Hochschule Fulda
University of Applied Sciences

A study of reinforcement learning algorithms in simulated robotics scenarios

Alejandro Pajares Chirre

Masters Thesis submitted to the Faculty of AI at HS Fulda

Matriculation No: 1331534
Supervisor: Prof. Dr. Alexander Gepperth
Co-Supervisor: Prof. Dr. David James

Submitted on dd.mm.yyyy

Abstract

The world of robotics has seen significant advancements in recent years, and this is largely due to the integration of machine learning techniques. Robots are now able to learn from their surroundings, make decisions, and carry out tasks with minimal human intervention. Machine learning has enabled robots to interact with humans and perform tasks that were previously considered impossible. In particular, reinforcement learning (RL) is a type of machine learning that models how humans learn from sensory input and motor responses in response to rewards. RL is based on the idea that an agent interacts with an environment by taking actions and receiving feedback in the form of rewards or punishments. Q-learning is a popular algorithm used in RL to learn the optimal policy, i.e., the best sequence of actions to maximize reward, for an agent. This thesis focuses on the application of reinforcement learning (RL) and Q-learning algorithms in controlling the motion of three joints of a six-degree-of-freedom (6-DOF) robotic arm in a simulated environment. To apply the Q-learning algorithms, the problem needs to be modeled as a Markov Decision Process, and during the learning process, the exploration and exploitation rate need to be balanced. The robotic arm is modeled in Gazebo, and the control commands are sent using the Robot Operating System (ROS 2). The objective of the robotic arm is to touch the target. The RL algorithm learns to maximize the reward function, which is based on the current and previous distance between the target and one end of the robotic arm and the angles of the three joints being controlled. The experimental results demonstrate that RL and Q-learning algorithms can effectively control the motion of a robotic arm in a simulated environment. The robotic arm successfully learns to approach and touch the target.

Contents

List of Figures

List of Tables

1 Introduction

Designing, building, operating, and programming robots is the primary objective of the engineering and scientific discipline of robotics. A robot is a device created to carry out operations mechanically or somewhat autonomously, frequently imitating human actions or behavior [1]. Numerous industries and applications, including agriculture, manufacturing, healthcare, transportation, entertainment, and the military, use robots. Robots are frequently utilized in industry to complete tasks like welding, creating art, integration, and packing that could be repetitious or dangerous for human workers. Robots are employed in agriculture to do duties including planting, harvesting, and crop monitoring [2]. Robots are utilized in the transportation industry for logistics, warehouse management, and autonomous driving. Robots are employed in the entertainment industry for activities including animatronics as well as special effects. Robots are employed in the military for operations including bomb disposal, surveillance, and reconnaissance. Robots are evolving rapidly in terms of versatility, intelligence, and adaptability, as well as in terms of possible applications. Robotics is an emerging discipline that has the potential to significantly alter numerous aspects of our everyday lives and will probably become more crucial in determining our future [3].

Numerous companies have embraced the use of robotics to aid humans in tasks that are monotonous, physically demanding, or hazardous. Yet, acquiring a robot and hiring a robotic engineer to create a tailored solution for a particular task requires a significant investment of resources. The duties of a robot engineer include setting up communication, designing control scripts, computing coordinate transformations, and creating error-handling programs. Typically, a technician takes on the task of operating the robot on a daily basis, or the robot operates on its own. However, if the task requirements or processes change, it is difficult to modify the existing robotic solution to suit a new configuration or application without the assistance of a robot engineer, despite the significant resources invested in acquiring and developing it. Instead of relying on a robotic engineer to manually program a robot's operations for a new application, companies could employ deep reinforcement learning to train an intelligent agent to control the robot specifically for that application. This approach would enable the resources invested in robotics to be more adaptable and versatile, suitable for a broader range of applications and purposes.

This thesis is about reinforcement learning which is a type of machine learning that involves an agent learning from its interactions with an environment in order to maximize a reward signal over time [4]. It is a method of learning that involves trial and error, with the agent receiving feedback in the form of rewards or punishments for its actions. According to Kaelbling, Littman, and Moore [5], reinforcement learning can be defined as "a problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment". Many different concepts and methodologies can be used to break down reinforcement learning. This work focuses on Deep Q learning and to study it a simulated robotic arm is trained to touch a can.

1.1 Context

1.1.1 Simulated Robotics

Simulated robotics is a rapidly developing field of study that strives to create intelligent systems that can communicate with virtual worlds in a manner that is comparable to how actual robots communicate with the real world [6]. Compared to traditional robots, simulated robotics has many benefits, such as lower costs, more flexibility, and the ability to test and improve algorithms in a secure setting. Simulated robotic systems can be applied to a variety of tasks, from straightforward ones like object detection and manipulation to more difficult ones including autonomous navigation, group decision-making, and experience-based learning. The application of virtual environments for autonomous vehicle training and testing is an illustration of simulated robotics [7]. To train and test the computer programs that operate autonomous vehicles, researchers can develop realistic simulations of numerous driving circumstances and scenarios, like traffic patterns, atmospheric conditions, and unforeseen incidents. With this strategy, researchers may evaluate the dependability and safety of autonomous vehicles in a safe setting before placing them on public roads. The application of virtual environments to train and test robots for search and rescue missions is another illustration of simulated robotics [8]. In order to train and test robotics that can aid in rescue operations, researchers can develop simulations of emergencies such as earthquakes or floods. With this method, researchers may test the efficacy and security of various robotic systems and algorithms in a range of circumstances without endangering human responders. Systems for industrial automation can also be developed using simulated robotics. For instance, manufacturers can build and test robotic assembly lines, improve production workflows, and spot possible bottlenecks or safety risks using simulations. Before installing physical systems in the real world, this method enables manufacturers to optimize their procedures and increase productivity.

1.1.2 Industrial Automation

Automation of industrial processes, such as manufacturing and construction, via the use of technological devices and control systems is known as industrial automation [9]. Industrial automation can be accomplished utilizing simulated robotic arms to carry out operations that would typically be carried out by human workers or actual robots. This entails creating the algorithms and control frameworks necessary for the virtual robotic arm to move and handle items precisely and effectively, as well as incorporating the virtual arm into a larger system that is capable of carrying out difficult tasks on its own. Applications for simulated robotic arms in industrial automation include material handling, manufacturing work, quality control examinations, and machine maintaining. Companies can decrease the expenses of real robots and human labor while boosting production and efficiency by deploying virtual robotic arms [10]. Additionally, for increased flexibility and scalability, simulated robotic arms can be flexibly customized and adapted to various industrial environments. All things considered, the employment of

simulated automated arms for automation in industry is a promising area of study and development, with tremendous potential for enhancing industrial procedures and developing the field of robotics.

1.1.3 Robotic Arm

Traditionally developed industrial robots have been confined to closed cells or limited-access warehouse areas [11]. They typically perform repetitive operations on standardized objects without human interaction. Programming these robots is usually a time-consuming process that requires specialized knowledge of the machine's software. However, current trends in robotics aim to make robots capable of operating in dynamic and open environments where they can work alongside humans. This presents new challenges that require equipping the robot with sensors to perceive its surroundings and interact with objects. However, integrating and utilizing sensor data for planning the robot's actions is not an easy task. A robotic arm functions like a human arm and is a mechanical system that typically includes an end effector for manipulating and interacting with the environment [12]. Robotic arms have various applications in industrial and service fields, such as pick and place, exploration, manufacturing, laboratory research, and space exploration. The 6 degrees of freedom allow the arm to pivot in six different directions, similar to a human arm. In industrial robotic arms, the mechanical structure and control mechanism are major factors of concern. These arms are commonly used in a variety of applications, such as manufacturing, assembly, material handling, and surgery. Robotic arms can be controlled by various means, including joystick or slider controls, keyboard-based interfaces, and programmed motions. They can be programmed to perform repetitive tasks with high precision and speed, which makes them ideal for use in industrial settings where consistency and efficiency are crucial. These arms can be equipped with various end-effectors, such as grippers, cameras, or welding tools, depending on the specific task that needs to be performed. They can also be designed to have multiple joints or degrees of freedom, which enables them to move in a wide range of directions and perform complex tasks. Advancements in robotics technology have led to the development of lightweight and portable robotic arms that can be easily integrated into various systems [13]. These arms are becoming increasingly popular in areas such as healthcare and rehabilitation, where they can assist with tasks such as lifting and moving patients. The primary focus of current research efforts is on training the robot's arm to carry out various tasks autonomously using deep learning technologies. However, due to the massive amount of data required to teach a robot effectively, a data-driven approach is necessary. This can be challenging to achieve using a physical robotic arm. Therefore, developers have turned to robot simulation software [14], [15] to overcome the limitations of data-intensive AI approaches and to provide a stable environment [16]. In a simulated environment, it is possible to control every aspect of the world, including impractical factors in reality. Moreover, there is no risk of damaging robots or human operators in simulations, and time control allows for faster data collection.

1.1.4 Robot Operating System (ROS)

The Robot Operating System (ROS) is a set of software libraries and tools that enables developers to build robotic applications [17]. It provides a framework for writing and running code across multiple computers and devices, making it easier to create complex robotic systems. ROS was first developed in 2007 by Willow Garage, at robotics research lab [18]. Since then, it has become widely adopted by the robotics community and is now supported by the Open Robotics organization. It offers a comprehensive platform for managing robotic systems. Originally designed to facilitate research in robotics, ROS is a unique framework. To grasp the fundamentals of the ROS framework, it is essential to comprehend the concept of message communication between nodes using topics. One of the key features of ROS is its ability to handle communication between different components of a robotic system, such as sensors, actuators, and controllers [19]. This communication is done using a publish-subscribe messaging system, which allows components to share data and commands in real-time. ROS also provides a wide range of tools and libraries for tasks such as perception, navigation, and manipulation, which can be used to build complex robotic applications. These tools include algorithms for object recognition, path planning, and motion control, among others. Another advantage of ROS is its open-source nature, which means that developers can contribute to the development of the software and share their own code with the community. This has led to a large and active community of developers working on ROS, which has helped to drive its development and adoption.

1.1.5 Simulations

Simulations serve as an entry point for Digital Twins, which are highly accurate depictions of the physical world [20]. These Twins can aid in boosting manufacturing output and improving the flexibility of supply chains. To streamline the implementation of manufacturing processes during production line changes, digital twinning involves linking simulation software to an actual self-governing robotic system. A robotic arm digital twin solution is showcased in a recent study [21], where the authors employed ROS [22] to achieve smooth functioning between the virtual and real worlds. Simulating software has its limitations as it cannot accurately represent the real world due to the imperfections in their physics engines. Additionally, simulations have the advantage of providing perfect data with no interference, which has supported the exploration of deep learning approaches in robotics research. Simulations are a powerful tool for designing and testing robotic arms. They allow engineers to create virtual models of robotic arms and simulate their behavior in different scenarios, without the need for a physical prototype. In a robotic arm simulation, the arm's mechanical structure, control system, and sensors are modeled in a virtual environment. The simulation can then be used to test the arm's performance in various tasks, such as picking and placing objects, assembling parts, or performing complex movements. One of the key benefits of using simulations for robotic arm design is that they can help identify potential issues or inef-

ficiencies before a physical prototype is built [23]. This can save time and resources, as well as improve the overall design of the arm. Simulations can also be used to optimize the control system of a robotic arm. By simulating the arm's behavior in different scenarios, engineers can identify the optimal control strategy for achieving a specific task or movement. The ROS framework includes a useful tool called RVIZ, which enables us to observe the robot's pose or estimation in a 3D environment [24]. With the correct configuration of the URDF file, the robot model can be visualized in RVIZ. Furthermore, simulations can help train and test algorithms for robotic arm control, such as RL methods or deep learning approaches. This can be done by running simulations with different environments and scenarios and using the resulting data to train and refine the algorithms.

1.1.6 Reinforcement Learning

Artificial neural networks (ANNs) are gaining importance in the field of robotics. In 2016, Levine et al.'s study [25] provided encouraging outcomes, indicating a direction toward a more straightforward approach to constructing robot behaviors. The end-to-end approach described in the study is more scalable than traditional programming methods. To meet the demand for autonomous systems that cater to societal needs, it is necessary to replace conventional, unsophisticated autonomous systems with intelligent ones. Intelligent systems can take on various forms of intervention, such as aiding humans with image and video analysis, language translation, simplifying sentences, solving math problems, managing portfolios, undertaking monotonous tasks in the manufacturing industry, driving cars, flying helicopters, and more. The complexity of certain tasks makes it impractical to solve them by specifying a set of rules due to the vast number of rules required, and programming an agent's behavior in advance is also difficult. However, machine learning techniques can be used to develop learned agents capable of addressing these challenges. Supervised learning techniques require large amounts of labeled data for training, making them less practical for certain scenarios. Unsupervised learning, on the other hand, is not well-suited to situations that involve interaction with the environment. Reinforcement learning (RL) [25] presents new opportunities for addressing various challenges. In the past, RL techniques were only effective in domains where handcrafted features or low-dimensional input were utilized, and could only handle discrete state and action spaces. However, recent advancements in Deep Learning (DL) have yielded promising results in several fields such as speech, vision, wireless communication, natural language translation, and assistive devices. These advancements in DL-based techniques have made it possible to overcome the challenges faced by RL, and to process raw data for better performance. The progress in deep learning techniques has paved the way for resolving the difficulties encountered in RL and processing raw data. The latest developments that merge RL with deep learning, such as those highlighted in [26], [27] increase their suitability to domains where handcrafted features are not present, and the state or action space is either vast or continuous. Such advanced methods can address both perception and planning issues,

whereas RL can only handle the planning problem and deep learning only the perception problem. Rewarding or punishing agents for their actions, reinforcement learning (RL) is a kind of machine learning that enables agents to learn through interactions with an environment. RL deals with the development of decision-making algorithms. Its main focus is on creating a set of actions that an agent can perform in a specific environment. At each time step, the agent takes an action and receives feedback in the form of an observation and a reward. The ultimate goal of RL is to maximize the agent's total reward through a learning process that involves experimentation and feedback. In the learning process, an agent creates a policy. The agent begins in a particular state 's' of the environment, performs an action 'a', and transitions to another state 's'. The agent receives scalar feedback in the form of a reward 'r' after taking the action. This cycle is repeated many times during the learning process, as illustrated in Figure 1.

The action space represents the possible actions that the agent can take in a given state. In the past, the tabular approach was used to store state and action values. However, in environments with a large number of states or actions, the approximation approach has replaced the tabular method. Neural networks are commonly used as approximation methods, as shown in Figures 2a and b. There are two types of action spaces, as illustrated.

Deep Reinforcement Learning (DRL) is a field of study in which neural networks are utilized as function approximators to improve the performance of RL. Recently, several DRL techniques have achieved remarkable success in learning complex behavior skills and solving challenging control tasks in high-dimensional state-space [28] environments. However, many benchmarked environments such as Atari [29] and Mujoco [30] lack complexity or realism, which is often present in robotics. Additionally, these benchmarked environments [31] do not use commonly used tools in the field such as ROS. The research conducted in the previous work requires a considerable amount of effort for each specific robot, and therefore, the scalability of previous methods for modular robots is questionable. Consequently, trial and error learning process is often needed to apply the previous research to real-world robots. Training robotic arms to carry out certain actions, including touching an object in a virtual environment, is an intriguing use of RL. The robotic arm in this scenario must navigate a complicated state space while selecting actions that would maximize its reward while avoiding collisions with the environment.

1.1.7 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) leverages deep learning architectures as function approximators to tackle high-dimensional data and address the challenge of approximating in the presence of large state and action spaces [32]. Unlike traditional methods such as decision trees or SVMs, DRL employs neural networks to map states to actions, enabling it to handle high-dimensional data types such as images, videos, and time-

series data. The approach also utilizes deep learning techniques such as convolutional or recurrent neural networks to address the limitations of traditional artificial neural networks which are unable to handle such data and often ignore input data topology. Prior research has focused on addressing various challenges in applying DRL to different fields, particularly control problems and games like Atari. Some of the key tasks involved in DRL include:

- **Exploration Exploitation:** The exploration refers to trying a new action, whereas exploitation makes use of learned knowledge to decide the action.
- **Generalization:** Generalization, on the other hand, refers to the capability of the agent to adapt to new environments, which can range from one task to another or from simulation to a real-life situation.
- **Finding Policy:** Finding a valuable policy involves identifying important states and actions that can help in learning an optimal policy for decision-making.
- **Finding a catastrophe:** Discovering a catastrophic event is crucial, as such events may cause significant harm. These events may include physical harm, offensive tweets, false stories, and so on. Avoiding such occurrences can help to improve the policy.
- **Handling Overestimation:** Overestimation occurs when inaccurate computation of action value takes place, often due to the use of the max operation in Q learning. It is important to handle overestimation to ensure that accurate values are computed.
- **Reducing Sample Size:** Deep reinforcement learning (DRL) requires a large number of samples for effective training, which may not always be feasible in real-world scenarios. This poses a challenge for DRL applications that deal with limited data availability.
- **Detection and prevention of overfitting:** overfitting is a common issue in DRL, especially when using high-capacity deep learning models. Overfitting occurs when the agent is too sensitive to small perturbations in the environment.
- **Robust Learning:** In recent years, there has been a growing interest in incorporating robustness into the DRL system using techniques such as deep learning. Researchers have proposed various adversarial attacks and defense mechanisms to address this challenge. Therefore, enhancing the robustness of DRL systems has become an active research topic in the community.

The field of Reinforcement Learning (RL) has seen significant contributions from various researchers who have proposed different network architectures and action selection criteria. Some methods include trial and error without human intervention, learning via demonstrations, learning via criticism, and learning with adversaries to

tackle complex problems. Despite these advancements, the challenge of discovering an optimal policy that is both robust and able to meet multiple goals remains a topic of active research and an open area of investigation.

1.1.8 Double Q-Learning

The Q-learning algorithm, a well-liked RL method, has a version called double Q-learning [33]. The fundamental tenet of Q-learning is to acquire knowledge of a function Q(s,a), which denotes the anticipated reward for performing action in a given state s. The agent employs this feature to determine the optimal course of action in each state. However, Q-learning may be hampered by an overestimation of the Q-values, which could result in ineffective policies. This problem is addressed by double Q-learning, which estimates the greatest expected reward by switching between two different Q-functions. The robot's arm would operate as the agent in a Double Q-learning simulation [34], interacting with its surroundings and receiving rewards or punishments as a result of its actions. The orientation and position of the robotic arm, along with the location and characteristics of the object it is attempting to touch, would all be included in the state space. The robotic arm's range of potential motions for approaching the object would make up the action space. The goal of the reward function is to persuade the robotic arm to touch the target without causing a collision or any other unfavorable outcomes. The Double Q-learning method is used by the robotic arm to investigate its environment during training and modify its Q-functions in response to incentives. By estimating the expected reward for every action using the two Q-functions, the algorithm would select the action featuring the largest expected reward. The robotic arm would develop a policy that enables it to consistently touch the target by continuing this process over a large number of attempts. In this thesis, we explore the use of Double Q-Learning in a simulated robotic arm to learn effective control policies for the arm.

1.2 Problem statement

Many studies have demonstrated that utilizing reinforcement learning (RL) presented a viable solution for addressing the limitations of conventional methods in tackling intricate robotics tasks. Numerous AI experts have created various frameworks and toolkits to examine and assess their algorithms' effectiveness in solving challenging problems [35]. Although the outcomes were remarkable, these applications were generally limited to simulated environments and seldom deployed in real-world scenarios. Numerous researchers are presently focused on a highly promising mission of bridging the gap between simulation and reality. However, proficiency in various domains is crucial in the intricate field of RL, which might be an obstacle to entry for roboticists. For Problems involving an agent interacting with the environment to maximize a reward signal, RL, a potent branch of machine learning (ML), is used. Gaming, robotics, and finance are just a few of the industries where RL has been successfully used. RL has been applied to robotics to tackle a variety of problems, including grasping, manipulating,

and navigating. One of these tasks involves teaching a virtual robotic arm to touch an object using reinforcement learning. The objective of this proposed study is to create a simulated robotic arm that can learn the Double Q Learning method for RL in order to learn how to touch an object. The object should not be knocked over or sustain any damage when the robotic arm moves its end-effector opposite direction and touches it. The following are the study's primary obstacles:

- Designing the incentive function: It is difficult to create a reward mechanism that encourages the robotic arm to interact with the object without damaging it or knocking it over.
- Tackling high-dimensional events and action spaces: It is difficult to apply RL methods to the robotic arm's highly dimensional state space and continuous action space.
- Safeguarding the surroundings and the robotic arm is essential while instructing the agent.

We will utilize the Double Q Learning technique to address this issue since it uses two Q-value functions to address the overestimation problem with Q Learning. The best course of action is chosen using one Q-value function, and the chosen course of action is assessed using the other Q-value function. The environment and robotic arm will be simulated using a physical simulator. The robotic arm will use the Double Q Learning method to choose an action after receiving state information from the simulator. In order to give the agent knowledge about the next state and a reward signal, a simulator will model the dynamics of the robotic arm and the object.

1.3 Goals

The following are the main goals of this study:

- Use RL to instruct the robotic arm to approach and grasp an object.
- To train a robot's arm, use the Double Q-learning algorithm.
- Establish a reward mechanism to motivate the robotic arm to effectively touch the target.
- Create a simulated space where the robotic arm is capable of touching things.
- Define the robotic arm's state space in the simulation environment.
- Define the robotic arm's reaction space in the simulation environment.
- Train the robot's arm on the best ways to touch an object so it can learn them.

- To get the greatest performance, fine-tune the Double Q-learning algorithm's hyperparameters.
- By counting the number of times, the robotic arm successfully touches an object, you may evaluate how well it is working.
- Continually enhance the training procedure by modifying the algorithm or its reward function as necessary.

Here, give a list of bullet points of quantifiable goals of the presented work. These achievement of these goals will be shown in the experiments section. The list should have 3-4 entries. No blabla here, hard goals!

1.4 Related Work

Deep Q-learning has been successfully applied in various robotic control tasks, including manipulators and robotic arms. In this section, we summarize the results and contributions of six relevant papers on deep Q-learning for robotic arm control.

In [36], the authors proposed a method for robotic grasping using deep Q-networks (DQNs). The results showed that the proposed method outperforms other traditional grasping methods in terms of grasping success rate and efficiency.

In [37], the authors presented a framework for real-time control of a robotic arm using deep reinforcement learning. They used a DQN-based algorithm to learn the control policy for the robotic arm, and the results demonstrated that the proposed method can achieve accurate and robust control.

List works that have a similar goals, plus a short explanation (3-4 sentences at most) as to how they differ from your work. Do NOT make comparisons here (better than my work or similar), that happens in the discussion section.

Admissible related work is (in descending order of acceptability):

- Peer-reviewed scientific publications, ideally with a DOI. Use Google scholar for searching (GS can export BibTeX entries that you can copy into the .bib file of this project).
- White papers and publicly available documents without review, cite with title, URL and date of access. In addition, you need to submit the PDFs in electronic form.
- Web pages, especially for software projects (e.g., TensorFlow, nginx, react, Django). Cite via URL and date of access. A github/gitlab/etc link is acceptable as well. Nothing needs to be submitted electronically, but only use such sources of there is no other way.

Literature is cited like this: as shown in clemen1989combining, blablaba. Or: [**?**] has a similar scope in the domain of perverted numerical integration, however without considering the aspect of cupidity. Or: In [**?**], a study of perverted diagonal matrix perversions is presented. You need not include page numbers.

See also Kap. 7.9, 7.10.

1.5 Contribution

Here, you present a bullet ist of your personal contributions to the topic of the thesis. For example:

- Implementation of a bash script that did not work and was hard to read
- Comparison of different implementations for matrix perversion
- Implementation of a web service that provides jokes about professors via a ReST API.

2 Foundations

2.1 Python

Python is a popular language for implementing deep learning algorithms because of its simplicity, adaptability, and abundance of tools and frameworks. It has grown in popularity as a programming language for machine learning, deep learning, and robotics due to its simplicity of use, wide library support, and adaptability. TensorFlow, PyTorch, Keras, OpenCV, NumPy, SciPy, ROS, and Gazebo are just a few of the libraries and frameworks available in Python for machine learning, deep learning, and robotics. These libraries and frameworks provide developers with a variety of tools and methods for developing and deploying machine learning, deep learning, and robotic applications. Python's readability and ease of use make it an appealing choice for machine learning and robotics developers. Its syntax is straightforward and clear, making it easier to develop and comprehend code. Furthermore, the dynamic nature of Python allows for quick prototyping and experimentation, which is vital for building and testing machine learning and robotics algorithms. Python's extensive library and frameworks, paired with its simplicity of use and adaptability, makes it an excellent choice for machine learning, deep learning, and robotics applications. Here are some of the most important Python modules and frameworks for these fields:

1. TensorFlow: Google's popular deep learning package that supports both static and dynamic computation graphs. TensorFlow is utilized in a variety of applications such as computer vision, natural language processing, and robotics.

2. PyTorch: A famous deep learning library created by Facebook that is noted for its simplicity and adaptability. PyTorch is utilized in a variety of applications such as computer vision, natural language processing, and robotics.

3. Keras: Keras is a high-level deep learning API that may be used in conjunction with TensorFlow, Theano, or CNTK. Keras is well-known for its simplicity and is frequently used for quick prototyping and experimentation.

4. OpenCV: OpenCV is a free and open-source computer vision toolkit that includes a variety of image processing and computer vision methods. OpenCV is frequently used in robotics applications like object identification and tracking.

5. NumPy: A Python library for numerical computing that supports massive, multi-dimensional arrays and matrices. NumPy is commonly used as a basis for many other scientific computing libraries, as well as in machine learning and robotics applications.

6. SciPy: A Python library for scientific computing that supports optimization, signal processing, and other scientific computing activities. For tasks such as optimization and control, SciPy is frequently utilized in machine learning and robotics applications.

7. Python: Python has a variety of robotics libraries, including ROS (Robot Operating System) and Gazebo, which are frequently used for designing and modeling robotic applications.

Python's popularity in robotics is growing due to its simplicity of use, adaptability, and wide library support. Here are some of the reasons why Python is useful in robotics:

1. Python features a basic and easy-to-learn syntax, making it accessible to both beginners and professionals. Python's readability and compact syntax make it simple to create and comprehend code, which is required while designing and testing robotics algorithms.

2. Python is a versatile programming language that may be used for a variety of purposes, including robots. Python is a versatile choice for robotics developers since it can be used for both high-level and low-level programming.

3. Extensive library support: Python has a number of libraries and frameworks intended expressly for robotics, such as ROS, Gazebo, and PyBullet. These libraries and frameworks provide developers a variety of tools and algorithms for developing and delivering robotics applications.

4. Python's dynamic nature enables quick prototyping and experimentation, which is vital for designing and testing robotics algorithms. Python's interactive shell also makes it simple to test code and try out new ideas.

5. Integration with other languages: Python can readily integrate with other programming languages used in robotics, such as C++ and MATLAB. This enables robotics engineers, regardless of programming language, to employ the finest tools for the job.

2.2 Pytorch

PyTorch is a Python-based open-source machine learning library based on the Torch library, which was created by Facebook's AI Research (FAIR) team [38]. It is generally employed for creating deep learning models for applications like speech recognition, computer vision, and natural language processing. PyTorch uses a dynamic computational network to construct and adapt models, which gives it a competitive advantage over competing deep learning frameworks. It also provides a variety of tools and utilities for developing, training, and assessing deep learning models, including as data loaders, optimizers, and loss functions. One of PyTorch's primary advantages is its ability to easily integrate with Python, making it simple to use for developers and researchers who are already familiar with Python. Furthermore, PyTorch provides a user-friendly interface for developing and training deep learning models, reducing the time and effort necessary to design and deploy models. PyTorch provides a versatile and fast framework for creating and training deep neural networks, making it an excellent candidate for DDQN implementation. Typically, the procedure entails establishing the neural network architecture, configuring the environment, and performing training loops to update the network weights depending on the agent's actions and rewards. PyTorch also has several important DDQN capabilities, such as the ability to build various optimization algorithms and loss functions, which can assist to increase the efficiency and efficacy of the learning process. Furthermore, PyTorch's automated differentiation capability can make it easier to construct and debug complicated DDQN algorithms by automatically calculating gradients during the training phase.

2.3 Gazebo

Gazebo is a free and open-source 3D simulation platform for robotics and automation [39]. It enables users to design and simulate complex systems like robots, sensors, and surroundings, and it is used for a variety of applications such as robot development, testing, and validation. Gazebo has a realistic physics engine that properly replicates object behavior and interactions with the environment, allowing developers to test and debug their algorithms in a safe and controlled setting. It also includes a variety of sensors and actuators for simulating various sorts of robotics and automation systems. Gazebo is built on top of the ODE (Open Dynamics Engine) physics engine and generates realistic images with the Ogre 3D rendering engine. It is developed in C++ and works with a variety of programming languages such as Python, Java, and MATLAB. Gazebo is frequently used in combination with other robotics libraries and frameworks,

such as ROS (Robot Operating System), which offers a comprehensive collection of tools and utilities for developing and testing robotic systems. ROS contains a Gazebo integration package that enables smooth integration of the two platforms, making it simple to model and test robotic systems with Gazebo. Gazebo is largely used in robotics and automation applications for simulation. It offers a realistic simulation environment for testing and evaluating algorithms and systems before deploying them in the real world. Users may use Gazebo to design and simulate complex systems such as robots, sensors, and surroundings, as well as test their algorithms in a safe and controlled environment. The simulation environment contains a physics engine that realistically models object behavior and interactions with the environment, allowing developers to test and debug their algorithms and systems under a variety of scenarios. Gazebo offers a wide range of sensors and actuators, such as cameras, lidar, and GPS, that may be used to mimic many sorts of robots and automation systems [40]. It also allows plugins, which let users customize and enhance the simulation environment's capabilities to match their individual needs. Gazebo is often used in robotics research and development, as well as in robotics and automation system teaching and training. It may be used in conjunction with other robotics libraries and frameworks, such as ROS (Robot Operating System), to offer a full simulation and development environment for robotic systems. Gazebo is commonly used for simulating robotic arms in robotics research and development. Gazebo offers a diverse set of sensors and actuators for simulating many sorts of robotic arms. Joint controllers, for example, can be used to regulate the position and velocity of the joints in the arm, and force/torque sensors can be used to mimic contact with objects and surfaces. Furthermore, Gazebo has a plugin system that allows developers to customize and enhance the simulation environment to match their individual requirements. Developers can write plugins, for example, to imitate certain sensors or actuators or to provide new control techniques for the robotic arm.

2.4 ROS 2

A collection of ROS software packages available for download is referred to as a ROS distribution, backed by the non-profit organization, Open-Source Robotics Foundations (OSRF) [41]. The ROS organization periodically updates these packages and assigns distinct titles to each distribution. ROS2 (Robot Operating System 2) [42] is the second edition of the popular open-source robotics middleware framework, ROS, launched in 2017 to address some of the limits and issues of the original ROS framework. ROS2's primary features and benefits include the following:

1. Enhanced real-time performance: In comparison to the original ROS middleware, ROS2 incorporates a new middleware layer called the Data Distribution Service (DDS), which delivers enhanced real-time speed and stability.

2. Improved support for multi-robot systems: ROS2 introduces the ROS2 Multi-Robot System (MRS) architecture, which improves communication and coordination between numerous robots.

3. ROS2 features support for Transport Layer Security (TLS) and X.509 certificates, which enables increased security and authentication for node-to-node connections.

4. Better non-Unix platform compatibility: ROS2 has enhanced support for non-Unix systems such as Windows and macOS.

5. ROS2 features better development tools and documentation, making it easier for developers to get started with the framework.

Conclusively, ROS2 is intended to provide a more robust, dependable, and adaptable platform for developing and deploying robotic systems. It is interoperable with a broad range of robotic hardware and software platforms, and it has a big and active development and user community. In robotics research and development, ROS2 is often used to operate robotic arms. Developers may use ROS2 to design a software stack that contains the control algorithms, sensors, and communication interfaces needed to control the robotic arm. ROS2 is a flexible and modular design that enables developers to construct software modules known as nodes that connect with one another via a publish-subscribe messaging model. For example, one node may be in charge of reading sensor data from the robotic arm, while another node may be in charge of operating the motors of the arm depending on the sensor data. A large selection of software libraries and tools referred to as packages, are also offered by ROS2 and may be used to create and test robotic arm control software. For instance, the *ros_control* package offers a variety of hardware interfaces and controllers for interacting with robotic arm hardware, while the *MoveIt* package offers a set of motion planning and control algorithms for robotic arms. Developers often begin by specifying the hardware interface for the arm, which comprises the joints, motors, and sensors, in order to control a robotic arm using ROS2. Then, using ROS2 nodes and packages, they create the control algorithms and sensor interfaces needed to control the arm. Finally, before implementing their robotic arm control software on the MoveItactual robot, engineers may test and evaluate it using simulation tools like Gazebo. This enables the software to be developed and iterated upon quickly without endangering the physical robot or its surroundings.

2.5 DDQN

Double Deep Q Learning (DDQN) [43] is a Q-Learning algorithm extension that overcomes Q-value overestimation in standard Q-Learning. DDQN is a DRL method that combines a deep neural network with the Q-Learning technique to develop an optimum policy for an agent to make decisions in an environment. The Double Deep Q-Learning (DDQN) method is an extension of the classic Q-Learning algorithm, which is widely used in Reinforcement Learning (RL) for robotics and robotic arm control. Deep RL method DDQN combines two deep neural networks to estimate the Q-values of state-action pairings in an environment. In this section, we will go through the DDQN idea, its benefits, and how it may be applied in robotics and robotic arm control. Q-Learning

is a well-known RL method that may be used to identify the best policy for an agent in a given environment. The agent in Q-Learning attempts to learn a function Q(s, a) that calculates the anticipated reward for doing an action in state s. The best policy is then found by choosing the action that maximizes the Q-value for a particular state. Q-Learning, on the other hand, can only be employed in tiny settings with a limited number of states and actions. It is unsuitable for big and complicated situations. Deep Q-Learning (DQL) is a Q-Learning variant that uses a deep neural network to estimate Q-values for state-action pairings in vast and complicated contexts. DQL has been demonstrated to be successful in a variety of difficult contexts, including video games and robotic control. However, it has been discovered that DQL can occasionally overstate Q-values, resulting in poor strategies. To overcome this issue, Double Deep Q-Learning (DDQN) was created, which estimates Q-values using two deep neural networks. One network, known as the target network, is used to produce Q-value targets, while the other, known as the online network, is used to generate Q-value estimates. To promote training stability, the Q-value objectives are updated less frequently than the Q-value estimations. This method decreases overestimation and enhances algorithm convergence. The DDQN algorithm is summarized below:

1. Set up two deep neural networks, one for the target network and one for the online network.
2. Create a replay buffer to record the agent's experience tuples (state, action, reward, next state).
3. Set the exploration strategy's parameters, such as the epsilon-greedy policy.
4. For every episode:
   (a) Return the environment to its original state.
   (b) Using the exploratory technique, choose an action.
   (c) Carry out the activity and watch for the reward and the following state.
   (d) Put the experience tuple into the replay buffer.
   (e) Take a random sample of experience tuples from the replay buffer.
   (f) Using the target network, compute the Q-value targets.
   (g) Using the online network, compute the Q-value estimations.
   (h) Determine the difference between the Q-value objectives and the Q-value estimations.
   (i) To minimize loss, update the online network via backpropagation.
   (j) Periodically update the target network with the weights of the online network.
   (k) Repeat steps b–j until the episode is finished.

5. Repeat step 4 for a fixed number of episodes or until the agent reaches a satisfactory level of performance.

There are various advantages of using DDQN over standard Q-Learning and DQL. It decreases overestimation, enhances stability, and speeds up convergence. DDQN is also capable of dealing with vast and complicated settings with multidimensional state and action spaces. These characteristics make DDQN an appealing candidate for robotic control applications. DDQN may be used in robotics and robotic arm control to determine an optimum policy for the agent to complete certain tasks such as item grabbing or assembly. Without any prior understanding of the environment or the work, DDQN may be used to learn the policy from the ground up. The agent may investigate its surroundings and learn from its experiences in order to better its performance.

2.6 Machine Learning Algorithms

Machine learning is a branch of AI that entails teaching computers to learn from data without being explicitly programmed. Machine learning seeks to develop models that can make predictions or judgments based on input data. In this post, we will go over the fundamentals of machine learning, such as its many kinds, techniques, and applications. There are several machine learning algorithms that are utilized in various sorts of machine learning jobs. Machine learning methods of many types can be applied to robotics and robotic arm control jobs.

A machine learning algorithm that can be used to operate robotic arms. Here are some examples of how to utilize decision trees in this context:

1. Classification of Arm Configurations: ML Algorithms can be used to categorize various arm configurations depending on sensor data input. A decision tree, for example, may be trained to classify various arm configurations based on the location and orientation of the arm joints.

2. Grasp Classification: Using sensor data as input, decision trees may be used to categorize several sorts of grasping actions. A decision tree, for example, may be taught to distinguish between power and precision grasps depending on the pressure applied to the item being clutched.

3. Object Recognition: Using sensor data as input, decision trees can be implemented to recognize various things. A decision tree, for example, may be trained to recognize different things based on their form and size, which could be beneficial for gripping and putting objects in precise spots.

4. Ideal Path Planning: Using input sensor data and the desired job, decision trees may be utilized to design the ideal path for the robotic arm to travel. A decision tree, for example, may be taught to identify the best path for the arm to take to pick up an object depending on its size and position.

Some machine learning algorithms and their uses in our context are:

- **Decision Trees:** Decision trees are used for classification jobs in which the purpose is to label incoming data. In robotics, decision trees may be used for a number of tasks like object detection, path planning, and motion control. A decision tree, for example, might be used to identify different sorts of objects in a robot's surroundings or to decide the best course for a robot to take based on sensor data. Decision trees can be used to classify different arm configurations or grab types for robotic arm control. Based on sensor data input, decision trees may be used to identify various arm configurations or grip kinds. They may also be utilized to forecast the best path for the arm to take in order to complete a certain job.
- **Random Forests:** Random forests are an ensemble approach for improving the accuracy and resilience of classification problems by combining several decision trees. Random forests can be used in robotics to increase the accuracy and resilience of classification tasks. A random forest, for example, can be taught to identify different types of terrain based on sensor data input or to classify different sorts of items in a robot's surroundings. They can be used in robotic arm control to identify various items or to forecast the best-grabbing approach for certain things.
- **SVMs (Support Vector Machines):** SVMs are used for classification and regression problems. SVMs may be used in robotics for a range of tasks including object detection, path planning, and control. An SVM, for example, can be trained to recognize different sorts of objects based on their shape and size or to predict the best course for a robot to take depending on sensor data input. They can be used in robotic arm control to forecast the best path for the arm to take or to categorize various arm configurations.
- **Neural Networks:** A common deep learning approach, neural networks may be used for a range of tasks such as classification, regression, and reinforcement learning. In robotics, neural networks may be utilized for a range of tasks like object identification, motion control, and reinforcement learning. A neural network, for example, may be used to recognize different sorts of objects based on sensor data input or to learn and optimize the control of a robot's movement over time. They may be used in robotic arm control to understand complicated patterns and correlations in arm motions and forecast the best course for the arm to take.
- **Clustering algorithms:** these algorithms can be used to group together comparable data points, which is important in robotic arm control for finding similar grasping actions or arm configurations.

The choice of machine learning algorithm will depend on the specific task and data available in robotic arm control. It is important to select an algorithm that can effectively learn from the available data and perform the desired task accurately and efficiently. The application of these algorithms can greatly enhance the control and operation of robotic arms in various fields including manufacturing, logistics, and healthcare.

2.7 Neural Networks

Neural networks are machine learning models inspired by the structure and function of the human brain. They are effective tools for addressing a wide range of complicated issues, including as image and audio recognition, natural language processing, and gameplay. This section will offer an overview of neural networks, covering their construction, training method, and applications.

2.7.1 Structure of Neural Networks

Layers of linked nodes or neurons form neural networks, which are organized into an input layer, one or more hidden layers, and an output layer. Each neuron takes input from neurons in the previous layer, processes it using an activation function, and generates an output signal that is transferred to neurons in the next layer [44]. A neural network's input layer accepts raw input data, such as an image or a written document, and routes it to the first hidden layer. Depending on the network's topology, each neuron in the hidden layer analyses this input and creates an output signal, which is then passed on to the next hidden layer or the output layer. The output layer generates the network's final output, which might be a classification label, a numerical value, or a collection of probabilities.

2.7.2 Training of Neural Networks

The process of training a neural network entail modifying the weights and biases of the neurons in order to minimize the discrepancy between the network's expected and actual output. This is accomplished using a technique known as backpropagation, which analyses the difference between the expected and actual output and propagates this mistake backward through the network layers to alter the weights and biases of the neurons. The backpropagation method adjusts the weights and biases of the neurons using an optimization approach such as gradient descent to minimize the error between the expected and actual output. The optimization procedure entails interactively changing the weights and biases of the neurons depending on the computed error until the error is reduced to an acceptable level.

2.7.3 Types of Neural Networks

There are several varieties of neural networks, each built for a unique issue or data format. Among the most frequent forms of neural networks are:

- **Feedforward neural networks** are the most basic sort of neural network, with information flowing from the input layer to the output layer in just one way.
- **Convolutional neural networks (CNNs)** are specialized neural networks developed for image and video processing, with a two-dimensional array of pixels as the input.

- R**ecurrent neural networks (RNNs)** are neural networks that are designed to process data sequences such as text or time series data.
- **Long short-term memory (LSTM) networks** are particularly developed for processing data sequences with long-term dependencies.

2.7.4 Applications of Neural Networks

Neural networks have several applications in domains such as computer vision, natural language processing, robotics, and finance. Among the most frequent neural network applications are:

- **Image and audio recognition:** Neural networks may be taught to accurately recognize and categorize pictures and sounds.
- **Natural Language Processing:** Neural networks may be used to analyze and create natural languages, such as machine translation and text production. Neural networks may be used to regulate and optimize robot motions such as grasping and manipulation.
- **Finance:** Neural networks may be used to forecast stock prices, assess credit risk, and detect fraud.

2.7.5 Neural networks in robotics

Because of their capacity to learn from data and make predictions or judgments based on that data, neural networks are commonly utilized in robotics. They are useful for a wide range of applications including object detection, path planning, motion control, and manipulation. Object recognition is one of the most popular applications of neural networks in robotics. Neural networks may be trained on vast datasets of images to recognize distinct objects and categories them. This can be applied to jobs like picking and arranging things, in which a robot must detect and grip an object in a chaotic environment. Neural networks can additionally be employed for route planning, which is the process of identifying a safe and efficient way for a robot to go from one site to another. By training a neural network on a dataset of maps and obstacle configurations, the network can learn to anticipate the optimum path for the robot to take. Neural networks can be used in motion control to regulate the movement of robotic joints and end-effectors. By training a neural network on a dataset of joint angles and related end-effector locations, the network may learn to anticipate the joint angles necessary to move the end-effector to a desired position [45]. Manipulation tasks, such as grasping and assembling, are another use of neural networks in robotics. A neural network may learn to anticipate the optimum gripping stance for a particular item by training it on a dataset of grasping poses and matching object attributes.

2.7.6 Neural Network for the robotic arm

In robotics, neural networks may be used to control the movement of robotic arms, among other things. One typical way is to employ a neural network as an arm controller, in which the network receives sensor readings and generates control signals to move the arm to a desired position. A neural network must be trained on a dataset of sensor readings and matching arm positions before it can be used for arms control. The network's inputs might comprise joint angles, velocities, and end-effector locations, with the output being the joint torques necessary to move the arm to the desired position. The training method generally consists of iteratively modifying the network's weights to minimize the gap between the network's expected and true outputs. This may be accomplished through the use of various optimization techniques, such as stochastic gradient descent. Once trained, the network may be utilized to operate the arm in real time. The network receives sensor values from the arm and predicts the control signals required to move the arm to the desired position. The control signals may then be delivered to the arm's actuators, causing the arm to move. Other tasks connected to robotic arms, such as object identification and grasping, can also be performed using neural networks. A neural network, for example, may be trained on a collection of photographs of items and their related grasping stances and then used to predict the optimal gripping pose for a specific object.

2.8 Reinforcement Learning Algorithms

For example:

- Specialized libraries and their use
- Complex concepts of the chosen programming language
- Basics of machine learning, or neural networks, or both
- Description of used databases or datasets
- rviz
- rqt
- urdf file
- sdf file
- stl file

In subsequent chapters, you can reference this one to avoid having to explain everything over and over again. This means that you just include things here that are necessary for the understanding of later chapters, nothing more.

2.9 Topic 1

2.10 Topic 2

2.11 Topic 3

3 Implementation

3.1 Robotic Arm

The robotic arm used in the simulation is the Kuka KR210. The simulation was taken from https://github.com/udacity/RoboND-Kinematics-Project. To use the simulated arm, a ROS2 package was created and after that the files of type *URDF, DAE,* and *STL* were copied into our package. To make the robotic arm compatible with ROS2, the necessary plugins ros2_control and gazebo_ros2_control were added to the *URDF* file.

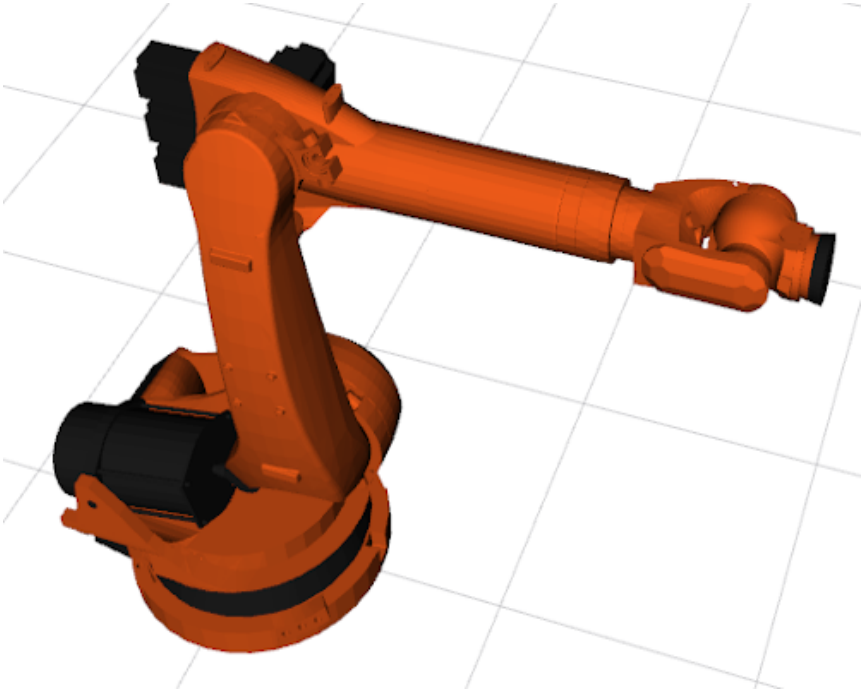


Figure 1: Kuka KR210 in Rviz2.

The robotic arm consists of six joints and seven links that connect them. The six joints of the KR210 are numbered J1 to J6, and they allow the robot to move in various directions and orientations.

- J1: The first joint is the base joint, which allows the robot to rotate horizontally around its vertical axis.

- J2: The second joint is the shoulder joint, which allows the robot to lift and lower its arm vertically.
- J3: The third joint is the elbow joint, which allows the robot to bend its arm vertically.
- J4: The fourth joint is the wrist roll joint, which allows the robot to rotate its wrist around its vertical axis.
- J5: The fifth joint is the wrist pitch joint, which allows the robot to tilt its wrist up and down.
- J6: The sixth joint is the wrist yaw joint, which allows the robot to rotate its wrist horizontally.

The arm also has links that connect the joints, including the base, lower arm, upper arm, wrist, and end-effector. These links are designed to provide strength and rigidity to the robot arm while allowing for smooth and precise movement. These links are:

- Base Link: This is the fixed part of the robot that is attached to the ground.
- Link 1: This is the first link that connects the base to joint 1.
- Link 2: This link connects joint 1 to joint 2.
- Link 3: This link connects joint 2 to joint 3.
- Link 4: This link connects joint 3 to joint 4.
- Link 5: This link connects joint 4 to joint 5.
- End Effector (Link 6): This is the final link that connects to the tool or object being manipulated.

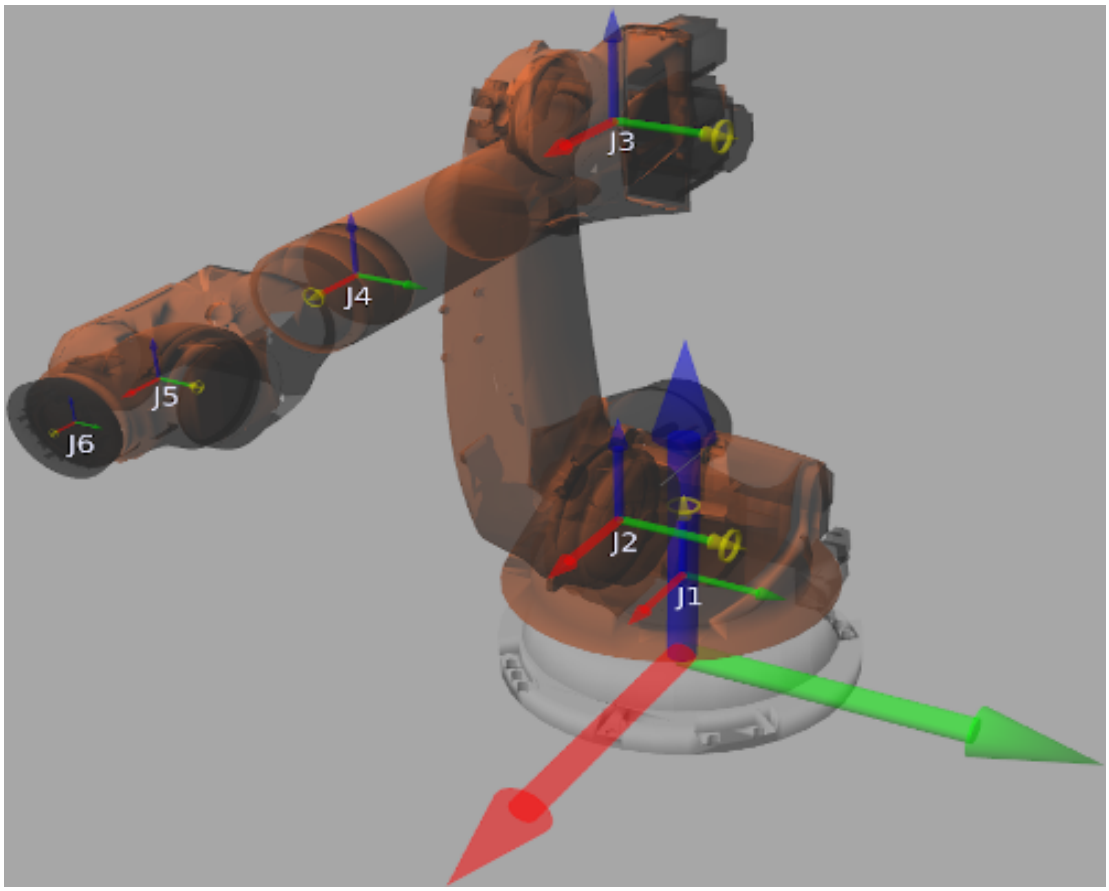
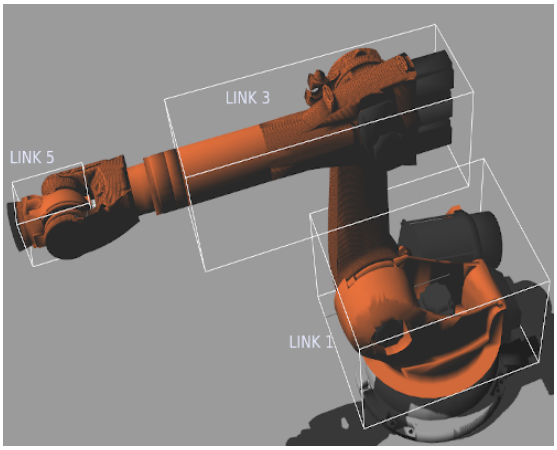
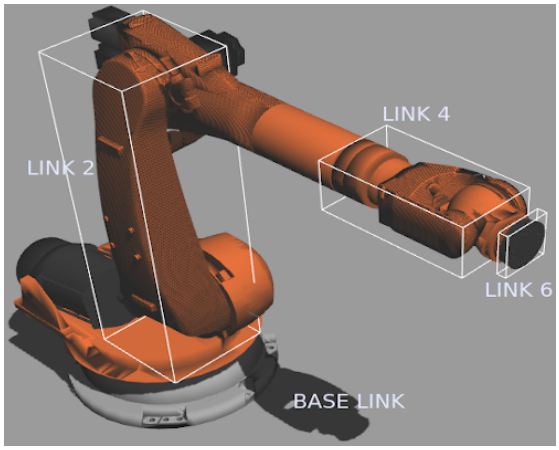


Figure 2: long caption



(a) subcaption.



(b) subcaption.

Figure 3: caption.

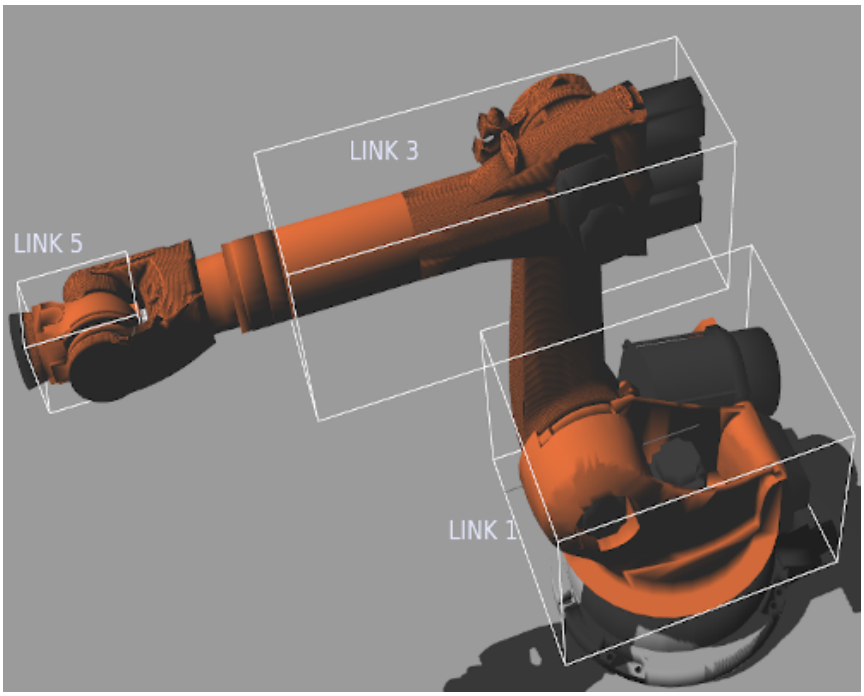


Figure 4: long caption

A Bumper sensor was added to links 4, 5, and 6 to detect collisions with these links (As shown in the code snippet) to the URDF.

A Camera sensor is added (As shown in the code snippet) to the URDF.

The object the robotic arm has to learn to touch is a can taken from gazebo models (http://models.gazebosim.org/beer). A world file is created containing the can.

The Gazebo ROS state plugin was added (as shown in 3) to the world file to have the positions, velocities, and other properties from our models published and to be able to modify them programmatically.

3.2 Moving the robotic arm

The AnglesPublisher node was created to move the arm by publishing the target angles for the joints. To do this the Joint Trajectory Controller is used, this controller generates and executes trajectories for the robot joints. It subscribes to a JointTrajectory message that specifies the desired trajectory, and then generates a control signal to move the robot's joints along the trajectory. By publishing messages of type JointTrajectory to the joint_trajectory topic the arm is moved.

3.3 Getting Images from the camera

The CameraSubscriber node was created to get images from the environment. The plugin previously used takes care of publishing the images to the topic /camera/image_raw,

the CameraSubscriber just subscribes to this topic and gets the images as messages of type Image.

ROS control module is used to move the arm. ROS Control is a framework for building and controlling robots in ROS (Robot Operating System). It provides a standardized way to manage hardware interfaces, controllers, and state machines, making it easier to develop and integrate robot applications.

The Joint State Broadcaster is a ROS node that broadcasts the state of robot joints. It reads joint positions, velocities, and efforts from a robot's hardware interfaces or controllers and publishes them as a JointState message on a ROS topic. This message contains the current state of all joints in the robot's model, allowing other ROS nodes to subscribe and use this information.

The Joint Trajectory Controller is a ROS controller that generates and executes trajectories for robot joints. It subscribes to a JointTrajectory message that specifies the desired trajectory, and then generates a control signal to move the robot's joints along the trajectory. It can use different algorithms to generate this control signal, such as PID, computed-torque, or model-predictive control. This controller is often used in combination with the Joint State Broadcaster to provide closed-loop control of the robot's joints.

Overall, the Joint State Broadcaster and Joint Trajectory Controller are two important components of ROS Control that enable the control of robot joints. The Joint State Broadcaster provides the current state of the joints to other ROS nodes, while the Joint Trajectory Controller generates and executes trajectories for the joints based on desired goals. Together, these components provide a powerful toolset for controlling robot motion in ROS.

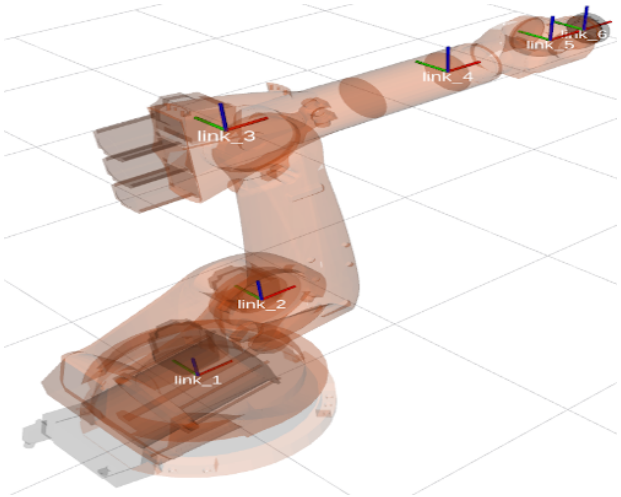


Figure 5: long caption

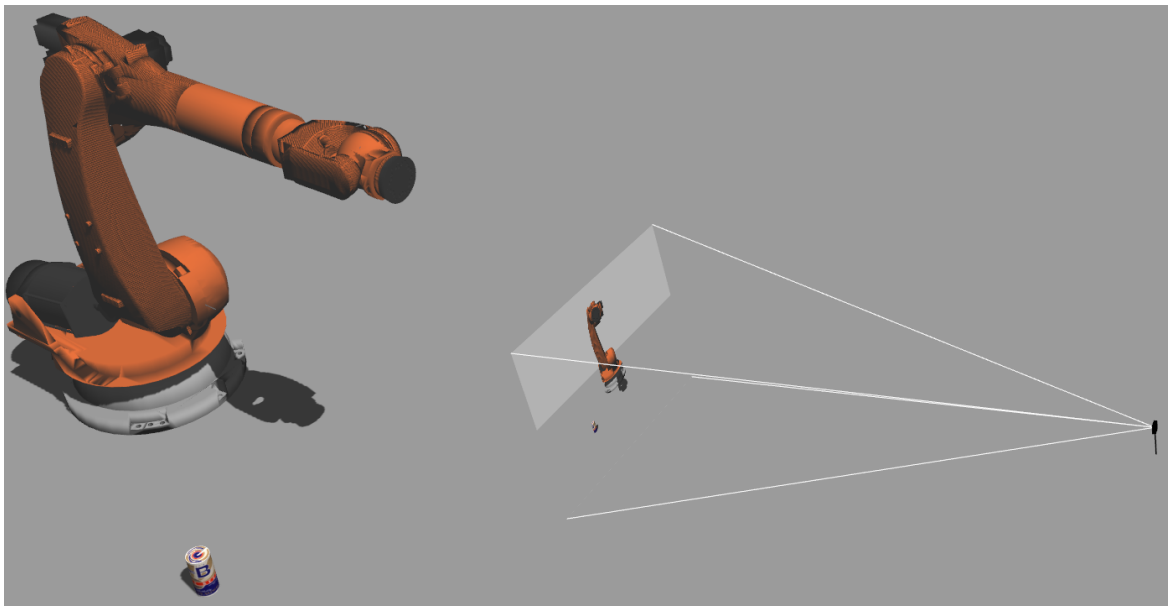


Figure 6: long caption

Figure 7: long description

Should refer, where possible, to the preceding chapter, e.e.: Singular value decomposition of the matrix Σ is conducted as explained in Sec. **??** using the *lapack* library (see Sec. 2.10).

For software development: what is the logic of the developed code, which of it was done by yourself? Sequence diagrams or UML are good tools here.

Please give code snippets only if they take up less that 0.25 pages, and only if it is unavoidable. Longer snippets go to the appendix and are referenced like this: see App. **??**.

4 Experiments

Show here that the goals from the introduction were achieved (or not achieved), you need at least one experiment per goal. Use screenshots, diagrams, plots, photos, etc. as necessary.

5 Discussion

2-3 pages are a good idea here. Picks up goals from the introduction (see 1.3) and experiments (see 4) and explains what was achieved and what was not (and why not in this case). Compares results with results from related work, see Sec. **??**. Draws a preliminary conclusion for the whole thesis.

6 Conclusion

Give an executive summary for important decision makers here, as well as an outlook (what would you do if you had another 3 months). 2-3 pages are ok here.

7 Using LaTeX, erase this chapter later

I was too lazy to translate this, it will be translated later. But I believe the ideas are clear!

7.1 Mathematische Gleichungen

Eine mehrzeilige Gleichung sieht so aus (die Symbole nach den und-Zeichen werden untereinander gesetzt). Die nonmber-Befehle verhindern dass die Gleichung nummertiert wird (Geschmackssache, ist nie falsch wenn eine Gleichung nummeriert ist). Aber: eine Gleichung auf die man refernziert (also die ein Label hat), muss nummeriert sein!

$$\begin{aligned} A &= \sum_{i=1}^{N} x_i \\ B &= \frac{\pi}{2} \end{aligned} \tag{1}$$

Eine inline-Gleichung: $x = 45b + \frac{2}{3}\pi$. Der Text geht weiter! Auf inline-Gleichungen kann man keine Refernzen erstellen.

7.2 Das ist eine Auflistung

1. Element 1
2. Element 2

7.3 Das ist eine Bullet-Liste

- Element 1
- Element 2

7.4 Eine Grafik bindet man so ein

Zulässige Formate sind generell eps, pdf und png.

Figure 8: Logo der HAW Fulda

7.5 So schreibt man einen Algorithmus

Algorithm 1: How to write algorithms

```
Data: this text
Result: how to write algorithm
initialization;
while not at end of this document do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end if
end while
```

7.6 So gestaltet man eine Tabelle

Table 1: Beispielstabelle

A	B	C
D	per gram	11.65
	each	1.01
E	stuffed	32.54
F	stuffed	73.23
G	frozen	8.39

7.7 Interne Referenzen

So wird ein Kapitel oder Unterkapitel referenziert: Kap. 1, Kap. 7.10. Auf Gleichungen bezieht man sich so: Wie in Gl. (1) gezeigt, sehen Gleichungen in der Regel gut aus. Auf Abb. 8 bezieht man sich so. Auf Tab. 1 referenziert man so. Algorithmen sind analog: siehe Alg. 1. Generell kann man alles zitieren was ein Label hat.

7.8 Textformatierung

So wird dick geschrieben und *so kursiv*.

7.9 Zitieren

Generell zitiert man so: wie in [**?**] gezeigt, blablaba. Für jedes zitierte Werk ist ein BibTex-Eintrag nötig! Eine gute Quelle ist Google Scholar!!

7.10 Webquellen zitieren

So wird eine Webquelle zitiert: [46], siehe auch den Eintag im BibTeX-File. Wichtig: für jede Web-Quelle ein BibTeX-Eintrag! Wenn Sie das auf die hier gezeigte Art machen, werden URLs (fast) automatisch getrennt. Kontrollieren Sie trotztdem die Literaturliste, es kann sein dass das nicht immer funktioniert.

7.11 Literaturverzichnis erstellen

Hierzu müssen BibTeX-Einträge in die Datei literatur.bib eingefügt werden. Die BibTeX-Keys sind jeweils Argumente für die cite-Kommandos! Wenn Sie literatur.bib ändern müssen Sie alles mindestens 5x compilieren: 3x mit latex, 1x mit BibTex und dann noch 2x mit LaTeX (in der Reihengfolge). Am besten Sie machen ein Skript dafür!

References

[1] S O'Sullivan, N Nevejans, C Allen, A Blyth, S Leonard, U Pagallo, and H Ashrafian. Legal, regulatory, and ethical frameworks for development of standards in artificial intelligence (ai) and autonomous robotic surgery. *The international journal of medical robotics and computer assisted surgery*, 15(1):e1968, 2019.

[2] Juan José Roldán, Juan del Cerro, David Garzón-Ramos, Pablo Garcia-Aunon, Mónica Garzón, Jonathan De León, and Antonio Barrientos. Robots in agriculture: State of art and practical experiences. In *Service robots*, pages 67–90. Springer, 2018.

[3] Nick Bostrom and Eliezer Yudkowsky. The ethics of artificial intelligence. In *Artificial Intelligence Safety and Security*, pages 57–69. Chapman and Hall/CRC, 2018.

[4] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[5] M. L. Littman L. P. Kaelbling and A. R. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[6] Hyungtae Choi, Colin Crump, Colin Duriez, Anna Elmquist, Gregory Hager, Dongheui Han, and Jeff Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1):e1907856118, 2021.

[7] Anna Elmquist, Radu Serban, and Dan Negrut. A sensor simulation framework for training and testing robots and autonomous vehicles. *Journal of Autonomous Vehicles and Systems*, 1(2), 2021.

[8] Carlos Sampedro, Alejandro Rodriguez-Ramos, Hriday Bavle, Adolfo Carrio, Pablo de la Puente, and Pascual Campoy. A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. *Journal of Intelligent & Robotic Systems*, 95:601–627, 2019.

[9] Paulo Leitão, Armando Walter Colombo, and Stamatis Karnouskos. Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges. *Computers in Industry*, 81:11–25, 2016.

[10] Muhammad Javaid, Abid Haleem, Ravi Pratap Singh, and Rakesh Suman. Substantial capabilities of robotics in enhancing industry 4.0 implementation. *Cognitive Robotics*, 1:58–75, 2021.

[11] Marco Lucchi, Florian Zindler, Stephanie Mühlbacher-Karrer, and Hannes Pichler. Robo-gym–an open source toolkit for distributed deep reinforcement learning on real and simulated robots. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5364–5371. IEEE, 2020.

[12] Patrick Ohta, Lorenzo Valle, Jared King, Katherine Low, Jeeyeon Yi, Christopher G Atkeson, and Yun Seong Park. Design of a lightweight soft robotic arm using pneumatic artificial muscles and inflatable sleeves. *Soft Robotics*, 5(2):204–215, 2018.

[13] B Singh, N Sellappan, and P Kumaradhas. Evolution of industrial robots and their applications. *International Journal of Emerging Technology and Advanced Engineering*, 3(5):763–768, 2013.

[14] O. Robotics. Ignition robotics. `https://ignitionrobotics.org/home`, Accessed: 2023-04-24.

[15] Erwin Coumans and Yun Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[16] Jerry Banks. *Introduction to simulation*. 1999.

[17] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, May 2009.

[18] Steve Cousins. Willow garage retrospective [ros topics]. *IEEE Robotics & Automation Magazine*, 21(1):16–20, 2014.

[19] Luis Emmi, Mariano Gonzalez-de Soto, Gonzalo Pajares, and Pablo Gonzalez-de Santos. New trends in robotics for agriculture: integration and assessment of a real fleet of robots. *The Scientific World Journal*, 2014, 2014.

[20] Yuxin Lu, Chen Liu, K I-K Wang, He Huang, and Xiaofei Xu. Digital twin-driven smart manufacturing: Connotation, reference model, applications and research issues. *Robotics and Computer-Integrated Manufacturing*, 61:101837, Feb 2020.

[21] P. Tavares, J. Silva, P. Costa, G. Veiga, and A. Moreira. Flexible work cell simulator using digital twin methodology for highly complex systems in industry 4.0. *Nov*, pages 541–552, 2018.

[22] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. `https://www.ros.org`. [Online; accessed 24-April-2023].

[23] T. R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, 2001.

[24] M. Kulkarni, P. Junare, M. Deshmukh, and P. P. Rege. Visual slam combined with object detection for autonomous indoor navigation using kinect v2 and ros. In *2021 IEEE 6th International Conference on Computing, Communication and Automation (ICCCA)*, pages 478–482. IEEE, 2021.

[25] J. E. Ball, D. T. Anderson, and C. S. Chan. Comprehensive survey of deep learning in remote sensing: theories, tools, and challenges for the community. *Journal of Applied Remote Sensing*, 11(4):042609–042609, 2017.

[26] Microsoft. Bonsai: Drl for industrial applications. `https://www.bons.ai/` and `https://aischool.microsoft.com/en-us/autonomous-systems/learning-paths`, 2014. [Online; accessed 30-May-2019].

[27] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[28] Yuxi Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in Neural Information Processing Systems*, pages 5285–5294, 2017.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[30] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[31] Tiago Nogueira, Simone Fratini, and Klaus Schilling. Autonomously controlling flexible timelines: From domain-independent planning to robust execution. In *2017 IEEE Aerospace Conference*, pages 1–15. IEEE, March 2017.

[32] Robert N Boute, Joren Gijsbrechts, Willem Van Jaarsveld, and Jeroen Vanvuchelen. Deep reinforcement learning for inventory control: A roadmap. *European Journal of Operational Research*, 298(2):401–412, 2022.

[33] Samuel Ogunniyi. Energy efficient path planning: the effectiveness of q-learning algorithm in saving energy. Master's thesis, University of Cape Town, 2014.

[34] Wenbo Weng, Himanshu Gupta, Nan He, Leslie Ying, and Rayadurgam Srikant. The mean-squared error of double q-learning. In *Advances in Neural Information Processing Systems*, volume 33, pages 6815–6826, 2020.

[35] Wei Wu, Tingting Huang, and Ke Gong. Ethical principles and governance technology development of ai in china. *Engineering*, 6(3):302–309, 2020.

[36] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[37] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, page 3389–3396. IEEE Press, 2017.

[38] Zhihua Wang, Kehan Liu, Jian Li, Yuchao Zhu, and Yifei Zhang. Various frameworks and libraries of machine learning and deep learning: a survey. *Archives of computational methods in engineering*, pages 1–24, 2019.

[39] Bulat Abbyasov, Roman Lavrenov, Andrey Zakiev, Konstantin Yakovlev, Mikhail Svinin, and Evgeny Magid. Automatic tool for gazebo world construction: from a grayscale image to a 3d solid model. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7226–7232. IEEE, 2020.

[40] Gonzalo Echeverria, Nicolas Lassabe, Arnaud Degroote, and Severin Lemaignan. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. IEEE, 2011.

[41] Emad Ebeid, Michael Skriver, Kristian H Terkildsen, Kjeld Jensen, and Ulrik Pagh Schultz. A survey of open-source uav flight controllers and flight simulators. *Microprocessors and Microsystems*, 61:11–20, 2018.

[42] Pattaraporn Phueakthong and Jinda Varagul. A development of mobile robot based on ros2 for navigation application. In *2021 International Electronics Symposium (IES)*, pages 517–520. IEEE, 2021.

[43] Khaled M Hamdia, Xiaoying Zhuang, and Timon Rabczuk. An efficient optimization approach for designing machine learning models based on genetic algorithm. *Neural Computing and Applications*, 33:1923–1933, 2021.

[44] K. M. Hamdia, X. Zhuang, and T. Rabczuk. An efficient optimization approach for designing machine learning models based on genetic algorithm. *Neural Computing and Applications*, 33:1923–1933, 2021.

[45] R. Villegas, J. Yang, D. Ceylan, and H. Lee. Neural kinematic networks for unsupervised motion retargetting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8639–8648, 2018.

[46] RStudio. Welcome to shiny. `https://shiny.rstudio.com/tutorial/written-tutorial/lesson1/`.

A Code Snippets

```
<gazebo reference="link_4">
  <!-- contact sensor -->
  <sensor name="end_effector_sensor" type="contact">
    <selfCollide>true</selfCollide>
    <alwaysOn>true</alwaysOn>
    <update_rate>500</update_rate>
    <contact>
      <collision>link_4_collision</collision>
    </contact>
    <!-- gazebo plugin -->
    <plugin name="gazebo_ros_bumper_sensor" filename="libgazebo_ros_bumper.so">
      <ros>
        <namespace>contact_sensor</namespace>
        <remapping>bumper_states:=bumper_link_4</remapping>
      </ros>
      <frame_name>link_4</frame_name>
    </plugin>
  </sensor>
</gazebo>
```

Listing 1: Bumper Sensor

```
<gazebo reference="camera_link">
  <material>Gazebo/Black</material>
  <sensor name="camera" type="camera">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>10</update_rate>
    <camera>
      <horizontal_fov>1.089</horizontal_fov>
      <image>
        <format>R8G8B8</format>
        <width>640</width>
        <height>480</height>
      </image>
      <clip>
        <near>0.05</near>
        <far>8.0</far>
      </clip>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <frame_name>camera_link_optical</frame_name>
    </plugin>
  </sensor>
</gazebo>
```

Listing 2: Camera Sensor

```
<plugin name='gazebo_ros_state' filename='libgazebo_ros_state.so'>
  <ros>
    <namespace>/gazebo_state</namespace>
    <argument>model_states:=model_states_demo</argument>
    <argument>link_states:=link_states_demo</argument>
  </ros>
  <update_rate>1.0</update_rate>
</plugin>
```

Listing 3: Gazebo ROS state plugin

```
  <gazebo>
    <plugin filename="libgazebo_ros2_control.so" name="gazebo_ros2_control">
      <robot_sim_type>gazebo_ros2_control/GazeboSystem</robot_sim_type>
      <parameters>/home/ros/ros2-projects/my-workspace/src/kuka_kr210/config/jtc.yaml</para
    </plugin>
  </gazebo>
```

Listing 4: Gazebo plugin and ROS2 controller configuration file.

```
    <ros2_control name="GazeboSystem" type="system">
      <hardware>
        <plugin>gazebo_ros2_control/GazeboSystem</plugin>
      </hardware>
      <joint name="joint_1">
        <command_interface name="position">
          <param name="min">-3.14</param>
          <param name="max">3.14</param>
        </command_interface>
        <command_interface name="velocity">
          <param name="min">-3.15</param>
          <param name="max">3.15</param>
        </command_interface>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
        <state_interface name="effort"/>
        <param name="initial_position">0.0</param>
      </joint>
      <joint name="joint_2">
        <command_interface name="position">
          <param name="min">-3.14</param>
          <param name="max">3.14</param>
        </command_interface>
        <command_interface name="velocity">
          <param name="min">-3.15</param>
```

```
      <param name="max">3.15</param>
    </command_interface>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
    <param name="initial_position">-1.57</param>
  </joint>
  <joint name="joint_3">
    <command_interface name="position">
      <param name="min">-3.14</param>
      <param name="max">3.14</param>
    </command_interface>
    <command_interface name="velocity">
      <param name="min">-3.15</param>
      <param name="max">3.15</param>
    </command_interface>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
    <param name="initial_position">0.0</param>
  </joint>
  <joint name="joint_4">
    <command_interface name="position">
      <param name="min">-3.14</param>
      <param name="max">3.14</param>
    </command_interface>
    <command_interface name="velocity">
      <param name="min">-3.2</param>
      <param name="max">3.2</param>
    </command_interface>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
    <state_interface name="effort"/>
    <param name="initial_position">-1.57</param>
  </joint>
  <joint name="joint_5">
    <command_interface name="position">
      <param name="min">-3.14</param>
      <param name="max">3.14</param>
    </command_interface>
    <command_interface name="velocity">
      <param name="min">-3.2</param>
      <param name="max">3.2</param>
    </command_interface>
    <state_interface name="position"/>
```

```
      <state_interface name="velocity"/>
      <state_interface name="effort"/>
      <param name="initial_position">0.0</param>
    </joint>
    <joint name="joint_6">
      <command_interface name="position">
        <param name="min">-3.14</param>
        <param name="max">3.14</param>
      </command_interface>
      <command_interface name="velocity">
        <param name="min">-3.2</param>
        <param name="max">3.2</param>
      </command_interface>
      <state_interface name="position"/>
      <state_interface name="velocity"/>
      <state_interface name="effort"/>
      <param name="initial_position">0.0</param>
    </joint>
  </ros2_control>
```

B Thesis defence

The defence is 15/20 minutes for Bachelor/Master, followed by questions and a discussion. Both examiners are present, and you can invite external persons since defences are generally public.

Targetted group are non-computer scientists, e.g., from higher management, NOT the examiners. Means that at least $\frac{1}{3}$ if the presentation is introduction/context/problem statement. You should re-use text/images/graphs/etc from the corresponding chapters here!

1 Slide per minute is a good guideline. If you can guess that some questions are going to be asked anyway, prepare some slides specifically for these questions, makes a good impression, and you can show them in the discussion time, not during the 15 minutes of the presentation.

Defences are not graded, you can only pass or not pass.

Students are responsible for finding dates for the defence and coordinating this with both supervisors.

Some common advice is:

- Speak slowly and loadly
- If you do not have enough time left for all slides, leave some out rather than rushing through all of them!!
- Slide numbers!

- In presence: be there 10 minutes ahead of time to check projectors etc. Makes a very bad impression if this is not working. Same for online presentations: be there 5 minutes ahead of time to verify screen sharing works.
- do not read text from the slides. These should contains key words only, and you explain the rest in free presentation
- Defences can by all means be online, more convenient for companies
- in presence: always carry a USB key with a PDF of your slides. If you have to use another PC than yours, PowerPoint slides may look very differently (fonts, page setup etc.)
- No-Go: spelling errors on slides!!!
- Do not use animations, they may not work in an online setting

C Extras

C.1 Markov Decision Process

A Markov Decision Process (MDP) is defined by a tuple $< S, A, P, R, \gamma >$, where:

- S is the set of states in the environment
- A is the set of actions that can be taken in each state
- P is the state transition probability matrix, where $P^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$
- R is the reward function, where $R^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$
- γ is the discount factor, where $\gamma \in [0, 1]$

The goal of an agent in a Markov Decision Process is to find a policy $\pi : S \to A$ that maximizes the expected discounted reward:

$$V_\pi(s) = \mathbb{E}\pi \left[\sum k = 0^\infty \gamma^k R_{t+k+1} \middle| S_t = s \right]$$

or the corresponding action-value function:

$$Q_\pi(s, a) = \mathbb{E}\pi \left[\sum k = 0^\infty \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]$$

C.2 Q learning Algorithm

The Q-learning algorithm is an off-policy temporal difference learning algorithm for finding the optimal action-value function $Q^(s, a)$ in a Markov Decision Process (MDP). The algorithm updates an estimate of $Q^(s, a)$ by iteratively applying the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big]$$

where s_t and a_t are the current state and action, R_{t+1} is the reward received after taking action a_t in state s_t and transitioning to state s_{t+1}, α is the learning rate, and γ is the discount factor.

The Q-learning algorithm can be summarized as follows:

1. Initialize the Q-value function $Q(s, a)$ for all state-action pairs.
2. Observe the current state s_t.
3. Choose an action a_t based on a policy, such as ϵ-greedy or softmax.
4. Take the action a_t and observe the next state s_{t+1} and reward R_{t+1}.
5. Update the Q-value function using the update rule: $Q(s_t, a_t) \leftarrow Q(s_t, a_t)+\alpha\Big[R_{t+1}+ \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big]$.
6. Set $s_t = s_{t+1}$ and repeat from step 3 until the end of the episode or termination of the task.

The Q-learning algorithm is guaranteed to converge to the optimal action-value function $Q^*(s, a)$ under certain conditions, such as the MDP being finite and the learning rate α decaying over time.

C.3 Deep Q Learning

Deep Q-learning is a variant of the Q-learning algorithm that uses a deep neural network to approximate the action-value function $Q(s, a)$ in a Markov Decision Process (MDP). The algorithm combines reinforcement learning with deep neural networks to enable learning in high-dimensional and continuous state spaces.

The Deep Q-learning algorithm can be summarized as follows:

Algorithm: Deep Q-learning

1. Initialize the replay memory buffer D with capacity N.
2. Initialize the Q-network with random weights θ.

3. Initialize the target Q-network with weights $\theta^- = \theta$.
4. For each episode $e = 1, 2, ..., E$ do the following:
   (a) Initialize the environment with initial state s_0.
   (b) For each step $t = 1, 2, ..., T$ do the following:
      i. With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$.
      ii. Execute action a_t and observe reward r_t and next state s_{t+1}.
      iii. Store the transition (s_t, a_t, r_t, s_{t+1}) in the replay memory buffer D.
      iv. Sample a mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from the replay memory buffer D.
      v. Compute the Q-learning target for each transition (s_j, a_j, r_j, s_{j+1}):
      $$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}.$$
      vi. Compute the loss between the predicted Q-value and the target Q-value: $L(\theta) = \frac{1}{B} \sum_{j=1}^{B} (y_j - Q(s_j, a_j; \theta))^2$.
      vii. Update the Q-network weights using stochastic gradient descent: $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$.
      viii. Every C steps update the target Q-network weights: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$.
      ix. Set $s_t = s_{t+1}$.

In the Deep Q-learning algorithm, the replay memory buffer D is used to store experiences in order to prevent overfitting and stabilize learning. The target Q-network is used to compute the target Q-value in the Q-learning update, and its weights are periodically updated from the Q-network to prevent target overestimation.

The Deep Q-learning algorithm has been successfully applied to various tasks, such as playing Atari games, controlling robots, and playing board games.

The Deep Q-learning algorithm uses experience replay and target networks to improve stability and convergence of the algorithm. Experience replay randomly samples transitions from the replay memory buffer to decorrelate the data and prevent overfitting. Target networks are used to stabilize the training by keeping a separate target network with fixed parameters and periodically updating it with the weights of the online network.

The Deep Q-learning algorithm has been successfully applied to various tasks, such as playing Atari games, controlling robots, and playing board games.