Reinforcement Learning in Multidimensional Continuous Action Spaces

Jason Pazis
Department of Computer Science
Duke University
Durham, NC 27708–0129, USA
Email: jpazis@cs.duke.edu

Michail G. Lagoudakis
Department of Electronic and Computer Engineering
Technical University of Crete
Chania, Crete 73100, Greece
Email: lagoudakis@intelligence.tuc.gr

***Abstract*—The majority of learning algorithms available today focus on approximating the state (V) or state-action (Q) value function and efficient action selection comes as an afterthought. On the other hand, real-world problems tend to have large action spaces, where evaluating every possible action becomes impractical. This mismatch presents a major obstacle in successfully applying reinforcement learning to real-world problems. In this paper we present an effective approach to learning and acting in domains with multidimensional and/or continuous control variables where efficient action selection is embedded in the learning process. Instead of learning and representing the state or state-action value function of the MDP, we learn a value function over an implied augmented MDP, where states represent collections of actions in the original MDP and transitions represent choices eliminating parts of the action space at each step. Action selection in the original MDP is reduced to a binary search by the agent in the transformed MDP, with computational complexity logarithmic in the number of actions, or equivalently linear in the number of action dimensions. Our method can be combined with any discrete-action reinforcement learning algorithm for learning multidimensional continuous-action policies using a state value approximator in the transformed MDP. Our preliminary results with two well-known reinforcement learning algorithms (Least-Squares Policy Iteration and Fitted Q-Iteration) on two continuous action domains (1-dimensional inverted pendulum regulator, 2-dimensional bicycle balancing) demonstrate the viability and the potential of the proposed approach.**

I. Introduction

The goal of Reinforcement Learning (RL) is twofold: accurately estimating the value of a particular policy and finding good policies. A large body of research has been devoted in finding effective ways to approximate the value function of a particular policy. While in certain applications estimating the value function is interesting in and of itself, most often our ultimate goal is to use such an estimate to act in an intelligent manner. Unfortunately, the majority of RL algorithms available today focus on approximating the state (V) or state-action (Q) value function and efficient action selection comes as an afterthought. On the other hand, real-world problems tend to have large action spaces, where evaluating every possible action is impractical. This mismatch presents a major obstacle to successfully applying RL in real-world problems.

When it comes to value-function-based algorithms, the goal is to learn a mapping from states or state-action pairs to real numbers which represent the desirability of a certain state or state-action combination. Two major problems exist with using state value functions. The first is that a state value function on its own tells us something about how good the state we are currently in is, but is not sufficient for acting. In order to figure out the best action, we need a model to compute what the expected next states given each action will be, so we can then compare the values of different actions based on their predicted effects. Unfortunately, in many applications, a model is not available or it may be too expensive to evaluate and/or store. The second problem is that, if the number of possible actions is too large, evaluating each one explicitly (which is required for exact inference unless we make strong assumptions about the shape of the value function) becomes impractical. State-action value functions address the first problem by directly representing the value of each action at each state. Thus, picking the right action becomes the *conceptually* simple task of examining each available action at the current state and picking the one that maximizes the value. Unfortunately, state-action value functions don't address the second problem. Picking the right action may still require solving a difficult non-linear maximization problem.

In this paper, we build upon our previous work on learning continuous-action control policies through a form of binary search over the action space using an augmented Q-value function [1]. We derive an equivalent V-value-function-based formulation and extend it to multidimensional action spaces by realizing (to our knowledge, for the first time) an old idea for pushing action space complexity into state space complexity [2], [3]. Efficient action selection is directly embedded into the learning process, where instead of learning and representing the state or state-action value function over the original MDP, we learn a state value function over an implied augmented MDP, whose states also represent collections of actions in the original MDP and transitions also represent choices eliminating parts of the action space. Thus, action selection in the original MDP is reduced to a binary search in the transformed MDP, whose complexity is linear in the number of action dimensions. We show that the representational complexity of the transformed MDP is within a factor of 2 from that of the original, without relying on any assumptions about the shapes of the action space and/or the value function. Finally, we compare our approach to others in the literature, both theoretically and experimentally.

II. Background

A. Markov Decision Processes

A *Markov Decision Process* (MDP) is a 6-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \mathcal{D})$, where $\mathcal{S}$ is the state space of the process, $\mathcal{A}$ is the action space of the process, P is a Markovian transition model $\big(P(s'|s,a)$ denotes the probability of a transition to state s' when taking action a in state $s\big)$, R is a reward function $\big(R(s,a)$ is the expected reward for taking action a in state $s\big)$, $\gamma \in (0,1]$ is the discount factor for future rewards, and $\mathcal{D}$ is the initial state distribution. A *deterministic policy* π for an MDP is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$ from states to actions; $\pi(s)$ denotes the action choice in state s. The value $V_\pi(s)$ of a state s under a policy π is defined as the expected, total, discounted reward when the process begins in state s and all decisions are made according to policy π:

$$V_\pi(s) = E_{a_t \sim \pi; s_t \sim P}\left[\sum_{t=0}^{\infty} \gamma^t R\big(s_t, a_t\big) \Big| s_0 = s\right].$$

The value $Q_\pi(s,a)$ of a state-action pair (s,a) under a policy π is defined as the expected, total, discounted reward when the process begins in state s, action a is taken at the first step, and all decisions thereafter are made according to policy π:

$$Q_\pi(s,a) = E_{a_t \sim \pi; s_t \sim P}\left[\sum_{t=0}^{\infty} \gamma^t R\big(s_t, a_t\big) \Big| s_0 = s, a_0 = a\right].$$

The goal of the decision maker is to find an optimal policy π^* for choosing actions, which maximizes the expected, total, discounted reward for states drawn from $\mathcal{D}$:

$$\pi^* = \arg\max_\pi E_{s \sim \mathcal{D}}\left[V_\pi(s)\right] = \arg\max_\pi E_{s \sim \mathcal{D}}\left[Q_\pi\big(s, \pi(s)\big)\right].$$

For every MDP, there exists at least one optimal deterministic policy. If the value function V_{π^*} is known, an optimal policy can be extracted, only if the full MDP model of the process is also known to allow for one-step look-aheads. On the other hand, if Q_{π^*} is known, a greedy policy, which simply selects actions that maximize Q_{π^*} in each state, is an optimal policy and can be extracted without the MDP model. Value iteration, policy iteration, and linear programming are well-known methods for deriving an optimal policy from the MDP model.

B. Reinforcement Learning

In Reinforcement Learning (RL), a learner interacts with a stochastic process modeled as an MDP and typically observes the state of the process and the immediate reward at every step, however P and R are not accessible. The goal is to gradually learn an optimal policy using the experience collected through interaction with the process. At each step of interaction, the learner observes the current state s, chooses an action a, and observes the resulting next state s' and the reward received r, essentially sampling the transition model and the reward function of the process. Thus, experience comes in the form of (s,a,r,s') samples. Several algorithms have been proposed for learning good or even optimal policies from (s,a,r,s') samples [4].

III. Related Work

A. Scope

Our focus is on problems where decisions must be made under strict time and hardware constraints, with no access to a model of the environment. Such problems include many control applications, such as controlling an unmanned aerial vehicle or a dynamically balanced humanoid robot. Extensive literature exists in the mathematical programming and operations research communities dealing with problems having many and/or continuous control variables. Unfortunately, the majority of these results are not very well suited for our purposes. Most assume availability of a model and/or do not directly address the action selection task, leaving it as a time consuming, non-linear optimization problem that has to be solved repeatedly during policy execution. Thus, our survey will be focused on approaches that align with the assumptions commonly made by the RL community.

B. The main categories

There are two main components in every approach to learning and acting in continuous and/or multidimensional action spaces. The first is the choice of what to represent, while the second is how to choose actions.

Even though many RL approaches have been presented in the context of some representation scheme (neural-networks, CMACs, nearest-neighbors), upon careful analysis we realized that, besides superficial differences, most of them are very similar to one another. In particular, two main categories can be identified.

The first and most commonly encountered category uses a combined state-action approximator for the representation part, thus generalizing over both states and actions. Since approaches in this category essentially try to learn and represent the same thing, they only differ in the way they query this value function in order to perform the maximization step. This can involve sampling the value function in a uniform grid over the action space at the current state and picking the maximum, Monte Carlo search, Gibbs sampling, stochastic gradient ascent, and other optimization techniques. One should notice however, that these approaches don't have any significant difference from approaches in other communities where the maximization step is recognized as a non-linear maximization and is tackled with standard mathematical packages. To our knowledge, all the methods proposed for the maximization step have already been studied outside the RL community.

The second category deals predominantly with continuous (rather than multidimensional) control variables and is usually closely tied to online learning. The action space is discretized and a small number of different, discrete approximators are used for representing the value function. However, when acting, instead of picking the discrete action that has the highest value, the actions are somehow "mixed" depending on their relative values or "activations". The mixing can be either between the discrete action with the highest predicted value and its closest neighbor(s), or even a weighted average over all discrete actions (where the weights are the predicted values). Online learning comes into play in the way the action values are updated. The learning update is distributed over the actions that were used to produce the action, thus, with multiple updates the values that each discrete action approximator represents, may drift far from what the value of that particular discrete action is for the domain in question. Although this allows the agent to develop preferences for actions that fall between approximators, it is unclear under what conditions these schemes converge. Additionally, such approaches scale poorly to multidimensional action spaces. Even for a small number of discrete actions from which one can interpolate in each dimension, the combinatorial explosion soon makes the problem intractable. In order to deal with this shortcoming, the domain is often partitioned into multiple independent subproblems, one for each control variable. However, by assigning a different agent to each control variable, we are essentially casting the problem into a multiagent setting, where avoiding divergence or local optima is much more difficult.

Bellow, we provide a brief description of what we believe is a representative sample of approaches that have appeared in the RL literature. This discussion does not, in any way, attempt to be complete. Our goal is to highlight the similarities and differences between these approaches and provide our own (biased) view of their strengths and weaknesses.

C. Approaches

Santamaría, Sutton, and Ram [5] provide one of the earliest examples of generalizing across both states and actions in RL. They demonstrate that a combined state-action approximator can have an advantage in continuous action spaces, where neighboring actions have similar outcomes. Their approach was originally presented in conjunction with CMACs, however it can be combined with almost any type of approximator. It has proven to be effective at generalizing over continuous action spaces and can be used with multiple control variables. Unfortunately, it does not address the problem of efficient action selection. Without further assumptions, it requires an exhaustive search over all available action combinations, which quickly becomes impractical as the size of the action space grows.

One popular approach to dealing with the action selection problem is sampling [6], [7]. The representation is the same as above, however, using some form of Monte-Carlo estimation, the controller is able to choose actions that have a high probability of performing well, without exhaustively searching over all possible actions [8]. Unfortunately, the number of samples required in order to get a good estimate can be quite high, especially in large and not very well-behaved action spaces.

Originally presented in conjunction with incremental topology-preserving maps, continuous-action Q-learning [9] by Millán, Posenato and Dedieu can be generalized to use other types of approximators. The idea is to use a number of discrete approximators and output an average of the discrete actions weighted by their Q-values. The incremental updates are proportional to each unit's activation. Ex<a> [10] by Martin and de Lope differs from continuous-action Q-learning in that it interprets Q-values as probabilities. When it comes to selecting a maximizing action and updating the value function, the idea is very similar to continuous-action Q-learning. In this case, the continuous action is calculated as an expectation over discrete actions.

Policy gradient methods [11] circumvent the need for value functions by representing policies directly. One of their main advantages is that the approximate policy representation can often output continuous actions directly. In order to tune their policy representation, these methods use some form of gradient descent, updating the policy parameters directly. While they have proven effective at improving an already reasonably good policy, they are rarely as effective in learning a good policy from scratch, due to their sensitivity to local optima.

Scaling efficient action selection to multidimensional action spaces has been primarily investigated in collaborative multi-agent settings, where each agent corresponds to one action dimension, under certain assumptions (factored value function representations, hierarchical decompositions, etc.). Bertsekas and Tsitsiklis [2] (Section 6.1.4) introduced a generic approach of trading off control space complexity with state space complexity by making incremental decisions over an augmented state space. This idea was further formalized by de Farias and Van Roy [3] as an MDP transformation encoding multidimensional action selection into a series of simpler action choices.

Finally, some methods exploit certain domain properties, such as temporal locality of actions [12], [13], modifying the current action by some Δ at every step. However not only do they not scale well to multidimensional action spaces, but their performance is also limited by the implicit presence or explicit use of a low pass filter on the action output, since they are only able to pick actions close to the current action.

IV. Action search in continuous and multidimensional action spaces

As described in the previous section, there are two main components in every approach to learning and acting in continuous and/or multidimensional action spaces. Each component aligns with one of the two problems that surface when the number of available actions becomes large. The first problem is how to generalize among different actions. It has long been recognized that the naive approach of using a different approximator for each action quickly becomes impractical, just as tabular representations become impractical when the number of states grows. We believe that many of the approaches available offer an adequate solution to this problem.

The second issue, that becomes apparent when the number of available actions becomes large, is selecting the right action using a reasonable amount of computation. Even if we have an optimal state-action value function, the number of actions available at a particular state may be too large to enumerate at every step. This is especially true in multidimensional action spaces where, even if the resolution of each control variable is low, the available action combinations grow exponentially. In our view, while many approaches offer a reasonable compromise between computational effort and accuracy in action selection, there is room for significant improvement.

It should be apparent from the previous discussion that most approaches deal with the two problems separately. We believe that in order to be able to provide an adequate answer to the action selection problem, we must design our representation to facilitate it and this is the approach we explore in this paper. The value function learned is designed to allow efficient action selection, instead of the latter coming as an afterthought. We are able to do this without making any assumptions about the shape of, or our ability to decompose, the action space.

We transform the problem of generalizing among actions to a problem of generalizing among states in an equivalent MDP (cf. [2], [3]), where action selection is trivial. Arguably such an approach does not offer any reduction in the complexity of what has to be learned (in fact we will show that in the case of exact representation the memory requirements are within a factor of 2 from the original). Nevertheless, the benefits of using such an approach are twofold. Firstly, it allows us to leverage all the research devoted to effective generalization over states. Instead of having to deal with two different problems, generalizing over states and generalizing over actions, we now have to deal with the single problem of generalizing over states. Secondly, it offers an elegant solution to the action selection problem, which requires exponentially less computation per decision step.

A. MDP transformation

Consider an MDP $\mathcal{M}(\mathcal{S}, \mathcal{A}, P, R, \gamma, \mathcal{D})$, where at each state $\mathcal{S}$ our agent has to choose among the set of available actions $\mathcal{A}$. We will transform $\mathcal{M}$ using a recursive decomposition of the action space available to each state. The first step is to replace each state s of $\mathcal{M}$ with 3 states s'_0, s'_1, and s'_2. State s'_0 has two actions available. The first leads deterministically to s'_1, while the second leads deterministically to s'_2. In state s'_1 we have access to the first half of the actions available in s while in s'_2 we have access to the other half. The transition between s'_0 to s'_1, or s'_0 to s'_2 is undiscounted and does not receive a reward. Therefore, we have that $V(s'_0) = V(s) = \max_{a \in \mathcal{A}} Q(s, a)$ while at least one of the following is also true: $V(s) = V(s'_1)$ or $V(s) = V(s'_2)$. We can think of the transformation as creating a state tree, where the root has deterministic dynamics with the go-left and go-right actions available, zero reward and no discount ($\gamma = 1$). Each leaf has half the number of available actions as the original MDP and the union of actions available to all the leaves is $\mathcal{A}$, the same as those available in the original MDP.

Applying this decomposition recursively to the leaves of the previous step, with individual leaves from each iteration having half the number of actions, leads to the transformed MDP $\mathcal{M}'$ where for each state in $\mathcal{M}$ we have a full binary tree in $\mathcal{M}'$ and each leaf has only one available action. If we represent the i-th leaf state under the tree for state s as s'_i, the value functions of $\mathcal{M}'$ and $\mathcal{M}$ are related by the equation $V(s'_i) = Q(s, a_i)$. Also note that by the way the tree was created, each level of the tree represents a $\max$ operation over its children.

Theorem 4.1: Any MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mathcal{D})$ with $|\mathcal{A}| = 2^N$ discrete actions can be transformed to another (mostly deterministic) MDP $\mathcal{M}' = (\mathcal{S}', \mathcal{A}', P', R', \gamma', \mathcal{D}')$, with $|\mathcal{S}'| = (2|\mathcal{A}| - 1)|\mathcal{S}|$ states and only $|\mathcal{A}'| = 2$ actions, which leads to the same total expected discounted reward.

Proof: The transformed MDP $\mathcal{M}'$ is constructed using the recursive decomposition described above. The new state space will include a full binary tree of depth $\log_2 |\mathcal{A}|$ for each state in $\mathcal{S}$. The number of states in each such binary tree is $(2^{N+1} - 1)$; 2^N leaf states —one for each action in $\mathcal{A}$— and $(2^N - 1)$ internal states. Therefore, the total number of states in $\mathcal{M}'$ must be $(2^{N+1} - 1)|\mathcal{S}| = (2|\mathcal{A}| - 1)|\mathcal{S}|$. The transformed MDP uses only two actions for making choices at the internal states; the original actions are hard-coded into the leaf states and need not be considered explicitly as actions, since there is no choice at leaf states. The transition model P' is deterministic at all internal states, as described above, and matches the original transition model P at all leaf states for the associated original state and action. The reward function R' is 0 and $\gamma' = 1$ for all transitions out of internal states, but R' matches the original reward function and $\gamma' = \gamma$ for all transitions out of leaf states[1]. Finally, $\mathcal{D}'$ matches $\mathcal{D}$ over the $|\mathcal{S}|$ root states and is 0 everywhere else. The optimal state value function V in the transformed MDP is trivially constructed from the optimal state-action value function Q of the original MDP. For $i = 1, \ldots, 2^N$, the value $V(s'_i)$ of each leaf state s'_i in the binary tree for state $s \in \mathcal{S}$, corresponding to action $a_i \in \mathcal{A}$, is trivially set to be equal to $Q(s, a_i)$ and the value of each internal state is set to be equal to the maximum of its two children. ■

The proposed action space decomposition can also be applied to arbitrary discrete or hybrid action spaces. If the number of actions is not a power of two, it merely means that some leaves will not be at the bottom level of the tree or equivalently the binary tree will not be full.

B. Action selection

Corollary 4.2: Selecting the maximizing action among $|\mathcal{A}|$ actions in the original MDP, requires $\mathcal{O}(\log_2 |\mathcal{A}|)$ comparisons in the transformed MDP.

Selecting the maximizing action is quite straightforward in the transformed MDP. Starting at the root of the tree, we compare the V-values of its two children and choose the largest (ties can be resolved arbitrarily). Once we reach a leaf, we have only one action choice. The action available to the i-th leaf in $\mathcal{M}'$ corresponds to action a_i in $\mathcal{M}$. Since this is a full binary tree with $|\mathcal{A}|$ leaves, its height will be $\log_2 |\mathcal{A}|$. The

[1]One could alternatively set the discount factor to $\gamma^{\frac{1}{\log_2 |\mathcal{A}|}}$ for all states. However, this choice makes the approximation problem harder, since nodes within the optimal path in the tree for the same state will have different values.

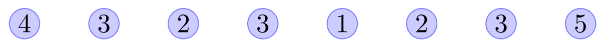

Fig. 1. The Q values for 8 available actions for some state.

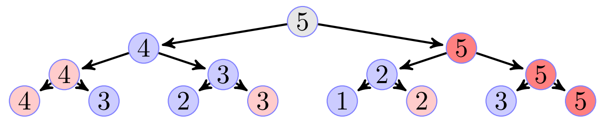

Fig. 2. An example binary action search tree for the state in Figure 1.

search requires one comparison per level of the tree, and thus the total number of comparisons will be $\mathcal{O}(\log_2|\mathcal{A}|)$. Notice that the value of the root of the tree is never queried and thus does not need to be explicitly stored.

To illustrate the transformation and action selection mechanism, Figure 1 shows an example with 8 actions along with their Q values for some state; finding the best action in this flat representation requires 7 comparisons. Figure 2 shows the action search tree in the transformed MDP for the same state; the 8 actions and the corresponding action values are now at the leaves. There are also 7 internal states along with their state values and the edges indicate the deterministic transitions from the root towards the leaves. Finding the best action in this representation requires only 3 comparisons.

C. Multidimensional action spaces

When the number of controlled variables increases, the number of actions among which the policy has to choose from grows exponentially. For example, in a domain with 4 controlled variables whose available action sets are $\mathcal{A}_0$, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$, the combined action space is $\mathcal{A} = \mathcal{A}_0 \times \mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$. If $|\mathcal{A}_0| = |\mathcal{A}_1| = |\mathcal{A}_2| = |\mathcal{A}_3| = 8$, then $|\mathcal{A}| = 4096$. The key observation is that there is no qualitative difference between this case and any other case where we have as many actions (e.g. one controlled variable with a fine resolution). Therefore, if we apply the transformation described earlier, with each one of the 4096 actions being a leaf in the transformed MDP, we will end up with a tree of depth 12. One convenient way to think about this transformation (that will help us when trying to pick a suitable approximator) is that each of the 4 controlled variables yields a binary tree of depth 3. On each "leaf" of the tree formed by the actions in $\mathcal{A}_0$, there is a tree formed by the actions in $\mathcal{A}_1$, and so forth[2]. Notice that while the number of leaves in the full tree is the same as the number of actions in the original MDP, the complexity of reaching a decision is once again exponentially smaller.

Corollary 4.3: The complexity (in the transformed MDP) of selecting the maximizing multidimensional action in the original MDP is linear in the number of action dimensions.

Consider an MDP with an N-dimensional action space $\mathcal{A} = \mathcal{A}_0 \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$. The number of comparisons required to select the maximizing action is:

$$\begin{aligned}\mathcal{O}(\log_2|\mathcal{A}|) &= \mathcal{O}(\log_2|\mathcal{A}_0 \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_N|) \\ &= \mathcal{O}(\log_2|\mathcal{A}_0| + \log_2|\mathcal{A}_1| + \cdots + \log_2|\mathcal{A}_N|)\end{aligned}$$

[2]Equivalently, one can interleave the partial decisions across the action variables in any desired schedule. However, it is important to keep the chosen schedule fixed throughout learning and acting, so that each action of the original MDP is reachable only through a unique search path.

D. Learning from samples

The transformation presented above provides a conceptual model of the space where the algorithm operates. However, there is no need to perform an explicit MDP transformation for deployment. Every sample of interaction (consistent with the original MDP) collected, online or offline, yields multiple samples (one per level of the corresponding tree) for the transformed MDP; the path in the tree can be extracted through a trivial deterministic procedure (binary search). Alternatively, the learner can interact directly with the tree in an online fashion, making binary decisions at each level and exporting action choices to the environment whenever a leaf is reached.

The careful reader will have noticed that, for every sample on the original MDP, we have $\log_2|\mathcal{A}|$ samples on the transformed MDP. This may raise some concern about the time required to learn a policy from samples, since the number of samples is now higher. A number of researchers have already noticed that the running time for many popular RL algorithms is dominated by the multiple $\max$ (policy lookup) operations at each iteration [14], which is further amplified as the number of actions increases. Our experiments have confirmed this, with learning being much faster on the transformed MDP. In fact, (unsurprisingly) for the algorithms tested, learning time increased logarithmically with the number of actions with our approach, while the expensive $\max$ operation quickly rendered the naive application of the same algorithms impractical.

E. Representation

One useful consequence of the transformed MDP's structure is that the V-value function is sufficient to perform action selection, without requiring a model. Each state in the original MDP corresponds to a tree in the transformed MDP. Starting at the root, a leaf can be reached by following the deterministic and known transitions of navigating through the tree. Once at the i-th leaf, there is only one available action, which corresponds to action a_i in the original MDP. Also notice that the value function of the root of the tree is never queried and thus does not need to be explicitly stored.

1) Exact representation:

Corollary 4.4: In the case of exact representation, the memory requirements of the transformed MDP are within a factor of 2 from the original.

In order to be able to select actions without a model in the original MDP, the Q-value function, which requires storing $|\mathcal{S}||\mathcal{A}|$ entries, is necessary. In the transformed MDP, the model of the tree is known, therefore storing the V-value function is sufficient. Since there are $|\mathcal{S}||\mathcal{A}|$ leaves and the number of internal nodes in a full binary tree is one less than the number of leaves, the V-value function requires storing less than $2|\mathcal{S}||\mathcal{A}|$ entries. Considering the significant gain in action selection speed, a factor-of-2 penalty in memory required for exact representation is a small price to pay.

2) Queries: When considering a deterministic greedy policy, most of $\mathcal{M}'$'s value function (and its corresponding state space) is never accessed. Consider a node whose right child has a higher value than the left child. Any node in the subtree below the left child will never be queried. Such a policy only ever queries $2|\mathcal{S}|\log_2|\mathcal{A}|$ values; those in the maximal path and their siblings. Of course, we don't know in advance which values these are, until we have the final value function. However, this observation provides some insight while considering approximate representation schemes.

3) Approximations: The most straightforward way to approximate the V-value function of $\mathcal{M}'$ would be to use one approximator per level of the tree. Since the number of values each approximator has to represent is halved every time we go up a level in the tree, the resources required (depending on our choice of approximator this could be the number of RBFs, the complexity of the constructed approximator trees, or the features selected by a feature selection algorithm) are within a factor of 2 of what would be required for approximating Q-values, just as in the exact case.

In practice, we've observed that a different approximator per level is not necessary. Using a single approximator, as we would if we only wanted to represent the leaves, and projecting all the other levels on that space (internal nodes in the tree will fall between leaves) seems to suffice. For example, for state s in $\mathcal{M}$, with $\mathcal{A} = \{1, 2, 3, 4\}$ we would have the leaves $s_1 = (s, 1)$, $s_2 = (s, 2)$, $s_3 = (s, 3)$, $s_4 = (s, 4)$. The nodes one level up would be $s_{12} = (s, 1.5)$ and $s_{34} = (s, 3.5)$ (remember that we don't need to store the root). The result is that each internal node ends up being projected between two leaf nodes.

An interesting observation is to see what happens when we are sampling actions uniformly (the probability that we reach a leaf for a particular state of the original MDP is uniform). The density of samples in each level of the tree is twice the density of the one below it. Since we have the same number of samples for each level of the tree and there are half the number of nodes on a level compared to the level below it, the density doubles each time we go up a level. For most approximators sample density acts as reweighing, therefore this approximation scheme assigns more weight to nodes higher up in the tree, where picking the right binary action is more important. Thus, in this manner we get a natural allocation of resources to parts of the action space that matter. As we will see from our experimental results, this scheme has proven to work very well in practice.

F. A practical action search implementation

A practical implementation for the general multidimensional case of the action search algorithm is provided in Figure 3. We are interested in dealing with action spaces where storing even a single instance of the tree in memory is infeasible. Thus, the search is guided by the binary decisions and relies on generating nodes on the fly based on the known structure (but not the values) of the tree. Note that while this is one implementation that complies with the exposition given above, it is not the only one possible.

V. Experimental Results

We tested our approach on two continuous-action domains. Training samples were collected in advance from "random episodes", that is, starting in a randomly perturbed state close to the equilibrium state and following a purely random policy. Each experiment was repeated 100 times for the entire horizontal axis to obtain average results and 95% confidence intervals over different sample sets. Each episode was allowed to run for a maximum of 3,000 and 30,000 steps for the pendulum and bicycle domains respectively, corresponding to 5 minutes of continuous balancing in real-time.

A. Inverted Pendulum

The inverted pendulum problem [15] requires balancing a pendulum of unknown length and mass at the upright position by applying forces to the cart it is attached to. The 2-dimensional continuous state space includes the vertical angle θ and the angular velocity $\dot{\theta}$ of the pendulum. The action space of the process is the range of forces in $[-50N, 50N]$, which in our case is approximated to an 8-bit resolution with 2^8 equally spaced actions (256 discrete actions). All actions are noisy (uniform noise in $[-10N, 10N]$ is added to the chosen action) and the transitions are governed by the nonlinear dynamics of the system [15]. Most RL researchers choose to approach this domain as an avoidance task, with zero reward, as long as the pendulum is above the horizontal configuration, and a negative reward, when the controller fails. Instead we chose to approach the problem as a more difficult regulation task, where we are not only interested in keeping the pendulum upright, but we want to do so while minimizing the amount of force we are using. Thus a reward of $1 - (u/50)^2$, was given, as long as $|\theta| \leq \pi/2$, and a reward of 0, as soon as $|\theta| > \pi/2$, which also signals the termination of the episode. The discount factor of the process was set to 0.98, and the control interval to 100msec.

In order to simplify the task of finding good features we used PCA on the state space of the original process[3] and kept only the first principal component pc. The state was subsequently augmented with the current action value u. The approximation architecture for representing the value function in this problem consisted of a total of 31 basis functions; a constant feature and 30 radial basis functions arranged in a 5×6 regular grid over the state-action space:

$$\left(1, e^{-\frac{\sqrt{\left(\frac{pc}{n_{pc}}-c_1\right)^2+\left(\frac{u}{n_u}-u_1\right)^2}}{2\sigma^2}}, \ldots, e^{-\frac{\sqrt{\left(\frac{pc}{n_{pc}}-c_3\right)^2+\left(\frac{u}{n_u}-u_3\right)^2}}{2\sigma^2}}\right)^\top$$

where the c_i's and u_i's are equally spaced in $[-1, 1]$, while $n_{pc} = 1.5$, $n_u = 50$ and $\sigma = 1$. Every transition in this domain corresponds to eight samples in the transformed domain, one per level of the corresponding tree.

Figure 4 shows the total accumulated reward as a function of the number of training episodes, when our algorithm is combined with Least-Squares Policy Iteration (LSPI) [16] and

[3]The two state variables (θ and $\dot{\theta}$) are highly correlated in this domain.

```
Action Search
  Input: state $s$, value function $V$, resolution bits vector $N$, number of action variables $M$, vectors $a_{\min}$, $a_{\max}$ of action ranges
  Output: joint action vector $a$
  $a \leftarrow (a_{\max} + a_{\min})/2$                                  // initialize each action variable to the middle of its range
  for $j = 1$ to $M$                                                    // iterate over the action variables
    $\Delta \leftarrow \mathbf{0}$                                                         // initialize vector $\Delta$ of length $M$ to zeros
    $\Delta(j) \leftarrow \Big(a_{\max}(j) - a_{\min}(j)\Big)\frac{2^{N(j)-1}}{(2^{N(j)}-1)}$     // set the step size $\Delta$ for the current action variable
    for $i = 1$ to $N(j)$                                              // for all resolution bits of this variable
      $\Delta \leftarrow \Delta/2$                                                  // halve the step size
      if $V(s, a - \Delta) > V(s, a + \Delta)$                          // compare the two children
        $a \leftarrow a - \Delta$                                                  // go to the left subtree
      else
        $a \leftarrow a + \Delta$                                                  // go to the right subtree
    end for
  end for
  return $a$
```

Fig. 3. A practical implementation of the action search algorithm

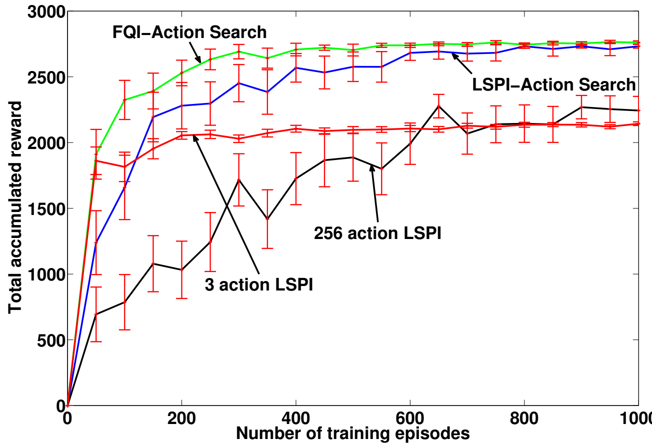

Fig. 4. Total accumulated reward versus training episodes for the inverted pendulum regulation task. The green and blue lines represent the performance of action search when combined with FQI and LSPI respectively, while the red and black lines represent the performance of 3 and 256-action controllers learned with LSPI and evaluated for every possible action at each step.

Fitted-Q iteration (FQI) [14]. For comparison purposes, we show the performance of a combined state-action approximator using the same set of basis functions, learned using LSPI and evaluated at each step over all 256 actions. We chose this approach as our basis for comparison, because it represents an upper bound on the performance attainable by algorithms that learn a combined state-action approximator and approximate the $\max$ operator. In order to highlight the importance of having continuous actions, we also show the performance achieved by a discrete 3-action controller learned with LSPI.

It should come as no surprise that we are able to outperform the discrete controller, when the number of samples is large, since the reward of the problem is such that it requires fine control. What is more interesting is that on the one hand, learning in the transformed MDP appears to be as fast as learning the discrete controller, achieving good performance with few training episodes and on the other hand the performance achieved is at least as good (and in fact in this case even better), as learning a combined state-action approximator and evaluating it over all possible actions in order to find the best action. We believe that the reason the naive combined state-action approximator does not perform very well, is that the highly non-linear dynamics of the domain give little opportunity for generalizing across neighboring actions with a restricted set of features. While we don't expect to always outperform the combined state-action approximator, the fact that we are able to have comparable performance with only a fraction of the computational effort (8 versus 255 comparisons per step) is very encouraging.

B. Bicycle Balancing

The bicycle balancing problem [14], has four state variables (angle θ and angular velocity $\dot{\theta}$ of the handlebar and angle ω and angular velocity $\dot{\omega}$ of the bicycle relative to the ground). The action space is two dimensional and it consists of the torque applied to the handlebar $\tau \in [-2, +2]$ and the displacement of the rider $d \in [-0.02, +0.02]$. The goal is to prevent the bicycle from falling, while moving at constant velocity.

Once again we approached the problem as a regulation task, rewarding the controller for keeping the bicycle as close to the upright position as possible. A reward of $1 - |\omega|(\pi/15)$, was given, as long as $|\omega| \le \pi/15$, and a reward of 0, as soon as $|\omega| > \pi/15$, which also signals the termination of the episode. The discount factor of the process was set to 0.9, the control interval was set to 10msec and training trajectories were truncated after 20 steps. Uniform noise in $[-0.02, +0.02]$ was added to the displacement component of each action.

As with the pendulum problem, after doing PCA on the original state space and keeping the first principal component, the state was augmented with the current action values. The approximation architecture consisted of a total of 28 basis functions; a constant feature and 27 radial basis functions arranged in a $3 \times 3 \times 3$ regular grid over the state-action space with $n_{pc} = 2/3, n_d = 0.02, n_\tau = 2$ and $\sigma = 1$.

Using 8-bit resolution for each action variable we have 2^{16}

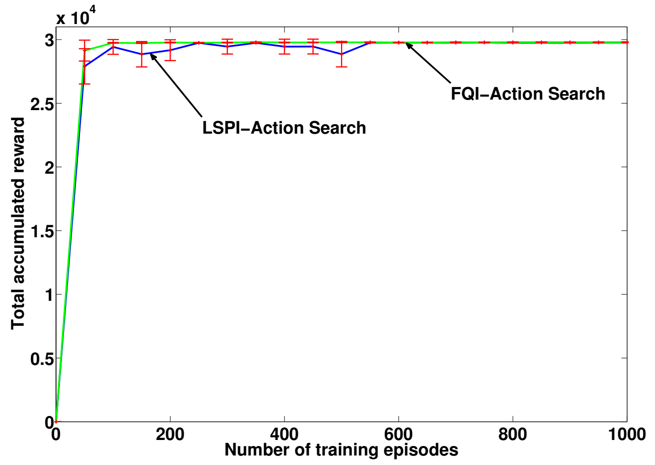

Fig. 5. Total accumulated reward versus training episodes for the bicycle balancing task using action search combined with FQI and LSPI.

(65,536) discrete actions, which brings us well beyond the reach of exhaustive enumeration. With the approach presented in this paper we can reach a decision in just 16 comparisons. Figure 5 shows the total accumulated reward as a function of the number of training episodes.

VI. Conclusion and future work

In this paper we have presented an effective approach for efficiently learning and acting in domains with continuous and/or multidimensional control variables. The problem of generalizing among actions is transformed to a problem of generalizing among states in an equivalent MDP, where action selection is trivial. There are two main advantages to this approach. Firstly, the transformation allows leveraging all the research devoted to effective generalization over states, to generalize across both states and actions. Secondly, action selection becomes exponentially faster, speeding up policy execution, as well as learning when the learner needs to query the current policy at each step (such as in policy iteration). In addition, we have shown that the representation complexity of the transformed MDP is within a factor of 2 from the original and the learning problem is not fundamentally more difficult. As discussed in Section IV-E, only a small subset of the value function is accessed during policy execution. Future work will investigate whether an exponential reduction in representation complexity over Q-value functions can be achieved as well.

This paper assumes that learning in continuous-state MDPs with binary actions is a solved problem. Unfortunately, the performance of current algorithms quickly degrades as the dimensionality of the state space grows. The action variables of the original problem appear as state variables in the transformed MDP, therefore the number of state variables, quickly becomes the limiting factor. Oftentimes the choice of features is more critical than the learning algorithm itself. As the dimensionality of the state space grows, picking features by hand is no longer an option. Combining action search with popular feature selection algorithms and investigating the particularities of feature selection on the state space of the transformed MDP is a natural next step.

Our approach effectively answers the question of how to select among a large number of actions, which is the case with continuous and/or multidimensional control variables. There are, however, a number of questions we do not address. We use an "off-the-self" learner and approximator as a black box. It would be interesting to investigate whether the unique structure of the transformed MDP offers advantages to certain learning algorithms and approximation architectures. Finally, while we have used batch learning algorithms, our scheme could also be used in an online setting. An interesting future research direction is investigating how we can exploit properties of the transformed MDP to guide exploration.

Acknowledgment

The authors are grateful to Ron Parr for helpful discussions.

References

[1] J. Pazis and M. Lagoudakis, "Binary action search for learning continuous-action control policies," in *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009, pp. 793–800.
[2] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.
[3] D. P. de Farias and B. V. Roy, "On constraint sampling in the linear programming approach to approximate dynamic programming," *Mathematics of Operations Research*, vol. 29, pp. 462–478, 2004.
[4] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 1998.
[5] J. C. Santamaría, R. S. Sutton, and A. Ram, "Experiments with reinforcement learning in problems with continuous state and action spaces," *Adaptive Behavior*, vol. 6, pp. 163–218, 1998.
[6] B. Sallans and G. E. Hinton, "Reinforcement learning with factored states and actions," *Journal of Machine Learning Research*, vol. 5, pp. 1063–1088, 2004.
[7] H. Kimura, "Reinforcement learning in multi-dimensional state-action space using random rectangular coarse coding and Gibbs sampling," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007, pp. 88–95.
[8] A. Lazaric, M. Restelli, and A. Bonarini, "Reinforcement learning in continuous action spaces through sequential Monte Carlo methods," in *Advances in Neural Information Processing Systems (NIPS) 20*, 2008, pp. 833–840.
[9] J. D. R. Millán, D. Posenato, and E. Dedieu, "Continuous-action Q-learning," *Machine Learning*, vol. 49, no. 2-3, pp. 247–265, 2002.
[10] J. A. Martín H. and J. de Lope, "`Ex<a>`: An effective algorithm for continuous actions reinforcement learning problems," in *Proceedings of the 35th Annual Conference of IEEE on Industrial Electronics*, 2009, pp. 2063–2068.
[11] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006, pp. 2219–2225.
[12] M. Riedmiller, "Application of a self-learning controller with continuous control signals based on the DOE-approach," in *Proceedings of the European Symposium on Neural Networks*, 1997, pp. 237–242.
[13] J. Pazis and M. G. Lagoudakis, "Learning continuous-action control policies," in *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2009, pp. 169–176.
[14] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.
[15] H. Wang, K. Tanaka, and M. Griffin, "An approach to fuzzy control of nonlinear systems: Stability and design issues," *IEEE Transactions on Fuzzy Systems*, vol. 4, no. 1, pp. 14–23, 1996.
[16] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.