Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Reinforcement Learning with Parametrized Actions for Imitation Learning

Marek Majsner

Supervisor: Mgr. Karla Štěpánová, Ph.D.
January 2024

Acknowledgements

First and foremost, I would like to use this opportunity to thank my supervisor for her advice and endless patience during my work on this thesis. I would also like to thank my family for providing me with support during my studies and I want to thank all my teachers for providing me with the knowledge necessary for the creation of this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 8. January 2024

Abstract

This thesis explores the benefits of temporal abstraction in reinforcement learning by examining two approaches for action generalization: Parameterized Action Markov Decision Processes (PAMDPs) and the options framework. In this work, we employ the Manipulation Primitive-augmented reinforcement Learning (MAPLE) algorithm founded on the PAMDPs framework. We describe the inner workings of the MAPLE algorithm and its underlying Robostuite simulation framework and evaluate the framework in new tasks and with different parameters. Moreover, we propose an extension to the framework to include observational information.

Keywords: MAPLE, reinforcement learning, PAMPDS, options, Robosuite

Supervisor: Mgr. Karla Štěpánová, Ph.D.

Abstrakt

Tato práce zkoumá výhody časové abstrakce v posilovacím učení zkoumáním dvou přístupů pro zobecnění akcí: Parameterized Action Markov Decision Processes (PAMDP) a rámce options. V této práci používáme zesílení Manipulation Primitive-augmented reinforcement Learning (MAPLE) algoritmus založený na struktuře PAMDPs. Popisujeme vnitřní fungování algoritmu MAPLE a jeho základního simulačního platformu Robostuite, navíc hodnotíme metodu v nových úlohách a s různými parametry. Kromě toho navrhujeme rozšíření metody informacemi z pozorování.

Klíčová slova: MAPLE, posilovací učení, PAMPDS, options, Robosuite

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Motivation

The recent success of predictive language models and the progress achieved in advanced robot control systems (for instance, the large multimodal model GPT-4 [21] and Bostons dynamics Atlas research platform[6]) have instilled the idea that personal robotics is close to integrating with humans in the home and workplace environments. Despite their success, state-of-the-art Deep reinforcement learning (DRL) models fall far from tackling long-horizon tasks due to the exploration and observation challenge in complex environments. In order to come up with a solution, the robot has to explore a large number of actions and observe the effects of each individual action on its environment. For example, in a simple environment where the task is to pick up a cube, random motor motions of a 6DOF robot will hardly ever get close to the cube. Moreover, the learning time and the number of actions to explore this simple environment using random exploration is unfeasible. This limits the ability of DRL models to perform long-horizon tasks (such as handling and cutting up ingredients) effectively and efficiently in stochastic environments. Since the introduction of DRL, there have been many attempts to aid the exploration burden by introducing higher levels of abstraction, for example, using semi-Markov decision processes [1] or improved exploration strategies [2].

To get around the problems of exploration in complex environments, an abundance of work in robotics has been dedicated to learning robot control for individual tasks, such as grasping [3]. With the combination of algorithms for optimal motion planning, recent works tried to solve the necessity of exploring complex environments by creating a hierarchical model [7] [8].

Nevertheless, there is a growing recognition in the field of the importance of transferring knowledge between tasks to speed up the learning process to accomplish long-horizon tasks. One promising approach to transferring knowledge of learned controls for robots is to use a parameterized action space, which allows for a compact representation of the robot's behavior in DRL algorithms.

Parameterized Action Markov Decision Processes (PAMDPs) allow for representation that gives the underlying solution to the given task in the form of parameterized action, which could be transferred to similar tasks, thus aiding the strain of exploration in DRL [4]. It also promises the ability to solve long-horizon tasks in complex environments by dividing them into a series of parameterized actions that lead to the successful completion of the complex tasks [5].

1.2 Project goals

To assist long-horizon solutions by incorporating demonstration data, our first goal is to identify a suitable method. We will explore approaches that leverage hierarchical structures, such as options (generalized primitive actions) [1] and parameterized action [13], and whose solutions lead to reusable policies. We will:

- Describe each method in detail.
- Compare the advantages and limitations of options and parameterized action.

Secondly, once we have identified the most suitable method, we will:

- Evaluate the method's training sensitivity to its parameters.
- Assess the method's capability to adapt to newly designed environments.
- Create and evaluate additional parametrized action.

Finally, our goal is to determine whether it is feasible to expand the selected method by:

- Integrating additional information (e.g., demonstrations, task descriptions) into the learning process.
- Testing whether this extension can expedite the learning process.

Chapter 2

Background

In this section, we briefly introduce the core concepts in reinforcement learning with Markov decision processes, deep reinforcement learning, options, and finally the fundamentals behind Reinforcement Learning with Parameterized Action Markov Decision Processes.

2.1 Reinforcement Learning with Markov Decision Processes

A Markov Decision Process (MDP) is defined by the tuple (S, A, P, R, γ), where S denotes a set of possible states that describe the current state in the environment, A A set of possible actions that the agent can take in each state, P is a transition function that defines the probability of moving from one state to another state as a result of taking an action, R represents the reward function that assigns a real-valued reward to each state-action pair and γ is the discount factor [1]. The core idea of reinforcement learning (RL) is to learn a Markov policy, a mapping from states to probabilities of taking each available action, $\pi : S \times A \Rightarrow [0, 1]$, that maximizes the expected discounted future reward from each state $s \in S$.

RL agent interacts with an environment following a policy $\pi(a_t|s_t)$, which represents the agent's behavior at each time step t and perceived state $s_t \in S$. The agent selects an action $a_t \in A$ (by mapping from state s_t to actions a_t) and as a result of the action receives a reward r_t and transitions to the next state s_{t+1}, which is the result of the agent's interaction with the environment. The agent's goal is to maximize the discounted long-term returned reward $R_t = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$, where the discount factor is $\gamma \in (0, 1]$. The expected return $v_\pi(s) = E[R_t|s_t = s]$ of the following policy π is called the value for given state s. The value function then works as a prediction of the expected accumulative future reward giving the representation of quality for each state-action pair. The optimal state-value policy π^* maximizes the value achievable by any policy for state s and action a. To learn the optimal policy, the agent explores the state and action spaces relevant to the task by executing various sequences of state-action-next-state transitions. The average length of such sequences is called the task horizon. If the horizon is long while the task involves large state and action spaces, then the exploration space also becomes large. This results in the poor performance of the standard RL algorithms on such long-horizon tasks without sophisticated exploration techniques. Conventionally, the MDPs do not carry any temporal abstraction or higher-level representation of behavior [9].

2.2 Reinforcement learning for option discovery

The term 'options' (generalized primitive actions) was introduced in the semi-Markov decision process (SMDP) in [1] as a step forward to incorporating temporal abstraction.

The addition of options as courses of temporally extended actions allows RL agents to learn and create a set of internal policies that in effect represent temporal abstraction. An option ω is a tuple (I, π, β) where $\pi : S \times A \rightarrow [0, 1]$ consists of a policy, $I \subseteq S$ denotes the option's initiation set and $\beta : S \rightarrow [0, 1]$ represents the termination condition. Termination condition β is the probability that option ω will terminate at a given state. In an initialization state I the agent carries out actions according to the option's policy ω, until the termination conditions are satisfied.

The benefit of having the initiation set I and termination condition β as part of an option is that it limits the range over which the option's policy needs to be defined [1]. This means that a robot follows a policy ϕ of washing dirty dishes only if the initial state, for example, the presence of dirty dishes, is met. Furthermore, options introduce a level of temporal abstraction and allow for hierarchical representation and hierarchical planning. However, the discovery and creation of options in high-dimensional continuous spaces is a long-standing problem.

2.3 Reinforcement Learning with Parameterized Action Space Markov Decision Processes

An alternative to using options to generate the compositional structure of long-horizon tasks is reinforcement learning with parameterized actions [13] [14]. In Parameterized Action Space Markov Decision Processes (PAMDPs) is the state space considered continuous $S \in \mathbb{R}^n$, but the actions are parameterized. The action set is a finite set of discrete actions $A_d = (a_1, a_2, a_3, ..., a_k)$, where each discrete action $a_n \in A_d$ has a set of continuous parameters $X_a \in \mathbb{R}^{m_a}$. Action is defined as a tuple (a, x), where a represents a discrete primitive action, and x denotes the parameters for that primitive action. Therefore, the agent must select both primitive action a_i and its corresponding parameters x_i at each decision-making step. Union over all discrete actions with all the possible parameters for that action $A = \cup_{a \in A_d} \{(a, x) | x \in X_a\}$ is the action space.

In PAMDPs, each task can be efficiently addressed by identifying the suitable set of parametrized actions. The goal of reinforcement learning with PAMDPs is to find this ideal set autonomously.

2.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) combines the artificial neural networks of deep learning with reinforcement learning algorithms. Training neural networks provide the means to solve complex, decision-making problems [31],[32]. The deep learning techniques enable us to approximate the optimal policy π_ϕ or value function V_ψ in RL with parameters ψ and ϕ. Nevertheless, deep online RL methods encounter evaluation challenges. Inadequate account for statistical variability[33] and sensitivity to the hyperparameter selection[34].

2.5 Soft Actor-Critic

The concepts of DRL in an on-policy manner necessitate new samples for nearly every policy update [26]. This escalates the samples needed to develop an effective policy. To address the challenges of high sample complexity, the Soft Actor-Critic (SAC) algorithm employs off-policy DRL [27]. Furthermore, SAC addresses the brittleness of convergence properties connected with the model-free Deterministic Policy Gradient (DDPG) methods [36] by maximizing the policy entropy at each timestep. Maximizing entropy in the framework nudges the agent to explore the environment more effectively and increases the likelihood of uncovering new behaviors that may result in greater rewards [26].

The network architecture is outlined in Figure 2.2. SAC concurrently learns a parametrized policy π_ϕ and two parametrized Q-functions Q_{θ_1}, Q_{θ_2}. Doubling of Q-functions helps to ensure the policy π_ϕ could not exploit inconsistencies in the approximated Q function by taking the minimum of two Q-values. SAC algorithm 1 first carries out the exploration endeavor inside the simulated environment and stores the observations of action, state, and reward tuples in replay pool D see Figure 2.1. Batches of data sampled from the replay pool D are used in the second phase of SAC for the computation of stochastic gradients, updating the parameters θ_1, θ_2, ϕ.

To train the policy π_ϕ SAC alters the goal of maximizing the expected sum of rewards

$$\sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[(r(s_t, a_t)\right],$$

where the environment returns a bounded reward $r : S \times A \to [r_{\min}, r_{\max}]$, by including the maximum entropy objective

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[(r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t)))\right],$$

where $\rho_\pi(s_t, a_t)$ denotes the state-action marginal of the trajectory distribution induced by the policy $\pi(a_t|s_t)$, and α is a temperature parameter that scales the importance of the entropy term. The policy parameters ϕ are then updated by minimizing the expected augmented Kullback-Leibler divergence

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim D}\left[\mathbb{E}_{a_t\sim\pi_\phi}\left[\alpha\log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t,a_t)\right]\right].$$

To evaluate the gradient of objective J_π the original authors in [26] used the reparameterization trick

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim D,\epsilon_t\sim\mathcal{N}}\left[\log\pi_\phi(f_\phi(\epsilon_t;s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t;s_t))\right],$$

where the policy is replaced with a neural network transformation $a_t = f_\phi(\epsilon_t; s_t)$ with noise vector ϵ_t sampled from some fixed distribution as an input. Since SAC is composed of two Q-functions the update uses the minimum of the two approximators

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim D,\epsilon_t\sim\mathcal{N}}\left[\log\pi_\phi(f_\phi(\epsilon_t;s_t)|s_t) - \min_{j=1,2} Q_{\theta_j}(s_t, f_\phi(\epsilon_t;s_t))\right].$$

The Q-function parameters θ_1 and θ_2 are obtained by minimizing the Bellman residual

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t)\right)^2\right],$$

$$\hat{Q} = r(s_t,a_t) + \gamma\left(\min_{j=1,2} Q_{\bar{\theta}_j}(s_{s+1}, a_{t+1}) - \alpha\log\pi_\phi(a_{t+1}|s_{t+1})\right),$$

where the next action a_{t+1} is a newly sampled action from the stored observations from replay pool D see Figure 2.2. Parameters $\bar{\theta}$ refer to the use of target Q-functions, which are exponentially moving averages of the Q-functions.

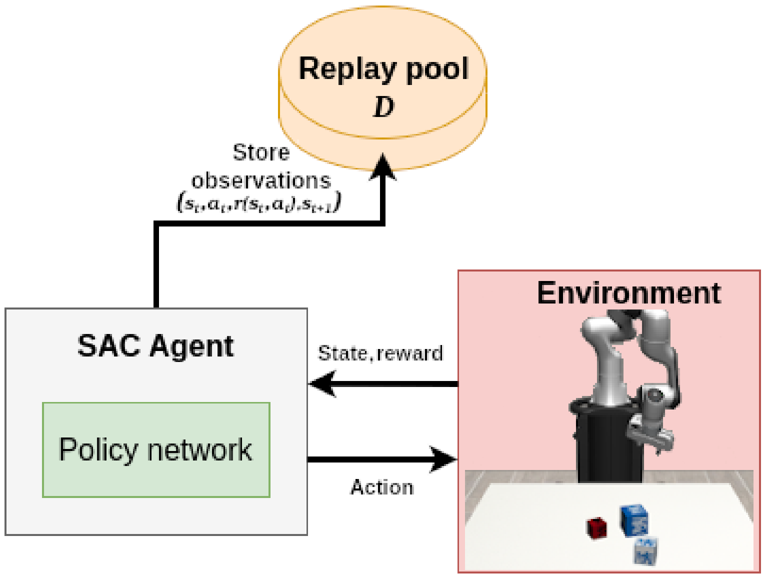


Figure 2.1: Diagram of SAC's replay pool D.

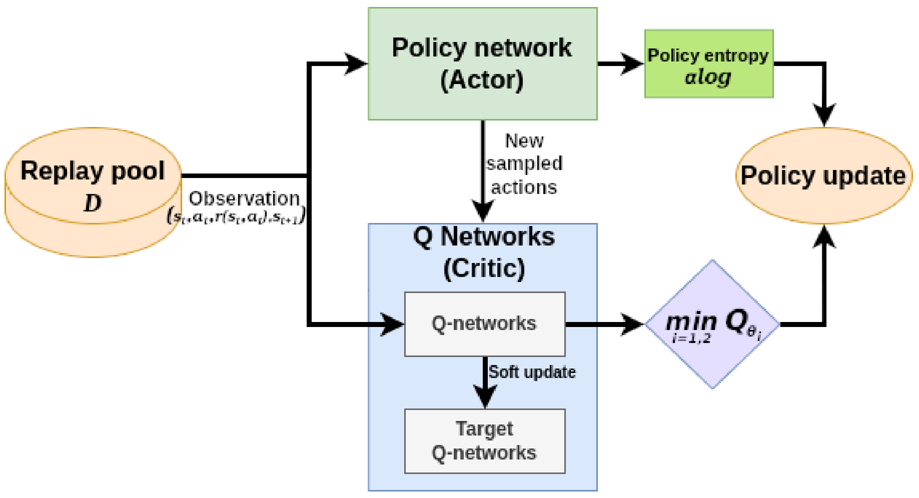


Figure 2.2: Diagram of SAC architecture.

Algorithm 1 Soft Actor-Critic

```
1:  Initialize Q networks weights $\theta_1, \theta_2$ and policy network weights $\phi$
2:  Initialize target network weights $\bar{\theta}_1 \leftarrow \theta_1$, $\bar{\theta}_2 \leftarrow \theta_2$
3:  Initialize replay buffer $D$
4:  for each iteration do
5:      for each environment step do                              ▷ Exploration Phase
6:          Sample action $a_t$ from the policy $\pi_\phi(a_t|s_t)$
7:          Sample transition from the environment $s_{t+1}$
8:          Add transition to replay buffer $D \leftarrow D \cup \{(s_t, a_t, r, s_{t+1})\}$
9:      end for
10:     for each gradient step do                                 ▷ Training Phase
11:         Update the Q networks $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
12:         Update policy weights $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
13:         Entropy adjustment $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
14:         Update target networks $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
15:     end for
16: end for
```

Chapter 3

Related Work

In this chapter, we discuss related works in solving long-horizon problems using options and parametrized actions. We compare the two introduced approaches in Table 3.1 and pick a suitable method for incorporating prior knowledge. We conclude this chapter with different approaches to integrating prior knowledge.

3.1 Options frameworks

One design of hierarchical structures in RL arises from Skill Chaining based upon the proposed Options framework in [1]. Skill chaining presumes that long-horizon tasks can be deconstructed into a sequence of smaller skills, which are comparatively easier to learn. Each skill is represented as an option tuple (I, π, β), transforming the problem from regular MDPs into Options as defined in Section 2.2. Options reduce the complexity of representing the task's value function by segmenting it into multiple local value functions. Each option then centers around its own specific local value function. This segmentation helps the approximation in high-dimensional problems[10]. Therefore Skill chaining appears to be a good solution for solving long horizon tasks.

Skill chains are created from the goal region of the task by linking options backward to the starting point vis Figure3.1. Moreover, the method in [10] proposed forming skill trees to allow for multiple solutions or start states see Figure 3.2. The complexity of the skill tree is limited by avoiding the overlap of options' initiation sets targeting the same goal region. Agents using skill chaining in [10] showed significantly better performance than those with a flat policy. Furthermore, agents starting with pre-learned options achieved faster learning. However, the experiments in [10] were run and tested only in a simple two-dimensional Pinball Domain, where the goal is to guide a small ball to a desired space in a maze.

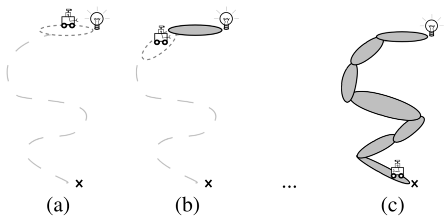


Figure 3.1: Option discovery using Skill Chaining.
Source: https://www.semanticscholar.org/paper/Skill-Discovery-in-Continuous-Reinforcement-Domains-Konidaris-Barto/bdc5a10aa5805808cfca58ac527ddc23e73 7bee8

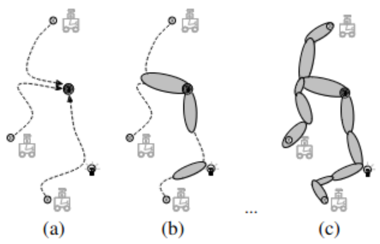


Figure 3.2: Creation of skill trees.
Source: https://www.semanticscholar.org/paper/Skill-Discovery-in-Continuous-Reinforcement-Domains-Konidaris-Barto/bdc5a10aa5805808cfca58ac527ddc23e73 7bee8

3.1.1 Options discovery using Deep Skill Chaining

Deep Skill Chaining (DSC) [11] modifies the Skill chaining method by introducing policy over options $\pi_{opt} : s_t \in S \to o \in O$, which at decision-making step t determines what option to execute based on the current state s_t. Each option o_i is represented by a tuple $(I_i, \pi_{\theta_i}, \beta_i)$, where the option's policy π_{θ_i} is parameterized. The policy over options π_{opt} starts with only one global option o_g representing primitive actions with its initiation set being the whole state space $I_g = S$ and termination condition being the task's goal state. The algorithm first repeats the global option o_g until it gathers N number of successful trajectories to train a new option policy π_{θ_i}. As a result, long-horizon tasks are broken down into a series of standard RL problems. DSC adds each new option to its repertoire while chaining backward from the task's goal similar to Skill Chaining in Figure 3.1.

The advantage of using policy over options is that each skill in DSC can develop its unique state-abstraction, in contrast to Skill Chaining [10] where each option uses fixed state-abstraction. As a result, DSC was able to outperform non-hierarchical agents and state-of-the-art skill discovery algorithms in five domains within the MuJoCo physics simulator [46] (for example Four Rooms with a Lock and Key). Nevertheless, the current implementation uses a basic exploration method of adding Gaussian noise to actions and could benefit from a more sophisticated exploration strategy aided with observation information.

3.1.2 Flexible Option Learning

Flexible Option Learning (FOL)[12] edits the option policy learning by updating all relevant options simultaneously. Instead of learning one option at a time as described in DSC, FOL sets a hyperparameter to control the probability of updating all options versus only updating the executed option. Updating all relevant options improves performance and sample efficiency in hierarchical RL. Moreover, the algorithm creates temporally extended options with meaningful behavior since its transferred options outperform other hierarchical agents in experiments within the MuJoCo framework [24] (visualized in Figure 3.3).

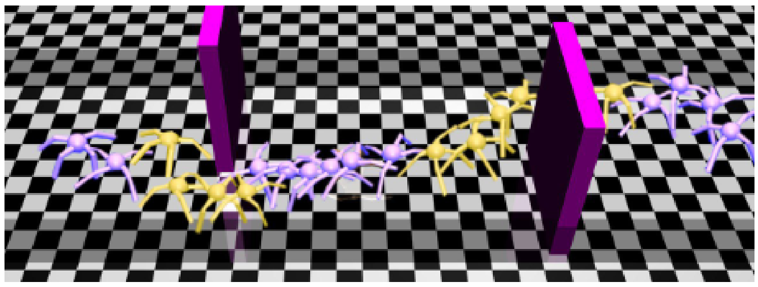


Figure 3.3: Option visualization in AntWalls experiments within the MuJoCo simulator.
Source: https://openreview.net/pdf?id=L5vbEVIePyb

3.2 Frameworks with Parameterized Actions

An alternative approach to divide long-horizon problems into a series of primitive actions are PAMDPs (defined in Section 2.3). The concept of parameterized actions was first proposed in [13] to allow for a distinct set of simple actions with continuous adjustability in RL. To achieve this the proposed method employs a two-level hierarchical policy with the top level being a discrete-action policy $\pi_d(a|s)$ and the lower level consisting of action-parameter policies $\pi_{a_i}(x|s)$. For example in the simplified soccer scoring problem in Figure 3.4, the white agent's action set could be the actions $A_d = (kick, run, shoot\_goal)$ with each action being associated with a set of parameters such as force and target position. The higher policy $\pi_d(a|s)$ decides whether to kick, run, or shoot at the goal depending on the relative positions and speeds of the keeper, the ball, and the agent. If the agent's higher-level policy chooses to shoot at the goal, the corresponding lower-level policy $\pi_{shoot\_goal}(x|s)$ selects the shot target position along the goal line.

The strong performance of the proposed Q-PAMDP [13] in the half-field soccer offense was expanded upon with DRL in [14]. The addition of parameterized policies using deep neural networks laid the groundwork for the use of PAMDPs in high-dimensional long-horizon tasks.

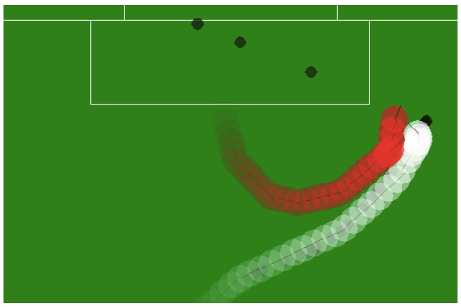


Figure 3.4: A soccer goal episode using a converged Q- PAMDP policy. White agent goes around the goalkeeper to score a goal.
Source: https://www.semanticscholar.org/paper/Reinforcement-Learning-with-Parameterized-Actions-Masson-Ranchod/3f59bce00434b432dfd0b9ab20903aca daefd456

3.2.1 Accelerating Robotic Reinforcement Learning via Parameterized Action Primitives

Robot Action Primitives (RAPS) [19] were used to demonstrate the effectiveness of PAMDPs on several multi-stage, long-horizon manipulation tasks in a kitchen simulation environment [47] visualized in Figure 3.5. In environments mimicking real-world RAPS displayed robust performance, outperforming other methods. With a manually specified library of robot actions, RAPS was able to speed up each episode 32 times compared to using raw robotic action. Moreover, RAPS achieved a 100% success rate transferring policy trained on a 7DOF robot to a 6DOF robot in Robosuite door opening environment.

A disadvantage of the RAPS is the lack of dynamic behavior. When a primitive action begins it does not stop or alter the execution based on environmental feedback. Another problem is that most sets of primitive actions do not guarantee a general solution. To get around this, RAPS incorporates primitive action corresponding to full non-parametrized action space, giving the method ways to fill the lack of suitable primitive action.

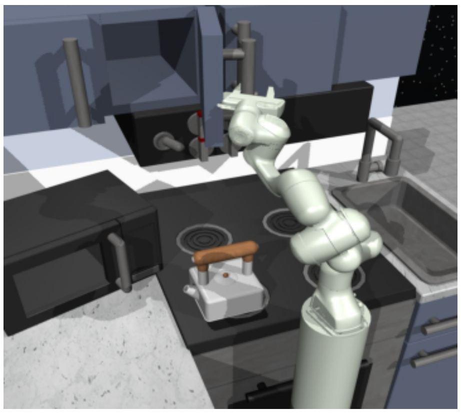


Figure 3.5: Kitchen environment for simulating long-horizon tasks used with RAPS.
Source: https://relay-policy-learning.github.io

3.3 Augmenting Reinforcement Learning with Behavior Primitives

Manipulation Primitive-augmented reinforcement Learning (MAPLE) [5] employs a parametrized set of actions to solve multistep manipulation tasks in Robosuite [20]. Figure 3.6 presents a series of parametrized actions, MAPLE's hierarchical policy employed to solve the tasks. The sequence $(Grasp, Reach, Release)$ resembles a compositional structure that can be further exploited to accelerate learning when transferred to similar tasks. Another advantage of the compositional structure is the resemblance to the human task description.

In the Door opening task, MAPLE achieved a 100% success rate outperforming the Flat, Dual Actor-Critic, and Open Loop baselines, as well as in other complex tasks, such as Peg insertion and Cube stacking. Furthermore, the framework was successfully evaluated on real-world tasks. The Robosutie benchmarking tasks are described in detail in Table 6.1. To achieve this MAPLE algorithm incorporates an additional affordance score to its reward function. This score evaluates the relevance of each discrete action for the given state. We explain the MAPLE algorithm in detail in Section 4.2.

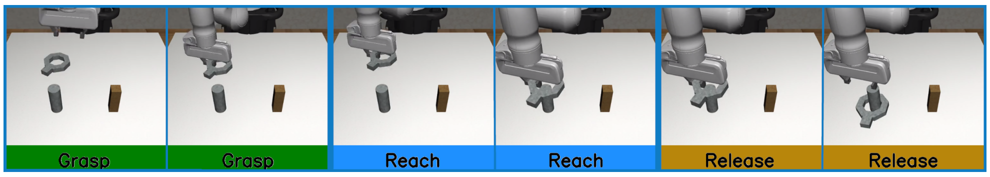


Figure 3.6: MAPLE's solution to the Robosuite tasks of Nut Assembly.

3.4 Comparison of options versus PAMDPs

In the following Table 3.1, we summarize the major differences and similarities in the Options and PAMDPs frameworks.

The main difference between the two approaches is that options learn their own skills, while in PAMDPs, the skills are predefined parametrized actions. Another distinction between the two approaches is that the represented options methods revolve around navigational tasks. On the other hand, the depicted PAMDPs methods are used in solving robot manipulation tasks as can be seen in Table 3.1. Furthermore, the options approaches benefit from high sample efficiency in contrast to the PAMDPs approaches, although the predefined parametrized actions allow for greater insight into the robot's behavior [5].

Feature	Options	Parameterized Action
Policy Type	Policy over option tuples $(I_i, \pi_{\theta_i}, \beta_i)$.	Higher level action policy $\pi_d(a\|s)$ with the lower level action-parameter policies $\pi_{a_i}(x\|s)$.
Skills Form	Algorithm learned option policy neural networks.	Hard coded set of functions.
Transferability of policies	Options transfer across similar tasks.	Higher-level policy transfers in tasks with similar compositional structure.
Learning process	Goal oriented Skill chaining.	Off-policy SAC update.
Exploration	Dependent on a good exploration algorithm.	SACs maximum entropy objective.
Sample Efficiency	High sample efficiency, updating multiple options per observation step.	Low efficiency, a great number of samples is needed to train both actor policies and critic networks.
Domain	MuJoCo[24]	Robosuite[20]
Environments	The ant and The walker MuJoCo environments.	One-arm robot manipulation tasks outlined in Table 6.1.
Skill scalability	Algorithm updated set of options.	Limited by hand-coding each parametrized action function.

Table 3.1: Comparison of Options [10],[11],[12] and Parameterized Action Frameworks [5],[13],[19].

3.5 Methods for including the prior knowledge

Human demonstrations promise to solve the problem of exploration versus exploitation (the challenge of determining when to take exploratory actions rather than following the policy). Recent works use human demonstrations to guide the learning process with the promise of more efficient exploration strategies by guiding the policy to uncover relevant behaviors for task completion [15],[30],[29],[37]. We recognize that there are many different methods to incorporate demonstration data and in the following subsections, we outline some of the common methods.

3.5.1 Learning from demonstrations

Approaches of learning complex behavior directly from human demonstrations involve the following two techniques:

- **Behavioral Cloning (BC)** [41] tries to directly replicate the demonstrated actions.
- **Inverse Reinforcement Learning (IRL)** [42] attempts to infer a reward function (from the demonstrations) to train a policy maximizing this deduced reward.

Although these methods can speed up the initial learning phase, states not included in the training demonstrations limit the robustness of trained policies. Moreover, human training datasets inherently limit the quality of trained policies. Given that human physical skill in long-horizon tasks may be less precise and slower when compared with the abilities of modern robot systems. However, modern methods can surpass the performance in the demonstration dataset and deal with unencountered states by using generalizations from existing demonstrations.

3.5.2 Human-in-the-loop learning

Another method of dealing with unencountered states in demonstrations is interactive Human-Assisted Learning [29],[43]. This approach leverages active human feedback to gather additional demonstrations or domain-specific knowledge to guide the learning process further. Different types of human feedback are used:

- **Binary critique** provides positive or negative feedback (that reflexes the benefit of the last chosen action) as a reward or as policy information[44].
- **Guidance** describes goals in an environment by specifying the objects of interest.
- **Action advice** is a direct human intervention by manipulating or shadowing the robot's believed best action.
- **Query** presents a small set of rewards to gather information about the unknown rewards in the environment.

Although these approaches have shown better results than the traditional non-human-assisted methods, they require a significant amount of human time to give the feedback [30]. New methods make use of generative adversarial networks to minimize the amount of feedback (therefore limiting the time cost) [37].

3.5.3 Reward Shaping

The next approach employs prior knowledge to guide the agent toward desired behavior in certain states of the environment by adapting the reward function. The goal-oriented reward function is supplemented by additional rewards encoded with domain-specific knowledge. Adding the supplementary reward can significantly speed up the learning process, however, the changes to the reward function can result in a completely different goal [45].

3.5.4 Skill-based learning

Hierarchical policies can take advantage of divided action space by pretraining primitive skills from demonstrations. Leveraging these skills in long-horizon tasks improves learning efficiency [28]. Furthermore, pretrained skills are reusable in new unseen tasks.

3.6 Selection of the suitable method for incorporating prior knowledge

We opted for the MAPLE [5] method based on the PAMDP [13] framework defined in Section 2.3. We chose the generalization of skills provided by parametrized action over the options because we believe the predefined structure of actions makes mapping prior knowledge more practical. Moreover, the aforementioned methods in Section 3.2 involve the solution of manipulation tasks, which matches our ambition for solving long-horizon tasks in contrast to the goal-oriented tasks of options frameworks as can be seen in Table 3.1.

We chose the MAPLE method specifically for their set of parametrized actions, which in task solutions resemble human action depiction as illustrated in Figure 3.6. Furthermore, MAPEL demonstrates the ability to transfer the human-like task description across similar tasks. The last determining factor in the selection of the MAPLE method was the provided integration with the established Robosuite simulation framework [20] and the diverse set of benchmarking environments.

Chapter 4

MAPLE and Robosuite framework

The main focus of this chapter is the detailed description of the MAPLE framework and the underlying simulation framework Robosuite.

4.1 Robosuite framework

To provide insight into the MAPLE framework, we first introduce the Robosuite simulation framework[20] which is built upon the MuJoCo physics engine from DeepMind [24]. This framework presents a standardized set of benchmarking tasks for policy development with seven models of robotic manipulators (for instance Franka Emika's Panda, KUKA's LBR iiwa robot systems). The great benefit of this framework is the modularity of simulation and environment APIs which allows for simple expansion to accommodate new tasks and training algorithms. Consequently, Robosuite has been profusely used to develop robot learning algorithms to solve both custom and robosuite's benchmark tasks [5][22]. Furthermore, Robosuite was used to learn task policies in simulated environments, which were later transferred into real-world equivalent environments[5] [29].

MuJoCo is used to initialize the simulation environment (which stores information about the object models and robotic manipulator models). The robot movement is determined at each simulation step by joint torques computed by the Robosuite controller, as seen in the diagram in Figure4.1. Controller input action can be for example end-effector pose or joint configuration. The results of the robotic action can be then observed with sensors with realistic data collection (simulating realistic sensor sampling and delay). Additionally, the Robosuite framework presents insight into the task progress and includes task reward functions for learning purposes.

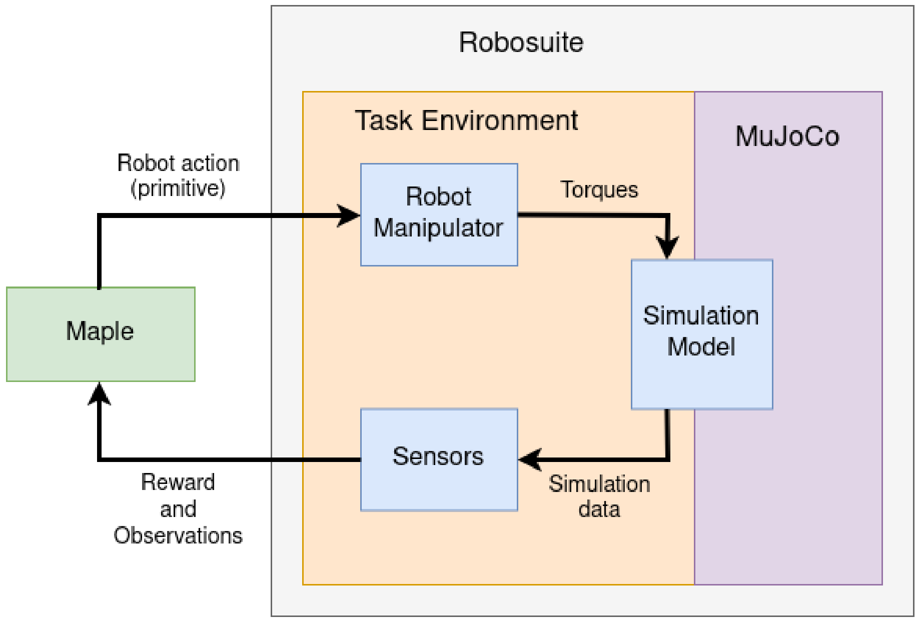


Figure 4.1: Diagram of the connections in the MAPLE [5] framework.

4.2 MAPLE framework

In the following part, we will describe how individual actions are represented within the MAPLE framework. As shown in Figure 4.2, MAPLE's policy architecture is divided hierarchically. The higher-level policy network determines at each decision-making step which one of the behavioral primitives to use. To illustrate the concept we provide the selection of primitives of the higher-level policy network employed to complete the Pick and place benchmark environment in Figure 4.3.

Each primitive consists of a hard-coded, closed-loop function. We list the definitions of all the original MAPLE behavioral primitives as described in [5] and our additional primitives in Table 4.1. The parameter sub-policy in the diagram in Figure 4.2 represents the set of low-level parameter networks where each sub-policy network determines the parameters for one of the primitives. All these parameter policy sub-networks output a universal distribution over parameters that fit all, and the parameters are truncated to the length of the chosen primitive at execution.

The diagram in Figure 4.1 depicts the connection between the MAPLE and Robosuite framework. Observation data is gathered from the simulation using object-based observables which return the corresponding object's positional and rotational data. This data is subsequently processed by the environment function into a vector of parameters relevant to the executions of parametrized primitives in Table 4.1.

For example, in the benchmarking environment Cube stacking (visualized in Figure 4.4) the observation vector is composed of the position and rotation of the red cube, the position, and rotation of the blue cube, vectors representing the relative position of cubes to the gripper and the relative position between the cubes. MAPLE policy networks process the relative environmental composition into robot actions in the form of a 5-DoF end-effector pose (3 degrees of freedom to control the position of the end effector, 1 degree to control the yaw angle, and 1 degree to open and close the gripper.). Afterward, Robosuite invokes the operational space controller to translate the generated robotic action into the joint torques necessary to run the Robot actuators inside the simulation.

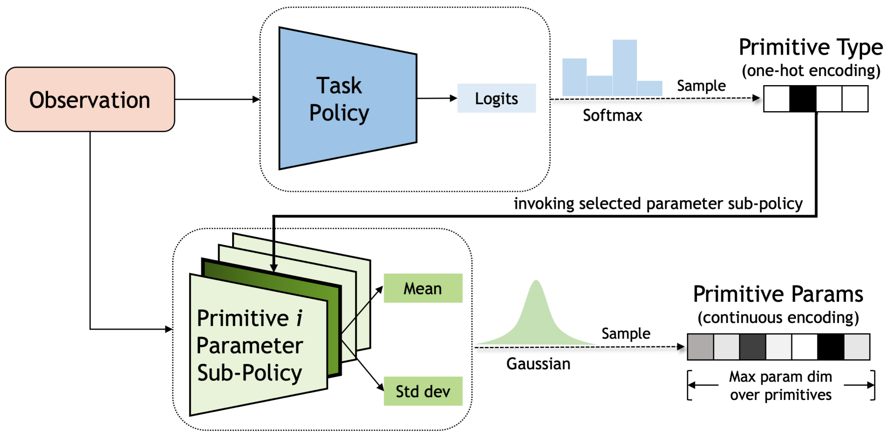


Figure 4.2: Architecture of MAPLE's hierarchical policy.
Source: https://ut-austin-rpl.github.io/maple/

Primitive	Description	Atomic Actions
Reaching	Move end-effector to a target location (x, y, z)	Up to 15
Grasping	Move end-effector to a pre-grasp location (x, y, z) at a yaw angle ψ and close gripper	Up to 20
Pushing	Reaches a starting location (x, y, z) at a yaw angle ψ then move the end-effector by a displacement $(\delta x, \delta y, \delta z)$	Up to 20
Gripper Release	Apply atomic actions to open gripper	4
Atomic	Applies an atomic action $a \in \mathbb{R}^{d_A}$	1

Table 4.1: Set of primitive actions used in MAPLE [5].

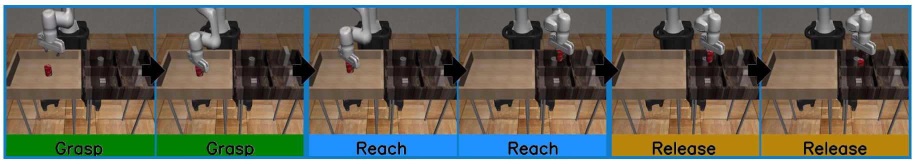


Figure 4.3: Selection of primitive action in the Pick and Place environment.

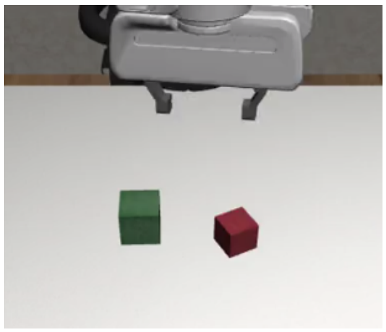


Figure 4.4: Example composition of Cube Stacking environment

4.2.1 MAPLE algorithm

Intending to train hierarchical policy, the MAPLE framework[5] introduced an augmented version of the Deep Reinforcement Learning (DRL) algorithm based on the Soft Actor-Critic (SAC) [26] architecture. The key concept of the algorithm is alternately collecting on-policy transitions within the simulated Robosuite environment and performing off-policy training using data sampled from a replay buffer.

The first of the changes to the SAC algorithm [26],[27] is the replacement of the policy actor network $\pi_\phi(a_o|s)$ with task policy network $\pi_{tsk_\phi}(a \mid s)$ and parameter policy network $\pi_{p_\psi}(x \mid s, a)$, where $s \in S$ is a state in the state space S. Policy parameterization in SAC is possible as long as we can evaluate the policy for any state-action pair [27]. In the original SAC [26] a_o denotes robotic action however in the two networks a represents behavioral primitive and $x \in \mathbb{R}^{d_A}$ is the distribution over parameters, where $d_A = \max_a d_a$ (d_a is the number of parameters for primitive a, vis Table 4.1). Further, the parametrization of actions a_o alters the standard critic Q-network $Q_\theta(s, a_o)$ into MAPLE's critic network $Q_\theta(s, a, x)$. Network parameters are symbolized ϕ and ψ for policy networks respectively and parameters θ for Q-networks. Training of the MAPLE Q network $Q_\theta(s, a, x)$ and the task policy $\pi_{tsk_\phi}(a \mid s)$, the parameter policy $\pi_{p_\psi}(x \mid s, a)$ networks involves the optimization through gradient descent. Learning is therefore done by directly minimizing altered

SAC losses defined:

$$J_Q(\theta) = (Q_\theta(s,a,x) - (r(s,a,x) + \\ \gamma \left(Q_{\bar{\theta}}(s',a',x') - \alpha_{tsk}\log(\pi_{tsk}\phi(a'|s')) - \alpha_p \log(\pi_p\psi(x'|s',a'))\right))^2$$

$$J_{\pi_{tsk}}(\phi) = \mathbb{E}_{a\sim\pi_{tsk}\phi}\left[\alpha_{tsk}\log(\pi_{tsk}\phi(a|s)) - \mathbb{E}_{x\sim\pi_p\psi}Q_\theta(s,a,x)\right]$$

$$J_{\pi_p}(\psi) = \mathbb{E}_{a\sim\pi_{tsk}\phi}\mathbb{E}_{x\sim\pi_p\psi}\left[\alpha_p\log(\pi_p\psi(x|s,a)) - Q_\theta(s,a,x)\right]$$

Coefficients α_p and α_{tsk} control the maximum entropy objective for the task policy and parameter policy. While this optimization assumes continuous primitive parameters, discrete parameters could be also included by reparameterization with the Gumbel-Softmax trick [25].

Algorithm 2 presents a complete summary of the original MAPLE algorithm. The exploration phase is simulated in an episodic manner. Each of these episodes is time and primitive length limited. We include an example of the primitive selection during the episodic exploration of the Stacking benchmark in Figure 4.5. In this exploration, the limit was set to 100 primitives.

The major benefit of exploration through behavioral primitives is the separation of action space into compositional structures. This temporal abstraction means faster exploration of action space in a meaningful way. However, to promote the use of primitives with purpose the framework includes an additional reward function in the form of object affordances. These affordances are expressed by adding an auxiliary affordance score to the reward function. For the atomic and gripper release primitives, an affordance score of 1 is always given, allowing universal applicability. As for the remaining primitives affordance score is calculated as:

$$s_{\text{aff}}(s,x;a) = \max_{p\in P}\left(1 - \tanh\left(\max\left(\|x-p\| - \tau, 0\right)\right)\right), \qquad (4.1)$$

where the keypoint p depends on the primitive a and the current state s, threshold τ for the primitive parameters x is also set depending on the primitive type. These keypoints are the locations of objects' interest for the specific primitive (location to push for the push primitive, the locations of objects to grasp for the grasp primitive, etc).

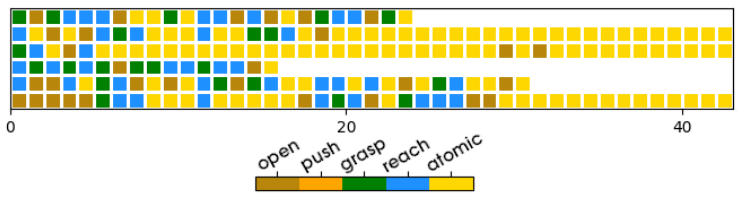


Figure 4.5: MAPLE's six exploration sequences in the Cube stacking task.

Algorithm 2 Manipulation Primitive-augmented reinforcement Learning (MAPLE)

```
1: Initializatize Q-network $Q_\theta(s, a, x)$, task policy $\pi^{\varphi}_{\text{tsk}}(a|s)$, parameter policy
   $\pi^{\psi}_{\text{p}}(x|s, a)$, replay buffer $D$
2: for $i = 1$ to $N$ do                                                ▷ Exploration Phase
3:     for $j = 1$ to $M$ do                                            ▷ Episode
4:         Initialize timer $t \leftarrow 0$
5:         Initialize episode $s_0$
6:         while episode not terminated do
7:             Sample primitive type $a_t$ from task policy $\pi^{\varphi}_{\text{tsk}}(a_t|s_t)$
8:             Sample parameters $x_t$ from parameter policy $\pi^{\psi}_{\text{p}}(x_t|s_t, a_t)$
9:             Execute $a_t$ and $x_t$ in environment
10:            Obtain reward $r_t$ and next state $s_{t+1}$
11:            Add affordance $r_t \leftarrow r_t + \lambda_{\text{saff}}(s_t, x_t; a_t)$
12:            Add transition to replay buffer $D \leftarrow D \cup s_t, a_t, x_t, r_t, s_{t+1}$
13:            Update timer $t \leftarrow t + 1$
14:        end while
15:    end for
16:    for $k = 1$ to $K$ do                                            ▷ Training Phase
17:        Update Q network: $\theta \leftarrow \theta - \lambda_{\text{lr}} \nabla_\theta J_Q(\theta)$
18:        Update task policy: $\varphi \leftarrow \varphi - \lambda_{\text{lr}} \nabla_\varphi J_{\pi_{\text{tsk}}}(\varphi)$
19:        Update parameter policy: $\psi \leftarrow \psi - \lambda_{\text{lr}} \nabla_\psi J_{\pi_{\text{p}}}(\psi)$
20:    end for
21: end for
```

Chapter 5

The proposed extension of the MAPLE framework

In this chapter, we describe the proposed extension of the MAPLE framework (see Section 4.2) by incorporating prior knowledge from human demonstrations.

5.1 Leveraging prior human knowledge for task completion

In many instances, we can rely on human demonstrations to guide our understanding of task execution [30]. These demonstrations can provide the optimal set of primitive actions, as well as reveal the sequence necessary to achieve the task objective. When available, taking advantage of this insight can improve the initial search in RL. Instead of randomly exploring, the human-informed sequences can direct the order of actions necessary for effective learning and task completion.

5.1.1 Type of prior knowledge

As stated earlier in Section 3.5 models benefiting from temporal abstraction have demonstrated improved performance in learning from human datasets. The temporal abstraction that arises from the hierarchical policy structure in MAPLE mirrors a human-like task description. As can be seen in Figure 4.3 the MAPLE's selection of primitive actions Grasp, Reach, and Release for the aptly named Pick and Place environment. This similarity to the semantic solution of the task objective is only possible due to the particular definition of each primitive (see Table 4.1 for primitives descriptions). Moreover, MAPLE framework [5] demonstrated faster learning of parameter sub-policies with a fixed sequence of primitives. Considering the advantages of temporal abstraction we augment the MAPLE framework with human sequence description.

To ensure that our demonstration dataset is both compact and easy to use within the existing MAPLE framework, knowledge prior(human demonstration data) is in the form of a sequence description as can be seen in the mapping diagram in Figure 5.1. Each primitive sequence is written in a manner that would most efficiently lead to the completion of the task. For example, the primitive sequence for the Pick and Place task would be $(Grasp, Reach, Release)$.

5.1.2 Prior knowledge mapping

In our proposed extension, we make use of a mapping function to mediate the semantic description to the MAPLE framework as can be seen in Figure 5.2. Description data is one-to-one mapped from the human-constructed semantic solution Z to the set of corresponding primitive robot actions O_p. For example the mapping function $f : Z \rightarrow O_p$ maps the suggestion of using grasping action to the primitive of grasping. The mapping function directly applies the semantic significance of each primitive from Table 4.1. The suggested primitive represents the correct primitive function and connected parameter sub-policy in the decision-making step. The observed primitives are mapped into a matrix, where each column represents one primitive suggestion from the description sequence and rows represent the type of primitive (see Figure 5.1). Each column vector has the same length as the output of MAPLE's task policy (number of defined primitives). Even though we recognize the ineptness of direct mapping (for example for mapping hand trajectories), the vector representation allows for further replacement of our mapping function with more sophisticated methods. This improvement could be in the form of a supplementary neural network for hand trajectory and gesture mapping. Moreover, using neural networks could map any other human descriptive guidance from human speech. These mapping methods could replace ours by simply replacing the mapping function with the appropriate neural network within our outline in Figure 5.2.

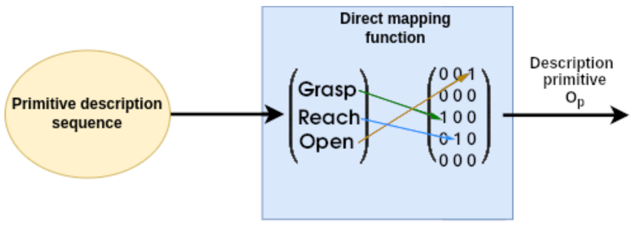


Figure 5.1: Direct mapping function.

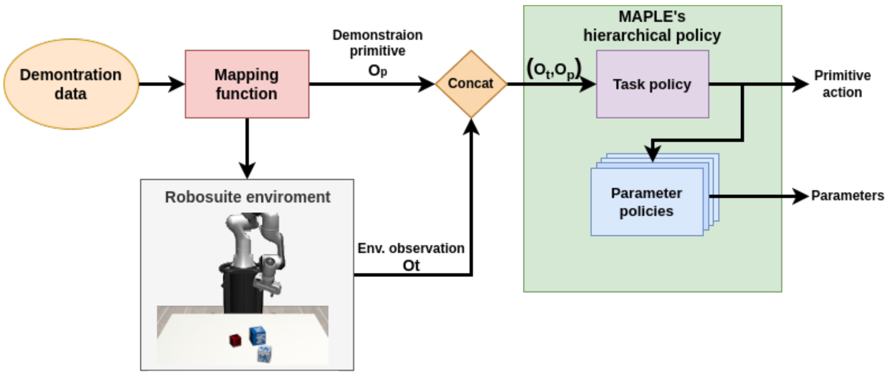


Figure 5.2: Proposed MAPLE framework incorporating human demonstrations.

5.2 Augmenting MAPLE architecture with prior knowledge

Although there exist several methods for integrating prior knowledge, as discussed in Section 3.5, our approach involves incorporating sequence information in contrast to directly mimicking actions from observations. Consequently, we do not employ behavioral cloning. Additionally, the SAC's off-policy learning introduces constraints on the in-loop methods that rely on probing the user for additional information during the learning. Maple could benefit from Demonstration-Guided RL[15], where the algorithm learns short-horizon skills from offline datasets and then leverages this knowledge in new tasks[16]. However, this method parallels pretraining the primitive parameter sub-policy networks.

Acknowledging these factors, the MAPLE framework can benefit from reward shaping and the incorporation of prior knowledge into the policy model architecture. These strategies can be easily incorporated into the preexisting SAC-based MAPLE framework. In the following subsections, we describe the two extensions. The overall extended architecture is visualized in Figure 5.2 and pseudocode is in the Algorithm 3.

5.2.1 Reward shaping

Reward shaping involves modifying the reward function based on the provided primitive sequence to guide the learning toward the desired behavior. We adjust the reward function by adding an additional reward to behavior mimicking demonstration primitive. Task policy network outputs softmax-normalized vector x which decides what primitive action will be enacted. We use Cosine similarity multiplied by scaling constant α_o to compute additional observation reward $\lambda_o(o_p, x)$ from the primitive type vector x and description vector o_p:

$$\lambda_o = \alpha_o * (x \cdot o_p)/(||x|| * ||o_p||). \tag{5.1}$$

When the task policy network grants a probability value to the same primitive as the one from the observed primitive, it gains an overlap bonus reward.

Therefore nudging the policy to pick the same primitive as the one from observations.

5.2.2 Incorporating prior knowledge into the policy network

We additionally incorporate prior knowledge into the task policy neural network input. We concatenate the observation vector from the Robosuite environment with the mapped demonstration data. Thus supplying the compositional knowledge directly to the MAPLE policy as described in Figure 5.2. supplying the compositional knowledge directly to the MAPLE policy as In the visualized task in Figure 5.3, the first step of the decision-making process involves the task policy neural network receiving a vector composed of $[observed\,scene, grasp]$. In the following step, the task policy neural network's input is the vector containing $[observed\,scene, reach]$. While the task policy network is not required to use the prior knowledge, imitating it benefits the network. Therefore, leading to learning advantages.

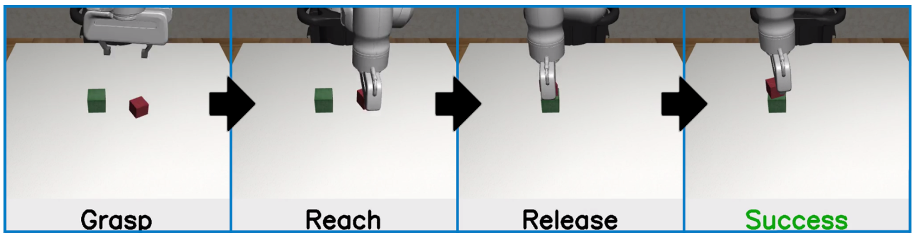


Figure 5.3: Example of primitive sequence in the Cube Stacking environment.

Algorithm 3 Extended MAPLE (extensions marked in blue)

```
 1: Input: Demonstration data $z$
 2: Initializatize Q-network $Q_\theta(s, a, x)$, task policy $\pi^{\varphi}_{\text{tsk}}(a|s)$, parameter policy
    $\pi^{\psi}_{\text{p}}(x|s, a)$, replay buffer $D$
 3: Get description primitives $O_p$ by mapping $O_p \leftarrow f(z)$
 4: for $i = 1$ to $N$ do                                              ▷ Exploration Phase
 5:     for $j = 1$ to $M$ do                                                   ▷ Episode
 6:         Initialize timer $t \leftarrow 0$
 7:         while episode not terminated do
 8:             Sample primitive $a_t$ from $\pi^{\varphi}_{\text{tsk}}$ and parameters $x_t$ from $\pi^{\psi}_{\text{p}}$
 9:             Execute $a_t$ and $x_t$ to obtain reward $r_t$ and next state $s_{t+1}$
10:             Get demonstration primitive vector $o_p \leftarrow O_p(t)$
11:             Add primitive overlap reward $r_t \leftarrow r_t + \lambda_{\text{o}}(a_t, o_p)$
12:             Add transition to replay buffer $D \leftarrow D \cup \{s_t, a_t, x_t, r_t, s_{t+1}, o_p\}$
13:             Update timer $t \leftarrow t + 1$
14:         end while
15:     end for
16:     for $k = 1$ to $K$ do                                              ▷ Training Phase
17:         Update Q network: $\theta \leftarrow \theta - \lambda_{\text{lr}} \nabla_\theta J_Q(\theta)$
18:         Update task policy: $\varphi \leftarrow \varphi - \lambda_{\text{lr}} \nabla_\varphi J_{\pi_{\text{tsk}}}(\varphi)$
19:         Update parameter policy: $\psi \leftarrow \psi - \lambda_{\text{lr}} \nabla_\psi J_{\pi_{\text{p}}}(\psi)$
20:     end for
21: end for
```

Chapter 6

Experimental setup

This section aims to provide information for constructing new environments within the MAPLE framework and present our methodology for creating reward functions and new behavioral primitives. Furthermore, this section describes employed benchmarking environments and the hyperparameters used in training.

An overview of all the employed benchmarking environments is available in Table 6.1, and all hyperparameters used during the training of policy and SAC networks across all benchmark environments are available in Table 6.2.

6.1 Robosuite environments

In our experimentation with the MAPLE algorithm, we train new policies in simulated environments constructed using Robosuite [20]. Every Robosuite task includes a robot model, task-specific items, and an arena (example environment rendered in Figure 6.1). All objects use the underlying MuJoCo physics engine to simulate realistic physics (including robot dynamics, contact mechanics, and friction). The following sections detail the Python implementation of the Environment class, object classes, and reward functions necessary for new task creation.

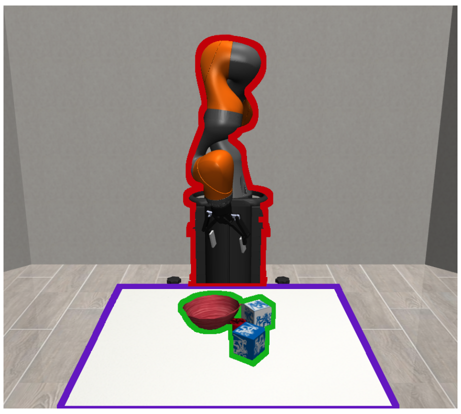


Figure 6.1: Render of Robosuite environment.

6.1.1 Environment class

The class diagram in Figure 6.3 outlines the dependency structure of the Robosuite's environment. Complete class diagram of all Robosuite classes available with code in [48]. Moreover, we highlight four environment methods that are important for the creation of new tasks and for understanding the Robosuite framework.

1. **The load_model method** sets up the MuJoCo instance by loading the appropriate robot, arena, and object XML model files into the simulation. The method loads specific attributes, textures, and geometry meshes of task items and defines the initial task composition.

2. **The reset method** sets the environment at the start of each episode with a valid, non-colliding placement of all objects in the scene.

3. **The get_observation method** is used to get all the necessary information about the current state. In the Maple framework observation includes the task objects positions and rotations and relative positions and angles to the gripper. For example, observations in the peg insertion benchmark comprise the position and rotation of the peg, the relative positions between the gripper and the peg, and the relative position and angle between the hole and the peg as demonstrated in Figure 6.2 .

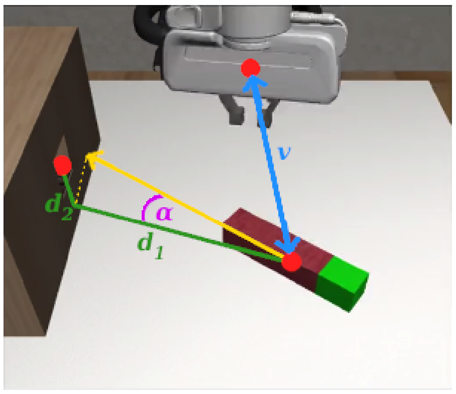


Figure 6.2: Visualization of observation data passed from the simulated peg insertion task environment. The relative position between the gripper and the peg v, as well as the peg orientation denoted by angle α, and the relative position of the hole and the peg denoted by (d_1, d_2).

4. **The reward method** function is an essential component of RL, the Maple framework incorporates staged rewards to compute the reward of the current state of the environment. The staged reward provides the agent with feedback even when the task is not successfully completed. For instance, in the peg insertion benchmark, the reward function consists of a reward for grasping the peg, a reward for the alignment of the peg with the hole, and a reward for successful insertion.

5. **The check_success method** is a binary task completion check. This method defines the goal of the task by implementing constraints that need to be accomplished. For example, in the cube stacking benchmark, success is achieved when the smaller cube is placed on top of the other. However, since the simulation runs for a fixed number of timesteps, this method also checks for the stability of the discovered policy. If the smaller cube is placed on the edge of the base cube, the stacked tower can fall over invalidating the temporary success of the task.

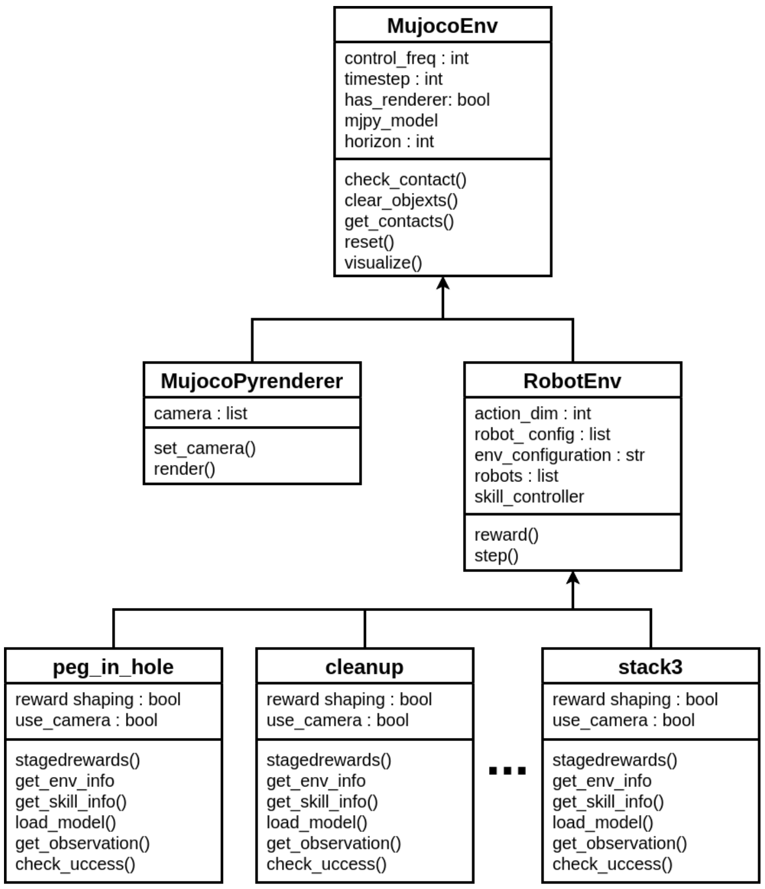


Figure 6.3: Class diagram of Robosuite's environment implementation.

6.1.2 Object classes

Each task contains an arena model, a manipulator model, and a list of object model instances. All three subclasses represent a key component of the environment (the robot, the workspace, and the task items) as rendered in the example environment in Figure 6.1. This modularization allows for a simple addition of new tasks by including the three subclasses in the load_model method of the Environment class. We provide a brief description of each subclass below:

1. **The Robot class** represents the real-life robot manipulator defined by

a corresponding robot arm XML and gripper XML. These files define manipulator characteristics, such as the degrees of freedom and the control method. The robot model used in all of our tests was the same as in [5] the 7-DoF Panda. The control method is Operational Space Control guided by the behavioral primitive close loop controls.

2. **The Arena class** defines the physical space where the robot operates and performs tasks. The workspace is defined depending on the task. In the pick and place benchmark, the Arena class defines the size and height of the table and bins for the items. While in the door opening task, the Arena is composed of a door and a handle.

3. **The MujocoObjec class** represents the objects, the robots manipulator interacts with in the tasks. These can be defined as simple geometric shapes or complex models. The objects have their physical properties, such as dimension, mass, and friction.

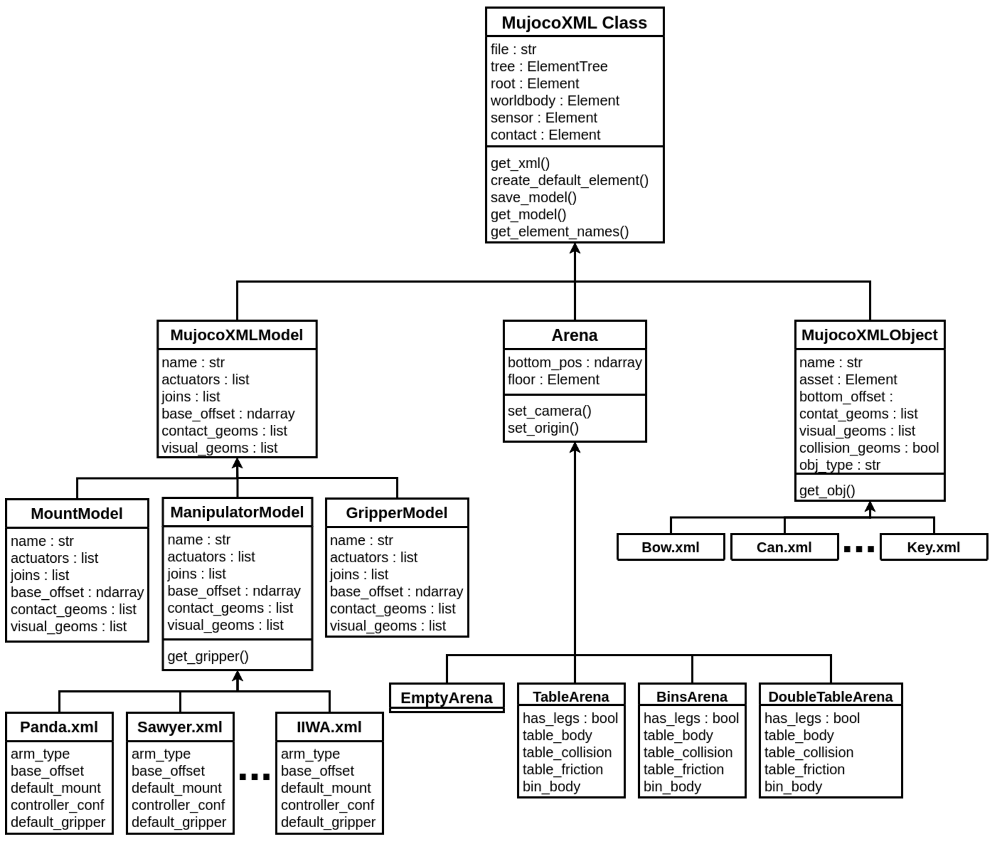


Figure 6.4: Class diagram of Robosuite's object implementation.

6.1.3 MAPLE-specific reward and observation methods

MAPLE's algorithm implementation makes changes to Robosuite environments by implementing additional methods below:

1. **The staged_rewards methods** are used with affordance to nudge the hierarchical policy to use parametrized actions instead of atomic actions, speeding up learning. Each task has its own set of staged rewards. For example, in the peg insertion task one staged reward is added for peg alignment and another for grasping and lifting the peg in Figure 6.2. Staged rewards are all added up together and then normalized.

2. **The get_skill_info method** returns environment observation for keypoint computation of the affordance score. In the case of the before-mentioned peg task in Figure 6.2, the observation contains points of interest, the grasping position of the green tip, and the hole position.

3. **The get_aff_reward method** computes additional rewards from observed key points as outlined in Section 4.2.1.

6.1.4 MAPLE's parametrized action implementation

MAPLE employs a **SkillController class** to enact a parametrized action using Robosuite's inbuilt operational space control (OSC). To execute the parametrized action (skill) the **SkillController** repeatedly calls **the get_pos_ac()** method of the chosen skill until **the is_success()** method returns a True value. Each parametrized action is defined in its skill subclass as a closed loop of hard-coded end effector states computed from input parameters (parameters generated by its primitive parameter sub-policy). In Figure 6.5, we provide an example of a parametrized action instance.

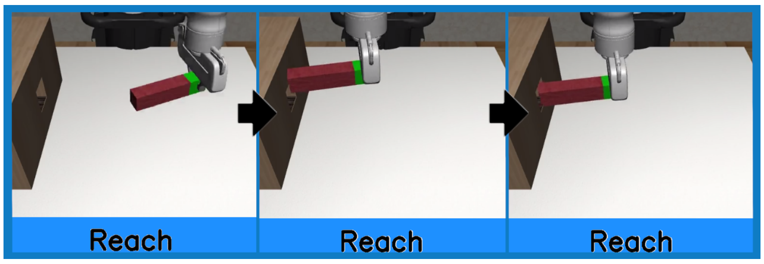


Figure 6.5: lustration of the reach parametrized action in the peg insertion task. After the task policy has elected to use the reach primitive, the corresponding primitive sub-policy network has chosen the target position (x, y, z) near the hole. Subsequently, the ReachSkill passes a series of positions to the OSC controller to move the gripper to (x, y, z).

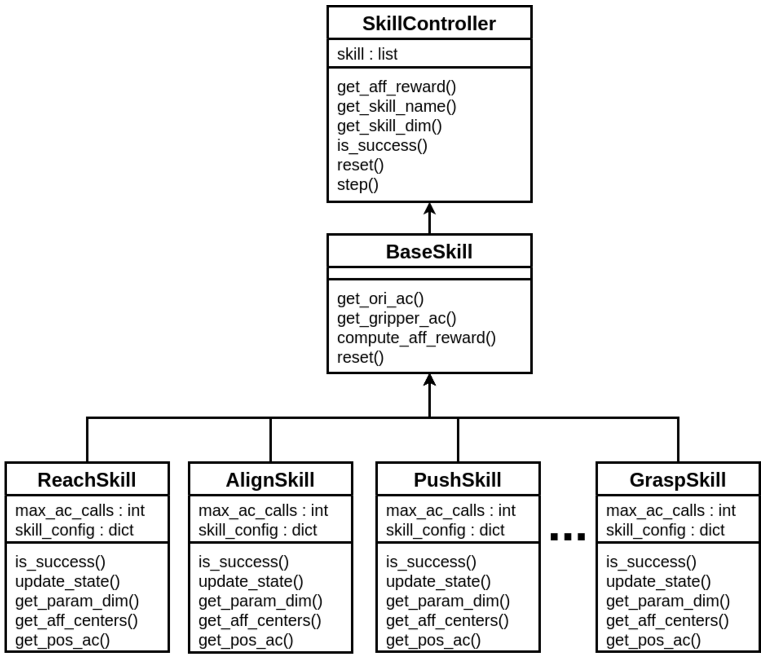


Figure 6.6: Class diagram of MAPLE's parametrized action implementation.

6.2 Our extensions to the Robosuite environments

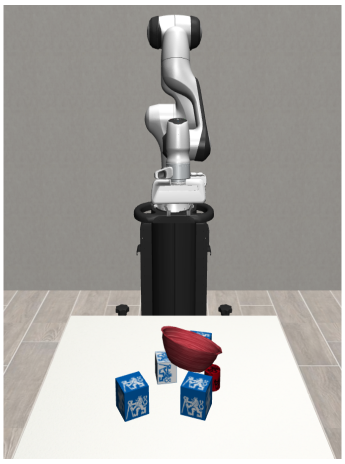
As a part of our experiments, we make the following additions to the MAPLE framework, which include:

1. Expanding Robosuite benchmarks by introducing "clutter."
2. Introducing a new reward function from prior knowledge.
3. Creating new tasks with multiple task-relevant objects.

6.2.1 Addition of YCB objects

To represent realistic household items and tools, we decided to use the YCB standardized benchmark models [23]. From the YCB objects dataset, we have imported five objects to the Maple framework to use them to generate artificial clutter. These objects are the Key, the Bowl, the Clamp, the Scissors, and the Plate in image 6.7b. The four selected common items were specifically included for the diversity they provide in the evaluation of the maple framework. Each of the objects' geometries amplifies the complexity of the benchmarking task. For example, task objects are spawned in the Bowl, the Key can block grasping action, and the plate can act as an uneven moveable base for stacking. An example of a high-clutter environment is depicted in Figure 6.7a.

All of the YCB model meshes are processed using Meshlab [35]. This program allowed us to simplify the detailed geometric meshes into more generalized models with Laplace smoothing and quadric edge collapse decimation as illustrated in Figure 6.8.


(a) : Render illustrates the stack. Forironment with multiple objects forming obstacles for task completion.

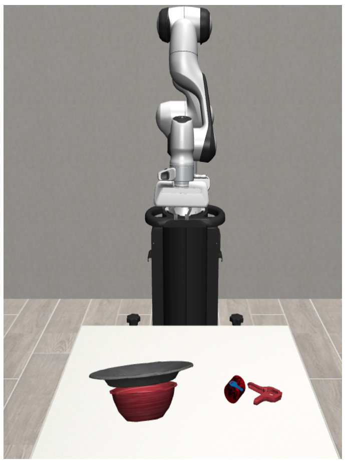
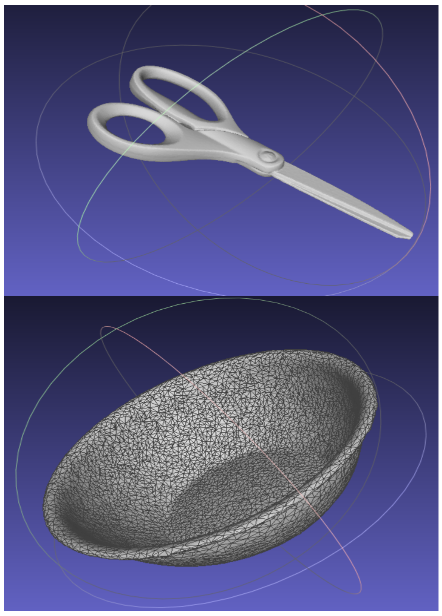

(b) : Render of imported YCB objects

Figure 6.7: Example environments containing YCB objects.


(a) : Original high vertex density meshes.

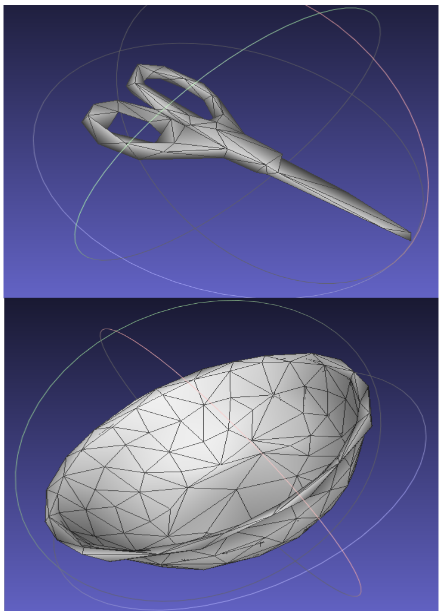

(b) : Our simplified YCB object meshes.

Figure 6.8: YCB's bowl and scissors laser scans before and after mesh simplification.

6.2.2 Reward shaping

Our implementation of prior knowledge is outlined in Section 5.2.1. The additional observation reward is implemented inside the **SkillController class** using Pytorch's **torch.nn.CosineSimilarity** function. The primitive sequence is loaded and mapped into the environment as a numpy array. The observation reward is computed after each environmental step similarly to the affordance score.

6.2.3 Creation of new benchmarks

A new task is created by implementing the environment class as depicted in the class diagram in Figure 6.3. MAPLE environment creation is composed of the following steps:

1. Implementing the **load_model** method which defies the composition and objects used.
2. Defining the **reward** functions and task success conditionions.
3. Defining the **get_observation** method which serves as the input to MAPLE's hierarchical policy.
4. Configuring the **get_skill_info** method for affordance score.

A detailed description of how to implement new tasks within the MAPLE framework is provided with code in [48].

We created two new manipulation tasks to test the MAPLE framework [5] in multi-stage long-horizon tasks:

1. **The cube sorting environment** provides the challenge of sorting three cubes of different sizes. The task's goal is to sort the cubes on the grey part of the table in ascending order, with the smallest on the left and the largest on the right.
2. **The triple stacking environment** benchmark is a more complex scenario of cube stacking. We increased the number of cubes needed stack to complete the task by adding an extra cube and augmenting the reward functions and cube positions.

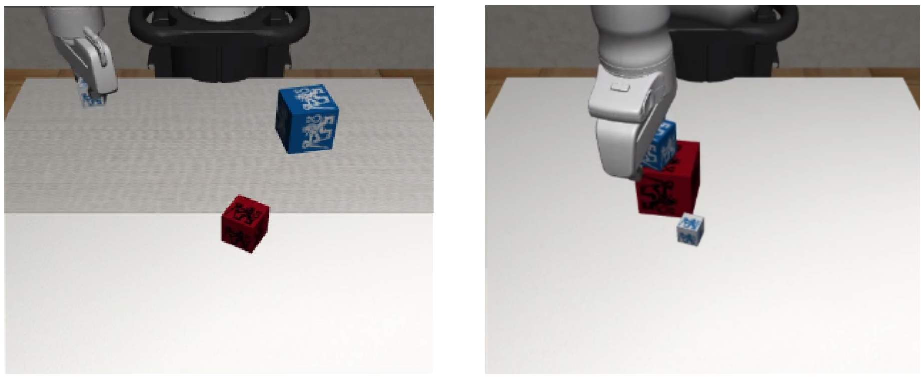
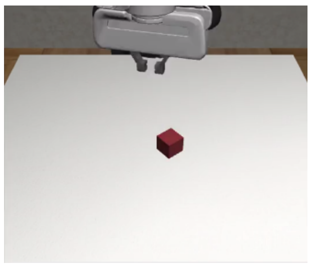
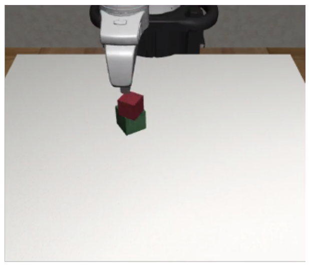
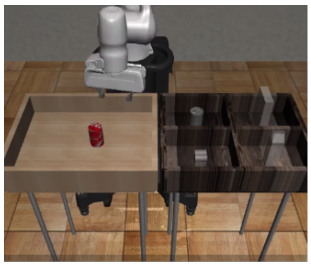
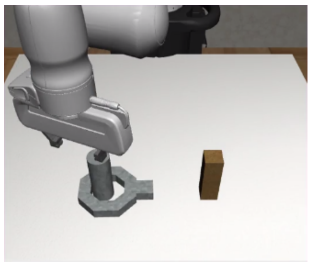
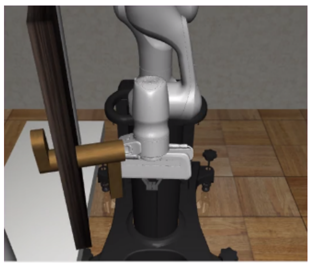

(a) : Cube sorting environment **(b) :** Triple cube stacking task.

Figure 6.9: Renders of our Robosuite manipulation tasks.

6.3 Benchmarking environments

We run our experiments on ten different robot manipulation tasks. Eight of these tasks are part of the MAPLE framework [5]. Additional two tasks, triple cube stack, and cube sort, were created to further evaluate MAPLE's performance. In Table 6.1, we provide a brief description of each task.

Task: env_lift (Block Lifting) **Scene Description:** A cube is placed on the tabletop in front of a single robot arm. **Goal:** The robot arm must lift the cube above a certain height. **Start State Distribution:** The cube location is randomized at the beginning of each episode.	
Task: env_stack (Block Stacking) **Scene Description:** Two cubes are placed on the tabletop in front of a single robot arm. **Goal:** The robot must place one cube on top of the other cube. **Start State Distribution:** The cube locations are randomized.	
Task: env_pnp (Pick and Place) **Scene Description:** Four objects are placed in a bin in front of a single robot arm. There are four containers next to the bin. **Goal:** The robot must place each object into its corresponding container. **Start State Distribution:** The object locations are randomized.	
Task: env_nut_assembly (Nut Assembly) **Scene Description:** One square peg and one round peg are mounted on the tabletop, and two nuts (one square and one round) are placed on the table in front of a single robot arm. **Goal:** The robot must fit the square nut onto the square peg and the round nut onto the round peg. **Start State Distribution:** The nut locations are randomized at the beginning of each episode.	
Task: env_door (Door Opening) **Scene Description:** A door with a handle is mounted in front of a single robot arm. **Goal:** The robot arm must turn the handle and open the door. **Start State Distribution:** The door location is randomized at the beginning of each episode.	

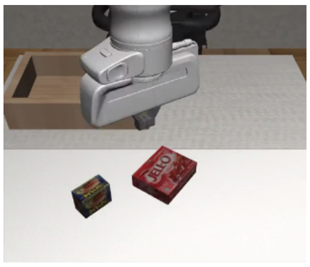
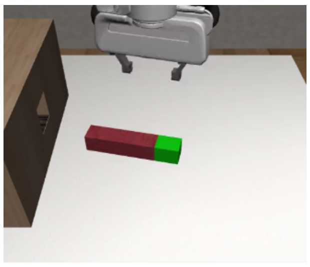
Task: env_wipe (Table Wiping) **Scene Description:** A table with a whiteboard surface and dark markings is placed in front of a single robot arm, which has an eraser mounted. **Goal:** The robot arm must learn to wipe the whiteboard surface and clean all of the markings. **Start State Distribution:** The whiteboard markings are randomized at the beginning of each episode.	
Task: env_cleanup (Clean Up) **Scene Description:** Spam box and jello box are placed on the tabletop in front of the single robot arm. **Goal:** The robot arm must learn to place the spam box into the storage bin and move the jello onto the grey part of tabletop next to the bin. **Start State Distribution:** Locations of both boxes are randomized.	
Task: env_peg_in_hole (Peg Insertion) **Scene Description:** Peg is placed on the tabletop in front of a single robot arm next to a box with a square hole. **Goal:** The robot arm must lift the peg and insert it a certain distance into the box. **Start State Distribution:** The peg location is randomized at the beginning of each episode.	
Task: env_stack3 (Triple Stack)[NEW TASK] **Scene Description:** Three cubes, each of a different size, are placed on the tabletop in front of a single robot arm. **Goal:** The robot must learn to stack all three cubes on top of each other. **Start State Distribution:** The cubes locations are randomized at the beginning of each episode.	
Task: env_cube_sort(Cube Sort) [NEW TASK] **Scene Description:** The tabletop is divided into a grey sorting part and a white part, on which there are placed three different cubes. **Goal:** The robot must sort all three cubes by size onto the grey part. **Start State Distribution:** The cubes' locations are randomized within the white part of the tabletop.	

Table 6.1: Descriptions of all environments used in our experiments (Triple stack and Cube sort are newly introduced tasks).

6.4 Hyperparameters used in training new Maple policies

Hyperparameter selection in RL algorithms significantly impacts performance, stability, and convergence of the learning process [38],[39]. Therefore to reliably replicate results achieved with Maple [5], we retained the training hyperparameters of the SAC architecture see Table 6.2. In all conducted experiments with Maple, the optimizer employed was Adam [40], SAC's automatic entropy tuning was enabled, and the sizes of both critic networks were kept the same. Moreover, the Maple hierarchical structure makes use of two entropy targets, the entropy target for the task policy (primitive selection network) had an increased entropy target for the first 200 epochs to $0.97 \cdot \log(k)$, where k is the number of primitives.

Maple-specific hyperparameters used within our tests are listed in Table 6.3. The hyperparameters employed with our extension reflect the lesser necessity for exploration as we provide foundational knowledge of the task structure to the environment. Consequently, we skip the initial high entropy phase in training our extended Maple policies. Furthermore, we limit the number of primitives in each episode to 100. However, to allow a direct comparison of learning speeds across the two methods, we retained the number of primitives per exploration phase.

Tuning hyperparameters allows for deeper insights into the algorithm's behavior. In Section 7.1, we present findings on how the hidden size and reward influence policy training. All experiments detailed in chapter 7 were carried out on Google's Colab research infrastructure with Tesla K40 GPU and a cluster with NVIDIA GeForce 1080 Ti.

Parameter	**Value**
Optimizer	Adam [40]
Batch Size	1024
Policy Learning Rate	3×10^{-5}
Q-function Learning Ratef	3×10^{-5}
Discount factor	0.95
Replay Buffer Size	10^6
Number of Hidden Layers	2
Hidden size	256
Target Task Policy Entropy	$0.50 \cdot \log(k)$, k is number of primitives
Target Parameter Policy Entropy	$-\max_a d_a$
Activation Nonlinearity	ReLU
Target Smoothing Coefficient (τ)	0.001
Target Update Interval	1

Table 6.2: SAC Parameter Values.

Parameter	**Maples Value**	**Extension Value**
First exploration phase length	30000 primitives	20000 primitives
Training steps per epoch	1000	1000
Number of primitives per exploration phase	3000	3000
Max episode length	150 primitives	100 primitives
Reward scale	5	5
Affordance scale	3	Not in use
Observation scale α_o	Not in use	4

Table 6.3: Maple Parameter Values.

Chapter 7

Results

In this chapter, we present results attained from testing the MAPLE algorithm's sensitivity to hyperparameter, reward, and environment changes and provide results achieved with the changes made to the MAPLE framework outlined in Section 6.2.

7.1 Evaluation of MAPLE sensitivity

To evaluate the robustness of the MAPLE algorithm, we have run the following experiments:

1. Impact of hidden layer size on learning performance.
2. Effects of staged reward multipliers on the success of learning new policies.
3. Evaluation of policies in cluttered environments.

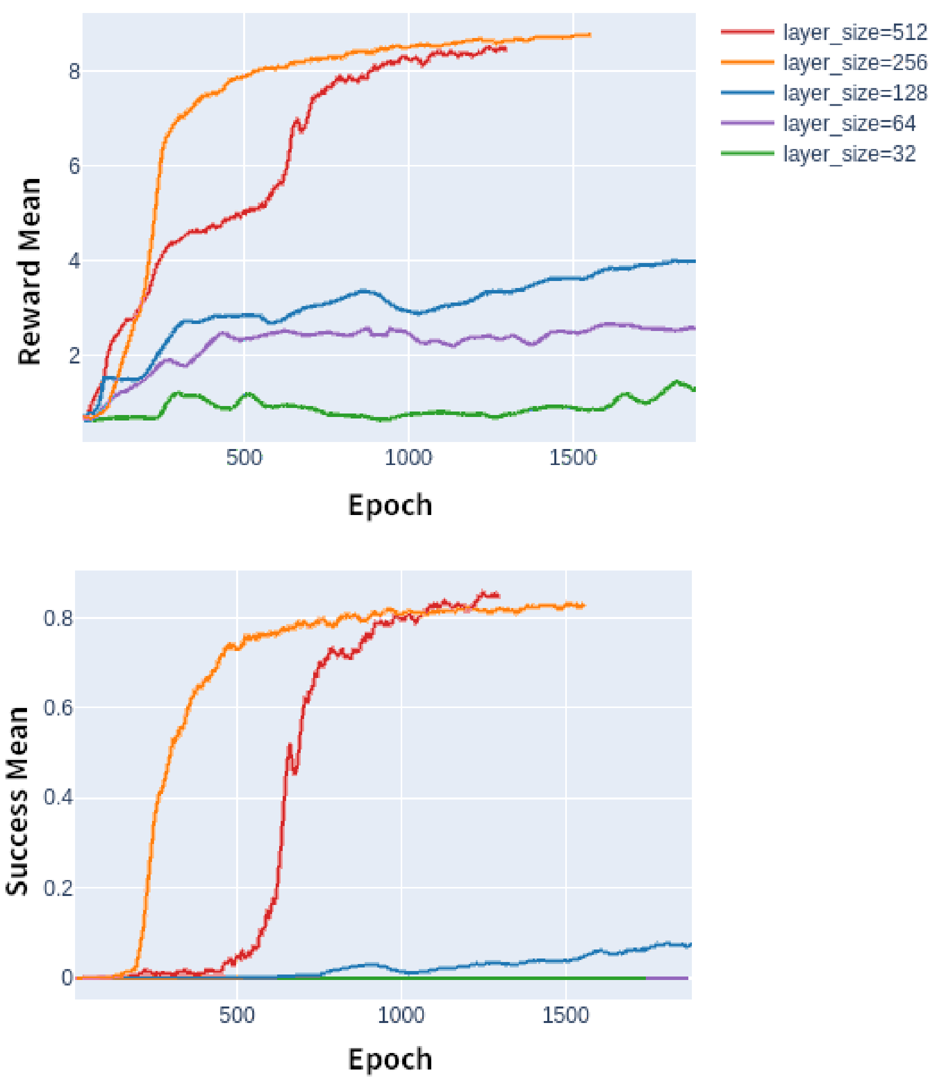
7.1.1 Hidden layers sizes

The graphs in Figure 7.1, display the effects of hidden layer size in learning new MAPLE policies in the Cleaning up environment, the Peg insertion environment, and the Cube stacking environment.

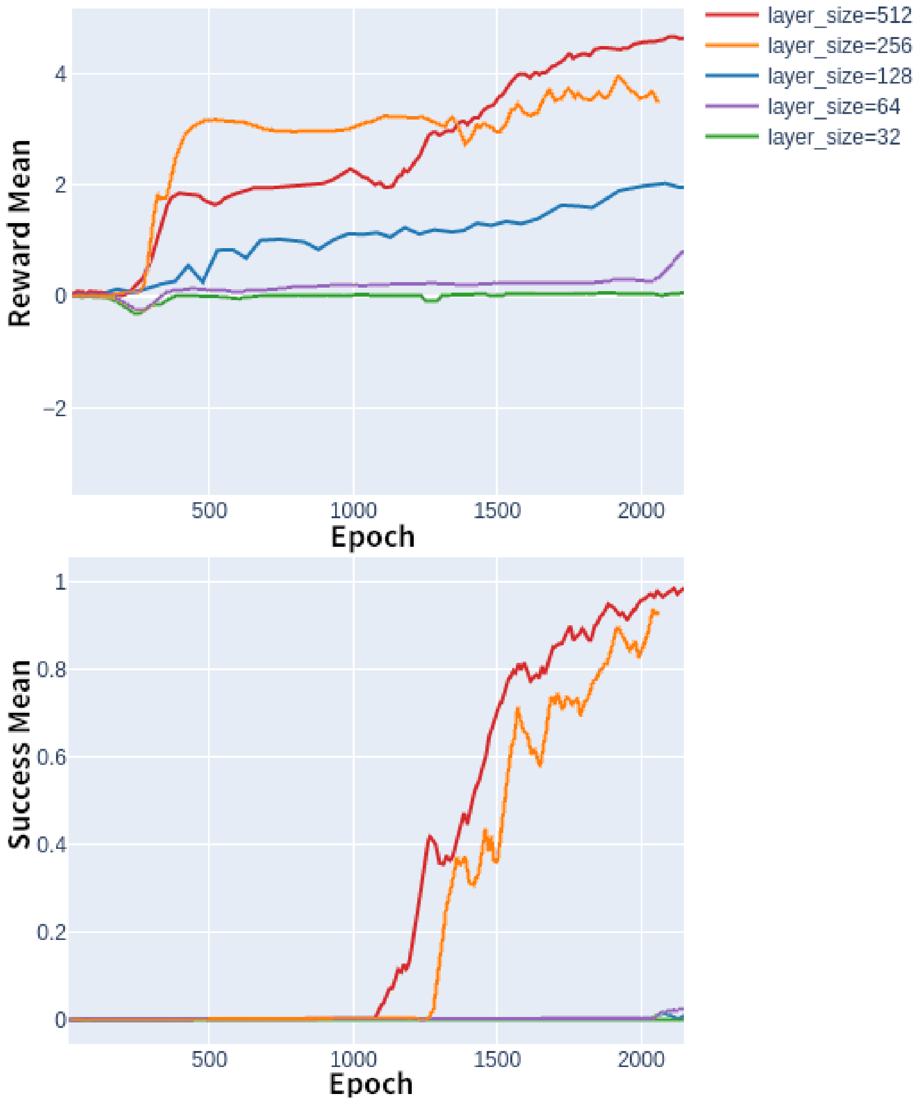
Policies trained with smaller hidden layer sizes of 64 and 32 failed to solve all three tasks see Table 7.1. Policies with a higher number of neurons generally outperformed smaller ones. The baseline size of 256 achieved comparatively the best learning speed-to-success ratio. However, in the complex task of pin insertion, the policy with the highest number of neurons 512 outperformed the baseline in learning speed. All experiments are averaged over 5 seeds. (Each set of runs in one environment was done on a single platform to eliminate the effects of cross-platform on the Robosuite simulator.)

Hidden Size	32	64	128	256	512
Clean up	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 12.8	83.0 ± 4.3	86.0 ± 5.6
Peg insert	0.0 ± 0.0	4.0 ± 4.8	2.0 ± 3.0	98.0 ± 2.0	93.0 ± 3.4
Cube stack	0.0 ± 0.0	0.5 ± 1.8	1.0 ± 0.9	99.0 ± 1.2	100.0 ± 2.0

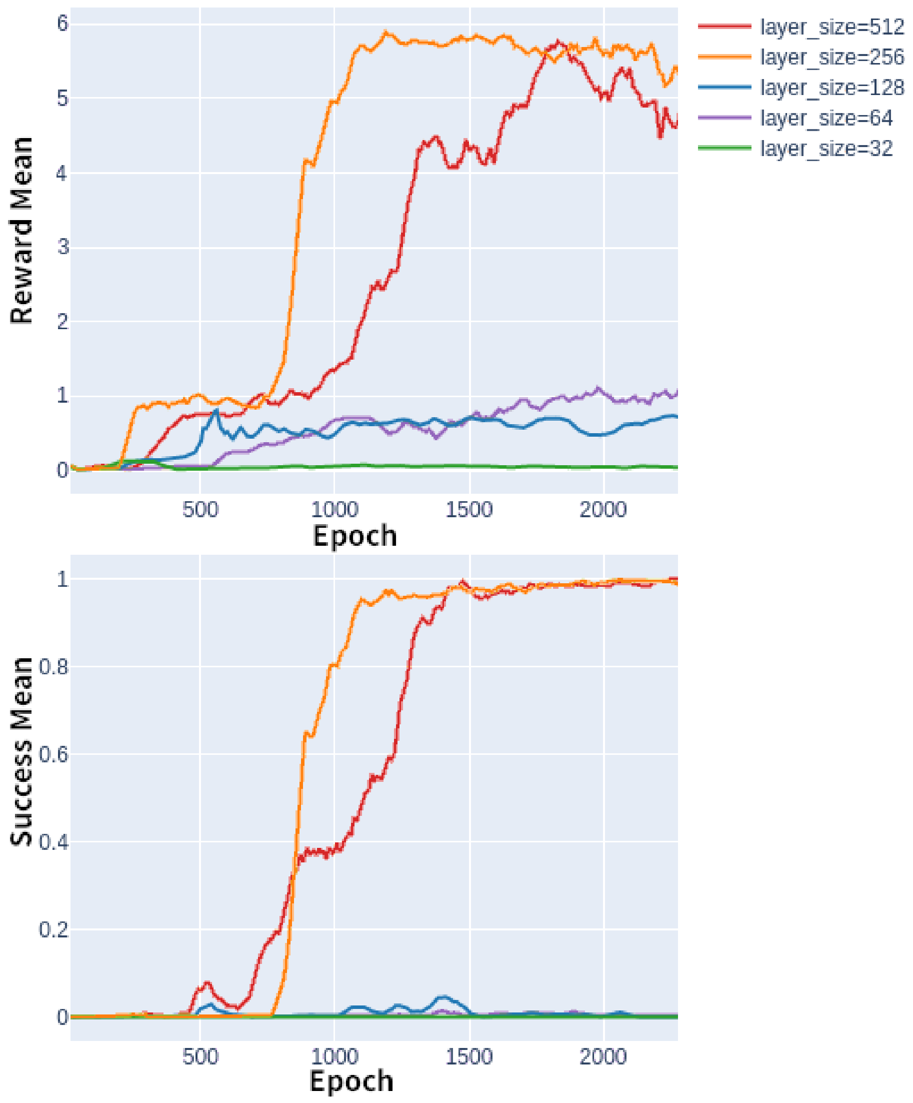
Table 7.1: Final Hidden size Success Rates (%)



(a) : Clean up



(b) : Peg Insertion



(c) : Cube Stacking

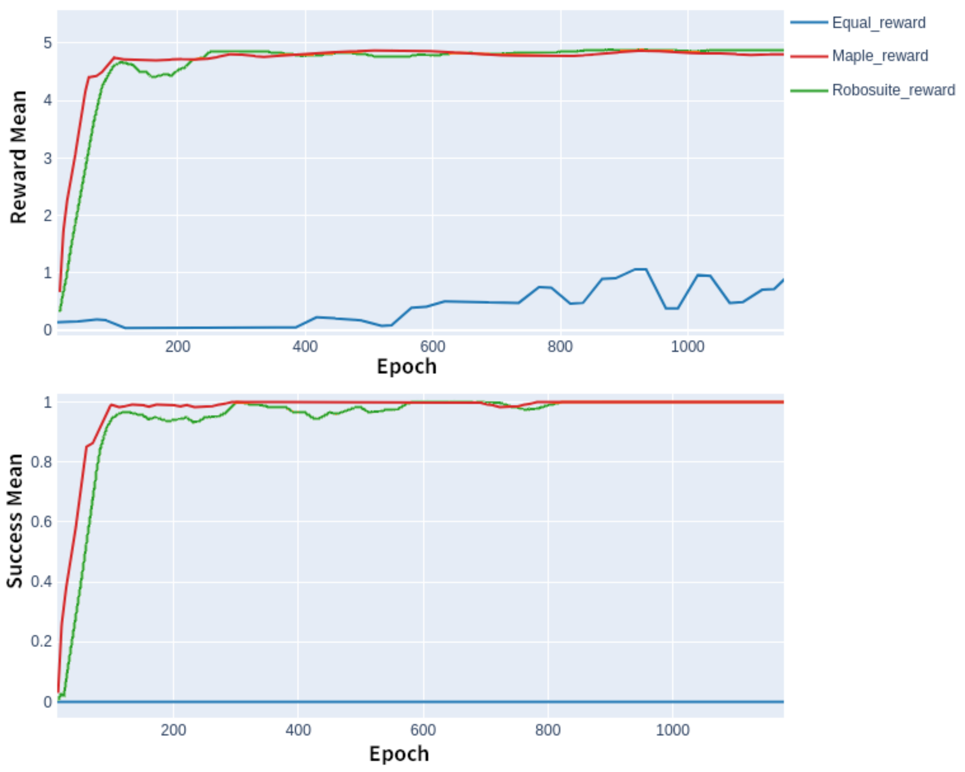
Figure 7.1: Learning curves showing average episodic task rewards and success rates for different hidden layer sizes throughout training.

7.1.2 Reward sensitivity

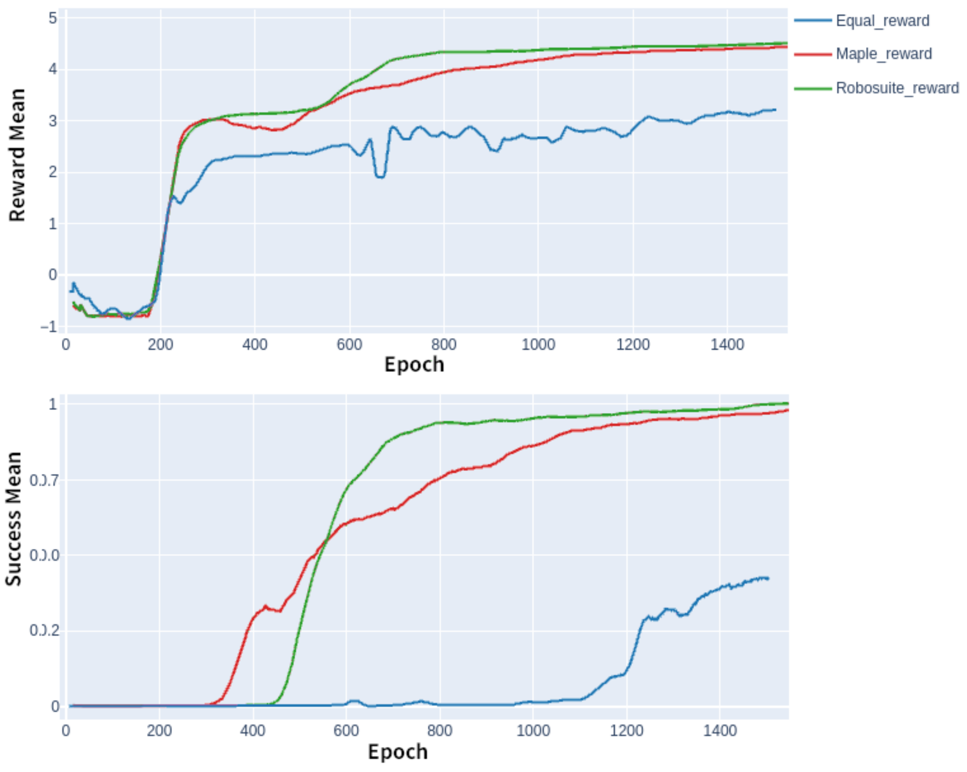
To test the MAPLE's sensitivity to the reward function divergence, we use three different sets of staged reward multipliers. Each of the Robosuite benchmarking environments comes with staged reward functions where each function has a different emphasizing multiplier. We compare MAPLE's multipliers [5] with the multipliers from Robosuite [20] and lastly with all multipliers set to a single value.

The learning curves in Figure 7.2 show the effects of staged reward multipliers on learning new policies. Each learning curve is averaged over 5 seeds.

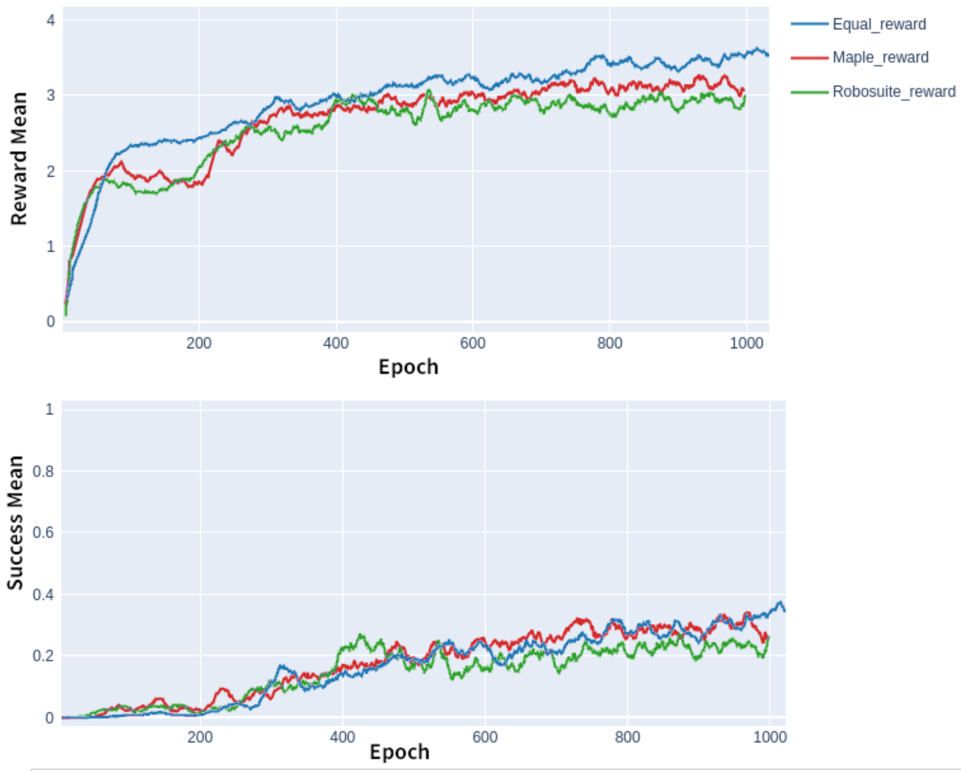
Robosuite reward multipliers produced learning times and success rates similar to the original MAPLE multipliers in all four experiments. In contrast, uniform reward multipliers failed to teach successful policies in the Door Opening and Nut Assembly tasks. In the Wiping environment, all sets of multipliers yielded similarly successful policies. Learning curves from the Nut Assembly task (in Figure 7.2b) show similar behavior for around the first 300 epochs. However, policies trained with uniform reward multipliers reached a local optimum and failed to meaningfully complete the task for another 1000 epochs.



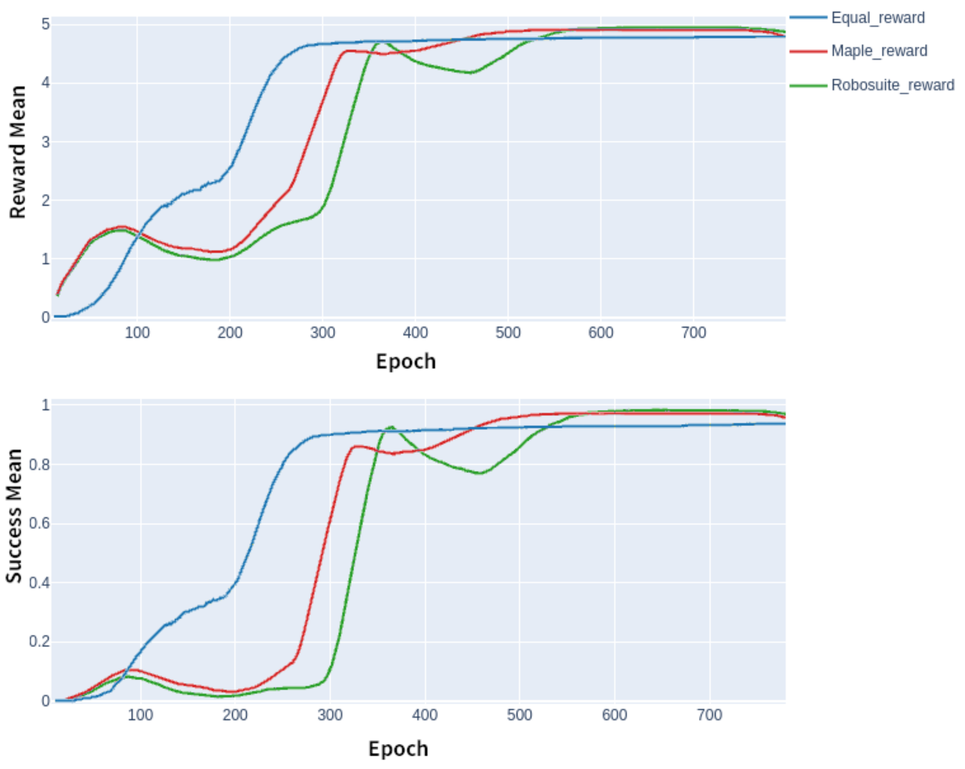
(a) : Door Opening



(b) : Nut Assembly



(c) : Table Wiping



(d) : Lift

Figure 7.2: Learning curves showing the average episodic task rewards and success rates of different staged reward multipliers.

7.1.3 Environmental changes

In Table 7.2, we provide results from evaluating MAPLE policies in cluttered environments. Low-clutter environments contain a single nonrelevant object, and high-clutter environments contain four nonrelevant objects. Success rates are averaged over 5 evaluation runs (one evaluation run spans over 20 episodes).

All evaluated policies showed decreased success rates in clutter environments apart from policies in the Wipe environment. In low-clutter environments, MAPLE policies managed similarly high success rates in all but one environment. On the other hand, success rates plummeted in highly cluttered environments. Mainly in the high-clutter Pick and Place environment, MAPLE policies failed completely.

Environments with	No Clutter	Low-clutter	high-clutter
Nut Assembly	100.0 ± 0.0	73.0 ± 34.0	7.0 ± 12.8
Wipe	34.0 ± 10.3	35.0 ± 9.3	29.0 ± 15.3
Lift	100.0 ± 0.0	99.0 ± 2.2	46.0 ± 30.9
Stack	99.0 ± 1.2	98.0 ± 5.8	26.0 ± 47.9
Pick and Place	97.0 ± 2.7	95.0 ± 6.2	0.0 ± 0.0
Clean Up	83.0 ± 4.3	83.0 ± 6.9	15.0 ± 34.9
Peg Insertion	98.0 ± 2.0	91.0 ± 8.0	33.0 ± 28.9

Table 7.2: Success rates (%) of environments with clutter

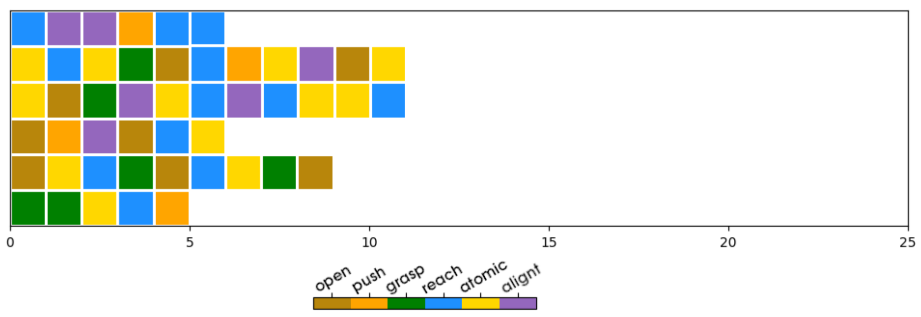
7.2 Evaluation of MAPLE framework extensions

In this section, we analyze the effects of our additions to the MAPLE framework. We provide the results of the following extensions:

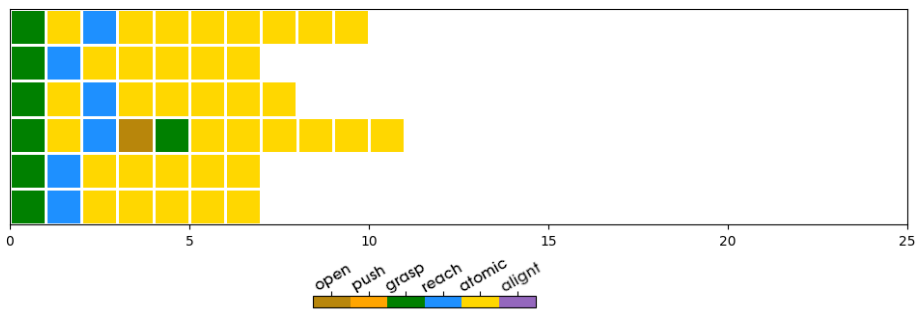
1. Inclusion of new parametrized action, the align primitive.
2. Creation of new benchmarking environments.
3. Incorporation of prior knowledge.

7.2.1 Skill addition

Our newly introduced parameterized action (*Align*) aimed to solve the use of atomic actions in the peg insertion task. *Align* action was used during the early exploration phases of learning see Figure 7.3a. However, as shown in Figure 7.3b, the hierarchical policy did not apply our parametrized action in the final solution (i.e., the policy still elected the atomic primitive to align and insert the peg to solve the task).



(a) : Action sequences used in the exploration of epoch 100.



(b) : Action sequences used in the evaluation of epoch 2000.

Figure 7.3: Illustrations of skills employed in the Peg Insertion task. Each row corresponds to a single simulated episode.

7.2.2 New environments

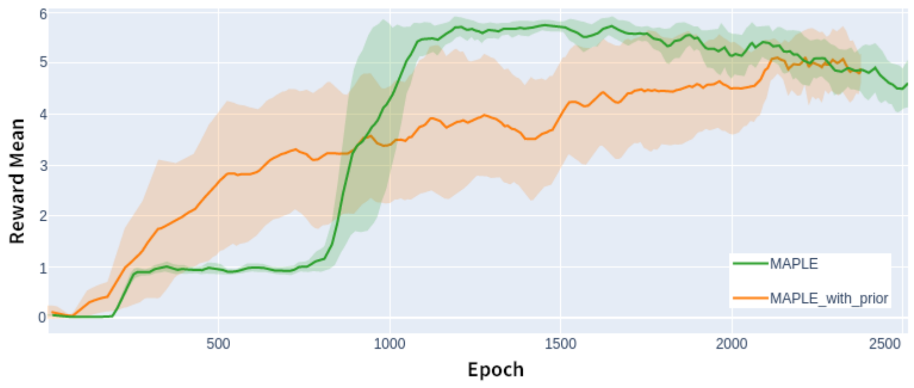
In Table 7.3 are averaged success rates over 5 evaluation runs in the environments outlined in Section 6.2.3. Each policy has been training for 2500 epochs (approximately 48 hours on our setup). The first stage in Cube sorting is pushing the largest cube into its position and in Triple Stack it signifies two cubes being stacked on top of each other. The second stage is equivalent to the task goal condition as listed in Table 6.1

Success Rate (%)	First Stage	Second Stage
Cube Sort	100.0 ± 0.0	76.0 ± 38.0
Triple Stack	100.0 ± 0.0	23.0 ± 9.8

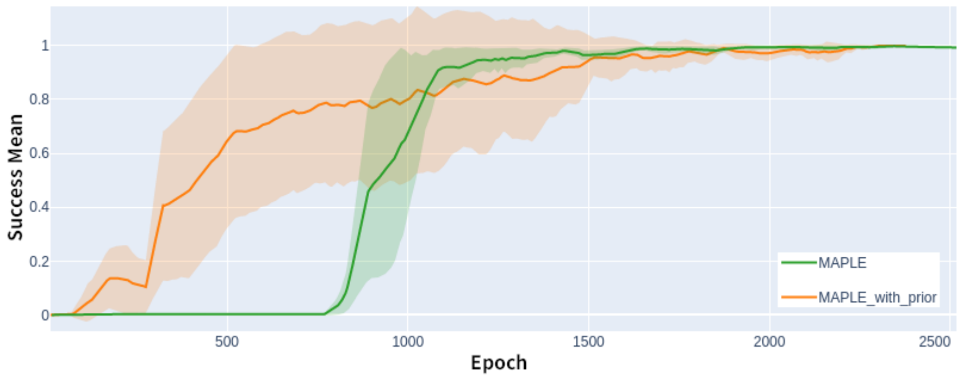
Table 7.3: Final success rates (%) for different reward multipliers

7.2.3 Prior knowledge

Figure 7.4 presents results of extended MAPLE with priors compared with original MAPLE algorithm [5]. The observation sequence used in training extended MAPLE was (*Grasp*, *Reach*, *Realease*). As can be seen in Figure 7.4b, the introduction of prior knowledge accelerates the learning process, as is evident from the success rate rising significantly earlier around epoch 300. Eventually, both methods achieve the same success rate.



(a) : Comparison of average episodic task rewards. Shaded areas indicate the standard deviation.



(b) : Comparison of average episodic success rates. Shaded areas indicate the standard deviation.

Figure 7.4: Learning curves of training with and without prior knowledge.

Chapter 8
Discussion

We segment our analysis of our results from Chapter 7 into two corresponding discussions.

- In the first discussion, we interpret the results of sensitivity testing of the MAPLE algorithm under different parameters and reward functions. Furthermore, we discuss the evaluation of MAPLE policies in cluttered environments. We try to explain the different behaviors seen in our data and provide context for outlier results.

- The second discussion is regarding the results we achieved with our extensions of the MAPLE framework. We expand on the results from newly included tasks and review the failed addition of new parametrized action. Finally, we analyze the results of the proposed learning with prior knowledge.

8.1 MAPLE framework sensitivity

In testing with different hidden layer sizes, MAPLE displayed the best learning-to-success ratio for the baseline hidden layer size 256, see Table 7.1. In general, the number of epochs to complete the tasks slowed with the biggest networks with hidden layer size 512, as shown in Figure 7.1, most likely due to the added size complexity of the network. The 512 networks achieved slightly better success rates when comparing the three manipulation tasks: clean up, peg insert, and cube stack. However, success rate results are within the standard deviations of each other. Moreover, each epoch was more computationally demanding, and on average, the 512 networks took 20 percent more time to train. This, combined with the relative sample inefficiency, meant training policies spanned over multiple days. We can conclude that the solutions to the tested manipulation tasks require policy networks made up of at least 2 layers with hidden sizes of 256. Moreover, this size makes a perfect compromise to maximize learning efficiency.

Policies trained with different sets of staged reward multipliers depict the dependency of MAPLE exploration on staged rewards. This can be seen when comparing the learning curves of the Nut assembly task and the Lift task in Figure 7.2. Policies trained with uniform reward multipliers achieve the staged reward for grasping the task-relevant object. This allows the policy to solve the simpler Lift task. However, in the Nut assembly tasks, the learning in Figure 7.2b lacks the "reward push" of the original MAPLE staged multipliers to manipulate the nut object onto the peg (i.e., learning achieves local optima and only after an additional 1000 epochs discovers and learns the complete set of actions for reaching and releasing the nut onto the right peg).

Evaluation in cluttered environments displayed the robustness of learned policies in low-clutter environments. Policies were able to avoid unnecessary objects and complete tasks. However, evaluation in the high-clutter environments has not demonstrated any new complex behavior. This can be seen in the Pick and Place high-clutter environment, where the policy failed completely, see Table 7.2, 3rd column. The number of items in the left bin, see the Pick and Place render in Table 6.1, made it impossible for the policy to pick up any objects. One exception was the evaluation of the Wipe task, where the applied cutter had seemingly no effect. This was due to the nature of the task, where the robot solution repeatedly applied the push primitive to clean spots on the table. This amounted to seemingly random swiping motions around brown spots, pushing everything out of the way.

8.2 Our extension of the MAPLE framework

The sequences of parametrized actions used to solve the Peg insertion task in Figure 7.3b illustrate that our *Align* parametrized action was completely omitted from the solution as can be seen in Figure 7.3b. This could be the result of improper action design or insufficient affordance rewards. The creation of new tasks was plagued with difficulties regarding the staged reward functions. Policies would often find loopholes in reward definitions, which resulted in nondesired behavior (for example, continuously picking and releasing one cube). The multi-object tasks entailed longer episodes affecting the learning times. Nonetheless, the MAPLE algorithm managed to solve our multi-object long-horizon tasks with a 76% success rate in the Cube sorting task and a 23% success rate in the Triple stack task (see results for Triple stack and Cube sorting tasks in Table 7.3). Last and foremost, our extension of the MAPLE framework with prior knowledge improved the exploration strategy of MAPLE policy networks, resulting in accelerated learning. However, the high sample demand for MAPLE learning is time and resource-consuming even with the benefits of our modification.

Chapter 9

Conclusion and future work

In this work, we introduced and compared two methodologies: PAMDPs and options. Both approaches are employed in state-of-the-art works to achieve hierarchical reasoning with temporarily extended actions (skills or parametrized actions). We decided to leverage PAMDPs as the predefined parametrized actions allow for more straightforward demonstration mapping than the algorithm-learned options. Moreover, we chose the MAPLE framework [5] for its ability to solve complex manipulation tasks with sequences of parametrized actions resembling human task descriptions.

This thesis describes the implementation of the MAPLE algorithm in Python and produces a simple overview of the most important classes and methods necessary for building on to the existing framework. This description also involves the explanation of the underlying Robosuite simulation framework [20]. We detail the inclusion of the YCB objects [23] and two new environments: Triple Stack and Cube Sort; see renders in Figure 6.9. Furthermore, we propose our extension of the MAPLE framework with mapped observations (task-solving human action sequences). The proposed extension incorporates prior knowledge directly into the neural networks and by reward shaping.

Additionally, we presented the results showing the MAPLE algorithm sensitivity to different hidden layer sizes, concluding the networks with the hidden layer sizes of 256 showed the best trade-off between success and learning speed. Analysis of staged reward tests showed the dependency of exploration on staged reward multipliers. Furthermore, we highlighted the MAPLE policies' sensitivity to environmental changes.

We have introduced a new *Align* primitive. Nevertheless, our parametrized action was not used in the final solution of the Peg insertion task. On the other hand, we presented a 76% success rate in the newly introduced Cube sorting task and a 23% success rate in the Triple stack task. We successfully improved the MAPLE framework by incorporating prior knowledge, achieving faster learning in the Cube stacking task.

Further expansion of the mapping function could lead to obtaining prior knowledge from gesture recordings or internet videos [17],[18]. Moreover, future work could attempt to transfer the MAPLE policies to real-life environments.

Bibliography

[1] SUTTON, Richard S.; PRECUP, Doina; SINGH, Satinder. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 1999, 112.1-2: 181-211.

[2] BELLEMARE, Marc, et al. Unifying count-based exploration and intrinsic motivation. Advances in neural information processing systems, 2016, 29.

[3] BOHG, Jeannette, et al. Data-driven grasp synthesis—a survey. IEEE Transactions on robotics, 2013.

[4] MASSON, Warwick; RANCHOD, Pravesh; KONIDARIS, George. Reinforcement learning with parameterized actions. In: Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[5] NASIRIANY, Soroush; LIU, Huihan; ZHU, Yuke. Augmenting reinforcement learning with behavior primitives for diverse manipulation tasks. In: 2022 International Conference on Robotics and Automation (ICRA). IEEE, 2022. p. 7477-7484.

[6] KUINDERSMA, Scott, et al. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. Autonomous robots, 2016, 40: 429-455.

[7] KARAMAN, Sertac; FRAZZOLI, Emilio. Sampling-based algorithms for optimal motion planning. The international journal of robotics research, 2011, 30.7: 846-894.

[8] GARRETT, Caelan Reed, et al. Integrated task and motion planning. Annual review of control, robotics, and autonomous systems, 2021, 4: 265-293.

[9] MACHADO, Marlos C., et al. Temporal abstraction in reinforcement learning with the successor representation. Journal of Machine Learning Research (JMLR) 24, 1-69, 2023.

[10] KONIDARIS, George; BARTO, Andrew. Skill discovery in continuous reinforcement learning domains using skill chaining. Advances in neural information processing systems, 2009, 22.

[11] BAGARIA, Akhil; KONIDARIS, George. Option discovery using deep skill chaining. In: International Conference on Learning Representations. 2020.

[12] KLISSAROV, Martin; PRECUP, Doina. Flexible option learning. Advances in Neural Information Processing Systems, 2021, 34: 4632-4646.

[13] MASSON, Warwick; RANCHOD, Pravesh; KONIDARIS, George. Reinforcement learning with parameterized actions. In: Thirtieth AAAI Conference on Artificial Intelligence. 2016.

[14] HAUSKNECHT, Matthew; STONE, Peter. Deep reinforcement learning in parameterized action space. In: Proceedings of the International Conference on Learning Representations (ICLR). 2016.

[15] ARGALL, Brenna D., et al. A survey of robot learning from demonstration. Robotics and autonomous systems, 2009, 57.5: 469-483.

[16] PERTSCH, Karl, et al. Demonstration-Guided reinforcement learning with learned skills. In: Conference on Robot Learning (CoRL). 2021.

[17] SHAW, Kenneth; BAHL, Shikhar; PATHAK, Deepak. VideoDex: Learning Dexterity from Internet Videos. In: Conference on Robot Learning (CoRL). 2022.

[18] YANG, Yezhou, et al. Robot learning manipulation action plans by" watching" unconstrained videos from the world wide web. In: Proceedings of the AAAI conference on artificial intelligence. 2015.

[19] DALAL, Murtaza; PATHAK, Deepak; SALAKHUTDINOV, Russ R. Accelerating robotic reinforcement learning via parameterized action primitives. Advances in Neural Information Processing Systems, 2021, 34: 21847-21859.

[20] ZHU, Yuke, et al. robosuite: A modular simulation framework and benchmark for robot learning. arXiv preprint arXiv:2009.12293, 2020.

[21] OPENAI(2023). GPT-4 technical report. arXiv: 2303.08774, 2023.

[22] ZHU, Yifeng; STONE, Peter; ZHU, Yuke. Bottom-up skill discovery from unsegmented demonstrations for long-horizon robot manipulation. IEEE Robotics and Automation Letters, 2022, 7.2: 4126-4133.

[23] CALLI, Berk, et al. The YCB object and model set: Towards common benchmarks for manipulation research. In: 2015 international conference on advanced robotics (ICAR). IEEE, 2015. p. 510-517.

[24] TODOROV, Emanuel; EREZ, Tom; TASSA, Yuval. Mujoco: A physics engine for model-based control. In: 2012 IEEE/RSJ international conference on intelligent robots and systems. IEEE, 2012. p. 5026-5033.

[25] MADDISON, Chris J.; MNIH, Andriy; TEH, Yee Whye. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In: International Conference on Learning Representations. 2016.

[26] HAARNOJA, Tuomas, et al. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International conference on machine learning. PMLR, 2018. p. 1861-1870.

[27] HAARNOJA, Tuomas, et al. Soft actor-critic algorithms and applications. arXiv preprint arXiv:1812.05905, 2018.

[28] PERTSCH, Karl; LEE, Youngwoon; LIM, Joseph. Accelerating reinforcement learning with learned skill priors. In: Conference on robot learning. PMLR, 2021. p. 188-204.

[29] LIU, Huihan, et al. Robot Learning on the Job: Human-in-the-Loop Autonomy and Learning During Deployment. Robotics: Science and Systems (RSS), 2023.

[30] MANDLEKAR, Ajay, et al. What Matters in Learning from Offline Human Demonstrations for Robot Manipulation. In: Conference on Robot Learning. PMLR, 2022. p. 1678-1690.

[31] MNIH, Volodymyr, et al. Playing atari with deep reinforcement learning. In: NIPS Deep Learning Workshop, 2013.

[32] SILVER, David, et al. Mastering the game of Go with deep neural networks and tree search. nature, 2016, 529.7587: 484-489.

[33] AGARWAL, Rishabh, et al. Deep reinforcement learning at the edge of the statistical precipice. Advances in neural information processing systems, 2021, 34: 29304-29320.

[34] HENDERSON, Peter, et al. Deep reinforcement learning that matters. In: Proceedings of the AAAI conference on artificial intelligence. 2018.

[35] CIGNONI, Paolo, et al. Meshlab: an open-source mesh processing tool. In: Eurographics Italian chapter conference. 2008. p. 129-136.

[36] LILLICRAP, Timothy P., et al. Continuous control with deep reinforcement learning. In : Proceedings of the International Conference on Learning Representations (ICLR). 2016.

[37] ZHAN, Huixin; TAO, Feng; CAO, Yongcan. Human-guided robot behavior learning: A gan-assisted preference-based reinforcement learning approach. IEEE Robotics and Automation Letters, 2021, 6.2: 3545-3552.

[38] HUSSENOT, Léonard, et al. Hyperparameter selection for imitation learning. In: International Conference on Machine Learning. PMLR, 2021. p. 4511-4522.

[39] HENDERSON, Peter, et al. Deep reinforcement learning that matters. In: Proceedings of the AAAI conference on artificial intelligence. 2018.

[40] KINGMA, Diederik P.; BA, Jimmy. Adam: A Method for Stochastic Optimization. international conference on learning representations. 2015.

[41] POMERLEAU, Dean A. Alvinn: An autonomous land vehicle in a neural network. Advances in neural information processing systems, 1988.

[42] ABBEEL, Pieter; NG, Andrew Y. Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the twenty-first international conference on Machine learning. 2004. p. 1.

[43] ARZATE CRUZ, Christian; IGARASHI, Takeo. A survey on interactive reinforcement learning: Design principles and open challenges. In: Proceedings of the 2020 ACM designing interactive systems conference. 2020. p. 1195-1209.

[44] GRIFFITH, Shane, et al. Policy shaping: Integrating human feedback with reinforcement learning. Advances in neural information processing systems, 2013, 26.

[45] NG, Andrew Y.; HARADA, Daishi; RUSSELL, Stuart. Policy invariance under reward transformations: Theory and application to reward shaping. In: Icml. 1999. p. 278-287.

[46] DUAN, Yan, et al. Benchmarking deep reinforcement learning for continuous control. In: International conference on machine learning. PMLR, 2016. p. 1329-1338.

[47] GUPTA, Abhishek, et al. Relay Policy Learning: Solving Long-Horizon Tasks via Imitation and Reinforcement Learning. In: Conference on Robot Learning. PMLR, 2020. p. 1025-1037.

[48] Augmented MAPLE framework. [Accessed 9 January 2024]. Available from: https://gitlab.ciirc.cvut.cz/majsnmar/pamdps-with-priors

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Majsner Marek**	Personal ID number:	**492336**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Reinforcement Learning with Parametrized Actions for Imitation Learning

Bachelor’s thesis title in Czech:

Posilované učení parametrizovaných akcí pro imitační učení

Guidelines:

Children learn to sequence individual movements to perform the given task (e.g. get a toy). Depending on whether their actions were successful or not, they correct their behaviour. Similarly, a teacher (parent) can adjust their behaviour, which can help them to achieve their goal quicker. Similarly, reinforcement learning methods with parametrized action primitives are learning how to sequence and parametrize individual primitive motions to fulfil the given task. This thesis aims to compare and evaluate available methods and extend them by new tasks or actions, and possibly incorporate into learning the prior knowledge about the suitable primitive actions and their sequencing coming from the supervisor (demonstrator).
1. Review and qualitatively compare methods of deep reinforcement learning with parametrized action space (e.g. [1], [2], [3],[4]). The comparison should focus on the type of algorithm, loss function, testing environment, set of primitive actions, etc..
2. Select one of the methods and evaluate its sensitivity to various settings, such as environment (configuration of objects, the complexity of the task, etc.), type of the search, loss function, initialisation, etc..
3. For a selected set of tasks evaluate and visualise the quality of recognition of the sequence of primitive actions leading to the fulfilment of the given task by the chosen method.
4. For the same set of tasks, evaluate and visualise the quality of recognition of the sequence of primitive actions from gestures demonstrated by a human. Use the available Gesture toolbox [5].
5. Extend and possibly improve the method's performance by one of the following options:
a) Adaptation to new environment and tasks (new object types and configuration, various complexity of the tasks, etc.)
b) Extension by own parametrized actions
c) Initialize the method with demonstration data (the demonstrator shows the primitive actions by gestures).

Bibliography / sources:

[1] Nasiriany, Soroush, Huihan Liu, and Yuke Zhu. "Augmenting reinforcement learning with behavior primitives for diverse manipulation tasks." 2022 International Conference on Robotics and Automation (ICRA). IEEE, 2022.
[2] Z. Wu, N. M. Khan, L. Gao and L. Guan, "Deep Reinforcement Learning with Parameterized Action Space for Object Detection," 2018 IEEE International Symposium on Multimedia (ISM), Taichung, Taiwan, 2018, pp. 101-104, doi: 10.1109/ISM.2018.00025.
[3] Bagaria, Akhil, and George Konidaris. "Option discovery using deep skill chaining." International Conference on Learning Representations. 2020.
[4] Dalal, Murtaza, Deepak Pathak, and Russ R. Salakhutdinov. "Accelerating robotic reinforcement learning via parameterized action primitives." Advances in Neural Information Processing Systems 34 (2021): 21847-21859.
[5] Vanc, Petr, Jan Kristof Behrens, and Karla Stepanova. "Context-aware robot control using gesture episodes." 2022 International Conference on Robotics and Automation (ICRA). IEEE, 2023.

Name and workplace of bachelor’s thesis supervisor:

Mgr. Karla Štěpánová, Ph.D. Robotic Perception CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **04.02.2023** Deadline for bachelor thesis submission: **09.01.2024**

Assignment valid until: **22.09.2024**

Mgr. Karla Štěpánová, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature