HIERARCHICAL REPRESENTATION LEARNING FOR MARKOV DECISION PROCESSES

Lorenzo Steccanella *, **Anders Jonsson**
Dept. Information and Communication Technologies,
Universitat Pompeu Fabra,
Barcelona, Spain.

Simone Totaro
Mila – Quebec Artificial Intelligence Institute,
Montreal, Canada.

ABSTRACT

In this paper, we present a novel method for learning reward-agnostic hierarchical representations of Markov Decision Processes. Our method works by partitioning the state space into subsets, and defines subtasks for performing transitions between the partitions. At the high level, we use model-based planning to decide which subtask to pursue next from a given partition. We formulate the problem of partitioning the state space as an optimization problem that can be solved using gradient descent given a set of sampled trajectories, making our method suitable for high-dimensional problems with large state spaces. We empirically validate the method, by showing that it can successfully learn useful hierarchical representations in domains with high-dimensional states. Once learned, the hierarchical representation can be used to solve different tasks in the given domain, thus generalizing knowledge across tasks.

1 INTRODUCTION

In reinforcement learning, an agent attempts to learn behaviors through interaction with an unknown environment. By observing the outcome of actions, the agent has to learn from experience which action to select in each situation to maximize the expected cumulative reward. To learn, the agent has to balance exploration, i.e. discovering the effect of actions on the environment, and exploitation, i.e. repeating action choices that have proven successful in the past.

In hierarchical reinforcement learning (Barto & Mahadevan, 2003), the task is decomposed into subtasks, and the solutions to the subtasks are combined to form a solution to the overall task. If each subtask is easier to solve than the overall task, the decomposition can significantly speed up learning. The subtasks can also help explore the environment more efficiently, since one high-level decision typically brings the learning agent multiple steps in a promising direction, rather than exploring locally one step at a time.

In most applications of hierarchical reinforcement learning, the subtask decomposition is provided by a domain expert that exploits extensive domain knowledge to define appropriate subtasks. Though many automatic methods have been proposed, learning a useful subtask decomposition from experience is still mostly an open research question. In addition, several of the proposed methods are not appropriate for high-dimensional problems since they maintain statistics about individual states.

In this paper, we propose a novel method for learning a hierarchical representation for reinforcement learning. The idea is to partition the state space into subsets and define subtasks that perform transitions between the partitions. We formulate the problem of learning a hierarchical representation as an optimization problem that can be solved using gradient descent given a set of sampled trajectories. The resulting method can be applied to high-dimensional states (e.g. images) and combined with state-of-the-art function approximation techniques for reinforcement learning. In experiments, we show that our method can learn useful subtask decompositions in several domains with high-dimensional observations in the form of images. We also show that the learned hierarchical representation can be used to transfer knowledge to new, previously unseen tasks, thus generalizing knowledge across tasks.

1.1 RELATED WORK

Hierarchical reinforcement learning using hand-crafted subgoals to guide exploration, either as part of the value function representation (Nachum et al., 2018a; Schaul et al., 2015; Sutton et al., 2017), or as pseudo-rewards (Eysenbach

* Correspondence to: Lorenzo Steccanella <lorenzo.steccanella@upf.edu>

et al., 2019; Florensa et al., 2017), can solve hard exploration tasks with sparse rewards more efficiently than a flat learner, even for high-dimensional states.

Early work on learning hierarchical representations for reinforcement learning focused on analyzing the state transition graph (Menache et al., 2002; Şimşek et al., 2005), clustering nearby states (Mannor et al., 2004), discovering common substructure (Pickett & Barto, 2002) or identifying landmarks (McGovern & Barto, 2001; Şimşek & Barto, 2004; Solway et al., 2014a). However, most of these methods rely on enumerating states, which is not possible in high-dimensional domains. Lakshminarayanan et al. (2016) used a spectral clustering algorithm on the state space representation learned using an unsupervised model prediction network (Oh et al., 2015). Their method scales to high-dimensional states but strongly relies on the latent representation of the neural network and does not perform clustering directly on the original state space.

Skill learning (Konidaris & Barto, 2007; Da Silva et al., 2012) identifies initiation sets of options by searching backward from a given set of target states where options terminate. Similar to our method, skills can be reused in a range of similar tasks. The option-critic framework (Bacon et al., 2017) and similar algorithms such as MODAC (Veeriah et al., 2021) use gradient descent to learn the components of each option from trajectories. However, the resulting options are not easily interpretable, unlike our options that always transition between partitions. DDO (Fox et al., 2017) leverages an Expectation-Maximization algorithm to train a hierarchy of options end-to-end for imitation learning. Unlike our method, they do not explicitly consider initiation sets of options.

Corneil et al. (2018) learn a latent state model given a sequence of observations, akin to learning a mapping from states to abstract states, using a neural network architecture similar to a variational autoencoder. Shang et al. (2019) use variational inference to construct a partition similar to ours, but unlike our model-free option learning, the option policies are trained using dynamic programming, which requires knowledge of the environment dynamics. Bagaria et al. (2023) introduces the concept of proto-goals, a generalization of goals. Their approach assumes prior knowledge of the proto-goal space, which may not always be available. One advantage of their representation is that the proto-goal space only needs to contain useful goal components and can be extended to a combinatorially larger goal space with logical operations. In contrast to our work, they do not learn a partition of the state space.

Machado et al. (2017) use learned proto-value functions to identify subtask structure, in which a proto-value function becomes a local reward function for a given option. Eysenbach et al. (2019) build distance estimates between pairs of states, and use the distance estimate to condition reinforcement learning in order to reach specific goals, which is similar to defining temporally extended actions. Ghosh et al. (2018) learn an actionable representation that encodes distances between states in terms of the KL-divergence between the policies that transition to them. They assume access to a trained goal-conditioned policy capable of reaching any state. Nachum et al. (2018a;b) propose a two-level hierarchy where a high-level policy operates on a compact goal space and selects sub-goals for a low-level goal-conditioned policy. The compact goal space is either known or learned through interaction with the environment. Unlike these works, our method is designed to balance the size of the partitions, and create partitions that are strongly connected (cf. the properties listed in Appendix A).

Several authors have used state space partitions to handcraft hierarchical structures. Ecoffet et al. (2019) use a partition to learn to play Montezuma Revenge, using random search to find transitions between the partitions. Wen et al. (2020) take advantage of equivalent partitions with the same local dynamics to reuse option policies in multiple partitions. When the number of termination states of options is relatively small, the resulting algorithm has much better sample complexity properties than flat learning.

2 Background

In this section we define Markov decision processes and hierarchical reinforcement learning in the form of the options framework. For any finite set X, let $\Delta(X) = \{p \in \mathbb{R}^X : \sum_{x \in X} p(x) = 1, p(x) \geq 0 \ (\forall x)\}$ be the probability simplex on X, i.e. the set of all probability distributions on X.

2.1 Markov Decision Processes

A finite Markov decision process (MDP) (Puterman, 2014) is a tuple $\mathcal{M} = \langle S, A, P, r \rangle$, where S is a finite state space, A is a finite action space, $P : S \times A \to \Delta(S)$ is a transition kernel and $r : S \times A \to \mathbb{R}$ is a reward function. At time t, the learning agent observes a state $s_t \in S$, takes an action $a_t \in A$, obtains a reward r_t with expected value $\mathbb{E}[r_t] = r(s_t, a_t)$, and transitions to a new state $s_{t+1} \sim P(\cdot|s_t, a_t)$. We refer to (s_t, a_t, r_t, s_{t+1}) as a *transition*.

A stochastic policy $\pi : S \to \Delta(A)$ is a mapping from states to probability distributions over actions. The aim of reinforcement learning is to compute a policy π that maximizes some notion of expected future reward. In this work,

we consider the discounted reward criterion, for which the expected future reward of a policy π can be represented using a value function V^π, defined for each state $s \in S$ as

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(S_t, A_t) \middle| S_1 = s\right].$$

Here, random variables S_t and A_t model the state and action at time t, respectively, and the expectation is over the action $A_t \sim \pi(\cdot|S_t)$ and next state $S_{t+1} \sim P(\cdot|S_t, A_t)$. The discount factor $\gamma \in (0, 1]$ is used to control the relative importance of future rewards, and to ensure V^π is bounded.

As an alternative to the value function V^π, one can instead model expected future reward using an action-value function Q^π, defined for each state-action pair $(s, a) \in S \times A$ as

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} r(S_t, A_t) \middle| S_1 = s, A_1 = a\right].$$

The value function V^π and action-value function Q^π are related through the well-known Bellman equations:

$$V^\pi(s) = \sum_{a \in A} \pi(a|s) Q^\pi(s, a),$$
$$Q^\pi(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^\pi(s').$$

The aim of learning is to find an optimal policy π^* that maximizes the value in each state, i.e. $\pi^*(s) = \arg\max_\pi V^\pi$. The optimal value function V^* and action-value function Q^* satisfy the Bellman optimality equations:

$$V^*(s) = \max_{a \in A} Q^*(s, a),$$
$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s').$$

2.2 Model-Based Methods

If the reward function r and transition probabilities P are known (and the state and action spaces are not very large), we can use dynamic programming methods such as Value Iteration to compute an estimate $\widehat{Q}$ of the optimal action-value function, using the following update rule:

$$\widehat{Q}_{t+1}(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \max_{a'} \widehat{Q}_t(s', a').$$

Value iteration repeatedly computes $\widehat{Q}_{t+1}$ for all state-action pairs (s, a) until $\|\widehat{Q}_{t+1} - \widehat{Q}_t\|_\infty < \varepsilon$ for some desired accuracy ε, i.e. until the difference between subsequent iterations is small enough.

2.3 Function Approximation

Since the state space S is usually large, it is common to define a set of *features* Φ, and an associated mapping $\phi : S \to \Phi$ from states to features. Value-based methods such as Deep Q-learning (Mnih et al., 2013) maintain an estimate $\widehat{Q}_\theta : \Phi \times A \to \mathbb{R}$ of the optimal action-value function, defined on features instead of states and parameterized on a vector θ. Given a transition $\langle s_t, a_t, r_t, s_{t+1} \rangle$, DQN updates the parameter vector according to the gradient method:

$$\theta \leftarrow \theta + \alpha \left(\mathrm{T}_{\widehat{Q}} - \widehat{Q}_\theta(s, a)\right) \nabla_\theta \widehat{Q}_\theta(s, a)$$

where $\mathrm{T}_{\widehat{Q}}$ is a target value based on optimal bellman equation calculated as:

$$\mathrm{T}_{\widehat{Q}} = r\left(s, a\right) + \gamma \max_{a'} \widehat{Q}_\theta\left(s', a'\right)$$

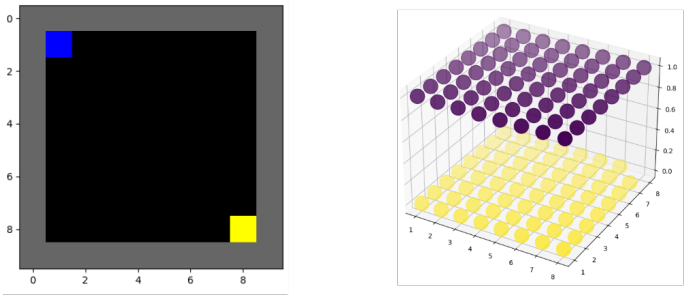

Figure 1: Results on Key-Door0 gridworld environment. On the left the Key-Door0 gridworld environment, on the right the corresponding learned deterministic compression functions, where different colors represent different abstract states $z \in Z$.

2.4 Options

Given an MDP $\mathcal{M} = \langle S, A, P, r \rangle$, an option is a temporally extended action $o = \langle I^o, \pi^o, \beta^o \rangle$, where $I^o \subseteq S$ is an initiation set, $\pi^o : S \to \Delta(A)$ is a policy and $\beta^o : S \to [0, 1]$ is a termination function (Sutton et al., 1999). An option can be applied in any state $s \in I^o$, selects actions using policy π^o, and terminates in a state $s' \in S$ with probability $\beta^o(s')$. A primitive action $a \in A$ is a special case of an option $\langle I^a, \pi^a, \beta^a \rangle$ with initiation set $I^a = S$, $\pi^a(a|s) = 1$ and $\beta^a(s) = 1$ $(\forall s)$, i.e. the option can be applied in any state, terminates with probability 1 in any state, and the associated policy always selects action a with probability 1.

Given a set of options O, we can form a semi-Markov decision process (SMDP) $\mathcal{S} = \langle S, O, P', R' \rangle$, where P' and R' model the transition probabilities and expected reward of options. Such an SMDP enables a learning agent to act and reason on multiple timescales. At the top level, the learning agent observes a state s_t, selects an option o_t, executes option o_t until termination, and observes the next state s_{t+k}, where k is the time it takes option o_t to terminate. The reward $R_t = \sum_{u=t}^{t+k-1} \gamma^{u-t} r_u$ is the discounted sum of rewards obtained during the execution of option o. Hence (s_t, o_t, R_t, s_{t+k}) is a high-level transition, and with minor modifications, reinforcement learning algorithms can be adapted to estimate an optimal SMDP policy π^*, even when the SMDP dynamics P' and R' are unknown.

To learn the option policy π^o, it is common to define an option-specific reward function r^o, which defines an option-specific Markov decision process $\mathcal{M}^o = \langle S, A, P, r^o \rangle$. The policy π^o is then implicitly defined as the optimal solution to $\mathcal{M}^o$. Even though the original definition of SMDPs considers options with fixed policies, in practice one can learn the option policies and the SMDP policy in parallel.

3 Contribution

In this section we present our main contribution, a method for learning a hierarchical representation of a given MDP.

3.1 Compression Function

The first step is to learn a compression function from MDP states to abstract states. We first define a set Z of abstract states that will represent the partitions of the state space. Without loss of generality, the elements of Z are simply integers, i.e. $Z = \{0, \ldots, |Z| - 1\}$, where $|Z|$ is an input parameter of the method. Our goal is to learn a parameterized compression function $f_\psi : S \to \Delta(Z)$ that maps MDP states to probability distributions over abstract states. Ideally, f_ψ should be *deterministic*, but the learning framework we consider favors probabilistic compression functions.

Intuitively, for abstract states to represent partitions of the state space, on a given trajectory the abstract state should remain the same most of the time, and only change occasionally. We formalize this intuition as a loss term, which will later be part of the objective that the learner attempts to minimize. Let $\tau = \langle s_t, a_t, r_t, s_{t+1} \rangle$ be a transition, and let $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ be a set of transitions. The loss associated with $\mathcal{T}$ is given by

$$\mathcal{L}_Z(\mathcal{T}) = -\sum_{\tau \in \mathcal{T}} \sum_{z \in Z} f_\psi(z|s_t) \log f_\psi(z|s_{t+1}).$$

Here, $-\sum_{z \in Z} f_\psi(z|s_t) \log f_\psi(z|s_{t+1})$ is the cross-entropy loss for consecutive states s_t and s_{t+1} in τ, measuring the distance between the distributions $f_\psi(\cdot|s_t)$ and $f_\psi(\cdot|s_{t+1})$.

On its own, the above loss term will not yield a meaningful compression function, since it can be minimized by mapping all states to the same abstract state. To ensure that all abstract states appear in the compression, we define a second loss term equivalent to the negative entropy of the compression function across the same set of transitions $\mathcal{T}$. Given an abstract state $z \in Z$, let $F(z|\mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} f_\psi(z|s_t)$ be the average probability of being in z across the first state s_t of each transition $\tau \in \mathcal{T}$. We define a loss term

$$\mathcal{L}_H(\mathcal{T}) = -H(F(\cdot|\mathcal{T})) = \sum_{z \in Z} F(z|\mathcal{T}) \log F(z|\mathcal{T}),$$

where $H(F(\cdot|\mathcal{T}))$ is the entropy of the function $F(\cdot|\mathcal{T})$. This loss is minimized when the probabilities of abstract states are uniform, i.e. each abstract state is equally likely.

Finally, as already stated, we would like the compression function f_ψ to be as deterministic as possible. For this reason, we define a third loss term equivalent to the entropy of the compression function for individual states. We use the same set of transitions $\mathcal{T}$, and define this loss term as

$$\mathcal{L}_D(\mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} H(f_\psi(\cdot|s_t)) = -\frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \sum_{z \in Z} f_\psi(z|s_t) \log f_\psi(z|s_t).$$

This loss term is minimized when the compression function $f_\psi(\cdot|s_t)$ is deterministic, i.e. assigns probability 1 to a single abstract state, for the first state s_t of each transition $\tau \in \mathcal{T}$.

The overall loss function $\mathcal{L}(\mathcal{T})$ is a combination of the three individual loss terms, i.e.

$$\mathcal{L}(\mathcal{T}) = \mathcal{L}_Z(\mathcal{T}) + w_H \mathcal{L}_H(\mathcal{T}) + w_D \mathcal{L}_D(\mathcal{T}), \tag{1}$$

where w_H and w_D are weights that we can tune to determine the relative importance of each loss term. Note that for $w_D = -1$, $\mathcal{L}_Z(\mathcal{T}) + w_D \mathcal{L}_D(\mathcal{T})$ is the average Kullback-Leibler divergence between $f_\psi(\cdot|s_t)$ and $f_\psi(\cdot|s_{t+1})$; however, our intention is to use positive values of w_D.

To train our compression function we use experience replay (Mnih et al., 2013) to randomize the transitions in $\mathcal{T}$. We first sample a set of trajectories using some exploration policy, which constitutes our memory. However, learning directly from consecutive transitions along the same trajectory is inefficient, due to the strong correlations between the samples. Instead, we form the set of transitions $\mathcal{T}$ by randomly sampling individual transitions from the memory. Randomizing the sampled transitions this way breaks the correlations and therefore reduces the variance of the updates.

We discuss the properties of the learned representation in Appendix A.

3.2 Hierarchical Representation

Once we have learned a compression function f_ψ for a given MDP $\mathcal{M} = \langle S, A, P, r \rangle$, we use it to define a set of options O and an SMDP $\mathcal{S}$. First, we introduce a *deterministic* compression function $g : S \to Z$, defined in each state s as $g(s) = \arg\max_z f_\psi(z|s)$. Given an abstract state $z \in Z$, let S_z be the subset of states that map to z, i.e. $S_z = \{s \in S : g(s) = z\}$.

Our algorithm then uses the compression function in an online manner, by exploring the environment and finding *abstract transitions*, i.e. consecutive states s_t and s_{t+1} such that $g(s_t) \neq g(s_{t+1})$. Let $\mathcal{Y} \subseteq Z \times Z$ be the subset of pairs of distinct abstract states (z, z') that appear as abstract transitions while exploring, i.e. there exist two consecutive states s_t and s_{t+1} such that $g(s_t) = z$ and $g(s_{t+1}) = z'$. For each pair $(z, z') \in \mathcal{Y}$, we introduce an option $o_{z,z'} = \langle I^o, \pi^o, \beta^o \rangle$ whose purpose is to perform an abstract transition from z to z'. Option $o_{z,z'}$ is applicable in abstract state z, i.e. $I^o = S_z$, and terminates as soon as we reach an abstract state different from z, i.e. $\beta^o(s) = 0$ if $s \in S_z$ and $\beta^o(s) = 1$ otherwise.

To learn the policy π^o of option $o_{z,z'}$, we define an option-specific Markov decision process $\mathcal{M}^o = \langle S_z, A, P, r^o \rangle$. Note that $\mathcal{M}^o$ needs only be defined for states in S_z, since option $o_{z,z'}$ always terminates outside this set. The local reward function r^o is defined for each state-action pair as $r^o(s, a) = r(s, a)$, i.e. equal to the environment reward. We also introduce a bonus $+1$ for terminating in a state s such that $g(s) = z'$. As a consequence, the policy π^o has an incentive to leave abstract state z, and prefers to transition to abstract state z' whenever possible.

Let $O = \{o_{z,z'} : (z, z') \in \mathcal{Y}\}$ be the set of options for performing abstract transitions, and let $O_z = \{o \in O : I^o = S_z\}$ be the subset of options applicable in abstract state z. We define an SMDP $\mathcal{S} = \langle S, O, P', R' \rangle$, i.e. the high-level choices of the learning agent are to select abstract transitions to perform. Once the individual option policies have been trained, exploration is typically more efficient since the single decision of which option to execute results in a state that is many steps away from the initial state. In addition, one can approximate the SMDP policy as $\pi : Z \to \Delta(O)$, i.e. the

choice of which option to execute only depends on the current abstract state. This has the potential to significantly speed up learning if $|Z| \ll |S|$.

The system is trained using a Manager-Worker architecture (Dayan & Hinton, 2000). The Manager performs tabular Value Iteration over the SMDP. The motivation for using tabular learning is that the number of abstract states $|Z|$ is typically small, even if states are high-dimensional. On the other hand, the Worker uses off-policy value based methods to learn the policies of the options $o_{z,z'}$.

3.3 Controllability

According to the definition of the option reward function r^o, option $o_{z,z'}$ is equally rewarded for reaching any boundary state between abstract states z and z'. However, all boundary states may not be equally valuable, i.e. from some boundary states, the options in $O_{z'}$ may have a higher chance of terminating successfully. To encourage option $o_{z,z'}$ to reach valuable boundary states and thus make the algorithm more robust to the choice of compression function g, we add a reward bonus when the option successfully terminates in a state s' belonging to abstract state z'.

One possibility is that the reward bonus depends on the value of state s' of options in the set $O_{z'}$. However, this introduces a strong coupling between options in the set O: the value function $V_{z,z'}$ of option $o_{z,z'}$ will depend on the value functions of options in $O_{z'}$, which in turn depend on the value functions of options in neighboring abstract states of z', etc. We want to avoid such a strong coupling since learning the option value functions may become as hard as learning a value function for the original state space S.

Instead, we introduce a reward bonus which is a proxy for controllability, by counting the number of successful applications of subsequent options after $o_{z,z'}$ terminates. Let M be the number of options that are selected after $o_{z,z'}$, and let $N \leq M$ be the number of such options that terminate successfully. We define a controllability coefficient ρ as

$$\rho(z) = \frac{N}{M}. \tag{2}$$

We then define a modified reward function $\bar{r}^o$ which equals r^o except when $o_{z,z'}$ terminates successfully, i.e. $\bar{r}^o(s,a,s') = r^o(s,a,s') + \rho(z)$ if $s' \in z'$. In experiments we use a fixed horizon $M = 4$ after which we consider successful option transitions as irrelevant.

3.4 Transfer

The hierarchical representation in the form of the SMDP $\mathcal{S}$ defined above can be used to transfer knowledge between tasks. Concretely, we assume that the given MDP $\mathcal{M}$ can be extended to form a task by adding states and actions. Imagine that $\mathcal{M}$ models a navigation problem in a given environment. A task can be defined by adding objects in the environment that the learning agent can manipulate, while navigation is still part of the task.

Formally, given an MDP $\mathcal{M} = \langle S, A, P, r \rangle$, a task $\mathbb{T}$ is an MDP $\mathcal{M}_\mathbb{T} = \langle S \times S_\mathbb{T}, A \cup A_\mathbb{T}, P \cup P_\mathbb{T}, r \cup r_\mathbb{T} \rangle$. The states in $S_\mathbb{T}$ represent information about task-specific objects, and the actions in $A_\mathbb{T}$ are used to manipulate these objects. The transition kernel $P_\mathbb{T} : (S \times S_\mathbb{T}) \times A_\mathbb{T} \to \Delta(S_\mathbb{T})$ governs the effects of the actions in $A_\mathbb{T}$, which may depend on the states of the original MDP (e.g. the location of the agent). Finally, the reward function $r_\mathbb{T} : (S \times S_\mathbb{T}) \times A_\mathbb{T} \to \mathbb{R}$ models the reward associated with actions in $A_\mathbb{T}$.

To solve a task, we can replace the MDP $\mathcal{M}$ with the learned SMDP $\mathcal{S} = \langle S, O, P', R' \rangle$, forming a task SMDP $\mathcal{S}_\mathbb{T} = \langle S \times S_\mathbb{T}, O \cup A_\mathbb{T}, P' \cup P_\mathbb{T}, R' \cup r_\mathbb{T} \rangle$. Here, the options in O are used to navigate in the original state space S, while the actions in $A_\mathbb{T}$ are used to manipulate the task-specific objects. If the policies of the options in O have been previously trained, the task SMDP $\mathcal{S}_\mathbb{T}$ can significantly accelerate learning compared to the task MDP $\mathcal{M}_\mathbb{T}$. To ensure that the learning agent can navigate to individual objects inside a partition of Z, we consider states in $S_\mathbb{T}$ to be different abstract states; hence our algorithm will automatically add options for manipulating objects.

4 Experimental Results

The experiments are designed to answer the following questions [1]:

- Is the learned compression function suitable for learning a hierarchy?
- Does the learned hierarchy transfer across different tasks in the same environment?

[1] The code is publicly available at: https://github.com/lorenzosteccanella/HRL-MDP

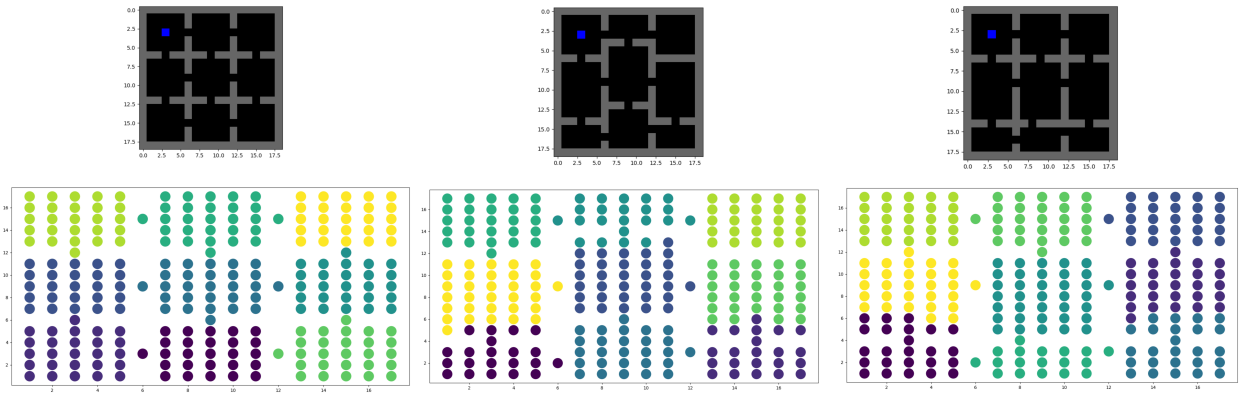

Figure 2: Results on geometric variations of the NineRooms0 gridworld environments. The first row represents the environments, and the second row illustrates examples of the corresponding learned deterministic compression functions, where different colors represent different abstract states $z \in Z$.

- How does our HRL algorithm compare against state-of-the-art flat algorithms, such as Self Imitation Learning (Oh et al., 2018) and Double-DQN with Prioritized Experience Replay (Schaul et al., 2016)?

- How does our HRL algorithm compare against state-of-the-art HRL algorithms such as Option Critic (Bacon et al., 2017)?

4.1 Learning a Compression Function

We designed two different empty navigation environments without tasks, KeyDoor0 (c.f. Figure 1), with grid size 10×10, where an agent (blue square) always starts in position (1, 1), has to collect a key (yellow square). And NineRooms0 (c.f. Figure 2), a nine rooms grid environment with grid size 19×19 where at each episode the agent is placed at a random initial position, which promotes exploration. For all the environments the states are (x, y)-positions which are mapped to images, and the discrete action space is $A = \{up, down, left, right\}$.

The first step of our procedure consists in a pre-training phase where we form a replay memory of trajectories. We use a random exploration policy to repeatedly generate trajectories from the random initial states, using a fixed episode length of 100. During this phase, we can vary the number of trajectories generated to test the robustness of the approach.

We then use the replay memory and a number of abstract states $|Z| = 2$ for KeyDoor0 and $|Z| = 9$ for NineRoom0 to train the compression function f_ϕ using the AdamW optimizer (Loshchilov & Hutter, 2017) by minimizing the loss in equation 1 over 4000 iterations, randomly sampling a set of 32 transitions $\mathcal{T}$ from the replay memory in each iteration. The learned compression functions for 1000 trajectories are shown in Figures 1 and 2, respectively.

To asses the robustness of our procedure, in Figure 2 we evaluate how the compression function changes by introducing different geometries of the NineRooms0 environment. As we can see, if we make the room sizes imbalanced, the resulting compression function does not exactly match the shape of the rooms, due to the second term $\mathcal{L}_H$ of the loss in equation 1, which promotes all abstract states z to be equally likely. However, the resulting compression function still partitions the states and translates into a correct SMDP.

In Figure 4, we evaluate how the size of the replay memory affects the accuracy of the compression function in terms of the absolute error deviation with respect to a correct representation. For this experiment we use the left-most room in Figure 2 with balanced room sizes, and vary the number of trajectories in the replay memory. When the replay memory contains at least 200 trajectories, the procedure converges to an absolute error very close to 0, while less than 200 trajectories results in an increasing absolute error.

We present additional experiments with learning a compression function in the MountainCar environment in Appendix D. We also list all the hyperparameters of the algorithm in Appendix E.

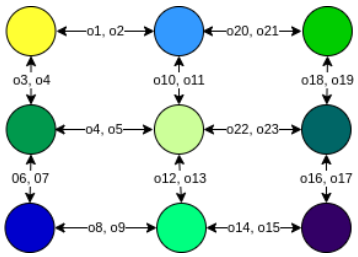

Figure 3: The discovered invariant SMDP on the NineRooms0 environment.

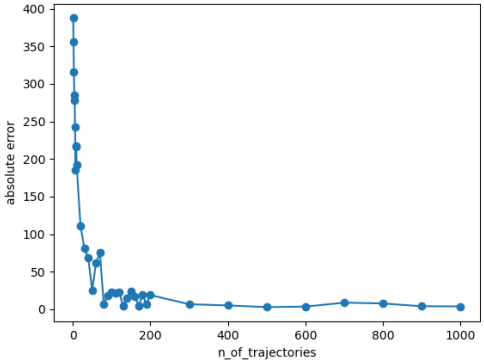

Figure 4: Absolute error of the compression function, evaluated on increasing replay memory size.

4.2 Hierarchical Reinforcement Learning

Following the pre-training phase, we can use the learned compression function to solve any task in the same environment. In what follows we use the compression function learned on the left-most room in Figure 2 with a replay memory of 1000 trajectories. We distinguish between a *manager* in charge of solving the task SMDP $\mathcal{S}_{\mathbb{T}}$ and *workers* in charge of solving the option MDPs $\mathcal{M}^o$.

4.2.1 Manager

Our algorithm iteratively grows an estimate of the SMDP S. Initially, the agent only observes a single state $s \in S$ and associated abstract state $z = g(s)$. Hence the state space Z contains a single abstract state z, whose associated option set O_z is initially empty. In this case, the only alternative available to the agent is to $explore$. For each abstract state z, we add an exploration option $o_z^{exploration} = \langle z, \pi_z^{exploration}, \beta_z \rangle$ to the option set O. This option has the same initiation set and termination condition as the options in O_z, but the policy $\pi_z^{exploration}$ is an exploration policy that selects actions uniformly at random, terminating when it leaves abstract state z or exhausts a given budget.

Once the agent discovers a neighboring abstract state z' of z, it adds z' to the set Z and the associated option $o_{z,z'}$ to the option set O. The agent also maintains and updates a directed graph whose nodes are abstract states and whose edges represent the neighbor relation. Hence next time the agent visits abstract state z, one of its available actions is to select option $o_{z,z'}$. When option $o_{z,z'}$ is selected, it chooses actions using its policy $\pi_{z,z'}$ and updates $\pi_{z,z'}$ based on the rewards of the option MDP $\mathcal{M}^o$. Figure 3 shows an example representation discovered by the algorithm on the NineRooms0 environment.

Algorithm 1 in the appendix shows the pseudo-code of the algorithm. As explained, Z is initialized with the abstract state z of the initial state s, and O is initialized with the exploration option $o_z^{exploration}$. In each iteration the algorithm selects an option o which is applicable in the current abstract state z. If we transition to a new abstract state z', it is added to Z and the exploration option $o_{z'}^{exploration}$ and transition option $o_{z,z'}$ are appended to O. The process then repeats from the next state s'.

The subroutine GetOption that selects an option o in the current abstract state z can be implemented in different ways; we use an ϵ-greeedy policy. Since the set of abstract states Z is small, the manager performs tabular Value Iteration over the task SMDP $\mathcal{S}_{\mathbb{T}}$. In order to recognize new goal states, while exploring we define any terminal state in the environment as a new abstract state z; hence the manager will introduce options for reaching this terminal state.

4.2.2 Planning with a learned SMDP

We have seen how the state space of the task SMDP $\mathcal{S}_{\mathbb{T}}$ is discovered online given a compression function $g(s)$. In order to apply a model-based method on this learned compression function, we still need to be able to estimate the transition kernel $P_{\mathcal{S}_{\mathbb{T}}}$ and reward function $r_{\mathcal{S}_{\mathbb{T}}}$ of $\mathcal{S}_{\mathbb{T}}$.

Estimating the transition probability associated with an option $o_{z,z'}$ of our task SMDP is not easy, since the policy π^o is trained online while exploring the environment, making the transition probability non-stationary. In order to alleviate

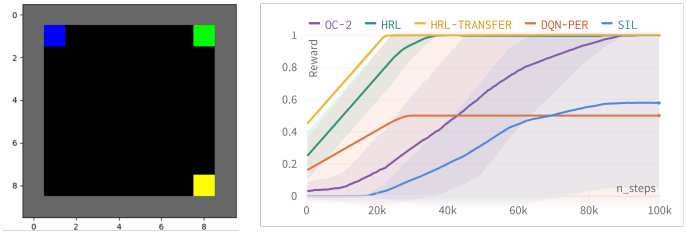

Figure 5: Results in the KeyDoor environment.

the cost of estimating the transition probability, we assume that $o_{z,z'}$ will become deterministic once the training phase terminates, i.e. $\widehat{P}_{\mathcal{S}_\mathbb{T}}(z'|z, o_{z,z'}) = 1$. Though this is an approximation, the aim of option $o_{z,z'}$ is precisely to reach abstract state z', and constructing the SMDP is intended to simplify the high-level decision making.

On the other hand, for each state-option pair (z, o) of the task SMDP $\mathcal{S}_\mathbb{T}$, we estimated the task SMDP reward $r_{\mathcal{S}_\mathbb{T}}$ as an average of the reward encountered in the environment $\widehat{r}_{\mathcal{S}_\mathbb{T}}(s, o) = \frac{\sum_{i=1}^{N(z,o)} R_i(z,o)}{N(z,o)}$, where $N(z, o)$ counts the number of times the state-option pair (z, o) has been observed, and R_i is the cumulative reward obtained while applying option o for the i-th time.

Since the model changes over time, the subroutine UPDATEPOLICY updates the Q values of the Manager at regular intervals by applying value iteration on the learned SMDP $\widehat{\mathcal{S}}_\mathbb{T}$.

4.2.3 WORKERS

The workers are in charge of learning the policies of each option $o_{z,z'}$ in O, allowing the manager to transition between two abstract states z, z'. We use Double DQN (van Hasselt et al., 2015), a version of DQN that addresses the overestimation of Q-values, combined with Prioritized Experience Replay (PER) (Schaul et al., 2016) that improves the way experience is sampled from the Experience Replay. The rewards that the worker observes are defined in the Hierarchical Representation Section and implemented in the routine TRAINOPTION from Algorithm 1 in the appendix.

Since Double DQN is able to evaluate Q-values off-policy, one can relabel failed transitions to speed up learning of the correct option behavior, similar to Hindsight Experience Replay (Andrychowicz et al., 2017). The architecture is made of a neural network Q_θ parametrized on θ, and a frozen target network $Q_{\bar{\theta}}$ used to alleviate the non-stationarity of the targets $\mathrm{T_Q} = r(s, a) + \gamma \max_{a'} Q_\theta(s', a')$.

The parameters of the neural network are updated as:

$$\theta \leftarrow \theta + \alpha \left(\mathrm{T_D} - Q_\theta(s, a)\right) \nabla_\theta Q_\theta(s, a),$$

where $\mathrm{T_D}$ is the target value computed as:

$$\mathrm{T_D} = r\left(s, a\right) + \gamma Q_{\bar{\theta}}\left(s_{t+1}, \underset{a}{\mathrm{argmax}} Q_\theta\left(s_{t+1}, a\right)\right).$$

The target network is then updated with Polyak updates (Heess et al., 2015):

$$\bar{\theta} = \tau\, \theta + (1 - \tau)\, \bar{\theta}.$$

4.3 EXPERIMENTS

In our experiments, we evaluate the performance of our agent in two environments, on a KeyDoor environment in Figure 5 where an agent has to collect a key (yellow square) and open a door (green square) and a NineRooms environment in Figure 6 where an agent has to reach the goal (green square). In both environments the initial position is fixed to $(1, 1)$. In the NineRooms environment, we defined three variants where the goal is positioned at an increasing distance from the initial state to make exploration harder, c.f. NineRooms1, NineRooms2 and NineRooms3 in Figure 6. Results are averaged over 5 seeds and each experiment is run for 100,000 iterations. Even though the compression function is given, the goal location is unknown, so the agent has to explore the environment in order to find the goal location for the first time.

We set the maximum number of steps in the environment to 40 for the KeyDoor environment and to 200 for NineRooms environment, making exploration hard, especially in NineRooms3 where the goal is at the maximum distance from the

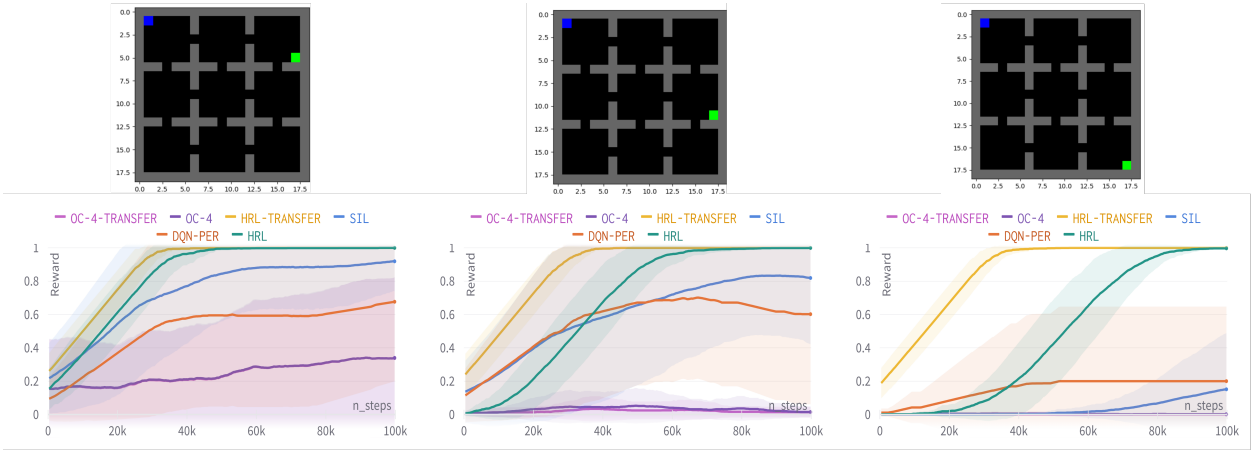

Figure 6: Results on the variations of nine room gridworld environments where the goal (green square) is placed at an increasing distance from the agent (blue square). From left to right: NineRoom1, NineRoom2, NineRoom3.

initial state. Results in Figures 5 and 6 show the total reward with a running average smoothing of 100 episodes and shaded standard deviation. In the KeyDoor environment, the agent receives a reward of +1 only once it opens the door (green square) with the key (yellow square) while in NineRooms the agent receives a reward of +1 when it reaches the goal position (green square) and a reward of 0 elsewhere.

We compare our algorithm against state-of-the-art flat reinforcement learning agents designed to perform well in sparse reward settings, namely Self Imitation Learning (SIL) (Oh et al., 2018) and Double DQN with Prioritized Experience Replay (DQN-PER) (Schaul et al., 2016), implemented on top of the Reinforcement Learning framework Machin (Li, 2020). Moreover, we include a comparison to state-of-the-art Hierarchical Reinforcement Learning (HRL) algorithm, namely Option Critic (OC) (Bacon et al., 2017), for which we tune the number of options selecting the best performing alternative. We refer to OC-2 as the algorithm with 2 options and OC-4 as the algorithm with 4 options. We also include a comparison to a transfer learning variant of Option Critic, OC-TRANSFER-4, where the agent is trained in sequence on environments NineRooms1, NineRooms2 and NineRooms3.

We refer to "HRL" as our algorithm in which the task SMDP $\mathcal{S}_{\mathbb{T}}$ is learned online while exploring, but the compression function g is given. "HRL-TRANSFER" refers to our algorithm where the agent is first pretrained in order to learn the options in KeyDoor0 and NineRooms0 in Figure 1 and Figure 2 without any task and then exposed respectively to KeyDoor and NineRooms1, NineRooms2, NineRooms3 in sequence. In this case, the algorithm benefits from the transfer of the SMDP S while the manager policy (i.e. Q-values) are reset to 0 after training in each environment.

We can observe that the HRL algorithm learns faster than SIL, DQN-PER and OC in all the environments. SIL and DQN-PER both rely only on random exploration, but once they find a positive reward, they can exploit it. In contrast, the exploration of HRL and HRL-TRANSFER are aided by the hierarchical structure. Both SIL, DQN-PER and OC present high variance, and for some seeds they are not even able to solve the task, given the budget of 100,000 iterations. We can also observe that HRL-TRANSFER does improve over HRL, and we would argue that this improvement could be larger if we choose harder tasks were the option policies for transitioning between abstract states become harder to learn.

5 Conclusion

We present a novel method for learning a hierarchical representation from sampled transitions in high-dimensional domains. The idea is to generate abstract states that partition the original state space, and introduce options for performing transitions between abstract states. Experiments show that the learned representation can successfully be used to solve multiple tasks in the same environment, significantly speeding up learning compared to a flat learner.

An important direction for future work is to sample trajectories using a more informed exploration policy, since learning the compression function depends on having a variety of trajectories in different states. Another possible extension is to interleave representation learning with policy improvement, which may successively improve the quality of the sampled trajectories. Yet another possibility is to correct the compression function in states from which some abstract transitions are not possible.

ACKNOWLEDGMENTS

Lorenzo Steccanella is partially funded by ACCIONA through the Reinforcement Learning for Water Treatment (RELEWAT) project.

Anders Jonsson is partially funded by TAILOR, AGAUR SGR and Spanish grant PID2019-108141GB-I00.

REFERENCES

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pp. 5048–5058, 2017.

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

Akhil Bagaria, Ray Jiang, Ramana Kumar, and Tom Schaul. Scaling goal-based exploration via pruning proto-goals. *arXiv preprint arXiv:2302.04693*, 2023.

Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1–2):41–77, January 2003. ISSN 0924-6703.

Dane S. Corneil, Wulfram Gerstner, and Johanni Brea. Efficient modelbased deep reinforcement learning with variational state tabulation. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 1057–1066, 2018. URL http://proceedings.mlr.press/v80/corneil18a.html.

Ö. Şimşek and A. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 21:751–758, 2004.

Ö. Şimşek, A. Wolfe, and A. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. *Proceedings of the International Conference on Machine Learning*, 22, 2005.

Bruno Castro Da Silva, George Konidaris, and Andrew G. Barto. Learning parameterized skills. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, pp. 1443–1450, Madison, WI, USA, 2012. Omnipress. ISBN 9781450312851.

Peter Dayan and Geoffrey Hinton. Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, 5, 09 2000. doi: 10.1002/0471214426.pas0303.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 15220–15231, 2019.

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

Roy Fox, Sanjay Krishnan, Ion Stoica, and Ken Goldberg. Multi-level discovery of deep options. *arXiv preprint arXiv:1703.08294*, 2017.

Dibya Ghosh, Abhishek Gupta, and Sergey Levine. Learning actionable representations with goal-conditioned policies. *arXiv preprint arXiv:1811.07819*, 2018.

Nicolas Heess, Greg Wayne, David Silver, Timothy Lillicrap, Yuval Tassa, and Tom Erez. Learning continuous control policies by stochastic value gradients, 2015.

George Konidaris and Andrew Barto. Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pp. 895–900, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

Aravind S Lakshminarayanan, Ramnandan Krishnamurthy, Peeyush Kumar, and Balaraman Ravindran. Option discovery in hierarchical reinforcement learning using spatio-temporal clustering. *arXiv preprint arXiv:1605.05359*, 2016.

Muhan Li. Machin. https://github.com/iffiX/machin, 2020.

Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving markov decision problems, 2013. URL https://arxiv.org/abs/1302.4971.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. A laplacian framework for option discovery in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2295–2304. JMLR.org, 2017.

S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. *Proceedings of the International Conference on Machine Learning*, 21:560–567, 2004.

A. McGovern and A. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. *Proceedings of the International Conference on Machine Learning*, 18:361–368, 2001.

I. Menache, S. Mannor, and N. Shimkin. Q-Cut – Dynamic Discovery of Sub-Goals in Reinforcement Learning. *Proceedings of the European Conference on Machine Learning*, 13:295–306, 2002.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Near-optimal representation learning for hierarchical reinforcement learning. *arXiv preprint arXiv:1810.01257*, 2018a.

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018b.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. *arXiv preprint arXiv:1507.08750*, 2015.

Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. In *International Conference on Machine Learning*, pp. 3878–3887. PMLR, 2018.

M. Pickett and A. Barto. Policyblocks: An Algorithm for Creating Useful Macro-Actions in Reinforcement Learning. *Proceedings of the International Conference on Machine Learning*, 19:506–513, 2002.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pp. 1312–1320, 2015.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

W Shang, A Trott, S Sheng, C Xiong, and R Socher. Learning World Graphs to Accelerate Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1907.00664*, 2019.

Özgür Şimşek and Andrew Barto. Skill characterization based on betweenness. *Advances in neural information processing systems*, 21, 2008.

Alec Solway, Carlos Diuk, Natalia Cordova, Debbie Yee, Andrew G. Barto, Yael Niv, and Matthew M. Botvinick. Optimal behavior hierarchy. *PLOS Comp. Bio.*, 10(8), 2014a.

Alec Solway, Carlos Diuk, Natalia Córdova, Debbie Yee, Andrew G Barto, Yael Niv, and Matthew M Botvinick. Optimal behavioral hierarchy. *PLoS computational biology*, 10(8):e1003779, 2014b.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pp. 212–223. Springer, 2002.

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

Richard S Sutton, Joseph Modayil, Michael Delp Thomas Degris, Patrick M Pilarski, and Adam White. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. 2017.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

Vivek Veeriah, Tom Zahavy, Matteo Hessel, Zhongwen Xu, Junhyuk Oh, Iurii Kemaev, Hado van Hasselt, David Silver, and Satinder Singh. Discovery of Options via Meta-Learned Subgoals. *CoRR*, abs/2102.06741, 2021.

Zheng Wen, Doina Precup, Morteza Ibrahimi, Andre Barreto, Benjamin Van Roy, and Satinder Singh. On efficiency in hierarchical reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 6708–6718. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/4a5cfa9281924139db466a8a19291aff-Paper.pdf.

A Representation Properties

In this section, we discuss the property of the representation presented in this paper. We adapt the definition of a state partition from Wen et al. (2020).

Definition 1 *Given an MDP $\mathcal{M} = \langle S, A, P, r \rangle$, consider a partition of the states S into L disjoint subsets $Z = \{S_i\}_{i=1}^{L}$, i.e. $S = S_1 \cup \cdots \cup S_L$ and $S_i \cap S_j = \emptyset$ for each pair $(S_i, S_j) \in Z^2$.*

We define an induced subMDP $\mathcal{M}_i = \langle S_i \cup E_i, A, P_i, r_i, E_i \rangle$ as follows:

- *S_i is the internal state set, and the action space is still A.*
- *The exit state set E_i is defined as $E_i = \{e \in S \backslash S_i : \exists (s, a) \in S_i \times A \text{ s.t. } P(e \mid s, a) > 0\}$. Is important to notice that the exit state set E_i will belong to a different partition in $S_j \in \mathcal{Z}$ with $j \neq i$ that is reachable in one step from some state in S_i.*
- *The state space of $\mathcal{M}_i$ is $S_i \cup E_i$.*
- *$P_i : S_i \times A \to \Delta(S_i \cup E_i)$ and $r_i : S_i \times A \times (S_i \cup E_i) \to \mathbb{R}$ are respectively the restriction of P and r to domain $S_i \times A$.*
- *The subMDP $\mathcal{M}_i$ terminates once it reaches a state in E_i (i.e., an exit state).*

We start to introduce some properties that a state partition should have to define a good representation for Hierarchical Reinforcement Learning.

- **Induced subMDP must be easy to solve**:
  The maximum size of an induced subMDP M is defined as:

  Definition 2 $M = \max_i |S_i \cup E_i|$.

  If M is small, all subMDPs have small size $|S_i \cup E_i| \leq M$, so they would be relatively easy to solve. This definition characterizes the hardness in terms of the state space size of a subMDP. Complexity results with tabular representation have shown that finite MDPs can be solved in polynomial time in the size of the state space and action space (Littman et al., 2013) when the transition matrix P is known, proving that the smaller the state space size is, the easier it is to solve the MDP.

  Note that this is the easiest and most general definition of hardness since it does not take into account the reward r_i nor transition probability P_i inside the subMDPs.
- **Bottleneck states**:
  The set of all exit states for a given partition is defined as:

  Definition 3 $\mathcal{E} = \cup_i^L E_i$.

  If $|\mathcal{E}|$ is small, intuitively we have a few states that connect the sub-problems. We can think of these as "bottleneck" states in $\mathcal{M}$, which have been shown before to enable computationally efficient planning (see e.g. Sutton et al. 1999; McGovern & Barto 2001; Stolle & Precup 2002; Şimşek & Barto 2008; Solway et al. 2014b).

  We highlight that a trade-off exists between the maximal size of an induced subMDP M and the size of the set of all exit states $|\mathcal{E}|$. It is desirable for M to be small to simplify the solving process for every subMDP. However, trivially partitioning every single state as an independent subMDP can result in a significant increase in the size of $|\mathcal{E}|$.
- **Strongly connected SubMDPs**:
  Another desired property of a partition Z is that it should induce strongly-connected subMDPs.

  Definition 4 *A subMDP $\mathcal{M}_i$ is strongly connected if for each pair of states $s_i, s_j \in S_i \cup E_i$ there exists a policy π which, when starting in s_i, reaches s_j with a positive probability.*

  This property ensures that when we enter a subMDP $\mathcal{M}_i$ we can choose a policy π that with positive probability will let us reach a desired exit state $e \in E_i$.

We now motivate why the methodology presented in this paper learns representations that respect these properties.

- **Induced subMDP must be easy to solve**: The parametrized compression function $f_\psi : S \to \Delta(Z)$ proposed in this paper is able to control and balance the size of each subMDP by tuning the number of regions $|\mathcal{Z}|$ to fit on the state space $\mathcal{S}$. Moreover, the second term in the loss proposed in Equation 1 constrains the representation to balance the probabilities of belonging to regions, forcing the region size to be balanced and allowing us to control the maximum size M of each induced subMDP.
- **Bottleneck states**: The compression function presented here implicitly favors partitions that minimize the set of all exit states $\mathcal{E}$ as we can see in Figures 1 and 2. From the loss in Equation 1 we can tell that the two terms $\mathcal{L}_\mathcal{Z}$ and $\mathcal{L}_H$ will be minimized when the compression function f_ψ clusters strongly connected states together and at the same time balances the probabilities of belonging to regions. By tuning the weights w_H we could allow different sizes of regions and better match clusters that minimize the set of all exit states $\mathcal{E}$.
- **Strongly connected SubMDPs**: By minimizing the first term in Equation 1 we incentivize partitions that induce strongly connected subMDPs. The learned representation demonstrates a coupling to the behavior policy used to collect the dataset, which indirectly defines the distance between states (i.e. which states should be clustered together).

B Additional Empirical Evaluation

In this section we present an additional empirical evaluation of our approach to learn a compression function, complementing the analysis reported in the main text.

Concretely, we evaluate our approach in the MountainCar environment, in which the state consists of the current location and velocity of the agent. In this environment, we collected a replay memory consisting of 200 trajectories of length 200 using a sub-optimal policy that can reach the goal state and can approximately cover all the state space, and learned a compression function with 20 abstract states.

In Figure 7 we show the result of this compression function where different colors represent different abstract states $z \in Z$. Note that the compression function is able to cluster together states that are close in the environment, i.e. states where the car is at similar position and velocity. In particular, states with low velocity near the center are *not* very similar to states with high velocity in the same location, and this is captured by the compression function.

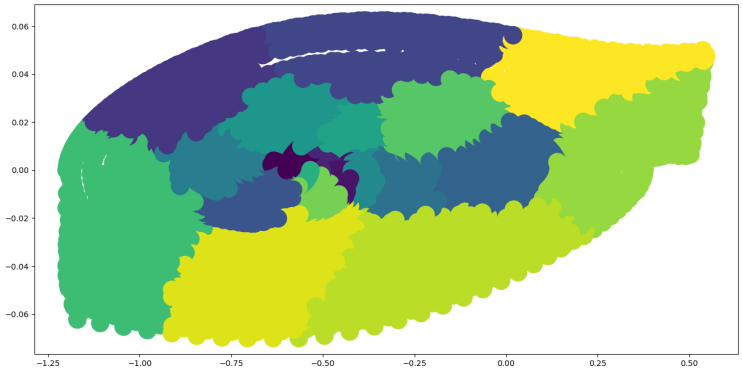

Figure 7: Results of the compression function in the MountainCar environment (axes represent location and velocity); different colors represent different abstract states $z \in Z$

C Computational Complexity Discussion

The proposed algorithm relies on off-policy DQN to learn the options while at the SMDP level we used tabular SMDP Q-learning on a small finite state space with a negligible computational cost.

The computational complexity of the proposed algorithm scales with the number of offline updates that we perform to train the options. This parameter can be tuned to adapt to the computational budget available.

D PSEUDO-CODE

Pseudo-code of the HRL algorithm at the manager level.

Algorithm 1 MANAGER

Input: environment e, previously discovered SMDP $\mathcal{S}$ in case of transfer learning, compression function g
$s \leftarrow initialstate$
$z \leftarrow g(s)$
if $z \notin Z$ **then**
  $Z \leftarrow Z \cup \{z\}$
  $O \leftarrow o_z^{exploration}$
end if
$\pi_{\mathbb{T}} \leftarrow$ initial policy
$o \leftarrow$ None
while within budget **do**
  if o is None or Terminate **then**
    $o \leftarrow$ GETOPTION$(\pi_{\mathbb{T}}, z, O)$
    $R = 0$
  end if
  $s', r, done \leftarrow e(o(s))$
  TRAINOPTION$(o, s, r, s', done)$
  $R = R + r$
  $z' \leftarrow g(s')$
  if $z' \notin Z$ **then**
    $Z \leftarrow Z \cup \{z'\}$
    $O \leftarrow O \cup \{o_{z'}^{exploration}, o_z^{z,z'}\}$
  end if
  if $z \neq z'$ **then**
    UPDATEPOLICY$(\pi_{\mathbb{T}}, z, o, R, z')$
    $o \leftarrow$ GETOPTION$(\pi_{\mathbb{T}}, z', O)$
  end if
  if s' is terminal and s' not in Z **then**
    $O \leftarrow O \cup \{o_z^{z,s'}\}$
    $Z \leftarrow Z \cup \{s'\}$
  end if
  $(z, s) \leftarrow (z', s')$
end while

E HYPERPARAMETERS

Table 1 reports the values of the hyperparameters used to train the compression function and the HRL agent. Table 2 reports the value of the hyperparameters used to train the DQN-PER and SIL agents.

Hyperparameters	***Value***
Worker Hyperparameters	
Neural Network Architecture	CONV1(32, (7, 7), (1, 1)) FC1(32) FC2(32)
Activation Function	Relu
Learning rate	0.001
Optimizer	Adam
E-Greedy decay	0.9998
Batch size	100
Target network poliak update	0.05
Discount Factor	0.95
Replay buffer size	$5 * 10^5$
Replay type:	PrioritizedExperience Replay
Exponent for prioritization	0.6
Bias Correction	0.1
Manager Hyperparameters	
E-Exploration to learn the model	0.995
Discount Factor	0.95
Compression Function Hyperparameters	
Neural Network Architecture	CONV1(16, (1, 1), (1, 1)) BatchNorm2D(16) CONV2(32, (5, 5), (1, 1)) BatchNorm2D(32) CONV3(32, (3, 3), (1, 1)) BatchNorm2D(32) FC1(64) BatchNorm1D(64) FC1(1)
Activation Function	Selu
w_H, w_D on GridWorld	0.2, 0.1
w_H, w_D on Mountain Car	2, 0.1
Learning rate	0.001
Optimizer	AdamW
Batch size	32, 64
Epochs	4000

Table 1: Hyperparameters used to train HRL and HRL-Transfer agents.

Hyperparameters	***Value***
DQN-PER Hyperparameters	
Same as Worker	
SIL Hyperparameters	
Neural Network Architecture	Same as Worker
Discount Factor	0.95
Replay type:	Prioritized Experience Replay
Exponent for prioritization	0.6
Bias Correction	0.1
Additional Hyperparameters	Same as reported in Self Imitation Learning paper
OC Hyperparameters	
Neural Network Architecture	Same as Worker
Discount Factor	0.99
N of Options	Best between [2, 4, 8]
Rollout Length	5
Entropy weight	0.001

Table 2: Hyperparameters used to train SIL, DQN-PER and OC agents.