WITS UNIVERSITY
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science and Applied Mathematics

Progressive Skill Extraction with a Curriculum of Tasks

Shahil Mawjee
558028

WITS UNIVERSITY

Supervised by:
Dr Pravesh Ranchod and Prof Benjamin Rosman

Declaration

I, Shahil Mawjee, declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Masters of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

Shahil Mawjee (558028)

30 October 2020

Date

Abstract

When using reinforcement learning to acquire behaviours, solving complex tasks is often infeasible due to the curse of dimensionality and the need to explore the search space fully. An approach humans use to address this is making use of structured curricula which proceed from simple tasks to more complex tasks, allowing the learner to leverage knowledge gained in more straightforward tasks when solving more complex tasks.

However, this problem not yet been solved for artificial learning agents. In order to achieve this and be fully autonomous, the agent needs to be able to learn to select related tasks, learn how tasks are related and have a mechanism to transfer knowledge between tasks.

In this dissertation, we present a training strategy for a reinforcement learning agent to progressively and autonomously extract skills (as options, from hierarchical reinforcement learning) for a given curriculum of tasks. With each iteration, the library of options is refined to be more robust for transfer among the tasks from the curriculum. Our training strategy makes use of the NPBRS algorithm (non-parametric Bayesian reward segmentation) a recently developed technique for extracting skills from demonstrations. We extend the NPBRS algorithm to operate in an iterative manner, so that we can generalise skill policies across the range of tasks and prevent recovering duplicate skill policies.

After that, we augment the NPBRS output to include an inferred initiation set and termination condition. This transforms the trajectory segments and skill policies into usable options. In our benchmark task in the *code-game-world environment*, our reinforcement learning agent was able to learn the target task using around 71.32% fewer samples when given access to a library of options learned using a well-designed curriculum.

Contents

List of Figures

Chapter 1

Introduction

In an ideal world, robots would be able to perform a wide range of tasks in a specific environment. Consider a robot in the kitchen where the robot would be expected to carry out a large number of different tasks, such as *"bake a cake"*, or *"prepare a cup of coffee"*. The challenge here is that the robot will need a wide range of skills. It will need to be able to perform a set of known tasks as well as have the ability to learn novel tasks. Implementing such a robot in the real world is very difficult to achieve.

We can take some inspiration from how humans learn to perform a range of tasks. We humans operate at a level of abstraction, instead of thinking about every muscle twitch, we are able to decompose a task into small sub-tasks. The same sub-task is encountered when solving many different tasks hence instead of solving the sub-task over and over; we learn a skill that we can reuse when solving a range of tasks. We can identify and utilise a variety of skills which are necessary to perform each task. If we have a skill available that could be useful in solving a particular task, we will use it. If we do not, we can construct new skills in an online manner. For example, when learning to *"prepare a cup of tea"*, we incidentally learn the skills of *"pouring water"* and *"boiling water"*. If subsequently, we need to *"prepare a cup of coffee"* we can use our existing skills to simplify the process.

It can be difficult for us to learn a complex task when we have no prior knowledge of the environment and no skills available. Similarly, this problem exists in robotics when learning to solve a new task. The robot is typically not able to take advantage of skills learned in previously solved tasks, and must thus learn, typically by trial-and-error in the solution space. This is both time-consuming and potentially dangerous to the robot and its environment.

As humans, we can exploit the skills we have already learned while simultaneously acquiring new skills. In order to fully exploit this ability to learn skills incrementally, human learning

makes use of carefully designed curricula, allowing us to learn necessary skills initially and build up more and more complex skills while progressing through the curriculum of tasks of increasing difficulty. For example, we first learn to *add* and *multiply* numbers before learning to *integrate* an equation. We wish to mimic this learning system to allow robots to build a library of skills and learn a range of tasks more efficiently.

We make use of a framework that is able to make decisions under uncertainty. This framework is called Reinforcement Learning (RL). The decision-making entity is known as the *agent*, and in our case, this is the robot. The RL agent learns by interacting with the environment, by taking an action and in turn, the environment provides feedback. The lowest level (primitive) actions could be similar to a muscle twitch or a single joint movement of a robot. The number of correct sequential decisions that must be made in order to successfully solve a task could thus become very large. The probability of selecting such an unbroken correct sequence thus becomes very small, requiring us to explore for a very long time before we discover good sequences.

Skills can be thought of as higher-level actions which combine a large number of primitive actions into a single decision. The sequence of correct decisions which must be made when using skills is thus shortened. However, this is only effective if these high-level actions are appropriate for the task being learned. An important open problem is how to construct libraries of skills.

In this work, we have developed a training strategy to build a library of skills by progressing through a curriculum. We make use of a curriculum of tasks, which we define manually, in which tasks are in ascending order of complexity. For each task from the curriculum, new concepts are introduced, and our system extracts new useful skills and refines previous skills to generalise across tasks. The library of skills grows while progressing through the curriculum, thereby starting each new task with more prior knowledge.

For each task from the curriculum, our proposed training process can be divided into two steps. The first step is learning to solve the task with the aid of the library of skills that were acquired from previously solved tasks. We make use of the SMDP Q-Learning algorithm [Bradtke and Duff 1995] to solve each task. The goal for step two is to extract transferable and reusable skills utilised during the task. We make use of the NPBRS (*non-parametric Bayesian reward segmentation*) algorithm [Ranchod *et al.* 2015] to identify and extract useful skills from the demonstration of the task.

In this dissertation, we provide three main contributions: (1) we extended the NPBRS algo-

rithm to operate in an iterative manner. (2) We developed a method to learn the initiation set and termination condition for the trajectory segments from NPBRS, and along with the skill policy, we are able to define an option (in a hierarchical RL sense) [Cockcroft *et al.* 2020]. (3) We created a training strategy to build a library of options while progressing through the curriculum of tasks.

In chapter 2, we provide formal definitions to the concepts we use throughout the dissertation. Chapter 3 focuses on the proposed training strategy and the details of each step. Chapter 3 also provides details of the modifications made to the NPBRS algorithm. In chapter 4, we investigate the advantages of learning with the aid of a library of skills. Finally, in chapter 5, we summarise our findings and provide avenues for future research.

Chapter 2

Background

In this chapter, we provide a brief background of the concepts and notation that we use throughout the document. In section 2.1, we briefly visit the idea of learning with the aid of curriculum. This leads us into section 2.2 where we discuss the Reinforcement Learning framework for learning to solve a task. Then we explore inverse reinforcement learning methods, in section 2.3, to learn from expert demonstrations, followed by section 2.4, we investigate *non-parametric Bayesian reward segmentation* (NPBRS) which is a recently developed technique for extracting skills from demonstrations. In section 2.5 we formally define the notation of a skill. We than discuss common metrics when dealing with transfer, in section 2.7.

2.1 Curriculum Learning

Humans learn much better when the tasks are not randomly presented but rather when organised in a meaningful order which introduces new concepts progressively, and gradually more complex tasks [Bengio *et al.* 2009]. The idea is to start simple, learn simple concepts and then gradually increase the task's complexity to learn more complex concepts. By choosing which tasks to present and in which order to introduce them to the *learner*, one can guide training and notably increase the speed at which learning can occur. Elman [1993] states that a curriculum learning strategy could make it possible for humans to learn what might otherwise prove to be "unlearnable".

Sanger [1994] explored a similar concept in robotics by gradually making the learning task more difficult. In the field of reinforcement learning, using a curriculum is a type of training strategy [Narvekar *et al.* 2016]. Narvekar *et al.* [2016] experimentally show that training using a curriculum has a substantial impact on the learning speed or performance of an agent and that the sequence of tasks in the curriculum does matter. A sequence of tasks represents the curriculum, where the tasks are ordered from simple to more complex tasks. The agent is required

to learn how the tasks are related and effectively transfer knowledge while progressing through the curriculum. In the next section, we describe in detail how each task can be solved and then explore mechanisms to accumulate knowledge.

2.2 Reinforcement Learning

This section provides an introduction to reinforcement learning (RL), a field involved with learning by interaction within an environment. Reinforcement learning is an area of machine learning inspired by behaviourist psychology and neuroscience [Hassabis *et al.* 2017]

Sequential decision problems are a large subset of problems in the field of artificial intelligence (AI). In these problems, the decision-making entity is known as the agent. The agent attempts to maximise its utility by making a sequence of decisions. At each time step, the agent receives observations from its environment and acts accordingly. As a consequence of its action, the agent receives feedback and then the agent transitions to the new state. Whereas the feedback (reward) represents only the immediate outcome, the utility captures the long-term consequences of actions. The goal of the agent is to maximise the rewards it receives over time.

More formally, at time t, the agent receives the current state as an observation $s_t \in S$ and selects an action $a \in \mathcal{A}(s_t)$. Action a is selected according to the agent's policy π. The agent then receives a numerical reward $r_{t+1} \in \mathbb{R}$ and transitions to the new state s_{t+1}. The series of observations, actions and rewards constitute the agent's experience. Figure 2.1 provides a visual of this process. The agent learns from this experience to modify the policy π and maximise the total expected reward, given by

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

where $0 \leq \gamma \leq 1$ is used to discount future rewards.

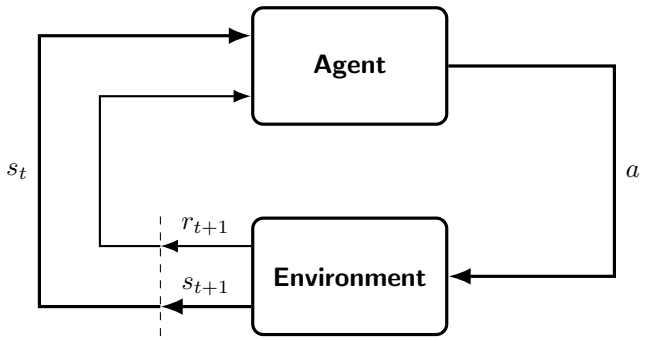


Figure 2.1: The Agent-Environment Interaction [Sutton and Barto 1998]

A sequential decision problem can be modelled as a Markov decision process (MDP) only if its transition function is Markovian - that is, the conditional probability of reaching state s_{t+1} depends only upon the present state s_t and action a_t, not on the sequence of events that preceded it [Puterman 1995], i.e.,

$$p\left(s_{t+1}|s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t, a_t\right) = p\left(s_{t+1}|s_t, a_t\right) \tag{2.2}$$

The Markov decision process (MDP) can be defined as

$$M = (S, \mathcal{A}, T, R, \gamma)$$

where S is a set of possible states in the environment, and $\mathcal{A}(s_t)$ is the set of actions available at state s_t. The transition function $T(s_t, a, s_{t+1})$, is the probability the agent will transition to state s_{t+1}, given that agent is at state s_t and taking action a. The transition function may be stochastic such that the action a taken at state s_t may not always transition to the same state s_{t+1} each time. The reward function $R(s_t, a, s_{t+1})$, is the reward obtained by the agent for taking action a at state s_t and transitioning to state s_{t+1}. The discount rate γ, with $0 \leq \gamma \leq 1$, indicates how strongly the agent prefers rewards to occur sooner rather than later.

A policy $\pi : S \times \mathcal{A} \rightarrow [0, 1]$ is a function mapping current state to action choice. For a policy π, the value function $V^\pi : S \rightarrow \mathbb{R}$ gives the value of a state s as a long-term expected cumulative reward incurred from the state by following policy π. The value of policy π is,

$$V^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(s, a) \left[\sum_{s'} T(s, a, s') \left(R\left(s, a, s'\right) + \gamma V^\pi\left(s'\right) \right) \right] \tag{2.3}$$

The goal of solving MDP M is to find an optimal policy π^* such that $V^{\pi^*}(s) = V^*(s)$, for all $s \in S$. In order to find an optimal policy we need to know which action a to take from state s that will lead us to the state s' with the highest value. The Q-value function, $Q^\pi : S \times \mathcal{A} \rightarrow \mathbb{R}$, captures the value of each action taken from each state. The policy π then compares the state-action values for a state s and selects the action a with the maximum state-action value. The optimal Q-value function $Q^*(s, a)$ can be written in terms of $V^*(s)$,

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R\left(s, a, s'\right) + \gamma V^*\left(s'\right) \right] \tag{2.4}$$

2.2.1 Temporal Difference Learning

The equation above requires the complete model of the environment's dynamics. The model is not always completely known or accessible. Temporal difference (TD) methods can learn directly from raw experience without a model of the environment's dynamics [Sutton and Barto 1998]. The TD learning algorithm of Sutton [1985] uses a technique known as bootstrapping

and alleviates the need to wait for the episode to complete before making an update. Temporal difference methods update the estimate of the value of the state $V(s_t)$ based on the values of subsequent states. The most basic form of temporal difference algorithms, TD(0), uses only the value of the successor state s_{t+1}.

Temporal difference methods update state values in the direction of a target value. Let δ_t be the error between the state and target value. Then state-value updates take the form

$$V\left(s_t\right) \leftarrow V\left(s_t\right) + \alpha\delta_t \tag{2.5}$$

where α is a step-size parameter that controls the learning rate. The error δ_t for TD(0) is

$$\delta_t = r_{t+1} + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \tag{2.6}$$

Algorithm 1 shows how the TD update is preformed when evaluating policy π. The $Step$ function takes the action a and returns the immediate reward r and next state s', based on the environment transition dynamics.

Temporal difference learning need not just update its values from the next state, but also the values many steps into the future can be used. This is known as the TD(λ) algorithm, where $0 \leq \lambda \leq 1$ refers to the span over which bootstrapping occurs: TD(0) bootstraps from the immediate successor, while TD(1) updates from the final return. In other words, TD(1) and Monte-Carlo methods are equivalent [Sutton and Barto 1998].

Algorithm 1: Tabular TD(0), for policy evaluation

Initialisation: An arbitrary initialised vector V

Result: Value of policy π

$s \leftarrow$ initial state

while s *is not terminal* **do**

$a \leftarrow \pi s$

$r, s' \leftarrow$ `Step(`s`)`

$V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$

$s \leftarrow s'$

end

The TD algorithm only solves the policy evaluation problem but not the control one. The equation 2.5 can be updated as the action-value function,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma\left(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)\right] \tag{2.7}$$

Sarsa (algorithm 2) [Rummery and Niranjan 1994; Singh and Sutton 1996] is an on-policy control algorithm that makes use of the TD for policy evaluation with an exploration strategy. Exploration is when the agent chooses an action to learn more about the environment. This must be balanced with exploitation, when the agent selects what it currently values to be the best action. A straightforward approach that balances the two is ϵ-greedy action selection. The agent selects a random action with the probability ϵ, and the best action is selected with probability $1 - \epsilon$ (where $\epsilon \in [0, 1]$). The policy π is derived from the current estimate of the action-value function, where a random action is selected with the probability of ϵ and the action with the maximum value is selected with the probability of $1 - \epsilon$. *Sarsa* is guaranteed to converge to an optimal policy, as the number of state-action samples goes to infinity (as long as the ϵ decays to 0) [Sutton and Barto 1998].

Algorithm 2: Sarsa: On-policy TD Control

Parameters : step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialisation: $Q(s, a)$, for all $s \in S$, $a \in \mathcal{A}(s)$ arbitrarily, except that $Q(terminal, \cdot) = 0$

Result: An optimal action-value function Q for control

for *episode* **do**

  $s \leftarrow$ initial state

  $a \leftarrow$ `SelectAction` (s)

  while x *is not terminal* **do**

    $r, s' \leftarrow$Step(a)

    $a' \leftarrow$ `SelectAction` (s')

    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$

    $s \leftarrow s'$

    $a \leftarrow a'$

  end

end

Since TD learning is an on-policy learning method, the choice of exploration strategy directly impacts the convergence to the estimation policy. *Q-learning* (algorithm 3) [Watkins 1989; Sutton 1988] decouples the exploration and evaluation and allows for any policy to be followed while still converging to optimal action-value function Q^*. The update of the action-value function is,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \left(\max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right) \right] \tag{2.8}$$

Algorithm 3: Q-learning: Off-policy TD Control

Parameters : step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialisation: $Q(s, a)$, for all $s \in S$, $a \in \mathcal{A}(s)$ arbitrarily, except that $Q(terminal, \cdot) = 0$

Result: An optimal action-value function Q for control

for *episode* **do**

  $s \leftarrow$ initial state

  while x *is not terminal* **do**

    $a \leftarrow$ `SelectAction` (s)

    $r, s' \leftarrow$Step(a)

    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a' \in \mathcal{A}} Q\left(s', a'\right) - Q(s, a) \right]$

    $s \leftarrow s'$

  end

end

2.3 Inverse Reinforcement Learning

When we humans teach someone to do a task, rather than providing feedback once they have completed the task, it is often much easier and more natural to demonstrate the task to them, so that they can learn from the demonstration. If we already know the desired behaviour and not the reward function, then the logical step is to learn from a demonstration and infer the reward function.

Inverse reinforcement learning (IRL) is the problem of learning the preferences of an expert using its observed behaviour, thereby avoiding a manual specification of its reward function [Russell 1998; Ng *et al.* 2000]. We assume that the expert is behaving according to an underlying policy π_E. While roughly specified reward functions are sufficient for many tasks to obtain expected behaviour, more complex tasks can require much trial-and-error of delicate balance of multiple conflicting attributes [Coates *et al.* 2009], which becomes cumbersome. For example, instead of defining a reward function for the robot to learn how to *"hold a cup of coffee"*, we would be required to explicitly provide a reward at each step of the process of holding the mug upright to ensure the coffee is not spilt. This hand-crafted reward function could easily embed implicit domain-specific knowledge. Russell [1998] points out that the reward function is inherently more transferable compared to the expert agent's policy π_E.

The need to pre-specify the reward function limits the applicability of RL to problems where a reward function can be easily specified or simulated. IRL offers a way to reduce the manual design of task specification, given a demonstration of the desired behaviour is available. Access to demonstrations of behaviours is often in the form of recorded data, which are also known

as *trajectories*. Thus, IRL forms a critical method for learning from demonstration (LfD) [Ng *et al.* 2000; Abbeel and Ng 2004; Argall *et al.* 2009]. The reward function is unknown but some structure is assumed which helps when learning. Common assumptions include a linearly weighted combination of reward features, a probability distribution over reward functions, or a neural network representation [Abbeel and Ng 2004; Ziebart *et al.* 2008; Wulfmeier *et al.* 2015].

Approaches to IRL predominantly depend on using a Markov decision process to model the interaction of the observed expert with its environment, and whose solution is a policy that maps states to actions [Arora and Doshi 2018]. The reward function of this MDP is unknown, and the observed expert is assumed to follow an optimal policy for the MDP. We use the same MDP definition from section 2.2, with an additional parameter S_0, where S_0 is the initial state distribution, from which the start state s_0 is drawn. Let MDP/R denote an MDP without a reward function

$$M/R = (S, A, T, \gamma, S_0)$$

Given a set of m demonstrations $\mathcal{D} = \{\zeta_i\}_{i=1}^{m}$ by an expert, following policy π_E, consisting of trajectories $\zeta_i = ((s_0^i, a_0^i), (s_1^i, a_1^i), ..., (s_{n-1}^i, a_{n-1}^i), (s_n^i, a_n^i))$. The aim is to recover the underlying reward function R which explains the behaviour of the expert π_E.

Another perspective to the value function (equation 2.3) involves multiplying the reward with the converged state-visitation frequency $\psi^\pi(s)$, which is the number of times the state s is visited using policy π.

$$\psi^\pi(s) = \psi^0(s) + \gamma \sum_{s' \in S} T\left(s, \pi(s), s'\right) \psi^\pi\left(s'\right) \tag{2.9}$$

where $\psi^0(s)$ is initialised as 0 for all states. Let Ψ be the space of all ψ functions. Iterating the above until the state-visitation frequency stops changing yields the converged frequency function, ψ_*^π [Ziebart *et al.* 2008]. The value function can be written in the form,

$$V^*(s) = \sup_\pi \sum_{s \in S} \psi_*^\pi(s) R(s, \pi(s)) \tag{2.10}$$

The reward function is then a linear sum of weighted features,

$$\begin{aligned} R(s,a) &= w_1\phi_1(s,a) + w_2\phi_2(s,a) + \ldots + w_k\phi_k(s,a) \\ &= w^T\phi(s,a) \end{aligned} \tag{2.11}$$

where $\phi_k : S \to \mathbb{R}$ is a feature function and weight $w_k \in \mathbb{R}$. Then, define the expected feature count for policy π and feature ϕ_k as [Abbeel and Ng 2004],

$$\mu^{\phi_k}(\pi) = \sum_{t=0}^{\infty} \psi^\pi\left(s_t\right) \phi_k\left(s_t, \pi\left(s_t\right)\right) \tag{2.12}$$

The expected feature count can be used to define the expected value of a policy [Arora and Doshi 2018],

$$\begin{aligned} V^{\pi} &= w^{T}\mu^{\phi}(\pi) = \sum_{s,a} \psi^{\pi}(s) w^{T}\phi(s,a) \\ &= \sum_{s,a} \psi^{\pi}(s) R(s,a) \end{aligned} \tag{2.13}$$

The expected visitation-frequency of features in the demonstrations $\psi(\zeta)$ provided by the expert π_E should equal the expected visitation-frequency of features visited by the learner π_L following the recovered reward function. This is known as feature-expectation matching,

$$\mathbb{E}_{\pi_L}[\psi(\zeta)] = \mathbb{E}_{\pi_E}[\psi(\zeta)] \tag{2.14}$$

Inverse reinforcement learning is challenging because the optimisation associated with finding a reward function that best explains observations is essentially ill-posed. There are many possible reward functions that could explain the trajectories [Ng *et al.* 2000]. In fact, a reward function in which every move is given a reward of zero makes every possible trajectory optimal, leading to a trivial degenerate solution [Ng *et al.* 2000]. Hence IRL suffers from ambiguity in solution.

Abbeel and Ng [2004] overcome this problem by implementing two additional criteria to identify better choices of R. The first criterion finds the maximum margin hyperplane separating the best solution and the second-best solution. This penalises uninformative reward functions such as the zero reward since all solutions, for the underlying MDP, are equally good. The second additional criterion introduces a bias towards simple rewards. This selects reward functions which have small rewards in most states, and large rewards in a few states.

Ziebart *et al.* [2008] take a probabilistic approach to reason the uncertainty in imitation learning. Maximum Entropy (MaxEnt) IRL employs the principle of maximum entropy [Jaynes 1957] to resolve the ambiguity in choosing a distribution over decisions. This attempts to give the least biased estimate of the reward distribution that matches the given trajectories $\Pr(\tau)$, meaning that the reward function should not lead to any preferences that are not present in the demonstrations. The maximum entropy approach provides a principled method of dealing with this uncertainty.

MaxEnt Deep IRL uses a neural network to approximate the reward function. This allows for complex and non-linear reward functions [Wulfmeier *et al.* 2015]. Both MaxEnt methods are challenging to apply to complex tasks, where the MDP is itself complicated, particularly in robotic systems with unknown dynamics. This is because they require carefully designed features to infer the structure of the reward function as well as solve the MDP, which is finding

an optimal policy given the current reward function, in the inner loop of an iterative reward function optimisation.

The problem with IRL is that it will recover a single reward function for the entire demonstration of the skill. This will require an expert to demonstrate the same skill repeatedly. It is more natural to demonstrate a complete task and find all the skills employed. The same skill can be demonstrated many times, with variations, in the same task.

2.4 Task Segmentation

When learning from a demonstration, we humans can identify and learn the individual skills that are present within the demonstrated task. In other words, we can acquire various component skills when learning to perform the task. Each of these skills could have a different sub-goal, and these skills can be chained together in order to achieve the task goal.

In section 2.3, we explored methods of inferring the goal from demonstrations of a task. These methods recover a single reward function and policy for the complete task, which might limit the usefulness of the policy, besides when performing the complete task. The ability to identify and extract the various repeated sequences of actions would be more useful and reusable.

The demonstrations are in the form of trajectories, as described in section 2.3. We need to extract the skills encoded within the trajectories by segmenting the demonstrated trajectories according to the skill employed in each segment. However, we have no access to a description of the component skills or annotations of when they are employed. We thus need to encode some biases to avoid classifying random trajectory sub-sequences as skills. We thus need some characteristics and metrics that can be used to evaluate a segment to determine if it is a valid skill.

Konidaris *et al.* [2012] present the CST (for Constructing Skill Trees) algorithm, which segments trajectories using change-point detection. Where each segment is mapped to a single value function, value functions are defined over state-space; this might limit skill reuse. The CST method produces options (refer to section 2.5) which can be chained together with the termination set of an option corresponding to the initiation set of its successors.

Bacon *et al.* [2017] developed a gradient-based approach for learning skills (as options) which is known as the Option-Critic framework. The Option-Critic framework is an end-to-end procedure which only requires specifying the number of options to extract for the task on hand.

Ranchod *et al.* [2015] developed non-parametric Bayesian reward segmentation (NPBRS), a method to discover skills from demonstration trajectories. NPBRS hypothesises that specific properties will determine whether a trajectory-subsequence is a skill. These properties are:

- **Reusability** – A skill is more valuable if it is usable in multiple tasks.
- **Usefulness** – Skills which do not accomplish any desirable outcome are not worth retaining.
- **Linked** – The use of skill will cause a bias in the choice of the next skill.

Ranchod *et al.* [2015] use a statistical model to encode these properties. In order to encode the bias towards reusability, it is necessary to assign higher likelihoods to hypotheses in which common skills occur frequently. A model which encodes this bias is the Dirichlet process, in which a concentration parameter dictates the increase in likelihood assigned to a common skill versus an uncommon one. By using the Dirichlet process, NPBRS will penalise skill hypotheses in which many infrequent skills are employed rather than a smaller number of frequently encountered skills, resulting in the creation of a more compact skill set.

To ensure a skill is useful, the skill must be goal-directed. Given the skill's trajectories segments, inverse reinforcement learning can recover the skill's reward function, which encodes the goal of the demonstrator. If IRL is performed on a particular skill and no reward function can represent the skill's goal, then it is likely that the segmentation incorrectly identified a single skill in a segment when multiple skills would be more appropriate. For example, if we incorrectly classify *"running and jumping"* as a single skill, then it could be difficult to represent the goal of the combined skill as a reward function.

In order to ensure that the transitions between skills are coherent, we first note that the demonstrator is assumed to be performing a task rather than a sequence of random skills. We would thus expect there to be meaningful transition dynamics between the skills conditioned upon the task. For example, we would expect that in soccer, it would be more likely to transition from *dribbling* directly to *passing* than from *dribbling* directly to *taking a free-kick*. Given that the demonstrator can be performing different tasks in each demonstration, we would expect to observe different transition dynamics, for instance, in *baking* as opposed to *soccer*. In order to recover these transition dynamics, we model the system using a Hidden Markov Model. If the skill transition dynamics do not fit those of other instances of the same task, then the skills are unlikely as it would be expected that the skill transition dynamics would be similar within the same task.

Ranchod *et al.* [2015] enforces all three properties using a Beta Process Hidden Markov Model (BP-HMM) [Fox 2009]. The Beta process generates skill membership which defines the probability of utilising the skill within a task. That means the transition distribution is mode-set specific, in other words depending on which skills are used in the particular trajectory, the transition probabilities are related to other trajectories that have the same skills in them. Then we use a Dirichlet process generated from the Beta process, so the hypothesised skill is more likely to be true if it occurs often. Hence the Dirichlet process ensures that the skill satisfies the property of being reusable.

The Hidden Markov Model (HMM) represents the skill transition dynamics with each hidden mode representing a skill. The HMM transition dynamics are drawn from the Dirichlet process. Since the mode is hidden, we only have access to the observations, which are drawn from the skill-specific emission distribution. In our context, the emission distribution is a skill-specific action selection policy.

More formally, this BP-HMM is the forward generative model.

$$
\begin{aligned}
B|B_0 &\sim \mathrm{BP}\left(1, \beta_0\right) \\
X_i|B &\sim \mathrm{BeP}(B) \\
\pi_j^{(i)}||f_i, \gamma, \kappa &\sim \mathrm{Dir}\left([\gamma, \ldots, \gamma+\kappa, \gamma, \ldots] \otimes f_i\right) \\
z_t^{(i)} &\sim \pi^{(i)}_{z_{-1}^{(i)}} \\
\mathrm{y}_t^{(i)} &= \sum_{j=1}^{r} A_{j, z_t^{(i)}} \mathrm{y}_{t-j}^{(i)} + \mathrm{e}_t^{(i)}\left(z_t^{(i)}\right)
\end{aligned}
\tag{2.15}
$$

B is drawn from a Beta process (BP) which provides a set of global probabilities for each skill. These probabilities are used to produce a Bernoulli process (BeP), from which X_i is drawn. For the *ith* trajectory, distribution X_i is used to construct a binary feature vector f_i, indicating which skills are present for this trajectory. Next, for mode j, we define the transition probabilities for time series i using the transition probability vector $\pi_j^{(i)}$, which is drawn from a Dirichlet distribution. The hyperparameter κ is used to place extra expected mass on the *jth* component, making the selection "sticky" since skills are likely to be employed for multiple sequential time steps. For each time step t, a mode $z_t^{(i)}$ is drawn from the transition distribution at time step $t-1$. The observation for the *ith* time series, at time t, is represented by $y_t^{(i)}$. If the order of the model is r, then $y_t^{(i)}$ is computed as a sum of mode-dependent linear transformations using the previous r observations, as well as the model-dependent noise term $\mathrm{e}_t^{(i)}$ [Ranchod 2017].

Given that we have demonstration trajectories, there is a need to compute the forward model's parameters which is most likely to generate these trajectories. Metropolis-Hastings and Gibbs sampling are used in order to compute the likelihood of the model generating the trajectories. Ranchod *et al.* [2015] makes use of the birth and death reversible jump Markov Chain Monte Carlo (MCMC) sampler developed for BP-AR-HMM [Fox 2009]. The Metropolis-Hastings algorithm will propose birth or death moves to either create or remove skills. Then Gibbs sampling is used to figure out all the parameters of the model, including the action selection distribution, which is inferred using Inverse Reinforcement Learning.

The BP-AR-HMM emissions are modelled as VAR (vector autoregression) processes by Fox *et al.* [2011]. Instead, Ranchod *et al.* [2015] models the emissions as MDPs. For the given the mode sequence $z_t^{(i)}$, Ranchod *et al.* [2015] models the dynamics of the system as follows,

$$\begin{aligned} P(a)|z_t^{(i)} &= \frac{e^{\tau Q_t^{(i)}}\left(y_{t-1}^{(i)}, a\right)}{\sum_a e^{\tau Q^{z_t^{(i)}}}\left(y_{t-1}^{(i)}, a\right)} \\ a_t^{(i)} &\sim P(a)|z_t^{(i)} \\ y_t &\sim T\left(y_{t-1}, a_t^{(i)}\right) \end{aligned} \tag{2.16}$$

This model removes the step of sampling of parameters A and e for the BP-AR-HMM. Instead, it uses IRL to determine the agent's reward function and accompanying policy, and calculate the transition probabilities from a combination of the policy and the known dynamics of the environment. The graphical representation is shown in figure 2.2. The optimal policy for $Q^{z_t^{(i)}}$ gives the likelihood of each of the demonstration trajectories and selecting the set with the highest likelihood [Ranchod *et al.* 2015].

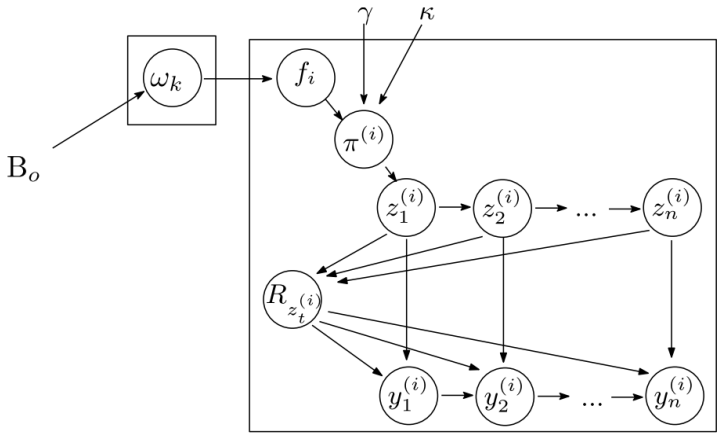


Figure 2.2: Graphical model of the BP-HMM applied to MDPs. $R_{z_t^{(i)}}$ represents the reward function of the MDP in mode $z_t^{(i)}$ [Ranchod *et al.* 2015]

The NPBRS algorithm outputs trajectory segments and skill policies. When using these skill

policies, the agent has no way of determining when to start or end using the skill policy. This is a problem, as the agent has to check all possible action and skill policies at every time-step, possibly reducing the learning performance. To alleviate this, we need a way to know when the policy can be initiated and a method to determine when to terminate. In this way it can be called in a *call-and-return* manner, without making a decision at each step. We explore a representation to do this in section 2.5.

2.5 Skill Representation

A skill can be thought of as a macro-action that is executed from a specific starting region until the end region, following a specific policy. These are precisely known as partial policies as they are generally defined for a subset of the state space. These partial policies are sometimes called temporally-extended actions, options [Sutton *et al.* 1999], skills [Thrun and Schwartz 1995] or macro-actions [Hauskrecht *et al.* 2013]. Sutton *et al.* [1999] developed a framework that is widely used and provides a clear notion of a skill in the reinforcement learning field. This framework is known as the "options framework".

The options framework [Sutton *et al.* 1999] defines an option $o \in \mathcal{O}$ as,

$$o = (I, \pi, \beta)$$

which consists of an initiation set $I \subseteq S$, a policy $\pi : S \times \cup_{s \in S} A_s \rightarrow [0, 1]$ and a termination condition $\beta : S \rightarrow [0, 1]$. The option is available in state s if and only if $s \in I$. This initiation set is meant to facilitate the decision making procedure by reducing the number of possible options to consider at a given state. In practice this initiation set is sometimes neglected by some authors, assuming that all options are available everywhere. If the option is executed, then actions are selected according to π until the option terminates stochastically according to the termination condition β [Sutton *et al.* 1999]. In other words, the option o can be executed in a *call-and-return* manner. For example, if the current state is s and action a is selected by the option with probability $\pi(s, a)$ then the environment makes a transition to state s', where the option either terminates with probability $\beta(s')$ or else continues, determining the next action a' with probability $\pi(s', a')$, and so on. When the option terminates, the agent can select another option from the set of those available at the termination state (i.e. options whose initiation sets contain the termination state), or it can select a primitive action [Barto and Mahadevan 2003].

The options framework is at the crossroad of Markov Decision Processes and semi-Markov Decision Processes (SMDP) of Bradtke and Duff [1995]. An MDP and a set of pre-defined options form a semi-Markov Decision Process, hence the theory of SMDPs can be used when using op-

tions. Formally this is stated in Theorem 1 and shown in figure 2.3.

Theorem 1[Sutton *et al.* 1999, Theorem 1] (MDP+Options = SMDP): *For any MDP, and any set of options defined on that MDP, the decision process that selects only among those options, executing each to termination, is an SMDP.*

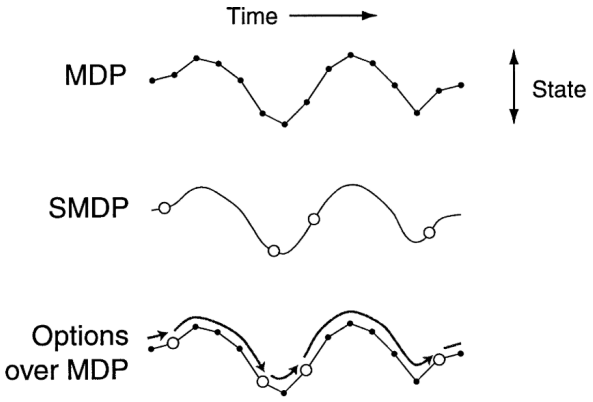


Figure 2.3: The state transitions of an MDP is a fixed discrete-time transition, whereas its larger and continuous-time transition in an SMDP. Options allow for an MDP trajectory to be seen as an SMDP trajectory [Sutton *et al.* 1999].

Let $\mu : \mathcal{S} \times \mathcal{O} \to [0,1]$ be a policy where $\mathcal{O}$ is a set of options. The primitive action $a \in \mathcal{A}$, is a special case option with the duration of one time-step $k = 1$, which are included in $\mathcal{O}$. The value of μ is defined as [Sutton *et al.* 1999],

$$V^{\mu}(s) = \mathbb{E}\left[R_{t+1} + \cdots + \gamma^{k-1} R_{t+k} + \gamma^{k} V^{\mu}\left(S_{t+k}\right) | \mathcal{E}(s,\mu,t)\right] \tag{2.17}$$

where k is the duration of the option selected by μ. The expected value is conditioned on $\mathcal{E}(s,\mu,t)$, the policy over options μ is executing at time t at state s. The corresponding equation for the value of an option o in state $s \in I$ is [Sutton *et al.* 1999],

$$\begin{aligned} Q^{\mu}(s,o) =& E\left[r_{t+1} + \cdots + \gamma^{k} r_{t+k} + \gamma^{k} V^{\mu}\left(s_{t+k}\right) | \mathcal{E}(s,o,t)\right] \\ =& E\Bigg[r_{t+1} + \cdots + \gamma^{k} \quad {}^{1} r_{t+k} \\ & + \gamma^{k} \sum_{o' \in \mathcal{O}_s} \mu\left(s_{t+k}, o'\right) Q^{\mu}\left(s_{t+k}, o'\right) | \mathcal{E}(s,o,t)\Bigg] \\ =& r_s^o + \sum_{s'} p_{ss'}^{o} \sum_{o' \in \mathcal{O}_{s'}} \mu\left(s', o'\right) Q^{\mu}\left(s', o'\right) \end{aligned} \tag{2.18}$$

where $\mathcal{O}_{s'}$ is the subset of options which can be initiated from state s'. The state-prediction part

of the model of o for state s is,

$$p_{ss'}^{\mathcal{O}} = \sum_{k=1}^{\infty} p\left(s', k\right) \gamma^k \tag{2.19}$$

for all $s' \in S$, where $p(s', k)$ is the probability that the option terminates in s' after k steps.

Finding an optimal policy over a set of options $\mathcal{O}$ can also be addressed by learning methods. Due to the fact that the MDP paired with a set of options $\mathcal{O}$ is an SMDP, the use of the existing theory of semi-Markov decision processes [Puterman 1995; Bradtke and Duff 1995] can be applied to finding an optimal policy over a set of options $\mathcal{O}$. With experience, the approximate option-value function $Q(s, o)$ can be computed. The SMDP version of one-step Q-learning, known as SMDP Q-learning [Bradtke and Duff 1995], updates after each option termination by,

$$Q(s, o) \leftarrow Q(s, o) + \alpha \left[R + \gamma^k \max_{o' \in \mathcal{O}_{s'}} Q\left(s', o'\right) - Q(s, o) \right] \tag{2.20}$$

holds the under the event of $\mathcal{E}(s, \mu, t)$ where option o is executed for k time steps with cumulative discounted reward R. The estimate $Q(s, o)$ converges to the optimal value $Q^*(s, o)$ for all $s \in S$ and $o \in \mathcal{O}$ under conditions similar to the Q-Learning algorithm [Parr and Russell 1998].

Options provide a level of abstraction; thus allowing the agent to learn and plan at a level of abstraction. The agent can learn to chain options together in order to solve the task. Therefore, the agent needs to have the ability to discover new options and improve existing options.

2.6 Support Vector Machine

In section 2.4, we discussed methods for extracting useful skills yet as the number of skills grows, the agent has more available actions to choose from, at each state. In an ideal situation, only the possible and fit-for-purpose skills should be available to the agent at a given state. In order to define these skills into Options, we need a method of learning the initiation and termination sets. In section 3.4.3, we discuss how we can use a supervised learning approach to achieve these sets.

A commonly used algorithm for supervised classification problems is the Support Vector Machine (SVM) [Vapnik 2013]. The SVM aims to split a given dataset into two separate classes by finding a boundary with the maximum margin between the two classes. The standard SVM require the training data to be labelled into two distinct classes. In some cases only one of the class's labels are available and so the standard SVM approach will not be able to separate the classes. An alternate approach could be to use an anomaly detection method such as the One-Class SVM (OC-SVM) [Schölkopf *et al.* 2001]. The OC-SVM determines a decision space which is in the form of a hypersphere. It contains much of the training data as possible while

attempting to minimise its volume. Given training data $X_1, \ldots, X_n \in d$, the OC-SVM first project this data into a higher-dimensional space, by mapping $\Phi : \mathcal{R}^d \to \mathcal{F}$. The hypersphere is then parameterised by a centre c and a radius r. These are computed by minimising the equation:

$$\min_{r,c,\xi} \left[r^2 + \frac{1}{vn} \sum_i \xi_i \right] \tag{2.21}$$

such that $\|\Phi(X_i) - c\|^2 \leq r^2 + \xi_i, \quad \xi_i \geq 0, \quad i = 1, \ldots, n$, where $v \in (0, 1)$ is a trade-off parameter between the radius and the number of training data points that fall inside the hypersphere, and ξ_i are slack variables which determine how far away outliers lie from the hypersphere surface.

2.7 Transfer Learning

As a human, we take advantage of our previously gained knowledge when solving a new task. This ability allows us to transfer knowledge between tasks; therefore it reduces the need to relearn skills that were present in previous tasks. The central idea of transfer is to use the knowledge gained from previous tasks to aid the learning of related but different tasks [Taylor and Stone 2009]. Transfer learning has been studied in many fields, transfer between machine learning tasks [Caruana 1995; Thrun 1996] and for planning tasks [Fern *et al.* 2004; Ilghami *et al.* 2005]. We are interested in transfer between reinforcement learning tasks. Taylor and Stone [2009] provides an extensive survey of this particular type of transfer.

Transfer learning in terms of RL, is the idea that the transfer of knowledge gained from previously solved task(s), known as source task(s), aids the learning of one or more target task(s) faster than if any transfer was not used. Transfer techniques have varying degrees of autonomy. To be fully autonomous, the agent would have to perform all of the following steps [Taylor and Stone 2009]:

1. Given a target task, select an appropriate source task or set of tasks from which to transfer.
2. Learn how the source task(s) and target task are related.
3. Effectively transfer knowledge from the source task(s) to the target task.

While the mechanisms used for these steps will necessarily be interdependent, Taylor and Stone [2009] states that "most transfer learning research has focused on each independently, and no transfer learning methods are currently capable of accomplishing all three goals".

An essential aspect of transfer learning is the set of the evaluation metrics to measure how

successful was the transfer. There are different goals where transfer may be beneficial. A possible goal of transfer could be to reduce the overall time required to learn a complex task. For this scenario, a *total time scenario*, which includes the time needed to learn the source task(s), would be most appropriate. Another possible goal of transfer is to reuse previously gained knowledge in a novel task effectively. For this scenario, a *target task time scenario*, which only accounts for the time spent learning in the target task, is reasonable. Refer to Figure 2.4.

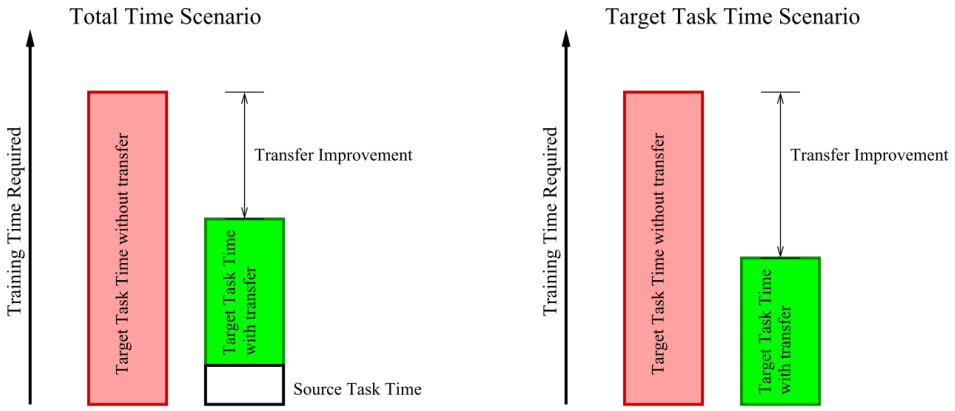


Figure 2.4: Successful transfer learning may be able to reduce the total training time (on the left). Alternatively, the source task time could be treated as a sunk cost (on the right) [Taylor and Stone 2009].

Taylor and Stone [2007] defined a few metrics to measure the benefits of transfer,

1. *Jumpstart*: The initial performance of the agent for a task could be improved by transfer from previously solved tasks.

2. *Asymptotic Performance*: The final performance of the agent for a task could be improved by transfer.

3. *Total Reward*: The total reward accumulated by an agent could be improved by transfer. The total reward accumulated is the area under the learning curve.

4. *Transfer Ratio*: The ratio of the total reward accumulated by the agent with transfer and without transfer.

5. *Time to Threshold*: The learning time needed by the agent to achieve a pre-specified performance level could be reduced by transfer.

Metrics one to four are most appropriate in the fully autonomous techniques as the learning of any source task's sunk cost is not included. Using the metric five accounts for the time spent learning the source tasks. Figure 2.5 provides a visual of these metrics.

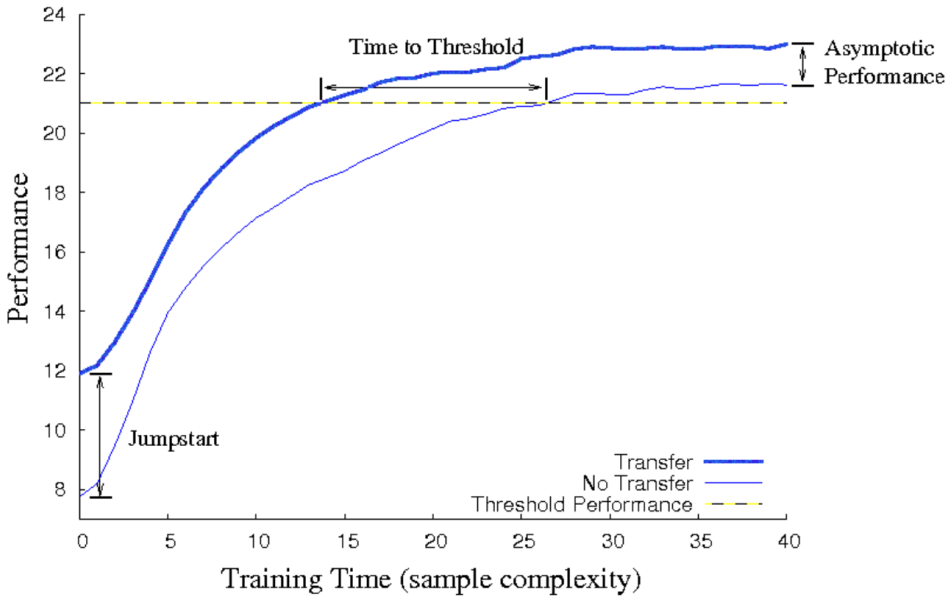


Figure 2.5: This graph show benefits to the jumpstart, asymptotic performance, time to threshold, and total reward, which is the area under the learning curve (AUC) [Taylor and Stone 2007].

Chapter 3

Building a Library of Options using a Curriculum

3.1 Introduction

The aim of this work is to develop a system capable of autonomously building a library of usable options. When viewed individually, each section described in the background contains pitfalls, but when pieced together, they can be used to develop an effective training strategy that leverages the strengths of each concept. Traditional RL methods are capable of solving the tasks but produce long sequences of actions, which can be alleviated through the use of options. To avoid having to manually hand-craft these options, which can be a timeous process, we can perform autonomous option discovery. The NPBRS method (see section 2.4) can successfully segment expert demonstrations into component skill policies. To be able to efficiently use these policies as options, we present a method to learn the initiation sets and the termination conditions. In order to build a library of usable options, we need to be able to extract different skills from separate tasks. Curriculum learning provides us with a method to relate these tasks so that we can solve more complicated problems, and in turn learn more complex options. We have created a training strategy to achieve this autonomously.

In section 3.2, we provide the overall process of the training strategy. In section 3.3, we discuss the curriculum of tasks in detail. The training strategy consists of two steps, for each task, the first is to solve the task (section 3.4.1) and the second is to extract useful options (section 3.4.2 & 3.4.3).

3.2 Training Strategy

Given a curriculum of tasks, the goal is to build a library of useful options which are capable of solving these tasks. Using the curriculum, we solve each task in order. Once a task has been solved, we extract useful options which are leveraged in the following tasks; this is a form of knowledge transfer. In this way, we can extend and refine our library so that it progressively contains more general and usable options.

Taking each task from the curriculum in order, we proceed to solve the task using previously discovered options. Once the task has been solved, we can generate a set of expert trajectories using the optimal policy that we have learned for the given task. These trajectories are then combined with the collection of trajectories generated from previous tasks. This is used as input into the NPBRS method, which segments and returns the skill policy for the corresponding collection of segments. Then for each collection, we learn the initiation set and termination condition, which, together with the corresponding skill policy, forms an option. These options are combined with our previous options set to update the library of options. Refer to figure 3.1 for a visual model of the process, while algorithm 4 shows a detailed procedure.

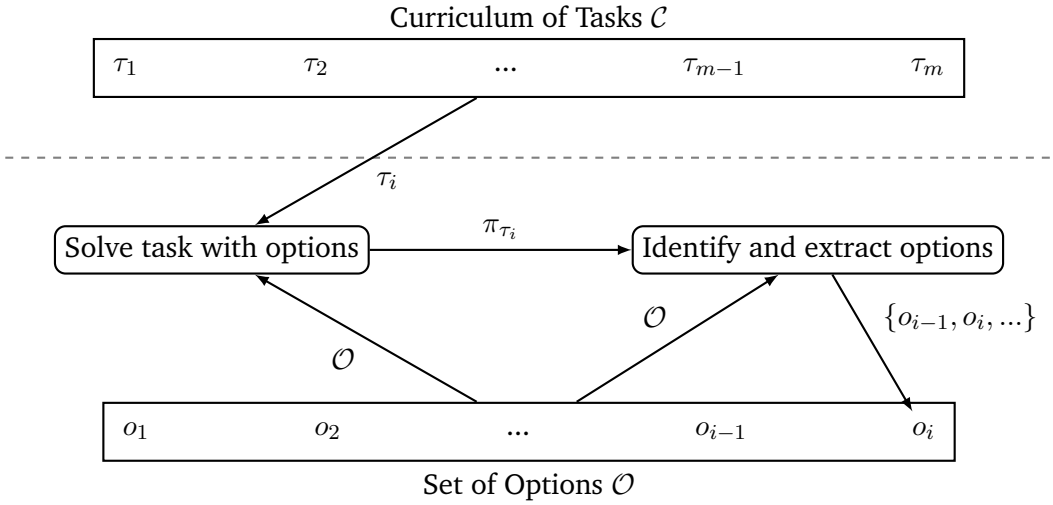


Figure 3.1: Progressively Building a Library of Options: 1) Select task τ_i from curriculum. 2) Solve Task τ_i using set of options $\mathcal{O}$. 3) Identify and extract possible options present in policy π_{τ_i}. 4) Add newly extracted options $\{o_{i-1}, o_i, ...\}$ to set $\mathcal{O}$.

Algorithm 4: Progressively Building a Library of Options

Input: Curriculum of Tasks $\mathcal{C}$

Result: Library of Options $\mathcal{O}$

for *task* τ_i *in Curriculum* $\mathcal{C}$ **do**

$\pi_{\tau_i} \leftarrow$ solve task τ_i with $\mathcal{O}$

$D^{\tau_i} \leftarrow$ generate trajectories using policy π_{τ_i}

$D \leftarrow$ append D^{τ_i} to all set of trajectories D

$z_t^{(i)}, Q^{z_t^{(i)}} \leftarrow$ perform NPBRS on trajectories D, with $\mathcal{O}$

$\mathcal{O} \leftarrow$ compute initiation set and termination condition for segmentations $z_t^{(i)}$ and corresponding skill policy $Q^{z_t^{(i)}}$

end

3.3 Structuring the curriculum

Here we focus solely on the development of the curriculum. We formally define the curriculum as a sequence of tasks $[\tau_1, \tau_2, \cdots, \tau_m]$ where the tasks are ordered from simple to more complex tasks. Ng *et al.* [2000] states that the reward function is the most succinct and transferable definition of a task; hence each task will have a different reward function. In other words, the tasks will share a common domain $\mu = MDP/R$, where domain μ encompasses the state space S, action space A and transition probability $T(s, a, s')$ that is constant among all the tasks. We can define a task $\tau_i = (\mu, R_{\tau_i})$, where R_{τ_i} is the reward function of task τ_i. This allows the agent's environment representation to be the same from task to task. This makes it easier to transfer knowledge between tasks.

We then need to define a *target task*, which is the challenging complex task we want the agent to ultimately be able to solve. The idea of this task is to provide direction for building a library of options. For example, the *target task* may be to *"prepare a cup of coffee"*. This is a difficult task, as it involves many different sub-tasks.

The first step is to generate a set of tasks that are related to our defined *target task*. For example, tasks such as *"preparing a cup of sugar milk"* and *"preparing a glass of water"*. These tasks should have some skill overlap with the *target task*. The set of tasks must share a common domain μ, and the only differences between the tasks are the reward functions R_{τ_i}.

The next step is to arrange the set of tasks into a sequence ordered from simple to more complex tasks, such that the agent can leverage previously acquired options. For example, learning to *"prepare a glass of sugar water"* can use the options extracted from the task of *"preparing a glass of water"*. This sequence of tasks ordered from simple to more complex tasks forms the curricu-

lum. The idea of the curriculum is to introduce a new concept with each task, where each task increases in complexity.

The automation of these two steps are an active area in curriculum learning [Narvekar *et al.* 2017; Forestier *et al.* 2017; Held *et al.* 2017]. For this research, we define the curriculum manually and instead focus on the mechanisms by which the agent can strategically make use of the curriculum.

3.4 Progressing through the curriculum

The agent progresses through the curriculum task by task, where for each task, the agent needs to solve the task, and after that, identify and extract useful options to transfer between tasks.

3.4.1 Solving the task

Given the structured curriculum of tasks, as discussed in section 3.3, the process of selecting a task is simple, as the tasks are in order, by construction. For each task in the curriculum, the first part of the process is to solve the task τ_i.

We use a library of options $\mathcal{O}$ to transfer knowledge amongst tasks. The task can be solved using a reinforcement learning algorithm which can learn using the set of previously acquired options $\mathcal{O}$, as well as the primitive actions space $\mathcal{A}$ [Sutton *et al.* 1999; Bacon *et al.* 2017]. In other words, the RL agent needs to learn how the current task τ_i is related to the previous tasks $[\tau_1, \cdots, \tau_{i-1}]$ through the options $\mathcal{O}$. The particular case of the first task τ_1 does not have any prior knowledge to leverage, since the curriculum of tasks is ordered from simple to more complex tasks; we can assume a straightforward reinforcement learning algorithm will be able to solve the first task, within a reasonable time and without any options.

In order to accumulate a library of options $\mathcal{O}$, we need to identify and extract useful options from the tasks. We discuss this process in section 3.4.2.

3.4.2 Identify and Extract option's policy: Segmentation

Once the task τ_i has been solved, the next step is to identify and extract useful options from the task τ_i. We use *Non-parametric Bayesian Reward Segmentation* (NPBRS) [Ranchod *et al.* 2015] as the segmentation method to identify possible component skill policies, as discussed in section 2.4. NPBRS requires expert demonstrations (trajectories) of the task τ_i. As discussed in section 2.4, this requires a human designer to produce optimal trajectories, which is often a non-trivial

task. We overcome this problem by using the agent's optimal policy π_{τ_i} to generate the trajectories D_{τ_i}, removing the need for human demonstrations. NPBRS is then used to segment the trajectories into trajectory fragments which are grouped according to the underlying skill policy. After NPBRS has extracted a set of skill policies (one per segment), we have to transform each skill policy into an option. We discuss this process in detail in section 3.4.3.

NPBRS is typically performed on a fixed set of provided trajectories. A naive approach to applying it to a curriculum of tasks would be to treat each task in the curriculum separately. As NPBRS discovers and extract skill policies from the task τ_i, it does so independently of the library of options $\mathcal{O}$, which were acquired from the range of tasks $[\tau_1, \cdots, \tau_{i-1}]$. Hence NPBRS rediscovers marginally different skill policies and does not generalise these skill policies across a set of tasks. This is a waste of extra computation time, which also may cause the library of options $\mathcal{O}$ to contain redundant skill policies.

In order to take advantage of the curriculum, a mechanism is required to transfer skill policies between the tasks. This allows an agent to build up a library of options $\mathcal{O}$ while progressing through the curriculum of tasks.

In order to facilitate this, we modify the way NPBRS is initialised. In the original algorithm, the set of skill policies is initially populated by a single skill policy, which represents a global task policy. This global task policy is generated by performing inverse reinforcement learning on the full, unsegmented set of input trajectories. We modify this step by setting the initial set of skill policies in task τ_i to the skill policies ($Q^{z_t^{(i)}}$) extracted from the range of tasks $[\tau_1, \cdots, \tau_{i-1}]$.

The trajectories D_{τ_i} of the current task τ_i are added to the set of trajectories of all previous tasks $[D_{\tau_1}, \cdots, D_{\tau_{i-1}}]$. The previous trajectories then retain their existing segments and skill policy assignments ($z_t^{(i)}$ and $Q^{z_t^{(i)}}$ $\forall i$ in equation 2.16), and new segmentation and skill policy allocation are performed on the new trajectories. This allows for segments, and skill policies to be shared across tasks, allowing IRL to find more general reward functions and in turn, more general skill policies.

Due to the lack of an initiation set and a termination condition for the skill policy, the skill policy can be not executed in a *call-and-return* manner. Instead, the agent will have to select an action or skill policy at each time-step. Hence this might not provide a speed-up during learning. Another issue is that as the number of skill policies increases the number of possible choices at each time-step increases. In order to alleviate this problem and make use of these skill policies, we need to learn an initiation set and a termination condition for each skill policy.

3.4.3 Learning where to Initiate and Terminate

Given the set of segmented trajectories produced by NPBRS, as discussed in section 3.4.2, we need to generate estimates for the initiation set I and termination condition β for an option o. We phrase this as an anomaly detection problem to classify the final trajectory state as a termination state or a non-termination state, and similarly classify start states as to whether they are initiation states or not [Cockcroft *et al.* 2020].

Given a set of trajectories, we can use the OC-SVM (section 2.6) to define separate decision boundaries, using the trajectory's start state and end state as the training data. We then consider any state which lies within the boundaries learned during training on the start states as an initiation state and any state which lies within the boundaries learned during training on the end states as a termination state.

It is important to note that while the end states are classified as termination states, options require a termination condition β. Thus, we use the set of termination states to generate a probability of being a termination state for any particular state and use this as the option termination condition β. We can form an option $o = (I, \pi, \beta)$, by combining the initiation set I and termination condition β, which we have obtained along with the corresponding skill-policy π.

3.5 Conclusion

We have proposed a method to build a library of options autonomously. With the aid of a curriculum, we are then able to build a library of useful options to transfer knowledge from basic tasks to the complex task. This was achieved by adapting the existing segmentation method NPBRS to operate in an iterative manner and along with a method to learn the initiation set and termination condition, we were able to transform the skill policy into a useful option. With this process, we allowed the options to be autonomously refined and more general, thus allowing the options in the library to be more robustly transferable to other tasks.

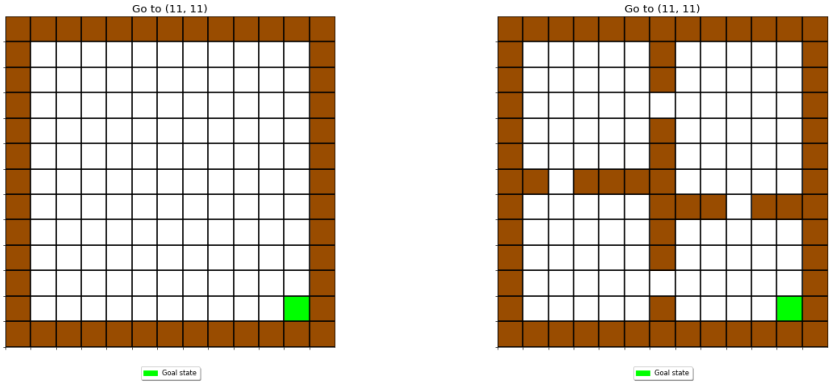
Chapter 4

Experiments

In this chapter we show four experiments. First, we investigate if it is possible to recover the underlying reward function given the policy or demonstrations. Next we demonstrate the benefits of learning with options after which we show the process of extracting these options automatically. Finally we show that we can combine these methods to solve a curriculum of tasks, while taking advantage of the knowledge gained from previously solved tasks. We first introduce the types of domains we use for the experiments, in section 4.1.

4.1 Environments

Initially we begin with a discrete *grid-world* environment of size 11 by 11, as seen in figure 4.1a. There are four navigation actions available to the agent. These are *Up*, *Right*, *Down* and *Left*. The environment can be easily navigated by the agent and so to increase the complexity of navigation, we additionally make use of the *four-rooms* environment [Sutton *et al.* 1999]. This environment splits the *grid-world* into four rooms separated by walls and connected by hall-ways, as seen in figure 4.1b.



(a) *grid-world* domain (b) *four-rooms* domain

Figure 4.1: Grid world domains with the goal being in the *bottom-right*

In order to allow for option transfer, we need a environment in which an (exact) individual option is applicable in multiple situations, in the same task. We thus introduce a mini-game which can, in principle, be solved by using an individual option. We call this environment *code-game-world*.

We achieve this by modifying the *four-rooms* environment. We include a total of three *code-game* states, in the *top-left*, *top-right* and *bottom-left* corners, as seen in figure 4.2a. In this environment the agent has four more actions, making a total of eight actions. This consists of the original four navigation actions along with new actions to enter/exit a *code-game*, *input-A*, *input-B* and *input-C*, described below.

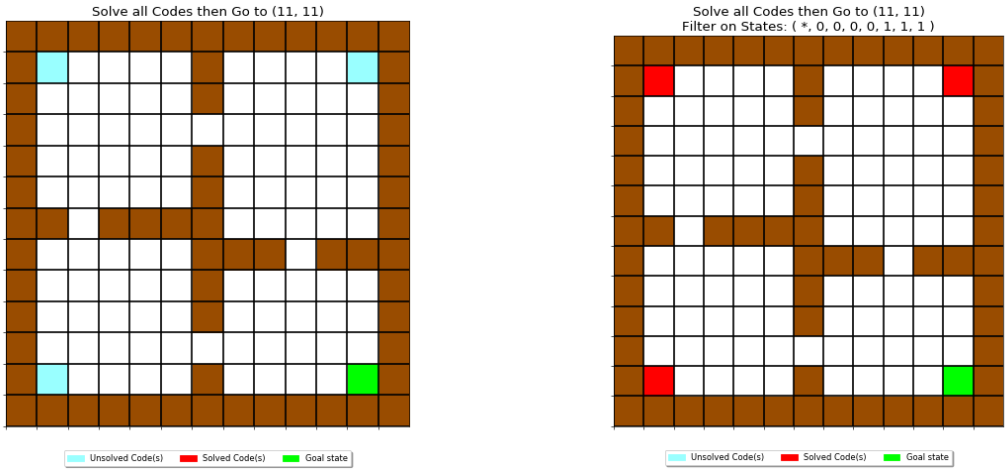
Each *code-game* state is a separate mini-game (shown in figure 4.3) where the agent has to solve the password, consisting of three characters of any permutation of the letters A, B, C. The password is the same for all *code-game* states. For example, if the password is B-C-A then the sequence of actions to solve this *code-game* is as follows,

$$\textit{enter} \rightarrow \textit{input-B} \rightarrow \textit{input-C} \rightarrow \textit{input-A} \rightarrow \textit{exit}$$

Due to the fact that we wish to model this environment as an MDP we need to preserve the Markovian property. Hence when the agent enters a *code-game* the observation has to reflect this to the agent, and the same goes for inputting password characters and completion of *code-games*. This is achieved by appending boolean indicators to the agent's observation vector, as seen below:

- *position* $\leftarrow$ the position on grid
- *inCodeGame* $\leftarrow 1$ if in a *code-game* otherwise 0
- *pwdChar-1*, *pwdChar-2*, *pwdChar-3* $\leftarrow 1$ if the correct input was provided
- *completedCodeGame-1*, *completedCodeGame-2*, *completedCodeGame-3* $\leftarrow$ this indicates if the *code-game* is solved.

Figure 4.2b shows the state space when all the *code-games* are solved, after which the agent is required to navigate to a specific position, which in this case the position is the *bottom-right* corner.



(a) Three unsolved *code-game* states

(b) All three *code-games* are solved

Figure 4.2: *code-game-world*

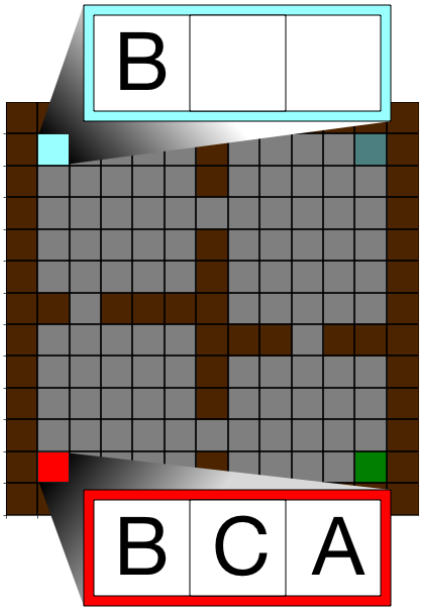


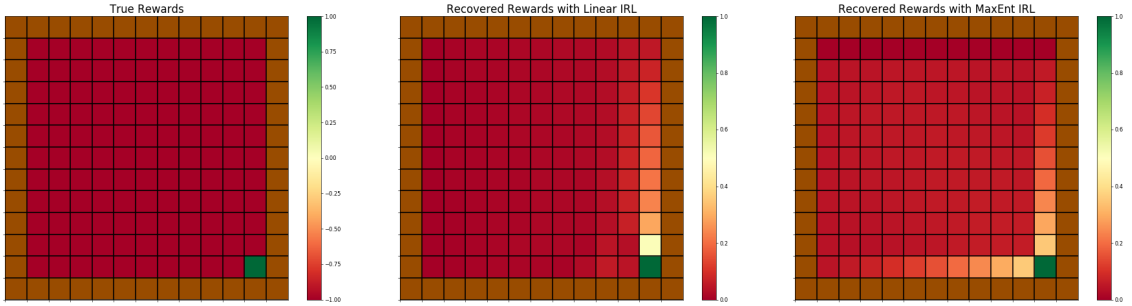
Figure 4.3: *Code-games*, the solved *code-game-3* (red) with the password B-C-A, while *code-game-1* (cyan) is in the process of being solved.

4.2 Inferring the reward function

The aim of this experiment is to show that using inverse reinforcement learning algorithms, we are able to recover the reward function. We use the *grid-world* environment to compare the Linear IRL algorithm [Ng *et al.* 2000] against the Maximum Entropy (MaxEnt) IRL algorithm [Ziebart *et al.* 2008]. For the Linear IRL algorithm, the optimal policy π is provided as input along with the discount factor $\gamma = 0.5$. For the MaxEnt algorithm, we provide 500 trajectories as input, which are generated using the optimal policy π, along with the learning rate $\alpha = 0.2$

and threshold $\delta = 0.01$.

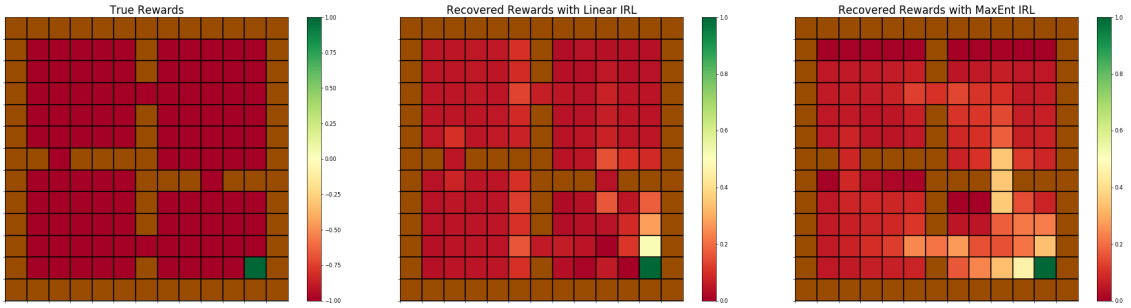
Figure 4.4 shows that the Linear IRL algorithm and the MaxEnt IRL algorithm are both successful in the recovery of the underlying reward function from the provided policy and trajectories respectively. We use Value-iteration to solve environments with true rewards and the recovered rewards. The learned state-value function and policy for each reward functions are shown in appendix A.1.



(a) (b) (c)

Figure 4.4: Rewards in the *grid-world* environment with goal being the *bottom-right* corner. (a) True rewards, (b) Recovered rewards using Linear IRL algorithm and (c) Recovered rewards using the MaxEnt IRL algorithm.

It can be seen that the goal state is clearly the highest reward value, for both Linear IRL and MaxEnt IRL. In comparison to the true reward, the Linear IRL and MaxEnt IRL methods show path preference of policy and trajectories respectively, by also assigning them a non-zero reward. We investigate this fact when considering environments that contain bottleneck states. Figure 4.5 shows that MaxEnt IRL can identify these bottleneck states in the *four-rooms* environment, which are the *hall-ways*. These bottleneck states can be useful in identifying possible sub-goals.

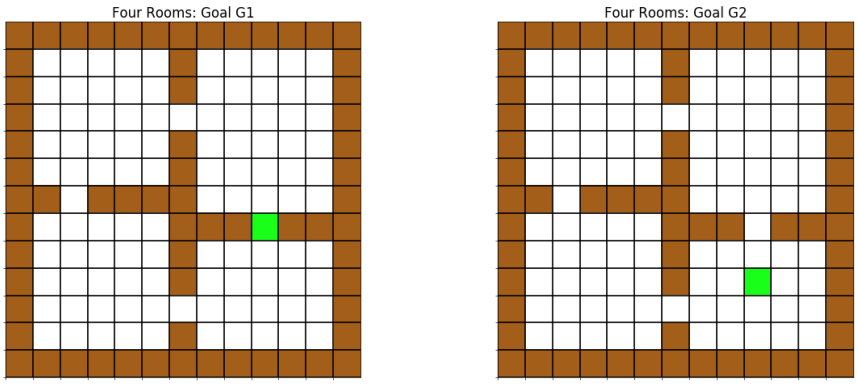


(a) (b) (c)

Figure 4.5: Rewards in the *four rooms* environment with goal being the *bottom-right* corner. (a) True rewards, (b) Recovered rewards using Linear IRL algorithm and (c) Recovered rewards using the MaxEnt IRL algorithm. The learned state-value function and policy for each reward function are shown in appendix A.2.

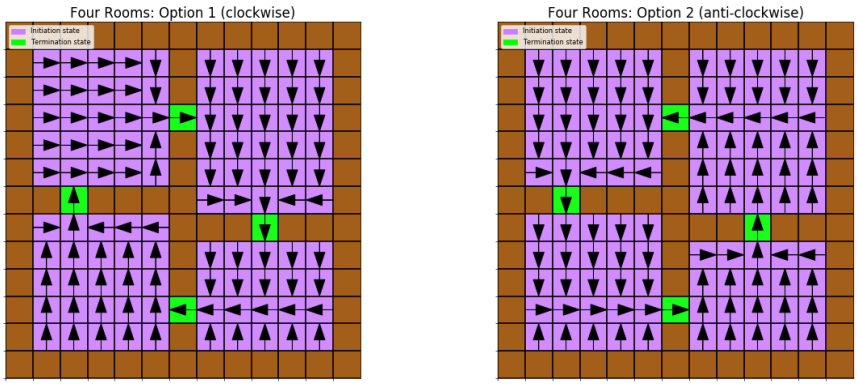
4.3 Learning with Options

Once again consider the *four-rooms* environment, with the goals at *G1* and *G2* as seen in figures 4.6a and 4.6b respectively. We provide some hand-crafted options, that take the agent to the clockwise *hall-way* or anti-clockwise *hall-way* from each room, as seen in figure 4.7. All states in the initiation set I are highlighted in purple and the termination condition $\beta(s) = 1$ is highlighted in green. The arrows indicate the action a at each state s, selected by the option's policy π.



(a) Goal G1 (b) Goal G2

Figure 4.6: *four-rooms* environment, with goals *G1* and *G2*

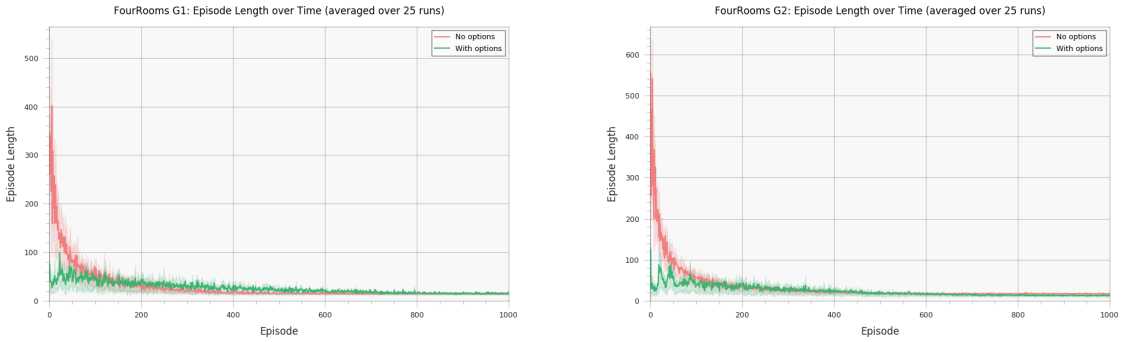


(a) Option 1 (clockwise) (b) Option 2 (anti-clockwise)

Figure 4.7: Hand crafted options for the *four-rooms* environment.

Using the SMDP Q-Learning algorithm we recreate the experiments performed in *four-rooms* [Sutton *et al.* 1999] to show the benefits of learning with options. For the SMDP Q-Learning algorithm, we use the following parameters: a discount factor $\gamma = 0.9$, a learning rate $\alpha = 0.125$ and with an ϵ-greedy action selection strategy ($\epsilon = 0.1$).

Goal *G1* can be solved with only the use of the options, where as goal *G2* requires the agent to use primitive actions to get from the *hall-way* to *G2*. The learning curves can be seen in figure 4.8. The use of options provides a clear *jumpstart* during learning and converges faster compared to when using only the primitive actions.



(a) Goal: G1 (b) Goal: G2

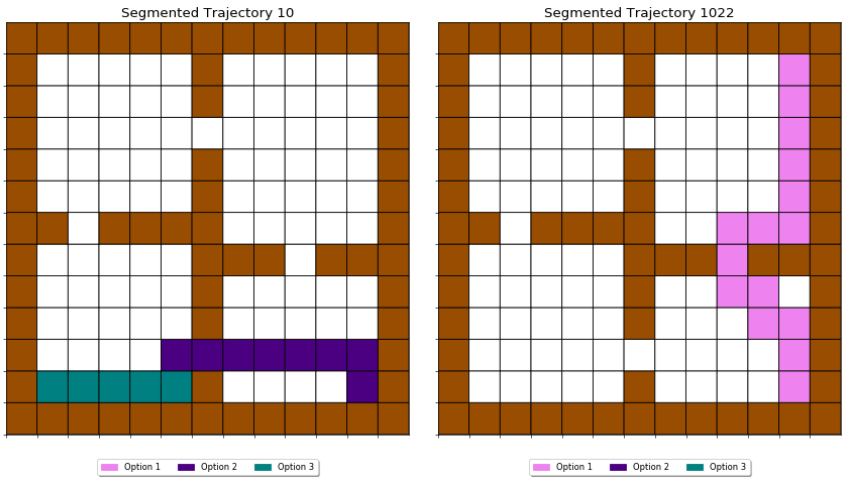
Figure 4.8: Number of steps per episode. The accompanying reward learning curves are provided in appendices B.1 and B.2 respectively.

4.4 Option discovery

As seen in section 4.2, we were able to recover the global reward function. Given that this reward function is for the entire task, it could be less transferable to related tasks, which contain

similar sub-tasks. To allow for more robust transfer, we would like to identify and extract skills for smaller sub-tasks. We make use of the NPBRS segmentation method to decompose these tasks into sub-tasks. Given a set of trajectories, generated by an optimal policy, NPBRS is able to segment these trajectories into skills segments, along with the corresponding skill policy. Figure 4.9 shows a few examples of segmented trajectories. We can see in figure 4.9a, the trajectory is segmented into two sub-tasks. The first sub-task is to go to the hall-way, and then the second sub-task is to go from the hall-way to the goal.

For this experiment, we define two tasks in the *four-rooms* environment. Task 1 is to go to the *bottom-left* corner, and Task 2 is to go to the *top-right* corner. Both tasks have the same starting location, which is the *bottom-right*. We use the optimal policy to generate a 1000 trajectories for each task. The NPBRS method segments the trajectories and computes the likelihood of the segment with the MaxEnt IRL algorithm. We use the following parameters for the IRL algorithm, a learning rate starting at $\alpha = 0.2$ with linear decay and threshold error $\delta = 0.01$.

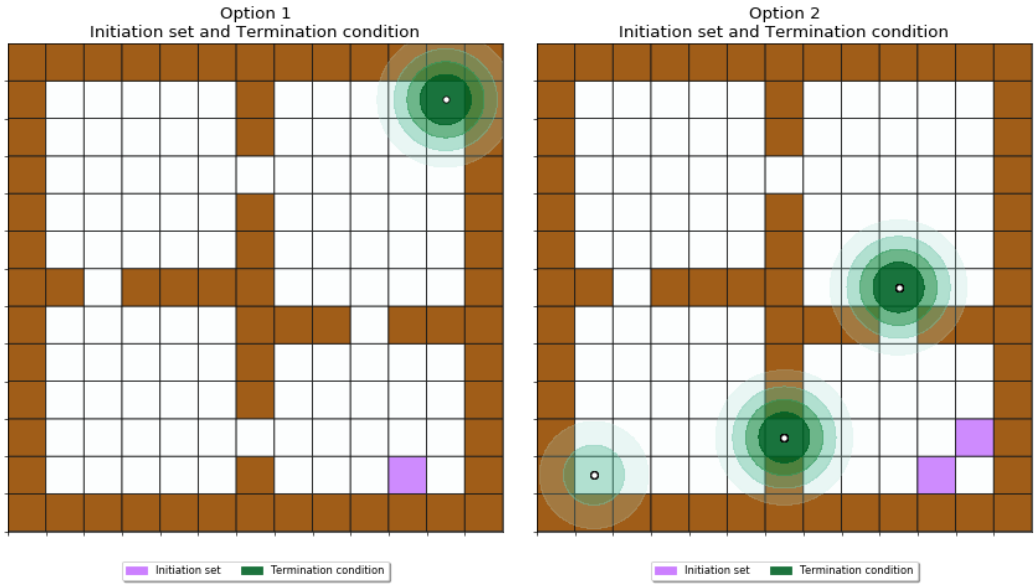


(a) Trajectory 10 from Task 1 (b) Trajectory 1022 from Task 2

Figure 4.9: Segmented Trajectories

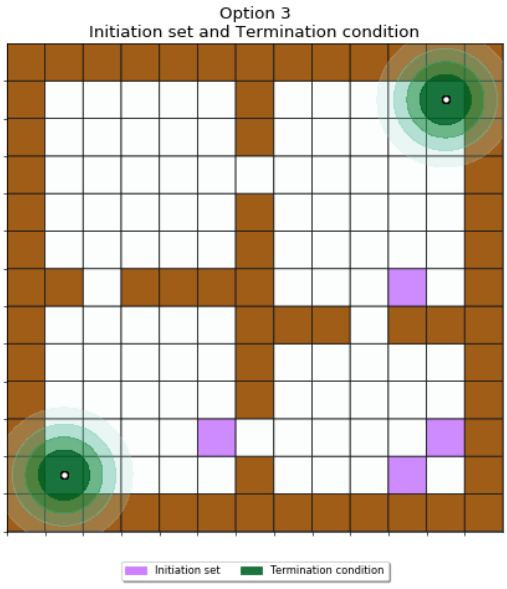
The NPBRS algorithm returns only the skill policy for each segment, but in order to define the skill as an option we also require an initiation set and a termination condition. We make use of the one-class SVM to learn the initiation set and the termination condition for each skill (section 3.4.3). Given the segmented trajectories, the SVM is able to learn a bound around the start states, and all points which lay within this bound are classified as being in the initiation set. Similarly for the end states we can define a probability of being in this set, giving us the termination condition. Figure 4.10 shows the initiation set as states which are highlighted in purple and the termination condition, with the probability of terminating shown by the green concentric circles. Given the skill policy, the initiation set, and the termination condition of each

segment, we can define each skill as an option. We can see where the option's policy can initiate from and where it will terminate, which explains the segmentation of the trajectories above.



(a) Option 1

(b) Option 2



(c) Option 3

Figure 4.10: Options after Task 2

4.5 Building a Library of Options, with a Curriculum

As discussed in section 3.3, we need to structure a curriculum manually. We use the *code-game-world* environment for each task in the curriculum. For a fixed code, there are a total of eight tasks in the curriculum. The curriculum of tasks is:

- Task 1 - Navigate to *bottom-left* corner

- Task 2 - Navigate to *top-right* corner
- Task 3 - Navigate to *top-left* corner
- Task 4 - Navigate to *bottom-right* corner
- Task 5 - Navigate to *bottom-left* corner and Solve *code-game-3*
- Task 6 - Navigate to *top-right* corner and Solve *code-game-2*
- Task 7 - Navigate to *top-left* corner and Solve *code-game-1*
- Task 8 - Solve all three *code-games* and then navigate to *bottom-right* corner

The first fours tasks are simple navigation tasks, to go to a specific location. Task 5 to Task 7 build on the navigation tasks with the addition of a single *code-game* for each task. Task 8 is the complex target task which is a combination of previous tasks. Task 8 contains a total of three *code-games* at different locations.

A random start state could provide an advantage in terms of exploration. For a more controlled comparison, we eliminate this possible advantage by using a fixed start state-space (i.e. all the start states contain the same *position* within the observation vector). A visual and the details of the state-space, for each task, are provided in appendix C.1.

4.5.1 Progressing through the curriculum

In this experiment, we make use of our proposed training strategy along with the curriculum of tasks defined above. The training strategy progressively solves the curriculum task by task. For each task, there are two steps, as described in section 3.4. The first step is to solve the task and in the second we discover useful options to transfer between tasks.

For comparison purposes, we solve each task with three different approaches. We make use of the Q-Learning algorithm to solve the task without any transfer (i.e. does not use the library of options). We use this as the benchmark (red). The other two approaches both make use of the SMDP Q-Learning algorithm, where the difference is in the library of options $\mathcal{O}$. We want to compare how well the agent performs with the library of options constructed while learning to solve previous tasks $\mathcal{O}_{\tau_{i-1}}$ (blue), against the library of options including the options extracted from the current task $\mathcal{O}_{\tau_i}$ (green). The options extracted from the current task should provide an advantage over options from the previous tasks only. Due to the increasing number of options in the library, it could also have a negative effect as the agent will need to explore more options. For Task 1, the library of options from the previous task $\mathcal{O}_{\tau_{i-1}}$ is not available as it is the first task. Hence the for Task 1 results, the blue bar does not exist in figure 4.14 and 4.15.

The parameters for the SMDP Q-Learning algorithm and the Q-Learning algorithm were kept the same, which are, the discount factor $\gamma = 0.9$ and a learning rate $\alpha = 0.5$. In order to allow the agent to explore, we make use of the ϵ-greedy action selection with $\epsilon = 0.1$. For the options extraction process, we use the agent's converged policy to generate 1000 trajectories for each task, which is then passed to NPBRS. For the inverse reinforcement learning algorithm used within the NPBRS process, we make use of the MaxEnt IRL algorithm, with the parameters as follows, a learning rate starting at $\alpha = 0.2$ with linear decay and threshold error $\delta = 0.01$. For each task, we average over 10 runs for each approach.

For Task 1 the goal is to navigate to the *bottom-left* corner and the goal for Task 2 is to navigate to the *top-right* corner. It can be seen in figure 4.11, when learning with options extracted from Task 1 (blue), that the agent takes a period of time to realise that the options are not useful. This is because these options take the agent towards the *bottom-left* instead of *top-right*, as is shown by Option 2 from Task 1, in figure 4.12a.

After solving Task 2, and given the fact that options from Task 1 were not useful for the agent while solving Task 2, it is sensible for the agent to update/refine the options from Task 1. This is shown in figure 4.12. This refinement process was done when performing NPBRS with trajectories from Task 1 and Task 2. This shows that options can be generalised as more tasks are solved. When using the refined and expanded library of options after solving Task 2 (green), the learning is much better as shown in figure 4.11.

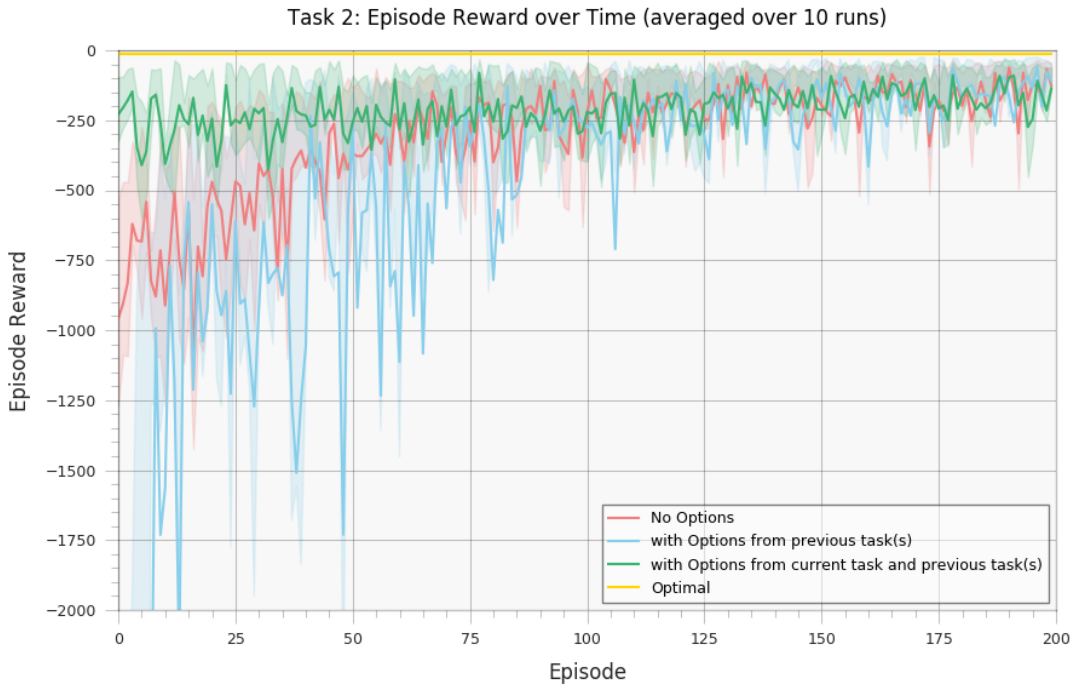
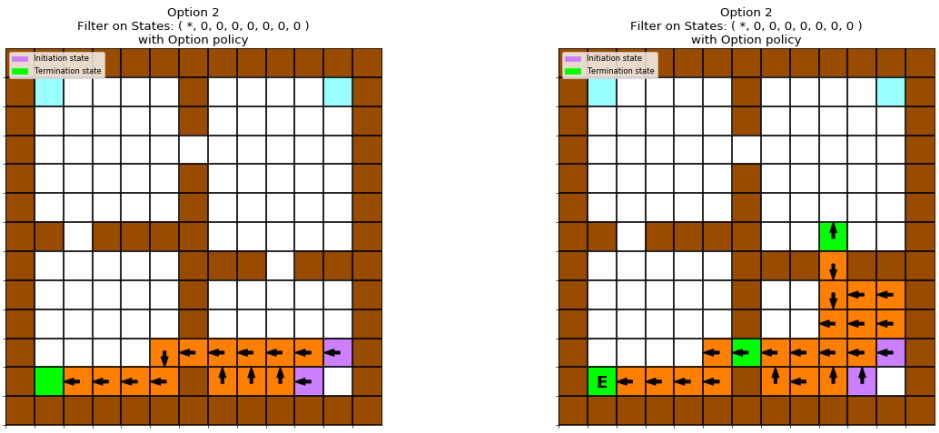


Figure 4.11: Task 2: Total reward per episode, for the first 200 episodes



(a) Option 2 after Task 1

(b) Option 2 after Task 2

Figure 4.12: Refinement of Option 2, from Task 1 to Task 2

Given a curriculum of tasks, each task can be solved independently without any mechanism to share knowledge. Our training strategy makes use of a library of options as a mechanism to transfer knowledge amongst the tasks. We treat the time of solving previous tasks as a sunk cost. Thus we only compare the time of learning to solve the target task. After progressively solving all the Tasks, from Task 1 to Task 7, our training strategy has built a comprehensive library of options to aid when solving the target task (Task 8). We evaluate our method using the transfer learning metrics defined by Taylor and Stone [2009]. These metrics are *jumpstart*, *asymptotic performance*, *total reward*, *transfer ratio* and *time to threshold*.

When learning to solve to target task (Task 8), it is clear that the library of options provide a *jumpstart*. This is shown in figure 4.13, when comparing the learning curve with no options (red) and the learning curves using options (blue and green). The library of options clearly provide an advantage in the initial performance. With regards to *asymptotic performance* the learning with options does not provide a benefit, as given with enough episodes both, with and without options, are able to converge to an optimal policy. In some cases the option's policy might be sub-optimal thus the agent will learn to select primitive actions over these options.

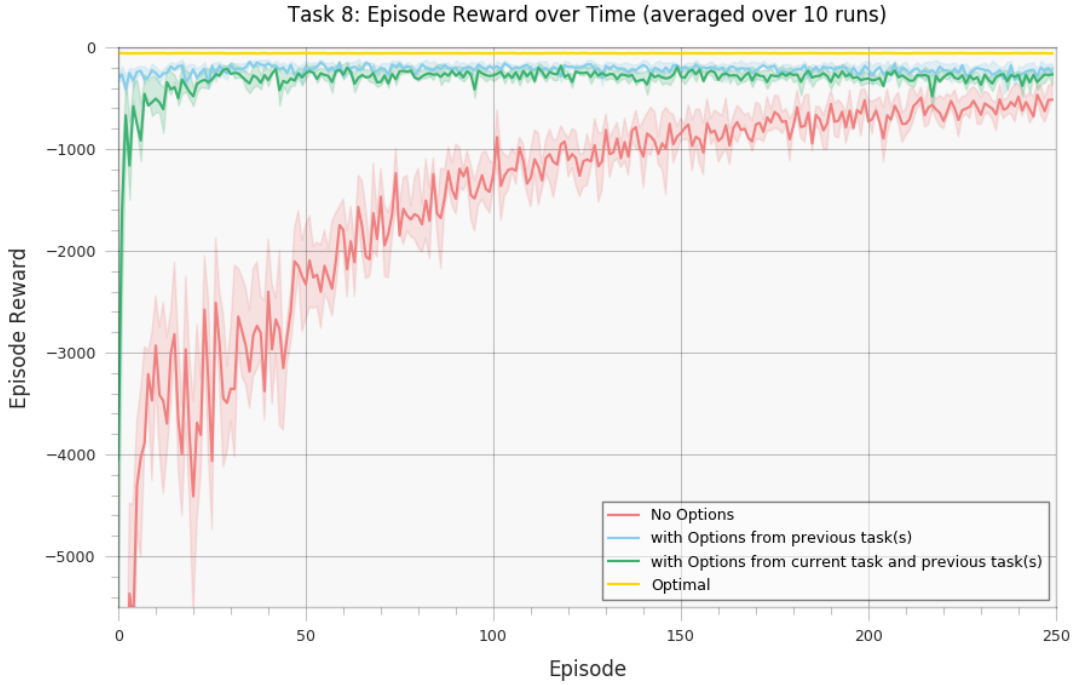


Figure 4.13: Task 8: Total reward per episode, for the first 250 episodes

The *total reward* is calculated as the area under the learning curve (AUC). For this experiment we used the trapezoidal rule when calculating the AUC. We can see that the *total reward* when learning with options is greater than when learning without options. The *transfer ratio* provides a method to give a normalised comparison of the *total reward* of different approaches. The ratio can be computed by the following:

$$AUC_{(with\_transfer)}/AUC_{(without\_transfer)}$$

In figure 4.14, we show the *transfer ratio* for each task. The Q-learning algorithm (red) is without transfer. The SMDP Q-Learning algorithm (blue & green) is with transfer. We compare with the optimal policy (gold) to provide an upper bound. If the *transfer ratio* is less than 1.0, it indicates the library of options were not beneficial when learning to solving the task. When the *transfer ratio* is greater than 1.0, we can state that the library of options is successful in transfer

and provides an advantage when learning to solving the task. This is particularly clear for the target task (Task 8), which can be viewed as a combination of the previous tasks.

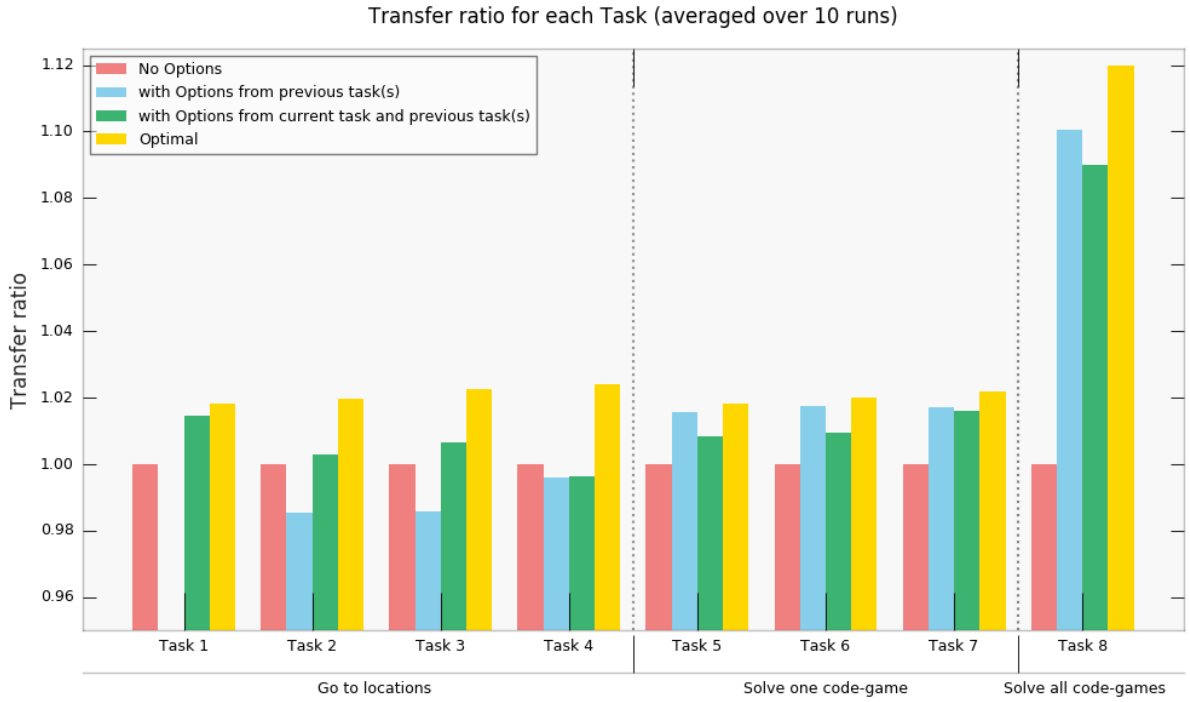


Figure 4.14: Transfer ratio for each task from the curriculum. The learning curves for each task is provided in the appendix C.2.

In order evaluate *time to threshold*, we need to determine an appropriate threshold for each task. We set the threshold to double the number of steps required to reach the goal using an optimal policy. For example, if the optimal number of steps to the goal is 11 steps, then the threshold is set to 22 steps. Figure 4.15 shows the total samples required to reach the threshold performance. If threshold performance is not met within total number of episodes, the total number of steps is taken. We can see that learning with the library of options (blue & green), the agent is more sample efficient, when the *transfer ratio* is greater than 1.0. We notice that from Task 5 to Task 8, the library of options contain useful options from the previous tasks, and this leads to a significant reduction in the *time to threshold*. We can see that for Task 5 & 6, when solving the task with the library of options from extracted from previous tasks $\mathcal{O}_{\tau_{i-1}}$ (shown in blue) has a huge improvement in sample efficiency. When learning to solve the target task (Task 8) with the library of options, we see that around 71.32% fewer steps were required to reach threshold performance (shown in figure 4.15). Tasks 2, 3 and 4 do not share common options when solving each task. The agent learns, by exploration, that the options from previous tasks (blue) are not helpful, when solving the task. In these cases, either new options are gained or refinement of options has taken place, as it is evident that the *time to threshold* has improved, when learning with the newly refined and expanded library of options (green).

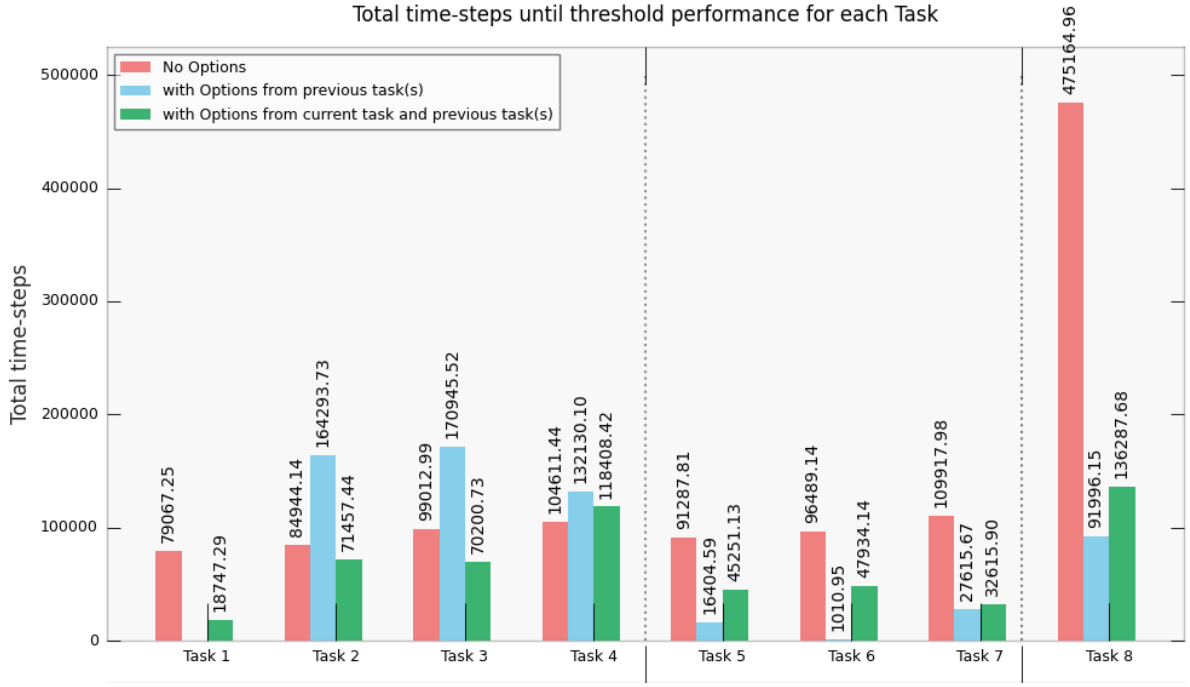


Figure 4.15: Number of samples required to reach threshold performance for each task.

4.6 Conclusion

In section 4.2, we showed that the underlying reward function can be recovered from demonstrations. In section 4.3, we explored that learning with options provide a significant boost in the initial performance. In section 4.4 we investigated the extensions we made to the NPBRS algorithm. We show that with these extensions, we are able to autonomously discover and extract options from demonstrations. Finally in section 4.5, we demonstrated our proposed training strategy is able to build a library of usable options. Based on the results it is clear that the library of options is effective as a mechanism for transfer. Our method was successful in autonomously building a library of options, as progressing through the curriculum, which provided a number of significant advantages when solving the successive task.

Chapter 5

Conclusion & Future Work

In this dissertation, we introduced a training strategy to build a library of options progressively and autonomously. This encompasses many different techniques to provide an effective strategy. We showed that it was possible to extract options from a simple task to accelerate the learning of a more complex task. The options provided a considerable *jumpstart* in the learning performance and are more sample efficient when solving the complex task. Our training strategy learns a policy for each task in the curriculum using options from previous tasks. It subsequently extracts the new options from the trajectories, generated with the policy, by using NPBRS. The curriculum by design ensured that the sequence of tasks are related, thus allowing the options to be transferable and reusable between tasks. After training on a curriculum of tasks, we showed that the library of options was beneficial when solving the complex target task, which consisted of a combination of sub-tasks similar to the previous tasks.

The NPBRS algorithm provides an advantage when extracting skill policies, due to the criteria it enforces when deciding if the trajectory segment is a viable skill. To reiterate, these properties are that the skills should be useful and reusable and that options selection are correlated (see section 2.4). We modified the NPBRS algorithm to be initialised with all previous trajectory segments and associated skill policies. With this modification, NPBRS can operate in an iterative manner. This enforces these three properties across all skill policies and segments from all task demonstrations. Our method of transforming the skill policies and corresponding segments into beneficial options was vital to make the learning with options possible [Cockcroft *et al.* 2020].

There are three criteria (see section 2.7), defined by Taylor and Stone [2009], in determining if the system is fully autonomous. Our proposed training strategy achieves two out of three. The agent is able to learn how the options from previous tasks are related to the current task. The agent learns which options are useful from the library of options, for the current task. The library of options provides an effective system of transfer between tasks. This is shown by the

significant *jumpstart* in performance. The current system requires a human designer to construct a curriculum with a sequence of tasks that iteratively uncovers skills. This is a burden on a human designer. In future work, it would be interesting to automate the curriculum development process. If a curriculum can be developed for a given target task, it would fulfil the criterion by selecting an appropriate sequence of source tasks from which to transfer. Hence with an automatic curriculum development process, our training strategy could be fully autonomous.

Another avenue of future work is to adapt the NPBRS algorithm to use variational inference (VI) methods instead of MCMC. The variational inference method would provide a principled method for checking for convergence, instead of running MCMC for a fixed amount of time or number of iterations. VI methods have displayed substantial computational gains over MCMC methods, with only a small sacrifice in terms of quality.

Angelov *et al.* [2020] present an interesting approach to composing diverse polices with varied representation. This could make it possible to share a library of options amongst the similar type of robots. Angelov *et al.* [2020] developed a Goal Score estimator which transforms the hierarchical learning problem into a planning problem. We can use the estimator along with the library of options in a multi-agent setting for an agent to plan at an abstracted level.

References

[Abbeel and Ng 2004] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[Angelov *et al.* 2020] Daniel Angelov, Yordan Hristov, Michael Burke, and Subramanian Ramamoorthy. Composing diverse policies for temporally extended tasks. *IEEE Robotics and Automation Letters*, 5(2):2658–2665, 2020.

[Argall *et al.* 2009] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[Arora and Doshi 2018] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *arXiv preprint arXiv:1806.06877*, 2018.

[Bacon *et al.* 2017] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[Barto and Mahadevan 2003] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

[Bengio *et al.* 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[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.

[Caruana 1995] Rich Caruana. Learning many related tasks at the same time with backpropagation. In *Advances in neural information processing systems*, pages 657–664, 1995.

[Coates *et al.* 2009] Adam Coates, Pieter Abbeel, and Andrew Y Ng. Apprenticeship learning for helicopter control. *Communications of the ACM*, 52(7):97–105, 2009.

[Cockcroft *et al.* 2020] Matthew Cockcroft, Shahil Mawjee, Steven James, and Pravesh Ranchod. Learning options from demonstration using skill segmentation. In *2020 International SAUPEC/RobMech/PRASA Conference*, pages 1–6. IEEE, 2020.

[Elman 1993] Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

[Fern *et al.* 2004] Alan Fern, SungWook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias. In *Advances in neural information processing systems*, pages 847–854, 2004.

[Forestier *et al.* 2017] Sébastien Forestier, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *arXiv preprint arXiv:1708.02190*, 2017.

[Fox *et al.* 2011] Emily B Fox, Erik B Sudderth, Michael I Jordan, and Alan S Willsky. Joint modeling of multiple related time series via the beta process. *arXiv preprint arXiv:1111.4226*, 2011.

[Fox 2009] Emily Beth Fox. *Bayesian nonparametric learning of complex dynamical phenomena*. PhD thesis, Massachusetts Institute of Technology, 2009.

[Hassabis *et al.* 2017] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[Hauskrecht *et al.* 2013] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas L Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. *arXiv preprint arXiv:1301.7381*, 2013.

[Held *et al.* 2017] David Held, Xinyang Geng, Carlos Florensa, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. *arXiv preprint arXiv:1705.06366*, 2017.

[Ilghami *et al.* 2005] Okhtay Ilghami, Hector Munoz-Avila, Dana S Nau, and David W Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd international conference on Machine learning*, pages 337–344, 2005.

[Jaynes 1957] Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.

[Konidaris *et al.* 2012] George Konidaris, Scott Kuindersma, Roderic Grupen, and Andrew Barto. Robot learning from demonstration by constructing skill trees. *The International Journal of Robotics Research*, 31(3):360–375, 2012.

[Narvekar *et al.* 2016] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 566–574. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[Narvekar *et al.* 2017] Sanmit Narvekar, Jivko Sinapov, and Peter Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 147, page 149, 2017.

[Ng *et al.* 2000] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670, 2000.

[Parr and Russell 1998] Ronald Edward Parr and Stuart Russell. *Hierarchical control and learning for Markov decision processes*. University of California, Berkeley Berkeley, CA, 1998.

[Puterman 1995] Martin L Puterman. Markov decision processes: Discrete stochastic dynamic programming. *Journal of the Operational Research Society*, 46(6):792–792, 1995.

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

[Ranchod 2017] Pravesh Ranchod. *Skill Discovery from Multiple Related Demonstrators*. PhD thesis, University of the Witwatersrand, Dec 2017.

[Rummery and Niranjan 1994] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[Russell 1998] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[Sanger 1994] T.D. Sanger. Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE Transactions on Robotics and Automation (Institute of Electrical and Electronics Engineers); (United States)*, 10:3(1), 6 1994.

[Schölkopf *et al.* 2001] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[Singh and Sutton 1996] Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

[Sutton and Barto 1998] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*. 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.

[Sutton 1985] Richard S Sutton. Temporal credit assignment in reinforcement learning. 1985.

[Sutton 1988] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[Taylor and Stone 2007] Matthew E Taylor and Peter Stone. Cross-domain transfer for reinforcement learning. In *Proceedings of the 24th international conference on Machine learning*, pages 879–886, 2007.

[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.

[Thrun and Schwartz 1995] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In *Advances in neural information processing systems*, pages 385–392, 1995.

[Thrun 1996] Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in neural information processing systems*, pages 640–646, 1996.

[Vapnik 2013] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[Watkins 1989] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[Wulfmeier *et al.* 2015] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.

[Ziebart *et al.* 2008] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

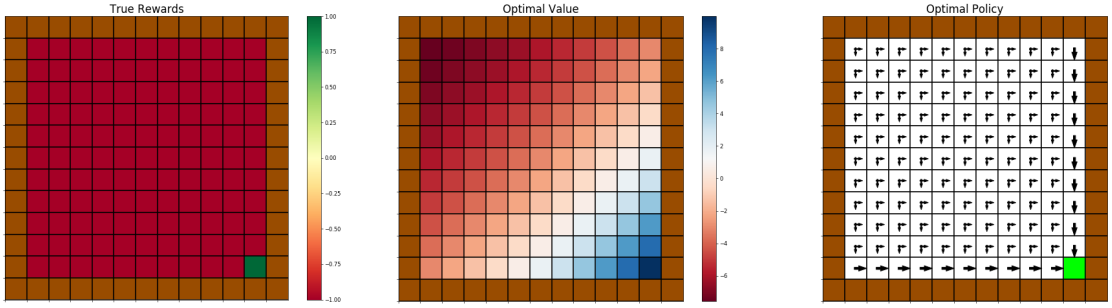
Appendices

Appendix A

Inferring the reward function

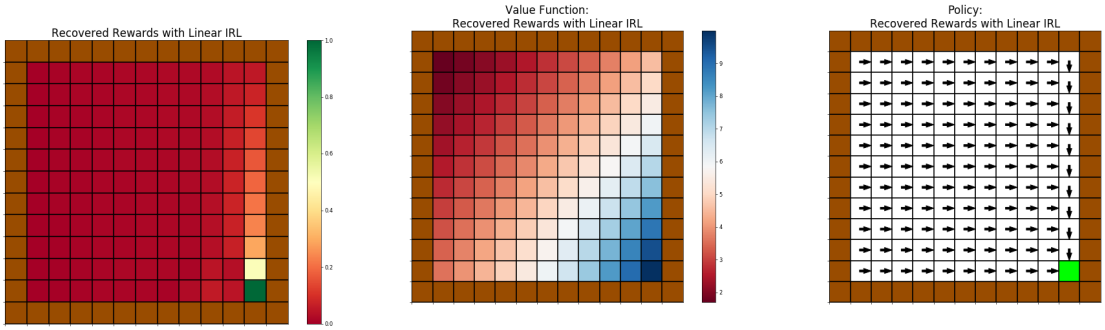
For the experiments discussed in section 4.2, we make use of the *grid-world* and the *four-rooms* environments. For both environments, a random start state is selected for each episode. We use Value-iteration to solve environments with true rewards and the recovered rewards. For the Linear IRL algorithm, the optimal policy π is provided as input along with the following parameters, discount factor $\gamma = 0.5$ and $l1 = 10$. For the MaxEnt algorithm, we provide 500 trajectories as input, which are generated using the optimal policy π, with a learning rate $\alpha = 0.2$ and threshold $\delta = 0.01$.

A.1 Grid-World Environment



(a) True Rewards (b) Optimal State-values (c) Optimal Policy

Figure A.1: Optimal state values and policy learned using true rewards, in the *grid-world* environment with the goal being the *bottom-right* corner.

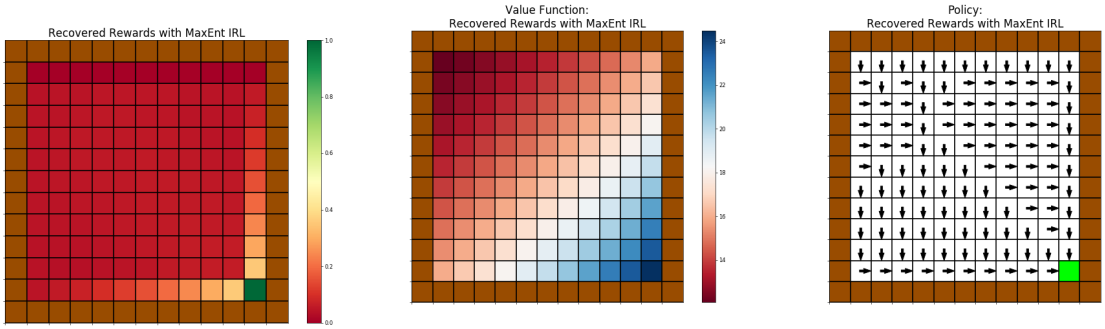


(a) Recovered Rewards

(b) State-values using recovered rewards

(c) Policy using recovered rewards

Figure A.2: State-values and policy learned using recovered rewards, in the *grid-world* environment with the goal being the *bottom-right* corner. Rewards were recovered using the Linear inverse reinforcement learning algorithm.



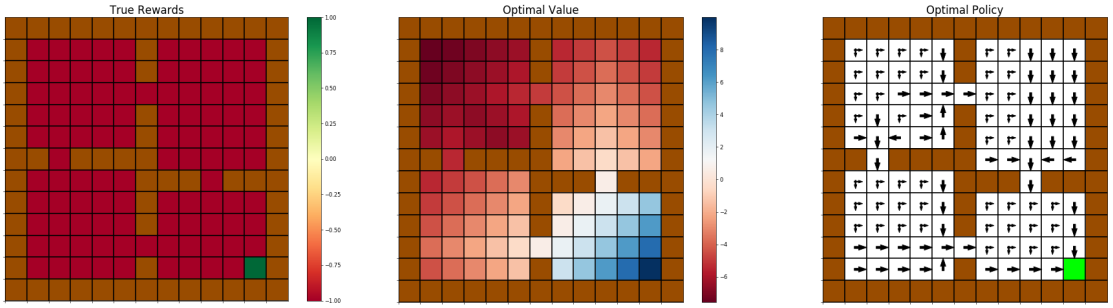
(a) Recovered Rewards

(b) State-values using recovered rewards

(c) Policy using recovered rewards

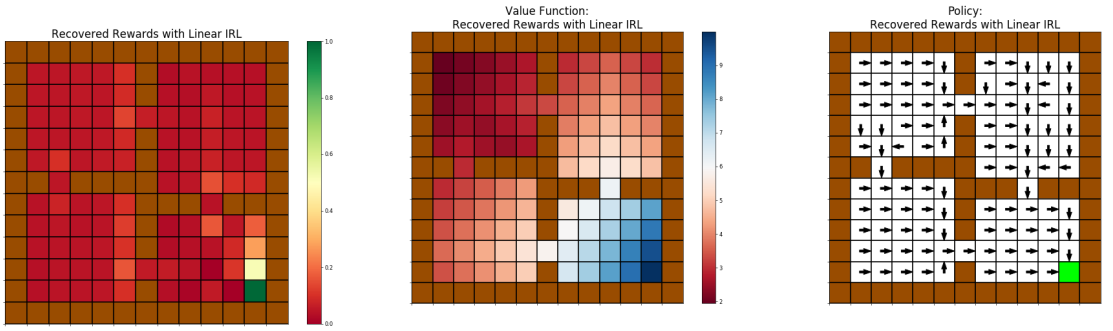
Figure A.3: State-values and policy learned using recovered rewards, in the *grid-world* environment with the goal being the *bottom-right* corner. Rewards were recovered using the Maximum Entropy inverse reinforcement learning algorithm.

A.2 Four-Rooms Environment



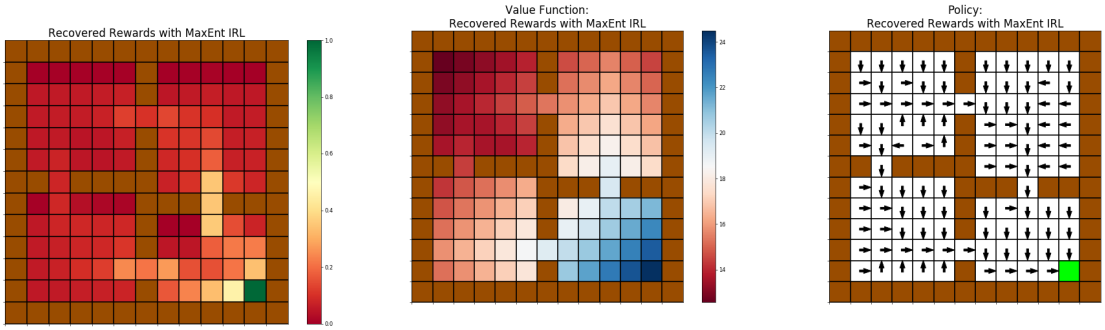
(a) True Rewards (b) Optimal State-value (c) Optimal Policy

Figure A.4: Optimal state values and policy learned using true rewards, in the *four-rooms* environment with the goal being the *bottom-right* corner.



(a) Recovered Rewards (b) State-values using recovered rewards (c) Policy using recovered rewards

Figure A.5: State-values and policy learned using recovered rewards, in the *four-rooms* environment with the goal being the *bottom-right* corner. Rewards were recovered using the Linear inverse reinforcement learning algorithm.



(a) Recovered Rewards

(b) State-values using recovered rewards

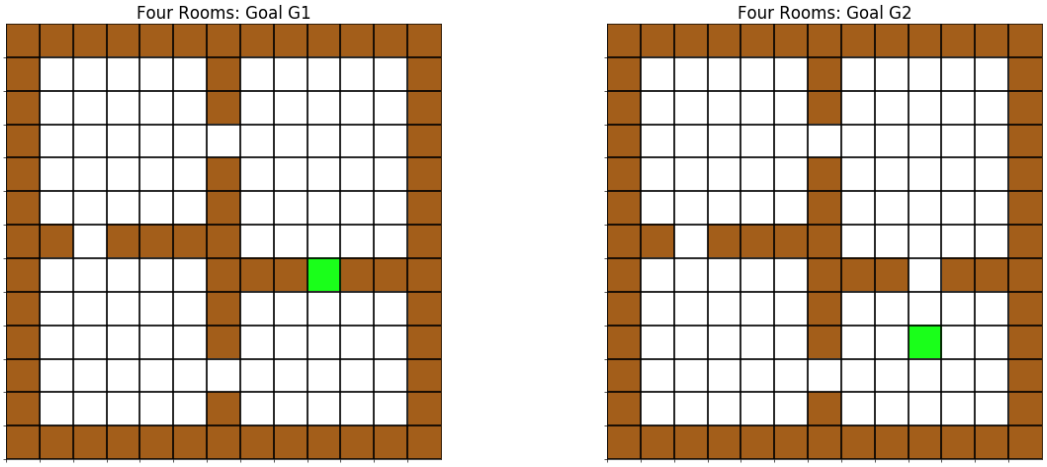
(c) Policy using recovered rewards

Figure A.6: State-values and policy learned using recovered rewards, in the *four-rooms* environment with the goal being the *bottom-right* corner. Rewards were recovered using the Maximum Entropy inverse reinforcement learning algorithm.

Appendix B

Learning with Options

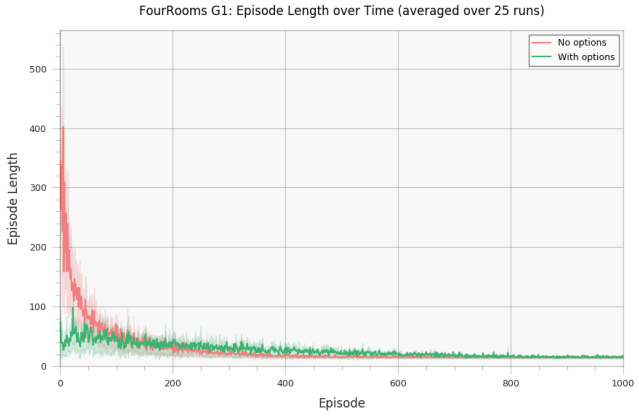
For the experiments discussed in section 4.3, we make use of the *four-rooms* environment, with a random start state selected for each episode. For the SMDP Q-Learning algorithm, we use the following parameters: a discount factor $\gamma = 0.9$, a learning rate $\alpha = 0.125$ and with an ϵ-greedy action selection strategy ($\epsilon = 0.1$).



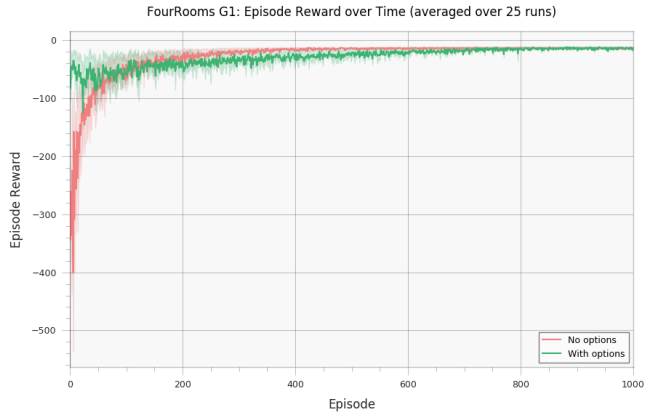
(a) Goal G1 (b) Goal G2

Figure B.1: The *four-rooms* environment with goal *G1* and *G2*

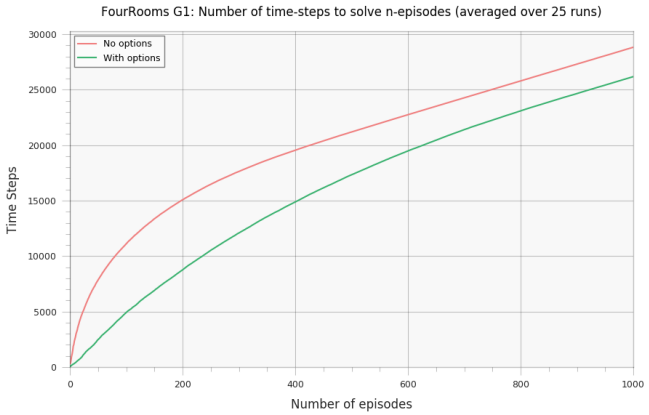
B.1 Solving Goal G1



(a) Number of steps per episode



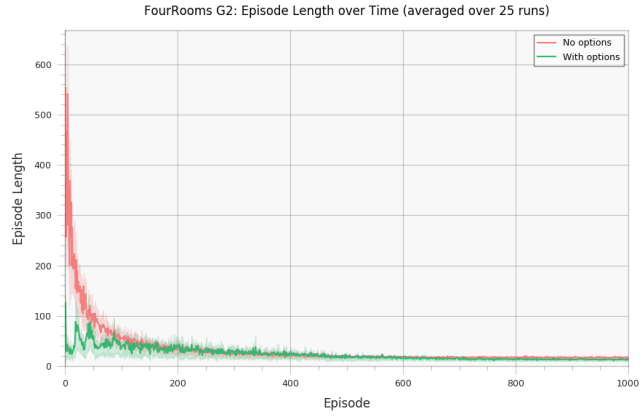
(b) Total reward per episode



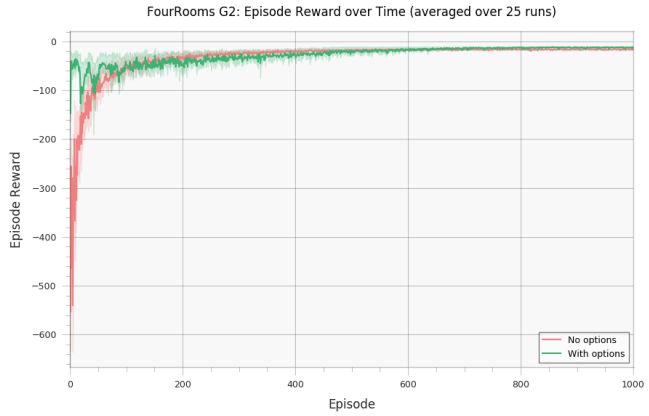
(c) Number of time-steps to solve n-episodes

Figure B.2: Solving Goal G1 with hand-crafted options.

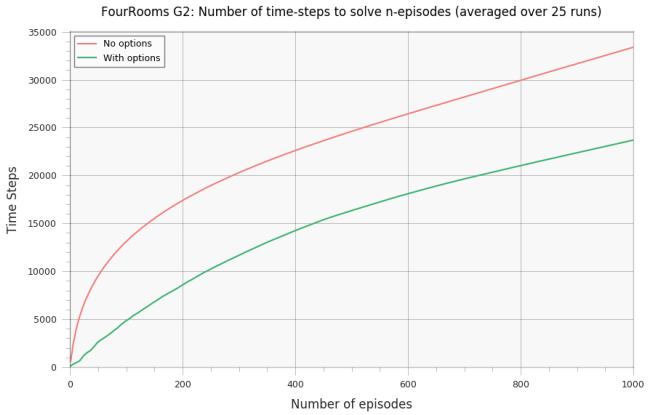
B.2 Solving Goal G2



(a) Number of steps per episode



(b) Total reward per episode



(c) Number of time-steps to solve n-episodes

Figure B.3: Solving Goal G2 with hand-crafted options.

Appendix C

Build a Library of Options, with a curriculum

C.1 Curriculum of Tasks

We use the *code-game-world* environment for each of task in the curriculum. The environment contains a total of three *code-games*, with the password of A-B-C. The agent's observation is tuple of the following:

- *position* $\leftarrow$ the position on grid
- *inCodeGame* $\leftarrow 1$ if in a *code-game* otherwise 0
- *pwdChar-1*, *pwdChar-2*, *pwdChar-3* $\leftarrow 1$ if the correct input was provided
- *completedCodeGame-1*, *completedCodeGame-2*, *completedCodeGame-3* $\leftarrow$ this indicates if the *code-game* is solved.

Task 1

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$
Goal state-space: $(\textit{bottom-left}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$
Reward: $+1$ for going to goal otherwise -1

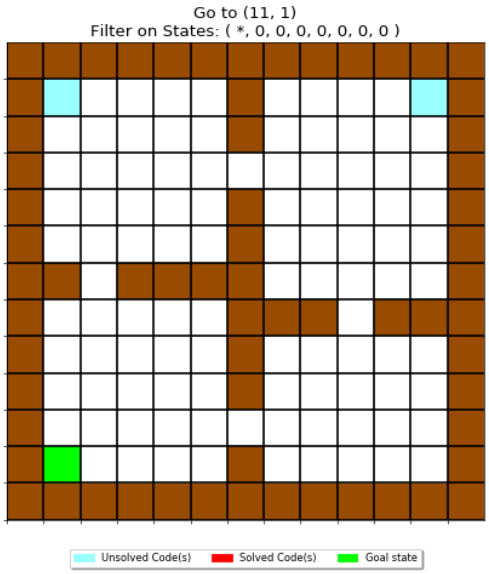


Figure C.1: Task 1: Navigate to *bottom-left* corner

Task 2

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{top-right}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

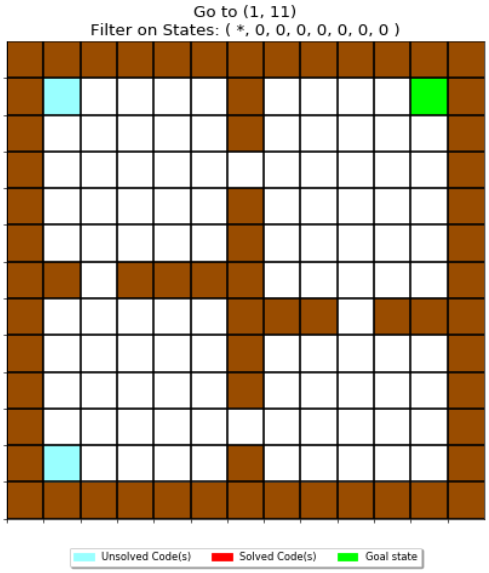


Figure C.2: Task 2: Navigate to *top-right* corner

Task 3

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{top-left}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

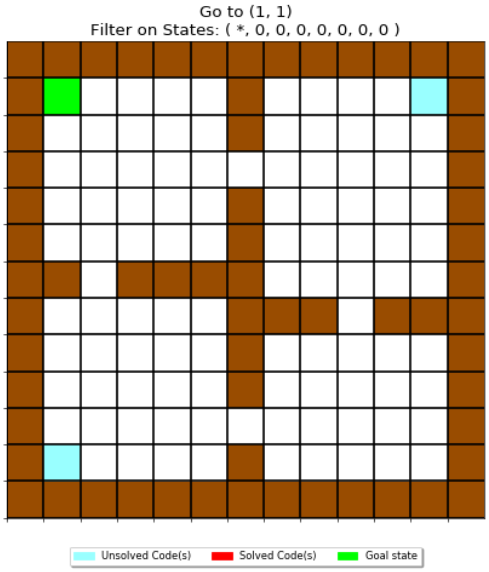


Figure C.3: Task 3: Navigate to *top-left* corner

Task 4

Start state-space: $(\textit{top-left}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, *, *)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

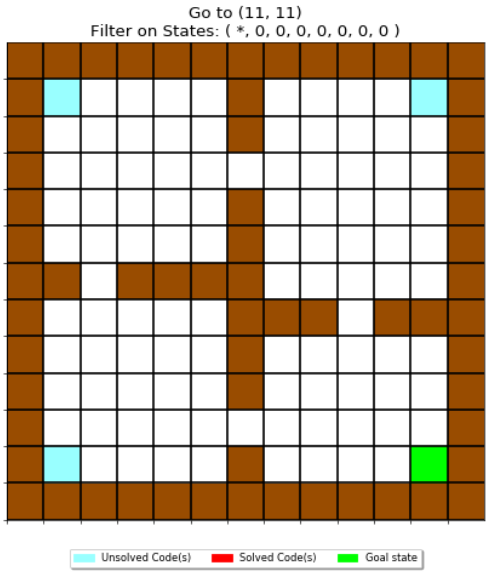


Figure C.4: Task 4: Navigate to *bottom-right* corner

Task 5

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, *, 0)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{bottom-left}, 0, 0, 0, 0, *, *, 1)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

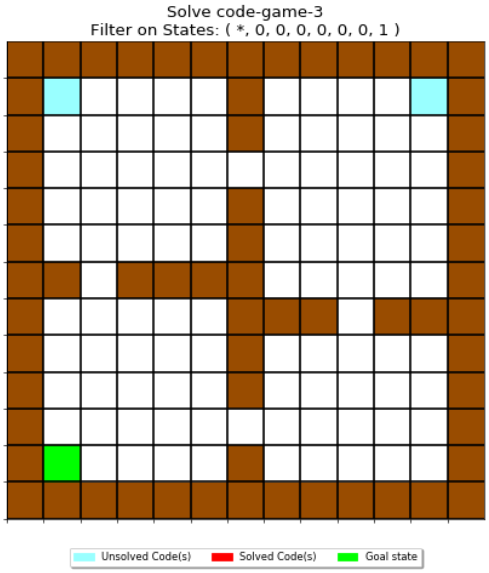


Figure C.5: Task 5: Navigate to *bottom-left* corner and Solve *code-game-3*

Task 6

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, *, 0, *)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{top-right}, 0, 0, 0, 0, *, 1, *)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

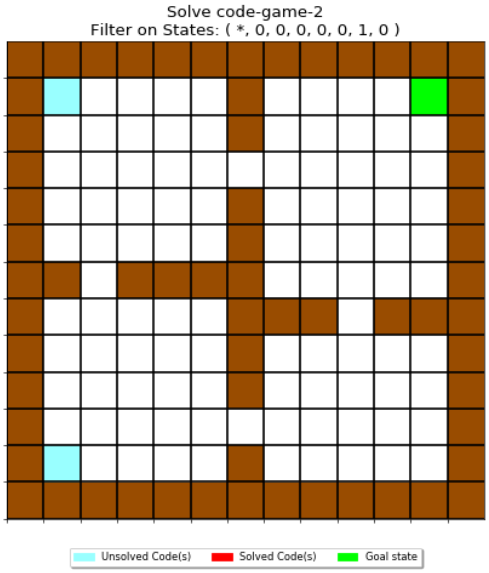


Figure C.6: Task 6: Navigate to *top-right* corner and Solve *code-game-2*

Task 7

Start state-space: $(\textit{bottom-right}, 0, 0, 0, 0, 0, *, *)$, where $* \in \{0, 1\}$

Goal state-space: $(\textit{top-left}, 0, 0, 0, 0, 1, *, *)$, where $* \in \{0, 1\}$

Reward: $+1$ for going to goal otherwise -1

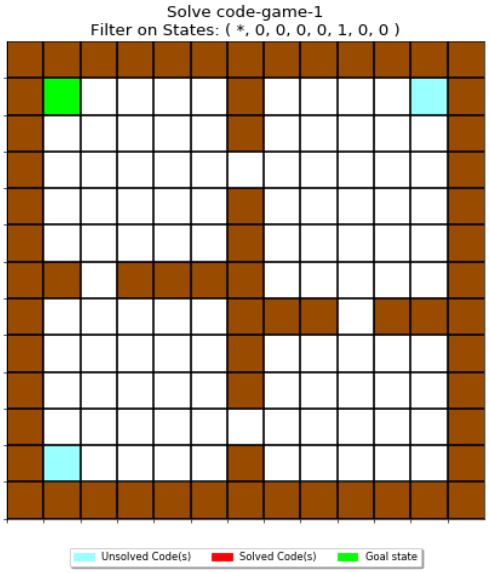


Figure C.7: Task 7: Navigate to *top-left* corner and Solve *code-game-1*

Task 8

Start state: $(\textit{bottom-right}, 0, 0, 0, 0, 0, 0, 0)$

Goal state: $(\textit{bottom-right}, 0, 0, 0, 0, 1, 1, 1)$

Reward: $+1$ for going to goal otherwise -1

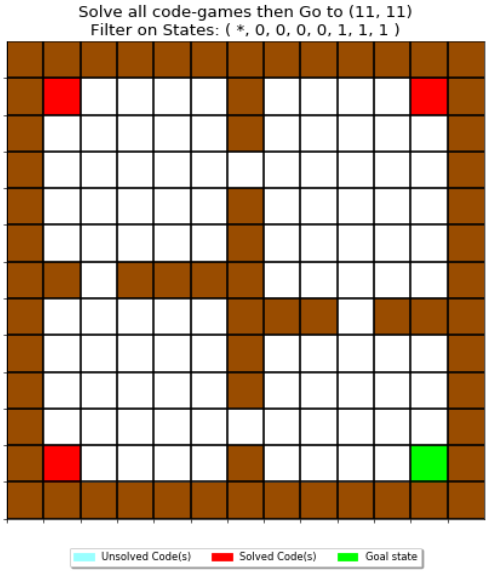


Figure C.8: Task 8: Solve all three *code-games* and then navigate to *bottom-right*

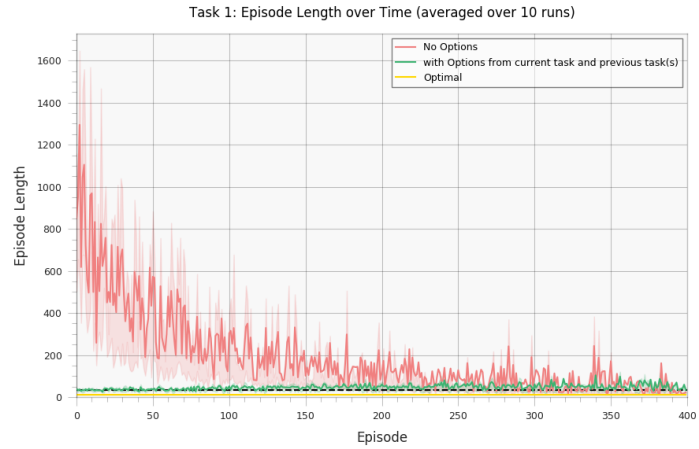
C.2 Solving each Task

When solving each task for this experiment, we use three different approaches for comparison purposes. We make use of a standard Q-Learning algorithm to solve the task without any transfer (i.e. does not use the library of options). We use this as the benchmark, which is red in colour, for all graphs in this experiments section. The other two approaches both make use of the SMDP Q-Learning algorithm, and the difference is in the library of options. We want to compare how well the agent performs with the library of options constructed while learning to solve previous tasks (blue), against the library of options including the options extracted from the current task at hand (green). We expect that the options extracted from the current task would provide an advantage over options from previous tasks only.

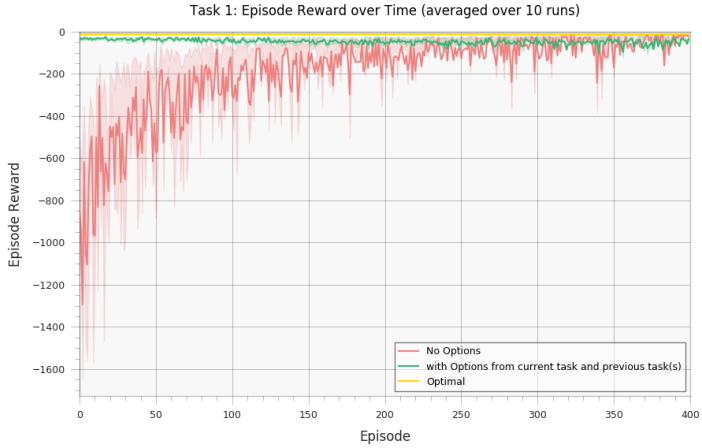
The parameters for SMDP Q-Learning and standard Q-Learning were kept the same, which are, the discount factor $\gamma = 0.9$ and a learning rate $\alpha = 0.5$. In order to allow the agent to explore, we make use of the ϵ-greedy action selection with $\epsilon = 0.1$. Each approach was averaged over ten randomly seeded runs.

For the options extraction process, we use the agent's policy to generate 1000 trajectories for each task, which is then passed to NPBRS. The NPBRS method segments the trajectories and computes the likelihood of the segment with the aid of an inverse reinforcement learning algorithm. We use the MaxEnt IRL algorithm, with learning rate starting at $\alpha = 0.2$ with linear decay and threshold error $\delta = 0.01$. Once the segment's reward function is uncovered, we can learn the skill policy. With the trajectory segments and corresponding skill policy, we want to determine the initiation set and learn the termination condition in order to form an option. We used the one-class SVM, as discussed in section 3.4.3, in order to form these options. After that, the library of options can be used when solving the subsequent tasks.

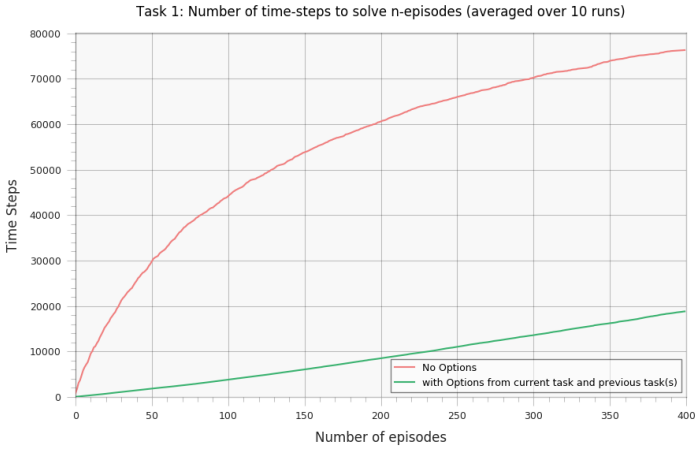
Task 1



(a) Number of steps per episode

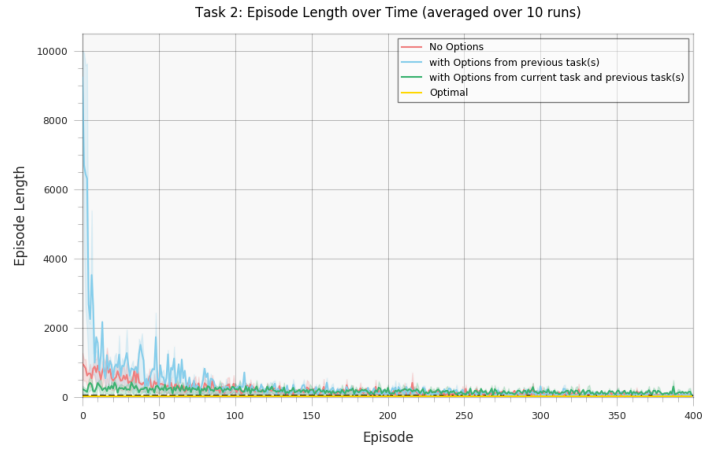


(b) Total reward per episode

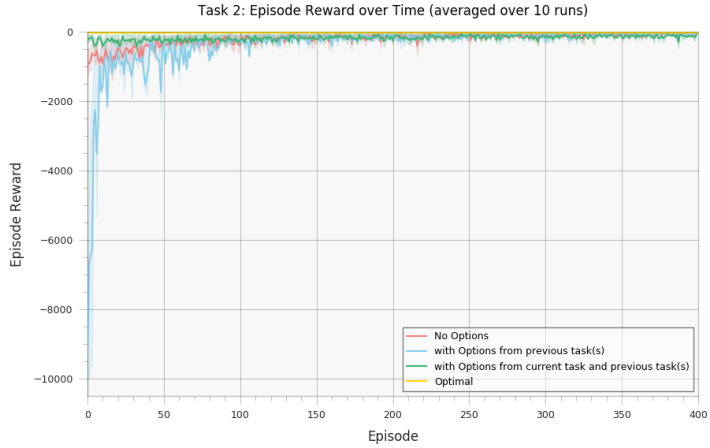


(c) Number of time-steps to solve n-episodes

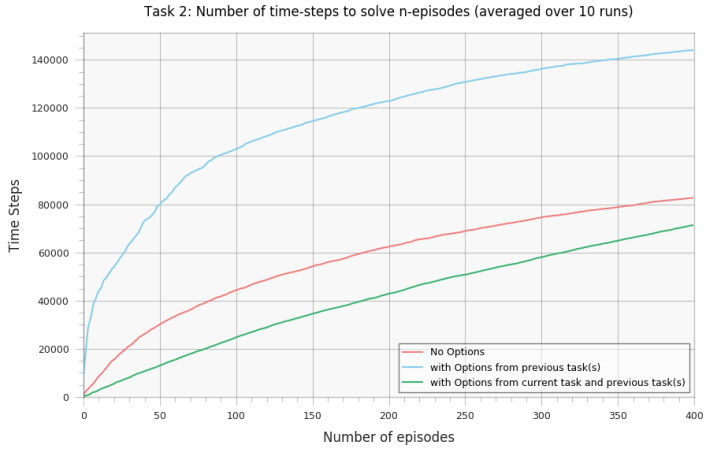
Task 2



(a) Number of steps per episode

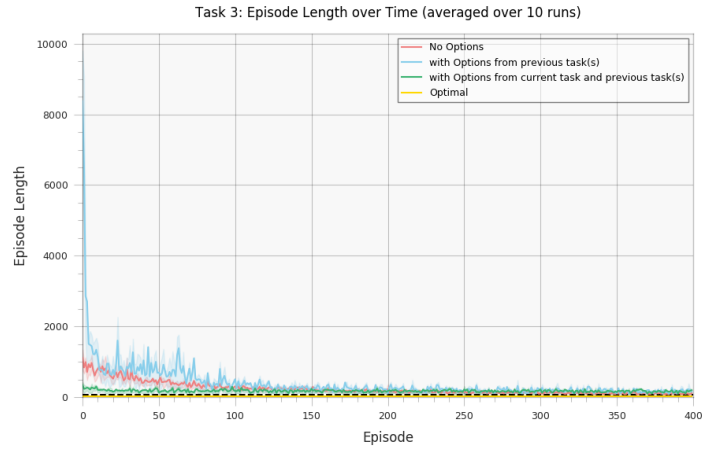


(b) Total reward per episode

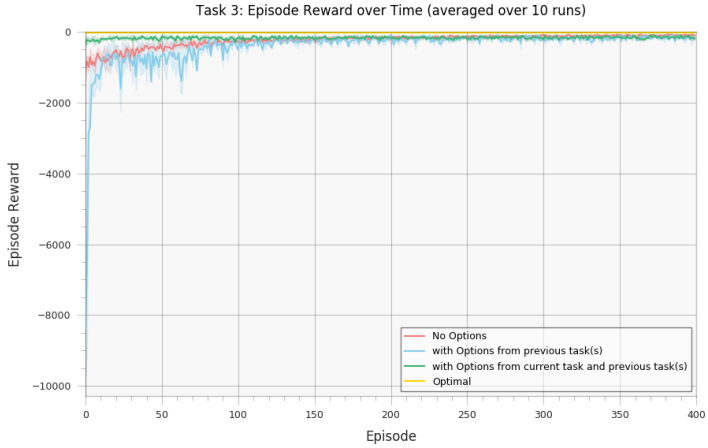


(c) Number of time-steps to solve n-episodes

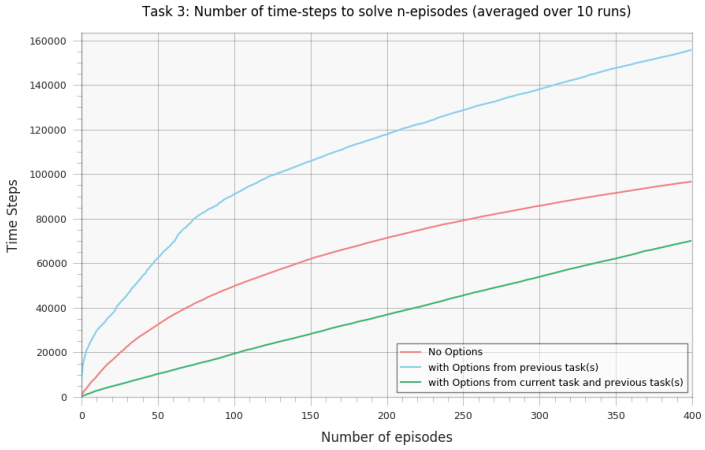
Task 3



(a) Number of steps per episode

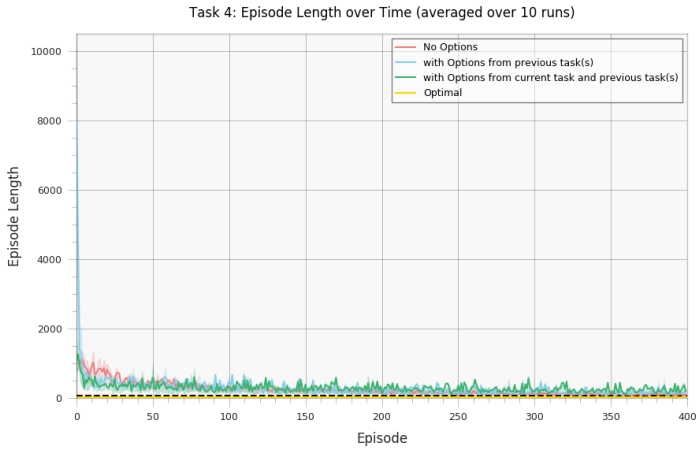


(b) Total reward per episode

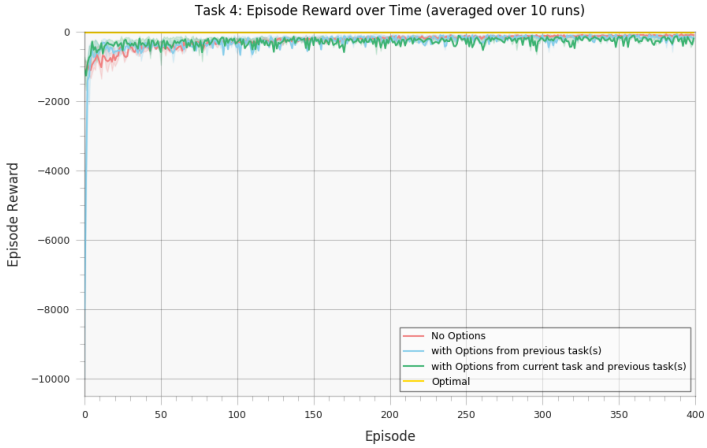


(c) Number of time-steps to solve n-episodes

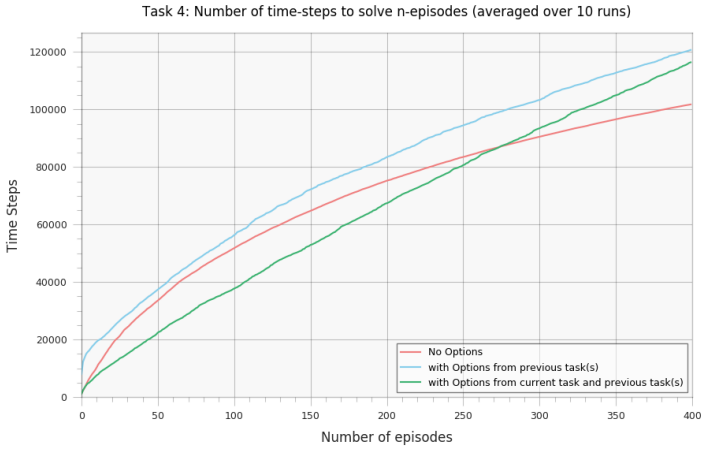
Task 4



(a) Number of steps per episode

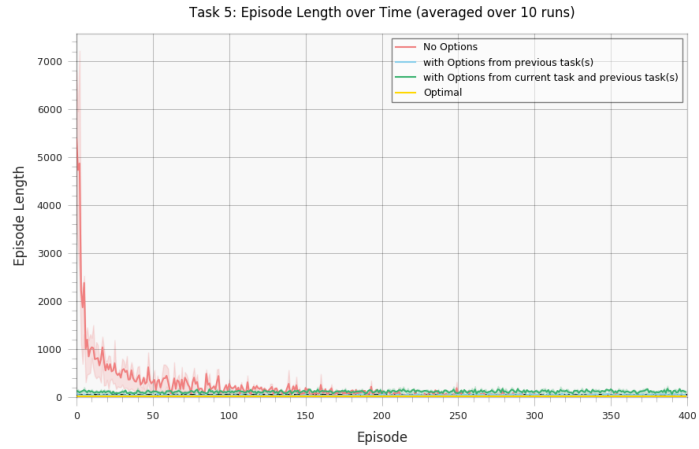


(b) Total reward per episode

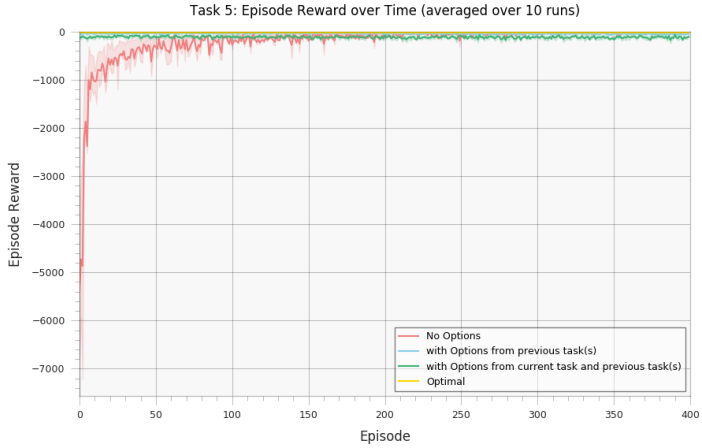


(c) Number of time-steps to solve n-episodes

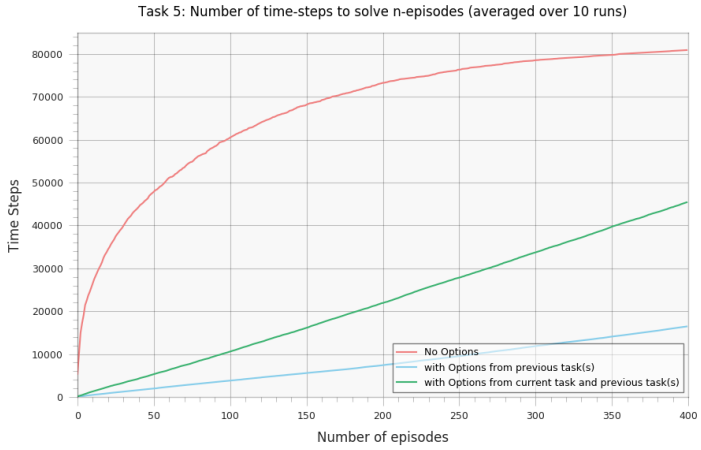
Task 5



(a) Number of steps per episode

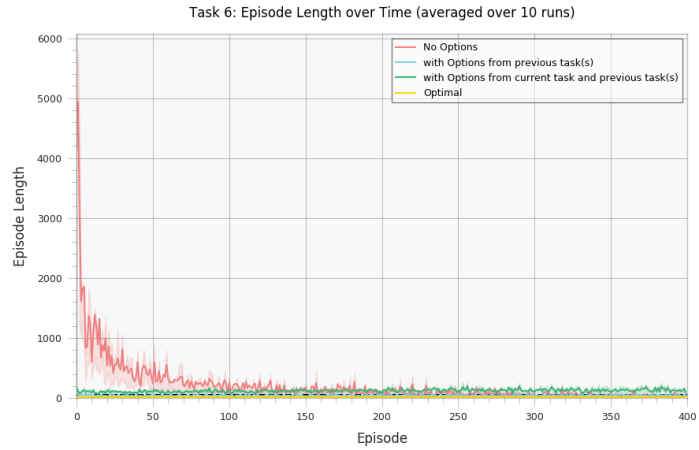


(b) Total reward per episode

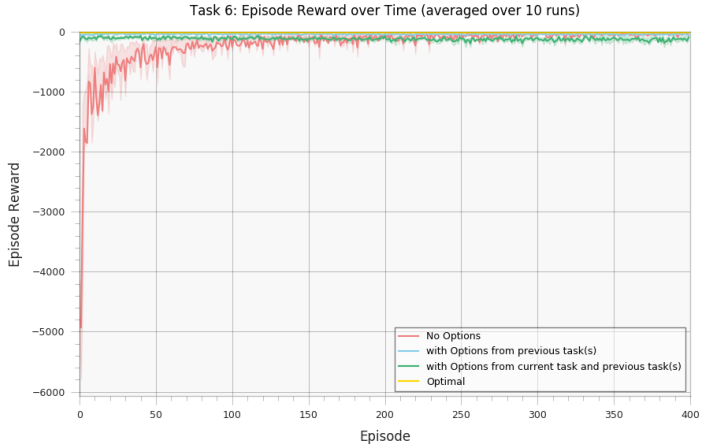


(c) Number of time-steps to solve n-episodes

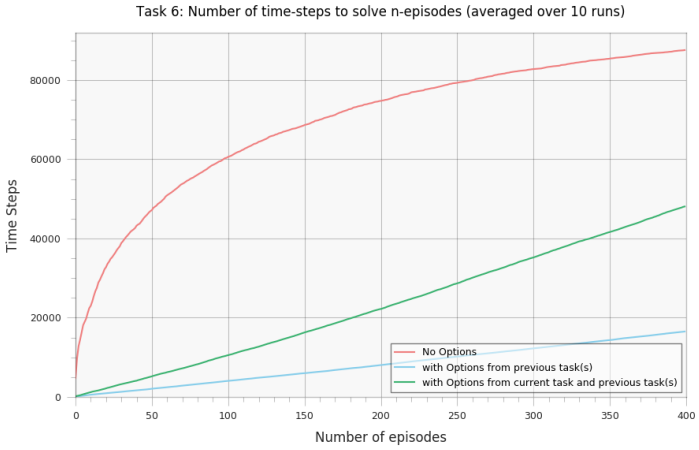
Task 6



(a) Number of steps per episode

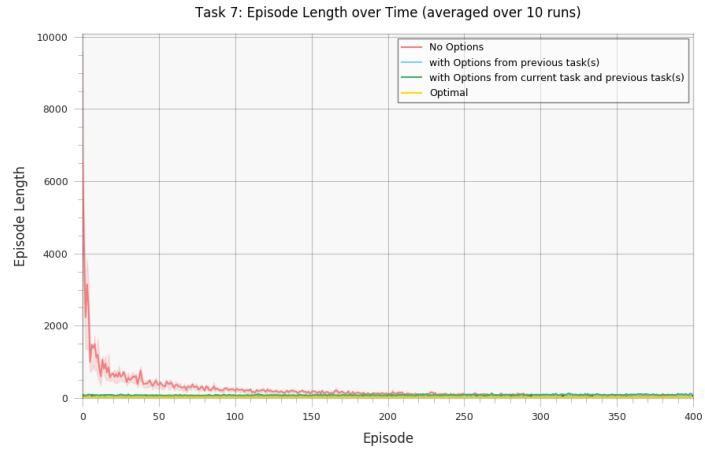


(b) Total reward per episode

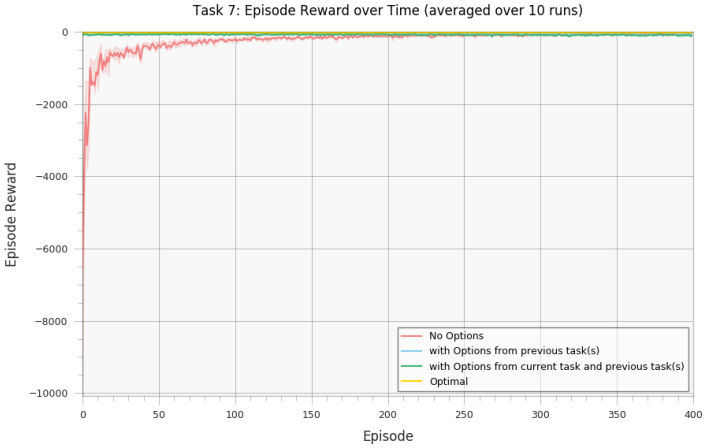


(c) Number of time-steps to solve n-episodes

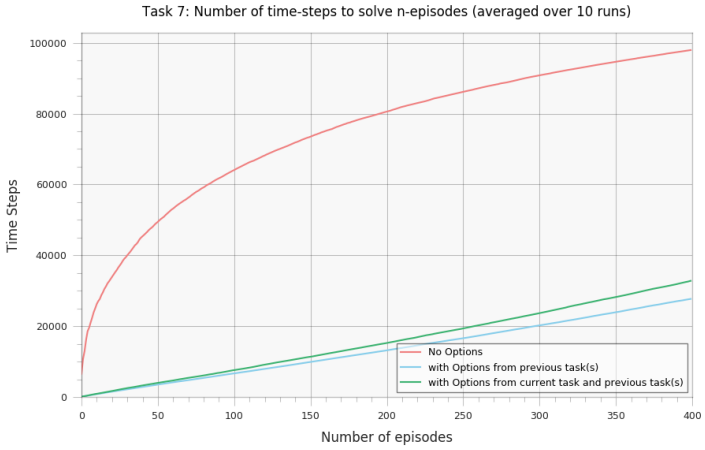
Task 7



(a) Number of steps per episode

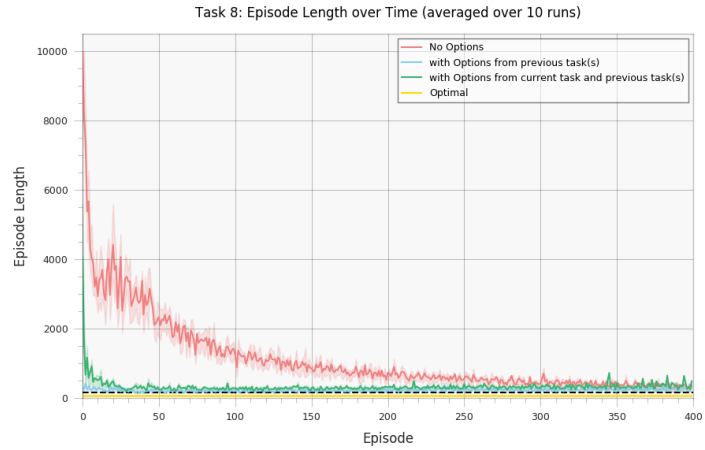


(b) Total reward per episode

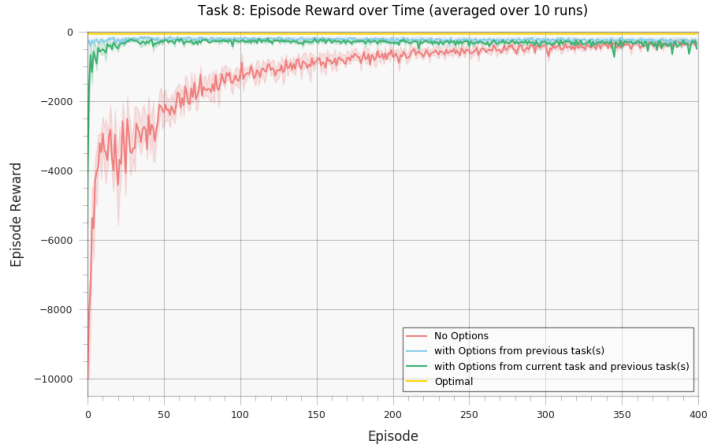


(c) Number of time-steps to solve n-episodes

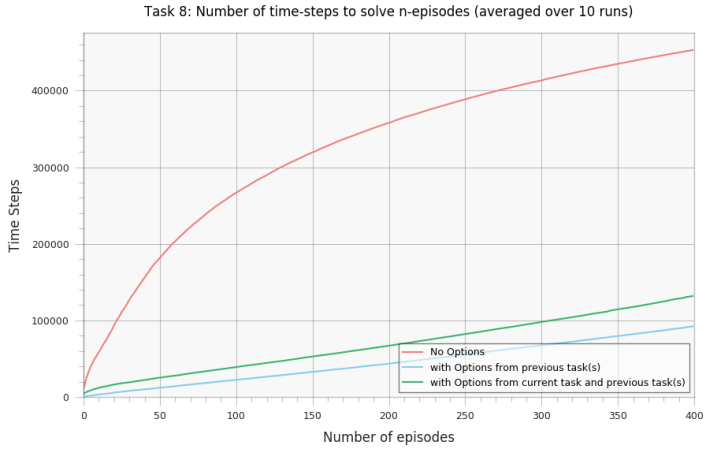
Task 8



(a) Number of steps per episode



(b) Total reward per episode



(c) Number of time-steps to solve n-episodes