Learning Options for an MDP from Demonstrations

Marco Tamassia, Fabio Zambetta, William Raffe, and Xiaodong Li

RMIT University, Melbourne VIC, Australia
{first.last}@rmit.edu.au

Abstract. The options framework provides a foundation to use hierarchical actions in reinforcement learning. An agent using options, along with primitive actions, at any point in time can decide to perform a macro-action made out of many primitive actions rather than a primitive action. Such macro-actions can be hand-crafted or learned. There has been previous work on learning them by exploring the environment. Here we take a different perspective and present an approach to learn options from a set of experts demonstrations. Empirical results are also presented in a similar setting to the one used in other works in this area.

Keywords: reinforcement learning, options.

1 Introduction

A Markov Decision Process (MDP) is a formal model of decision processes in a stochastic stationary environment. MDPs are currently being used in a wide variety of problems including robotics ([9]), games ([19, 2]), and control ([13]).

An MDP includes a set of states, a set of possible actions, a model of the environment (that is, information on how the state of the environment varies in response to the agent's actions) and a reward function specifying how good each state is.

Given an MDP, a variety of techniques can be used to compute an optimal behavior with respect to the specified reward. A behavior is termed *policy* and, in the simplest form, it is a mapping from states to actions. Among the existing techniques, the most used are dynamic programming, temporal differences and Monte Carlo methods.

The field studying these techniques is called *Reinforcement Learning* (RL) ([20]). RL can be considered as a branch of Machine Learning, which deals with strategic behaviors.

This model, however, suffers the so-called "curse of dimensionality": each of the currently known algorithms has at least a quadratic computational time complexity in the number of states. This translates in a prohibitive computational time required for states sets of important magnitude.

This is more evident in cases where a state is represented by a combination of features, making thus the states set exponentially large in the number of features.

S.K. Chalup et al. (Eds.): ACALCI 2015, LNAI 8955, pp. 226–242, 2015.

To address this issue, [21] introduced a framework for abstraction of the states of MDPs. The central concept of their work is that of an *option*, which is an extended course of action, and can be thought as an abstract action or a macro-action. Options are not actions that are really available to the agent but, rather, represent sequences of primitive actions performed over an extended period of time.

When options are used along with actions in a dynamic programming algorithm, an approximation of the optimal policy can be computed in a short amount of time (compared to that of the same setting but with no options). Options can also be used within a variation of the Q-learning algorithm ([21]). Also in this case the time necessary to complete the process is cut by means of using options.

As the "no free lunch" principle suggests, the computational burden is not eliminated but, rather, is moved to a precomputation step in which options are learned. However, since options are defined over a subset of the states set, computing their internal policy is potentially much faster than finding the optimal policy without options.

Such an abstraction mechanism arguably offers a strong advantage, but the drawback is that of having to manually specify the options to be adopted. Previous work has been done to automate the learning of options ([11, 18, 4, 17, 10, 22, 3]): note that these approaches rely on the agent exploring the state space and inferring potentially good options.

The purpose of this work is to introduce an algorithm that learns options from one or more demonstrations provided by experts. This is a different use case than that in the mentioned literature. Whereas in those works options were learnt by interacting with the environment, we do not assume such interaction possible but, rather, we assume data about other agents interacting with the environment is available.

It could be said that learning from a demonstration is a form of Supervised Learning (SL). However, while on an abstract level this is correct, the reader should not form the idea that Supervised Learning algorithms (such as Neural Networks or Support Vector Machines, to name the most famous) would work well on a Reinforcement Learning problem.

This is likely not to happen because RL algorithms take advantage of information of the structure of the environment to plan actions in order to maximize the cumulative reward *over time*, while SL algorithms do not.

In the context of learning from a demonstration, a SL algorithm would copy state-actions pairs from the demonstration, but it would not be able to generalize effectively from them because it would have no model of the environment.

This is a well known difference between SL and RL and it is furtherly explained in [20].

The idea of learning parts of an MDP from a demonstration is not new. The field of *Inverse RL* (IRL) studies methods of learning the reward function from a demonstration. The first problem has been formulated as learning the reward function used by an expert ([14]) while in later work the objective has

been relaxed to learning a reward function that makes the agent behave like the expert ([1, 16, 25, 12, 8]).

Nevertheless, to the best of our knowledge, learning options from demonstrations is a still unexplored area. It is argued in [7] that by using options in a Q-learning process, bias is introduced. This can potentially be an advantage if the bias is well directed. In a settings where options are learned from experts demonstrations, this is possibly the case.

2 MDPs and Options

This section introduces the Markov Decision Process (MDP) formalism which is used in this work (2.1) and gives an introduction to the options framework (2.2).

2.1 MDP and Reinforcement Learning

An MDP is a model of a stochastic, stationary environment; that is, an environment whose response to an agent's actions is non-deterministic but follows a distribution that does not change over time.

Formally, an MDP M is a tuple

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

where $\mathcal{S}$ is the set of states; $\mathcal{A}$ is the set of actions; $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a function expressing the environment state transition probabilities; $\gamma < 1$ is a discount factor used to decrease future rewards; $R : \mathcal{S} \to \mathbb{R}$ is a function associating a reward to each state.

At each point in time t, the environment is in a state s_t, and an agent receives a reward $r_t = R(s_t)$. After collecting the reward, an agent performs an action a_t. The purpose of an agent is to maximize the discounted, cumulative reward collected over time: $\sum_t \gamma^{t-1} r_t$. The agent, therefore, wants to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes such reward.

Such a policy can easily be computed if the expected future reward of taking action a in state s is known for all a and s. It is possible to define a function $Q^* : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ to capture this notion. Such function is called *state-action value function* and is recursively defined as follows:

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} T(s, a, s') \left(R(s) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a') \right).$$

An optimal policy executes the action with the highest expected future reward:

$$\pi^*(s) = \arg\max_a Q^*(s, a).$$

The optimal Q^* function can be learned by interacting with the environment step by step. The most widely used algorithm to this end is Q-learning, introduced in [24]. The algorithm approximates the optimal Q-function by updating any initial estimate of Q at each state transition (s_t, a_t, s_{t+1}) as follows:

$$Q(s_t, a_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t),$$

where $x \overset{\alpha}{\leftarrow} y$ is a short for $x \leftarrow x + \alpha y$ and $0 < \alpha \leq 1$ is a learning factor. This algorithm converges with probability 1 to the optimal Q^* function.

A good introduction to RL is [20].

2.2 Options Framework

The concept of *option* generalizes that of action (which, in the options framework, are called *primitive actions*) to include temporally extended courses of action. It has been introduced in [21] to accelerate the convergence rate of RL algorithms. An option, like any other action, can be chosen by a policy; however, differently from primitive actions, when an option is chosen, an auxiliary policy is followed for some period of time - until the option execution terminates.

An option o is defined as a tuple $o = (\mathcal{I}, \pi, \beta)$ where:

- $\mathcal{I}$ is the *initiation set*; that is, the set of states from which it is possible to select option o;
- π is the policy to be used when option o is selected;
- $\beta : \mathcal{S} \to \mathbb{R}$ is a function expressing the termination probability; that is, $\beta(s)$ expresses the probability with which option o terminates when in state s.

The intuitive idea behind this is to generalize a policy so that it chooses what to do next not just from the set of primitive actions $\mathcal{A}$ but more generally from a set of options $\mathcal{O}$, which may or may not include the primitive actions in $\mathcal{A}$. Policies over options are denoted with the symbol μ.

Notice that the best policy $\mu^*_{\mathcal{O}}$ over a set of options $\mathcal{O}$ is not guaranteed to be as good as the best policy $\pi^*_{\mathcal{A}}$ over the set of primitive actions $\mathcal{A}$. This is because the options in $\mathcal{O}$ may not necessarily have the *granularity* required to achieve an optimal behavior. However, an approximation of $\mu^*_{\mathcal{O}}$ can be computed in potentially fewer steps than $\pi^*_{\mathcal{A}}$, because one option execution can transition to a state that is reachable only by many primitive actions executions.

Furthermore, if $\mathcal{O} \supseteq \mathcal{A}$, the best policy over $\mathcal{O}$ is at least as good as the best policy over $\mathcal{A}$. This framework, while maintaining the advantages of reasoning with policies, only adds a slight computational burden with respect to the setting that reasons only with actions ([21]).

When a policy over options in state s_t selects an action $a \in \mathcal{A}$ as the next move, the action is executed as in a usual MDP policy. On the other hand, when it selects an option $o \in \mathcal{O}$, the option policy determines the actions to be performed until it randomly terminates in s_{t+k}, at which time a new action or option is selected.

Notice that, to treat options as primitive actions - in order to be able to use existing RL algorithms - a model of each option o is required. A complete model of o in this formalism consists of the transition probabilities $T_o(s, s')$ associated to o. T_o expresses the probability of macro-transitioning from state s

to state s' when selecting option o. That is, actually, the probability that option o terminates in state s' when started in state s.

If such a model is not known, it is still possible to learn the Q-function (now called *state-option value function*) by exploring the environment. For this purpose, a generalization of the Q-learning algorithm for options, introduced in [21], can be used. The Q-learning update rule, for each option execution (s_t, o_t, s_{t+k}), is defined as follows:

$$Q(s_t, o_t) \overset{\alpha}{\leftarrow} r + \gamma^k \max_{o \in \mathcal{O}} Q(s_{t+1}, o) - Q(s_t, o_t),$$

where k is the number of steps between s and s' and $r = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k}$ is the discounted cumulative reward obtained over this time. Using such rule at the termination of each option leads Q to converge to the optimal state-option value function Q^*. Notice that, in case o_t is a primitive action, $k = 1$ and the update rule is the same as the original Q-learning, except that the choice for the next action is extended to options.

Performing updates only at the end of options executions, however, takes a long time to converge. To accelerate this process, *intra-option learning* can be used ([21]). On every single state transition (s_t, a_t, s_{t+1}), all options $o = (I, \pi, \beta)$ such that $\pi(s_t) = a_t$ are updated as follows:

$$Q(s_t, o) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma U(s_{t+1}, o) - Q(s_t, o),$$

where

$$U(s_t, o) = (1 - \beta(s_t)) Q(s_t, o) + \beta(s_t) \max_{o'} Q(s_t, o').$$

This algorithm also converges to the optimal state-option value function.

It is also possible to use a variation of the mechanism so to allow policies to terminate when termination is a better alternative. Indeed, given two policies μ and μ' that are identical except for the fact that μ' terminates an option o in states s where $Q^{\mu'}(s, o) < V^{\mu'}$, it can be proven that $V^{\mu'} \geq V^{\mu}$ ([21]). The computational burden added by this variation is negligible.

3 Learning Options

This section explains how it could be possible to learn options from one or more demonstrations. As mentioned above, this approach is different from [11, 18] since in these works the agent needs to explore the environment while in our work one or more expert demonstrations are required.

This approach has, however, some constraints. The first constraint is that the state and the action spaces must be finite. The second constraint of this mechanism is that the policy of each option o must be based on a **single-peak** reward function R_o; in other words, option o represents the high-level intent of reaching a specific subgoal s_{k_o}. This is consistent with work in [11, 18, 4, 17].

That is to say, given s_{k_o}, R_o assumes a non-zero value c only for state s_{k_o}. Formally, for some $c > 0$, $R_o(s) = c \cdot \delta_{s,s_{k_o}}$, where δ is the Kronecker's delta:

$$\delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

Such a restriction reduces the space of reward functions to a tractable size. This allows the use of **memoization** over calls of any RL algorithm A. By means of memoization, $A(R)$ needs to be called only the very first time for each R: by storing the result in memory, subsequent calls to $A(R)$ can be avoided. Notice that this is possible since states and action spaces are finite.

It can be argued that memoization is not critical in this task, but the cost of computing optimal policies with model-based RL algorithms is high, and we believe the speed up given by not having to recalculate the policy of options is important. However, we do not have quantitative data to support such claim.

Furthermore, this very specific rewards space allows for a very **concise representation** of a reward function. In fact, supposing parameter c is fixed, all the necessary information to represent a reward function R is the state in which R assumes the non-zero value. For example, the reward function expressed above can be represented entirely by the scalar k_o.

3.1 Identify Useful Subgoals

Our procedure elaborates on one or more demonstrations given as input. The first step of our approach is to identify useful subgoals. We base our algorithm on the assumption that, if a subgoal is useful, at least one of the demonstrations makes good use of it.

The idea for identifying such subgoals, is to find the smallest set $\mathcal{L}$ of options that allows to equivalently rewrite all of the demonstrations expressing the steps in the form $(s_i, \pi_o(s_i))$, with $o \in \mathcal{L}$.

Formally, this means, given a demonstration d, which is a sequence of state-action pairs (s_i, a_i), $d = \langle (s_1, a_1), (s_2, a_2), \ldots, (s_{n-1}, a_{n-1}), (s_n, a_n) \rangle$, to find the smallest set $\mathcal{L} = \{o_1, o_2, \ldots, o_m\}$ such that

$$\begin{aligned} a_1 &= \pi_{o_1}(s_1) \\ a_2 &= \pi_{o_1}(s_2) \\ &\vdots \\ a_i &= \pi_{o_1}(s_i) \\ a_{i+1} &= \pi_{o_2}(s_{i+1}) \\ a_{i+2} &= \pi_{o_2}(s_{i+2}) \\ &\vdots \\ a_n &= \pi_{o_m}(s_n). \end{aligned}$$

By using options in set $\mathcal{L}$, it is possible to identify subsequences in d. Each such subsequence d_x is composed of state-action pairs that can be rewritten in the form $(s_i, \pi_{o_x}(s_i))$:

$$d_x = \langle (s_{i+1}, \pi_{o_x}(s_{i+1})), (s_{i+2}, \pi_{o_x}(s_{i+2})), \ldots, (s_j, \pi_{o_x}(s_j)) \rangle.$$

Given this, it is possible to rewrite d as $\langle d_1, d_2, \ldots, d_m \rangle$ (with a little abuse of notation), which expands to:

$$\begin{aligned} d = \langle & (s_1, \pi_{o_1}(s_1)), (s_2, \pi_{o_1}(s_2)), \ldots, (s_i, \pi_{o_1}(s_i)), \\ & (s_{i+1}, \pi_{o_2}(s_{i+1})), (s_{i+2}, \pi_{o_2}(s_{i+2})), \ldots, (s_j, \pi_{o_2}(s_j)), \\ & \vdots \\ & (s_{k+1}, \pi_{o_m}(s_{k+1})), (s_{k+2}, \pi_{o_m}(s_{k+2})), \ldots (s_n, \pi_{o_m}(s_n)) \rangle. \end{aligned}$$

3.2 Reduce the Number of Options

Set $\mathcal{L}$ includes all the options that are necessary to equivalently rewrite d as specified above. However, some of these options may not differ much from each other. For this reason, we perform a clustering step to group "similar" options.

The similarity concept that is desired here is captured by the likelihood of transitioning from one state to another state, assuming the best action to this end is chosen. We thus define the distance between states s_i and s_j as follows:

$$\Delta(s_i, s_j) = \min_{a \in \mathcal{A}} \frac{1}{T(s_i, a, s_j)},$$

where $T(s_i, a, s_j)$ is the probability of transitioning from state s_i to state s_j when executing action a.

This distance measure is then used to build a probability matrix. To this end, a graph is built and then the *all-pairs shortest path* algorithm is used ([6]). This algorithm has a complexity $O(|S|^3)$, however, the idea is that this algorithm is run only once in a precomputation phase. The distance matrix is then fed to the *DBSCAN* clustering algorithm ([5]). DBSCAN has been chosen because it does not require a specification of the number of clusters, unlike many other clustering algorithms.

The clusters are sorted by size and a random representative from the top k clusters is selected, where k is an arbitrary constant. The so-chosen representatives form the set of learned subgoals, and can then be enriched with an initiation set, a policy and a distribution of termination probabilities and so be transformed in options.

3.3 Limitations

This method relies on an RL algorithm that computes the policies induced by the subgoals. However, this goes against the advantage of computing options at all. Indeed, for large state spaces, this computation would be prohibitive.

Similarly, if the options model is computed by using the environment model (in the form of the transition probabilities function), the cost of such a computation would be prohibitive.

A simple approximation is to limit the computation performed by the RL algorithm to only a subset $\mathcal{S}' \subset \mathcal{S}$ with a much lower dimensionality. This would make the computation of policies feasible.

Such simplification would produce a larger variety of options. While most of these options may actually be redundant[1], the computational burden added would not be significant ([21]).

The procedure we propose makes use of the transition probabilities $T(s_i, a, s_j)$ for computing the similarity of options. This is a strong assumption, but such information could also be approximated by using the data contained in the demonstrations.

Notice that, even knowing T in advance, traditional model-based RL could not be used because the reward function is not assumed to be known. The purpose of the learnt options is to speed up a later on-line learning phase.

4 Experiments

This section details the experimental setup and presents the results of our experiments.

4.1 Experimental Setup

The whole procedure has been implemented in Python using the Numpy library[2] ([23]) and the Scikit Learn toolkit[3] ([15]).

The Q-learning algorithm has been tested in a grid-world. The shape of such environment is very similar to that used in [21], where a square 13×13 world is divided in 4 areas by means of barriers. Figure 1 shows a representation of such environment. Barriers are interrupted to allow traveling from one room to another; these holes are called "hallways". The agent can move in the four cardinal directions north, south, east or west. The probability of transitioning in the desired direction is $\frac{2}{3}$; the agent moves in one of the other cardinal directions with probability of $\frac{1}{3}$, that is $\frac{1}{9}$ for each of them.

We used this topology in two different flavors.

- The four areas are separated by "walls" which are impassable no matter what. This is the setting used in [21].
- The four areas are separated by "ponds", which are not impassable, but just have a much lower reward. The rationale behind this choice is to make the hallways part of the paths *chosen* by the experts rather than *unavoidable* steps. This is because our algorithm detects as subgoals only states that

[1] In case they were generated only as substeps of another option that could not be completely computed due to this approximation

[2] `http://www.numpy.org/`

[3] `http://scikit-learn.org/`

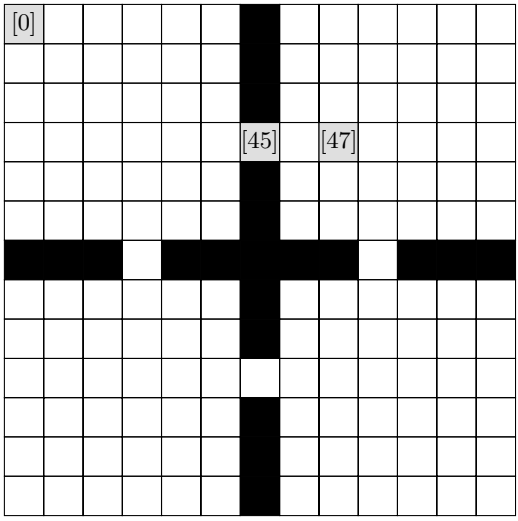

Fig. 1. The grid-world used in the experiments. The gray squares represent the destinations used in the different experiments: the one in the upper corner is labeled as [0], the one in the hallway is labeled [45] and the other one is labeled [47]. These labels will be used to present the results of the experiments. The black squares represent unreachable states in the wall setting and low rewards states in the ponds setting.

the experts explicitly chose among the others - as opposed to states that were simply not avoidable.

Specifically, while normal states have a reward of 0, ponds have a reward of -10. The final state has reward 1000 and, when reached, terminates the execution.

The key difference between the two flavors emerges because the demonstrations are generated by using a reward function that is unknown to the algorithm analyzing them. Consequently, the best choice to reach a particular state can appear suboptimal to the algorithm analyzing the demonstration. For example, a demonstration in which the expert is avoiding a pond by taking a longer path will be considered sub-optimal by our algorithm which does not know about the existence of the pond, whose existence is encoded in the unknown reward function.

In each of the flavors, different sets of experiments are run: each set of experiments sets a different destination for the agent. The states we chose as destinations are shown in Figure 1. We selected these destinations for specific reasons: one of them is in a corner, away from most common paths; another one is in a hallway, which is one of the hand-crafted options; the third one is close to a hallway and, as such, is part of many common paths.

By choosing different destinations, it is possible to compare the performance of all sets of options in different situations: in fact, if the options are close to the destination, the agent is advantaged because its exploration is biased in a

convenient way; on the other hand, if the options are far from the destination, for the same reason, this is a disadvantage for the agent.

The value of the discount factor of the MDP is $\gamma = 0.97$. The options are generated from the subgoals o by using a reward function $R_o(s) = c \cdot \delta_{s,s_{k_o}}$ with $c = 50$, where s_{k_0} is the subgoal captured by option o by making use of the model of the environment T. The exploration strategy used in the experiments is ϵ-greedy, with $\epsilon = 0.1$.

The purpose of the experiments is to compare the quality of hand-crafted options, the hallways, denoted by $\mathcal{H}$, and that of options learned from demonstration, denoted $\mathcal{L}$. These sets of options are also compared against the set of primitive actions only, denoted $\mathcal{A}$. Sets $\mathcal{A} \cup \mathcal{H}$ and $\mathcal{A} \cup \mathcal{L}$ are also tested,

We select the handcrafted options as the baseline for comparison, rather than a random selection of states, because the former are supposed to perform better. This is because, due to the structure of the environment, a hallway state is for sure in any path that travels between two different rooms. Given this, their contribution to the Q-learning algorithm is supposedly more useful than that of other random locations.

4.2 Results

In the following we will often refer to "bad" options. By such term, we refer to options that lead to an area in the state space that is far away from the destination state the agent is pursuing.

It could be argued that, since all the options selected by our algorithm are part of the optimal path in a demonstration, there cannot be such a thing as a "bad" option. This can, however, happen since options are inferred from more than one demonstration, and are then likely to be sparse around the state space.

Figures 2, 3, 4, 5, 6 and 7 show the result of the experiments that have been run. On the X axis, the number of episodes is presented, while on the Y axis the (average) number of steps per episode is reported.

The plots show the average performance on 200 experiments run over 3000 episodes. In particular, for the curves representing the performance of learned options, for each of the 200 repetitions, a new set of demonstrations was randomly generated. The number of experiments is high because, since the perfomance depends on the random starting state, there is high variance.

All of the experiments have been run on both the "pond" and "wall" settings and on all of the destinations, [0], [45], [47] (see Figure 1 to visually identify these destinations). Results are presented using a sliding window with size of 20 to average values and hence smooth the curves.

The demonstrations used to learn the options have been generated by means of an automatic process. Each demonstration was computed by selecting a random initial state and using the optimal policy (computed by means of the value iteration algorithm) to simulate the behavior of an agent pursuing the destination. For each of environment, options set, destination and repetition, 4 demonstrations were generated and were used to learn options.

In the following, it will often be said that an option is close or far from a destination point: this is actually a short way to say that the subgoal which is captured by an option is close or far from that destination point. For the sake of conciseness, we will stick to the former, shorter version.

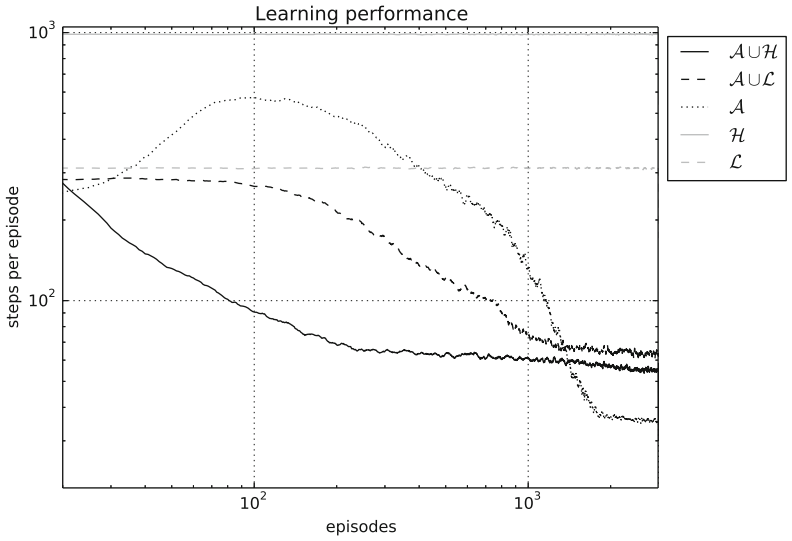

Fig. 2. This and the following figures show the result of experiments in the simulated environment described in Subsection 4.1. In all the figures, $\mathcal{A}$ denotes the set of primitive actions, $\mathcal{H}$ denotes the set of handcrafted options, which lead to each of the four hallways, and $\mathcal{L}$ denotes the set of learnt options. This figure shows the results obtained in the "pond" setting with destination [0]. For details, see Subsection 4.2.

Figures 2 and 3 show the results of the experiments where the agent's destination is the top-left corner, labeled as [0] in Figure 1.

In the pond setting, in Figure 2, both sets $\mathcal{A} \cup \mathcal{H}$ and $\mathcal{A} \cup \mathcal{L}$ both converge, but the former is consistently better than the latter. Set $\mathcal{A}$ performs worse at the beginning but ends up being the best performer, as one would expect. We interpret this difference to be due to the ϵ-greedy strategy: when an agent randomly chooses a "bad" option, it ends up further away than it would by just choosing a "bad" action. Set $\mathcal{H}$ performs very poorly here, because the distance of the options from the destination make it very unlikely that the destination is ever reached. Set $\mathcal{L}$ performs quite poorly as well, even though it shows a slight advantage.

In the "wall" setting, in Figure 3, set $\mathcal{A} \cup \mathcal{L}$ and set $\mathcal{L}$ both outperform sets $\mathcal{A} \cup \mathcal{H}$ and $\mathcal{A}$. While the former is expected to perform worse, it is not obvious for the latter. After investigating in the logs, we found out that this is due to the learned options in this setting. As a consequence of the topology, all of the expert's decisions, inferred by the demonstrations, are explained by the topology (rather than the reward function, as in the pond setting). As a consequence, most

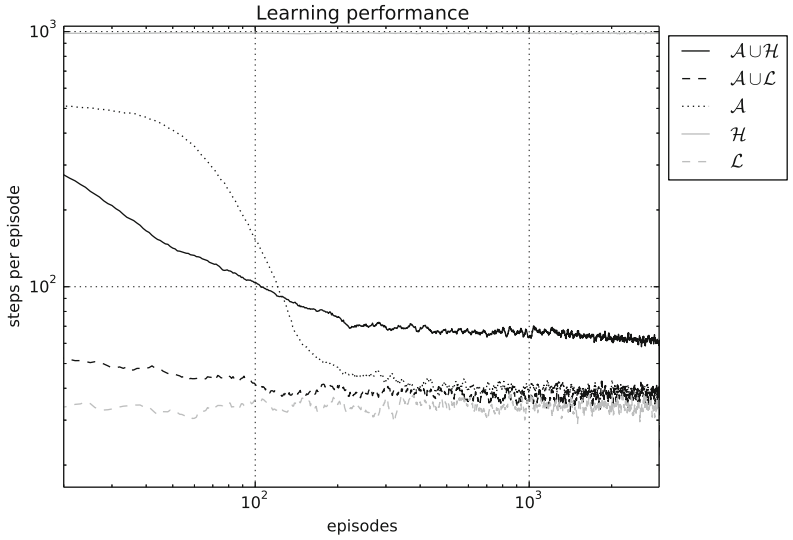

Fig. 3. This figure shows the results obtained in the "wall" setting with destination [0]. For further information, see Figure 2.

of the times the only option that is learned in this setting is the destination itself. The fact that virtually only one option is available, beside the primitive actions, eliminates the disadvantage caused by the ϵ-greedy exploration strategy. Finally, for the same reason as in the pond setting, set $\mathcal{H}$ performs very poorly.

Figures 4 and 5 show the results of the experiments where the agent's destination is the hallway, labeled as [45] in Figure 1.

In the pond setting, in Figure 4, the best performers are clearly sets $\mathcal{H}$ and $\mathcal{A} \cup \mathcal{H}$. This is because the destination is exactly one of the options, making it very quick for the agent to find a good path to the destination. Set $\mathcal{A}$ also performs well after an exploration phase. Set $\mathcal{A} \cup \mathcal{L}$ converges with a slight disadvantage with respect to set $\mathcal{A}$. Finally, set $\mathcal{L}$ is outperformed by all of the others: this shows how having options distant from the destination and not having primitive actions make it unlike that the destination is reached.

In the wall setting, in Figure 5, the sets $\mathcal{H}$ and $\mathcal{A} \cup \mathcal{H}$ both perform very well, once more, because the destination is one of the subgoals. In this case, also sets $\mathcal{L}$ and $\mathcal{A} \cup \mathcal{L}$ perform well, again because the topology makes it so that, most of the times, the only learned option is the destination. Also set $\mathcal{A}$ performs well, even though it takes longer to converge.

Figures 6 and 7 show the results of the experiments where the agent's destination is the state on the left of the hallway, labeled as [47] in Figure 1.

In the pond setting, in Figure 6, sets $\mathcal{A} \cup \mathcal{L}$, $\mathcal{A} \cup \mathcal{H}$ and $\mathcal{A}$ perform very well. Sets $\mathcal{H}$ and $\mathcal{L}$ perform unexpectedly poorly. We hypothesize that, due to the low rewards received, the algorithm does not have sufficient information to tell which of the option is the best one to choose since they all appear equally bad.

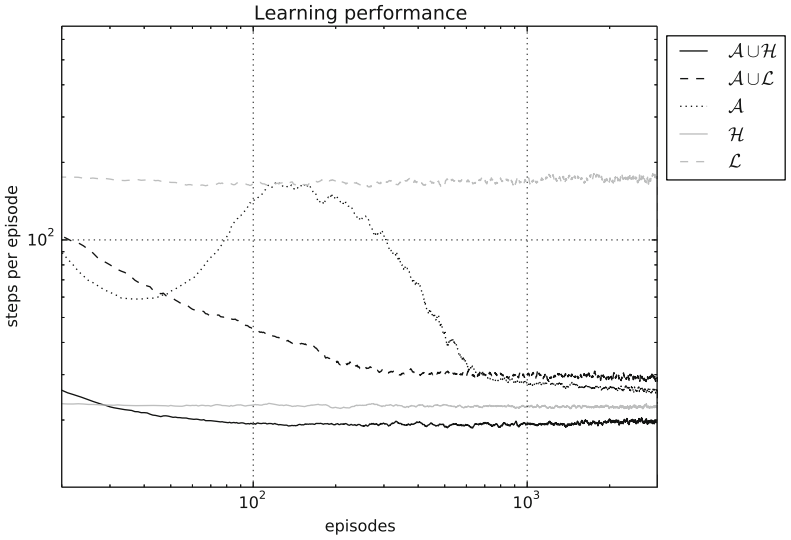

Fig. 4. This figure shows the results obtained in the "pond" setting with destination [45]. For details, see Figure 2.

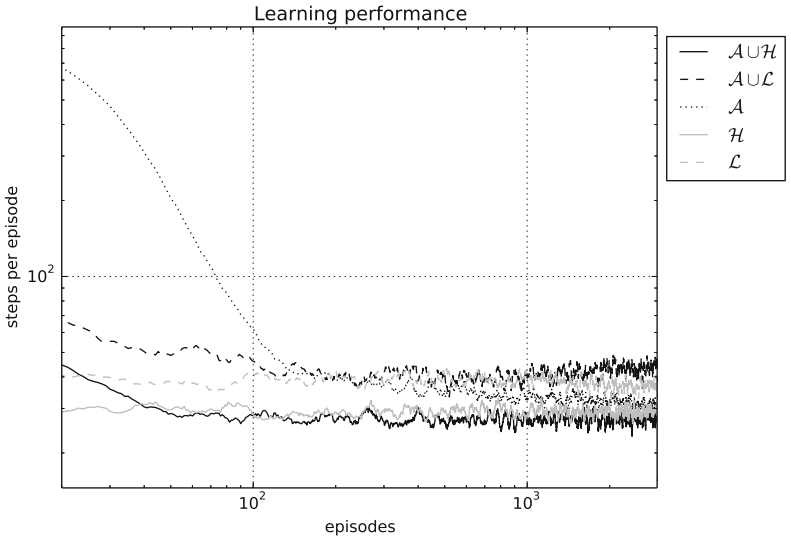

Fig. 5. This figure shows the results obtained in the "wall" setting with destination [45]. For details, see Figure 2.

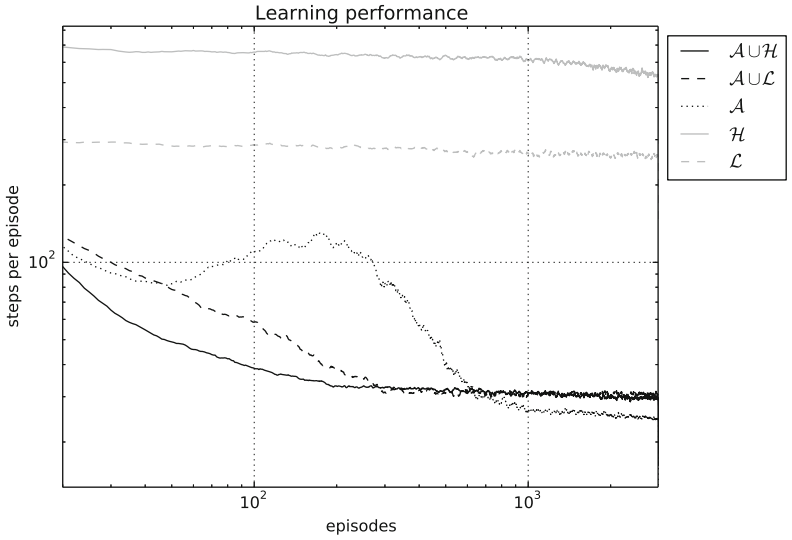

Fig. 6. This figure shows the results obtained in the "pond" setting with destination [47]. For details, see Figure 2.

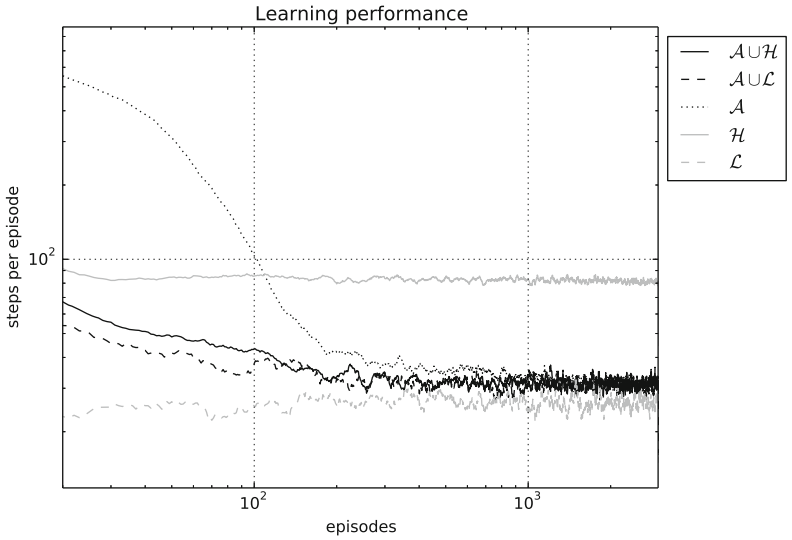

Fig. 7. This figure shows the results obtained in the "wall" setting with destination [47]. For details, see Figure 2.

In the wall setting, in Figure 7, the performance of sets $\mathcal{A} \cup \mathcal{L}$ and $\mathcal{L}$ are about the same as the pond setting. Set $\mathcal{A}$ reaches convergence more quickly and set$\mathcal{A} \cup \mathcal{H}$ performs better in general. Set $\mathcal{H}$ also perform better than in the pond setting; however, since the destination is only close to the options but not coincident, reaching it takes some luck (i.e. time).

When used along with primitive actions, learned options perform about as well as hand-crafted options, with a slight advantage in most cases. Furthermore, it can be noticed that the bias introduced by the options in the random exploration provides a speed-up in the learning process, letting the agents reach a stable performance after only a few tens of episodes. This is one of the cases in which the introduction of a bias (as mentioned in [7]) is benefiting the learning process; since it is derived by the behavior of experts, the exploration tends to follow the footsteps of the experts.

5 Conclusion

We have showed that learning options for an MDP from a demonstration is a viable choice. Options can greatly improve learning and planning performance. Previous work has shown that they can be hand-crafted ([21]) or learned by exploring the environment ([11, 18, 4, 17, 10, 22, 3]). With this work we show that options can also be learned from demonstration, which is a good choice in a context where exploration is not possible but significant amounts of data are available.

We also showed that different settings give rise to different behaviors both in learned and hand-crafted options. In particular, when the topology constrains the experts' behavior, learned options usually correspond to the destination; on the other hand, when a behavior is a consequence of the rewards, which are unknown to the agent, learned options are arguably more significative (in the sense that they resemble more to landmarks) but are less efficient in the learning phase.

Future works include testing this approach against real data, possibly in the form of a bigger dataset. A possible extension of this research is to let the algorithm work in continuous states and actions spaces.

References

1. Abbeel, P., Ng, A.Y.: Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the 21st International Conference on Machine Learning, ICML 2004, pp. 1–8. ACM, New York (2004), http://doi.acm.org/10.1145/1015330.1015430, doi:10.1145/1015330.1015430
2. Baxter, J., Tridgell, A., Weaver, L.: Knightcap: A chess programm that learns by combining TD(lambda) with game-tree search. In: Proceedings of the Fifteenth International Conference on Machine Learning, ICML 1998, pp. 28–36. Morgan Kaufmann Publishers Inc., San Francisco (1998), http://dl.acm.org/citation.cfm?id=645527.657300

3. Cobo, L.C., Subramanian, K., Jr., C.L.I., Lanterman, A.D., Thomaz, A.L.: Abstraction from demonstration for efficient reinforcement learning in high-dimensional domains. Artificial Intelligence 216(0), 103 (2014), http://www.sciencedirect.com/science/article/pii/S0004370214000861, doi:10.1016/j.artint.2014.07.003
4. Şimşek, Ö., Wolfe, A.P., Barto, A.G.: Identifying useful subgoals in reinforcement learning by local graph partitioning. In: Proceedings of the 22nd International Conference on Machine learning, ICML 2005, pp. 816–823. ACM, New York (2005), http://doi.acm.org/10.1145/1102351.1102454, doi:10.1145/1102351.1102454
5. Ester, M., Kriegel, H.-P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Simoudis, E., Fayyad, U., Han, J. (eds.) Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, vol. 96, pp. 226–231. AAAI Press (1996)
6. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM 5(6), 345–349 (1962), http://doi.acm.org/10.1145/367766.368168, doi:10.1145/367766.368168
7. Jong, N.K., Hester, T., Stone, P.: The utility of temporal abstraction in reinforcement learning. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2008, vol. 1, pp. 299–306. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2008), http://dl.acm.org/citation.cfm?id=1402383.1402429
8. Klein, E., Geist, M., Pietquin, O.: Batch, off-policy and model-free apprenticeship learning. In: Sanner, S., Hutter, M. (eds.) EWRL 2011. LNCS, vol. 7188, pp. 285–296. Springer, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29946-9_28
9. Kober, J., Peters, J.: Reinforcement learning in robotics: A survey. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning. Adaptation, Learning, and Optimization, vol. 12, pp. 579–610. Springer, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-27645-3_18, doi:10.1007/978-3-642-27645-3_18
10. Mannor, S., Menache, I., Hoze, A., Klein, U.: Dynamic abstraction in reinforcement learning via clustering. In: Proceedings of the 21st International Conference on Machine Learning, ICML 2004, pp. 71–78. ACM, New York (2004), http://doi.acm.org/10.1145/1015330.1015355, doi:10.1145/1015330.1015355
11. McGovern, A., Barto, A.G.: Automatic discovery of subgoals in reinforcement learning using diverse density. In: Proceedings of the Eighteenth International Conference on Machine Learning, ICML 2001, pp. 361–368. Morgan Kaufmann Publishers Inc., San Francisco (2001), http://dl.acm.org/citation.cfm?id=645530.655681
12. Lacasse, A., Laviolette, F., Marchand, M., Turgeon-Boutin, F.: Learning with randomized majority votes. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part II. LNCS, vol. 6322, pp. 162–177. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15883-4_25
13. Ng, A., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., Liang, E.: Autonomous inverted helicopter flight via reinforcement learning. In: Ang Jr, M.H., Khatib, O. (eds.) Experimental Robotics IX. Springer Tracts in Advanced Robotics, vol. 21, pp. 363–372. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11552246_35

14. Ng, A.Y., Russell, S.J.: Algorithms for inverse reinforcement learning. In: Proceedings of the Seventeenth International Conference on Machine Learning, ICML 2000, pp. 663–670. Morgan Kaufmann Publishers Inc., San Francisco (2000), http://dl.acm.org/citation.cfm?id=645529.657801
15. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. The Journal of Machine Learning Research 12, 2825–2830 (2011), http://dl.acm.org/citation.cfm?id=1953048.2078195
16. Ramachandran, D., Amir, E.: Bayesian inverse reinforcement learning. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI 2007, pp. 2586–2591. Morgan Kaufmann Publishers Inc, San Francisco (2007), http://dl.acm.org/citation.cfm?id=1625275.1625692
17. Şimşek, Ö., Barto, A.G.: Using relative novelty to identify useful temporal abstractions in reinforcement learning. In: Proceedings of the 21st International Conference on Machine Learning, ICML 2004, pp. 95–102. ACM, New York (2004), http://doi.acm.org/10.1145/1015330.1015353, doi:10.1145/1015330.1015353
18. Stolle, M., Precup, D.: Learning options in reinforcement learning. In: Koenig, S., Holte, R. (eds.) SARA 2002. LNCS (LNAI), vol. 2371, pp. 212–223. Springer, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45622-8_16
19. Stone, P., Sutton, R.S.: Scaling reinforcement learning toward robocup soccer. In: Proceedings of the Eighteenth International Conference on Machine Learning, ICML 2001, pp. 537–544. Morgan Kaufmann Publishers Inc., San Francisco (2001), http://dl.acm.org/citation.cfm?id=645530.655674
20. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning, 1st edn. MIT Press, Cambridge (1998)
21. Sutton, R.S., Precup, D., Singh, S.: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence 112(1–2), 181–211 (1999), http://www.sciencedirect.com/science/article/pii/S0004370299000521, doi:http://dx.doi.org/10.1016/S0004-37029900052-1
22. Vigorito, C., Barto, A.: Intrinsically motivated hierarchical skill learning in structured environments. IEEE Transactions on Autonomous Mental Development 2(2), 132–143 (2010), doi:10.1109/TAMD.2010.2050205
23. Walt, S., van, d. C.S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering 13(2), 22–30 (2011), http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2011.37, doi: http://dx.doi.org/10.1109/MCSE.2011.37
24. Watkins, C.J.C.H.: Learning from delayed rewards. Ph.D. thesis, University of Cambridge (1989)
25. Ziebart, B.D., Maas, A., Bagnell, J.A., Dey, A.K.: Maximum entropy inverse reinforcement learning. In: Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3, AAAI 2008, pp. 1433–1438. AAAI Press (2008), http://dl.acm.org/citation.cfm?id=1620270.1620297