Department of Electrical Engineering
Indian Institute of Technology, Madras
Chennai - 600036, India

Adaptive Augmentation of Option Indices in Hierarchical Reinforcement Learning

Dual Degree Project Report

Submitted by

Akash Reddy A

EE17B001

in fulfillment of requirements
for the award of the Dual Degree of

Bachelor of Technology

and

Master of Technology

in

Electrical Engineering

June 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **Adaptive Augmentation of Option Indices in Hierarchical Reinforcement Learning**, submitted by **Akash Reddy**, to the Indian Institute of Technology, Madras, for the award of the degree of **Dual Degree in Electrical Engineering**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Balaraman Ravindran
Research Guide
Professor
Dept. of Computer Science and Engineering
Robert Bosch Center for Data Science and AI
IIT Madras, 600 036

Place: Chennai
Date: 18 June 2022

Acknowledgements

I would like to express my heartfelt gratitude, first and foremost, to Prof. Balaraman Ravindran, Department of Computer Science and Engineering, IIT Madras, for offering me his exceptional guidance coming from the cumulative knowledge of decades. His encouragement and understanding have helped me bolster my understanding of this field, and shape this final year project into what it stands to be today. I would also like to thank Prof. Arun Rajkumar and Prof. Nandan Sudarsanam for their valuable time and inputs as panel members during the mid-year review of this project.

I would next like to thank Pradeep Shenoy, Kushal Chauhan, and Soumya Chatterjee of Google Research India, for showing me interesting research directions, providing resources, helping to clarify ideas, and even taking the time out to assist me with implementation in certain tough spots.

I would like to thank my family and friends for providing a supportive ecosystem in which I can continue to pursue my goals, professional and personal. Heartfelt gratitude also goes out to the Electrical Engineering department of IIT Madras, and the institution of IIT Madras as a whole, for providing crucial exposure and a valuable set of peers who have exposed me to a variety of learning experiences, and also the opportunity to work on my Master's thesis.

Finally, I would also like to extend my gratitude towards my colleagues Nishant Prabhu and Sawni Gupta, who provided me with access to computational resources when I needed them.

Abstract

In hierarchical reinforcement learning, a task is broken down into several subtasks for learning. This promotes easier learning of long-horizon tasks, as well as allows for reuse of subtasks across several related tasks. This reuse finds strong utility in a continual learning setting, where a stream of tasks are to be solved by the agent while retaining information on previous tasks and potentially reusing it for future tasks, a concept known as *positive forward transfer*. A continual learning agent in real life might be expected to have a large repository of knowledge from solving previous tasks throughout its lifetime. If there is such a large library of skills (or options, as they are called in the Options framework), there would be no need to re-learn the subtasks end-to-end for each task, provided the tasks themselves are related. In such a situation, an option retrieval mechanism that retrieves only the relevant options drastically simplifies the learning task at hand, avoiding the need to learn a policy over the entire option library. However, if even one option that is necessary for the task is missed out, the learning collapses and fails. The work in this thesis extends a project that implements an option retrieval mechanism, by implementing an adaptive augmentation mechanism, that uses an augmented neural network to learn a policy over parts of the state space that the set of retrieved options does not cover, preventing full collapse of the option retrieval mechanism in situations where some options are missed out. Some positive results of this idea are demonstrated, first on a simple gridworld environment called CraftWorld, and then on a complex 3D environment called iTHOR. Finally, some ideas are proposed to extend the adaptive augmentation mechanism into a true continual learning framework for options, where new skills/options may be added to the option library as the continual learning agent progresses through tasks.

List of Tables

List of Figures

Abbreviations

Abbreviation	Full Form
RL	Reinforcement Learning
HRL	Hierarchical Reinforcement Learning
OIHRL	Option-Indexed Hierarchical Reinforcement Learning
MDP	Markov Decision Process
SMDP	Semi-Markov Decision Process
A2C	Advantage Actor-Critic
A2T	Attend, Adapt and Transfer

Contents

1 Introduction

1.1 Overview

Hierarchical approaches to artificial intelligence are inspired from significant evidence in behavioural science and neuroscience [1, 2] that animal activities can be described in terms of a small set of important behaviours. Even when humans think of going from, say, home to the office, we tend to break it down into the several steps involved, such as going downstairs, walking to the bus station, exiting the bus at the right stop, and climbing up to the office. In the actual execution of these steps, some actions such as climbing stairs are performed almost subconsciously because they are behaviours that we have internalised through repetition across a variety of other tasks, despite the recruitment of complex sensorimotor systems to move several parts at once. As such, breaking down tasks into subtasks within the realm of Hierarchical Reinforcement Learning has shown to benefit learning of complex tasks and learn skills that are transferable.

1.2 Motivation and Objective

However, skills acquired through such learning can only transfer to related tasks, where such behaviours are required again. A truly versatile learning agent of general intelligence must be able to continuously perform sequences of varied tasks in the real world, while not completely forgetting how to perform previously learnt tasks. Also, these new tasks may bring with them the potential to even learn more useful skills which can transfer to an even broader set of tasks.

As a human example, consider a non-athletic person who has had no exposure to any kind of sport. They possess useful life skills such as walking, conversation, and perhaps a musical talent as well. Let us imagine this person decides to learn tennis. At this juncture, none of the skills that they possess transfer well to the task at hand. The ability to adapt and learn outside of existing skills is important even for an agent that already possesses useful skills.

Furthermore, the sport of tennis has several transferable skills to other sports as well, such as powerful stroking with the arm, bracing the wrist tight, explosive jumps, and agility. Internalising these fundamental skills of athleticism could directly transfer to new physically demanding tasks in the future, such as arm wrestling for example. If somehow these fundamental skills were not memorised, arm wrestling would be an entirely new task, despite having gone through the process of learning tennis. It could be learnt a lot easier through direct application of previously developed skills, if they were retained instead. As such, it may help an agent to not only retain the performance on tennis, but also certain useful skills for better learning future tasks. This is known as *positive forward transfer.*

On the other hand, if there is a large library of skills that an agent can always try, learning a strategy to use them may be significantly slower. Skill indexing is an approach to retrieve the relevant skills by looking at the state and items in the environment available in the current task (more on this in Section 2.5). This allows efficient learning of a higher-level strategy over only the relevant skills. In such an approach, however, an indexing mechanism may sometimes miss out on one or more relevant options to the task (or the entire option library itself may not contain a relevant option in the first place). When this happens, it is quite useful to be able to adapt to the task at hand anyway.

The goal of this project is to empower hierarchical agents possessing a certain set (either a library or a retrieved index) of skills to adapt to new unseen tasks, where the existing/retrieved skills may not suffice. Additionally, we propose that useful behaviours encountered in these new tasks may be captured and encapsulated as new skills, progressively expanding the arsenal of skills that the agent holds.

2 Background

2.1 Reinforcement Learning

Reinforcement Learning is an artificial intelligence paradigm developed in the context of optimal control, where the agent learns an optimal policy π to take actions to move from one state to another in an environment. The learning happens through scalar feedback that the agent receives in the form of rewards. The environment is usually encoded as a Markov Decision Process (MDP) represented by the tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ), where $\mathcal{S}$ is the state space, $\mathcal{A}$ the action space, $\mathcal{P}$ the matrix of transition probabilities $p(s'|s,a)$ for each state-action pair, $\mathcal{R}$ the reward function, and γ the factor by which future rewards are discounted per time-step. This framework allows agents to learn complex behaviours even in stochastic environments without explicit supervision as to how to reach a goal - the agent learns through trial and error and the reward it receives from the environment.

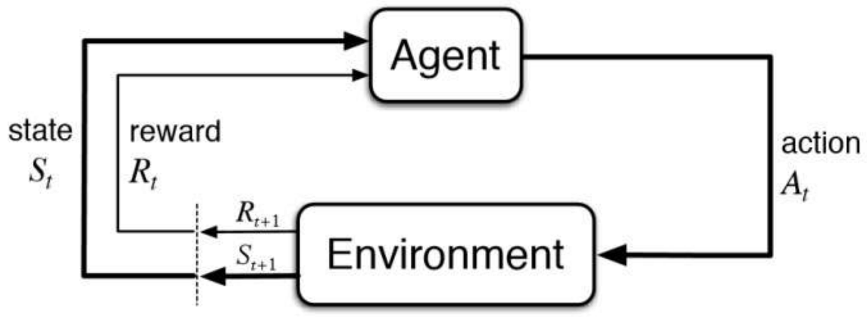


Figure 1: The Reinforcement Learning framework

2.2 Hierarchical Reinforcement Learning

However, this learning mechanism that relies solely on reward signals can sometimes hinder learning. In long-horizon sparse-reward tasks, the reward is received rarely, say, at the end of a successful episode. In early training, this may only happen through a series of lucky actions in the trial-and-error process. Therefore, learning can be quite slow. One of the solutions that has proven to effectively tackle this issue in several complex scenarios, such as robot continuous control [3], is Hierarchical Reinforcement Learning (HRL).

In HRL, the longer-horizon task is broken down into a hierarchy of smaller subtasks. A higher-level policy decides which subtask to use, and the lower-level policies in turn execute the subtasks. At the lowest level, the actual actions in the environment would be executed. This framework can drastically accelerate learning as, at the higher level, the shorter horizon leads to better reward feedback, and at the lower level, the subtasks may be easier to learn as well. There are several approaches today for HRL - some of the recent work is in multi-agent HRL [4] and graph-based subtask learning [5]. Some of the original classical approaches are MAXQ [6], Hierarchical Abstract Machines [7], and Options [8].

Options

The Options framework uses the theory of Semi-Markov Decision Processes (SMDPs), which are an extension of the classical MDPs to a continuous-time case, where the actions can take varying amounts of time (temporally extended). The time-varying subtasks in an HRL setting, which may be similarly temporally extended, therefore fit neatly into the SMDP framework in the form of temporally-extended actions.

Therefore, the SMDP representing the reinforcement learning problem in the hierarchical setting can be formulated as the MDP + a set of options $\mathcal{O}$ which correspond to the available set of temporally extended actions. An option is represented as $< I_o, \pi_o, \beta_o >$, which correspond to the initiation state set (states where an option can be taken), the option policy, and the option termination function which dictates where an option will end. The higher-level policy π can learn to choose an option from the set of options (which may also include the lower-level primitive actions) in any given MDP state. When an option is picked, its primitive actions are executed in the MDP (according to π_o) until it terminates, the higher-level policy selects another option, and so on.

In classical SMDP theory, the temporally extended actions are treated as atomic units which cannot be further divided. Here, however, an option is composed of primitive actions. Therefore, one can consider options to be a bridge between MDPs and SMDPs, which allows one to dig deeper into what the options are doing at the MDP level, and even change/learn the subtasks that they perform.

2.3 Literature Review

2.3.1 Continual Learning

A versatile, real-life, general-purpose learning agent should be able to incrementally learn new tasks using both current data and (potentially) behaviours from previous tasks that it has learned, while retaining previously learned abilities. This is the premise of continual learning [9], where several algorithms have emerged from various approaches towards the problem - some algorithms are episodic-memory-based [10, 11], where a buffer of experiences is maintained, but used in a specific manner amenable to continual learning, unlike the usual experience replay used in deep RL algorithms such as DQN [12]. Some approaches store latent parameters and tune them as tasks are observed, e.g., ELLA [13], which maintain a latent basis that represents tasks, uses it to solve a new task, and refines the basis using this task. The related PG-ELLA [14] extends this algorithm into a policy gradient method for continual RL.

We are interested in the problem of learning new options in a continual manner when new tasks arrive. In Bagaria et al. 2020 [15], skills are learnt through identification of salient regions of the state space and learning options between them. Brunskill and Li 2014 [16] present a mathematically justified algorithm for adding options with PAC-guarantees, but it assumes the knowledge of near-optimal actions for each state in the MDP. Aubret et al. 2020 [17] maintain a hierarchy of intrinsically-motivated skills which are progressively broken down and expanded, making them transferable. However, the transferability is not specific to a potential new task in a stream of sequential tasks, as the algorithm learns a general set of reusable skills for the currently available tasks.

Perhaps the closest work to this project is by Tessler et al. 2016 [18], which contains several lower-level Deep Skill Networks (DSNs) which function as options, and a higher-level Hierarchical Deep RL Network (H-DRLN) that uses a variation of DQN to learn to choose between these DSNs and the primitive actions. While this method can adapt the existing set of options to new tasks, it would find large or continuous action spaces harder to handle, as a continuous action cannot be selected categorically, alongside a set of options.

2.3.2 Transfer Learning

A different, yet related, paradigm to continual learning is the transfer learning setting. Here, the information from several source tasks is used for effectively learning a target task. However, the agent and policy for the target task is separate, and it is not expected to retain the performance of the source tasks. The ability to learn a new policy in this framework, when combined with a hierarchical learning agent, can be utilised to learn a new option - to adapt an available option index to a new task. This idea will be the foundation of the Adaptive Augmentation method described below.

In particular, the method is inspired by Attend, Adapt and Transfer (A2T) by Rajendran et al. [19], a transfer learning method where several source tasks $K_1, K_2, ..., K_N$ are available, and a base policy network K_B is available in order to learn the target task in the parts of the state space where the source tasks transfer negatively. A deep attention network learns state-dependent attentions for each network, including the base network, and the final target policy K_T in any state is the linear combination of the output of each network weighted by its respective attention in that state. In the problem of interest, where the option library available

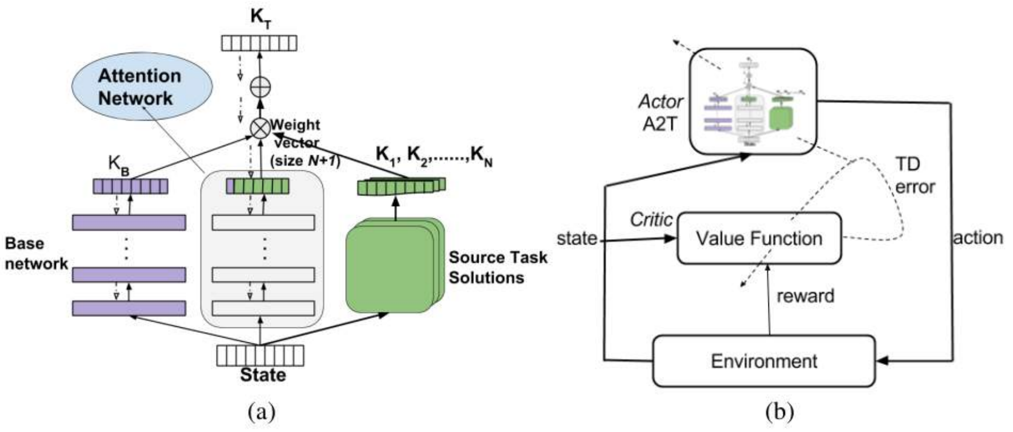


Figure 2: **From the A2T paper:** (a) The A2T architecture (b) Actor-Critic using A2T as the Actor

(or index retrieved) for a task may be insufficient, this idea can be used to adapt a base network to learn an augmented policy for states where the options are not useful. Furthermore, it is interesting to look into expanding this idea to allow continual learning of options, through the annexation of the learnt base network as a new option into the option index, to be used for future tasks.

2.4 Actor-Critic Methods

In policy gradient methods, gradient steps are taken to improve a parametrised policy according to a performance metric $J(\theta)$. If we wish to maximise the cumulative expected future reward (i.e., value function) as a metric when we follow a policy, the policy gradient theorem [20] allows the gradient step to take the following basic form in the REINFORCE algorithm:

$$\begin{aligned}\theta_{t+1} &\longleftarrow \theta_t + \alpha\overline{\nabla J(\theta)} \\ &\longleftarrow \theta_t + \alpha(G_t - b(S_t))\nabla \log \pi(A_t|S_t, \theta_t)\end{aligned}$$

where G_t is the cumulative discounted reward obtained in an episode, and $b(S_t)$ is a state-dependent, action-independent baseline that acts as a reference point with which to compare all actions in the current state S_t, and can reduce variance in training. A natural choice for the baseline is the value-function estimate for the state.

The issue with G_t is that one must wait until an episode ends to obtain it. This can become a problem especially in long-horizon settings. The idea of temporal-difference learning [21] can be used to bootstrap the value of G_t using the current reward and the value-function estimate of the next state, to learn in an online fashion. This bootstrapping can also reduce variance in learning, and leads us into the idea of actor-critic methods [22]:

$$\begin{aligned}\theta_{t+1} &\longleftarrow \theta_t + \alpha(R_t + \gamma\hat{v}(S_{t+1}) - \hat{v}(S_t))\nabla \log \pi(A_t|S_t, \theta_t) \\ &\longleftarrow \theta_t + \alpha\delta_t\nabla \log \pi(A_t|S_t, \theta_t)\end{aligned}$$

Here, the policy $\pi(a|s, \theta)$ is referred to as the actor, and the value-function estimate $\hat{v}(s)$ is called the critic, as it provides a baseline and a bootstrapping estimate of the return that critiques the actor and pushes it in the right direction.

2.5 Option Indexed Hierarchical Reinforcement Learning (OI-HRL)

The work described in this thesis extends a project by Google Research India on the task of retrieving the options relevant to the task at hand from a larger library of pre-trained or pre-encoded options [23]. In a regular end-to-end HRL setting, the agent attempts to partition a long-horizon task into subtasks, learn how to perform these subtasks, and also sequence these subtasks in the right order. This is a difficult task to successfully learn, and requires high amounts of training.

Provided that the task is one among many related tasks, if one allows the agent to learn useful behaviours from previous related tasks, then this can be leveraged to improve performance on a new unseen task. The knowledge from previous related tasks can be encoded as a large library of options, over which direct hierarchical training would prove to be extremely slow. It can be assumed that this large library of options has been acquired across the lifetime of a continual learning agent. This is the situation in which the OI-HRL approach proves beneficial, as it allows hierarchical training to happen over a much smaller set of relevant options to the task.

OI-HRL learns an index over the option library that generates a vector key for each option, and a Query Generation Network (QGN) that generates a vector query based on the initial state of the environment. Using a dot product between the key for each option and the vector query of the current task, from the abstract of the project: "...we learn an affinity function between options and the items present in the environment." This affinity function is used to generate a probability vector consisting of the probabilities that each option is present in successful trajectories of the current task. These probabilities are then used to rank the options and the top-p [24] set of options is retrieved as the relevant set of options for the current task. If the retrieved set contains all the required options, learning a higher-level policy over options is made much more efficient.

In the case where the retrieved set is insufficient (or the option library itself does not possess the right options for the current task in the first place), however, the higher-level policy completely fails to complete the task. This is the context in which the ability to adapt to such situations is highly valuable. This is done using the presence of a base network that can learn the policy in parts of the state space that are not covered by existing options, in a very similar way to A2T. The methodology is described in the next section.

3 Adaptive Augmentation: Methodology

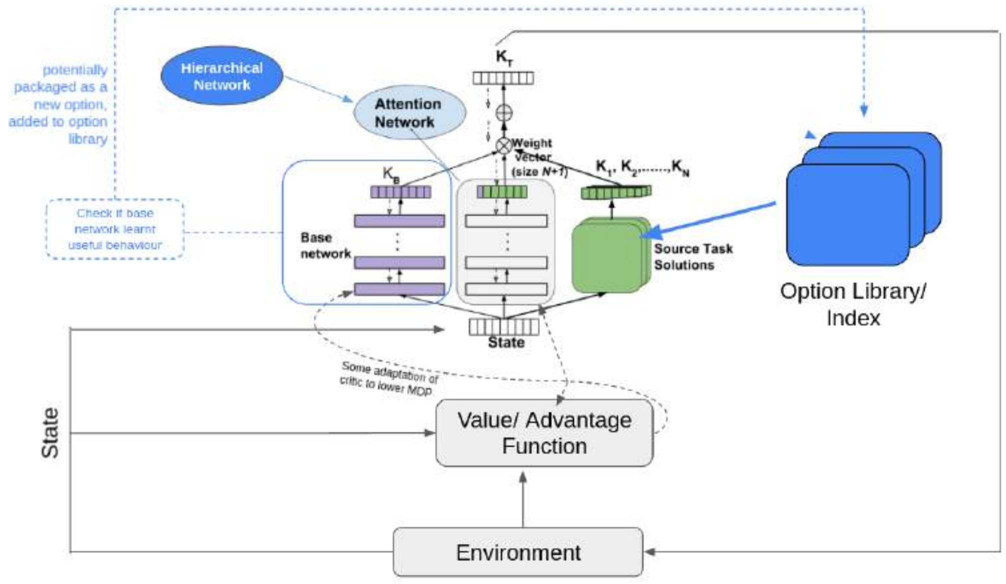

potentially packaged as a new option, added to option library
Hierarchical Network
Attention Network
K_T
Weight Vector (size N+1)
K_B
$K_1, K_2, \ldots, K_N$
Check if base network learnt useful behaviour
Base network
Source Task Solutions
Option Library/ Index
State
Some adaptation of critic to lower MDP
State
Value/ Advantage Function
Environment

Figure 3: Visual Flowchart for the Adaptive Augmentation Strategy inspired by the A2T architecture. The dotted blue lines represent parts of future work.

The adaptation of an option index to new tasks is done through augmentation, using an architecture inspired by A2T. The attention network would play the role of a hierarchical higher-level network. The source tasks that receive the attentions would be replaced by the options from the option index, and the base network would function as an augmented "extra" option, which can learn a policy for the parts of the state space where the existing options do not suffice.

3.1 Modifications and Complications Relative to A2T

- However, the SMDP setting of the Options framework is a little different. For one, the attention network in A2T outputs attentions at the higher level, and uses the resultant lower-level policy for training, at the timescale of primitive actions. On the other hand, a hierarchical network chooses an option at the higher level, and the option plays actions until its termination. Control is not returned to the hierarchical network at every time step, but after the option terminates. Therefore, there is a need to train the hierarchical network in an intra-option manner, perhaps using lower-level policy gradients. It can also be assumed that control is, in fact, returned to the hierarchical network after every primitive action. In such a framework for Options, there is no termination function. However, the inability to play options to directly complete subtasks without the hierarchical policy diving into primitive actions would mean that the hierarchical training would be considerably slower.

- Secondly, if we consider the setting where options are played until termination, it is not straightforward to combine the ”source” and ”base” options using a soft-attention mechanism as in A2T, where it allows flexibility to combine multiple lower-level solutions. With options, one does obtain several lower-level policies which could be combined in a soft manner, but the varying timescales and terminations present a roadblock. What does it mean to combine several options? Does the resultant policy run for as long as option 1, or option 2, or the base network (which just plays lower level actions, leading to control returning to the hierarchical network after just one time step)? As a result, hard attentions have been used, which simply select one of the lower-level choices with no combination.

- Next, it is worthwhile to note that A2T prescribes a learning of the base network which is off-policy and intra-option. That is, even if the effective policy hardly uses the base policy, the actual actions played in the environment can be used to train the base network as if the action was played from the base network itself:

$$\theta_a \longleftarrow \theta_a + \alpha_{\theta_a}\delta_t \nabla \log \pi_T(s_t, a_t)$$
$$\theta_b \longleftarrow \theta_b + \alpha_{\theta_b}\delta_t \nabla \log \pi_B(s_t, a_t)$$

  for the case of actor-critic learning. (Gradient normalisation has not been displayed here for the sake of brevity, but it prevents the problem of slow crawling when the gradients are too small.)

  This method of training the base network can result in significantly faster learning than simply training the base network only when it is picked. This is a benefit that could be leveraged in our application, however, it is important to note that this is not possible if the options are rule-based, indivisible units. In the experiments below, the options used are indivisible units, and therefore this kind of training was not possible. The base network is trained only on those time steps when it is picked.

- Finally, if the base network learns useful behaviour over a significant part of the state space in any particular task (or series of tasks) which is not captured by the option library, the base network can potentially be packaged as another option and annexed to the option library, thereby learning and adding options in a truly continual manner. This might require generalisation of the learnt base network policy over a larger portion of the state space than initially learnt upon. This idea is discussed in Section 6.

4 Experimental Setup and Results

4.1 CraftWorld

The algorithm was first implemented on a 12×12 gridworld-like environment called CraftWorld [24]. It is similar to the video game Minecraft in that objects can be collected, and combined in workshops to produce complex objects, which in turn can be used in recipes for even more complex objects (built in possibly different workshops). Some of the recipes from the Cookbook of the CraftWorld are displayed below in Table 1.

`plank`	use: `wood`
	at: `workshop0`
`paper`	use: `wood`
	at: `workshop1`
	use: `grass`
	at: `workshop2`
`bridge`	use: `wood, iron`
	at: `workshop2`
`axe`	use: `rod, iron`
	at: `workshop1`
	use: `stick, iron`
	at: `workshop4`

Table 1: Some recipes from CraftWorld

An SMDP was built on top of this CraftWorld environment, with a set of options. This set comprises of the base network, and all other options were picked from a set of pre-encoded, rule-based options which perform subtasks such as picking up basic objects like wood and grass. These rule-based options had been constructed beforehand, and were indivisible units - for simplicity, they were constructed as ”teleporting” options that let the agent directly jump to the required objects and pick them up, and calculate and return the corresponding discounted reward (referring to the true reward which would be returned if the option played actual primitive actions). This is calculable due to the gridworld-nature of the environment: the optimal option would simply follow the shortest path (Manhattan distance path) to the object and pick it up, which means the length of the option (and therefore discount applied) is known.

Such an environment allows the construction tasks that are of interest to the adaptive augmentation problem, where the set of available/retrieved options is involved in, but not sufficient for, solving a task. Let us represent any task with the notation `"[options available] goal object"`. The experiments were conducted on these tasks: `"[wood, workshop0] plank"`, `"[wood] plank"`, `"[wood, iron, rock, grass] plank"`, and `"[wood, iron, workshop2] bridge"`, `"[wood, workshop2] bridge"`, with and without the base network. The last of these tasks was also included to investigate the effect of the presence of irrelevant options on the sample complexity and convergence rate of a task.

The details of the CraftWorld environment are listed below:

- **Gridworld size:** 12×12
- **Number of kinds of items:** 41

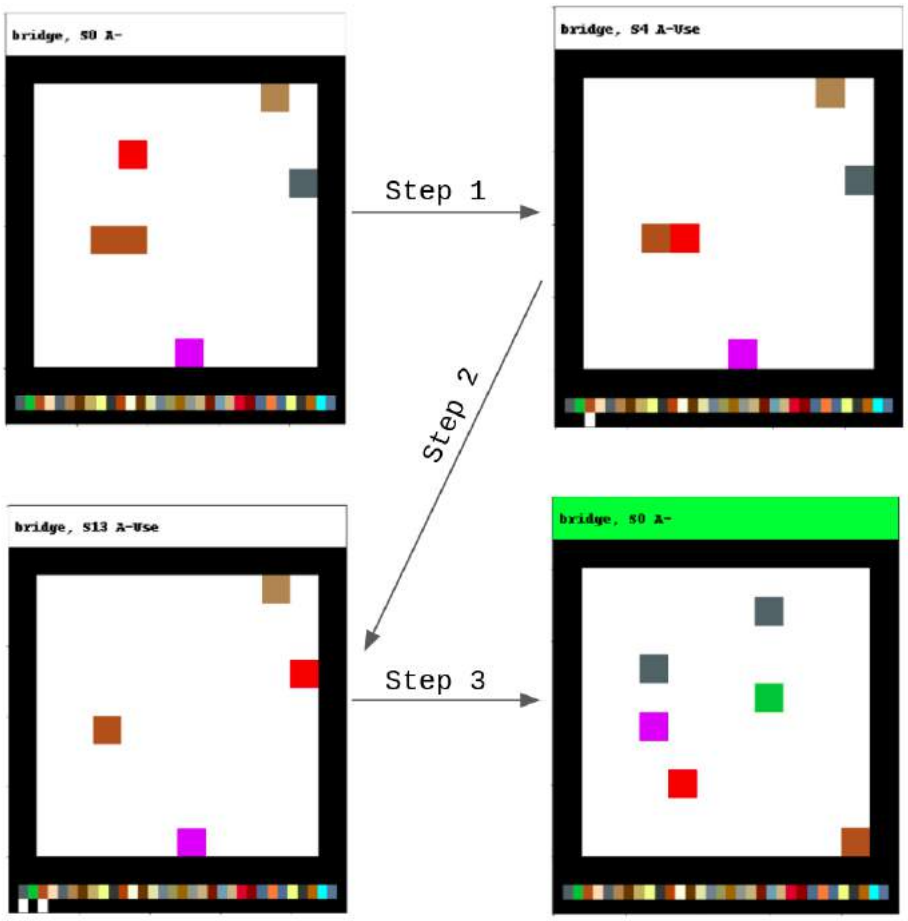


Figure 4: The CraftWorld environment, and a representation of the `bridge` task step-by-step, using the teleporting, pre-encoded options. Step/Option 1: pick up `wood`, Step/Option 2: pick up `iron`, Step/Option 3: use them at `workshop2`

- **Agent state representation in underlying MDP:** For the purposes of Option Indexing in OI-HRL, it was sufficient to consider a state representation that was global, as the teleporting options did not need positional information of the agent relative to the objects of interest. For adaptation, however, the base network uses primitive actions to actually move in the environment, and therefore it is important for the agent to have an ego-centric view so that it may understand when an object is nearby and be encouraged to move towards it. The final state is a $(2091,)$ vector which contains the concatenation of:
  - one hot vector representations of presence of objects in a local 5×5 grid around the agent (flat vector of size 5*5*41)
  - one hot vector representations of presence of objects in a larger local $5^2 \times 5^2$ grid (in this case, this covers the entire gridworld) around the agent, where each 5×5 subgrid in the grid is maxpooled into one entry in the one-hot vector (if an object is present in the subgrid, the corresponding cell in the final vector takes the value 1) (flat vector of size 5*5*41)
  - one hot vector representation of the inventory of the objects that the agent has currently (flat vector of size 41)
- **Agent action space:** In the CraftWorld MDP, the 5 actions are

  - 0: `DOWN`
  - 1: `UP`
  - 2: `LEFT`
  - 3: `RIGHT`
  - 4: `USE`: use an item that is present in the current cell. If an object, pick it up. If a workshop, use the workshop to build a complex object that the objects currently in the inventory allow.

  In the SMDP, the actions are the options, $K+1$ in number. K is the number of rule-based options available, and the extra option is the base network.

- **Reward:** The agent is rewarded +1 when it reaches the state of holding the right items in the inventory, and is penalised (-0.3) in case it picks up extra items. Also, when an option is executed, any reward received at the end of the option is discounted appropriately.

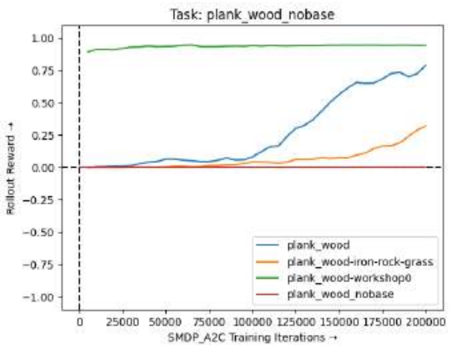
Algorithmic Details in CraftWorld Experiments

The SMDP was trained using Advantage Actor-Critic (A2C) [], a form of Actor-Critic that uses the advantage function as the critic instead of simply a value function, reducing variance.

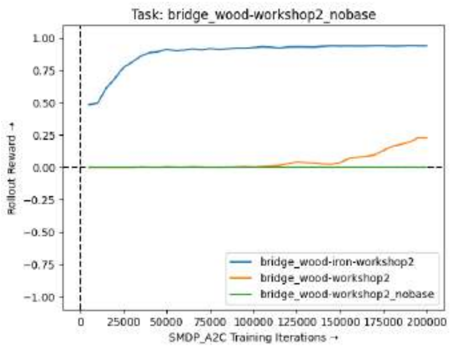
- Number of parallel subprocess-environments: 10, each with a different training seed
- Higher-level/attention policy network architecture: 2091 × 64 × 64 × K+1
- Value network architecture: 2091 × 64 × 64 × 1
- Activation function: tanh()
- Learning rate: 3×10^{-4}
- Optimizer: RMSProp ($\epsilon = 10^{-5}$)
- Gradient clipping (max = 0.5) and normalisation used
- Discount factor γ: 0.99
- Value function loss coefficient (critic loss): 0.5
- Generalized Advantage Estimate (GAE) used to estimate advantages, $\lambda = 0.99$
- Rollout buffer used for mini-batch training, size = 5

The base network was trained using A2C in a similar fashion, however, the same state-value critic as the higher-level network was used for the base network too.

- Base network policy architecture: 2091 × 64 × 64 × 5
- Activation function: tanh()
- Learning rate: 3×10^{-4}
- No rollout buffer used (size = 1), therefore training is online and stochastic
- All other hyperparameters same as the higher-level network. A clear hyperparameter search has not been performed yet as the implementation of the algorithm is yet to complete.



(a) various `plank` tasks



(b) various `bridge` tasks

Figure 5: Mean episode reward curves during rollouts

Results

In both `"[wood, workshop0] plank"` and `"[wood, iron, workshop2] bridge"`, the agent had all the required pre-trained options for the task and did very well with a reward close to 1 (the maximum). Here, the hierarchical policy had a very straightforward task of putting together the options in the right order.

In `"[wood] plank"` and `"[wood, iron, rock, grass] plank"` without base network, the options were not sufficient to solve the task. Therefore, even after 200,000 training steps, there was no learning and the average reward was 0.

Once the base network was added, the agent was able to leverage the idea of adaptation. The base network was capable of learning a policy in the parts of the state space where the available options were insufficient. However, in the case of `"[wood, iron, rock, grass] plank"`, learning was significantly slower than in `"[wood] plank"`, because the presence of excess options slowed down the higher-level hierarchical learning, and the base network learning as a result of the base network being picked less frequently as the agent explored. Perhaps if the base network is allowed to learn off-policy and intra-option, after the pre-trained teleporting options are replaced with options that play primitive options, it can learn faster even through actions that other options execute, and the training speed becomes more comparable with the `"[wood] plank"` case.

Similarly, `"[wood, workshop2] bridge"` without a base network learned nothing as one ingredient option was missing. When a base network was added, learning happened, but it was significantly slower than in `"[wood] plank"`, which is a 1-ingredient-missing analogue for the `plank` object. Once again, this was probably happening due to less frequent selection and training of the base network as the agent explored (due to there being more options to choose from).

In conclusion, encouraging results in favour of the idea of adaptation to improve the performance of option indexing/retrieval were obtained. Table 2 summarises the values of mean episodic rewards at the end of 200,000 steps for each of these tasks.

4.2 AI2THOR

Upon obtaining supportive results, the next step was to attempt adaptive augmentation in a more complex 3D environment. AI2THOR [25], developed by the Allen Institute for Artificial

Final Target Task	**Options Available**	**R$_{\mathbf{av}}$**
`plank`	[wood, workshop0]	**0.947**
	[wood] with base network	0.890
	[wood, iron, rock, grass] with base network	0.371
	[wood] without base network	0.0
	[wood, iron, rock, grass] without base network	0.0
`bridge`	[wood, iron, workshop2]	**0.941**
	[wood, workshop2] with base network	0.223
	[wood, workshop2] without base network	0

Table 2: Mean episodic rollout reward (R_{av}) for the various tasks in CraftWorld

Intelligence, is a framework comprised of 3D simulation environments that facilitate interaction with several objects. It comprises of iTHOR (an environment with a set of interactive objects and scenes, with accurate physics modelling), ManipulaTHOR (an environment for visual object manipulation using a robotic arm), and RoboTHOR (a navigation environment with simulated scenes which have counterparts in the real world). The iTHOR scenes have been used for the experiments in this project.

iTHOR consists of 120 scenes, 30 each of kitchens, living rooms, bedrooms, and bathrooms in simulation. Since the task setting for the problem requires the usage of recipes (sequences of subtasks/steps taken to achieve the goal), we selected one of the kitchen scenes as our final environment. The possible recipes from the kitchen scenes are quite intuitive to construct and evaluate on, as there are food items, such as apple, bread, and egg, and they can be changed into a Cooked state from Uncooked using, say, a stove-burner. Additionally, bottles and mugs can be filled with coffee or water, for example.

Figure 6: Some scenes from iTHOR. The full set of objects available in iTHOR is: *AlarmClock, AluminumFoil, AppleSliced, ArmChair, BaseballBat, BasketBall, Bathtub, BathtubBasin, Blinds, Book, Boots, Bottle, Bowl, Box, Bread, BreadSliced, ButterKnife, Cabinet, Candle, CD, CellPhone, Chair, Cloth, CoffeeMachine, CoffeeTable, CounterTop, CreditCard, Cup, Curtains, Desk, DeskLamp, Desktop, DiningTable, DishSponge, DogBed, Drawer, Dresser, Dumbbell, Egg, EggCracked, Faucet, Floor, FloorLamp, Footstool, Fork, Fridge, GarbageBag, GarbageCan, HandTowel, HandTowelHolder, HousePlant, Kettle, KeyChain, Knife, Ladle, Laptop, LaundryHamper, Lettuce, LettuceSliced, LightSwitch, Microwave, Mirror, Mug, Newspaper, Ottoman, Painting, Pan, PaperTowelRoll, Pen, Pencil, PepperShaker, Pillow, Plate, Plunger, Poster, Pot, Potato, PotatoSliced, RemoteControl, RoomDecor, Safe, SaltShaker, ScrubBrush, Shelf, ShelvingUnit, ShowerCurtain, ShowerDoor, ShowerGlass, ShowerHead, SideTable, Sink, SinkBasin, SoapBar, Sofa, Spoon, SprayBottle, Statue, Stool, StoveBurner, StoveKnob, TableTopDecor, TargetCircle, TeddyBear, Television, TennisRacket, TissueBox, Toaster, Toilet, ToiletPaper, ToiletPaperHanger, Tomato, TomatoSliced*, Towel, TowelHolder, TVStand, VacuumCleaner, Vase, Watch, WateringCan, Window, WineBottle.*

A robotic agent is allowed to navigate in the iTHOR environment and manipulate objects by moving them and modifying their state, through the usage of actions provided by the iTHOR API. iTHOR provides access to the metadata of the ”controller” (agent + environment), which includes information such as `objectType`, `objectId` (a string of the format ’`Egg_0|+00.12|-00.45|+00.97|`’ - a combination of the `objectType` and initial spawn location), position, rotation, visibility, temperature, the objects they contain (called ”receptacle objects”), and the objects they are contained in (called ”parent receptacles”) of each object. Further, the metadata contains information about the agent: its own position and rotation, as well as camera rotation angle. The metadata also contains a Boolean variable indicating whether the last action was successfully executed or not.

It is important to note that an action played by the iTHOR agent on an object only affects the object if it is in visible range of the agent, which is specified with a default value of 1.5 metres. Objects farther than this from the agent cannot be manipulated unless the agent moves closer to them.

Constructing the SMDP over iTHOR

The recipes used for all the tasks in our case do not require that all objects in iTHOR be considered. For example, Vase is not an object used in any recipe. The list of the 20 considered objects is:

Object No.	Object Name
0	Apple
1	Bottle
2	Bowl
3	Bread
4	BreadSliced
5	CounterTop
6	Cup
7	DiningTable
8	Egg
9	EggCracked
10	Microwave
11	Mug
12	Pan
13	PepperShaker
14	Plate
15	Pot
16	SaltShaker
17	StoveBurner
18	Toaster
19	WineBottle

We ignore all the other objects in the iTHOR environments. The objectIds of duplicate objects (more than one instance of the same object type) are also included in a list and ignored during the execution of the program. This is for simplicity in the state representation - so that there is no non-toasted slice of bread after a bread slice has been put in a toaster and toasted, for example.

4.2.1 Action Space

Part of the infrastructure needed to construct an MDP over the iTHOR environment is provided within the AI2THOR framework - the agent actions and the transition probabilities (which are mostly deterministic, except for certain actions that fail with a miniscule probability due to bugs) are packaged with the iTHOR agent and the environment. The actions that the agent is capable of using, which perform navigation, and object movement and manipulation are described in Table 3.

Many actions that the agent is able to play on the iTHOR simulator, allow continuous

Action Type	Action Name and Important Arguments

Navigation	`ctr.step(action="MoveAhead", moveMagnitude=float)`
	`ctr.step(action="RotateLeft", degrees=float)`
	`ctr.step(action="RotateRight", degrees=float)`
	`ctr.step(action="LookUp"/"LookDown", degrees=float)`
	`ctr.step(action="Crouch"/"Stand")`
	`ctr.step(action="Teleport", position=dict(x, y, z), rotation=dict(x, y, z), horizon=float)`

Object Movement	`ctr.step(action="PickupObject", objectId=str)`
	`ctr.step(action="PutObject", objectId=str(targetobject))`
	`ctr.step(action="DropHandObject")`
	`ctr.step(action="ThrowObject", moveMagnitude=float)`
	`ctr.step(action="MoveHeldObjectAhead", moveMagnitude=float)`
	`ctr.step(action="RotateHeldObject", pitch=float, yaw=float, roll=float)`
	`ctr.step(action="DirectionalPush", objectId=str, moveMagnitude=float, pushAngle=float)`

Object State Changes	`ctr.step(action="OpenObject", objectId=str)`
	`ctr.step(action="CloseObject", objectId=str)`
	`ctr.step(action="BreakObject", objectId=str)`
	`ctr.step(action="CookObject", objectId=str)`
	`ctr.step(action="SliceObject", objectId=str)`
	`ctr.step(action="ToggleObjectOn", objectId=str)`
	`ctr.step(action="ToggleObjectOff", objectId=str)`
	`ctr.step(action="OpenObject", objectId=str)`
	`ctr.step(action="DirtyObject", objectId=str)`
	`ctr.step(action="FillObjectWithLiquid", objectId=str, fillLiquid=str)`
	`ctr.step(action="EmptyLiquidFromObject", objectId=str)`
	`ctr.step(action="UseUpObject", objectId=str)`

Table 3: The main original actions that the iTHOR API provides. Most of these actions also have an Boolean argument called `forceAction` which, if set to `true`, executes an action ignoring visibility and interactability constraints. Some auxiliary actions have not been displayed, such as `GetReachablePositions` which returns a list of positions that the agent can reach (useful in tandem with the `Teleport` action). Each object-manipulating action requires the specification of the `objectID` of the object which is to be manipulated, which is a string that is part of the object's metadata.

values for movement magnitude, rotation angle, etc. Since extremely precise control is not required to perform tasks in iTHOR, it is possible to reduce this continuous action space into a discrete one for simplicity. The navigation actions have therefore been specified with certain fixed values of movement magnitude and rotation angle, and made available to the agent as discrete actions. Additionally, the PickUpObject, PutObject, SliceObject, OpenObject, and CloseObject actions for each of the 20 considered objects (only if any given object is pickupable, receptacle, sliceable, etc.) is included as a separate action. The full list of available primitive actions in the MDP's action space is described below in Table 4.

Primitive Action	**Corresponding Real Action Used in iTHOR**
MoveAhead	`ctr.step(action="MoveAhead", moveMagnitude=0.5)`
RotateLeft/ RotateRight	`ctr.step(action="RotateLeft"/"RotateRight", degrees=30)`
RotateLeftHard/ RotateRightHard	`ctr.step(action="RotateLeft"/"RotateRight", degrees=60)`
Rotate180	`ctr.step(action="RotateRight", degrees=180)`
LookUp/LookDown	`ctr.step(action="LookUp"/"LookDown", degrees=30)`

PickupBottle	
PickupEgg	
·	`ctr.step(action="PickupObject", objectId =`
·	`corresponding pickupable object)`
PickupBread	

PutPan	
PutPot	
·	`ctr.step(action="PutObject", objectId =`
·	`corresponding receptacle object)`
PutCountertop	

SliceApple	`ctr.step(action="SliceObject", objectId =`
SliceBread	`corresponding sliceable object)`

BreakEgg	`ctr.step(action="BreakObject", objectId = egg['objectId']`

ToggleOnMicrowave	`ctr.step(action="ToggleObjectOn", objectId =`
ToggleOnToaster	`corresponding toggleable object)`

OpenMicrowave	`ctr.step(action="OpenObject", objectId =`
	`microwave['objectId']`
CloseMicrowave	`ctr.step(action="CloseObject", objectId =`
	`microwave['objectId']`

FillMugWithCoffee	`ctr.step(action="FillObjectWithLiquid", objectId =`
FillBottleWithCoffee	`corresponding object with canFillWithLiquid==True,`
	`fillLiquid = "coffee")`

Table 4: The index of primitive (or base) actions that form the action space in the MDP, derived from the API actions available in iTHOR by providing fixed, discrete values for the parameters

4.2.2 Pre-Encoded Options

Similar teleportation options to the CraftWorld setting were created for iTHOR and used as the pre-encoded options available in the option library. The teleportation was implemented using the inbuilt `Teleport` action in iTHOR. The arguments to the `Teleport` action, i.e., the `position`, `rotation`, and `horizon` (camera pitch angle) that the agent needs to teleport to were calculated using iTHOR metadata of the agent and the objects, and using geometry and trigonometry of triangles formed by the involved vectors, in a manner very similar to the calculation of the relative pitch and relative angle of objects (described in Section 4.2.3).

Figure 7 displays the execution of four options: `pickup_egg, break_egg, pickup_eggcracked, puton_countertop`.

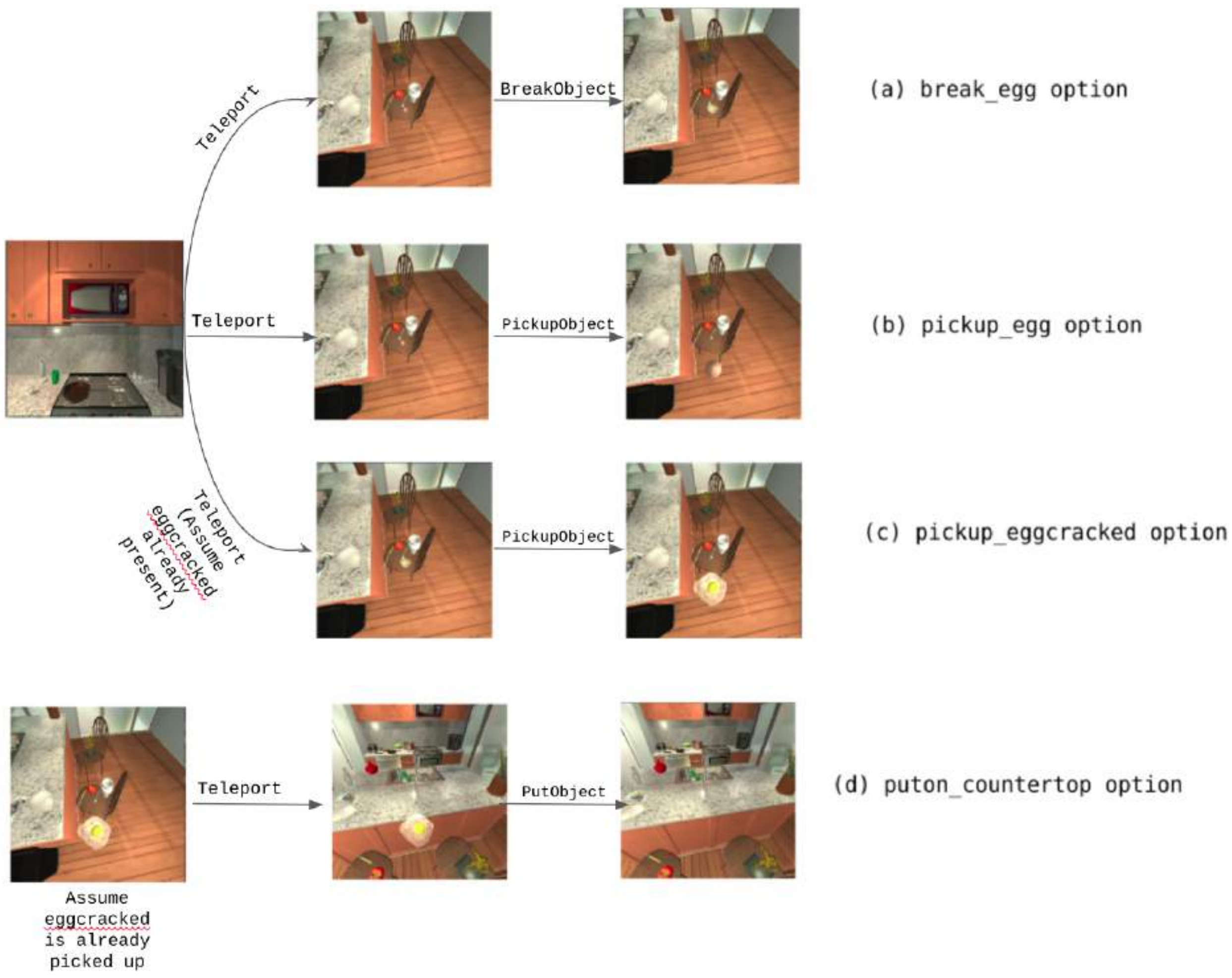

Teleport
BreakObject
(a) break_egg option
Teleport
PickupObject
(b) pickup_egg option
Teleport (Assume eggcracked already present)
PickupObject
(c) pickup_eggcracked option
Teleport
PutObject
(d) puton_countertop option
Assume eggcracked is already picked up

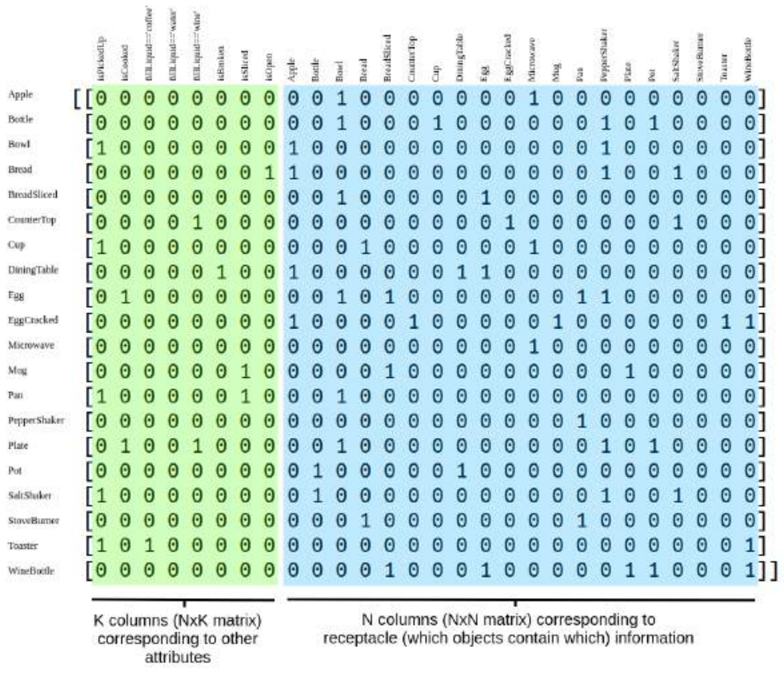
Figure 7: The visualisation of 4 pre-encoded options. Each step visualised happens directly after the previous one, without intermediate steps of navigation or manipulation. The name of each iTHOR action used for each step is written above the arrow leading from step to step.

4.2.3 State Space and Reward Functions: Some Challenges Faced and their Solutions

The metadata of the controller provided by iTHOR can be used to construct the state of the MDP. The state is encoded first as an $N \times (N+K)$ multi-binary matrix containing information about all considered objects. N corresponds to the number of considered objects, and K corresponds to the number of "extra attributes", described in the following paragraphs. The latter $N \times N$ square section of the matrix contains information about which objects are contained in which other objects - if the cell (x, y) is set to 1, it means that object x is contained in object y. This is meaningful information to include in the state as many of the tasks involve placing some object on another receptacle object, say, on a DiningTable or a CounterTop while serving a dish.

The remaining first K columns in the state matrix correspond to K other attributes obtained from the metadata, for each object corresponding to each row of the matrix. For the option indexing task, an abstracted environment capturing essential features of the actual iTHOR environments was used, where the available options could all "teleport" and perform their corresponding sub-task in a single step. Therefore, the current visibility of an object to an agent did not matter. With the usage of this abstracted environment, the actual graphical iTHOR environment with an agent capable of executing primitive actions could be omitted. Therefore, the abstract "objects" in this abstraction only possessed the following hand-picked attributes: `breakable, canFillWithLiquid, compatibleRecepticles, cookable, pickupable, sliceable, isPickedUp, temperature, isCooked, fillLiquid, receptacleObjects, parentRecepticles, isPresent`. They did not possess attributes of real objects in the real iTHOR environment, such as `visible, position`, and `interactable`.

For the recipe tasks in the option indexing context, it was therefore sufficient to consider only the following object attributes in the abstracted environment: `isPickedUp, isCooked, isBroken, isSliced, isOpen, fillLiquid=='coffee', fillLiquid=='water', fillLiquid=='wine'` as the K columns in the state matrix.



	isPickedUp	isCooked	fillLiquid=='coffee'	fillLiquid=='water'	fillLiquid=='wine'	isBroken	isSliced	isOpen	Apple	Bottle	Bowl	Bread	BreadSliced	CounterTop	Cup	DiningTable	Egg	EggCracked	Microwave	Mug	Pan	PepperShaker	Plate	Pot	SaltShaker	StoveBurner	Toaster	WineBottle
Apple	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Bottle	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0
Bowl	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Bread	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
BreadSliced	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
CounterTop	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
Cup	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
DiningTable	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
Egg	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
EggCracked	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
Microwave	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Mug	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Pan	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PepperShaker	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Plate	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
Pot	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
SaltShaker	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
StoveBurner	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Toaster	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
WineBottle	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	1

K columns (NxK matrix) corresponding to other attributes

N columns (NxN matrix) corresponding to receptacle (which objects contain which) information

Figure 8: The Older State Representation in the Previous Option Indexing Context

However, in the adaptation setting, in the presence of a base network which needs to learn a policy over primitive actions, it became necessary to work in the actual iTHOR environment. Here, the visibility and position of objects were important to consider - since we expected the agent to know when to walk forward, when to pick up an object, and when to execute other actions. In other words, the local context of the agent became important, and it was now necessary to shift to an egocentric view of the environment. As a result, the state representation required certain modifications.

The first step was to indicate the attributes of an object only when that object is visible. For a currently invisible object, the current state would have all zeroes in the corresponding row. This changes the state to an egocentric view, where only the attributes of the visible objects are known to the state. This was thought to encourage the agent to first seek out visibility of the relevant objects which, in turn, leads to successful trajectories - therefore, first the agent would align and/or navigate itself towards the relevant object, and then manipulate it as required.

However, this state representation failed despite several improvements to the learning algorithm over several weeks. It was, eventually, improvements to the MDP itself, discussed below, that yielded positive results.

Reward Structure: The tasks are all sparse-reward, therefore a reward of 1.0 is obtained only when the final goal is reached. This reward was encoded by checking whether only the relevant aspects of the current state match with the goal state. The goal state was generated by running the known recipe of the task on an instance of the environment in advance. If the relevant aspects matched, a reward of 1.0 was considered, else 0.0. In addition, a length penalty of -0.002 was added to the reward for every step, to discourage long policies and prevent the agent from getting stuck. A few examples of tasks and their reward conditions using the state information are tabulated in Table 5.

One of the important improvements had to do with the reward structure for the base network, and is described below:

1. In the option indexing and adaptation setting, it is a sequence of options (including the base network) that leads to a successful episode. This means there are two possible cases for the base network:

    - **Case 1:** The base network has to compensate for the <u>first</u> or an <u>intermediate</u> option in the recipe: In this case, the base network does not receive any reward directly. The reward for completing the task is only given to the hierarchical network. Therefore, the base network needs some kind of pseudo-reward to learn that the state it ends on is, in fact, valuable. One candidate for this pseudo-reward is the value function maintained by the hierarchical policy, which possesses the higher-level awareness that the end state of the base network can lead to reward upon playing other options. The reward, then, at each time step for the base network can be considered to be equal to `true reward+higher-level value`, effectively combining the true reward and the pseudo-reward.
    - **Case 2:** The base network has to compensate for the <u>last</u> option in the recipe: In this case, the base network receives the actual reward from the environment as it is the one that completes the task. Considering this, `true reward+higher-level value` is no longer a good candidate for the base reward, as adding the higher-level value as a pseudo-reward leads to a moving target for the A2C algorithm used by

the base network for learning. The higher-level value of the last state does not make sense in this context, as it is the true last state of the episode. Moreover, the value function of both the higher-level network and the base network indicate a similar value in this setting, as there is no future trajectory after the base network reaches success. Loosely assuming that both the value functions are the same, we have the following problem with the TD-error used in updates:

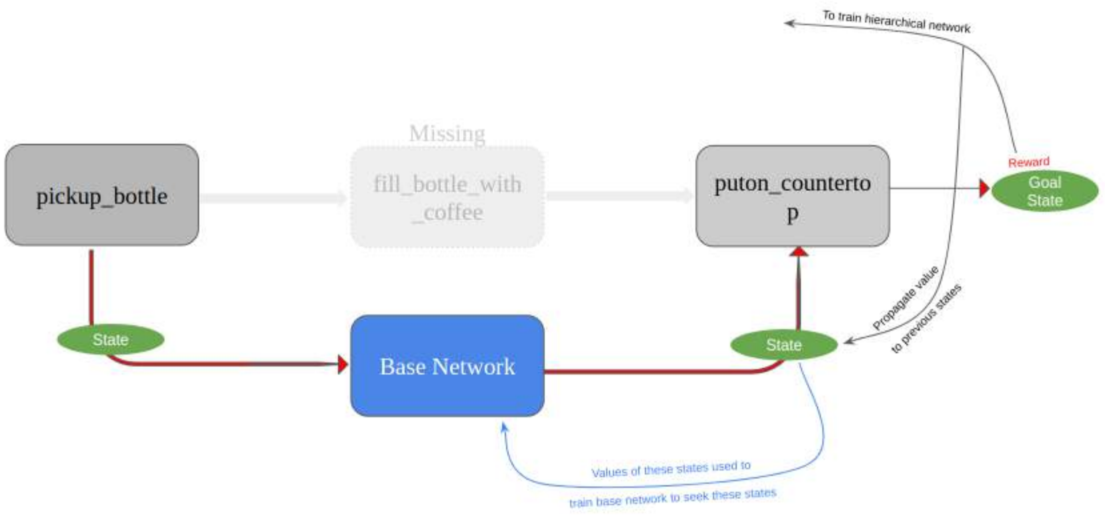
$$\begin{aligned} Error_{TD} &= [r + V_{high}(s_t)] + V_{base}(s_{t+1}) - V_{base}(s_t) \\ &\approx [r + V_{base}(s_t)] + V_{base}(s_{t+1}) - V_{base}(s_t) \\ &\approx r + V_{base}(s_{t+1}) \end{aligned}$$

which proves to be an exploding target for the learning algorithm. While this description is not mathematically accurate, it is an indicator of the problem with using `true reward+higher-level value` as the reward signal for the base network when the last option is missing.

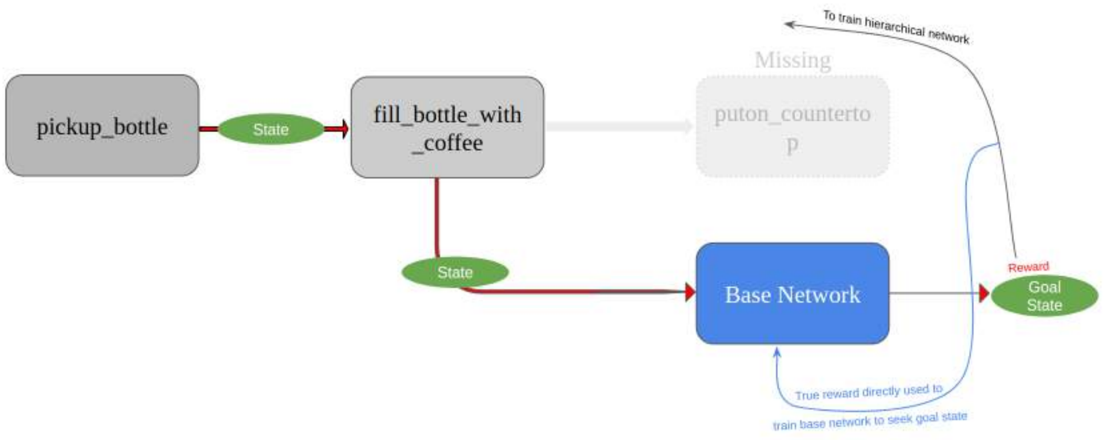
As a result, whenever it is the last option that is missing, the base network simply learns using `true reward` as the base reward.

2. As the primary goal was the implementation of the adaptive augmentation idea on the complex iTHOR environment, changing the base-reward formulation depending on which option was missing was the temporary adopted solution for this problem. However, incorporating these different cases is only possible if we know the missing option in advance, which is not the case in a real option retrieval and adaptation setting. A formulation for the pseudo-reward which is not dependent on the ideal position of the base network in the recipe would be better suited.

The next page contains Figure 9 which represents both the above cases in graphical form for better understanding.


To train hierarchical network
Missing
pickup_bottle
fill_bottle_with _coffee
puton_countertop
Reward
Goal State
State
Base Network
State
Propagate value to previous states
Values of these states used to train base network to seek these states

(a) when base network compensates intermediate option



(b) when base network compensates final option

Figure 9: The propagation of reward signal to the base network in both the cases, Case 1 and Case 2, discussed above. The task considered is `serve_beverage_coffee_on_table_countertop-v000`, whose recipe in terms of the pre-encoded options is `1. pickup_bottle, 2. fill_bottle_with_coffee, 3. puton_countertop`. In Case 1, an intermediate option is missing: the true task reward cannot propagate directly to the base network for training, therefore some pseudo-reward (in this case the higher-level value function) must be used to encourage the base network towards policies that drive it towards the reward state. In Case 2, the final option is missing: the reward is a direct consequence of the base policy, and can be used to directly train the base network.

The algorithm, however, still failed for most tasks - showing reasonable success only for tasks involving objects close to the starting location. Eventually, upon visualisation of the learned policies, it was discovered that the agent was stuck when it was far away from the objects of interest, spinning circles or running into the wall, essentially failing to learn the navigation aspect of such tasks.

It was then understood that the agent currently had no way of telling through the state, whether an object was nearby or far away, to the left or to the right, as only visibility (and the lack of it) was encoded. The original iTHOR environment treated only interactable (nearby) objects as visible, and the visibility distance itself was very short (1.5 metres). This means that a faraway object, even if present in the frame, was effectively invisible to the agent. Therefore, for example, the agent would not know when it can use the `MoveAhead` action to get closer to an object. The agent merely had a small region of visibility and interactability. As a result, the agent was unable to learn how to transition from an object being invisible to being visible, and once visible, how to get closer within interactable distance of the object.

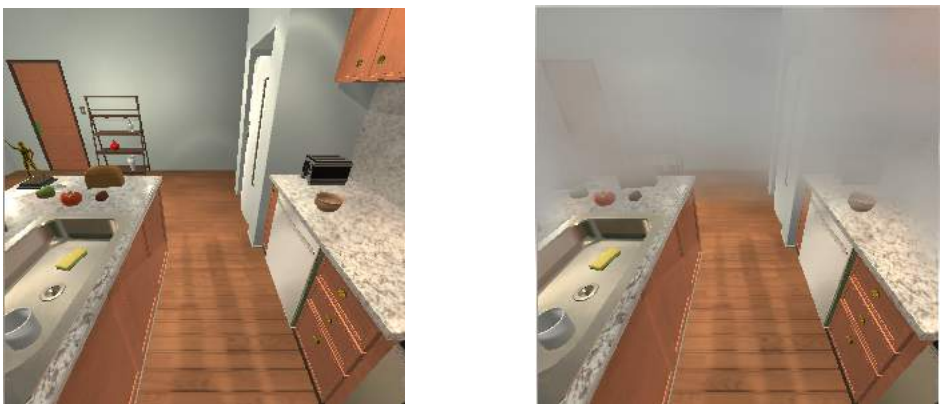


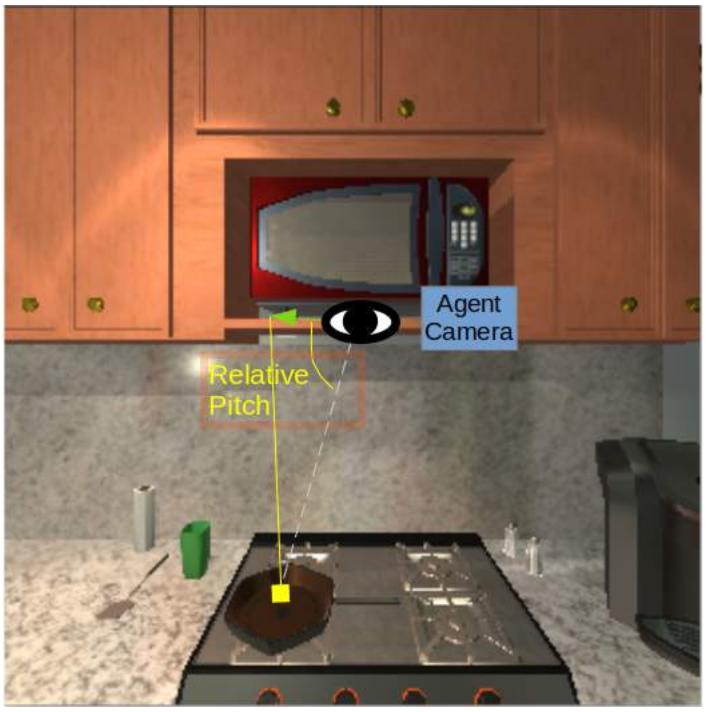
Figure 10: A true scene in an iTHOR environment (left) and what the agent could see at any given time (represented approximately) according to the default definition of visibility and interactibility (right). According to how visibility was defined by default, the agent could only see nearby objects that it could interact with. This hindered the agent's ability to learn to move towards an object that it sees in the distance, making learning navigation impossible.

The state representation was subsequently enriched to include more information about objects:

1. The visibility distance of the agent was increased so that all objects in the frame are visible. However, they are not all interactable in a realistic setting. Therefore, an extra condition was added for interactability: the relative distance between the agent and the object was calculated using the metadata, and only objects within the original visiblity distance of 1.5 metres were made interactable.

2. The relative pitch between the agent's camera angle and the objects' y-coordinates (height or depth) was calculated geometrically. The field of view of the agent was divided into three horizontal sections. If the relative pitch was $< -35°$, the object was categorised as being "down" and the agent would hopefully learn to use the `LookDown` action to keep the object in the field of view as it moved closer. If the relative pitch was $> 35°$, the object was categorised as being "up", and the agent would hopefully learn to use the `LookUp` action.

3. Similarly, the relative left-right angle between the agent's camera angle and the objects' x- and z-coordinates was calculated geometrically, and the field of view was divided into 3 regions: relative angle $< -35°, -35° <= angle <= 35°, > 35°$, so that the agent may learn to use the `RotateLeft` and `RotateRight` actions when required.



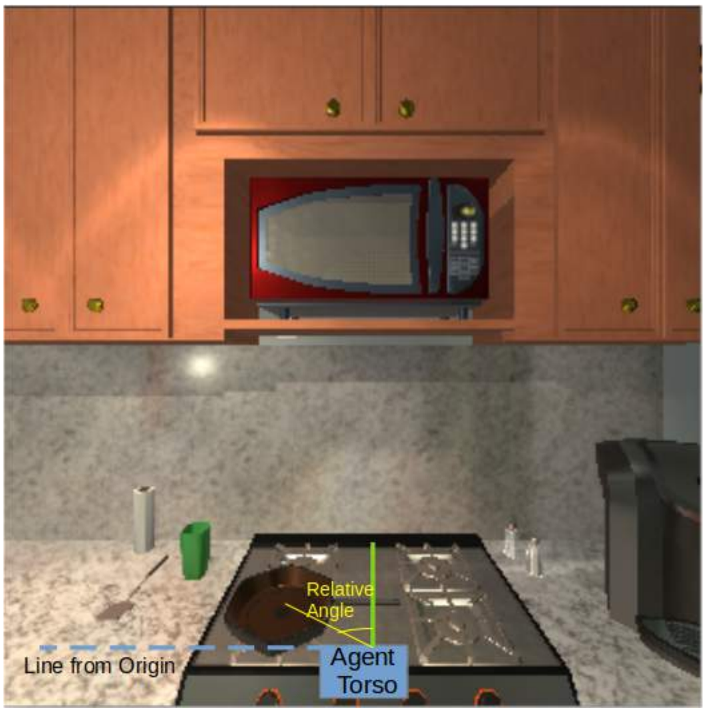
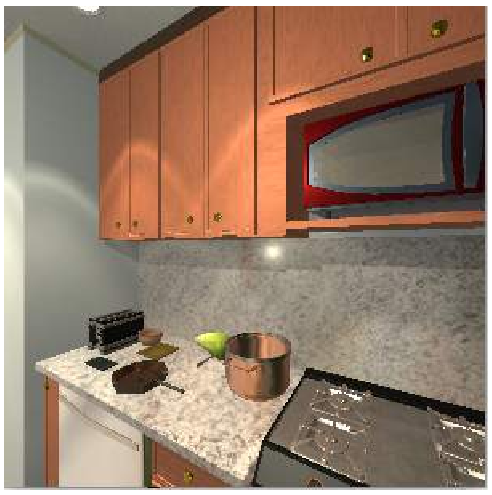
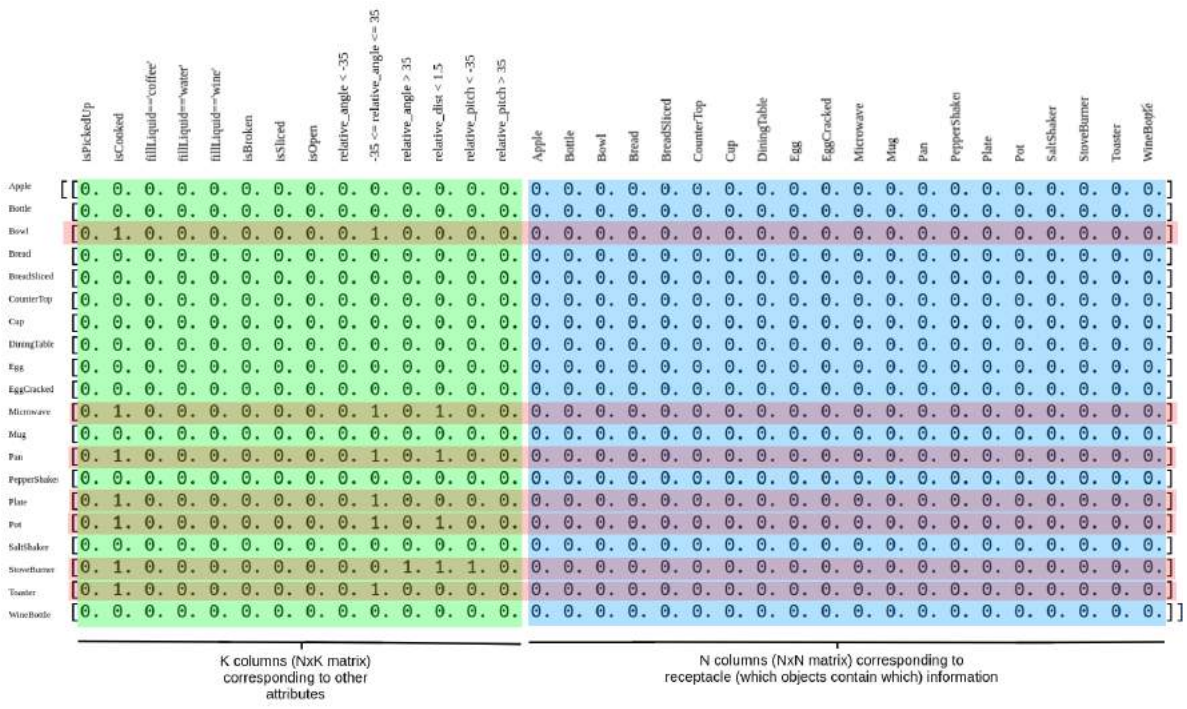


Figure 11: Calculation of the angles with respect to the object Pan: (a) Relative pitch calculated taking the difference between the agent camera pitch angle, and the object pitch angle calculated using the *arctan* of the y-coordinate distance divided by the xz-plane distance between the agent camera and the pan (b) Relative left-right angle calculated taking the difference in angle between the agent rotation, and the object position vector with respect to the agent torso in the xz-plane

The final state representation is shown graphically below in matrix form for better understanding. The actual state representation fed to the agent is the flattened version of the matrix, so that it can be used as input to the agent's MLP neural networks.





	isPickedUp	isCooked	fillLiquid=='coffee'	fillLiquid=='water'	fillLiquid=='wine'	isBroken	isSliced	isOpen	relative_angle < -35	-35 <= relative_angle <= 35	relative_angle > 35	relative_dist < 1.5	relative_pitch < -35	relative_pitch > 35	Apple	Bottle	Bowl	Bread	BreadSliced	CounterTop	Cup	DiningTable	Egg	EggCracked	Microwave	Mug	Pan	PepperShaker	Plate	Pot	SaltShaker	StoveBurner	Toaster	WineBottle
Apple	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Bottle	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Bowl	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Bread	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
BreadSliced	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
CounterTop	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Cup	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
DiningTable	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Egg	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
EggCracked	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Microwave	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Mug	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Pan	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
PepperShaker	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Plate	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Pot	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
SaltShaker	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
StoveBurner	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	1.	1.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Toaster	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
WineBottle	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.

K columns (NxK matrix) corresponding to other attributes

N columns (NxN matrix) corresponding to receptacle (which objects contain which) information

Figure 12: An example scene (top) and the corresponding final, enriched state representation (bottom). Only the rows in red, which correspond to objects that are visible (Bowl, Microwave, Pan, Plate, Pot, StoveBurner, Toaster) have non-zero entries in them: this gives the state an egocentric view typical of any navigation agent. In this example, Toaster and Plate are not within the interactable range of 1.5 metres even though they are visible. The existence of such states allows the agent to learn to use the `MoveAhead` action in such contexts to move towards objects of interest. Also, StoveBurner has a relative pitch of $< -35°$ and a relative angle of $> 35°$, which means it is downwards and rightwards. This state information allows the agent to learn to use the `RotateRight` and `LookDown` actions in this situation, if StoveBurner was an object of interest.

With this enriched state representation, adaptive augmentation finally showed success on the complex iTHOR environment.

4.2.4 Tasks

The tasks that the agent is trained and evaluated on, are generated using set sequences (or recipes) of the pre-encoded options. A few examples of tasks and their recipes are tabulated in Table 5.

Task	Recipe	Reward Conditions
put_crockery_bottle_on _table_countertop-v000	-pickup_bottle -puton_countertop	- bottle in countertop['receptacleObjs']
serve_beverage_coffee_on _table_countertop-v002	-fill_mug_with_ coffee -pickup_mug -puton_countertop	- mug in countertop['receptacleObjs'] - fillLiquid=='coffee' in mug
put_egg_on_plate-v000	-pickup_egg -puton_plate	- egg in plate['receptacleObjs']
put_egg_on _countertop-v000	-pickup_egg -puton_countertop	- egg in countertop['receptacleObjs']
serve_coffee_egg_on _table_countertop-v000	-fill_bottle_with _coffee -pickup_bottle -puton_countertop -puton_countertop -break_egg -pickup_eggcracked -puton_countertop	- bottle in countertop['receptacleObjs'] - fillLiquid=='coffee' in bottle - egg['isBroken']==True - eggcracked in countertop['receptacleObjs']
serve_microwaved_egg_on _table_countertop-v000	-break_egg -pickup_eggcracked -puton_microwave -cookon_microwave -pickup_eggcracked -puton_countertop	- eggcracked in countertop['receptacleObjs'] - eggcracked['isCooked'] ==True - microwave['isOpen']==True

Table 5: A few task recipes used for training and evaluation of the agent

Algorithmic Details for AI2THOR Experiments

Both the higher-level network and the base network were trained using A2C once again. The same state-value function as the higher-level network was not used for the base network, as in the CraftWorld experiments (as this is incorrect). Instead a separate value function network (critic) was maintained for the base network.

- State Space: 20×34 multi-binary matrix flattened into a $680-$dimensional vector
- Primitive (Base) Action Space: 44 object manipulation and navigation actions derived from the iTHOR API actions, giving them fixed parameter values
- Higher-level/attention policy network architecture: $680 \times 64 \times 64 \times$ K+1
- Value network architecture: $680 \times 64 \times 64 \times 1$
- Activation function: tanh()
- Learning rate: 3×10^{-4}
- Optimizer: RMSProp ($\epsilon = 10^{-5}$)
- Gradient clipping (max = 0.5) and normalisation used
- Discount factor γ: 0.99
- Value function loss coefficient (critic loss): 0.8
- Entropy coefficient: 0.5
- Generalized Advantage Estimate (GAE) used to estimate advantages, $\lambda = 0.99$
- Rollout buffer used for mini-batch training, size = 10

- Base network policy architecture: $680 \times 64 \times 64 \times 44$
- Value network architecture: $680 \times 64 \times 64 \times 1$
- Activation function: tanh()
- Learning rate: 3×10^{-4}
- Optimizer: RMSProp ($\epsilon = 10^{-5}$)
- Value function loss coefficient (critic loss): 0.8
- Entropy coefficient: 0.5
- Rollout buffer used (size = 10), advantages are calculated using GAE again in a backward accumulative manner. However, in time steps where the base network is not used, the accumulation has to be broken and GAE calculation is restarted whenever the base network is picked again, even within a single buffer of 10 transitions.
- All other hyperparameters same as the higher-level network. A clear hyperparameter search has not been performed.

Results

The reward curves for 5 of the 6 tasks tabulated in Table 5 are displayed below. Three kinds of scenarios are considered:

- Hrl-Adapt_N-1 and Hrl-Adapt_N-2: The scenarios relevant to adaptation, with one missing option and two missing options for completing the task respectively.
- Hrl-N: The exact set of options required for the task is available, and a hierarchical agent is trained over them.
- Hrl-Full: The full set of 42 pre-encoded options required for the task is available, and a hierarchical agent is trained over them.

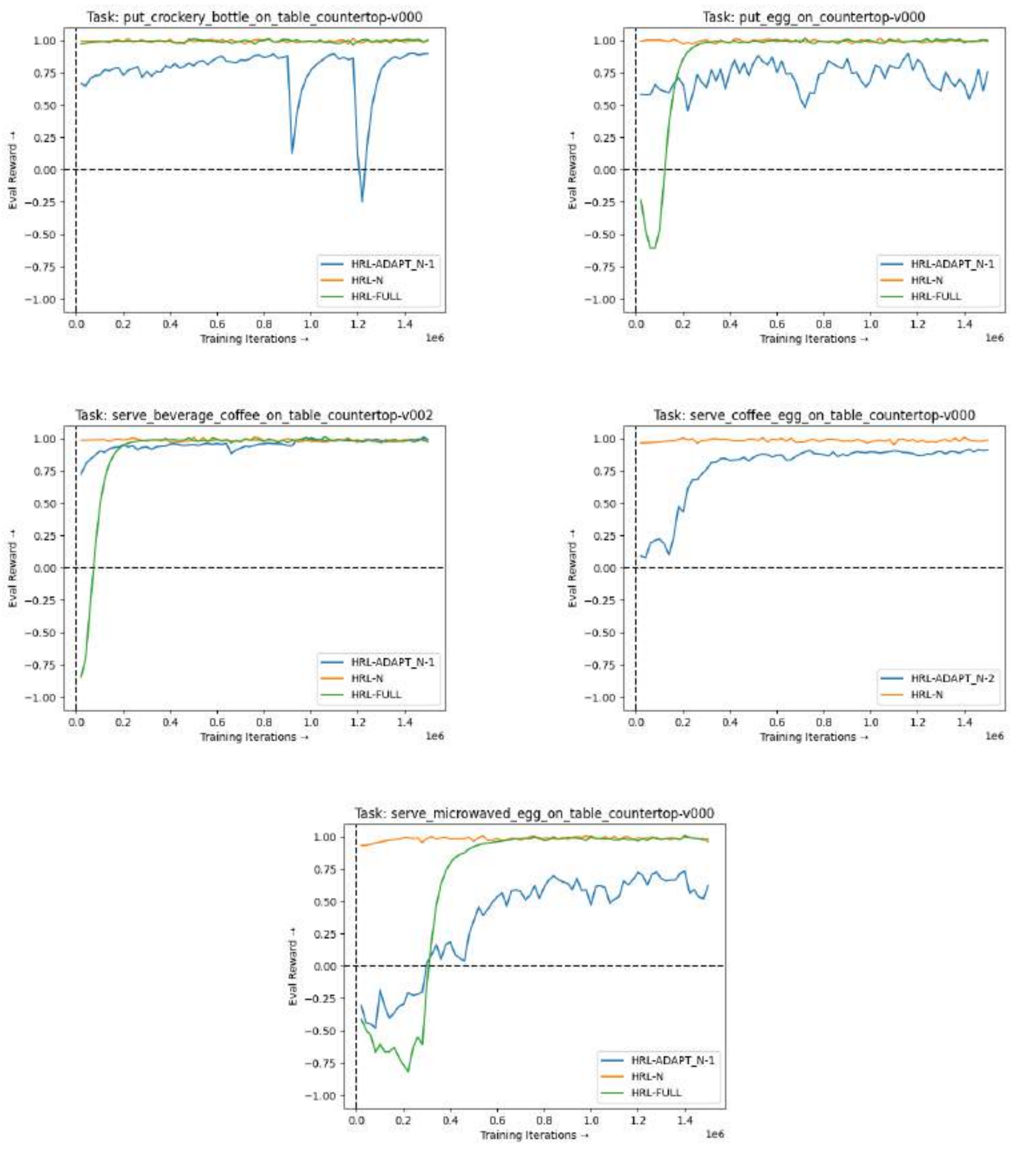


Figure 13: Reward curves for 1 training run of each task, upto 1.5M training steps

The value of final reward reached at convergence cannot be compared in this evaluation, as there is a varying length penalty on the reward depending on whether a pre-encoded option or the base network accomplishes any particular subtask. Currently, as the pre-encoded options are "teleporting", there is no good way to impose a length penalty on them that is indicative of the actual number of steps that the option might take if it used primitive actions (this was calculable in CraftWorld as the Manhattan distance is easy to obtain in the gridworld setting). Therefore, the teleporting options and the primitive actions both have an equal length penalty of 0.002. This means that a teleporting option might accomplish a subtask with just 0.002 length penalty, while the base network might use a long sequence of primitive actions to do the same subtask, and therefore incur a larger length penalty (0.002 for each primitive action). This discrepancy would be eliminated if the pre-encoded options were also replaced by neural policies that are actually pre-trained to perform their corresponding subtasks.

It is worthwhile to look at the convergence rate or sample complexity. In `put_egg_on_countertop-v000` and `serve_beverage_coffee_on_table_countertop-v002`, it appears that HRL-ADAPT_N-1 converges faster than the HRL-FULL setting, and at nearly the same rate as HRL-FULL in `serve_microwaved_egg_on_table_countertop-v000`. These conclusions demonstrate that adaptive augmentation, with some refinement, can be used effectively in the option retrieval/option indexing context with one missing option.

In `serve_coffee_egg_on_table_countertop-v000`, HRL-ADAPT_N-2 is also considered, and adaptation does well to compensate for two missing options. This is encouraging evidence towards the use of this idea in the context of several missing options. At the very least, having the adapting base network would do better than not having it. The reward curves for all the experiments above where options are missing without a base network would be all -1, as there is no way to perform the task if the retrieved options are insufficient and there is no way to adapt.

The missing options in each HRL-ADAPT_N-1 (or HRL-ADAPT_N-2) scenario are tabulated below.

Task	Missing Option(s)
`put_crockery_bottle_on_table_countertop-v000`	`-pickup_bottle`
`put_egg_on_countertop-v000`	`-puton_countertop`
`serve_beverage_coffee_on_table_countertop-v002`	`-puton_countertop`
`serve_coffee_egg_on_table_countertop-v000`	`-pickup_bottle`
	`-break_egg`
`serve_microwaved_egg_on_table_countertop-v000`	`-cookon_microwave`

Table 6: The missing options in the evaluated HRL-ADAPT_N-1 and HRL-ADAPT_N-2 tasks

Visualisation of the base network policies: As further evidence of the policies learnt by the base network compensating for the missing option, shown below is several trajectories of the base policy learnt in the task `serve_beverage_coffee_on_table_countertop-v002` with `puton_countertop` missing. The base network learns to put the mug on the counter-top in several ways after the agent has filled the mug with coffee and picked it up using the available pre-encoded options missing.

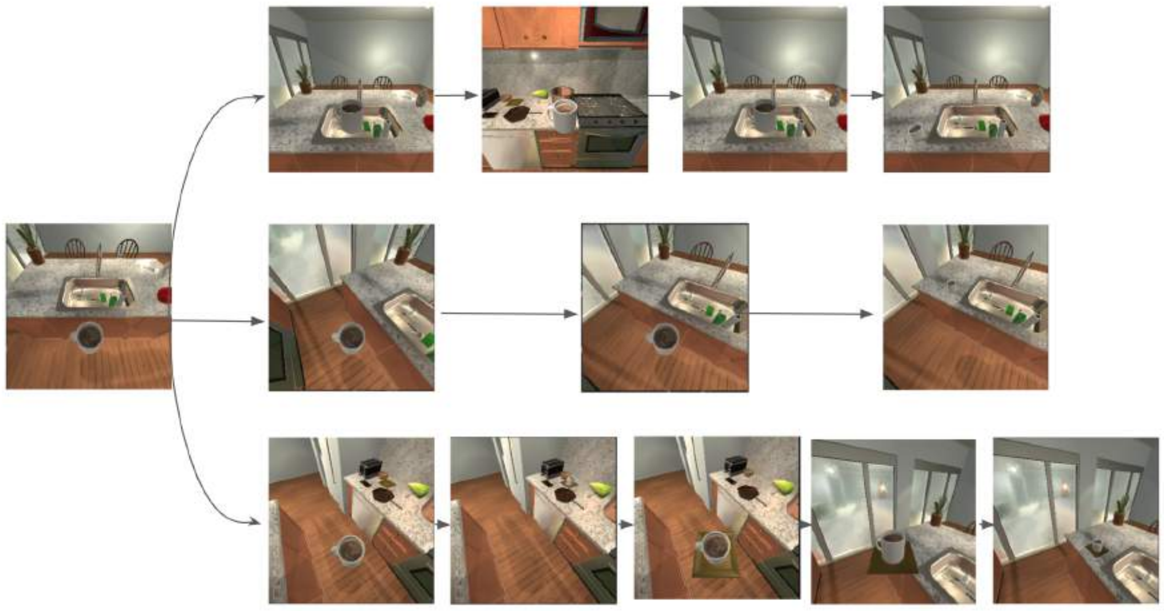


Figure 14: Several trajectories of the `puton_countertop` segment of the task `serve_beverage_coffee_on_table_countertop-v002` that the base network learns to perform. There are several spurious actions in between that have not been visualised - therefore, the lengths of successful trajectories are not 3, 4, or 5 for example.

5 Learning Options out of Base Network Policies

The base network in the Adaptive Augmentation idea learns a policy to compensate for a step (or steps) in the recipe of a task. This means it is learning to do a useful subtask - which might help in future tasks. Therefore, it would be helpful to encapsulate this policy as an option, and add it to the option library. A framework of this nature would enable a true continual learning framework of options, which is a meaningful eventual goal for this project.

However, in the current state, there exist a few obstacles towards achieving this framework.

1. The learnt base network policy from a single task is usually not generalisable. Say, the base network learns to compensate for a `puton_countertop` option in the task `put_crockery_bottle_on_table_countertop-v000`, it is able to perform the `puton_countertop` subtask, but only from the part of the state space it has seen.

2. Secondly, the `puton` options in the iTHOR environment, for example, are not only required to generalise positionally, but also in an object-agnostic manner (as the picked up object information is also a part of the state). A base policy that learnt to put a bottle on the countertop should be able to extend that to placing other objects on the countertop.

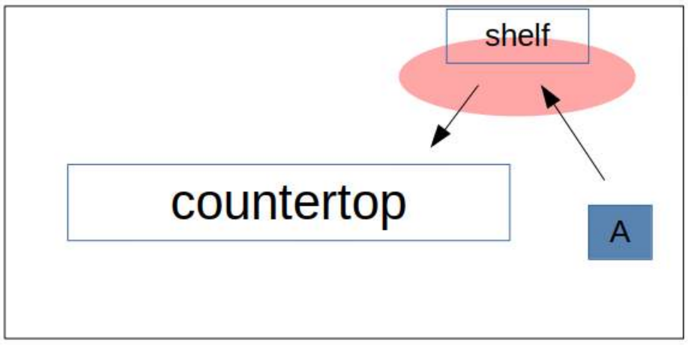


Figure 15: The policy learnt by a base network is local only to some states around the place where it starts off from. In the above figure, the agent is represented by the blue square marked ’A’. Assuming the bottle is on the shelf, and the task is `put_crockery_bottle_on_table_countertop-v000` as discussed above, with `pickup_bottle` available and `puton_countertop` missing, the `pickup_bottle` option takes the agent near the shelf. The base network then learns to ”put on countertop” from only this local region of the state space, and does not transfer as a generalised option.

3. If the base network learns to compensate for the absence of two (or more) options, the base network learns a combination of two subtasks (for different parts of the state space). In such a case, before we think about generalisation over the state space, it might be necessary to split the policy into two (or more) policies that perform one subtask each, so that we can continually expand the option library without redundancy.

The following are some heuristics that might lay the pathway towards future work:

- **For positional generalisation:** Consider a sequence of tasks, say Task 1, Task 2, and so on. Assume the case where only one of the required options is missing from the set of retrieved options. The base network might learn a localised policy for a subtask during adaptation for Task 1. For Task 2, we allow this base policy from Task 1 to continue training in the form of a ”new option” added to the set of options. If the missing option in Task 2 is quite similar to what the base policy learnt in Task 1, the ”new option” finetunes its policy to perform a similar task in a generally different part of the state space. In order to prevent catastrophic forgetting, the updates could be done in a manner similar to methods such as GEM, where the gradient step is taken in a direction that does not decrease performance on previous tasks.
- Alternatively, when training, the agent could be made to undergo a random positional reset after the execution of every option. This allows the base network to perform the subtask of the missing option from all over the state space, leading to a generalisable policy being learnt.
- Finally, state abstraction methods could be looked into, so that several states can be interpreted as similar according to the base network if it only focuses on the important features of the state. However, an immediate problem that arises here is that the important features of the state depend on the option that is missing, which we do not know in advance. An idea to overcome this, is to identify the common features of the state recurring frequently in successful trajectories of the training task.
- **For object-agnostic generalisation:** For iTHOR environments, the only object-agnostic generalisation required is that of the picked up object. A `puton_countertop`-compensating base network that learns to put a bottle on the countertop should easily

generalise to other objects. This could be done by abstracting the `isPickedUp` Boolean attribute of all objects (stored in the state) into a single bit, which is 1 if any object is picked up, and 0 if none is picked up. This collapses all states where an object is picked up into one.

- When the base network learns to compensate for multiple (say, two) missing options, it learns two different subtasks locally in two different parts of the state space. In such a case, it might help to split the learnt policy into several policies based on discrete regions of the state space that it covers, and then attempt generalisation.

6 Future Research Directions

In summary, some directions for future work and improvements to the present work are:

- Extending this work into true continual learning for options, by:
  - generalising the base network policies, from whatever tasks something useful is learnt, to large parts of the state space, and subsequently packaging them as options to add into the option library,
  - splitting the base network policies effectively when it learns two different subtasks, in order to fulfill the above objective, and
  - distilling the option library into an encoded basis, similar to [25], to keep the option library compact and eliminate redundancy as new options are continually added to the library
- Implementation of off-policy intra-option base network training when actual options made of primitive actions are used, as mentioned in the third point under Section 4.1

7 References

1. Gordon J. Berman, Daniel M. Choi, William Bialek, and Joshua W. Shaevitz. 2014. *Mapping the stereotyped behaviour of freely moving fruit flies. Journal of The Royal Society Interface, 11(99), 20140672.*

2. Matthew M. Botvinick, Yael Niv, and Andrew G. Barto. 2009. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition, 113(3), 262–280.*

3. Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. 2018. Data-Efficient Hierarchical Reinforcement Learning. *Proceedings of the Conference on Advances in Neural Information Processing Systems, 2018.*

4. Mohammad Ghavamzadeh, Sridhar Mahadevan, and Rajbala Makar. 2006. Hierarchical multi-agent reinforcement learning. *Autonomous Agents and Multi-agent Systems 13, 2 (2006), 197–229.*

5. Matheus R. F. Mendonça, Artur Ziviani, and André M. S. Barreto. 2019. Graph-based skill acquisition for reinforcement learning. *ACM Computing Survey 52, 1 (Feb. 2019).*

6. Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition

7. Ronald Parr and Stuart Russell. 1998. Reinforcement learning with hierarchies of machines. *Proceedings of the Conference on Advances in Neural Information Processing Systems, 1997.*

8. Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence 112, 1–2 (Aug. 1999), 181–211.*

9. Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. 2020. Towards Continual Reinforcement Learning: A Review and Perspectives.

10. David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient Episodic Memory for Continual Learning. *In Proceeding of the Conference on Advances in Neural Information Processing Systems 2017.*

11. Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K. Dokania, Philip H. S. Torr, and Marc'Aurelio Ranzato. 2019. On Tiny Episodic Memories in Continual Learning.

12. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *Conference on Advances in Neural Information Processing Systems Deep Learning Workshop, 2013.*

13. Paul Ruvolo and Eric Eaton. 2013. ELLA: An Efficient Lifelong Learning Algorithm. *Proceedings of the 30th International Conference on Machine Learning, 2013.*

14. Haitham Bou Ammar, Eric Eaton, Paul Ruvolo, and Matthew E. Taylor. 2014. Online Multi-Task Learning for Policy Gradient Methods. *Proceedings of the 31st International Conference on Machine Learning, 2014.*

15. Akhil Bagaria, Jason K Senthil, and George Konidaris. 2021. Skill Discovery for Exploration and Planning using Deep Skill Graphs. *Proceedings of the 38th International Conference on Machine Learning, 2021.*

16. Emma Brunskill and Lihong Li. 2014. PAC-inspired Option Discovery in Lifelong Reinforcement Learning. *Proceedings of the 31st International Conference on Machine Learning, 2014.*

17. Arthur Aubret, Laetitia Matignon, and Salima Hassas. 2020. ELSIM: End-to-end learning of reusable skills through intrinsic motivation. *Proceedings of the European Conference on Machine Learning, 2020.*

18. Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J. Mankowitz, and Shie Mannor. 2016. A Deep Hierarchical Approach to Lifelong Learning in Minecraft.

19. Janarthanan Rajendran, Aravind Srinivas, Mitesh M. Khapra, P Prasanna, and Balaraman Ravindran. 2017. Attend, Adapt and Transfer: Attentive Deep Architecture for Adaptive Transfer from multiple sources in the same domain. *Proceedings of the 5th International Conference of Learning Representations, 2017.*

20. Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation.

21. Richard S. Sutton. 1988. Learning to Predict by the Methods of Temporal Differences.

22. Vijay R. Konda and John N. Tsitsiklis. 1999. Actor-Critic Algorithms.

23. Kushal Chauhan, Soumya Chatterjee, Akash Reddy, Balaraman Ravindran, Pradeep Shenoy. 2022. Matching Options to Tasks Using Option-Indexed Hierarchical Reinforcement Learning. *https://arxiv.org/pdf/2206.05750.pdf*

24. Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. *In International Conference on Learning Representations. URL: https://openreview.net/forum?id=rygGQyrFvH.*

25. The CraftWorld Environment. *https://github.com/jacobandreas/psketch*

26. Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *https://arxiv.org/abs/1712.05474*

27. Option Encoder: A Framework for Discovering a Policy Basis in Reinforcement Learning. 2020. Arjun Manoharan, Rahul Ramesh, and Balaraman Ravindran. *Proceedings of the European Conference on Machine Learning, 2020.*