Generalized Value Functions for Large Action Sets

Jason Pazis JPAZIS@CS.DUKE.EDU
Ronald Parr PARR@CS.DUKE.EDU
Department of Computer Science, Duke University, Durham, NC 27708 USA

Abstract

The majority of value function approximation based reinforcement 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 a unified view of V and Q functions and arrive at a new space-efficient representation, where action selection can be done exponentially faster, without the use of a model. We then describe how to calculate this new value function efficiently via approximate linear programming and provide experimental results that demonstrate the effectiveness of the proposed approach.

1. Introduction and motivation

One of the most basic decisions researchers are presented with in value function approximation for Markov decision processes is whether to use the state (V) or state-action (Q) value function. Model based approaches commonly use the V function, while model free approaches universally use the Q function. While there has been extensive research on the properties of these two types of value functions, there seems to be an assumption that these two approaches are fundamentally different and the only ones that are useful. In this paper we challenge both of these assumptions.

The first part of this paper demonstrates that V and Q functions are just the two extremes of a large family of value functions and shows how to approximate any value function in this family, using exact or approximate linear programming. The second part investigates the properties of a particular instantiation of this family termed the H-value function. The H function is a new space-efficient representation, where action selection can be done exponentially faster than in V or Q functions, without the use of a model. Finally we provide experimental results that demonstrate the effectiveness of the proposed approach.

2. Background

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, 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 a 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. The goal of the decision maker is to find an optimal policy π^* for choosing actions, which yields the optimal value function $V^*(s)$, defined recursively via the Bellman optimality equation:

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')$$

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. Once again our goal is to find an optimal policy π^* for choosing actions, which yields the optimal value function $Q^*(s,a)$:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

Appearing in *Proceedings of the 28^{th} International Conference on Machine Learning*, Bellevue, WA, USA, 2011. Copyright 2011 by the author(s)/owner(s).

2.1. Solving MDPs exactly via linear programming

One of the popular ways to solve for the optimal value function V^* is via linear programming. We can model finding the optimal value function as a minimization problem, where every state $s \in \mathcal{S}$ is a variable and the objective is to minimize the sum of the states' values under the constraints that the value of each state must be greater than or equal to the reward for taking any action at that state, plus the discounted value of the expected next states.

$$\begin{aligned}&minimize \sum_s V^*(s)\\&subject\ to:\\&(\forall s,a) V^*(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')\end{aligned}$$

Extracting the policy is fairly easy (at least conceptually), by looking at the dual variables and picking the action that corresponds to the non-zero variable for the state in question, or equivalently picking the action that corresponds to the constraint that has no slack in the current state.

2.2. Approximate linear programming (ALP)

In many real world applications the number of states is too large (or even infinite if the state space is continuous), rendering exact representation intractable. In those cases we approximate the value function via a linear combination of (possibly non-linear) basis functions or features. The variables in the approximate linear program are now the weights assigned to each basis function and the value of each state is computed as $\phi(s)^T w$, where $\phi(s)$ is the feature vector for that state, while w is the weight vector. The linear program becomes:

$$\begin{aligned}&minimize \sum_s \phi^T(s)w\\&subject\ to:\\&(\forall s,a)\ \phi^T(s)w \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)\phi^T(s')w.\end{aligned}$$

Using features dramatically reduces the number of variables in our program, however it does not reduce the number of constraints. Since the number of constraints is larger than the number of variables in the exact linear program, we have to find a way to reduce the number of constraints as well. Making certain assumptions over our sampling distribution (De Farias & Van Roy, 2004), or if we incorporate regularization (Petrik et al., 2010), we can sample constraints and bound the probability that we will violate a non-sampled constraint, or bound the performance degradation that will occur as a result of missing constraints.

Unfortunately this approximate formulation does not allow for easy extraction of a policy from the dual. Not only is the number of dual variables large (the same as the number of samples) but it does not offer a straightforward way to generalize to unseen states.

2.3. Reinforcement learning

In reinforcement learning, a learner interacts with a stochastic process modeled as an MDP and typically observes the state and immediate reward at every step; however the transition model P and the reward function R are not accessible. The goal is to 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.

3. Value functions

The choice of whether to use V or Q functions depends both on the problem and on the method of approximating the value function. In approximate linear programming, researchers universally use the V function.

In a number of disciplines the value of a state is an interesting quantity in itself, making the V function a natural choice. On the other hand, in the kinds of problems addressed herein (such as controlling a dynamic system in realtime), the value function is only useful as long as it can provide information about how to act, for which traditional V functions are often not sufficient absent a model. One workaround is to try to learn a model of the environment as well and use that model to predict the effects of actions, computing the best action with the help of the V function. Of course this has the added burden of approximating and repeatedly evaluating a model along with the value function. The second problem with V functions is that even with a model, when the number of actions available is too large, the computational cost of evaluating each one and picking the max can be prohibitive. This, once again, is a big problem especially in realtime and/or embedded applications.

Q functions solve the first problem, at the expense of increased representation complexity. Since the Q function is defined over states and actions, the dimension of the approximator increases, which can increase sample complexity. Additionally, while the space complexity to store the (exact) V function is $|S|$, the Q function requires $|S||A|$ space. Finally Q functions don't provide a solution to the second problem. One still must search over all actions exhaustively to pick the best one at each decision step.

3.1. A unified view of value functions

We can view V and Q functions as the two extremes of value function representation. A V function represents the value of the action that maximizes the total expected discounted reward for any given state. In contrast the Q function represents the value of any action for every state. Although these two extremes have been studied extensively, to our knowledge, no work has investigated the use of schemes that represent more information than a V function but less than a Q function.

Consider a value function that partitions the action space into sets and represents only the max value for each set at each state. In the case of the V function there is only one set containing all the actions. In contrast, in the Q function representation there is one set per action. Varying the number of sets and the number of actions per set specifies an entire family of value functions.

The following observation will be useful: While each action should belong to at least one set, there is no requirement for each action to appear in only one set. This suggests redundant representation schemes, where all (or some) actions appear in multiple sets; i.e., the sets should be jointly exhaustive but not necessarily disjoint.

3.2. Generalizing the Bellman optimality equation

For a value function X with action sets $u_i \subseteq A$ we can generalize the Bellman optimality equation as follows:

$$\begin{aligned}&\forall u \in \{u_1, \ldots, u_n\},\\ &X_u^*(s) = \max_{a \in u} R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_u X_u^*(s')\end{aligned}$$

In the case of the V function the innermost $\max$ is unnecessary since there is only one set, while for the Q function the outermost $\max$ in not necessary since there is only one action per set. Note that if the sets u_i are not disjoint we may need to examine only a subset of the action sets to evaluate the innermost $\max$ operator.

3.3. Solving for the generalized Bellman equation via linear programming

The following linear program solves for the generalized Bellman equation:

$$\begin{aligned}&minimize \sum_u \sum_s X_u^*(s)\\ &subject\ to:\\ &\forall u \in U, \forall (s,a) \in (S,u),\\ &\quad X_u^*(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')\\ &\forall (u,s) \in (U,S),\ V^*(s) \geq X_u^*(s)\end{aligned}$$

Here U is the set of action sets u, with the property that $u_1 \cup u_2 \cup \cdots \cup u_n = A$.

Note that this is not the only way to solve for the generalized Bellman equation. The first alternative is to solve for V^* first, and use it as a constant in the original problem. In that case, for n action sets, the original LP decomposes into n independent LPs (one for each $u \in U$) of the form:

$$\begin{aligned}&minimize \sum_s X_u^*(s)\\ &subject\ to:\\ &\forall (s,a) \in (S, A_u)\\ &\quad X_u^*(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')\end{aligned}$$

For the exact case these LPs are trivial to solve. Everything on the right hand side of the inequalities is already known (and thus constant), while there are no constraints between the values of the variables. In the solution, one constraint per set will hold with equality (the one with the largest right hand side), thus we don't even need an LP solver to find the X^* values.

If a subset of the action sets completely covers the action space, we can solve the original problem for this subset, as in section 3.2, extract V^*, and then solve the trivial LPs as above.

3.3.1. The approximate case

The ALP formulation is very similar to the exact formulation. The only difference is that the X variables will be replaced by a set of basis functions times their respective weights $\phi_u^T(s)w$ and we'll have one V variable per state/sample. Again, once we know $V(s)$ we can use it to solve a number of independent and much simpler programs, where everything on the right hand side of the inequalities is already known (and thus constant). However, assuming we have more samples than features, not all constraints are going to hold with equality, so this time we do need to treat the problem as an LP.

4. The H-value function

The remainder of the paper focuses on a particular instantiation of the value function family presented above, with $2\log_2|A|$ sets and $|A|/2$ actions per set.

Without loss of generality assume that we have $|A| = 2^d$ actions, each one corresponding to a vertex of an d-dimensional hypercube[1] (hence the name H function). Ev-

[1] If the number of actions is not a power of 2 we can simply assign some action(s) to multiple vertices of the hypercube.

ery dimension i of the hypercube partitions all the vertices into two sets: the ones that have coordinate i equal to zero and the ones that have coordinate i equal to one, called i_0 and i_1 respectively. Consider a value function that is composed of the $2d$ sets formed by the pairs over each dimension of the hypercube. This representation requires $2|S|\log_2|A|$ space to store in the exact case, and permits finding the maximizing action in $\log_2|A| = d$ comparisons if the maximizing action is unique.

Comparing the value of i_0 and i_1 for a given state reveals in which half of the hypercube the optimal action lies, essentially fixing one of the d coordinates. By performing this comparison over all pairs corresponding to the same dimension of the hypercube, we can narrow the set to a single vertex, corresponding to a single action. Figure 1 shows a graphical example for the trivial case of 3 dimensions.

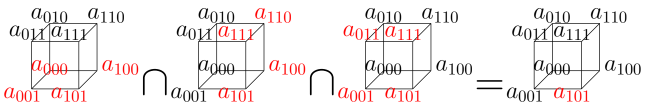

a_{010} a_{110} a_{011} a_{111} a_{000} a_{100} a_{001} a_{101}
$\bigcap$
a_{010} a_{110} a_{011} a_{111} a_{000} a_{100} a_{001} a_{101}
$\bigcap$
a_{010} a_{110} a_{011} a_{111} a_{000} a_{100} a_{001} a_{101}
$=$
a_{010} a_{110} a_{011} a_{111} a_{000} a_{100} a_{001} a_{101}

Figure 1. Example of finding the maximizing action over a 3-dimensional hypercube. Actions in red represent the actions in the set with the highest value.

For the H function the Bellman optimality equation can be expressed as follows, for $x \in \{0,1\}$:

$$H^*_{ix}(s) = \max_{a\in A_{ix}} R(s,a) + \gamma \sum_{s'} P(s'|s,a)\max\{H^*_{i0}(s'), H^*_{i1}(s')\}$$

Even though the number of vertices per facet A_{ix} grows linearly with the number of actions, the number of facets and thus the amount of information we need to store, grows logarithmically with the number of vertices/actions. In multidimensional action spaces, where the number of actions grows exponentially with the number of dimensions, the amount of information we need to store grows linearly with the number dimensions.

Using the H function speeds-up policy lookup immensely when compared to the V or Q functions, however it does not make the planning step any faster. If one wants to find the exact value function H^*, one has to consider every possible action. Of course as with the V function one should be able to find a reasonable approximation by sampling only a subset of the actions.

4.1. LP formulation

The H function can be formulated as a set of d linear programs that can be solved independently. Each LP will have the following form:

$$\begin{aligned}
&minimize \sum_s H^*_{i0}(s) + \sum_s H^*_{i1}(s)\\
&subject\ to:\\
&\forall(s,a) \in (S, A_{i0})\\
&\quad H^*_{i0}(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')\\
&\forall(s,a) \in (S, A_{i1})\\
&\quad H^*_{i1}(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')\\
&\forall(s \in S)\ V^*(s) \geq H^*_{i0}(s)\\
&\forall(s \in S)\ V^*(s) \geq H^*_{i1}(s)
\end{aligned}$$

where A_{i0} and A_{i1} are the sets of actions having their i-th coordinate 0 or 1 respectively and similarly for H^*_{i0} and H^*_{i1}.

For the approximate case the only difference is that the H variables will be replaced by a set of weights, applied to a set of basis functions; there will be one V variable per state/sample.

As was explained in section 3.3, once we have $V(s)$ we can break each of the linear programs above in two trivial LPs, one for $x = 0$, and another for $x = 1$:

$$\begin{aligned}
&minimize \sum_s H^*_{ix}(s)\\
&subject\ to:\\
&\forall(s,a) \in (S, A_{ix}),\\
&\quad H^*_{ix}(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')
\end{aligned}$$

4.2. Avoiding degenerate solutions

The exposition above made the assumption that there exists exactly one maximizing action for every state. If we relax that assumption, we can arrive at solutions where for some states the H function is not enough to determine a maximizing action uniquely. We can use a simple trick to avoid this degeneracy with probability 1. We will add a random constant $\epsilon \in (0, \epsilon_{\max})$, to every reward $R(s,a)$. It is easy to see that in the modified program, the probability of having two actions have the same value tends to 0 with high probability.

4.3. Parallel evaluation

Notice that once the H function is known, determining which side of the hypercube has the action with the highest value, when at state s, can be evaluated for all dimensions in parallel. Even in a single agent environment, almost linear evaluation speedup is possible by spreading the computation over different processors. Perhaps even more importantly, if different (groups of) dimensions of the hypercube correspond to different agents in the world, they can make decisions and act without the need for communication, as long as they all have access to the global state s.

4.4. Embedding continuous spaces on a hypercube

There are two cases that can result in a large number of actions. The first is when we have many action variables. If these action variables are binary, then mapping one action variable per dimension of the hypercube is an obvious solution. The second case is when we have a continuous action space that has been finely discretized, yielding a large number of actions. In this case mapping actions to dimensions of the hypercube may not be straightforward.

Consider a domain that requires controlling a motor using 8-bit PWM [2] (all zeros corresponding to 0% duty cycle and all ones corresponding to 100% duty cycle). The most obvious solution would be to map one dimension per bit of the PWM register. Notice, however, what happens when switching from action 01111111 to action 10000000 (49.8% to 50.2% duty cycle). All of the action variables must change at the same time. If due to approximation errors, or insufficient sampling, the most significant bit fails to change we can end up with an action that is very far from the optimal. On the other hand, in continuous spaces we usually assume that actions that are nearby have similar values. A way to remedy this problem is to switch to using Gray codes (Gray, 1953). In Gray codes, all unit increments in magnitude correspond to exactly one bit changes in representation, leading to much greater robustness to single or even multiple bit errors. In our experiments Gray codes performed significantly better than the naive mapping.

Two general design principles can help mapping action sets to faces of the hypercube. First, if it is possible to identify subsets of actions with differences in value that will be larger than anticipated function approximation errors, it is useful to map them to opposing faces of the hypercube. Second, if errors are unavoidable, then it can be useful to arrange the actions such that small numbers of edge traversals in the hypercube correspond to small changes in the action selected, thereby mitigating the effects of errors.

4.5. Enhanced Error Robustness

Gray codes are a natural way to increase robustness to errors, but their use assumes prior knowledge of an action space in which small changes in action do not correspond to big changes in value. If this prior knowledge is not available, or if its not enough to achieve the desired performance, additional measures may be needed to ensure robustness. In these cases, we can use redundant bits, similar to error-correcting codes (Clark & Cain, 1981), to detect and correct errors. In telecommunication and information theory, this is called forward error correction (FEC) or channel coding. For our hypercube representation this amounts to adding redundant dimensions and having multiple neighboring vertices correspond to the same action. Error correcting codes have already been exploited in supervised learning (Allwein et al., 2001).

5. Related Work

Extensive literature exists in the mathematical programming and operations research communities dealing with problems having many and/or continuous control variables. Unfortunately most results are not very well suited for reinforcement learning. 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 reinforcement learning community.

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.

The first and most commonly encountered category of methods for dealing with large action spaces uses a combined state-action approximator for the representation part (Santamaría et al., 1998), 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 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 (Lazaric et al., 2008), Gibbs sampling (Kimura, 2007), 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.

The second category deals predominantly with continuous (rather than multidimensional) control variables. The action space is discretized and a small number of different, discrete approximators are used for representing the value function (Millán et al., 2002). 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".

Policy gradient methods (Peters & Schaal, 2006) circum-

[2] Pulse Width Modulation.

vent 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. 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 used for learning a good policy from scratch, due to their sensitivity to local optima.

Another approach to dealing with large action spaces is factored value functions (Guestrin et al., 2002). The value function is assumed to be decomposable to a set of local value functions, with few interactions between different action variables. Action selection complexity is exponential to the induced width of the dependence graph.

The approach of Pazis and Lagoudakis (2011) transforms an MDP with a large number of actions, to an equivalent MDP with binary actions, where action complexity is cast to state complexity. The resulting tree shaped value function can easily by modeled as a collection of $\log |A| - 1$ value functions with set sizes being powers of two, ranging from the Q function up to but not including the V function. Specifically the last level is the Q function with $|A|$ sets, the one above it is the value function that represents the max over each pair of actions in Q, ($|A|/2$ sets) and so on. The time complexity for action selection is the same as for the H-function, however the space complexity is $2|S|(|A| - 1)$ in the exact case (roughly twice that of storing the Q function).

6. Experimental Results

In this section we use the H-value function to solve for the continuous action versions of three popular domains. In addition to the naive ALP formulation (H-ALP), we combine our approach with regularized approximate linear programming (Petrik et al., 2010) (H-RALP).

Regularized ALP (RALP) combines ALP with regularization of the weight vector of the solution. For L_1 norm, this corresponds to adding $O(k)$ constraints to the LP, where k is the number of weights in the linear value function approximation. The effect of L_1 regularization in RALP is similar to that of L_1 regularization in Lasso (Tibshirani, 1996) in that it produces sparse solutions and resistance to overfitting. In the case of ALP, overfitting has a slightly different interpretation than in regression. Overfitting is the result of an interaction between an expressive basis and constraint sampling. Without regularization, this combination can result in unbounded solutions, or bounded solutions where missing constraints allow the LP solver to find solutions with extremely low values at certain states.

For H-ALP we solve one linear program calculating the value of the first dimension, calculate an approximate V from that, and then solve two trivial LPs for each of the remaining dimensions. For H-RALP we first solve for the V-value function and then solve the resulting two LPs for each dimension using the same regularization parameter. While in both cases we may have been able to get better approximation by solving a new LP for each dimension separately, we chose to prioritize computational efficiency.

We also compare to the performance of RALP with the V-value function and the approach of Pazis and Lagoudakis (2011) with LSPI (P&L-2011). Since RALP is used as the first step in H-RALP, any error in the approximation of V is carried over to H. Thus when comparing the graphs for H-RALP and RALP, the difference can be seen as the price we pay in approximation performance, for the exponential increase in execution performance and the fact that we no longer need a model.

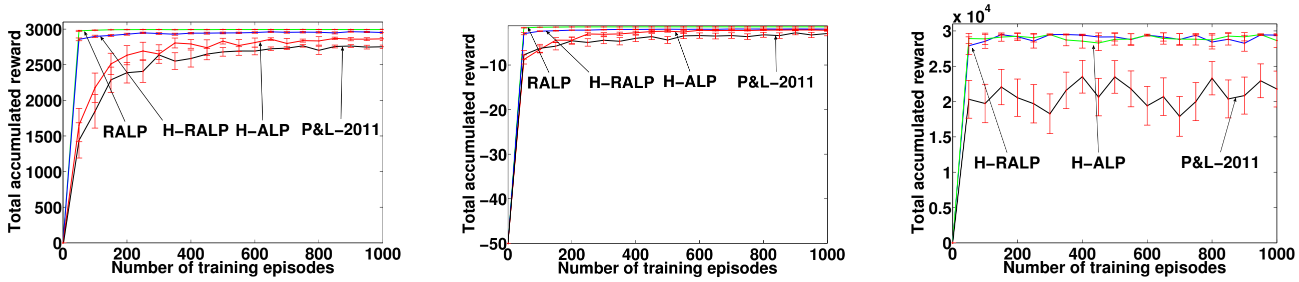
Cross validation was used to find a good value for the regularization parameter ψ for RALP and H-RALP. We should also note that all our samples come in the form of (s, a, r, s') samples; we only have one constraint per sampled state, which amounts to sampling the right hand side of the Bellman equation[3].

6.1. Inverted Pendulum

The inverted pendulum problem (Wang et al., 1996) requires balancing a pendulum of unknown length and mass at the upright position by applying forces to the cart to which it is attached. 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 a resolution of 2^8 equally spaced actions (256 discrete actions). Most researchers in reinforcement learning 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 100ms. To simplify the task of finding good features we standardized the state space and used PCA, keeping only the first principal component[4]. For H-ALP our

[3] All the domains tested in this paper are noiseless, thus there is only one possible next state for every state-action combination.

[4] Note that θ and $\dot{\theta}$ are highly correlated with only a small part (around the diagonal $\theta = a\dot{\theta}$) of the state space having any sam-



(a) (b) (c)

Figure 2. Total accumulated reward vs. training episodes: (a) inverted pendulum, (b) double integrator, (c) bicycle balancing. Graphs show averages and 95% confidence intervals over 100 independent runs.

features were a grid of 50 equally spaced Gaussian radial basis functions (RBFs) in $[-1.5, 1.5]$ with $\sigma = 10$ plus a constant feature. For RALP and H-RALP our features were a grid of 200 RBFs in $[-1.5, 1.5]$ with $\sigma = 1$ plus a constant feature, and the regularization parameter was set to $\psi = 100$. For P&L-2011 our features were a grid of 5×6 RBFs over the joint state-action space plus a constant feature.

Figure 2 (a) shows the total accumulated reward as a function of the number of training episodes. Training samples were collected in advance from “random episodes”, that is, starting the pendulum in a randomly perturbed state close to the equilibrium state $(0, 0)$ and following a purely random policy. Each episode was allowed to run for a maximum of 3,000 steps corresponding to 5 minutes of continuous balancing in real-time.

The figure demonstrates that all controllers perform well and regularization offers a significant performance advantage. Significantly, the penalty in performance for the exponential reduction in action selection cost (RALP to H-RALP) is minimal.

6.2. Double Integrator

The double integrator problem requires the control of a car moving on a one-dimensional flat terrain. The 2-dimensional continuous state space (p, v) includes the current position p and velocity v. The goal is to bring the car to the equilibrium state $(0, 0)$ by controlling the acceleration $a \in [-1, 1]$, under the constraints $|p| \leq 1$ and $|v| \leq 1$. The cost function $p^2 + a^2$ penalizes positions differing from the home position ($p = 0$), as well as large acceleration (action) values. The linear dynamics of the system are: $\dot{p} = v$ and $\dot{v} = a$. For H-ALP our features were a grid of 10×10 RBFs in $[-1.0, 1.0]$ with $\sigma = 1$ plus a constant feature. For RALP and H-RALP our features were a grid of 20×20 RBFs in $[-1.0, 1.0]$ with $\sigma = 1$ plus a constant feature and the regularization parameter was set to $\psi = 10^{1.5}$. For P&L-2011 we used a polynomial set of basis functions $(1, p, v, a, p^2a, v^2a, a^2, pv, pa, va, a^2p, a^2v)$.

Training samples were collected in advance from “random episodes” with a maximum length of 200 steps. 100 controllers were trained in each case and tested starting at state $(1, 0)$ (maximum allowed p, zero v) for a maximum of 200 steps. The discount factor of the process was set to 0.98 and the control interval to 100ms.

Figure 2 (b) shows the total accumulated reward as a function of the number of training episodes. Once again the penalty for not using the model is very modest.

6.3. Bicycle Balancing

The bicycle balancing problem (Ernst et al., 2005), has four state variables (angle θ and angular velocity $\dot{\theta}$ of the handlebar as well as angle ω and angular velocity $\dot{\omega}$ of the bicycle relative to the ground). The action space is 2D and 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.

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 $1 - |\omega| \times (\pi/15)$, was given, as long as $|\omega| \leq \pi/15$, 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 0.98, the control interval was set to 10ms and training trajectories were truncated after 20 steps. To simplify the task of finding good features we standardized the state space and used PCA, keeping only the first principal component. For H-ALP our features were a grid of 50 RBFs in $[-2.5, 2.5]$ with $\sigma = 10$ plus a constant feature. For H-RALP our features were a grid of 200 RBFs in $[-2.5, 2.5]$ with $\sigma = 1$ plus a constant feature, and the regularization parameter was set to $\psi = 0.1$. For P&L-2011 our features were a grid of $3 \times 3 \times 3$ RBFs over the joint state-action

ples at all. Using a regular grid over the original state space would result in unbounded or ill-conditioned programs for the unregularized ALP.

space plus a constant feature.

Using 8 resolution bits for each action variable we have 2^{16} (65,536) discrete actions, which brings us well beyond the reach of exhaustive enumeration (thus we were unable to obtain results using RALP with the V-value function). With the approach presented in this paper we can reach a decision in just 16 comparisons. Figure 2 (c) shows the total accumulated reward as a function of the number of training episodes. Each episode was allowed to run for a maximum of 30,000 steps, corresponding to 5 minutes of continuous balancing in real-time. In this case regularization didn't seem to offer an advantage, however with both H-ALP and H-RALP we achieve very good performance. H-function based controllers were almost always able to balance the bicycle for the entire time with as little as 50 training episodes. For our choice of discount factor P&L-2011 with LSPI did not perform competitively. For a lower discount factor (0.9), P&L performs on par with H-ALP.

7. Conclusion and future work

In this paper we have presented a unified view over V and Q functions and arrived at a new space-efficient representation, where action selection can be done exponentially faster, without the use of a model. We have described how to calculate this new value function efficiently via approximate linear programming and experimentally demonstrated the effectiveness of the proposed approach.

As mentioned in section 4.5 we can use redundant bits to improve our performance even in the presence of errors. Error correcting codes is a subject that has been extensively studied in the field of telecommunications systems as well as in the context of supervised learning. Capitalizing on that knowledge is an exciting direction for future research.

This paper assumes that learning the value function of an MDP is a solved problem. While for small numbers of state variables this is arguably true, the performance of current algorithms quickly degrades as the number of state variables grows. The number of state variables that our learning algorithm can handle, quickly becomes the limiting factor on the size of the problems we can negotiate. Oftentimes the problem isn't so much the algorithm that we are using, but our choice of features. As the dimensionality of the state space grows, picking features by hand is no longer an option. Although we have already used RALP in our experiments, investigating the particulars of feature selection for the H function is a natural next step.

Finally we should point out that while we have used batch-mode learning in our experiments, our scheme could also be used in an online setting. An interesting future research direction is investigating how we can exploit the properties of the H function to guide exploration.

Acknowledgments

We thank the reviewers for helpful comments and suggestions. This work was supported by NSF IIS-0713435. Opinions, findings, conclusions or recommendations herein are those of the authors and not necessarily those of NSF.

References

Allwein, E. L., Schapire, R. E., and Singer, Y. Reducing multiclass to binary: a unifying approach for margin classifiers. *JMLR*, 1:113–141, September 2001.

Clark, G.C. Jr. and Cain, J. B. *Error-Correction Coding for Digital Communications*. Springer, 1981.

De Farias, D.P. and Van Roy, B. On Constraint Sampling in the Linear Programming Approach to Approximate Dynamic Programming. *Mathematics of OR*, 29(3):462–478, 2004.

Ernst, D., Geurts, P., and Wehenkel, L. Tree-based batch mode reinforcement learning. *JMLR*, 6:503–556, 2005.

Gray, F. Pulse code communication, 1953. US Patent 2,632,058.

Guestrin, C., Lagoudakis, M., and Parr, R. Coordinated Reinforcement Learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 2002.

Kimura, H. Reinforcement learning in multi-dimensional state-action space using random rectangular coarse coding and gibbs sampling. In *Proceedings of the IEEE/RSJ Intl Conference on Intelligent Robots and Systems*, pp. 88 –95, 2007.

Lazaric, A., Restelli, M., and Bonarini, A. Reinforcement learning in continuous action spaces through sequential Monte Carlo methods. In *NIPS*, pp. 833–840. 2008.

Millán, J. del R., Posenato, D., and Dedieu, E. Continuous-action Q-learning. *Machine Learning*, 49(2-3):247–265, 2002.

Pazis, J. and Lagoudakis, M. Reinforcement learning in multi-dimensional continuous action spaces. In *Proceedings of the IEEE Intl Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 97–104, April 2011.

Peters, J. and Schaal, S. Policy gradient methods for robotics. In *Proceedings of the IEEE/RSJ Intl Conference on Intelligent Robots and Systems*, pp. 2219–2225, 2006.

Petrik, M., Taylor, G., Parr, R., and Zilberstein, S. Feature selection using regularization in approximate linear programs for markov decision processes. In *ICML*, pp. 871–878. Omnipress, June 2010.

Santamaría, J. C., Sutton, R. S., and Ram, A. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6:163–218, 1998.

Tibshirani, R. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

Wang, H., Tanaka, K., and Griffin, M. An approach to fuzzy control of nonlinear systems: Stability and design issues. *IEEE Trans. on Fuzzy Systems*, 4(1):14–23, 1996.