Abstraction from Demonstration for Efficient Reinforcement Learning in High-Dimensional Domains[☆]

Luis C. Cobo[a], Kaushik Subramanian[b], Charles L. Isbell, Jr.[b], Aaron D. Lanterman[a], Andrea L. Thomaz[b]

[a]*School of Electrical and Computer Engineering, Georgia Tech, Atlanta, GA, 30332*
[b]*College of Computing, Georgia Tech, Atlanta, GA, 30332*

Abstract

Reinforcement learning (RL) and learning from demonstration (LfD) are two popular families of algorithms for learning policies for sequential decision problems, but they are often ineffective in high-dimensional domains unless provided with either a great deal of problem-specific domain information or a carefully crafted representation of the state and dynamics of the world. We introduce new approaches inspired by these two techniques, which we broadly call *abstraction from demonstration*. Our first algorithm, state abstraction from demonstration (AfD), uses a small set of human demonstrations of the task the agent must learn to determine a state-space abstraction. Our second algorithm, abstraction and decomposition from demonstration (ADA), is additionally able to determine a task decomposition from the demonstrations. These abstractions allow RL to scale up to higher-complexity domains, and offer much better performance than LfD with orders of magnitude fewer demonstrations. Using a set of videogame-like domains, we demonstrate that using abstraction from demonstration can obtain up to exponential speed-ups in table-based representations, and polynomial speed-ups when compared with function approximation-based RL algorithms such as fitted Q-learning and LSPI.

Keywords: Reinforcement learning, learning from demonstration, Dimensionality reduction, function approximation

1. Introduction

One of the hardest challenges in the field of machine learning is to build agents that can autonomously learn new tasks that they were not pre-programmed to tackle, without the intervention of an engineer. Agents such as robotic assistants in homes and hospitals or non-player characters in videogames may

[☆]This work is supported by ONR Grant No. N000141210483

Email addresses: luisca@gatech.edu (Luis C. Cobo), kausubbu@gatech.edu (Kaushik Subramanian), isbell@cc.gatech.edu (Charles L. Isbell, Jr.), lanterma@ece.gatech.edu (Aaron D. Lanterman), athomaz@cc.gatech.edu (Andrea L. Thomaz)

be required to perform tasks beyond the imagination of their designers, and different end users may require different behaviors. This makes it impractical to manually engineer a policy for all the possible tasks the agent may have to perform.

Reinforcement learning (RL) [57] and Learning from Demonstration (LfD) [4] are popular families of algorithms for learning policies for sequential decision problems, but they are often ineffective in high-dimensional domains unless provided with either a great deal of problem-specific domain information or a carefully crafted representation of the state and dynamics of the world. Unfortunately, autonomous agents learning new tasks usually do not have access to such domain information nor to an appropriate representation.

This paper proposes a middle ground between LfD and RL, introducing *abstraction from demonstration* algorithms, which take a small set of demonstrations from human teachers in order to infer the relevant features of the state space at each moment. Then using this abstracted state space allows significant speed-ups in reinforcement learning algorithms with respect to the number of state features in complex domains. Our experiments show that these speed-ups are exponential when using table-based representations and polynomial when used in conjunction with function approximation-based RL algorithms such as fitted Q-learning or LSPI. Compared with learning a policy with LfD, our approach can obtain significantly better performance with orders of magnitude fewer samples.

We describe two abstraction from demonstration algorithms. The first, state abstraction from demonstration (AfD) [12], builds a single state-space abstraction from a set of demonstrations. The second, abstraction and decomposition from demonstration (ADA) [10], uses the demonstrations to break down the task into smaller subtasks and finds a suitable state space representation for each subtask.

Our algorithms reduce the need for manual feature engineering, so they can be useful for multipurpose autonomous agents that must learn tasks that are not known in advance. Most research on machine learning assumes a suitable representation, *i.e.,* state space, for the problem to be solved, but finding this representation is often the hardest part of solving real-world tasks. Usually, this representation is found by machine learning practitioners manually, with a combination of experience, domain insight, and intuitive knowledge [16]. Our work opens the door to more automatic approaches to deriving appropriate representations for a problem. Even though our algorithms require a set of human demonstrations, they are obtained from normal people or end-users, which need not have any engineering or machine learning knowledge. This improves the agent's ability to learn new tasks after deployment without the intervention of an engineer.

The rest of this introduction first points out some shortcomings of LfD and RL to later show how our approach addresses them. We discuss the applicability of our approach and finally summarize the contents of the rest of the paper.

1.1. Learning from Demonstration

Learning from demonstration is an approach to robot/agent learning that takes as input demonstrations from a human in order to build action or task models. There are a broad range of approaches that fall under the umbrella of LfD research [4] but our work is focused on the class of algorithms that learn a policy for a sequential decision problem by reducing it to a supervised learning problem on a set of human demonstrations. These demonstrations are typically represented as state-action tuples, and the LfD algorithm learns a policy mapping from states (input) to actions (output) based on the examples seen in the demonstrations.

In principle, LfD could be effective in large state spaces because it focuses on the interesting regions of the state space, which are the same regions that are shown in the demonstrations. However, these approaches often do not work well unless either a detailed model of the problem is available or the learned policy is just used as a starting point for other learning techniques [2, 9]. The number of samples required to derive a good policy may be large, and obtaining human demonstrations is costly and time-consuming. Thus, an important consideration is how to use human demonstrations efficiently. Further, training and testing distributions for the algorithm may not match when the agent has not perfectly mimicked the teacher, because imitation errors can accumulate in consecutive time steps. This can lead the agent to an unknown part of the state space, where the performance of the algorithm may be extremely poor because there are no demonstrations from that region.

1.2. Reinforcement Learning

Reinforcement learning algorithms solve sequential decision problems posed as Markov decision processes (MDPs), learning a policy by letting the agent explore the effects of different actions in different situations while trying to maximize a sparse reward signal. RL has been successfully applied to a variety of scenarios [63, 44]; however, RL tends to not scale well to high-dimensional state spaces because of the *curse of dimensionality*. As the number of features that compose the state space grows linearly, the size of the state space grows exponentially, and so does the time required by the algorithm to converge on a policy.

This problem has been hitherto addressed by two different but related approaches: manual engineering of the features and function approximation. In manual feature engineering, an expert analyzes the task to be learned and, starting from a large set of low-level features, composes a small set of high-level features with which the agent can learn a compact and good policy for the task. This approach shifts the engineering task from designing a policy to designing the feature space in which the policy will be learned. The approach can be useful if the latter task is easier than the former; however, we cannot talk of true autonomous learning agents if a significant amount of manual feature engineering is needed for each task to be learned.

Function approximation, the other approach to deal with the curse of dimensionality in RL, offers significant speed-ups with respect to learning on a

flat tabular representation of the state space. Unfortunately, function approximation often requires manual feature engineering, as the hypothesis space of the approximation algorithm (*e.g.,* linear combinations of the features) may not be appropriate if used with low-level features.

1.3. Our Approach: Abstraction from Demonstration

Our approach is to devise algorithms that can automatically focus on a small set of relevant features at each point in time. We call these kinds of algorithms *abstraction from demonstration algorithms*.

We draw inspiration from humans. Our brains receive, at any given moment, up to 11 million pieces of information, but we can be consciously aware of at most 40 of these [66]. In addition, adults can only hold a maximum of three to five meaningful items or "chunks" in their working memory [13]. This indicates that humans are able to cope with the complexity of the tasks they face using aggressive dimensionality reduction of the sensory information they receive. This dimensionality reduction enables them to quickly learn compact policies that are nearly optimal. This ability is called *selective attention*; it allows us, for example, to focus on a specific conversation in a crowded environment with many people talking at the same time.

One way to leverage human selective attention would be to directly ask people which features are important for a task, but this is problematic for two reasons. First, even if humans had a clear and unbiased idea of what is important to them when performing a task, most people do not understand the internal representation of the state space in a learning agent, and therefore they will not be able to convey the information to the agent. Second, humans perform selective attention without being consciously aware of it [45], so they may not have accurate insight into which aspects are important to them when carrying out a task.

We sidestep those problems by obtaining all the necessary information from demonstrations by human teachers performing the task. With an adequate set of demonstrations, we can measure the mutual information between each feature of the state space (as the learning agent sees it) and the actions that the teachers take. With these measurements, we can determine which features are actually relevant to the teachers and focus on those features in subsequent learning. AfD, our first algorithm introduced in Section 4, builds a state-space abstraction for the whole task that excludes irrelevant features. ADA, our second algorithm introduced in Section 5, goes one step further and uses the demonstrations to find a subtask structure in the task to be learned, along with a separate state-space abstraction for each subtask.

1.4. Application Domains

Abstraction from demonstration algorithms exploit common patterns in typical real-world tasks, trying to imitate the tricks that humans use to leverage these patterns. Obviously, there is no perfect solution for the curse of dimensionality, which means that our algorithms will not be useful in arbitrary domains.

For example, a domain where the optimal action depends on an arbitrary function that depends on all the features of the domain would not benefit from our methods.

However, abstraction from demonstration methods can extend reinforcement learning to handle many real-world tasks that could not be tackled before because of their high dimensionality. Specifically, we focus on tasks that humans can carry out easily, but can be challenging for a computer. As we have seen, humans quickly find policies for complex tasks by performing aggressive feature selection on their sensory input. With such a strong simplification of the environment, humans do not look for optimal policies but for *satisficing* ones [53]. Satisficing is a portmanteau of the words satisfy and suffice, *i.e.,* satisficing policies are policies that are close enough to an optimal policy for the incentive of the optimal policy to offset the cost of finding it.

Abstraction from demonstration algorithms aim for satisficing policies for two reasons. First, we will often face NP-hard problems, like many videogames [3] or a simple transformation to an MDP of a travelling salesman problem. This means that, unless $P = NP$, there is no option but to sacrifice optimality for the sake of efficiency in complex domains. The second reason is that, often, a satisficing policy is more convenient than the optimal one because it can transfer better to slightly different versions of the same task. A perfect optimal policy for solving a level of a game of Pacman is not going to be useful for a different level. However, a policy that uses patterns in the area of the screen close to the acting agent may not be optimal but, if enough neighborhoods are recognized, it will likely work with any level. In short, satisficing policies are necessary for complex domains, and they are also beneficial for transfer learning.

With respect to representation, our algorithms are most useful with state representations relative to the agent and deictic representations [20]. In general, it is fair to assume that these representations are native representations because this will be the case for embodied agents that perceive the world through a series of sensors. It is also the natural representation for humans.

In our experiments we have used several videogames because they are convenient domains to simulate and for which to obtain human demonstrations. They also make it straightforward to match the human representation and the agent internal representation of the problem. However, our work applies to any kind of autonomous agent that needs to learn after deployment, for example, manufacturing robots in industry or robot assistants aiding at domestic tasks, automatizing procedures in hospitals, or providing assistance to elderly or disabled individuals. These multipurpose agents typically have a wide range of sensors that can provide a high-dimensional signal about the world. In general, this high-dimensional signal is too complex for direct learning. Abstraction from demonstration is likely not useful on top of this raw signal either, but these signals can be preprocessed using unsupervised feature learning techniques such as deep learning [33] to provide a set of features that is still too large for direct RL, but adequate for abstraction from demonstration approaches.

We focus on stochastic, episodic, infinite-horizon MDPs. In our domains, the state is encoded in a factored representation because these representations

enable generalization, *i.e.,* extending what is learned in one state to similar states. These representations also allow the application of other machine learning algorithms, such as dimensionality reduction or unsupervised learning, to improve learning efficiency. Furthermore, factored representations more naturally represent a world state that is usually the result of a combination of different elements.

1.5. Paper organization

Section 2 situates our ideas among related work, and Section 3 presents our notation. Sections 4 and 5 describe our two algorithms, AfD and ADA, and Section 6 discusses how we have combined them with function approximation-based RL algorithms such as fitted Q-learning and LSPI and why this was necessary. Section 7 describes our experimental settings. Section 8 discusses some subtle details of our algorithms. We conclude in Section 9.

2. Related work

AfD and ADA were first described in conference proceedings [12, 10], but their interactions with function approximation-based RL algorithms had not been considered before. Our algorithms are related to work in different subfields in machine learning. The rest of this section details these subfields.

2.1. Selective attention

Previous work has already considered applying concepts related to selective attention to machine learning. The U-tree [40] and G [8] algorithms use an interesting approach to selective attention that first treats the state space as a single state. This state is later iteratively subdivided using a Kolmogorov-Smirnov test that decides which feature encodes the most important distinction. These methods do not need domain information, but it is impractical to find these subdivisions for large or continuous state spaces, especially without an initial good policy to aid exploration. RL has also been used for the problem of selective visual attention [43], but that is a different pursuit than using selective attention as a preprocessing step to enable RL in complex state spaces.

OF-Q [11] is also connected to the concept of selective attention, but it derives this information not from demonstrations but from generic models of the world along with the experience of the agent performing the task. It is suited for situations where demonstrations are not available but the algorithm has access to an object-based representation of the world.

2.2. Previous LfD and RL combinations

A significant branch of LfD focuses on combining demonstrations with conventional RL methods, including using demonstrations to guide exploration [55, 28] or learn a reward function [1]. We use LfD to build state-space abstractions and task decompositions that are then combined with RL algorithms.

Previous work using LfD to induce task decompositions [42, 68] required a dynamic Bayesian network (DBN) model of the environment, while our methods are model free.

An algorithm with a similar goal to ours is the CST algorithm [31]. CST segments demonstrations into a set of goal-oriented sequential skills and organizes them in a skill tree. If a library of candidate abstractions is provided, it can use a different one for each skill and therefore find, like our ADA algorithm, skill-specific decompositions in a lower-dimensional state space where it is easier to learn them. The difference with our ADA algorithm is that our approach finds subtasks that correspond to regions of the state space. CST, however, defines skills sequentially based on expected discounted reward, which is a problem for tasks where there is no unique temporal order in which subtasks/skills must be performed, or where some subtask/skills may be used more than once. Also, CST requires a library of abstractions and a probability model for skill duration, which are not needed by our algorithms.

Another branch of LfD uses demonstration data for learning plans, *e.g.,* learning pre and post conditions; however, this work typically assumes that additional information, such as annotations, is provided to the algorithm. We limit our work to cases where only demonstrations (at most) are available, reflecting our focus on agents that can learn autonomously from non-experts.

2.3. Hierarchical Abstractions

MAXQ [15] is one of the best known hierarchical RL algorithms. In MAXQ, a system designer engineers a task hierarchy, and decomposes the reward to indicate which levels of the hierarchy are potentially responsible for the reward. MAXQ cannot find globally optimal policies, but *hierarchically optimal* ones, this is, the best policies that are consistent with the imposed hierarchy. However, it can speed up learning significantly by dramatically reducing the complexity of the policy that must be learned at each level of the hierarchy. HEXQ [23] extends MAXQ by using a heuristic, based on the relative frequency of change of the features, to try to automatically build a MAXQ hierarchy. Unfortunately, this is only useful in a limited set of domains; for example, it would not help in a continuous domain where the values of all the features change at each step.

Other hierarchical algorithms try to reduce complexity by implementing temporally extended actions or macroactions [5] that represent sequences of actions. The most representative of this class of approaches is the *options* framework [60]. The original work on options assumes that the options are provided to the learning algorithm, including their policy. Several extensions try to learn options automatically using different heuristics such as relative novelty [14], bottleneck states [56, 41], or state clusters [39]. Other work focuses on learning options assuming there is a separate reward signal available for each option to learn [26]. Other authors are critical of the options framework, arguing that options can harm learning due to pathological exploration patterns, even in simple domains, and that the speed-up associated with the options framework is a just a consequence of its implicit experience replay [24].

There are also promising algorithms based on intrinsically motivated learning [64], but so far these are restricted to domains where the agent can control all the variables of the state space.

Alternatives to hierarchies include a flat compositions of local models [62], but these algorithms need highly domain-specific knowledge that must also be specified by a system designer.

In general, hierarchical approaches to RL are useful and conceptually appealing, but their use in autonomous learning systems is difficult due to the necessity of either an engineer manually designing a task structure or the task fitting a specific heuristic that allows an algorithm to recover the task structure automatically.

2.4. State-Space Abstractions

State-space abstractions simplify learning by clumping together or generalizing among similar states. *Regularization*, for example, is closely related with function approximation. Previous work has combined RL with L_1 regularization [29], L_2 regularization [19], and "forward-selection" style feature selection from features generated based on Bellman error analysis [27, 46].

Like our work, regularization tries to limit the complexity of the solution. However, our approach allows a more aggressive state abstraction than regularization, because the policy to be approximated is not the optimal one but a satisficing human-like policy based on a small set of features. This trade-off is beneficial in problems that, because of their complexity, prevent conventional algorithms from converging in any reasonable amount of time.

A different approach to state-space abstraction is to consider a set of predefined state-space representations and find the most useful one [30, 52]. Unfortunately, this approach is impractical in high-dimensional domains if there is no previous knowledge about useful representations or which features are useful together. In addition, these approaches pick a fixed representation for a given task, and they cannot switch to use a different representation dynamically as our algorithm does, unless provided with a library of candidate abstractions that define, as well as the abstraction itself, the region of the state space where it should be used [51]. U-tree [40] can find its own representations, but it requires too many samples to scale well in realistic domains. Instead of using human traces to derive a state space representation, U-tree uses the experience of the agent itself. This requires a great deal of initial exploration, and does not scale well to large domains. An agent following an initial random policy will visit only very rarely the interesting areas of the state space where it can gather the information necessary to build good state space abstractions.

3. Notation

Markov decision Processes are formalized as a tuple

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

with the following elements:

$\mathcal{S}$ is the *state space*, a set of possible states.

$\mathcal{A}$ is the *action space*, a finite set of actions that can be taken by the agent.

$\mathcal{P} = \mathcal{P}^a_{ss'} = Pr\,(s'|s,a)$ is the *transition function*, *i.e.,* the probability of transitioning to state $s' \in \mathcal{S}$ when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

$\mathcal{R} = \mathcal{R}^a_s = r(s,a)$ is the *reward function*, which determines the immediate reward obtained when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. It can be stochastic.

γ is the *discount factor* that encodes an implicit preference between immediate rewards or larger rewards in the future, where $0 < \gamma \leq 1$.

An MDP episode is organized sequentially as a series of steps $t = 0, 1, 2, \ldots$, with step $t = 0$ being the initial step. If at each time step t, with state $s_t \in \mathcal{S}$, the agent takes action $a_t \in \mathcal{A}$, and receives reward $r_t = r(s_t, a_t)$, the total reward for the agent in these settings is the *sum of discounted rewards*

$$\sum_t \gamma^t r_t = \sum_t \gamma^t \mathcal{R}^{a_t}_{s_t}.$$

A *policy* $\pi : S \to A$ determines which action $a \in \mathcal{A}$ to take in each state $s \in \mathcal{S}$. An *optimal policy* π^* is a policy that obtains the highest expected sum of discounted rewards.

Given an MDP M and a policy π, the state-value of a state s with respect to policy π is

$$V^\pi(s) = \mathcal{R}^\pi_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^\pi_{ss'} V^\pi(s').$$

$V^*(s) = V^{\pi^*}(s)$ is the state-value of state s when following π^*. This value is also referred to as the value of state s, or $V(s)$.

Q-value

$$Q^\pi(s,a) = \mathcal{R}^a_s + \gamma \sum_{s' \in S} \mathcal{P}^a_{ss'} Q^\pi(s', \pi(s')), \tag{1}$$

is the discounted reward obtained when taking action a in state s and subsequently following policy π.

$Q(s,a) = Q^*(s,a) = Q^{\pi^*}(s,a)$ are analogous to their state-value function counterparts. Note that the state-values and Q-values are always defined with respect to a given policy that is not necessarily optimal.

We represent $\mathcal{S}$ with a *factored representation* or cross product of n features,

$$S = F_1 \times \cdots \times F_n,$$

where each feature F_i has an independent range of values. Each possible state $s \in \mathcal{S}$ is then defined by the values of these features

$$s = (f_1, \ldots, f_n), f_i \in F_i.$$

Human demonstrations are defined as a set of episodes, each comprising a list of state action pairs

$$H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s_i \in S, a_i \in A.$$

We define

$$\vec{\mathrm{mi}}_E = (I(F_1; A), \ldots, I(F_n; A))$$

as a vector whose elements are the mutual information between each feature of the state space and the action taken by the human teacher, according to the samples of H in the state-space subregion $E \subset S$. We compute each component with

$$I(F_i; A) = \sum_{f_i \in F_i} \sum_{a \in A} p(f_i, a) \log \left(\frac{p(f_i, a)}{p_{f_i}(f_i) p_a(a)} \right), \tag{2}$$

where $p(f_i, a)$ is the joint *probability density function* (pdf) of the feature and the action and; and $p_{f_i}(f_i)$ and $p_a(a)$ the respective marginal pdfs of each random variable, all computed from the samples of H that fall in region E. This equation is used explicitly in our ADA algorithm and implicitly by the feature selection algorithms that AfD uses.

4. AfD: Automatic feature selection

Our first algorithm, state abstraction from demonstration (AfD) learns a policy for an MDP by building an abstract space S^α and using reinforcement learning to find an optimal policy that can be represented in S^α. AfD obtains S^α by selecting a subset of features from the original state space S with which it can predict the action that a human teacher has taken in a set of demonstrations H. Learning in S^α can be significantly more efficient because a linear reduction in the number of features leads to an exponential reduction in the size of the state space.

4.1. Algorithm

AfD, shown in Algorithm 1, is composed of two steps. First, a feature selection algorithm is applied to human demonstrations to choose the subset of features to use. In the second step, which corresponds to the loop in Algorithm 1, the algorithm learns a policy for the MDP M using a modified version of a Monte-Carlo learning algorithm with exploring starts. Instead of learning the Q-values of states in the original state space $Q(s, a), s \in S$, it learns the Q-values of states in the transformed state space $Q(s^\alpha, a)$, where $s^\alpha \in S^\alpha$. We stop when policy performance converges, *i.e.,* when the expected discounted reward remains stable.

We have used two different feature selection algorithms for AfD. The first, which we call C4.5-greedy, is a simple greedy backward selection algorithm. Starting with the full feature set, it iteratively removes features, one at a time, that have little impact on performance. In each iteration, the feature whose

Algorithm 1 Generic AfD algorithm with $\gamma = 1$; the general case is a simple extension.

Require: MDP $M = (S, A, P^a_{ss^\alpha}, R^a_s, \gamma)$, $S = \{F_1 \times \ldots \times F_n\}$, feature selector F, human demonstrations $H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s \in S, a \in A$.

```
chosenFeatures ← F(H)
π ← arbitrary policy
Initialize all Q(s^α, a) to arbitrary values
Intialize all Rewards[(s^α, a)] to []
while π performance has not converged do
  visited ← []
  Start episode with random state-action, then follow π
  for all Step (state s, action a, reward r) do
    for all (s^α, a) in visited do
      EpisodeReward[(s^α, a)] += r
    end for
    s^α ← getFeatureSubset(s, chosenFeatures)
    if (s^α, a) not in visited then
      Add (s^α, a) to visited
      EpisodeReward[(s^α, a)] ← 0
    end if
  end for
  for all (s^α, a) in visited do
    Add EpisodeReward[(s^α, a)] to Rewards[(s^α, a)]
    Q(s^α, a)=average(Rewards[(s^α, a)])
  end for
  Update greedy policy π
end while
```

absence affects accuracy the least is removed. To determine this, we build a set of decision trees using the C4.5 algorithm, each one omitting one of the features, and test the accuracy of each tree using cross-validation on the demonstrations. If the best feature to drop affects accuracy by more than 2% with respect to the current feature set, we stop. We also stop if dropping a feature results in an accuracy loss greater than 10% with respect to the original feature set. These stopping parameters do not appear to be sensitive. In experiments we have tested parameter values of 1% and 5% and found no significant difference in the feature set selected. Note that we use *relative accuracy* for these stopping criterion, *i.e.,* the amount of accuracy gained with respect to the majority classifier.

The second feature selection algorithm, Cfs+voting, uses Correlation-based Feature Subset Selection (Cfs) [22]. Cfs searches for features with high individual predictive ability but low mutual redundancy. The Cfs algorithm is used separately on the demonstrations of each teacher. A feature is chosen if it is included by at least half of the teachers. Both feature selection algorithms use mutual information as the metric to construct decision trees.

Our algorithm can be used with a wide range of feature selection algorithms, there is nothing special about the particular ones we use. The reason we use

these two algorithms is that we were curious whether there would be a difference using all demonstrations as if they were coming from a single teacher, or taking into account that they come from different sources. From our experiments, both approaches work. Typically LfD algorithms cannot merge demonstrations from different teachers because they may have different and even contradicting styles. However, even if the policies of two teachers are different, it seems likely that the state space representation that they effectively use is similar.

4.2. Policy invariance

State abstractions that, like AfD, collapse states with the same optimal action are called policy-invariant. These abstractions can be problematic; even if they can represent a policy, they might not be good enough to learn it when using bootstraping algorithms such as Q-learning or Sarsa.

In the transformed state space, a single state $s^\alpha \in S^\alpha$ represents several states of the original space $s_1, \ldots, s_k \in S$. Assuming the subset of features selected is sufficient for predicting the teacher's action, every original state corresponding to s^α should share the same action. Because only states with the same associated action are collapsed, the teacher's policy can be represented in S^α. The original state value function $V(s)$ may not be representable in S^α because the collapsed states $s_1, \ldots, s_k$ may have different true state values in S, but in S^α they all share the state value of s^α. The same holds for the Q-values $Q(s, a)$.

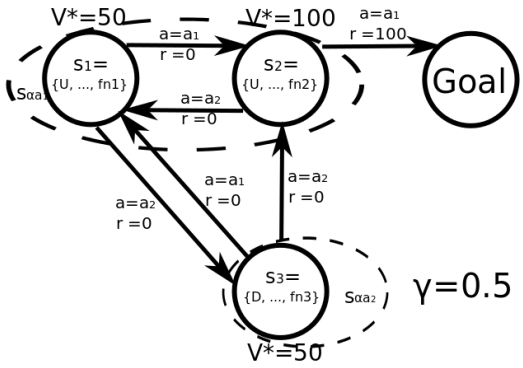


Figure 1: Example of an MDP that loses the Markov property upon state abstraction. Circles represent states in S and arrows transitions when taking action a, providing an immediate reward r. Dashed circles represent states in the transformed state space S^α.

For example, Figure 1 shows a deterministic MDP with states defined by features $\{F_1, \cdots, F_n\}$. F_1 can have values U or D depending whether the state is up or down in the figure, and the rest of the features are policy-irrelevant. The available actions are a_1 and a_2, and the only reward is obtained by taking action a_1 in state s_2. With $\gamma < 1$, the value of s_1 and s_2 is different; however,

because their optimal action is the same (a_1), AfD collapses both into a single state $s^{\alpha}_{a_1} \in S^{\alpha}$.

Conversion to the transformed space also breaks the Markov property. In Figure 1, we can see that in the transformed space, actions a_1 and action a_2 in collapsed state $s^{\alpha}_{a_2}$ are identical. However, we know that the two actions are very different. The value of taking action a_1 from collapsed state $s^{\alpha}_{a_1}$ depends upon how we got to $s^{\alpha}_{a_1}$; specifically, what action was taken in state $s^{\alpha}_{a_2}$ to get there. This means reinforcement learning algorithms that use bootstrapping will not converge to the optimal policy in the transformed state space. Bootstrapping algorithms such as Q-learning compute Q-values considering the immediate reward and the value of the next state. In our example, with the immediate reward and next state being identical for both actions, Q-learning would be unable to learn the best action, even though one leads to higher discounted rewards.

Previous work on policy-invariant abstractions [25] encapsulates the state abstraction as an option that will not be used if it degrades performance. We work around problems of policy-invariant abstractions by using non-bootstrapping methods like Monte Carlo.

4.3. Theoretical properties of AfD

AfD's feature elimination process is benign: in the limit of infinite data, the feature subset it yields will not negatively affect the accuracy of the learner. To see this, consider that AfD only removes features it judges will not reduce the accuracy of the learner. These judgements are based on some held out portion of the data.[1] In the limit of infinite data, these judgements will be accurate, and the elimination process will never remove a feature if it negatively impacts accuracy. Thus, the final feature subset cannot negatively impact accuracy.

Similarly, AfD's policy solver is sound. As shown in Section 4.2, in a policy-invariant abstraction like ours, bootstrapping algorithms like Q-learning may not find the best stationary policy; however, the abstracted space creates a POMDP, where Monte-Carlo control is sound [54]. Thus, applying the policy solver will not lower policy performance.

From these properties, we can show that, in the limit, the worst-case policy performance of AfD is the same as direct LfD. In the limit, LfD will yield a policy with the best prediction accuracy.

5. ADA: Automatic task decomposition

Often, multipurpose agents have a high number of input signals, but only a subset of them are relevant for any specific task. Unfortunately, AfD does not help if all state features are relevant at some point in a task.

The key insight that we want to incorporate is that tasks where all features are relevant at some point can often be decomposed into subtasks, such that

[1] We actually use cross-validation for higher sample efficiency, but the same principle holds.

for any given subtask, there is an abstraction in which a good policy can be expressed. For example, when we drive a car, we focus our attention almost completely on the car keys at the start and end of a drive, but completely ignore them for the rest of the trip.

Our goal is to infer this shifting selective attention, *i.e.,* the particular task decomposition and state abstraction that the human teacher uses during her demonstration. We define a subtask as a region of the state space where only a subset of the original features is relevant, with this subset differing from those of other subtasks. Thus, we look for a decomposition that maximizes our ability to apply AfD in each part.

Our algorithm, *automatic task decomposition and state abstraction from demonstration* (shortened to automatic decomposition and abstraction, ADA), uses human demonstrations to both decompose a task into its subtasks and find independent state abstractions for each subtask. ADA can build more powerful abstractions than AfD, finding compact state-space representations for more complex tasks in which all state features are relevant during some point of the task.

To determine which features are relevant to a particular subtask, we measure the mutual information between each state feature and the action taken by a human teacher in a set of demonstrations. Once the state space is decomposed into different subtasks, the agent can learn and represent a compact policy by focusing only on the features that are relevant at each part of the task.

5.1. Algorithm overview

Given an MDP M and a set of human demonstrations H for a task to be learned, ADA finds a policy in three conceptual steps and an optional fourth step:

1. **Problem decomposition.** Using H, partition the state space S into different subtasks $T = \{t_1, t_2, \ldots\}$, $\cup T = S$, $t_i \cap t_j = \emptyset$ if $i \neq j$.
2. **Subtask state abstraction.** Using H, determine, for each subtask $t_i \in T$, the relevant features $\hat{F}_i = \{F_{i_1}, F_{i_2}, \ldots\}, F_{i_j} \in F$, and build a projection from the original state space S to the abstract state space $\phi(s) = \{i, f_{i_1,s}, f_{i_2,s}, \ldots\} \in S^\alpha, s \in t_i$. $\phi(s)$, therefore, includes i, which is the identifier of the subtask t_i, and the values of the features that are active in this subtask.
3. **Policy construction.** Build a stochastic policy $\pi(s^\alpha)$, projecting the samples from H into S^α.
4. **Policy improvement (optional).** Use reinforcement learning to improve the policy found in the previous step. We refer to our algorithm as ADA+RL when it includes this step.

We list the first two steps as if they were sequential; however, they are concurrent. The decomposition of the state space depends on the quality of the abstractions that can be found on different subspaces. We describe them as separate steps just to make the algorithm description more straightforward.

5.2. Problem decomposition

Definition 1. *A set of* ***subtasks*** $T = \{t_1, t_2, \ldots\}$ *of an MDP* M *is a set of regions of the state space* S *such that:*

- *The set of all subtasks* T *forms a partition of the original state space* S*, i.e.,* $\cup T = S$ *and* $t_i \cap t_j = \emptyset$ *if* $i \neq j$*.*
- *A subtask* t_i *is identified by having a local satisficing policy* π_i *that depends only on a subset of the available features. This subset is different from spatially neighboring subtasks.*
- *The global policy* $\pi(s^\alpha)$*, combination of the policies of each subtask, is also satisficing.*

While this definition is not the typical one for subtasks in a sequential decision problem, it turns out to be a useful one, particularly if we focus on human-like activities. For example, cooking an elaborate recipe requires multiple steps, and each of these steps will involve different ingredients and utensils. It is possible that two conceptually different subtasks may depend on the same features, but in our framework, and arguably in general, the computational benefits of separating them are not significant.

With ADA, we can identify these subtasks given a set of human demonstrations H with two requirements:

- There must be a sufficient number of samples $\min_{ss}$ from each subtask in the set of demonstrations H.
- The class of possible boundaries between subtasks $B = \{b_1, b_2, , \ldots\}, b_i \subset S$ must be defined. Each boundary is a threshold that divides the state space in two subspaces, b_i and $S - b_i$. ADA will be able to find subtasks for subspaces that can be expressed as combinations of these boundaries.

The necessary number of samples is determined in the first step of the ADA algorithm. This minimum sample size is needed due to the metric we use to infer feature relevance. Mutual information is sensitive to the *limited sampling bias*, and will be overestimated if the number of samples considered is too low.

The decomposition is described in Algorithm 2. At each iteration of the while loop, we consider a subspace E, with $E = S$ in the first iteration. We then consider all valid boundaries. If there are none, then E itself is a subtask. If there are valid boundaries, we score them and choose the one with the highest score. We then split E according to the boundary and add the two new subspaces to the list of state subspaces to be evaluated, so they will be further decomposed if necessary.

The boundaries B can have any form that is useful for the domain. In our experiments, we consider thresholds on features, *i.e.,* axis-aligned surfaces. Using these boundaries, Algorithm 2 is just building a decision tree with a special split scoring function and stopping criteria.

The following subsections discuss the details of the split scoring function, the discriminator of valid boundaries, and the estimator of the minimum number of samples. These contain the most interesting insights about ADA.

Algorithm 2 ADA problem decomposition.

Require: MDP $M = (S, A, P^a_{ss'}, R^a_s, \gamma)$, $S = \{F_1 \times \ldots \times F_n\}$, human demonstrations $H = \{\{(s_1, a_1), (s_2, a_2), \ldots\}, \ldots\}, s \in S, a \in A$, boundaries $B = \{b_1, b_2, \ldots\}, b_i \in S$, ϵ.

```
min_ss ← min_sample_size(H, ε)
T ← {}
𝕊 ← {S}
while 𝕊 ≠ ∅ do
    {pop removes the element from 𝕊}
    E ← 𝕊.pop()
    B_E ← {b ∈ B, valid_split(b, E, min_ss)}
    if B_E = ∅ then
        T.push(E)
    else
        b_best ← arg max_{b∈B_E}(boundary_score(b, E)))
        𝕊.push(b_best ∩ E)
        𝕊.push((S − b_best) ∩ E)
    end if
end while
Return T
```

5.2.1. Boundary discriminator

Given a subspace $E \subset S$, m_{ss} and H, we consider a boundary $b \subset S$ to be valid if it meets three conditions:

1. There are enough samples in the set of human demonstrations H to ensure we can measure mutual information with accuracy on both sides of the boundary, *i.e.,*

$$|\{\{s, a\} \in H, s \in b \cap E\}| > \min_{ss},$$
$$|\{\{s, a\} \in H, s \in (S - b) \cap E\}| > \min_{ss}.$$

2. At least on one side of the boundary, either $b \cap E$ or $(S - b) \cap E$, it is possible to find a state abstraction, *i.e.,* some features are policy-invariant and can be ignored. We detail how we find these features in Section 5.3.
3. The state abstractions at each side of the boundary are not the same.

This boundary discriminator works as the stopping criteria of the algorithm. When there are no more valid boundaries to be found, the decomposition step finishes.

5.2.2. Boundary scoring

The boundary scoring function determines the quality of b as a boundary between different subtasks within a region $E \in S$:

$$\text{boundary\_score}(b, E) = \left|\left| \frac{\vec{\text{mi}}_{b \cap E}}{\|\vec{\text{mi}}_{b \cap E}\|} - \frac{\vec{\text{mi}}_{(S-b) \cap E}}{\left|\left|\vec{\text{mi}}_{(S-b) \cap E}\right|\right|} \right|\right|. \tag{3}$$

The score is thus the Euclidean distance between the normalized mutual information vectors on both sides of the boundary.

We are therefore measuring the difference between the relative importance of each feature on both sides of the boundary. Since want to find subtasks that rely on different features, we choose the boundary that maximizes this difference (see Algorithm 2).

5.2.3. Minimum samples

Due to the *limited sampling bias*, mutual information is overestimated if it is measured with an insufficient number of samples. The minimum number of samples in ADA, given a set of demonstrations H and parameter $\epsilon \approx 0.1$, is

$$m_{ss} = \arg\min_n \text{average} \left(\frac{\vec{\text{mi}}_S - \vec{\text{mi}}_{S,n}}{\vec{\text{mi}}_S} \right) < \epsilon, \tag{4}$$

where $\text{mi}_{S,n}$ is the mutual information vector on the original state space S, taking only a subset of n randomly chosen samples from all the samples in H. The subtraction and division are elementwise and the average function takes the average of the values of the resulting vector. Because of the variability of mutual information, it is necessary to evaluate (4) several times for each possible n, each time with a different and independently chosen set of samples. Because of the limited sampling bias, the difference between $\vec{\text{mi}}_S$ and $\vec{\text{mi}}_{S,n}$ will grow as n decreases, and a binary search can be used to find m_{ss} efficiently.

5.3. Subtask state space abstraction

Given a region of the state space $E \subset S$, we consider $\vec{\text{mi}}_E$ in order to estimate policy-irrelevant features. Even if a feature is completely irrelevant for the policy in a region of the state space, its mutual information with the action will not be zero due to the limited sampling bias. Therefore, ADA groups the values of $\vec{\text{mi}}_E$ into two clusters separated by the largest gap among the sorted values of the vector. If the value difference between any two features in different clusters is larger than the distance within a cluster, we found a good abstraction that discards the features in the lower value cluster.

Note that this step occurs concurrently with the previous one, since the decomposition step needs to know in which regions of the state space there are good abstractions. Once these steps complete, we can build the projection function from the original state space S to the abstract state space $\phi(s) = \{i, f_{i_1}, f_{i_2}, \ldots\} \in S^\alpha, s \in t_i$.

5.4. Policy construction

Once the task decomposition and state abstraction are completed, and we have the projection function $\phi(s)$, we use the demonstrations H to build a stochastic policy that satisfies

$$P(\pi(s^\alpha) = a_i) = \frac{\left|\{\{s, a_i\} \in H, \phi(s) = \hat{s^\alpha}\}\right|}{\left|\{\{s, a\} \in H, \phi(s) = \hat{s^\alpha}, a \in A\}\right|}, \tag{5}$$

where $\hat{s^\alpha}$ equals s^α if $|\{\{s, a\} \in H, \phi(s) = s^\alpha, a \in A\}\}| > 0$. Otherwise, $\hat{s^\alpha}$ equals the nearest neighbor of s^α for which the denominator in (5) is not zero.

To compute the policy, we project the state of each sample of H into the abstracted space and make a normalized histogram of each action. This concludes the basic ADA algorithm.

5.5. Policy improvement

ADA+RL adds another step, policy improvement, in which we use reinforcement learning techniques to find the optimal policy that can be represented in the abstract state space S^α. Unlike most previous LfD techniques, ADA was designed so that the resulting policy can be easily improved with the agent own experience through reinforcement learning. In this way, we can potentially obtain a better policy than that of the human teacher.

Given the kind of abstraction that ADA performs, bootstrapping methods such as Sarsa or Q-learning are not guaranteed to converge in the abstract state space [37]. As such, we can use either Monte-Carlo methods or direct policy search.

6. Scalability using Function Approximation

AfD and ADA are able to derive useful state representations using a small set of demonstrations, yielding exponential speed-ups in learning. However, the performance of both techniques was first measured only with tabular state-space representations. Function approximation-based RL algorithms perform implicit feature selection that may overlap with abstraction from demonstration, so it was initially unclear whether the benefits of our algorithms would still be significant when combined with function approximation.

Function approximation (FA) makes it possible to successfully scale up reinforcement learning for use in complex real-world domains. However, FA often requires careful and time consuming feature engineering: depending on the chosen state representation, an FA-based RL algorithm can be successful or useless in a given domain.

Further experiments have shown that ADA and AfD combined with function approximation offer significant advantages over FA alone. In particular:

- Abstraction from demonstration can be used with function approximation-based RL algorithms to obtain near-optimal policies.
- Abstraction from demonstration enhances the performance and applicability of RL algorithms with function approximation in the following ways:
  - Increasing scalability, *i.e.,* making these algorithms able to deal with domains that were previously out of their scope due to high-dimensional state spaces.
  - Speeding up learning. The speed-up grows with the scale of the problem and is large enough to justify the cost of acquiring demonstrations.

– Extending the hypothesis space of function approximation-based RL algorithms, broadening the class of problems that such algorithms can solve.
– Making the algorithms more automatic. Abstraction from demonstration can make manual feature engineering less necessary by finding a good state-space representation. This allows agents to learn new tasks without intervention from an engineer.

Section 7 justifies these claims experimentally, measuring the effect of abstraction from demonstration on two popular function approximation-based RL algorithms. In our experiments, we use two of the most widely used FA-based RL algorithms: fitted Q-learning and LSPI. The rest of this section briefly introduces them and details the specifics of our implementations. Later, we comment on the limitations of linear models, which are used in our FA-based algorithms. Finally we overview other FA-based algorithms and explain why they are less suitable for our purposes.

6.1. Fitted Q-Learning

Fitted Q-learning [61] is a simple function approximation-based RL algorithm that is widely used and works well in practice. We chose linear regression, because it is popular and easy to tune. Our implementation was online; instead of running the regressor after having collected all available information from the domain, we made small updates to the linear approximation after each interaction with the environment.

Online implementations are convenient and often the only option in large environments, since it may not be feasible to store all the transitions that need to be observed to obtain a good model. Additionally, by updating the model more frequently, the algorithm explores the interesting parts of the state space faster, and as the policy improves, the approximation focuses on accurately representing the part of the state space that is most often visited by the optimal policy.

6.2. LSPI

LSPI [32] is a stable algorithm that is more sample-efficient than fitted Q-learning, but it has a higher computational cost. LSPI is an actor-critic method that uses a linear approximation to the value function.

We use three variants of LSPI: classic LSPI, IncLSPI, and LARS-TD. The original version of LSPI needs to store all the samples, which is impractical in large domains, and also needs a matrix inversion step of time complexity $O(n^3)$, where n is the number of features. This makes LSPI impractical for our domains when learning in the original state space, so in our experiments we use an incremental implementation with lower computational complexity that we call IncLSPI.

Our IncLSPI is similar to iLSTD [21], but in our implementation, we run a fixed behavioral policy and periodically update the policy [7] and reset the

μ and A matrices to satisfy the assumptions made by iLSTD. Like iLSTD, IncLSPI is particularly efficient in sparse domains where the number of non-zero features k is small with respect to the total number of features n; however, it has complexity $O(n^2)$ in general.

LARS-TD [29] is an L1-regularized version of LSPI with complexity $O(mnk^3)$ where m is the number of samples and n and k are defined above. This algorithm selects features that are most likely to affect the function being approximated in order to make LSPI robust to stochastic features as well as more feasible in large domains. This set of features is actively modified by adding and removing features until a chosen regularization criterion is satisfied. Our implementation of LARS-TD was modified to learn Q-functions.

6.3. Limitations of Linear Models

Linear models are usually easier to tune, analyze, and debug than others, so they are a frequent choice for complex domains. However, a poor choice of features can make a linear representation useless. Imagine a simple grid domain with the coordinates of the agent and the coordinates of the goal position as features, and actions to move north, south, east and west. A linear model cannot represent a policy that leads the agent from an arbitrary cell to an arbitrary goal cell, because whether one action is preferred over another does not depend on any of the features of the state space, but on the relation between different features (the coordinates of the agent and the coordinates of the goal).

However, a linear model can use any number of nonlinear features based on any combination of the *native* features of the original state space. Therefore, any value function can be approximated if the right features are used. In the extreme case, we can imagine a single feature that outputs the true value function, and the linear approximation would be perfect by assigning a weight of one to that feature and zero to the rest. Unfortunately, finding the right features for complex domains can be a daunting task, often involving time-consuming trial and error. Furthermore, manual feature engineering is not an option for autonomous learning agents. For this reason, our abstraction from demonstration algorithms try to subdivide the function needing approximation into smaller problems that will be more likely properly approximated by a linear model on the native feature space of the domain.

6.4. Other Algorithms

There are many other approximation algorithms [61] that we do not use in our work because their requirements or slow convergence limit their usefulness in general domains and make them unsuitable for our purposes. Nonlinear approximation algorithms [50, 18], for example, offer a larger hypothesis space, but they often do not perform well in practice without a time-consuming manual parameter tuning and feature space optimization. Some algorithms try to automate this process [65], but are not yet practical for large domains. Policy gradient approaches usually need an initial policy with decent performance, and are generally slower than value function approximation [6, 67, 59]. Additionally,

they require a parametric formulation of the policy that is differentiable, which is often challenging to design. Residual gradient algorithms are also slow [35, 48].

Recent TD methods [58] offer attractive theoretical properties, but so far these methods are limited to prediction, *i.e.,* determining the value function V^{π} of a given policy π, as opposed to finding the optimal policy π^{*}. Greedy-GQ [38] solves this problem and can find an approximation of the optimal value function as long as the behavior policy is fixed. Unfortunately, in complex domains, a random behavior policy does not visit the interesting parts of the state space often enough for effective learning, and the algorithm is no longer stable with a varying behavior policy.

There are also efforts to adapt the KWIK[2] learning model to approximation algorithms [36]. We believe this is a promising approach, but the current implementation assumes access to a perfect oracle that predicts Q-values with no error, which is not available in most domains.

In order to provide a complete evaluation of our methods, we have implemented the Tree-based batch mode Q-Iteration [18] algorithm. It is a non-linear value function approximation method that performs automatic state space abstraction. The algorithm iteratively trains a regression decision tree using a batch of training samples which are in the form of state, action pairs as input and Q-values (computed from the previous iterate) as output. The predicted Q-values are iteratively bootstrapped until a predefined stopping condition is satisfied. The algorithm is guaranteed to converge under standard stochastic approximation conditions.

7. Experimental Evaluation

This section details the experiments that we carried out to measure the performance of our algorithms. We first introduce our experimental domains and then analyze the performance, using a tabular representation, of AfD and ADA. Finally, we analyze the speed-up obtained when combining our abstraction from demonstration algorithms with previous function approximation-based methods.

7.1. Domains

We implemented several videogame domains to test our algorithms. Some of these domains have a clear subtask structure, while others do not, so we can highlight the differences between AfD and ADA.

7.1.1. Pong

We used this simple domain for an early test of the soundness of AfD. Pong is a form of tennis where two paddles move to keep a ball in play. Our agent uses one paddle while the other paddle follows a fixed policy to move in the

[2]KWIK stands for *knows what it knows*; it is a family of PAC (probably approximately correct) algorithms for solving MDPs that includes R-max.

Figure 2: Screen capture of the Frogger domain.

direction that best matches the ball's Y position when the ball is approaching, moving randomly otherwise. There are five features: `paddle-Y`, `ball-X`, `ball-Y`, `ball-angle`, and `opponent-Y`. `Y` coordinates and `ball-angle` have 24 possible values while `ball-X` has 18. There are two possible actions: `Up` or `Down`. The reward is 0, except when successfully returning a ball, yielding +10. The game terminates when a player loses or after 400 steps, implying a maximum policy return of 60.

7.1.2. Frogger

Our non-hierarchical high-dimensional domain is a version of the classic Frogger game (Figure 2). In the game, the player must lead the frog from the lower part of the screen to the top, without being run over by a car or falling in the water.

At each time step, cars advance one position in their direction of movement, and the player can leave the frog in its current position or move it up, down, left or right. The positions and directions of the cars are randomly chosen for each game, and the frog can be initially placed in any position in the lower row. The game was played at 7 steps per second, chosen empirically as a challenging enough speed for the game to be fun.

The screen is divided into a grid, and the state features are the contents of each cell relative to the current position of the frog. For example, the feature `3u2l` is the cell three rows up and two columns to the left of the current frog position, and the feature `X1r` the cell just to the right of the frog. The possible values are `empty`, if the cell falls out of the screen; `good`, if the cell is safe; and `water`, `carR`, and `carL` for cells containing water, or a car moving to the right or left. There is a positive reward $r = 1000$ for reaching a goal cell and a negative reward $r = -100$ for failure. The discount factor is $\gamma = 0.99$.

There are 10x9 cells, so 306 features are needed to include the screen in the state representation. With 5 possible actions and 5 possible values per cell, a table-based Q-learning algorithm might need to store up to $5^{307} \approx 10^{215}$ Q-values.

As a comparison, the estimated number of atoms in the observable universe is just 10^{80}. This means the problem is intractable in the raw state space, and approaches that use human demonstrations as policy priors [28] may not be directly applicable.

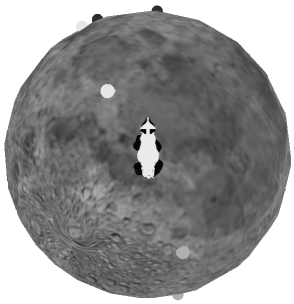


Figure 3: RainbowPanda domain. The agent must pick all balls, aiming at those that match the agent color at each moment. With 12 continuous state features, conventional RL does not converge in a reasonable amount of time. ADA finds a satisficing policy quickly by decomposing the problem into simpler subtasks.

7.1.3. Panda domains

To measure the effect on FA of the subtask decomposition of ADA, we used two different domains in which the player controls an agent represented as a panda bear walking on a spherical surface. The agent has to collect balls of different colors, as shown in Figure 3. At each time step, the panda can turn right/left or move forward/backward. The agent moves slower backward than forward. The position of the balls is chosen randomly at the start of each episode.

In the PandaSequential domain, there are several balls of different colors to be picked up in a specific fixed order. In the RainbowPanda domain, there are six balls, two of each possible color, and at any given time the panda is tinted by the color of the balls it must pick up. The active color always changes when there are no balls of the active color left and it may also change, with a small probability, at any time step. The state features are the angle and distance of each ball relative to the agent and, for RainbowPanda, the active color. The agent receives a reward $r = 1000$ for each ball collected and there is a discount factor $\gamma = 0.99$.

In RainbowPanda, there are 12 continuous variables (relative angle and distance of each ball) and one discrete variable, the color the agent is currently allowed to pick up. In this 13-dimensional state space, raw state-space tabular RL takes an unreasonable amount of time to converge. Further, the complexity of the policy grows exponentially with the number of balls. We will show that with ADA we can automatically decompose this problem into a set of subtasks, one per color, with each one needing to pay attention only to the closest ball of the target color. These two-dimensional policies are easy to obtain, and the complexity of the global policy grows linearly with the number of balls.

Player	Average Return	Episodes
Human	56.5	–
Sarsa	60.0	2554
Direct LfD	15.7	–
AfD - C4.5-greedy	60.0	59

Table 1: Performance on Pong.

7.2. Results on non-hierarchical domains

7.2.1. Setup

We compare the performance of AfD with that of using demonstrations alone and that of using RL alone (using Sarsa(λ)). For direct LfD, we use a C4.5 [49] decision tree classifier to learn a direct mapping. The classifier uses all the features of the state space as input attributes and the actions of the particular domain as labels.

Pong serves as a proving ground for demonstrating the correctness of AfD, using just 23 episodes as training data. The more complex Frogger gauges generalization and real-world applicability. For Frogger, we recruited non-expert human subjects—six males and eight females—to provide demonstrations. Each had three minutes to familiarize themselves with the controls of the game; they were then asked to provide demonstrations by playing the game for ten minutes.

7.2.2. Results on Frogger and Pong

Table 1 compares the performance of various learned policies in Pong. Human results are provided for reference. As we can see from the results, AfD is able to learn an optimal policy, outperforming direct LfD; moreover, while RL also learns an optimal policy, AfD is significantly faster. AfD's speed-up corresponds directly to the smaller abstract state space. In particular, AfD ignored the feature `opponent-Y`, which did not influence the teacher's actions.

With Frogger, we focus on how well AfD works with non-expert humans. The 14 human teachers obtained a success rate (percentage of times they lead the frog to the goal) between 31% and 55%. For learning in AfD, we filtered the demonstrations to keep only successful games and to remove redundant samples caused by the player not pressing keys while thinking or taking a small break. Each user provided on average 33.1 demonstrations ($\sigma = 9.3$), or 1230.1 samples ($\sigma = 273.6$). Note that a demonstration is a complete episode of the game and a sample is a single state-action pair.

We compared the algorithms using two sets of the demonstrations: the aggregated samples of all users, 464 demonstrations (17221 samples) in total, and the 24 demonstrations (1252 samples) from the best player.

For feature selection in AfD, we used the Cfs+voting and C4.5-greedy algorithms (Section 4). Before using C4.5-greedy in this domain, we reduced the number of variables using the Cfs algorithm, because iterative removal on 306 variables was too time consuming and our tests showed that the chosen features were the same as when using only C4.5-greedy. Cfs+voting was not used on

Player	Success rate
Human	31%-55%
Direct LfD	43.9%
AfD - Cfs+voting	97.0%
AfD - C4.5-greedy	97.4%

Table 2: Frogger domain, all demonstrations (17221 samples).

Player	Success rate
Human	55%
Direct LfD	17.1%
AfD - C4.5-greedy	88.3%

Table 3: Frogger domain, best player demonstrations (1252 samples).

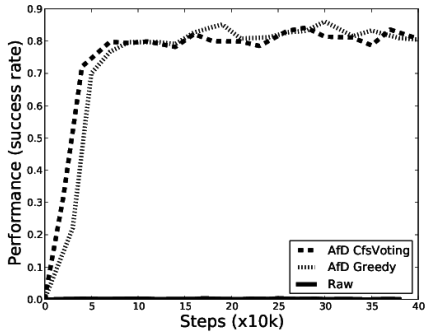
the second data set, as it is designed to work with a set of demonstrations from different teachers.

Table 2 shows that, using all demonstrations, AfD achieved a significantly higher success rate than direct LfD. AfD achieved close to 100% success regardless of the feature selection algorithm used, while direct LfD did not reach 44%. Note also that the AfD policies performed better than the best human.

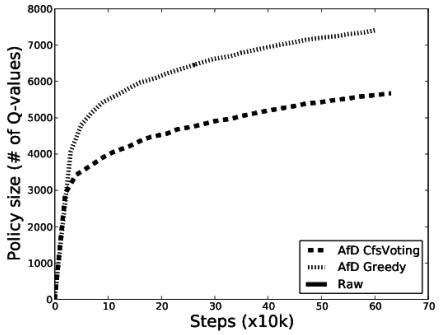
Table 3 shows results using only demonstrations from the best player. Even with only 7% the number of demonstrations of the previous experiment, AfD performance decreased only slightly. By contrast, direct LfD is much worse. Again, AfD performed better than the teacher. Comparing both tables, we can appreciate that AfD was much more sample efficient than LfD, performing better with 20 times fewer demonstrations.

By inspection, we see that AfD identified "key features" of the domain (the ones that would be included in a hand-crafted set) using either of the two datasets. The five key features for this domain are the cells at both sides of the frog and the three closest cells in the row immediately above. Of the original 306 features, the algorithms selected 9 to 12, and the five key features were included in these sets. Only when using just the best player demonstrations AfD did fail to include one of these key features, to which we attribute the slight decrease in performance. The other features selected were also useful: cells in the three rows contiguous to the frog and others that allowed the frog to line up with a goal cell when in a safe row.

We also compared the performance of AfD to that of applying RL directly in the raw feature space. Figure 4(a) shows that working in the large raw state space did not achieve significant learning: states are rarely visited for a second time, which makes learning slow. Additionally, memory consumption grows linearly as a function of the number of steps taken. In our experiments, the raw method had consumed 19GB of memory before it had to be killed. At that point, it had taken almost 1.7 million steps, but the success rate was still below 0.2%. By contrast, AfD was performing better within a thousand



(a) Performance (% of games won) of AfD and learning in the raw state space.



(b) Policy size vs. number of steps of AfD and learning in the raw state space. The raw line is not visible because it matches the vertical axis.

Figure 4: Results on the Frogger domain

steps (the success rate is lower than in Table 2 because exploration was not disabled). This dramatic difference reflects the reduction in state space, and the corresponding exponential reduction in computational cost. Figure 4(b) illustrates the difference in state-space size by showing the growth of policy sizes (the raw method is aligned with the y-axis). Policy sizes for the AfD methods begin to level off immediately.

Learning in the raw domain, the size of the representation of the policy grows linearly with the number of steps, since at each time step the algorithm visits a previously unseen state for which a Q-value must be remembered. It cannot be seen in Figure 4(b) because the line for learning without demonstrations is aligned with the y-axis, but after 1.7 million steps, 92% of the steps were a previously unseen state-action.

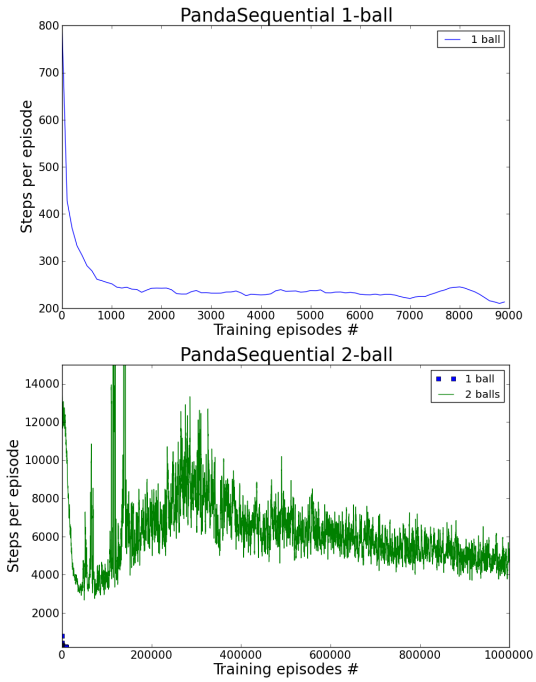


Figure 5: Results using the Sarsa(λ) algorithm on a simplified version of the domain with only one or two balls. One extra ball decreases performance by two orders of magnitude, even if the algorithm trains 2000 times longer.

7.3. Results on hierarchical domains

Using the hierarchical Panda domains described in Section 7.1.3, we compare ADA with reinforcement learning using Sarsa(λ), learning from demonstration using a C4.5 [49] decision tree, and abstraction from demonstration. We captured a set of human demonstrations; 400 episodes for the PandaSequential domain (about 2 hours of gameplay) and 200 and 300 episodes, from different individuals, for the RainbowPanda domain (1.5h and 2.25h, respectively). We first discuss RL and LfD, then our algorithm, and finally we compare the abstractions that our ADA and AfD find.

7.3.1. Reinforcement learning using Sarsa on Panda domains

For Sarsa, we discretized the continuous values into 64 bins. The results were poor in these domains because of the high dimensionality of the problems. The simpler domain, PandaSequential, still has 64^6 possible states, for a total of about 343 billion Q-values. To ensure our implementation of the algorithm was correct and find its limits, we tested it with simplified versions of the game,

Table 4: LfD results over 10 thousand episodes, using a C4.5 decision tree. Average steps computed only over successful episodes.

Domain/Player	# Demonstrations	Success rate	Avg. steps
Sequential	100	0.28%	241.21
	200	0.44%	227.79
	400	0.82%	242.83
Rainbow/A	100	8.1%	1543.84
	200	2.76%	1650.37
Rainbow/B	100	0.27%	1518.70
	200	0.27%	1994.92
	300	0.73%	1901.51

Table 5: Comparison of average human performance (on the demonstrations used), ADA, and ADA+RL, measured in number of steps to task completion (algorithm performance averaged over 10 thousand episodes.)

Domain/Player	# Demonstrations	Human	ADA	ADA+RL
Sequential	100	322.67	360.75	267.11
	200	318.71	360.11	
	400	311.30	335.92	
Rainbow/A	100	525.33	-	-
	200	504.71	626.27	466.59
Rainbow/B	100	540.03	596.46	
	200	536.43	593.45	
	300	533.56	583.34	

with only one and two balls. The results are shown in Figure 5. We can see that Sarsa performed reasonably well for the case of only one ball (4096 states), but no longer did so for the two-ball game (16.8 million states), even though we let the algorithm run for 8 days on a modern computer, this is, 2000 times longer than it took for the one-ball policy to converge. The policy performance keeps improving, but at an extremely slow rate. Therefore, Sarsa is not an effective option for these domains.

7.3.2. Learning from demonstration using C4.5 on Panda domains

To compare with the performance of LfD techniques, we trained a C4.5 [49] decision tree with the demonstrations captured from human players, in a purely supervised learning fashion. Table 4 shows that LfD also performed poorly. The best result we obtained, using all available demonstrations, was a policy that would reach the goal state in less than 3% of the episodes.[3]

[3]It should not be a surprise that sometimes, with more samples, the number of average steps on successful episodes increases. This is due to the policy being able to deal with more difficult episodes (remember that the initial placement of the balls is random) that require more steps to complete.

7.3.3. Automatic decomposition and abstraction from demonstration on Panda domains

To use ADA in our domains, we discretized the continuous values into 64 bins (same as for RL/Sarsa), used $\epsilon = 0.1$, and considered as candidate boundaries every possible threshold on every feature of the domain. ADA was much more effective than the other methods on both domains, and led to near-optimal polices that succeeded on every episode. Table 5 shows that we obtained policies comparable to those of the human teacher. Even though the average number of steps is slightly higher than for the LfD policy in the PandaSequential domain, this is averaged over all episodes, while the number for LfD is only averaged over the small percentage of episodes that LfD is able to resolve.

The success of ADA, compared with LfD and Sarsa, is due to its finding the right decomposition of the domains. For both domains, the algorithm builds an abstraction that focuses only on the angle between the agent and the next ball to be picked up. Which ball is the target depends on what balls are present for the PandaSequential domain, and on the current color the agent is targeting for the RainbowPanda domain. The algorithm was able to identify the right boundary on each domain. It was a surprise that only the angle, and not the distance to the ball, was necessary, but it is easy to see that a satisficing policy can be found using only the angle: rotate until the ball is in front of the agent and then go forward. In fact, this was what the human players were doing, except in the rare case where the ball of interest was right behind the agent; since the agent moves faster forward than backward, it was usually not worth moving backwards.

Only one case in Table 5 did not produce the abstraction described above. Rainbow/B-100 episodes did not find any abstraction. This was due to m_{ss} being higher than a third of the total number of samples; therefore, it could not find any of the three subtasks, one per color and roughly of the same size, that were found in the other cases. We tested a lower value for ϵ and in that case the usual abstraction was found.

The same table shows results for $ADA + RL$, applying policy search on top of the policy found by ADA. The abstraction built by ADA may prevent bootstrapping algorithms such as Sarsa or Q-learning from converging, but with only 192 states in the abstraction and a good starting policy, we can use direct policy search methods. We could obtain good results by just iteratively changing the policy of each state and evaluating the effect on performance using rollouts.

Using this additional policy improvement step, we can find policies that are better than those demonstrated by the human teachers. The policies found were better than those demonstrated in three ways. First, the preferred action for states that were rarely visited was sometimes incorrect in the ADA policy, because there were not enough samples in the demonstrations. ADA+RL could find the best action for these uncommon states. Second, human players would make the agent turn to face the target ball and then move forward when the relative angle to the ball was less than 15 degrees. ADA+RL found it was more efficient to turn until the angle to the ball was less than three degrees, and

only then move forward. Third, ADA policies assign some probability to each action depending how often it is taken in the demonstrations for a particular state. ADA+RL can identify which actions were not appropriate for the state and never execute them, even if they appear in the demonstrations, perhaps because of distractions or errors by the teacher. In short, the policy found by ADA+RL was a *more precise* and *less noisy* version of the policy derived directly from the demonstrations.

7.3.4. Abstraction from demonstration on Panda domains

Finally, we tried AfD in the domains, using the abstraction algorithm described in Section 5.3 for the whole state space. In the PandaSequential domain, AfD would identify the position of the first ball as the only useful feature. This abstraction leads to a policy that can find the first ball quickly, but can only perform a random walk to find the other two balls. The large difference in mutual information between each ball position and the action is due to the fact that while the first ball is present, its position is significant for the policy; however, the second ball is significant for the policy only half of the time it is present, and the third ball only a third of the time it is present.

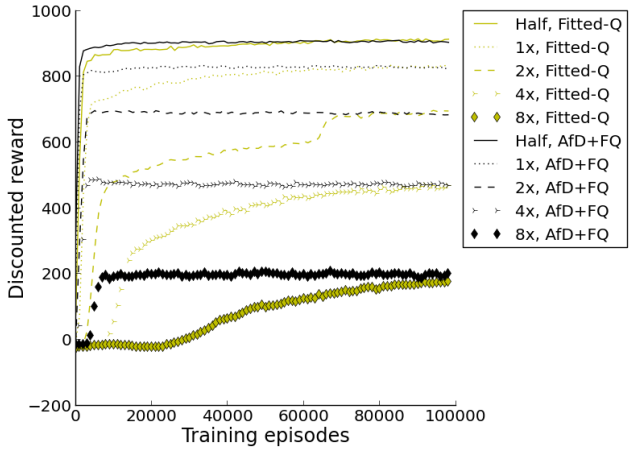
Regarding AfD for the RainbowPanda domain, because the active color at each moment is chosen at random, the mutual information measures between each ball's relative position and the action are similar. In this case, AfD is able to identify the true relevant features, *i.e.,* the relative position to the closest ball of each color. Due to the nature of AfD abstraction, we could not use bootstrapping algorithms such as Sarsa, and $64^3 = 262144$ states are too many for our naive policy search, so we tried to obtain a policy using Monte-Carlo methods. Unfortunately, these are known to be much slower to converge than Sarsa. Even after experimenting with various exploration parameters, we could not reach a policy better than a random walk. We can thus conclude that for complex domains that can be decomposed in different subtasks, ADA can find policies better than those demonstrated by humans, while LfD, RL, and AfD cannot find policies significantly better than a random walk.

7.4. Function approximation

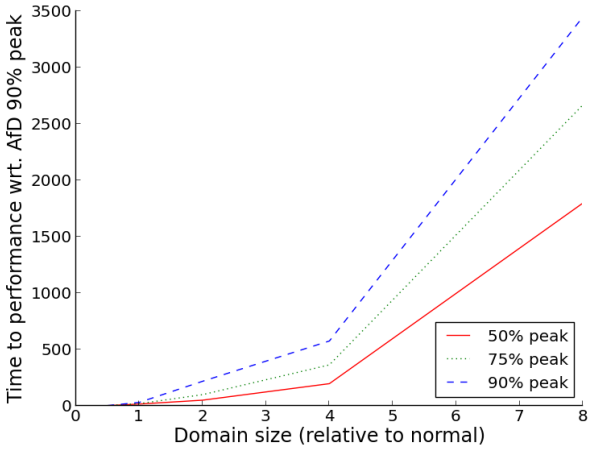
This section experimentally confirms that abstraction from demonstration provides significant speed-ups in high-dimensional domains when used in combination with previous function approximation-based RL algorithms. We tried the combination of both AfD and ADA with some of the most popular algorithms, namely, fitted Q-learning and LSPI, described in Section 6. The abstractions we use are the same as those found in our tabular-based experiments.

To measure how the size of the problem affects the speed-up provided by AfD, we experimented with different sizes of the Frogger domain. We tried variations with half, two, four, and eight times the number of car lanes on the domain. For hierarchical domains, we also experimented varying PandaSequential from one to three balls to see the effects of ADA with respect to the dimensionality of the problem.

7.4.1. AfD with fitted Q-learning on Frogger domain



(a) Reward vs. episodes across domain sizes.



(b) Speed-up provided by AfD vs. domain size, as time ratio.

Figure 6: Results with fitted Q-learning on Frogger domain, averaged over 10 runs.

Figure 6(a) shows the average discounted rewards vs. number of training episodes for fitted Q-learning, with and without the AfD abstraction (AfD+FQ and fitted Q). The peak performance is lower for bigger domain sizes, because the final reward is fixed but discounted by the number of steps, and it takes more steps to reach the goal on larger domains. Even though AfD+FQ uses fewer state features to approximate a policy, its final performance does not degrade with respect to the non-AfD algorithm. Learning on the original state

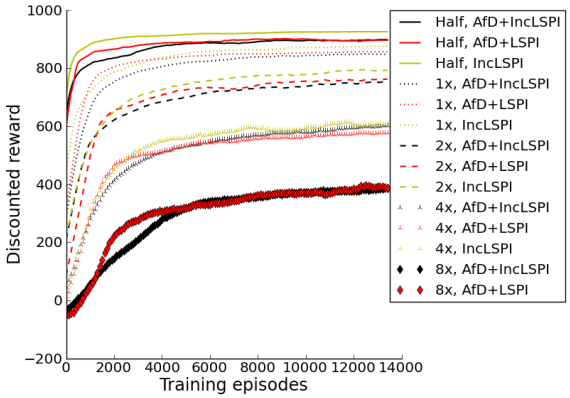
space is much slower, and the difference grows with the size of the domain. The performance of AfD+FQ also deteriorates, but most of that effect is because the average time needed to find one of the goals by chance grows with the square of the domain size, since the agent is only avoiding death and performing a random walk in early learning. AfD+FQ converges, with a steep performance slope, to the optimal policy, regardless of the size of the domain. Whereas for non-AfD fitted Q-learning, the convergence slope degrades significantly with the problem size.

Figure 6(b) shows the time it takes fitted Q-learning on the original state space to reach a level of performance comparable to AfD+FQ with respect to the size of the domain. The differences here are larger because, besides requiring fewer episodes for convergence, AfD fitted Q-learning requires fewer steps per episode and each step is faster (fitted Q-learning step complexity is linear in the number of features). The y-axis shows the computational time that the non-AfD algorithm needed to achieve a given fraction of the peak performance, divided by time required by the AfD version to reach 90% of the peak performance. For the largest domain, the non-AfD algorithm needed 3500 times longer to reach the performance of AfD+FQ, approximately 10 days instead of 4 minutes.

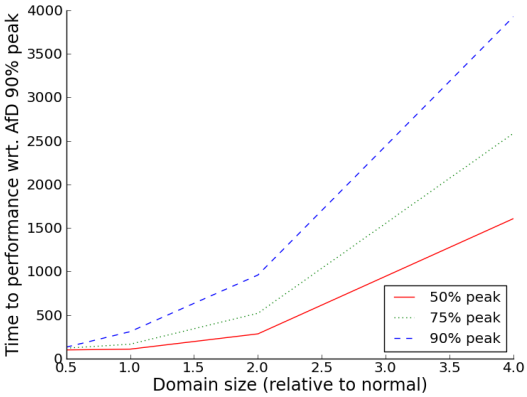
7.4.2. AfD with LSPI on Frogger domain

Figure 7 shows the results for LSPI, with a linearly decaying learning rate $\alpha = 0.001$ for IncLSPI. This experiment used both LSPI and IncLSPI combined with AfD, obtaining similar results. Due to the high dimensionality of the problem, it was not possible to use LSPI with the original state space, even for the smallest version of the domain. Again, in our experiments we used different sizes of the domain by adding more lanes to the road, using the original domain, a domain with half the number of lanes and domains with two, four and 8 times as many lanes as the original domain, which we denominate 2x, 4x and 8x domains.

We could expect the non-AfD version to perform slightly better since it has access to the whole state space, but we were surprised to see that the convergence speed in samples was the same with or without abstraction from demonstration. We believe this is due to the simplicity of the Frogger transition model, which allows model-based LSPI to learn in parallel how the state features are connected. Even though abstraction from demonstration does not make a difference in the number of samples required for convergence, it still offers significant polynomial speed-ups, as can be seen in Figure 7(b). The more efficient incremental implementation, IncLSPI, still has $O(n^2)$ complexity per sample with respect to the number of features. Non-AfD IncLSPI on the 4x domain took three weeks to complete 15000 episodes, while the AfD version ran the same number of episodes with similar performance in just 87 minutes. We were unable to run non-AfD LSPI on the 8x version: the raw state space has a total of 55000 features, which made it impossible to run a single episode on a 16-core 12GB RAM machine. From this experiment, we conclude that by drastically reducing the number of features, abstraction from demonstration makes LSPI applicable to larger domains.



(a) Reward vs. episodes accross domain sizes.



(b) Speed-up provided by AfD vs. domain size.

Figure 7: Results with LSPI on the Frogger domain, averaged over 10 runs.

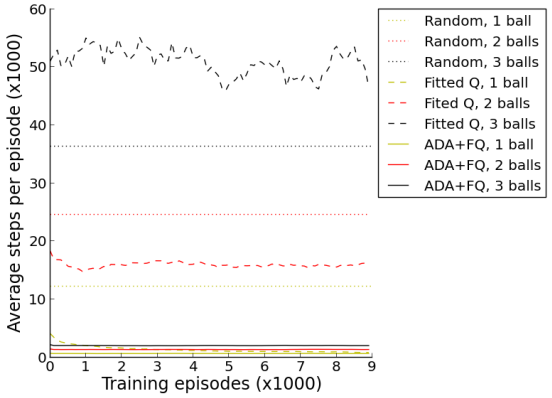
We also tried using LARS-TD in this domain to compare feature selection through regularization and our feature selection with human demonstrations. With a regularization criteria of 0.001, LARS-TD selected 300 non-zero features after the first iteration. This is many more than the nine features suggested by AfD. We tried different regularization parameter settings (ranging from 0.1 to 0.0001) and in all cases found the results to be consistent. This behavior can be explained by highlighting the difference in objectives between the respective algorithms.

LARS-TD is performing feature selection by iteratively computing the L1 regularized fixed points that reduce the TD error. In other words it is (implicitly) selecting features useful to represent the value function of the MDP. However the AfD approach is motivated to select features useful to represent

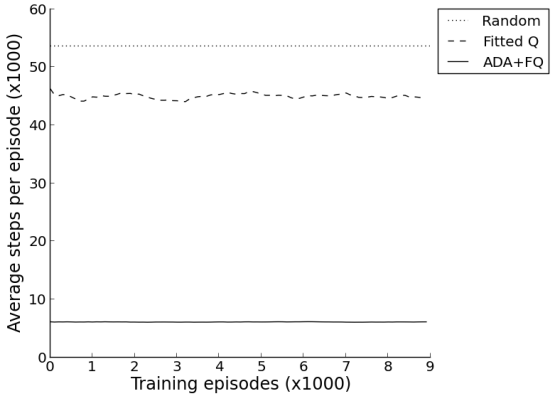
the human demonstrator's policy. These features are what the human is paying attention to when making decisions and are not directly concerned with the value function. It is this characteristic that leads to marked distinctions in the performance of the two algorithms. In our domains, the features necessary to represent the human's policy were significantly smaller than those necessary to learn the value function.

Additionally the LARS-TD time complexity of $O(mnk^3)$ outweighs the computational gains provided by the feature selection, because in our domains $m >> k, p$. The algorithm could not resolve even the smallest version of the domain after weeks of running time.

7.4.3. ADA with fitted Q-learning on Panda domains



(a) Average steps per episode for PandaSequential.



(b) Average steps per episode for RainbowPanda.

Figure 8: Results with fitted Q-learning on Panda domains, averaged over 10 runs.

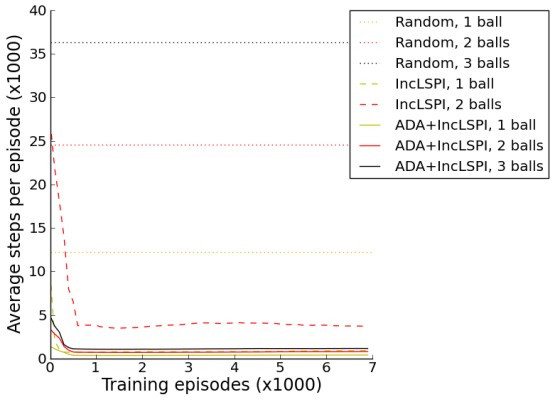
We now compare linear fitted Q-learning on the original state space and on the ADA abstraction in the PandaSequential and RainbowPanda domains. For PandaSequential, we compare versions with one, two, and three balls to study the scalability of the algorithms. The results are shown in Figure 8(a). For clarity, these graphs show the average number of steps that it took to complete the task, so a lower value means better performance. With only one ball, there is only one subtask; therefore, the ADA abstraction is equivalent to AfD. The convergence of the non-ADA version is extremely slow and takes almost 10000 episodes, while the ADA version needs only a few episodes. This is an interesting result because the difference between both algorithms in the one-ball case is whether they take into account only the distance to the ball or both distance and angle to the ball. Using both parameters could lead to slightly better policies, but we confirmed that, even after 50000 episodes, the policy of the non-ADA algorithm was not better than ADA.

The two-ball and three-ball cases of PandaSequential are also interesting. We expected that the non-ADA version would be able to learn a good policy for this domain. Because the balls are always picked in the same order, it is possible to express a near-optimal policy as a linear combination of the state features: replicate the weights for the features of the first ball scaled down for the subsequent balls, so that the weights associated with the next ball to pick up always dominate; however, fitted-Q was not able to find these policies and performed just a bit better than the random policy for two balls and even worse than the random policy for three balls. The ADA version of the algorithm converges equally fast in number of episodes, regardless of number of balls, and the length of the episodes grows linearly with the number of balls, as expected for a near-optimal policy.

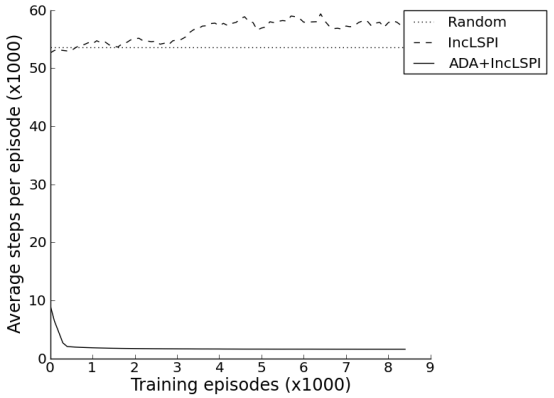
We correctly expected poor results for RainbowPanda, because in this domain it is not possible to represent a global policy using only a linear combination of features. We could represent a global policy only if we add features like the Cartesian product of the position of each ball and the active color, but that entails manually coding domain-specific information, which is what abstraction from demonstration tries to avoid. Figure 8(b) shows that, unable to represent the policy, the non-ADA version of the algorithm performs poorly while the ADA version converges quickly to a near-optimal policy.

7.4.4. ADA with LSPI on Panda domains

Figures 9(a) and 9(b) show an effect on LSPI similar to the one on fitted Q-learning. ADA expands the hypotheses space of LSPI (IncLSPI with initial $\alpha = 10^{-8}$) so that it can find near-optimal policies for more complex domains. For PandaSequential, the non-ADA policy for two balls is three times quicker than the policy found with non-ADA fitted Q-learning. However the three-ball policy for non-ADA LSPI (not shown in the figure) is much worse, oscillating between 120 and 160 thousand steps, more than four times slower than the random policy.



(a) Average steps per episode for PandaSequential.



(b) Average steps per episode for RainbowPanda.

Figure 9: Results with LSPI on Panda domains, averaged over 10 runs.

7.4.5. Tree-based non-linear approximation and automatic abstraction

We tested the decision tree regression methods implemented in Scikit-learn [47] on Frogger and Panda domains. The training samples where obtained by following an epsilon-greedy policy with a decaying epsilon. The input component of the training samples were the state (represented using the original state variables) and the action taken (represented as a categorical variable). We used a fixed number of iterations as the stopping condition.

We report that the algorithm was able to solve the smallest version of the Frogger domain with performance matching that of the LSPI variants (see Figure 10), however failed to solve larger, more complex, versions of Frogger as well as any version of the Panda domain. We tested both CART decision trees and Extra Trees ensemble methods (using 5 to 15 trees in the forest) along with al-

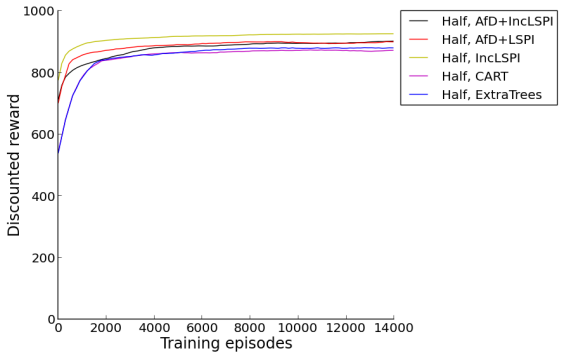


Figure 10: Results with Tree-based Q-Iteration on the Frogger domain, averaged over 10 runs.

ternate representations for the training data (those used for IncLSPI and Fitted Q-learning) and the results were consistent.

We explain this behavior by highlighting some design aspects of the algorithm. Tree-based Fitted Q-Iteration is a batch algorithm and does not lend itself to online adaptation. For high-dimensional domains like the ones used in our work, this requires the storage of a large number of samples (necessary to cover interesting parts of the state space) which leads to prohibitively expensive memory requirements.[4] This reasoning also explains our choice behind using an incremental version of LSPI, as the standard LSPI algorithm is a batch method. Given insufficient training samples (and the lack of a learning rate), the stochastic approximation conditions of Q-learning are no longer satisfied and the algorithm is not guaranteed to converge.

We would also like to highlight the differences in the abstractions learned from decision trees and AfD. From the trees learned on the smallest version of Frogger, we observed that the features considered to be important were the ones useful to regress on the Q-function and not particularly related to the underlying policy. This characteristic makes this approach prone to regression errors while the AfD approach does not concern itself with predicting the correct value.

8. Discussion

Direct LfD tries to approximate the mapping from states to actions; however, in reality, the teacher's policy is not necessarily deterministic or stationary. Different teachers may also teach different but equally valid policies. Thus, for any given state, there may be a set of equally appropriate actions; hence, we are in a multi-label supervised learning setting [69], but the data is not multi-label.

[4]The authors of the Tree-based Q-Iteration algorithm tested their approach on relatively simple domains involving seven dimensional state variables.

For any single action, we see examples of when it should be used, but never examples of when it should not be. This problem stems from the fundamental fact that when a teacher shows an appropriate action for a given state, they are not necessarily indicating that other actions are inappropriate. This produces a situation commonly referred to as "positive and unlabeled data" [34]. One approach to dealing with positive and unlabeled data is to assume that observed positive examples are chosen randomly from the set of all positive examples [17]. Unfortunately, this is not true for human demonstrations. Abstraction from demonstration approaches avoid this issue by only using the data to identify relevant features. It does not matter how well we can approximate the demonstrated policy, only that a feature has a positive contribution toward the approximation.

One "unfair" advantage of ADA over some other methods is that we must provide it with a set B of candidate boundaries, which is a form of domain information. In principle, we could chose as candidate boundaries every possible subset of S, but this would be computationally intractable, so we must explicitly choose the candidate boundaries. This is a small price to pay for the performance gains of the algorithm. As a default choice, axis-aligned boundaries, i.e., thresholds on a single feature, are a compact class that works well across a diverse range of domains. They would work for the Taxi domain [15], which is the typical example of task decomposition in RL, using as a boundary whether the passenger has been picked up or not. If the features are learned from low-level sensing information using unsupervised feature learning techniques, it is likely that one of the generated features will provide adequate thresholds. Additionally, many learning algorithms have similar kinds of bias, *e.g.,* decision trees also only consider thresholds on a single feature, just like ADA in our experimental settings.

A significant limitation of ADA is that it does not consider second-order mutual information relationships, and these can be relevant. For example, we can imagine a domain where the desired action depends on whether two independent random variables have the same value. The mutual information between each variable and the action might be 0, but the mutual information between both variables and the action would account for all the entropy of the action. We decided to use only first-order mutual information because we believe it is enough to obtain a good decomposition of a wide range of problems, and because the number of samples needed to get an accurate estimate of higher-order relationships is much larger. However, if a large number of demonstrations is available, ADA can be easily extended to use these additional mutual information measures.

One additional advantage of ADA is that it can be used as part of a larger system of *transfer learning*. Once an autonomous agent learns a new task and the subtasks it decomposes into, the subtask policies can be useful for other tasks that may be decomposed in a similar way. For example, an agent might, as part of the policy improvement step for a specific subtask, try policies of previously learned subtasks that have the same abstraction, perhaps after comparing policies and determining that the subtasks are similar. Demonstrating

the utility of ADA for transfer learning is an important area of future work.

9. Conclusions

Abstraction from demonstration approaches perform better than policies built using direct LfD, even when LfD uses an order of magnitude more demonstrations. Sample efficiency is one of the key advantages of our algorithms, given that obtaining good human demonstrations is often expensive and time-consuming.

Because of the cost of acquiring human samples, it might be desirable to avoid it altogether and use direct RL algorithms without using demonstrations; however, AfD and ADA achieve significant speed-ups by taking advantage of the exponential savings of dimensionality reduction, and converge to a high-performance policy in minutes, while learning without demonstrations did not show improvement over the initial random policy, even after days of computation. This speed-up suggests that, in many domains, even including the time and cost required to acquire the human demonstrations, AfD will be more cost-effective and time-effective than learning without using human demonstrations.

Another advantage of abstraction from demonstration is that its performance is not limited to that of the teacher. AfD and ADA use the reward signal to obtain the best policy that can sometimes be expressed in their reduced feature space. This policy, as our results show, can be significantly better than that of the teacher.

Abstraction from demonstration can also extend the class of sequential decision problems that can be solved with FA-based RL algorithms such as fitted Q-learning and LSPI, simultaneously reducing the amount of manual feature engineering necessary to adapt the algorithms to new domains. This is achieved through polynomial speed-ups with respect to the number of state features and extensions of the hypothesis space by task decomposition.

References

[1] P. Abbeel, A.Y. Ng, Apprenticeship learning via inverse reinforcement learning, Proc. Int. Conf. on Machine Learning (2004) 1.

[2] R. Aler, O. Garcia, J. Valls, Correcting and Improving Imitation Models of Humans for Robosoccer Agents, 2005 IEEE Congress on Evolutionary Computation (2005) 2402–2409.

[3] G. Aloupis, E. Demaine, A. Guo, Classic Nintendo Games are (NP-) Hard, Arxiv preprint arXiv:1203.1895 (2012) 1–21.

[4] B.D. Argall, S. Chernova, M. Veloso, B. Browning, A survey of robot learning from demonstration, Robotics and Autonomous Systems 57 (2009) 469–483.

[5] A. Barto, Recent advances in hierarchical reinforcement learning, Discrete Event Dynamic Systems 13 (2003) 341–379.

[6] J. Beitelspacher, J. Fager, G. Henriques, A. McGovern, Policy Gradient vs. Value Function Approximation: A Reinforcement Learning Shootout, Technical Report February, School of Computer Science, University of Oklahoma, 2006.

[7] L. Busoniu, A. Lazaric, M. Ghavamzadeh, R. Munos, R. Babuska, B.D. Schutter, Least-squares methods for policy iteration, Reinforcement Learning (2012) 75–109.

[8] D. Chapman, L. Kaelbling, Input generalization in delayed reinforcement learning: An algorithm and performance comparisons, Proc. Int. Joint Conf. on Artificial Intelligence (1991) 726–731.

[9] A. Coates, P. Abbeel, A. Ng, Apprenticeship learning for helicopter control, Communications of the ACM (2009) 144–151.

[10] L.C. Cobo, C.L. Isbell, A.L. Thomaz, Automatic task decomposition and state abstraction from demonstration, Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (2012) 483–490.

[11] L.C. Cobo, C.L. Isbell, A.L. Thomaz, Object Focused Q-learning for Autonomous Agents, Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (2013) 1061–1068.

[12] L.C. Cobo, P. Zang, C.L. Isbell, A.L. Thomaz, Automatic state abstraction from demonstration, Proc. Int. Joint Conf. on Artificial Intelligence (2011) 1243–1248.

[13] N. Cowan, The Magical Mystery Four: How Is Working Memory Capacity Limited, and Why?, Current Directions in Psychological Science 19 (2010) 51–57.

[14] O. Şimşek, A.G. Barto, Using relative novelty to identify useful temporal abstractions in reinforcement learning, Proc. Int. Conf. on Machine Learning (2004) 751–758.

[15] T. Dietterich, The MAXQ method for hierarchical reinforcement learning, Proc. Int. Conf. on Machine Learning (1998) 118–126.

[16] P. Domingos, A few useful things to know about machine learning, Communications of the ACM 55 (2012) 78–87.

[17] C. Elkan, K. Noto, Learning classifiers from only positive and unlabeled data, Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data mining (2008) 213–220.

[18] D. Ernst, P. Geurts, L. Wehenkel, Tree-based batch mode reinforcement learning, Journal of Machine Learning Research 6 (2005) 503–556.

[19] A.M. Farahmand, M. Ghavamzadeh, C. Szepesvari, S. Mannor, Regularized policy iteration, Advances in Neural Information Processing Systems (2009) 441–448.

[20] S. Finney, N. Gardiol, L. Kaelbling, T. Oates, Learning with deictic representation, Technical Report April, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2002.

[21] A. Geramifard, M. Bowling, M. Zinkevich, R. Sutton, iLSTD : Eligibility Traces and Convergence Analysis, Advances in Neural Information Processing Systems (2007) 441–448.

[22] M. Hall, Correlation-based feature selection for machine learning, Ph.D. thesis, The University of Waikato, 1999.

[23] B. Hengst, Discovering hierarchy in reinforcement learning with HEXQ, Proc. Int. Conf. on Machine Learning (2002) 234–250.

[24] N. Jong, T. Hester, P. Stone, The utility of temporal abstraction in reinforcement learning, Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (2008) 299–306.

[25] N.K. Jong, P. Stone, State Abstraction Discovery from Irrelevant State Variables, Proc. Int. Joint Conf. on Artificial Intelligence (2005) 752–757.

[26] A. Jonsson, A. Barto, Automated state abstraction for options using the U-tree algorithm, Advances in Neural Information Processing Systems (2001) 1054–1060.

[27] P.W. Keller, S. Mannor, D. Precup, Automatic basis function construction for approximate dynamic programming and reinforcement learning, Proc. Int. Conf. on Machine Learning (2006) 449–456.

[28] W. Knox, P. Stone, Combining Manual Feedback with Subsequent MDP Reward Signals for Reinforcement Learning, Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (2010) 10–14.

[29] J.Z. Kolter, A.Y. Ng, Regularization and feature selection in least-squares temporal difference learning, Proc. Int. Conf. on Machine Learning 94305 (2009) 1–8.

[30] G. Konidaris, A. Barto, Efficient skill learning using abstraction selection, Proc. Int. Joint Conf. on Artificial Intelligence (2009) 1107–1112.

[31] G. Konidaris, S. Kuindersma, R. Grupen, A. Barto, Robot learning from demonstration by constructing skill trees, The International Journal of Robotics Research 31 (2012) 360–375.

[32] M.G. Lagoudakis, R. Parr, Least-Squares Policy Iteration, Journal of Machine Learning Research 4 (2003) 1107–1149.

[33] H. Lee, R. Grosse, R. Ranganath, A. Ng, Unsupervised learning of hierarchical representations with convolutional deep belief networks, Communications of the ACM 54 (2011) 95–103.

[34] F. Letouzey, F. Denis, R. Gilleron, Learning from positive and unlabeled examples, Algorithmic Learning Theory (2009) 71–85.

[35] L. Li, A worst-case comparison between temporal difference and residual gradient with linear function approximation, Proc. Int. Conf. on Machine Learning (2008) 560–567.

[36] L. Li, M.L. Littman, Reducing reinforcement learning to KWIK online regression, Annals of Mathematics and Artificial Intelligence 58 (2010) 217–237.

[37] L. Li, T. Walsh, M. Littman, Towards a unified theory of state abstraction for MDPs, Proc. Int. Symp. on Artificial Intelligence and Mathematics (2006) 531–539.

[38] H. Maei, C. Szepesvári, S. Bhatnagar, R. Sutton, Toward off-policy learning control with function approximation, Proc. Int. Conf. on Machine Learning (2010) 719–726.

[39] S. Mannor, I. Menache, A. Hoze, U. Klein, Dynamic abstraction in reinforcement learning via clustering, Proc. Int. Conf. on Machine Learning (2004) 71–78.

[40] A. McCallum, Learning to use selective attention and short-term memory in sequential tasks, From animals to animats 4 (1996) 315–324.

[41] A. McGovern, A. Barto, Automatic discovery of subgoals in reinforcement learning using diverse density, Proc. Int. Conf. on Machine Learning (2001) 361–368.

[42] N. Mehta, S. Ray, P. Tadepalli, T. Dietterich, Automatic discovery and transfer of MAXQ hierarchies, Proceedings of the 25th international conference on Machine learning (2008) 648–655.

[43] S. Minut, S. Mahadevan, A reinforcement learning model of selective visual attention, Proc. Int Conf. on Autonomous Agents (2001) 457–464.

[44] A.Y. Ng, H.J. Kim, M.I. Jordan, S. Sastry, Autonomous helicopter flight via Reinforcement Learning, Advances in Neural Information Processing Systems 16 (2004).

[45] T. Nørretranders, The user illusion: Cutting consciousness down to size., Viking, 1991.

[46] R. Parr, C. Painter-Wakefield, L. Li, M. Littman, Analyzing feature generation for value-function approximation, Proc. Int. Conf. on Machine Learning (2007) 737–744.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[48] D. Precup, R. Sutton, Off-policy temporal-difference learning with function approximation, Proc. Int. Conf. on Machine Learning (2001) 417–424.

[49] J. Quinlan, C4.5: programs for machine learning, Morgan Kaufmann, 1993.

[50] M. Riedmiller, Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method (2005) 317–328.

[51] H. van Seijen, S. Whiteson, L. Kester, Efficient abstraction selection in reinforcement learning, SARA 2013: Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation (2013).

[52] H.V. Seijen, S. Whiteson, L. Kester, Switching between representations in reinforcement learning, Interactive Collaborative Information Systems (2010) 65–84.

[53] H.A. Simon, Administrative behavior, volume 4, Cambridge University Press, 1957.

[54] S. Singh, T. Jaakkola, M. Jordan, Learning without state-estimation in partially observable Markovian decision processes, Proc. Int. Conf. on Machine Learning (1994) 284–292.

[55] W. Smart, L. Pack Kaelbling, Effective reinforcement learning for mobile robots, Proceedings 2002 IEEE International Conference on Robotics and Automation (2002) 3404–3410.

[56] M. Stolle, D. Precup, Learning Options in Reinforcement Learning, Abstraction, Reformulation, and Approximation (2002) 212–223.

[57] R. Sutton, A. Barto, Reinforcement learning: An introduction, Cambridge University Press, 1998.

[58] R. Sutton, H. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, E. Wiewiora, Fast gradient-descent methods for temporal-difference learning with linear function approximation, Proceedings of the 26th Annual International Conference on Machine Learning (2009) 993–1000.

[59] R. Sutton, D. McAllester, S. Singh, Policy gradient methods for reinforcement learning with function approximation, Advances in Neural Information Processing Systems (2000) 1057–1063.

[60] R.S. Sutton, D. Precup, S. Singh, Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning, Artificial Intelligence 112 (1999) 181–211.

[61] C. Szepesvari, Algorithms for Reinforcement Learning, Synthesis Lectures on Artificial Intelligence and Machine Learning (2010) 1–98.

[62] E. Talvitie, S. Singh, Simple Local Models for Complex Dynamical Systems, Advances in Neural Information Processing Systems (2008) 1617–1624.

[63] G. Tesauro, Programming backgammon using self-teaching neural nets, Artificial Intelligence 134 (2002) 181–199.

[64] C.M. Vigorito, A.G. Barto, Intrinsically Motivated Hierarchical Skill Learning in Structured Environments, IEEE Trans. on Autonomous Mental Development 2 (2010) 132–143.

[65] S. Whiteson, P. Stone, Evolutionary function approximation for reinforcement learning, The Journal of Machine Learning Research 7 (2006) 877–917.

[66] T. Wilson, Strangers to ourselves: Discovering the adaptive unconscious, Harvard University Press, 2002.

[67] X. Xu, D. Hu, X. Lu, Kernel-based least squares policy iteration for reinforcement learning., IEEE Trans. on Neural Networks 18 (2007) 973–92.

[68] P. Zang, P. Zhou, D. Minnen, C. Isbell, Discovering options from example trajectories, Proc. Int. Conf. on Machine Learning (2009) 1217–1224.

[69] M. Zhang, Z. Zhou, ML-KNN: A lazy learning approach to multi-label learning, Pattern Recognition 40 (2007) 2038–2048.