令和 5 年度博士前期課程

学位論文

Spaghetti grasping robot simulation using Deep Reinforcement Learning on pybullet

九州工業大学情報工学府

学際情報工学専攻

学籍番号　21677602

氏名　Promma Sornsiri

指導教官　林 英治

研究室　林英治研究室

Table of contents

1 Introduction 4
1.1 Motivation 4
1.2 Background 6
1.3 Purpose and outline 7
1.4 Overview of the dissertation 8
2 System overview 9
2.1 Choice of simulation 9
2.1.1 Unity 9
2.1.2 Pybullet 12
2.2 Soft body mesh 13
2.3 Arm manipulator 16
2.4 OpenAI Gym 17
2.5 Stable Baseline3 19
3 Related Theory 20
3.1 Reinforcement Learning 20
3.2 Markov Decision Process 22
3.3 Deep Reinforcement Learning 23
3.3.1 Proximal Policy Optimization 25
3.3.2 Soft Actor-Critic 26
4 Experiment 28
4.1 Implemented system 28
4.1.1 Soft object setup 28
4.1.2 Timesteps and episodes setting 30
4.1.3 Observation configuration 32
4.1.4 Action configuration 33
4.1.5 Reward function structure 33
4.2 Results 37
4.2.1 Training by Proximal Policy Optimization, using direct observation 37
4.2.2 Training by Soft Actor-Critic, using direct observation 39
4.2.3 Training by Proximal Policy Optimization, using image observation 41
4.2.4 Training by Soft Actor-Critic, using image observation 43

5 Discussion 45
5.1 Model results comparison 45
5.2 Reward graph pattern 48
5.3 Training speed 49
5.4 Challenges 49
5.4.1 Object parameters setting 49
5.4.2 Soft object stability 51
5.4.3 Object interaction detection 52
6 Conclusion 53
6.1 Summary 53
6.2 Future work 54
6.2.1 Object parameter setting 54
6.2.2 Computer vision 54
6.2.3 Adapt custom policy 55
6.2.4 More simulation choices 55
7 References 56

1 Introduction

1.1 Motivation

Robots have been a part of human society for decades, but their importance has grown exponentially in recent years. With the rapid advancements in technology and artificial intelligence, robots are becoming increasingly important in various industries, such as food industry. The food industry is constantly evolving, with new technologies and processes being developed to meet changing consumer demands. As shown in Figure 1.1, the diagram presents an overview of the Food Robotics Market, highlighting the various applications of robots within the food production industry. It clearly indicates a growing demand for these automated systems, reaffirming their significance in this sector.

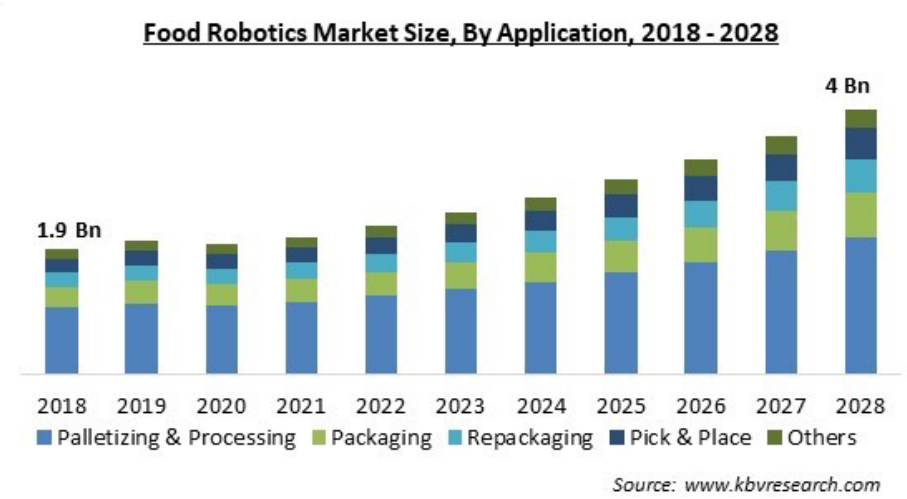

Figure 1.1 Food Robotics Market size, 2018-2028 [1]

One area of the food industry that is being revolutionized by technology is food packaging, and robots are playing a critical role in this transformation. Food packaging is an important process in the food industry. It ensures that food products are protected from contamination and packaged equally. However, traditional packaging methods can be time-consuming, labor-intensive, and prone to errors. This is where robots come in, offering a range of benefits that can transform the food packaging process. Robots can perform repetitive tasks with greater precision and consistency than human workers, reducing the risk of errors and improving the overall quality of the packaging. This results in faster production times and lower costs for food manufacturers.

Figure 1.2 Food packaging automation line with pick and place task

Still, some tasks are hard to be done repeatedly, such as to pick and place food into the box, known as Bento in Japan. This task may seem simple for humans, but they are quite complex for robots, as they require the robot to accurately perceive and manipulate objects in an unstructured environment. The pick and place task are a complex and dynamic process that requires the robot to perceive the environment, identify objects, and manipulate them accurately. Manually coding the rules and algorithms for each step of this process is not practical, as the robot needs to be able to adapt to different objects, positions, and orientations. So, Artificial Intelligence (AI) was introduced due to more flexibility and adaptable approach to pick and place tasks. To specify, machine learning algorithms such as Reinforcement Learning, can enable the robot to learn from experience and improve its performance over time.

Reinforcement Learning is a type of artificial intelligence that can be used to train robots to perform complex tasks like pick and place operations. In pick and place tasks, the robot must be able to perceive its environment, identify the object it needs to pick up, and then manipulate the object to place it in the desired location. These tasks can be challenging for robots, especially when the objects are of different shapes and sizes or are in different orientations. Still, Reinforcement Learning allows the robot to learn from experience, by receiving feedback on its actions and adjusting its behavior over time. It helps the robot learn through trial and error, which empowers the food robot to improve its performance over time, optimizing its pick and place actions based on previous experiences. Together with simulation, robots can undergo rapid training iterations, honing their skills and strategies before transitioning to real-world operations. Simulation serves as a valuable tool for accelerating the learning curve, facilitating faster adaptation to real-world scenarios.

1.2 Background

Nowadays, the robot arm has been widely used in food production lines for various purposes such as picking and placing the object for packaging the food box, commonly known as Bento in Japan. Within each Bento box, a diverse range of food items, including rice, chicken, and fruits, must be delicately handled. While many of these items can be effortlessly grasped by robot arms, others may be challenging when dealing with ingredients like pasta or spaghetti. These items pose difficulties in accurately estimating their weight and achieving consistent grasping, making it a complex task to accomplish every time. Currently, human intervention is still necessary to tackle these challenges. When it comes to ingredients like spaghetti, it's easy for them to slip away because of their deformable characteristic. On the other hand, grasping by putting too much force can damage the spaghetti as well. Furthermore, estimating its weight accurately is pretty difficult. Ensuring that each box has an equal amount of spaghetti can be quite challenging when relying on a robot arm.

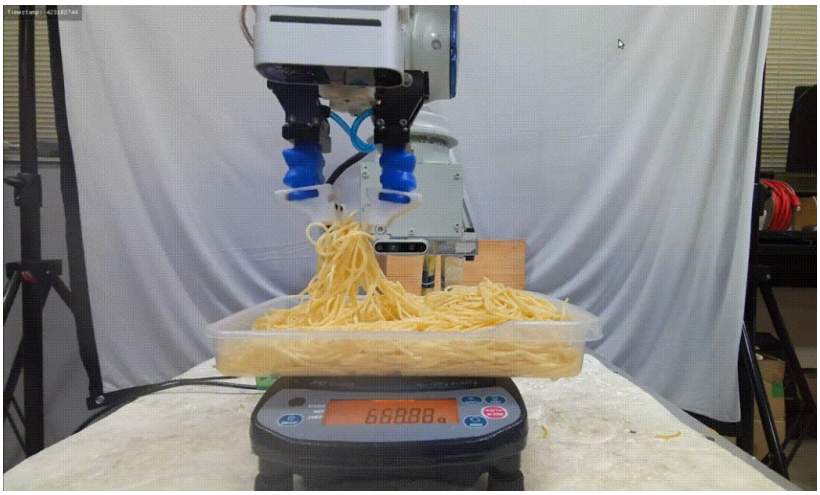

Figure 1.3 Training robot arm to grasp the spaghetti at the exact weight in real environment [2]

To enable the robot arm to grasp spaghetti at the desired weight, we employed a technique called Deep Reinforcement Learning. In this approach, a physical robot arm utilizes an RGB-D camera to detect the tray of spaghetti [2]. It then isolates the specific area of the spaghetti using image masking techniques and attempts to grasp it repeatedly. During each grasp, the weight is measured, and a reward is calculated based on how close it is to the expected weight. This reward serves as feedback to the learning process, allowing the robot arm to improve its performance through trial and error. However, training the robot arm in a real-world

environment presents some challenges. One of these challenges is the changing condition of the spaghetti over time. Initially, the boiled spaghetti is slippery and doesn't stick together. But as time passes during the training session, the spaghetti becomes drier and more prone to tearing apart and sticking together. This variance in the spaghetti's state can lead to discrepancies in the grasping results, even if the robot arm attempts to grasp the same amount each time. Therefore, it becomes necessary to boil fresh spaghetti every half an hour during the training process, which is inconvenient for the training process.

Regarding the challenges mentioned, we plan to shift our training approach from a real-world environment to a simulation. By utilizing simulation, we can overcome the obstacles more effectively. In the simulated environment, we can prepare the necessary conditions and settings such as the robot arm and the soft object which acts like spaghetti, allowing us to accelerate the training process without requiring human intervention. This approach will enable us to fine-tune and optimize the robot's grasping capabilities before deploying it in the real-world production line by using Deep Reinforcement Learning same as previous one.

1.3 Purpose and outline

According to all the issues above, the purpose of this research is to create an environment for spaghetti grasping robot and train with Deep Reinforcement Learning to make the robot able to grasp the object which represents the spaghetti in the simulation. Rather than using solid objects, we focus on creating a soft object that mimics the characteristics of real spaghetti. The robot arm should be capable of detecting the position of the object and performing a gentle grasp without causing damage to it.

While there have been successful studies on using robot arms to grasp various shapes of solid objects, there is limited research on soft-body objects, making it a more challenging task. When grasping solid objects, the main factor typically revolves around precise positioning. However, when grasping soft objects like spaghetti, additional considerations such as proper force, elasticity, and shape both before and after touching them.

1.4 Overview of the dissertation

In the first chapter, we have described the motivation and background for this research which contains the application of robot technology in the food industry and how to apply Artificial Intelligence such as Reinforcement learning to train the robot to do various tasks in the real-world situation. Furthermore, the obstacles for training the robot arm in the physical environment have also been explained, together with the purpose of the research by using simulation to solve those problems.

The second chapter will describe in detail the system configuration such as the simulation platform and the reason that we chose it, how to create the soft body object that has similar characteristics to the spaghetti, the robot arm in the simulation, and the framework like OpenAI gym, together with stable baseline that are the main parts using for training with Deep Reinforcement Learning.

The third chapter of the research will introduce and present the relevant theoretical concepts that form the foundation of the research. It will provide an overview of the key theories and concepts necessary to understand the research topic, covering three main areas of focus which are Reinforcement Learning, Markov Decision Process and Deep Reinforcement Learning.

The fourth chapter will outline the details of how the experiment was conducted, including the methodology and procedures employed. It will present the results obtained from the experiment, along with an analysis of the findings.

The fifth chapter will discuss and interpret all of the results. It will analyze the performance of different models and configurations, highlighting their strengths and weaknesses. Moreover, it will address the challenges encountered during the training process as well.

The sixth chapter will serve as the conclusion of the study, providing a comprehensive overview of the outcomes, including the achievements, and limitations of the research. Additionally, the chapter will offer suggestions for further development and improvement of the research topic for enhancing the efficiency of the proposed approach.

2 System overview

2.1 Choice of simulation

2.1.1 Unity

Choosing the suitable simulation is essential, especially when we specify the object as a soft body. At first, we experimented with another simulation platform like Unity. Unity is a game development engine that has gained widespread popularity in the gaming industry, providing a set of tools and features to create interactive and immersive experiences across various platforms. With its impressive level of physics, it can also be used for physics simulation, including in the robotic field. The built-in physics engine enables the simulation of robot movements, interactions, and physical behaviors. Unity also allows the customization and extension of the simulation through scripting, making it possible to control robot behaviors, implement algorithms, and integrate external libraries or plugins. Various kinds of sensors commonly used in robotics, such as cameras, lidar, and proximity sensors, are also provided. This enables the development and testing of perception algorithms, object detection, localization, and mapping techniques in a controlled virtual environment. Furthermore, Unity provides important extensions such as ML-Agent, which is used to create environments for Deep Reinforcement Learning. Additionally, there are assets like Obi Rope that can be used as spaghetti-like objects for training purposes.

- **ML agent**: Not only can Unity be used for normal simulations, but it can also be utilized in Machine Learning. In Unity, it is possible to create scenes as environments where agents can interact and learn through Reinforcement Learning, imitation learning, and other machine learning techniques [3]. It provides tools for defining agent behaviors, setting up reward structures, and visualizing the training process. ML-Agents supports a wide range of AI techniques, including Reinforcement Learning, which allows agents to learn through trial and error. Using Reinforcement Learning, agents interact with the environment and receive feedback in the form of rewards or penalties based on their actions. Over time, they learn to maximize their cumulative rewards by discovering optimal strategies and behaviors. Along with the sensors mentioned earlier, agents can handle high-dimensional data, such as images or sensor readings, and feed them forward

as inputs to deep neural networks, which serve as the underlying models for agent decision-making.

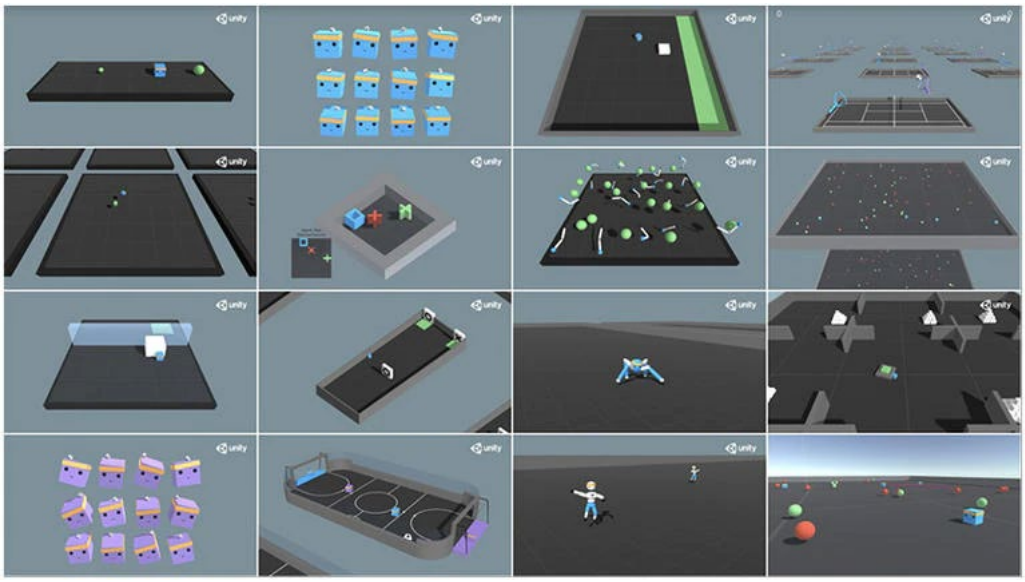

Figure 2.1 Example scene of the environment that training by Reinforcement Learning in Unity [3]

- **Asset:** One of the important reasons why we chose Unity is Unity provided asset store, which provides a variety of assets like 3D models, textures, scripts, and plugins. Because our main objective is to simulate the robot's ability to grasp a soft object, it is important to have an object with suitable properties and accurate physics. Initially, we tried to create the object in both Unity and other 3D modeling software such as CAD, Blender, and MeshLab. However, there were many obstacles to making it look like the real object, such as the required spaghetti noodle. We searched for a ready-to-use model and found an asset named Obi Rope [4].

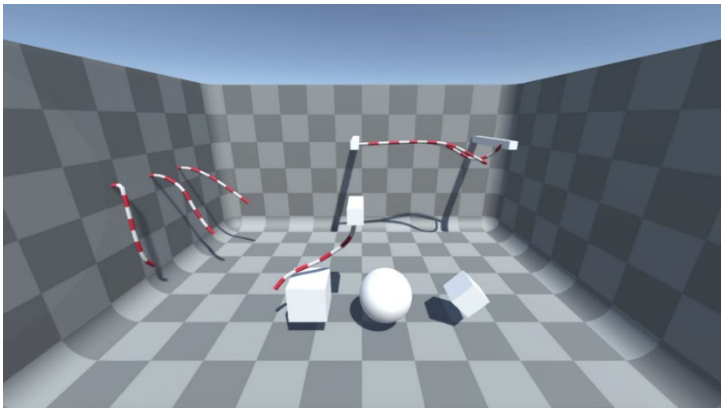

Figure 2.2 Obi Rope asset [4]

Obi Rope is a Unity asset that creates realistic simulations of ropes and flexible objects within Unity. It is part of the Obi Physics framework, designed specifically for simulating cloth, fluids, and soft bodies. With Obi Rope, it is possible to easily incorporate dynamic and interactive ropes into games or applications. It utilizes advanced physics algorithms to accurately simulate the behavior of flexible objects, allowing for realistic animations and interactions. It supports various customizable properties such as stiffness, bending, and damping. The asset also includes features like self-collision detection, constraints, and attachment points, enabling ropes to interact with themselves and other objects in the environment. Additionally, Obi Rope supports scripting, allowing for seamless integration with other Unity components or scripts to create complex and interactive simulations.

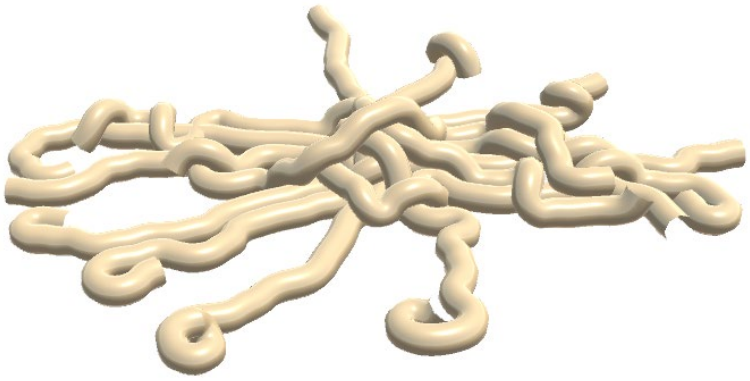

Figure 2.3 Soft object which represent the spaghetti, created by Obi rope asset

After trying to adjust various parameters, Obi Rope amazingly can act like a spaghetti-like object. It has self-collision, is able to move smoothly, and has great performance even when adding a large number of objects to the scene. Unfortunately, it still lacks the most significant feature, which is the ability to be picked up. Obi Rope is primarily designed for simulating the behavior of ropes and flexible objects, but it does not have built-in capabilities for grasping or manipulating objects because it is not based on rigid bodies. Its focus is on providing realistic animations and interactions for ropes and similar flexible structures within Unity. To solve this problem, we can indicate the center point for grasping and using other objects, which may be another 3D shape like a sphere, that have normal collision. In this case, the robot will be able to grasp the object. However, this approach would not meet our objective, which requires grasping the soft object. Additionally, it will

lead to other problems, such as a too specific grasping position, whereas a long object should be able to be grasped at any area of the object.

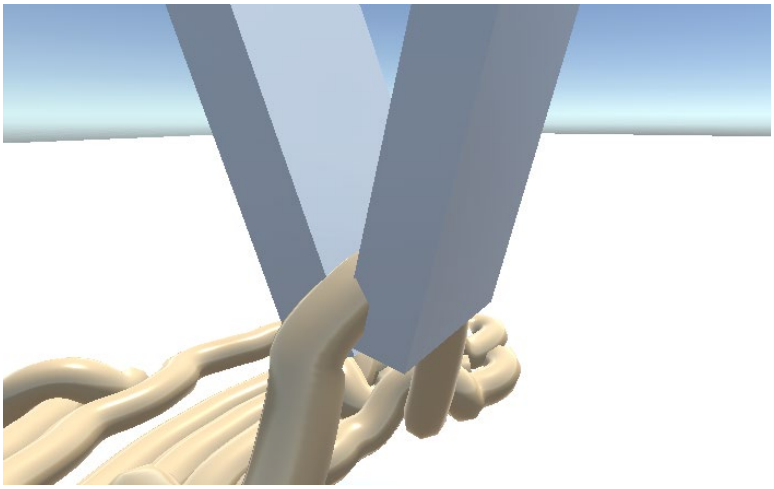

Figure 2.4 Soft object is being grasped in Unity with low accuracy of collision

Considering these limitations, we try to find another solution by looking for another asset that can be used as a soft object. Still, there are no objects that have the suitable properties as we desire. So, we chose to change the simulation platform instead, which is pybullet.

2.1.2 Pybullet

Regarding the issue in Unity, we decided to choose pybullet as the simulation platform used for this research. Pybullet is an open-source physics simulation that is widely used in the robotic field. It can be used to recheck the robot's motion with the environment before deploying them on a physical robot. Not only the articulated bodies for the robot, pybullet is also capable of importing various formats of mesh, providing a high degree of customization and flexibility through plenty of parameters to make the object in simulation close to the real one. This simulation can simulate both robot, objects and environment with high accuracy of collision, making it suitable for the environment that needs the interaction between the robot itself and the object in the environment such as the pick and place task.

Not only solid objects, but it can import objects as soft bodies as well, which makes it stand out from other simulation platforms. Designed for high-performance computing, pybullet is capable of simulating complex soft body physics calculations in real-time, including realistic deformations, dynamics, and accurate collisions. The object can be

adjusted in its properties such as elasticity, stiffness and friction, able to make the object in a simulation similar to the real-world one. Moreover, pybullet also supports a wide range of robot models and environments and can be easily integrated with other software tools such as ROS (Robot Operating System), OpenAI Gym and Stable Baseline which also make it suitable for training the robot with Reinforcement Learning.

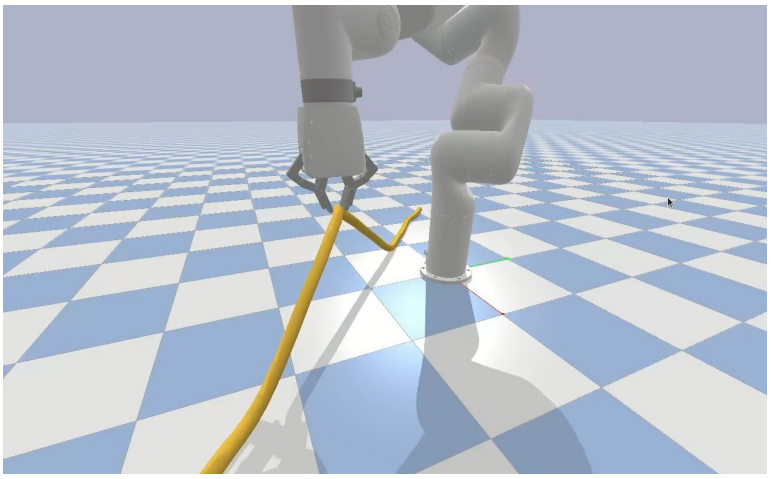

Figure 2.5 Soft object in pybullet able to be grasped by the robot arm

2.2 Soft body mesh

After choosing pybullet as the simulation platform, we need to create the suitable object which will be represented as the spaghetti. Still, to import object as soft body into pybullet, the object needs to be in VTK format which refers to a 3D geometric model that is represented using the Visualization Toolkit (VTK) library's data file format. VTK format allows for efficient storage and exchange of 3D models, supports a wide range of geometric primitives and data attributes, and is widely used in various fields for visualization, analysis, and simulation of 3D graphics and scientific data. In VTK, an object is typically represented as a collection of geometric primitives, such as points, lines, polygons, or volumes, along with associated attributes such as colors, textures, and scalar or vector values. These geometric primitives are organized into data structures such as unstructured grids, structured grids, and multi-block datasets.

To create an object in VTK format, there are 3 main steps [5]:

1. **Create plain 3D model**

Create a 3D model of the object in a 3D modeling software, such as Blender. The cylinder model should be saved as STL format.

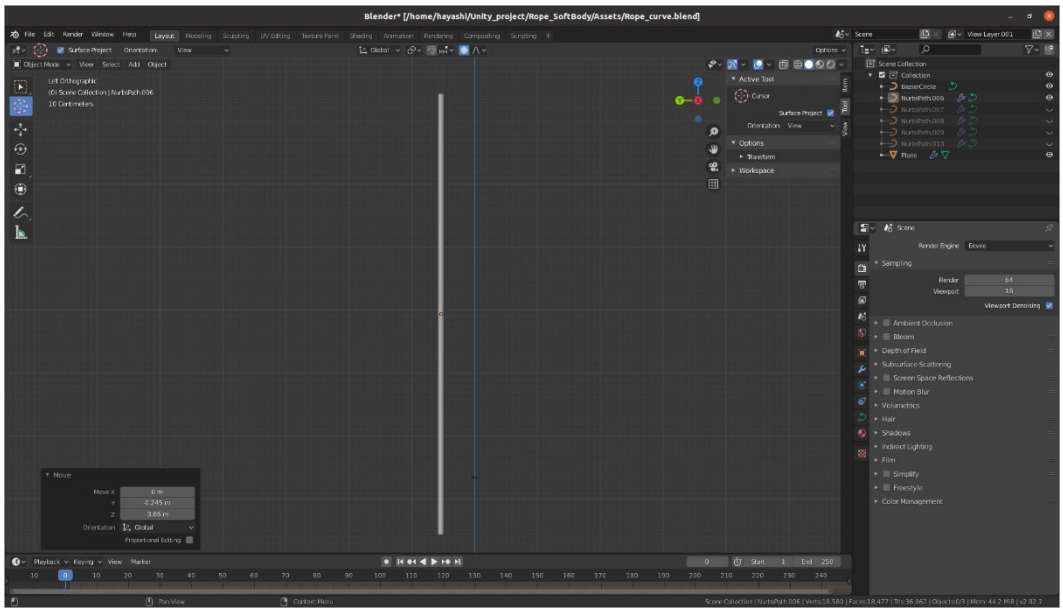

Figure 2.6 Cylinder model created in Blender

2. **Convert 3D model to tetrahedral mesh using Tetwild**

In order to make it have volume and be usable in the simulation, the model needs to be in tetrahedral mesh form. Tetrahedral mesh is a type of three-dimensional mesh structure commonly used in computer graphics, scientific simulations, and engineering applications. This kind of geometry consists of a collection of tetrahedra, which are four-sided polyhedral with triangular faces. Each tetrahedron is defined by its vertices, representing points in space, and its triangular faces that connect these vertices. It is widely used in various fields for simulations and analysis that require a volumetric representation, as tetrahedra provide a good approximation of the underlying continuous space. Additionally, tetrahedral meshes can capture the volumetric properties of objects, such as volume, density, and material properties, enabling accurate physical simulations and calculations.

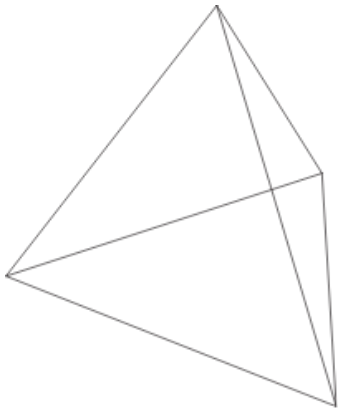

Figure 2.7 Tetrahedral shape [6]

In this step, we used TetWild [7] to generate high-quality tetrahedral meshes. TetWild provides a tetrahedral meshing technique that is easy to use and can directly convert any shape of mesh into a volumetric mesh. For installation and usage instructions, please refer to this link. The result from this step will be a 3D model in MSH format, which stores information about the mesh geometry, element connectivity, and other properties related to the mesh.

3. Convert MSH file to VTK file using GMSH

The last step involves making the 3D model importable to pybullet as a soft body, which requires it to be in VTK file format. We accomplish this by using GMSH, a user-friendly meshing tool that allows for the creation and manipulation of meshes. GMSH supports the generation of meshes for both two-dimensional (2D) and three-dimensional (3D) geometries, offering a wide range of element types, including tetrahedra.

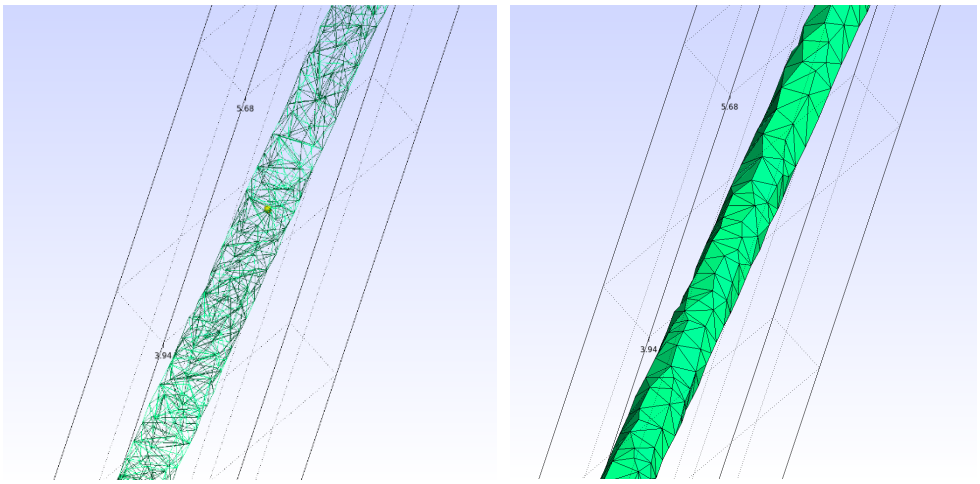

Figure 2.8 Cylinder object which contain of tetrahedral mesh in GMSH

After importing the model, we directly export it from GMSH without changing any parameters or properties. GMSH generates a VTK file that contains information about the tetrahedral mesh geometry and connectivity. This exported VTK file can be directly used in pybullet as a soft body for simulation purposes.

To reduce mesh in order to improve calculation time, we use MeshLab to simplify mesh before we convert it to tetrahedral mesh in TetWild. Firstly, we import mesh to MeshLab and select 'Filters', then 'Remeshing, Simplification and Reconstruction'. We choose 'Simplification: Quadric Edge Collapse Decimation' and reduce mesh as the percentage.

While reducing the mesh complexity can improve calculation time during training the agent in pyBullet, it is important to consider the trade-off between computation efficiency and the flexibility of the object during grasping. If the mesh contains too few vertices and faces, the object may become less flexible, affecting the ability to grasp and adjust its parameters effectively. In the pyBullet library, the mesh used for an object has a resolution of around 600 points. After converting the mesh to the VTK format from software like GMSH, we suggest maintaining a suitable number of points close to the original resolution, to ensure that the object does not significantly increase computation time while still providing a sufficient resolution for effective grasping. The balance between mesh complexity and computational efficiency is crucial in creating a realistic and flexible environment for training the agent. Finding the optimal number of points in the mesh that allows for efficient calculations while maintaining sufficient resolution for effective grasping, is essential in achieving successful results.

2.3 Arm manipulator

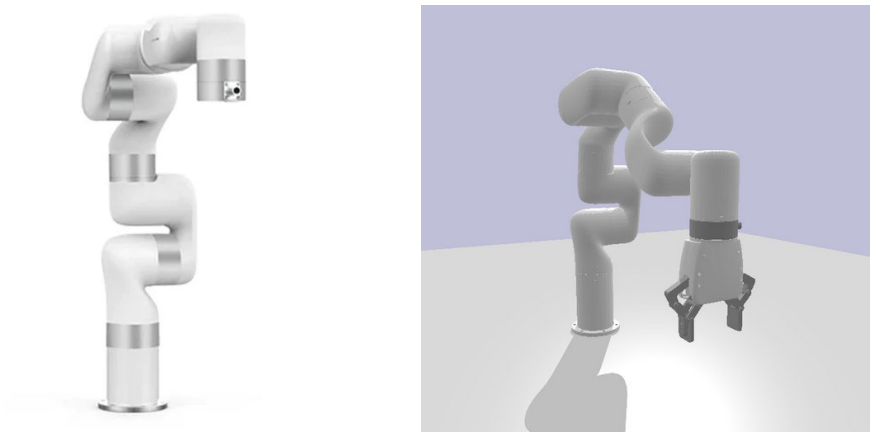

Figure 2.9 Xarm7, both physical robot arm and the model with gripper in simulation [8]

The robot arm that uses in this research is Xarm7, also known as XArm 7-DOF Robotic Arm. It is a versatile and advanced robotic arm designed for various industrial and research applications. The XArm7 is equipped with seven degrees of freedom (DOF), which refers to the number of independent movements or axes the arm can perform. This flexibility allows it to mimic human-like movements and perform a wide range of tasks with precision and dexterity. In the real-world environment, the payload weighs approximately 3.5 kg, and the overall weight of the system is around 14 kg. The working space of the system is approximately 103 centimeters in height and 70 cm in horizontal reach. In the simulation environment, the XArm7 is equipped with a pair of grippers attached to the end effector, allowing it to grasp objects.

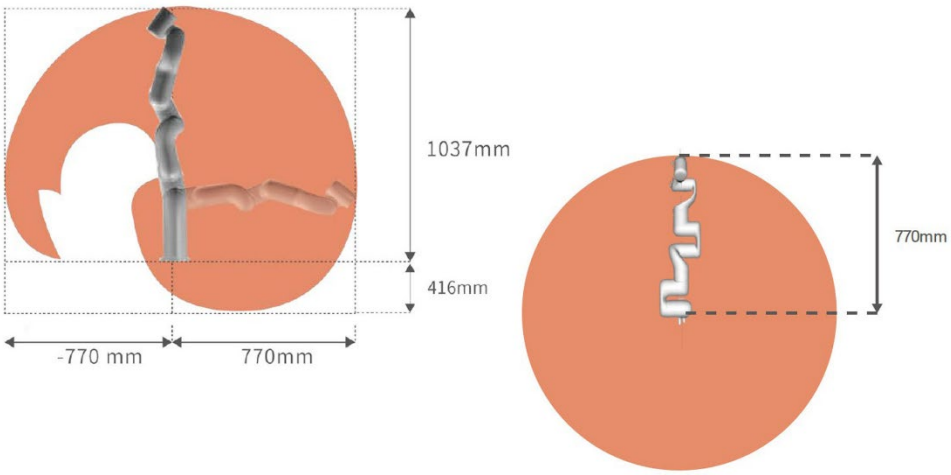

Figure 2.10 Xarm7 working space in real-environment [8]

2.4 OpenAI Gym

To construct an environment that is able to integrate with Reinforcement Learning, we use OpenAI Gym, which is an open-source toolkit for developing and comparing Reinforcement Learning (RL) algorithms that provides a collection of environments that is trainable with Reinforcement Learning [9]. Its standardized environments and resources make it easier to compare algorithm performance and develop more advanced AI systems. OpenAI Gym create the concept of the RL training process by dividing the training process into episodes in which the agent will randomly take action in each timestep and be rewarded according to the result. The goal for each episode is to maximize the total reward with the least number of episodes. The agent experience will come from a series of episodes and be improved repeatedly to get more rewards.

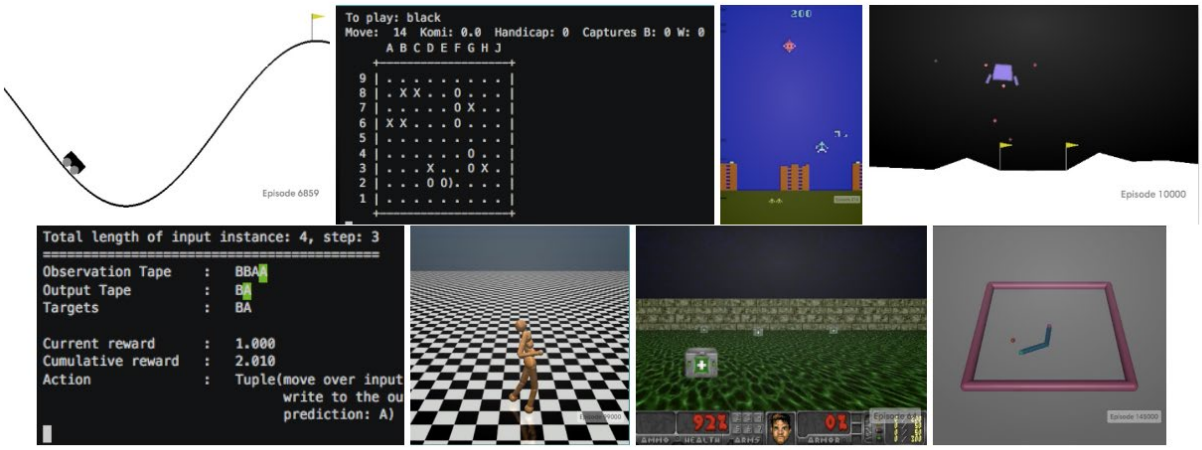

Figure 2.11 Example of environments which are part of OpenAI Gym [9]

Throughout the course of interaction, the environment may reach a terminal state, often due to an undesirable event like a robot crash. In such instances, the environment will be reset to initiate and continue the learning process. To signify the occurrence of a terminal state, the environment sends a "done" signal to the agent. Still, a “done” signal is not only because of the failures but it also can be triggered after a fixed number of timesteps or when the agent completes a particular task within the environment.

```
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
   action = policy(observation)  # User-defined policy function
   observation, reward, terminated, truncated, info = env.step(action)

   if terminated or truncated:
      observation, info = env.reset()
env.close()
```

Figure 2.12 Example of environment implementation with OpenAI gym

In this research, we utilize a custom environment that has been implemented in pybullet. This environment includes the XArm7 robotic arm [10], which will be used as the agent. However, we have made certain customizations to various modules within the environment to align with our specific goals. These customizations include modifications to the object setup, reward function design, timestep settings, and observation configurations, to make it well-suited for our research objectives which are going to be explained in chapter four.

2.5 Stable Baseline3

There are several libraries and frameworks that can use for implementing Reinforcement Learning algorithms. However, using with OpenAI gym framework, Stable Baseline becomes the best choice because it is built on top of OpenAI Gym, which is a toolkit for developing and comparing Reinforcement Learning algorithms. Stable Baseline provides a collection of state-of-the-art algorithms, which are pre-implemented and ready to use for training and testing Reinforcement Learning agents.

Stable Baseline also supported a wide range of algorithms, both on-policy and off-policy such as Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), Trust Region Policy Optimization (TRPO), Soft Actor-Critic (SAC), and more. These algorithms are designed to handle different types of action spaces, including both continuous and discrete spaces. Moreover, they are also known for their stability and robustness. They incorporate techniques such as value function approximation, policy optimization, and exploration strategies to ensure effective learning and convergence. Still, Stable Baselines also offer extensibility. The algorithm is able to customize and extend to suit the specific needs which empowers the researchers to experiment with novel ideas, incorporate domain-specific knowledge, or adapt the algorithms to unique problem domains.

3 Related Theory

3.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that focuses on the interaction between an agent and its environment, which involves the concept of learning through trial and error. Unlike supervised and unsupervised learning, Reinforcement Learning does not require pre-existing labeled data for training. Instead, the agent generates data in real time as it interacts with the environment. It is concerned with how an agent can learn to make decisions or take actions in order to maximize its cumulative reward over time.

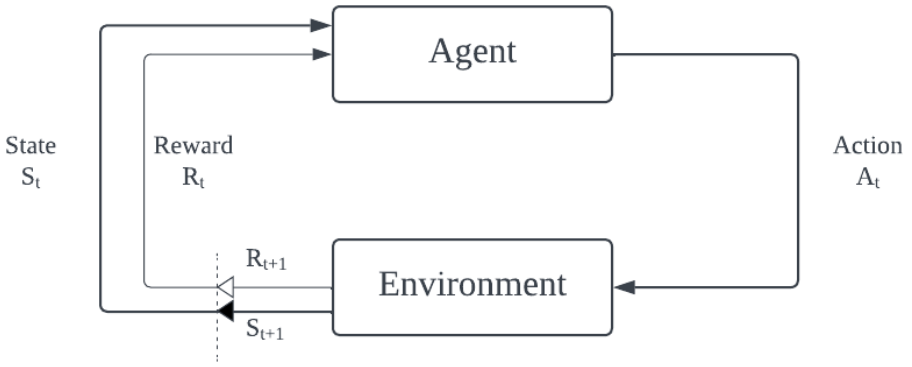

Figure 3.1 Reinforcement Learning process diagram

From Figure 3.1, there are 5 key terms in Reinforcement Learning which are agent, environment, action, state and reward. The agent starts by interacting with the environment, taking actions based on its current state, and receiving feedback in the form of rewards or penalties depending on the success or failure of the agent's actions. By continuously repeating this process, the agent tries to find a policy that maximizes the expected cumulative reward over the course of its interactions with the environment. The objective of the agent is to learn an optimal policy, a strategy or set of rules that dictate which action to take in each state, that maximizes its long-term expected reward.

In Reinforcement Learning, an agent learns through a process of exploration and exploitation. Exploration refers to the agent's ability to actively seek out and gather new information about its environment. By exploring, the agent can acquire new information and update its knowledge about the environment, potentially leading to improved long-term performance. Exploitation is the ability to select actions or strategies based on the agent's current knowledge to maximize its immediate rewards. Exploitation focuses on maximizing

short-term rewards by taking actions that it has already learned to be valuable. The agent exploits the knowledge it has gained from past interactions with the environment to make decisions that are likely to yield high rewards.

However, exploration may come at the cost of immediate rewards, as the agent might need to sacrifice short-term gains to gain better knowledge about the environment, while exploitation may prevent the agent from discovering potentially better actions or strategies that have not been thoroughly explored. Therefore, the agent needs to effectively combine exploration and exploitation so that it can improve its policies over time, achieving better performance in more complex environments.

To solve the RL problem, the approach can be divided into 3 common strategies, which are:

1. Policy-Based approach: In this approach, the main goal is to directly optimize the policy of the agent. The policy defines the agent's decision-making process, specifying which actions to take given a certain state. The agent learns to improve its policy through trial and error, using various policy optimization algorithms like Proximal Policy Optimization (PPO) or Soft Actor-Critic (SAC).
2. Value-Based approach: Instead of directly optimizing the policy, the agent focuses on optimizing a value function. The value function estimates the expected future reward the agent can receive from a given state and action. Q-learning and Deep Q-Networks (DQNs) are popular value-based methods that try to maximize the expected cumulative reward.
3. Model-Based approach: In this approach, the agent builds a model of the environment to predict the consequences of its actions. It tries to learn the dynamics of the environment, which helps in planning and decision-making. Model-based methods can include techniques like Monte Carlo Tree Search (MCTS) or model-based Reinforcement Learning algorithms.

3.2 Markov Decision Process

Markov Decision Process (MDP) is a fundamental algorithm that provided mathematical framework used in Reinforcement Learning and decision-making problems.

An MDP consists of 5 main components which are

1. States: A set of possible states that the system can be in. Each state represents a particular configuration of the environment.
2. Actions: A set of actions available to the agent. The agent selects an action based on the current state.
3. Transition Probabilities: For each state-action pair, there is a probability distribution over the possible next states. These probabilities represent the likelihood of transitioning from the current state to the next state when the agent takes a specific action.
4. Rewards: The agent receives a reward after each action based on the current state and the action taken. The goal of the agent is to maximize the cumulative rewards over time.
5. Policy: A policy determines the agent's behavior by mapping states to actions. It defines the strategy for selecting actions based on the current state.

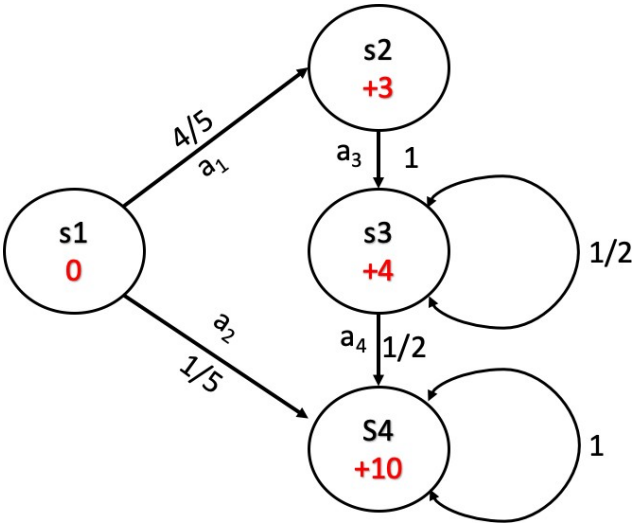

Figure 3.2 Example of Markov Decision Process schematic diagram

In an MDP, a decision maker or agent interacts with an environment in a series of discrete time steps. At each time step, the agent observes the current state of the environment and selects an action randomly to take. The environment then transitions to a new state based on the action taken, and the agent receives a reward or a numerical feedback signal. Markov Decision Process can be represented as mathematic equation which relates the value of a state or state-action pair to the expected cumulative rewards obtained by following a given policy as

$$V(s) = R(s) + \gamma * \Sigma P(s' \mid s, a) * V(s'),$$

where:

- V(s) is the value function for state s, representing the expected cumulative rewards starting from state s and following a given policy.
- R(s) is the immediate reward obtained when transitioning to state s.
- γ (gamma) is the discount factor, a value between 0 and 1 that determines the importance of future rewards compared to immediate rewards.
- P(s' | s, a) is the transition probability, representing the probability of transitioning to state s' when taking action a in state s.
- V(s') is the value function for state s', representing the expected cumulative rewards starting from state s' and following a given policy.

From the above equation, it states that the value of a state is equal to the immediate reward obtained in that state, plus the discounted sum of the expected values of the subsequent states, which means the uncertainty of the environment by considering the transition probabilities and future expected rewards.

3.3 Deep Reinforcement Learning

Reinforcement Learning is an approach to make the agent learn from trial and error, receiving the reward according to its action. Still, Traditional RL algorithms often struggle with problems that have large or continuous state spaces, as the size of the state space grows exponentially with the number of dimensions, while as, deep learning is capable of automatically learning hierarchical representations from raw sensory inputs, making them well-suited for handling complex and high-dimensional input spaces. Integrating Reinforcement Learning and deep learning, it will increase the ability to handle high-dimensional problems.

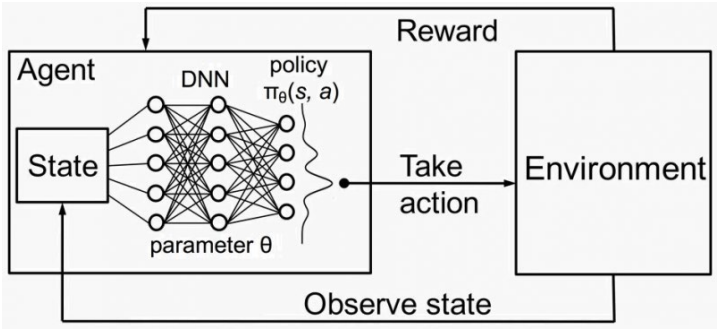

Figure 3.3 Deep Reinforcement Learning process diagram

The workflow of DRL involves the agent iteratively interacting with the environment like normal RL problem, observing states, taking actions, and receiving rewards. The agent then uses this experience to update its neural network to improve its policy and decision-making capabilities. This process continues until the agent learns a good policy that maximizes the cumulative reward over time.

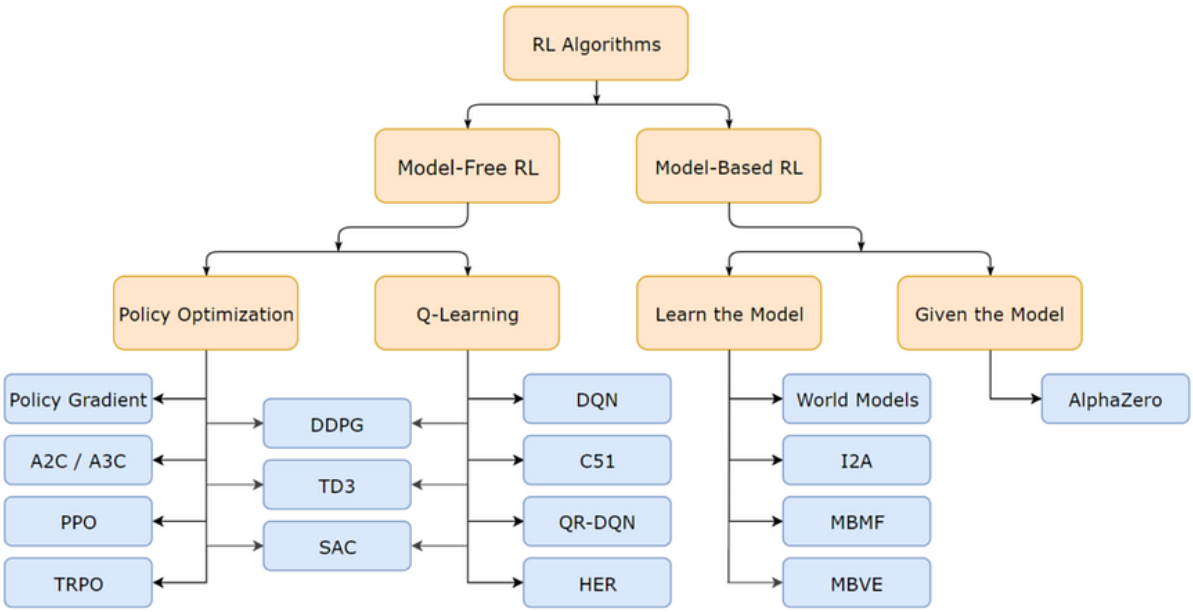

Figure 3.4 Deep Reinforcement Learning algorithm taxonomy, defined by OpenAI [11]

As shown in the above diagram, in Deep Reinforcement Learning, there are two main types of algorithms which are model-based and model-free algorithms. As explained in Reinforcement Learning section, Model-based algorithms involve the agent trying to learn a model of the environment, such as transition probabilities and rewards, and then using this model to make decisions. However, these models can be less flexible when facing new and complex environments. On the other hand, model-free algorithms do not try to learn an explicit model of the environment. Instead, they directly learn from the interactions with the environment.

Furthermore, model-free algorithms can be categorized into on-policy and off-policy algorithms. In our research, we select Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) as two examples that represent each category.

3.3.1 Proximal Policy Optimization

Proximal Policy Optimization algorithm or PPO is an on-policy model-free DRL algorithm, that uses a policy gradient approach to optimize a policy by avoiding policy to not change too drastically from one iteration to the next which will make training more stable. On-policy algorithms like PPO update their policy using the most recent data they collect while interacting with the environment. The agent learns from the data generated by its current policy and tends to be more conservative in its updates. It avoids the need for a replay buffer, which simplifies the implementation and reduces memory requirements. PPO will perform policy updates while ensuring that the new policy remains close to the previous one to avoid large policy changes that might make the training become unstable. This concept is called **clipped surrogate objective** as its objective is "clipped" to prevent large changes to the policy distribution. It will approximate the policy's performance improvement during updates which makes PPO maintains stability during training and avoids divergence issues. It also helps balance between exploiting the current policy and exploring new possibilities, leading to robust and effective learning in both continuous and discrete action spaces.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

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

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

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

typically via some gradient descent algorithm.
8: **end for**

Figure 3.5 Pseudocode of Proximal Policy Optimization

The agent collects experience by interacting with the environment using its current policy. Then uses this data to update the policy in a way that maximizes the expected reward. PPO ensures that the policy updates are performed in a stable and conservative manner, avoiding large policy updates that could lead to catastrophic results. This property makes PPO particularly well-suited for tasks that involve continuous control and complex environments, as it allows for more stable and reliable learning.

3.3.2 Soft Actor-Critic

Soft Actor-Critic or SAC is an off-policy model-free DRL algorithm. Normally, many Deep Reinforcement Learning algorithms will focus on maximizing reward function only, while SAC focuses on maximizing the entropy term as well. The entropy term encourages the policy to be stochastic, promoting exploration and robustness in learning. Off-policy algorithms like are more flexible as they learn from a combination of data generated by the current policy and data generated by an older policy which helps it to potentially learn from a wider range of experiences and be more sample-efficient. Its characteristics let it learn from a replay buffer, storing past experiences, enabling more efficient use of data and improved sample efficiency. During training, SAC samples mini-batches of data from the replay buffer and uses the current policy and value function to update the actor and critic networks via gradient descent. Additionally, SAC employs a variant of the target network technique to improve stability during training.

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
  for each iteration **do**
    for each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
    end for
    for each gradient step **do**
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
      $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
    end for
  end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 3.6 Pseudocode of Soft Actor-Critic [12]

SAC uses an actor-critic architecture, where the actor is responsible for selecting actions based on the current policy, and the critic evaluates the value of those actions. The actor acts as the policy network and is responsible for selecting actions based on the current policy. It takes the current state of the environment as input and outputs the agent's action. In the case of SAC, the actor network typically outputs a probability distribution over actions, making it a stochastic policy. This stochasticity helps the agent to explore different actions, which can be beneficial for learning in complex environments. The next component is the Critic or Value Network, which is used to evaluate the value of the actions taken by the actor. It estimates the expected cumulative reward, also known as the state-action value (Q-value), given a particular state and action. This component helps the actor by providing feedback on the quality of its actions, which guides the learning process.

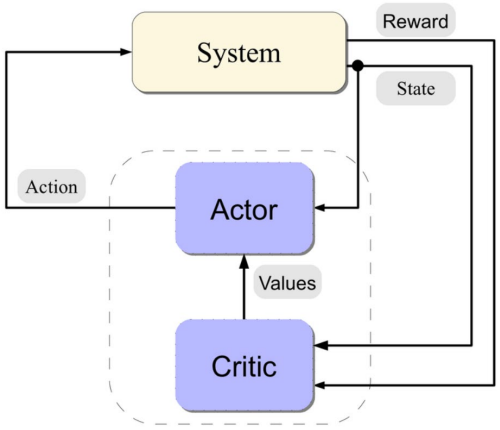

Figure 3.7 Actor-Critic architecture

With Actor-Critic architecture, it makes SAC become more stable learning, The critic's feedback can be used to update the policy and improve its decision-making. Moreover, using value functions in the critic network also add the ability for the exploration-exploitation trade-off, as it guides the actor to select actions that lead to higher expected rewards.

4 Experiment

4.1 Implemented system

The training environment is simulated in pybullet with OpenAI Gym as the framework, in order to train with Deep Reinforcement Learning. As mentioned in chapter 2, we use Xarm7 environment that has already implemented [10] and customize some configurations to make it suit our research purpose. There are several parts that we have adjusted, including both simulation and Deep Reinforcement Learning components.

4.1.1 Soft object setup

According to our research objective, we want to experiment on grasping the spaghetti by using soft object as the representer. We changed from a solid cube to a soft cylinder object that we have created in VTK format which has 500 double points, not too heavy for calculation time and not too less resolution for grasping.

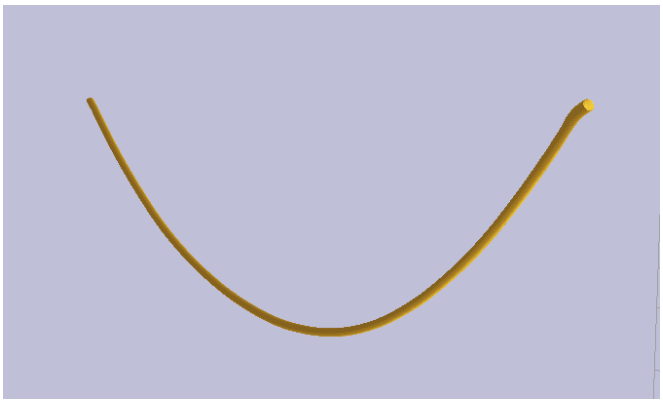

Figure 4.1 Object is soft and bendable after setting various parameters in pybullet

The object can be loaded into the simulation using the pb.loadSoftBody function, which offers various adjustable parameters. One of these parameters is "usingNeoHookean," which allows for simulating the behavior of elastic materials such as rubber or soft tissue. The Neo-Hookean model can be used in order to adjust the stretching and shearing properties of the material, providing an approximation of the stress-strain relationship. After applying Neo Hookean parameters, specific parameters within the Neo-Hookean model, such as Mu, Lambda, and damping, are adjusted to enhance the stability of the object, making it more suitable for interaction with the gripper.

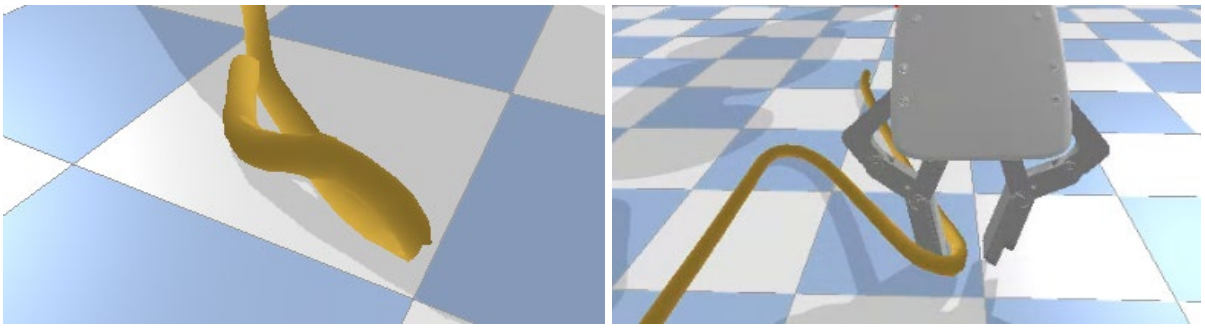

Figure 4.2 Object properties comparison. The object that did not apply Neo Hookean parameter, on the left image is less bendable compared to the right that have applied it.

```
def add_obj(self, pos, orientation):
    filename = "Rope.vtk"
    id = p.loadSoftBody("Rope.obj",
        simFileName=filename,
        basePosition=pos,
        baseOrientation = orientation,
        scale= 1,
        mass = 2,
        collisionMargin=0.005,
        useBendingSprings=1,
        springBendingStiffness = 0.001,
        useNeoHookean=1,
        NeoHookeanMu = 2000,
        NeoHookeanLambda = 1500,
        NeoHookeanDamping = 0.1,
        frictionCoeff=2,
        useSelfCollision = 1,
        )
    p.changeVisualShape(id, -1, flags=p.VISUAL_SHAPE_DOUBLE_SIDED)
    p.changeVisualShape(id,-1, rgbaColor =[1,.8,.3,1])
    print("Add soft object")
    return id
```

Figure 4.3 Parameter setting for the soft object that represent the spaghetti

To set the parameter for the object, we have tried different shapes of the object as well. We found that due to the spaghetti shape which is long and narrow, it tends to be more sensitive to an external force that the shape that is wider. There are fewer nodes inside the narrow object which make it crash easily. To solve this problem, we recommend setting the collision margin to extend the object area to make it wider and easier to grasp without damaging the object.

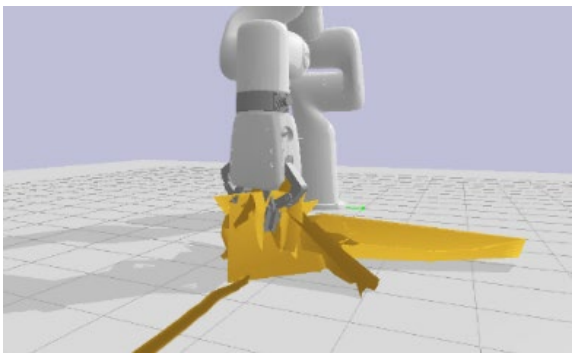

Figure 4.4 Unstable object can be crashed when the gripper touches it.

4.1.2 Timesteps and episodes setting

Each timestep, the robot moves to the assigned position which received from the Deep Reinforcement Learning algorithm as policy and receives the reward from its actions. After finishing a number of timesteps which we called episodes, the policy will be updated to make it able to get more rewards in the next one. Once a predefined number of timesteps, referred to as an episode, is completed, the policy is updated. The purpose of updating the policy is to improve its performance and enable the robot arm to achieve higher rewards in subsequent episodes.

Initially, each episode in the training process consisted of 3,000 timesteps. However, upon observing the behavior of the robot arm during training, it was noticed that the arm tended to remain stationary or not move towards the end of the episode. To address this issue, adjustments were made to the number of timesteps per episode. We adjust the number of timesteps to 300 to avoid excessively long episodes that would result in unnecessary time wastage and not too short to ensure that the robot arm had sufficient opportunity to explore the environment and learn from its interactions. In the first phase of training, for most of the total timesteps, the robot arm tries to reach the middle of the object. Once the robot arm has successfully learned to reach the middle of the object, the arm shifts its focus from simply reaching the object's center to grasping the object from an elevated position. The training algorithm encourages the arm to spend less time on reaching the object and instead prioritizes learning the appropriate grasping techniques.

Based on Figure 4.5, at the end of each action performed by the robot arm, the number of timesteps is checked to determine if it has reached the maximum limit of 300 timesteps. If the limit is reached, it indicates that the arm was unable to bring the object to the predefined height, resulting in a penalty. However, there is a distinction in the penalty depending on the extent to which the object was lifted. If the arm is able to lift the object even a small amount, it gets a lower penalty compared to cases where the object was not grasped or touched at all. This distinction in penalties encourages the arm to make progress towards lifting the object, even if it falls short of the predefined height.

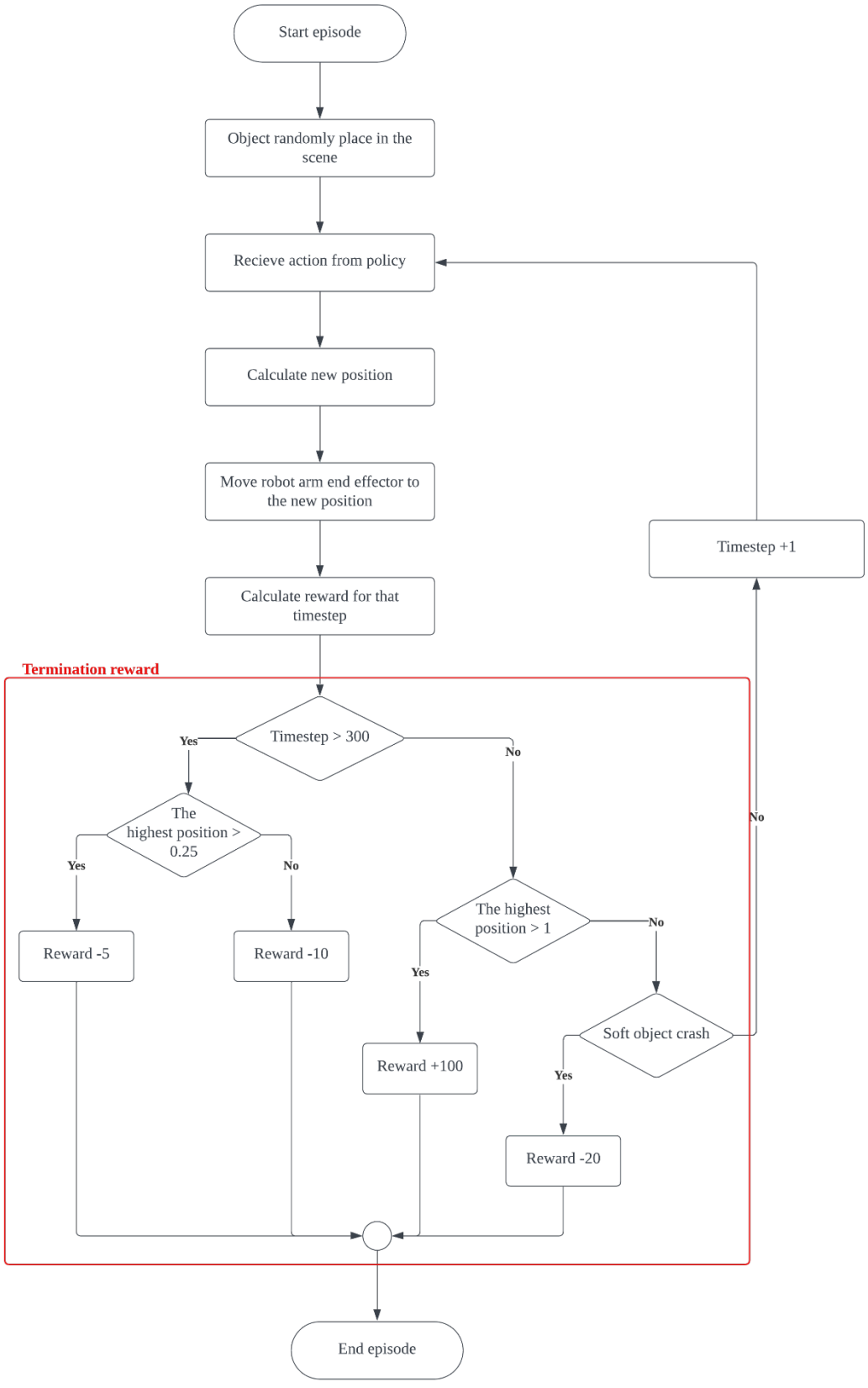

Figure 4.5 Flowchart diagram represent the process of each training episode

4.1.3 Observation configuration

Observations refer to the information that an agent receives from its environment which provides the agent with a representation of the current state or the relevant aspects of the environment necessary for decision-making. In our experiment, we employed two types of observations which are direct observation, using the information directly from the simulation, and the other one is using the image as the observation.

Direct observation involves using the positions of the robot arm, specifically the end effector, along with the positions of the object and the gripper width. This observation was collected as box-type observation in OpenAI gym, containing 7 elements. These positional values serve as the direct observations for the Reinforcement Learning process so that it is able to make decisions based on the current state of the robot arm, object, and gripper.

On the other hand, for image observation, we implemented a camera near the end effector to capture RGB images. These images have a resolution of 100x100 pixels. Before inputting them into the Reinforcement Learning process, the RGB images are converted to grayscale and also reduced resolution to 64x64 pixels. This conversion simplifies the data by reducing the dimensionality while still retaining the necessary visual information. By creating a wrapper function, the existing environment can be extended to handle image observations without requiring significant modifications or the creation of a separate environment.

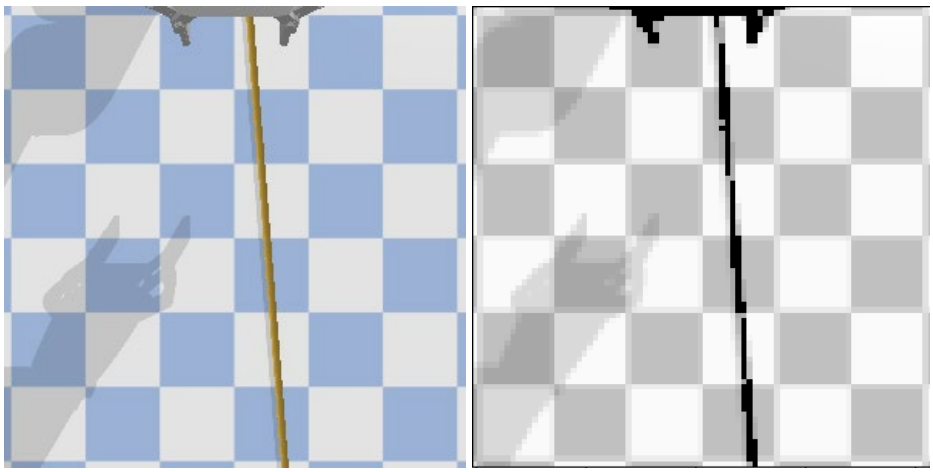

Figure 4.6 Left image is directly from the camera, then convert to gray scale and reduce resolution from 100x100 pixels to 64x64 pixels.

4.1.4 Action configuration

An action refers to a specific decision or choice made by an agent in response to a given state. In our experiment, actions are continuous, meaning that the action space is a continuous range of values. We represent actions as a box type, similar to the observation, consisting of four elements which are robot movement along the x, y, and z axis, as well as the gripper width. These values are between -1 and 1. For the gripper width, we apply a threshold to convert it into a discrete value in order to control the gripper. If the value is negative, the gripper will close, otherwise, it will remain open.

4.1.5 Reward function structure

To ensure that the agent understands the consequences of its actions, we need to define a proper reward function. The shaped reward function is used in order to evaluate each timestep. The reward will be calculated from three main factors which are object position, end effector position and gripper width. Typically, the objects used for training are relatively uniform in dimensions such as width, height, and depth. However, in this case, the position of the object cannot be obtained from the middle point of the object. In our experiment, soft object that we use is long, narrow and free-shaped soft object, making the agent possible for grasping the object from its head or tail without reaching the middle point. This makes it inadequate to rely solely on the position obtained from the simulation to calculate the precise position of the object.

Regarding the object height, the object position value that we get from the simulation is not sufficient because it only considers the middle point of the object. From Figure 4.7, even if the robot arm successfully lifts the object to the predetermined height, if the middle point of the object which marks a red arrow, remains on the plane, it would not receive the reward. To find its actual height, we iterate through every node of the object and identify the highest node, pointing by the green arrow, which would represent the height of the object instead.

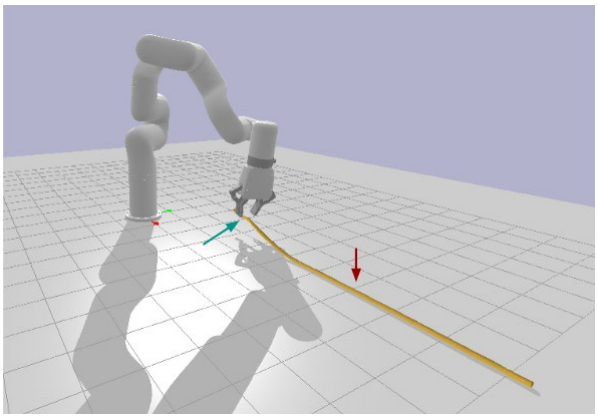

Figure 4.7 The object height will be determined by the green arrow instead of the red arrow which is the middle point of the object.

As mentioned in the first paragraph, we use object position, robot arm end effector position and gripper width to calculate the reward which can be represented as the equation below.

$$R_{total} = R_{distObjGoal} + R_{distObjGripper} + R_{gripper}$$

where:

- R_{total} is total reward for each timestep

- $R_{distObjGoal}$ is the reward that considers the distance between the object and goal is z-axis, based on how close the object is to the desired goal position in terms of vertical displacement.

- $R_{distObjGripper}$ is the reward that considers the distance between the object and gripper, indicating successful grasping or approaching the object. The position of the gripper is represented by the end effector position.

- $R_{gripper}$ is calculated from gripper width, representing whether the gripper is open or closed. It encourages the agent to learn appropriate grasping actions.

From the above equation, $R_{distObjGoal}$ is calculated based on the distance between the object and the goal solely along the z-axis. The higher the value, the greater the reward. The maximum reward for $R_{distObjGoal}$ is achieved when the gripper successfully grasps the object at a z-axis value of 1.

Next, $R_{distObjGripper}$ is calculated from distance between object and gripper in 3-axis (x, y, z) However, due to the fluctuation in the distance between the object and the gripper at the start of each episode, directly using the distance value may lead to inconsistencies in

the rewards. For instance, if the object happens to spawn near the gripper at the start of an episode, the agent would receive a higher reward compared to episodes where the object spawns far from the agent.

To solve this problem, we use the percentage of the distance between the object's initial position and the gripper's position along the x and y axes. The closer the gripper gets to the object, the higher the reward for the agent. This percentage acts as a coefficient value, with a maximum of 100 percent for each axis make it get the maximum reward if it can go to the middle point of the object. By using a percentage-based reward calculation, we can avoid the fluctuation in the initial placement of the object. Still, the agent can still grasp the object up even it is not at the middle of the object. So, we use a weighting factor to adjust the reward in the case where the agent can grasp the object from a position other than the object's middle point. This weighting factor ensures that the agent still receives a reward even if it grasps the object from a non-middle position. In case agent moves gripper away from the object which makes the percentage be negative

For the z-axis, the reward assignment depends on the agent's ability to lift the object while keeping the gripper within a designated range, typically around the object's center point. If the agent successfully lifts the object within this range, it receives a positive reward. However, if the agent is in the grasping position but fails to lift the object into the air, a small penalty is applied.

```
# Calculate distance in x and y axis as the percentage of the distance between starting point and gripper
perc_dist_x = 100 - distance_gripper_obj_x*100 /self.distance_obj_start_x
perc_dist_y = 100 - distance_gripper_obj_y*100/self.distance_obj_start_y

# Threshold the negative reward to prevent too much penalty
perc_dist_x = perc_dist_x if perc_dist_x > 0 else -10
perc_dist_y = perc_dist_y if perc_dist_y > 0 else -10

# Reward for z axis if the gripper is able to grasp the object in the defined range
# It will receive penalty if it is in grasping area but cannot bring the object up
rew_z = -.2 if perc_dist_x < 30 or distance_gripper_obj_y >.2 else (1 - distance_gripper_obj_z) *.5

# Sum above reward with weighting factor
reward_distance_gripper_obj = (perc_dist_x/2 + perc_dist_y) / 2 * .01 + rew_z

# Threshold the negative reward to prevent too much penalty
if reward_distance_gripper_obj < 0:
    reward_distance_gripper_obj = -0.1
```

Figure 4.8 $R_{distObjGripper}$ calculation

Lastly, the $R_{gripper}$ represents the reward associated with the gripper's state, specifically whether it opens or closes in the right position for grasping the object. Initially, a collision between the gripper and the object is detected to determine if the gripper is close enough to the object to attempt grasping. If the gripper is able to touch the object and it is not open widely, it will receive a certain amount of reward. This reward reflects the successful proximity of the gripper to the object and encourages the agent to approach the object for grasping. Next step, If the gripper is in a touching state and in the proper grasping position, meaning it is in the correct position to securely grasp the object, the reward will be doubled. Still, if the gripper is far from the object along the vertical axis (z-axis), a reward will be given if the gripper opens. However, if the gripper is already near the object along the z-axis, the reward will be zero, as the gripper does not need to be opened further.

```
if Gripper touch object and Gripper is close then
    Reward for gripper = (1.5 - Gripper width)
    if Gripper is in grasping area then
        Reward for gripper x 2
else
    if Gripper is higher than object and Gripper is open then
        Reward for gripper = -(Gripper width)
    else
        Reward for gripper = 0
```

Figure 4.9 $R_{gripper}$ calculation as pseudocode

Not only is there a reward provided at each timestep, but there is also a termination reward at the end of the episode. This termination reward informs the agent about the overall outcome of its series of actions. There are three types of termination rewards which are penalty for maximum timesteps, penalty when soft object crashed due to external force and positive reward for goal achievement, as shown below.

```
if reach 300 timesteps then
    if it is able to lift the object up for a bit then
        Reward = -5
    else
        Reward = -10
else
    if soft object crashed then
        Reward = -20
    if reach the goal in z-axis then
        Reward = +100
```

Figure 4.10 Termination reward calculation structure as pseudocode

4.2 Results

For the training process, we conducted training sessions with the objective of teaching the agent to grasp the soft cylinder, which represents the spaghetti. Each training round consisted of approximately 2 million timesteps or roughly 6,600 episodes. As for monitoring and assessing the progress of the training, the results were saved as a model after training for every 50,000 timesteps. We employed both the SAC (Soft Actor-Critic) and PPO (Proximal Policy Optimization) algorithms for training, using both direct observation and image observation approaches.

4.2.1 Training by Proximal Policy Optimization, using direct observation

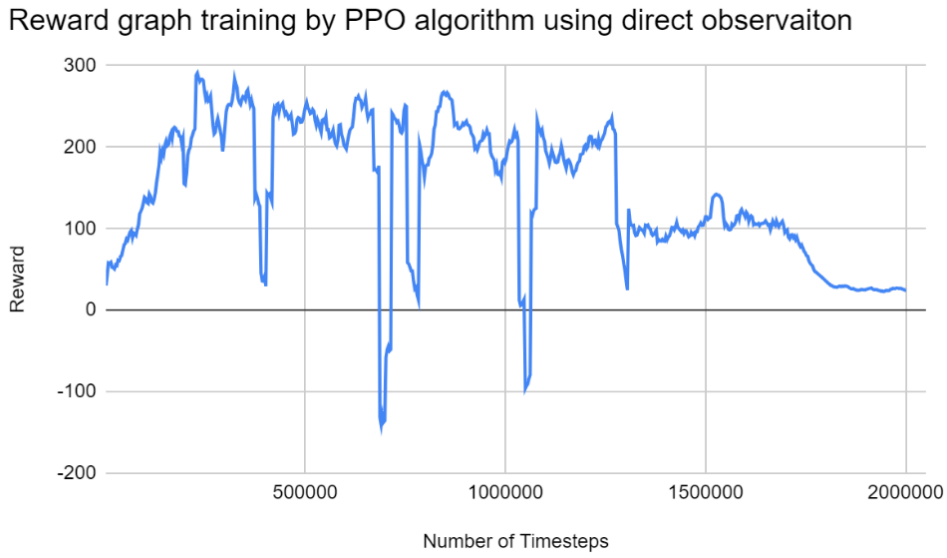

Figure 4.11 Agent's reward graph training by PPO algorithm, using direct observation

Based on Figure 4.11, the graph represents the results obtained after training the agent with Proximal Policy Optimization algorithms using direct observation, which includes the positions of the robot's end effector, object position, and gripper width. The reward demonstrates a significant increase from the initial stages until reaching approximately 350,000 timesteps or 1,160 episodes. Through testing and environmental observations, at that timesteps, the agent successfully reaches the midpoint of the object, as indicated by a reward exceeding 200. Even in cases where the agent does not precisely reach the middle position, it demonstrates the ability to grasp other parts of the object successfully, earning rewards as long as the gripper makes contact with the object.

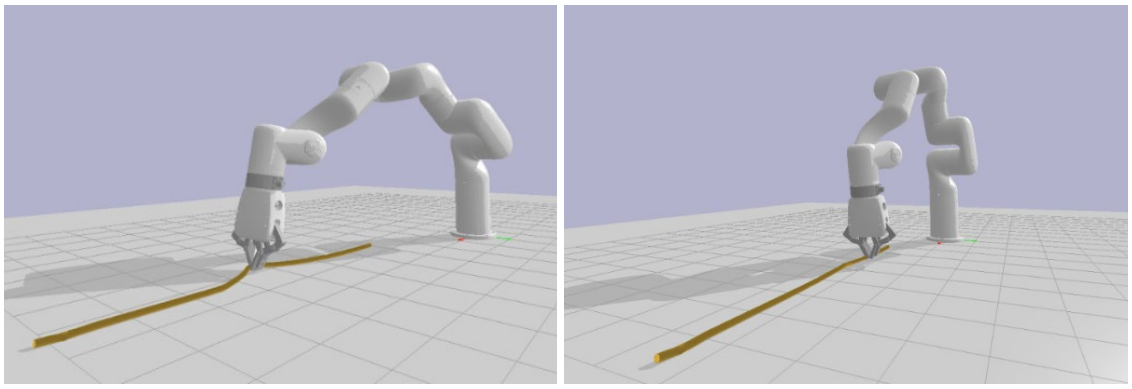

Figure 4.12 Agent's grasping behavior after training 350,000 timesteps

However, during attempts to lift the object around 400,000 timesteps, there are instances where the gripper adjusts too narrowly or moves too quickly when close to the object. As a result, the object experiences damage and crashing, leading to a substantial penalty, as depicted in the graph. This significant penalty causes the reward to sharply decrease before gradually rising again. This pattern emerges as the agent continues to attempt grasping actions despite the penalties incurred. Still, there is no clear upward trend observed in the reward after reaching that point. The agent continues to explore alternative strategies to lift the object, as indicated by the reward consistently staying above 200 until reaching 1.3 million timesteps or 4,300 episodes. However, beyond this point, the reward tends to decline. This decline can be regarding to the agent tend to stay in the same place to avoid incurring penalties, resulting in reduced exploration. As a result, the agent's ability to discover new and more rewarding actions becomes limited.

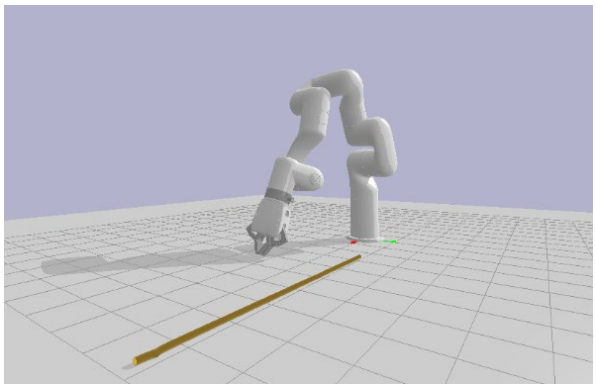

Figure 4.13 Agent's grasping behavior after training 1.3 million timesteps

4.2.2 Training by Soft Actor-Critic, using direct observation

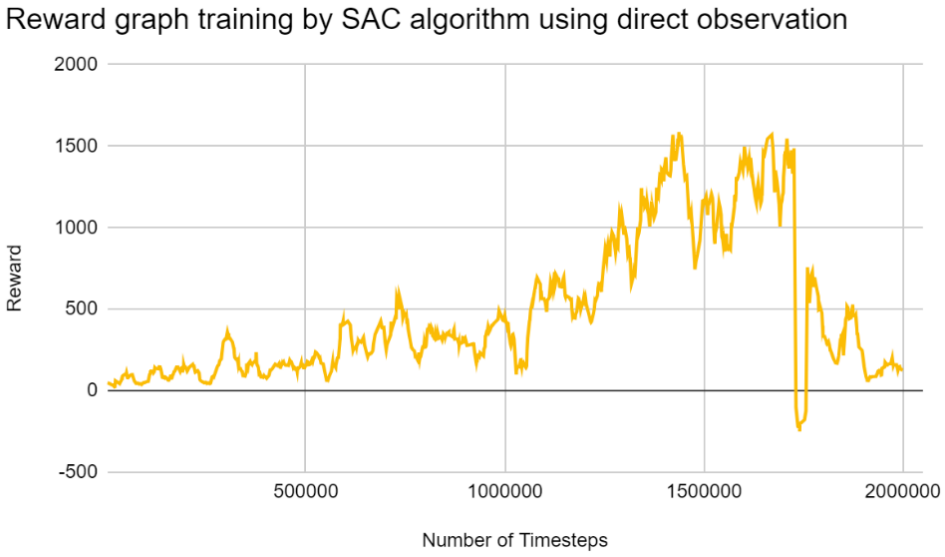

Figure 4.14 Agent's reward graph training by SAC algorithm, using direct observation

Using the Soft Actor-Critic algorithm with the same custom policy employed in PPO, the agent undergoes training and achieves notable progress. According to Figure 4.14, which illustrates the training progress using the SAC algorithm, after completing approximately 300,000 timesteps or 1,000 episodes, the agent demonstrates the capability to move itself to the grasping area successfully on multiple occasions.

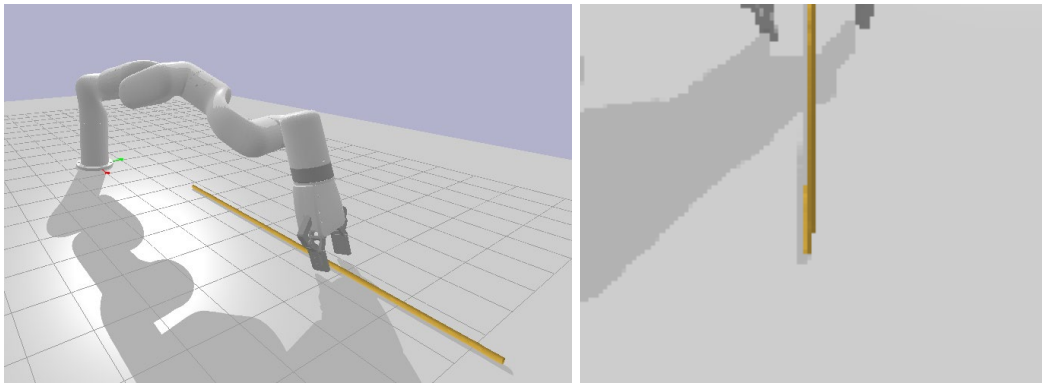

Figure 4.15 Agent's grasping behavior after training 300,000 timesteps

Around 450,000 timesteps later, specifically at approximately 750,000 timesteps or around 2,500 episodes, the gripper starts showing the ability to position itself in the grasping position and lift the object into the air. This action is reflected in the reward graph, which

exceeds 500 at this stage. However, the gripper's approach to lifting the object is different from our initial expectations.

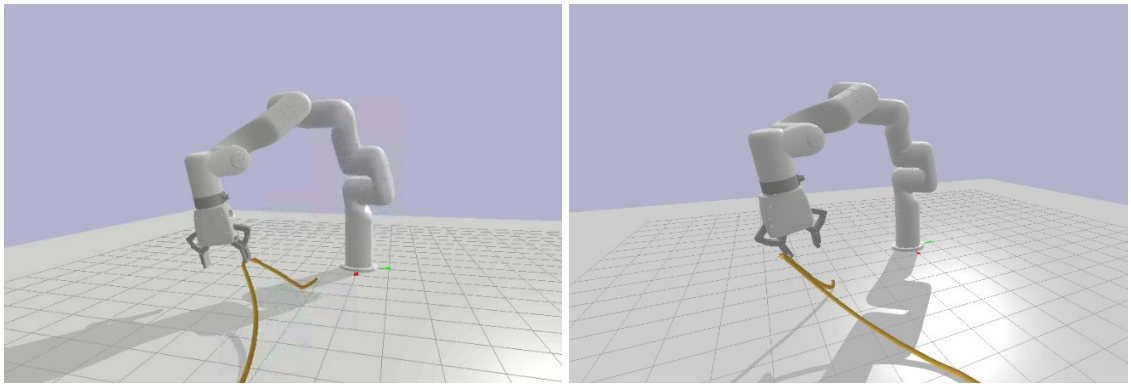

Figure 4.16 Agent's behavior when try to bring the object higher after training 1.4 million timesteps

The reward continuously shows a significant increase after reaching 1 million timesteps. However, upon observing the results obtained from the model saved at approximately 1.4 million timesteps, it became apparent that the agent had discovered a new method for elevating the object. Following a considerable training period, the gripper successfully reaches the midpoint position of the spaghetti. Instead of employing a direct lifting motion, the gripper strategically positions itself adjacent to the object and accelerates its movement to make itself stick to the soft object before lifting motion. The faster it moves, the longer it will remain attached to the gripper. With this action, the agent primarily adjusts its position while keeping the gripper width constant This is because, once the object is securely attached to the gripper, it will not fall even if the gripper is open. Still, there comes a point when it moves so fast that its force damages the object instead, leading to a sharp decrease in the reward and stop exploring after training around 1.7 million timesteps.

4.2.3 Training by Proximal Policy Optimization, using image observation

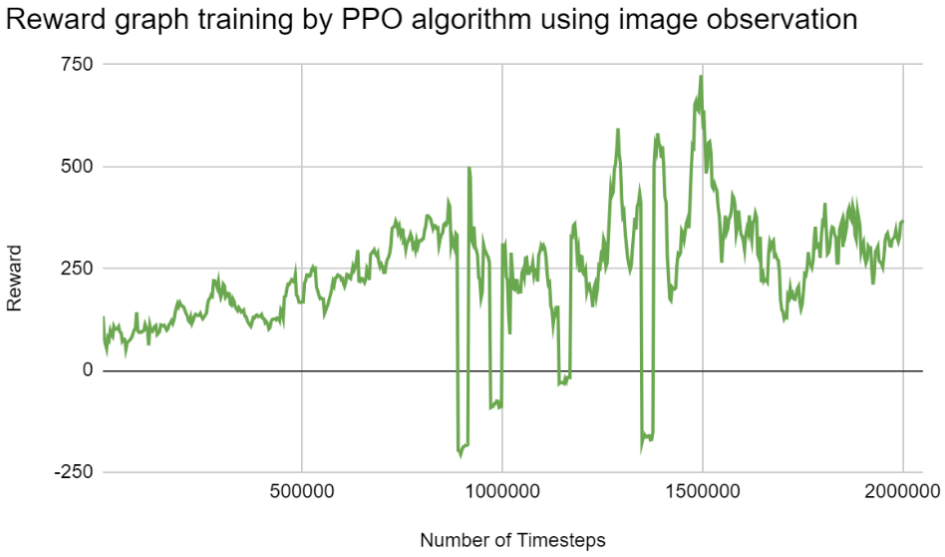

Figure 4.17 Agent's reward graph training by PPO algorithm, using image observation

In this training, we trained the model with Proximal Policy Optimization algorithm same as the first one, but instead of using direct observation, we use an image that got from the synthetic camera in the pybullet instead. According to Figure 4.17, the reward graph provides insights into the training progress of the model. Around 300,000 timesteps after starting training, the agent is able to move to the object's position. Then, at approximately 500,000 timesteps, the reward approaches a value of 250, indicating that the agent has successfully moved closer to the object which randomly spawned within the grasping area.

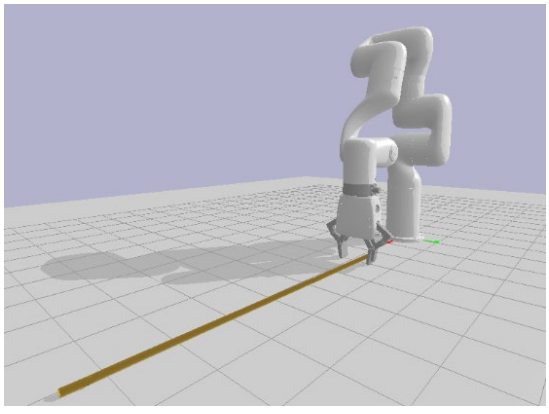

Figure 4.18 Agent's grasping behavior after training 300,000 timesteps

After that, the reward continues to gradually increase until around 900,000 timesteps. At this point, there is a sharp drop in the reward due to the agent's attempt to grasp the object, which results in unintended damage.

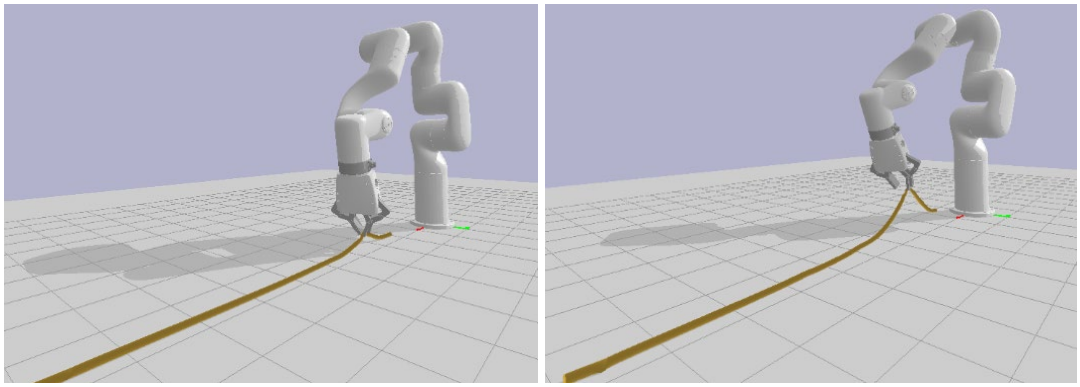

Figure 4.19 Agent behavior when try to bring the object higher, training around 950,000 timesteps

Only 50,000 timesteps later, approximately 950,000 timesteps, the gripper is finally able to reach the grasping position accurately and able to touch the object as in Figure 4.19. The agent then proceeds to lift the object using a grasping motion that involves closing the gripper to securely hold it. Nevertheless, during this process, there are occasions where the object sticks to the gripper unintentionally, especially when the gripper comes into contact with the object while it is on a plane surface. The moment that it moves itself up, there are instances where the gripper suddenly opens shortly after being closed even though the object would not fall even if the gripper stays open. This motion occurs occasionally, as there are instances where the gripper may or may not open after lifting the object up, even if it was closed initially. Despite the fact, keeping the gripper closed during the lift would result in a higher reward.

Moreover, there is a noticeable result when the agent reaches approximately 1.45 million timesteps or 4,800 episodes, with the reward at that point exceeding 600. Upon observing the environment, we discover an interesting agent’s behavior. Initially, the agent follows the grasping action described earlier, where it moves to the grasping area, closes the gripper, and attempts to lift the object. However, if the agent finds itself moving upward without an increase in reward due to the object not being lifted along with it, it adjusts its height by moving back down and tries to pick up the object again within the same episode which leads to the increase in reward.

This action is a result of the movement that the gripper occasionally opens after successfully lifting the object into the air, as mentioned previously, to avoid the grasping motion that resulted in significant penalties earlier in the training process. Still, attempting to touch the object for a second time can sometimes render the object unstable, leading to a subsequent crash, resulting in a decline in the reward after reaching 1.5 million timesteps.

4.2.4 Training by Soft Actor-Critic, using image observation

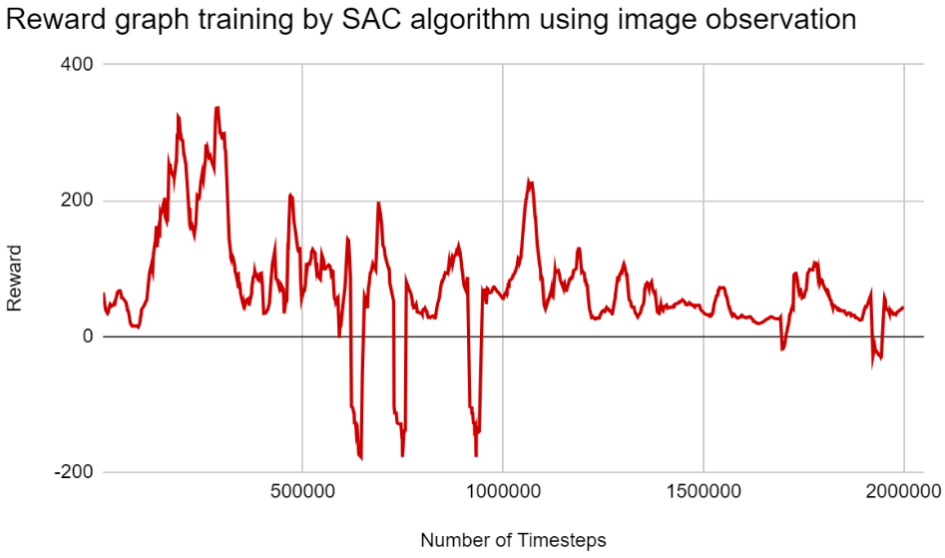

Figure 4.20 Agent's reward graph training by SAC algorithm, using image observation

Based on Figure 4.20, the graph illustrates the training results of the model using the Soft Actor-Critic algorithm with image observation. The reward curve initially shows a sharp upward trend at the beginning of the training process. After approximately 150,000 timesteps, the reward surpasses 200, indicating a notable improvement in the agent's performance. At this stage, the agent becomes capable of moving toward the object, although its accuracy in landing near the object is still pretty low.

Continuing the training for an additional 50,000 timesteps brings the total to 200,000 timesteps. At this point, significant progress is observed. The agent demonstrates the ability to approach the object, grasp it, and successfully lift it. However, despite completing the task, the agent's actions appear to be relatively random, lacking a consistent strategy.

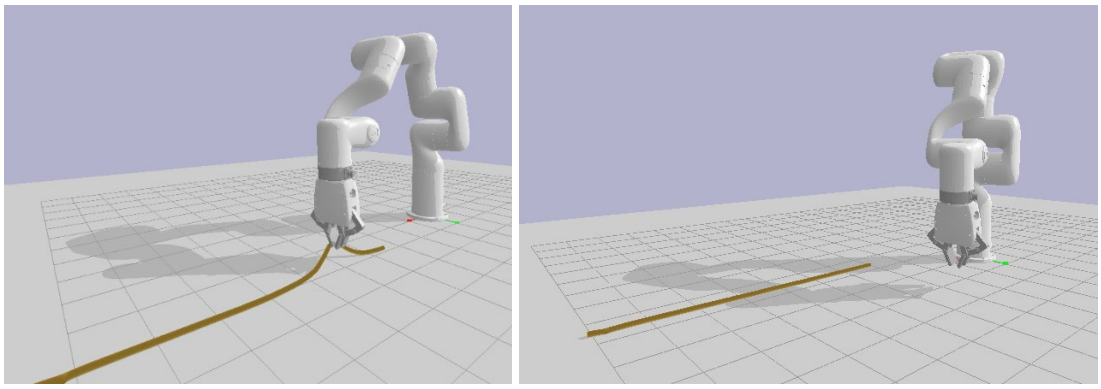

Figure 4.21 Agent’s grasping behavior after training around 200,000 timesteps

By observing the environment, we found that the agent achieves successful lifting of the object only 6 out of 20 times, equal to 30%. Still, the other 6 times, the agent moves randomly which is not going close to the object. These behaviors, including the random movements and inconsistent approach to the object, contribute to the relatively low reward observed in the graph when compared to the results obtained from other models. Although the agent is capable of successfully grasping and lifting the object, its performance varies, leading to inconsistent outcomes.

After that, the reward dropped once and rise again. During this phase, the agent continued to do similar actions as observed during testing with the model trained for 200,000 timesteps. The agent's behavior became more random, with instances where it successfully grasped and lifted the object, while other times it remained near the grasping position without completing the task. These results indicate that the agent attempted to explore different behaviors rather than adhering to a single approach. Despite its efforts to explore the environment further, the agent was unable to discover a more effective strategy to consistently increase its total reward. As a result, the reward graph displayed a downward trend towards the end of the training process.

5 Discussion

5.1 Model results comparison

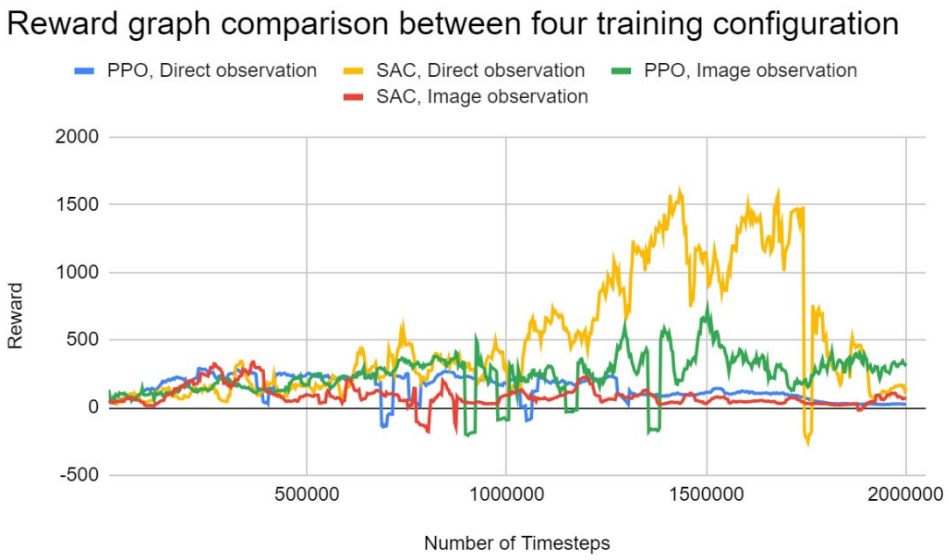

Figure 5.1 Agent's reward graph comparison between four training configurations

According to Figure 5.1, the graph displays the results of training the grasping environment using 4 different configurations. It is evident that the model trained with the Soft-Actor Critic algorithm, using direct observation, achieves the best performance. The reward surpasses 500 at around 750,000 timesteps and goes beyond 1,000 at approximately 1,300,000 timesteps. Testing the environment further confirms that this model is capable of lifting the object to the highest position, despite its approach differing from the other models due to its explorability of SAC algorithm.

The next model that shows good results is the one trained with the Proximal Policy Optimization algorithm, using image observations. The reward surpasses 500 at around 900,000 timesteps, and testing in the environment also indicates positive outcomes. The agent can lift the object into the air using the grasping motion initially and sometimes opens the gripper after a moment.

The model trained with the SAC algorithm using image observations also demonstrates great results, achieving success in a relatively short training period. The agent is able to start grasping tasks after only 150,000 timesteps, and within an additional 50,000 timesteps, it can

both touch and grasp the object simultaneously. However, it exhibits some random movements, indicating less consistency compared to the other models.

The last result, the model trained with PPO using direct observations, even it cannot reach the goal, which is lifting the object, but it shows great consistency. Almost every testing round, the agent can move the gripper to the grasping position perfectly, which can be calculated at approximately 90%.

Table 1 The speed at which each model allows the agent to accomplish the actions, measured in the number of timesteps

Agent abilities	Number of Timesteps			
	Direct observation		Image observation	
	PPO	SAC	PPO	SAC
Move to grasping position	200,000	300,000	300,000	150,000
Touch and grasp the object	350,000	600,000	500,000	200,000
Lift object from the plane	-	750,000	700,000	200,000

From the Table 1, it presents the speed of each training configuration, measured in terms of how quickly each task can be accomplished based on the number of timesteps. To evaluate the performance, we conducted tests on the saved models at intervals of 50,000 timesteps using the same environment as during training. The grasping task was divided into three subtasks:

1. Moving to the grasping position: This subtask focuses on the gripper's position. If the robot arm is able to move the gripper towards the object's direction and maintain proximity to it, we consider this subtask to be successfully completed.
2. Touching and grasping the object: Once the gripper is in close proximity to the object, the agent attempts to touch or grasp it. If the agent successfully touches or attempts to grasp the object, we consider this subtask to be accomplished.
3. Lifting the object up from the plane: The ultimate goal of the task is to lift the object up into the air. If the agent is able to hold the object above the plane for a significant distance, we consider this subtask to be successfully completed.

Based on the categorization by task, the model that allows the agent to move closely to the object fastest is the one trained with SAC using image observation. The agent is able to reach the object's proximity within approximately 150,000 timesteps. Subsequently, after 50,000 additional timesteps, the agent can successfully grasp and lift the object up into the air.

Compared to the other models, this configuration shows the shortest training time for achieving the grasping task. However, while this model shows faster initial progress, it also exhibits less consistency in its performance. As mentioned about the training results in the previous chapter, the performance of this model tends to decline after reaching a peak, indicating a lack of stability in maintaining consistent success rates.

Apart from the reward graph that can provide insights into the agent's training progress, it may not always directly correspond to the agent's performance in the testing environment. The reward function used during training might not capture all aspects of the desired behavior or may not fully align with the evaluation criteria used during testing. Therefore, it is important to evaluate the agent's performance in the actual testing environment to obtain a more accurate assessment of its capabilities.

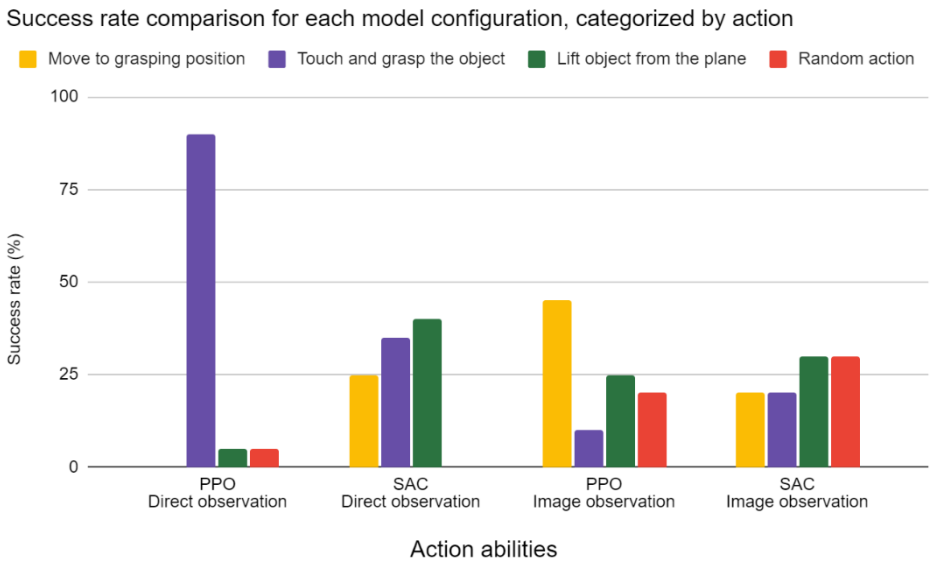

Figure 5.2 Success rate comparison for each model configuration, categorized by action

To evaluate the performance of each model, we conducted testing in the environment using the specific models that demonstrated the best performance at different timesteps. For each model, we ran the testing scenario 20 times to assess its capabilities. As shown in Figure 5.2, the diagram shows the consistency of each configuration categorized by the action. The model trained with the PPO algorithm using direct observations shows a high level of consistency in terms of reaching the grasping position. In almost every testing round, the agent successfully moves the gripper to the desired grasping position, achieving an accuracy rate of approximately 90%. On the other hand, the model trained with SAC using image observation may display

more random behavior compared to the others. For 20% of the testing rounds, the agent can move to the grasping position. In 30% of the rounds, it successfully lifts the object, while in the remaining 30%, it moves randomly without approaching the object's position.

This variability in performance highlights the agent's exploration capabilities, but it also indicates the need for further refinement to improve the consistency of its actions and increase the overall success rate in accomplishing the task.

5.2 Reward graph pattern

According to the graphs in the above section, we notice that there is a pattern which cannot be found in training with solid objects. When the reward is gradually increased, at some point, it will drop sharply as it gets a large penalty. This pattern comes from the behavior that the agent can reach the object and try to grasp it, suddenly, it makes the object crash when it grips too hard which leads to lots of negative reward. The agent repeats this behavior for a period until it recognizes which actions are problematic and attempts alternative solutions.

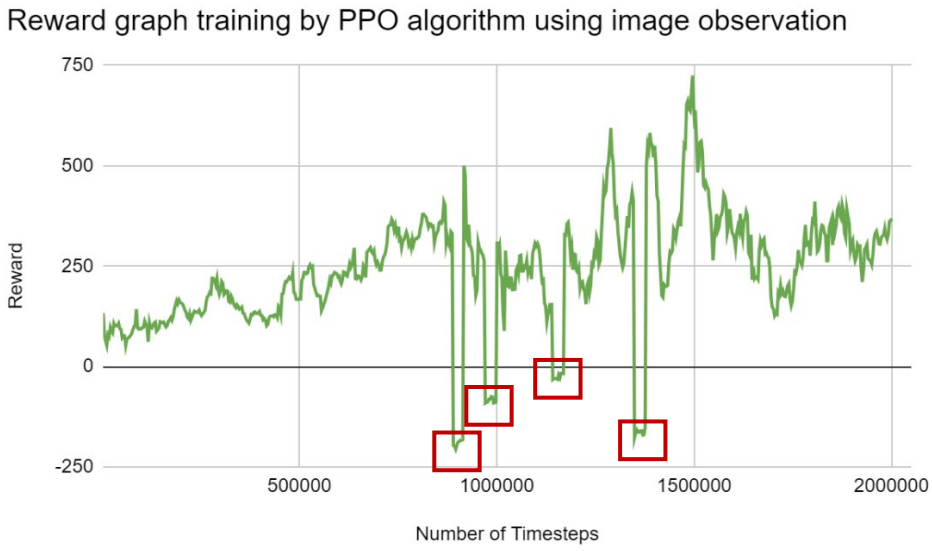

Figure 5.3 Example of reward graph that show the pattern while training with soft object

Due to using the soft object, it tends to be difficult to make the object stable enough for grasping action, especially with asymmetric objects. To prove this assumption, we have tried training by using Deep Reinforcement Learning, the same algorithm as the current soft object but different shape, and more symmetry compared to the current object we created. The result is, using a more symmetric soft object, the reward graph does not show the pattern which

happens when training with the current object. The symmetric object tends to be more stable because it contains more node inside the object which make it durable against external force.

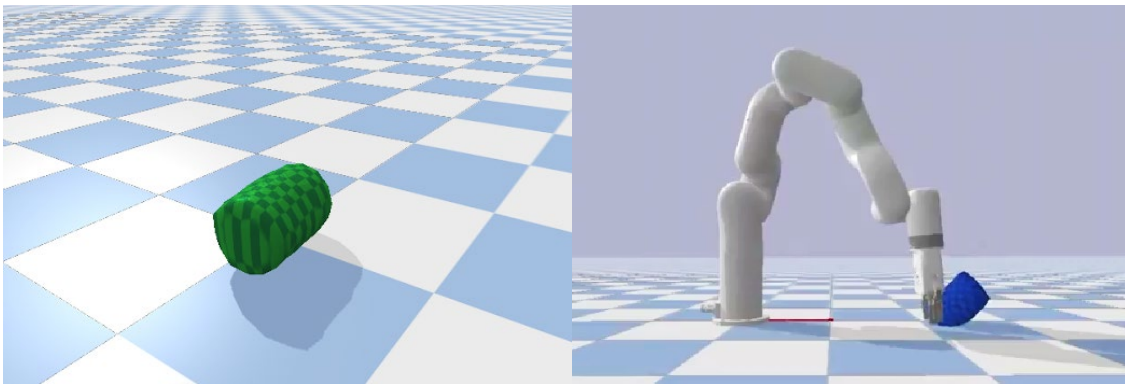

Figure 5.4 Example of more symmetric object, tend to be more resistant to the external force

5.3 Training speed

For this research, the training for all of the models was conducted on a Dell G15 5515 with an AMD Ryzen 5800h processor. When using direct observation as the input, the training process achieved an average speed of 18 frames per second (fps) or 70 – 100 ms per timestep, with the model taking approximately one and a half days to complete the training. On the other hand, when utilizing image observation, the training speed decreased by approximately 33%, resulting in an average speed of around 12 fps.

Furthermore, when we conducted tests using this object manipulation task environment, same agent but with solid objects instead of soft objects, we observed a notable increase in speed. The training process using solid objects was approximately 35% faster compared to the training with soft objects by using direct observation.

5.4 Challenges

5.4.1 Object parameters setting

Finding suitable parameters for the object in the training process can be a time-consuming task. As described, the initial approach of using Unity as the simulation platform involved creating the soft object within Unity itself. However, it was discovered that the created soft object did not work as intended, so we decided to use the existing assets instead. Nonetheless, these assets still required various adjustments to ensure that the soft object has

characteristics similar to real spaghetti, such as being bendable, having low friction, and not consuming excessive computational resources during training, as we planned to use plenty of them in training. Once finished the parameter settings, it was found that although the soft object had a good collision with other objects, it did not lend itself well to firm grasping. Consequently, a change in the simulation platform became necessary, leading to the adoption of pybullet.

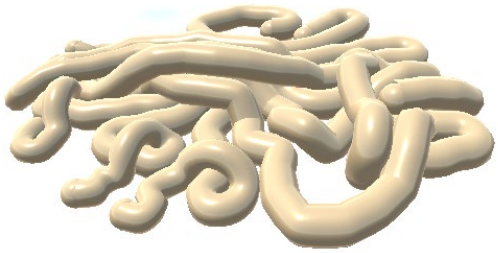

Figure 5.5 Object that was set various parameters in Unity

Transitioning to pybullet also introduced a new set of challenges, as there were fewer examples available for creating the training environment with soft objects. It was necessary to spend time understanding and configuring the environment settings specific to soft objects. Additionally, due to the unavailability of a suitable long cylinder model, a custom object needed to be created. This involved obtaining or generating the object model in the rarely used VTK format, which required specialized software. Furthermore, finding the appropriate parameters for the object in the pybullet environment demanded further experimentation. Resulting in spending more time finding suitable parameters, which may include parameters for the simulation and parameters for creating the model. As a result, we need to adjust and iterate repeatedly to fine-tune these parameters.

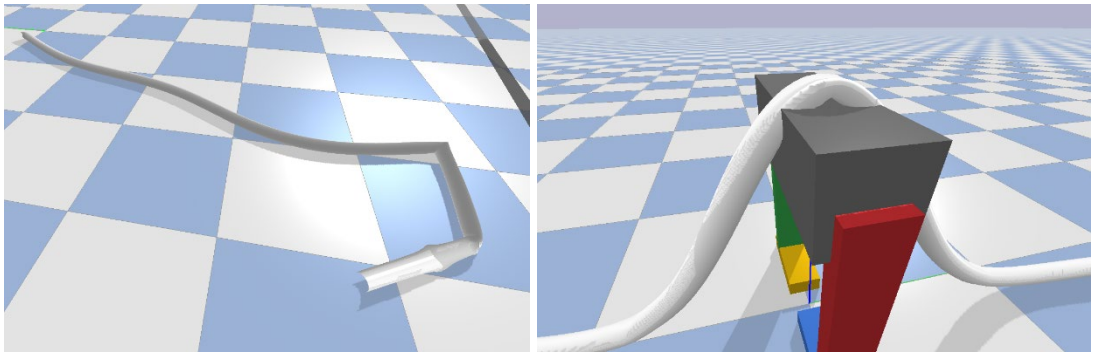

Figure 5.6 The cylinder that was imported to the pybullet, before setting the parameters

This process often requires extensive experimentation and fine-tuning to identify the optimal parameters that enable the agent to perform effectively in the object manipulation task. Each adjustment requires training the model and evaluating its performance to determine if the changes have resulted in improved outcomes. Due to the complexity of the task and the multitude of possible parameter combinations, it is common for this process to require a significant amount of time and effort. However, by exploring and adjusting the parameters, it becomes possible to find the configuration that gives the good performance for the given object manipulation task.

5.4.2 Soft object stability

Even though we can adjust various parameters for the soft cylinder object to mimic the properties of real spaghetti, during the training process, the object behaves differently. When the object is pushed by external forces, there are times when the pybullet is unable to accurately calculate the resulting shape and the simulation crashes. This instability of the object becomes a significant obstacle for the agent to explore the environment and achieve the goal of grasping the object. Additionally, even when the robot arm is able to successfully lift the object, there are occasions where the object crashes, resulting in penalties for the agent instead. This issue highlights the challenge of object stability and the impact it has on the training process. The unpredictable behavior of the object impedes the agent's ability to learn and execute effective grasping strategies. Finding ways to improve object stability within the simulation environment is important for enhancing the agent's performance and achieving more consistent and successful grasping outcomes.

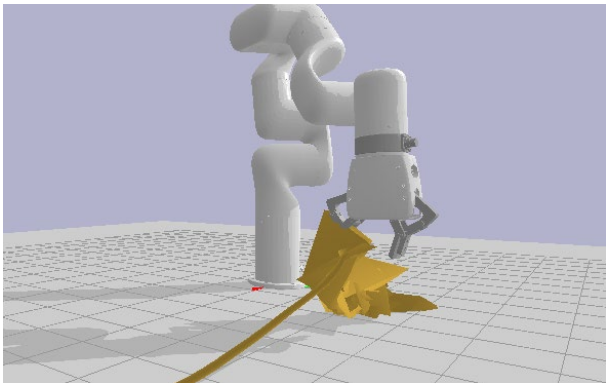

Figure 5.7 Soft object was damaged while the agent is grasping it

5.4.3 Object interaction detection

In the grasping object task, how the gripper holds the object is an important factor for the agent's learning success. When dealing with solid objects, the object's position may vary, but its shape and properties remain consistent throughout the task. On the other hand, when working with soft objects, particularly with the model that we currently use that is long and narrow, ensuring a secure grasp becomes more challenging. When the grippers contact the object at the designated grasping position, it is not always able to determine whether they are just touching the object or genuinely holding onto it. The deformable nature of soft objects makes the task more complicated as they can easily change shape and slip through the grippers' grasp.

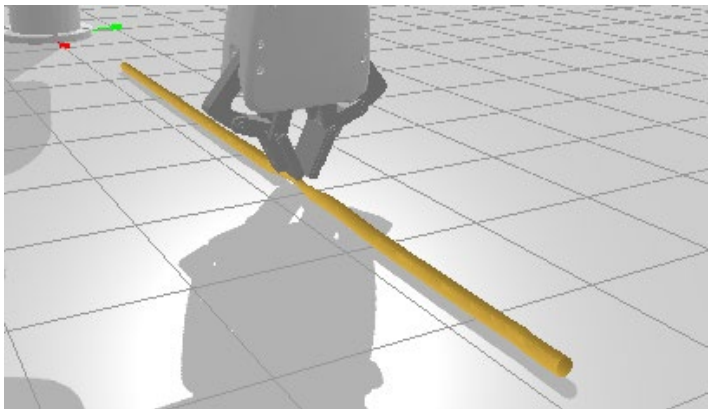

Figure 5.8 Agent’s behavior sometimes hard to be determined if it is grasping or just touching

Furthermore, differentiating between successfully grasping the object and damaging it becomes less clear-cut when working with soft materials. Even a slight change in gripper width can lead to the object being crushed. The agent must learn to apply just the right amount of force and pressure to hold the object securely without causing harm or deformation.

6 Conclusion

6.1 Summary

In conclusion, we have developed a specific environment for the object manipulation task, focusing on a soft, narrow, and long-shaped object resembling spaghetti. This environment was trained using Deep Reinforcement Learning techniques to enable the agent to successfully grasp and lift the soft cylinder object. The results obtained from training the models using different configurations show that the agent is capable of lifting the soft object. The models trained using PPO (Proximal Policy Optimization) and SAC (Soft Actor-Critic) algorithms show different characteristics. PPO tends to display more consistency in its performance, but explores fewer alternative approaches compared to SAC. On the other hand, SAC demonstrates greater exploration capabilities, allowing the agent to try different strategies to maximize its reward. However, SAC's actions tend to be more random compared to PPO.

Evaluate by the speed of training the configuration that leads the agent to the goal in the shortest time is the model trained with SAC using direct observation. The agent can reach and lift the object up after approximately 200,000 timesteps or 660 episodes, which takes around 6 hours for training. However, this model shows less consistency as it sometimes leads the agent to perform random actions. On the other hand, the model trained with PPO using direct observation tends to show the most consistent results. Although it cannot lift the object, it successfully moves the gripper to the grasping position with a success rate of 90%.

From the reward graph, the two best-performing models are the one trained with PPO using image observation and the other trained with SAC using direct observation. After testing these trained models in the environment, we found that both models are capable of lifting the object high, but they use different approaches to achieve this. The PPO model uses a grasp motion to grasp and lift the object up, while the SAC model tries to avoid direct grasping. Instead, it accelerates itself to make the object stick to it and then lifts itself up, resulting in a higher reward. These results demonstrate the strengths of the SAC algorithm since it is known for its ability to balance exploration and exploitation, allowing it to discover various strategies and find optimal policies in complex environments.

Still, there is an important obstacle that makes the training process become more challenging. The object stability becomes the important factor to make the agent able to complete the task. The soft object's tendency to deform and lack rigidity makes it difficult for the agent to maintain a firm grip and manipulate it effectively. Despite adjusting various parameters, the soft object was still susceptible to external forces exerted by the gripper. This limitation prevents the agent's ability to explore the environment further, as some exploration attempts resulted in penalties due to damage to the soft object.

6.2 Future work

6.2.1 Object parameter setting

Currently, the main challenge lies in the stability of the object. As mentioned before in Chapter 5, the object needs to be more stable for the agent to effectively touch and lift it up. The stability issue includes further adjustments to the object's parameters, such as its material properties, friction coefficients, or structural characteristics.

Moreover, exploring advanced physics simulations or introducing additional constraints during object interactions could help improve stability. Focusing on enhancing the object's stability, we can create a more realistic and reliable training environment that enables the agent to develop better grasping strategies and achieve higher success rates.

6.2.2 Computer vision

Using images as observations and employing a Convolutional Neural Network (CNN) policy from stable baselines for feature extraction, the agent can effectively extract relevant features from the raw image inputs, enabling it to gain a deeper understanding of the object. Moreover, converting the images to grayscale can simplify the input data while retaining essential information, aids in better feature extraction and decreases computation time. This approach helps agents to process images faster, allowing for more rapid decision-making and action execution during training and testing.

So, we believe that enhancing the agent's vision further in the object manipulation task, such as using a depth camera or performing image recognition, would lead to better performance, enable the agent to accurately identify and locate the soft object in the environment.

6.2.3 Adapt custom policy

At present, we use the PPO and Soft Actor-Critic algorithms, originally from stable baselines, together with a few layers of custom policy. When training the agent, various parameters such as learning rate and batch size are adjusted to gain better results. However, further adjustments are required to increase the success rate in the future.

For the custom policy, when comparing the results to those obtained without it, we find significant improvements for both types of observations. This underscores the importance of custom policy design. Tailoring a policy specifically for the object manipulation task, taking into account its unique characteristics and challenges, can enhance the agent's decision-making and grasping strategies. Custom policies have the capacity to incorporate domain knowledge and heuristics, guiding the agent's actions and improving overall performance. However, it's essential to consider that adding more layers to the neural network will increase computational time. This aspect should be taken into account when adjusting the custom policy.

6.2.4 More simulation choices

While the current simulation environment using pybullet is able to complete this task, it is still worth investigating alternative simulation platforms that offer more advanced physics simulations and better handling of soft objects since pybullet’s accuracy when calculating the physics simulation is not good enough. It may be possible to find a platform that provides better stability and more realistic interactions with the soft object, which leads to more effective training and improved agent performance.

7 References

[1] KBVresearch, Jan 2023. [Online]. Available: https://www.kbvresearch.com/food-robotics-market/.

[2] P. Gamolped, S. Chumkamon, C. Piyavichyanon and E. Hayashi, "Online Deep Reinforcement Learning on Assigned Weight Spaghetti Grasping in One Time using Soft Actor-Critic," *International Conference on Artificial Life and Robotics,* pp. 554-558, 2022.

[3] Unity, "Unity ML-Agents Toolkit," 2018. [Online]. Available: https://unity-technologies.github.io/ml-agents/.

[4] V. Method, "Obi Rope," 11 Jun 2016. [Online]. Available: https://assetstore.unity.com/packages/tools/physics/obi-rope-55579.

[5] Renatyv, "Generating .vtk mesh for Pybullet softbody simulation," 30 Sep 2022. [Online]. Available: https://medium.com/@renatyv/generating-vtk-mesh-from-for-pybullet-softbody-simulation-ccc5f2f39139.

[6] CK-12, "Polyhedrons," 27 Jul 2022. [Online]. Available: https://flexbooks.ck12.org/cbook/ck-12-basic-geometry-concepts/section/11.1/primary/lesson/polyhedrons-bsc-geom/.

[7] Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin and D. Panozzo, "TetWild," 2018. [Online]. Available: https://github.com/Yixin-Hu/TetWild.

[8] "UFACTORY xArm 7 DoF Robotic Arm," [Online]. Available: https://jp.robotshop.com/en/products/xarm-7-dof-robotic-arm.

[9] G. Brockman, V. Cheung, L. Pettersson, JonasSchneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540v1,* 2016.

[10] C. Pan, "gym-xarm," 26 Jul 2021. [Online]. Available: https://github.com/jc-bao/gym-xarm.

[11] OpenAI, "Kinds of RL Algorithms," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[12] OpenAI, "Soft Actor-Critic," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html.

[13] Szepesvári and Csaba, "Algorithms for Reinforcement Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning,* vol. 4, 2010.

[14] M. Breyer, F. Furrer, T. Novkovic, R. Siegwart and J. Nieto, "Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning," *arXiv:1803.04996v2,* 2019.

[15] Abdeetedal and Mahyar, "OpenAI Gym Environments with PyBullet," 17 Apr 2020. [Online]. Available: https://www.etedal.net/2020/04/pybullet-panda_2.html.