FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Abstraction in Reinforcement Learning
Student:	Ondřej Bíža
Supervisor:	Robert Platt, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2019/20

Instructions

The ability to create useful abstractions automatically is a critical tool for an autonomous agent. Without this, the agent is condemned to plan or learn policies at a relatively low level of abstraction, and it becomes hard to solve complex tasks. What we would like is the ability for the agent to learn new skills or abstractions over time that gradually increase its ability to solve challenging tasks. This thesis will explore this in the context of reinforcement learning.

We plan to do the following:

1. Review the research literature on MDP Homomorphisms, their properties, and algorithms for finding them.

2. Develop novel algorithms that integrate homomophism-finding with online learning methods. The resulting algorithm should enable an autonomous agent to find environmental structure while simultaneously solving tasks in model-free environments.

3. Demonstrate ideas in the context of toy domains and later with simulated robotic manipulation problems.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 5, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Abstraction in Reinforcement Learning

Ondřej Bíža

Department of Applied Mathematics
Supervisor: Robert Platt, Ph.D.

May 8, 2019

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 8, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Ondřej Bíža. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Bíža, Ondřej. *Abstraction in Reinforcement Learning.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Abstrakce je důležitý nástroj pro inteligentního agenta. Pomáhá mu řešit složité úlohy tím, že ignoruje nedůležité detaily. V této práci popíši nový algoritmus pro hledání abstrakcí, Online Partition Iteration, který je založený na teorii homomorfismů Markovských rozhodovacích procesů. Můj algoritmus dokáže vytvořit abstrakce ze zkušeností nasbíraných agentem v prostředích s vysokodimenzionálními stavy a velkým množství dostupných akcí. Také představím nový přístup k přenášení abstrakcí mezi různými úlohami, který dosáhl nelpších výsledků ve většině mých experimentů. Nakonec dokážu správnost svého algoritmu pro hledání abstrakcí.

Klíčová slova strojové učení, posilované učení, abstrakce, robotická manipulace, homomorfismy Markovsých rozhodovacích procesů, deep learning

Abstract

Abstraction is an important tool for an intelligent agent. It can help the agent act in complex environments by selecting which details are important and which to ignore. In my thesis, I describe a novel abstraction algorithm called Online Partition Iteration, which is based on the theory of Markov Decision Process homomorphisms. The algorithm can find abstractions from a stream of collected experience in high-dimensional environments. I also introduce a technique for transferring the found abstractions between tasks that outperforms a deep Q-network baseline in the majority of my experiments. Finally, I prove the correctness of my abstraction algorithm.

Keywords machine learning, reinforcement learning, abstraction, robotic manipulation, markov decision process homomorphisms, deep learning

Contents

1 Introduction **1**
- 1.1 Abstraction and control 1
- 1.2 Structure of the thesis 3

2 Abstraction in Reinforcement Learning **5**
- 2.1 Reinforcement Learning 5
- 2.2 State abstraction 6
- 2.3 MDP homomorphisms 9
- 2.4 Finding homomorphisms 11
- 2.5 Transfer Learning 12
- 2.6 Deep Reinforcement Learning and Fully Convolutional Neural Networks 13

3 Online Partition Iteration **17**
- 3.1 Abstraction 17
- 3.2 Partitioning algorithm 18
- 3.3 Increasing robustness 19
- 3.4 Transferring abstract MDPs 21

4 Empirical Analysis of Online Partition Iteration **23**
- 4.1 Grid world puck stacking 23
- 4.2 Discrete blocks world 27
- 4.3 Continuous pucks world 28
- 4.4 Discussion 32

5 Proof of Correctness of Online Partition Iteration **33**

Conclusion **37**

Bibliography **39**

A Acronyms **45**

B Contents of enclosed CD **47**

List of Figures

1.1 (a) an example of a simple two puck stacking task in a 4×4 grid world environment (bottom) and its abstract representation (top). (b) UR5, a robotic arm that may attempt to solve this task [1]. . . 2

2.1 Comparison between state abstractions induced by bisimulation and MDP homomorphisms. The fourteen images in the left panel are examples of states from the fourteen distinct abstract states induced by bisimulation. Abstraction based on MDP homomorphisms results in only three abstract states (right panel). 8

2.2 (a) and (b) are an example of semantic segmentation with fully convolutional neural networks from [2]. Analogously, (c) is an input depth map and (d) is a map of output Q-values from a fully convolutional deep Q-network. The deep Q-network is trained to stack two pucks in a pseudo-continuous environment. 13

3.1 Projection (Algorithm 4) of a single state s. s is evaluated under actions a_1, a_2, a_3 and a_4. For each pair (s, a_i), the classifier g predicts its state-action block b_j. s belongs to a state block identified by the set of state-action blocks $\{b_1, b_3\}$. 20

4.1 A diagram of the architecture of our convolutional network. "Conv" stands for a convolutional layer. Each convolutional and fully-connected layer is following with a ReLU activation function. . . . 24

4.2 Cumulative rewards for two transfer learning experiment in a 4×4 puck stacking environment. (a) In the first 1000 episodes, the agent learns the initial task: stacking two pucks. Subsequently, the agent tries to learn a second, harder, task: stacking three pucks. (b) In the first 1500 episode, the agent learns the initial task: stacking three pucks. The second task requires the agent to make two stacks of two pucks. The results were averaged over 20 runs. 26

4.3 Comparison with Wolfe et al. [3] in the Blocks World environment. The horizontal line marks the highest mean reward per time step reached by Wolfe et al. We averaged our results over 100 runs with different goals. 27
4.4 The goal states of the four types of tasks in our continuous pucks world domain. a) the task of stacking three pucks on top of one another, b) making two stacks of two pucks, c) arranging three pucks into a connected component, d) building stairs from three pucks. 30
4.5 Grid searches over state-action block size thresholds and prediction confidence thresholds (described in Subsection 3.3). The y-axis represents the average per-episode reward (the maximum is 10) obtained by planning in the quotient MDP induced by the resulting partition. Stack, component and stairs represent the three puck world tasks shown in Figure 4.4. We report the means and standard deviations over 20 runs with different random seeds. 31

List of Tables

4.1 Transfer experiments in the pucks world domain. We measure the number of time steps before the agent reached the goal in at least 80% of episodes over a window of 50 episodes and report the mean and standard deviation over 10 trials. Unreported scores mean that the agent never reached this target. The column labeled *Options* represents our transfer learning method (Subsection 3.4), *Baseline* is deep Q-network described in Subsection 4.3 that does not retain any information from the initial task and *Baseline, share weights* copies the trained weights of the network from initial task to the transfer task. The bolded scores correspond to a statistically significant result for a Welch's t-test with $P < 0.1$. 29

Chapter 1

Introduction

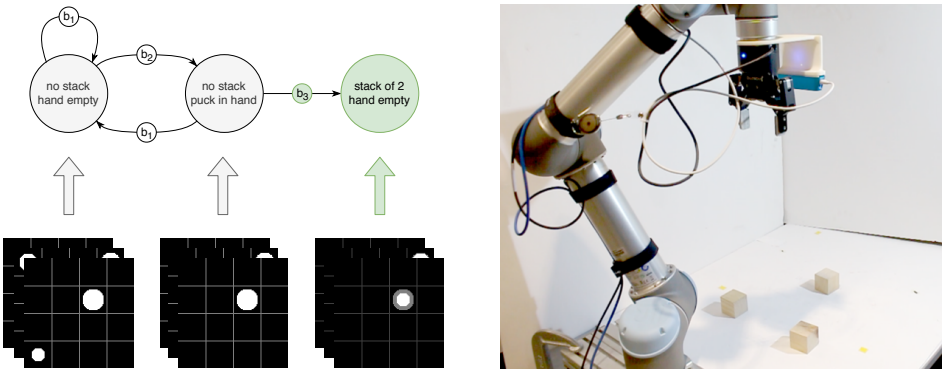
Abstraction is integral to our everyday reasoning about the world. When I plan my trip to an academic conference in Montreal, Canada, I first search for the flight I will take, ignoring the concrete steps such as packing my luggage and getting to the airport. It is only when the day of the travel comes that I need to concern myself with the taxi company I will use and the time when I should leave. At the bottom level of this imaginary hierarchy lie the precise movements I need to perform to get in and out of a taxi or the coordination of my fingers to open my passport at the right page for the flight attendant to check.

People seamlessly traverse this hierarchy of abstractions, from the conscious act of buying a plane ticket to a subconscious motion plan. In contrast, many robotic systems only plan their behavior in terms of elementary actions, such as moving their arm by two centimeters to the right or rotating their wrist by ten degrees [4]. Hence, it is exceedingly difficult for robots to perform complex manipulation tasks (e.g. packing a box in an Amazon warehouse, assembling Ikea furniture), without the explicit programming of abstract concepts by their designers.

In the rest of this chapter, I delve deeper into the types of abstraction considered in this work and outline the structure of my thesis.

1.1 Abstraction and control

We can divide abstractions into two kinds: temporal and spatial. Temporal abstraction encapsulate some sequence of movements into a single abstract concept. For instance, we could introduce the concept ”pick up a cup” to a robot, which would encapsulate a series of torque commands required to reach this goal. Temporal abstractions are important if we wish to plan on a higher level than individual torque commands. Spatial abstractions, on the other hand, group different states of the world around us into a single abstract concept. We could introduce the concept ”cup is on the table” to our robot,



(a) (b)

Figure 1.1: (a) an example of a simple two puck stacking task in a 4×4 grid world environment (bottom) and its abstract representation (top). (b) UR5, a robotic arm that may attempt to solve this task [1].

so that it can decide if the abstract action ”pick up a cup” can be executed. This concepts holds true regardless of the position of the cup on the table. Hence, it is a spatial abstraction.

Figure 1.1 (a) shows an example of a task that includes both temporal and spatial abstractions. The environment the robot is interacting with is a 4×4 grid world with two pucks placed in random cells. The pictures you see in the bottom part of the figure are taken from the top of the work space by a simulated depth camera. To make the task simpler, temporal abstractions ”pick at (x, y)” and ”place at (x, y)” are introduced. For instance, the abstraction ”pick at (1, 1)” will instruct the robot to attempt to pick up a puck in the cell (1, 1). Analogously, the abstraction ”place at (1, 1)”, which can be executed only when the robot is holding something in its hand, executes a series of actions to this end.

The goal of my work is to automatically find the spatial abstractions depicted in the top part of Figure 1.1 (a). These abstractions should group different states of the environment into meaningful concepts, such as ”no stack, puck in hand” for all states, in which the robot is holding a puck in its hand and the second puck is in an arbitrary position and of an arbitrary size. Usually, abstractions are introduced to make reaching some goal easier: in this case, the goal is to stack two pucks on top of one another. To complete this example, Figure 1.1 (b) shows an instance of this task in the real world.

Furthermore, I aim to find these abstraction ”online”, meaning that they are discovered based on a stream of experience collected by an agent interacting with an environment, such as the robotic arm in Figure 2.1 (b). In contrast, many previous abstraction algorithms require the full model of the environment [5, 6], which tends to be difficult to specify. Finally, the found abstractions should not only be useful for solving the present task, but also

transferrable to other problems, so that the robot can reuse skills it learned in the past.

1.2 Structure of the thesis

My thesis consists of five chapters. I review various theoretical frameworks and state-of-the-art algorithms for finding abstractions in Chapter 2 and mention several transfer learning techniques relevant to my work. I also review Deep Learning methods used by our algorithm. I present our algorithm, Online Partition Iteration, in Chapter 3 and prove its correctness in Chapter 5. Chapter 4 presents an empirical evaluation of our method.

The description of Online Partition Iteration, the proof of correctness and our empirical evaluation were published as a full conference paper [7], which I co-authored with professor Robert Platt.

Chapter 2

Abstraction in Reinforcement Learning

I study abstraction in the context of Reinforcement Learning (RL) because it has proven to be an useful framework for many robotic manipulation problems. For example, the RL framework has been successfully used to find grasps in unconstrained environments [8, 9, 10, 11], learn to manipulate tools [12] and to solve miscellaneous manipulation tasks [13, 14]. One desirable aspect of RL is that it considers an agent that is embodied in an environment and can change the states of the environment with its actions. RL can model both tasks with a set of goal states (as in traditional shortest-path problems [15]) as well as a continuous spectrum of rewards for different states. RL is also readily extensible to cover partially-observable environments [16] and can be augmented with actions that span more than one time step [17].

This chapter reviews the basics of Reinforcement Learning and introduces two methods of abstraction: bisimulation and MDP homomorphisms. I also cover transfer in the context of RL and review neural network based approaches to RL that appear in my work.

2.1 Reinforcement Learning

The Reinforcement Learning framework models an agent's interactions with an environment as a Markov Decision Process (MDP) [18]. An MDP is a tuple $\langle S, A, \Phi, P, R \rangle$, where S is the set of states, A is the set of actions, $\Phi \subset S \times A$ is the state-action space (the set of available actions for each state), the transition function $P : S \times A \times S \to [0, 1]$ assigns the probability of transitioning into each state given the current state and a selected action, and $R : S \times A \to \mathbb{R}$ is the reward function.

As an example, we can formalize the task of picking up an object with a robotic arm as an MDP. The state could be represented as the positions and

velocities of all the joints of the arm, perhaps together with an image taken by a camera located next to the arm; we can choose to directly output joint torques as actions. The transition function P, which is usually unknown, then relates the joint torques with the changes in the position of the arm. Finally, the function R rewards successful grasps.

An agent selects actions based on a policy $\pi : S \times A \rightarrow [0, 1]$, a probability distribution over all admissible actions given a state. The policy is usually learned, either explicitly or through the learning of a model of the environment or values of states. Given a policy π, we might want to know the future rewards we will get when we start in a certain state and act according to π thereafter. This is called the *state value*

$$v_\pi(s) = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right],$$

where γ is the discount factor that encourages the agent to pursue immediate rewards. It is also convenient to define the *state-action value*

$$q_\pi(s, a) = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right].$$

Notice that since we output the joint torques in our example, the agent might need to choose actions very frequently, perhaps 20 times per second as in [10]. Hence, picking up an object according to a policy π might take hundreds of time steps. A convenient *temporal abstraction* of such a long sequence of actions is an **option** [17]. An option $\langle I, \pi, \beta \rangle$ is a temporally extended action: it can be executed from the set of states I and selects primitive actions with a policy π until it terminates. The probability of terminating in each state is expressed by $\beta : S \rightarrow [0, 1]$. It can also be viewed as a closed-loop controller.

2.2 State abstraction

In general, the aim of state abstraction is to ignore the aspects of states unimportant for solving the task at hand. It should reduce the computational complexity of the reasoning of the agent, but also help it make better decisions. If we return back to our grasping example from the previous section, vital information about the current state include the orientation of the robotic arm, the shape of the object and so on. On the other hand, features such as the color of the object to be picked or a clutter in the background should be ignored.

This process is commonly formalized as a partitioning of the state space. Let $B = \{B_1, B_2, ..., B_N\}$ be a partition, such that $\bigcup_{i=1}^{N} B_i = S$ and $B_i \cap B_j = \emptyset$. We call B a state partition and each B_i a state block. The question then becomes which states should belong to the same state block. For instance,

Algorithm 1 Partition Improvement [5]

Input: State partition B.
Output: Refined state partition B'.

```
1: procedure PartitionImprovement
2:     B' ← B
3:     for each block B_i ∈ B do
4:         while B contains block B_j for which B' ≠ SPLIT(B_j, B_i, B') do
5:             B' = SPLIT(B_j, B_i, B')
6:         end while
7:     end for
8: end procedure
```

if our task is to pick up a cup from a table, all states, where the cup is in a particular location, should be in the same state block regardless of other objects on the table.

One way of achieving this goal is finding state partitions that satisfy stochastic bisimulation homogeneity, or *bisimulation* in short [19].

Definition 1. A state partition $B = \{B_1, B_2, ..., B_N\}$ has the bisimulation property with respect to an MDP M if and only if for each $B_i, B_j \in B$, for each action $a \in A$ and for each pair of states $p, q \in B_i$, $\sum_{r \in B_j} P(p, a, r) = \sum_{r \in B_j} P(q, a, r)$ and $R(p, a) = R(q, a)$.

Figure 2.1 shows an example of a bisimilar state partition for the task of stacking two pucks in a grid world environment. The state is represented as a depth image plus a binary variable indicating if the hand is full or empty. There are four actions, each corresponding to "pick" or "place" (depending on the state of the robot's hand) in one of the four grid cells. In this domain, the actions of picking and placing the pucks do not depend on the sizes of the pucks. Therefore, each bisimulation block, represented as a single image in the figure, contains many images of pucks of different sizes in the same configuration.

The question then becomes how do we find such partitions. If we are given a fully-specified MDP with discrete states and actions, we can apply the Partition Iteration algorithm [19, 5]. The algorithm applies Partition Improvement (Algorithm 1) until it arrives to the coarsest bisimilar partition. Partition Improvement iteratively splits state blocks (using the SPLIT operation) until they all satisfy the bisimulation property with respect to each other. See [5] for more details.

It is often the case that we do not have access to the underlying dynamics of the MDP. One possible approach is to learn the model. In the puck stacking task, we could estimate the reward and transition functions from many

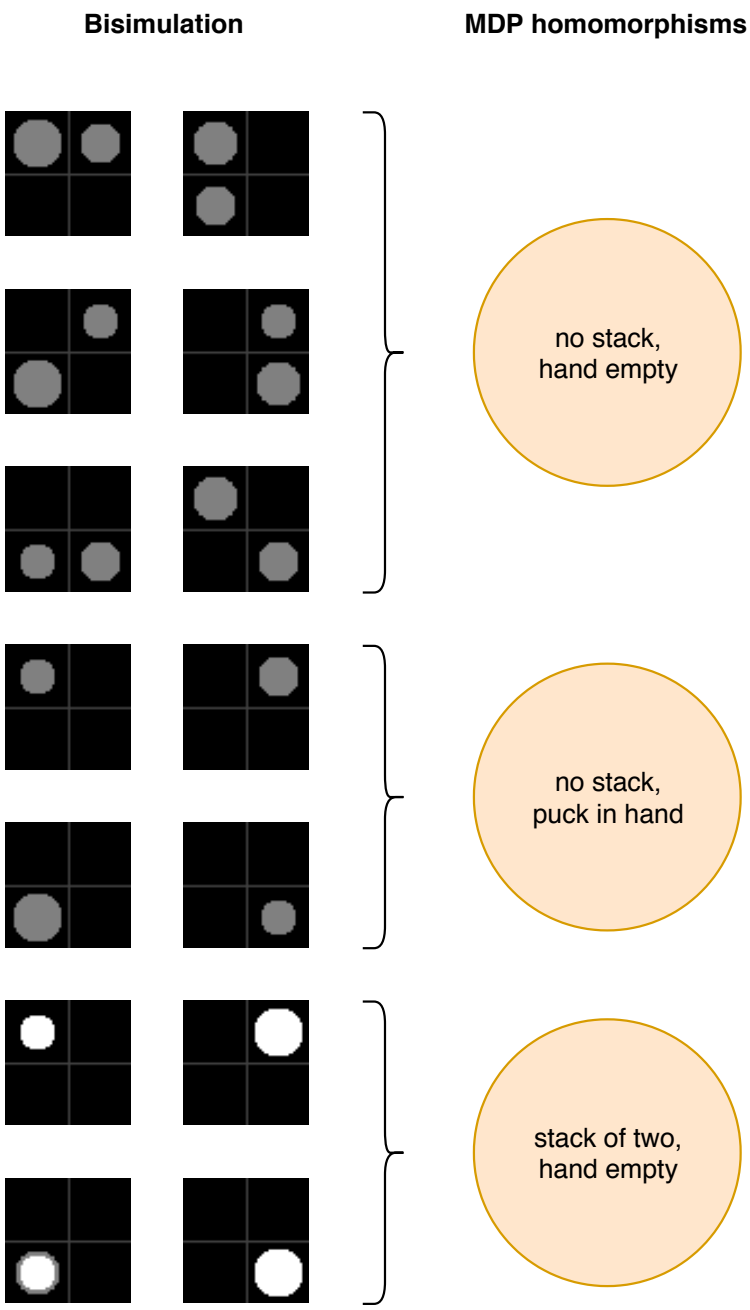


Figure 2.1: Comparison between state abstractions induced by bisimulation and MDP homomorphisms. The fourteen images in the left panel are examples of states from the fourteen distinct abstract states induced by bisimulation. Abstraction based on MDP homomorphisms results in only three abstract states (right panel).

recorded attempts to stack the two pucks. However, the learned models often contain inaccuracies, whereas bisimulation requires transition probabilities and expected rewards to be exactly equal. This problem has been addressed through approximate bisimulation, which allows the dynamics of a pair of states in the same state block to differ up to some constant ϵ [20]. The drawback is that the approximate SPLIT operation has more than one solution and finding the sequence of splits that lead to the coarsest bisimilar partition is a NP-complete problem. Hence, heuristics are needed. Approximate bisimulation can also be turned into a distance metric between pairs of states [21].

Bisimulation is not the sole approach to state abstraction. Li et al. relate bisimulation to four other approaches to abstraction based on equivalence of state values and policies [22]. It is shown that these methods are more lax than bisimulation. Notably, Jong et al. developed a method for abstracting away irrelevant aspects of states that do not affect optimal policies, without needed the full model of the environment [23]. But, their approach is limited to discrete MDPs.

2.3 MDP homomorphisms

Returning back to the example in Figure 2.1, it is evident that the number of bisimulation blocks will grow with the size of the action space. If we make the action grid finer, say 112×112 instead of 2×2 depicted in the figure, the number of blocks in the coarsest partition explodes to millions. Such abstraction would be difficult to learn and is not particularly useful for our agent. The key piece missing from bisimulation-based abstraction is the reduction of the *action space*.

The following definitions build up to the concept of MDP homomorphisms: an abstraction framework that enables us to reduce both the state and the action space [6]. I cover this concept in much more detail than bisimulation because MDP homomorphisms underpin my algorithm described and evaluated in the next three chapters. The review of MDP homomorphisms is also the first part of my assignment.

First, I define a partition of the *state-action space* Φ. Intuitively, the state-action space provides us with the set of admissible actions for each state. For example, if our agent is holding an object in its hand, picking up a second object with the same hand is not an admissible action. Therefore, $\Phi \subset S \times A$.

Definition 2. A *partition of an MDP* $M = \langle S, A, \Phi, P, R \rangle$ is a partition of Φ. Given a partition B of M, the *block transition probability of* M is the function $T : \Phi \times B|S \to [0, 1]$ defined by $T(s, a, [s']_{B|S}) = \sum_{s'' \in [s']_{B|S}} P(s, a, s'')$.

Similarly to Partition Iteration, my algorithm also iteratively splits blocks, leading to a *refinement* of the partition.

Definition 3. A partition B' is a *refinement* of a partition B, $B' \ll B$, if and only if each block of B' is a subset of some block of B.

Partitioning of the state-action space presents more challenges than a state space partition. To recover an abstraction of the state space from the state-action partition, we need to project it onto the state space.

Definition 4. Let B be a partition of $Z \subseteq X \times Y$, where X and Y are arbitrary sets. For any $x \in X$, let $B(x)$ denote the set of distinct blocks of B containing pairs of which x is a component, that is, $B(x) = \{[(w, y)]_B \mid (w, y) \in Z, w = x\}$. The *projection of B onto X* is the partition $B|X$ of X such that for any $x, x' \in X$, $[x]_{B|X} = [x']_{B|X}$ if and only if $B(x) = B(x')$.

An example of projection is given in Figure 3.1. Here, we have four state-action pairs $(s, a_1), ..., (s, a_4)$. The first three belong to the state-action block b_1 and the last one is a member of b_4. Assuming that there are only four actions, the state s is then projected onto the state block $\{b_1, b_3\}$. All states that constitute only state-action pairs in b_1 and b_3 will be projected onto this state block.

Next, I define two desirable properties a state-action partition should have. These properties are analogous to the bisimulation property of a state partition (Definition 1).

Definition 5. A partition B of an MDP $M = \langle S, A, \Phi, P, R\rangle$ is said to be *reward respecting* if $(s_1, a_1) \equiv_B (s_2, a_2)$ implies $R(s_1, a_1) = R(s_2, a_2)$ for all $(s_1, a_1), (s_2, a_2) \in \Phi$.

Definition 6. A partition B of an MDP $M = \langle S, A, \Phi, P, R\rangle$ has the *stochastic substitution property* (SSP) if for all $(s_1, a_1), (s_2, a_2) \in \Phi$, $(s_1, a_1) \equiv_B (s_2, a_2)$ implies $T(s_1, a_1, [s]_{B|S}) = T(s_2, a_2, [s]_{B|S})$ for all $[s]_{B|S} \in B|S$.

Having a partition with these properties, we can construct the *quotient MDP* (I also call it the abstract MDP). The quotient MDP encapsulates the abstraction we found by partitioning the state-action space.

Definition 7. Given a reward respecting SSP partition B of an MDP $M = \langle S, A, \Phi, P, R\rangle$, the *quotient MDP* M/B is the MDP $\langle S', A', \Phi', P', R'\rangle$, where $S' = B|S$; $A' = \bigcup_{[s]_{B|S} \in S'} A'_{[s]_{B|S}}$ where $A'_{[s]_{B|S}} = \{a'_1, a'_2, ..., a'_{\eta(s)}\}$ for each $[s]_{B|S} \in S'$; P' is given by $P'([s]_f, a'_i, [s']_f) = T_b([(s, a_i)]_B, [s']_{B|S})$ and R' is given by $R'([s]_{B|S}, a'_i) = R(s, a_i)$. $\eta(s)$ is the number of distinct classes of B that contain a state-action pair with s as the state component.

The top part of Figure 1.1 (a) depicts a quotient MDP for the task of stacking two pucks. It is obvious that each state of the quotient MDP corresponds to many states in the underlying MDP. But notice that each action of the quotient MDP also aggregates multiple elementary actions. For instance,

the abstraction action "pick puck" corresponds to any of the two actions that pick in the occupied grid cells. To decide which actions to choose in the underlying MDP given the abstract MDP, we need a mapping from states to abstract states and from actions in a particular state to abstract actions. One set of such mappings are MDP homomorphisms.

Definition 8. An *MDP homomorphism* from $M = \langle S, A, \Phi, P, R\rangle$ to $M' = \langle S', A', \Phi', P', R'\rangle$ is a tuple of surjections $\langle f, \{g_s : s \in S\}\rangle$ with $h(s,a) = (f(s), g_s(a))$, where $f : S \to S'$ and $g_s : A \to A'$ such that $R(s,a) = R'(f(s), g_s(a))$ and $P(s,a,f^{-1}(f(s'))) = P'(f(s), g_s(a), f(s'))$. We call M' a *homomorphic image* of M under h.

Through the constraints on the transition and reward function, MDP homomorphisms ensure that the abstract MDP captures all important dynamics of the underlying environment. As states by the next theorem, we can recover homomorphisms from a reward respecting SSP partition.

Theorem 1 ([6])**.** Let B be a reward respecting SSP partition of MDP $M = \langle S, A, \Phi, P, R\rangle$. The quotient MDP M/B is a homomorphic image of M.

Computing the optimal state-action value function in the quotient MDP usually requires fewer computations, but does it help us act in the underlying MDP? The last theorem states that the optimal state-action value function *lifted* from the minimized MDP is still optimal in the original MDP:

Theorem 2 (Optimal value equivalence, [6])**.** Let $M' = \langle S', A', \Phi', P', R'\rangle$ be the homomorphic image of the MDP $M = \langle S, A, \Phi, P, R\rangle$ under the MDP homomorphism $h(s,a) = (f(s), g_s(a))$. For any $(s,a) \in \Phi$, $Q^*(s,a) = Q^*(f(s), g_s(a))$.

2.4 Finding homomorphisms

The concept of MDP homomorphisms together with a sketch of an algorithm for finding them were first proposed in Balaraman Ravindran's Ph.D. thesis [6]. Ravindran's algorithm directly follows Partition Iteration, an algorithm for finding bisimulations. The only major difference is that the SPLIT operation splits a *state-action block* (instead of a state block) with respect to a state block. As with bisimulation, this algorithm assumes the MDP is discrete and fully specified; the environments for my robotic manipulation tasks are neither discrete nor is it possible to easily specify the transition probabilities.

The literature on finding homomorphisms in real-world MDPs is sparse. One option is to learn the model of the environment and then search for approximate homomorphisms [24], which are analogous to approximate bisimulation. That is, since the learned reward and transition function will always introduce approximation error, approximate homomorphisms allow the dynamics of two states in the same state block to vary up to some ϵ. A method for finding approximate homomorphisms is described in [25].

However, model learning can be difficult, especially when the states are represented as images taken by a camera. To my knowledge, the only model-free algorithm for finding homomorphisms was developed by Wolfe et al. [3]. They embed their algorithm inside of a decision tree–inspired by the UTree algorithm [26]–that first splits states into state blocks and then partitions actions for a given state block. The main difference between our approaches is that their method operates over Controlled Decision Processes, which augment the reward function of the standard MDP with additional supervision. This supervision makes the task of finding homomorphisms easier. Their method also cannot handle continuous state spaces–an important requirement for my algorithm.

2.5 Transfer Learning

Transfer learning is a broad subfield of Machine Learning concerned with accelerating the learning of new tasks through previously attained knowledge. This ability is a part of human reasoning [27] and could be considered a vital component of an intelligent agent [28]. Since my work is concerned with transferring abstractions between Reinforcement Learning tasks, I will narrow the focus of this section only to the relevant Reinforcement Learning literature. See the following surveys for a full review of Transfer Learning [29, 30].

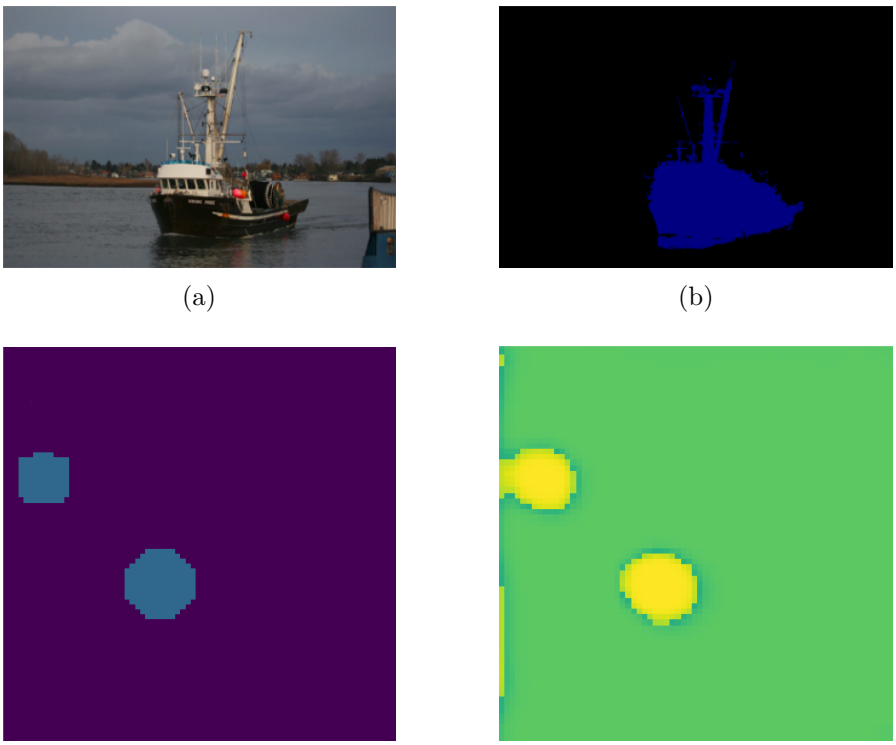
Methods for transfer between Reinforcement Learning domains vary along many axes. Do we transfer the learned Q-values, policy, model or something else? Do the MDPs between which we transfer knowledge vary in terms of the reward function, transition function or both? Do we have access to some mapping between tasks? Taylor et al. categorized around 40 different approaches according to these considerations and more [31].

Both the notions of bisimulation and MDP homomorphisms can be used for transfer learning. The underlying idea is that if we learn some useful policy π_A in an MDP M_A and we have a mapping between M_A and a new MDP M_B, then we can act according to π_A in M_B. As a simple example, imagine that the environment M_A contains a robotic arm and a blue cup and we train a policy π_A to pick up the blue cup. The next day, we are given a red cup–MDP M_B–and are tasked with picking it up with the robotic arm. Then, if we introduce some kind of a mapping stating that a red cup is identical to a blue cup, we can use the previously trained π_A. The two major questions are how do we find these mappings and what policies are worth transferring.

Castro et al. used a metric derived from lax-bisimulation [21] to transfer skills from a small discrete MDP to a large one [32]. The metric calculates the distance between two state-action pairs from different MDPs, which allows for the transfer of policies. Transfer learning with MDP homomorphisms was explored by Soni et al. and Sorg et al. [33, 34]. Both of these approaches only involve the learning of a mapping between the states of the two MDPs,

assuming that the mapping of actions is given. Since the main goal of my work is to abstract the action space and use the abstraction to transfer knowledge, I cannot assume that the mapping of actions between different MDPs is given. Finally, [35] proposed a high-level description of a system that uses homomorphisms to create a collection of skills to apply to different problems.

2.6 Deep Reinforcement Learning and Fully Convolutional Neural Networks



(a) (b) (c) (d)

Figure 2.2: (a) and (b) are an example of semantic segmentation with fully convolutional neural networks from [2]. Analogously, (c) is an input depth map and (d) is a map of output Q-values from a fully convolutional deep Q-network. The deep Q-network is trained to stack two pucks in a pseudo-continuous environment.

I end this chapter with a review of Deep Reinforcement Learning: a recent trend of merging Reinforcement Learning algorithms with deep neural networks. Deep RL is important for my research because it is currently the only technique that can directly map high-dimensional inputs, such as camera images, to actions to be performed by the RL agent. An alternative (and time-

consuming) approach would be to first handcraft filters that extract relevant information from the camera images and then fit a standard Machine Learning algorithm (e.g. support vector machines [36]) to the curated representation.

Mnih et al. introduced the deep Q-network in their seminal Deep RL paper [37]. Their method learns to predict the Q-values of each state-action pair using the TD(0) update [38]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)].$$

Unlike standard RL algorithms, deep Q-network represents the Q-function as a neural network trained through mini-batch gradient descent: an approach that has next to none theoretical guarantees, is unstable and prone to overfitting on the portion of the state-action space, where the agent is currently located. Through clever tricks, Mnih et al. attained human-level performance across a test suite of 49 ATARI games, a previously insurmountable challenge for RL algorithms. The stability problem was addressed with a target neural network, a second neural network that predicts the $Q(S_{t+1}, a)$ term in the TD(0) update, preventing the system from entering feedback loops that lead to exploding gradients. The target network is synchronized with the main network every T steps. Second, deep Q-network contains a long replay buffer so that it can sample diverse experience at each training step–an approach that overcomes the overfitting problem. Many improvements to the base architecture have been proposed since 2015, including a more sophistical replay buffer [39], better ways of predicting the current and target Q-values [40, 41, 42] and a "rainbow" deep Q-network that integrates the three years of research into a single RL agent [43].

Actor-Critic methods represent the second popular branch of Deep RL. Instead of recovering the policy by taking the action with the maximum predicted Q-value, the Actor in the Actor-Critic framework learns to output actions directly. The Critic, which predicts state or state-actions values, usually only assists the Actor during the training phase by providing more accurate gradient estimates. An obvious advantage of having an Actor is that we can predict continuous actions without much difficulties, whereas the deep Q-network is suited for discrete action spaces. Silver et al. and Lillicrap et al. developed Actor-Critic agents represented by neural network that can learn a range of continuous control tasks; e.g. learning to walk in a 2D environment and controlling a car in a driving simulator. A second advantage of this framework is that more theoretical guarantees can be derived for the Actor neural networks. Notably, the trust-region methods [44] ensure that a single gradient update does not ruin the Actor network, which greatly increases the stability of the algorithm. Proximal Policy Optimization, a trust region method, has been shown to be significantly more stable and sample efficient than previous Actor-Critic agents. See [45] for a comprehensive review of the Deep RL literature.

Although my robotic manipulation problems could be parameterized by continuous actions, a more usual approach is to overlay a fine grid over the workspace and introduce the "pick" and "place" actions for each cell in the grid. Figure 2.2 (c) and (d) show an example of such parameterization. The image in (c) is a 112×112 depth image and (d) represents the Q-values for the 12544 actions in the 112×112 action grid. You can see that the (trained) agent prefers to execute the "pick" action in the location of the two pucks (yellow represents a higher Q-value than green). However, learning the Q-values of so many actions is non-trivial. I commonly work with datasets that contain 10 000 transitions (tuples of state, action, reward and the next state); hence, some actions are not executed even once. We need an approach that generalizes over locations in the workspace. A standard deep Q-network is not capable of such generalization, as it predictions the Q-values of each action using a fully-connected layer–no information about the structure of the action space is encoded in the design.

Two recent papers solved this challenge in different ways: a fully convolutional network was used in [13] to learn "push" and "pick" actions over a fine grid of actions, whereas [1] introduced a new type of spatial abstraction with a local representation for each action. I focus on the former because it is easier to integrate into my system, which uses neural networks for both classification of state-action blocks and prediction of Q-values. Fully convolutional networks are the go-to method for semantic segmentation (see an example in Figure 2.2 (a) and (b)) [46, 2]. Their implementation is simple, since they consist only of a stack of convolutional layers interlaid with non-linear transformations and possibly batch normalization [47]. The ability to generalize over locations is the result of not having any parameters specific to a particular position in the input image–the convolutional operation slides a small filter over the image, treating each position equally. Finally, two useful improvements to fully convolutional networks are dilated convolutions [48], a method that stretches convolutional filters to increase the portion of the input they see, followed by Dilated Residual Networks [49]. The latter combines the benefits of Deep Residual Learning [50]–the ability to learn convolutional networks with tens or hundreds of layers–with dilation to achieve state-of-the-art performance on semantic segmentation tasks.

Chapter 3

Online Partition Iteration

I solve the problem of abstracting an MDP with a discrete or continuous state-space and a discrete action space. The MDP can have an arbitrary reward function, but I restrict the transition function to be deterministic. This restriction simplifies my algorithm and makes it more sample-efficient (because I do not have to estimate the transition probabilities for each state-action pair).

This chapter starts with an overview of my abstraction process (Section 2), followed by a description of my algorithm for finding MDP homomorphisms (Section 3.2). I describe several augmentations to the base algorithm that increase its robustness in Section 3.3. Finally, Section 3.4 contains the description of my transfer learning method that leverages the learned MDP homomorphism to speed up the learning of new tasks. The description of the algorithm is based on our publication [7].

3.1 Abstraction

Algorithm 2 gives an overview of my abstraction process. Since I find MDP homomorphisms from experience, I first need to collect transitions that cover all regions of the state-action space. For simple environments, a random exploration policy provides such experience. But, a random walk is clearly not sufficient for more realistic environments because it rarely reaches the goal of the task. Therefore, I use the vanilla version of a deep Q-network [37] to collect the initial experience in bigger environments.

Subsequently, I partition the state-action space of the original MDP based on the collected experience with my Online Partition Iteration algorithm (Algorithm 3). The algorithm is described in detail in Subsection 3.2. The state-action partition B–the output of Algorithm 3–induces a quotient, or abstract, MDP according to Definition 7.

The quotient MDP enables both planning optimal actions for the current task (Section 3.2) and learning new tasks faster (Section 3.4).

Algorithm 2 Abstraction

1: **procedure** Abstraction
2: $E \leftarrow$ collect initial experience with an arbitrary policy π
3: $g \leftarrow$ a classifier for state-action pairs
4: $B \leftarrow OnlinePartitionIteration(E, g)$
5: $M' \leftarrow$ a quotient MDP constructed from B according to Definition 7
6: **end procedure**

3.2 Partitioning algorithm

My online partitioning algorithm (Algorithm 3) is based on the Partition Iteration algorithm from [5]. It was originally developed for stochastic bisimulation based partitioning, and I adapted it to MDP homomorphisms (following Ravindran's sketch [6]). Algorithm 3.2 starts with a reward respecting partition obtained by separating transitions that receive distinct rewards (*SplitRewards*). The reward respecting partition is subsequently refined with the *Split* (Algorithm 5) operation until a stopping condition is met. *Split(b, c, B)* splits a state-action block b from state-action partition B with respect to a state block c obtained by projecting the partition B onto the state space.

The projection of the state-action partition onto the state space (Algorithm 4) is the most complex component of my method. I train a classifier g, which can be an arbitrary model, to classify state-action pairs into their corresponding state-action blocks. The training set consists of all transitions the agent experienced, with each transition belonging to a particular state-action block. During State Projection, g evaluates a state under a sampled set of actions, predicting a state-action block for each action. For discrete action spaces, the set should include all available actions. The set of predicted state-action blocks determines which state block the state belongs to.

Figure 3.1 illustrates the projection process: a single state s is evaluated under four actions: a_1, a_2, a_3 and a_4. The first three actions are classified into the state-action block b_1, whereas the last action is assigned to block b_3. Therefore, s belongs to the state block identified by the set of the predicted state-action blocks $\{b_1, b_3\}$.

The output of Online Partition Iteration is a partition B of the state-action space Φ. According to Definition 7, the partition induces a quotient MDP. Since the quotient MDP is fully defined, I can compute its optimal Q-values with a dynamic programming method such as Value Iteration [38].

In order to act according to the quotient MDP, I need to connect it to the original MDP in which I select actions. Given a current state s and a set of actions admissible in s, A_s, I predict the state-action block of each pair (s, a_i), $a_i \in A_s$ using the classifier g. Note that Online Partition Iteration trains g in the process of refining the partition. This process of predicting state-action block corresponds to a single step of State Projection: I determine which state

Algorithm 3 Online Partition Iteration

Input: Experience E, classifier g.
Output: Reward respecting SSP partition B.

```
1: procedure OnlinePartitionIteration
2:     B ← {E}, B′ ← {}
3:     B ← SplitRewards(B)
4:     while B ≠ B′ do
5:         B′ ← B
6:         g ← TrainClassifier(B, g)
7:         B|S ← Project(B, g)
8:         for block c in B|S do
9:             while B contains block b for which B ≠ Split(b, c, B) do
10:                B ← Split(b, c, B)
11:            end while
12:        end for
13:    end while
14: end procedure
```

block s belongs to. Since each state in the quotient MDP corresponds to a single state block (by Definition 7), I can map s to some state s' in the quotient MDP.

Given the current state s' in the quotient MDP, I select the action with the highest Q-value and map it back to the underlying MDP. An action in the quotient MDP can correspond to more than one action in the underlying MDP. For instance, an action that places a puck on the ground can be executed in many locations, while still having the same Q-value in the context of puck stacking. I break the ties between actions by sampling a single action in proportion to the confidence predicted by g: g predict a state-action block with some probability given a state-action pair.

3.3 Increasing robustness

Online Partition Iteration is sensitive to erroneous predictions by the classifier g. Since the collected transitions tend to be highly unbalanced and the mapping of state-action pairs into state-action blocks can be hard to determine, I include several augmentations that increase the robustness of my method. Some of them are specific to a neural network classifier.

- **class balancing**: The sets of state-action pairs belonging to different state-action blocks can be extremely unbalanced. Namely, the number of transitions that are assigned a positive reward is usually low. I follow

the best practices from [51] and over-sample all minority classes so that the number of samples for each class is equal to the size of the majority class. I found decision trees do not require oversampling; hence, I use this method only with a neural network.

- **confidence calibration**: The latest improvements to neural networks, such as batch normalization [47] and skip connections [50] (both used by my neural network in Subsection 4.3), can cause miscalibration of the output class probabilities [52]. I calibrate the temperate of the softmax function applied to the output neurons using a multiclass version of Platt scaling [53] derived in [52]. The method requires a held-out validation set, which consists of 20% of all data in my case.

- **state-action block size threshold and confidence threshold**: During State Projection, the classifier g sometimes makes mistakes in classifying a state-action pair to a state-action block. Hence, the State Projection algorithm can assign a state to a wrong state block. This problems usually manifests itself with the algorithm "hallucinating" state blocks that do not exist in reality (note that there are $2^{\min\{|B|,|A|\}} - 1$ possible state blocks, given a state-action partition B). To prevent the *Split* function from over-segmenting the state-action partition due to these phantom state blocks, I only split a state-action block if the new blocks contain a number of samples higher than a threshold T_a. Furthermore,

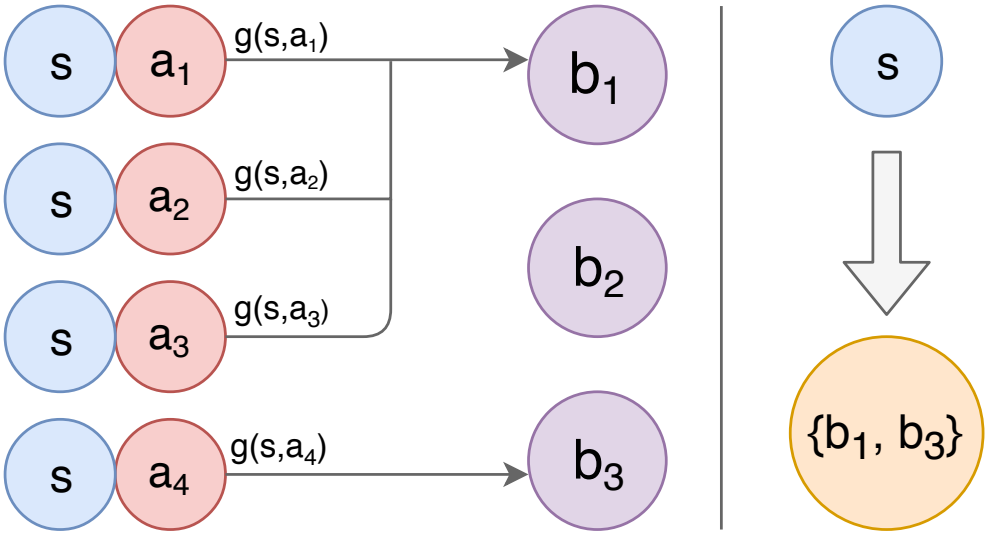


Figure 3.1: Projection (Algorithm 4) of a single state s. s is evaluated under actions a_1, a_2, a_3 and a_4. For each pair (s, a_i), the classifier g predicts its state-action block b_j. s belongs to a state block identified by the set of state-action blocks $\{b_1, b_3\}$.

Algorithm 4 State Projection

Input: State-action partition B, classifier g.
Output: State partition $B|S$.

```
1: procedure PROJECT
2:     B|S ← {}
3:     for block b in B do
4:         for transition t in b do
5:             A_s ← SampleActions(t.next_state)
6:             B_s ← {}
7:             for action a in A_s do
8:                 p ← g.predict(t.next_state, a)
9:                 B_s ← B_s ∪ {p}
10:            end for
11:            Add t to B|S using B_s as the key
12:        end for
13:    end for
14: end procedure
```

I exclude all predictions with confidence lower than some threshold T_c. Confidence calibration makes it easier to select the optimal value of T_c.

3.4 Transferring abstract MDPs

Solving a new task from scratch requires the agent to take a random walk before it stumbles upon a reward. The abstract MDP learned in the previous task can guide exploration by taking the agent into a starting state close to the goal of the task. However, how do we know which state block in the abstract MDP is a good start for solving a new task?

If we do not have any prior information about the structure of the next task, the agent needs to explore the starting states. To formalize this, I create $|B|S|$ options, each taking the agent to a particular state in the quotient MDP from the first task. Each option is a tuple $\langle I, \pi, \beta \rangle$ with

- I being the set of all starting states of the MDP for the new task,
- π uses the quotient MDP from the previous task to select actions that lead to a particular state in the quotient MDP (see Subsection 3.2 for more details) and
- β terminates the option when the target state is reached.

Algorithm 5 Split

Input State-action block b, state block c, partition B.
Output State-action partition B'.

```
1: procedure Split
2:     b_1 ← {}, b_2 ← {}
3:     for transition t in b do
4:         if transition.next_state ∈ c then
5:             b_1 ← b_1 ∪ {t}
6:         else
7:             b_2 ← b_2 ∪ {t}
8:         end if
9:     end for
10:    B' ← B
11:    if |b_1| > 0 && |b_2| > 0 then
12:        B' ← (B' \ {b}) ∪ {b_1, b_2}
13:    end if
14: end procedure
```

The agent learns the Q-values of the options with a Monte Carlo update [38] with a fixed α (the learning rate)–the agent prefers options that make it reach the goal the fastest upon being executed. If the tasks are similar enough, the agent will find an option that brings it closer to the goal of the next task. If not, the agent can choose not to execute any option.

I use a deep Q-network to collect the initial experience in all transfer learning experiments. While my algorithm suffers from the same scalability issues as a deep Q-network when learning the *initial task*, my transfer learning method makes the learning of new tasks easier by guiding the agent's exploration (Table 4.1).

Chapter 4

Empirical Analysis of Online Partition Iteration

Per the third part of my assignment, I evaluate Online Partition Iteration both in toy domains and in a simulated robotic manipulation task. For the toy domains, I implemented a grid-world-like puck stacking task (Section 4.1) and compared my method with Wolfe and Barto in a discrete blocks world domain from their paper [3] (Section 4.2). Next, I designed a series of simulated robotic manipulation tasks much closer to the real world: a pucks world domain, which integrates for distinct manipulation tasks, with a very fine grid of actions similar to a parameterization used by a real robotic arm (Section 4.3).

I aim to answer the following questions with my experiments:

1. Can Online Partition Iteration find homomorphisms in environments with continuous state spaces and high-dimensional action spaces (characteristic for robotic manipulation tasks)?
2. Do options induced by quotient MDPs speed-up the learning of new tasks?
3. How does Online Partition Iteration compare to the only previous online approach to finding homomorphisms [3]?

The results of my comparison with Wolfe et al. and the experiments in the pucks world domain have been published [7].

4.1 Grid world puck stacking

The first testing environment is a grid world with pucks inside some of the cells (see the bottom part of Figure 1.1). The agent can execute a "pick" action in

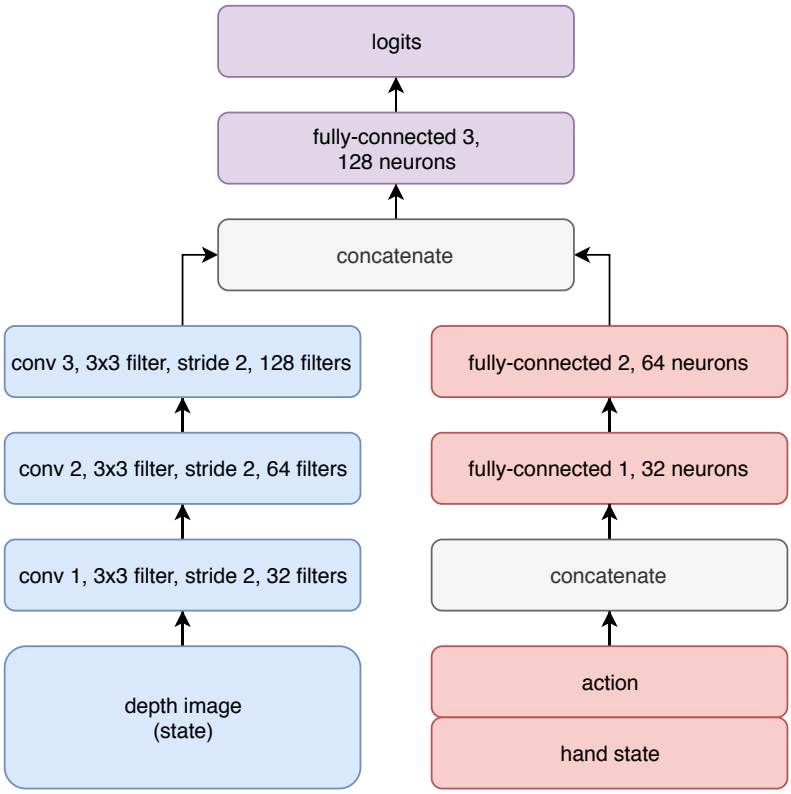


Figure 4.1: A diagram of the architecture of our convolutional network. "Conv" stands for a convolutional layer. Each convolutional and fully-connected layer is following with a ReLU activation function.

any of the cells, which changes the environment only when the corresponding cell is occupied. In that case, the puck is transferred into the robot's hand and it can then execute a "place" action in any of the cells. The state is represented as a depth image of size $(28 * C)^2$, where C is the number of cells along one axis, together with the state of the robot's hand: either full or empty. The task is episodic and the goal is to stack a target number of pucks on top of each other. The task terminates after 20 time steps or when the goal is reached. Upon reaching the goal, the agent is awarded 10 reward, other state-action pairs get 0 reward.

The convolutional network described in Figure 4.1 was used as the classifier g. The agent collected experience with a deep Q-network [37] of the same structure as g, except for the number of output neurons. The number of state-action blocks was limited to 10: the purpose of the limit was mostly to speed-up faulty experiments, but an over-segmented partition can still perform well in some cases. Confidence thresholding (Section 3.3) was not used. The replay buffers of the partitioning algorithm and the deep Q-network were both limited to 10000 transitions–it sufficed for the purpose of finding a reward

respecting SSP partition. The learning rate of the convolutional network was set to 0.001, the batch size to 32 and the weight decay for all layers to 0.0001. The early stopping constant T_s was set to 1000 steps. The deep Q-network settings were as follows: 0.0001 learning rate and update the target network every 100 steps; for the exploration, I linearly decayed the value of ϵ from 1.0 to 0.1 for 5000 steps. The batch size was set to 32.

I first test my algorithm on a sequence of two tasks, where getting to the goal of the first task helps the agent reach the goal of the second task. Specifically, the first task is stacking two pucks in a 4×4 grid world environment and the second task requires the agent to stack three pucks in an arbitrary location. Both of the tasks run for 1000 episodes. I compare my method, which first executes the option that brings the agent to the goal of the first task and then acts using a deep Q-network, with two baselines:

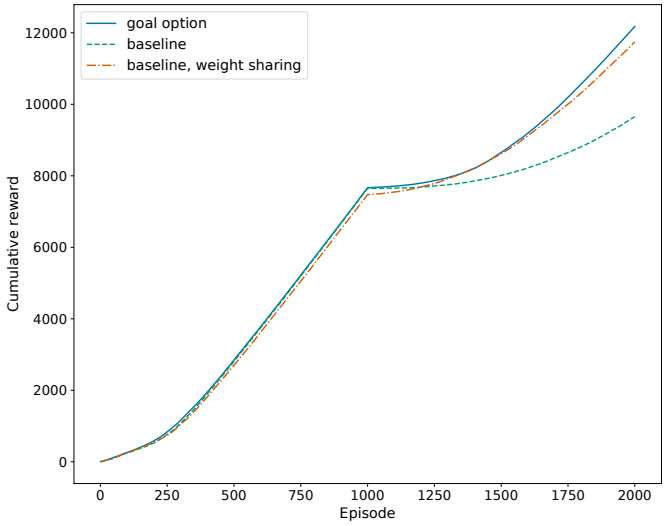
- **baseline**: A vanilla deep Q-network. I reset its weights and replay buffer after the end of each task so that it does not retain any information from the previous task it solved.

- **baseline, weight sharing**: The same as the above, but I do not reset its weights. When a new task starts, it goes to the goal state of the previous tasks and explores from there.

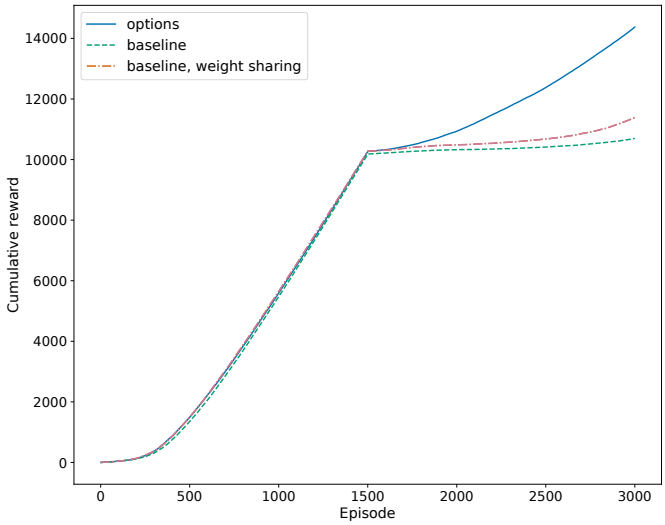
My agent augmented with the goal option reaches a similar cumulative reward to the baseline with weight sharing (Figure 4.2). I expected this result because creating an abstract MDP of the whole environment does not bring any more benefits than simply knowing how to get to the goal state of the previous task.

To show the benefit of the abstract MDP, I created a sequence of tasks in which reaching the goal of the first task does not help: the first task is stacking three pucks and the second task is making two stacks of height two. Upon the completion of the first task, my agent is augmented with options for reaching all state blocks of the abstract MDP. The agent learns the Q-value of the options with a Monte Carlo update [38] with the learning rate α set to 0.1. The baselines are the same as in the previous experiment.

Figure 4.4 shows that my agent learns to select the correct option that brings it closer to the goal of the second task, reaching a significantly higher cumulative reward than both of the baselines. An unexpected result is that the baseline with weight sharing performs better than the other even when reaching the goal of the first task is not as beneficial. I hypothesize that the deep Q-network can learn the second policy easier due to all convolutional layers being pre-trained to spot the pucks from the first task–pre-training convolutional network has been shown to help in tasks such as image classification [54].



(a)



(b)

Figure 4.2: Cumulative rewards for two transfer learning experiment in a 4×4 puck stacking environment. (a) In the first 1000 episodes, the agent learns the initial task: stacking two pucks. Subsequently, the agent tries to learn a second, harder, task: stacking three pucks. (b) In the first 1500 episode, the agent learns the initial task: stacking three pucks. The second task requires the agent to make two stacks of two pucks. The results were averaged over 20 runs.

4.2 Discrete blocks world

Next, I compare Online Partition Iteration with the decision tree method from [3] in their blocks world environment. The environment consists of three blocks that can be placed in four positions. The blocks can be stacked on top of one another, and the goal is to place a particular block, called the *focus block*, in a goal position and height. With four positions and three blocks, 12 tasks of increasing difficulty can be generated. The agent is penalized with -1 reward for each action that does not lead to the goal; reaching the goal state results in 100 reward. Unlike the puck stacking tasks, the state of the blocks world is represented as discrete positions of all the blocks.

Although a neural network can learn to solve this task, a decision tree trains two orders of magnitude faster and often reaches better performance. I used a decision tree from the scikit-learn package [55] with the default settings as the classifier g. All modifications from Section 3.3 specific to a neural network were omitted: class balancing and confidence thresholding. I also disabled the state-action block size threshold because the number of unique transitions generated by this environment is low and the decision tree does not make many mistakes. Despite the decision tree reaching high accuracy, I set

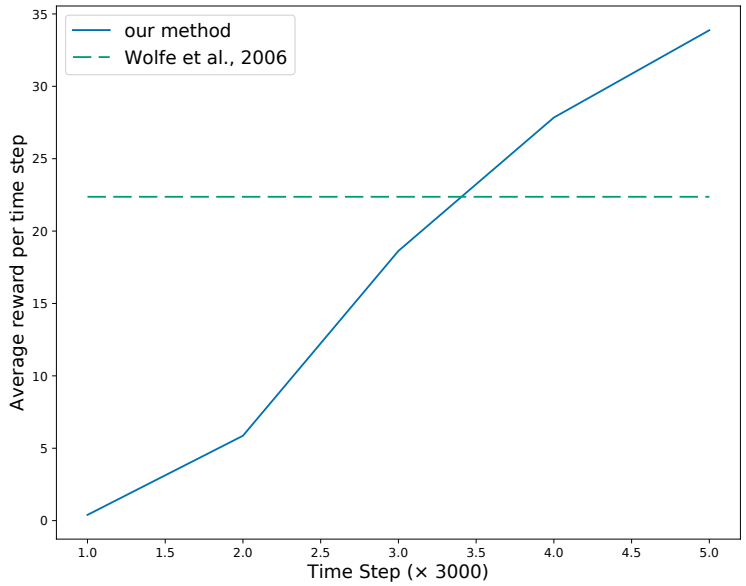


Figure 4.3: Comparison with Wolfe et al. [3] in the Blocks World environment. The horizontal line marks the highest mean reward per time step reached by Wolfe et al. We averaged our results over 100 runs with different goals.

a limit of 100 state-action blocks to avoid creating thousands of state-action pairs if the algorithm fails. The abstract MDP was recreated every 3000 time steps and the task terminated after 15000 time steps.

Figure 4.3 compares the decision tree version of my algorithm with the results reported in [3]. There are several differences between my experiment and the algorithm in [3]: Wolfe and Barto's algorithm works with a Controlled Markov Process (CMP), an MDP augmented with an output function that provides richer supervision than the reward function. Therefore, their algorithm can start segmenting state-action blocks before it even observes the goal state. CMPs also allow an easy transfer of the learned partitions from one task to another; my algorithm solve each task separately. On the other hand, each action in Wolfe's version of the task has a 0.2 chance of failure, but I omit this detail to satisfy the assumptions of my algorithm. Even though each version of the task is easier in some ways are harder in others, I believe the comparison with the only previous algorithm that solves the same problem is valuable.

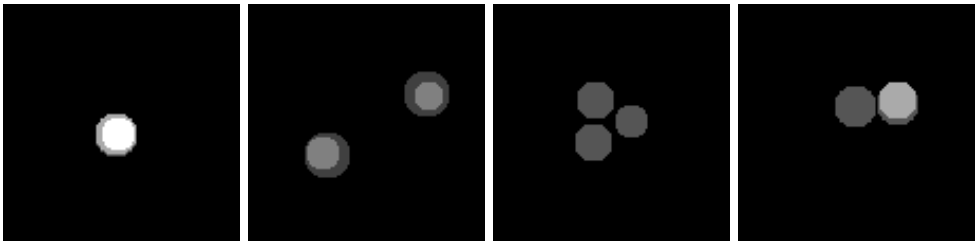
4.3 Continuous pucks world

Finally, I designed the pucks world domain (Figure 4.4) to approximate real-world robotic manipulation tasks. The state is represented by a 112×112 depth image and each pixel in the image is an admissible action. Hence, 12544 actions can be executed in each state. Environments with such a high branching factor favor homomorphisms, as they can automatically group actions into a handful of classes (e.g. "pick puck" and "do nothing") for each state. If an action corresponding to a pixel inside of a puck is selected, the puck is transported into the agent's hand. In the same way, the agent can stack pucks on top of each other or place them on the ground. Corner cases such as placing a puck outside of the environment or making a stack of pucks that would collapse are not allowed. The agent gets a reward of 0 for visiting each non-goal state and a reward of 10 for reaching the goal states. The environment terminates when the goal is reached or after 20 time steps. I implemented four distinct types of tasks: stacking pucks in a single location, making two stacks of pucks, arranging pucks into a connected component and building stairs from pucks. The goal states of the tasks are depicted in Figure 4.4. I can instantiate each task type with a different number of pucks, making the space of possible tasks and their combinations even bigger.

To gather the initial experience for partitioning, I use a shallow fully-convolutional version of the deep Q-network. My implementation is based on the OpenAI baselines [56] with the standard techniques: separate target network with a weight update every 100 time steps and a prioritized replay buffer [39] that holds the last 10 000 transitions. The network consists of five convolutional layers with the following settings (number of filters, filter

Task	Options	Baseline	Baseline, share weights
2 puck stack to 3 puck stack	$\mathbf{2558 \pm 910}$	5335 ± 1540	10174 ± 5855
3 puck stack to 2 and 2 puck stack	$\mathbf{2382 \pm 432}$	-	3512 ± 518
2 puck stack to stairs from 3 pucks	$\mathbf{2444 \pm 487}$	4061 ± 1382	4958 ± 3514
3 puck stack to stairs from 3 pucks	$\mathbf{1952 \pm 606}$	4061 ± 1382	5303 ± 3609
2 puck stack to 3 puck component	2781 ± 605	3394 ± 999	6641 ± 5582
stairs from 3 pucks to 3 puck stack	3947 ± 873	5335 ± 1540	6563 ± 4299
stairs from 3 pucks to 2 and 2 puck stacks	5552 ± 3778	-	5008 ± 1998
stairs from 3 pucks to 3 puck component	3996 ± 2693	3394 ± 999	4856 ± 3600
3 puck component to 3 puck stack	$\mathbf{3729 \pm 742}$	5335 ± 1540	8540 ± 4908
3 puck component to stairs from 3 pucks	3310 ± 627	4061 ± 1382	2918 ± 328

Table 4.1: Transfer experiments in the pucks world domain. We measure the number of time steps before the agent reached the goal in at least 80% of episodes over a window of 50 episodes and report the mean and standard deviation over 10 trials. Unreported scores mean that the agent never reached this target. The column labeled *Options* represents our transfer learning method (Subsection 3.4), *Baseline* is deep Q-network described in Subsection 4.3 that does not retain any information from the initial task and *Baseline, share weights* copies the trained weights of the network from initial task to the transfer task. The bolded scores correspond to a statistically significant result for a Welch's t-test with $P < 0.1$.



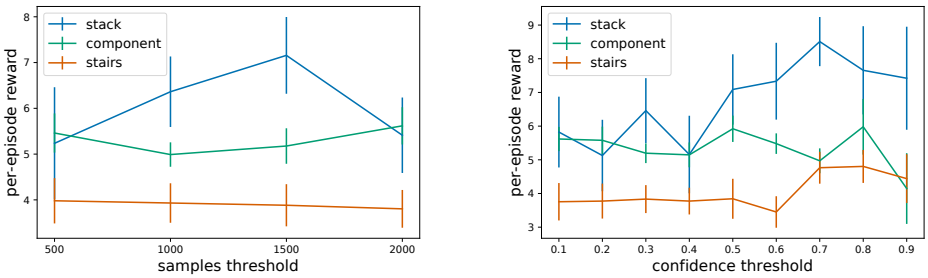
(a) (b) (c) (d)

Figure 4.4: The goal states of the four types of tasks in our continuous pucks world domain. a) the task of stacking three pucks on top of one another, b) making two stacks of two pucks, c) arranging three pucks into a connected component, d) building stairs from three pucks.

sizes, strides): (32, 8, 4), (64, 8, 2), (64, 3, 1), (32, 1, 1), (2, 1, 1). The ReLU activation function is applied to the output of each layer except for the last one. The last layer predicts two maps of Q-values with the resolution 14×14 (for 112×112 inputs)–the two maps correspond to the two possible hand states: "hand full" and "hand empty". The appropriate map is selected based on the state of the hand, and bilinear upsampling is applied to get a 112×112 map of Q-values, one for each action. I trained the network with a Momentum optimizer with the learning rate set to 0.0001 and momentum to 0.9, batch size was set to 32. The agent interacted with the environment for 15000 episodes with an ϵ-greedy exploration policy; ϵ was linearly annealed from 1.0 to 0.1 for 40000 time steps.

Online Partition Iteration requires a second neural network–the classifier g. My initial experiments showed that the predictions of the architecture described above lack in resolution. Therefore, I chose a deeper architecture: the DRN-C-26 version of Dilated Residual Networks [49]. I observed that the depth of the network is more important than the width (the number of filters in each layer) for my classification task. Capping the number of filters at 32 (the original architecture goes up to 512 filters in the last layers) produces results indistinguishable from the original network. DRN-C-26 decreases the resolution of the feature maps in three places using strided convolutions, I downsample only twice to keep the resolution high. I train the network for 1500 steps during every iteration of Online Partition Iteration. The learning rate for the Momentum optimizer started at 0.1 and was divided by 10 at steps 500 and 1000, momentum was set to 0.9. The batch size was set to 64 and the weight decay to 0.0001.

Figure 4.5 reports the results of a grid search over state-action block size thresholds and classification confidence thresholds described in Subsection 3.3. Online Partition Iteration can create a near-optimal partition for the three pucks stacking task. On the other hand, my algorithm is less effective in the



(a) (b)

Figure 4.5: Grid searches over state-action block size thresholds and prediction confidence thresholds (described in Subsection 3.3). The y-axis represents the average per-episode reward (the maximum is 10) obtained by planning in the quotient MDP induced by the resulting partition. Stack, component and stairs represent the three puck world tasks shown in Figure 4.4. We report the means and standard deviations over 20 runs with different random seeds.

component arrangement and stairs building tasks. These two tasks are more challenging in terms of abstraction because the individual state-action blocks are not as clearly distinguishable as in puck stacking.

Next, I investigate if the options induced by the found partitions transfer to new tasks (Table 4.1). For puck stacking, a deep Q-network augmented with options from the previous tasks significantly outperforms both of our baselines. "Baseline" is a vanilla deep Q-network that does not retain any information from the initial task, whereas "Baseline, share weights" remembers the learned weights from the first task. Options are superior to the weights sharing baseline because they can take the agent to any desirable state, not just the goal. For instance, the 2 and 2 puck stacking task benefits from the option "make a stack of two pucks"; hence, options enable faster learning than weight sharing. I would also like to highlight one failure mode of the weight sharing baseline: the agent can sometimes get stuck repeatedly reaching the goal of the initial task without going any further. This behavior is exemplified in the transfer experiment from 2 puck stacking to 3 puck stacking. Here, the weight sharing agent continually places two pucks on top of one another, then lifts the top puck and places it back, which leads to slower learning than in the no-transfer baseline. Options do not suffer from this problem.

As reported in Figure 4.5, the learned partitions for the stairs building and component arrangement tasks underperform compared to puck stacking. Regardless, I observed a speed-up compared to the no-transfer baseline in all experiments except for the transfer from stairs from 3 pucks to 3 puck component. Options also outperform weight sharing in 3 out of 5 experiments with the non-optimal partitions, albeit not significantly.

4.4 Discussion

I return to the questions posited at the beginning of this section. Online Partition Iteration can find the right partition of the state-action space as long as the individual state-action blocks are clearly identifiable. For tasks with more complex classification boundaries, the partitions found are suboptimal, but still useful. I showed that options speed-up learning and outperform the baselines in the majority of the transfer experiments. My algorithm also outperformed the only previous method of the same kind [3] in terms of finding a consistent partition.

The main drawback of Online Partition Iteration is that is it highly influenced by the accuracy of the classifier g. During state projection, it takes only one incorrectly classified action (out of 12544 actions used in my pucks world experiments) for the state block classification to be erroneous. Confidence thresholding helps in the task of stacking pucks (Figure 4.2b), as it can filter out most of the errors. However, trained classifiers for the other two tasks, arranging components and building stairs, often produce incorrect predictions with a high confidence.

Moreover, the errors during state projection get amplified as the partitioning progresses. Note that the dataset of state-action pairs (inputs) and state-action blocks (classes) is created based on the previous state partition, which is predicted by the classifier. In other words, the version of the classifier g at step t generates the classes that will be used for its training at step $t+1$. A classifier trained on noisy labels is bound to make even more errors at the next iteration. In particular, I observed that the error rate grows exponentially in the number of steps required to partition the state-action space.

In these cases, the partitioning algorithm often stops because of the limit on the number of state-action blocks (10 for the pucks domain). That is why the performance for the component arrangement and stairs building tasks is not sensitive to the state-action block size threshold (Figure 4.2a). Nevertheless, these noisy partitions also help with transfer learning, as shown in Table 4.1.

Chapter 5

Proof of Correctness of Online Partition Iteration

I prove the correctness of Online Partition Iteration with a 1-nearest-neighbor classifier that has access to an infinite stream of experience. The proofs follow the theoretical analysis of bisimulation by Givan et al. [5], as there any many parallels between algorithms for finding bisimulations and homomorphisms.

The first lemma and corollary ensure that Partition Iteration over the state-action space Φ finds reward respecting SSP partitions.

Lemma 1. Given a reward respecting partition B of an MDP $M = \langle S, A, \Phi, P, R \rangle$ and $(s_1, a_1), (s_2, a_2) \in \Phi$ such that $T(s_1, a_1, [s']_{B|S}) \neq T(s_2, a_2, [s']_{B|S})$ for some $s' \in S$, (s_1, a_1) and (s_2, a_2) are not in the same block of any reward respecting SSP partition refining B.

Proof. Following the proof of Lemma 8.1 from [5]: proof by contradiction.

Let B' be a reward respecting SSP partition that is a refinement of B. Let $s' \in S$, $(s_1, a_1), (s_2, a_2) \in b \in B$ such that $T(s_1, a_1, [s']_{B|S}) \neq T(s_2, a_2, [s']_{B|S})$. Define B' such that $(s_1, a_1), (s_2, a_2)$ are in the same block and $[s']_{B|S} = \bigcup_{i=1}^{k} [s'_i]_{B'|S}$. Because B' is a reward respecting SSP partition, for each state block $[s'']_{B|S} \in B'|S$, $T(s_1, a_1, [s'']_{B|S}) = T(s_2, a_2, [s'']_{B|S})$. Then, $T(s_1, a_1, [s']_{B|S}) = \sum_{1 \leq i \leq k} T(s_1, a_1, [s'_i]_{B'|S}) = \sum_{1 \leq i \leq k} T(s_2, a_2, [s'_i]_{B'|S}) = T(s_2, a_2, [s']_{B|S})$. This contradicts $T(s_1, a_1, [s']_{B|S}) \neq T(s_2, a_2, [s']_{B|S})$.

□

Corollary 1. Let B be a reward respecting partition of an MDP $M = \langle S, A, \Phi, P, R \rangle$, b a block in B and c a union of blocks from $B|S$. Every reward respecting SSP partition over Φ that refines B is a refinement of the partition $Split(b, c, B)$.

Proof. Following the proof of Corollary 8.2 from [5].

Let $c = \bigcup_{i=1}^{n} [s_i]_{B|S}$, $[s_i]_{B|S} \in B|S$. Let B' be a reward respecting SSP partition that refines B. $Split(b, c, B)$ will only split state-action pairs $(s_1, a_1), (s_2, a_2)$ if $T(s_1, a_1, c) \neq T(s_2, a_2, c)$. But if $T(s_1, a_1, c) \neq T(s_2, a_2, c)$, then there must be some k such that $T(s_1, a_1, c_k) \neq T(s_2, a_2, c_k)$ because for any $(s, a) \in \Phi$, $T(s, a, c) = \sum_{1 \leq m \leq n} T(s, a, c_m)$. Therefore, we can conclude by Lemma 1 that $[(s_1, a_1)]_{B'} \neq [(s_2, a_2)]_{B'}$.

□

The versions of Partition Iteration from [5] and [6] partition a fully-defined MDP. I designed my algorithm for the more realistic case, where only a stream of experience is available. This change makes the algorithm different only during State Projection (Algorithm 4). In the next lemma, I prove that the output of State Projection converges to a state partition as the number of experienced transitions goes to infinity.

Lemma 2. Let $M = \langle S, A, \Phi, P, R \rangle$ be an MDP with a finite A, a finite or infinite S, a state-action space Φ that is a separable metric space and a deterministic P defined such that each state-action pair is visited with a probability greater than zero. Let $SampleAction(s) = A_s, \forall s \in S$ (Algorithm 4, line 5). Let t_1, t_2, ... be i.i.d. random variables that represent observed transitions, g a 1 nearest neighbor classifier that classifies state-action pairs into state-action blocks and let $(s, a)_n$ the nearest neighbor to (s, a) from a set of n transitions $X_n = \{t_1, t_2, ..., t_n\}$. Let B_n be a state-action partition over X_n and $S_n = \bigcup_{t \in X_n} t.next\_state$. Let $(B_n|S_n)'$ be a state partition obtained by the State Projection algorithm with g taking neighbors from X_n. $(B_n|S_n)' \to B_n|S_n$ as $n \to \infty$ with probability one.

Proof. $B_n|S_n$ is obtained by projecting B_n onto S_n. In this process, S_n is divided into blocks based on $B(s) = \{[(s', a)]_{B_n} | (s', a) \in \Phi, s = s'\}$, the set of distinct blocks containing pairs of which s is a component, $s \in S_n$. Given $SampleAction(s) = A_s, \forall s \in S_n$, line 8 in Algorithm 4 predicts a $b \in B$ for each $(s', a) \in \Phi$, such that $s = s'$. By the Convergence of the Nearest Neighbor Lemma [57], $(s, a)_n$ converges to (s, a) with probability one. The rest of the Algorithm 4 exactly follows the projection procedure (Definition 3), therefore, $(B_n|S_n)' \to B_n|S_n$ with probability one.

□

Finally, I prove the correctness of my algorithm given an infinite stream of i.i.d. experience. While the i.i.d. assumption does not usually hold in Reinforcement Learning, the Deep Reinforcement Learning literature often leverages the experience buffer [37] to ensure the training data is diverse enough. My algorithm also contains a large experience buffer to collect the data needed to run Online Partition Iteration.

Theorem 3 (Correctness)**.** Let $M = \langle S, A, \Phi, P, R \rangle$ be an MDP with a finite A, a finite or infinite S, a state-action space Φ that is a separable metric

space and a deterministic P defined such that each state-action pair is visited with a probability greater than zero. Let $SampleAction(s) = A_s, \forall s \in S$ (Algorithm 4, line 5). Let t_1, t_2, ... be i.i.d. random variables that represent observed transitions, g a 1 nearest neighbor classifier that classifies state-action pairs into state-action blocks. As the number of observed t_i goes to infinity, Algorithm 3 computes a reward respecting SSP partition over the observed state-action pairs with probability one.

Proof. Loosely following the proof of Theorem 8 from [5].

Let B be a partition over the observed state-action pairs, S the set of observed states and (B—S)' the result of StateProjection(B,g) (Algorithm 4).

Algorithm 3 first splits the initial partition such that a block is created for each set of transitions with a distinct reward (line 2). Therefore, Algorithm 3 refines a reward respecting partition from line 2 onward.

Algorithm 1 terminates with B when $B = Split(b, [s]_{(B|S)'}, B)$ for all $b \in B$, $[s]_{(B|S)'} \in (B|S)'$. $Split(b, [s]_{(B|S)'}, B)$ will split any block b containing $(s_1, a_1), (s_2, a_2)$ for which $T(s_1, a_1, [s]_{(B|S)'}) \neq T(s_2, a_2, [s]_{(B|S)'})$. According to Lemma 2, $(B|S)' \to B|S$ as $N \to \infty$ with probability one. Consequently, any partition returned by Algorithm 3 must be a reward respecting SSP partition.

Since Algorithm 3 first creates a reward respecting partition, and each step only refines the partition by applying $Split$, we can conclude by Corollary 1 that each partition encountered, including the resulting partition, must contain a reward respecting SSP partition.

□

Conclusion

The aim of my work was to create a novel abstraction algorithm that could find MDP homomorphisms "online" (i.e. from a stream of experience). I fully completed this task, both in analyzing the theoretical basis of abstraction and state-of-the-art abstraction methods, as well as in developing a new algorithm called Online Partition Iteration that satisfies all stated requirements. I rigorously tested this algorithm in three different environments–puck stacking in a grid world, placing a block in a target position, and various arrangement tasks in a continuous pucks world–and proved its correctness under certain assumptions. My algorithm also outperformed the only previous online method for finding MDP homomorphisms [58].

As I discussed in Section 4.4, choosing the right state-action classifier g was a major challenge because my partitioning algorithm is highly susceptible to errors made by the classifier. I addressed this problem by choosing the correct architecture and training settings for g, but also by changing the partitioning algorithm itself to be more robust. In particular, I implemented a balancing procedure to make sure g pays equal attention to all classes and thresholded the confidence of the classifier. Modern neural networks tend to severely overestimate or underestimate their confidence; hence, I included a simple confidence calibration algorithm [52] to make sure the confidence predicted by g are meaningful.

A promising direction of future work is learning the abstract MDP in a single training run of a neural network. This approach would prevent the errors made by the classifier from growing exponentially with each step of Partition Improvement (see Section 4.4). One approach could be learning a model of the environment while imposing some kind of a compression constraint on the latent representation of the model (e.g. [59]). That is, the model would be incentivized to use as few states and as few actions as possible to encode the dynamics of the environment. This constraint matches the goal of my algorithm: find an abstract MDP as compact as possible that encodes all the important dynamics of the original MDP.

Bibliography

[1] Platt, R.; Kohler, C.; et al. Deictic Image Maps: An Abstraction For Learning Pose Invariant Manipulation Policies. *CoRR*, volume abs/1806.10045, 2019.

[2] Chen, L.; Papandreou, G.; et al. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Trans. Pattern Anal. Mach. Intell.*, volume 40, no. 4, 2018: pp. 834–848, doi:10.1109/TPAMI.2017.2699184.

[3] Wolfe, A. P.; Barto, A. G. Decision Tree Methods for Finding Reusable MDP Homomorphisms. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, AAAI Press, 2006, ISBN 978-1-57735-281-5, pp. 530–535.

[4] Spong, M. W. *Robot Dynamics and Control*. New York, NY, USA: John Wiley & Sons, Inc., first edition, 1989, ISBN 047161243X.

[5] Givan, R.; Dean, T.; et al. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, volume 147, no. 1, 2003: pp. 163 – 223, ISSN 0004-3702, doi:https://doi.org/10.1016/S0004-3702(02)00376-4, planning with Uncertainty and Incomplete Information.

[6] Ravindran, B. *An Algebraic Approach to Abstraction in Reinforcement Learning*. Dissertation thesis, 2004, aAI3118325.

[7] Biza, O.; Platt, R. Online Abstraction with MDP Homomorphisms for Deep Learning. In *Proceedings of The 18th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '19, Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[8] Mahler, J.; Pokorny, F. T.; et al. Dex-Net 1.0: A cloud-based network of 3D objects for robust grasp planning using a Multi-Armed Bandit model

with correlated rewards. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016: pp. 1957–1964.

[9] Mahler, J.; Liang, J.; et al. Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics. *CoRR*, volume abs/1703.09312, 2017.

[10] Levine, S.; Finn, C.; et al. End-to-end Training of Deep Visuomotor Policies. *J. Mach. Learn. Res.*, volume 17, no. 1, Jan. 2016: pp. 1334–1373, ISSN 1532-4435.

[11] ten Pas, A.; Gualtieri, M.; et al. Grasp Pose Detection in Point Clouds. *Int. J. Rob. Res.*, volume 36, no. 13-14, Dec. 2017: pp. 1455–1473, ISSN 0278-3649, doi:10.1177/0278364917735594.

[12] Fang, K.; Zhu, Y.; et al. Learning Task-Oriented Grasping for Tool Manipulation from Simulated Self-Supervision. *CoRR*, volume abs/1806.09266, 2018.

[13] Zeng, A.; Song, S.; et al. Learning Synergies Between Pushing and Grasping with Self-Supervised Deep Reinforcement Learning. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018: pp. 4238–4245.

[14] Lee, M. A.; Zhu, Y.; et al. Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks. *CoRR*, volume abs/1810.10191, 2018.

[15] Russell, S.; Norvig, P. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, third edition, 2009, ISBN 0136042597, 9780136042594.

[16] Kaelbling, L. P.; Littman, M. L.; et al. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, volume 101, no. 1, 1998: pp. 99 – 134, ISSN 0004-3702, doi:https://doi.org/10.1016/S0004-3702(98)00023-X.

[17] Sutton, R. S.; Precup, D.; et al. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, volume 112, no. 1, 1999: pp. 181 – 211, ISSN 0004-3702, doi:https://doi.org/10.1016/S0004-3702(99)00052-1.

[18] Bellman, R. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, volume 6, no. 5, 1957: pp. 679–684, ISSN 00959057, 19435274.

[19] Dean, T.; Givan, R. Model Minimization in Markov Decision Processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial*

Intelligence, AAAI'97/IAAI'97, AAAI Press, 1997, ISBN 0-262-51095-2, pp. 106–111.

[20] Dean, T.; Givan, R.; et al. Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, UAI'97, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN 1-55860-485-5, pp. 124–131.

[21] Ferns, N.; Panangaden, P.; et al. Metrics for Finite Markov Decision Processes. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, Arlington, Virginia, United States: AUAI Press, 2004, ISBN 0-9749039-0-6, pp. 162–169.

[22] Li, L.; Walsh, T. J.; et al. Towards a Unified Theory of State Abstraction for MDPs. In *ISAIM*, 2006.

[23] Jong, N. K.; Stone, P. State Abstraction Discovery from Irrelevant State Variables. In *IJCAI*, 2005.

[24] Ravindran, B.; G. Barto, A. Approximate Homomorphisms: A framework for non-exact minimization in Markov Decision Processes. 2003.

[25] Taylor, J. J.; Precup, D.; et al. Bounding Performance Loss in Approximate MDP Homomorphisms. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, USA: Curran Associates Inc., 2008, ISBN 978-1-6056-0-949-2, pp. 1649–1656.

[26] Mccallum, A. K. *Reinforcement Learning with Selective Perception and Hidden State*. Dissertation thesis, 1996, aAI9618237.

[27] Harlow, H. F. The Formation of Learning Sets. *Psychological Review*, volume 56, no. 1, 1949: pp. 51–65.

[28] Lake, B. M.; Ullman, T. D.; et al. Building Machines That Learn and Think Like People. *CoRR*, volume abs/1604.00289, 2016, 1604.00289.

[29] Pan, S. J.; Yang, Q. A Survey on Transfer Learning. *IEEE Trans. on Knowl. and Data Eng.*, volume 22, no. 10, Oct. 2010: pp. 1345–1359, ISSN 1041-4347, doi:10.1109/TKDE.2009.191.

[30] Weiss, K.; Khoshgoftaar, T. M.; et al. A survey of transfer learning. *Journal of Big Data*, volume 3, no. 1, May 2016: p. 9, ISSN 2196-1115, doi:10.1186/s40537-016-0043-6.

[31] Taylor, M. E.; Stone, P. Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.*, volume 10, Dec. 2009: pp. 1633–1685, ISSN 1532-4435.

[32] Castro, P. S.; Precup, D. Automatic Construction of Temporally Extended Actions for MDPs Using Bisimulation Metrics. In *Proceedings of the 9th European Conference on Recent Advances in Reinforcement Learning*, EWRL'11, Berlin, Heidelberg: Springer-Verlag, 2012, ISBN 978-3-642-29945-2, pp. 140–152, doi:10.1007/978-3-642-29946-9_16.

[33] Soni, V.; Singh, S. Using Homomorphisms to Transfer Options Across Continuous Reinforcement Learning Domains. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, AAAI Press, 2006, ISBN 978-1-57735-281-5, pp. 494–499.

[34] Sorg, J.; Singh, S. Transfer via Soft Homomorphisms. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, ISBN 978-0-9817381-7-8, pp. 741–748.

[35] Rajendran, S.; Huber, M. Learning to generalize and reuse skills using approximate partial policy homomorphisms. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, Oct 2009, ISSN 1062-922X, pp. 2239–2244, doi:10.1109/ICSMC.2009.5345891.

[36] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN 0387310738.

[37] Mnih, V.; Kavukcuoglu, K.; et al. Human-level control through deep reinforcement learning. *Nature*, volume 518, no. 7540, 2015: pp. 529–533, doi:10.1038/nature14236.

[38] Sutton, R. S.; Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[39] Schaul, T.; Quan, J.; et al. Prioritized Experience Replay. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[40] van Hasselt, H.; Guez, A.; et al. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, pp. 2094–2100.

[41] Wang, Z.; Schaul, T.; et al. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 1995–2003.

[42] Bellemare, M. G.; Dabney, W.; et al. A Distributional Perspective on Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 2017, pp. 449–458.

[43] Hessel, M.; Modayil, J.; et al. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 2018, pp. 3215–3222.

[44] Schulman, J.; Levine, S.; et al. Trust Region Policy Optimization. *CoRR*, volume abs/1502.05477, 2015, 1502.05477.

[45] Li, Y. Deep Reinforcement Learning. *CoRR*, volume abs/1810.06339, 2018, 1810.06339.

[46] Long, J.; Shelhamer, E.; et al. Fully Convolutional Networks for Semantic Segmentation. *CoRR*, volume abs/1411.4038, 2014, 1411.4038.

[47] Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, 2015, pp. 448–456.

[48] Yu, F.; Koltun, V. Multi-Scale Context Aggregation by Dilated Convolutions. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[49] Yu, F.; Koltun, V.; et al. Dilated Residual Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, 2017, pp. 636–644, doi:10.1109/CVPR.2017.75.

[50] He, K.; Zhang, X.; et al. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016: pp. 770–778.

[51] Buda, M.; Maki, A.; et al. A systematic study of the class imbalance problem in convolutional neural networks. *Neural networks : the official journal of the International Neural Network Society*, volume 106, 2018: pp. 249–259.

[52] Guo, C.; Pleiss, G.; et al. On Calibration of Modern Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning,*

ICML 2017, Sydney, NSW, Australia, 6-11 August 2017, 2017, pp. 1321–1330.

[53] Platt, J. C. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*, MIT Press, 1999, pp. 61–74.

[54] Yosinski, J.; Clune, J.; et al. How Transferable Are Features in Deep Neural Networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, Cambridge, MA, USA: MIT Press, 2014, pp. 3320–3328.

[55] Pedregosa, F.; Varoquaux, G.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830.

[56] Dhariwal, P.; Hesse, C.; et al. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[57] Cover, T.; Hart, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, volume 13, no. 1, January 1967: pp. 21–27, ISSN 0018-9448, doi:10.1109/TIT.1967.1053964.

[58] Wolfe, A. P. Defining Object Types and Options Using MDP Homomorphisms. 2006.

[59] Tishby, N.; Pereira, F. C.; et al. The Information Bottleneck Method. 1999, pp. 368–377.

Appendix A

Acronyms

RL Reinforcement Learning

Deep RL Deep Reinforcement Learning

MDP Markov Decision Process

CMP Controlled Markov Process

SSP Stochastic Substitution Property

DRN Deep Residual Network

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `thesis.pdf` ..... the thesis text in PDF format
- `src` ..... the directory of source codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `abstraction` ..... the directory of the implementation
    - `README.md` ..... instructions regarding the implementation