

https://dspace.org/

Learning factored skill representations for improved transfer

Cockcroft, Matthew

2022
https://hdl.handle.net/10539/36949

Downloaded from a DSpace repository. https://dspace.org/

University of the Witwatersrand
School of Computer Science and Applied Mathematics
Faculty of Science

Learning Factored Skill Representations for Improved Transfer

Matthew Cockcroft
(1415624)



WITS
UNIVERSITY

Supervisors:
Prof Benjamin Rosman
Dr Steven James

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

August 2022, Johannesburg

Abstract

The ability to reuse skills gained from previously solved tasks is essential to building agents that can solve more complex, unseen tasks. Typically, skills are specific to the initial task for which they were learned to assist in solving and it remains a challenge to determine which features between a set of two tasks need to be similar in order for that skill to apply to both tasks. Current approaches have shown that learning generalised skill representations allows for the skills to be successfully transferred and reused in accelerating learning across multiple similar new tasks. However, these approaches require large amounts of domain knowledge and handcrafting to learn the correct representations. We propose a novel framework for autonomously learning factored skill representations, which consist only of the variables which are relevant to executing each particular skill. We show that our learned factored skills significantly outperform traditional unfactored skills, and match the performance of other methods, without requiring the prior expert knowledge that those methods do. We also display our framework's applicability to real-world settings by demonstrating its ability to scale to a realistic simulated kitchen environment.

Declaration
University of the Witwatersrand, Johannesburg
School of Computer Science and Applied Mathematics
Senate Plagiarism Policy

I, Matthew Cockcroft, (Student number: 1415624) am a student registered for COMS8003A - Masters by Dissertation in the year 2022.

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that ALL the work submitted for assessment for the above course is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this in not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: 

Signed on 9th day of August , 2022 in Johannesburg.

Acknowledgements

First and foremost, thank you to Benji and Steve for all the guidance, support and understanding that you have shown me over the last two years. Knowing that you would be there to meet with me and discuss my progress each week without fail has been a driving factor in providing me with the motivation to persist through this dissertation. An additional thank you to Steve for assisting me with CHPC compute, so that I no longer had to worry whenever the cluster was down (which was often).

To Pravesh, thank you for getting me interested in reinforcement learning in the first place and helping me to formulate the initial idea upon which this research was based.

To my friends and family, thank you for putting up with all my whinging and moaning, whether it be research-related or not. I am greatly indebted to all of you for the time that you have dedicated to me. Last, but not least, a big thank you to my parents for the patience that you have shown me and for always being there to encourage me whenever I was feeling downhearted. A particularly large thank you to my mom for taking the time to wade through all the technical jargon while proof reading this dissertation.

This research has been supported in part by the National Research Foundation of South Africa (Grant Number: 17808). Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Contents

Abstract i

Declaration ii

Acknowledgements iii

1 Introduction 1

1.1 Contributions 2

1.2 Structure 3

2 Background and Related Work 4

2.1 Reinforcement Learning 4

2.1.1 Q-Learning 5

2.2 Hierarchical Reinforcement Learning 6

2.3 Factored Representations 7

2.3.1 Object-Oriented MDPs 7

2.3.2 Factored MDPs 9

2.4 Decision Tree Based Models 10

2.5 Transfer Learning 11

2.6 Related Work 13

2.7 Summary 14

3 A Framework For Learning Factored Skills 15

3.1 Learning Factored Options 15

3.1.1 Minigrid Example 18

3.2 Refining Representations 21

3.3 Variable Grouping 21

3.4 Evaluation of Transferability of Factored Options 23
3.5 Summary 24

4 Experimentation **25**

4.1 Environments 25
4.2 Baseline Comparisons 25
4.3 Initial Experiments 27
4.4 Locally Observable Domains 30
4.4.1 Variable Groupings 30
4.5 Redundant and Noisy Variables 35
4.6 AI2-THOR 37
4.7 Summary 43

5 Conclusion **44**

5.1 Future Work 45

Appendices **49**

A Decision Tree Models **50**

B Grid Size Experiments **52**

C Redundant and Noisy Domain Experiments **53**

D Hardware Specifications **55**

Chapter 1

Introduction

The ability to leverage previously gained knowledge is crucial in accelerating the learning of new tasks when designing intelligent agents [Taylor and Stone 2009]. However, one of the main difficulties in this approach lies in learning initial knowledge which will be applicable to future tasks. If the knowledge is too specific, then it will very rarely be useful for learning new tasks, but if the knowledge is too general, then the performance gain from this knowledge will be minimal in the current task. Consequently, it would be desirable for an agent to learn knowledge in a representation that can generalise across multiple future tasks, while still being specific enough so as to provide a significant acceleration in learning those tasks.

We focus on the skills learned when solving previous tasks, which can be thought of as the *knowledge* that we wish to reuse when attempting to solve new tasks. The *options* framework [Sutton *et al.* 1999] has shown to be successful in constructing reinforcement learning agents which are capable of learning such skills. These skills are represented as action hierarchies, which consist of primitive action sequences, which are chained together to form higher-order actions or *options*. The effectiveness of these skills in new environments depends on the similarity between the environment in which the skill was learned and the environment in which the skill is being reused. However, if agents can learn more general representations of these skills, then the required similarity between environments is reduced and these skills become applicable to a greater number of tasks.

We can take inspiration from how humans are able to focus on a specific task and what is required to solve it, while ignoring any extraneous information presented to them. For example, consider an agent working in a kitchen environment, as shown in Figure 1.1. The agent has sensors which provide it with various information, including the location of objects, such as the frying pan, stove, sink and an apple. If the agent is given the task to cook an egg, it should be interested in the location of the stove and the pan, but should be able to ignore the data regarding the sink and apple. Once the agent has learned the skill of cooking eggs, it should then be able to effectively reuse that skill in a new kitchen environment, even if some unrelated items are no longer present or if the room colours and lighting varies from the original environment.

If we consider all of the sensory information available to the agent as the global set, then we wish to learn a *factored* representation of this set, which would be a subset containing only information which is relevant to the task which is being learned. The two most common approaches in reinforcement learning for learning such representations are *Factored Markov Decision Processes* [Boutilier *et al.* 2000] and *Object-Oriented Markov Decision Processes* [Diuk *et al.* 2008]. While both of these frameworks are effective at modeling factored domain representations, they require knowledge of the structure of the environments to be known beforehand and rely heavily on handcrafted features which need to be manually changed each time the environment is updated. This process does not scale well, as it becomes significantly more time consuming as the domain complexity increases.

As a result, the aim of our work is to build a framework which would allow for agents to autonomously learn *factored* representations, without requiring prior domain knowledge and expert handcrafting. Such a framework should be able to learn these representations quickly, from few training samples, while the representations themselves should allow for the agent's skills to generalise across various similar tasks. This will ultimately lead to agents that can effectively leverage the skills learned when solving one task to accelerate learning of new unseen tasks, and successfully scale to more complex environments without requiring prior domain knowledge.

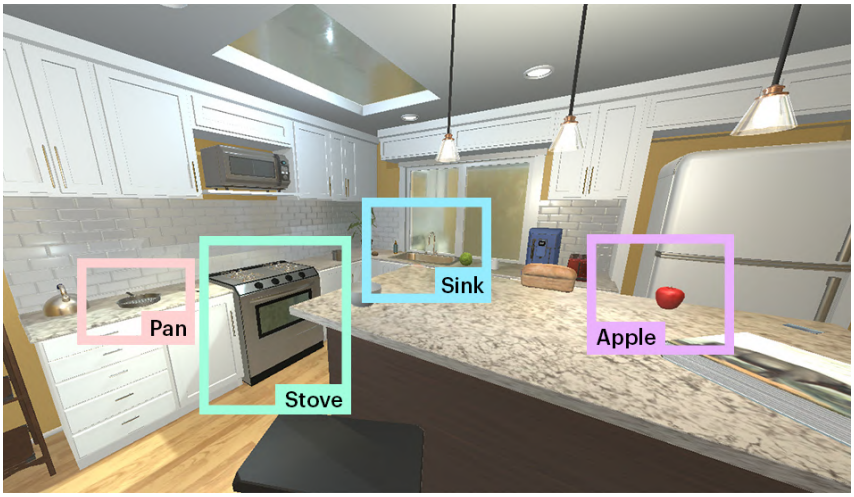


Figure 1.1: Example of a simulated kitchen environment. The different coloured blocks indicate objects within the environment for which the agent has sensory data, but not all of these objects may be important for the current task being carried out.

1.1 Contributions

To tackle the issue of laborious handcrafting of features and domain structures, we propose a framework which is able to learn factored skill representations autonomously. We extend the options framework, by introducing *factored options* which are represented in a way that improves generalisability and transfer across domains. Our approach learns these factored option representations using a continuous refinement process, in which the factoring is updated after each new training domain presented to it. We show that the framework is able to converge to a learned factoring quickly, without having to interact with an excessive number of training domains.

We present empirical results which show that our learned factored options consistently outperform traditional options when evaluated on their ability to accelerate learning in new domains. We also compare our framework to baselines that require extensive human expertise and handcrafted domain knowledge. Our method is fully autonomous, and yet we demonstrate that it is able to match the performance of those baseline methods.

We also demonstrate that our agent is able to adapt to domains containing several noisy variables, similar to what is likely to be encountered in real-world settings. We show that in such domains our framework is able to significantly outperform other baseline methods, despite all of the prior expert knowledge they have been provided with. Finally, we also showcase our framework's ability to scale to a larger, realistic simulated kitchen environment, in which our agent is able to learn factored representations of cooking-related skills, that allows it to successfully transfer the skills to new kitchen configurations.

1.2 Structure

The structure of this dissertation is organised as follows: in Chapter 2, we detail the concepts, notations and definitions upon which our work is built, as well as providing an overview of the current work in this field. In Chapter 3, we discuss the methodology of this work and elaborate on the key contributions, including how we create a framework for factored skill representations and how we intend to utilise it to accelerate learning. Then, in Chapter 4 we detail the experiments which we use to evaluate the performance of our framework compared to other methods, elaborating on the experiment set-up and analysing the results from each experiment. Finally, Chapter 5 summarises the contributions of this work and looks at future avenues upon which it may be extended.

Chapter 2

Background and Related Work

In this chapter, we expand upon the idea of learning generalisable skills in reinforcement learning and introduce the fundamental concepts required for implementing these ideas. We provide a thorough explanation of reinforcement learning in Section 2.1, and then expand upon the field of hierarchical reinforcement learning and how skills can be incorporated into reinforcement learning domains in Section 2.2. We then detail the concept of factored representations and their benefits with regard to generalisation in Section 2.3, specifically focusing on two methods commonly used in conjunction with hierarchical reinforcement learning, namely Factored Markov Decision Processes and Object-Oriented Markov Decision Processes. Finally, in Section 2.5, we discuss relevant methods and evaluation criteria for transfer learning.

2.1 Reinforcement Learning

Reinforcement learning is a trial-and-error based learning paradigm in which a decision-making entity, known as an agent, performs a set of actions in an attempt to maximise its numerical reward. The agent repeatedly performs actions, receiving a reward after each action and transitioning to a new state, until a terminal state is reached. A state s refers to all of the information used to describe the environment at a particular timestep, some of which may not be known to the agent.

Typically, reinforcement learning methods focus on problems that preserve the Markov property, where all relevant information required to determine the agent's transition after performing an action is included in the state data it receives. More formally, the probability of transitioning from state s to state s' is only dependent on state s and the action performed to cause the transition, rather than depending on any of the preceding states. Sutton and Barto [1998] describe how problems such as these can be modelled using a Markov Decision Process (MDP):

An MDP is a tuple defined as $M = (S, A, P, R, \gamma)$, whereby:

- S is the set of all possible states of the environment
- A is the set of actions available to the agent
- $P(s'|s, a)$ is the transition probability function, which describes the probability that taking action a in state s will result in a transition to state s'
- $R(s, a, s')$ is the reward function, which is the reward obtained when transitioning to state s' by taking action a in state s
- $\gamma \in [0, 1]$ is the discount factor, which determines how much future rewards are valued

The goal of reinforcement learning is to learn an action policy which maximises the cumulative expected reward of the agent. An action policy $\pi(a|s)$ gives the probability of an agent choosing action a when in state s. To learn the optimal policy π^* the notion of state values is required. State value $V^\pi(s)$ is the expected return from following policy π when starting in state s. To solve for the optimal policy, the value of state s can be defined in a recursive form using the *Bellman equation*, where $V^*(S) = V^{\pi^*}$ is an optimal value function and π^* is an optimal policy:

$$V^*(s) = \max_a \sum_{s'} P\left(s'|s,a\right)\left[R\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right] \tag{2.1}$$

This optimal value function can then be solved using dynamic programming techniques such as *value iteration* [Bellman 1957]. Value iteration works by first arbitrarily initialising the values of V, and then sweeping through all state values and updating them using the Bellman equation (2.1).

2.1.1 Q-Learning

The issue with value iteration is that for each state value update a one-step lookahead is required, which become infeasible for large state spaces. An alternative is to make use of *Q-values*, which map state-action pairs and are represented by $Q^\pi(s,a)$. They represent the expected cumulative reward obtained when performing action a in state s, and thereafter following action policy $\pi(a|s)$ until reaching a termination state.

Q-Learning [Watkins and Dayan 1992] is an algorithm which is used to iteratively learn an MDP's optimal Q-value function. The Q-value function is defined as $Q^*(s,a) = R(s,a,s') + \gamma V^*(s')$, where $0 < \gamma < 1$ is the discount factor, and V^* represents the expected cumulative reward for optimal policy π.

Before learning Q, all values are first initialised to an arbitrary fixed value. Then at a given time-step t, the agent in state s executes action a and transitions to state s', receiving reward r. The update rule for the approximation of Q, which is denoted as $\hat{Q}$, can then be calculated as:

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)] \tag{2.2}$$

where α is the *learning-rate*. The agent is now able to learn this Q-value function by acting directly with the environment, rather than having to iterate through all possible states and actions [Watkins and Dayan 1992]. Once the Q-value value function has been obtained, it is then trivial to solve for the optimal policy π^* by acting greedily with respect to $Q^*(s,a)$.

2.2 Hierarchical Reinforcement Learning

The use of single-step actions in a standard MDP, known as primitive actions, is typically adequate for solving reinforcement learning problems in which there are abundant rewards. Oftentimes though the reward is sparse and is only received upon completion of the task. Such problems require long sequences of correct actions during exploration before a reward is received, which is infeasible for larger state spaces. Hierarchical reinforcement learning is a form of action abstraction which combats this problem by chaining sequences of low-level actions into high-level skills which can be instantiated together, allowing the agent to take greater and more meaningful steps through the state space.

For example, consider again the kitchen agent discussed in Chapter 1. Given the task of cooking eggs, we may decompose this task into a set of smaller sub-tasks, such as walking to the eggs, picking the eggs up and placing the eggs in the pan. To solve one of these sub-tasks, such as fetching the eggs, we may chain all of the actions required into one temporally-extended action. Once the agent has fetched the eggs it will need to decide on the next action or skill to execute, but while it is executing the *fetch eggs* skill it does not need to think about each individual action taking it towards the fridge. If the agent only receives a reward once the entire task is complete and we are operating with low level actions such as motor torques to move the agent, then it will take thousands of correct actions before a reward is received, which is not feasible. In contrast, with higher level skills and task decomposition, the agent will only need to make three correct decisions, before a reward is received.

A common approach to task decomposition is through the use of the options framework [Sutton *et al.* 1999], which provides a formal definition of a skill in a reinforcement learning setting. Options make use of the original low-level actions to construct a single temporally-extended action. An option o is formally defined by the tuple (I_o, π_o, β_o), where $I_o \subseteq S$ is an initiation set that specifies all the states from which option o can be initiated, $\pi_o : S \times A \to [0, 1]$ is a policy which provides the probabilities of option o executing each action for each state, and $\beta_o : S \to [0, 1]$ is the termination condition, which gives the probability of the option terminating for each particular state. The learned set of options can then be augmented with the original primitive actions to form a *Semi-Markov Decision Process* (SMDP) [Sutton *et al.* 1999]. All transitions and rewards in an SMDP are dependent on the time taken for an option to execute, which can take $t > 1$ timesteps.

An SMDP is a tuple defined as $M = (S, O, P', R', \gamma)$, where whereby:

- S is the set of all possible states of the environment
- O is the set of options and primitive actions available to the agent
- $P'(s', t|s, a)$ is the transition probability function, which is now a function of time t
- $R'(s, a, s', t)$ is the reward function, which is also a function of time t
- $\gamma \in [0, 1]$ is the discount factor

Learning options is an active field of research: examples include using inverse reinforcement learning and reward segmentation to discover skills [Ranchod *et al.* 2015], constructing chains of skills through change point detection [Konidaris *et al.* 2010] and using policy gradient theorems to learn intra-option policies and termination functions [Bacon *et al.* 2017].

The majority of attention in the area of learning options in MDPs has focused on using the same state representation for every option. While this successfully achieves action abstraction, the skills are still specific to the domain in which they are learned. Learning more general option representations would make it more feasible to reuse these options in new domains that consist of different settings, such as colours and lighting, but contain similar sub-problems.

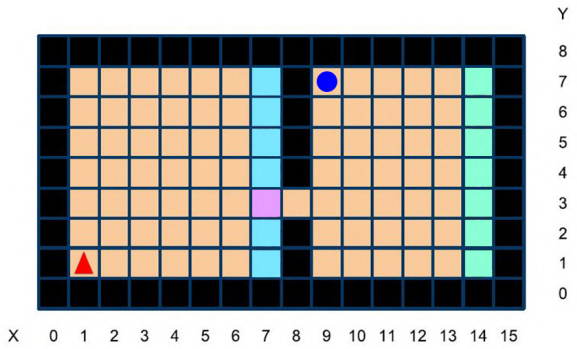
2.3 Factored Representations

The task decomposition and factored state representations, which partition the state space into distinct sub-states, are complementary concepts. Without a factored representation, the agent will be able to decompose the given task into appropriate sub-tasks, but the skills used to solve these sub-tasks will remain restricted to the domain in which they are learned. This is an issue when transferring these skills to a new domain, as it may not contain the same features as the original domain and determining which features need to be similar across the two domains for the skill to apply is not trivial.

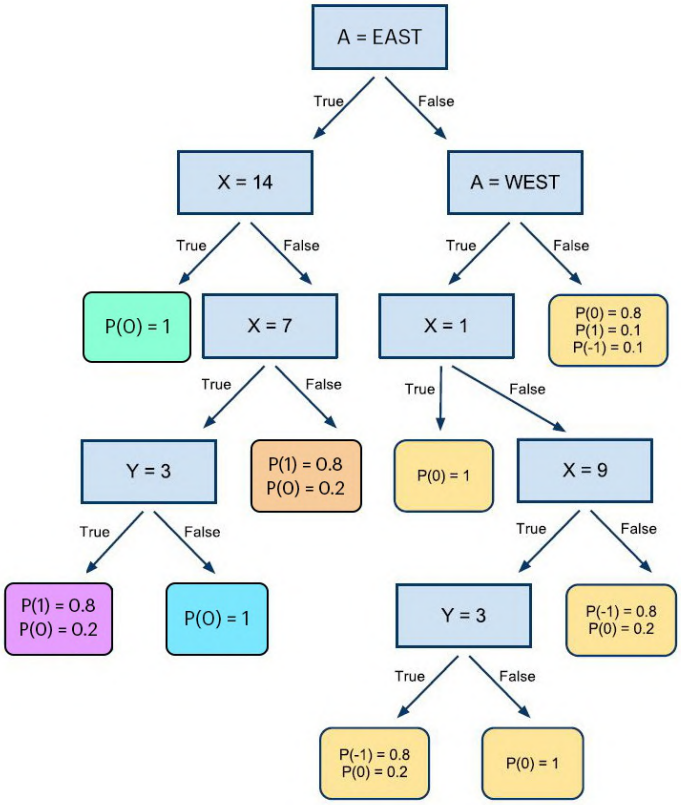
Referring back to the kitchen agent example, we may decompose the task of making eggs into various sub-tasks such as fetching the eggs, or placing them in the pan. Each one of these sub-tasks is focused on a specific region of the kitchen and particular items. Learning a factored representation for the sub-task to fetch the eggs for example would show that there need to be eggs in the environment and the agent would need to be within a certain range of those eggs, with an accessible path to them. Other state information such as the lighting or colours in the room, or the location of other objects can be considered irrelevant to that sub-task and do not need to be consistent with the original domain when reusing the skill to fetch eggs in a new domain. The two most common frameworks used for learning such factored state representations in hierarchical reinforcement learning settings are Object-Oriented MDPs and Factored MDPs.

2.3.1 Object-Oriented MDPs

Object-Oriented MDPs (OO-MDPs) decompose the agent's environment into a set of n objects, $O = \{o_1, ..., o_n\}$ [Diuk *et al.* 2008]. Each object belongs to an *object class*, which contains a set of attributes associated with it. The current state of an object, denoted $o.state$, is given by the set of assigned values for all of the attributes of the object. The MDP state s is then defined as the union of all object states at a given point in time, $s = \bigcup_{i=1,..,n} o_i.state$. For example, consider the gridworld domain shown in Figure 2.1a, with object classes *agent*, *ball* and *wall*. All classes contain position attributes x and y, the agent class also contains a *direction* attribute and the ball class contains a *colour* attribute and a *carried* attribute to denote whether it has been picked up by the agent.



(a) Gridworld domain consisting of two rooms connected by a hallway. Cells are coloured to match the corresponding leaves in the tree below. The red triangle indicates the agent's location and the blue circle indicates the location of a ball object.



(b) Decision tree model for predicting the change in the X state variable for the domain shown in Figure 2.1a

Figure 2.1: This figure shows the decision tree model that predicts how the X state feature will change for a specific two room stochastic gridworld environment. The tree is split based on actions and states, indicated by the blue rectangles. The coloured rounded rectangles are the leaves of the tree and display the probabilities of changes in X. For example $P(0) = 1$ predicts that X will not change with a probability of 1, while $P(1) = 0.8$ indicates a prediction that X will change with probability 0.8 [Hester and Stone 2013].

OO-MDPs also capture interactions between objects as *relationships*. Formally, a relationship $r : C_i \times C_j \rightarrow Boolean$ is a function defined at class level, whose value is dependent on two objects $o_1 \in C_i$ and $o_2 \in C_j$. Again looking at the gridworld domain in Figure 2.1a, the relationship $touching(o_1, o_2)$ defines whether object o_1 is one block away from object o_2. If relationship $touching(agent_i, ball_j)$ is true, then the effect of action *pickup* is defined as $ball_j.carried \leftarrow true$. If a new object of an existing object class is added to the environment, then it will inherit all of the attributes and relationships that belong to that class. However, if a new object class is added to the environment, then the attributes of that class and the relationships with other classes will need to be created manually.

The deictic object-oriented framework [Marom and Rosman 2018] is an extension of OO-MDPs, which explores learning more transferable transition dynamics. Traditionally the transition dynamics would be described in terms of first-order predicates, which indicate to which object the transition applies to. For example, when the agent is unlocking the yellow door, it specifically requires the yellow key object. If the number of key objects in the domain changes, then this will affect the number of preconditions that the transition dynamics depend on. Deictic OO-MDPs overcome this by learning deictic predicates, which are grounded with respect to a specific reference object. For example, the predicates required for unlocking the yellow door would be grounded with respect to the yellow key, rather than any key in the domain. This allows the transition dynamics to be effective when transferred to new tasks, even if the number of objects differs from the original training domain.

Another framework which looks to leverage the structure provided by OO-MDPs to transfer learned knowledge to new domains is Transfer Options (TOPs) [MacGlashan 2013]. TOPs are options which are learned in an OO-MDP with the specific goal of being transferable to multiple different target domains. Similar to regular options, TOPs consist of an initiation set and a termination condition, but also require a state space mapping between the source and target task. This mapping is then used to apply the entire source policy to the target domain. The limitation with the TOPs framework is that the state space mapping needs to be handcrafted by a domain expert, which can be a time-consuming process. To tackle this issue, Topin *et al.* [2015] present a framework that makes use of an unsupervised learning method to automatically generate mappings between domains. While this is effective in guiding the agent as to which policies are most appropriate to transfer to the target domain, the method still suffers from the limitations of the underlying OO-MDPs in that each domain's dynamics are required to be manually created by an expert.

2.3.2 Factored MDPs

Another commonly used framework when working with factored representations is the factored MDP (FMDP) framework [Boutilier *et al.* 2000], which allows for reward and transition functions to be represented compactly. FMDPs are MDPs where state variables are represented by a finite set of random variables and the state space of the FMDP is defined by the Cartesian product of the domains of these random variables, $\mathbf{S} = \{S_1, ..., S_n\}$. Thus, a single state in an FMDP is represented as a vector containing values assigned to each of the state variables. Since the number of states increases exponentially as the number of variables in the FMDP increases, it is desirable to represent the transition function in a way so that the inter-variable dependencies can be best exploited, and that the adverse effects of exponential growth are reduced.

Assuming that the problem MDP is finite, then the transition function $P(s'|s,a)$, as described in Section 2.1, consists of a finite set of probabilities. If the state space can then be decomposed into a finite set of random variables as described above, then these transition probabilities can also be decomposed, resulting in a product of probabilities [Degris *et al.* 2006]. This allows the transition function to be represented more compactly, by exploiting the independencies between the random variables. Similarly the reward function can be represented as the average over all factored rewards.

Vigorito and Barto [2008] present a method to learn options within an FMDP, by using the Variable Influence Structure Analysis (VISA) algorithm [Jonsson and Barto 2006]. *VISA* models the FMDP as a Dynamic Bayesian Network (DBN), which allows it to identify the causal relationships between state variables. Options are then discovered within this FMDP using standard option discovery techniques and these options are used for exploration to improve on the state model, which is then factored again in a recursive process.

DBN's ability to represent dependencies between random variables makes them the data structure of choice when working with FMDPs, but they are also one of the main limiting factors. Most research within FMDPs assumes that the structure of the DBN is known, and while there have been attempts to autonomously learn this structure, it has been shown that learning the structure of a DBN from data is an NP-hard problem [Chickering *et al.* 2004]. As a result, similar to OO-MDPs, designing FMDPs is also a time-consuming process in which an expert is required to manually encode the variable relationships in the domain and adjust the network structure each time the domain changes.

Another limitation of FMDPs is that they make the assumption that the agent's environment and the reward function are always factored. This assumption is quite restrictive as it is not always possible to decompose the reward function into factored local rewards. Similarly, if the environment contains any state variables which have causal relationships and dependencies that don't allow them to be decomposed, then the problem cannot be represented as an FMDP.

2.4 Decision Tree Based Models

Another data structure which is capable of modeling transition and reward dynamics is a decision tree. While decision trees are used much less frequently in conjunction with FMDPs due to their inability to capture inter-variable dependencies, they are far more robust in adapting to changes in the domain. *TEXPLORE* [Hester and Stone 2013] is a model learning algorithm, which learns models of the domain using random forests of decision trees to model transition and reward dynamics. The algorithm is similar to the way in which DBNs are used for learning FMDPs, in that it learns separate predictions for each individual state feature, in conjunction with the reward.

Decision trees are used in *TEXPLORE* to approach model learning as a supervised learning problem, where the learner is predicting the next state s' and reward r, from the input pair (s,a) comprising of the current state and action. Each node of the decision tree is split using the C4.5 algorithm [Quinlan 1986], which uses information gain to determine the optimal split.

Given a vector containing the n state features and action a, each tree makes a prediction for a specific feature or reward. The *TEXPLORE* algorithm works in an on-line manner, meaning that the tree is generated and refined while the agent is acting within the MDP. Beginning with an empty tree, the algorithm will continue to refine the tree based on new agent experiences. Eventually the tree will contain leaves for each individual state in which the transition dynamics vary from the global state dynamics.

Consider the simple gridworld domain shown in Figure 2.1a, consisting of two rooms connected by a hallway. The domain is stochastic, with an 80% probability of the agent moving in the desired direction of a given action, and a 20% probability that the agent remains in the same state. Figure 2.1b shows the decision tree model for predicting the change in the X state variable. The tree is split on both the agent's actions and individual state variables, as shown by the nodes in the blue rectangles. The gridworld is coloured to match the corresponding leaves in the tree when taking the action *EAST*. While the agent is acting within the domain, the tree concurrently gets built in an on-line manner. To observe how the tree divides the state space, consider taking action *EAST* in state *X=14*. Unlike the global state dynamics where the agent's X variable would increase by 1 when taking action *EAST*, in this case the agent would not transition to a new state as it is against a wall. For this reason, we say that the tree splits the states up into sections where the transition dynamics vary from the global state dynamics. Once the tree is complete we can see that all of the leaves give probabilistic predictions of how the X variable will change based on the outcomes that the agent experienced when exploring the domain.

The use of decision trees provides a sample efficient way of modelling the MDP, but can often be incorrect for infrequently visited state-actions. To provide a more accurate prediction *TEXPLORE* uses the random forest model [Breiman 2001], which consists of a collection of decision trees. Each decision tree in the random forest is only trained on a subset of the agent's experiences within the MDP. Given a state-action pair (s, a), the random forest model makes a prediction by averaging all of the predictions from each of the trees. For example, if three trees predict *outcome 0* and *outcome 1* with equal probability of 0.5 and a fourth tree predicts *outcome 0* with a probability of 1, then the random forest model will predict *outcome 0* to occur with a probability of 0.625 and outcome 1 with a probability of 0.375.

While the focus of *TEXPLORE* is on learning transition dynamics to aid agent planning, the algorithm's ability to separate the state space into different regions in which the transition dynamics differ is what is most relevant to our research. These groupings allow us to find commonalities between tasks, which may allow for the transfer of skills, rather than spending time redundantly learning new skills for tasks we have seen before.

2.5 Transfer Learning

The ability to reuse rather than relearn skills is highly beneficial in improving the efficiency of a reinforcement learning agent. To achieve this, knowledge needs to be transferred from tasks that have been seen before to new, unseen tasks. Taylor and Stone [2009] provide an extensive overview of the concepts of transfer with respect to reinforcement learning tasks. Knowledge gained from previously solved tasks, known as source tasks, is used to improve the learning of a new unseen task, called the target task.

There are various transfer learning approaches, which transfer different types of knowledge, ranging from action-value functions, to policies, to full task models [Konidaris and Barto 2007; Pickett and Barto 2002; Lazaric 2012]. We are particularly interested in the transfer of options and their dynamics models between tasks. The most straightforward form of option transfer would be to transfer the source options directly to the target task, but this is often not beneficial as the source options are often overfitted to the source task and require strong state overlap for their policy to be applicable in the target domain [Taylor and Stone 2009]. One way to counteract this is to provide a state space mapping between source and target tasks as in the TOP framework [Topin *et al.* 2015] discussed in Section 2.3.1.

Another approach explores separating the state space into *agent-space* and *problem-space* [Konidaris and Barto 2007]. *Agent-space* is the set of features that remain consistent across tasks, such as observations by robot sensors, while *problem-space* defines the set of states, transition probabilities and reward function based on the current environment that the agent is in. Options learned in both agent-space and problem-space can be used to significantly increase the total reward and initial performance of the target task. It is suggested that agent-space options transfer better in cases where the source task and target task differ more, while problem-space options are useful when the source task and target task are more similar [Konidaris and Barto 2007].

Given these various transfer learning approaches it is important to be able to distinguish how effective they are. When considering the entire process of transfer learning one must decide whether or not to incorporate the time spent learning the source task when evaluating the efficiency of the approach. It is suggested that if the goal of transfer is to reuse knowledge in novel tasks, then it is reasonable to only account for time spent learning in the target task [Taylor and Stone 2009]. Given this, Taylor and Stone [2009] present five metrics which can be used to evaluate the performance gain of transfer learning approaches, in relation to learning without transfer:

- *Jumpstart:* How much does the initial performance of the agent in the target task improve, given transferred knowledge from a source task?
- *Asymptotic Performance:* How much has the final performance of the agent in the target task improved, given transferred knowledge from a source task?
- *Total Reward:* How much has the total reward accumulated by the agent increased? This can be calculated by taking the area under the learning curve as seen in Figure 2.2
- *Transfer Ratio:* What is the ratio of the total reward accumulated by the agent with transfer and without transfer? *Transfer ratio* can be considered as the ratio of areas defined by learning curves with and without transfer. It can be calculated by the equation:

$$r = \frac{\text{area under curve with transfer} - \text{area under curve without transfer}}{\text{area under curve without transfer}} \tag{2.3}$$

- *Time to Threshold:* How much faster does the agent reach a pre-specified performance level when learning the target task, given transferred knowledge from a source task?

Figure 2.2 provides a visual representation of some of these transfer metrics.

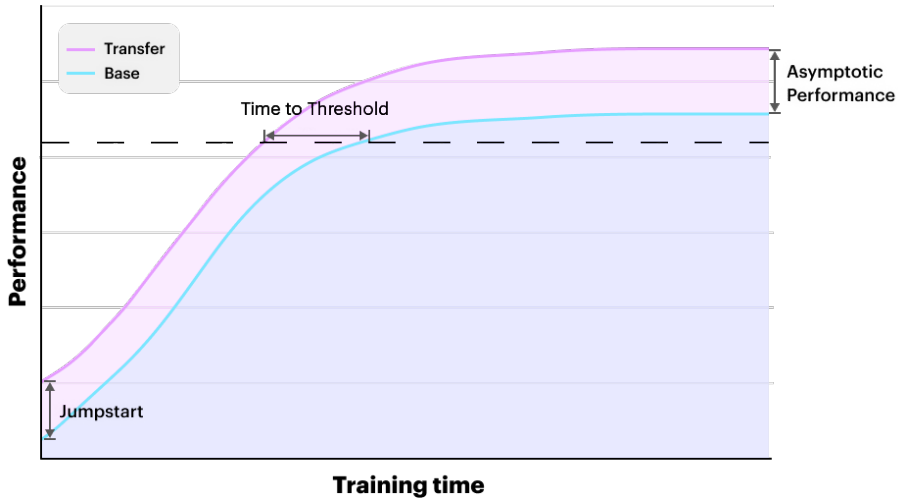


Figure 2.2: Graph depicting the different metrics for evaluating transfer. Jumpstart, asymptotic performance and time to threshold are indicated on the graph, while total reward can be calculated as the area under the learning curves [Taylor and Stone 2009].

2.6 Related Work

While the Transfer Options framework [Topin *et al.* 2015], discussed in Section 2.3.1, is the most comparable work to our framework, the field of learning specialised representations for hierarchical learning remains an active research area. One such example is the *TeXDYNA* method which looks at option discovery within FMDPs [Kozlova *et al.* 2010]. *TeXDYNA* breaks down a single environment with the FMDP structure and then solves it using hierarchical reinforcement, by using options that are learned within that factored environment. This is inherently different from our work in that we wish to learn factored representations of the options themselves, with the aim of transferring them to new domains to reuse. *TeXDYNA* would not be practical to use for our approach as it would require learning the factored state representation of each new environment, rather than learning a one-off representation which could apply across multiple environments.

The object-oriented dynamics predictor framework [Zhu *et al.* 2018], presents a method for autonomously predicting OO-MDP structures for a given environment. The predictor uses a neural network to decompose pixel-based states into objects and learn the object-level dynamics and relationships, although it does not focus on any hierarchical reinforcement learning applications of the learned OO-MDPs. Extending this framework to operate in skill based settings is a potential avenue for future work, as it would provide an interesting comparison with the framework that we present in this study.

2.7 Summary

In this chapter we introduced the background work upon which this research is based. This includes presenting an overview of reinforcement learning, hierarchical reinforcement learning and factored representations. We then discuss previous works that have successfully combined hierarchical reinforcement learning and factored representations, and elaborate on why these two concepts can be integrated to accelerate learning, particularly through transfer.

Despite the successes of the research described here, these works still rely heavily on knowledge of the structure of the environments to be known beforehand and on input from the researcher to handcraft the domains and experiments [Topin *et al.* 2015; Vigorito and Barto 2008]. In the following chapter we outline the methodology of our research, which provides a framework which can autonomously learn factored representations for skill based reinforcement learning.

Chapter 3

A Framework For Learning Factored Skills

In the previous chapter we outlined the background knowledge required for implementing this research. This included an introduction to reinforcement learning, followed by detailing the benefits of task decomposition, factored state representations and transfer learning. We wish to unify these concepts by learning generalised representations of skills that can be reused across an array of different tasks. In this chapter, we detail our framework for autonomously learning such representations and elaborate on how these factored skills can be used to accelerate learning of new tasks in Section 3.1, including a worked example of the framework methodology in Section 3.1.1. We then explain how the initial factored representations can be improved upon given additional training tasks in Section 3.2 and then present our method for grouping similar variables in Section 3.3.

3.1 Learning Factored Options

Given an initial training domain consisting of a set of tasks, and an agent with a set of options for solving these tasks, our framework learns representations for each of these options that only contains the variables relevant to solving each particular task. These factored option representations can then immediately be used in a new domain containing similar tasks to accelerate learning, without needing to adjust them to any variations in the new domain.

To explain our methodology, we decompose our framework down into four sections, shown in Figure 3.1. The sections can be described at a high level as follows:

1. Exploring the training domain with the agent to generate a set of trajectories, consisting of the states, actions and next states that the agent observes
2. Updating and formatting our set of trajectories, so that they are able to be used as input when training our decision tree models
3. Using the trajectories to learn individual decision tree models for each state variable
4. Learning factored representations from these models and applying these representations to our options to create *factored options*

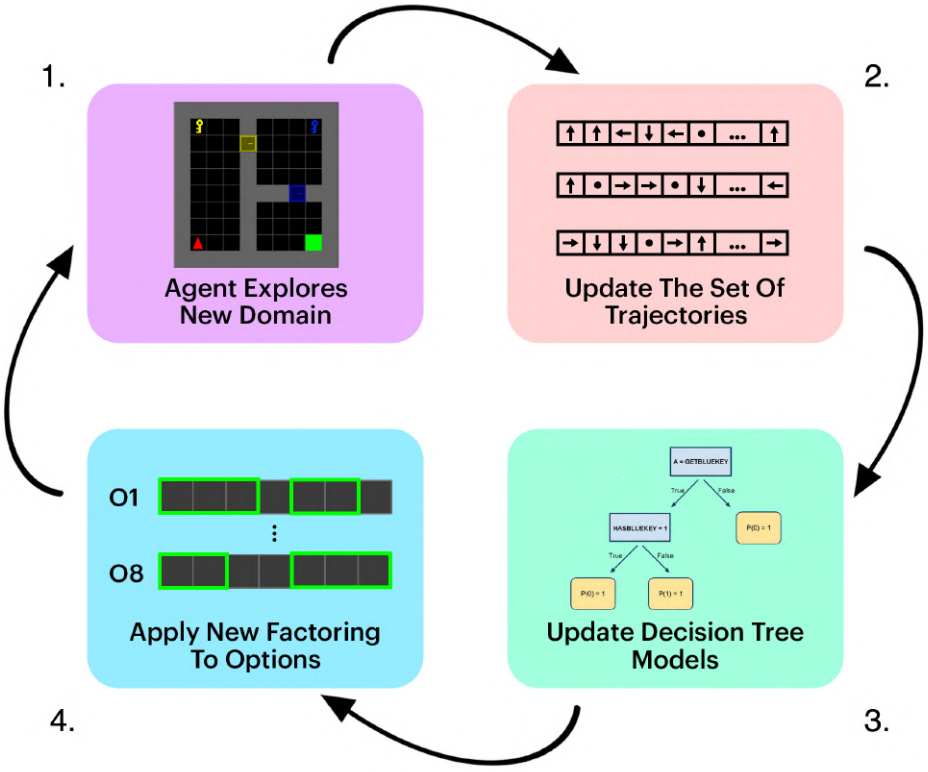


Figure 3.1: Diagram illustrating the framework's factor learning process. First, the agent explores the domain, attempting to solve the initial task. This generates a set of trajectories which are used as input for the decision tree models of the state variables. From these models factored representations are learned for each option available to the agent. These factored options are then made available to the agent, and it is given a new task to solve. This process continues and the factored representations are refined after each cycle.

Since our research focuses on learning factored option representations, we begin by making the assumption that a set of options for the initial tasks has already been learned. As our domains are deterministic, we learn the optimal initial options by generating a graph of the environment using *NetworkX* [Hagberg *et al.* 2008] and using the built in *single target shortest path* algorithm to obtain a policy for all reachable states that lead to the relevant terminal locations (key or door locations). While this method is used as the shortest path algorithm is efficient, it is perfectly reasonable to make use of one of the many other option discovery techniques [Bacon *et al.* 2017; Ranchod *et al.* 2015; Konidaris *et al.* 2010] to learn the initial options for our framework.

The agent uses SMDP Q-Learning [Sutton *et al.* 1999] to explore and attempt to solve the initial domain with these options. While the agent explores the domain, each state-action pair, along with the next state that the agent reaches, is logged to generate a set of trajectories. We want to use these trajectories to determine which state variables each option affects, and create a factored representation for the options which only uses those relevant variables. To do so we make use of decision tree models, which are learned for each state variable. Our input $\mathbf{x}$ for the decision trees is the set of states s with action a appended. Since we are interested

in the transition dynamics for each state variable, we calculate the change in each variable by looking at the difference between state s, and the next state s' after action a has been taken, and generating a binary indicator for each variable. Each variable is either assigned a 1 if that variable has changed from state s to s', or a 0 if that variable has remained constant. The target $\mathbf{y}$ for a particular decision tree model will then be the set of all binary indicators for a single variable. Our framework takes these inputs and target values and learns decision tree models for each state variable, using the *Scikit-Learn DecisionTreeClassifier* [Pedregosa *et al.* 2011], which makes use of the CART algorithm [Breiman *et al.* 1984].

While we draw inspiration from *TEXPLORE* [Hester and Stone 2013] in learning decision tree models of the transition dynamics for each state variable, our decision trees differ in that the branches are split on states and options, rather than states and actions. Any primitive actions that are able to change a particular variable are already encapsulated by the option which comprises of those actions and thus are redundant when learning the factored options' models. We only consider trajectories in which an option is executed and discard any trajectory which only contains primitive actions.

Once we have obtained the decision trees for each state variable, we wish to generate a factored option representation for each of the available options. Algorithm 1 presents the pseudocode for the process of learning these representations. To briefly describe this process, we loop through all of the decision trees and identify all of the trees in which the option appears (lines 1, 2). If an option appears in a tree traversal to a leaf node in which the state variable changed, then we know that the option is dependent on that state variable (lines 3, 4). The algorithm then returns a list of all the state variables that each option is dependent on (line 7), which we formally refer to as a *factored state set*.

Algorithm 1 Learn Option Factoring

```
1: procedure FACTOR_OPTIONS(state_var)
2:     for each decision tree d ∈ D do
3:         for each positive node n ∈ N do
4:             t ← traversal(n)              ▷ append traversed variables and options to set t
5:         for each option o ∈ O do
6:             if o ∈ t then
7:                 factored_set[o] ← state_var          ▷ append state variable to factored set
```

We provide a set of formal definitions to help ground the notion of *factored options* learned by our framework, as well as the *factored state space* over which they are learned and across which they are transferred.

Definition 1: A *factored state set* is a subset of the global state variable set, $\mathbf{S} = \{S_1, ..., S_n\}$, which consists of n random variables. Formally we denote the factored state set as $\phi \subset \mathbf{S}$.

Definition 2: A *factored state space* S^{ϕ}, contains the set of all possible combinations of the variables for a specific domain, given a factored state set ϕ.

To illustrate the difference between these two concepts, an example of a factored state set may be $\phi = \{\text{xPosition}, \text{yPosition}\}$, for which the factored state space would be $S^{\phi} = \{(0,0), (0,1), ..., (\text{n-1}, \text{n-1})\}$ for an agent acting in an $n \times n$ grid.

Definition 3: A *factored option* is defined as an option acting over a given factored state set ϕ. Therefore the initiation set and termination condition of a factored option are subsets of the factored state space S^{ϕ}. Formally this can be written as $FO = (I_o^*, \pi_o^*, \beta_o^*, \phi)$, where $I_o^* \subset S^{\phi}$ is the initiation set over the *factored state space* and β_o^* is the termination condition, defined as a probability distribution over S^{ϕ}.

Figure 3.2 shows how the factored state set can be used to apply a factored option to a new domain. The option requires the factored variables to be named the same across domains and to exhibit the same transition dynamics, for the option to be executable. For example, using the pickup action when next to a yellow key should always result in the *hasYellowKey* variable becoming true. This mapping between domains is invariant of both the location of the variables within each domain's global state variable set, and the total number of variables in the set.

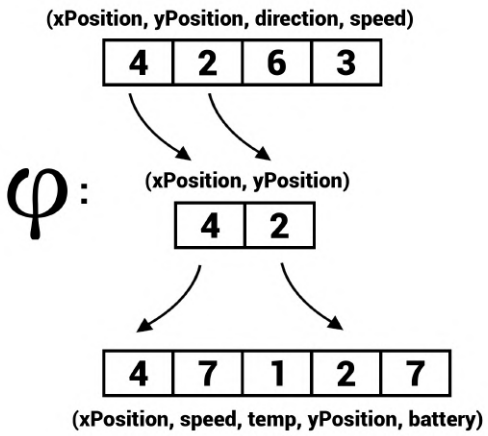


Figure 3.2: Example of how the factored state set is used to generate factored representations, which can then be mapped to new domains. In this example, *phi* refers to the set of variables *xPosition* and *yPosition*, on which the option depended. Assuming that the top state is an initiation set for the option, the factored option would be able to be initiated in the bottom state as the dependent variables match.

3.1.1 Minigrid Example

To further outline this process we provide a detailed breakdown of the methodology setup through the use of an example experiment. We make use of the Minigrid environment [Chevalier-Boisvert *et al.* 2018] for this worked example as that is the environment which we use for our initial experiments when evaluating our framework. In the Minigrid environment there are 5 actions available to the agent, namely turn left, turn right, move forward, the ability to pick up objects and the ability to use objects. In each domain the agent is posed with a set of tasks that are required to be solved sequentially.

To demonstrate this, consider the example domain shown in Figure 3.3, where the agent is represented by the red triangle and the goal by the green square. This domain consists of three rooms, connected by locked doors. To proceed to subsequent rooms the agent is required to unlock the doors by obtaining the key corresponding to the door colour. The sequence of tasks in this domain can be broken up as follows: obtain the yellow key, unlock the yellow door, obtain the blue key, unlock the blue door and proceed to the goal state. The sparse rewards in these domains make them a challenge for non-hierarchical methods. Formally, we define the reward function as 100 at the green goal state and -1 elsewhere.

We define the state vector for this domain as follows:

$(x, y, hasYellowKey, yellowDoorOpen, hasBlueKey, blueDoorOpen)$

where x and y describe the position of the agent, while the key and door variables are boolean indicators.

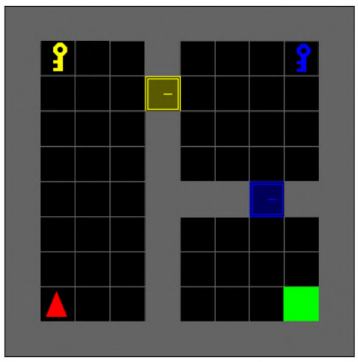
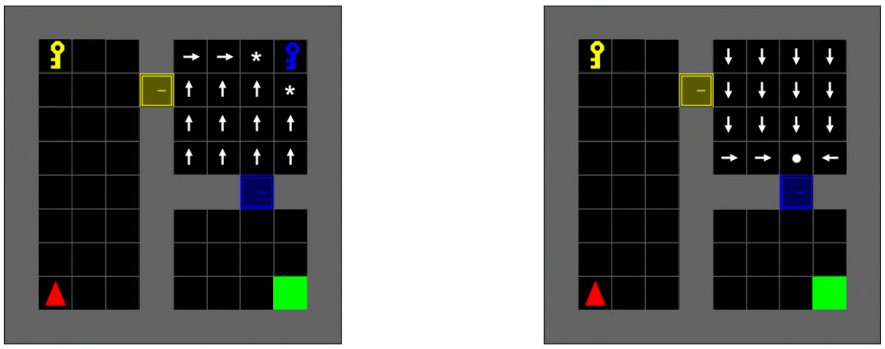


Figure 3.3: Example Minigrid domain, consisting of 3 rooms. Each room is connected by a locked door, which can only be unlocked by using the key corresponding to the door colour. The agent location is indicated by the red triangle and the goal state by the green square.

Continuing with our example domain, we begin with options that allow the agent to obtain each of the keys and options to unlock each of the doors. For example, the options to obtain the blue key and unlock the blue door are visualised in Figure 3.4. The *NetworkX* graph used for learning our initial options is built using data generated from an agent exploring the domain. Hence the initial options are dependent on the sequence of tasks that the agent is required to complete. This ensures that these options have the same conditions for their initiation sets and termination conditions as that of discovered options.

For example, consider the option that takes the agent to collect the blue key; when learning this option the yellow key will have already been collected and the yellow door will have been opened. As such, variables $hasYellowKey$ and $yellowDoorOpen$ are set to 1 in the initiation set of the option. Using the state vector described previously, we can formally define the initiation set as $(x^*, y^*, 1, 1, 0, 0)$ for the option to obtain the blue key (where x^* and y^* are any valid values of x and y, as shown in Figure 3.4). The termination condition is then given by $(x^*, y^*, 1, 1, 1, 0)$. The option policy can be seen in Figure 3.4a.



(a) Option policy to collect blue key (b) Option policy to unlock blue door

Figure 3.4: Visual representation of the option policies that take the agent to collect the blue key and to unlock the blue door respectively. Movement actions are indicated by arrows, the pickup action by stars and the use action by a dot.

We then learn the decision tree models of each of our state variables. For example, the decision tree for the *hasBlueKey* variable is shown in Figure 3.5. We then learn the factored state set for each option by following the procedure described in Algorithm 1. In this example the decision trees for the *hasYellowKey* and *yellowDoorOpen* do not contain the option to open the blue door, and as such these variables are excluded from the factored state set, even though they were required to be true in the original handcrafted option. The factored set for the option to open the blue door is (*hasBlueKey*, *blueDoorOpen*). The new initiation set can then be thought of as $(0, 0)$ for these two variables, and any arbitrary variable values for the rest of the state variables. This is useful as we can now transfer this option to a new domain containing a blue key and reuse it without having to collect the yellow key first. In fact, the new domain is not required to contain a yellow key at all. So long as the domain contains *hasBlueKey* and *blueDoorOpen* variables, this option will be able to be executed.

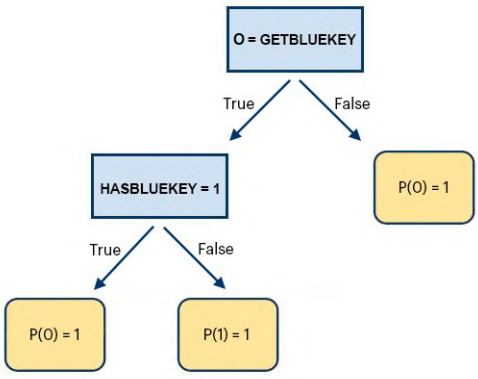


Figure 3.5: Decision tree for the *hasBlueKey* state variable. The tree is split on options and state variables, shown by the blue rectangles. The leaves provide probabilistic predictions on how the hasBlueKey variable will change.

3.2 Refining Representations

If we consider the framework as described in Section 3.1, the factoring that is learned is heavily dependent on the initial task that the agent is presented with. This is not ideal as it is highly likely that the decision tree models that have been learned have been overfitted to this training domain. This may result in the framework excluding certain state variables for which the domain provides limited or no data on or considering certain variables as dependent due to the structure of the domain.

TEXPLORE [Hester and Stone 2013] attempts to address overfitting by learning multiple decision tree models for a single domain, and averaging them using a random forest model. This approach results in better models for the given domain, but these models are still reliant on the information provided by that single domain. We show that given more training domains which consist of similar tasks, our framework is able to learn more general models that do not overfit to a single domain.

We implement this by providing the agent with a set of domains, consisting of similar tasks, for the agent to solve sequentially. Referring back to our Minigrid example, the overall structure of the domain would remain the same (3 rooms with keys and doors), but the tasks may differ by altering key and door colours and locations for example. After the framework has learned a set of factored option representations from the first domain, these options are then provided to the agent to use in exploring the second domain to generate a new set of trajectories. After generating trajectories for a new domain, they are then concatenated with the trajectories generated from all previous domains, and used to learn new decision tree models and factored representations. This cyclic process is repeated, with the agent continually refining its representations after exploring each new training domain.

Since the set of trajectories now span across multiple domains, the models learned from these trajectories are better able to generalise than with trajectories from a single domain alone. Referring back to Figure 3.1, we see that the four steps that our framework follows can be repeated in a closed-loop cycle, which continually refines the learned representations.

3.3 Variable Grouping

Consider an agent with a camera that provides RGB data. Based on the current framework, each sensor reading is considered independent, so it would be possible for example that the green or blue variables are considered irrelevant to a particular option and are excluded from the option factoring. It does not make sense in this case to deal with these camera inputs independently as they are not independent and identically distributed random variables. Rather, it would be more sensible if these variables could be grouped and treated as a single sensor input.

Pierce and Kuipers [1997] suggest that robotic sensors that are physically close in proximity tend to *behave similarly*. Similarly behaving sensors are defined as sensors which have a similar value at each timestep and a similar frequency distribution. For example, a robot in a real-world setting, may have sensors related to the robot itself, such as its position and direction, vision-based sensors which provide information about the orientation of objects around the robot, and

environment-based sensors such as temperature or wind speed. Figure 3.3 shows a toy example illustrating how these groupings could be applied.

Due to the fact that these grouped sensors exhibit similar behaviour and record similar variables, we make the assumption that if an option is dependent on the majority of a sensor groups variables, then it is probably dependent on all of that group's variables. Algorithm 2 shows the pseudocode for applying this grouping to our factored options. For each set of sensor groupings, we count the number of variables from that group that are in the option's factored set (lines 3, 4). If the count is greater than the group threshold, then we add all of that group's variables to the option's factored set (lines 5, 6).

Algorithm 2 Group Factoring

```
1: procedure GROUP_FACTORING(O, t)                          ▷ threshold t
2:     for each factored option o ∈ O do
3:         for each sensor grouping s do
4:             c ← group_count(s, o)                        ▷ count group variables in factored set
5:             if c ≥ t then
6:                 factored_set[o] ← s                      ▷ append entire group to factored set
```

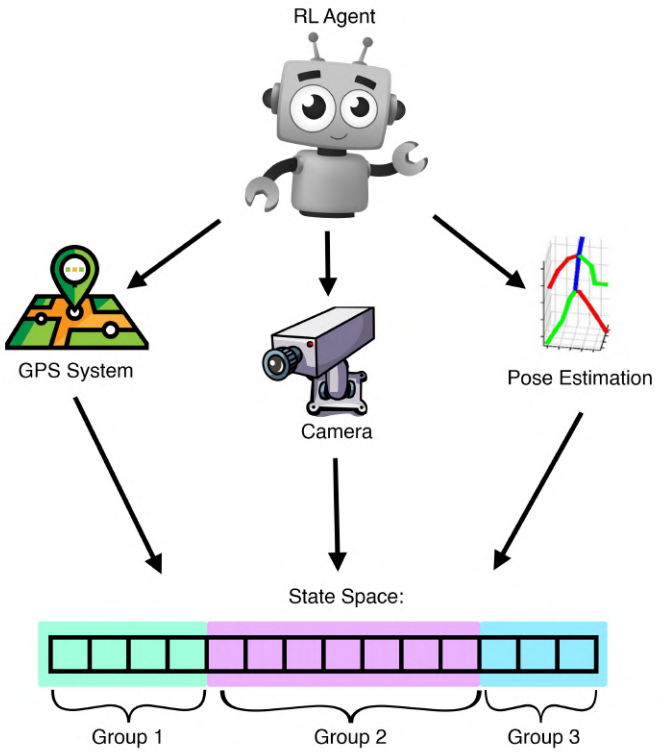
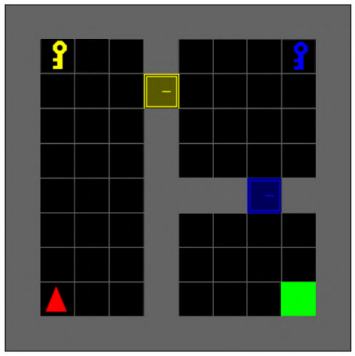


Figure 3.6: Diagram illustrating the application of variable groupings. The agent has information from three different systems: a GPS system providing the agent's location, a camera showing the agent's visual observation and a pose estimation system describing the joint angles of the agent. Observations from these three systems are concatenated together to form the entire state space. We aggregate all the individual readings or variables into three separate groups, one for each system.

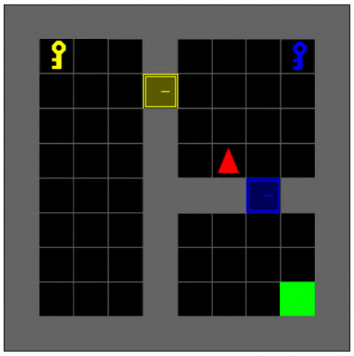
3.4 Evaluation of Transferability of Factored Options

Once we have learned the factored option models, we wish to evaluate their ability to improve learning in new tasks. To achieve this we need to transfer the options to our new domain. We transfer the entire factored option, including the option policy, initiation set, termination condition and factored state space. In this case transfer involves making these factored options available as part of the action space for the SMDP for the target domain. The agent will then have both Minigrid's primitive actions and the transferred set of options available to it when learning in the target domain.

To demonstrate the benefit of our factored options, consider the domain shown in Figure 3.7b, which is almost identical to the one in which we learnt the factored options, shown in Figure 3.7a. The only difference to the original domain is the starting position of the agent. Consider our initial handcrafted option to collect the blue key, which required the yellow key to be collected and the yellow door to be open. In this new domain the handcrafted option cannot be executed as it is impossible for the agent to obtain the yellow key. Conversely, our factored option is not dependent on the yellow door and yellow key and so would be executable in this new domain.



(a) Source domain in which the options were learned



(b) Target domain to which the options were transferred

Figure 3.7: Example of two domains across which transferring traditional options will fail, but *factored options* will work. The two domains are almost identical, except for the starting position of the agent.

While these two domains have been specifically tailored to display the benefits of factored options, we wish to discern the ability of these options to generalise across a multitude of different tasks. We generate new tasks by sampling domain features from a predefined distribution. The features that we sample for a new domain include agent start location, key location, key colour, door location, door colour, wall locations and goal location. Each of these variables are sampled from their own predefined distribution. For example, we may sample two key colours without replacement, from a set of four available colours, to be used in the domain.

This variable sampling results in different domains for similar tasks, for example two domains may have a task to collect the yellow key, but the agent start location, key location and colour of other keys in the domain are all different between the two domains. These are the cases in which we are interested in evaluating our factored options, although we are also interested in investigating the effect of additional variables, whether they be relevant or extraneous. Since the reward function depends on the goal location, which does not change across the two domains, the reward function also remains constant.

3.5 Summary

In this chapter we presented our framework for autonomously learning factored skill representations. In addition to the methodology of the framework, we also addressed how we combat various issues which appear when attempting to learn these representations. This includes the refinement process, which updates the representations given multiple tasks to prevent overfitting to a single task, and the use of variable groupings to prevent the exclusion of sensor readings for which there is limited or no data available during training.

In the following chapter we focus on demonstrating the performance of our framework. We present a set of baseline algorithms with which we compared our framework and detail how we evaluated the framework's performance. look at various settings where our framework excels against other methods and also highlight some of the limitations of the work. This includes investigating the effect of various parameters on the framework's functionality.

Chapter 4

Experimentation

In the previous chapter we outlined the methodology that we proposed for implementing our framework to obtain factored options. In this chapter, we present a set of experiments to evaluate the performance of our framework compared to various baseline algorithms. We begin by introducing the domains which we used for our experiments in Section 4.1 and elaborate on how we evaluate the performance of our factored options, including an explanation of the algorithms with which we compare our framework in Section 4.2. We then present a set of initial experiments in Section 4.3 to show the general performance of the framework, before expanding on to more detailed scenarios where the framework excels. This includes extending our framework to learn locally transferable options in Section 4.4 and evaluating the performance in domains with noise in Section 4.5. We conclude by demonstrating the framework's performance in a simulated real-world kitchen domain in Section 4.6.

4.1 Environments

For our initial experiments we make use of the Minigrid environment [Chevalier-Boisvert *et al.* 2018] that we referred to in Section 3.1.1 as a worked example for our methodology. Each domain consists of a 10×10 grid, although the outer cells are all walls, which means that the agent can only interact with an 8×8 space. The action space for all domains provides the agent with 5 actions – turn left, turn right, move forward, the ability to pick up objects and the ability to use objects. All domains consist of three rooms and comprise of the same five tasks (obtain key A, open door A, obtain key B, open door B and go to the goal location) in different configurations. The reward function is consistent across all of the domains, and is defined as 100 at the goal location and -1 elsewhere.

To generate domains consisting of these different configurations, we define a distribution of item locations, wall locations and item colours and generate each domain by randomly sampling from each of these three distributions. We repeat this process to generate a set of training domains and a set of test domains. Figure 4.1 shows an example of two different generated domains.

4.2 Baseline Comparisons

As discussed in Section 2.3.1, object-oriented algorithms are currently the favoured method for learning factored skill representations. Despite this, there is no common baseline algorithm available for us to compare our framework with. Instead, we make use of the OO-MDP learning algorithm DOORMAX [Diuk *et al.* 2008].

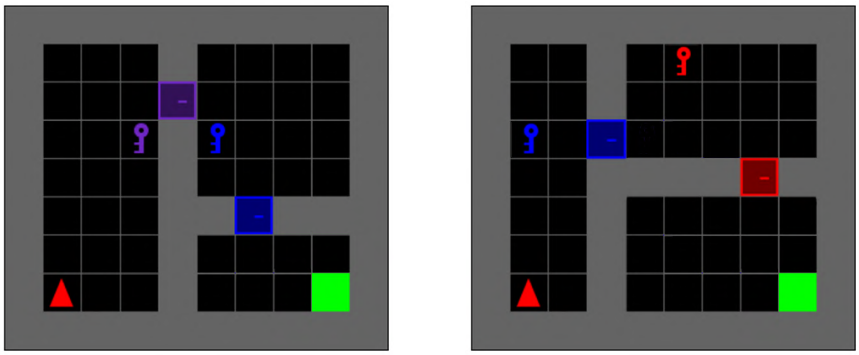


Figure 4.1: Example of two generated domains. The agent is represented by the red triangle and the goal location by the green square.

First we handcraft our OO-MDP object classes and attributes as shown in Table 4.1. We then define a set of five conditions between objects: $touching_N(o_1, o_2)$, $touching_E(o_1, o_2)$, $touching_S(o_1, o_2)$, $touching_W(o_1, o_2)$ and *direction*. The $touching_N(o_1, o_2)$ condition describes the situation where o_1 and o_2 share adjacent grid cells, where o_2 is to the *North* of o_1. For example, the proposition $touching_N(agent, wall) \wedge \text{direction} = \text{N}$ tells us that the agent is facing north and touching a wall to the north. The effect of taking the action *move forward* under this proposition is *no-change*.

Class Name:	Agent	Key	Door	Wall
Attributes:	xPosition yPosition	xPosition yPosition colour collected	xPosition yPosition colour opened	xPosition yPosition

Table 4.1: Table showing the object classes and attributes for the Minigrid OOMDP.

We then adapt the DOORMAX algorithm included in the *Efficient Reinforcement Learning* framework [Borea 2020] to solve these OO-MDP domains. As we are applying DOORMAX to the target domains, this can be seen as an upper bound of the transfer based OO-MDP algorithms mentioned in Section 2.3.1, such as TOPs, since Diuk *et al.* [2008] show that the DOORMAX algorithm is provably efficient.

Additionally, we evaluate our factored options against the following methods:

- Perfect options – we handcraft the optimal options for the target domain and solve the domain using SMDP Q-Learning [Sutton *et al.* 1999]. This can be viewed as an upper bound on the performance of the factored options, equivalent to the case where all of the learned options can be deployed everywhere in the target domain.
- No options – we solve the target domain using standard Q-Learning with primitive actions only. This can be viewed as a lower bound on the performance of the factored options, equivalent to the case where none of the learned options are applicable in the target domain.

- Unfactored options – we transfer the same options learned during training, but without applying any factoring to them.

When evaluating these algorithms we specifically focus on the *jumpstart*, *asymptotic performance* and *area under the curve* metric of the options, as discussed in Section 2.5. Jumpstart and asymptotic performance are useful metrics to assess the performance between algorithms that are able to solve the domain and do converge to the same reward. In the more complex domains though it is infeasible to run Q-Learning without options for enough episodes for it to solve the domain, and in these situations calculating the total reward from the area under the curve provides us with a way to evaluate these performances.

4.3 Initial Experiments

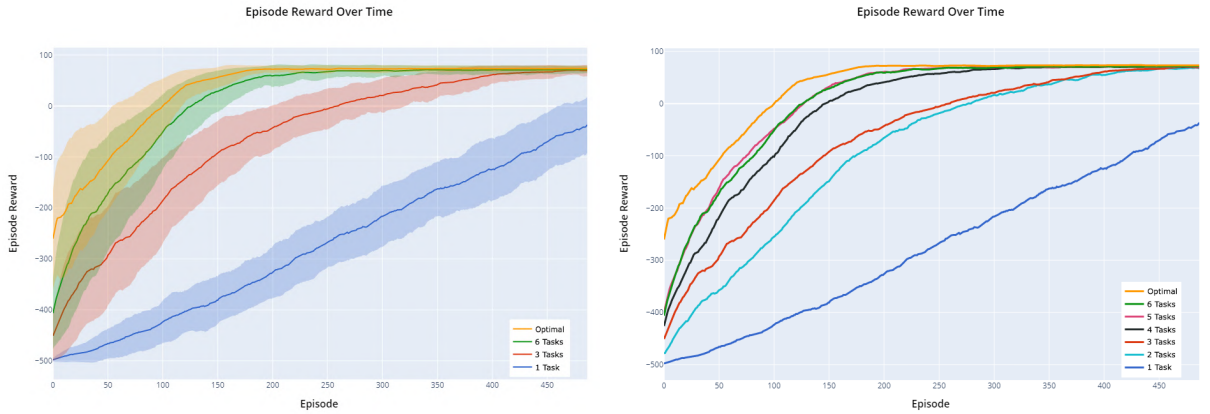
Our initial experiments test the validity of our framework. For these experiments we generate 50 training domains and 20 test domains. Each experiment involves randomly sampling six domains from the training set, which are given to the agent to solve sequentially. The agent begins with a set of four optimal options which can be used to solve the training domains. These options allow the agent to collect the first key, open the first door, collect the second key and unlock the second door.

For these experiments we focus on the performance of the *factored state set* ϕ that is learned by our method for each of the four tasks, rather than the performance of the options themselves. This is because the policy of the options in this current state is overfit to the specific task for which it was learned. For example, consider a domain where the yellow key in the first task is in the top right corner and we have an agent with the option to collect the yellow key. If we transfer this option as is to a new domain where the yellow key is in the top left corner, executing the option will still take the agent to the top right and this will not be beneficial for learning to solve the new task. In Section 4.4.1, we look at a solution to this by making use of a more agent-centric observation.

To evaluate the performance of ϕ, we apply the *factored state set* to the optimal options for the test domains, to obtain *factored options*. We then use these *factored options* to learn in the test domains. In the best case we expect the performance to be close to that of the optimal options for each task, once we have converged on the correct factoring.

The agent generates trajectories to learn ϕ by solving each of the training domains using SMDP Q-Learning [Sutton *et al.* 1999] with discount factor $\gamma = 0.9$, learning rate $\alpha = 0.15$ and ϵ-greedy action selection with $\epsilon = 0.05$. We train the agent for 500 episodes with a maximum episode length of 500 steps. For each entire run of our framework we sample a set of 5 test domains which we use to evaluate the performance. We use the same parameters for our SMDP Q-Learning agent when testing and training. After each new training domain that the agent sees, we evaluate the learned *factored state set* on all five test domains and obtain an average of the performance. Using multiple test domains helps us to assess the generalisability of our framework, and ensures that the performance is not greatly influenced by any single training domain, which may happen to be very similar to a testing domain in some cases given that they are both randomly generated from the same distributions.

Figure 4.2 shows the average episode reward for our factored options for various numbers of training tasks. We can see that even after only having seen a single task our framework performs better than training without the use of options, and by six tasks the performance converges close to that of the optimal option for the training domains, shown by the golden curve.



(a) Learning curves after 1, 3 and 6 training tasks, compared with the optimal option performance

(b) Learning curves for all number of tasks without confidence intervals for the sake of clarity

Figure 4.2: Average reward per episode for our learned factorings, for various numbers training domains compared to the optimal option performance. Graphs are plotted using a rolling average with a window size of 10 episodes.

Figure 4.4 shows a comparison between the decision tree for the *hasBlueKey* variable after one training domain, displayed in Figure 4.3, and after six training domains. Figure 4.4a demonstrates that after one domain the tree has overfit, considering variables such as the x-position of the agent (*xPosition*) and whether the yellow door has been opened (*yellowDoorOpen*) to be relevant to collecting the blue key. From Figure 4.3, we can see that this occurs due to the dynamics of the training domain. In this domain the agent will always be in an x-position ≤ 3.0 when collecting the blue key, and if the yellow door has been opened we know that the blue key will have been collected as well due to the sequential nature of the tasks.

The tree shown in Figure 4.4a can be interpreted as follows: the root node contains 2319 samples or trajectories, where 272 samples belong to class 0 and 2047 samples to class 1, as shown by the *value* field. This node splits the samples based on whether $xPos \leq 3.0$. The left child node is a leaf node, which contains 131 samples where this is true, and all of these samples result in a classification of class 0. The right child node will then have 2188 $(2319 - 131)$ samples, of which 272 are of class 1 and 1916 $(2047 - 131)$ are of class 0. This node now splits the tree on whether $hasBlueKey \leq 0.5$ and we repeat this process until we are only left with leaf nodes.

Figure 4.4b shows that after the agent has seen more examples it is able to learn a more general model of the state variable. Traversing up this tree from the positive orange leaf node, we can read this as: option $getBlueKey \leq 0.5$ must be *False* (we must execute option $getBlueKey$) and variable $hasBlueKey \leq 0.5$ must be *True* (we must not have already collected the blue key). From this it can be seen that once the model has converged, the only relevant variables to the task of collecting the blue key are the option *getBlueKey* and whether the key has already been collected or not, indicated by variable *hasBlueKey*. Appendix A shows the converged decision trees for other state variables.

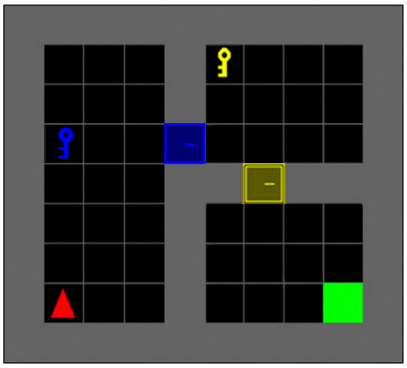
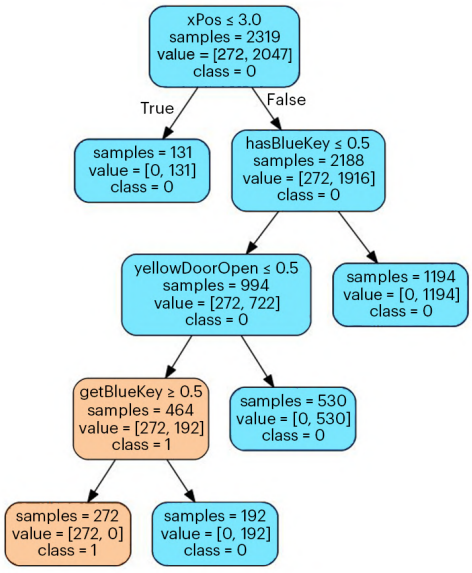
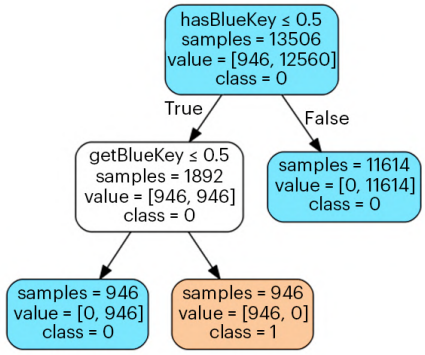


Figure 4.3: Initial training domain which resulted in the decision tree model shown in Figure 4.4a



(a) hasBlueKey decision tree model after one training domain (Figure 4.3)



(b) hasBlueKey decision tree model after six training domains

Figure 4.4: Example of two decision tree models learned during training. The first decision tree is learned after having only seen one training domain, while the second tree is the model learned at the end of training after having seen six training domains. The colour of each node indicates the majority class for that node (blue = class 0, orange = class 1 and white for a tie).

4.4 Locally Observable Domains

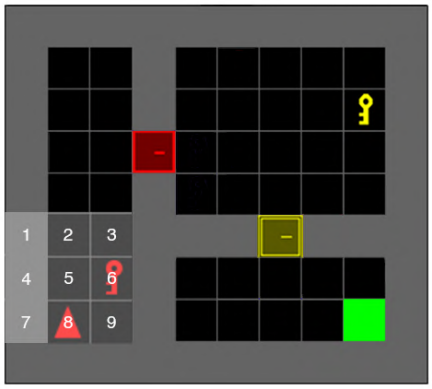
As mentioned in Section 4.3, directly transferring the options to new domains is generally ineffective as they are too restricted to the original domain in which they were learned. One method to tackle this is to decompose the problem into *agent-space* and *problem-space* [Konidaris and Barto 2007]. *Agent-space* contains the features that are consistent to the agent across all tasks, while the *problem-space* is defined by the environment in which the agent is currently acting. This allows the agent to learn more portable options which can be transferred between tasks that share the same agent-space. However, an issue with such agent-centric representations is that they are often non-Markov [Konidaris *et al.* 2012].

We have shown in Section 4.3 that the *factored state sets* that are learned by our framework are effective. We wish to draw inspiration from the *agent-space* approach, to apply these representations to a set of options that can easily be transferred across domains. To achieve this, we consider the case where the agent has a partially observable local view available. Figure 4.5a shows an example of this in a Minigrid domain, where the agent has a 3×3 vision grid. Each cell in this observation grid corresponds to a number depending on its contents. For example, an empty cell is a 0 and a cell containing a wall is a 10. We then flatten this 3×3 grid into a vector of size 9 and append it to the current state observation that the agent receives. We label the individual cell variables *v1, ..., v9*.

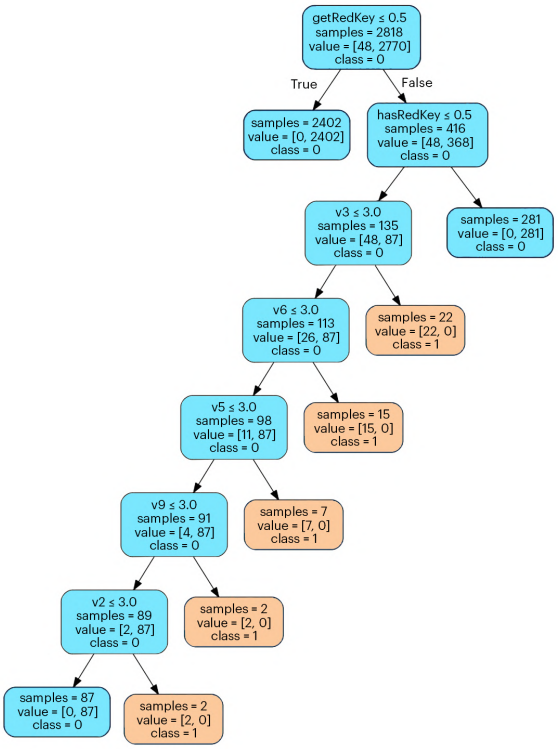
The agent's local observation now provides us with a more portable representation between domains. For our option to be executable in a new domain we now only require a set of 3×3 cells to match that of the original training domain. Using this new observation, we are no longer required to learn a new set of options for each new training domain. Instead we begin with a single set of options learned for the initial training domain. We then learn a factored representation for these options, apply this factoring, and then transfer these *factored options* to the next training domain where we repeat the process to refine our factored representation.

4.4.1 Variable Groupings

Given the partially observable domain seen in Figure 4.5a, the resulting decision tree model that is learned for the red key variable is shown in Figure 4.5b, which highlights a common issue that arises when dealing with such domains – not all vision cells are considered important. In this case the model only considers *v2, v3, v5, v6* and *v9* to be important to the red key, but excludes *v1, v4* and *v7* (the 8th cell is excluded as the agent always occupies this cell). This is due to the dynamics of a particular domain, as it is possible that the agent never sees certain items in specific vision cells. For example, due to the fact that the first room in the domain in Figure 4.5a is a 2-cell wide hallway, it is unlikely for the agent to see the red key in any of the left-hand side vision grid cells, unless it approaches the key from the opposite side of the room from which it starts.



(a) Training domain with a 3×3 local agent view, labelled 1 through 9.



(b) hasRedKey decision tree model learned from training domain shown in Figure 4.5a

Figure 4.5: Example of decision tree model which fails to capture certain vision variables due to the dynamics of the training domain.

If we now transferred this option to a new domain in which it was possible to see the red key in one of those vision cells, the option would not be able to execute in those cases. While the framework is still learning the correct factored models given the input data, we can see how the dynamics of the training domain can hurt the generalisability of these models. To counteract this we introduce the idea of grouping similar sensors [Pierce and Kuipers 1997], or in our case similar state variables, as discussed in Section 3.3.

We begin by handcrafting our state variable groupings, although there is room for future work to extend our framework to use autonomous grouping [Pierce and Kuipers 1997]. It is possible to derive groups from the structure of the environment, by estimating objects based on their state distributions and grouping similar objects. The agent also has information as to which sensors it contains and could simply group individual sensors. For these experiments though we use the following handcrafted groupings:

- agent variables (*direction, x-position, y-position*)
- object variables (*key collected, door opened*)
- vision variables (*vision cell 1, ..., vision cell n*)

Since the domains that we consider have two sets of key-door pairs, we have two object groups, one for each pair. We note that we group all vision cells together as in this case our vision variables can be considered equivalent to a single front-facing camera on the agent. There may be other cases, such as if the agent had one front-facing and one rear-facing camera, where it would be reasonable to split the vision variables into two groups or more.

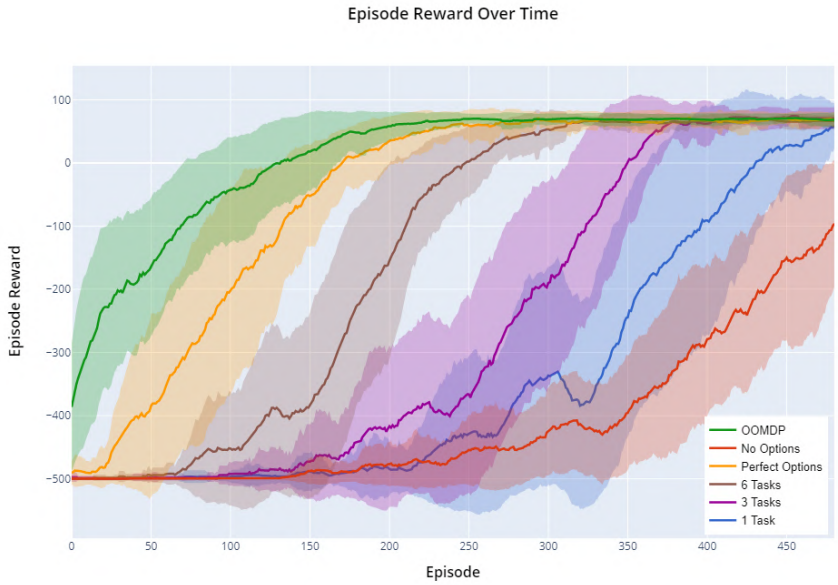
For these experiments, we consider two potential cases for the variable threshold, applying the grouping if the option depends on at least 50% of the group ($t = 0.5$), or applying the grouping if the option depends on at least one single variable from the group ($t = \frac{1}{\text{group size}}$). For example, if a vision sensor consists of 20 individual, concurrent readings or *variables*, then in the first case if 10 or more of these variables are considered important, then all readings from this sensor will be added to the *factored set*. In the second case, only one variable from this group is required to be considered important for the threshold to be reached and the entire sensor grouping to be added to the *factored set*. Again, here our threshold selections remain quite simplistic and there is room for developing a more elaborate formulation for selecting the threshold in future work.

Figures 4.6 and 4.7 show the performance of each of the two grouping methods separately, while Figure 4.8 provides a comparison between the two methods. The comparison figure shows the performance for both methods for 1, 3 and 6 tasks, with the single variable threshold indicated with solid lines and the half group threshold by the solid lines. From this comparison we observe that grouping with only one item in the group learns faster initially (shown by 1 task and 3 tasks performance), but converges to a lower performance than that of the method of grouping when the factoring contains at least half of the group (6 tasks performance).

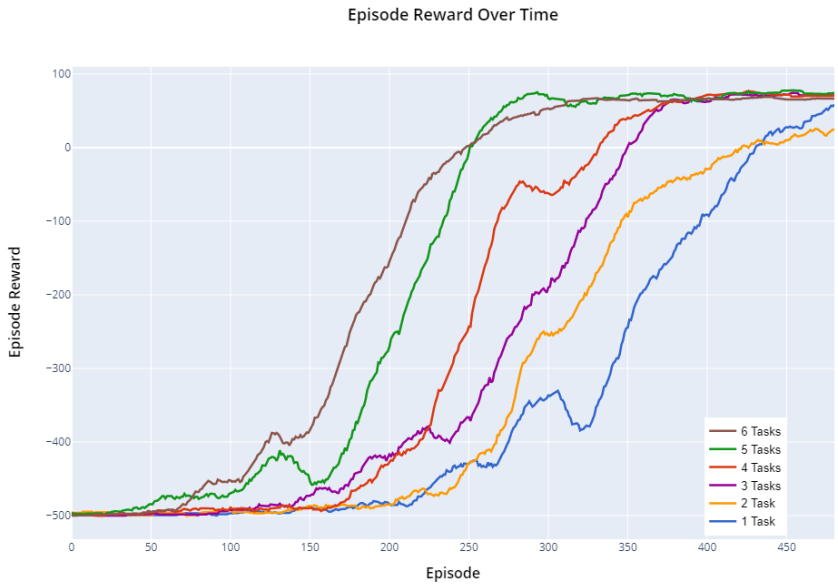
From the factoring obtained we see that this occurs because initially, occasionally more than half of the vision grid cells are often not considered important. In these cases the half group threshold struggles, but usually at least one cell is considered important and so the single variable threshold is able to cope early in the learning process. As the agent sees more training domains it becomes more likely that more than half of the vision state variables will be considered important and thus grouped to the factoring. On the other hand, the single item grouping is able to deal with these early training cases, but ends up marking many of the irrelevant variables as important. For example, if the *xPosition* is considered important, then it will group all of the other agent related variables as well, which means these *factored options* will struggle to generalise. We can see benefits from both methods, which once again reinforces the idea of developing a more complex method for selecting the threshold.

While running these experiments we found that the grid size of the agent's locally observable view played a large role in performance as well. Intuitively, the smaller the grid size, the more likely it is to generalise across new domains as it requires a smaller area to be similar between the two domains. On the other hand, the performance benefit received from the options decreases when dealing with a smaller grid size as the agent moves a shorter distance when executing each option. For example, with a 3×3 grid the agent can only execute the option to get the key if the key variable is visible in that grid, and so at most 3 grid cells away.

We test this intuition by comparing the performance of agents with grids of size 3×3, 5×5 and 7×7, and found that the 5×5 grid displays the greatest benefit in terms transfer performance. Thus for the experiments used in this chapter that make use of the visible grid, we utilised a grid size of 5×5. For the individual learning curves from these experiments, see Appendix B.

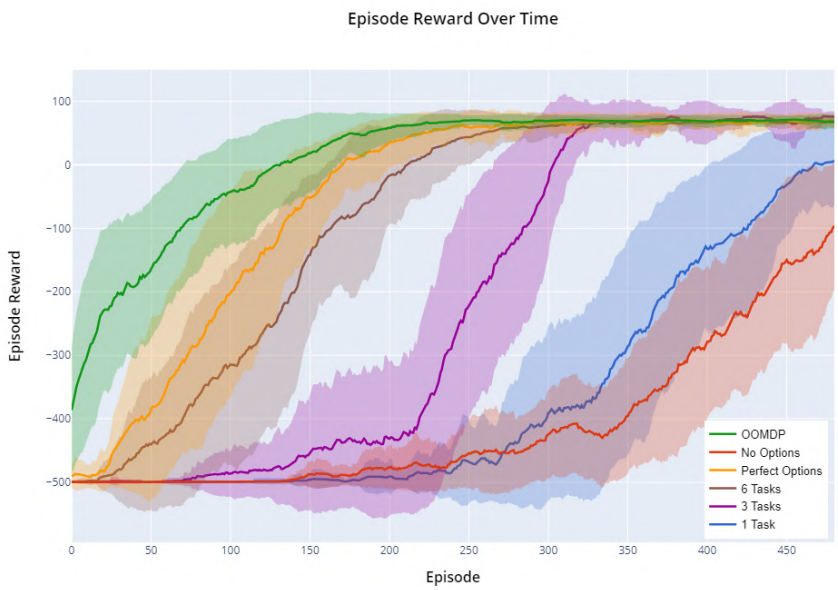


(a) Average reward per episode after 1, 3 and 6 training tasks given to our framework, compared with the optimal option performance. Shaded areas indicate a single standard deviation.

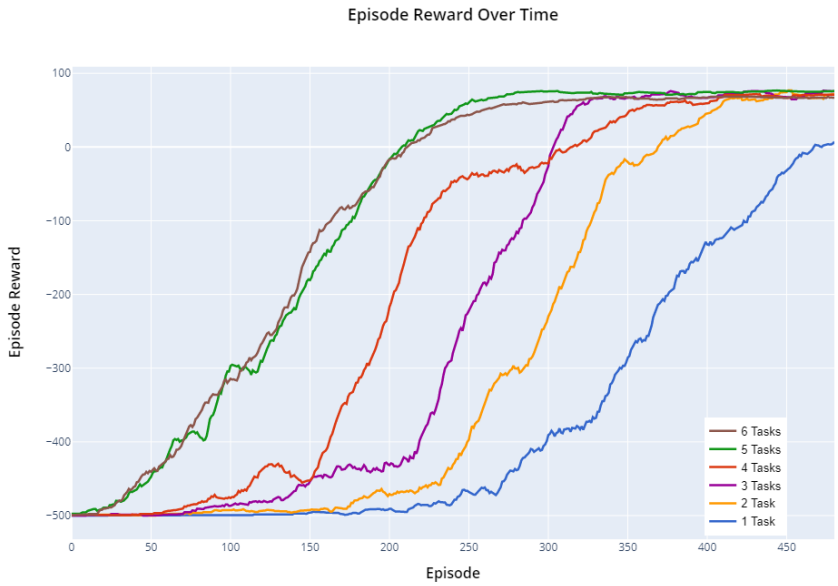


(b) Learning curves for all number of tasks without confidence intervals for the sake of clarity

Figure 4.6: Average reward per episode when applying a grouping threshold of 1 item in the group. Graphs are plotted using a rolling average with a window size of 10 episodes and are averaged over 20 random seeds.



(a) Average reward per episode after 1, 3 and 6 training tasks given to our framework, compared with the optimal option performance. Shaded areas indicate a single standard deviation.



(b) Learning curves for all number of tasks without confidence intervals for the sake of clarity

Figure 4.7: Average reward per episode when applying a grouping threshold of at least half of the items in the group. Graphs are plotted using a rolling average with a window size of 10 episodes and are averaged over 20 random seeds.

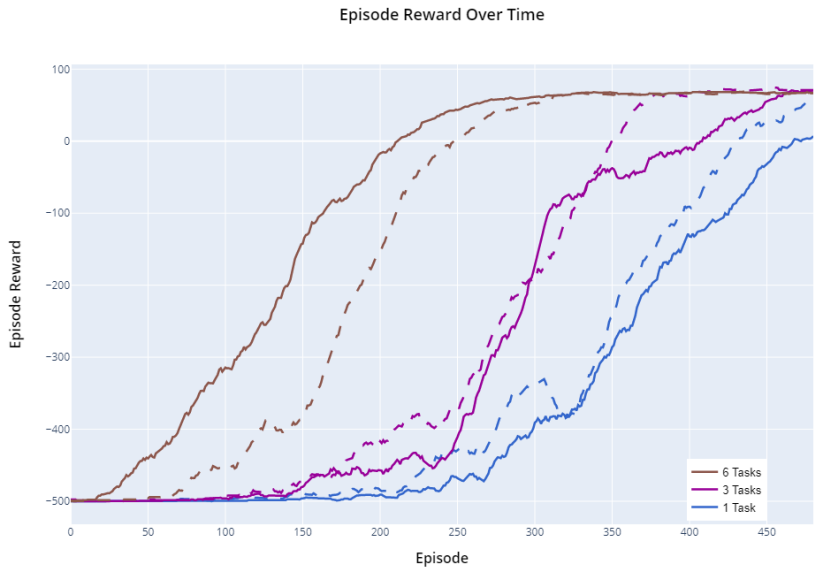


Figure 4.8: Comparison between 1, 3 and 6 training tasks for both grouping methods. The dotted lines show performance of the grouping threshold of 1 item, while the solid lines indicate the grouping threshold of at least half of the items.

4.5 Redundant and Noisy Variables

Next we explore two specific situations which are more akin to what an agent would encounter in real-world scenarios – domains containing multiple noisy or redundant variables. First, we test the case where the domain contains multiple redundant variables, but remains deterministic. If we consider agent's in real-world settings, there are typically a great number of objects present that are completely irrelevant to solving specific tasks. Thus if we wish to build a fully autonomous agent, it is important that they are still able to to transfer their learned skills to new environments where the configurations of these redundant variables or objects may differ. We showed in Section 4.4.1 that our framework can select the relevant variables from a small state space, but now we demonstrate that it is still able to do so, even when the complexity of the domain increases.

We set up our experiment in a similar fashion to the previous experiments with the local agent view, using a 5×5 vision grid. To generate redundant variables we append the distance between the agent and randomly selected wall cells to the state observation. We can consider this as equivalent to an agent that has information on the distance to various objects in the domain available, although most objects are not relevant to the current task. Given this new observation, we run our framework as described previously to learn a set of *factored options*. We then evaluate these options in a set of test domains, along with other baseline algorithms. We adjust the episode length of these experiments from 500 to 1000, so that the change in performance as additional variables are added can be clearly seen.

Figure 4.9 shows the total reward accumulated for each algorithm, for various numbers of redundant variables, calculated as the area under the curve for the each of the reward curves. We observe that the DOORMAX algorithm (labelled OOMDP) is able to scale well and outperforms any of the option based methods. Despite this, the DOORMAX approach becomes infeasible as the domain size grows, as it becomes increasingly more time consuming to manually add new objects and their attributes to the OO-OMDP. In contrast, our framework also displays the ability to scale with the addition of redundant variables, maintain performance as more redundant variables are added, but does not require any additional data as new variables are added. It is also observed that the unfactored options and regular Q-Learning without options struggle and their performance quickly diminishes as the complexity of the domain increases.

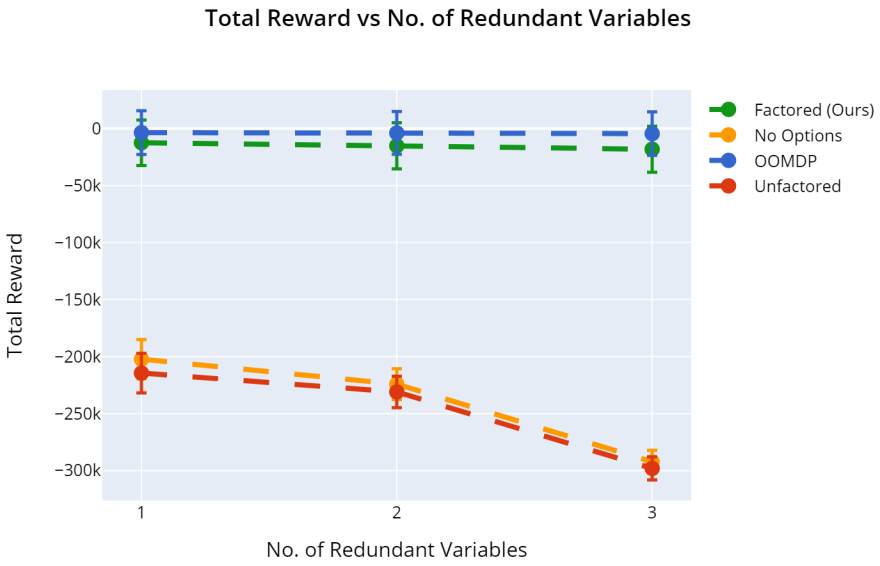


Figure 4.9: Comparison of total reward accumulated by different algorithms for domains containing various numbers of redundant variables. The bars on each point indicate one standard deviation, as each experiment is averaged over 10 random seeds.

Next we consider the case where the domain contains noisy, stochastic variables. We can consider this as equivalent to the case where an agent has a faulty sensor which provides it with noisy readings or if the sensor is capturing a particular variable which inherently contains noise. In real-world settings, this is a common occurrence and it is both unreasonable and often infeasible to expect a human to prune out these noisy readings and manually select the relevant variables to focus on for each experiment. Again we expect the ability to effectively deal with such scenarios to be a key characteristic of fully autonomous systems.

To generate these noisy variables we draw a sample from a normal distribution with mean $\mu = 10$ and standard deviation $\sigma = 10$, where the value is rounded to 2 decimal places. This reading is then appended to the agent's state. For each step that the agent takes, a new value is sampled and used to update the noisy variable in the state space. These variables are independent of the state, unlike the previous experiment in which state features were used to generate the additional variables.

Figure 4.10 shows the total reward accumulated for each algorithm, for various numbers of noisy variables. We can see that given a stochastic domain with noise, the DOORMAX algorithm struggles to solve the domain. This is known to be one of the main limiting factors of DOORMAX, as it requires the transition dynamics of each action and attribute to be representable as a full binary tree with effects at the leaf nodes and propositions at the non-leaf nodes. In non-deterministic domains it is not possible to represent the transition dynamics in this way and so the correct dynamics cannot be learned [Diuk *et al.* 2008].

Similarly, the unfactored options and standard Q-Learning perform poorly in this setup, particularly as more variables are added and the complexity of the domain increases. In contrast, our algorithm demonstrates that it is able to scale well in the case of stochastic domains with noisy variables. We see that even as the complexity of the domains increase, we are able to maintain performance by learning to ignore the noisy variables.

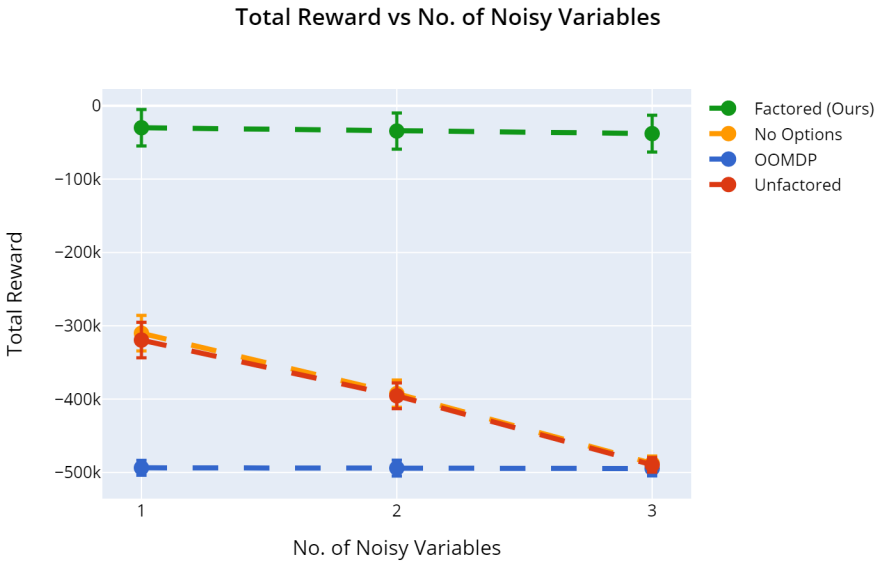


Figure 4.10: Comparison of total reward accumulated for domains containing various numbers of noisy variables. The bars on each point indicate one standard deviation, as each experiment is averaged over 10 random seeds

We provide the individual learning curves used to generate the total reward graphs seen in Figures 4.9 and 4.10 in Appendix C.

4.6 AI2-THOR

While experiments in the *Minigrid* domain have been useful as abstractions for testing separate components of our framework, we now show its effectiveness in a more realistic domain. *AI2-Thor* [Kolve *et al.* 2017] is a framework consisting of 120 interactive room-based environments in which an agent can act. These include kitchen, bedroom, bathroom and living room environments, which contain objects that can be manipulated and interacted with by an agent.

After each action taken in the environment, an *Event* is returned containing metadata pertaining to the agent, objects and the current scene that the agent is in. We extract specific metadata to create our state space that is used during these experiments. The state space is composed of the agent's position (x and y co-ordinates), the rotation of the agent and metadata for each object. The object metadata includes the object location, whether the object is visible to the agent or not, and whether the object is currently being held by the agent (if the object is marked as *pickupable*).

The action space available to the agent is as follows: *RotateRight* which rotates the agent 90° to the right, *RotateLeft* which rotates the agent 90° to the left, *MoveAhead* which moves the agent forward by a fixed amount defined by the *gridSize* parameter which we set to 0.25, *PickupObject* which will pick up the closest object within the agent's *visibilityDistance* that has the *pickupable* attribute and *PutObject* which will place the held object into a valid receptacle within the agent's *visibilityDistance*. In the case where there are no *pickupable* objects within the agent's visibility, or there are no valid receptacle to place a held object, executing either of these actions will result in no change in state. Although we have simplified our action space to be relevant to the specific tasks we give to our agent, the *AI2-Thor* framework provides several other actions which could be added in the future when dealing with more complex tasks, such as *Clean Object* or *Slice Object* used when cooking. There are also additional actions move the agent's camera angles, but for these experiments we fix the camera with $cameraHorizon = 0$.

For our experiments we work with the 30 kitchen environments, which we split into 20 training environments and 10 test environments. Figure 4.11 shows an example of one of the environments from the agent's view, as well as a top down view of the environment. These environments consist of a 16×16 grid that the agent can move across, although not all grid locations are reachable as some contain counters or other objects that the agent cannot move through. Each kitchen environment contains several objects with which the agent can interact. We remove certain objects such as cabinets, drawers and windows from the state space to simplify the domain slightly. Table 4.2 shows a list of all possible objects for a given environment, along with their attributes (whether they are visible, distance from agent, can be picked up or placeable). Note that this is not the same as the attributes of an OO-MDP class, as these variables are provided directly by the environment and we simply concatenate them together to form the state space. There is no prior knowledge provided by the environment and so the variables are not grouped into classes according to similar effects as with OO-MDP objects.

Object:	Apple	Bowl	Bread	CounterTop	Cup	Egg	Fork	Fridge
visible	✓	✓	✓	✓	✓	✓	✓	✓
isPickedUp	✓	✓	✓		✓	✓	✓	
distance	✓	✓	✓	✓	✓	✓	✓	✓
receptacle				✓				✓
Object:	Microwave	Mug	Pan	Plate	Pot	StoveBurner	Spoon	Toaster
visible	✓	✓	✓	✓	✓	✓	✓	✓
isPickedUp		✓	✓	✓	✓		✓	✓
distance	✓	✓	✓	✓	✓	✓	✓	✓
receptacle	✓					✓		

Table 4.2: Table showing the object classes and attributes for the AI2-Thor environments. The state space is defined as a combination of these attributes concatenated into a single vector. Note that not all environments will always contain all object and that the amount of each object present may differ across environments.



Figure 4.11: Top-down view of an example AI2-Thor environment, alongside the view provided to the agent by its camera. The agent's location can be seen in the top left corner of the top-down view, circled in red.

These environments provide a 4 times larger grid for the agent to move across, compared to that of *Minigrid*. The state space is also 12 times larger than *Minigrid*, due to all of the additional attributes that each object in the environment consists of. Despite these domains being technically large-scale gridworld domains, we believe that they are able to provide a more accurate representation of a real-world environment. This is because at a high level an agent in a real-world environment can treat it as a gridworld, just at a much finer granularity. If the agent moves and rotates at fixed increments, then this movement can be visualised as the agent moving over a very fine grid mapped over the environment. *AI2-Thor* provides similar functionality, whereby the agent can rotate by any specified number of degrees and move forward by a set distance. Although we do not make use of such fine actions for these experiments, they provide an interesting avenue to explore in future work.

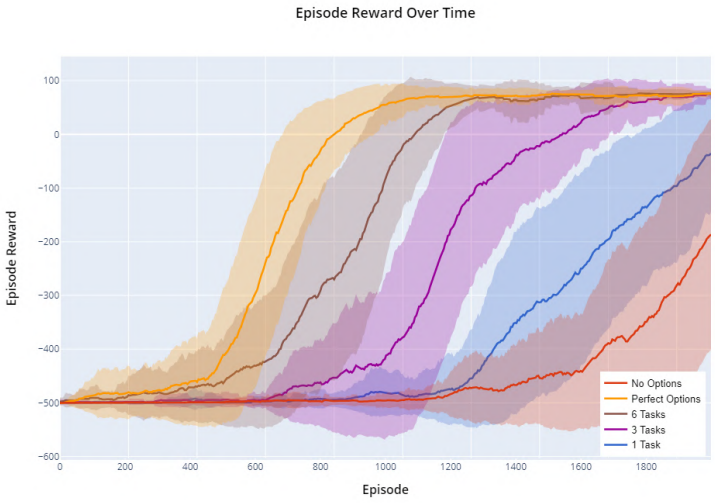
We provide the agent with two sequential sub-tasks that may be required when attempting to cook an egg:

1. Pick up the pan
2. Place the pan on the stove

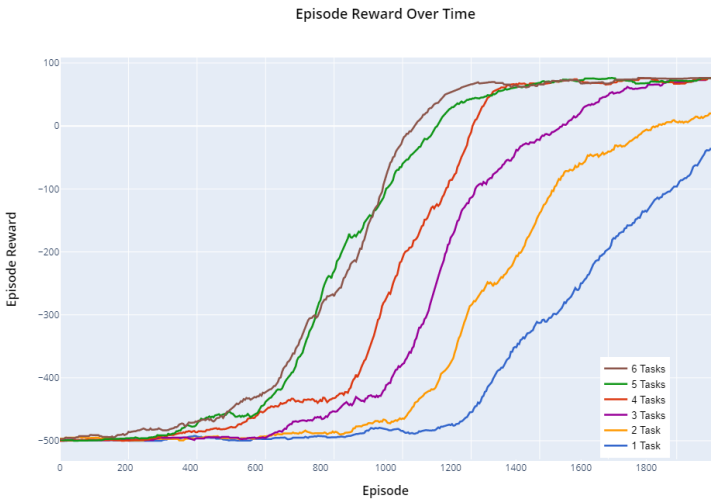
The reward function is defined as 100 once both sub-tasks have been completed, and -1 elsewhere. Again it is assumed that the agent has initial options available (*pickupPan, placePan*) for solving the first training environment. We then setup our experiment in a similar fashion to the Minigrid experiments. First we randomly select a set of six training environments, which are given to the agent to solve. The agent generates trajectories when exploring these environments, and these trajectories are used to learn our *factored options*. We then evaluate the performance of these options for each of a number of seen training environments, by transferring the learned *factored options* to three randomly selected test environments and averaging the performance. Since the start location of the agent plays a large role in these environments as to how quickly it can get to the pan, we randomise the start location for each run and average over 10 runs for each test environment.

To evaluate the performance of the learned *factored options* we make use of SMDP Q-Learning [Sutton *et al.* 1999] with discount factor $\gamma = 0.9$, learning rate $\alpha = 0.1$ and ϵ-greedy action selection with $\epsilon = 0.1$. We train the agent for 2000 episodes with a maximum episode length of 500 steps. Figure 4.12 shows the average episode reward for our factored options for various numbers of training tasks, compared to the optimal option performance and learning without options. We can see that the performance increases as the agent is given new training domains and converges close to the performance of the optimal options for the test results. Our factored options are still limited by the initial environment in which they were learned, which makes it difficult for them to reach the optimal performance. For example, if in the first training environment the agent had a large open space in front of the stove area, then the *placePan* option learned from this environment will struggle to generalise to a new environment where there are obstacles in front of the stove, and the agent may get stuck on these obstacles. Nevertheless, our factored options still display a clear benefit in accelerating learning in new environments.

Due to the fact that the agent has a *visibilityDistance* parameter for the objects that it interacts with, the learned factored options are portable to new domains with different configurations. For example, the agent learns that for the *pickupPan* option, the pan must have variable $isPickedUp = False$, $visible = True$ and also depends on the pan's *distance* variable to ground its location in the current environment. We can see an example of this from the converged decision tree for the *pan:isPickedUp* variable, shown in Figure 4.14



(a) Average reward per episode after 1, 3 and 6 training tasks given to our framework, compared with the optimal option performance. Shaded areas indicate a single standard deviation.



(b) Learning curves for all number of tasks without confidence intervals for the sake of clarity

Figure 4.12: Average reward per episode for various numbers of tasks in the *AI2-Thor* domain. Graphs are plotted using a rolling average with a window size of 20 episodes and averaged over 10 random seeds.

Figure 4.13 shows the top down view of two of the kitchen domains. Although the layout of these two domains is completely different, our *factored options* only require *isPickedUp, visible and distance* variables to match the initiation set of the original options. The shaded areas indicate the regions of the environment in which these variables that our framework has learned to be important to the particular options are all valid and hence the option can be executed in these regions. The blue area indicates where the *pickupPan* option is executable, while the pink area indicates where the *placePan* option is executable. With these experiments we have demonstrated that our framework is capable of learning factored skill representations which can transfer effectively to new environments and are no longer restricted to the initial environment in which they were learned. These transferred skills can then be used to accelerate learning of tasks in the new environment.



Figure 4.13: Top-down view of two different kitchen domains. The agent is highlighted by the red circle. The blue and pink areas indicate where our framework has learned that *pickupPan* and *placePan* options are executable respectively.

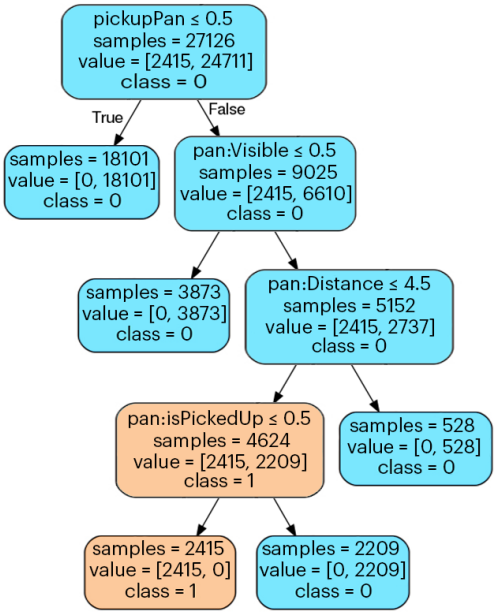


Figure 4.14: Converged decision tree for the *pan:isPickedUp* state variable.

4.7 Summary

In this chapter we presented various experiments which were used to evaluate the performance of our framework. We showed that our framework can autonomously learn *factored options* that successfully accelerate learning in new tasks, and match the performance of other benchmark algorithms which require domain knowledge and handcrafting, without requiring any of the prior expert knowledge that those algorithms do. We investigated a scenario which is more analogous to real-world environments, whereby the agent learns in domains containing noise. We demonstrated that in this situation our framework can considerably outperform other baseline algorithms. Finally, we showed that our framework can scale to larger domains which contain simulated real world environments.

Chapter 5

Conclusion

The use of task decomposition in conjunction with factored representations has been effective in learning skills that can generalise across multiple tasks. Until now one of the main limiting factors with this approach has been the laborious nature of composing structures which allow for learning such factored representations [Topin *et al.* 2015; Vigorito and Barto 2008]. In this work we proposed a framework for learning such factored skill representations in an autonomous manner.

We proposed *factored options*, which consist of a more generalised form of traditional options [Sutton *et al.* 1999], and allows for the learned options to transfer better to new domains. Our framework makes use of a closed-loop refinement system, which allows it to update the factored option representations that it learns after each new training domain presented to it. We showed that this method allows our framework to learn representations quickly, after only a few training domains, and that these representations can generalise across multiple test domains without overfitting to the domains on which they were trained.

To demonstrate the performance of our framework, we presented several different experiments. First we showed that in the Minigrid gridworld the factored options learned by our framework significantly outperform traditional options, when comparing the acceleration of learning new tasks provided. Next, we presented a method to learn more transferable factored options by grouping similar variables. We showed that by incorporating this methodology, along with a locally observable agent view, our learned factored options were able to display similar performance to that of other baseline methods, without requiring any of the prior expert knowledge that these methods need.

We also presented experiments which detail specific scenarios, which are more akin to what may be encountered in real-life robotics settings, in which our framework was able to outperform other baseline algorithms. First, we investigated the situation in which additional redundant variables were added to the agent's domain. Our framework displayed its ability to scale to such domains and maintained performance similar to that of the other baseline algorithms. Next we examined the case where noisy variables were added to the domain, and showed that again our framework was able to scale to such domains, while outperforming other baseline algorithms. Finally, we presented an experiment in the simulated real-world kitchen environment *AI2-Thor*. Here our framework showed promise in its ability to scale to larger, more realistic domains.

We have demonstrated that the *factored options* learned by our framework provide a clear benefit over traditional, unfactored options when evaluating their ability to transfer to new tasks. Additionally, we have shown that our framework displays similar performance to other methods which aim to learn transferable representations, without requiring the prior domain knowledge or time-consuming handcrafting of features that these methods do. We see this as a step towards building robust autonomous agents that can leverage the skills that they have learned to generalise, scale and adapt to new tasks and settings.

5.1 Future Work

This research also presents multiple avenues for future work. Some potential extensions that have been discussed previously include introducing autonomous variable groupings [Pierce and Kuipers 1997] and developing a more robust formulation to determine the threshold for applying those groupings.

While our framework is able to work with any option learning method, in this work we assume that the options are already available to the agent to use in the first training domain. Selecting a state of the art option learning framework, such as Option-Critic [Bacon *et al.* 2017], NPBRS [Ranchod *et al.* 2015] or FeUdal Networks [Vezhnevets *et al.* 2017], and integrating it into our framework to present a fully autonomous system is another interesting path worth exploring.

Finally, our experiments displayed that our framework is able to perform well in simulated real-world environments. An obvious extension here would be applying the framework to real robots and evaluating their performance in real-world environments to determine whether it can scale to such settings.

Bibliography

[Bacon *et al.* 2017] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[Bellman 1957] R Bellman. Dynamic programming: Princeton univ. press. *N J*, 95, 1957.

[Borea 2020] M Borea. *Efficient Reinforcement Learning.* `https://github.com/borea17/efficient_rl`, 2020.

[Boutilier *et al.* 2000] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial intelligence*, 121(1-2):49–107, 2000.

[Bradtke and Duff 1995] Steven J Bradtke and Michael O Duff. Reinforcement learning methods for continuous-time markov decision problems. In *Advances in neural information processing systems*, pages 393–400, 1995.

[Breiman *et al.* 1984] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.

[Breiman 2001] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Chevalier-Boisvert *et al.* 2018] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym.* `https://github.com/maximecb/gym-minigrid`, 2018.

[Chickering *et al.* 2004] Max Chickering, David Heckerman, and Chris Meek. Large-sample learning of bayesian networks is np-hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.

[Degris *et al.* 2006] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd international conference on Machine learning*, pages 257–264, 2006.

[Diuk *et al.* 2008] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247, 2008.

[Hagberg *et al.* 2008] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX.* Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[Hengst 2002] Bernhard Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 19, pages 243–250, 2002.

[Hester and Stone 2013] Todd Hester and Peter Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *Machine learning*, 90(3):385–429, 2013.

[Jonsson and Barto 2006] Anders Jonsson and Andrew Barto. Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, 7(Nov):2259–2301, 2006.

[Kolve *et al.* 2017] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017.

[Konidaris and Barto 2007] George Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[Konidaris *et al.* 2010] George Konidaris, Scott Kuindersma, Roderic Grupen, and Andrew G Barto. Constructing skill trees for reinforcement learning agents from demonstration trajectories. In *Advances in neural information processing systems*, pages 1162–1170, 2010.

[Konidaris *et al.* 2012] George Konidaris, Ilya Scheidwasser, and Andrew G Barto. Transfer in reinforcement learning via shared features. 2012.

[Kozlova *et al.* 2010] Olga Kozlova, Olivier Sigaud, and Christophe Meyer. Texdyna: hierarchical reinforcement learning in factored mdps. In *International Conference on Simulation of Adaptive Behavior*, pages 489–500. Springer, 2010.

[Lazaric 2012] Alessandro Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012.

[MacGlashan 2013] James MacGlashan. *Multi-source option-based policy transfer*. University of Maryland, Baltimore County, 2013.

[Marom and Rosman 2018] Ofir Marom and Benjamin Rosman. Zero-shot transfer with deictic object-oriented representation in reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[Pedregosa *et al.* 2011] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Pickett and Barto 2002] Marc Pickett and Andrew G Barto. Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. In *ICML*, volume 19, pages 506–513, 2002.

[Pierce and Kuipers 1997] David Pierce and Benjamin J Kuipers. Map learning with uninterpreted sensors and effectors. *Artificial Intelligence*, 92(1-2):169–227, 1997.

[Quinlan 1986] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Ranchod *et al.* 2015] P. Ranchod, B. Rosman, and G. Konidaris. Nonparametric bayesian reward segmentation for skill discovery using inverse reinforcement learning. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 471–477, Sep. 2015.

[Sutton and Barto 1998] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

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

[Taylor and Stone 2009] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[Topin *et al.* 2015] Nicholay Topin, Nicholas Haltmeyer, Shawn Squire, John Winder, James MacGlashan, et al. Portable option discovery for automated learning transfer in object-oriented markov decision processes. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[Vezhnevets *et al.* 2017] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549. PMLR, 2017.

[Vigorito and Barto 2008] Christopher M Vigorito and Andrew G Barto. Autonomous hierarchical skill acquisition in factored mdps. In *Yale Workshop on Adaptive and Learning Systems, New Haven, Connecticut*, volume 63, page 109, 2008.

[Watkins and Dayan 1992] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

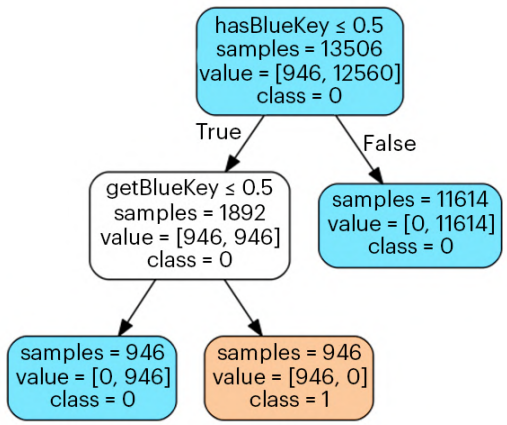
[Zhu *et al.* 2018] Guangxiang Zhu, Zhiao Huang, and Chongjie Zhang. Object-oriented dynamics predictor. *arXiv preprint arXiv:1806.07371*, 2018.

Appendices

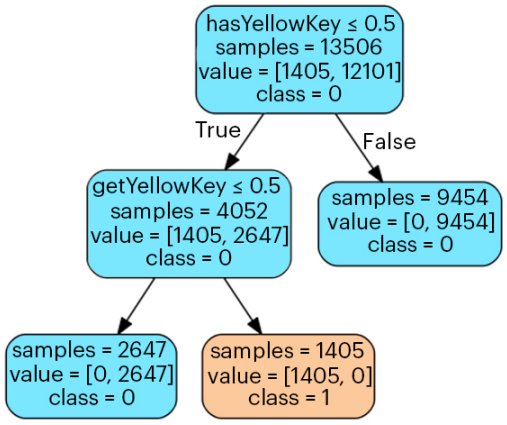
Appendix A

Decision Tree Models

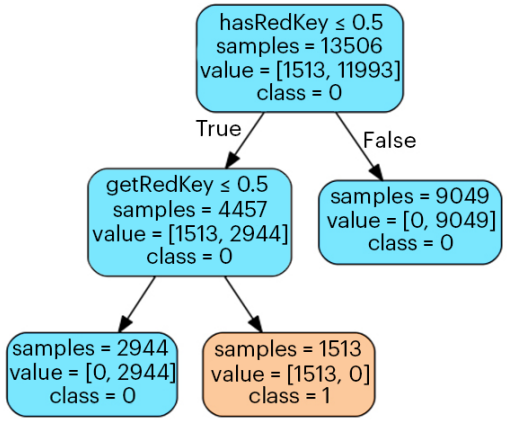
Figures A.1 and A.2 provides the individual decision tree models for each state variable once the framework has converged to the final *factored state sets* for the experiments described in Section 4.3.



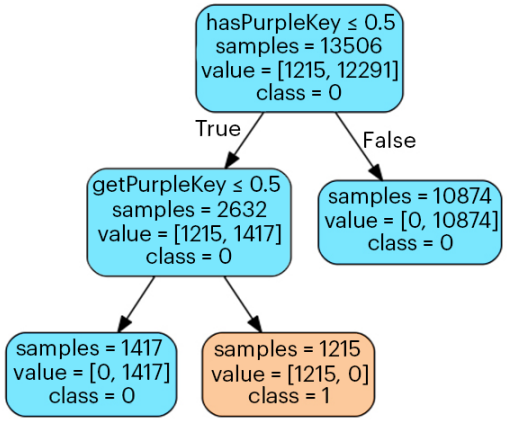
(a) hasBlueKey Tree



(b) hasYellowKey Tree

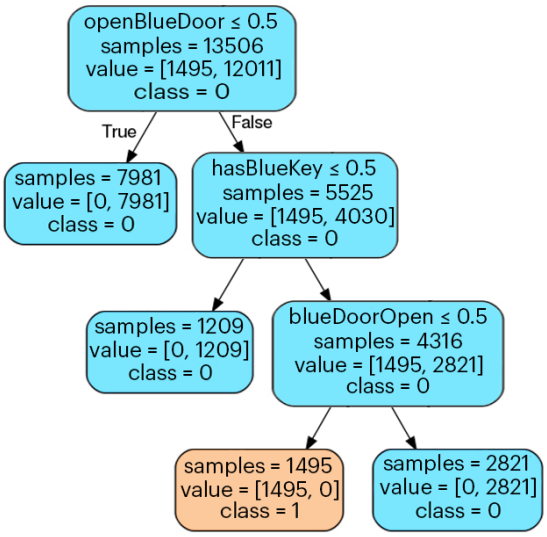


(c) hasRedKeyTree

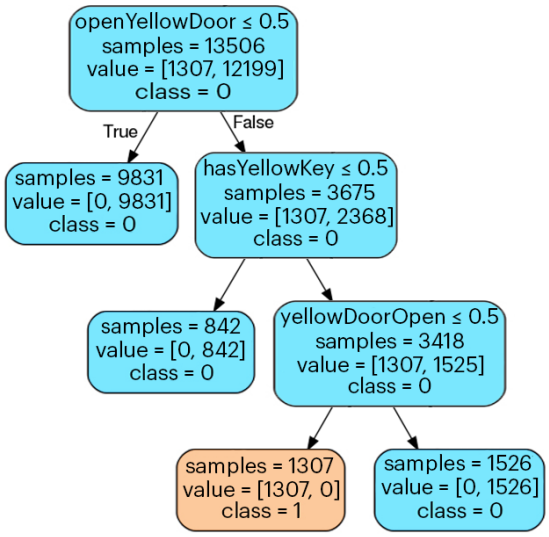


(d) hasPurpleKeyTree

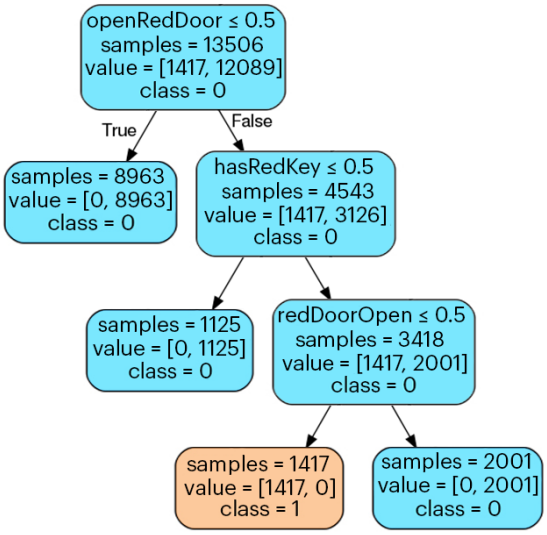
Figure A.1: Converged decision trees for the key variables for experiments described in Section 4.3



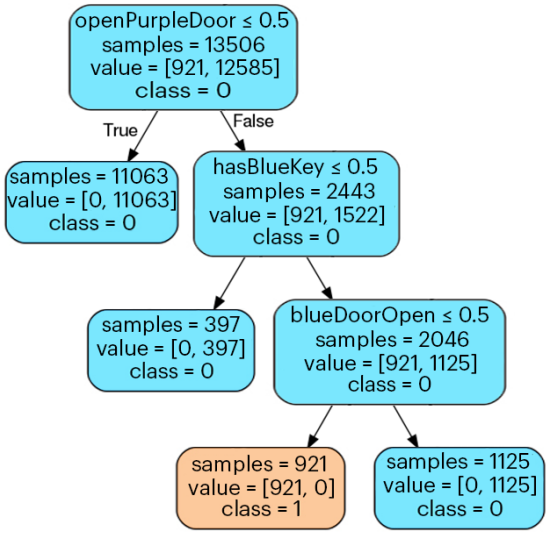
(a) blueDoorOpen Tree



(b) yellowDoorOpen Tree



(c) redDoorOpen



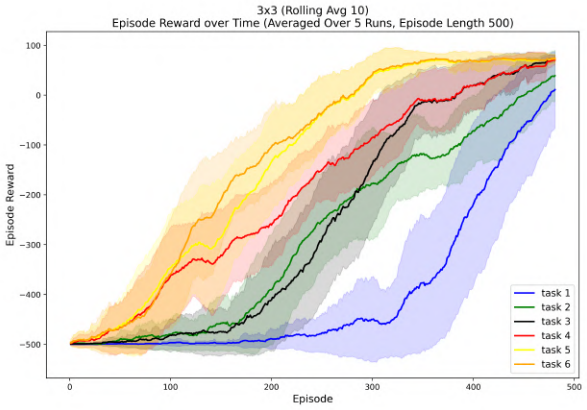
(d) purpleDoorOpen

Figure A.2: Converged decision trees for the door variables for experiments described in Section 4.3

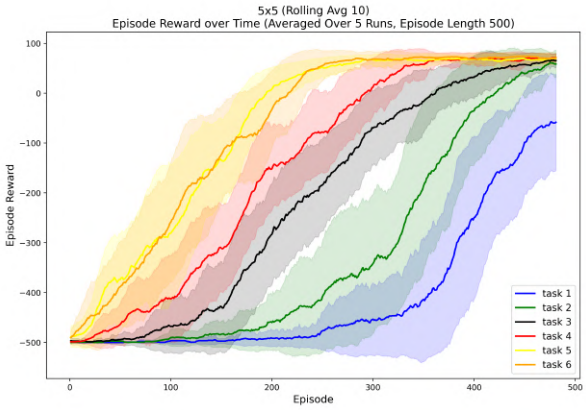
Appendix B

Grid Size Experiments

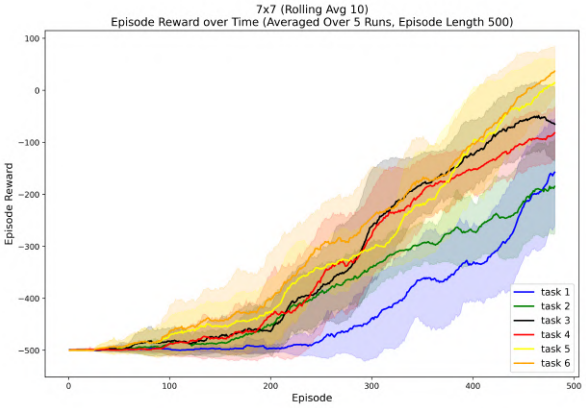
As discussed in Section 4.4.1, the grid size of the vision window available to the agent plays a large role in the performance. We evaluate agents with grids of size 3×3, 5×5 and 7×7 to determine which grid size to use for our experiments in Chapter 4. We set up our experiments using SMDP Q-Learning with the parameters the same as described in Section 4.3. Figure B.1 shows the performance of the factored agents in each scenario. Our framework shows consistent improvement given additional training domains, across all grid sizes. When looking at the performance in terms of transfer we can see that a 5×5 grid displays the greatest benefit in terms of jumpstart and total reward. The 3×3 grid outperforms the 7×7 as it is too restricted to the domain in which it was learned and cannot generalise well to the test domains. From these experiments, we choose to use the 5×5 grid in our experiments in Chapter 4.



(a) 3×3 Vision Grid Size



(b) 5×5 Vision Grid Size



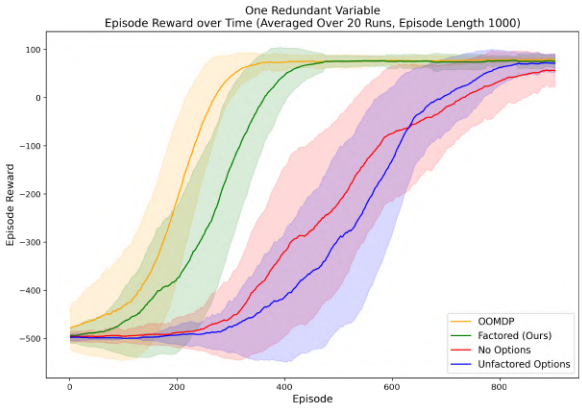
(c) 7×7 Vision Grid Size

Figure B.1: Comparison of factored option performance for various vision grid sizes.

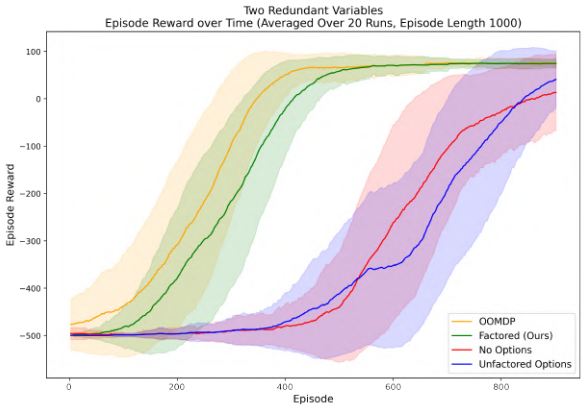
Appendix C

Redundant and Noisy Domain Experiments

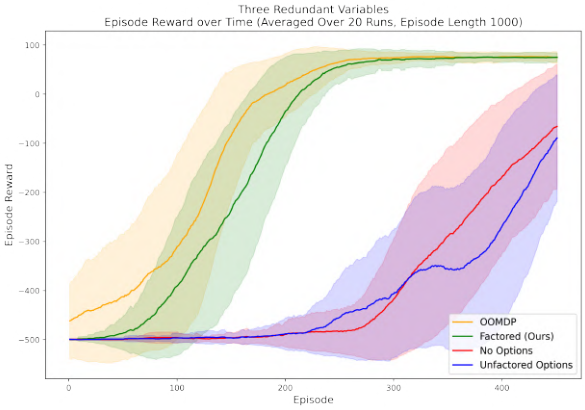
In Section 4.5 we detailed the experiment where we compared the performance of the factored, unfactored, non-option and Doormax algorithms when dealing with 1, 2 and 3 noisy variables. In Figure C.1 and C.2 we provide the individual learning curves for these algorithms which were used to generate the total reward plot shown previously in Figures 4.9 and 4.10 respectively.



(a) One Redundant Variable

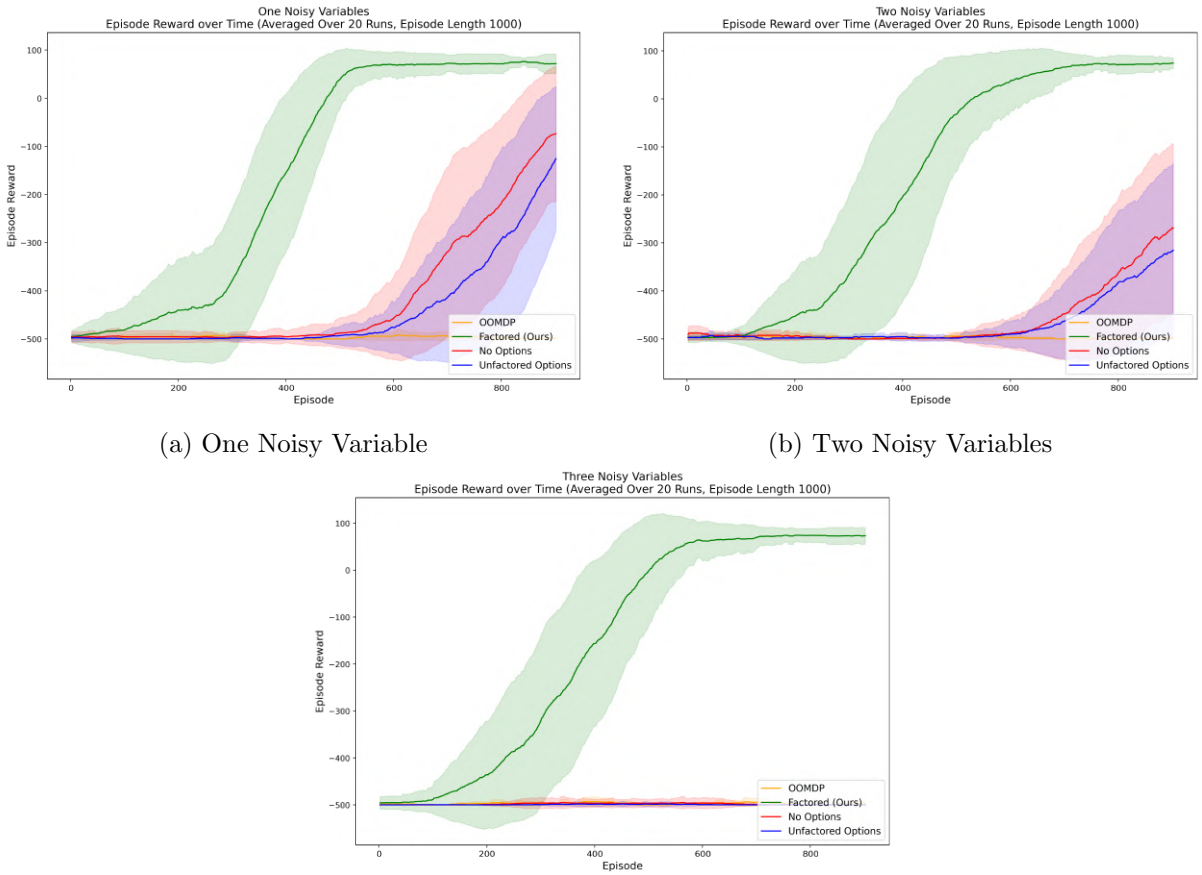


(b) Two Redundant Variables



(c) Three Redundant Variables

Figure C.1: Comparison average reward per episode for domains with various numbers of redundant variables. Our framework is compared to DOORMAX, no option Q-Learning and unfactored option SMDP Q-Learning. Graphs are plotted using a rolling average with a window size of 20 episodes.



(a) One Noisy Variable

(b) Two Noisy Variables

(c) Three Noisy Variables

Figure C.2: Comparison average reward per episode for domains with various numbers of noisy variables. Our framework is compared to DOORMAX, no option Q-Learning and unfactored option SMDP Q-Learning. Graphs are plotted using a rolling average with a window size of 20 episodes.

Appendix D

Hardware Specifications

Computations for this research were performed using infrastructure provided by the Center for High Performance Computing (CHPC). Experiments were run in parallel on their Dell Linux cluster consisting of Intel 5th Generation CPUs, with each node of the cluster containing 24 cores and 128GB of memory. Code was written in Python 3.9 and made use of the following libraries:

- *ai2thor 4.2.0*
- *gym 0.19.0*
- *matplotlib 3.1.1*
- *networkx 2.5*
- *numpy 1.19.5*
- *pandas 0.25.1*
- *plotly 5.1.0*
- *scikit-learn 0.21.3*