ONLINE ABSTRACTION WITH MDP HOMOMORPHISMS FOR DEEP LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Abstraction of Markov Decision Processes is a useful tool for solving complex problems, as it can ignore unimportant aspects of an environment, simplifying the process of learning an optimal policy. In this paper, we propose a new algorithm for finding abstract MDPs in environments with continuous state spaces. It is based on MDP homomorphisms, a structure-preserving mapping between MDPs. We demonstrate our algorithm's ability to learns abstractions from collected experience and show how to reuse the abstractions to guide exploration in new tasks the agent encounters. Our novel task transfer method beats a baseline based on a deep Q-network.

1 INTRODUCTION

Abstraction is a useful tool for effective control in complex environments. Instead of learning an entangled and uninterpretable policy with a deep neural network, as is the current practice in the deep reinforcement learning literature, we can abstract away unimportant details, which allows us to learn a much simpler policy. Such a policy can be examined by hand in its full form.

There are two approaches to abstraction: temporal abstraction and abstraction of the state space. The former is best suited for tasks that take hundreds or thousands of time steps to solve, as it creates temporally-extended actions that mitigate the difficulties of making decisions over long time periods. The latter groups together states that exhibit similar behaviors, which decreases the size of the state spaces and allows for simpler policies. The abstracted state space is the smallest when the actions are temporally extended, hence, temporal abstraction complements state abstraction. We focus on state abstraction in this paper.

To find useful abstraction, we adopt a theoretical framework called Markov Decision Process (MDP) homomorphism (Ravindran (2004)): a mapping between two MDPs that preserves their structure. Our goal is to find the smallest MDP homomorphic to the underlying problem. This can also be viewed as model minimization with MDP homomorphism serving the role of an equivalence relation between models–MDPs. We call the minimized MDP an *abstract MDP* because it abstracts away many unnecessary distinctions between states from the underlying MDP. Instead of proposing an ad-hoc algorithm, our approach leverages the theoretical foundations of Ravindran (2004).

We focus on solving end-to-end manipulation tasks from robotics characterized by continuous high-dimensional state spaces (i.e. the input from an RGB or a depth camera) and discrete high-dimensional action spaces: all possible positions where the robot can execute an action. Such environments are best handled with convolutional neural networks, which can learn from high-dimensional inputs without overfitting due to extensive weight sharing and various other desirable properties. However, our algorithm does not depend on a particular model–we replace the convolutional network with a decision tree in one experiment.

In this paper, we propose a new approach to learning abstract MDPs from experience. These are our key contributions:

- We propose an algorithm for creating abstract MDPs from experience in both discrete and continuous state spaces (Subsection 5.1). The algorithm first explores the environment with either a random uniform policy or a deep Q-network (Mnih et al. (2015)). Subsequently,

it generates an abstract MDP homomorphic to the underlying MDP and plans the optimal actions in the abstract MDP. It is limited to deterministic MDPs in its current version.

- We develop a classifier based on a convolutional network that learns to sort state-action pairs based on their behavior. We include several augmentations, such as sharing the weights of previously learned models and oversampling minority classes, for speeding-up learning and dealing with extreme class imbalance (Subsection 5.2).
- We propose a method for guiding exploration in a new task with a previously learned abstract MDP (Subsection 5.5). Our method is based on the framework of options (Sutton et al. (1999)); it can augment any existing reinforcement learning agent with a new set of temporally-extended actions. The method beats a baseline based on a deep Q-network in one class of tasks and performs equally well in another.

2 Related Work

Ravindran (2004) proposed Markov Decision Process (MDP) homomorphism together with a sketch of an algorithm for finding homomorphisms (i.e. finding the minimal MDP homomorphic to the underlying MDP) given the full specification of the MDP. The first and only algorithm (to the best of our knowledge) for finding homomorphisms from experience (Wolfe & Barto (2006)) operates over Controlled Markov Processes (CMP), an MDP extended with an output function that provides more supervision than the reward function alone. Homomorphism over CMPs was also used in Wolfe (2006) to find objects that react the same to a defined set of actions.

An approximate MDP homomorphism (Ravindran & Barto (2004)) allows aggregating together state-action pairs with similar, but not the same dynamics. It is essential when learning homomorphisms from experience in non-deterministic environments because the estimated transition probabilities for individual state-action pairs will rarely be the same, which is required by the MDP homomorphism. Taylor et al. (2008) built upon this framework by introducing a similarity metric for state-action pairs as well as an algorithm for finding approximate homomorphisms.

Sorg & Singh (2009) developed a method based on homomorphism for transferring a predefined optimal policy to a similar task. However, their method maps only states and not actions, requiring actions to behave the same across all MDPs. Soni & Singh (2006) and Rajendran & Huber (2009) also studied skill transfer in the framework of MDP homomorphisms.

3 Background

An agent's interaction with an environment can be modeled as a Markov Decision Process (MDP, Bellman (1957)). 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), $P(s, a, s')$ is the transition function and $R(s, a)$ is the reward function.

We aim to find minimal MDPs that retain the structure of the underlying MDP. An MDP homomorphism captures this intuition:

Definition 1 (Ravindran (2004))**.** 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.

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

Theorem 1 (Optimal value equivalence, Ravindran (2004))**.** 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 $(s, a) \in \Phi$, $Q^*(s, a) = Q^*(f(s), g_s(a))$.

During MDP minimization, Φ is partitioned and the partition subsequently induces the abstract MDP. Let $B = \{b_1, b_2, ..., b_N\}$ be the partition over Φ. We call B the *state-action partition*, with b_i being a block of state-action pairs. The state-action partition induces the *quotient MDP*:

Definition 2. 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 $R'([s]_{B|S}, a'_i) = R(s, a_i)$.

We omit the definition of the reward respecting SSP partition for brevity, please find it in Ravindran (2004). We call the induced quotient MDP an *abstract MDP*–it ignores many unimportant distinctions from the underlying MDP.

Finally, we adopt the framework of options (Sutton et al. (1999)) for the purpose of transferring learned homomorphisms between similar tasks. An option $\langle I, \pi, \beta \rangle$ is a temporally extended action; it can be executed from the set of states I, the individual primitive actions are selected with a policy π and each state it encounters has a probability of terminating the option, which is given by $\beta : S \rightarrow [0, 1]$.

4 Problem Formulation

Our algorithm is intended to abstract MDP with the following properties:

- The reward function is sparse. In particular, there is a set of goal states with an arbitrary positive reward and the rest of the states are assigned zero or negative reward.
- The transition and reward functions are deterministic.

Note that both the transition and the reward functions change when a new set of goal states is selected because the set of terminal states changes.

5 Methods

We define our base algorithm for partitioning a continuous state space in Subsection 5.1 and propose several modifications to increase its speed and robustness in Subsection 5.2. Subsection 5.3 and Subsection 5.4 describe how our approach acts and collects experience in an environment. Finally, we propose a method for guiding exploration with an abstract model in a new task in Subsection 5.5.

5.1 Partitioning algorithm

We adapt the Partition Iteration algorithm–originally developed for stochastic bisimulation based partitioning (Givan et al. (2003)) for the purpose of finding minimal MDPs homomorphic to the underlying MDP (Algorithm 1). The algorithm starts with a reward-respecting partition obtained by separating transitions that receive a positive reward from the ones that do not (*SplitRewards*). The reward-respecting partition is subsequently refined with the *Split* operation until it is consistent. *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 2) is the most critical part of our method. We train a classifier g, which can be an arbitrary model, to predict the state-action block given a state-action pair. The training set consists of all transitions the agent experienced, with each transition belonging to a particular state-action block. During the state projection, g evaluates a state under a sampled set of actions, predicting all possible state-action blocks that can be executed from the given state. For discrete action spaces, the set of actions should include all available actions. The set of executable state-action blocks determines which state block the state belongs to–each block in the state partition is identified by a set of state-action blocks that can be executed from it.

5.2 Speeding-up and increasing robustness

Retraining the classifier g from scratch at every step of Partition iteration can be time-consuming (especially if g is a neural network). Moreover, if the task is sufficiently complex, the classifier will make mistakes during the state projection. For these reasons, we developed several modifications to

Algorithm 1 Partition iteration

Input Experience E, classifier g.

procedure PARTITIONITERATION
  $B \leftarrow \{E\}, B' \leftarrow \{\}$
  $B \leftarrow SplitRewards(B)$
  $g \leftarrow TrainClassifier(B, g)$
  while $B \neq B'$ **do**
    $S/B \leftarrow Project(B, g)$
    $B' \leftarrow B$
    for block c in S/B **do**
      while B contains block b for which $B \neq Split(b, c, B)$ **do**
        $B \leftarrow Split(b, c, B)$
      end while
    end for
  end while
end procedure

Algorithm 2 State projection

Input State-action partition B, classifier g.
Output State partition S/B.

procedure PROJECT
  $S/B \leftarrow \{\}$
  for block b in B **do**
    for transition t in b **do**
      $A_s \leftarrow SampleActions(t.state)$
      $B_s \leftarrow \{\}$
      for action a in A_s **do**
        $p \leftarrow g.predict(t.state, a)$
        $B_s \leftarrow B_s \cup \{p\}$
      end for
      $Add\ t\ to\ S/B\ using\ B_s\ as\ the\ key$
    end for
  end for
end procedure

our partition algorithm that increase its speed and robustness. Some of them are specific to a neural network classifier.

- **weight reuse**: The neural network is initialized with the number of logits (outputs) equal to the maximum number of state-action blocks. First, the neural network is trained from scratch, but its weights are retained for all subsequent re-trainings, with old state-action blocks being assigned to the same logits and new state-action blocks to free logits.
- **early stopping of training**: We reserve 20% of the experience to calculate the validation accuracy. Having a measure of performance, we can select the snapshot of the neural network that performs the best and stop the training after no improvement for *N* steps.
- **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. We follow the best practices from Buda et al. (2018) and over-sample all minority classes so that the number of samples for each class is equal to the size of the majority class. We found decision trees do not require oversampling, hence we use this method only with a convolutional network.
- **state-action block size threshold**: During State projection, the classifier g sometimes makes mistakes in classifying a state-action pair to a state-action block. Hence, some transitions have an invalid next state block associated with them. To prevent the *Split* function from over-segmenting the state-action partition due to various misclassifications, we only create state-action blocks that contain a number of samples higher than a threshold T_a.

5.3 ACTING IN THE ENVIRONMENT

We observe that the state-action partition provides enough information to act in the environment. In particular, we do not need to evaluate in which abstract state the agent finds itself, the list of executable state-action blocks obtained by classifying all available actions in the current state with the classifier g suffices.

To decide which state-action block should be executed next, a directed graph of state action block that we call the *abstract graph* is constructed. The state-action blocks constitute the nodes of the abstract graph and each state-action blocks is connected to all state-action blocks that are available after it is executed by directed edges. The state-action block that leads to the goal set of states is marked as the goal node and we run a breadth-first search to compute the shortest path from each node to the goal.

Having constructed the abstract graph, our agent can choose the best state-action block to execute based on the length of the shortest path to the goal from each node. Similarly to the state projection, we sample actions available in the current state and use the classifier g to classify each action together with the current state–forming a state-action pair–to a state-action block. The agent selects the state-action block with the shortest path to the goal.

Note that multiple actions may be classified into a single state-action block. Since the assignment can be wrong, selecting the same action each time a particular state is encountered can lead to an undesired behavior. Instead, we sample from the set of actions belonging to the selected state-action block in proportion to the confidence of the classifier about each action. We also tested selecting an action from the state-action block at random, which performed slightly worse than the former strategy, and selecting the action with the highest confidence, which leads to a significant drop in performance.

5.4 Collecting experience

The initial run of the partitioning algorithm needs experience diverse enough to segment all possible success and failure modes of the task at hand. If the environment is simple, a random exploration policy provides such experience. However, in many types of tasks, a random policy rarely achieves the goal, making it hard for the classifier g to learn which state-action pairs lead to a positive reward. We address this issue by using the vanilla version of the deep Q-network (DQN, Mnih et al. (2015)) to generate the initial experience.

5.5 Transferring abstract MDPs

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

If we have the prior knowledge that the goal from the previous task is on the path to the goal of the new task, we can simply take the agent to the goal using the previously learn abstract MDP and continue from there. However, we do not need the whole abstract MDP to achieve this; any policy that reaches the goal of the first task suffices.

Our main contribution lies in solving the second case, where no prior information is available. We create $|S/B|$ options, each taking the agent to a particular state block in the abstract MDP. Each option is a tuple $\langle I, \pi, \beta \rangle$: the set of starting state I contains all starting states of the MDP for the new task, π uses the abstract graph to select actions that lead to a particular state block in the abstract MDP and the terminal condition β terminates the option when the selected state block is reached. The agent learns the q-values of the options with a Monte Carlo update (Sutton & Barto (1998)) 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.

6 Experiments

Our algorithm is intended to run in continuous state space environments. However, to compare with prior work, we also developed a version for discrete environments. We test our algorithm's ability to solve a single task and compare it with prior work in Subsection 6.2 and experiment with two kinds of task transfer in Subsection 6.3. All our neural networks were implemented in Tensorflow[1]. The source code for our work is available online, we will link it after the end of the review period.

6.1 Setup

We first describe the two environments in which we test our agent. All experiments are repeated 20 times and we plot the averaged results.

[1] `https://www.tensorflow.org/`, version 1.7.0

Puck stacking

For our continuous task, we chose stacking pucks in a grid world environment (see Figure 1d) with end-to-end actions: the agent can attempt to pick or place a puck in any of the cells. Executing the pick action in a cell with a puck places it into the agent's hand, whereas trying to pick an empty cell does not alter the state of the environment. The place action is allowed everywhere as long as the agent is holding a puck. The agent senses the environment as a depth image with a size $(28 * C)^2$, with C being the number of cells along an axis, together with a one-hot encoding of the state of its 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.

A convolutional network is used as the g classifier; we describe our architecture in Appendix A. The agent collects experience either with a random uniform policy or a deep Q-network (Mnih et al. (2015)) of the same structure as the g classifier, except for the number of output neurons. We list the settings of the deep Q-network in Appendix B. The number of state-action blocks is limited to 10: the purpose of the limit is mostly to speed-up faulty experiments, but an over-segmented partition can still perform well in some cases. The replay buffers of the partitioning algorithm and the DQN are both limited to 10000 transitions due to memory constraints.

Blocks world

We implemented the blocks world environment from Wolfe & Barto (2006). 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 the goal position and height. With 4 positions and 3 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.

Although a neural network can learn to solve this task, a decision tree trains two orders of magnitude faster and often reaches better performance. We use a decision tree from the scikit-learn package[2] with the default settings as our g classifier. All modifications from Subsection 5.2 specific to a neural network are omitted: weight reuse, early stopping of training and class balancing. We 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, we cap the number of state-action blocks at 100 to avoid creating thousands of state-action pairs if the algorithm fails. The abstract MDP is recreated every 3000 time steps and the task terminates after 15000 time steps.

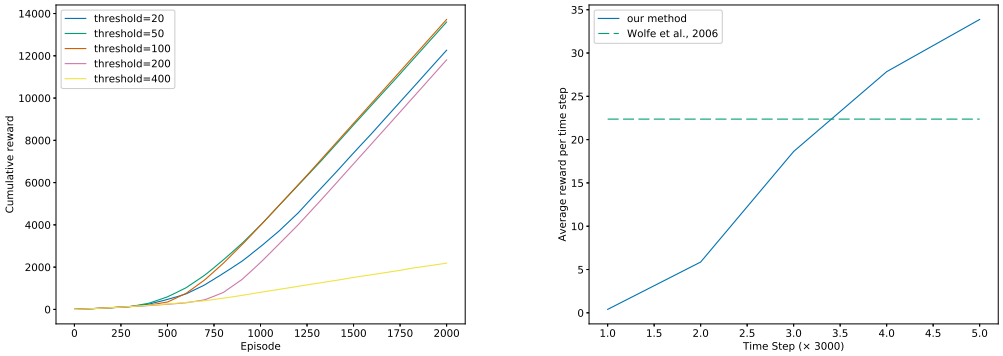
6.2 SINGLE TASK

We first demonstrate our algorithm's ability to find the minimal MDP homomorphic to the underlying problem and to act optimally given the minimal MDP. The agent is taught to solve the task of stacking 3 pucks in a 3x3 grid world environment; a random uniform policy selects actions in the first 100 episodes and we re-partition the minimal MDP every 100 episodes thereafter.

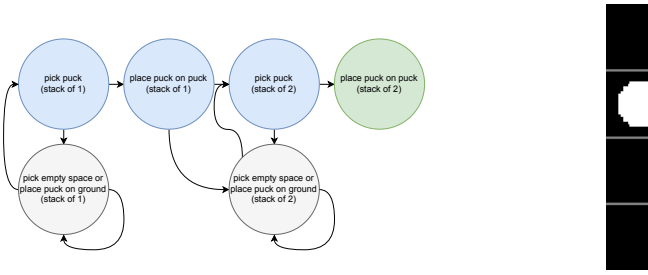
Figure 1a shows the cumulative reward of our algorithm over 2000 episodes with five different settings of the state-action block size threshold. If the threshold is set too high, the algorithm cannot split the necessary blocks, which leads to under-segmentation. A policy planned in an under-segmented abstract MDP does not fit well to the underlying MDP. All other settings eventually achieve a nearly optimal behavior, but the higher the threshold the longer it takes because more experience needs to be collected. Conversely, if the threshold is set too low our algorithm becomes more susceptible to misclassification and the performance drops. The minimal MDP learned by our algorithm contains 6 state-action blocks, see Figure 1c for an annotated abstract graph. Each node in the graph represent a state-action block and the arrows point to the blocks available after executing a particular state-action block.

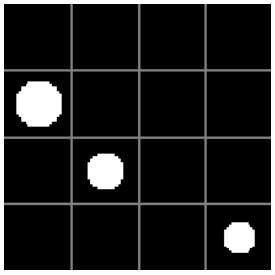
In figure 1b, we compare the decision tree version of our algorithm (as described in Subsection 6.1) with the results reported in Wolfe & Barto (2006). There are several differences between our experiments: the algorithm described in Wolfe & Barto (2006) works with a Controlled Markov

[2] `http://scikit-learn.org`, version 0.19.2



(a) (b)





(c) (d)

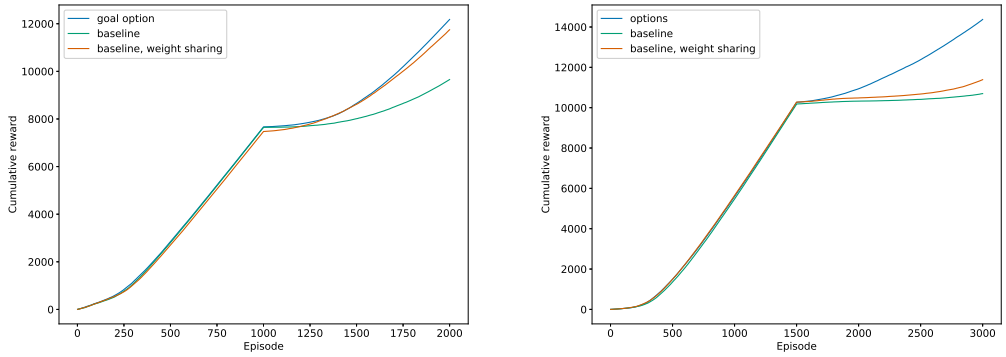
Figure 1: Experiment results for creating an abstract MDP for a single task: a) cumulative rewards over time for several state-action block size threshold settings for the task of stacking three pucks, b) comparison with Wolfe & Barto (2006) in the Blocks World environment, c) the minimal abstract MDP for the task of stacking three pucks learned by our algorithm, d) an example of the input to the neural network. All results are averaged over 20 runs with different random seeds.

Process (CMP), an MDP augmented with an output function that provides more supervision than the reward. Therefore, their algorithm can start segmenting state-action blocks before it even observes the reward. CMP also allows an easy transfer of the learned partition from one task to another; we solve each task separately. On the other hand, each action in the Wolfe's version of the task has a 0.2 chance of failure, but we omit this detail to satisfy the assumptions of our algorithm. Even though each version of the task is easier in some ways are harder in other, we believe the comparison with the only previous algorithm that solves the same problem is valuable.

6.3 Task transfer

We first test our 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 4x4 grid world environment and the second task requires the agent to stack three pucks. Both of the tasks run for 1000 episodes. We compare our method, which first executes the option that brings the agent to the goal of the first task and then acts with a vanilla deep Q-network, with two baselines:

- **baseline**: A vanilla deep Q-network. We reset its weights and replay buffer after the end of each task–it does not retain any information from the previous task it solved.
- **baseline, weight sharing**: The same as the above, but we do not reset its weights. When a new task starts, it goes to the goal state of the previous tasks and explores from there.



(a) (b)

Figure 2: Cumulative rewards for task transfer experiments: a) transferring the goal option from the task of stacking two pucks to stacking three pucks; the baseline is a vanilla deep Q-network and weight sharing means keeping the trained weights from the previous task; b) transferring all options from the task of stacking three pucks to the task of making two stacks of height two. All results are averaged over 20 runs with different random seeds.

Our agent augmented with the goal option reaches a similar cumulative reward to the baseline with weight sharing (Figure 2a). We 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, we 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, our agent is augmented with options for reaching all state blocks of the abstract MDP. The baselines are the same as in the previous experiment.

Figure 2b shows that our 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. We 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.

7 Conclusion and Future Work

We developed an algorithm for finding abstract MDPs in environments with continuous state spaces and demonstrated the algorithm works with medium-sized abstract MDPs. Furthermore, we devised a method for guiding exploration with an abstract MDP learned in a previous task based on the options framework. Our transfer method beats a deep Q-network baseline when the goal of the previous task is not on the path to the goal of the next task and it performs equally well otherwise.

In robotic manipulation tasks, most MDPs are deterministic with the exception of a small number of failure modes (e.g. the robot grasps an object by its edge and it subsequently falls out of its hand); in our future work, we aim to bound the loss in the performance of our algorithm when dealing with a certain degree of non-determinism in the MDP. Moreover, our algorithm creates a brand new partition every time it is run to avoid over-segmentation–if there are too many state-action blocks there is no way of merging them. We will address this limitation by introducing an operation that can merge two state-action blocks upon receiving more experience.

ACKNOWLEDGMENTS

REFERENCES

Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5): 679–684, 1957. ISSN 00959057, 19435274.

Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A systematic study of the class imbalance problem in convolutional neural networks. *Neural networks : the official journal of the International Neural Network Society*, 106:249–259, 2018.

Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1):163 – 223, 2003. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(02)00376-4. Planning with Uncertainty and Incomplete Information.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –, Feb 2015.

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

Balaraman Ravindran. *An Algebraic Approach to Abstraction in Reinforcement Learning*. PhD thesis, 2004. AAI3118325.

Balaraman Ravindran and Andrew G Barto. Approximate homomorphisms: A framework for non-exact minimization in markov decision processes. 2004.

Vishal Soni and Satinder Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pp. 494–499. AAAI Press, 2006. ISBN 978-1-57735-281-5.

Jonathan Sorg and Satinder Singh. Transfer via soft homomorphisms. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pp. 741–748, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-7-8.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181 – 211, 1999. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00052-1.

Jonathan J. Taylor, Doina Precup, and Prakash Panangaden. Bounding performance loss in approximate mdp homomorphisms. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, pp. 1649–1656, USA, 2008. Curran Associates Inc. ISBN 978-1-6056-0-949-2.

Alicia P. Wolfe. Defining object types and options using mdp homomorphisms. 2006.

Alicia Peregrin Wolfe and Andrew G. Barto. Decision tree methods for finding reusable mdp homomorphisms. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pp. 530–535. AAAI Press, 2006. ISBN 978-1-57735-281-5.

APPENDICES

A CONVOLUTIONAL NETWORK ARCHITECTURE AND SETTINGS

We ran a grid search over various convolutional network architectures. In particular, we searched for the architecture with the highest downsampling factor of the depth image that still achieves an acceptable accuracy. We chose the following settings for our experiment:

Branch 1

1. input: depth image
2. convolution 1, 3x3 filter size, stride 2, 32 filters
3. convolution 2, 3x3 filter size, stride 2, 64 filters
4. convolution 3, 3x3 filter size, stride 2, 128 filters
5. flatten the output into a vector

Branch 2

1. input: one-hot encoded state of the hand and one-hot encoded action
2. fully-connected 1, 32 neurons
3. fully-connected 2, 64 neurons

Head

1. concatenate output of Branch 1 and 2
2. fully-connected 3, 128 neurons

We set the learning rate to 0.001, the batch size to 32 and the weight decay for all layers to 0.0001 without an extensive grid search.

B DEEP Q-NETWORK SETTINGS

Our implementation of the vanilla deep Q-network is based on the OpenAI baselines[3] with a custom architecture described in Appendix A. We did not use prioritized experience replay or dueling networks. We ran a second grid search to find the best learning rate and target network update frequency with the following results: 0.0001 learning rate and update the target network every 100 steps. For the exploration, we linearly decayed the value of ϵ from 1.0 to 0.1 for 5000 steps in all experiments involving a deep Q-network. The batch size was set to 32.

[3] https://github.com/openai/baselines