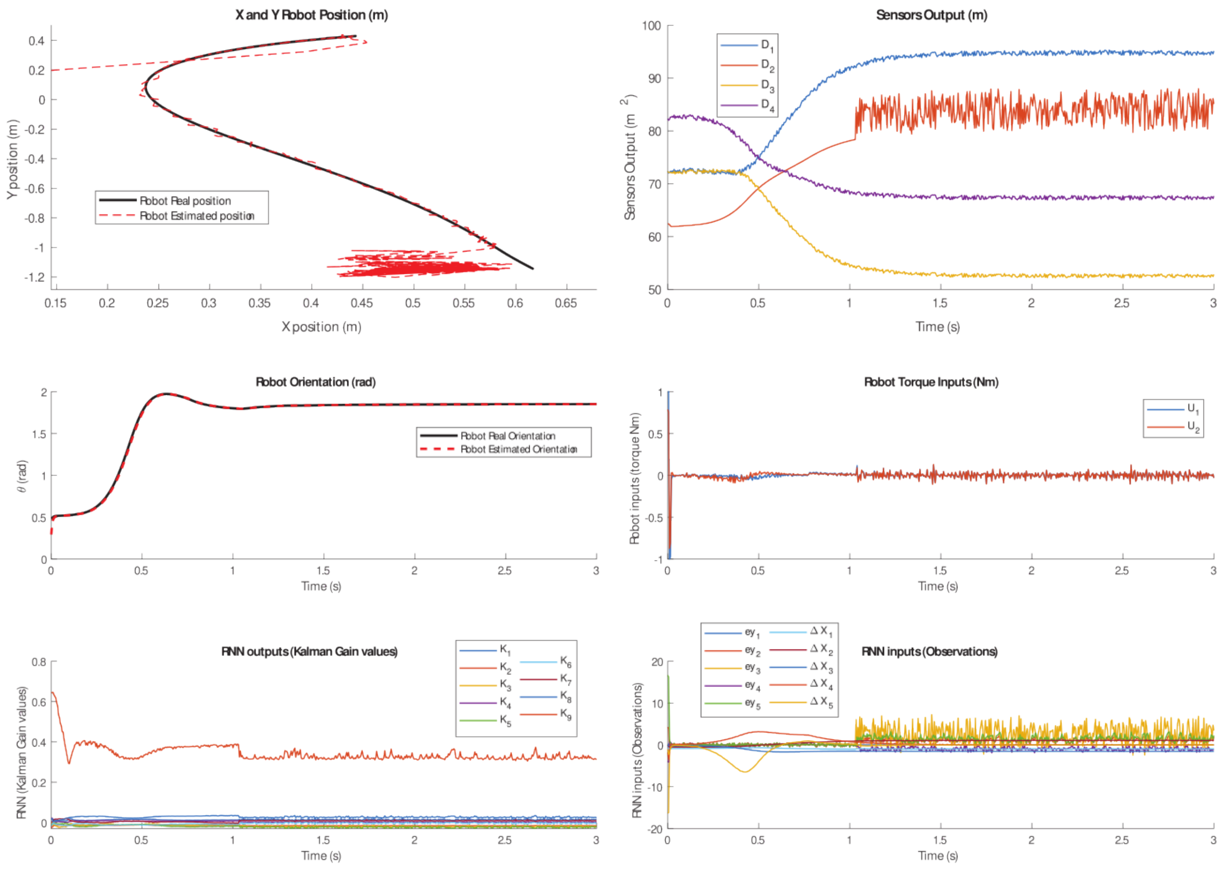

Figure 3.35: Testing the trained SAC agent (KalmanNet)

Analyzing the results:

Based on the training progress and the observed decrease in average reward, it appears that training the agent further may not lead to improved performance.

3.2.7.6 PPO agent

Agent architecture The actor network architecture used was the same as the SAC agent. For the critic network, the architecture used was the same as shown in Figure (3.36).

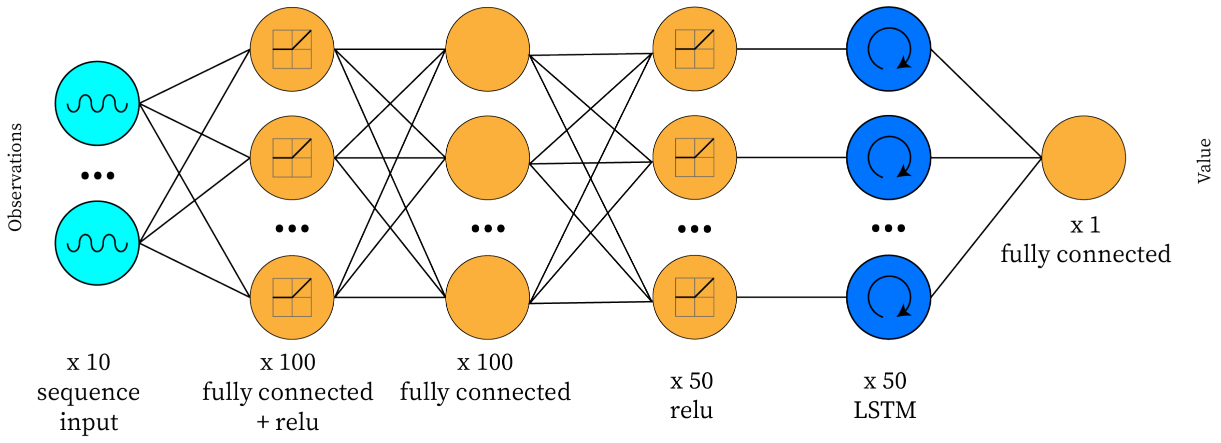

Figure 3.36: PPO Critic Network architecture for KalmanNet

PPO agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Horizon	600
Entropy Loss Weight	0.01
NumEpoch	3
Advantage Estimate Method	gae
Discount Factor	1

PPO agent training results: After 100 episode of training, the agent achieved an average reward of 396 out of 400 (or 350/400 with $r = 1-e$).

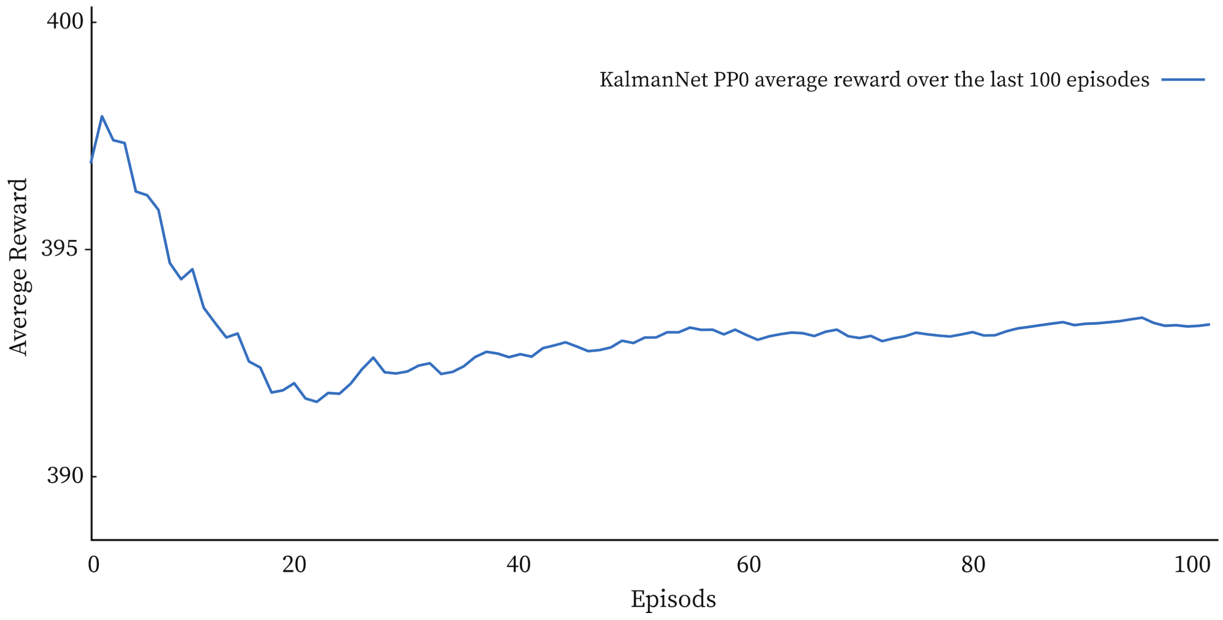

Figure 3.37: Training Progress of the PPO agent (KalmanNet)

Testing the trained PPO agent:

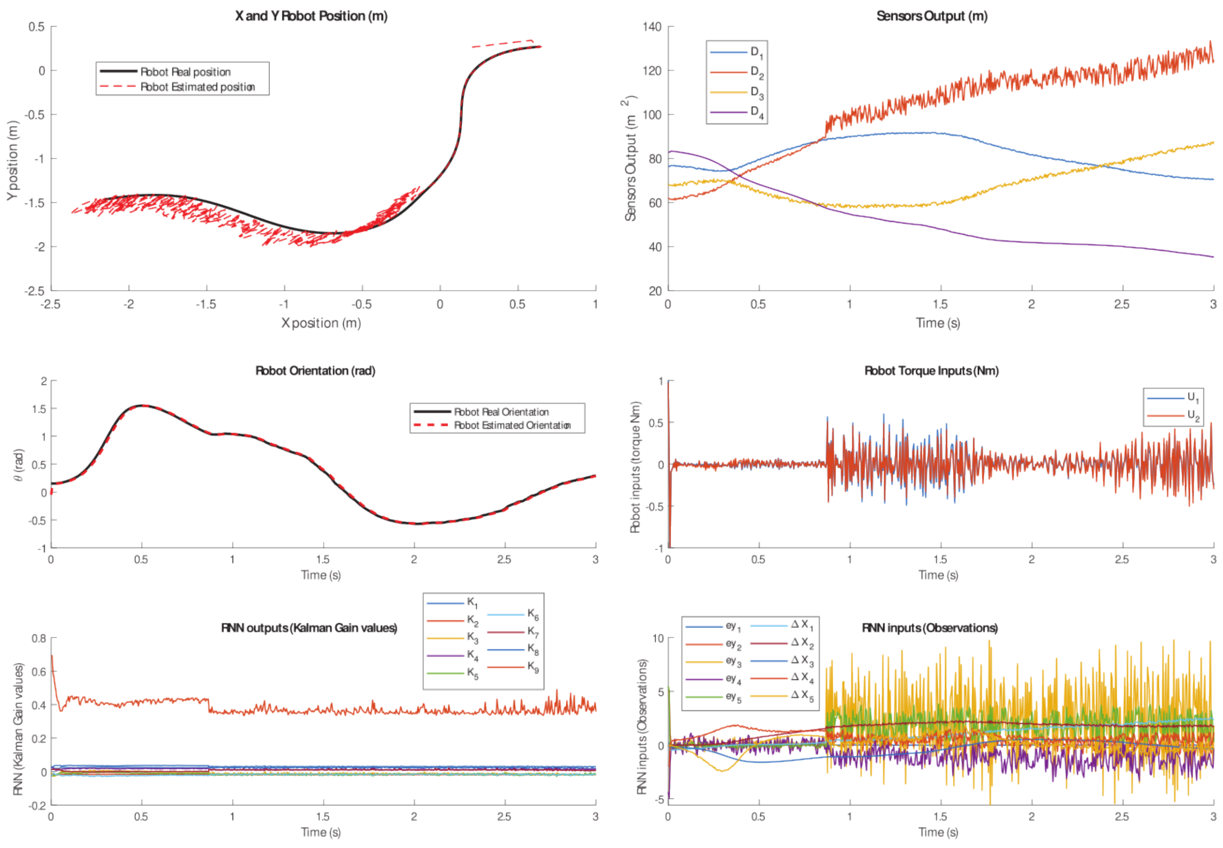

Figure 3.38: Testing the trained PPO agent (KalmanNet)

Analyzing the results:

From the findings and observations, it seems that the performance of the PPO agent, when compared to the RNN-only estimator, did not exhibit notable enhancements.

3.2.7.7 Performance comparison

Similar to the first approach, we conducted 200 simulations under different conditions and computed the integral of the square root errors, We then calculated the average across the 200 simulations. The results are presented in Figure (3.39).

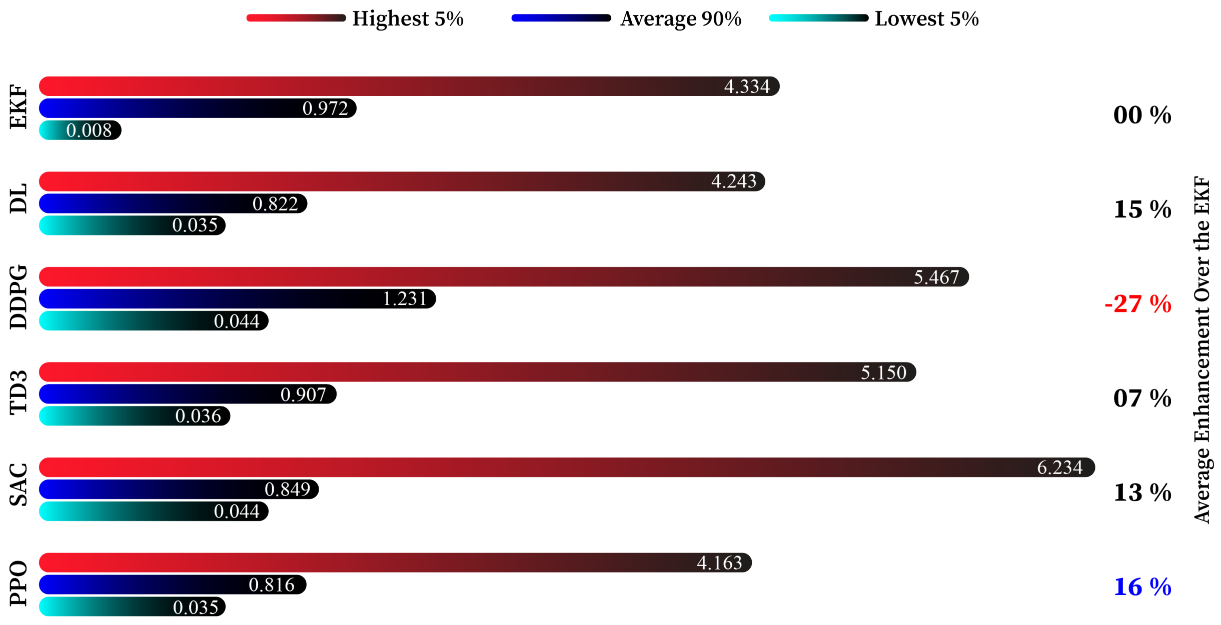

Figure 3.39: Comparison of the average error of 200 tests of the standard EKF, DL-EKF, DDPG agent, TD3 agent, SAC agent and PPO agent (KalmanNet)

Analyzing the results:

This test confirms the earlier results. The classic EKF had an average error of 0.972 in the 200 tests. Implementing the trained RNN with DL resulted in an error of 0.822, a 15% improvement over the EKF. The RL agents' performances are as follows: DDPG had an error of 1.231, 27% worse than the EKF. SAC had an error of 0.849, a 13% improvement. TD3 had an error of 0.907, nearly 7% improvement. PPO had the best performance with an error of 0.816, a 16% improvement over the EKF.

3.2.7.8 Results discussion and conclusion

These results, although not as good as the first approach, still demonstrate an improvement over the standard EKF. It is worth noting that the RL agents did not enhance the performance, in fact, they made it worse. This can be attributed to the fact that even with a pre-trained actor, the agent still needs to explore and gather experience to fill its buffer. Since our actor likely had limited room for improvement, any changes led to poorer results. This can be observed in the training progress of the DDPG agent shown in Figure (3.40). Despite training it for over 20 hours and 2200 epochs, the performance deteriorated. As a result, we terminated the training of all agents after only 100 episodes.

To improve the results of this approach, one potential solution is to employ a more complex

RNN architecture that can encode the data analysis process for selecting the best data to work with and incorporate the mechanisms of the EKF.

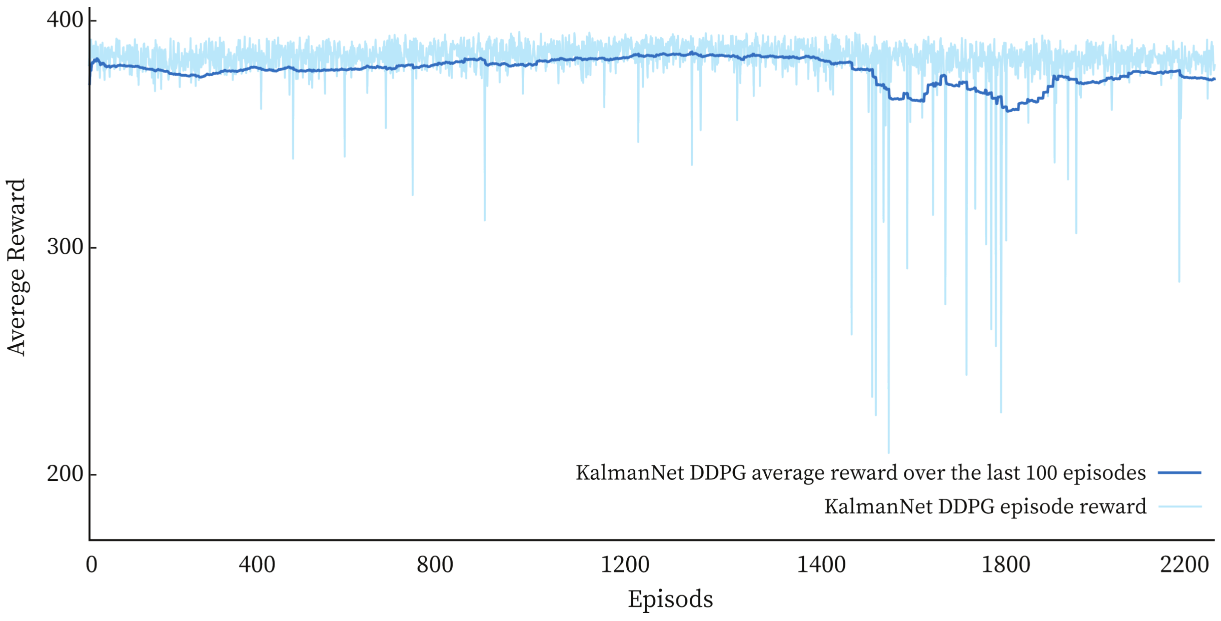

Figure 3.40: training progress of the DDPG agent for 2200 epochs (KalmanNet)

3.3 Aerial robot pose estimation

Aerial robots, including Unmanned aerial vehicles (UAVs) such as drones or quadrotors, have become increasingly popular in recent years, with a wide range of applications, including aerial photography, surveillance, delivery, and inspection. Pose estimation is a critical component of many aerial robots applications, enabling the robot to navigate and interact with its environment safely and efficiently.

To illustrate the concepts of aerial robot pose estimation, we will focus on the quadrotor as a specific example of an aerial robot.

3.3.1 Model presentation

Understanding the dynamic model of a quadrotor is essential for controlling its position and orientation accurately. The dynamic model of a quadrotor includes the equations of motion, which describe the acceleration and angular acceleration of the quadrotor in terms of the control inputs, such as the thrust and torques produced by the rotors. It also includes the kinematics of the quadrotor, which describe the relation between the position, velocity, and orientation. The dynamic model can be used to simulate the quadrotor's behavior under different conditions and to design control algorithms that achieve the desired behavior. In this section, we will present the quadrotor dynamic model and its mathematical formulation, as well as some control techniques that have been used.

3.3.1.1 Quadrotor dynamic model

Consider a quadrotor defined in the 3 dimentional space represented in the coordinate systems and the free body diagram as shown in Figure 3.41.

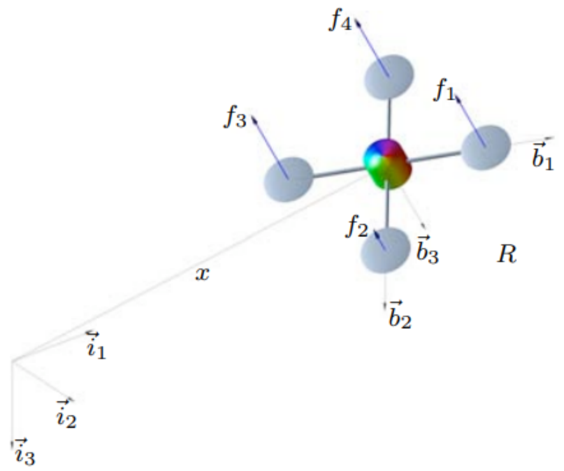

Figure 3.41: Quadrotor model [20]

The inertial frame $\{\vec{i}_1, \vec{i}_2, \vec{i}_3\}$ denotes the fixed reference frame with respect to which all motion can be referred to and the body-frame $\{\vec{b}_1, \vec{b}_2, \vec{b}_3\}$ is a frame attached to the center of mass of the vehicle. There is a vertical force for each rotor that comes from the rotation of the rotor. In addition to the forces, each rotor produces a moment perpendicular to the plane of propeller rotation.

3.3.1.2 Quadrotor motion

A quadrotor motion can be divided into two groups: rotational and translational motions. Rotational motion can be represented by Euler angles (ϕ, θ, ψ) where ϕ is roll angle, θ is pitch angle and ψ is yaw angle. Translational motion can be represented by a position vector (x, y, z).

The equations of motion of a quadrotor can be derived using Newton's second law of motion [20]. The quadrotor has four propellers, each generating a thrust force, which is used to control its motion. These equations take into account the forces acting on the quadrotor, such as gravity, thrust, and air resistance.

We define :

m	total mass of the quadrotor
g	force of gravity
l	arm length
r	position vector of the quadrotor (x, y, z)
v	velocity vector of the quadrotor $(\dot{x}, \dot{y}, \dot{z})$
ω	angular rates vector of the quadrotor (p, q, r)
I_{xx}	x moment of inertia
I_{yy}	y moment of inertia
I_{zz}	z moment of inertia
J	inertia matrix with respect to the body-fixed frame such us $J = \mathbf{diag}([I_{xx}, I_{yy}, I_{zz}])$
R	rotational matrix of the quadrotor from the body-fixed frame to the inertial frame
T_Σ	total sum fo thrusts along $-\vec{b}_3$ direction
M_i	moment on each control axis i
M	total moments acting on the body fixed frame

The rotational matrix R is given by: [2]

$$R = \begin{bmatrix} \cos\phi\cos\theta & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \sin\phi\cos\theta & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi \\ -\sin\theta & \cos\theta\sin\psi & \cos\theta\cos\psi \end{bmatrix} \tag{3.43}$$

3.3.1.3 Translational motion

The translational motion of a quadrotor refers to its linear motion in three-dimensional space. The quadrotor can move forward, backward, left, right, up, or down by adjusting the thrust of its four propellers. The rate of change for the 6 state translational variables ($\dot{x}$, $\dot{y}$, $\dot{z}$, $\ddot{x}$, $\ddot{y}$, $\ddot{z}$) can be obtained by the following translational motion equations:

$$\dot{\vec{\mathbf{r}}} = \vec{\mathbf{v}} \tag{3.44}$$

$$m\dot{\vec{\mathbf{v}}} = mg\,\vec{\mathbf{i}}_3 + \mathbf{R} \cdot T_\Sigma \cdot (-\vec{\mathbf{b}}_3) \qquad \text{(Newton's 2nd law: } \frac{d}{dt}(m\vec{v}) = \sum \vec{\mathbf{F}}_i\text{)} \tag{3.45}$$

In detail :

$$\begin{aligned} \ddot{x} &= -\frac{1}{m}\left(\sin\psi \cdot \sin\phi + \cos\psi \cdot \sin\theta \cdot \cos\phi\right) \cdot T_\Sigma \\ \ddot{y} &= -\frac{1}{m}\left(-\cos\psi \cdot \sin\phi + \sin\psi \cdot \sin\theta \cdot \cos\phi\right) \cdot T_\Sigma \\ \ddot{z} &= -\frac{1}{m}\cos\theta \cdot \cos\phi \cdot T_\Sigma + g \end{aligned} \tag{3.46}$$

3.3.1.4 Rotational motion

The rate of change for the other 6 state variables behind ($\dot\phi$, $\dot\theta$, $\dot\psi$, $\dot p$, $\dot q$, $\dot r$) can be obtained from the following rotational motion equations:

$$\dot{\mathbf{R}} = \mathbf{R} \cdot \hat{\vec{\omega}} \tag{3.47}$$

$$\mathbf{J}\,\dot{\vec{\omega}} + \vec{\omega} \cdot \mathbf{J}\,\vec{\omega} = \mathbf{M} \qquad \text{(Newton's 2nd law: } \frac{d}{dt}(\vec{\mathbf{H}}) = \sum \vec{\mathbf{M}}_i) \tag{3.48}$$

where $\hat{\cdot}$ is the hat map: $\mathbb{R}^3 \to SO(3)$, also know as skew-symmetric matrix. It is given by:

$$\hat{\vec{\omega}} = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \tag{3.49}$$

Using Euler angle parameterization:

$$\begin{pmatrix} \dot\phi \\ \dot\theta \\ \dot\psi \end{pmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \tag{3.50}$$

$$\begin{aligned} \dot p &= \frac{I_{yy} - I_{zz}}{I_{xx}} \cdot qr + \frac{1}{I_{xx}} \cdot M_1 \\ \dot q &= \frac{I_{zz} - I_{xx}}{I_{yy}} \cdot rp + \frac{1}{I_{xx}} \cdot M_2 \\ \dot r &= \frac{I_{xx} - I_{yy}}{I_{zz}} \cdot qp + \frac{1}{I_{xx}} \cdot M_3 \end{aligned} \tag{3.51}$$

3.3.2 Nonlinear quadrotor state-space model

A quadrotor has a total of 12 state variables defined below:

$$\mathbf{X} = [\,x \quad y \quad z \quad \dot x \quad \dot y \quad \dot z \quad \phi \quad \theta \quad \psi \quad p \quad q \quad r\,]^T \tag{3.52}$$

where x, y and z represents the quadrotor position in the inertial frame, $\dot x$, $\dot y$ and $\dot z$ are the quadrotor's velocity expressed in the body frame, ϕ, θ and ψ are the roll, pitch and yaw angles of the quadrotor and p, q and r are the angular rates of the quadrotor.

The nonlinear quadrotor state-space model $\dot X = f(X, U)$ can be summarized using equations

(3.44), (3.46), (3.50) and (3.51) as follows:

$$
\begin{aligned}
\dot{x} &= \dot{x} \\
\dot{y} &= \dot{y} \\
\dot{z} &= \dot{z} \\
\ddot{x} &= -\frac{1}{m}\left(\sin\psi \cdot \sin\phi + \cos\psi \cdot \sin\theta \cdot \cos\phi\right) \cdot T_\Sigma \\
\ddot{y} &= -\frac{1}{m}\left(-\cos\psi \cdot \sin\phi + \sin\psi \cdot \sin\theta \cdot \cos\phi\right) \cdot T_\Sigma \\
\ddot{z} &= -\frac{1}{m}\cos\theta \cdot \cos\phi \cdot T_\Sigma + g \\
\dot{\phi} &= p + \sin\phi \cdot \tan\theta \cdot q + \cos\phi \cdot \tan\theta \cdot r \\
\dot{\theta} &= \cos\phi \cdot q - \sin\phi \cdot r \\
\dot{\psi} &= \sin\phi \cdot \sec\theta \cdot q + \cos\phi \cdot \sec\theta \cdot r \\
\dot{p} &= \frac{I_{yy} - I_{zz}}{I_{xx}} \cdot qr + \frac{1}{I_{xx}} \cdot M_1 \\
\dot{q} &= \frac{I_{zz} - I_{xx}}{I_{yy}} \cdot rp + \frac{1}{I_{xx}} \cdot M_2 \\
\dot{r} &= \frac{I_{xx} - I_{yy}}{I_{zz}} \cdot pq + \frac{1}{I_{xx}} \cdot M_3
\end{aligned}
\tag{3.53}
$$

In this application, we take four control inputs which are total thrust T_Σ and moments M_i on each control axis i as follow:

$$
\mathbf{U} = [u_1, u_2, u_3, u_4]^T = [T_\Sigma, M_1, M_2, M_3]^T \tag{3.54}
$$

3.3.3 Quadrotor attitude control

Quadrotor attitude control involves complex technical processes to regulate its attitude accurately. Attitude control primarily focuses on maintaining the desired orientation of the quadrotor in three-dimensional space. The control system relies on sensor data to estimate the current attitude of the quadrotor. This attitude estimation is crucial for generating control commands that dictate the desired orientation. One commonly used control technique for attitude control is the Proportional-Integral-Derivative (PID) controller.

By assuming $\dot{\phi} \approx p$, $\dot{\theta} \approx q$ and $\dot{\psi} \approx r$ and near the hovering position, we can design a PID controller working in a moderate range of attitude commands as follow.

$$
\begin{aligned}
T_\Sigma &= mg - \left[K_{P,\dot{Z}}(\dot{Z}^{cmd} - \dot{Z}) + K_{I,\dot{Z}} \int (\dot{Z}^{cmd} - \dot{Z}) + K_{D,\dot{Z}}(\ddot{Z}^{cmd} - \ddot{Z})\right] \\
M_1 &= K_{P,\phi}(\phi^{cmd} - \phi) + K_{I,\phi} \int (\phi^{cmd} - \phi) + K_{D,\dot{Z}}(p^{cmd} - p) \\
M_2 &= K_{P,\theta}(\theta^{cmd} - \theta) + K_{I,\theta} \int (\theta^{cmd} - \theta) + K_{D,\theta}(q^{cmd} - q) \\
M_3 &= K_{P,\psi}(\psi^{cmd} - \psi) + K_{I,\psi} \int (\psi^{cmd} - \psi) + K_{D,\psi}(r^{cmd} - r)
\end{aligned}
\tag{3.55}
$$

3.3.4 Quadrotor pose estimation

In this section, we will explore and evaluate the classic EKF for pose estimation in a quadrotor with random values of its parameters considering its advantages, limitations and applicability in various scenarios.

3.3.4.1 EKF based pose estimation

We consider the state-space model of the quadrotor $\dot{X} = f(X, U)$ where f is given by equation (3.53) using as output:

$$Y = h(X) = [p, q, r]^T \tag{3.56}$$

In this can, we supposed that the quadrotor is equipped with IMU sensor. The IMU data contains the accelerations obtained from accelerometer and the orientation angles obtained from the gyroscope. In this application, we will focus on the orientation mesure. Usually, this mesure is passed through a low-pass filter to suppress the high frequency noise, but in this application, we don't need to.

We used the same discrete-time version of EKF algorithm (2) presented in the mobile robot application.

The process jacobian F_k such as $F = \begin{bmatrix} f_x \; f_y \; f_z \; \ldots \end{bmatrix}$ is given by:

$$F_k = \frac{df}{dX}\Big|_{\hat{X}_k, U_k} = \begin{bmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} & \frac{\partial f_x}{\partial z} & \cdots \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} & \frac{\partial f_y}{\partial z} & \cdots \\ \frac{\partial f_z}{\partial x} & \frac{\partial f_z}{\partial y} & \frac{\partial f_z}{\partial z} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.57}$$

The measurement jacobian H_k is given by:

$$H_k = \frac{dh}{dX}\Big|_{\hat{X}_k, U_k} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.58}$$

3.3.4.2 Simulations

In this part, we simulate the EKF algorithm of quadrotor pose estimation using random values of the process noise covariance Q and the measurement noise covariance R which are supposed to be constant. We perform test in differents situations under ideal and non-ideal conditions including modeling erros, noises and offset of the IMU sensor. The results are shown in figures (3.42), (3.43) and (3.44).

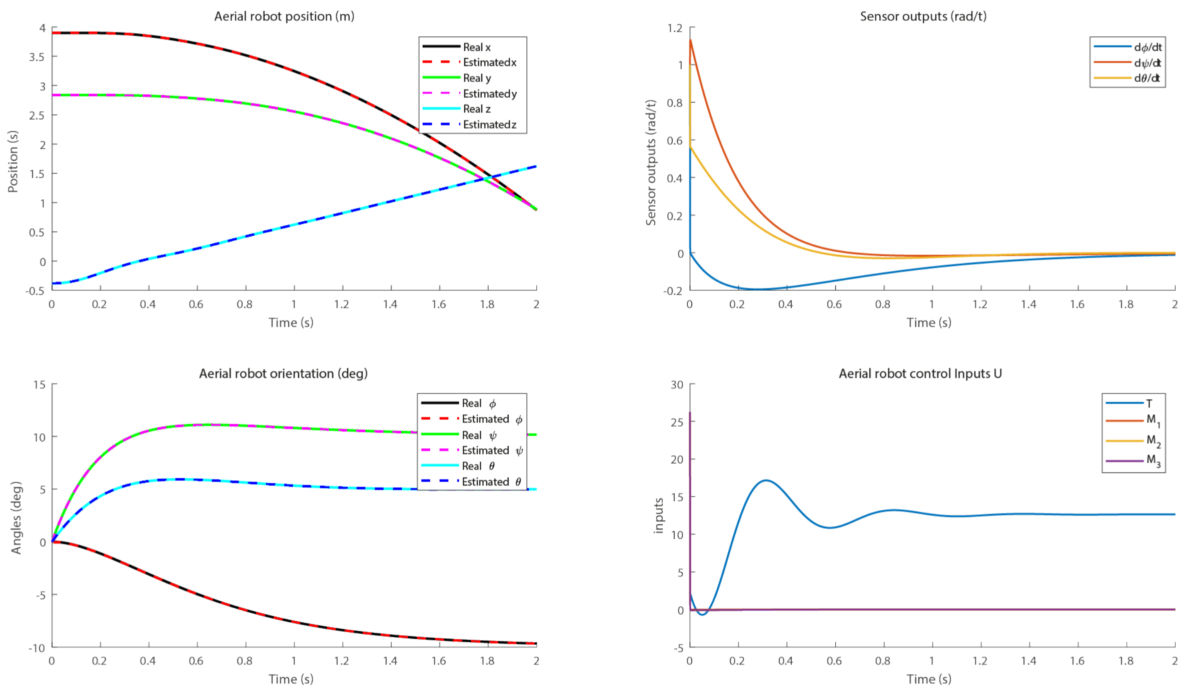

Figure 3.42: Testing the standard EKF for quadrotor pose estimation under ideal conditions

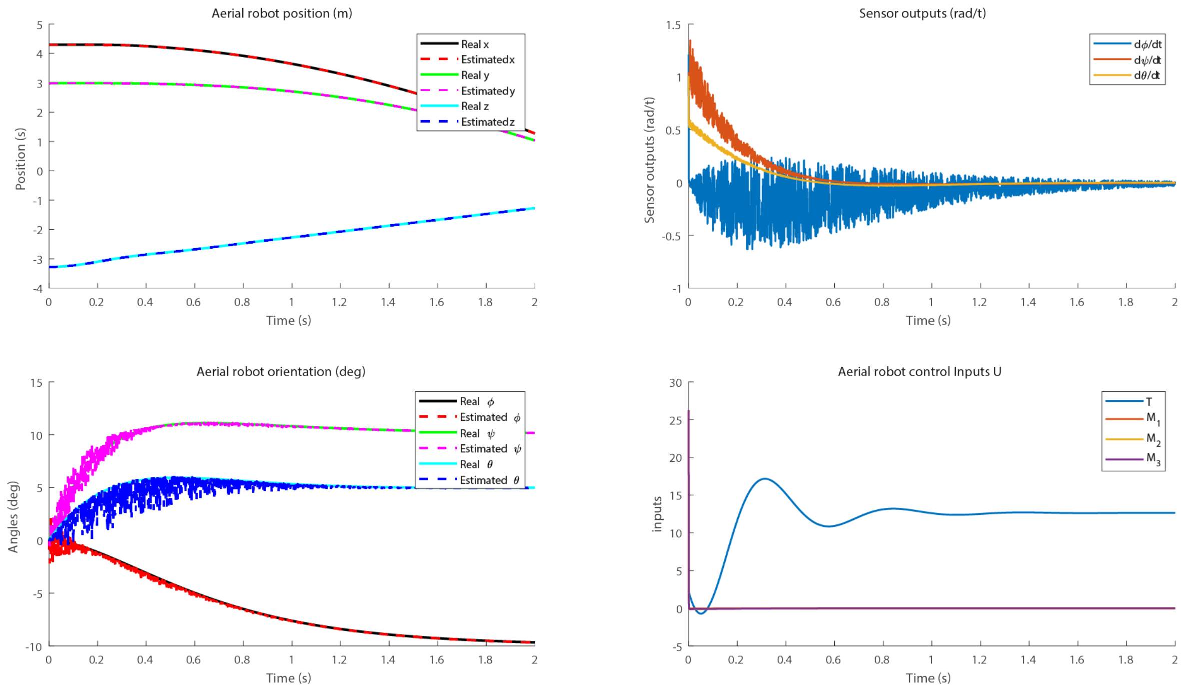

Figure 3.43: Testing the standard EKF for quadrotor pose estimation under non-ideal conditions (with modeling errors and noisy measurements)

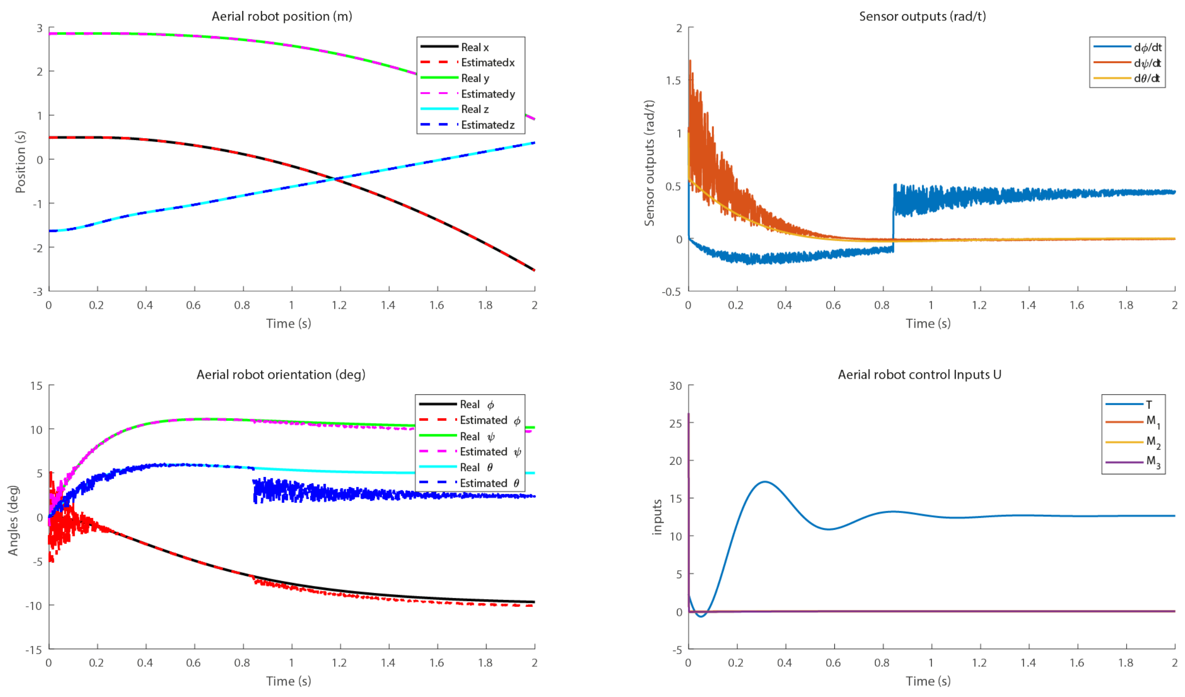

Figure 3.44: Testing the standard EKF for quadrotor pose estimation under non-ideal conditions with offset of the IMU sensor

3.3.4.3 Results and discussion

Under ideal conditions as illustrated in figure (3.42), our EKF pose estimation algorithm achieved excellent results. However, when we introduced non-ideal conditions such as modeling errors and sensor noise, we observed a noticeable degradation in the accuracy of the estimates as shown in figure (3.43).

We discovered that even small modeling errors had a significant impact on the pose estimation accuracy indicating the sensitivity of our algorithm to modeling errors.

Introducing sensor noise had a substantial effect on the pose estimation accuracy. As expected, the error increased with higher levels of sensor noise. The algorithm struggled to filter out the noise and accurately estimate the quadrotor's pose.

We also investigated the impact of sensor offset on the performance of our algorithm. By introducing a systematic bias in the sensor readings as shown in figure (3.44), we observed a consistent deviation in the pose estimation results which led to a divergence of the EKF algorithm indicating its limited ability to compensate for such biases.

3.3.4.4 Conclusion

Our tests highlighted the impact of modeling errors, sensor noise, and offset on the performance of the standard EKF pose estimation algorithm. These factors introduced significant challenges, leading to increased errors in the pose estimation. To deal with this kind of problem or limitation, we will introduce RL and DL based optimized versions of this algorithm by adding an adaptive part allowing to adapt the values of the Q and R parameters according to the changes in the environment.

3.3.5 QR estimator for quadrotor pose estimation

To improve the performance of the pose estimation process, we will introduce optimized versions of the EKF using Deep Learning and Reinforcement Learning. We will discuss various simulations for each version highlighting the strengths and limitations of each one.

Due to the poor performance observed with second approach using the KalmanNet architecture in the mobile robot application, we have decided to exclusively utilize the first approach for this application with only DDPG and TD3 agents which gives the best estimation rewards in the previous application.

As in the mobile robot application, the first approach was to use an RL agent to estimate Q_k and R_k for each k step. The EKF requires values for the process noise covariance and the measurement noise covariance, which can have a significant impact on the accuracy and convergence of the filter. RL can be used to learn the optimal values of this parameters by optimizing a reward function that measures the accuracy of the estimated state. To speed up the training of the RL agent, we pre-train the actor network using DL to obtain a well-approximated values of Q and R in each situation and conditions.

The application of the QR estimator for mobile robot pose estimation has provided valuable insights and proven its effectiveness. However, the transition to quadrotor pose estimation introduces a new level of difficulty due to the inherent complexity of the quadrotor model. Unlike a mobile robot with a simpler kinematic structure, the quadrotor exhibits a more complex dynamics involving multiple rotors, complex aerodynamic forces and higher degrees of freedom. This increased complexity presents a significant challenge in accurately estimating the Q and R parameters within the EKF estimator. The determination of appropriate Q and R values becomes more critical as the quadrotor model involves more state variables and noise sources.

In order to address the challenge of estimating the process noise covariance matrix Q, which consists of 12 elements for the quadrotor state model, a strategy was employed to simplify the problem while maintaining estimation accuracy. It was assumed that 6 states within the model are not significantly affected by noise sources, allowing the focus to be placed on estimating the remaining 6 noise covariance values for the remaining states. This approach was based on the understanding that certain states exhibit more deterministic behavior or are less susceptible to noise. By reducing the dimensionality of the problem, the estimation effort was concentrated on the states that are more likely to be affected by noise. The EKF algorithm was modified to update only the selected states' covariance matrix elements in Q, while keeping the others fixed. Careful consideration was given to selecting the noise-free states based on a comprehensive analysis of the quadrotor dynamics.

3.3.5.1 Environment setup

We create a simulation environment on SIMULINK with a quadrotor model and an IMU sensor.

Observation space: the size of the observation space is 15×1, consisting of the measurement error $e_Y = Y_k - \hat{Y}_k$ and the variation in the predicted state $X_k - X_{k-1}$.

Action space: the size of the action space is 9×1, consisting of elements q_i and r_i of matrices Q and R such as:

$$Q = \begin{bmatrix} 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & Q_{position} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & Q_{orientation} \end{bmatrix} \qquad R = \begin{bmatrix} r_1 & 0 & 0 \\ 0 & r_2 & 0 \\ 0 & 0 & r_3 \end{bmatrix} \tag{3.59}$$

where:

$$Q_{position} = \begin{bmatrix} q_{\dot{x}} & 0 & 0 \\ 0 & q_{\dot{y}} & 0 \\ 0 & 0 & q_{\dot{z}} \end{bmatrix} \qquad Q_{orientation} = \begin{bmatrix} q_p & 0 & 0 \\ 0 & q_q & 0 \\ 0 & 0 & q_r \end{bmatrix} \tag{3.60}$$

Reward: in this case, the reward function is based on the RMSE metric such as:

$$reward = 1 - e_{RMSE} = 1 - \sqrt{(X_k - \hat{X}_k)(X_k - \hat{X}_k)^T} \tag{3.61}$$

Stopping criteria: we use the "isdone" signal to stop the training episode in case of error divergence using the $error_{RMSE} > 10$ as stopping criteria.

3.3.5.2 Pre-training the actor network with Deep Learning

To speed up the learning process, we pre-train the actor network using Deep Learning. In this simulation, we have the control of the sensors noise and the modeling errors, so, we can easily get a good approximation of Q and R to be used to train an RNN.

Data generation: following the same approach of the mobile robot application, we create a function that takes environment parameters including modeling errors, noise color, characteristics and power as inputs and outputs an acceptable values of Q and R parameters. We saved these parameters as outputs along with the inputs (e_y and ΔX_p) as sequences. We created a dataset of 1200 simulations, each with a distinct environment characteristics. This data should be fine for pre-training the actor network to accurately estimate the appropriate Q and R parameters.

Actor network architecture: We used an RNN actor network with the following architecture:

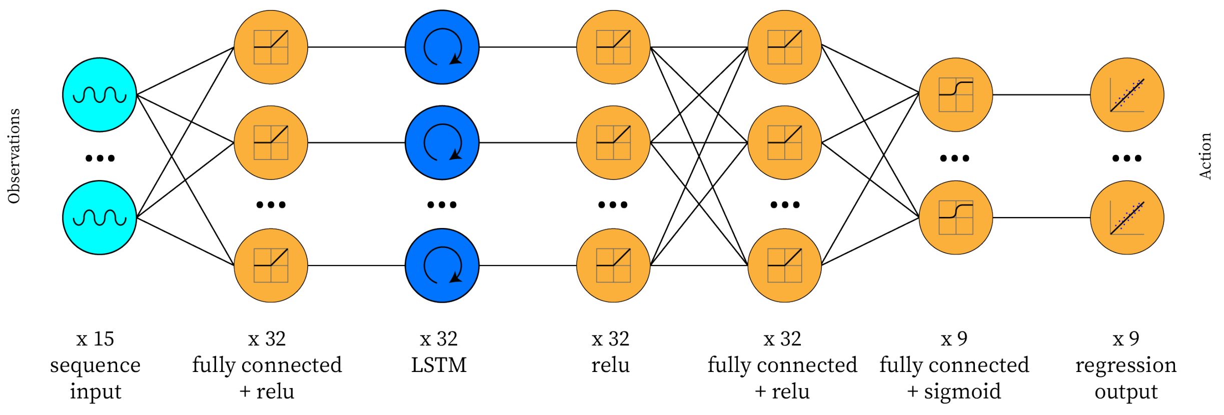

Figure 3.45: QR estimator Actor Network architecture for aerial robot

Actor network training: we trained the actor network with the following settings:

Optimizer	Adam	Mini-batch size	128
Initial learning rate	0.001	Shuffle	every-epoch
Learning rate drop factor	0.9	Sequence length	longest
Max epochs	30	Validation frequency	5

Training results: Figure (3.46) represents the learning curve of actor network training for quadrotor pose estimation. After 30 epochs of training, the training and validation loss reached 0.038. The RMSE was 0.27 for the training data and 0.28 for the validation data which is acceptable.

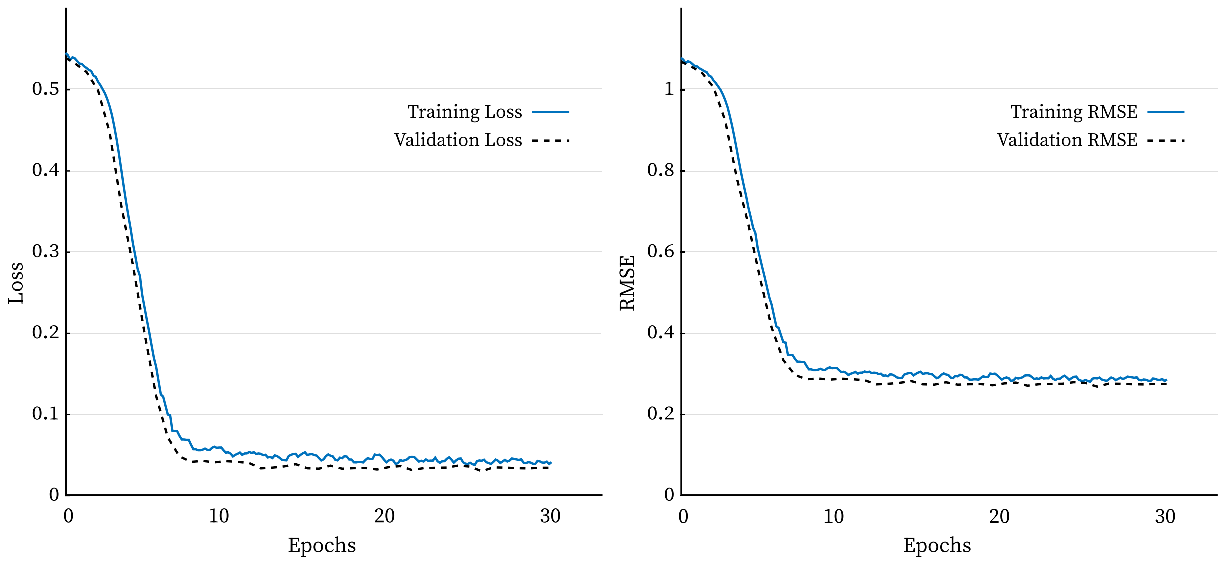

Figure 3.46: Learning curve of the actor network training for quadrotor pose estimation

After testing the pre-trained actor network, we get the following results:

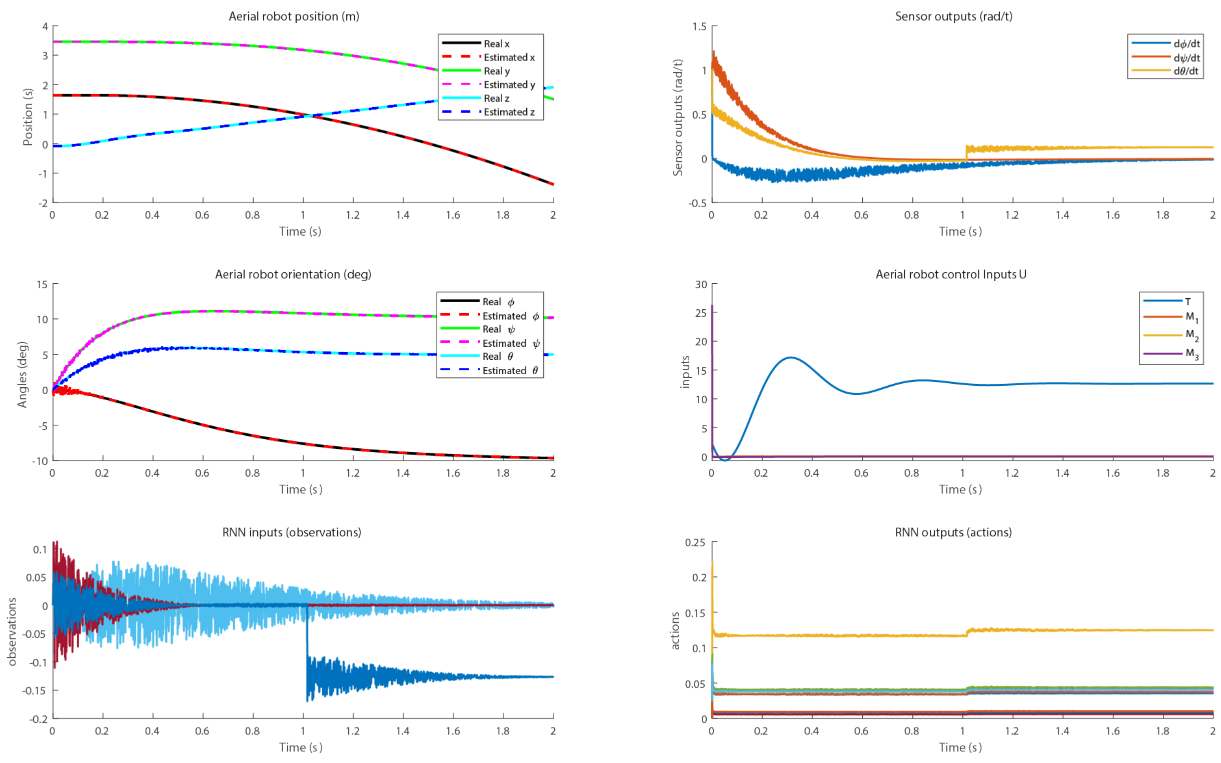

Figure 3.47: Testing the pre-trained Actor Network for quadrotor

Analyzing the results:

The training can be considered successful based on the achieved low loss and RMSE values. The results confirm the efficacy of the pre-trained model, as the error remains consistently below 0.3 during the test. Additionally, it is evident that the RNN exhibits adaptability to environmental variations, including sensor measurement failures, enabling accurate pose estimation.

These results highlights the possibility of further enhancement by incorporating RL techniques. Using RL, the agent can learn to adapt even more effectively to changing conditions and enhance the estimation accuracy.

3.3.5.3 DDPG agent

Agent architecture: The DDPG agent use an actor network that takes the observation and output the action, and a critic network who takes the state (observation) and the action as inputs, and outputs a scalar number which define the quality of the State/Action pairs.

For the actor network, we used the RNN that we pre-trained previously. For the critic, we used the following architecture:

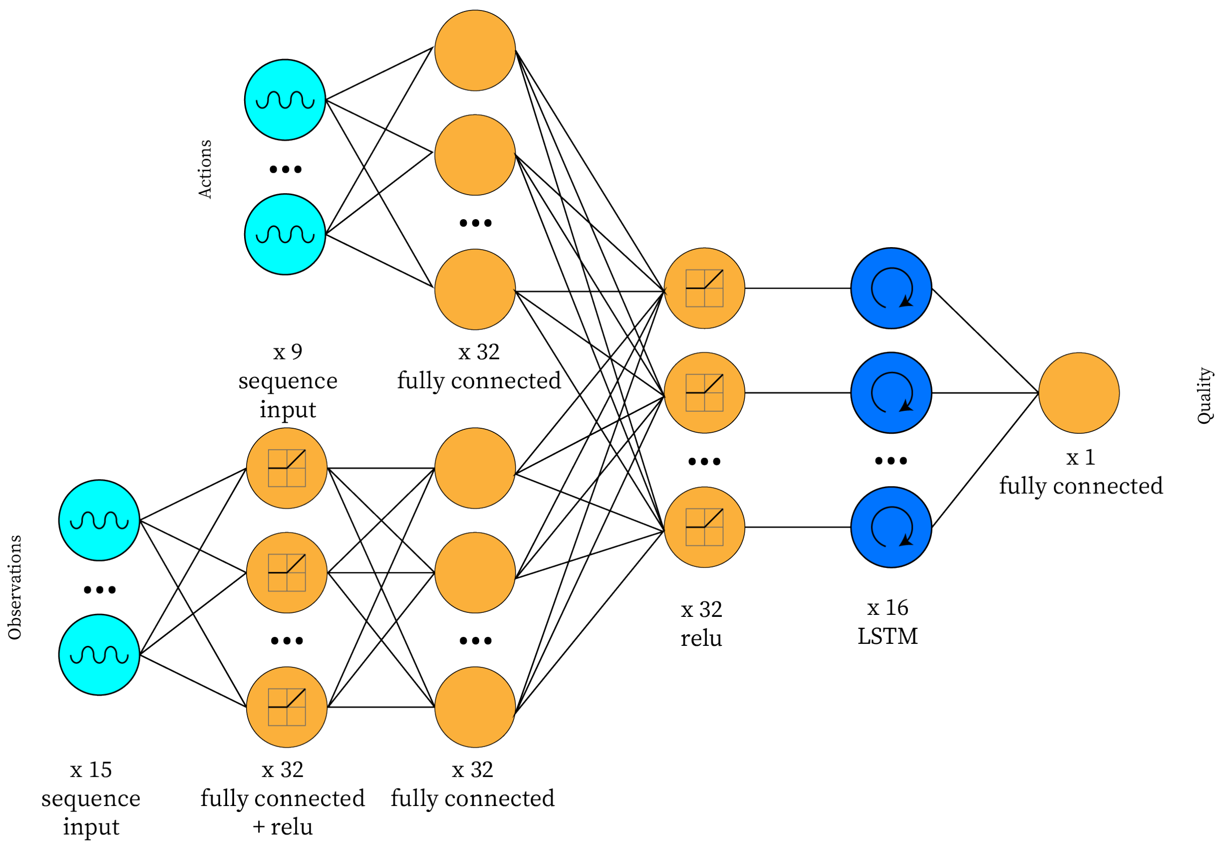

Figure 3.48: DDPG Critic Network architecture of QR estimator for aerial robot

DDPG agent training process: We trained the DDPG agent with these parameters:

For critic network

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	0.1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	0.1

Agent parameters

Sample time	0.001
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

DDPG agent training results: After 16 hours of training, the agent successfully completed 1000 episodes, achieving an average reward of 1918 out of 2000 over the last 100 episodes.

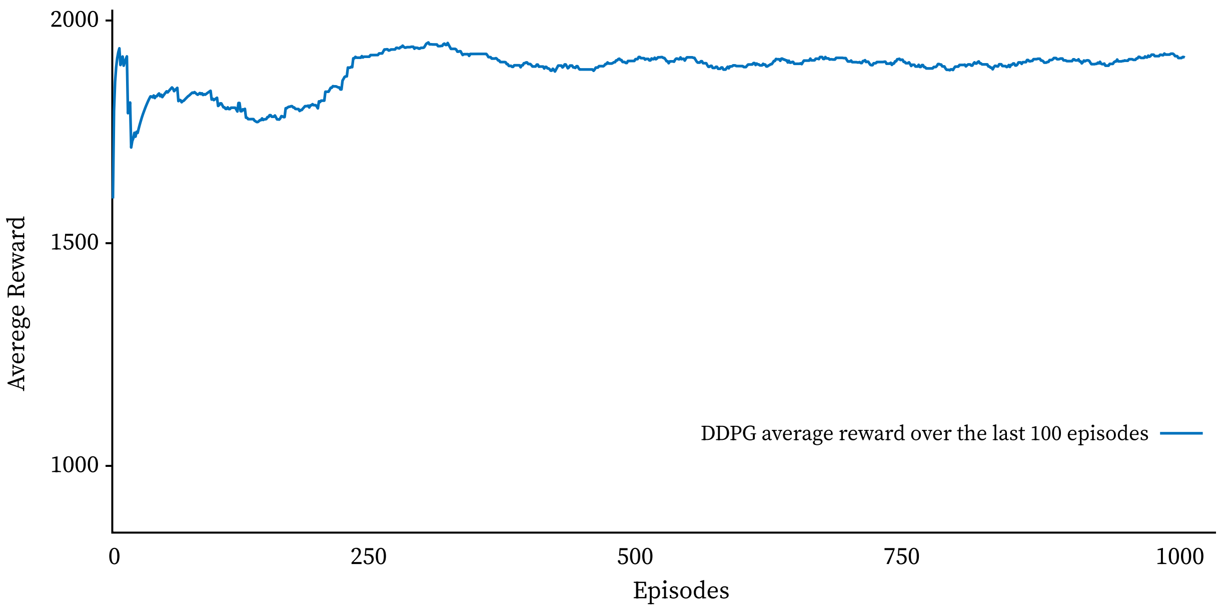

Figure 3.49: Training progress of the DDPG agent for quadrotor pose estimation

Testing the trained DDPG agent: The implementation and utilization of the optimized policy of the DDPG agent gives the result shown in the figure (3.50) bellow:

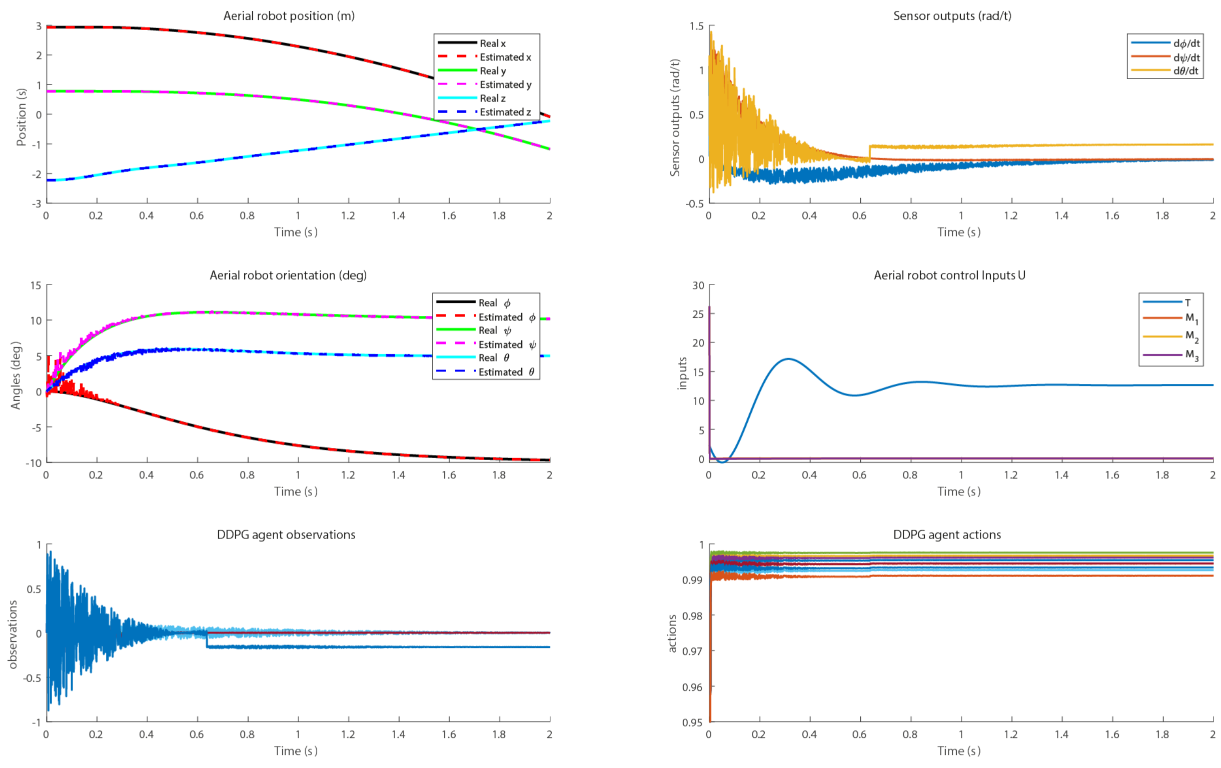

Figure 3.50: Testing the trained DDPG agent for quadrotor pose estimation

Analyzing the results:

Figure (3.49) present the learning curve, showing the agent's cumulative average reward per episode which steadily increases over training iterations, indicating improvement in the pose estimation task. The episode rewards plot reveals fluctuations, with a general upward trend, indicating the agent's ability to learn and adapt to the environment.

Temporary decreases in the value of the episode's reward during RL training can be interpreted as natural fluctuations in the learning process. RL algorithms often incorporate exploration strategies to discover optimal policies. Temporary decreases in the reward can indicate that the RL agent is exploring new actions or policies, which may initially result in suboptimal rewards. These fluctuations are expected as the agent gathers information about the environment and learns from its experiences.

Also, RL agents need to balance exploration and exploitation. Temporary decreases in reward can occur when the agent transitions from exploration to exploitation. Initially, the agent explores different actions to learn about the environment, but during the transition to exploitation, it focuses on exploiting the learned knowledge. This shift may cause a temporary drop in reward as the agent refines its policy.

Evaluation results demonstrate reduced estimation error, faster convergence and improved estimation accuracy of the QR estimator using the DDPG agent compared to the standard EKF.

3.3.5.4 TD3 agent

Agent architecture: As we did in the mobile robot application, we used the same actor and critic networks as the DDPG.

Agent training process: We trained the TD3 agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.001
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	32
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

TD3 agent training results: After 14 hours of training, the agent successfully completed 1000 episodes, achieving an average reward of 1948 out of 2000 over the last 100 episodes.

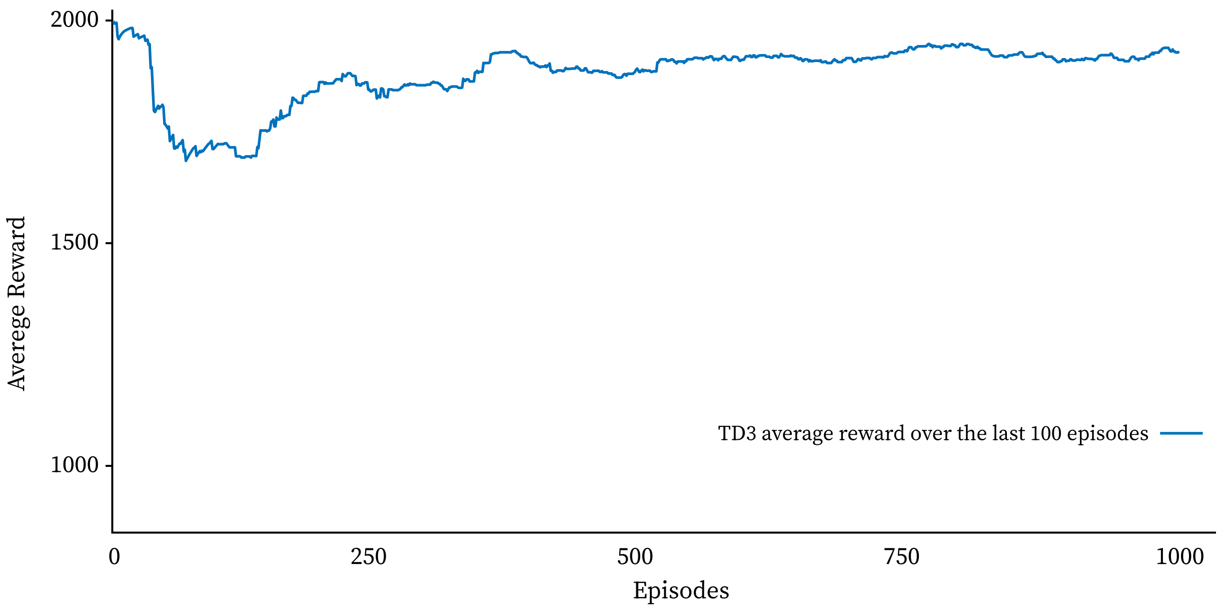

Figure 3.51: training progress of the TD3 Agent for quadrotor pose estimation

Testing the trained agent: The implementation and utilization of the optimized policy of the TD3 agent gives the result shown in the figure (3.52) bellow:

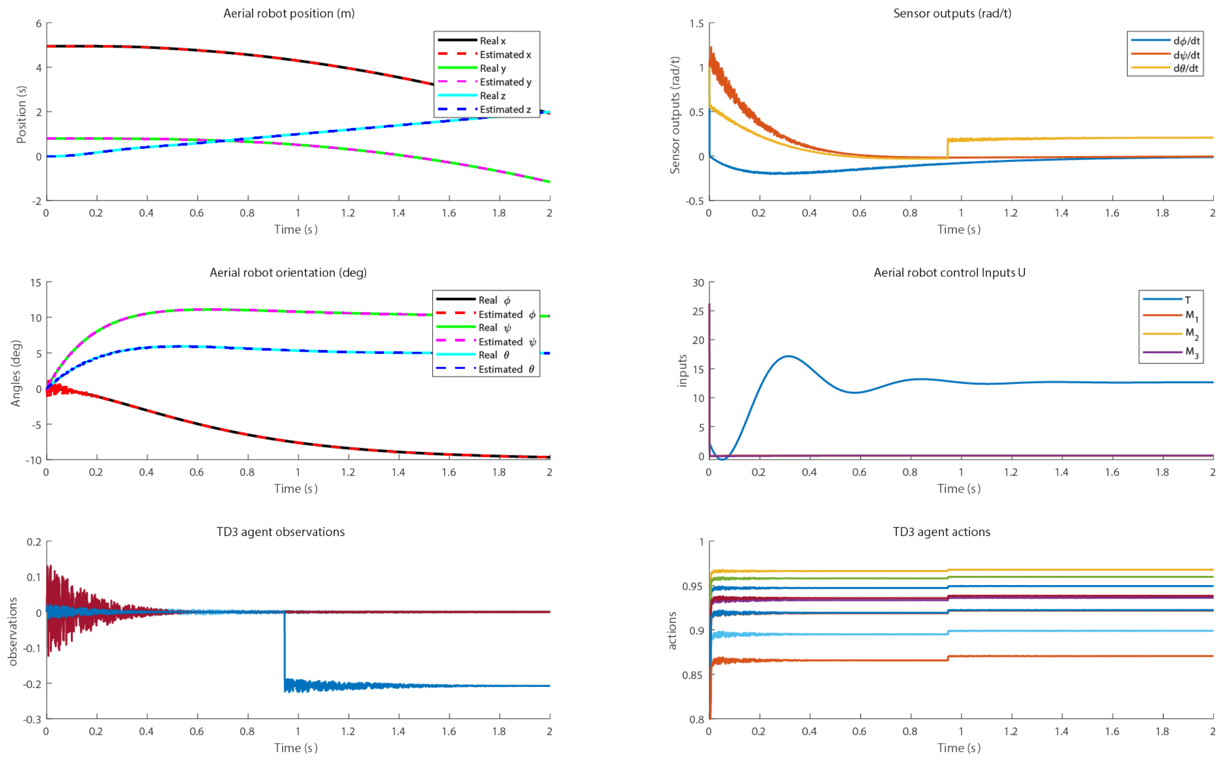

Figure 3.52: Testing the trained TD3 agent for quadrotor pose estimation

Analyzing the results:

Figure (3.51) present the learning curve, showing the TD3 agent's cumulative average reward per episode which steadily increases over training iterations, indicating improvement in the pose estimation process.

The results obtained from the TD3 agent demonstrate its superior performance compared to the previous DDPG agent. With less than 14 hours of training, the TD3 agent completed successfully 1000 episodes, allowing it to encounter and adapt to various environmental conditions more efficiently and rapidly.

Evaluation results demonstrate reduced estimation error, faster convergence and improved estimation accuracy of the QR estimator using the DDPG agent compared to the standard EKF.

3.3.5.5 Performance comparison

Similar to the mobile robot application, we conducted 200 simulations under different conditions and computed the sum of the square root errors using the formula:

$$\int_{t_0}^{t_f} \sqrt{(x-\hat{x})^2+(y-\hat{y})^2+(z-\hat{z})^2+(\phi-\hat{\phi})^2+(\psi-\hat{\psi})^2+(\theta-\hat{\theta})^2}\, dt \tag{3.62}$$

where x, y and z are actual positions, $\hat{x}$, $\hat{y}$ and $\hat{z}$ are estimated positions, ϕ, ψ and θ are actual orientation angles, $\hat{\phi}$, $\hat{\psi}$ and $\hat{\theta}$ are estimated orientation angles.

We then calculated the average across the 200 simulations. The results are presented in Figure (3.53).

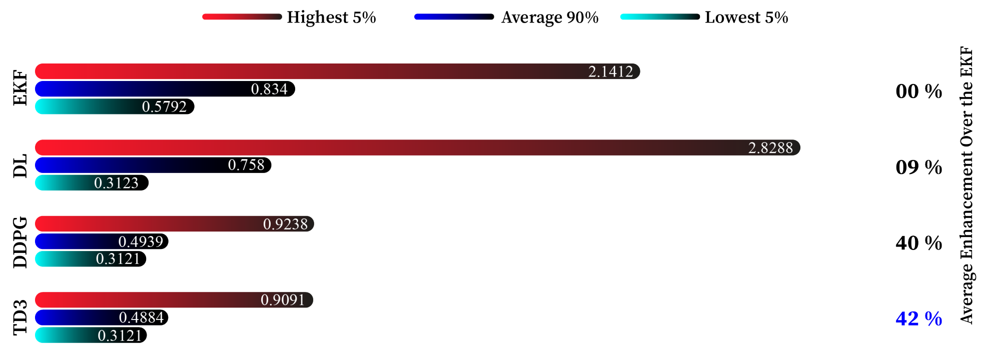

Figure 3.53: Comparison of the average error of 200 tests of the standard EKF, DL-EKF, DDPG agent and TD3 agent for quadrotor pose estimation

Analyzing the results:

The results of the test further confirm the earlier findings. The classic EKF demonstrated an average error of 0.834 across the 200 simulation tests. Implementing the trained RNN with DL resulted in an average error of 0.758, representing a significant improvement of 9% over the classic EKF. The DDPG and TD3 agents performed even better with an average error of 0.49, indicating an improvement of nearly 40% compared to the classic EKF. These results highlight the superior performance of RL-based agents in terms of estimation accuracy and adaptability in non-ideal conditions.

3.3.5.6 Results discussion and conclusion

The results demonstrate the potential of the QR estimator approach even with a relatively small RNN architecture. Despite the aerial robot model being more complex than the mobile robot one, we achieved a significant improvement of nearly 42% over the standard EKF.

The high adaptability of this approach to changes in the environment, sensor reliability, or the robot itself is a key advantage. It is important to mention that during all the tests, the model parameters used by the estimator were clearly distinct from the actual system parameters, demonstrating the estimator's robustness.

These results highlight the effectiveness and potential of using RL-based optimization approaches for pose estimation tasks even for complex dynamic systems.

3.4 Conclusion

The QR estimator approach for an adaptive version of the EKF for single-agent pose estimation have shown to be effective in improving accuracy and adaptability. The KalmanNet approach, although not as good as the first approach, still demonstrate an improvement over the standard EKF. These approaches open up possibilities for enhancing pose estimation capabilities in single-agent scenarios, providing valuable tools for various robotic applications. Future research should focus on further refining and validating these approaches, as well as exploring their performance in more complex and dynamic real-world environments.

Chapter 4

Pose estimation for multi-agent systems

4.1 Introduction

This chapter presents the proposed approach for collaborative pose estimation in a multi-agent systems. The objective is to improve the accuracy and robustness of pose estimation by leveraging the information from neighboring agents.

4.2 Pose estimation for mobile robot multi-agent system

In a multi-agent system, the concept of relative pose estimation plays a crucial role in improving the overall accuracy of individual robot's position estimation. Instead of relying solely on external fixed reference points or absolute localization methods, each robot leverages the presence of other robots as dynamic anchor points.

4.2.1 Proposed approach

By utilizing UWB sensors, the robot is able to measure the distances between itself and other neighboring robots. These measurements serve as valuable information for determining the relative positions and orientations among the robots. Additionally, the UWB sensors can also function as communication devices, allowing the robots to exchange estimated position information with one another.

This exchange of information enables each robot to have access to the estimated positions of all other robots in the system. Consequently, the output of the pose estimation process is no longer limited to the individual robot's position but also includes the estimated positions of other robots as well. This expanded output is represented by Equation (4.1).

$$Y = h(X) = \begin{cases} X_3 \\ (X_1 - AP_{1x})^2 + (X_2 - AP_{1y})^2 \\ (X_1 - AP_{2x})^2 + (X_2 - AP_{2y})^2 \\ \vdots \\ (X_1 - AP_{nx})^2 + (X_2 - P_{ny})^2 \\ (X_1 - R_{1x})^2 + (X_2 - R_{1y})^2 \\ (X_1 - R_{2x})^2 + (X_2 - R_{2y})^2 \\ \vdots \\ (X_1 - R_{nx})^2 + (X_2 - P_{ny})^2 \end{cases} \tag{4.1}$$

Where AP_i represents the position of the fixed anchor point i, and R_j denotes the estimated position of robot j.

However, one challenge that arises when employing this approach is the potential inefficiency and scalability issues in large swarm systems. As the number of robots in the swarm increases, the system size becomes massive, leading to computational and communication overheads. Furthermore, even in smaller swarm sizes, the presence of incorrect or unreliable data caused by factors such as inaccurate pose estimation or faulty distance measurements due to sensor failures, obstacles, or interference can adversely impact the relative pose estimation accuracy.

To address these challenges, a potential solution is to employ a filtering mechanism to classify the collected data, prioritizing the best-quality signals while discarding or minimizing the influence of unreliable or erroneous data. This can be achieved by passing the sensor data through a Recurrent Neural Network that assigns weights to each signal. Higher weights are assigned to signals that are deemed to be more reliable or accurate, while lower weights are given to signals that exhibit lower quality or reliability.

Subsequently, a simple algorithm can be utilized to select the signals (sensors or robots) that the robot will actively use for its pose estimation, based on the classification provided by the RNN. This process of data classification and selection is depicted in Figure (4.1). By employing this approach, the robot focuses on utilizing the signals with higher weights, which are indicative of better data quality.

This filtering and selection process helps mitigate the impact of unreliable data and enhances the overall robustness of the pose estimation system in the multi-agent setup. It enables the robot to make informed decisions by prioritizing more reliable sources of information and disregarding or minimizing the influence of potentially erroneous or inaccurate data.

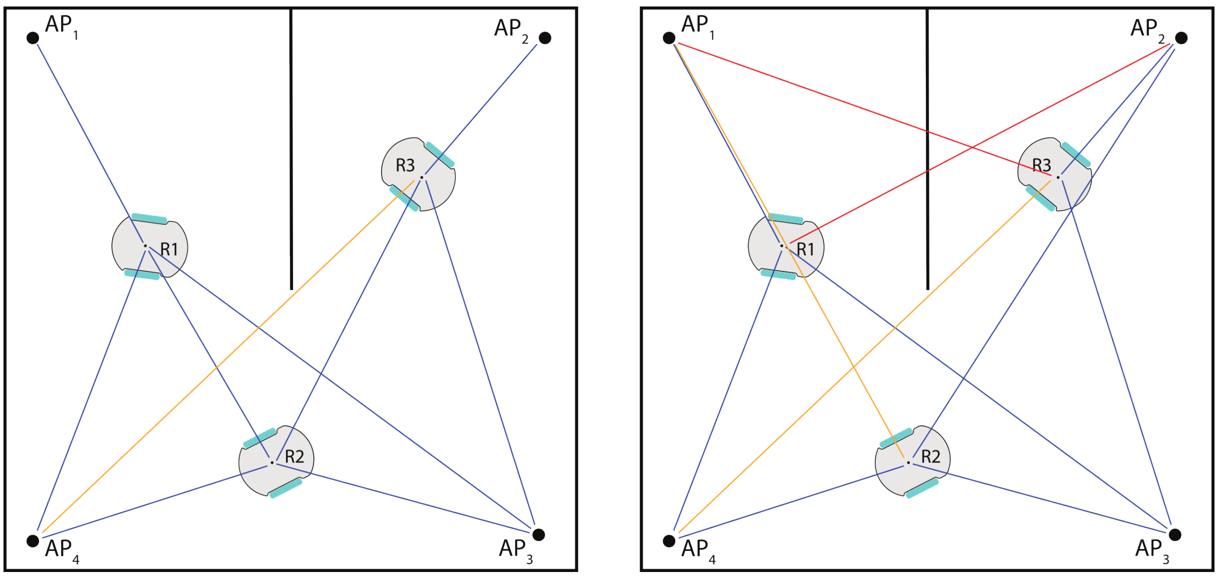

Figure 4.1: Difference between using fixed anchor points alone versus integrating a filtering and selection mechanism

Figure (4.1) illustrates the difference between utilizing only fixed anchor points and incorporating the proposed filtering and selection mechanism. While a schematic representation of the system is illustrated in Figure (4.2).

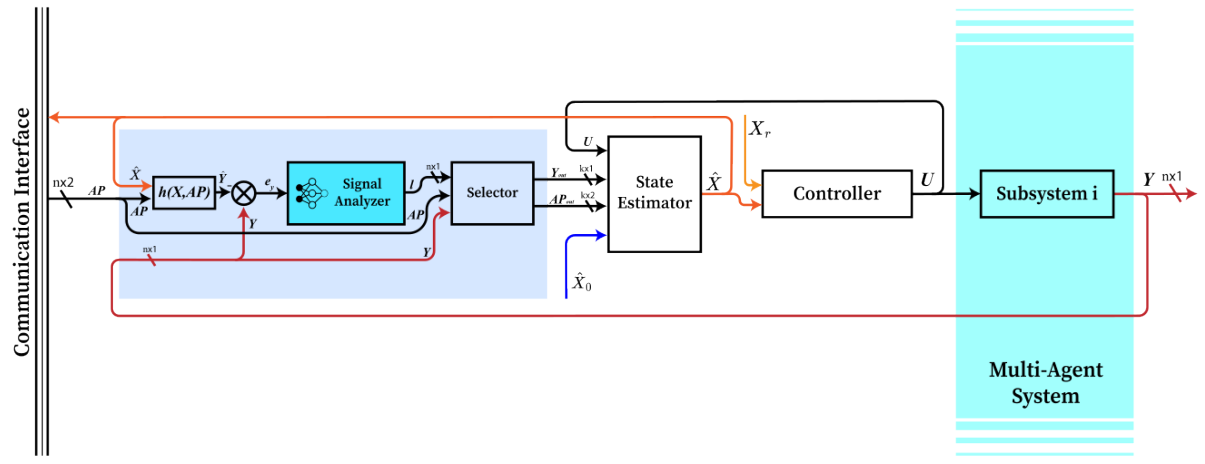

Figure 4.2: Schematic presentation of the multi-agent system with the sensors selector

4.2.2 Environment setup

Observation Space: In our application, we utilized three robots, resulting in an observation space size of 6×1. The observation space includes the error $e_Y = Y - \hat{Y}$, which represents the distance from four fixed anchor points and two other robots.

Action Space: The action space size in our application is also 6×1. Each output in the action space represents the probability of selecting a specific distance over the others.

Reward: For the reward mechanism, we provided the agent with a reward of $+1/6$ for each correct classification of a signal.

4.2.3 Pre-training the Actor Network

Similar to the single-agent approach, we will also pre-train the actor network to speed up the learning process.

Actor Network architecture: Since the problem involves signal processing and classification, we employed an RNN with a softmax output. The RNN architecture used for this application is shown in Figure (4.3).

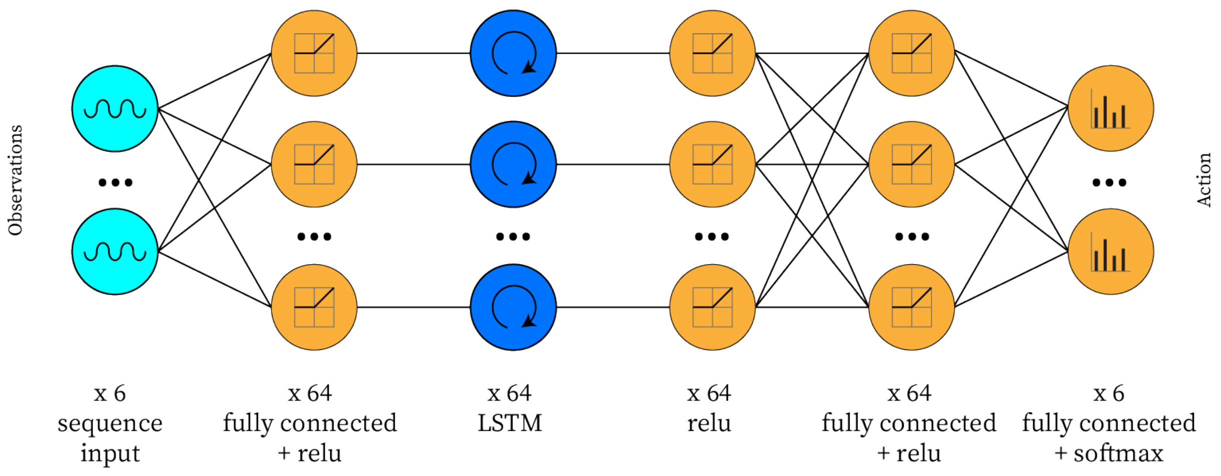

Figure 4.3: Actor Network RNN architecture for mobile robot multi-agent

Data generation: To generate the data, we assigned weights to each distance measurement based on the noise power and offset. A lower weight indicates a better quality measurement. We then applied a softmax function to obtain more unified data. Finally, we saved the weighted measurements along with the corresponding inputs (observations).

Actor Network training: We trained the actor network using the following settings:

Optimizer	Adam	Mini-batch size	1024
Initial learning rate	0.01	Shuffle	every-epoch
Learning rate drop factor	0.9	Sequence length	longest
Max epochs	2000	Validation frequency	50

DL training results: After 115 minutes of training, which corresponds to 300 epochs, a loss of 0.07 was achieved for both the training and validation data. The root mean square error (RMSE) for the training data was 0.35, while for the validation data, it was 0.38. Figure (4.4) provides a visual representation of the training progress.

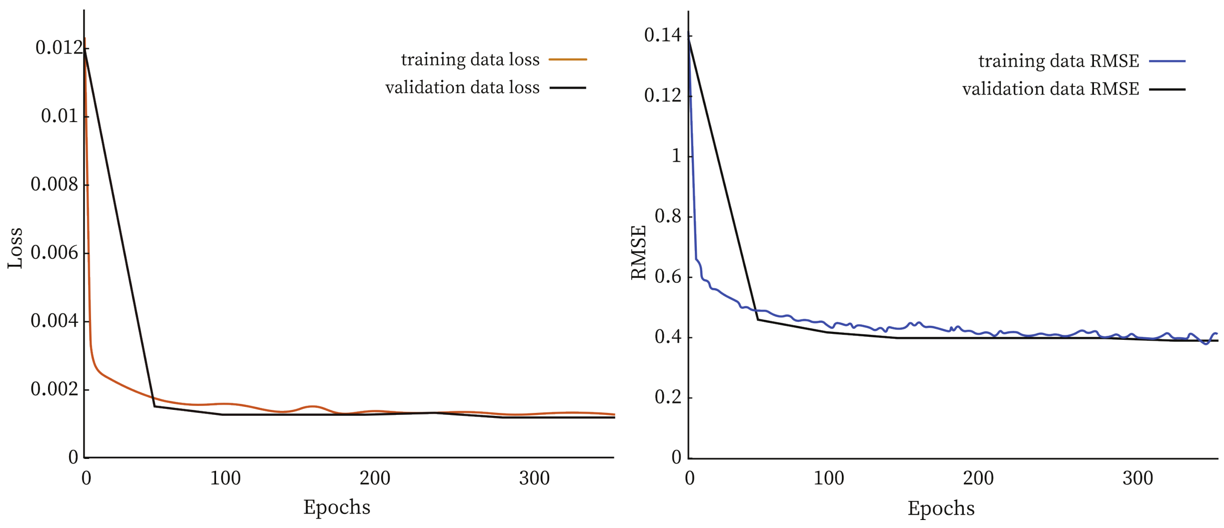

Figure 4.4: Training progress of sensors selector (multi-agent)

Testing the pre-trained Actor Network: Results are presented in Figure (4.5).

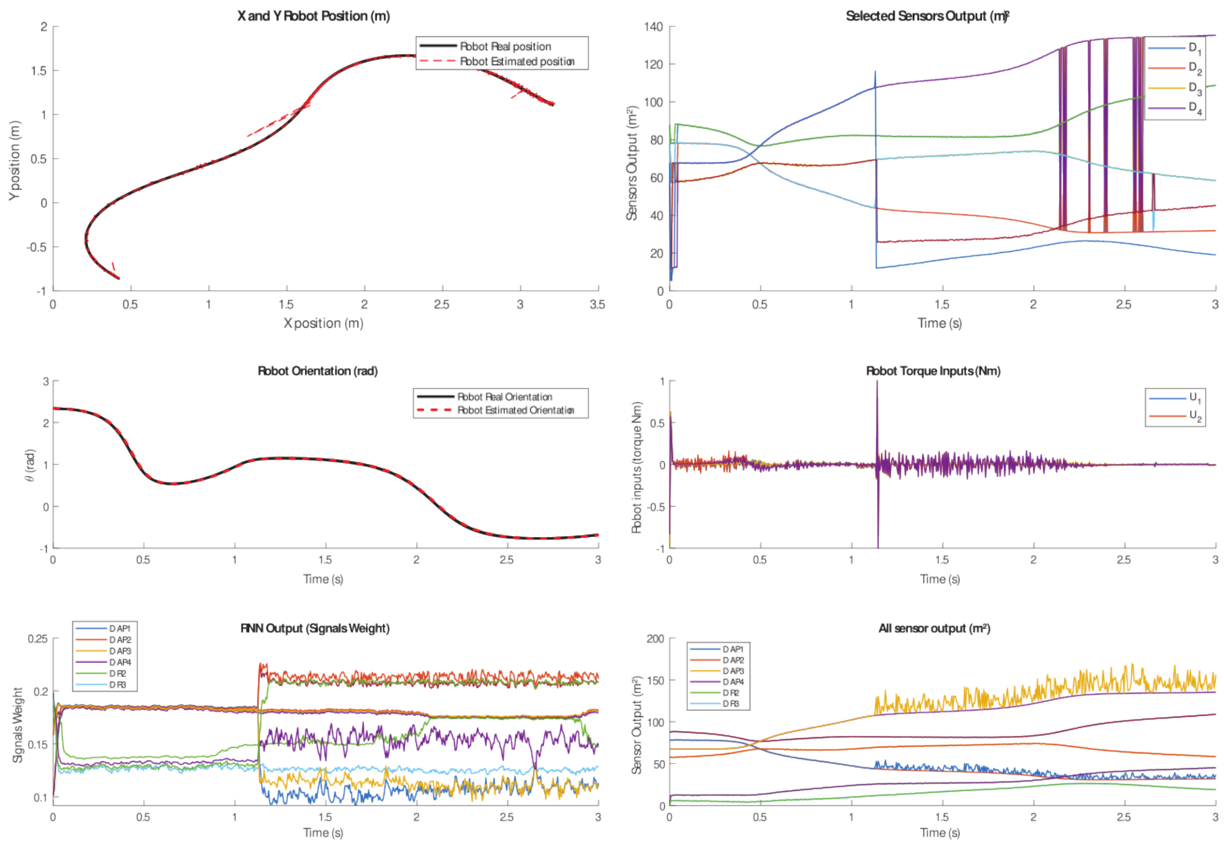

Figure 4.5: Testing the sensors selector (multi-agent)

Analyzing the results:

Test results presented in Figure (4.5) demonstrate the successful ability of this configuration to select the best signals. Although there is some initial and final fluctuation in finding the optimal selections, the structure effectively identifies the less noisy signals. It is worth noting that there is still room for further improvements in the performance of the system.

4.2.4 Collaborative estimation enhancement using RL

4.2.4.1 DDPG agent

Agent architecture: Similar to the single agent approach, we utilized the pre-trained actor network along with the critic network which is presented in Figure (4.6).

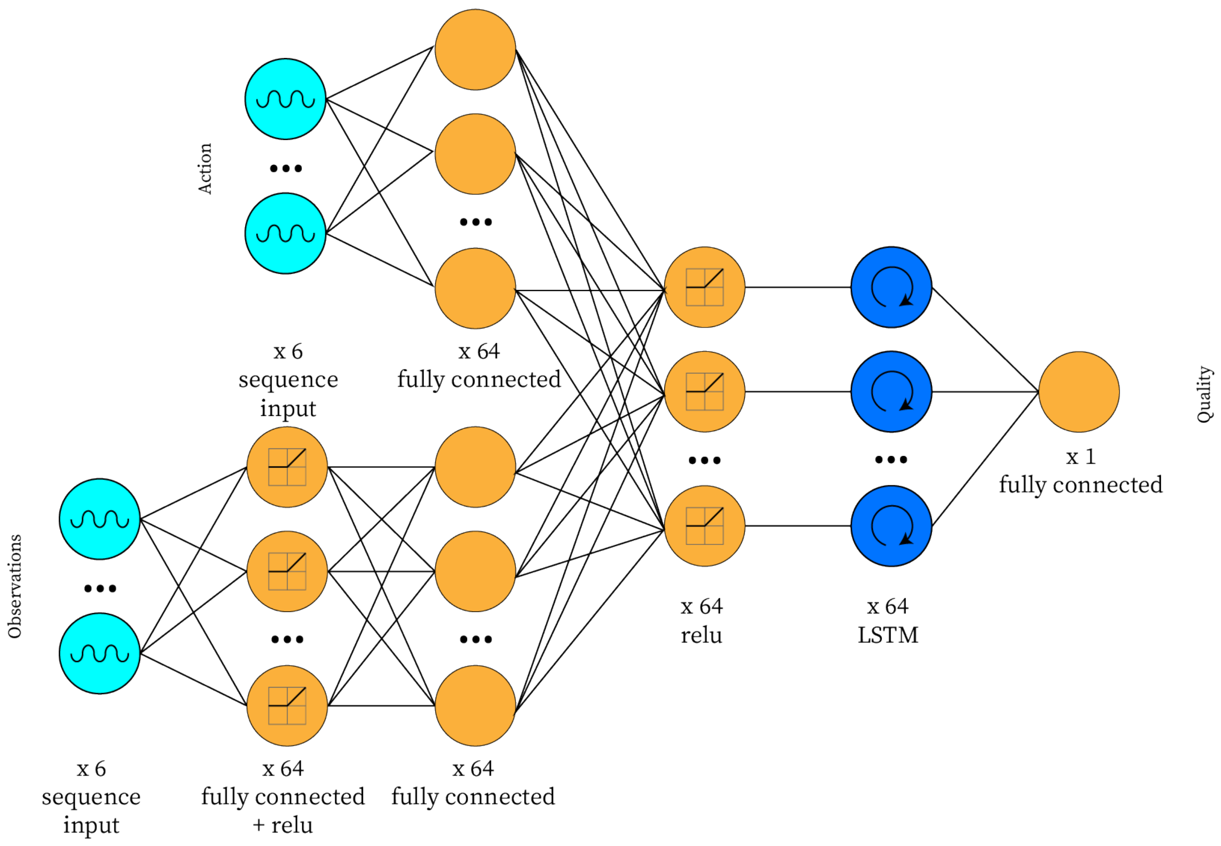

Figure 4.6: DDPG Critic Network architecture for multi-agent

DDPG agent training process: We trained the agent with these parameters:

For critic network

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	0.1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	0.1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Noise Variance	0.01
Variance Decay Rate	10^{-5}

DDPG agent training results: After training the agent for 400 episodes, the agent achieved an average reward of 110 over the last 100 episodes.

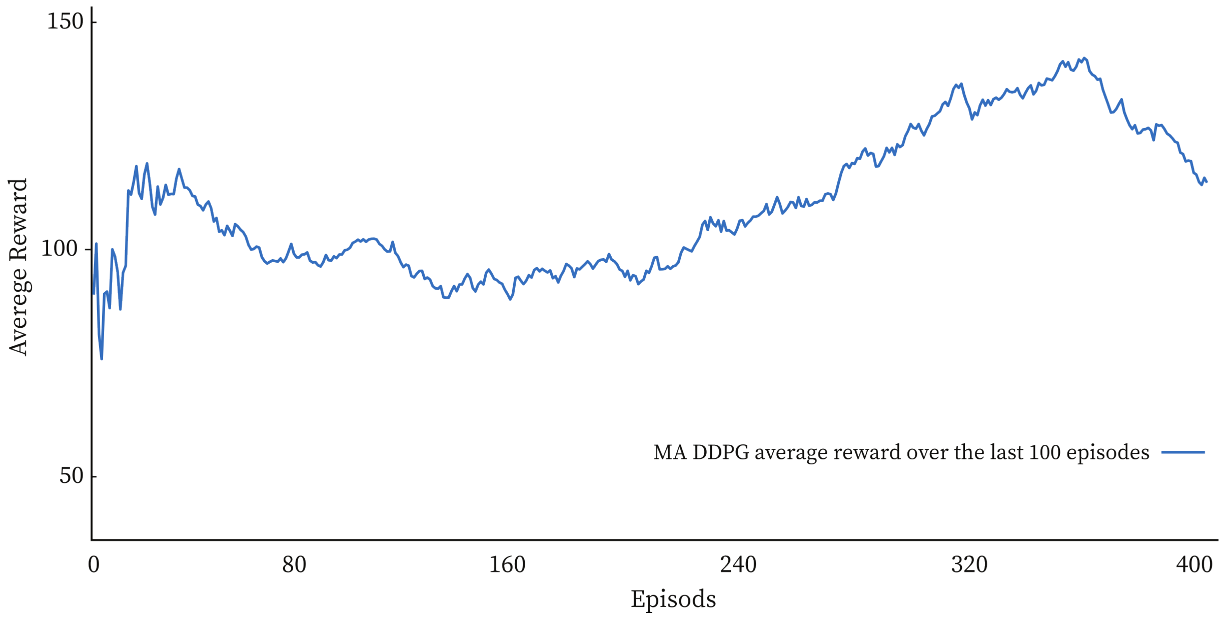

Figure 4.7: Training progress of the DDPG agent (multi-agent)

Testing the trained DDPG agent: Implementing and utilizing the optimized policy of the DDPG Agent resulted in the outcomes displayed in Figure (4.8).

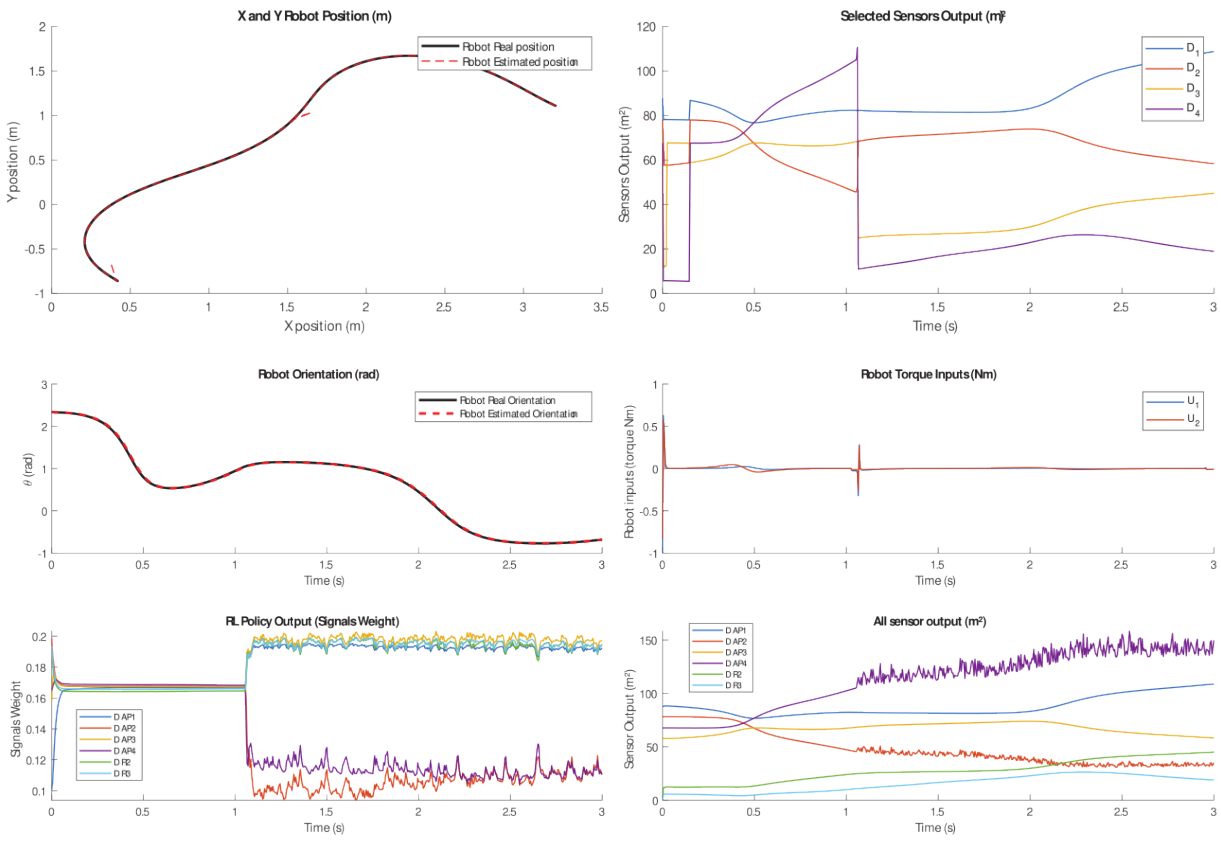

Figure 4.8: Testing the trained DDPG agent (multi-agent)

Analyzing the results:

Upon analyzing the training progress and the test results, it is evident that the agent did not demonstrate significant improvement. This could be attributed to the inherent characteristic of DDPG as a slow learner. However, these results suggest that further enhancements can be achieved by allowing the agent to train for more episodes. By extending the training duration, the agent may have the opportunity to acquire additional knowledge and refine its performance in the multi-agent system.

4.2.4.2 TD3 agent

Agent architecture: For the TD3 agent, we utilized the same actor and critic networks as the DDPG agent, with the only difference being the agent type.

TD3 agent training process: We trained the agent with these parameters :

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	32
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

TD3 agent training results: After 690 episodes of training, the TD3 agent achieved an average reward of 120 over the last 100 episodes.

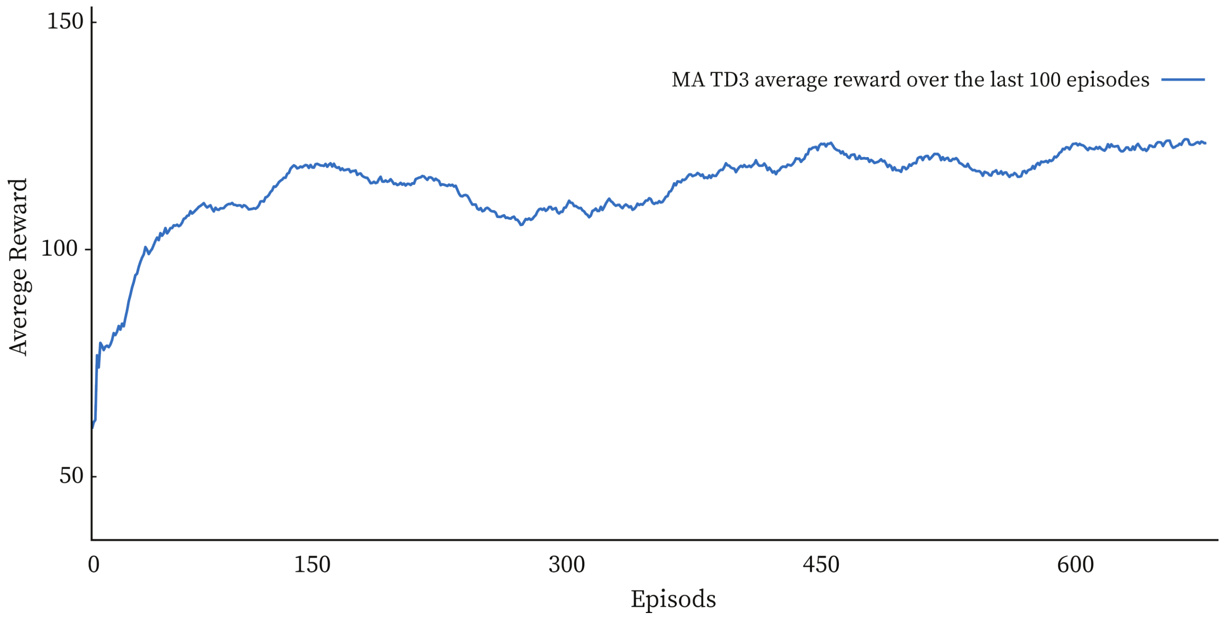

Figure 4.9: training progress of the TD3 agent (multi-agent)

Testing the trained TD3 agent: Implementing and utilizing the optimized policy of the TD3 agent resulted in the outcomes presented in Figure (4.10).

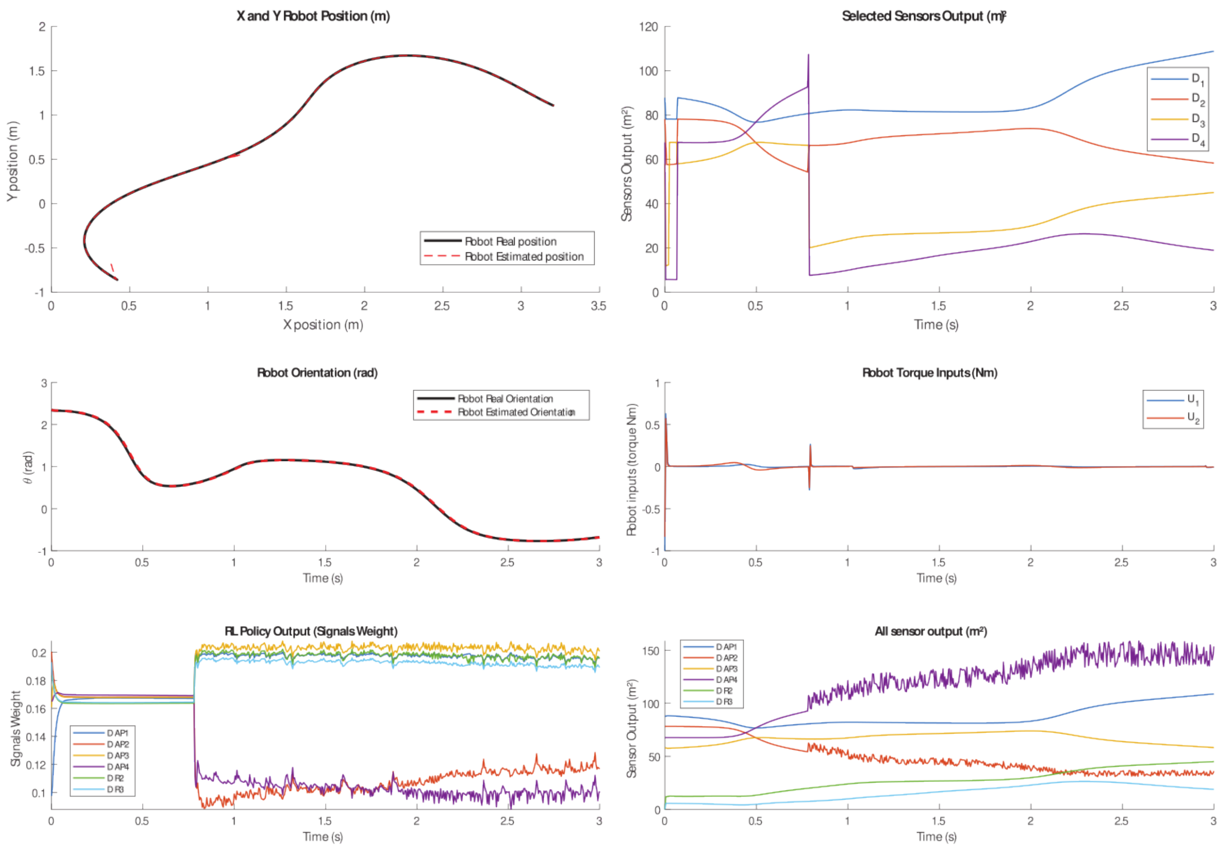

Figure 4.10: Testing the trained TD3 agent (multi-agent)

Analyzing the results:

Examining the training progress and the test results, it is evident that the TD3 agent outperforms the DDPG agent significantly. The TD3 agent demonstrates faster learning capabilities and exhibits greater stability in terms of its average reward. Moreover, the TD3 agent efficiently selects the best signals, exhibiting a smoother and more consistent performance throughout the test. These findings suggest that the TD3 agent is the most effective and reliable among the agents considered in this study.

4.2.4.3 SAC agent

SAC agent architecture: For the TD3 agent, we utilized the TD3 critic network along with a modified version of the trained actor network. The architecture of this new actor network is presented in Figure (4.11). By incorporating this modified actor network.

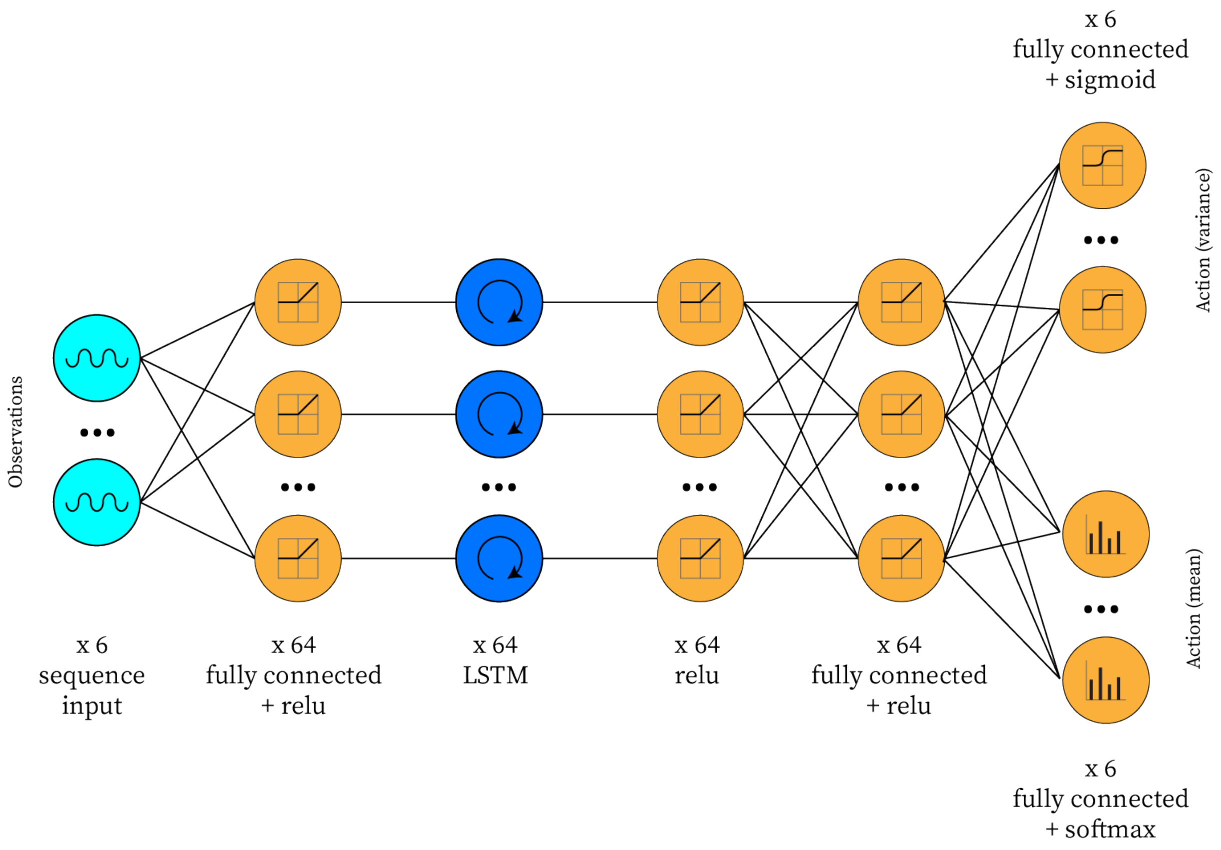

Figure 4.11: SAC Actor Network architecture for mobile robot multi-agent

SAC agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-6}
Gradient Threshold	0.01

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	16
Discount Factor	1

SAC agent training results: After 100 episodes, the average reward was 90.

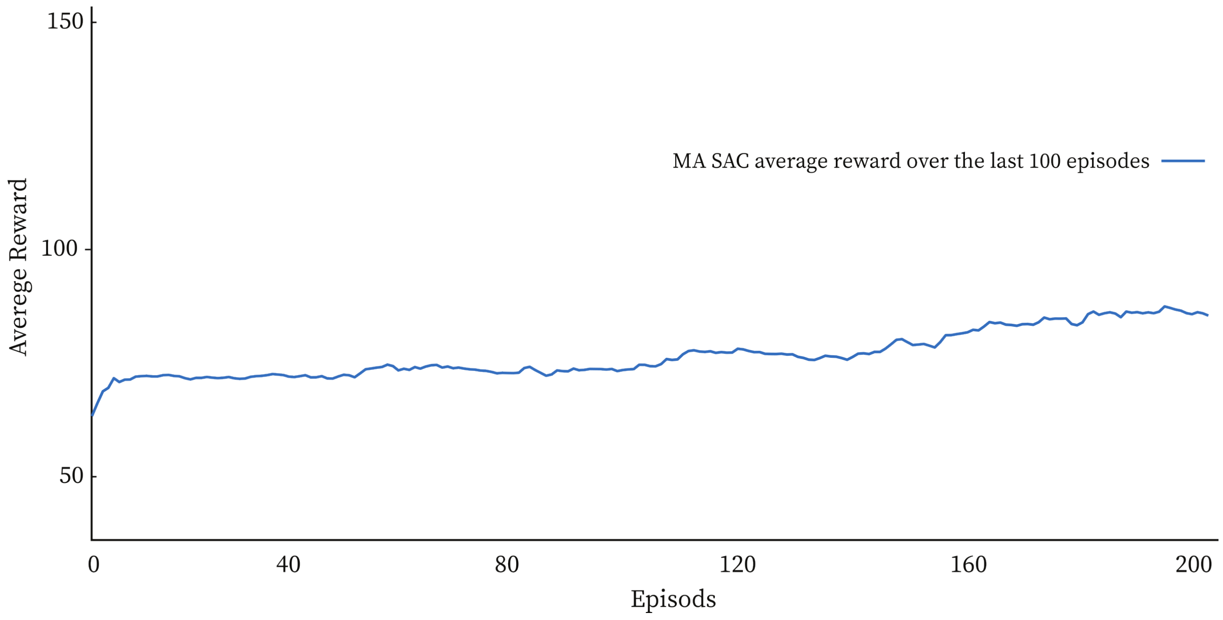

Figure 4.12: Training progress of the SAC agent (multi-agent)

Testing the trained SAC agent:

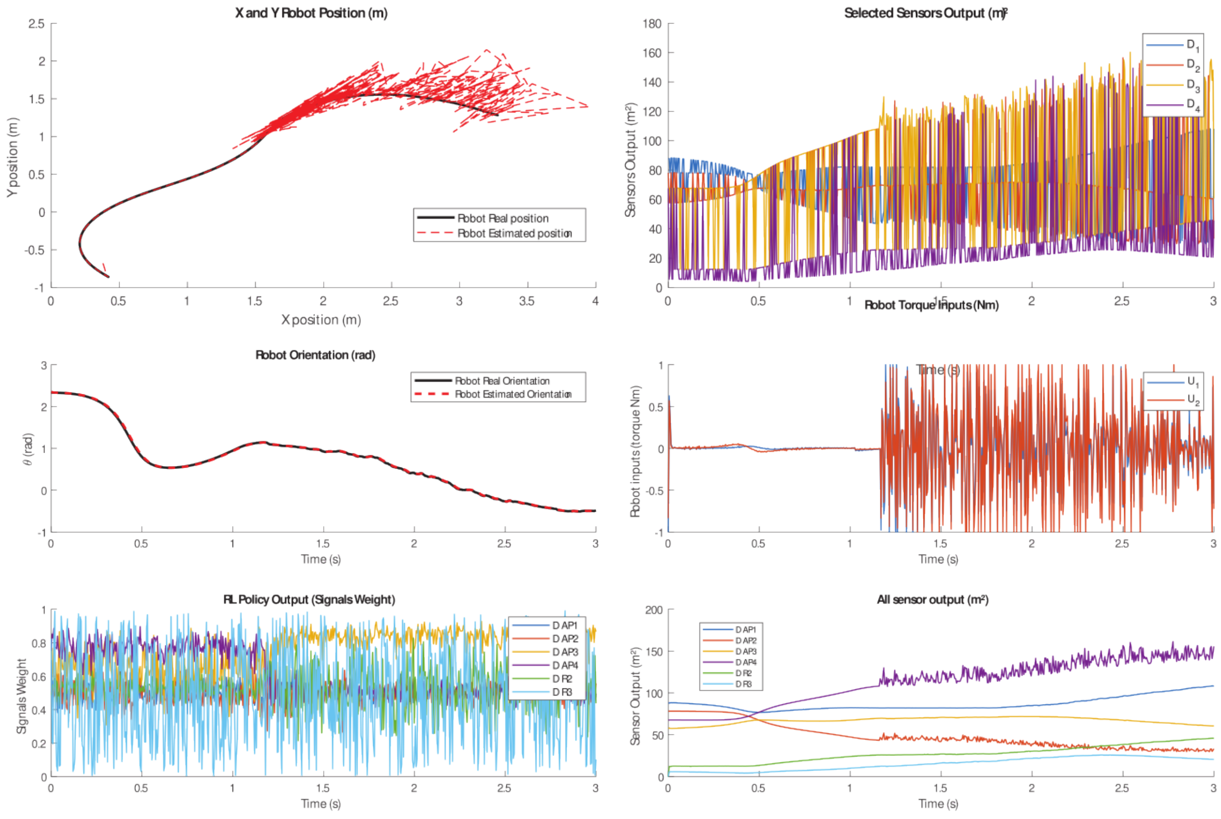

Figure 4.13: Testing the trained SAC agent (multi-agent)

Analyzing the results:

From the training progress and the test results, it appears that the SAC agent did not show significant improvement and even performed worse than the DL (Deep Learning) approach.

The SAC agent seemed to struggle in its decision-making process, exhibiting random behavior without demonstrating any learned knowledge or improvement. These observations suggest that the SAC agent may not be well-suited for the specific task or may require further optimization and fine-tuning to achieve better performance in the multi-agent system.

4.2.4.4 PPO agent

Agent architecture: The actor network for the PPO agent was identical to the one used in the SAC agent, as presented in Figure (4.11). Similarly, the critic network utilized the architecture outlined in Figure (4.14).

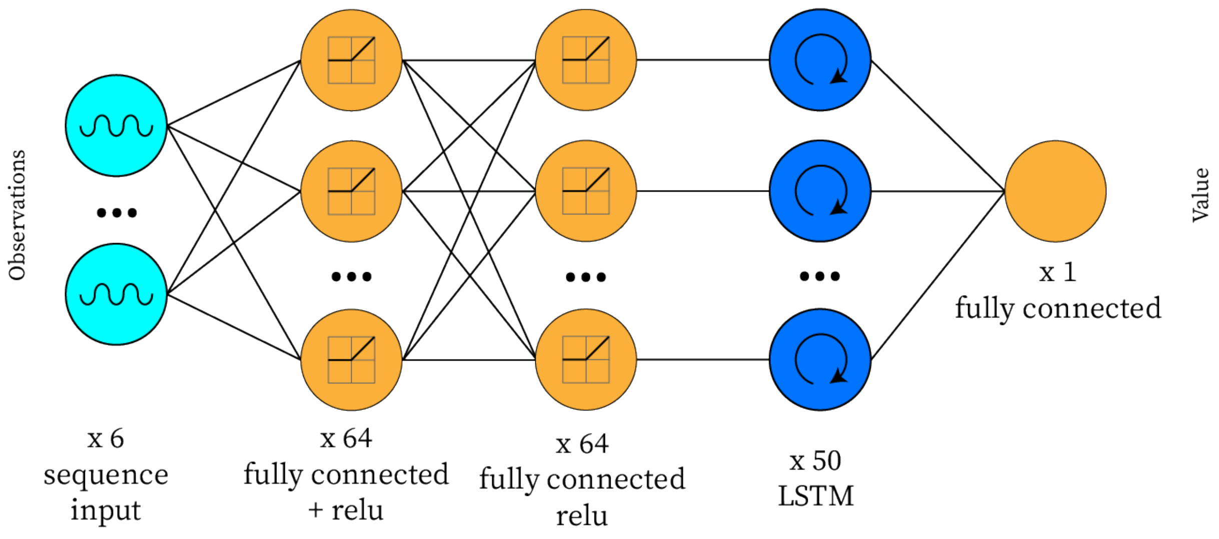

Figure 4.14: PPO Critic Network architecture for mobile robot multi-agent

PPO agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Horizon	600
Entropy Loss Weight	0.01
NumEpoch	3
Advantage Estimate Method	gae
Discount Factor	1

PPO agent training results: In 1000 episodes, the average reward was 112.

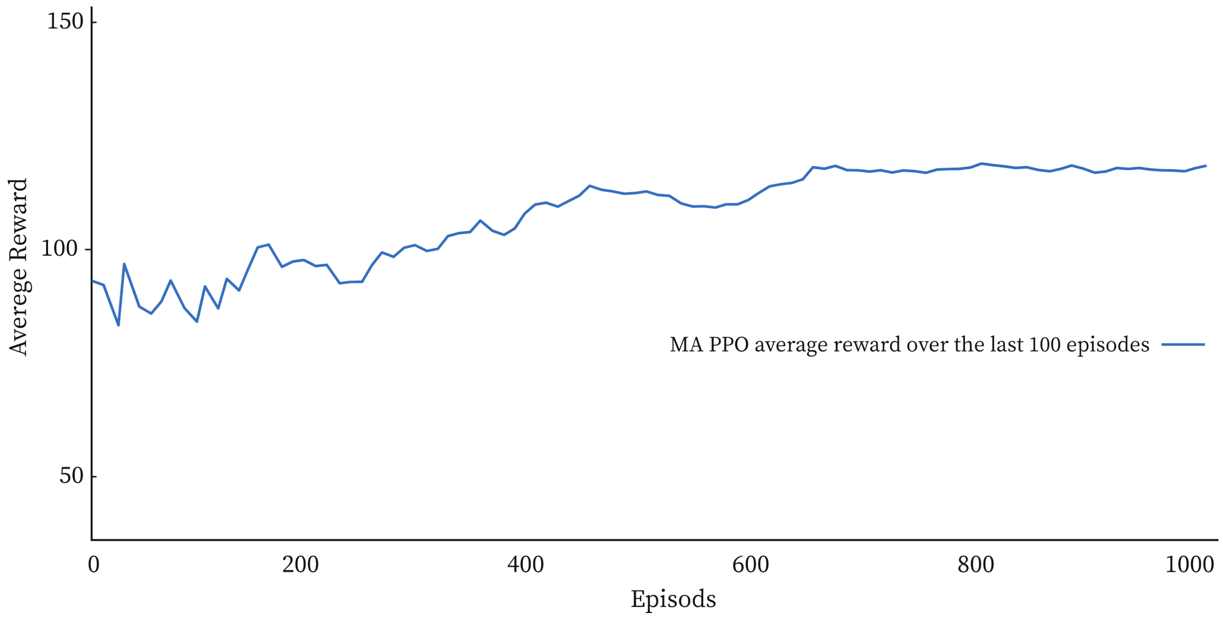

Figure 4.15: Training progress of the PPO agent (multi-agent)

Testing the trained PPO agent:

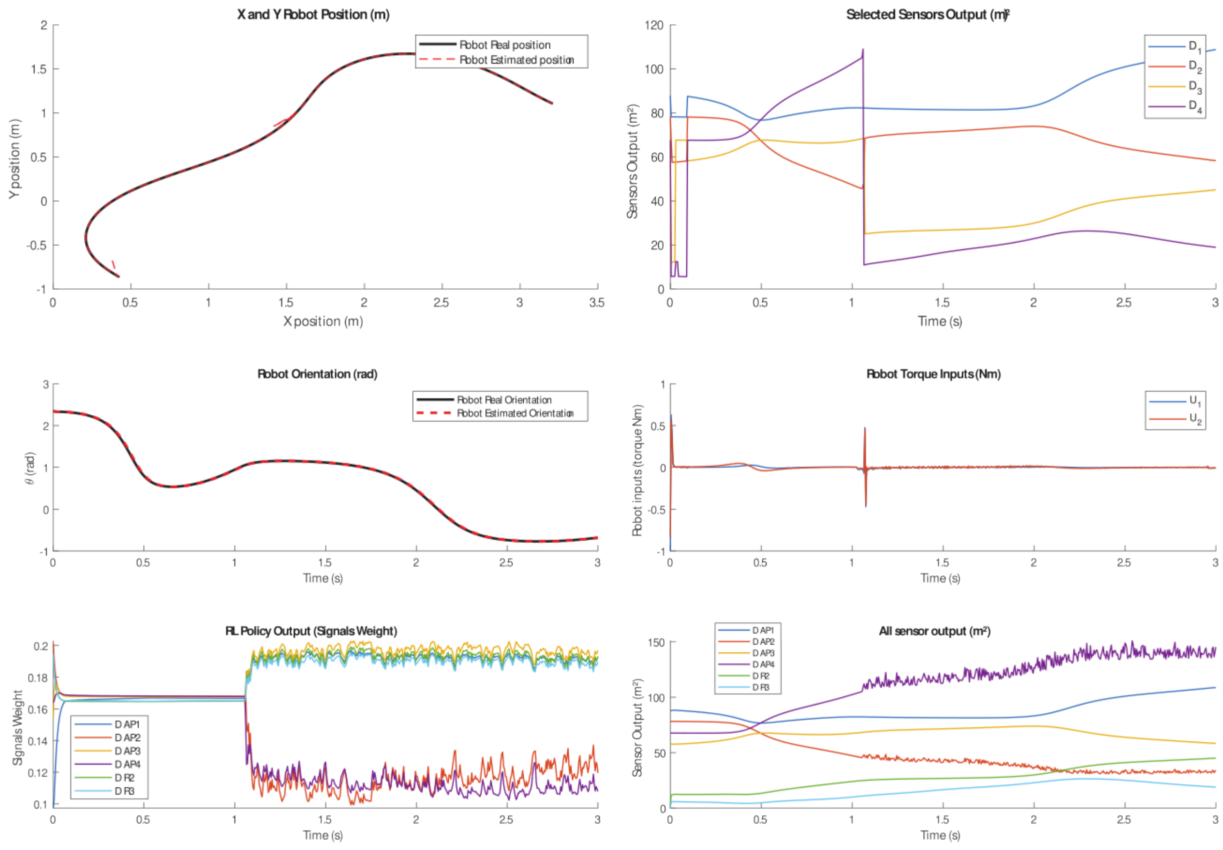

Figure 4.16: Testing the trained PPO agent (multi-agent)

Analyzing the results:

Analyzing the training progress and the test results, it is observed that after 1000 episodes, the

PPO agent's performance improved slightly, but it remained comparable to the DL approach. Despite training for a significant number of episodes, the agent did not exhibit substantial progress or outperform the DL approach.

4.2.5 Performance comparison

To compare the performances of the agents, a total of 100 simulations were conducted under various conditions. The sum of the square root error was calculated for each robot individually using the formula:

$$\sum_{k=1}^{3} \int_{t_0}^{t_f} \sqrt{(X^k - \hat{X}^k)^2 + (Y^k - \hat{Y}^k)^2 + (\theta^k - \hat{\theta}^k)^2}\, dt \tag{4.2}$$

where X^k, Y^k and θ^k are the real pose for the robot k, while $\hat{X}^k, \hat{Y}^k$ and $\hat{\theta}^k$ are the estimated pose for the robot k.

The errors for each robot were then summed, and the average error was computed over the 100 simulations. The results of this analysis are presented in Figure (4.17), allowing for a comparison of the performances of the different agents.

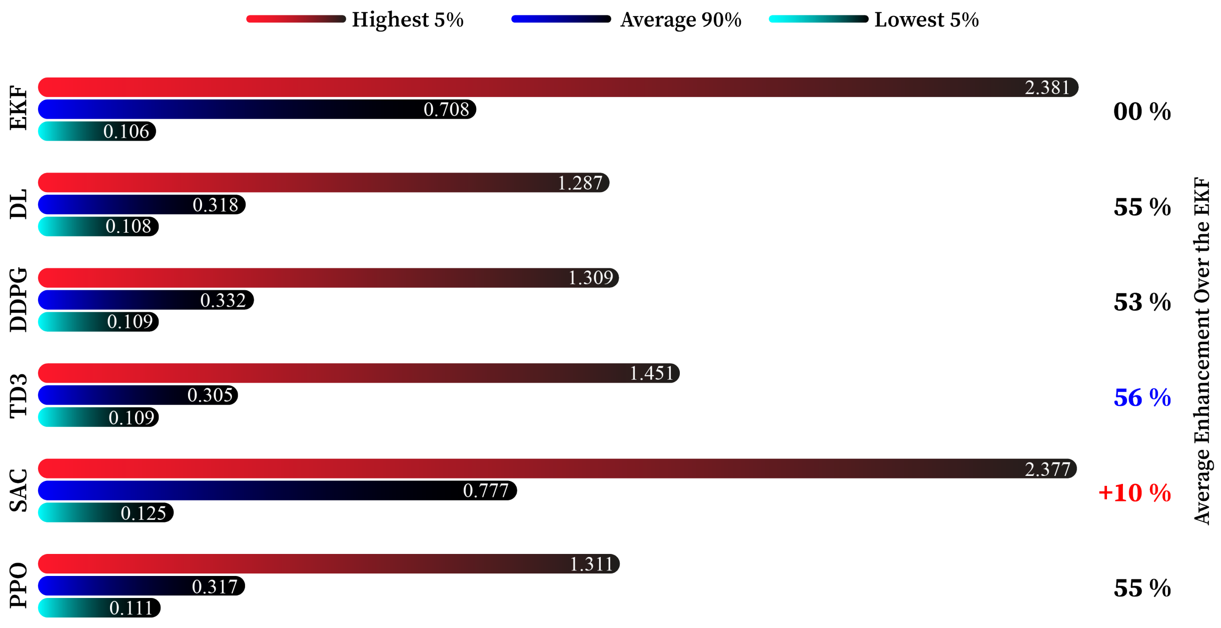

Figure 4.17: Comparison of the average error of 100 tests of the standard EKF, DL-EKF, DDPG Agent, TD3 Agent, SAC Agent and PPO Agent for the multi agent system

4.2.6 Results discussion

Comparing the performance of the agents, the classic EKF showed an average error of 0.708 in 100 tests. Implementing the trained RNN with the pre-trained agent resulted in a significant improvement, with an average error of 0.318, which is a 55% improvement over the classic EKF.

Of the RL agents, the SAC agent performed the worst, with an average error of 0.777, which is 10% worse than the EKF. The DDPG agent achieved an average error of 0.332, a 53% improvement over the EKF. The PPO agent performed similarly to the pre-trained actor, with an average error of 0.317, also a 55% improvement over the EKF.

The TD3 agent showed the best performance among the agents, with an average error of 0.305, representing an impressive 56% improvement over the classic EKF.

Focusing on the lowest 5% and the highest 5% of the performance, all agents delivered similar results under ideal conditions (lowest 5%), with a maximum error of 0.125 recorded by the SAC agent. However, for the highest 5%, it is evident that all agents except the SAC agent outperformed the EKF by over 50%, demonstrating superior performance even in the worst-case scenarios for the RL agents.

4.3 Pose estimation for aerial robot multi-agent system

This section presents the proposed approach for collaborative pose estimation in a multi-agent system composed of multiple quadrotors. The objective is to improve the accuracy and robustness of pose estimation by leveraging the information from neighboring quadrotors. The approach is based on the exchange of relative pose measurements.

4.3.1 Proposed approach

To enable collaborative pose estimation in a multi-agent system composed of multiple quadrotors, one approach is by distributed estimation. Each quadrotor can independently run its own extended Kalman filter (EKF) to estimate its pose based on IMU sensor measurements. To incorporate collaborative pose estimation, we enable neighboring quadrotor to exchange there relative pose measurements using UWB sensors. By fusing these relative measurements into the EKF, each quadrotor can enhance its own pose estimation accuracy and benefit from the collective information.

The proposed approach is based on the following steps:

Independent EKF Estimation: Each quadrotor runs its own EKF to estimate its pose based on IMU sensor measurements. The EKF is a recursive estimation algorithm that combines prediction and measurement updates to estimate the state of a system with uncertainty.

Relative Pose Measurements: Quadrotors exchange relative pose measurements with neighbors to enhance their individual pose estimation accuracy. These measurements can be obtained using onboard sensors like cameras, lidars, or distance sensors. Relative pose measurements provide information about the positions and orientations of other quadrotors relative to each quadrotor's own frame of reference.

Data Fusion: Each quadrotor incorporates the received relative pose measurements into its own EKF by fusing them with its local measurements. This fusion can be achieved by modifying the EKF's state and measurement update equations to include the relative pose measurements.

Weighting and Covariance Update: When fusing relative pose measurements, it's crucial to consider the reliability of each measurement. We assign weights to the measurements based on their accuracy, confidence, or distance to the quadrotor. The weights influence the contribution of each measurement to the EKF's estimation process. Additionally, the covariance matrix of the EKF should be updated to reflect the increased information from the relative pose measurements.

Cooperative Decision-Making: The distributed estimation process enables the quadrotors to collaboratively estimate their poses. This collaborative information can be utilized for cooperative decision-making, such as collision avoidance, formation flying, or distributed task allocation.

4.3.2 Implementation

Following the approach described above, we will implement a EKF-based pose estimation algorithm for multi-agent system with multiple quadrotors.

The exchange of information between quadrotor enables each one to have access to the estimated positions of all other robots in the system. Consequently, the measurement vector of the pose estimation process is no longer limited to the individual quadrotor's IMU sensor measurements but also includes the estimated positions of other quadrotor as well using distances. This expanded measurement vector is represented by Equation (4.3).

$$Y_i = h_i(X_i) = \begin{cases} p_i \\ q_i \\ r_i \\ d_{i1}^2 \\ d_{i2}^2 \\ \vdots \\ d_{in}^2 \end{cases} \tag{4.3}$$

Where p_i, q_i and r_i are IMU's angular rates measurements for agent i, d_{ij}^2 are square distances between agent i and its neighbors agents j given by:

$$d_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 \tag{4.4}$$

However, employing this approach poses certain challenges, particularly in large swarm systems. As the number of robots in the multi-agent system increases, scalability becomes an issue, leading to potential inefficiencies in computation and communication. The system size grows significantly, resulting in increased computational and communication overheads. Additionally, even in smaller multi-agent system sizes, the accuracy of relative pose estimation can be compromised due to the presence of incorrect or unreliable data. Factors such as inaccurate pose estimation, faulty distance measurements caused by sensor failures, obstacles or interference can all have a detrimental effect on the accuracy of relative pose estimation.

To address these challenges, a potential solution is to employ a filtering mechanism to select the best signals to use which can be achieved by passing the sensor data through a Recurrent Neural Network classifier. In our case, we select best 3 distances in the case of a multi-agent system composed of more than 3 quadrotors.

Figure (4.18) illustrates a multi-agent system of quadrotors and the selection mechanism.

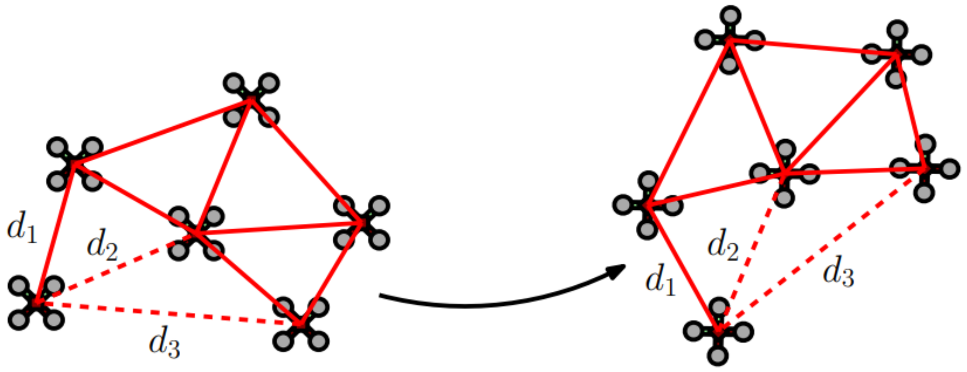

Figure 4.18: Selection mechanism in multi-agent system of quadrotors [31]

The simplfied measurement vector for each agent i is given then by:

$$Y_i = h_i(X_i) = \begin{cases} p_i \\ q_i \\ r_i \\ d_{i1}^2 \\ d_{i2}^2 \\ d_{i3}^2 \end{cases} \tag{4.5}$$

4.3.3 Environment setup

Observation space: The size of the observation space is related to the number of quadrotors n in the multi-agent system, resulting in a size of $n \times 1$. In this application, we used 4 quadrotors with 2 fixed points, resulting in an observation space size of 5×1, consisting of the measurement error $e_Y = Y_k - \hat{Y}_k$ and the variation in the predicted state $X_k - X_{k-1}$.

Action space: The size of the action space is also $n \times 1$. In this case, it is 5×1. Each output in the action space represents the probability of selecting a specific distance over the others.

Reward: We provided the agent with a reward of $+1/6$ for each correct classification of a signal.

4.3.4 Pre-training the Actor Network

Similar to the single-agent approach, we will also pre-train the actor network to speed up the learning process.

Actor Network architecture: Since the problem involves signal processing and classification, we employed an RNN with a softmax output. The architecture of the RNN used for this multi-agent application is presented in Figure (4.19).

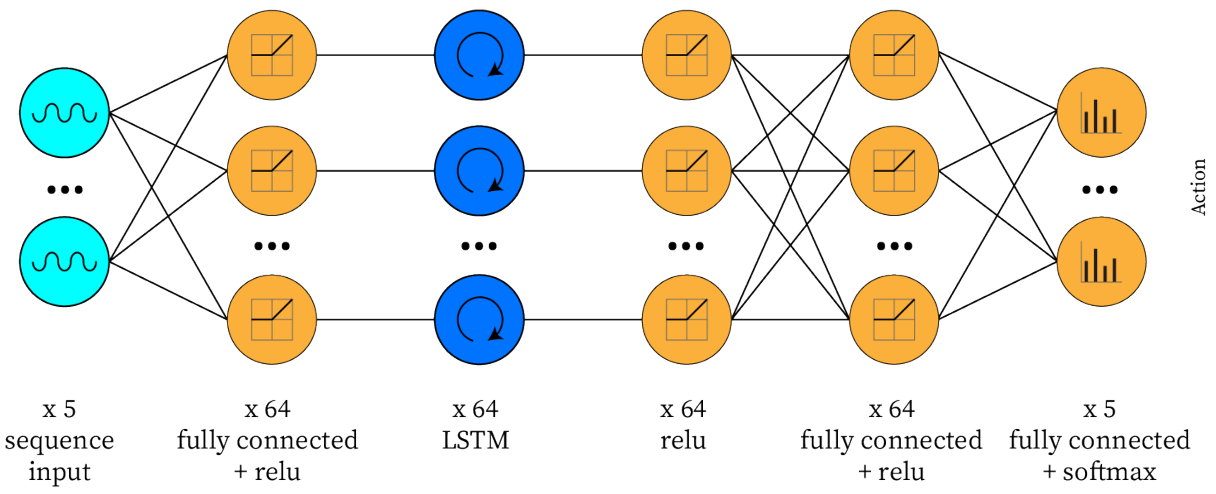

Figure 4.19: Actor Network architecture for aerial robot multi-agent

Data generation: To generate the data, we assigned weights to each distance measurement based on the noise power and offset. A lower weight indicates a better quality measurement. We then applied a softmax function to obtain more unified data. Finally, we saved the weighted measurements along with the corresponding inputs (observations).

Actor Network training: We trained the actor network using the following settings:

Optimizer	Adam	Mini-batch size	128
Initial learning rate	0.01	Shuffle	every-epoch
Learning rate drop factor	0.9	Sequence length	longest
Max epochs	100	Validation frequency	5

DL training results: After 45 epochs of training, a loss of 0.08 was achieved for both the training and validation data. The RMSE for the training data was 0.38, while for the validation data, it was 0.4. Figure (4.20) provides a visual representation of the training progress.

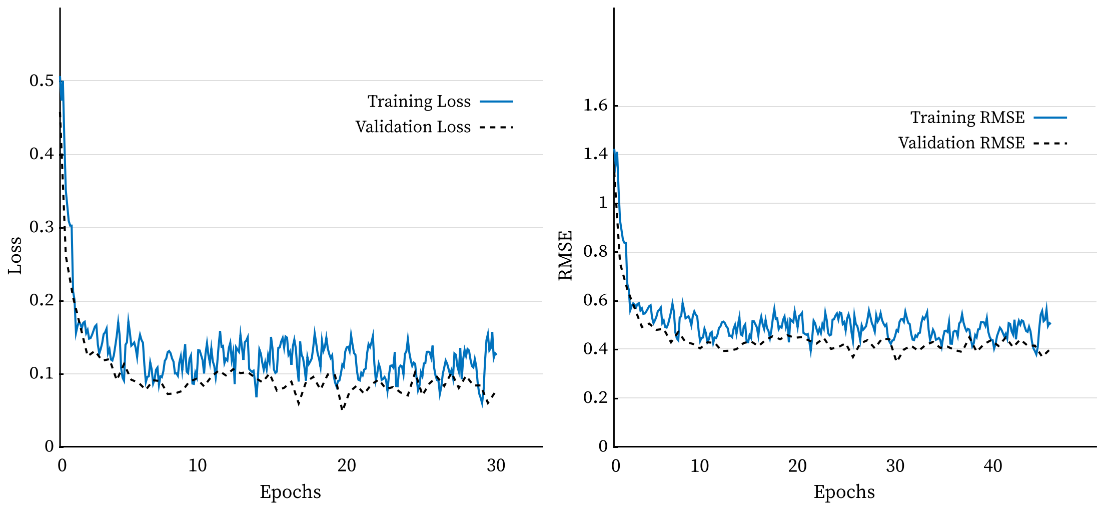

Figure 4.20: Training progress of sensors selector in aerial robot multi-agent

4.3.5 Collaborative estimation enhancement using RL

Now, we will try to improve the collaborative estimation process by employing the TD3 RL agent which has demonstrated good performance in the mobile robot application.

TD3 agent architecture: Similar to the previous application, we used the pre-trained actor network along with the critic network which is presented in Figure (4.21).

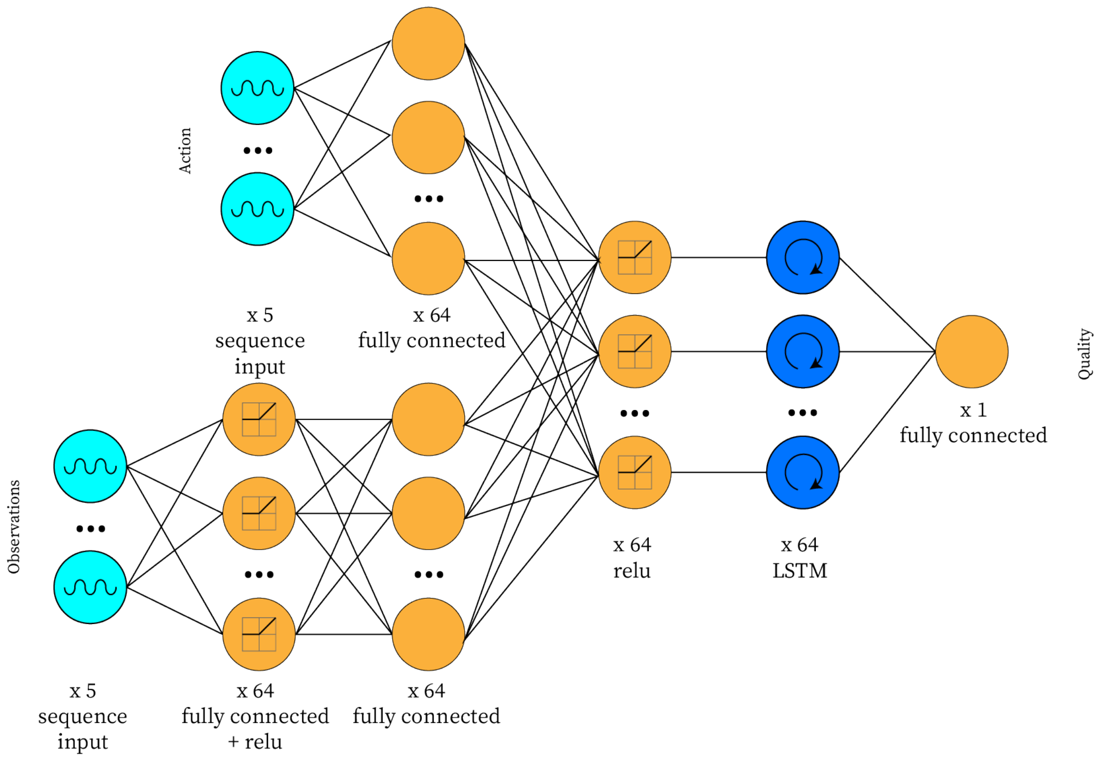

Figure 4.21: TD3 Critic Network architecture for aerial robot multi-agent system

TD3 agent training process: We trained the agent with these parameters:

For critic networks

Optimizer	Adam
Learning rate	0.001
Gradient Threshold	1

For actor network

Optimizer	Adam
Learning rate	10^{-7}
Gradient Threshold	1

Agent parameters

Sample time	0.005
Experience Buffer Length	1000
Sequence Length	10
Mini-batch size	32
Exploration noise variance	0.01
Variance Decay Rate	10^{-5}

TD3 agent training results: After 500 episodes of training, the TD3 agent achieved an average reward of xxx over the last 100 episodes.

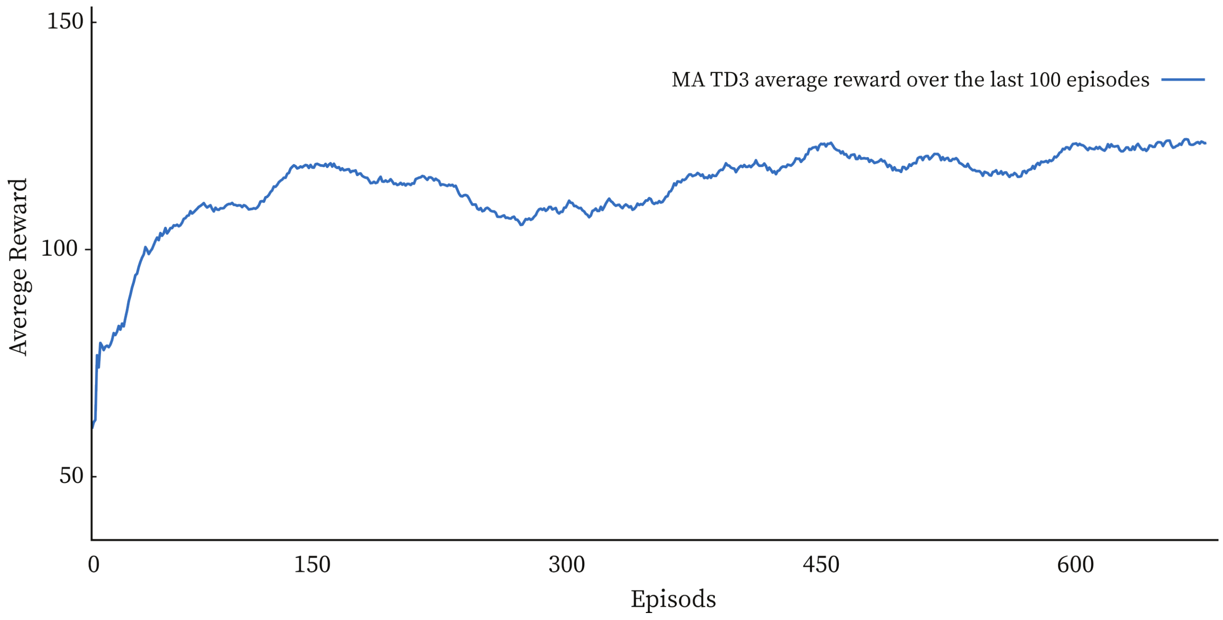

Figure 4.22: training progress of the TD3 agent for aerial robot multi-agent

Testing the trained TD3 agent: Results are presented in Figure (4.23).

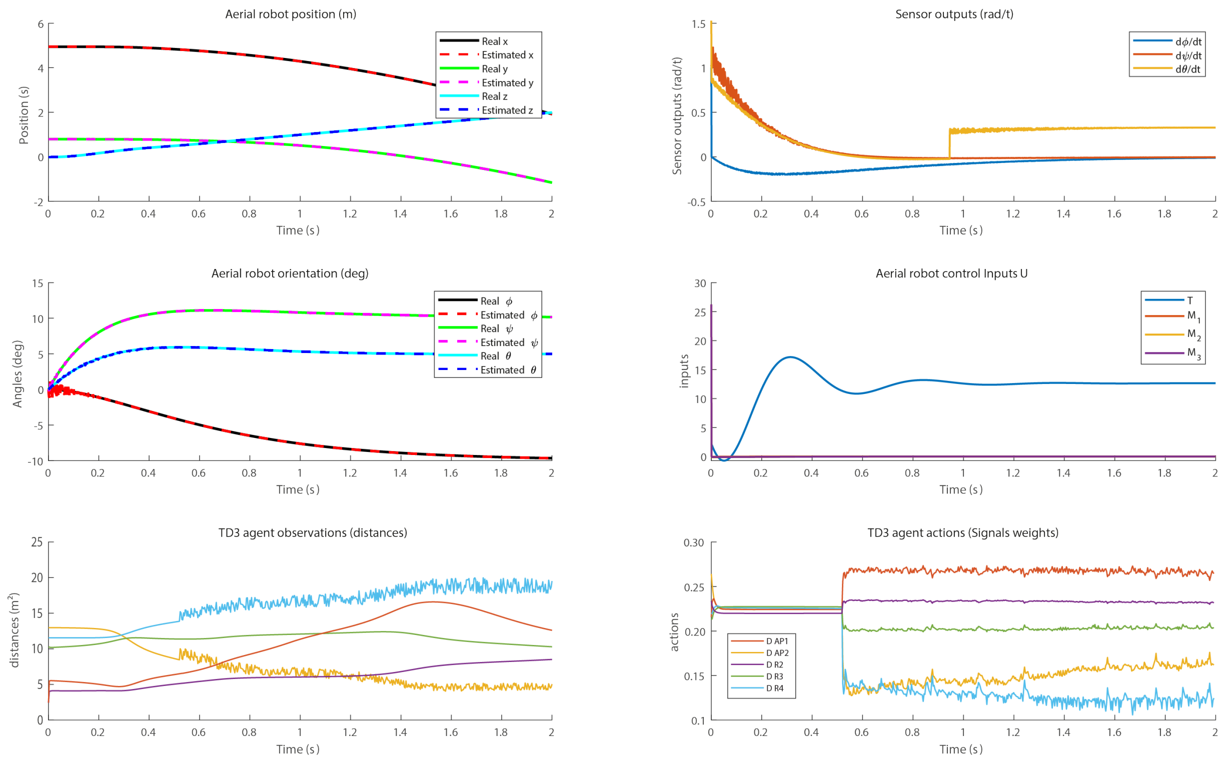

Figure 4.23: Testing the trained TD3 agent for aerial robot multi-agent

4.3.6 Results discussion

The test results presented in Figure (4.23) demonstrate the successful ability of this configuration to select the best signals. The TD3 agent demonstrates faster learning capabilities and present greater stability in terms of its average reward. Moreover, the TD3 agent efficiently selects the best signals for a smoother and more consistent performance throughout the test.

4.4 Conclusion

The test results of this approach demonstrate its potential for enhancing pose estimation in multi-agent systems, as it achieved a significant improvement over the standard EKF. This approach offers improved performance without adding additional complexity to the model.

It is worth noting that the comparison between DL and RL agents did not show a noticeable improvement in performance. This can be attributed to the nature of the problem, which is primarily a classification task. The DL approach, which focuses on selecting the right signals from the available options, already performed well and left limited room for further improvement.

In contrast, the SAC agent's performance was poor due to its random exploration strategy. Without a well-defined plan or strategy, the agent struggled to make effective decisions and achieve good results.

The results highlight the effectiveness of the proposed approach in enhancing pose estimation in multi-agent robotic systems for both mobile and aerial robots while also show the importance of selecting appropriate strategies for the given problem.

Chapter 5

Simulation in Gazebo

5.1 Introduction

In this chapter, we will demonstrate the practical application of our findings in a real-world scenario. The primary objective is to test the effectiveness of our solution as a component of an actual system and identify the challenges and obstacles it must overcome. To accomplish this, we will leverage the Robot Operating System (ROS) with its 3D simulation environment, Gazebo. This software combination offers us the opportunity to thoroughly test, experiment, observe outcomes, and troubleshoot within an environment that closely resembles reality. Furthermore, the code developed in ROS during this simulation will be highly adaptable for implementation in a physical robot, considering that ROS is extensively utilized as an operating system for real robots, so coupling it with Gazebo creates an optimal environment to evaluate our proposed solutions.

Tests will be executed using the ROS 1 noitic and Gazebo 11 software. Initially, we will proceed with designing the simulation environment, commonly referred to as the simulation world. Our approach involves creating an indoor-like world with strategically placed walls intended to obstruct sensor signals in specific areas. To obtain robot orientation information, we utilized the built-in odometry sensors. However, for distance measurements, we encountered a limitation as the software lacked predefined UWB sensors. Consequently, we create our custom sensors by leveraging the high-precision built-in odometry sensors on the robot. This enabled us to obtain the robot's exact position, which was subsequently utilized to calculate the required distances. We incorporated noises and offsets into the results, taking into account the robot's position, in order to emulate real-world measurements.

5.2 Robot model

One of the advantages of utilizing ROS and Gazebo in this test is that the simulation of the robot does not rely on mathematical models or state space representations. Instead, the robot is represented in its 3D form, with each part is represented as a link containing information such as mass, inertia, and material properties. These links are connected using joints, each has its own properties, including friction, maximum force, and limits.

Figure (5.1) displays the 3D model of the robot employed in the test. The model was created using Blender software and exported in URDF (Universal Robot Description Format), which is written in XML language using the Blender plugin 'Phobos'.

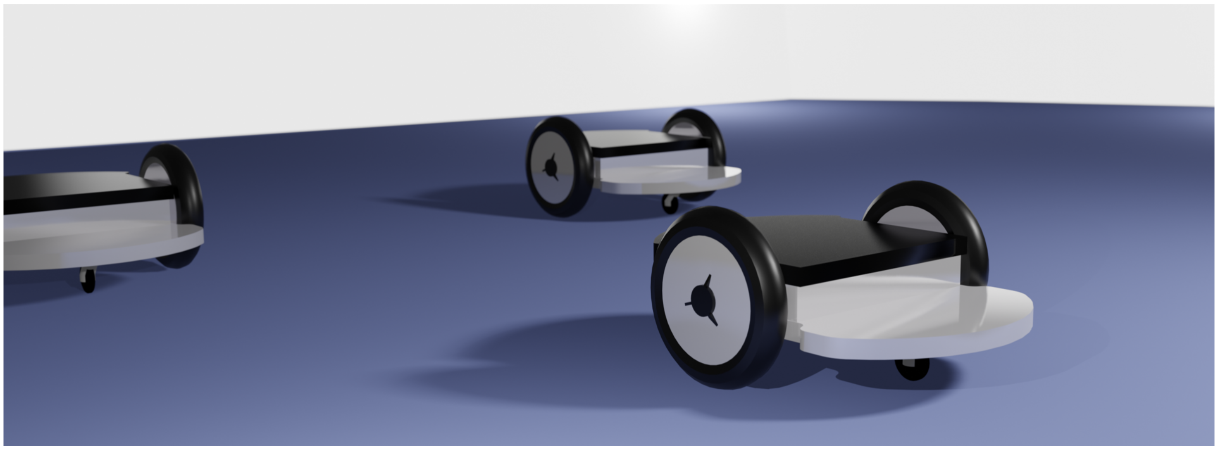

Figure 5.1: 3D model of the robot used in the Gazebo simulation

To control the robot, we used the Gazebo plugin, which only requires the command velocity and automatically adjusts the wheel movements to achieve the desired speed. We set the maximum torque to 5, and other parameters such as friction and force align with the properties defined for the joints.

The model used in the EKF is even simpler compared to the one used for training. It tracks only the X and Y position and the angle θ, without incorporating the dynamics of angular and linear velocities. It directly accepts the references, assuming a transfer function of 1. This simulation setup provides a valuable testing environment, as the robot is represented in high detail while the EKF utilizes a smaller, less detailed model.

5.3 Single agent simulation in Gazebo

5.3.1 Single agent simulation structure

Figure (5.2) illustrates the RQT graph of the single agent simulation setup, depicting the relationships between various nodes represented by ellipses, and the topics represented by rectangles. Each node corresponds to a Python code segment responsible for data processing and control operations, while the topics serve as the communication interface between these nodes.

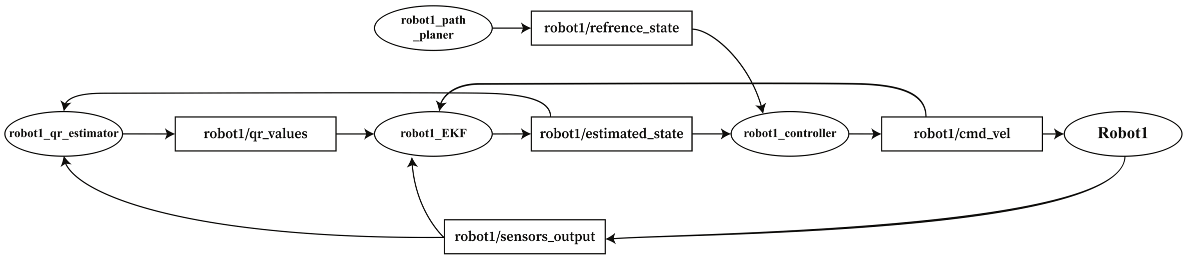

Figure 5.2: RQT graph of the simulation setup for single agent

The 'Robot1' node represents the dynamics of the first robot within the Gazebo software. It receives its commands from the 'robot1/cmd_vel' topic, which provides the reference linear and angular velocity. The node then outputs the sensor measurements to the 'robot1/sensors_output' topic. It is important to note that we only provide the robot with the reference speed, and the built-in controller is responsible for achieving it. Therefore, we do not have direct knowledge of the actual dynamics of the robot.

The 'robot1_sensors_selector' node takes the sensor outputs and forwards them to the sensors selector RL policy, trained in the previous section. After selecting the most suitable sensors to work with, the measurements from the selected sensors are published to the 'robot1/selected_sensors' topic, while the position of the anchor points is published to the 'robot1/AP_location' topic.

The 'robot1_qr_estimator' node takes the estimated robot state, anchor point locations, and selected sensor measurements to calculate the inputs (e_y and ΔX) for the QR estimator. One of the initial challenges we encountered was exporting the trained agent from MATLAB to use it in ROS. To accomplish this, we first extracted the RNN from the TD3 agent policy (QRestimator) and then utilized a specific command called 'exportNetworkToTensorFlow' to enable its usage in Python scripts, the programming language employed in the nodes. The resulting Q and R parameters are then published to the 'robot1/qr_values' topic.

Next, the 'robot1_EKF' node uses the Q and R parameters, along with the sensor measurements and the command to execute the EKF algorithm. This estimation process aims to determine the estimated state of the robot, which is subsequently published to the 'robot1/estimated_state' topic.

The 'robot1_controller' node receives the estimated state, along with the reference state published by the path planner node in 'robot1/reference_state'. Using this information, the controller calculates the command velocity (the reference linear and angular velocity) and publishes it to the 'robot1/cmd_vel' topic, thus completing the control loop.

5.3.2 Single agent simulation results

We conducted the test in the indoor environment depicted in Figure (5.3). Initially, we performed a test without any reference (or with the reference set equal to the initial state) to verify the estimator's ability to converge towards the real state. Subsequently, in the second test, we provided the robot with a constant command to observe whether the estimated state accurately tracked the robot's actual state. In the final test for the single agent, we closed the control loop by enabling the robot to autonomously follow a reference trajectory based on its estimated position.

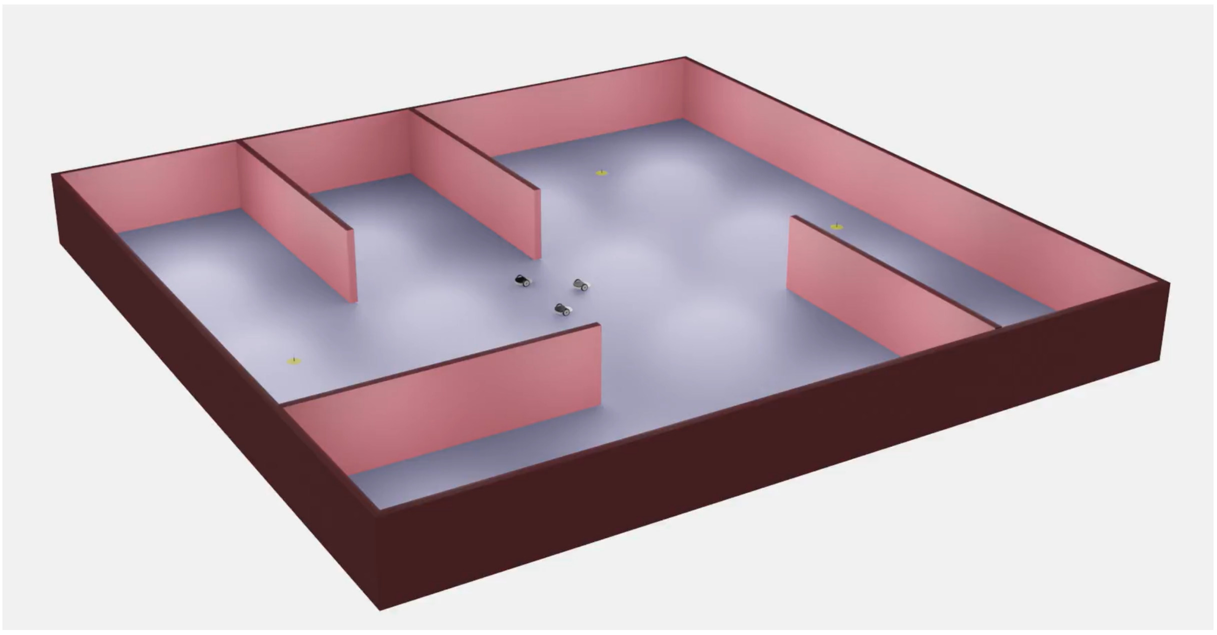

Figure 5.3: Gazebo simulation world

As an ultimate test, we utilized every component of the system. The robots were initialized at positions $(-7.5, 3, -\pi/2)$, $(-7.5, 4.5, -\pi/2)$, and $(-7.5, 6, -\pi/2)$ for robots 1, 2, and 3, respectively. The robots were then directed towards end positions of $(7, 6, 0)$, $(7, 3, 0)$, and $(7, 0, 0)$ following a specific path. This path took the robots through a long trajectory where the quality of the sensor measurements varied based on the robot's position.The fixed anchor points were positioned at $(-7.5, 0)$, $(0, -7.5)$, $(8, 0)$, and $(5, 8)$. Several screen captures of the simulation are presented in Figure(5.4). While, the tragectory of the robot with the references and the estimated state are presented in Figure (5.5). The Q and R parameters outputted from qr_estimator node are presented in Figure (5.6) and the sensors output in Figure (5.7)

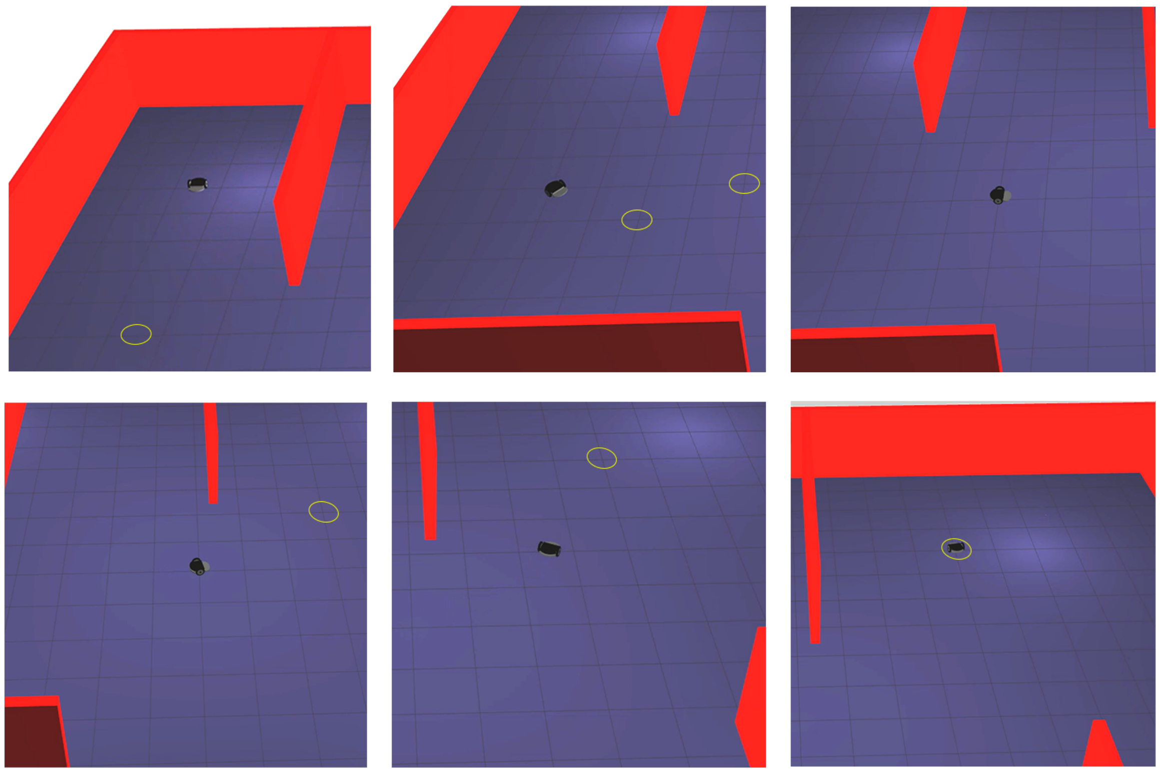

Figure 5.4: Screen captures of the simulation for single robot in Gazebo

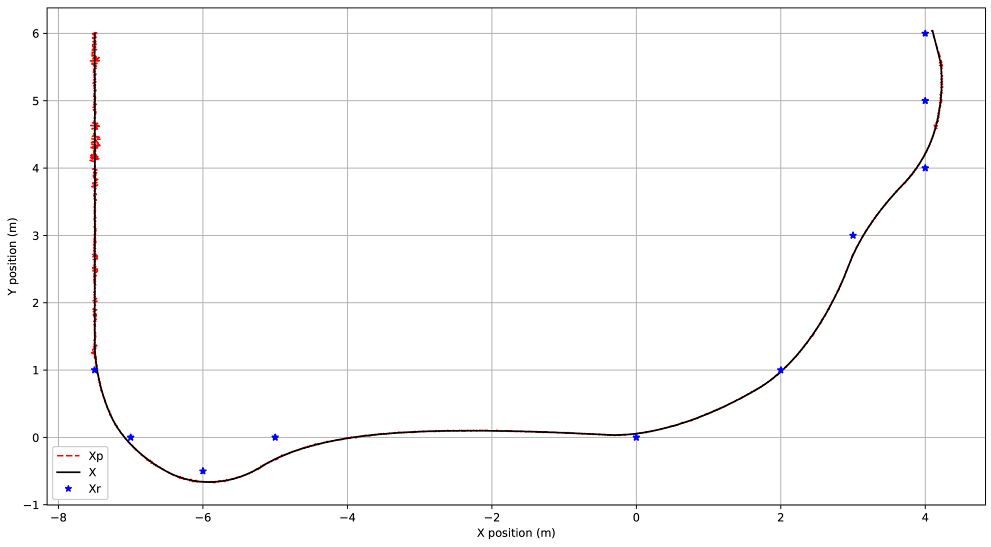

Figure 5.5: Robot Path of the Single Robot in the Gazebo Simulation

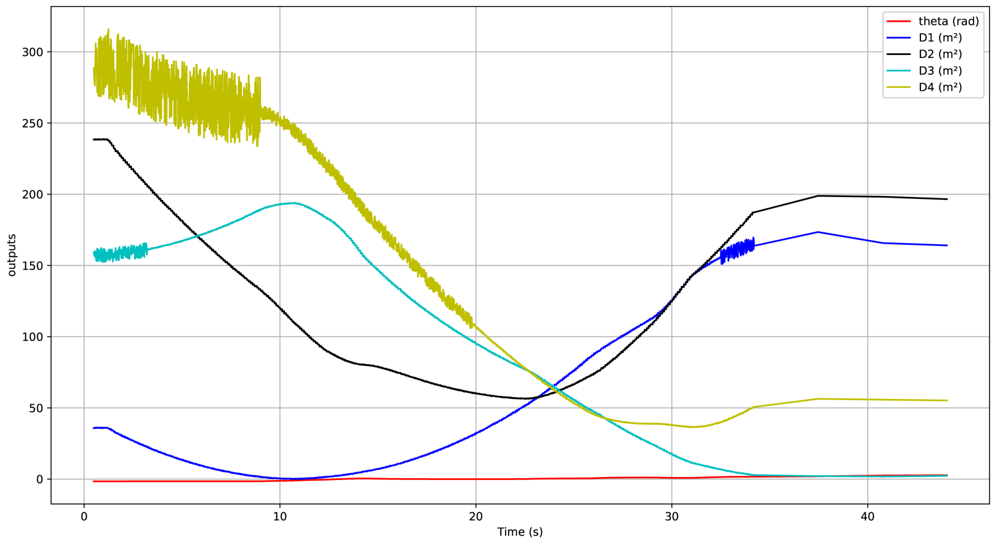

Figure 5.6: Sensors output of the single robot in the Gazebo simulation

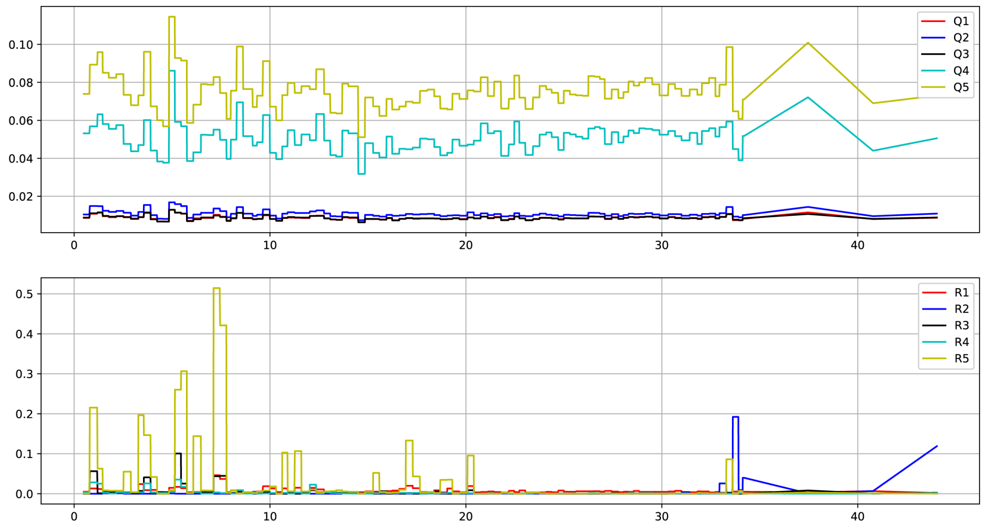

Figure 5.7: QR parameters used by the robot in the Gazebo simulation

5.4 Multi-agent simulation in Gazebo

5.4.1 Multi-agent simulation structure

Figure (5.8) illustrates the RQT graph of the setup, depicting the relationships between various nodes represented by ellipses, and the topics represented by rectangles. Each node corresponds to a Python code segment responsible for data processing and control operations, while the topics serve as the communication interface between these nodes.

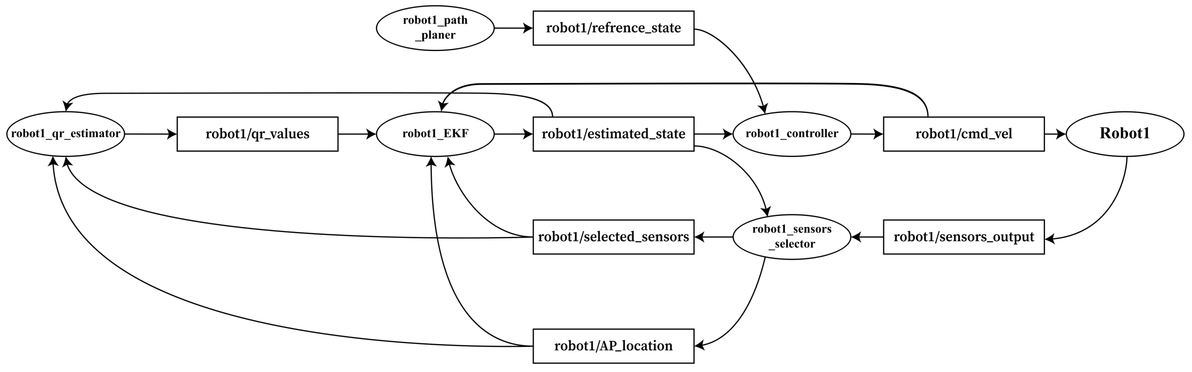

Figure 5.8: RQT graph of the simulation setup for an Agent in a Multi Agent System

The 'Robot1' node represents is the same as before, the only deference is that this time robot1 is one of multipule robots in the system and the sensor measurements outputted to the 'robot1/sensors_output' topic, are now not only the distances to to the fixed anchor points but also to the other robots.

The 'robot1_sensors_selector' node takes the sensor outputs and forwards them to the sensors selector RL policy, trained in the previous section, exported from matlab in the same way as the QRestimator. After selecting the most suitable sensors to work with, the measurements from the selected sensors are published to the 'robot1/selected_sensors' topic, while the position of the anchor points is published to the 'robot1/AP_location' topic.

The 'robot1_qr_estimator' and the 'robot1_EKF' node are the same as for the single robot.The only deference is that the are now taking also take the anchor point position from 'robot1/AP_location' topic. While the 'robot1_controller' node is exactly the same as before

5.4.2 Multi-agent simulation results

Like for the single agent, we performed a static test (without any reference) to verify the estimator's ability to converge towards the real state. Then, we provided the robot with a constant command, after that we closed the control loop.

Like with the single robot, we did an ultimate test. The robots were initialized at positions $(-7.5, 3, -\pi/2)$, $(-7.5, 4.5, -\pi/2)$ and $(-7.5, 6, -\pi/2)$ for robots 1, 2, and 3, respectively. The robots were then directed towards end positions of $(7, 6, 0)$, $(7, 3, 0)$ and $(7, 0, 0)$ following a specific path. This path took the robots through a long trajectory where the quality of the sensor measurements varied based on the robot's position.The fixed anchor points were positioned at $(-7.5, 0)$, $(0, -7.5)$, $(8, 0)$ and $(5, 8)$. Several screen captures of the simulation are presented in Figure(5.9). While, the trajectory of the robot with the references and the estimated state are presented in Figure (5.10). The Q and R parameters outputted from qr_estimator node are presented in Figure (5.11) and the sensors output in Figure (5.12)

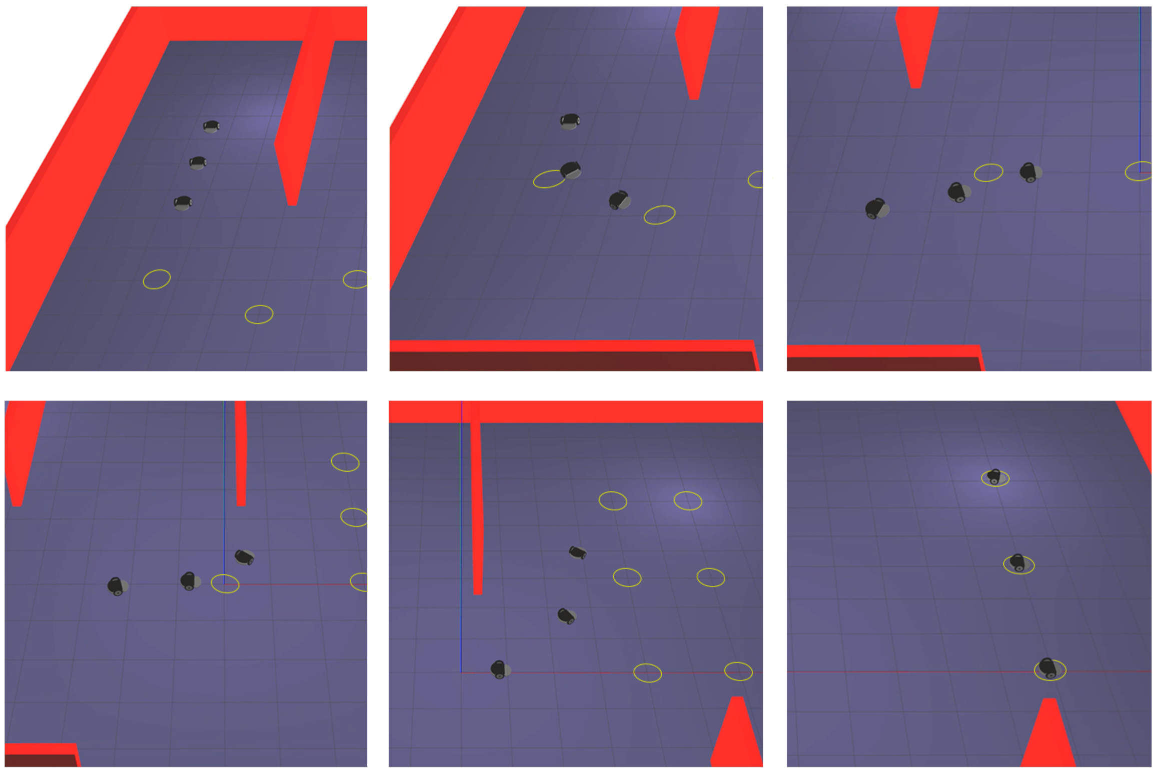

Figure 5.9: Screen captures of the multi-robots simulation in Gazebo

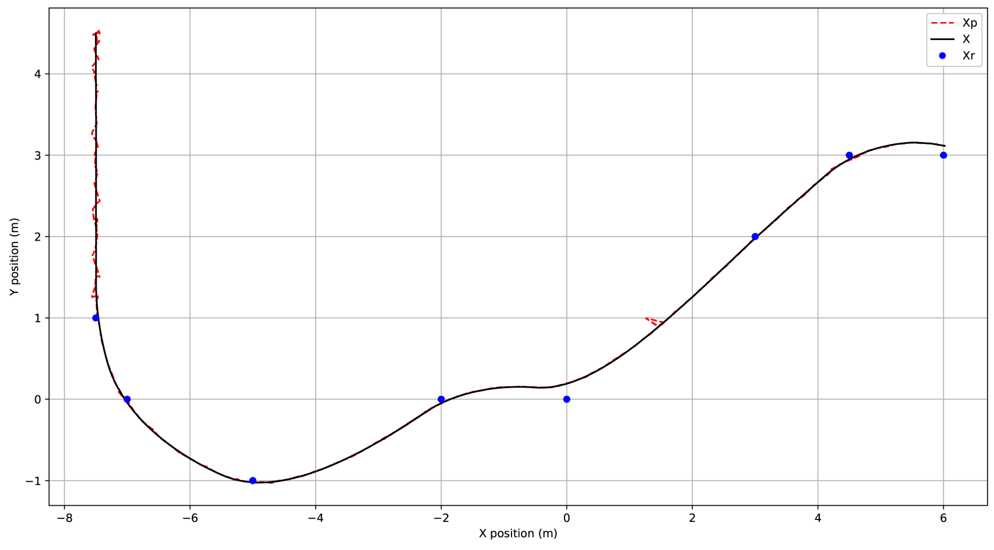

Figure 5.10: Robot 2 path in the Gazebo simulation for the multi-robots system

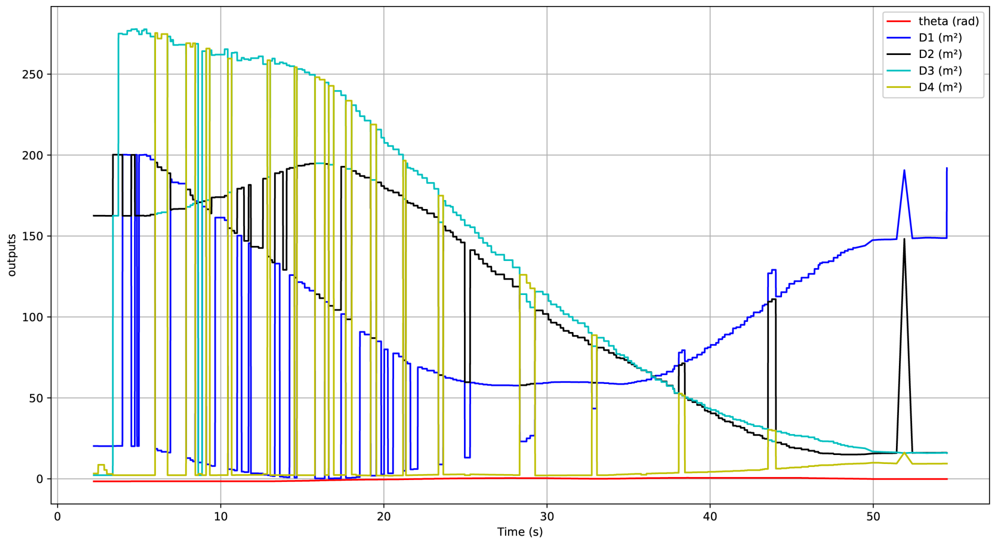

Figure 5.11: Sensors output of the robot 2 in the Gazebo simulation for the multi-robots system

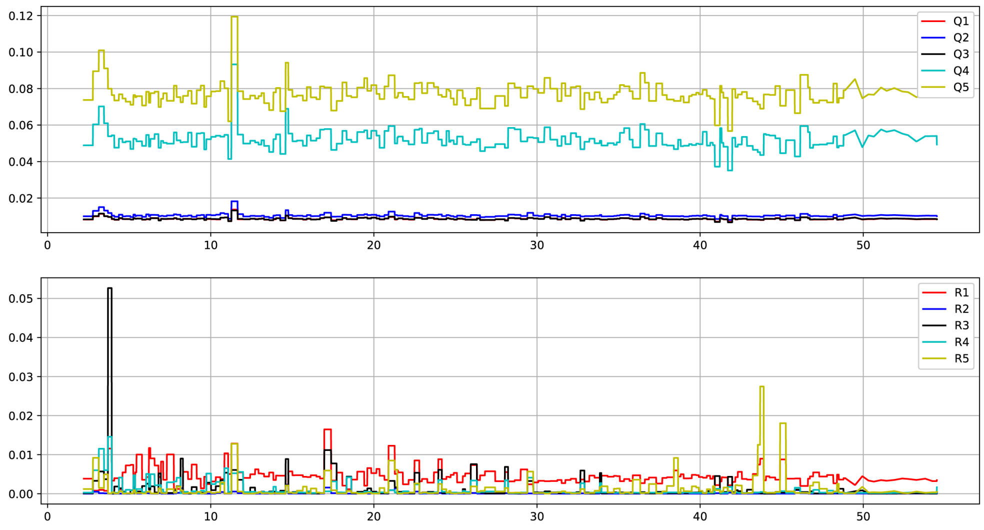

Figure 5.12: QR parameters used by the robot 2 in the Gazebo simulation for the multi-robots system

5.5 Results discussion

During these tests, we observed that the robot successfully completed the task, and this achievement was only possible because all nodes functioned correctly. We watched the QR-estimator adapting the Q and R parameters, and the sensor selector adjusting the selected measurements based on their quality. Remarkably, this success was attained despite the implementation of an inaccurate model in the EKF.

However, this significant accomplishment did come with some challenges worth mentioning. Firstly, running two RNNs for each robot posed a considerable challenge for micro-controllers, particularly when operating at high frequencies. Fortunately, a simple solution was available by reducing the frequency of the QR estimator and sensor selector nodes. In this test, the robot controllers and EKF estimator nodes were set to 100 Hz, while the QR estimator operated at 3 Hz and the sensor selector at only 1 Hz.

Another issue was founded during testing is when the estimated state initially converged to the wrong position. This occurred because the sensor selector selected the other two robots as anchor points, whose estimated states were still incorrect at the beginning. Since we were working now with only two fixed anchor points, there were two positions that satisfied the measurements. Consequently, the state estimator could converge to the wrong position. Fortunately, the solution was straightforward. We forced the state estimator to initially work with the fixed anchor points and employed the sensor selector only after 5 seconds or once the state estimator had converged. This problem could also be avoided by improved initialization of the robot's initial state.

5.6 Conclusion

These tests were conducted with the aim of testing the feasibility of implementing our solutions in real robots and identifying and resolving any challenges that may arise during the implementation process. The results of these tests were highly encouraging, as they demonstrated that our solution is well-suited for implementation.

Despite encountering a few minor issues, the implementation process proceeded smoothly. These challenges, such as the computational load of running multiple RNNs and the initial convergence of the estimated state to the wrong position, were easily overcome through simple adjustments and improvements.

The outcomes of both the single agent and multi-agent systems were remarkable. In both cases, the estimated state closely matched the real state of the robot. This level of accuracy and consistency indicates that our solution is nearly ready for implementation in real robots.

The successful completion of these tests signifies a major milestone in our development process. It provides strong evidence that our solution can be effectively applied to real-world scenarios. With further refinement and fine-tuning, we are confident that our solution will be fully prepared for deployment in real robots, thereby contributing to advancements in robotic systems and their capabilities.

General Conclusion

This thesis has focused on the optimization of pose estimation techniques in robotics, with a specific focus on aerial and mobile robots in both single and multi-agent systems. By leveraging the power of deep learning and reinforcement learning, we have proposed novel approaches to enhance the accuracy, robustness, and adaptability of pose estimation algorithms.

First, we provided a comprehensive literature review, highlighting the different approaches, techniques, and tools that have been developed in the field of pose estimation. It identified the strengths and limitations of existing methods, laying the foundation for the subsequent chapters. We introduced deep learning and reinforcement learning as powerful tools for optimizing pose estimation techniques. Then, we present two approaches of pose estimation for single agents: the QR estimator for an adaptive version of the EKF and the KalmanNet approach for direct estimation of the gain K. Through extensive simulations, we demonstrated the effectiveness of these approaches, highlighting their strengths and limitations, including a comparison between theme in two different applications : mobile and aerial robots. After that, we extended our investigation to collaborative pose estimation in multi-agent systems. We proposed a novel approach based on measurements' selection, where an RL agent selects the best measurements for each agent based on signal characteristics. This approach aimed to improve accuracy and robustness by leveraging information from neighboring agents. Finally, we validated our findings in a real-world scenario using the ROS and Gazebo simulation environment.

Overall, this thesis has made significant contributions to the field of pose estimation in robotics. The adaptive versions of the EKF estimator, the novel approach for collaborative pose estimation and the validation in a realistic simulation environment demonstrate the effectiveness and potential of our proposed approaches.

However, it is important to acknowledge that there are still challenges and limitations that need to be addressed. Factors such as non-linearities, outliers and highly dynamic environments require further investigation and refinement of the proposed approaches. Additionally, the transferability of the approaches to other robotic platforms and real-world settings should be explored.

In conclusion, this thesis has provided valuable insights and advancements in pose estimation for aerial and mobile robots. The combination of deep learning, reinforcement learning, and collaborative estimation techniques offers promising avenues for improving the performance and capabilities of robotic systems. Future research should focus on addressing the remaining challenges and further enhancing the proposed approaches to enable their practical deployment in real-world scenarios.

Bibliography

[1] Tim Bailey et al. "Consistency of the EKF-SLAM algorithm". In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2006, pp. 3562–3568.

[2] Randal W. Beard. "Quadrotor Dynamics and Control Rev 0.1". In: 2008.

[3] Richard Bellman. "On the Theory of Dynamic Programming". In: *Proceedings of the National Academy of Sciences* 38.8 (1952), pp. 716–719. DOI: 10.1073/pnas.38.8.716. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.38.8.716.

[4] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019. DOI: 10.1017/9781108380690.

[5] José A. Castellanos, José Neira, and Juan D. Tardós. "Limits to the consistency of EKF-based SLAM". In: *IFAC Proceedings Volumes* 37.8 (2004). IFAC/EURON Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal, 5-7 July 2004, pp. 716–721. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)32063-3.

[6] Jakub Čerkala, Tomáš Klein, and Anna Jadlovska. "Modeling and Control of Mobile Robot with Differential Chassis". In: *Electrical Engineering and Informatics 6 : proceedings of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice* (Sept. 2015), pp. 651–656.

[7] Felix L Chernousko. *State estimation for dynamic systems*. CRC Press, 1993.

[8] Junyoung Chung et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[9] Jesse Clifton and Eric Laber. "Q-Learning: Theory and Applications". In: *Annual Review of Statistics and Its Application* 7 (Mar. 2020), pp. 279–301. DOI: 10.1146/annurev-statistics-031219-041220.

[10] Nobuhiro Funabiki et al. "Range-aided pose-graph-based SLAM: Applications of deployable ranging beacons for unknown environment exploration". In: *IEEE Robotics and Automation Letters* 6.1 (2020), pp. 48–55.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[12] J.E. Humphreys. *Introduction to Lie Algebras and Representation Theory*. Graduate Texts in Mathematics. Springer New York, 2012. ISBN: 9781461263982.

[13] The MathWorks Inc. *MATLAB version: 9.13.0 (R2022b)*. Natick, Massachusetts, United States, 2022.

[14] The MathWorks Inc. *Reinforcement learning Toolbox (R2022b)*. Natick, Massachusetts, United States, 2022.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore. *Reinforcement Learning: A Survey*. 1996. arXiv: cs/9605103 [cs.AI].

[16] Christoforos Kanellakis, Sina Sharif Mansouri, and George Nikolakopoulos. "Dynamic visual sensing based on MPC controlled UAVs". In: *2017 25th Mediterranean Conference on Control and Automation (MED)*. IEEE. 2017, pp. 1201–1206.

[17] Jed M Kelsey et al. "Vision-based relative pose estimation for autonomous rendezvous and docking". In: *2006 IEEE aerospace conference*. IEEE. 2006, 20–pp.

[18] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, pp. 2149–2154.

[19] Manon Kok, Jeroen D. Hol, and Thomas B. Schön. "Using Inertial Sensors for Position and Orientation Estimation". In: *Foundations and Trends in Signal Processing* 11.1-2 (2017), pp. 1–153. ISSN: 1932-8346. DOI: 10.1561/2000000094.

[20] Taeyoung Lee, Melvin Leok, and N. Harris McClamroch. "Geometric tracking control of a quadrotor UAV on SE(3)". In: *49th IEEE Conference on Decision and Control (CDC)*. 2010, pp. 5420–5425. DOI: 10.1109/CDC.2010.5717652.

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

[22] Sina Sharif Mansouri et al. "Cooperative coverage path planning for visual inspection". In: *Control Engineering Practice* 74 (2018), pp. 118–131.

[23] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533.

[24] Zahra Mohajerani. "Vision-based UAV pose estimation". In: 2008.

[25] Sajad Mousavi, Michael Schukat, and Enda Howley. "Deep Reinforcement Learning: An Overview". In: June 2018, pp. 426–440. ISBN: 978-3-319-56990-1. DOI: 10.1007/978-3-319-56991-8_32.

[26] Siti Nurmaini, Kemala Dewi, and Bambang Tutuko. "Differential-Drive Mobile Robot Control Design based-on Linear Feedback Control Law". In: *IOP Conference Series: Materials Science and Engineering* 190 (Apr. 2017), p. 012001. DOI: 10.1088/1757-899X/190/1/012001.

[27] Python Software Foundation. *Python Language Reference*. Available at: https://www.python.org/doc/versions/3.9/. 2023.

[28] Guy Revach et al. "KalmanNet: Neural Network Aided Kalman Filtering for Partially Known Dynamics". In: *IEEE Transactions on Signal Processing* 70 (2022), pp. 1532–1547. DOI: 10.1109/tsp.2022.3158588.

[29] José Salvador, João Oliveira, and Mauricio Breternitz. *REINFORCEMENT LEARNING: A LITERATURE REVIEW (September 2020)*. Oct. 2020. DOI: 10.13140/RG.2.2.30323.76327.

[30] M. Schuster and K.K. Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681. DOI: 10.1109/78.650093.

[31] Mohammed Shalaby et al. “Relative Position Estimation in Multi-Agent Systems Using Attitude-Coupled Range Measurements”. In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4955–4961. DOI: 10.1109/LRA.2021.3067253.

[32] Dan Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches.* John Wiley & Sons, 2006.

[33] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System.* Version ROS Melodic Morenia. May 23, 2018.

[34] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction.* Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN: 9780262352703.

[35] Avraam Tsantekidis, Nikolaos Passalis, and Anastasios Tefas. “Chapter 5 - Recurrent neural networks”. In: *Deep Learning for Robot Perception and Cognition.* Ed. by Alexandros Iosifidis and Anastasios Tefas. Academic Press, 2022, pp. 101–115. ISBN: 978-0-323-85787-1. DOI: https://doi.org/10.1016/B978-0-32-385787-1.00010-5.

[36] Greg Welch and Gary Bishop. “An Introduction to the Kalman Filter”. In: *Proc. Siggraph Course* 8 (Jan. 2006).

[37] Chao Yu et al. “The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games”. In: *Advances in Neural Information Processing Systems.* Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 24611–24624.

[38] Adamantios Zaras, Nikolaos Passalis, and Anastasios Tefas. “Chapter 2 - Neural networks and backpropagation”. In: *Deep Learning for Robot Perception and Cognition.* Ed. by Alexandros Iosifidis and Anastasios Tefas. Academic Press, 2022, pp. 17–34. ISBN: 978-0-323-85787-1. DOI: https://doi.org/10.1016/B978-0-32-385787-1.00007-5.

[39] Adamantios Zaras, Nikolaos Passalis, and Anastasios Tefas. “Neural networks and backpropagation”. In: *Deep Learning for Robot Perception and Cognition.* Academic Press, 2022, pp. 17–34. ISBN: 978-0-323-85787-1. DOI: https://doi.org/10.1016/B978-0-32-385787-1.00007-5.

[40] Fengjiao Zhang, Jie Li, and Zhi Li. “A TD3-based multi-agent deep reinforcement learning method in mixed cooperation-competition environment”. In: *Neurocomputing* 411 (2020), pp. 206–215. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2020.05.097.

Appendices

A Simulation parameters for mobile robot

initial state	X position	randomly selected $[-1, 1]$ m
	Y position	randomly selected $[-1, 1]$ m
	the angle θ	randomly initialized $[-\pi, \pi]$ rad
	linear velocity	0 m/s
	angular velocity $\dot{\theta}$	0 rad/s
final state	X position	randomly selected $[-3, 1]$ m
	Y position	randomly selected $[-3, 1]$ m
	the angle θ	randomly selected $[-\pi, \pi]$ rad
	linear velocity	0 m/s
	angular velocity $\dot{\theta}$	0 rad/s
initial inputs	τ_r	0 N.m
	τ_l	0 N.m
controller variables	K_x	3
	K_y	6
	K_θ	0.1
	K_p	0.5
	K_i	0.1
	K_d	0
robot variables	mass m	0.5 ± 0.25 kg (randomly selected)
	wheels radios r	0.01 ± 0.005 m (randomly selected)
	moment of inertia I	$0.01 \pm 0.005 g.m^2$ (randomly selected)
	distance between robot wheels L	0.2 ± 0.1 m (randomly selected)
robot model variables used by the state estimator	mass m	0.5 kg
	wheels radios r	0.01 m
	moment of inertia I	$0.01 Kg.m^2$
	distance between robot wheels L	0.2 m

Environment parameter variables for tests and training	noise power	$[0, 10]$ (randomly selected)
	noise color	[white, pink, brown, blue, violet] (randomly selected)
	Measurement offset	$5 \rightarrow 10\%$ (randomly selected)
	Failed sensor.	one or two (randomly selected)

Table 1: Used Parameters in the mobile robot simulation, test and training

B Simulation parameters for aerial robot

initial state	X position	randomly selected $[-5, 5]$ m
	Y position	randomly selected $[-5, 5]$ m
	Z position	randomly selected $[-5, 0]$ m
	Pitch (Euler) angle	θ 0 rad
	Roll (Euler) angle	ϕ 0 rad
	Yaw (Euler) angle	ψ 0 rad
initial inputs	Thrust T_Σ	0 N
	Moment acting along x axis M_1	0 $N.m^2$
	Moment acting along y axis M_2	0 $N.m^2$
	Moment acting along z axis M_3	0 $N.m^2$
PID controller gains	$K_{P,\dot{Z}}$	10
	$K_{P,\phi}$	0.2
	$K_{P,\psi}$	0.8
	$K_{P,\theta}$	0.2
	$K_{I,\dot{Z}}$	0.2
	$K_{I,\phi}$	0
	$K_{I,\psi}$	0
	$K_{I,\theta}$	0
	$K_{D,\dot{Z}}$	0
	$K_{D,\phi}$	0.15
	$K_{D,\psi}$	0.3
	$K_{D,\theta}$	0.15
aerial robot variables	mass m	1.25 ± 0.3 kg (randomly selected)
	Gravitational acceleration g	9.807 kg
	Arm length l	0.265 ± 0.01 m (randomly selected)
	x moment of inertia, I_{xx}	$0.232 \pm 0.01 kg.m^2$ (randomly selected)
	y moment of inertia, I_{yy}	$0.232 \pm 0.01 kg.m^2$ (randomly selected)
	z moment of inertia, I_{zz}	$0.468 \pm 0.01 kg.m^2$ (randomly selected)

aerial robot model variables used by the state estimator	mass m	1.25 kg
	Gravitational acceleration g	9.807 kg
	Arm length l	0.265 m
	x moment of inertia, I_{xx}	0.232 $kg.m^2$
	y moment of inertia, I_{yy}	0.232 $kg.m^2$
	z moment of inertia, I_{zz}	0.468 $kg.m^2$

Table 2: Used Parameters in the aerial robot simulation, test and training