Hierarchical Reinforcement Learning with Universal Policies for Multi-Step Robotic Manipulation

Xintong Yang[1], Ze Ji[1] *Member, IEEE*, Jing Wu[2], Yu-Kun Lai[2], Changyun Wei[3], Guoliang Liu[4], Rossitza Setchi[1]

***Abstract*—Multi-step tasks, such as block stacking or parts (dis)assembly, are complex for autonomous robotic manipulation. A robotic system for such tasks would need to hierarchically combine motion control at a lower level and symbolic planning at a higher level. Recently, reinforcement learning (RL) based methods have been shown to handle robotic motion control with better flexibility and generalisability. However, these methods have limited capability to handle such complex tasks involving planning and control with many intermediate steps over a long time horizon.**

Firstly, current RL systems cannot achieve varied outcomes by planning over intermediate steps (e.g., stacking blocks in different orders). Secondly, exploration efficiency of learning multi-step tasks is low, especially when rewards are sparse.

To address these limitations, we develop a unified hierarchical reinforcement learning framework, named Universal Option Framework (UOF), to enable the agent to learn varied outcomes in multi-step tasks. To improve learning efficiency, we train both symbolic planning and kinematic control policies in parallel, aided by two proposed techniques: 1) an auto-adjusting exploration strategy (AAES) at the low level to stabilise the parallel training, and 2) abstract demonstrations at the high level to accelerate convergence.

To evaluate its performance, we performed experiments on various multi-step block-stacking tasks with blocks of different shapes and combinations and with different degrees of freedom for robot control. The results demonstrate that our method can accomplish multi-step manipulation tasks more efficiently and stably, and with significantly less memory consumption.

***Index Terms*—Robotic manipulation, multi-step tasks, hierarchical reinforcement learning, universal policy, option framework, planning and control.**

I. Introduction

HUMANS solve complex manipulation tasks by dividing them into multiple steps. Similarly, for a robot to accomplish a task such as assembly, it is required to decompose the task into a sequence of intermediate steps, e.g., moving the gripper to a place, grasping an object, placing the object, etc. Such a decomposition enables different combinations and ordering of steps to achieve various desired outcomes.

This work seeks to solve such multi-step planning and control tasks with a reinforcement learning-based solution. Fig. 1 shows an example, where a robot is trying to stack three blocks together. Learning such a task is difficult in two ways. Firstly, the robot needs to learn various desired outcomes: grasping and placing the blocks in different orders to build different towers, e.g., green-blue-red or blue-green-red (in the top-down order). Secondly, the robot also needs to learn the step dependencies to achieve a desired outcome. For example, it is impossible to place a block if the block has not been successfully grasped. Previous studies either deal with only one particular order of steps or ignore the orders [1], [2]. This means that, to achieve multiple desired outcomes, repetitively training multiple policies is required, costing additional computations, memory and time.

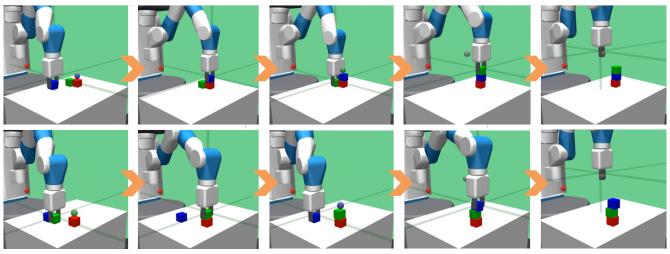


Figure 1: Interdependent steps of a block-stacking task investigated in this paper.

For tasks involving multiple steps or outcomes, we hypothesise that the learnt skills, knowledge or experience can be shared and reused. For example, placing a block at two locations depends on the same former step of grasping the block. This motivates us to learn high-level planning and low-level kinematic control with universal policies in the context of multi-goal reinforcement learning [3], [4].

However, the inter-dependency of steps makes such learning extremely difficult when rewards are sparse. Considering the task where an RL agent is learning to stack 2 distinct blocks (A and B), with a reward function that only rewards task completion, the agent will never learn to place A on B before it successfully learns to grasp A. In other words, exploration (i.e., collecting useful data) becomes more inefficient when learning towards later steps, as the probability of encountering meaningful rewards decreases.

One popular approach to this problem is to hand-craft a human prior-based reward function, which eliminates the reward sparsity but demands complex design and introduces human bias [5]. Another way is to use demonstrations [6]. For example, kinematic trajectories performed by a human can help improve learning efficiency. However, such demonstrations are exhausting to obtain and not readily available for many tasks [1], [2], [6].

In short, given the research gaps identified above, this paper studies the following questions:

1) How to learn the various desired outcomes of a multi-

Corresponding author: Ze Ji (email: jiz1@cardiff.ac.uk)

[1]Centre for Artificial Intelligence, Robotics and Human-Machine Systems (IROHMS), School of Engineering, Cardiff University, Cardiff, UK {yangx66, jiz1, setchi}@cardiff.ac.uk

[2]School of Computer Science and Informatics, Cardiff University, Cardiff, UK {wuj11, laiy4}@cardiff.ac.uk

[3]Robotics Engineering, Hohai University, Changzhou, China c.wei@hhu.edu.cn

[4]School of Control Science and Engineering, Shandong University, Jinan, China liuguoliang@sdu.edu.cn

step task?
2) How to improve the learning efficiency for such tasks with easy-to-collect knowledge/demonstrations?

For question 1) we unify hierarchical reinforcement learning (HRL) and universal policies, leading to our novel Universal Option Framework (UOF). HRL enables the robot to learn long-horizon multi-step tasks via high-level planning and low-level motion control. It typically decomposes a task into ordered steps (e.g., grasping followed by placing the grasped object in a block stacking task), while universal policies, one at the planning level and one at the control level, enable multi-goal learning for various outcomes (planning level) and manipulation skills (control level).

We propose to train both the high-level and low-level policies in parallel. This will allow more computationally efficient learning without the need of repetitive data collection. However, parallel training can be highly unstable. This is because the data for training the high-level policy is produced by an exploring low-level policy, and thus is noisy and not informative for most of the time [7]. Such noisy data prevents the high-level policy from stable improvement. Most previous works attempted to avoid this issue by simply training them separately [8]–[10], while, in this work, we propose to stabilise parallel training by adaptively reducing exploration of the low-level policy based on the performance of achieving goals in different steps.

To address question 2), we introduce an abstract form of demonstrations, i.e., the correct orders of steps, to deal with extremely sparse rewards. This is inspired by real-world examples such as instructions of building a Lego toy or assembling a piece of furniture. Instead of collecting demonstrations of kinematic trajectories or altering the reward function, using abstract demonstrations, in our experiments, has been shown to significantly improve the learning efficiency.

We test our methods by simulating a 7-DoF Fetch Robot to learn a set of block-stacking tasks, as illustrated in Fig. 1. Other similar tasks can also be easily accommodated. The low-level (control) policy is trained with the DDPG (Deep Deterministic Policy Gradient) algorithm with Hindsight Experience Replay (HER) [4], [11]. For the high-level (planning) policy, we adapt the Intra-Option Learning algorithm [12] to a deep learning and experience replay version (Section V-A). The source codes including the simulation environment and algorithms will be made available on GitHub[1].

The rest of this paper is organised as follows. We review related studies in Section II. Preliminaries (including standard RL, the Option Framework and Goal-conditioned RL) are reviewed in Section III. Our novel Universal Option Framework (UOF) is presented in Section IV, with training algorithms in Section V. Section VI introduces the tasks and experimental setup, while Section VII analyses the experiment results. Section VIII concludes this research.

II. Related work

This work relates closely to three topics, including hierarchical reinforcement learning (HRL), goal-conditioned reinforcement learning (GRL), and robotic block-stacking tasks.

Options and HRL: The Option Framework (OP) [13] provides a promising architecture to enable temporally extended actions. One research direction is to define or pre-train a fixed set of options (low-level policies), given which one can train an inter-option (high-level, gating) policy for symbolic planning [14], [15] or skill composition [10].

Two challenges studied in this work are related to HRL. The first is the non-stationary transition problem (see section IV-C for details) that occurs when training policies at different levels in parallel [7]. A typical approach for such problems is to train them independently [8]–[10]. We propose a novel exploration strategy to address this non-stationary issue during parallel training (see section V-B). Different from methods that focus on improving exploration efficiency [16]–[18], we instead aim to reduce unnecessary exploration of the low-level control policy and thus stabilise parallel training. Secondly, we train a high-level policy to reuse the low-level policies to achieve different final outcomes, e.g., stacking blocks in different orders, while recent hierarchical methods only focus on solving a single-outcome task [3], [8], [9], [19].

GRL: A goal-conditioned value function (or policy) integrates knowledge about pursuing different purposes in the same environment [3], [20], [21]. This broadened definition was further exploited by hindsight-goal-relabelling methods, which significantly improve the sampling efficiency of continuous Reinforcement Learning with sparse rewards [4]. Recent implementations in robotics include language-based goals [9], imitation learning [2], auxiliary tasks learning [22] and goal-generation (related to intrinsic motivation) [16], [17], [23].

Our work casts the idea of GRL into the Option Framework to obtain a universal hierarchical reinforcement learning architecture. By doing this, our framework can learn multiple outcomes and skills, while previous hierarchical frameworks lack this ability [3], [8], [9], [19].

Robotic Block-stacking Tasks: Block-stacking is deemed as a typical robotic task that requires long-horizon motion planning and control. Conventionally, this task has been widely used to validate various classic planning methods. These well-researched classic methods for motion and task planning have been proved to be effective in various settings [24].

The closest topic to our work would be the Task-motion planning (TMP) [25] that addresses the problem of generating high-level task sequences and uses a motion planner to solve each task. For task level planning, most works use symbolic planning methods [26], [27]. For the low-level task solver, algorithms such as probabilistic random map (PRM) or rapidly-exploring random trees (RRT) are common choices [24]. However, these classic methods rely heavily on expert knowledge and hard-coded rules, thus they are in general very domain-specific. In addition, classic methods require re-planning for every task, at both levels, while deep reinforcement learning-based frameworks can output valid solutions without any searching and provide better generalisibility [9].

Previously, some works have used RL approaches on simplified tasks either with a block being grasped in hand [28] or heavily shaped reward functions [5]. More recently, based on GRL [4], researchers started to solve more complex cases

[1]https://github.com/IanYangChina/UOF-paper-code

that require manipulating more blocks (up to 9 blocks) with only sparse rewards [1], [2]. They extensively use human-demonstrated kinematic motion trajectories [6], which are not readily available. These works regarding RL have merely learnt one particular order to stack blocks, typically in a fully end-to-end fashion, without considering the combinatorial nature of such multi-step problems.

III. Preliminaries

The following sub-sections introduce the preliminaries for the proposed UOF, namely the standard Reinforcement Learning, the Option Framework formalising a hierarchical architecture for learning low-level control and high-level planning [13], and the GRL for learning tasks with multiple goals [3]. Mathematical notations are summarised in supplementary material.

A. Standard Reinforcement Learning (RL)

Standard RL problems are based on discrete time Markov Decision Processes (MDPs), which is a model for sequential decision-making. MDPs assume that the underlying system dynamics is a Markov process, in which the future states are only affected by the current state and action [29]. An MDP is formalised by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a distribution of initial states $p(\mathbf{s}_0)$, a system transition function $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, a reward function $r : \mathcal{S}\times\mathcal{A} \to \mathbb{R}$ and a discount factor $\gamma \in [0, 1]$.

Given an MDP, a reinforcement learning agent interacts with the system in the discrete timesteps. At a timestep t, the agent observes a system state $\mathbf{s}_t$ and takes an action $\mathbf{a}_t$. The system then transits to the next state $\mathbf{s}_{t+1}$ according to the transition function, and emits a reward r_{t+1}. In most cases, it is hard to obtain an exact transition function for complex environment interactions. Thus, as many other works, we assume the transition function is unknown to the agent. The reward function is usually defined to reflect the success or failure of a given task.

The objective of the agent, i.e., of a standard RL problem, is to find a policy $\pi(\mathbf{a}|\mathbf{s}) : \mathcal{S} \to \mathcal{A}$ that produces actions for given states to maximise an expected discounted return (i.e., cumulative rewards) $\mathbb{E}[\mathcal{R}] = \mathbb{E}[\sum_{t=0}^{T} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)]$. The policy could also be implicitly represented by an action-value Q-function $Q^\pi(\mathbf{s}, \mathbf{a})$, which is itself the expected discounted return after taking an action at a state and following a policy thereafter [29].

B. The Option Framework (OF)

The OF is a classic hierarchical RL architecture. It introduces the notion of 'temporal abstraction' into the standard RL problem, enabling abstract planning with temporally-extended actions, called options. An option can be regarded as a subtask, a step or a skill, which may take several actions over a period of time [13]. For example, robotic arm motions for grasping and placing objects can be regarded as two options.

Formally, an option is denoted as $o\langle\mathcal{I}^o, \pi^o, \beta^o(\mathbf{s})\rangle$, where, $\mathcal{I}^o \subseteq \mathcal{S}$ is the set of initialisation states; π^o is an intra-option policy that can be pre-trained with standard RL using a reward function specific to the subtask; and $\beta^o(\mathbf{s}) \in [0, 1]$ is the termination function indicating whether an option is terminated at a state [13].

Given a set of pre-defined options, $\mathcal{O}$, the aim of OF is to find an inter-option policy $\Omega(o|\mathbf{s}) : \mathcal{S} \to \mathcal{O}$ that selects options (i.e. steps) for given states to maximise an expected discounted return. Similar to the standard RL, it can also be represented by an option-value function $Q^\Omega(\mathbf{s}, o)$ [13].

The inter-option policy plans over options to finish a task. For instance, given two options that can grasp and place blocks, the inter-option policy may learn to stack several blocks by selecting the options in a particular order.

In the rest of the paper, intra- and inter-option policies are referred to as low-level and high-level policies, denoted as $\pi^\mathcal{L}$ and $\pi^\mathcal{H}$, respectively.

C. Goal-conditioned Reinforcement Learning (GRL)

GRL formalises the problem of learning multiple goals in one environment. A policy in GRL is called a universal policy as it integrates knowledge about achieving different goals in one environment [3]. Our framework is an integration of OF and GRL.

In GRL, the MDPs remain the same with standard RL, except that the reward function depends additionally on goals: $r : \mathcal{S} \times \mathcal{G} \to \mathbb{R}$. It is assumed that an achieved goal can be easily found given a state. Typically, one can use part of the system states to represent goals, e.g., the desired Cartesian coordinates of blocks in a block stacking task.

It is assumed that every goal $g \in \mathcal{G}$ corresponds to a predicate $f_g : \mathcal{S} \to \{0, 1\}$, and a GRL agent aims to achieve any state s such that $f_g(s) = 1$. If a goal is a desired state and $\mathcal{S} = \mathcal{G}$, the predicate is simply: $f_g(s) = [s = g]$. More often the predicate is defined by some relationships between the achieved goals and desired goals, e.g., whether the L2-norm of their difference is within a threshold.

The reward function can then be defined as $r(s, a, g) = -[f_g(s) = 0]$, given a desired goal $g \in \mathcal{G}$. Such a function gives a reward of value 0 when a desired goal is achieved and -1 otherwise. The objective is then to maximise the expected return with respect to various goals [4].

IV. The Universal Option Framework

In this section, we propose the Universal Option Framework (UOF). In the original OF, a high-level policy plans over options to accomplish one task, while low-level policies of these options take low-level actions over a period of time to complete different subtasks [13]. We extend both the levels to be goal-conditioned, such that only one universal option is needed for different subtasks by setting different low-level goals, and only one universal high-level policy is needed to plan for different tasks.

Briefly, Section IV-A defines the universal option and the goal-conditioned high-level policy, followed by Section IV-B illustrating the links between the two levels. Section IV-C discusses the non-stationarity of parallel training. Section IV-D provides specific representations of the main components (states, actions, goals and rewards) of the UOF for multi-step

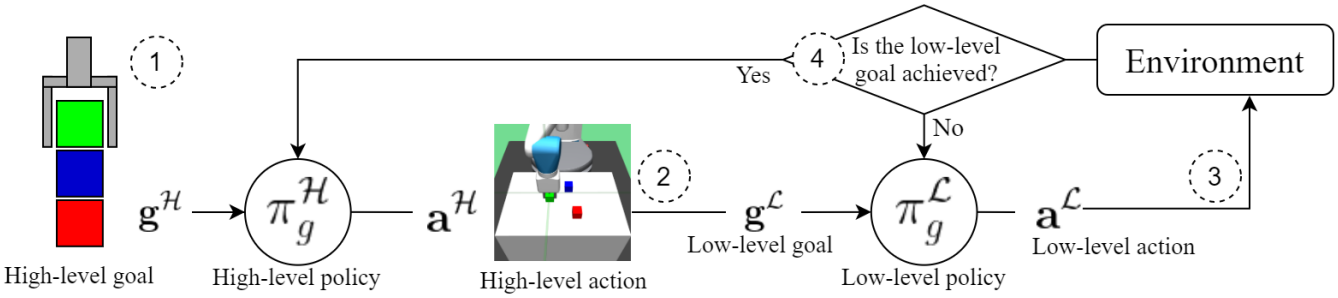


Figure 2: The procedure of the Universal Option Framework in the context of multi-step block-stacking tasks. The numbers in dashed circles indicate the four sub-processes: 1) an episode starts with a high-level goal (e.g., a desired order of blocks: Green→Blue→Red); 2) the high-level policy takes an action, which relates to a low-level goal (e.g., "Grasping Green Block"); 3) the low-level policy selects several actions (e.g., continuous gripper movements) to interact with the environment and try to achieve the low-level goal; 4) the low-level policy is terminated when the low-level goal is achieved, upon which the high-level policy selects another low-level goal.

block-stacking tasks. Mathematical notations are summarised in supplementary material.

A. Universal Option and High-level Policy

1) A universal option, denoted as $o_g\langle \mathcal{I}_g, \pi_g^{\mathcal{L}}, \beta_g^{\mathcal{L}} \rangle$, eliminates the need of training multiple low-level policies. It has three components.

$\mathcal{I}_g$ is the set of states where a goal is achievable. In this work, we assume that every goal can be achieved from any state, though it may need several intermediate actions. Thus, the initialisation set for any goal is the state space: $\mathcal{I}_g = \mathcal{S}$.

$\pi_g^{\mathcal{L}}(\mathbf{a}^{\mathcal{L}}|\mathbf{s}, \mathbf{g}^{\mathcal{L}})$ is a goal-conditioned low-level policy where $\mathbf{g}^{\mathcal{L}} \in \mathcal{G}^{\mathcal{L}}$ is a low-level goal. This policy produces actions according to different states and goals, e.g., controlling a gripper.

$\beta_g^{\mathcal{L}}(\mathbf{s})$ is the goal-conditioned termination function. For any goal, it gives the probability of that goal being achieved at a state and so the option terminates. We assume it to be a known deterministic mapping:

$f_{\mathbf{g}^{\mathcal{L}}} : \mathcal{S} \rightarrow \{0, 1\},\ \forall \mathbf{g}^{\mathcal{L}} \in \mathcal{G}^{\mathcal{L}}$, where $f_{\mathbf{g}^{\mathcal{L}}}$ is the human-specified predicate used in GRL problems to identify whether a goal is achieved at a state [4].

A sparse reward function for training the universal policy can thus be defined as $r_g^{\mathcal{L}}(\mathbf{s}, \mathbf{a}^{\mathcal{L}}) = -[f_{\mathbf{g}^{\mathcal{L}}} = 0]$.

2) A universal high-level policy is denoted as $\pi_g^{\mathcal{H}}(\mathbf{a}^{\mathcal{H}}|\mathbf{s}, \mathbf{g}^{\mathcal{H}})$, where $\mathbf{g}^{\mathcal{H}} \in \mathcal{G}^{\mathcal{H}}$ is a high-level goal. It learns to achieve various high-level goals by assigning low-level goals to the universal option. Given a predicate $f_{\mathbf{g}^{\mathcal{H}}}$ that indicates whether a high-level goal is achieved at a state, a sparse high-level reward function can be defined as $r_g^{\mathcal{H}}(\mathbf{s}, \mathbf{a}^{\mathcal{H}}) = -[f_{\mathbf{g}^{\mathcal{H}}} = 0]$.

B. Links between Low-level Control and High-level Planning

This subsection illustrates the links between a universal option (with a low-level control policy) and a high-level planning policy. In this paper, we manually decompose a task into N crucial steps. Then, we assume the available access to a mapping $\psi^N : \mathcal{S} \rightarrow \mathcal{G}_1^{\mathcal{L}}, \mathcal{G}_2^{\mathcal{L}}, ..., \mathcal{G}_N^{\mathcal{L}}$ from states to N subsets of desired low-level goals, each of which corresponds to a step.

To enable planning over steps, we assume the high-level policy has N discrete actions, corresponding to the N steps. When a high-level action is taken, the low-level policy receives a low-level goal related to the chosen step, generated by the mapping ψ^N based on the current state. Thus, the high-level policy acts at a higher level as it demands the low-level policy to achieve different steps. Note that, the low-level policy can only act according to the low-level goals assigned by the high-level policy.

Such a mapping that generates desired goals is usually deployed for simulation-based tasks [4], [8], [9]. Table I gives a mapping example for a two-step block-stacking task ($N = 2$). The first row is the state s. Two distinct low-level goals (second and third rows) that correspond to two distinct steps are mapped from the state. The second row corresponds to the step 'grasping the blue block' and the third row corresponds to the step 'placing the blue block on top of the red block'. The high-level policy thus has two actions, related to the two steps, whilst the low-level policy needs to learn to achieve these steps by controlling a gripper.

The universal high-level policy and the universal option linked by the mapping define the UOF. Fig. 2 summarises the running procedure of the UOF in the context of multi-step task learning. It comprises four sub-processes:

1) An episode starts with a random high-level goal $\mathbf{g}^{\mathcal{H}}$;
2) The high-level policy $\pi_g^{\mathcal{H}}$ takes an action $\mathbf{a}^{\mathcal{H}}$, i.e., selects a step, that maps to a low-level goal $\mathbf{g}^{\mathcal{L}}$ via ψ^N;

Table I: An example of ψ^N for a two-step block-stacking task

	Pos.Red	Pos.Blue	Pos.Grip.	Wid.Fin.
$\mathbf{s} \in \mathcal{S}$	$(x_r, y_r, z_f, \|\|$	$x_b, y_b, z_f, \|\|$	$x_{gr}, y_{gr}, z_{gr}, \|\|$	$w_{gr})$
$\mathbf{g}_1 \in \mathcal{G}_1^{\mathcal{L}}$	$(x_r, y_r, z_f, \|\|$	$x_b, y_b, z_f, \|\|$	$x_b, y_b, z_f, \|\|$	$s_{block})$
$\mathbf{g}_2 \in \mathcal{G}_2^{\mathcal{L}}$	$(x_r, y_r, z_f, \|\|$	$x_r, y_r, z_f + h, \|\|$	$x_r, y_r, z_f + h, \|\|$	$s_{block})$

Abbreviation: **Pos.**: position (as Cartesian coordinates); **Grip.**: gripper; **Wid.Fin.**: gripper finger width; $x/y_{r/b/gr}$: the x and y coordinates of the red or blue block or the gripper tip; z_f: the absolute height of the block centre when laying on the workbench; h: the absolute height of a block; s_{block}: size of blocks; ||: Vector concatenation.

3) The low-level policy $\pi_g^{\mathcal{L}}$ (i.e., the universal option) then tries to achieve the given goal. Each low-level action $\mathbf{a}^{\mathcal{L}}$ causes the environment to transit to a new state and emit rewards $(r_g^{\mathcal{H}},\ r_g^{\mathcal{L}})$;
4) The low-level policy is terminated when a low-level goal is achieved ($\beta_g^{\mathcal{L}}(\mathbf{s}) = 1$), only upon which can the high-level policy select a new action.

This architecture summarises recent studies in combining goal-conditioned and hierarchical reinforcement learning [8], [9], [16], [17], [22], [23], emphasising the idea of knowledge sharing and integration in a hierarchical RL system.

C. Parallel Training Instability

Parallel training is promising as it allows the high-level policy to learn simultaneously with the low-level one. There are two benefits of doing so. First, parallel training is less computationally expensive as the high-level policy can start learning without waiting for the pre-training of the low-level policy to finish. Secondly, when trained separately, the pre-trained low-level policy needs to be fine-tuned while training the high-level policy [9], [14], [15], whereas parallel training allows both levels to adapt intermediately.

Parallel training is unstable for the high-level policy due to an exploratory low-level policy [7]. This can be examined from the transition function for the high-level MDPs, i.e., the probability of the policy entering a new state $\mathbf{s}'$, written as

$$\begin{aligned} p^{\mathcal{H}}(\mathbf{s}') &= p^{\mathcal{H}}(\mathbf{s})\ \pi_g^{\mathcal{H}}(\mathbf{a}^{\mathcal{H}}|\mathbf{s})\ p^{\mathcal{H}}(\mathbf{s}'|\mathbf{a}^{\mathcal{H}},\mathbf{s}) \\ &= p^{\mathcal{H}}(\mathbf{s})\ \pi_g^{\mathcal{H}}(\mathbf{a}^{\mathcal{H}}|\mathbf{s})\ \pi_g^{\mathcal{L}}(\mathbf{a}^{\mathcal{L}}|\mathbf{a}^{\mathcal{H}},\mathbf{s})\ p^{E}(\mathbf{s}'|\mathbf{a}^{\mathcal{L}},\mathbf{s}) \end{aligned}$$

where, p^E is the system dynamics. The equation reveals that $p^{\mathcal{H}}(\mathbf{s}')$ depends on the probability of the low-level policy selecting an action, e.g., $\pi_g^{\mathcal{L}}$, given a low-level goal chosen by the high-level policy $\pi_g^{\mathcal{H}}$. This means that a randomly exploring low-level policy will cause the high-level transition probability distribution to be non-stationary.

A policy with a constant exploring ratio will let the agent have the same probability to deviate from the correct trajectory at every timestep. This benefits the low-level policy in terms of better exploration [4], [11]. However, it harms the high-level policy in parallel training because the high-level only obtains reward when the low-level achieves the desired goal. For a task that takes several steps to finish, a constantly exploring low-level policy and the massive search space impede the success of the task and, hence, considerably hinder the learning efficiency of the high-level policy. Previous works separately trained both levels because separate training ensures the low-level policy to be fully deterministic, and thus the high-level policy to learn stably.

In this work, we propose a novel exploration strategy that adapts the low-level exploration to resolve this problem (in Section V-B). We demonstrate that with this strategy, parallel training can achieve better performance with lower computational demands (in Section VII-C).

D. Task-specific Definitions for UOF

This subsection describes the numerical representations of states, actions, goals and rewards for the block-stacking tasks in this work, as an example of applying our proposed UOF. The two levels have different actions, goals and reward functions, but share states and the initial state distribution.

*1) **States & initial state distribution**:* For a block-stacking task with M blocks, we define a state $\mathbf{s}$ as a vector of states for the gripper $\mathbf{s}_{grip}$ and the blocks $(\mathbf{s}_1^b||\mathbf{s}_2^b||...||\mathbf{s}_M^b)$, where $||$ denotes vector concatenation. The gripper state is a vector concatenated by the absolute Cartesian coordinates and the linear velocity of the gripper, the linear velocity of the gripper fingers (symmetric), and the gripper stroke width, formulated as $\mathbf{s}_{grip} = (\mathbf{x}_{grip}^a||\mathbf{v}_{grip}^a||v_{finger}^a||w_{finger})$. The state of the i-th block is represented by a vector concatenating its relative Cartesian coordinates, linear and angular velocities with respect to the gripper's local frame, formulated as $\mathbf{s}_i^b = (\mathbf{x}_i^r||\mathbf{v}_i^r||\mathbf{w}_i^r), \forall i \in \{1, ..., M\}$. Therefore, a state of the system can be defined as $\mathbf{s} = (\mathbf{s}_{grip}||\mathbf{s}_1^b||\mathbf{s}_2^b||...||\mathbf{s}_M^b)$.

The initial pose of the robot gripper is fixed at the same place for all tasks, while the initial positions of blocks are randomised and the initial orientations are always aligned with the world frame. For the i-th block, its initial position, (x_0^i, y_0^i), is uniformly sampled on the planar workspace within a square centred at the gripper position (x_0^{gr}, y_0^{gr}), i.e., $x_0^i \sim \mathcal{U}(x_0^{gr} - \delta, x_0^{gr} + \delta)$ and $y_0^i \sim \mathcal{U}(y_0^{gr} - \delta, y_0^{gr} + \delta)$, where δ is half of the square edge length. In this work, $\delta = 15\ cm$.

*2) **Actions**:* A low-level action comprises four elements, including the motion of the end effector in the Cartesian space $(\Delta x^{gr}, \Delta y^{gr}, \Delta z^{gr})$ and the gripper stroke width w_{finger}, denoted as $\mathbf{a}^{\mathcal{L}} = (\Delta x^{gr}||\Delta y^{gr}||\Delta z^{gr}||w_{finger})$. All dimensions are continuous within $[-1, 1]$. An exception is the Rotation Task (see section. VI-A2), where the agent is additionally allowed to rotate the gripper around its Z-axis.

As discussed in Section IV-B, there are N high-level actions related to the N steps, i.e., $\mathbf{a}^{\mathcal{H}} \in \{1, 2, ..., N\}$.

*3) **Low-level goals & reward function**:* Low-level goals are represented by the absolute Cartesian coordinates of blocks and the gripper as well as the gripper stroke width. Thus, given M blocks, a low-level goal is described as $\mathbf{g}^{\mathcal{L}} = (\mathbf{x}_{grip}^a||w_{finger}||\mathbf{x}_1^a||\mathbf{x}_2^a||...||\mathbf{x}_M^a)$.

We measure the difference between an achieved low-level goal $\mathbf{g}^{\mathcal{L}\prime}$ and a desired low-level goal $\mathbf{g}^{\mathcal{L}}$ via the L2-Norm of their difference, formulated as $||\mathbf{g}^{\mathcal{L}} - \mathbf{g}^{\mathcal{L}\prime}||_2$. Given a threshold ϵ_g, a predicate $f_{\mathbf{g}^{\mathcal{L}}}(\mathbf{s}') = [\ ||\mathbf{g}^{\mathcal{L}} - \mathbf{g}^{\mathcal{L}\prime}||_F < \epsilon_g\ ]$ is used to define a sparse reward function:

$$r^{\mathcal{L}}(\mathbf{s}, \mathbf{a}^{\mathcal{L}}, \mathbf{g}^{\mathcal{L}}) = \begin{cases} 0, & f_{\mathbf{g}^{\mathcal{L}}}(\mathbf{s}') = 1 \\ -1, & f_{\mathbf{g}^{\mathcal{L}}}(\mathbf{s}') = 0 \end{cases}$$

where $\mathbf{s}'$ is the state that occurs after an action $\mathbf{a}$ is executed at state $\mathbf{s}$. In this work, we set $\epsilon_g = 0.02$.

It is worth mentioning that most existing works represent goals for the block-stacking tasks with only the absolute Cartesian coordinates of the blocks [1], [2], [4], and we call such goals *block-informed goals*. In contrast in our work, the goals consist of not only the coordinates of blocks but also the gripper and its stroke width, which we call *block-gripper-informed*

goals. We notice through experiments (Section VII-A) that training with block-informed goals is inefficient and results in inconsistent behaviours. We hypothesise that this is due to the lack of information about gripper motion provided by the goals (and thus by rewards).

On the one hand, when rewards are only related to block positions, exploration is difficult in a block-stacking task with sparse rewards. If goals contain information about the gripper, then some rewards are expected to 'teach' the policy about the direct effects of actions, which are correspondingly gripper movements in this context. It is therefore expected that the policy would first learn to control its gripper to move around before learning to use it to manipulate blocks.

On the other hand, block-informed goals incline to induce a mixture of undesirable or unpredictable behaviours, including slicing, pushing, grasping and moving several objects simultaneously. We hypothesise that this is because block-informed goals (and corresponding rewards) encourage the robot to finish tasks, but ignore how the tasks are achieved. Thus, only the blocks' positions are considered insufficient for representing the goals in accomplishing tasks that favour certain behaviours.

We therefore adopt block-gripper-informed goals and demonstrate empirically (in Section VII-A) that such goals alleviate the severe exploration problem and produce more consistent behaviours.

*4) **High-level goals & reward function**:* We have experimented on two types of representations for high-level goals, including the block-gripper-informed representation (discussed in section IV-D3) as applied to the low-level goals and a binary representation.

The binary representation is a binary vector with N dimensions, where N is the number of steps for a task. The value of each dimension is computed using the predicate of the according low-level goal, $f_{\mathbf{g}^{\mathcal{L}}}$. Using the task given in Table I as an example which has 2 steps ($N = 2$) with two distinct low-level goals ($\mathbf{g}_1$ and $\mathbf{g}_2$), a high-level goal is then represented by a 2-dimensional binary vector: $\mathbf{g}^{\mathcal{H}} = (\ f_{\mathbf{g}_1^{\mathcal{L}}}, f_{\mathbf{g}_2^{\mathcal{L}}}\ )$, where $f_{\mathbf{g}_n^{\mathcal{L}}} \in \{0, 1\}$.

As an example, in Table I, a high-level goal with values $(1, 0)$ means $\mathbf{g}_1^{\mathcal{L}}$ should be achieved, while $(0, 1)$ means that $\mathbf{g}_2^{\mathcal{L}}$ should be achieved. The high-level reward function is defined simply by checking whether the desired high-level goal is achieved, using the element-wise equality with the actual achieved goal $\mathbf{g}^{\mathcal{H}\prime}$, as formulated below:

$$r^{\mathcal{H}}(\mathbf{s}, \mathbf{a}^{\mathcal{L}}, \mathbf{a}^{\mathcal{H}}, \mathbf{g}^{\mathcal{H}}) = \begin{cases} 0, & \mathbf{g}^{\mathcal{H}} = \mathbf{g}^{\mathcal{H}'} \\ -1, & \mathbf{g}^{\mathcal{H}} \neq \mathbf{g}^{\mathcal{H}'} \end{cases} \tag{1}$$

where $\mathbf{s}'$ is the state after a low-level action $\mathbf{a}^{\mathcal{L}}$ is executed at the last state s.

V. Training Approach

In this section, we first propose a deep learning version of the Intra-Option Learning algorithm [12], [13], named Deep Intra-Option Learning (DIOL), for training the high-level policy. Next, we propose an exploratory strategy to stabilise the parallel training, named Auto-Adjusting Exploration Strategy (AAES), and abstract demonstrations to improve the learning efficiency. Finally, we review the popular goal-relabelling method, Hindsight Experience Replay (HER), that boosts the efficiency for GRL problems [4].

For updating the low-level policy, we apply the Deep Deterministic Policy Gradient (DDPG [11]) algorithm following the implementation in [4], but with a second critic to reduce the overestimation error [30]. The pseudo-code of the training process is presented in Algorithm 1. Algorithm parameters are given in Appendix A. Mathematical notations are provided in the supplementary material.

Algorithm 1 Parallel training pseudo-codes

Input: maximum epochs, cycles and episodes M_0, M_1, M_2
Initialise DIOL (section V-A) and DDPG [11]
Initialise *AAES* (Section V-B)
for $epoch = 1$ to M_0 do
  for $cycle = 1$ to M_1 do
    for $episode = 1$ to M_2 do
      Sample a high-level goal
      for $t = 0$ to $T - 1$ do
        if *use_demonstrations* (section V-C, VII-D)
          Obtain the correct next low-level goal
        else
          Sample a low-level goal from the high-level policy
        end if
        while not *low_level_goal_achieved*
          Sample an action from the low-level policy with AAES
          Execute the low-level action and observe the next state
          Store the transition
        end while
      end for
    end for
    Perform *HER* on $R^{\mathcal{L}}$ with the "*episode*" strategy [4]
    Perform T_{opt} optimisation steps with DDPG [11]
    Perform T_{opt} optimisation steps with DIOL (Eq. 2, 3)
  end for
  Perform tests to obtain current low-level performance
  Update *AAES* (Section V-B, Eqs. 4, 5)
end for

A. Deep Intra-Option Learning (DIOL)

To train a high-level policy, a typical approach is to represent the policy by an option-value function $Q(s, o)$ updated by the classic Semi-MDP Q-Learning method, which has been adopted in recent hierarchical RL studies [8], [9]. It updates $Q(s, o)$ with a discounted return, only when an option terminates. This is not data efficient as it discards all the data generated by a low-level policy except at the point of termination. An alternative is the Intra-Option Learning (IOL) method, which learns more efficiently from data generated at every timestep [12], [13]. The original IOL update rule for tabular problems is defined as:

$$Q(s, o) \leftarrow Q(s, o) + \alpha\ [(r + \gamma\ U(s', o)) - Q(s, o)]$$

where α is a step size and $U(s', o)$ is the *option value upon arrival* defined as:

$$U(s', o) = (1 - \beta(s'))\ Q(s', o) + \beta(s') \max_{o' \in \mathcal{O}} Q(s', o')$$

We extend it to a goal-conditioned, experience replay-based mini-batch update for neural network approximators. In training, off-line transitions are uniformly drawn from the replay buffer D. A transition is denoted by $\xi = (\mathbf{s}, \mathbf{g}^{\mathcal{H}}, \mathbf{a}^{\mathcal{H}}, r, b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}}, \mathbf{s}')$, where, $\mathbf{s}$ is the encountered state, $\mathbf{g}^{\mathcal{H}}$ is the desired high-level goal, $\mathbf{a}^{\mathcal{H}}$ is the action (a low-level goal) selected by the policy, r is the reward, $b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}}$ is a binary value indicating whether the selected low-level goal is achieved and $\mathbf{s}'$ is the next state. The loss function for computing the gradient is:

$$L_{\theta_i} = \mathbb{E}_{\xi\sim\mathcal{U}(D)}[(r + \gamma U(\mathbf{s}', \mathbf{g}^{\mathcal{H}}, \mathbf{a}^{\mathcal{H}\prime}, b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}}; \theta_i^-) - Q(\mathbf{s}, \mathbf{g}, \mathbf{a}^{\mathcal{H}}; \theta_i)] \tag{2}$$

where $U(\mathbf{s}', \mathbf{g}^{\mathcal{H}}, \mathbf{a}^{\mathcal{H}\prime}, b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}}; \theta_i^-)$ is the *goal-conditioned option value upon arrival* estimated by the target network, defined as:

$$\begin{aligned} U(\cdot) = {} & (1 - b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}})\ Q(\mathbf{s}', \mathbf{g}^{\mathcal{H}}, \mathbf{a}^{\mathcal{H}}; \theta_i^-) \\ & +\ b_{\mathbf{s}'\mathbf{g}^{\mathcal{L}}} \max_{\mathbf{a}^{\mathcal{H}\prime}\in\mathcal{A}^{\mathcal{H}}} Q(\mathbf{s}', \mathbf{g}^{\mathcal{H}}, \mathbf{a}^{\mathcal{H}\prime}; \theta_i^-) \end{aligned} \tag{3}$$

where θ_i^- denotes the target network parameters (a partial or delayed copy of the parameters of the main network θ_i [11], [30], [31]).

Equations 2 and 3 are used to update the high-level policy in our framework. For better learning stability with neural network approximations, we employ two individual value networks, each of which has a target network, to reduce value overestimation as suggested by [30]. The estimated option value upon arrival (Equation 3) takes the minimum value between those computed by the two target networks. Both target networks are updated w.r.t. their main networks softly with a parameter $\tau \ll 1$, i.e., $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$ [11]. We name this algorithm 'Deep Intra-Option Learning' (DIOL), as it is a deep learning and experience replay extension of the original IOL algorithm [12].

B. Auto-Adjusting Exploration Strategy (AAES)

As mentioned in Section IV-C, when training policies at different levels of a hierarchical RL system in parallel, the transition of higher level MDPs is non-stationary due to the instability of the low-level policy caused by the random exploration nature [7]. Previous works typically avoid this issue by training two levels separately, keeping the low-level policy unchanged when training the high-level one [8], [10]. Separate training is a viable but costly solution, especially when the data-collection process is expensive (e.g., real-life tasks or complex simulations). One existing approach to this, Hierarchical Actor Critic (HAC), modifies the high-level transitions as if the low-level policy is an optimal policy [7]. However, it cannot fundamentally eliminate the non-stationarity.

Based on the theoretical analysis in section IV-C, we propose the AAES method, which adaptively reduces unnecessary exploration of the low-level policy according to its performance. AAES builds upon a common exploration strategy for continuous reinforcement learning agents [11]. Specifically, the agent will either sample a random action from a uniform distribution with a probability of α or takes an action (with a probability of $1-\alpha$) using the learnt policy $\pi(a|s,g)$ with added Gaussian noises according to $\mathcal{N}(0,\sigma^2)$. The policy can be then formalised as $\pi_b(a|s,g) = \pi(a|s,g) + \mathcal{N}(0,\sigma^2)$. In the former case, the agent explores the environment more aggressively by randomly sampling actions in a large search space globally. On the contrary, the noisy actions will explore more locally. On top of this, AAES adjusts α and σ after each epoch according to the testing success rate of each step:

$$\boldsymbol{\alpha}_{e+1} = c_\alpha\ (1 - \mathbf{S}_e);\ \boldsymbol{\sigma}_{e+1} = c_\sigma\ (1 - \mathbf{S}_e) \tag{4}$$

where $c_\alpha, c_\sigma \in (0, 1]$ are the upper bound constants of the random action probability and noise standard deviation; the subscript e denotes the epoch index; $\boldsymbol{\alpha}_{e+1}$ and $\boldsymbol{\sigma}_{e+1}$ are N-dimensional vectors of random action probabilities and noise standard deviations at epoch $e+1$; and $\mathbf{S}_e$ is a N-dimension vector of the averaged success rates of the N steps after epoch e. Equations 4 allow the agent to adjust its exploration adaptively such that an increase of the testing success rate, after the e-th epoch, will result in a decreased probability of taking random actions and a reduced deviation of action noise, in the $(e+1)$-th training epoch.

In other words, at the beginning of training, AAES assigns the highest probability of taking random actions and the highest deviation of sampling action noise, since the success rates are all 0. As the low-level policy becomes more skilled at achieving a particular step (as its success rate grows), AAES reduces the random action probability and action noise deviation related to that step. When a step is well-learnt by the agent (with its success rate approaching 1), AAES tends to stop exploration and only takes greedy actions without noises. This thus ensures the high-level policy to stably move forward to later steps while learning a task. The upper bound constants c_α and c_σ are empirically determined (see Appendix A).

In practice, directly computing $\boldsymbol{\alpha}_e$ and $\boldsymbol{\sigma}_e$ using the original success rate results in bumping changes of the two values that may not reflect the real performance of the policy. For example, if a success rate occasionally grows up significantly after an epoch, it is more possible that it is tested in a more familiar task distribution region than that it truly performs well in all cases. Thus, we smooth out the bumping changes using a delayed copy of the original success rate vector: $\mathbf{S}_e^-$. This value is updated slowly via

$$\mathbf{S}_{e+1}^- \leftarrow \tau_s\mathbf{S}_e + (1-\tau_s)\mathbf{S}_e^- \tag{5}$$

with $\tau_s \ll 1$. Finally, we use this delayed copy to compute $\boldsymbol{\alpha}_e$ and $\boldsymbol{\sigma}_e$, obtaining smoothly changing curves of the two values and thus more smoothly auto-adjusting exploration.

C. Abstract Demonstrations

We introduce an abstract form of demonstrations to accelerate the learning for multi-step tasks with sparse rewards. As mentioned, most existing works use human demonstrations at the trajectory level that can be difficult to obtain and are usually not readily available [2], [6], [16].

In this work, we provide the agent with the correct orders of steps at the symbolic level for achieving final outcomes, hence named 'abstract demonstrations'. For example, if a desired step is to place a blue block on top of a red block, the demonstration would be a sequence of steps: 1) grasping the blue block and 2) placing it on top of the red block. In our implementation, they are represented by different ordering of step indexes. Such abstract demonstrations are relatively easier to obtain, and resemble many human instructions in real-world scenarios, e.g., instructions of building a Lego house, assembling/disassembling a machine, etc.

Despite its simplicity, we consider that this form of demonstrations would benefit both levels. For the low-level policy, it serves as a pre-designed curriculum, guiding the low-level policy to learn to achieve goals in a reasonable order (i.e., from easy to hard). For the high-level policy, it serves as the right action sequences that lead to desired outcomes.

D. Hindsight Experience Replay (HER)

HER is a data augmentation method, which relabels rewards and desired goals of experienced transitions, aiming to improve the sample efficiency for sparse reward GRL problems [4]. A transition is collected after the system passes one timestep. In GRL, a transition commonly consists of a state, a desired goal, the next state, an achieved goal, and a reward. Before pushing the transitions of a full trajectory into a replay buffer, HER samples (k) goals using a sampling strategy. Then, another k synthetic trajectories are produced by copying the old trajectories and replace their desired goals with the ones sampled previously. Rewards for these new transitions are recomputed according to the goal-conditioned reward function.

We use the *episode* strategy for all low-level policies to sample goals, which samples the k goals uniformly within a collected trajectory, with $k = 4$ working reasonably well [4].

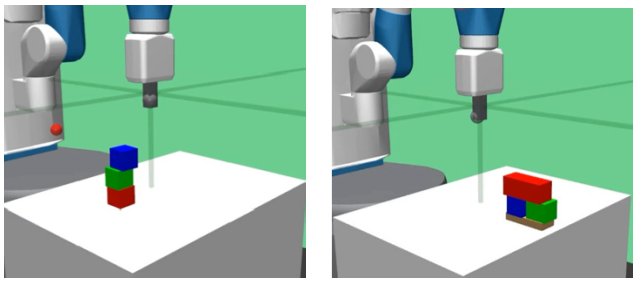
VI. Experiments Setup

This section introduces a set of block-stacking tasks experimented in this work and the training and testing procedures. To begin with, we declare two general assumptions used in this work. First, we assume that any given task to be solved by the agent can be decomposed into a finite number of intermediate steps (based on human prior). Second, we assume that there is always an achieved goal given a system state, which is the required assumption for using the hindsight experience replay technique [4].

A. Task configurations

We conduct simulation experiments on eight variants of block-stacking tasks in the Open AI Gym environment with the MuJoCo engine [32]. A 7-DOF Fetch robot is used in the simulation.

1) Basic tasks: Table II lists the configurations of four basic block-stacking tasks, where a robot needs to stack some cuboid blocks on top of one another to form a tower. Three blocks of different colours (Red, Blue, Green) are used in these tasks.



(a) (b)

Figure 3: (a) The 'B→G→R' outcome of the fourth basic task; (b) the 'R→BG' outcome of the pyramid task.

Each task has a different number of desired outcomes (high-level goals), and thus requires different number of steps to accomplish them all.

For example, 'B→G→R' is one desired planning outcome that denotes the desired top-down order of three blocks of task 4 (Fig. 3a). This 'B→G→R' order would require five steps: 1) grasping block G, 2) placing G on the top of R, 3) grasping block B, 4) placing B on the top of G and R, and 5) moving the grip away. Since the robot also has to learn the other order 'G→B→R', there are in total 10 steps in task 4. 'Training timesteps' and 'Testing timesteps' represent the number of actions the robot can take before the system resets (i.e., the length of a training or testing episode).

Among the four basic tasks, we analyse the effect of different representations of goals with tasks 1 and 2. The effects of the AAES, parallel training and abstract demonstrations are validated on only task 2. All basic tasks are used to evaluate the ability of learning diverse desired planning outcomes with a single policy.

2) Additional tasks: We also propose four additional tasks in different configurations that are considered more complex than the basic block-stacking task. The first one is designed to demonstrate that the proposed UOF can handle tasks of different stacking types, while the second one is the same with the task 1 in Table II, except that the gripper is allowed to rotate along the Z-axis, providing one more degree-of-freedom in control. The third and fourth tasks are designed to test the generalisation ability of our method by randomising the sizes of blocks. For the sake of clarity, the first task is termed as 'pyramid task', the second task is termed as 'rotation task' and third and forth are termed as 'randomised block size (RBS) tasks'.

The pyramid task is more difficult than the basic tasks as it requires the robot to place one or two blocks at one level (compare Fig. 3a and 3b). In Table III, 'R→BG' in the 'Desired outcomes' column denotes an outcome where a blue and a green blocks are placed at the bottom closely, and a red block longer than the others is placed on top of them (shown by Fig. 3b). This would require seven steps to finish: 1) grasping block G, 2) placing G on the front side of the tray, 3) grasping block B, 4) placing B on the back side of the tray close to G, 5) grasping block R, 6) placing R on top of B and G, and 7) moving the grip away. With the other order,

Task	Blocks	No. of steps	Desired outcomes	Training epochs	Training timesteps	Testing timesteps
1	R, B	3	B→R	150	25	50
2	R, B, G	6	B→R; G→R	800	25	50
3	R, B, G	15	B→R; B→G; R→B; R→G; G→R; G→B	1000	25	50
4	R, B, G	10	B→G→R; G→B→R	1500	40	60

Table II: Basic Block-stacking Tasks.

Task	Blocks	Control	No. of steps	Desired outcomes	Training epochs	Training timesteps	Testing timesteps
Pyramid	R, B, G	Position, finger	14	BG→R; R→BG	2000	60	80
Rotation	R, B	Position, finger, Z-rotation	3	B→R	300	25	50
RBS 1	R, B	Position, finger	3	B→R	Test only	-	50
RBS 2	R, B, G	Position, finger	6	B→R; G→R	Test only	-	50

RBS: random block size.

Table III: Additional Block-stacking Task.

'BG→R', the total step count is 14.

The rotation task is more difficult as the extra degree of freedom enlarges both the state and action spaces of the agent. In particular, we add the Euler angles of the end-effector and blocks into the agent's state representation, which, for task 1, adds 9 more dimensions into the state space. As such, we train the agent for 300 epochs, which is twice longer than the original basic task 1. Except for the state and action spaces and the training time, all other definitions of the rotation task and algorithm remain the same, as presented in subsection IV-D and Appendix A.

The RBS tasks are modified from tasks 1 and 2 presented in the basic tasks (Table II). They aim to test the zero-shot generalisation ability of the trained policies. Specifically, they are the same with the basic tasks except that, at the beginning of a test episode, the block sizes are sampled uniformly from an interval of $[15cm, 35cm]$. Since the agent is only trained with a fixed block size of $25cm$, its performance on the RBS tasks without fine-tuning will showcase its zero-shot generalisation ability (see results and discussion in section VII-G).

B. Training Details

The training process is summarised in Algorithm 1 in supplementary material, with detailed parameter settings given in Appendix A.

Training: Both levels are trained in parallel, The training process consists of Epochs, Cycles and Episodes. An epoch includes 50 cycles, and each cycle is composed of 16 episodes. States and goals are normalised using a running average of the mean and variance. At the end of each cycle, we apply the HER method using the *episode* strategy with $k = 4$ [4] to relabel the low-level trajectories. Then, along with the original trajectories, they are merged into the low-level replay buffer. For the networks at both levels, samples are uniformly drawn from the buffers to perform 40 optimiser steps after each cycle.

Testing: For performance evaluation, we test both levels at the end of each epoch with only greedy actions for 30 episodes, and record the average returns, success rates and time steps towards the completion.

It is worth mentioning that the performance of the low-level policy is related to the abstract demonstrations. This is because of the step inter-dependency, as some steps require others to be reached in advance. In a testing episode, the low-level goals are passed to the low-level policy according to the abstract demonstrations.

For the high-level policy, its performance is evaluated regardless of how well the low-level policy performs at the time of testing. This is different from testing its stand-alone planning performance using an optimal low-level policy, since we aim to improve the parallel training process instead of separate training.

The numbers of training epochs, training and testing timesteps of each task differ (see Table II). All basic tasks and the rotation task were run with 3 random seeds to calculate the means and deviations of the results, while the Pyramid task was run with one seed. The experiment is computationally expensive and takes a long time to run, ranging from 2 days to a week in our work. We use a workstation with an Intel i7-8700 CPU and a Nvidia RTX-2080 GPU.

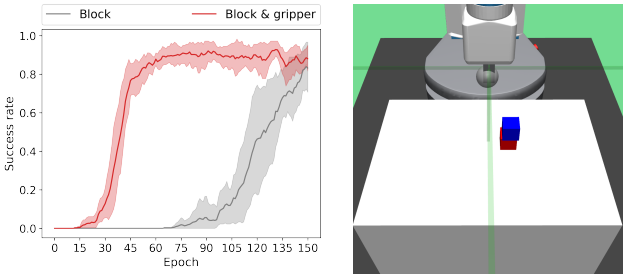
VII. Results

We evaluate the performance of the following aspects, namely, representations of goals, AAES, abstract demonstrations, and parallel training. We also give a comparison with a state-of-the-art algorithm, HAC, and experiments on the additional tasks. The main performance metric used in this section is the success rate, while we also provide the curves of test returns in the supplementary material.

A. Representations of Goals

1) ***Low-level goals****:* To evaluate the performance difference introduced by the proposed low-level goal representation, named block-gripper-informed goals (defined in section IV-D3), we perform a comparative study with the block-informed goals used in previous works [1], [2], [4]. Since this experiment is to evaluate the low-level control performance, only low-level policies were trained, without using AAES. In block-gripper-informed cases, desired low-level goals are provided in the correct order; while in the block-informed cases, desired goals (blue block position) were uniformly

sampled on the table, in the air or on top of the red block, as this is the only way to succeed without using kinematic-level demonstrations, suggested by [4].



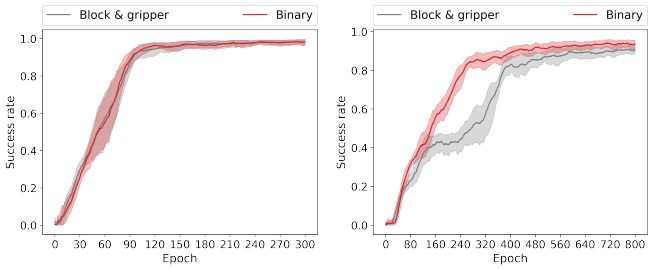
(a) (b)

Figure 4: (a) Average success rates of low-level kinematic control of the final step of task 1 with different goal representations; (b) Visualisation of the step.

In Fig. 4a, the red line depicts the success rate of block-gripper-informed goals of the final step ('Moving Gripper Away While Keeping The Blocks Stacked', as shown in Fig. 4b), and the grey line depicts the success rate of block-informed goals for stacking the blue block on top of the red block.

The grey line shows a higher variance and grows much more slowly compared to the red line. This means that using block-gripper-informed goals converges significantly faster with a lower variance. This result indicates that learning from only block-informed goals is inefficient. It is clear that introducing the extra gripper information in describing the goals effectively increases the performance.

*2) **High-level goals**:* This subsection analyses the performance of the high-level policy with the high-level goals in the block-gripper-informed representation and in the binary representation (see Section IV-D).



(a) Task 1 (b) Task 2

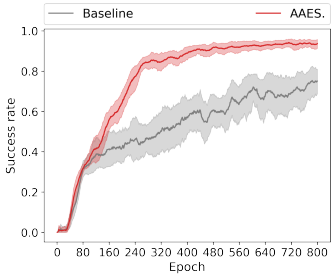
Figure 5: Average success rates with block-gripper-informed and binary high-level goals.

Fig. 5 displays the average testing success rates of the high-level policy using the two high-level goal representations. Comparative experiments were performed on basic task 1 and 2 (see Table II). The binary representation outperforms the block-gripper-informed representation. Though Fig. 5a shows no performance difference, Fig. 5b shows a clear advantage of deploying the binary representation (red line) over the block-gripper-informed representation (grey line) with task 2.

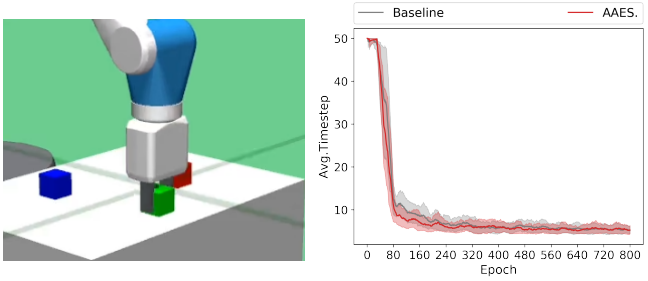
Based on the results above, in the rest of the paper, we use the binary representation to describe high-level goals.

B. Auto-adjusting Exploration Strategy (AAES)

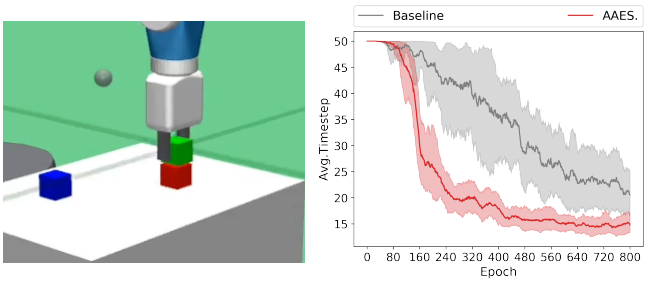
This subsection evaluates the AAES for parallel training. The evaluations are focused on the high-level planning policy, as the AAES is introduced to stabilise the high-level MDP



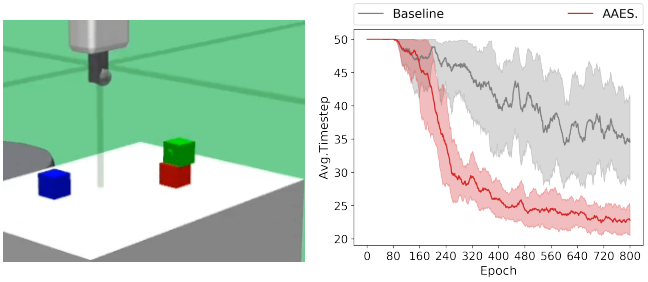
(a)



(b)



(c)



(d)

Figure 6: (a): Average success rate of high-level planning in task 2; (b)-(d): Three consecutive steps (left column) of task 2 (grasping blue block, placing on the red block, and leaving) and the average timesteps required for completion (right column). The red lines represent the performance with AAES and the grey lines depict the baseline without AAES.

transitions. The baseline performance is obtained with a non-adaptive version of the AAES, where $\boldsymbol{\alpha}$ and $\boldsymbol{\sigma}$ are kept equal to the constant bounds ($c_\alpha = 0.2$, $c_\sigma = 0.05$, section IV in supplementary material), as an exploration strategy equivalent to that in [4].

Fig. 6a displays the average success rates of task 2 (see Table II). It shows that using AAES (red line) improves the learning efficiency over the baseline (grey line) and reaches a success rate of 0.941 ± 0.003. Moreover, it starts to diverge from the grey line at about epoch 120, as expected, showing that AAES is considerably more effective for later steps.

Figs. 6b–6d display the average timesteps required to complete 3 consecutive steps: "Grasping Green Block", "Placing Green Block On Red Block", and "Moving Gripper Away". From the start to the later steps, (correspondingly from Figs. 6b to 6d), one can observe that, as expected, the least required number of timesteps for finishing a task step increases (around 5, 15 and 23 timesteps respectively) after the training stabilises. The advantage of deploying AAES becomes clearer gradually through the three steps. In particular, for the first step (Fig. 6b), it shows no clear improvement, while Figs. 6c and 6d demonstrate clear reductions of about 10 and 20 timesteps on average relative to the baseline at the end of the training process. In addition to the reduction of required timesteps, AAES considerably improves the speed of convergence for learning the later steps. As shown in Figs. 6a, 6c and 6d, one can see that the red lines converge at around the 450-th epoch, while the grey lines do not show any obvious trend of reaching convergence within 800 epochs.

In short, these results empirically prove that, for tasks comprising inter-dependent steps, the proposed AAES improves the learning for the high-level planning in parallel training, especially for later steps, in terms of convergence speed and learnt performance.

C. Parallel and separate training

This subsection analyses the performance of high-level planning, when training both levels in parallel and separately. Fig. 7 displays the average success rates of the high-level policy in task 2 (see Table II). The grey line depicts the success rate of high-level planning trained in 300 epochs with a pre-trained low-level policy; and the red line depicts the one that was trained in parallel with the low-level policy from scratch, aided by the AAES.

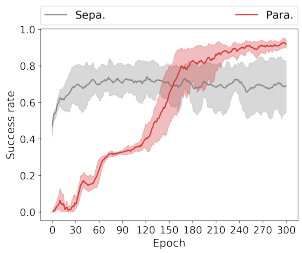


Figure 7: Average success rates of high-level planning. **Sepa.**: Trained with a pre-trained low-level policy; **Para.**: Parallel training with low-level policy from scratch.

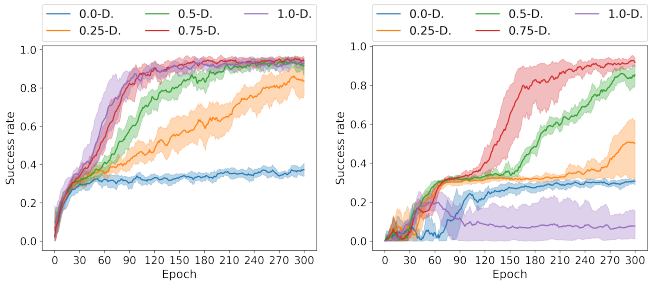
In Fig. 7, the grey line starts with a higher success rate (around 0.4), but has no obvious improvement afterwards (stay at around 0.661 with a standard deviation of 0.152). In comparison, the red line starts from zero because both the policies start from scratch. However, it surpasses the grey line at around 150-th epoch and converges at a higher average success rate (0.896) with a lower deviation (0.017).

Training a planning policy with a pre-trained control policy, as deployed in [8], [10], starts learning faster. However, parallel training achieves a higher and more stable performance. In addition, parallel training is more time efficient without the need to pre-train the low-level policy. For example, the low-level policy for task 2 needs to be pre-trained for 300 epochs, thus, parallel training is roughly twice more time-efficient than separate training in this case.

D. Abstract demonstrations

This subsection discusses the effect of abstract demonstrations. Fig. 8 shows the success rates of both low-level (Fig. 8a) and high-level policies (Fig.8b) given different numbers of demonstrated episodes.

The number of episodes to be provided with demonstrations is specified by a proportion of each cycle, denoted by x. Each cycle has 16 episodes in total. For instance, when $x = 0.25$, the number of demonstrated episodes within each training cycle will be $16 \times x = 4$. In that case, each cycle would contain 4 episodes of high-level actions from the provided demonstrations. We conducted experiments with various values of x: $\{0, 0.25, 0.5, 0.75, 1\}$. All cases are trained in parallel with the AAES.



(a) Low-level (b) High-level

Figure 8: Average success rates with different proportions of demonstrated episodes in the task 2. 0.0-D, 0.25-D, 0.5-D, 0.75-D, and 1.0-D denote the respective proportions of demonstrations added in the episodes.

Overall, Fig. 8 shows that, given half of the episodes being demonstrated (green lines in both subfigures), both levels reach a success rate that roughly triples those without demonstrations (blue lines in both subfigures). This indicates that abstract demonstrations significantly benefit both low-level and high-level policies.

Fig. 8a shows that, for the low-level control policy, demonstrating more than half of the training episodes does not achieve much more further improvements on performance, as the green, red and purple lines grow closely together. Besides,

demonstrating 0.75 proportion of the episodes did further accelerate the convergence, but increasing upon 0.75 did not.

On the other hand, Fig. 8b shows a different phenomenon that too many demonstrations result in a destructed high-level policy (purple line). This is because an RL algorithm needs not only rewards, but also random exploration to collect experiences without rewards for distinguishing good behaviours from the bad ones [29]. Our result also shows that exploration and exploitation for the high-level policy was well-balanced by selecting 0.75 proportion of the episodes to be demonstrated.

On the contrary, providing full demonstration to the low-level policy dose not degrade its performance. This is because the given demonstrations are the correct sequences of low-level goals, and this does not directly influence the exploration behaviour of the low-level policy, i.e., what gripper control actions to be selected. Instead, adding more abstract demonstrations will further improve the low-level performance as shown in Fig. 8a, because it serves as a kind of curriculum and frees the low-level policy from learning the dependencies between low-level goals from scratch.

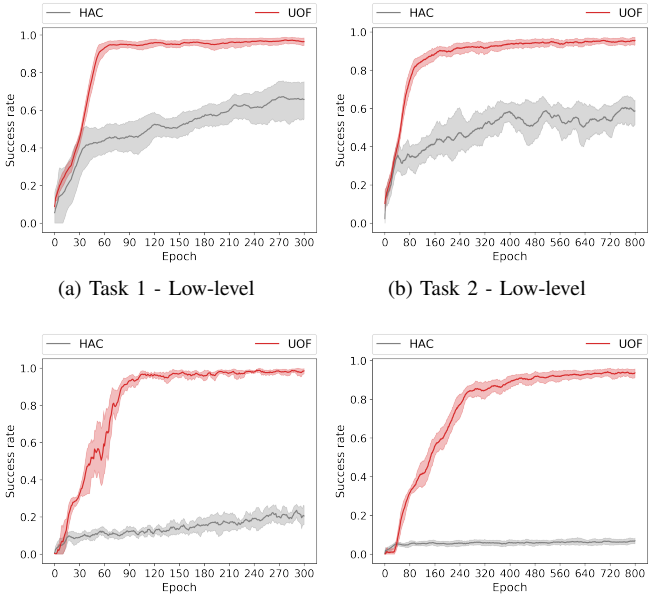
In short, these results prove that abstract demonstrations, which can be easily obtained, can significantly accelerate the learning efficiency of both low level (kinematic) control and high level (symbolic) planning.

E. Comparison with Hierarchical Actor Critic

We compare our method with Hierarchical Actor Critic (HAC) [7], which is considered a state-of-the-art goal-conditioned hierarchical reinforcement learning algorithm, and suits the multi-outcome tasks as it is goal-conditioned in all of its hierarchies. In HAC, three modifications are applied to the collected transitions to deal with the non-stationary transition problem described in section IV-C. However, it can not be directly applied to the multi-step settings discussed in our work because the high-level policy in HAC produces a continuous multi-dimensional vector as a low-level goal. This cannot be changed as their transition modification approaches depend on continuous high-level actions.

Note that, though the comparison is not perfectly fair, it provides an intuition of how hard it is to learn in a continuous goal space. The UOF proposed in this work starts by task decomposition and only learns the sequences of sub-tasks at the higher level, while HAC was designed to learn in continuous goal spaces at every level. To ensure the comparison to be as fair as possible, we also provide abstract demonstrations to the HAC agent[2].

Results show that, for tasks 1 and 2, the average success rate of the low-level policy of the HAC agent can only reach around 0.5, while the high-level policy success rates are even lower (grey lines in Fig. 9). This means the HAC agent can only learn the first step of the task (grasping a block), while our method (red lines) can solve all tasks satisfactorily. This is expected since HAC is not designed to handle multi-step tasks [7]. This implies that task decomposition, which may be done automatically in future research, is essential for long horizon manipulation tasks as it enables reasoning in a discrete space with vastly lower dimensionality. On the other hand, though HAC provides techniques to alleviate the non-stationary transition problem, it does not completely erase the exploratory behaviours of the low-level policy as what AAES does when a step is well-mastered.



(a) Task 1 - Low-level (b) Task 2 - Low-level

(c) Task 1 - High-level (d) Task 2 - High-level

Figure 9: Average success rates of HAC and UOF.

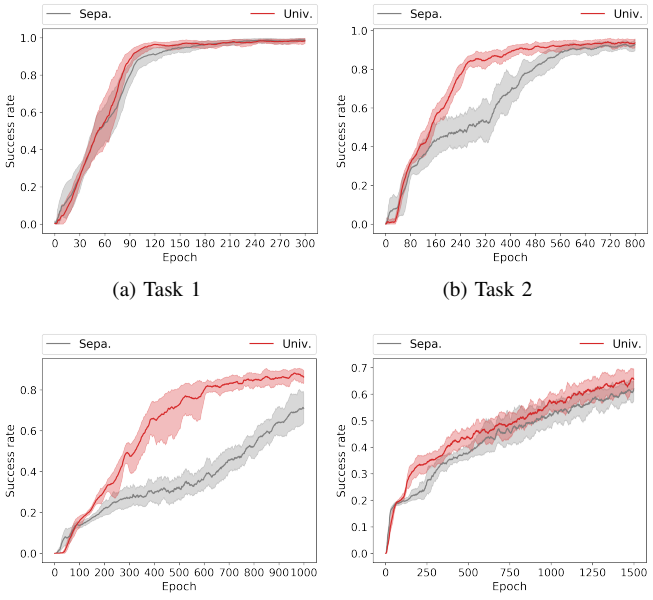
F. Learning Diverse Combinatorial Results

This subsection evaluates the high-level planning performance obtained by a universal high-level policy and separated policies. For a universal policy, we merge data into one replay buffer; while for separated policies, we store data of different final goals in different replay buffers, each of which is used to train a corresponding policy. The replay buffers have a capacity of $1e6$ datapoints and discard the oldest data, when new data comes in if it is full. Thus, they are trained with the same amount of data, but each separate policy only learns one task.

Fig. 10 shows that, for tasks 1-4 (Fig. 10a to 10d), the planning performance is not sacrificed when training with only one universal policy (red lines). Instead, it even surpasses separate policies (grey lines). Moreover, the more final planning outcomes needed to be learned, the greater the advantage of universal policy over separated policies (task 1 has 1 final outcome; task 2 and task 4 have 2; and task 3 has 6).

One reason for the better performance with the universal policy could be due to the knowledge sharing mechanism within a single policy that allows the learnt experience to be shared among different desired planning outcomes, while

[2]Abstract demonstrations are represented by the correct sequences of steps towards some desired final outcomes. They are lists of integer indices, with each index related to a set of sub-goals of a unique step. Thus, given the simulator, we can obtain a sub-goal vector related to a step by selecting an index. As for the UOF agent, the high-level policy takes the indices as demonstrated actions, while for the HAC agent, it takes the multi-dimensional sub-goal vectors as demonstrated actions.



(a) Task 1 (b) Task 2

(c) Task 3 (d) Task 4

Figure 10: Average success rates of planning over multiple steps with universal and separated policies for the four tasks. **Univ**: single universal policy; **Sepa**: separated policies.

separated policies will have to learn everything from scratch for each desired planning outcome.

However, as shown in Fig 10d, although knowledge sharing improves the sampling efficiency for learning multiple outcomes, its advantage is less obvious when solving longer horizon tasks (task 4). We consider that it is because, with more data required by longer horizon tasks, using separate policies for each of the final outcomes will reduce the amount of required training data. Therefore, for universal policies, the advantage of knowledge sharing is counter-balanced by the increased data requirement.

On the other hand, another advantage of such a universal policy is the reduced memory usages. A clear difference of memory usages for training these policies tasks is shown in Table IV. The "Step Num." column shows the number of steps that are required for each task.

The 'Buffer' column specifies the memory size of filled-up replay buffers, each of which contains $1e6$ transitions (datapoints). This column shows an increasing buffer size as the task becomes more complex (from rows 1 to 4), and a noticeable increased memory occupancy with separated policies than universal policies (in total, from $\sim$ 1.72G to $\sim$ 2.60G for a universal policy, and from $\sim$ 5.16G to $\sim$ 36.30G for separated policies).

The 'Network' column shows the memory consumption of neural network parameters of these policies. For separated policies, the increment of memory needed to store these parameters is also noticeably greater than a universal policy. Since we use Multi-Layer Perceptrons (MLPs), these networks are relatively small. However, such an increment of memory consumption would become considerably more severe when using more complicated neural networks.

Table IV: Memory Usages for Training High-Level Policies

Task	Step Num.	Policy	Network	Buffer
1	3	Universal	$\sim$1.08M	$\sim$1.72G
		Separate	$\sim$3.25M	$\sim$5.16G
2	6	Universal	$\sim$1.12M	$\sim$2.42G
		Separate	$\sim$6.73M	$\sim$14.52G
3	15	Universal	$\sim$1.12M	$\sim$2.42G
		Separate	$\sim$16.80M	$\sim$36.30G
4	10	Universal	$\sim$1.31M	$\sim$2.60G
		Separate	$\sim$11.30M	$\sim$26.00G

Fig. 10 and Table IV altogether suggest that training a single universal policy without sacrificing its performance has clear advantages in terms of better knowledge sharing and lower memory consumption.

G. Additional task

This subsection examines the performance of our framework on the additional block-stacking tasks. Specifically, we further test our method on four additional tasks: the pyramid task, the rotation task and two randomised block size (RBS) tasks as shown by Table III.

For the pyramid task, results show that the low-level policy achieved 0.6 average success rate and the high-level achieved 0.4. This is expected because each of the two final desired outcomes of the additional task requires 7 consecutive steps to finish, while those of the hardest basic task (the fourth in Table II) only require 5.

For the rotation task, results show that the low-level policy achieved 0.95 average success rate and the high-level achieved 0.8. The rotation task is the same as task 2 in Table II except that the gripper has one more degree of freedom (DoF). This extra DoF enlarges the search space of solution and thus doubles the training time required for the agent to converge.

Finally, the results of the RBS tasks show that the trained policies can achieved an average success rate of 0.66 in both tasks in 30 testing episodes for each step.

This successful zero-shot generalisation is potentially due to the use of block-gripper-informed-goal representation (see section IV-D3). After training, the agent develops a connection between its goals, observations and actions in a non-trivial way, such that it is able to match the selected gripper width (action) according to the block size (goals and observations).

We have also observed some failures when the blocks are too small or too large (near the extremes of the sampling interval $[15cm, 35cm]$. However, the zero-shot generalisation performance is considered satisfactory and a fine-tuning process can be conducted for a different size of blocks if higher performance is demanded.

Overall, these results demonstrate that the UOF is able to handle different types of tasks that can be more complicated and that our method can achieve a satisfactory level of zero-shot generalisation performance. Improving the performance would require more efforts such as more complicated neural networks, more informative representation of goals and fine-tuning on tasks with different target objects.

VIII. Conclusions

In this paper, we propose a hierarchical reinforcement learning framework, the Universal Option Framework (UOF), to formalise multi-step manipulation tasks learning more universally. We then extend the Intro-Option Learning algorithm to a deep learning and experience replay version for training the high-level policy. We also propose the Auto-Adjusting Exploration Strategy and Abstract Demonstrations to stabilise and accelerate parallel training. The UOF is tested on a set of block-stacking tasks with a 7-DOF Fetch robot in the Gym simulation environment [32].

We empirically demonstrate that our method is able to learn multiple combinatorial outcomes from multi-step manipulation tasks with universal low- and high-level policies. Compared to separate and repetitive training, our parallel training considerably reduces the memory consumption and computational costs.

Concretely, the parallel training is stabilised bottom-up by the proposed auto-adjusting exploration strategy at the low-level and accelerated top-down by an abstract form of demonstrations that provides the correct orders of steps at the high level.

There are two main limitations of the proposed method. First, as an extended version of goal-conditioned reinforcement learning, UOF requires a mechanism for goal generation [4]. Such a requirement prevents our method from handling tasks that cannot generate goals in advance. Another limitation is that our method requires users to manually separate the space of sub-goals in a way that each of the subsets relates to a high-level goal (a task step).

Future research will try to address the two aforementioned limitations. First, to improve the generalisability of the UOF, it is essential to develop a more generalised goal generation mechanism. For example, for cases where goals could only be represented by images or in other complex forms, a goal generation mechanism may be developed based on Generative Neural Networks (GNNs). Regarding the second limitation, it is valuable to develop a technique that can automatically define a set of meaningful sub-goals, such that these sub-goals correspond to some critical steps for the overall task. Besides, we will also try to improve the degree of knowledge integration of universal policies, either for planning or kinematic control, by, e.g., learning from different types of reward functions.

Appendix A
Algorithm and Parameter Settings

This section elaborates the implementation details of the training algorithm used in this work. Experimental programs, including environments and algorithms, are based on Python. Simulation environments are adapted from the Open AI Gym environments [32]. Neural network implementation is based on the PyTorch library [33]. The parallel training procedure is summarised in the Algorithm 1 in supplementary material.

The goal-conditioned low-level policy of a universal option is trained using the DDPG algorithm [11], while the goal-conditioned high-level policy is trained using the DIOL proposed in Section V-A. They both use a secondary critic to reduce value estimation error [30], with the discount factor $\gamma = 0.98$.

These components are represented by MLPs of the same size (3×256), activated by ReLU. The final layer of the low-level actor network is activated by Hyperbolic Tangent (Tanh); final layers of the low-level critics and option-value function do not use activation functions.

The states and goals are concatenated as the inputs for the policy and the option-value networks. Low-level actions, states and goals are concatenated as the inputs of the critic networks of the low-level policies. The states and goals are normalised using a running average of the mean and variance.

The replay buffers for both levels have the same size of $1e6$. The networks are optimised using Adam with the same learning rate of $1e-3$ and batch size of 128. All networks take 40 optimiser steps after each cycle. Estimated target action values for updating low-level policies are clipped within $[-25, 0]$; estimated target option values for updating high-level policies are clipped within $[-t, 0]$ where t is the maximal training timesteps of an episode (see Table II). After each optimiser step, corresponding target networks are updated softly with $\tau = 0.1$.

The AAES for the low-level policy uses constant upper-bounds $c_\alpha = 0.2$ and $c_\sigma = 0.05$. The copy of the performance $S^-_{n,e}$ is updated with $\tau_s = 0.05$. These three values are selected according to comparative experiments, with resultant figures given in section IV of the supplementary materials.

The high-level policy uses an episode-wise decaying-ϵ-strategy modified from [31], with ϵ being decayed using the following equation:

$$\epsilon_i \leftarrow \epsilon^- + (\epsilon^+ - \epsilon^-)\ e^{\frac{-i}{\rho}}$$

where ϵ^+ is the upper-bound, ϵ^- is the lower-bound, e is the natural exponential base, i is the total number of past episodes and ρ is the decaying efficiency parameter. In all tasks, we use $\epsilon^+ = 1.0$ and $\epsilon^- = 0.02$. For basic tasks 1 and 2, we set $\rho = 3e4$; for basic task 3, $\rho = 8e5$; for basic task 4, $\rho = 1e6$; for the additional task, $\rho = 2e6$;. Note that ϵ does not change within an episode.

Acknowledgment

The authors thank the China Scholarship Council (CSC) for financially supporting Xintong Yang in his PhD programme.

References

[1] Y. Duan, M. Andrychowicz, B. Stadie, O. J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba, "One-shot imitation learning", in *NIPS*, 2017, pp. 1087–1098.

[2] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming exploration in reinforcement learning with demonstrations", in *ICRA*, 2018, pp. 6292–6299.

[3] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators", in *ICML*, 2015, pp. 1312–1320.

[4] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight experience replay", in *NIPS*, 2017, pp. 5048–5058.

[5] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, "Data-efficient deep reinforcement learning for dexterous manipulation", *ArXiv:1704.03073*, 2017.

[6] B. Piot, M. Geist, and O. Pietquin, "Bridging the gap between imitation learning and inverse reinforcement learning", *IEEE transactions on neural networks and learning systems*, vol. 28, no. 8, pp. 1814–1826, 2016.

[7] A. Levy, G. Konidaris, R. Platt, and K. Saenko, "Learning multi-level hierarchies with hindsight", *ICLR*, 2019.

[8] O. Nachum, S. S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning", in *NIPS*, 2018, pp. 3303–3313.

[9] Y. Jiang, S. Gu, K. P. Murphy, and C. Finn, "Language as an abstraction for hierarchical deep reinforcement learning", in *NIPS*, 2019, pp. 9414–9426.

[10] X. B. Peng, M. Chang, G. Zhang, P. Abbeel, and S. Levine, "Mcp: Learning composable hierarchical control with multiplicative compositional policies", in *NIPS*, 2019, pp. 3681–3692.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning", *ICLR*, 2016.

[12] R. S. Sutton, D. Precup, and S. P. Singh, "Intra-option learning about temporally abstract actions.", in *ICML*, vol. 98, 1998, pp. 556–564.

[13] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning", *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[14] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman, "Meta learning shared hierarchies", *ICLR*, 2018.

[15] A. C. Li, C. Florensa, I. Clavera, and P. Abbeel, "Sub-policy adaptation for hierarchical reinforcement learning", *ICML*, 2019.

[16] N. Dilokthanakul, C. Kaplanis, N. Pawlowski, and M. Shanahan, "Feature control as intrinsic motivation for hierarchical reinforcement learning", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 11, pp. 3409–3418, 2019.

[17] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning", in *ICML*, 2017, pp. 3540–3549.

[18] I. J. Sledge, M. S. Emigh, and J. C. Príncipe, "Guided policy exploration for markov decision processes using an uncertainty-based value-of-information criterion", *IEEE transactions on neural networks and learning systems*, vol. 29, no. 6, pp. 2080–2098, 2018.

[19] Z. Yang, K. Merrick, L. Jin, and H. A. Abbass, "Hierarchical deep reinforcement learning for continuous action control", *IEEE transactions on neural networks and learning systems*, vol. 29, no. 11, pp. 5174–5184, 2018.

[20] D. Foster and P. Dayan, "Structure in the space of value functions", *Machine Learning*, vol. 49, no. 2-3, pp. 325–346, 2002.

[21] R. S. Sutton, J. Modayil, M. D. T. Degris, P. M. Pilarski, and A. White, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction", in *AAMAS*, 2011.

[22] M. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. Van de Wiele, V. Mnih, N. Heess, and J. T. Springenberg, "Learning by playing-solving sparse reward tasks from scratch", *ICML*, 2018.

[23] A. Nair, S. Bahl, A. Khazatsky, V. Pong, G. Berseth, and S. Levine, "Contextual imagined goals for self-supervised robotic learning", *CoRL*, 2019.

[24] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, S. Thrun, and R. C. Arkin, *Principles of robot motion: Theory, algorithms, and implementation*. MIT press, 2005.

[25] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach.", in *Robotics: Science and systems*, Ann Arbor, MI, USA, vol. 12, 2016, p. 00 052.

[26] S. J. Levine, "Monitoring the execution of temporal plans for robotic systems", PhD thesis, Massachusetts Institute of Technology, 2012.

[27] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains", *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[28] M. P. Deisenroth, C. E. Rasmussen, and D. Fox, "Learning to control a low-cost manipulator using data-efficient reinforcement learning", *Robotics: Science and Systems VII*, pp. 57–64, 2011.

[29] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[30] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods", *ICML*, 2018.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, p. 529, 2015.

[32] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, and P. Welinder, "Multi-goal reinforcement learning: Challenging robotics environments and request for research", *ArXiv:1802.09464*, 2018.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *NIPS*, 2019, pp. 8024–8035.