SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Deep Reinforcement Learning for Robotic Grasping in an Unstructured Environment

Matthias Pouleau

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Deep Reinforcement Learning for Robotic Grasping in an Unstructured Environment

Author:	Matthias Pouleau
Supervisor:	Prof. Dr.-Ing. habil. Alois Christian Knoll
Advisor:	Xiangtong Yao, Lin Hong
Submission Date:	18.01.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 18.01.2025 Matthias Pouleau

Acknowledgments

I would like to express my gratitude to my supervisors, Xiangtong Yao and Lin Hong, for their support and guidance throughout the course of this thesis. Their feedback and expertise have been essential for completing this work.
I am also sincerely grateful to Prof. Dr.-Ing. habil. Alois Christian Knoll and the entire team of the chair of "Robotics, Artificial Intelligence, and Real-time Systems" for giving me the opportunity to conduct my master's thesis at this chair.
Finally, I would like to thank my friends and family for their support throughout my studies. Their assistance and encouragement have been invaluable during this journey, and I could not have done it without them.

Abstract

This thesis concentrates on the task of vision-based robotic grasping of an unknown or partially known object in an unstructured environment. For this purpose, a robotic agent learns its optimal end-to-end policy, which consists of high-level continuous Cartesian commands, by being trained with an off-policy reinforcement learning algorithm inside a simulated lunar environment. This work focuses on improving grasping performance compared to the baseline implementation by Orsula et al. [43]. To this end, a new curriculum function was developed, with changes to the composite and the persistent reward function. Additionally, a new method for feature extraction was introduced, where the point-cloud data is directly processed by the encoder instead of first being converted into an octree. Two different feature extractor approaches were implemented using this method, one based on the PointNet network [45] and the other based on the encoder network of the 3D Diffusion Policy paper [55]. While an improved performance could be observed from the new curriculum function, the feature extractor approaches directly processing the point-cloud data could not compete with the octree-based baseline approach.

Contents

1 Introduction

Grasping objects is a key manipulation task in robotics, with applications across many fields, such as industrial manufacturing, automated logistics, domestic robotics, agriculture, and healthcare. While the successful grasp of a fully known object has been mastered for several years, difficulties still arise when handling unknown or partially known objects in an unstructured environment. With the emergence of current Reinforcement Learning (RL) methods, one could observe an increasing capacity of robotic agents to perform more complex tasks with only partial environment knowledge, such as the grasping of unknown objects. In contrast to analytical methods, RL methods are able to learn a policy that can handle novel, unseen scenarios [23]. The first key challenge is to extract valuable features from sensory input so that the agent can adequately learn a well-performing policy. To accomplish this, one could imagine utilizing known powerful 3D machine learning models as feature extractors such as VoxNet [36], O-CNN [52], or PointNet [45] to leverage their efficient geometric feature embeddings. Another challenge in robotic grasping that frequently arises is the sparse reward given by the environment. When training the agent, it usually receives only a success or a failure at the end of an episode, making it harder to learn from the reward function. Implementing a well-designed curriculum with an advanced composite reward function can help to alleviate this issue.
The task presented in this master's thesis is based on the implementation of the paper "Learning to Grasp on the Moon from 3D Octree Observations with Deep Reinforcement Learning" by Andrej Orsula et al. [43]. It encompasses a vision-based robotic grasping task inside an unknown, unstructured environment. The problem is modeled as an episodic Markov Decision Process (MDP), where the observations are viewed as RGB-D images, and the actions are represented as high-level Cartesian movements of the end-effector. The agent learns an end-to-end policy by getting trained inside a simulated lunar environment via an off-policy RL algorithm. In the implementation of the base paper, the RGB-D images get converted into an octree and then processed by a compatible feature extractor so that the algorithm can handle the observations.
The works presented in this thesis aim to implement modifications to the system pipeline that improve the overall performance compared to the baseline implementation. An agent's performance is primarily evaluated using two key metrics: the overall success rate, which gives out the proportion of episodes in which the agent successfully

grasps an object and lifts it above a certain predefined height threshold, and the mean episode reward obtained during evaluation. The first area of improvement focuses on the curriculum function, which provides the agent feedback at each time step, based on the stages reached. The other changes implemented concern the approach to handle the RGB-D observations. Instead of converting the input into an octree, the data is transformed into a colored point cloud, which is subsequently processed by a suitable feature extractor. The performance of various point-cloud-based feature extractors will be evaluated, including ones based on the PointNet network as well as the 3D Diffusion Policy (DP3) network.

Performance improvements can be observed from several changes in the curriculum function, including implementing an incremental reward function for the "Lift" stage of the curriculum and introducing a persistent time penalty, which grows over the training time. When looking at the implemented feature extractors directly processing the point-cloud data, both the PointNet-based and DP3-based feature extractors were unfortunately not able to match the performance of the octree-based feature extractor.

This master's thesis is organized as follows. First, the theoretical background is laid out, with a brief overview of 3D data representation and reinforcement learning (RL) basics. The third chapter will then present the related works in the fields of robotics and deep learning with 3D data. Chapter 4 describes the problem formulation, as well as the architecture of the implemented system. In chapter 5, the different experiments are presented, and the obtained results are discussed. Finally, chapter 6 gives us a conclusion about the findings of this thesis, followed by an outlook that lays out the encountered limitations and possible future works.

2 Theoretical Background

2.1 Basics 3D Data Representation

2.1.1 Representation Approaches for 3D Data

In most robotic tasks, the agent needs some understanding of its three-dimensional environment and needs to represent it in a certain way. Several euclidean methods exist to represent 3D space, like depth maps or volumetric approaches, as well as non-euclidean methods, such as point clouds or meshes.

Depth Maps and RGB-D Images

Depth maps or RGB-D images typically consist of a single channel for depth data or four channels when combined with RGB information. These representations are considered 2.5D because they capture depth data through projection onto a 2D plane. Depth maps and RGB-D images offer easy neighboring queries due to their regular grid structure, making them computationally and memory efficient compared to volumetric representations. Additionally, they provide a more compact and effective surface representation, yet they suffer from information loss due to the projection process, making them more limited than fully 3D representation methods. Additionally, generalizing over different spatial orientations can be more challenging due to their inherent 2D nature.

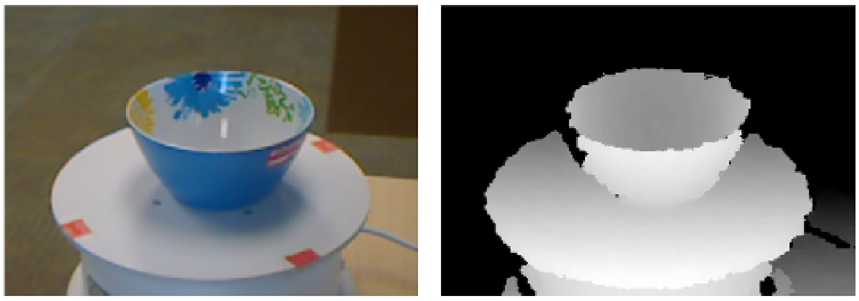

Figure 2.1: Example of an RGB-D image with color channels on the left side and the depth channel on the right side [27]

Meshed Representation

A polygon mesh represents a surface using a finite set $M = (V, E, F)$ of simple polygons, where V is the set of vertices, E is the set of edges, and F is the set of faces (often triangles). A vertex is defined by its 3D position in (x, y, z), an edge connects two vertices, and a face is a closed set of edges. The intersection of two polygons can be either empty, a vertex, or an edge, and every edge is part of at least one polygon. Meshes provide a piece-wise linear approximation of a surface with $O(h^2)$ error, allowing for a reasonably accurate representation of curved surfaces and smoother approximations with finer meshes [12]. This, as well as its ability to handle arbitrary topologies and its capacity to allow efficient rendering, make the meshed representation popular in 3D modeling and simulations. However, the connectivity between vertices, edges, and faces adds complexity, making polygon meshes more challenging to manage than point clouds or voxel grids [45].

Point Cloud Representation

In the point cloud representation method, the data is treated as an unordered set of N points in the three-dimensional space, each typically defined by (x, y, z) coordinates. These points can also carry additional attributes such as color or surface normals. Since point clouds are unordered, they are invariant to permutations, making them a flexible representation. They closely resemble the raw output of sensors like lidars or depth cameras and provide a more efficient surface representation than dense voxel grids. Unlike meshes, point clouds do not have the same combinatorial irregularities and complexities, making them easier to learn from in machine-learning tasks. However, lacking inherent spatial structure makes tasks like querying neighboring points less efficient [45].

Volumetric Voxel Representation

The voxel-based approach is a volumetric representation method where the data is discretized into a regular grid structure composed of 3D voxels analogous to pixels in two dimensions. The discretized structure is often represented as an occupancy grid, where each voxel contains a probabilistic estimation of whether the contained space is free, occupied, or unknown but can also comprise information about colors or normals. The voxel representation method offers several advantages, such as efficiently estimating free and occupied space and the ability to represent any 3D object with arbitrary structure or connectivity. It uses a simple and efficient data structure, making neighbor queries trivial and enabling easy manipulation and operations. However, a significant drawback is the cubic growth in memory requirements with increasing resolution, leading to memory inefficiency when representing detailed surface geometry [37].

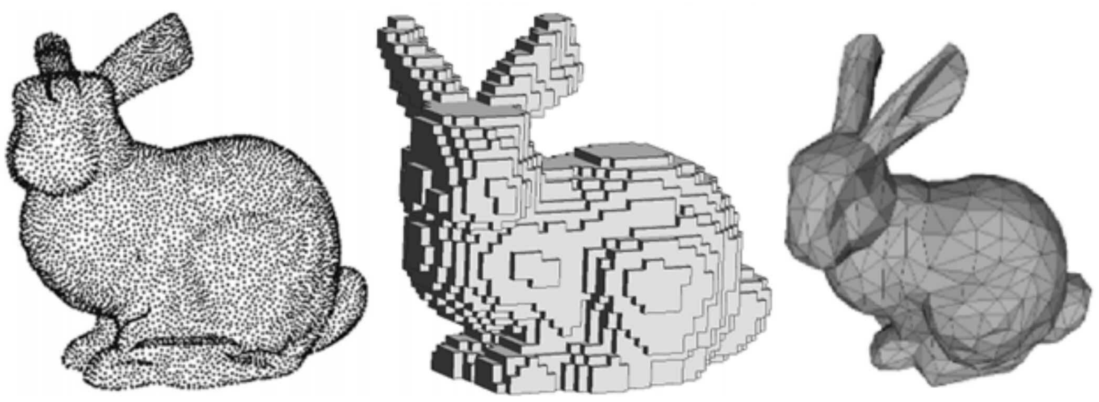

Figure 2.2: Visualization of the point-cloud-based, voxel-based, and mesh-based 3D representation methods using the Stanford bunny model [14]

Octree Representation

Octrees are a hierarchical volumetric 3D Data representation method initially conceptualized by Donald Meagher in 1980 [38], where the space is divided into a tree-like structure, and the cells are usually categorized as occupied, empty, or unknown. The octree decomposition process begins with a root node representing the entire workspace or volume to decompose. At each step, the current node's volume is subdivided into eight smaller subvolumes, each corresponding to a child node. These subvolumes are then evaluated and classified into one of three categories: occupied (the subvolume contains objects or obstacles), empty (the subvolume is clear of objects), or unknown (the subvolume is unclassified or mixed, and may require further refinement). If a subvolume is classified as either occupied or empty, it becomes a leaf node, indicating that no further subdivision is necessary. However, the process continues for subvolumes marked as unknown, by further subdividing them into smaller subvolumes. This recursive process usually continues until a certain predetermined partitioning depth (e.g., 4) or until all subvolumes are classified as occupied or empty, resulting in a complete space decomposition. The octree representation can provide the benefits of volumetric approaches without the same memory and computation inefficiency: due to their adaptive resolution, octrees allocate finer subdivisions only in regions with more detail, resulting in a higher percentage of cells being surface cells compared to the uniform grid of voxels. This efficiency allows octrees to better capture surface features while minimizing unnecessary subdivisions in empty or uniform regions, thus reducing memory usage. Additionally, octrees enable efficient spatial searching and collision detection by quickly narrowing the search space through their hierarchical structure. On the other hand, performing operations on octree structures can be pretty challenging, as their hierarchical structure leads to an increased complexity.

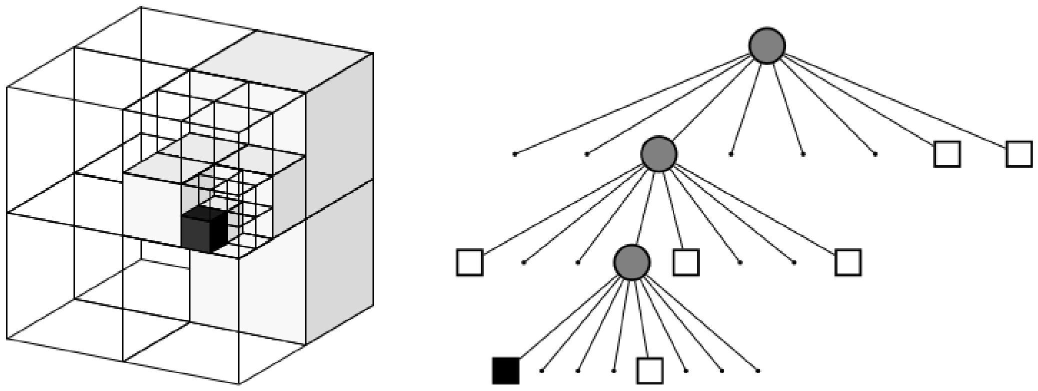
Figure 2.3: Visual representation of a 3D octree mapping from the works of Hornung, et al. [15]

2.1.2 Conversion Between Representation Approaches

Meshgrid to Point Cloud

Given a triangle mesh containing M triangles, the task is to generate a point cloud with N points on the surface of the mesh. Two commonly used approaches for this task are Uniform Sampling and Farthest Point Sampling (FPS).
Uniform Sampling Approach: generate random points on the surface of a triangle mesh with a uniform probability distribution proportional to each triangle's area. The process begins by calculating the area of each triangle, and this value is then normalized by dividing it by the total area to establish a triangle sampling distribution. For each point P_i, the following steps are executed: first, a triangle is uniformly sampled according to the established triangle sampling distribution. Next, two random values, $r1$ and $r2$, are uniformly sampled from the interval $[0, 1]$. Finally, the position of the point P_i is determined using the vertices V_1, V_2, and V_3 of the selected triangle with the following formula:

$$P_i = (1 - \sqrt{r_1}) \cdot V_1 + \sqrt{r_1} \cdot (1 - r_2) \cdot V_2 + \sqrt{r_1} \cdot r_2 \cdot V_3$$

This sampling formula ensures that the point is uniformly distributed across the triangle's surface.
Farthest Point Sampling Approach: technique used to select a new set of points A from a

previously sampled set of points S, where the number of points K in S is significantly larger than the number of points N to be sampled for A. The process begins by randomly selecting one point from S, removing it from S, and adding it to A. This initial point serves as a seed for the subsequent selections. For $N-1$ times, the following steps are performed: for each remaining point in S, the distances to all points currently in A are calculated. The distance to the closest point in A is identified for each point in S, and the point in S with the largest minimum distance to any point in A is selected. This selected point is then removed from S and added to A [9]. The FPS method results in points more evenly distributed across the surface area than uniform sampling. However, it has a higher computational complexity of $O(KN)$ than the $O(N)$ complexity of uniform sampling [32].

Depth Image to Point Cloud

Given a depth image D of pixel-width W and pixel-height H from a camera with horizontal field of view θ_x and vertical field of view θ_y. The principal point coordinates c_x and c_y equal $W/2$ and $H/2$, respectively, and the horizontal focal length f_x and the vertical focal length f_y in pixels are given by:

$$f_x = \frac{W}{2 \cdot \tan\left(\frac{\theta_x}{2}\right)} \quad f_y = \frac{H}{2 \cdot \tan\left(\frac{\theta_y}{2}\right)}$$

From this we can derive the camera intrinsics matrix K:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

For each pixel (u, v), with u being the horizontal pixel coordinate and v the vertical pixel coordinate of the image, we have $D(u, v) = Z_{cam}$ as the depth value at this pixel. Following the perspective projection formula [13], one could derive the the image coordinates P_{img} from the 3D coordinate point P_{cam}:

$$P_{img} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_{cam}} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \end{bmatrix} = \frac{1}{Z_{cam}} \cdot K \cdot P_{cam}$$

To obtain P_{cam}, one must then inverse the matrix K:

$$P_{cam} = \begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \end{bmatrix} = Z_{cam} \cdot K^{-1} \cdot P_{img} = D(u, v) \begin{bmatrix} \frac{u-c_x}{f_x} \\ \frac{v-c_y}{f_y} \\ 1 \end{bmatrix}$$

Using this formula, each pixel (u, v) can be transformed into a 3D point in the camera reference frame. The final step would involve transforming these 3D points into a different reference frame, such as the world frame or the robot base frame, with the transformation matrix T_{cam}^{base}, where R is a 3x3 rotation matrix and t is a 3x1 translation vector:

$$T_{cam}^{base} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

To transform P_{cam}, it must be converted into its homogeneous form, so that one can obtain P_{base}:

$$P_{base} = \begin{bmatrix} X_{base} \\ Y_{base} \\ Z_{base} \\ 1 \end{bmatrix} = T_{cam}^{base} \cdot P_{cam} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{bmatrix}$$

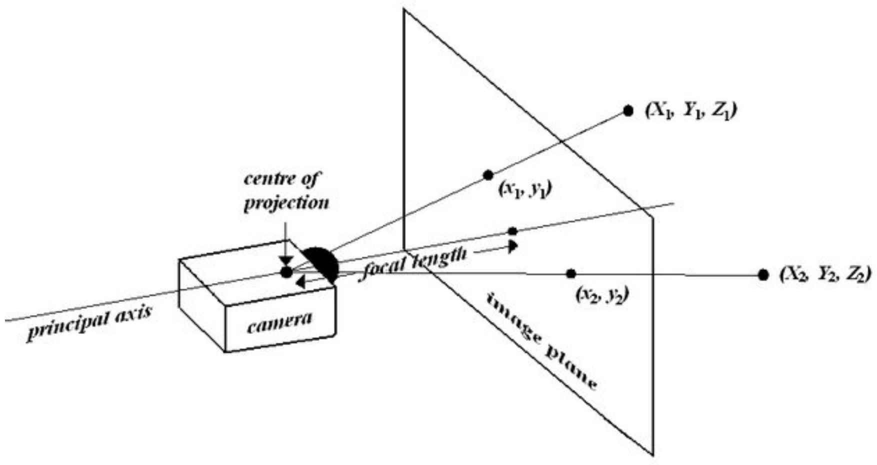

Figure 2.4: Visual representation of the perspective projection via the pinhole camera model [6]

Point Cloud to Octree

In the works of Wang et al. [52], a method is proposed that uses an Octree-based CNN approach for 3D shape analysis. As the input data is often in the form of a 3D point cloud, an approach to construct an octree from it was developed, with the resulting octree represented as a hierarchical tree. Each internal node stores references to its eight child nodes, known as octants, and each octant holds a list of indices corresponding to the points from the point cloud it contains. The process begins by defining a base octant as a 3D bounding cube that encloses all the points, and a maximum depth level is set to determine how many times the octants will be subdivided. Via breadth-first search, at each depth level l, the algorithm goes through every non-empty octant, dividing it into eight child octants at the next depth level $l + 1$. For each child octant, it checks whether it contains the points from within the parent octant. Octants become leaf nodes when they belong to the maximum depth level or when the number of encompassed points is below a predefined threshold, usually one point per octant. The process terminates once no more internal nodes are left to divide or the maximum depth is reached.

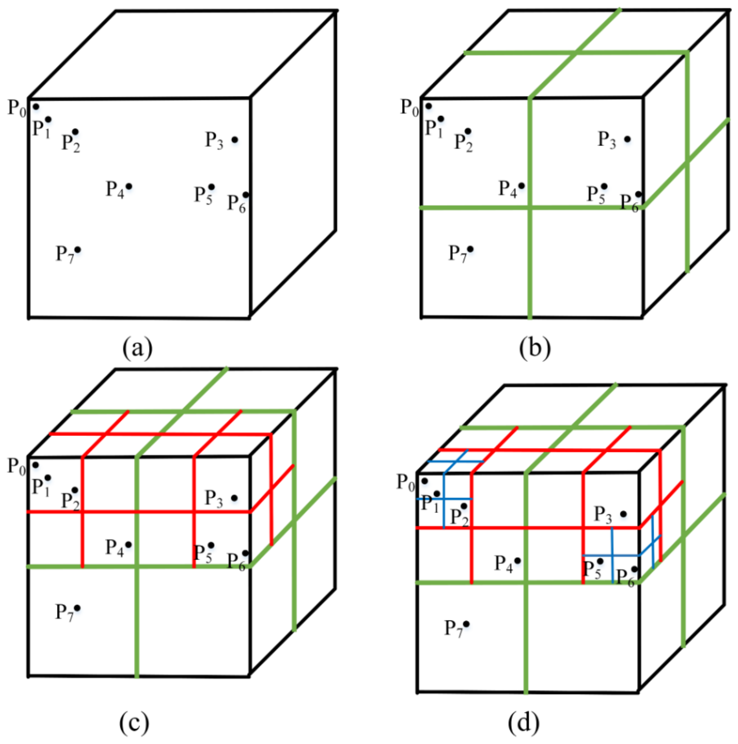

Figure 2.5: Example of the portrayal of the octree construction process with a decomposition threshold of two points points per octant [53]

2.2 Reinforcement Learning

2.2.1 Reinforcement Learning Problem

In RL, an agent interacts with its environment over time, aiming to learn which actions to take in order to maximize the cumulative reward received from the environment. The problem is modeled as an MDP, defined by a 5-tuple (S, A, P, R, γ), where S is the set of states, A is the set of actions, P represents the state transition probabilities $P(s_{t+1}|s_t, a_t)$, R denotes the reward function $R(s, a)$, and $\gamma \in (0, 1]$ is the discount factor. At each time step t, the agent observes the current state s_t, selects an action a_t following a learned policy $\pi(a_t|s_t)$, and receives a reward r_t based on its action and the current state. The process starts from an initial state $s_0 \sim P(s_0)$, and the Markov property ensures that the next state only depends on the current state and action rather than the history of prior states or actions [33]. The discounted cumulative reward the agent is supposed to maximize to learn the optimal policy is defined as:

$$R_t = \sum_{t' \geq 0} \gamma^{t'} \cdot r_{t+t'} \tag{2.1}$$

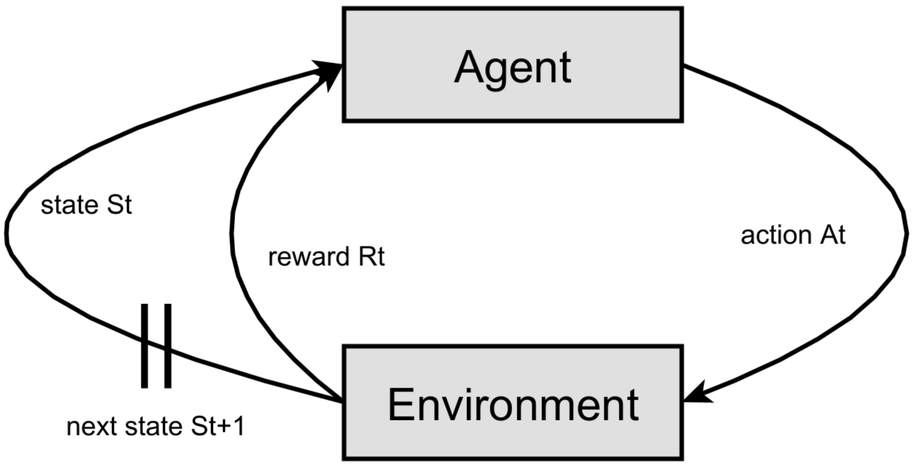

Figure 2.6: Visual representation of the reinforcement learning problem, where the agent is episodically interacting with its environment

2.2.2 Value Function and Q-Value Function

The value function at a state s predicts its expected cumulative reward, representing the expected return when following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\tau}\left[R_t \mid s_t = s, \pi\right] = \mathbb{E}_{\tau}\left[\sum_{t' \geq 0} \gamma^{t'} \cdot r_{t+t'} \mid s_t = s, \pi\right] \tag{2.2}$$

The Q-value function predicts the expected cumulative reward when taking action a in state s, while following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau}\left[R_t \mid s_t = s, a_t = a, \pi\right] = \mathbb{E}_{\tau}\left[\sum_{t' \geq 0} \gamma^{t'} \cdot r_{t+t'} \mid s_t = s, a_t = a, \pi\right] \tag{2.3}$$

Both functions are connected via the following relations:

$$Q^{\pi}(s,a) = \mathbb{E}_{r_t, s_{t+1}}\left[r_t + \gamma \cdot V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a, \pi\right] \tag{2.4}$$

$$V^{\pi}(s) = \mathbb{E}_{a_t}\left[Q^{\pi}(s, a_t) \mid s_t = s, \pi\right] \tag{2.5}$$

From this we get the following recursive relations:

$$V^{\pi}(s) = \mathbb{E}_{r_t, s_{t+1}}\left[r_t + \gamma \cdot V^{\pi}(s_{t+1}) \mid s_t = s\right] \tag{2.6}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{r_t, s_{t+1}, a_{t+1}}\left[r_t + \gamma \cdot Q^{\pi}(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\right] \tag{2.7}$$

There both exist an optimal state value $V^*(s)$, representing the maximum possible expected cumulative reward for a state s, as well as an optimal Q-value function $Q^*(s,a)$, which is the maximum possible expected cumulative reward for a state-action pair (a,s). From the optimal Q-value function, one can derive the optimal policy $\pi^*(s)$. They can be defined through the following equations:

$$V^*(s) := max_{\pi}\mathbb{E}_{\tau}\left[\sum_{t' \geq 0} \gamma^{t'} \cdot r_{t+t'} \mid s_t = s, \pi\right] \tag{2.8}$$

$$Q^*(s,a) := max_{\pi}\mathbb{E}_{\tau}\left[\sum_{t' \geq 0} \gamma^{t'} \cdot r_{t+t'} \mid s_t = s, a_t = a\pi\right] \tag{2.9}$$

$$\pi^*(s) := argmax_s Q^*(s,a) \tag{2.10}$$

Based on this we can derive the Bellman equation, where the optimal Q-function can be iteratively derived:

$$Q_i(s,a) = \mathbb{E}_{r_t, s_{t+1}}\left[r_t + \gamma \cdot max_{a_{t+1}} Q_{i-1}(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\right] \tag{2.11}$$

With:

$$Q_i \rightarrow Q^* \text{ for } i \rightarrow \infty$$

2.2.3 Difference Model-Based and Model-Free Learning

In the previous section, the agent models had the assumption that the transition probabilities of the Q-function can be viewed as a table with $\mid S \times A \mid$ entries. This is not scalable in most real-world tasks, especially if the observation space S or the action space A is continuous. One approach to deal with this would be to approximate the Q-function instead of viewing it as a lookup table and represent it in the following way [48]:

$$Q_i(s,a) \approx Q(s,a;\theta_i) \tag{2.12}$$

To approximate the Q-iteration one must then compute the approximator $Q(s,a;\theta_i)$ which minimizes the loss $L(\theta_i)$:

$$\theta_i = argmin_\theta L_i(\theta) \tag{2.13}$$

$$L_i(\theta) = \mathbb{E}_{s,a\sim\rho(s,a)}\left[(y_i - Q(s,a;\theta))^2\right] \tag{2.14}$$

With the target y_i as:

$$y_i = \mathbb{E}_{r,s'}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right] \tag{2.15}$$

2.2.4 On-Policy Learning

On-policy methods involve the agent directly learning and evaluating from its current behavior policy, requiring a balance between exploration and exploitation [50]. These methods are generally sample inefficient due to high variance but offer guaranteed convergence through direct gradient ascent. Common on-policy algorithms include REINFORCE, PPO, and SARSA [33].

2.2.5 Off-Policy Learning

Off-policy learning is a reinforcement learning approach where the agent iteratively learns an approximate value function by using stored experiences through exploration and experience replay. These experiences are stored in a buffer and reused for training at later stages. Off-policy methods involve indirect learning, where a distinction is made between the behavior policy used to generate training data and the target policy, which the agent is trying to optimize. One of the main advantages of off-policy learning is its sample efficiency, as it reuses past data to improve learning [50]. However, it has some drawbacks, such as the lack of guaranteed convergence and difficulties in handling high-dimensional action spaces (as it needs to compute $\max_a Q(s,a)$). Common algorithms that implement off-policy learning include the Deep Q-Network (DQN), the Deep Deterministic Policy Gradient (DDPG), and the Soft Actor-Critic (SAC).

Deep Q-Learning

DQN is a basic off-policy algorithm that approximates the loss and gradient using Monte Carlo integration, allowing efficient learning in environments with continuous state-action spaces. It utilizes a replay buffer for experience replay, storing past experiences to break the temporal correlation of the data, improving stability. To balance exploration and exploitation, the algorithm employs an ε-greedy strategy: with probability ε, it selects a random action to explore new possibilities, and with probability $1 - \varepsilon$, it exploits by choosing the action that maximizes the Q-value using the current best policy $\pi(s)$. The base algorithm can be seen below [39]:

Algorithm 1: Deep Q-learning with Experience Replay

```
Clear replay buffer $R$ with capacity $N$;
Initialize action-value function $Q(s, a; \theta)$ with random weights $\theta$;
for $episode = 1$ to $M$ do
    $\overline{\theta} \leftarrow \theta$, update weights of target network $Q(s, a; \overline{\theta})$;
    Set exploration policy $\hat{\pi}$ where action is selected
    $a \sim \hat{\pi}(a \mid s) = \begin{cases} \pi(s) = \max_a Q(s, a; \theta) & \text{with } p = 1 - \varepsilon \\ \text{random action} & \text{with } p = \varepsilon \end{cases}$ ;
    Initialize $s_1$;
    for $t = 1$ to $T$ do
        With exploration policy $\hat{\pi}(a_t \mid s_t)$ select an action $a_t$;
        Execute action $a_t$ and get reward $r_t$ and new observation $s_{t+1}$;
        Store transition samples $(s_t, a_t, r_t, s_{t+1})$ in $R$;
        Sample random minibatch of transitions $\{(s_k, a_k, r_k, s_{k+1})\}_K$ from $R$;
        $\forall\, k$, target $y_k = \begin{cases} r_k & \text{if } s_{k+1} \text{ is terminal} \\ r_k + \gamma \cdot \max_{a'} Q(s_{k+1}, a'; \overline{\theta}) & \text{otherwise} \end{cases}$ ;
        $\Delta \leftarrow -\frac{1}{K} \sum_{k=1}^{K} (y_k - Q(s_k, a_k; \theta)) \nabla_\theta Q(s_k, a_k; \theta)$ ;
        $\theta \leftarrow \theta - \alpha \cdot \Delta$ ;
    end for
end for
```

Double Deep Q-Learning

The off-policy algorithm of Double DQN, introduced by Van Hasselt et al. [51], improves upon the standard DQN algorithm by addressing the issue of overestimation bias. In Q-learning, the target value is typically computed as $y = r + \gamma \cdot max_{a'} Q(s', a'; \overline{\theta})$, where the target Q-function $Q(s', a'; \overline{\theta})$ is noisy and not necessarily reflective of the true optimal value. This leads to overestimation, as maximizing over noisy random variables results in $\mathbb{E}[max(X1, X2)] \geq max(\mathbb{E}[X1], \mathbb{E}[X2])$, where X_1 and X_2 are noisy estimates and the max operation overestimates the next values by maximizing over these noisy Q-values. Double DQN resolves this by decomposing the max operation, using one Q-network for action selection and another for value estimation. Specifically, the action is selected using one set of parameters θ_A, while the value is estimated using a different set of parameters θ_B, decorrelating the noise from both operations, thereby reducing the bias and improving the stability of the learning process. This is formulated as:

$$Q(s', a'; \theta_A) = r + \gamma \cdot Q(s', argmax_{a'}(s', a'; \theta_A); \theta_B)$$

$$Q(s', a'; \theta_B) = r + \gamma \cdot Q(s', argmax_{a'}(s', a'; \theta_B); \theta_A)$$

Usually, the target network is used as the second network, so when a sampling a transition from the buffer, the target y_k (in a non-terminal case) can be acquired via the following way:

$$a'_k = argmax_a Q(s_{k+1}, a; \theta) \rightarrow \text{ Action selection with current network } Q_\theta$$

$$y_k = r_k + \gamma \cdot Q(s_{k+1}, a'_k; \overline{\theta}) \rightarrow \text{ Action evaluation with target network } Q_{\overline{\theta}}$$

Deep Deterministic Policy Gradient

DDPG is an off-policy algorithm introduced by Lillicrap et al. [34] that differs from DQN in its ability to handle continuous action spaces. It uses the actor-critic structure, where the actor is responsible for updating the policy network parameters and selecting actions based on the current policy. At the same time, the critic evaluates the actions taken by the actor and estimates the Q-function parameters, providing feedback for the actor's policy updates. This separation of roles allows for more effective exploration than Q-Learning-based algorithms, as the critic's evaluation helps to guide the actor's learning process. To encourage exploration within the continuous action space, random noise (Ornstein–Uhlenbeck noise or zero-mean Gaussian noise) is added to the actions during training instead of using ε-greedy exploration, like in Q-learning. The pseudo-code for the DDPG algorithm can be observed in the following:

Algorithm 2: DDPG algorithm with actor and critic networks

```
Randomly initialize current networks $Q(s,a;\theta)$ & $\pi(s;\phi)$ with weights $\theta$ & $\phi$ ;
Initialize target networks $\overline{Q}(s,a;\overline{\theta})$ & $\overline{\pi}(s;\overline{\phi})$ with weights $\overline{\theta} \leftarrow \theta$ & $\overline{\phi} \leftarrow \phi$ ;
Initialize replay buffer $R$;
for $episode = 1$ to $M$ do
    Initialize a random noise process $\mathcal{N}$ for action exploration;
    Initialize $s_1$;
    for $t = 1$ to $T$ do
        Select an action $a_t = \pi(s_t;\phi) + \mathcal{N}_t$ ;
        Execute action $a_t$ and get reward $r_t$ and new observation $s_{t+1}$;
        Store transition samples $(s_t, a_t, r_t, s_{t+1})$ in $R$;
        Sample random minibatch of transitions $\{(s_k, a_k, r_k, s_{k+1})\}_K$ from $R$;
        $\forall$ $k$, set target $y_k = r_k + \gamma \cdot \overline{Q}(s_{k+1}, \overline{\pi}(s_{k+1};\overline{\phi});\overline{\theta})$ ;
        Update critic Q by minimizing loss $L(\theta) = \frac{1}{K}\sum_{k=1}^{K}\left(y_k - Q(s_k, a_k;\theta)\right)^2$ ;
        Update actor $\pi$ using the sampled gradient:
        $\nabla_\phi J(\phi) = \frac{1}{K}\sum_{k=1}^{K} \nabla_a Q(s_k, a;\theta)|_{a=\pi(s_k;\phi)} \nabla_\phi \pi(s_k;\phi)$ ;
        Update target networks by slowly tracking the current networks:
        $\overline{\theta} \leftarrow \tau\theta + (1-\tau)\overline{\theta}$ ;
        $\overline{\phi} \leftarrow \tau\phi + (1-\tau)\overline{\phi}$ ;
    end for
end for
```

Twin Delayed DDPG

The Twin Delayed DDPG (TD3) algorithm, developed by Fujimoto et al. [10], builds upon the actor-critic approach of DDPG but introduces specific modifications to address the overestimation bias issue. A distinguishing feature of TD3 is the use of two Q-networks, which allows it to take the minimum Q-value between them, thus reducing the risk of overly optimistic Q-value estimations. When updating Q-values, TD3 adds a small amount of Gaussian noise to the target policy action, increasing the model's robustness to outliers. Additionally, TD3 uses clipping to enforce action bounds without altering the output distribution. The target y is calculated as follows:

$$\hat{\pi}(s';\phi) = clip\left(\pi(s';\phi) + \mathcal{N}(0,\sigma), a_{low}, a_{high}\right)$$

$$y = r + \gamma \cdot \left(min_{i=1,2}\overline{Q}_i(s', \hat{\pi}(s';\phi);\overline{\theta}_i)\right)$$

Consequently, both Q-networks must be updated during each critic update step. Unlike the DDPG algorithm, which updates the actor and critic networks simultaneously, TD3 employs delayed policy updates, updating the policy network less frequently to improve the stability of Q-value estimates before each update.

Soft Actor-Critic

The off-policy algorithm Soft Actor-Critic (SAC) was conceived by Haarnoja et al. [11]. Similar to TD3, SAC utilizes two Q-networks to prevent overestimation bias; however, it distinguishes itself by employing a stochastic policy to enhance exploration in challenging environments, while TD3 applies this stochasticity only during Q-value updates. Additionally, SAC incorporates entropy into the reward function, encouraging the agent to favor actions with higher uncertainty, which leads to a broader exploration of the action space as well. Moreover, SAC updates its policy synchronously with the critic, resulting in more responsive learning - similar to DDPG - and contrasting with the delayed updates characteristic of TD3.

Algorithm 3: Soft Actor-Critic algorithm

```
Initialize replay buffer $R$, as well as learning rates $\lambda_Q, \lambda_\pi$, and parameters $\alpha, \tau, \gamma$;
Initialize actor net $\pi(s;\phi) = tanh\left(\mu(s;\phi) + \sigma(s;\phi) \cdot \mathcal{N}(0,1)\right)$ , with weights $\phi$ ;
Initialize current critic networks $Q_1(s,a;\theta_1), Q_2(s,a;\theta_2)$ with weights $\theta_1, \theta_2$ ;
Set target critic networks $\overline{Q}_1(s,a;\overline{\theta}_1), \overline{Q}_2(s;\overline{\theta}_2)$ with weights $\overline{\theta}_1 \leftarrow \theta_1, \overline{\theta}_2 \leftarrow \theta_2$ ;
for $episode = 1$ to $M$ do
    Initialize $s_1$;
    for $t = 1$ to $T$ do
        Select an action $a_t = \pi(s;\phi)$ ;
        Execute action $a_t$ and get reward $r_t$ and new observation $s_{t+1}$;
        Store transition samples $(s_t, a_t, r_t, s_{t+1})$ in $R$;
        Sample random minibatch of transitions $\{(s_k, a_k, r_k, s_{k+1})\}_K$ from $R$;
        $\forall\ k$, set $y_k = r_k + \gamma \cdot \left(min_{i=1,2}\overline{Q}_i(s_{k+1}, \pi(s_{k+1};\phi); \overline{\theta}_i) - \alpha \cdot \log \pi(s_{k+1};\phi)\right)$ ;
        Update $Q_1, Q_2$ via one step gradient descent:
        $\forall\ i \in \{1,2\}$: $L(\theta_i) = \frac{1}{K}\sum_{k=1}^{K}\left(y_k - Q_i(s_k, a_k; \theta_i)\right)^2$ ;
        $\forall\ i \in \{1,2\}$: $\theta_i \leftarrow \theta_i - \lambda_Q \cdot \nabla_{\theta_i} L(\theta_i)$ ;
        Update actor $\pi$ using the sampled gradient:
        $L(\phi) = \frac{1}{K}\sum_{k=1}^{K}\left(min_{i=1,2}Q_i(s_k, \pi(s_k;\phi); \theta_i) - \alpha \cdot \log \pi(s_k;\phi)\right)$ ;
        $\phi \leftarrow \phi - \lambda_\pi \cdot \nabla_\phi L(\phi)$ ;
        Update target networks by slowly tracking the current networks:
        $\forall\ i \in \{1,2\}$: $\overline{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\overline{\theta}_i$ ;
    end for
end for
```

Truncated Quantile Critics

Truncated Quantile Critics (TQC) is an advanced off-policy RL algorithm based on the SAC algorithm implemented by Kuznetsov et al. [26]. In order to further tackle the issue of overestimation bias, it introduced several key innovations: first, TQC uses a distributional representation of critics, where instead of only estimating the expected return $Q(s,a;\psi) = \mathbb{E}[Z(s,a;\psi)]$ for a state-action pair, it estimates the entire return distribution $Z(s,a;\psi)$, represented as:

$$Z(s,a;\psi) = \frac{1}{M}\sum_{m=1}^{M} \delta\left(\theta_{\psi}^{m}(s,a)\right)$$

where $\{\theta_{\psi}^{1}, ..., \theta_{\psi}^{M}\}$ are outputs of the critic network Z_{ψ}, with M denoting the number of atoms (quantiles) and δ as the Dirac delta function. Additionally, TQC leverages an ensemble of multiple critics by employing N separate critic networks $\{Z_{\psi_1}, ..., Z_{\psi_N}\}$, each approximating the return distribution, instead of using two Q-functions as in the SAC algorithm. Each network outputs M quantiles, forming a distribution set $\mathcal{Z}(s,a) = \{z_{(i)}(s,a) = \theta_{\psi_n}^{m}(s,a) \mid n \in [1..N],\ m \in [1..M]\}$. During implementation, quantiles from target networks $Z_{\overline{\psi_n}}$ populate $\mathcal{Z}$ as $z_{(i)}(s,a) = \theta_{\overline{\psi_n}}^{m}(s,a)$. Target values y_i, for $i \in [1..MN]$, are computed as:

$$y_i(s,a) = r(s,a) + \gamma \cdot [z_{(i)}(s',a') - \alpha \cdot \log \pi(s';\phi)]$$

To further mitigate overestimation, TQC truncates the critics' predictions by sorting the MN elements $z_{(i)}(s',a')$ of $\mathcal{Z}(s',a')$ and keeping only the kN lowest values. The target distribution is then:

$$Y(s,a) = \frac{1}{kN}\sum_{i=1}^{kN} \delta(y_i(s,a))$$

where $y_i(s,a)$ is taken from one of the kN lowest $z_{(i)}(s',a')$ values. The parameter k can be tuned based on the task; a larger k permits some overestimation, while a smaller k may result in information loss. Loss computation for the critic in TQC involves calculating the 1-Wasserstein distance between current critics $Z_{\psi_n}, n \in [1..N]$ and the target distribution $Y(s,a)$, where for each n:

$$L(s,a;\psi_n) = \frac{1}{kNM}\sum_{m=1}^{M}\sum_{i=1}^{kN} \rho_{\tau_m}^{H}\left(y_i(s,a) - \theta_{\psi_n}^{m}(s,a)\right)$$

with $\tau_m = \frac{2m-1}{2M}$ and $\rho_{\tau_m}^{H}(x) = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \le \tau_m \\ \tau_m \cdot (|x| - \frac{1}{2}\tau_m) & \text{for } |x| > \tau_m \end{cases}$

Finally, the loss for the actor involves a non-truncated estimation of the Q-value:

$$L(s;\phi) = \alpha \cdot \log \pi(s;\phi) - \frac{1}{NM} \sum_{m=1,n=1}^{M,N} \theta_{\psi_n}^{m}(s, \pi(s;\phi))$$

This structure allows TQC to effectively manage overestimation while leveraging ensemble critics and distributional representations for robust policy learning.

Algorithm 4: Truncated Quantile Critics algorithm

```
Set number of critics $N$, number of quantiles $M$, as well as truncation factor $k$ ;
Initialize replay buffer $R$, as well as learning rates $\lambda_\psi, \lambda_\pi$, and parameters $\alpha, \tau, \gamma$;
Initialize actor net $\pi(s;\phi) = tanh\,(\mu(s;\phi) + \sigma(s;\phi) \cdot \mathcal{N}(0,1))$ , with weights $\phi$ ;
Initialize current critic networks $Z(s,a;\psi_1), ..., Z(s,a;\psi_N)$ with weights $\psi_1, ..., \psi_N$ ;
Set target critic networks weights: $\overline{\psi}_1 \leftarrow \psi_1, ..., \overline{\psi}_N \leftarrow \psi_N$ ;
**for** *episode* $= 1$ *to* $M$ **do**
    Initialize $s_1$;
    **for** $t = 1$ *to* $T$ **do**
        Sample action $a_t = \pi(s;\phi)$ and perform forward step to obtain $r_t$ and $s_{t+1}$;
        Store transition samples $(s_t, a_t, r_t, s_{t+1})$ in $R$;
        Sample random minibatch of transitions $\{(s_b, a_b, r_b, s_{b+1})\}_B$ from $R$;
        **for** $b = 1$ *to* $B$ **do**
            $a_{b+1} = \pi(s_{b+1}, \phi)$ ;
            $\forall\ n \in [1..N],\ m \in [1..M]:\ z_{(i)}(s_{b+1}, a_{b+1}) = \theta_{\psi_n}^{m}(s_{b+1}, a_{b+1})$ ;
            Sort all atoms $z_{(i)}(s_{b+1}, a_{b+1})$ in ascending order ;
            Truncate highest $(k - M)N$ atoms ;
            Compute target distribution $\forall\ i \in [1..kN]$:
            $y_i(s_b, a_b) = r(s_b, a_b) + \gamma \cdot \left( z_{(i)}(s_{b+1}, a_{b+1}) - \alpha \cdot \log \pi(s_{b+1};\phi) \right)$ ;
        **end for**
        Update critic networks $\{Z_{\psi_1}, ..., Z_{\psi_N}\}$ via one step gradient descent:
        $L(\psi_n) = \frac{1}{B} \frac{1}{kNM} \sum_{b=1}^{B} \sum_{m=1}^{M} \sum_{i=1}^{kN} \rho_{\tau_m}^{H} \left( y_i(s_b, a_b) - \theta_{\psi_n}^{m}(s_b, a_b) \right)$ ;
        $\psi_n \leftarrow \psi_n - \lambda_\psi \cdot \nabla_{\psi_n} L(\psi_n)$ ;
        Update actor $\pi$ using the sampled gradient:
        $L(\phi) = \frac{1}{B} \sum_{b=1}^{B} \left( \alpha \cdot \log \pi(s_b;\phi) - \frac{1}{NM} \sum_{m=1,n=1}^{M,N} \theta_{\psi_n}^{m}(s_b, \pi(s_b;\phi)) \right)$ ;
        $\phi \leftarrow \phi - \lambda_\pi \cdot \nabla_\phi L(\phi)$ ;
        Update target critic networks by slowly tracking the current networks:
        $\forall\ n \in [1..N]$: $\overline{\psi}_n \leftarrow \tau\psi_n + (1 - \tau)\overline{\psi}_n$ ;
    **end for**
**end for**
```

3 Related Work

3.1 Motion Planning in Robotics

3.1.1 Global Path Planning

Single-Query Planners and Multi-Query Planners

Most path-planning approaches in robotics rely on sampling methods, as they can efficiently handle high-dimensional and complex workspaces where exhaustive exploration is unfeasible. Sampling-based planners are generally divided into single-query and multi-query methods [4]. Single-query planners construct paths specific to a unique start-goal pair, prioritizing speed and flexibility, making them ideal for robots performing one-off tasks in dynamic environments. Examples of single-query planners include Rapidly-exploring Random Trees (RRT) by LaValle and Kuffner [28] and Expansive Space Trees (EST) by Hsu et al. [16]. In contrast, multi-query planners typically use pre-built road-maps to represent the navigable space, allowing the efficient reuse of paths across multiple queries. This approach suits robots in known, static environments where paths are reused for different destinations, such as the navigation inside a warehouse. Prominent multi-query planners include the Probabilistic Road-map Method (PRM) by Kavraki et al. [19] and the SPARS Road-map Spanner algorithm (SPARS) by Dobson and Bekris [7].

Rapidly-exploring Random Trees

RRT is a single-query search algorithm that incrementally constructs a tree of feasible paths in a given configuration space X. The tree is defined as a graph $G = (V, E)$, where V represents the vertices and E the edges. Starting from an initial point x_{init} added to V, the algorithm iteratively extends the tree while navigating around an obstacle region $X_{obs} \subset X$ that must be avoided. In each iteration step, the algorithm samples a random point x_{rand} from the obstacle-free region $X_{free} = X \setminus X_{obs}$, finds the nearest node x_{near} to x_{rand} using a nearest neighbor search, and then generates a new point x_{new} on the path (x_{near}, x_{rand}) that is within a maximum incremental distance Δx to x_{near}. If the new path segment (x_{near}, x_{new}) is obstacle-free, it is added to the set of edges E, and x_{new} gets added to V. This process repeats until a stopping condition is met, either after a predefined number of steps or when a vertex reaches the goal region [28]. Below is the pseudo-code describing the steps of the algorithm:

Algorithm 5: Rapidly-exploring Random Trees algorithm (RRT)

```
Define the ending condition $T$ and the maximum step size $\Delta x$ ;
$V \leftarrow \{x_{init}\}; E \leftarrow \varnothing$ ;
repeat
    $x_{rand} \leftarrow SAMPLE\_CONFIG(X \setminus X_{obs})$ ;
    $\_, \_ \leftarrow EXTEND(G, x_{rand}, \Delta x)$ ;
until $T$ = True;
return $G = (V, E)$

Function $EXTEND$(Tree $G = (V, E)$, Target $x_{target}$, Threshold $\Delta x$) ;
    $x_{near} \leftarrow NEAREST\_NEIGHBOR(V, x_{target})$ ;
    $x_{new} \leftarrow \begin{cases} x_{target} & \text{if } |x_{target} - x_{near}| \leq \Delta x \\ x_{near} + SIGN(x_{target} - x_{near}) \times \Delta x & \text{otherwise} \end{cases}$ ;
    if $OBSTACLE\_FREE(x_{near}, x_{new})$ then
        Update G: $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{near}, x_{new})\}$ ;
        if $x_{new} = x_{target}$ then
            return $x_{new}$, Reached ;
        end
        else
            return $x_{new}$, Advanced ;
        end
    end
    return $\varnothing$, Collision ;
```

The RRT algorithm is known for its simplicity and limited computational costs, making it an efficient choice for path planning. One of its notable properties is the distribution of vertices, which closely aligns with the sampling distribution, allowing for an expansion that is biased towards unexplored spaces. The tree maintains its connectivity throughout its construction, ensuring that a path is guaranteed if the goal region is reached; however, this does not guarantee that the path will be (nearly) optimal. Overall, these features contribute to the high confidence in RRT's ability to find a solution path, making it a probabilistically complete algorithm [28].

RRT-Connect

RRT-Connect, an extension of the RRT algorithm introduced by Kuffner and LaValle in 2000 [24], builds two trees, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where one grows from the start node and the other from the goal node, with the goal of connecting them. This bidirectional growth approach enables the algorithm to find feasible paths more efficiently, reducing the required number of samples to reach the goal and thereby accelerating pathfinding. During each iteration, the first tree is extended with a newly sampled point, and a connect-heuristic is used to attempt to link the second tree to this new node by repeatedly extending it until it either connects with this node or encounters an obstacle. At the end of each iteration, the trees are swapped to continue the search from both directions.

Algorithm 6: RRT-Connect Algorithm

```
Given: start point $x_{start}$, goal point $g_{goal}$, maximum iterations $N$ and step size $\Delta x$ ;
Define start Tree: $G_1 = (V_1 \leftarrow \{x_{start}\}, E_1 \leftarrow \varnothing)$ ;
Define goal Tree: $G_2 = (V_2 \leftarrow \{x_{goal}\}, E_2 \leftarrow \varnothing)$ ;
repeat
    $x_{rand} \leftarrow SAMPLE\_CONFIG(X \setminus X_{obs})$ ;
    $x_{new1}, \_ \leftarrow EXTEND(G_1, x_{rand}, \Delta x)$ ;
    if $x_{new1} \neq \varnothing$ then
        $\_, S \leftarrow CONNECT(G_2, x_{new1}, \Delta x)$ ;
        if $S = Reached$ then
            return $PATH(G_1, G_2)$ ;
        end
        $SWAP(G_1, G_2)$ ;
    end
until $i = N$;
return $Failure$

Function $CONNECT($Tree $G = (V, E)$, Target $x_{target}$, Threshold $\Delta x)$ ;
    repeat
        $\_, S \leftarrow EXTEND(G, x_{target}, \Delta x)$ ;
    until $S \neq Advanced$;
return $S$
```

3.1.2 Local Path Planning via Inverse Kinematics Solving

Kinematics and Dynamics Library with Random Restarts

The Kinematics and Dynamics Library with Random Restarts (KDL-RR) is an improved version of the Orocos Kinematics and Dynamics Library (KDL) method [20]. The original KDL approach employs a pseudo-inverse Jacobian method with joint-limit constraints, using Newton's method for iterative solutions. KDL-RR builds on this by incorporating time-based iterations rather than count-based ones and introducing random restarts to help escape local minima. While KDL-based methods often achieve faster solutions for simpler problems, they have a lower success rate than Sequential Quadratic Programming (SQP) methods, which can handle nonlinear optimization problems.

Sequential Quadratic Programming - Sum of Squares

The Sequential Quadratic Programming - Sum of Squares (SQP-SS) method is based on the standard iterative SQP algorithm [25], which frames the Inverse Kinematics (IK) problem as a nonlinear optimization task. This approach enforces joint limit constraints and employs the Broyden-Fletcher-Goldfarb-Shanno algorithm, a Newton-approximation technique, as its iterative search method. It also incorporates random restarts to avoid local minima. Unlike base SQP, SQP-SS minimizes the sum of squares of the Cartesian error vector rather than joint space errors, making it computationally less expensive. While SQP generally takes longer to compute solutions than KDL-based methods, it outperforms them in more complex configurations.

TRAC-IK

TRAC-IK, developed by Beeson and Ames in 2015 [2], is a dual-threaded IK-solver that simultaneously runs the algorithms KDL-RR and SQP-SS. Whichever solver finds a solution first is selected, and both processes subsequently halt. By leveraging these solvers in parallel, TRAC-IK achieves a higher success rate in solving IK-problems, as well as improved solving speed.

3.2 Deep Learning in 3D Space

3.2.1 Deep Learning for 3D Point Cloud Processing

PoinNet

PointNet is a deep learning network designed to classify 3D objects and segment 3D objects or scenes, all represented as point clouds. Unlike other previous approaches, PointNet does not require a transformation of the point-cloud data into voxel grids or mesh representations. Instead, it directly operates on raw point-cloud data, treating each 3D point independently. This approach not only eliminates the risk of information loss associated with representation transformations and offers faster computation but also exploits the efficient surface representation of objects that come from point clouds [45].
The architecture of PointNet takes a point cloud as input, represented as a set of n 3D points, where each point p_i is a vector of dimension d that includes XYZ coordinates and may also incorporate normals or color features. In the point-cloud encoder, each individual point is mapped from the input dimension to first 64 and later to 1024 dimensions using shared point-wise multi-layer perceptron (MLP) layers and feature transformations to extract local structures and patterns from the point cloud. Subsequently, max pooling is employed to aggregate these extracted features into a single global feature vector. For classification tasks, PointNet then employs three consecutive MLP layers to reduce the dimensionality of the feature vector to k, where k represents the number of classes. In contrast, for semantic segmentation tasks, the model concatenates global features with point-wise features and passes them through several MLP layers to generate per-point scores for each of the n points. A visual representation of PointNet's architecture can be seen in Figure 3.1.
PointNet has demonstrated solid results on several 3D machine learning tasks. In 3D object classification, it achieves 89.2% accuracy on the ModelNet40 data-set, a widely used benchmark for 3D object classification tasks. For 3D semantic segmentation, it reaches a mean Intersection-over-Union (mIoU) of 47.6% and an overall accuracy of 78.5% on the S3DIS data-set, which is a standard benchmark for 3D semantic segmentation tasks [47].
Since point clouds are considered unordered sets, it is a crucial attribute of PointNet to be permutation invariant, which can achieved through a pooling operation. Additionally, the network is able to learn to be invariant to geometric transformations by aligning the point-cloud input to a canonical space. This robustness extends to input corruption: PointNet maintains nearly the same classification performance even when 50% of the points are missing, and it achieves over 80% classification accuracy, with 20% of the points being outliers. This capability makes PointNet highly suitable for

applications involving noisy data, such as those generated by imperfect real-world sensors like RGB-D cameras. Although newer models have surpassed PointNet in these tasks, PointNet remains popular due to its simple and efficient architecture, offering fast inference times that make it attractive for applications prioritizing speed over marginal gains in accuracy.

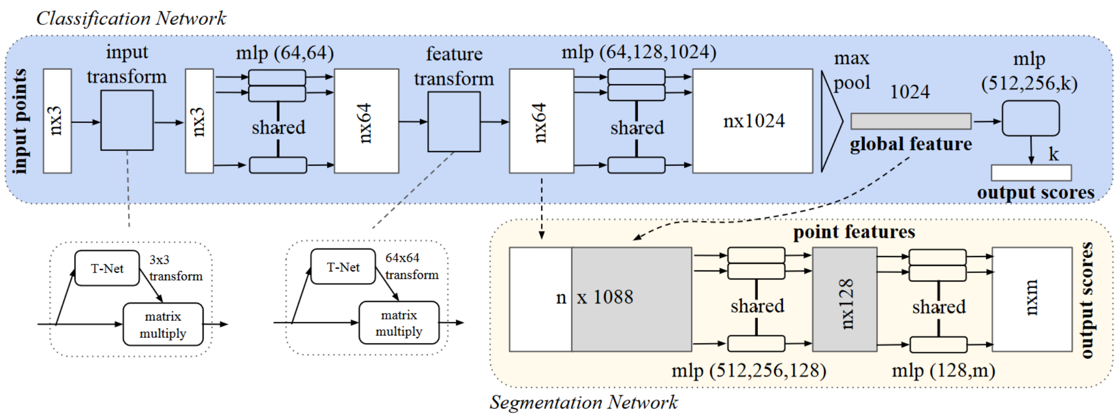

Figure 3.1: *The architecture of the PointNet classification and segmentation networks*: the encoder takes in a point cloud of shape (n, d) and maps it to a feature vector of shape $(n, 1024)$. These features are aggregated into a one-dimensional global feature vector, which can then be inputted into the final classifying layers for object classification. For segmentation, the intermediate point-wise features of shape $(n, 64)$ are concatenated with the tiled global features before going through the mlp-layers of the segmentation network.

PoinNet++

PointNet++ is a Convolutional Neural Network (CNN) model for processing 3D point clouds implemented by Qi et al. [46]. It builds upon the base PointNet model and addresses some of its limitations, thus improving overall performance.
The PointNet++ model encodes point-cloud data through a hierarchical feature learning approach that captures local neighborhood structures at various scales. This is achieved by recursively applying a set abstraction process consisting of a sampling layer, a grouping layer, and a PointNet layer. The sampling layer takes an input of shape $N \times (d + C)$, where N is the number of points, d is the coordinate dimension (typically 3), and C is the number of features. From this point set, a subset of N' centroids is selected via Farthest Point Sampling (FPS), with each centroid representing one local region. In the grouping layer, points are assigned to these local regions using K-Nearest

Neighbors (KNN) or a ball query that finds all points within a specified radius of each centroid. This results in an output of shape $N' \times K \times (d + C)$, where N' represents the number of local regions, and K is the number of points per region (which may vary). The grouped points are then processed by a PointNet encoder, yielding features of shape $N' \times (d + C')$, where C' is the number of features returned by the encoder. For classification tasks, PointNet++ first obtains a global feature vector by applying a final PointNet layer, followed by a series of MLP layers to produce class scores. For semantic segmentation, a decoder progressively upsamples the low-resolution feature map from the encoder, increasing the spatial resolution through consecutive interpolation and unit-PointNet layers applied to each local region. Skip connections link corresponding encoder and decoder layers, preserving local and global features lost during downsampling. The overall architecture of the model can be seen in Figure 3.2:

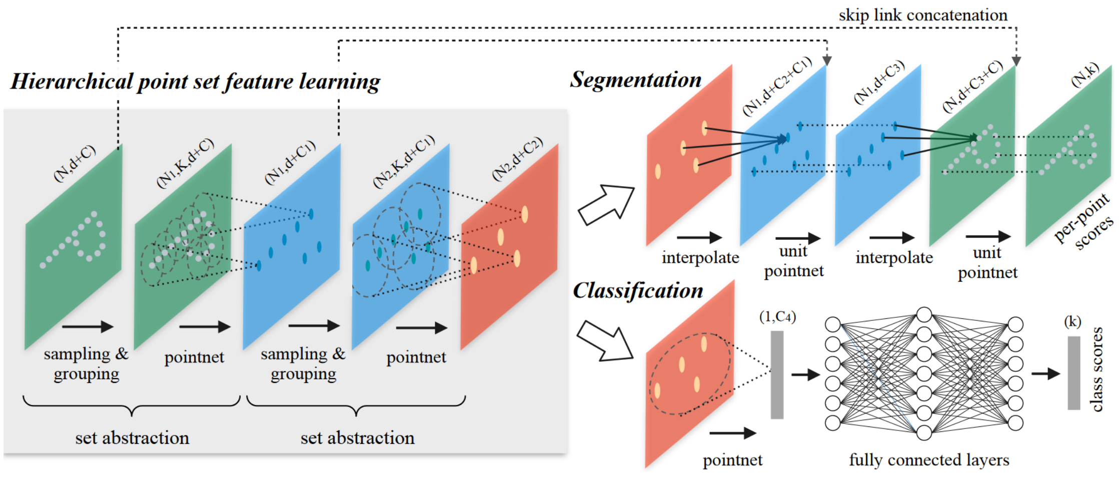

Figure 3.2: *The architecture of the PointNet++ network*: the point-cloud data is encoded via "hierarchical point set feature learning", consisting of consecutive sampling, grouping, and PointNet layers. In a classification task, the resulting features then are inputted into the classification network to obtain the object class scores. For semantic segmentation, the feature map is progressively upsampled while also utilizing the encoder features of the same dimension via a skip connection.

The PointNet++ model demonstrates superior performance over the base PointNet model in 3D object classification and semantic segmentation of 3D scenes. It achieves a 3D object classification accuracy of 91.9%, and it attains a mIoU of 54.5% and an overall accuracy of 81.0% for semantic segmentation tasks, highlighting its ability to accurately segment complex 3D scenes.

3.2.2 Deep Learning with Octree Representation

Octree-Based Convolutional Neural Network (O-CNN)

The O-CNN model, introduced by Wang et al. [52], presents an innovative approach to 3D shape analysis, utilizing an octree structure as input. By focusing operations solely on the occupied octants of a 3D shape, the model achieves significant efficiency gains compared to traditional voxel-based CNN methods, which process an entire volumetric grid regardless of occupancy. This reduces the computational load while enhancing the network's capacity to handle complex 3D shapes effectively.
One of the model's key features is its GPU-optimized octree structure, which facilitates efficient storage and processing on GPUs. Its main component is a sparse Octree-CNN operation, which restricts computations to surface octants, optimizing operations on the sparse octree representation. Additionally, the model uses a hash table for efficient neighbor searches, enabling convolutions on localized regions around each octant. The model architecture begins with converting a point cloud into an octree of a maximum depth d_{max}, with each octant containing f features, such as occupancy, normal vectors, and the mean distance from points to the octant center. Through a series of sparse octree convolutions, ReLU activations, and pooling layers, the data is encoded into a lower-dimensional octree, commonly with a depth of $d_l = 2$, following a similar structure as the LeNet network by LeCun et al. [29]. The last step of the encoding process consists of reshaping an octree with 16 features and depth 2 into a 4x4x4 voxel structure. For object classification tasks, this voxel data is flattened into a 1024-dimensional feature vector, which is then processed through two consecutive dropout and MLP layers before the final object scores are outputted via a softmax layer. For shape part segmentation tasks, the encoded data is passed through a mirrored decoder structure, using unpooling and deconvolution layers, to reconstruct the octree to its original maximum depth d_{max}.
In terms of performance, the O-CNN model demonstrates strong results for classification tasks on the ModelNet40 data-set. Models with an octree depth of $d_{max} = 6$ achieve an accuracy of 89.9%, even slightly outperforming the PointNet model. When reducing the octree depth to $d_{max} = 4$, the O-CNN model still maintains a competitive accuracy of 88.3%, albeit slightly below PointNet's results. Similar performance trends are observed on the ShapeNet data-set for object part segmentation [54], where O-CNN models with depths of 5 and 6 slightly surpass PointNet in accuracy, proving the model's effectiveness across various 3D shape analysis tasks.

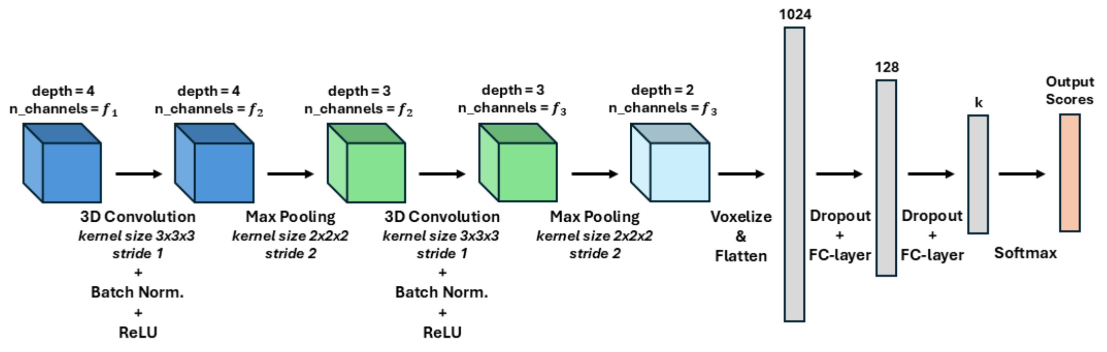

Figure 3.3: *The architecture of the O-CNN classification network of max depth 4*: the octree data is progressively encoded to a lower dimension via sparse 3D convolutions, batch normalization, and ReLU layers, as well as a max pooling operation. Once the octree data reaches depth level 2, it is converted into a voxel grid and flattened into a 1-D feature vector. The data then goes through two subsequent dropout and MLP layers before obtaining the class scores via a softmax operation.

3.3 Vision-Based Robotic Grasping

3.3.1 Task Subdivision

The planning phase of a vision-based robotic grasping task is typically divided into three key stages: object pose estimation, grasp pose determination, and path planning [21]. Both the analytic method and the model-based data-driven method follow this structured approach. In contrast, the model-free data-driven method bypasses the first intermediate step by directly estimating the grasp pose from sensory input, while the end-to-end method goes even further by directly mapping the sensory input to planned actions. After the planning stage, the task proceeds to execute the generated actions, which may involve closed-loop control to ensure the planned trajectory is accurately followed. A visual representation of the system's pipeline can be seen in Figure 3.4:

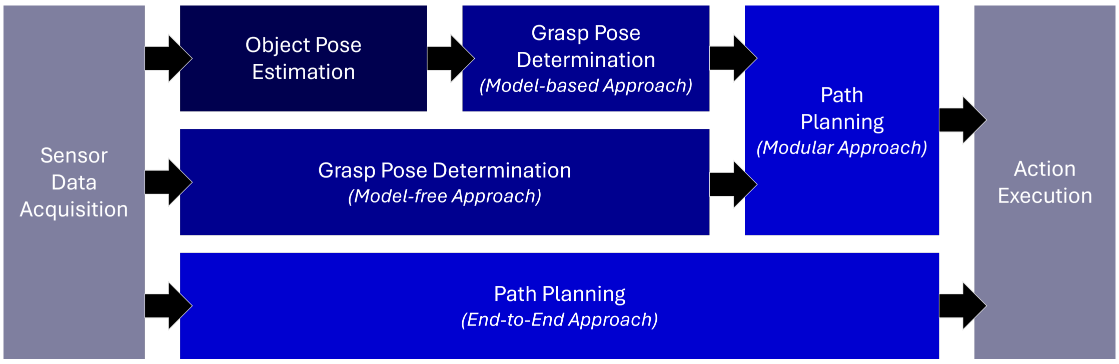
Sensor Data Acquisition
Object Pose Estimation
Grasp Pose Determination (Model-based Approach)
Grasp Pose Determination (Model-free Approach)
Path Planning (Modular Approach)
Path Planning (End-to-End Approach)
Action Execution

Figure 3.4: *Stages of the Vision-based Robotic Grasping Task*: differentiation between model-based methods, where the object's pose is determined before generating a grasping pose, and model-free methods, where the grasping pose is directly generated from the sensory input. In the end-to-end approach, the robot's actions are directly mapped from the sensory input.

Object Pose Estimation

The object pose estimation involves determining the position and orientation of an object relative to a sensor's reference frame based on sensory inputs such as RGB images, depth images, RGB-D images, or point clouds. This task is challenging due to factors like sensor noise, lighting variations, object occlusion, and diverse object characteristics such as shape, surface texture, and density. Training a supervised deep learning model using synthetic data generated in simulation environments is a practical and cost-effective approach, as it allows for the easy and inexpensive collection of large amounts of annotated 3D data. To ensure adequate performance in real-world scenarios, it is essential to employ sim-to-real transfer techniques, including domain randomization and domain adaptation [21].

Grasp Pose Determination

Grasp pose determination involves deriving a suitable grasp pose based on the estimated object pose and geometry, with the primary goal of generating a grasp that maximizes stability while minimizing the risk of slippage or damage to the object. While analytic methods can be employed for this task, the state-of-the-art approach typically leverages CNNs, which process object features to predict an optimal grasp pose.

Path Planning

The final stage involves generating a feasible, collision-free path to reach the desired grasp pose and carry out the grasp. Depending on the specific use case, the planning may involve high-level Cartesian pose changes, where lower-level joint actions are derived using an IK solver or low-level joint configurations directly. For a comprehensive overview of path planning in robotics, refer to Section 3.1.

Action Execution

In the works of Fantoni et al. [8], the grasping process in robotics is divided into several sequential steps to ensure successful object manipulation. It begins with the *object approaching* phase, where the robot positions itself within a suitable range of the target object. The next phase is *coming into contact with the object*, which involves getting the gripper into the proper grasp position and ensuring it is open so that the gripper fingers can grab the object. Once in position, the robot moves to *securing the object* by applying a force with the gripper's fingers sufficient to eliminate all degrees of freedom, creating a rigid attachment. With the object firmly grasped, the robot proceeds to the step of *moving the object* to the desired location. Finally, the process concludes with *releasing the object*, as the gripper opens its fingers to place the object in the target position.

3.3.2 Categorization of Methods

Analytic Method

The analytic method is a hard-coded approach that frames the task of finding a suitable grasp configuration as a constrained optimization problem. The criteria for this optimization problem are derived from knowledge of the environment, such as surrounding physics, the robot's characteristics, including joint kinematics and gripper forces, and the properties of the object to be grasped, such as geometry, texture, and weight. This method is highly efficient for more straightforward scenarios involving known objects but faces challenges when generalizing to cases with partial knowledge, also known as "grasping familiar objects". Moreover, it is entirely unsuitable for grasping unknown objects [3].

Model-based Data-driven Method

The model-based data-driven method leverages specific knowledge about the object to be grasped, such as its CAD model (if the object is fully known) or information derived from sensor data (if the object is partially known or unknown). This approach involves estimating the object's pose based on the available knowledge, determining an appropriate grasping pose, and finally generating the necessary actions for execution. While this method is highly efficient and accurate when detailed information about the object is available, it becomes impractical when the information is only partial in a more complex environment. The reliance on object-specific knowledge makes it unsuitable for scenarios requiring the grasping of completely unknown items [21].

Model-free Data-driven Method

In the model-free data-driven method, the agent directly estimates the grasping pose from sensor inputs without needing to explicitly determine the object's pose, thereby eliminating the requirement for prior knowledge about the object. This approach offers significant advantages, such as better generalization to novel objects, as it does not rely on object-specific knowledge and simplifies training by omitting the "Object Pose Estimation" step. However, the absence of detailed object information limits its applicability to lower-precision grasping tasks [21].

End-to-End Method

The end-to-end method is a specific case of the model-free data-driven approach where the task is no longer decomposed into distinct stages, and the agent directly maps actions from sensor inputs instead [31]. In this approach, the agent's policy can either directly output joint actions or output high-level actions at a lower frequency, with an inverse kinematics solving algorithm used for the low-level joint actions. A notable implementation of this method is the work by Kalashnikov et al. [18], where an end-to-end approach was applied to a vision-based robotic grasping task. In their study, the agent learns a policy using the *QT-Opt* algorithm, a novel RL method that integrates both off-policy and on-policy training. The process begins with off-policy learning, where the agent is trained on a large-scale data-set of real-world grasp attempts. The agent is then fine-tuned through on-policy training, where new data is collected from grasp attempts using the learned *QT-Opt* policy, along with some exploration to further improve the model's performance.

3.3.3 Learning Approaches

Supervised Learning Approach

The supervised learning approach in robotic grasping involves training a model with labeled data to learn grasping behaviors. The goal is to determine a grasp configuration, which typically includes the position, orientation, and gripper opening. Most supervised learning methods can be divided into the discriminative and the generative approach. The *discriminative approach* typically involves two stages. In the first stage, potential grasp candidates are sampled either exhaustively using a compact deep network or based on geometric features, such as high-gradient regions in the depth image. Then, a larger deep network ranks the rectangles based on their likelihood of representing a successful grasp. While effective, this method usually has a long run time due to the need to execute a forward pass for every candidate. It relies on labeled grasp data where the success or failure of each attempt is known [21]. Lenz et al. [30] proposed a deep learning framework for grasp detection from RGB-D images where the entire model was trained using the "Cornell Grasp Dataset" [17], which contains RGB-D images annotated with multiple grasp rectangles, each labeled as either successful or unsuccessful grasps. The "Dex-Net 2.0" method, proposed by Mahler et al. [35], utilizes a network trained on a data-set of the same name to rank grasp candidates formed by antipodal pairs of depth image pixels based on their quality scores. In contrast, the *generative approach* directly generates grasp hypotheses in a single forward pass, making it faster than the discriminative method. However, a large data-set is required to learn the joint distribution of grasp configurations [21]. Morrison et al. [40] proposed the Generative Grasping CNN, which uses depth images to predict the quality, angle, and width of potential grasps at every pixel. The best configuration is selected based on the pixel with the highest quality score.

Imitation Learning Approach

In Imitation Learning, a robotic agent is able to acquire the optimal behavior by learning from demonstrations provided by a teacher [1], as illustrated in Figure 3.5. These demonstrations are typically presented as sequences of state-action pairs, framing the problem as an MDP. In vision-based robotic manipulation tasks, the observations generally encompass the robot's sensory inputs and action measurements. Two primary approaches to Imitation Learning are Inverse Reinforcement Learning (IRL) and Behavioral Cloning (BC). IRL focuses on extracting the underlying reward function from the observed optimal behavior. It can offer strong generalization to unseen scenarios but often involves a complex implementation and requires high computational costs [41]. Conversely, BC, introduced by Pomerleau in 1989 [44], directly learns a policy that maps observations to actions by treating the problem as a supervised learning task, aiming to minimize the error between the predicted and demonstrated actions. However, BC struggles to generalize to unseen situations and demands diverse, high-quality demonstrations. A notable implementation of BC in vision-based robotic manipulation was done by Kopicki et al. [22], where a robot learned to grasp novel objects from kinesthetic demonstrations, using point clouds from a depth camera as sensory input.

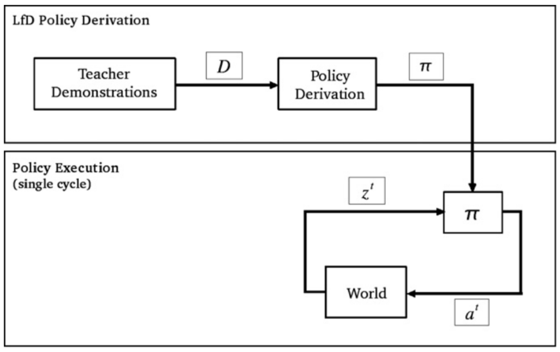

Figure 3.5: System Pipeline of the Imitation Learning Problem, from [1]

In the implementation of the DP3 paper [55], the agent learns to perform actions in a manipulation task in an end-to-end manner from expert demonstrations, which were mainly collected through human teleoperation or reinforcement learning agents. The network used for inference contains a perception module, where 3D single-view point clouds are processed into compact features via a lightweight MLP encoder, as well as a decision module, which uses a diffusion-based policy conditioned on the extracted features and the robot's proprioceptive observations, to denoise the latent Gaussian noise into an action sequence.

Reinforcement Learning Approach

In this approach, similar to imitation learning, the task is framed as an MDP, where the policy maps observations to actions at each time step [23]. The agent learns the optimal policy by interacting with the environment, receiving positive rewards for successful grasps and penalties for failures. Observations typically include sensory inputs from cameras or LiDARs and the robot's proprioceptive data. This method is well-suited for end-to-end learning, enabling the trained agent to handle novel, unseen task scenarios, provided overfitting to the training environment is prevented [18]. However, the approach also faces challenges: training can be computationally expensive and time-intensive, and sparse rewards often lead to slow and unstable learning [23]. Training is frequently conducted in simulation to speed up the training process and minimize risks associated with using a real robot. This can lead to a performance drop when transferring the trained model to real-world scenarios, necessitating techniques like *domain randomization* and *domain adaptation* to bridge the gap between simulation and reality [21]. For more information on the theoretical background of RL algorithms, please see section 2.2.

4 Problem Formulation and Implementation

4.1 Problem Formulation

Based on the implementation of the paper "*Learning to Grasp on the Moon from 3D Octree Observations with Deep Reinforcement Learning*" by Andrej Orsula et al. [43], the task involves the robotic grasping of unknown objects. The robot must handle objects with varying positions, sizes, shapes, textures, and quantities using visual RGB-D input as observations. The problem is modeled as an MDP, where the agent operates in an episodic environment of 100 time steps and derives actions from its current state at each step. The objective is for the agent to learn an end-to-end policy to perform high-level Cartesian actions in 3D space, enabling it to grasp objects and lift them above a specified threshold. This implementation addresses the challenge of developing effective grasping strategies in dynamic and uncertain environments.

4.2 Training Environment

The training environment is a lunar landscape simulated in Ignition Gazebo, following the implementation from the referenced base paper. The robot used is the Summit XL-GEN mobile manipulator from Robotnik, featuring a 7-DOF Kinova Gen2 robotic arm with a three-finger mechanical gripper attached. A simulated RGB-D camera with a resolution of 128x128 pixels and a frame rate of 15 Hz is statically mounted on the front of the robot. The camera's pose is randomized at the start of each episode. The working space for the robot is a 35x35x60 cm cube directly in front of it, while the observation space, defined by the bounds of the octree or the point cloud, is a 40x40x40 cm cube. Communication between the various modules is facilitated using ROS2 as middleware. The simulation operates with a low-level physics time step of 5 ms, while the high-level MDP steps occur at intervals of 400 milliseconds. While the high-level actions are handled by the learned policy, the low-level actions, in this case the joint positions, are generated using the MoveIt2 Motion Planning Framework [5], which employs the RRT-Connect algorithm [24] for efficient Cartesian path planning. Inverse kinematics for the robot arm are computed with the TRAC-IK solver [2], and precise control of the joint positions is maintained via a PID controller, allowing for refined adjustments and stable motion throughout the robot's movements. A high level of domain randomization was implemented to enhance performance during sim-to-real transfer. During each new episode, various environment and robot parameters are randomized, including the pose of the mobile manipulator in the environment, the initial joint configuration of its robotic arm, and the pose of the RGB-D camera mounted on the robot. Additionally, Gaussian noise is added to both the RGB and depth images to enhance their realism. Other randomized factors include the pose of the simulated sun, the ground's shape and surface texture, and the characteristics of objects to grasp, such as their amount, pose, shape, density, and surface texture. However, while domain randomization can improve the robustness of the model when transferred to the real world, it also leads to greater training instability and lower overall performance in simulation. This performance loss in simulation is due to increased variability, which makes it more difficult for the model to effectively learn the task, as observed in the base paper [43].

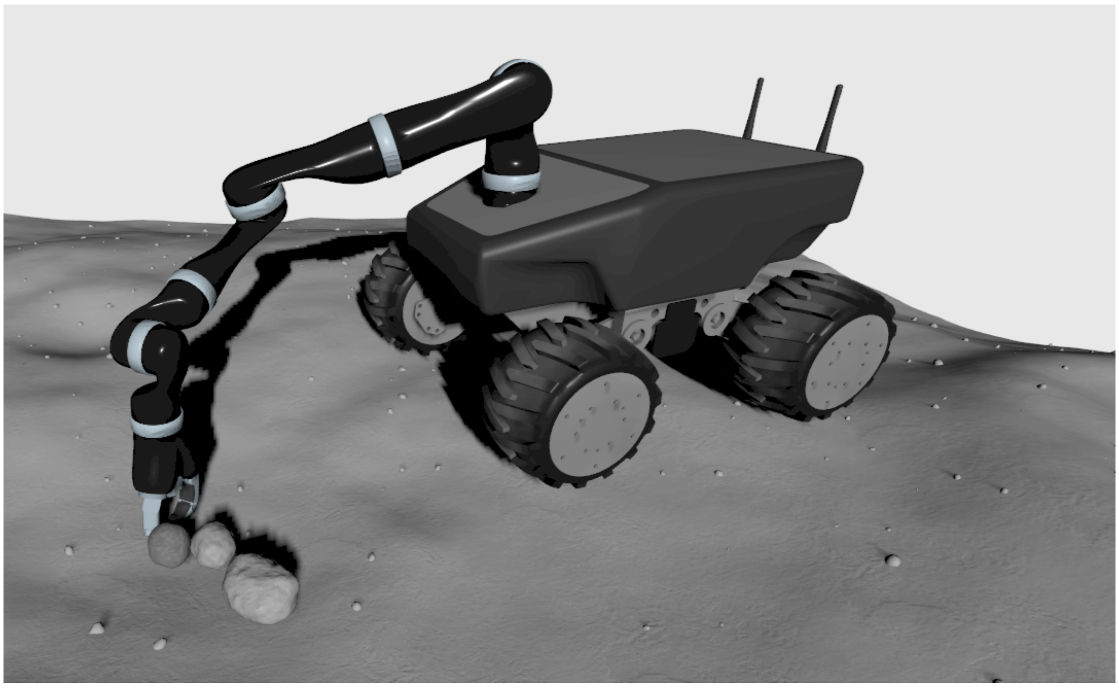

Figure 4.1: *Mobile Robot in Simulated Environment*: caption from the gazebo visualization of the Summit XL-GEN mobile manipulator inside the simulated randomized lunar environment, with three rocks to grasp in the workspace.

4.3 System Architecture

4.3.1 Main Pipeline

The observations of the current state get inputted into the feature extractor, which transforms the data into lower dimensional features so that the RL-algorithm can better process them. The algorithm's learned policy will then output a continuous action $a_t \in$ action space A. The environment will then perform a step, giving out the new state observations, as well as the step reward, which depends on the implemented curriculum function.

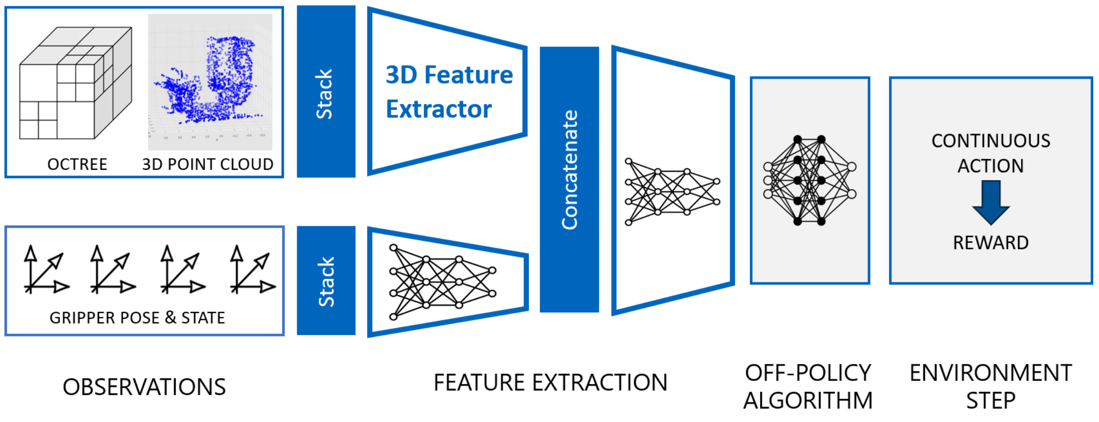

Figure 4.2: The visual representation of the main system pipeline passed through during each step.

4.3.2 Observation Space

The observation space consists of both visual and proprioceptive data. Visual observations are captured using an RGB-D camera and can take the form of the direct RGB-D input, point clouds, or octrees, depending on the used representation form. Point clouds can either be obtained by projecting RGB-D images or by directly retrieving them from a simulated sensor in Gazebo. These point clouds include each point's (x, y, z) coordinates relative to the robot base and may also include normalized (r, g, b) color values. For octree-based observations, normal vectors for each point in the point cloud are first computed, and the resulting data is then converted into an octree representation (see Fig. 4.3). Proprioceptive observations are represented as a 10-dimensional vector that captures the different states of the robot's gripper. This includes the (x, y, z) position of the tool center point (TCP) relative to the robot's base, the gripper's orientation as a continuous 6D rotation $(R_1, R_2, R_3, R_4, R_5, R_6)$, and the gripper state $g_s \in [-1, 1]$, where

$g_s = 1$ indicates a fully opened gripper and $g_s = -1$ represents a fully closed gripper. To capture temporal dynamics and motion, the observation includes frame stacking, comprising data from the current time step t and the previous time step $t-1$.

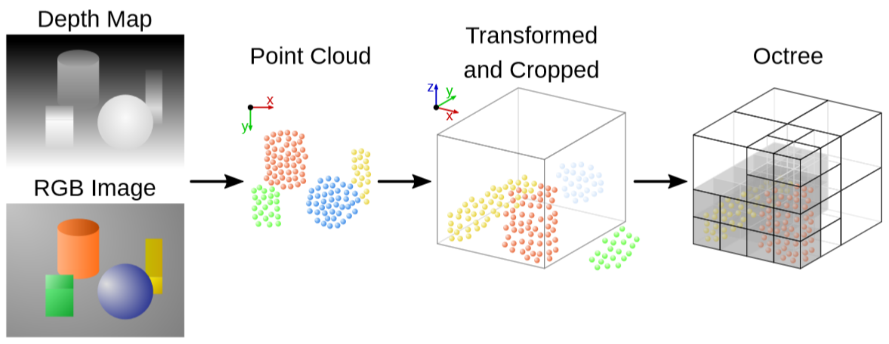

Figure 4.3: *Visual representation of the conversion of the RGB-D input into an Octree* [42]: first, a point cloud is gained from the RGB-D image. Then it is transformed and cropped, and its normal vectors are computed. Finally, an octree is constructed from this processed point cloud.

4.3.3 Feature Extractor

The feature extractor processes observations by mapping them into high-level, lower-dimensional features that RL algorithms can more effectively utilize. It typically includes learnable parameters and may incorporate a pre-trained section, usually in the initial layers that focus on extracting low-level features. The network is divided into two sections: one for processing visual observations and another for proprioceptive observations. The features extracted from both sections are concatenated and passed through a final MLP layer before being fed into the RL algorithm. Various approaches are implemented for the visual observation section, utilizing backbones such as O-CNN for octree-based observations and PointNet or DP3 for point-cloud-based observations.

4.3.4 Algorithm

The current implementation supports the TD3, SAC, and TQC algorithms from the Stable-Baselines3 framework [49]. These algorithms are model-free, off-policy, actor-critic methods. They train a policy $\pi(a \mid s, \theta)$, where θ represents the weights of the actor network, and estimate either the expected return or the return distribution to compute the Q-function $Q(s, a)$. During inference, the algorithms stochastically map the input features to actions using the learned policy.

4.3.5 Action Space

A high-level action $a_t \in A$ is executed in each time step of the MDP, occurring at a frequency of 2.5 Hz. The action space consists of continuous Cartesian 3D motions bounded to the range $[-1, 1]$ to simplify training and improve stability. These actions are then rescaled to their corresponding metric and angular ranges for execution. The ranges include the relative translations $\Delta x, \Delta y, \Delta z$ of the TCP within $[-0.1\text{m}, 0.1\text{m}]$ and a yaw rotation $\Delta\phi$ around the z-axis within $[-45°, 45°]$. Additionally, gripper control is represented by $g_a \in [-1, 1]$, where positive values indicate opening, and negative values indicate closing. At the simulation step scale, the lower-level actions are performed using the TRAC-IK framework from Beeson et al. [2], so they do not need to be learned by the algorithm.

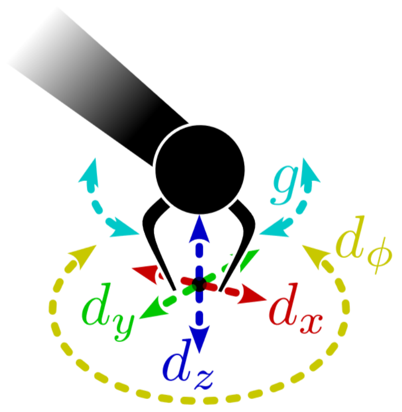

Figure 4.4: Depiction of the action space A, where (d_x, d_y, d_z) represent the translation direction, d_ϕ represents the yaw rotation, and g the gripper state (from [43])

4.3.6 Curriculum

The learning curriculum is based on a composite reward function that is supposed to guide the agent through four sequential stages: "Reach", "Touch", "Grasp", and "Lift". These four stages are analogous to the first four stages of the robotic grasping task described in the works of Fantoni et al [8]. At every time step t, the agent potentially earns a reward upon reaching one of the stages, as well as a negative persistent time penalty to discourage longer episodes without successful termination. For each stage $s \in [1, 4]$, the reward for reaching this given stage at the current time step is defined as $r(s) = r_0 \cdot r_b^{s-1}$, where r_0 is the base stage reward value, and r_b is the stage reward multiplier. The total episode reward is the sum of all time-step rewards within an episode, which depends on the highest stage reached and the

number of time steps before termination. In the current implementation, r_0 is set to 1, r_b is 8, and the persistent time reward is -0.1 per step. For example, if the agent reaches stage 3 and the episode lasts 100 steps, the total episode reward is calculated as $R_{eps} = 100 \cdot (-0.1) + \sum_{s=1}^{3} 8^{s-1} = -10 + 1 + 8 + 64 = 63$. This reward structure balances incentivizing progress through the stages with penalizing inefficient, prolonged episodes. There are various modifications to the curriculum that can be introduced to adapt the training process, each of which can be selectively activated or deactivated based on the desired implementation strategy:
Growing Persistent Reward Over Time: in the standard case, the persistent reward value remains constant throughout the entire training process. However, one could imagine increasing this value over time, for example, by letting it grow linearly or by doubling it every k time steps. This approach addresses a common issue where, during later stages of training, the agent frequently reaches lower stages and accumulates mostly positive episode rewards, which can lead to reduced exploration, preventing the agent from consistently reaching higher stages. By gradually increasing the persistent reward during the training process, episodes that only reach lower stages would become less rewarding or even penalized. In contrast, episodes that terminate early due to success would be more positively reinforced. This adjustment is expected to promote sustained growth in the agent's success rate during the later phases of training. To ensure consistent comparability during the entire training, the initial persistent value is applied across all evaluation sessions.
Single-Object Focus: in the standard implementation, a stage is considered attained if any object in the workspace reaches the specified stage during a given time step. However, if the single-object focus mechanism is implemented, progression to subsequent stages is restricted to the specific task object, with which a certain initial stage (e.g., "Reach" or "Touch") is attained. This encourages the agent to maintain its focus on a single target object rather than randomly achieving the consecutive reward stages with different objects in the workspace. The expectation is that while this approach may initially slow down learning due to the stricter progression criteria, it will improve the agent's long-term performance by fostering a more consistent and deliberate learning strategy.
Dynamic Lifting Height Threshold: in the standard implementation from the base paper, the required height for reaching the "Lift" stage can vary over time and grow with an increasing success rate. Initially, the required height lies at 7.5 cm and linearly grows up to 15 cm when reaching a success rate of 33%. This is meant to facilitate reaching the "Lift" stage at the beginning of the training process when the agent's policy does not yet perform well. The current approach introduces two major challenges. First, when the agent successfully lifts an object to the initial height, it tends to learn a suboptimal behavior by consistently lifting it to a height insufficient for later stages. As the success rate grows and the height requirement increases, the agent fails to progress, causing

the success rate to oscillate, dropping when the threshold becomes too high and only recovering when the requirement decreases again. Second, this dynamic adjustment complicates performance tracking, as the criteria for reaching the final stage do not remain constant, making it difficult to evaluate the agent's true capabilities over time. To fix this issue, one could implement a different reward function for the "Lift" stage, where the threshold stays constant at the maximum height of 15 cm and the reward is given progressively at different height increments.
Incremental Lift Stage Reward Function: this alternative approach for the reward function for the "Lift" stage is designed to encourage the agent to progressively lift an object to higher positions. When the agent successfully lifts the object to a certain first threshold (in our case, 40% of the maximum required height), it begins receiving a portion of the total "Lift" stage reward. The reward granted is proportional to the ratio of the first threshold to the maximum lift height (40% of the total "Lift" stage reward). Subsequently, the agent earns 10% of the total reward for each incremental lift achieved until reaching the maximum required height. Importantly, each increment can only be reached once, ensuring the total reward for the stage cannot exceed the maximum "Lift" reward. This gradual reward system aims to improve the agent's performance by encouraging progressively higher lifts, which could lead to better success rates when training the agent with this training curriculum. However, there are two important considerations. First, this method could distort the mean reward since portions of the "Lift" reward are given before the episode is successfully completed. Second, removing the dynamic lift height requirement makes the task more difficult to complete, especially when the success rate remains low, as the agent must always reach a minimum height of 15 cm, even early in training. This could lower success rates, particularly in the early stages of training, and might counterbalance the increased mean reward achieved through the gradual "Lift" reward. Therefore, achieving a comparable success rate under this fixed-height approach would signify improved performance, as the task is clearly more challenging.

4.4 Current Baseline Implementation

The baseline implementation uses octrees for the agent's visual observations, incorporating features like RGB color values and normal vectors. The feature extractor is built upon an O-CNN backbone, as shown in Figure 4.5. The feature extractors' hyperparameters can be seen in Table 4.1. The RL algorithm used is TQC, as it demonstrated the best performance among the three implemented off-policy algorithms in the trials of the base paper [42, 43]. The used curriculum implements a standard composite reward function with a constant persistent time penalty. Instead of using an incremental reward approach for the "Lift" stage, a dynamic lift height threshold is applied to determine the reward. Additionally, no object-specific focus is implemented, allowing the sequential stages to be achieved with any task objects.

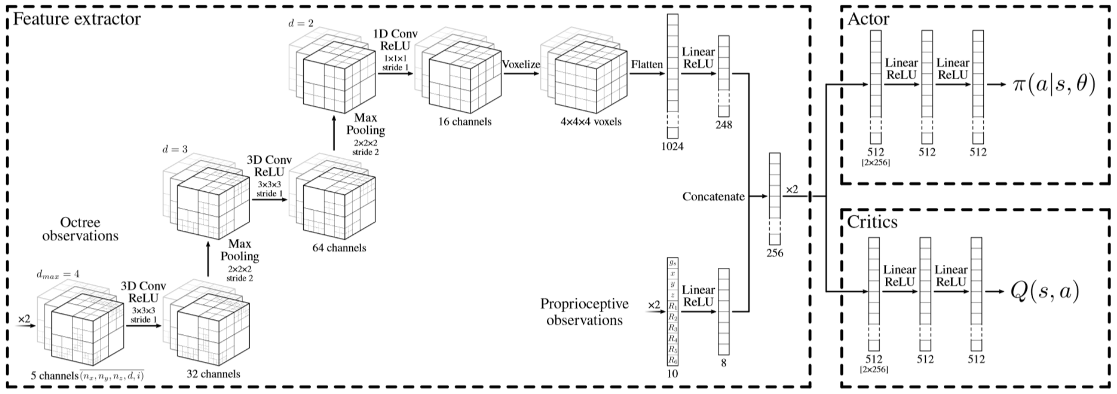

Figure 4.5: *Visual representation of the octree-based feature extractor based on the O-CNN network*: the stacked octree observations, as well as the proprioceptive observations, both enter their respective feature extractor section before having their features concatenated and being inputted into the actor and critic networks (from [43]).

Table 4.1: Hyperparameters of the octree feature extractor used in the baseline case. The "Full Depth" hyperparameter defines the depth at which the octree is converted into a voxel grid and flattened to a 1D vector.

Hyperparameter	Value
Backbone	O-CNN
Depth of Input Octree	4
Full Depth	2
Input Channels	7
Number of Channels at each Depth	[32, 64, 16]
Dimension of Visual Features	248
Dimension of Auxiliary Observations	10
Dimension of Auxiliary Features	8
Batch Normalization	False

5 Experiments and Results

5.1 Experiment Setup

5.1.1 Training Procedure

The training procedure involves running the training process three times with different random seeds and averaging the results to ensure robustness to outliers. The evaluation compares different approaches for curriculum learning and feature extraction while keeping the TQC algorithm consistent across all cases, as its superior performance over SAC and TD3 was already demonstrated in the works of Andrej Orsula's base implementation [43, 42]. Each training session runs for 500k time steps, with the current policy evaluated every 25k time steps over 50 episodes. In prior trial runs, a high variance between the evaluation sessions was observed. This is why a decision was made to reduce the evaluation frequency and increase the number of evaluation episodes in each evaluation session (keeping the proportion of evaluation steps the same) to provide more meaningful results and address the high variance during evaluation. Progress during training is tracked by plotting the mean episode reward and the success rate over time. Additionally, the best-performing model out of all 3 training runs is evaluated for 200 episodes under a new random seed, utilizing the same curriculum across all trials with an incremental "Lift" reward function.

5.1.2 Hyperparameters

To maintain a good comparability of the results, the main algorithm parameters were kept constant over all experiments, using the same ones as in the implementation of the base paper [43].

Table 5.1: Hyperparameters of the experiment used to train all agents.

Hyperparameter	Value
Algorithm	TQC
Optimizer Class	Adam
Learning Rate Schedule	Linear, $2.0 \cdot 10^{-4} \rightarrow 0$
Batch size	64
Learning Starts	20000
Replay Buffer Size	50000
Train Frequency	[1, "episode"]
Gradient Steps per Update	100
Discount Factor γ	0.99
Target Update Rate τ	$4 \cdot 10^{-5}$
Entropy Coefficient α	Automatic
Entropy Target	$-dim(A) = -5$
Number of Critics	2
Number of Quantiles	25
Number of Truncated Quantiles	3
Net Arch	[512, 512]
Exploration Noise	$\mathcal{N}(0, 0.025)$

5.2 Reproduction of Baseline Experiment

The first step involves reproducing the experiment from the base paper under theoretically identical conditions. In the base paper, the performance of three averaged runs in simulation achieved up to a 33% success rate, as shown in Figure 5.1. However, the success rate of three averaged runs reached only 8% in the reproduced experiment, indicating a significant discrepancy between the reproduced results and the original findings. The success rate over training time is visualized in Figure 5.3, while the mean episode reward over time is shown in Figure 5.2. The model that could demonstrate the best performance across all 3 runs was evaluated again for 200 episodes with a different random seed. The implemented curriculum used an incremental "Lift" reward function, where the threshold was 15cm (as this was the required height defined by the task). Under these conditions, it had a mean reward of 51.30 and could only show a success rate of 5.00 %. The goal in the following sections will be to outperform the reproduced experiment by introducing changes to the curriculum and the feature extractor approach to try to approach the performance shown in the base paper.

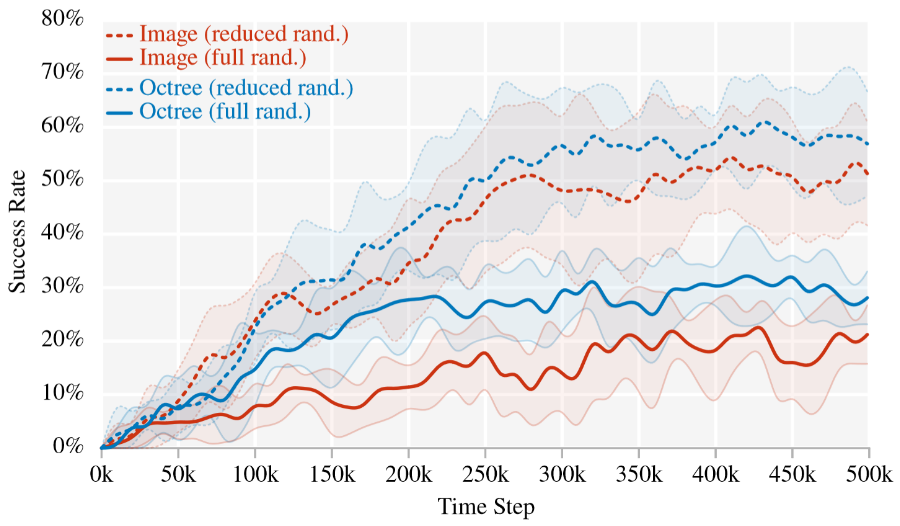

Figure 5.1: *Evaluation Performance over Training in Base Paper*: the "Octree (full rand.)" case is of interest, as it supposedly has the same conditions as our reproduced implementation (from [43]).

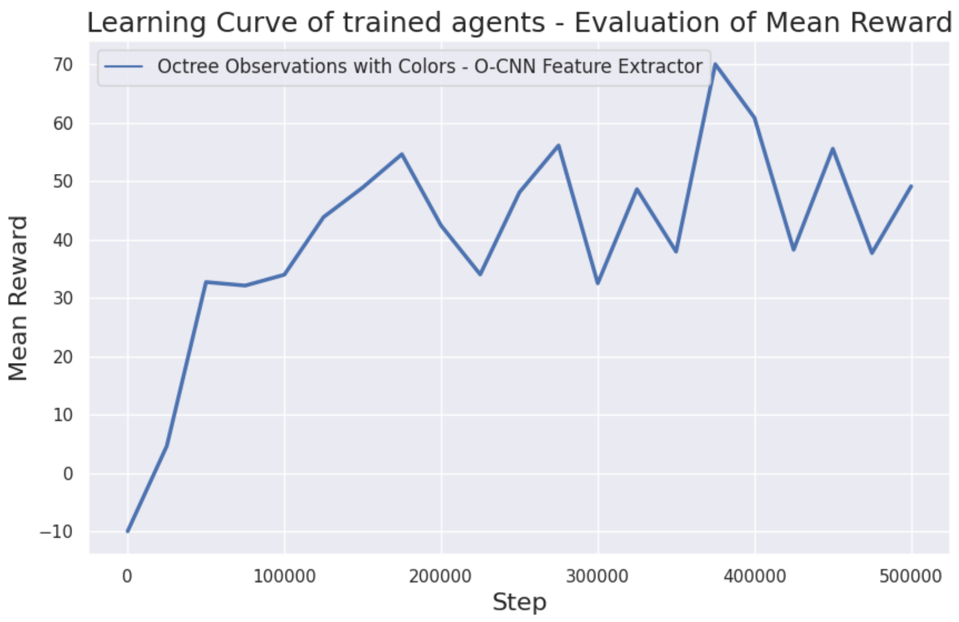

Figure 5.2: The plot of the mean reward over training of the 3 averaged runs from the reproduced experiment.

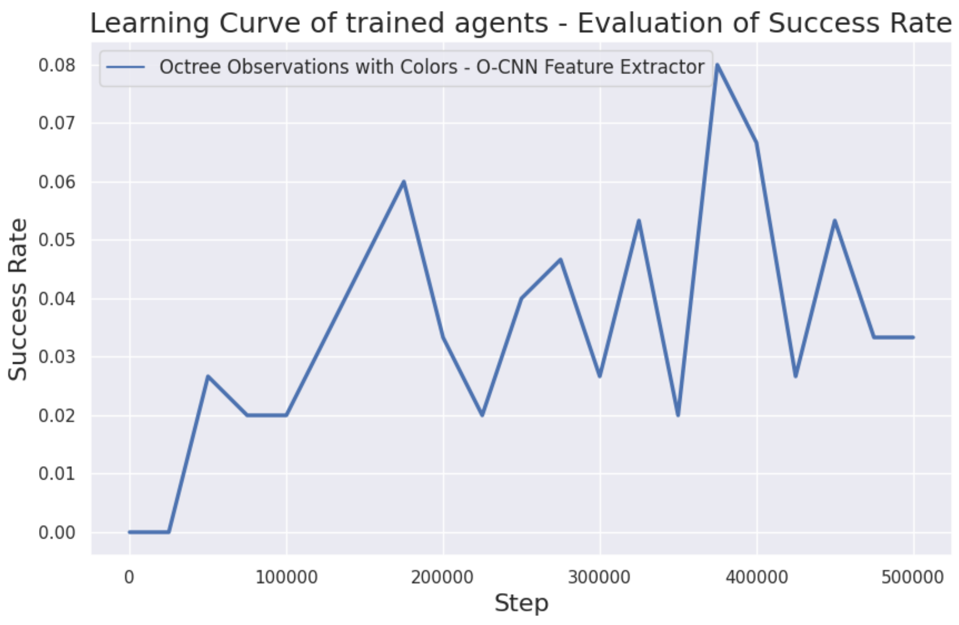

Figure 5.3: The plot of the success rate over training of the 3 averaged runs from the reproduced experiment.

5.3 Experiments on Curriculum Approaches

The experiments conducted in the following sections all use an O-CNN-based feature extractor derived from the baseline implementation and combined with the TQC algorithm to ensure comparability. The performance will be evaluated across three key aspects of the curriculum: the approach used for the "Lift" stage reward function, the method for applying a persistent reward penalty, and the strategy for focusing on a single object in the task.

5.3.1 Approach for "Lift" Stage Reward Function

The first experiment compares the performance of the dynamic "Lift" stage reward function with that of the incremental one, both of which include a constant, persistent reward penalty to maintain comparability. When comparing the success rate plots (see Fig. 5.5), it is important to consider the points made in section 4.3.6. Specifically, a success rate achieved with an incremental "Lift" stage reward function can be regarded as greater performance than that of a comparable rate achieved with the dynamic reward function, especially when the success rate did not come close to 33%. It is indeed easier to reach an episode success if the threshold is only half of the actual target height. Therefore, even if the success rate for the dynamic approach appears somewhat higher in the plot, it does not hold up when evaluating under the actual condition, where the threshold matches the target height. Indeed, when comparing the performance of the best-observed model for 200 episodes, the model using an incremental "Lift" stage reward reached a mean episode reward of 117.25, with a success rate of 12.50%. The baseline model, which used a dynamic "Lift" stage reward, performed significantly worse, with a mean reward of 51.30 and a success rate of 5.00%. This demonstrates that the incremental approach is preferable for later stages, especially since it appeared to be better compatible with the subsequent change in the curriculum - the growing persistent reward penalty. The learning curve of the mean episode reward (see Fig. 5.4) is not discussed in detail, as the reward is significantly altered by the change in the "Lift" stage reward function.

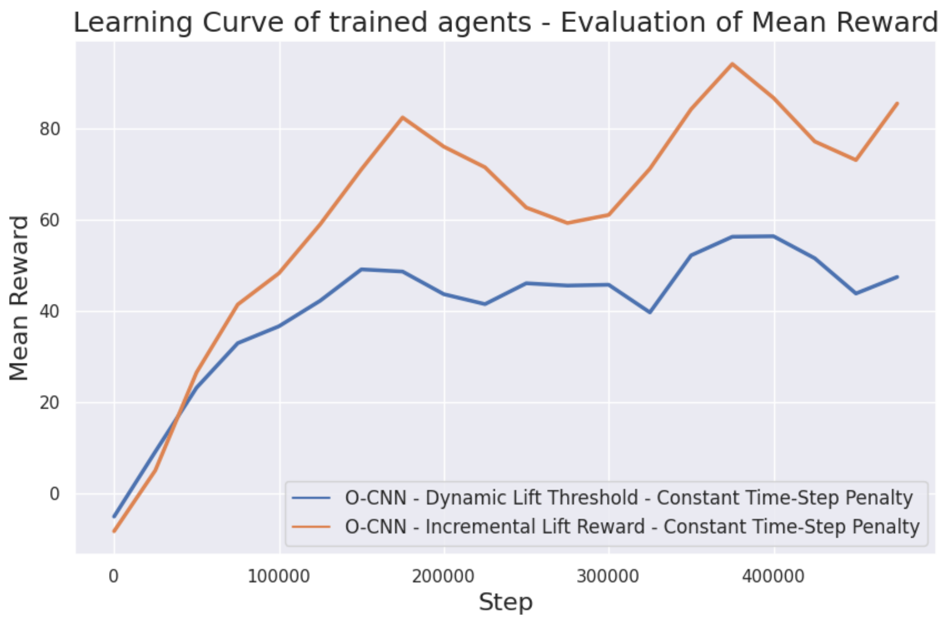

Figure 5.4: The plot of the mean reward for both "Lift" stage reward approaches is smoothed using a moving average with a window size of 3. Although the curve of the incremental approach is significantly higher than that of the dynamical approach, one cannot derive substantial, meaningful information from it, since it already rewards reaching intermediate height thresholds.

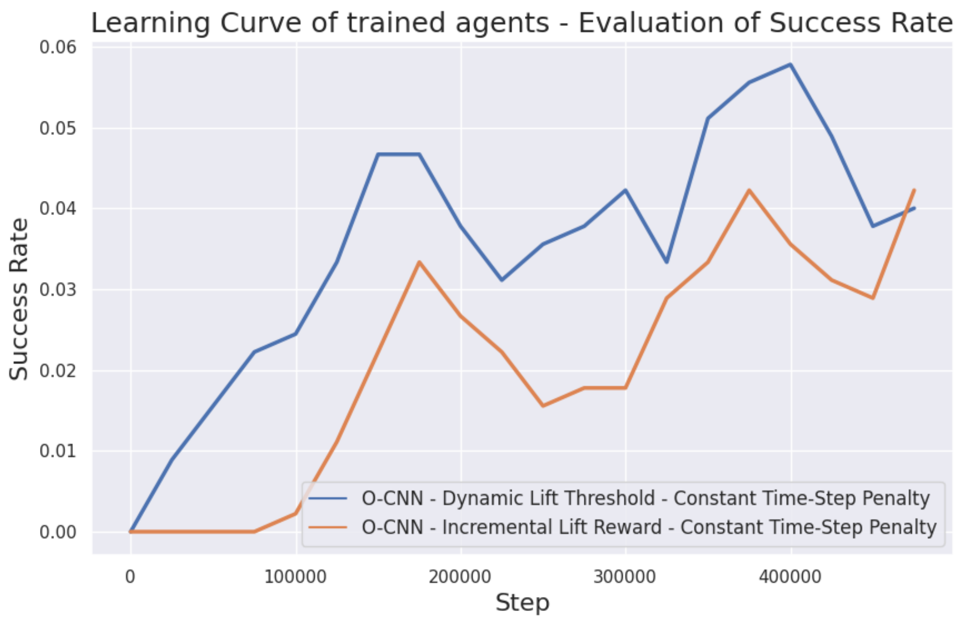

Figure 5.5: The plot of the success rate for both "Lift" stage reward approaches is smoothed using a moving average with a window size of 3. The curve of the dynamic approach mostly surpasses that of the incremental approach. Since neither approach reaches a rate close to 33%, the threshold for the dynamic approach is only approximately half that of the incremental approach, which prevented it from properly learning to lift the object to the actual target height.

5.3.2 Approach for Persistent Reward Penalty

In this section, three different approaches for the persistent reward penalty are implemented, all of which use an incremental 'Lift' reward function and an octree-based feature extractor. In the first baseline case, a constant penalty of -0.1 is applied at every time step throughout the entire training process. In the second case, the penalty grows linearly, starting at -0.1 per step and increasing by an additional -0.1 every 125k time steps. In the final case, the penalty also begins at -0.1 and doubles every 125k time steps. Several observations can be made when examining the learning curves of the three approaches (see Figures 5.6 and 5.7). Models using a growing persistent reward penalty outperform the constant penalty case in both mean episode reward and success rate, with a more significant difference in the success rate. For the exponential growth case, the model achieves its best performance in the first 250k time steps, after which its performance decreases. This decline could stem from the rapid penalty increase at 250k time steps to -0.4 and at 375k time steps to -0.8, which may be too drastic. In contrast, the linear growth model does not exhibit this decline and continues to improve throughout the entire training process. Another observation is that, despite having identical penalties for the initial 250k time steps (-0.1 until 125k time steps, then -0.2 until 250k time steps), the exponential growth models perform significantly better during this time frame than their linear counterparts. Given that all other training conditions are identical, this difference is likely attributable to variance, especially considering that the average variances for both the mean episode reward and the success rate are relatively high across all three configurations (see Table 5.3). This means that each training configuration would have to be run considerably more often than three times to confidently determine which approach works best, although it seems likely that increasing the penalty over time contributes to a higher success rate. Additionally, it can be inferred that a comparatively lower variance indicates the absence of (positive) outlier runs, which results in slightly more meaningful data that can be more effectively utilized in the following sections as a comparative result. Indeed, the linear case shows no outlier runs, with the best model clearly under-performing compared to the two other best models (see Table 5.2) and the overall variance being significantly lower as well (see Table 5.3). This is why the linear case will be used as the comparative baseline in the following sections.

Table 5.2: Comparison of the performance of the best model out of all three persistent reward approaches: constant penalty, linearly growing penalty, and exponentially growing penalty. The "Lift" reward function is incremental in all three approaches, and the feature extractor is O-CNN. One can observe that the model from the exponential approach performs the best, exhibiting a success rate of 16%, closely followed by the model from the constant approach. The model from the linear case performed the worst, with a success rate of only 5%.

Approach	Mean Episode Reward	Success Rate
Constant Time-Step Penalty	117.25	12.50%
Linearly Growing Time-Step Penalty	75.88	5.00%
Exponentially Growing Time-Step Penalty	129.71	16.00%

Table 5.3: The average variance of the mean rewards and success rates of evaluation sequences across three training runs in the different persistent reward cases. Across all three cases, the variances are very high, suggesting a low confidence in the results from the learning curve. The linear growth approach has the lowest variance of all three methods, with a ratio of ≈ 1.5 compared to the constant approach and ≈ 3 compared to the exponential growth approach.

Approach	Mean Reward Variance	Success Rate Variance
Constant Time-Step Penalty	946.63	15.3
Lin. Growing Time-Step Penalty	639.62	8.60
Exp. Growing Time-Step Penalty	1657.03	32.7
Ratio Constant to Linear	1.48	1.78
Ratio Exponential to Linear	2.59	3.80

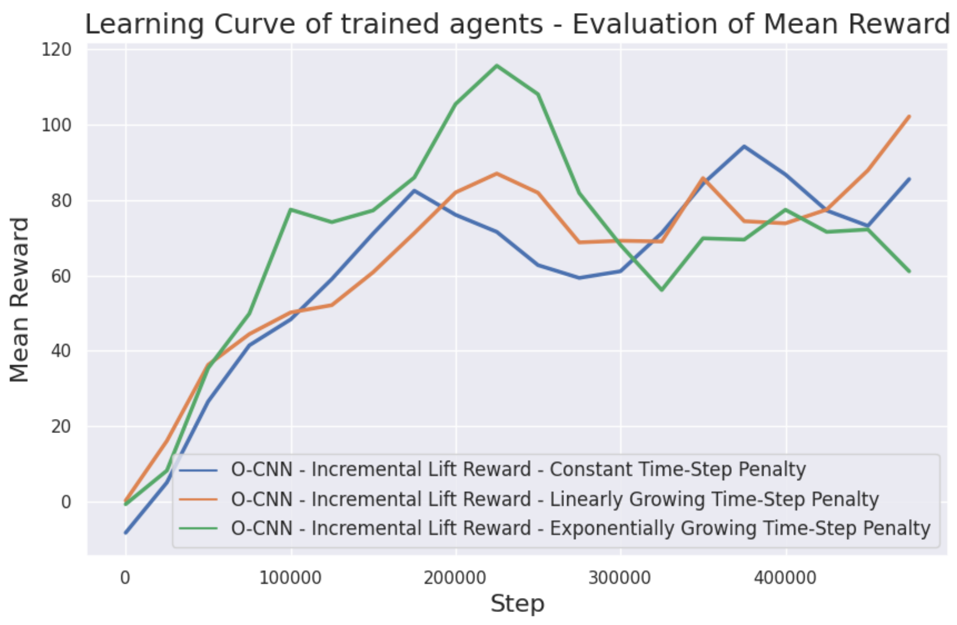

Figure 5.6: The plot of the mean reward of the different persistent time-penalty approaches smoothed is using a moving average with a window size of 3. The three approaches do not appear to vary significantly from one another, except for the time frame between steps 200k and 250k, during which the exponential approach seems to considerably outperform the other two models.

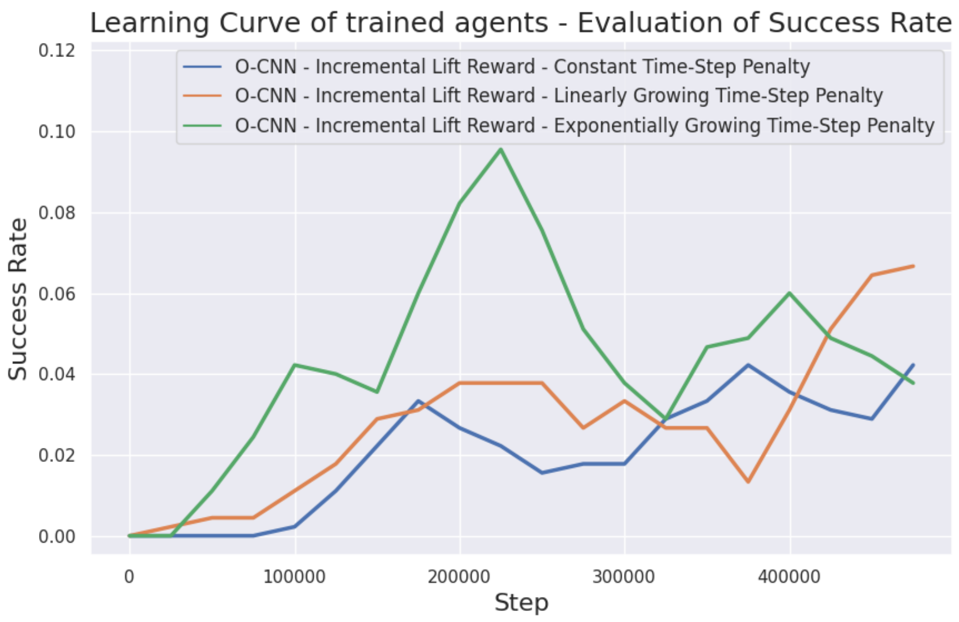

Figure 5.7: The plot of the success rate of the different persistent time-penalty approaches is smoothed using a moving average with a window size of 3. While both growing approaches consistently outperform the constant case, they do not reach their peak performance at the same time. The model from the exponential approach peaks at 225k time steps, with a success rate of nearly 10%, whereas the model from the linear approach reaches its peak near the final steps, at almost 7%.

5.3.3 Approach for Single-Object Focus

The last improvement axis in the curriculum concerned the implementation of a single-object focus after a given curriculum stage. As explained in Chapter 4, once a predetermined stage is reached with a certain object, the subsequent stages can only be reached with the same object. In the following experiment, we compared the performance of implementing a single-object focus after reaching the "Touch" stage with the performance of refraining from using any single-object focus, as done in the baseline case. In both approaches, the models were trained in otherwise identical conditions, using an incremental "Lift" stage reward function and a linearly growing persistent reward penalty. When examining the average learning curve of the two approaches, it is clear that the approach without single-object focus strongly outperforms the other in both mean episode reward (see 5.8) and success rate (see 5.9). The same cannot be said when comparing the evaluation performance of the best-observed model (the single-object focus was removed during this evaluation process to keep certain comparability): the best model using the single-object focus achieved a success rate of 8.00%, compared to 5.00% without it, and the mean episode reward was higher as well (91.93 versus 75.88). While this shows that the single-object focus approach might perform slightly better than initially expected, once the object focus is removed during evaluation, it still does not exhibit a strong performance, especially considering that the 5.00% success rate from the compared runs could indicate a slight underperformance (see previous section). Considering all results from comparing these two approaches, it seemed more sensible to refrain from using any single-object focus, as it might not align well with the given training instability and the relative rarity of successful episodes during the sampling of actions.

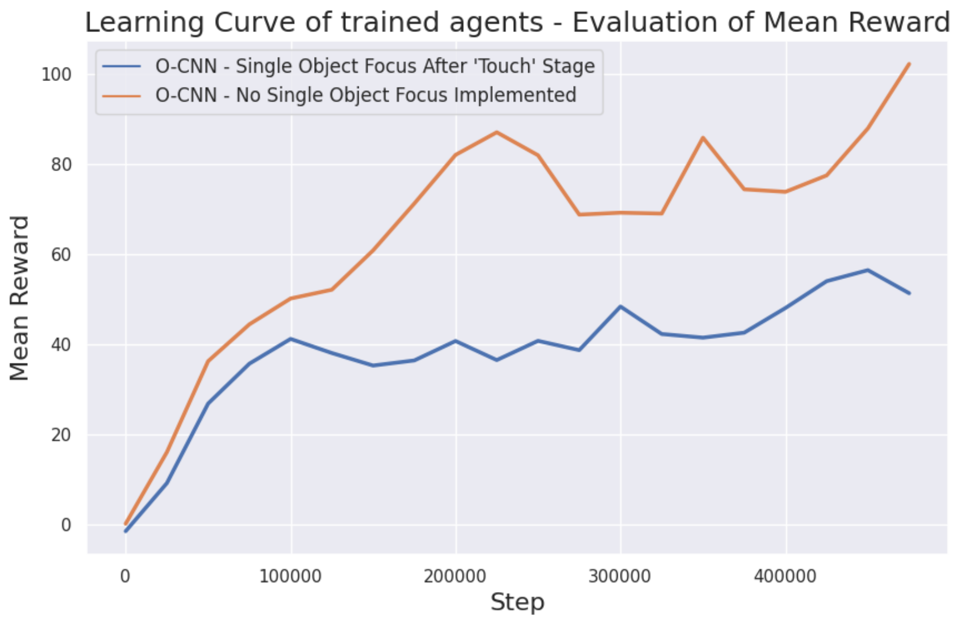

Figure 5.8: The plot of the mean reward of the different object-focus approaches smoothed is using a moving average with a window size of 3. From 150k time steps onward, the approach without any single-object focus consistently outperforms the one using a focus after the "Touch" stage by at least 20 points.

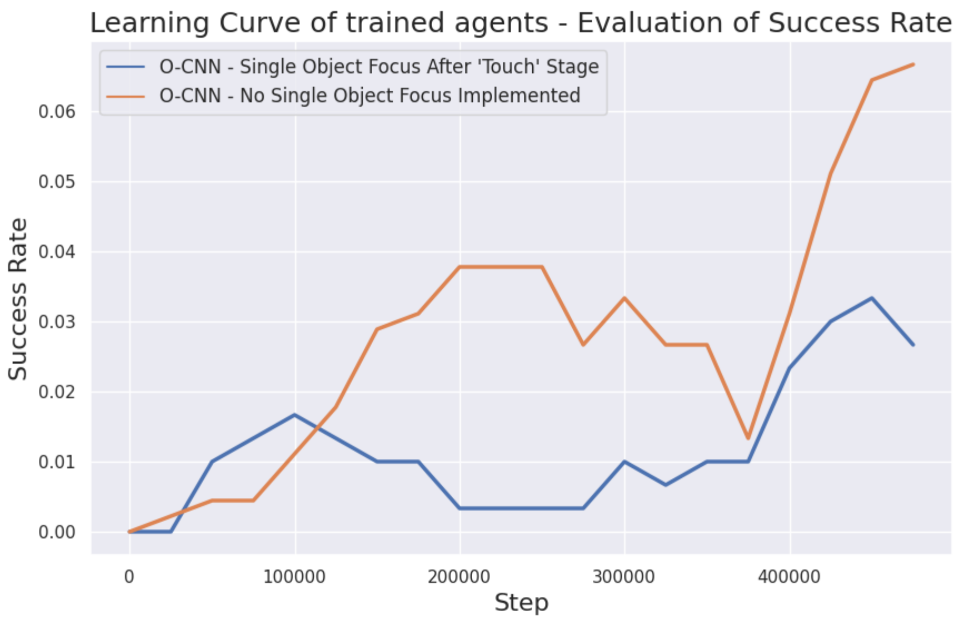

Figure 5.9: The plot of the success rate of the different object-focus approaches is smoothed using a moving average with a window size of 3. After the 125k time step, the approach without any single-object focus consistently achieves a higher success rate than the one using a focus after the "Touch" stage, with the highest achieved success rate being at least 3.00 % higher.

5.4 Experiments for PoinNet-Based Feature Extractor Implementation

5.4.1 PointNet-Based Model Selection

The newly implemented feature extractors were based on PointNet and PointNet++ backbones, and their forward pass durations were compared for both a single observation (batch size of 2) and an entire batch of 64 observations (batch size of 128). As shown in Table 5.4, PointNet-based feature extractors demonstrated slightly faster forward passes for single observations compared to the O-CNN-based extractor, making them suitable for real-time applications. However, their forward pass for an entire batch was 2-3 times slower, resulting in somewhat slower training. In contrast, PointNet++-based feature extractors had a significantly slower forward pass, taking approximately 45 times longer for a single observation, with a duration of $\approx$ 135 ms, which is in the same order of magnitude as the environment step duration of 400 ms, rendering them less suitable for real-time applications. Furthermore, the batch forward pass was around 53 times slower than the baseline, causing the training to be substantially slower and leading to CUDA memory issues. Although PointNet++ is known to perform better in 3D classification and segmentation tasks, the limitations encountered in this application made it unsuitable. As a result, the decision was made to exclusively use feature extractors based on the standard PointNet network for point-cloud observations.

Table 5.4: Duration (in ms) required to perform the forward pass on an NVIDIA RTX A5000 GPU with 24 GB of memory. The duration was measured over 100 averaged-out forward passes performed for a single observation (batch size of 2) or an entire batch (of size $2 \cdot 64$).

Feature Extractor	Forward Pass Duration for Single Observation	Forward Pass Duration for Entire Batch
O-CNN (baseline)	3.000	10.909
PointNet - using global features	2.413	22.419
PointNet - using point-wise features	2.806	33.283
PointNet++ - using global features	135.482	580.521

5.4.2 Feature Extractor Training Approach

The initial approach involves using the PointNet network's encoder component as our model's feature extractor and training the entire network, including the feature extractor, from scratch without freezing any weights. However, this training approach proved unsuitable, as the network was unable to meaningfully learn an optimal policy. Figure 5.10 shows the evolution of the mean reward during the training, and Figure 5.11 shows its success rate over time, each compared to the baseline model's performance. The results show a lack of significant improvement in the agent's performance beyond a certain training duration, causing it to fail to match the baseline model's performance. This inability to learn an optimal policy from the observations is likely due to the reward being too sparse to adequately train a complex encoder network like the one from PointNet. To address this issue, a pre-trained encoder with frozen weights will be used in the next stages, followed by 2-3 learnable layers at the end of the feature extractor pipeline. The encoder could get its weights from a network trained on a 3D object classification or semantic segmentation data-set, enabling the encoder to extract meaningful geometric features from the observation, hopefully facilitating learning and improving performance.

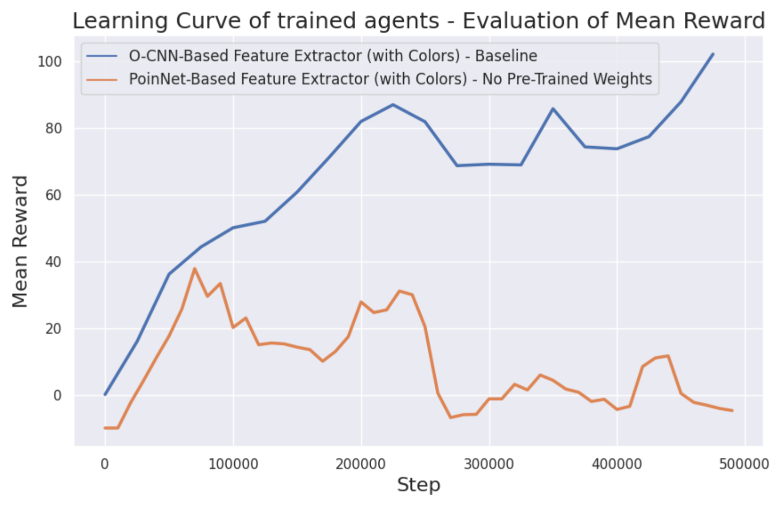

Figure 5.10: The plot of the mean reward over the training of 3 averaged runs using the plain PointNet encoder compared to the training progress of the baseline model. One can see that the average mean reward never reaches the value of 40 and starts to sharply decrease past 250k time steps.

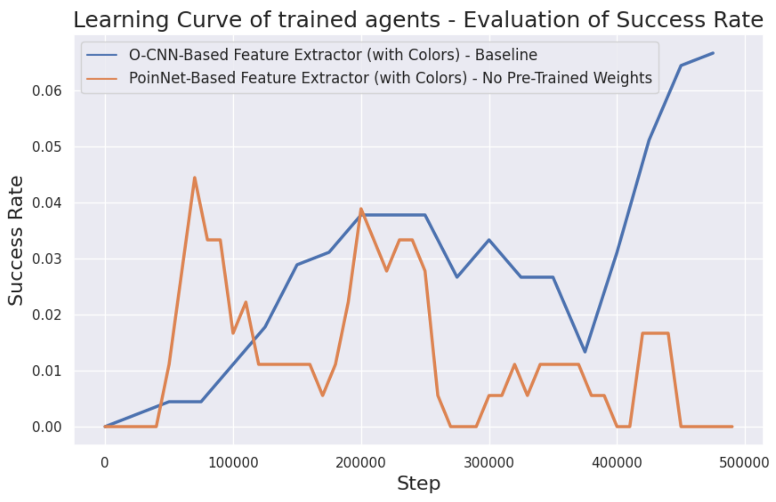

Figure 5.11: The plot of the success rate over the training of 3 averaged runs using the plain PointNet encoder compared to the training progress of the baseline model. The agent once reaches a success rate of around 4% in the early stages of the training process but is never able to learn to grasp the object at a consistent rate.

5.4.3 Utilized Observation Data

Potential Benefit of Using Normal Vectors

When using a PointNet classification model as the feature extractor base, one can only use the XYZ-coordinates or add the normal vectors to the observations. Both PointNet and PointNet++ were trained on the ModelNet40 data-set, where the point clouds include the normal vectors but no color channels. Table 5.5 lists the performance in both models' classification tasks once using these normal channels and once omitting them. In both cases, no significant performance benefit was observed from using the normal vectors, with the difference never exceeding 1.3%. This led to the decision not to use normal vectors in the observations, particularly because calculating them would add computational overhead to the current task.

Table 5.5: Performance comparison of trained PointNet and Pointnet++ models in the classification task with the ModelNet40 data-set. Each model was trained once with normal vectors and once without normal vectors in the observation. In both cases, the difference in performance in instance accuracy as well as class accuracy is minimal, and never exceeds 1.3 %.

Model	Instance Accuracy	Class Accuracy
PointNet - without Normals	0.904288	0.868665
PointNet - with Normals	0.916990	0.874906
PointNet++ - without Normals	0.926537	0.896751
PointNet++ - with Normals	0.928560	0.903051

Performance Using Different Numbers of Points

When using the PointNet semantic segmentation network as the feature extractor base, one can utilize the XYZ-coordinates and the RGB-color channels in the observations. Both Pointnet and Pointnet++ can be trained on the S3DIS data-set, and can handle varying input sizes in terms of the number of points. Table 5.6 shows the performance of both models trained on the data with 1024, 2048, and 4096 points in the observation. Since the networks trained with a higher number of points did not outperform the other models, it was decided to use the PointNet semantic segmentation model trained with 1024 points, which reduces the computational cost of a forward pass.

Table 5.6: Performance comparison of trained PointNet and PointNet++ models in the semantic segmentation task with the S3DIS data-set. Each model was trained once with 1024, 2048, and 4096 points in the observation. For both models, the difference in performance was negligible, never exceeding 2% in class mIoU.

Model	Overall Accuracy	Class mIoU
PointNet - 1024 points	0.793512	0.439452
PointNet - 2048 points	0.768758	0.426929
PointNet - 4096 points	0.786509	0.426158
PointNet++ - 1024 points	0.826655	0.520418
PointNet++ - 2048 points	0.834423	0.544301
PointNet++ - 4096 points	0.831059	0.546091

Benefit of Utilizing Color Channels

Two model approaches were compared for the PointNet-based feature extraction in this task. The first model utilized the classification network as the feature extractor backbone, using only XYZ coordinates as input observations. The second model employed the semantic segmentation network as the backbone, incorporating both XYZ coordinates and RGB color channels. In both models, the pre-trained backbone with frozen weights produced a single global feature vector of size 1024, which was subsequently processed by two additional learnable layers before being passed to the RL algorithm. While the model that included color features demonstrated better training performance than the one relying solely on spatial coordinates, both models struggled to learn a meaningful policy and failed to match the performance of the baseline model using octree-based observations (see Figures 5.12 and 5.13). For the next steps, the preferred approach is to include the color channels in the observations. However, it is essential to improve the architecture of the learnable component of the feature extractor, which is addressed in the next section.

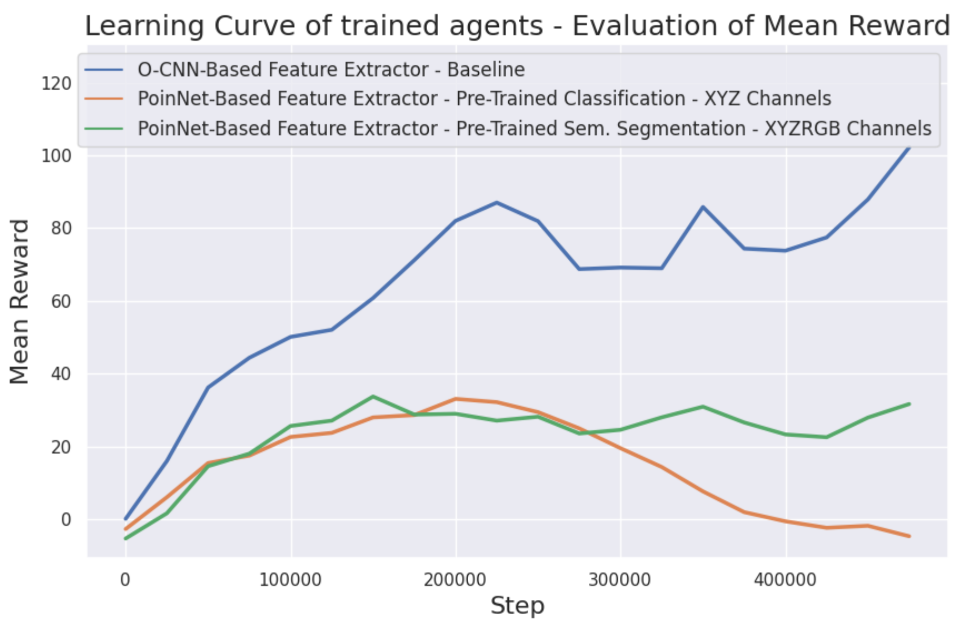

Figure 5.12: The plot of the mean reward of the different feature extractor approaches is smoothed using a moving average with a window size of 3. The different approaches include the PointNet classification network as backbone (with XYZ channels), the PointNet semantic segmentation network as backbone (with XYZ & RGB channels), and the O-CNN-based network, using octree observations.

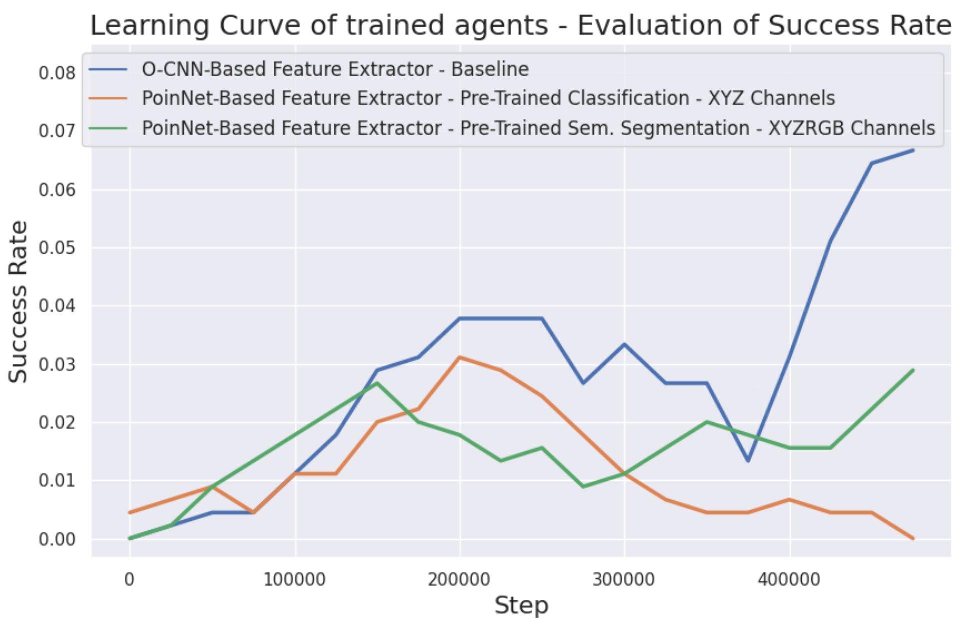

Figure 5.13: The plot of the success rate of the different feature extractor approaches is smoothed using a moving average with a window size of 3. The different approaches include the PointNet classification network as backbone (with XYZ channels), the PointNet semantic segmentation network as backbone (with XYZ & RGB channels), and the O-CNN-based network, using octree observations. Although it outperforms the classification network, the segmentation network cannot reach the same performance as the baseline model.

5.4.4 Architecture of Learnable Section

For the learnable section of the feature extractor, two different approaches were implemented, and their performances were compared. In the first approach, the global features extracted from the PointNet encoder go through two learnable fully-connected (FC) layers before being forwarded to the off-policy algorithm. The second approach utilizes both the global and the point-wise features from the PointNet encoder. The global features are processed by an initial FC-layer. Meanwhile, the point-wise features are concatenated with the XYZ and RGB channels before passing through a single learnable layer. The resulting point-wise features are then pooled using the mixed pooling strategy, where both max-pooling and average-pooling are applied to produce two separate feature vectors. The pooled features are then concatenated with the global features and go through one final FC-layer, before being fed into the algorithm. A visual representation of both architectures can be found in the Appendix. When looking at the learning curve of both approaches, one can see that while the second approach performs noticeably better in the mean episode reward (see Figure 5.14), it does not seem to outperform the first model in the success rate (see Figure 5.15). This is probably because both models have a very low achieved success rate, as they cannot compete with the performance of the octree-based baseline model. The main reason that the second model achieves a higher mean episode reward is that it reaches the second and third stages of the curriculum function at a much higher rate than the first model. Similar observations can be made when comparing the best respective models (see Table 5.7). Both PointNet-based models have a much lower success rate than the baseline model, with the second model performing slightly better than the first. From this experiment, as well as the previous one, one can deduce that the PointNet-based approach cannot demonstrate a similar performance as the octree-based approach.

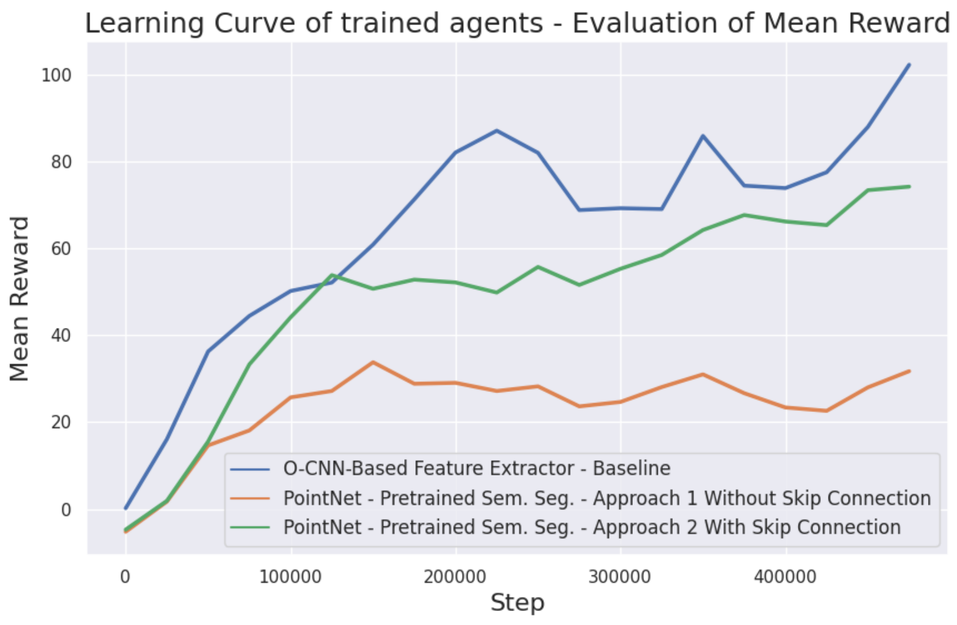

Figure 5.14: The plot of the mean reward of the different architectures for the learnable section of the PointNet-based feature extractors is smoothed using a moving average with a window size of 3. The second approach performs noticeably better than the first one, but cannot compete with the octree-based baseline model.

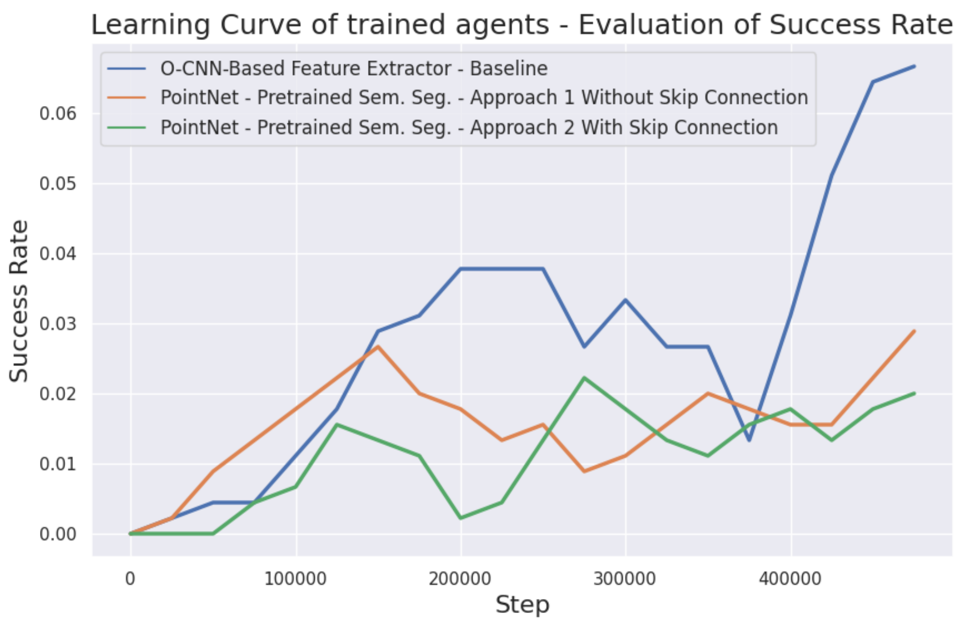

Figure 5.15: The plot of the success rate of the different architectures for the learnable section of the PointNet-based feature extractors is smoothed using a moving average with a window size of 3. Both approaches stagnate at a very low success rate, never even reaching 3%, and are clearly outperformed by the baseline model.

Table 5.7: The performance of the best model for two different architecture approaches for the learnable section of the PointNet-based extractor compared to the performance of the octree-based baseline model. One can observe that the baseline model has a significantly higher success rate than the PointNet-based models. While it still doesn't come close to the performance of the baseline model, the second model clearly performs better than the first one, with a noticeably higher mean episode reward.

Approach	Mean Episode Reward	Success Rate
PointNet-based - approach 1	33.35	0.50%
PointNet-based - approach 2	49.58	1.00%
Octree-based - baseline	75.88	5.00%

5.5 Experiments for Implementing 3D-Diffusion-Policy-Based Feature Extractor

The final feature extraction approach is based on the encoder network from the work by Ze et al. [55]. The feature extractor consists of several consecutive point-wise linear layers, followed by a pooling layer to obtain a global feature representation. These global features go through one final linear layer before being forwarded to the algorithm. A visual representation of the overall system architecture can be found in the Appendix. As shown in the learning curves for the mean episode reward (see Figure 5.16) and the success rate (see Figure 5.17), the best-performing PointNet-based and DP3-based approaches exhibit similar performance but underperform compared to the octree-based baseline. When evaluated under a new random seed, the best-performing model achieved a success rate of 5.50% and reached a mean episode reward of 60.21. This is almost on par with the top octree model, significantly outperforming the PointNet-based models. Due to its better evaluation performance and lower complexity, this approach appears more promising than the PointNet-based one. However, it still cannot compete with the baseline approach, primarily because it fails to consistently reach the final stage of the curriculum function. The root cause of this is unknown, but it seems evident that converting the point cloud into an octree enhances the network's ability to interpret and process 3D data efficiently. In this task, no approach that directly processes point-cloud data has been able to reproduce the performance of the octree-based baseline approach.

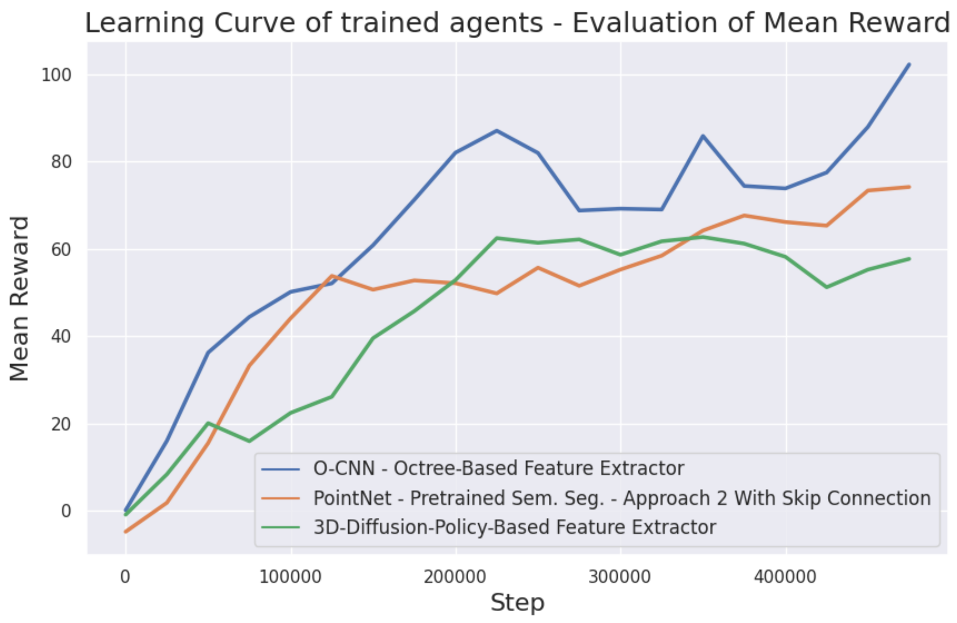

Figure 5.16: The plot of the mean reward of the different feature extractor approaches: octree-based, PointNet-based and DP3-based. The learning curve was is smoothed using a moving average with a window size of 3. While the PointNet- and DP3-based approaches have very similar performance, they both cannot match the baseline approach based on octrees.

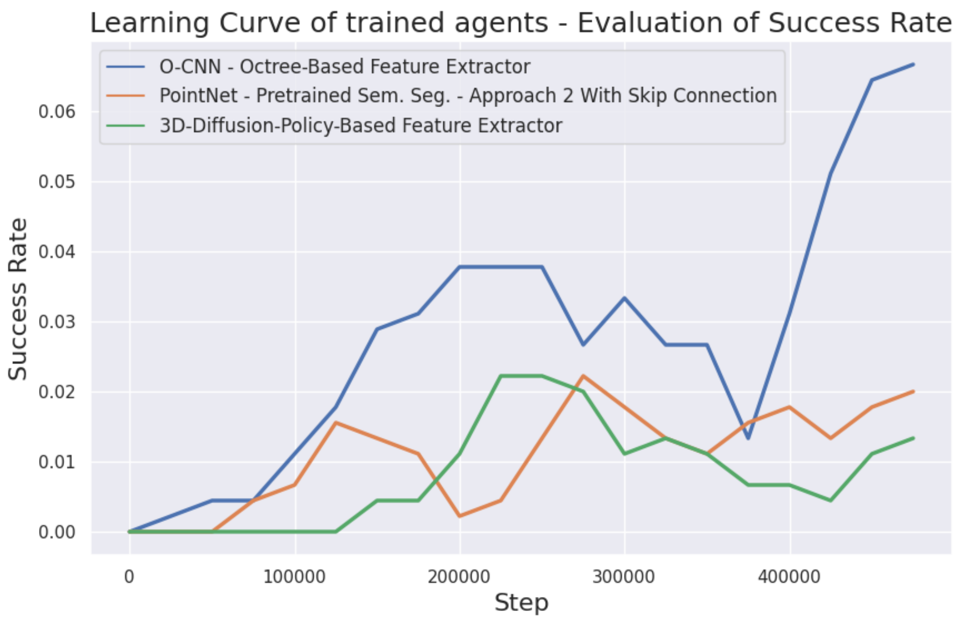

Figure 5.17: The plot of the success rate of the different feature extractor approaches: octree-based, PointNet-based and DP3-based. The learning curve was is smoothed using a moving average with a window size of 3. While the PointNet- and DP3-based approaches have very similar performance, they both cannot match the baseline approach based on octrees.

6 Conclusion

This master’s thesis addressed the problem of robotic grasping of unknown objects in an unstructured environment, based on the works of Orsula et al. [43]. The primary objective was to improve overall performance compared to the implementation upon which it is based by achieving a higher episode success rate. An episode can terminate successfully if it manages to grasp an object from the environment and lift it above a predetermined height threshold. The key findings and contributions of this thesis can be summarized as follows.

A modified curriculum function was implemented with three principal adjustment axes. First, an incremental reward function was introduced for the "Lift" stage, replacing the dynamic height threshold approach. Second, the persistent reward function, which previously applied a constant penalty at every time step, was modified by implementing a growing penalty that grows over the training time. Third, a single-object focus was enforced, ensuring that subsequent stages of the composite reward function could only be reached using the same object as in the previous stages. The findings from the experiments indicate that the incremental reward function for the "Lift" stage was more effective than using a dynamic height threshold, as it encouraged learning the actual target height, leading to improved performance. Additionally, a growing persistent reward penalty - whether linear or exponential - outperformed a constant penalty. However, due to high training instability, the number of training runs was insufficient to determine which of the two growth methods was superior. Lastly, implementing a single-object focus after the "Touch" stage did not improve performance compared to the baseline. Due to the high training instability, it resulted in fewer successful episodes during both training and evaluation.

Two feature extractor approaches were also implemented. The first was a PointNet-based feature extractor with variations in system architecture and task used for pre-training the encoder. The second was a feature extractor based on the encoder of the DP3 network. The experiments demonstrated that the best performance in the PointNet-based approach was achieved by pre-training the encoder on a semantic segmentation task and incorporating skip connections in the learnable section, although the performance improvement was relatively minor. While the DP3-based approach appeared more promising than the PointNet-based one, both underperformed compared to the octree-based baseline. From this, one could conclude that transforming point-cloud data into an octree before feature extraction leads to better performance than directly processing the point-cloud data.

7 Outlook

7.1 Limitations

The first limitation encountered was that each training process took considerable time due to the computationally expensive environment step. Simulating the physics, rendering, and low-level controls of the robot in its environment required considerable computational effort. Additionally, due to the risk of interference between the ros2 nodes, it was impossible to parallelize the training runs, further exacerbating the problem arising from the slow training. These two combined issues greatly limited the feasible number of potential training runs. Additionally, the extensive domain randomization led to low training stability, with high variance observed between runs. As a result, conducting numerous runs per configuration became necessary to obtain meaningful results, further increasing the difficulty of obtaining highly performant models during training. These two primary challenges - the limited number of feasible training runs and the high variance from unstable training - severely restricted the number of testable configurations. Furthermore, testing was limited to a single simulated environment, although this environment allowed for a wide range of domain randomization.

7.2 Future Works

The following additional experiments could be performed in the future to enhance the analysis and performance of the system: (1) Determine the optimal method for implementing a persistent reward penalty by conducting further training runs (e.g., 10 or 20) to obtain statistically significant results. One could determine the optimal parameters for each strategy, such as the doubling frequency for the exponential approach and the increasing frequency and the step size for the linear approach. (2) Explore potential architectural improvements for the learnable sections of the PointNet and DP3 feature extractors by training additional models with varying hyperparameters, such as the number and size of learnable layers. (3) Assess the performance of different point-cloud acquisition approaches, building on the already implemented code. The first baseline approach directly generates the point cloud inside the Gazebo simulation and down-samples it to 1024 points via random sampling. In the second approach, an RGB-D image from a simulated camera sensor is transformed into a point cloud via projection, followed by random sampling to down-sample to 1024 points. One can add Gaussian noise to the depth image to simulate a noisy real-world depth sensor and evaluate the performance drop compared to the ideal point cloud from the first approach. Finally, the third approach uses the same sensory input as approach 2 but replaces random sampling with FPS to improve point-cloud quality and mitigate potential performance degradation observed in approach 2. Another improvement to implement would be to incorporate the success rate measurements for the intermediate stages "Reach," "Touch," and "Grasp" during evaluation episodes. This would provide a more detailed understanding of the model's progress in each sub-category. Simultaneously, one should investigate strategies to address the high training instability observed during experimentation. Once these issues are all tackled, the implemented code should be extended to a different experimental setting with a different robot, potentially leveraging the setup from the chair lab. Finally, one could attempt to transfer this new experimental setting to the real-world scenario, bridging the gap between simulation and practical deployment.

Abbreviations

BC	Behavioral Cloning
CNN	Convolutional Neural Network
DDPG	Deep Deterministic Policy Gradient
DP3	3D Diffusion Policy
DQN	Deep Q-Network
DOF	Degrees of Freedom
FC	fully-connected
FPS	Farthest Point Sampling
IK	Inverse Kinematics
IRL	Inverse Reinforcement Learning
KDL	Kinematics and Dynamics Library
KNN	K-Nearest Neighbors
KDL-RR	Kinematics and Dynamics Library with Random Restarts
MDP	Markov Decision Process
mIoU	mean Intersection-over-Union

MLP multi-layer perceptron

O-CNN Octree-Based Convolutional Neural Network

RL Reinforcement Learning

RRT Rapidly-exploring Random Trees

SAC Soft Actor-Critic

SQP Sequential Quadratic Programming

SQP-SS Sequential Quadratic Programming - Sum of Squares

TCP tool center point

TD3 Twin Delayed DDPG

TQC Truncated Quantile Critics

List of Figures

List of Tables

Bibliography

[1] B. Argall, S. Chernova, M. Veloso, and B. Browning. "A survey of robot learning from demonstration." In: *Robotics and Autonomous Systems* 57 (May 2009), pp. 469–483. DOI: `10.1016/j.robot.2008.10.024`.

[2] P. Beeson and B. Ames. "TRAC-IK: An open-source library for improved solving of generic inverse kinematics." In: *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*. 2015, pp. 928–935. DOI: `10.1109/HUMANOIDS.2015.7363472`.

[3] J. Bohg, A. Morales, T. Asfour, and D. Kragic. "Data-Driven Grasp Synthesis—A Survey." In: *IEEE Transactions on Robotics* 30.2 (Apr. 2014), pp. 289–309. ISSN: 1941-0468. DOI: `10.1109/tro.2013.2289018`.

[4] H. Choset. "Principles of Robot Motion: Theory, Algorithms, and Implementations." In: *MIT Press google schola* 2 (2005), pp. 105–118.

[5] D. Coleman, I. A. Sucan, S. Chitta, and N. Correll. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study." In: *ArXiv* abs/1404.3785 (2014).

[6] J. Courtney. "Marker-Free Human Motion Capture with Applications in Gait Analysis." PhD thesis. June 2005. DOI: `10.13140/2.1.3143.9368`.

[7] A. Dobson and K. Bekris. "Sparse roadmap spanners for asymptotically near-optimal motion planning." In: *International Journal of Robotics Research* 33 (Jan. 2014), pp. 18–47. DOI: `10.1177/0278364913498292`.

[8] G. Fantoni, M. Santochi, G. Dini, K. Tracht, B. Scholz-Reiter, J. Fleischer, T. Kristoffer Lien, G. Seliger, G. Reinhart, J. Franke, H. Nørgaard Hansen, and A. Verl. "Grasping devices and methods in automated production processes." In: *CIRP Annals* 63.2 (2014), pp. 679–701. ISSN: 0007-8506. DOI: `https://doi.org/10.1016/j.cirp.2014.05.006`.

[9] C. Fd, N. Dodgson, and C. Moenning. "Fast Marching farthest point sampling." In: (May 2003).

[10] S. Fujimoto, H. van Hoof, and D. Meger. "Addressing Function Approximation Error in Actor-Critic Methods." In: *International Conference on Machine Learning*. 2018.

[11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." In: *ArXiv* abs/1801.01290 (2018).

[12] R. Hanocka, A. Hertz, N. Fish, R. Giryes, S. Fleishman, and D. Cohen-Or. "MeshCNN: a network with an edge." In: *ACM Transactions on Graphics* 38.4 (July 2019), pp. 1–12. ISSN: 1557-7368. DOI: 10.1145/3306346.3322959.

[13] R. Hartley. *Multiple view geometry in computer vision*. Vol. 665. Cambridge university press, 2003, pp. 153–158.

[14] L. Hoang, S.-H. Lee, O.-H. Kwon, and K.-R. Kwon. "A deep learning method for 3D object classification using the wave kernel signature and a center point of the 3D-triangle mesh." In: *Electronics* 8.10 (2019), p. 1196.

[15] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees." In: *Autonomous Robots* 34 (Apr. 2013). DOI: 10.1007/s10514-012-9321-0.

[16] D. Hsu, J.-C. Latombe, and R. Motwani. "Path planning in expansive configuration spaces." In: *Proceedings of International Conference on Robotics and Automation*. Vol. 3. 1997, 2719–2726 vol.3. DOI: 10.1109/ROBOT.1997.619371.

[17] Y. Jiang, S. Moseson, and A. Saxena. "Efficient grasping from RGBD images: Learning using a new rectangle representation." In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3304–3311. DOI: 10.1109/ICRA.2011.5980145.

[18] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. 2018. arXiv: 1806.10293 [cs.LG].

[19] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[20] *KDL Documentation*. https://docs.orocos.org/kdl/overview.html. Accessed: November 21, 2024.

[21] K. Kleeberger, R. Bormann, W. Kraus, and M. Huber. "A Survey on Learning-Based Robotic Grasping." In: *Current Robotics Reports* 1 (Dec. 2020), pp. 239–249. DOI: 10.1007/s43154-020-00021-6.

[22] M. Kopicki, R. Detry, M. Adjigble, R. Stolkin, A. Leonardis, and J. L. Wyatt. "One-shot learning and generation of dexterous grasps for novel objects." In: *The International Journal of Robotics Research* 35.8 (2016), pp. 959–976. DOI: 10.1177/0278364915594244.

[23] O. Kroemer, S. Niekum, and G. Konidaris. *A Review of Robot Learning for Manipulation: Challenges, Representations, and Algorithms*. 2020. arXiv: `1907.03146 [cs.RO]`.

[24] J. Kuffner and S. LaValle. "RRT-connect: An efficient approach to single-query path planning." In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 2. 2000, 995–1001 vol.2. DOI: `10.1109/ROBOT.2000.844730`.

[25] S. Kumar, N. Sukavanam, and R. Balasubramanian. "An Optimization Approach to Solve the Inverse Kinematics of Redundant Manipulator." In: 2010.

[26] A. Kuznetsov, P. Shvechikov, A. Grishin, and D. P. Vetrov. "Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics." In: *International Conference on Machine Learning*. 2020.

[27] K. Lai, L. Bo, and D. Fox. "Unsupervised feature learning for 3D scene labeling." In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 3050–3057. DOI: `10.1109/ICRA.2014.6907298`.

[28] S. M. LaValle. "Rapidly-exploring random trees : a new tool for path planning." In: *The annual research report* (1998).

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[30] I. Lenz, H. Lee, and A. Saxena. "Deep learning for detecting robotic grasps." In: *The International Journal of Robotics Research* 34.4-5 (2015), pp. 705–724.

[31] S. Levine, C. Finn, T. Darrell, and P. Abbeel. *End-to-End Training of Deep Visuomotor Policies*. 2016. arXiv: `1504.00702 [cs.LG]`.

[32] D. Li, Y. Wei, and R. Zhu. "A comparative study on point cloud down-sampling strategies for deep learning-based crop organ segmentation." In: *Plant Methods* 19 (Nov. 2023). DOI: `10.1186/s13007-023-01099-7`.

[33] Y. Li. "Deep Reinforcement Learning." In: *ArXiv* abs/1810.06339 (2018).

[34] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning." In: *CoRR* abs/1509.02971 (2015).

[35] J. Mahler, J. Liang, S. Niyaz, M. Laskey, R. Doan, X. Liu, J. A. Ojea, and K. Goldberg. "Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics." In: *arXiv preprint arXiv:1703.09312* (2017).

[36] D. Maturana and S. Scherer. "VoxNet: A 3D Convolutional Neural Network for real-time object recognition." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 922–928. DOI: `10.1109/IROS.2015.7353481`.

[37] D. Maturana and S. A. Scherer. "VoxNet: A 3D Convolutional Neural Network for real-time object recognition." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2015), pp. 922–928.

[38] D. J. Meagher. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[39] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. "Playing Atari with Deep Reinforcement Learning." In: *ArXiv* abs/1312.5602 (2013).

[40] D. Morrison, P. Corke, and J. Leitner. *Closing the Loop for Robotic Grasping: A Real-time, Generative Grasp Synthesis Approach*. 2018. arXiv: `1804.05172 [cs.RO]`.

[41] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning." In: *ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning* (May 2000).

[42] A. Orsula. "Deep Reinforcement Learning for Robotic Grasping from Octrees." Available at: `https://vbn.aau.dk/ws/files/421582447/Deep_Reinforcement_Learning_for_Robotic_Grasping_from_Octrees.pdf`, Accessed: November 21, 2024. Master's thesis. Aalborg University, 2021.

[43] A. Orsula, S. Boegh, M. A. Olivares-Méndez, and C. Martínez. "Learning to Grasp on the Moon from 3D Octree Observations with Deep Reinforcement Learning." In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2022), pp. 4112–4119.

[44] D. Pomerleau. "ALVINN: An Autonomous Land Vehicle In a Neural Network." In: *Proceedings of (NeurIPS) Neural Information Processing Systems*. Ed. by D. Touretzky. Morgan Kaufmann, Dec. 1989, pp. 305–313.

[45] C. Qi, H. Su, K. Mo, and L. J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 77–85.

[46] C. Qi, L. Yi, H. Su, and L. J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space." In: *ArXiv* abs/1706.02413 (2017).

[47] G. Qian, Y. Li, H. Peng, J. Mai, H. Hammoud, M. Elhoseiny, and B. Ghanem. "PointNeXt: Revisiting PointNet++ with Improved Training and Scaling Strategies." In: *ArXiv* abs/2206.04670 (2022).

[48] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.

[49] *Stable Baselines3*. `https://github.com/DLR-RM/stable-baselines3`. Accessed: November 21, 2024.

[50] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[51] H. Van Hasselt, A. Guez, and D. Silver. "Deep reinforcement learning with double q-learning." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[52] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong. "O-cnn: Octree-based convolutional neural networks for 3d shape analysis." In: *ACM Transactions On Graphics (TOG)* 36.4 (2017), pp. 1–11.

[53] X. Yao, J. Guo, J. Hu, and Q. Cao. "Using Deep Learning in Semantic Classification for Point Cloud Data." In: *IEEE Access* 7 (2019), pp. 37121–37130. DOI: `10.1109/ACCESS.2019.2905546`.

[54] L. Yi, V. G. Kim, D. Ceylan, I.-C. Shen, M. Yan, H. Su, C. Lu, Q. Huang, A. Sheffer, and L. Guibas. "A scalable active framework for region annotation in 3D shape collections." In: *ACM Trans. Graph.* 35.6 (Dec. 2016). ISSN: 0730-0301. DOI: `10.1145/2980179.2980238`.

[55] Y. Ze, G. Zhang, K. Zhang, C. Hu, M. Wang, and H. Xu. "3D Diffusion Policy." In: *ArXiv* abs/2403.03954 (2024).

Appendix

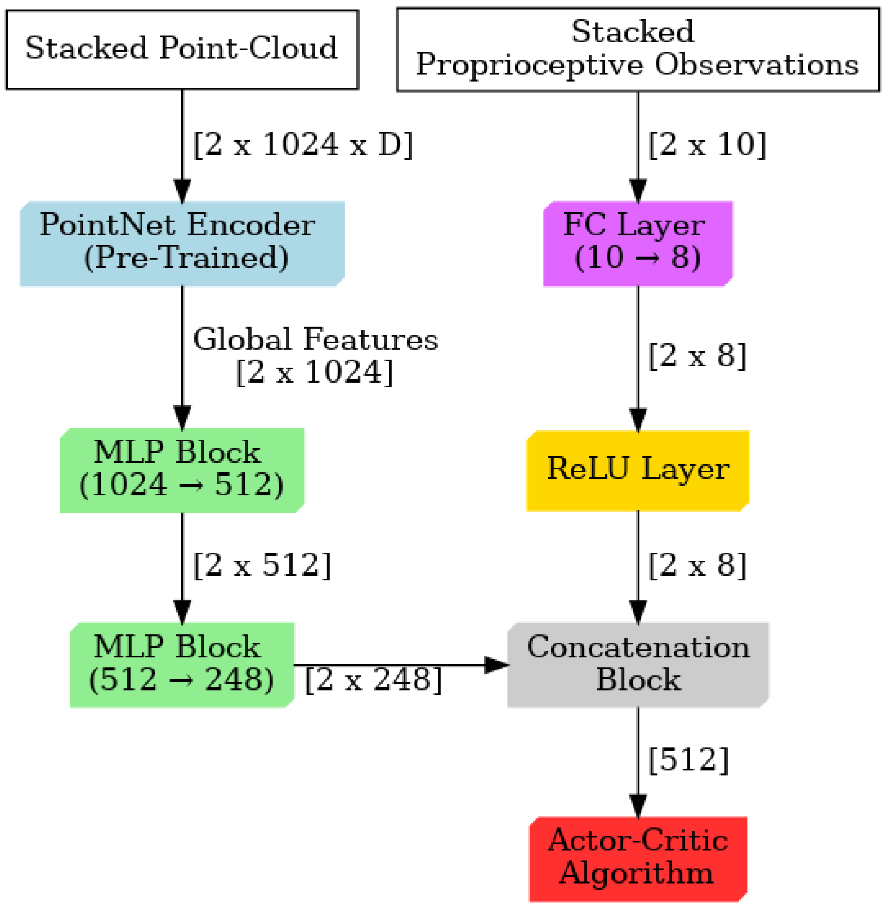

Figure 1: The architecture of the system pipeline from the first PointNet-based approach. The PointNet encoder was pre-trained on a semantic segmentation data-set and outputs the global features. The MLP block consists of an FC layer, followed by a batch-normalization layer and a ReLU activation function. The obtained visual features get concatenated with the proprioceptive features before being forwarded to the actor-critic algorithm.

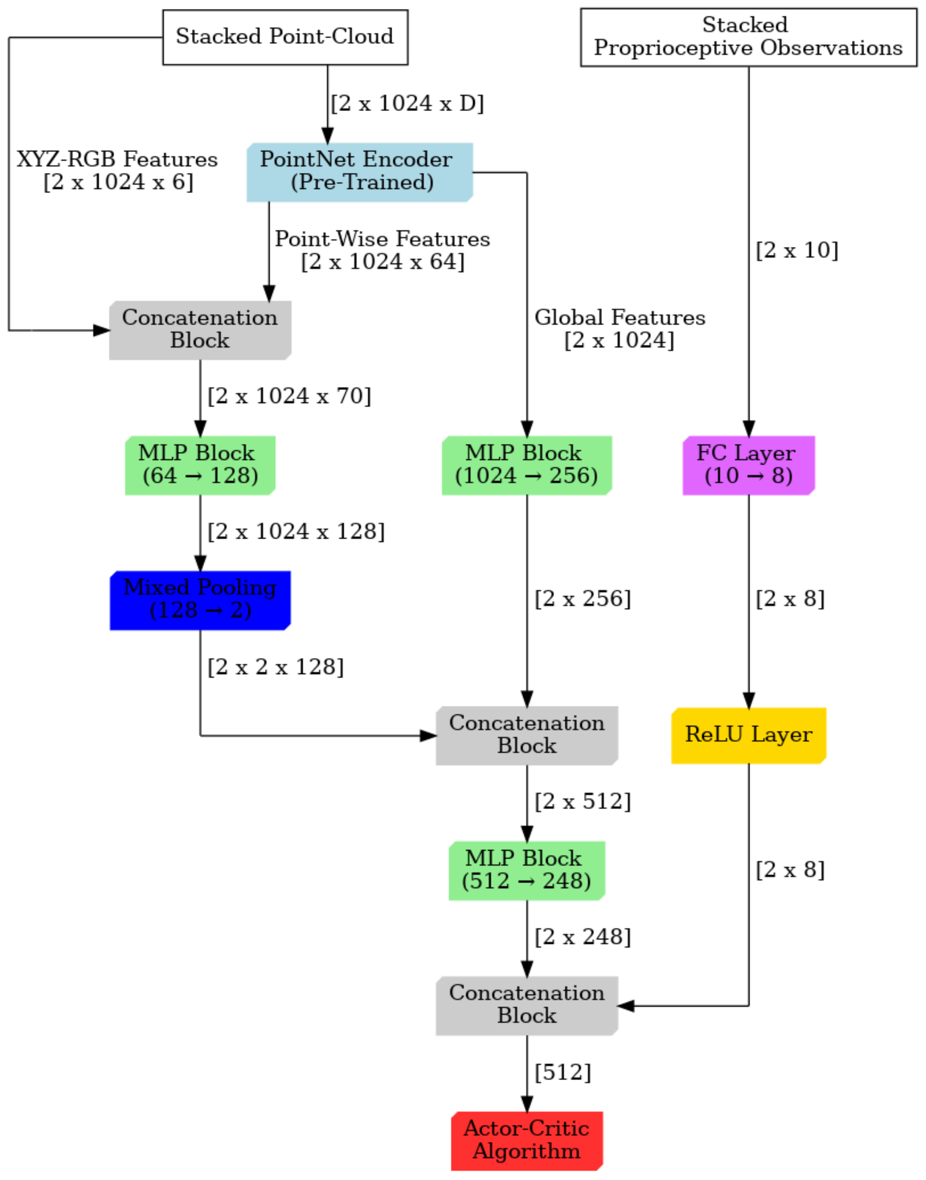

Figure 2: The architecture of the system pipeline from the second PointNet-based approach. The PointNet encoder was pre-trained on a semantic segmentation data-set and outputs the global and point-wise features. The MLP-block consists of a FC-layer, followed by a layer-normalization layer and a ReLU activation function. The mixed pooling layer outputs two feature vectors, combining max pooling with average pooling. The final visual features vector is concatenated with the proprioceptive features before being inputted into the actor-critic algorithm.

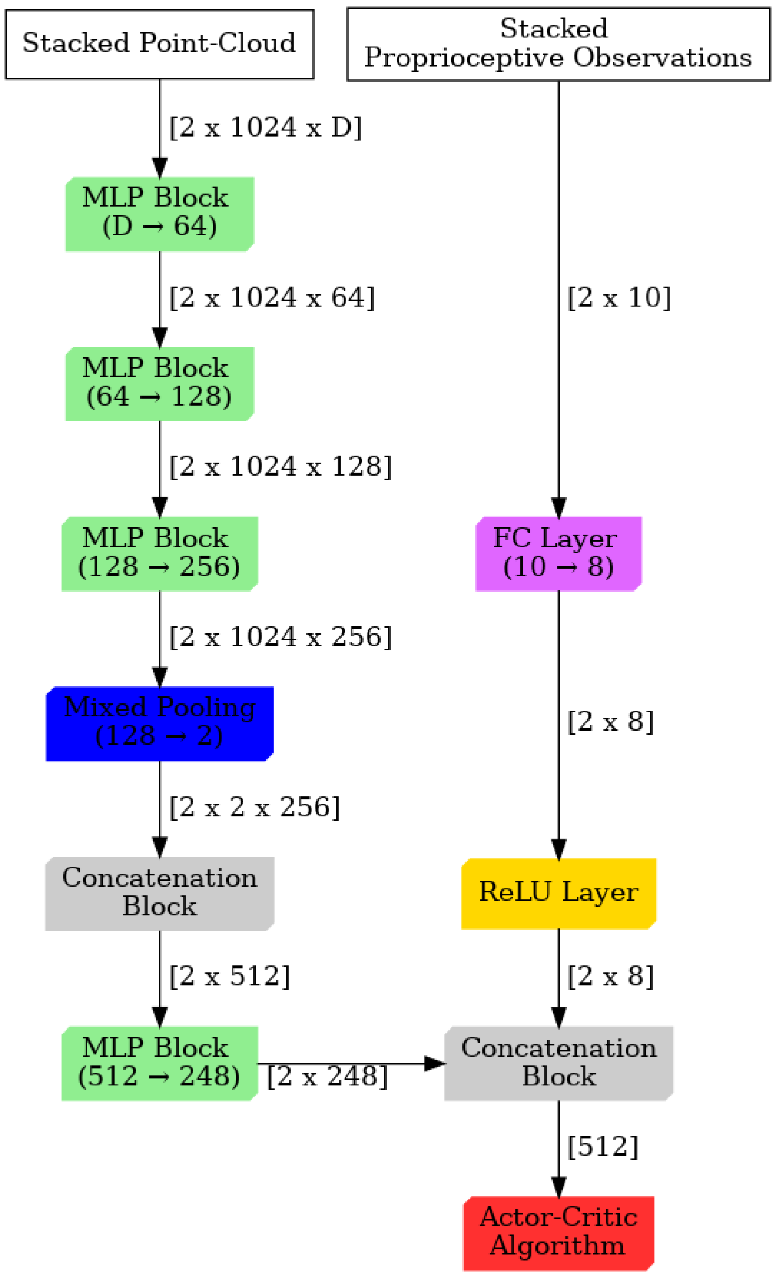

Figure 3: The architecture of the system pipeline from the first 3D-Diffusion-Policy-based approach. The MLP-block consists of a FC-layer, followed by a layer-normalization layer and a ReLU activation function. The mixed pooling layer outputs two feature vectors, combining max pooling with average pooling. The obtained visual features are concatenated with the proprioceptive features before being forwarded to the actor-critic algorithm.