# Reinforcement Learning

# in Multidimensional Continuous Action Spaces

by

Jason Pazis

Department of Electronic and Computer Engineering
Technical University of Crete, Greece

Date: _____

Approved:

_____
Michail G. Lagoudakis, Supervisor

_____
Michalis Zervakis

_____
Nikos Vlassis

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Electronic and Computer Engineering
of the Technical University of Crete, Greece
2012

# Ενισχυτική Μάθηση

# σε Πολυδιάστατους Συνεχείς Χώρους Ενεργειών

Ιάσων Πάζης

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών
Πολυτεχνείο Κρήτης

Ημερομηνία: _____

Εγκρίθηκε:

_____
Μιχαήλ Γ. Λαγουδάκης, Επιβλέπων

_____
Μιχάλης Ζερβάκης

_____
Νικόλαος Βλάσσης

Διατριβή που παρεδόθη για να καλύψει μερικώς τις απαιτήσεις απόκτησης
του Μεταπτυχιακού Διπλώματος Ειδίκευσης
στο Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών
του Πολυτεχνείου Κρήτης
2012

# Abstract

The majority of 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. This thesis presents 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 Markov Decision Process (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. This 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. Results with two well-known reinforcement learning algorithms (Least-Squares Policy Iteration and Fitted-$Q$ Iteration) on three continuous action domains (1-dimensional inverted pendulum regulator, 1-dimensional double integrator, and 2-dimensional

bicycle balancing) demonstrate the viability and the potential of the proposed approach.

# Περίληψη

Η πλειονότητα των αλγορίθμων ενισχυτικής μάθησης που είναι διαθέσιμοι σήμερα εστιάζουν στην προσέγγιση της συνάρτησης αξιολόγησης ως προς τις καταστάσεις ($V$) ή ως προς τα ζεύγη καταστάσεων-ενεργειών ($Q$) και η αποτελεσματική επιλογή ενεργειών αφήνεται σε δεύτερη μοίρα. Από την άλλη πλευρά, τα προβλήματα του πραγματικού κόσμου τείνουν να έχουν μεγάλους χώρους ενεργειών, όπου η αξιολόγηση κάθε δυνατής ενέργειας καθίσταται ανέφικτη. Η αναντιστοιχία αυτή προβάλλει ένα σημαντικό εμπόδιο στην επιτυχή εφαρμογή ενισχυτικής μάθησης σε προβλήματα του πραγματικού κόσμου. Η παρούσα εργασία παρουσιάζει μια αποτελεσματική προσέγγιση στη πρόβλημα της μάθησης και της επιλογής ενεργειών σε πεδία με πολυδιάστατες ή/και συνεχείς μεταβλητές ελέγχου, όπου η αποδοτική επιλογή ενέργειας είναι ενσωματωμένη στη διαδικασία μάθησης. Αντί να μαθαίνει και να αναπαριστά κάποιος τη συνάρτηση αξιολόγησης ($V$ ή $Q$) της Μαρκωβιανής Διεργασίας Αποφάσεων (Markov Decision Process - MDP), μαθαίνει μια συνάρτηση αξιολόγησης πάνω σε μια μετασχηματισμένη (επαυξημένη) διεργασία, όπου οι καταστάσεις αντιπροσωπεύουν συλλογές ενεργειών της αρχικής διεργασίας και οι μεταβάσεις αντιπροσωπεύουν επιλογές εξάλειψης τμημάτων του χώρου ενεργειών σε κάθε βήμα. Η επιλογή ενέργειας στην αρχική διεργασία ανάγεται σε δυαδική αναζήτηση από τον πράκτορα στη μετασχηματισμένη διεργασία, με πολυπλοκότητα λογαριθμική ως προς το πλήθος των ενεργειών, ή ισοδύναμα γραμμική ως προς τις διαστάσεις του χώρου ενεργειών. Η μέθοδος αυτή μπορεί να συνδυαστεί με οποιονδήποτε αλγόριθμο ενισχυτικής μάθησης για διακριτές

ενέργειες για την εκμάθηση πολιτικών για πολυδιάστατους συνεχείς χώρους ενεργειών με τη χρήση προσέγγισης για τη συνάρτηση αξιολόγησης ως προς τις καταστάσεις στη μετασχηματισμένη διεργασία. Τα αποτελέσματα σε συνδυασμό με δύο γνωστούς αλγορίθμους ενισχυτικής μάθησης (Least-Squares Policy Iteration και Fitted-$Q$ Iteration) σε τρία πεδία με συνεχείς ενέργειες (1-dimensional inverted pendulum regulator, 1-dimensional double integrator, και 2-dimensional bicycle balancing) αναδεικνύουν τη βιωσιμότητα και τις προοπτικές της προτεινόμενης προσέγγισης.

# Contents

# List of Figures

# List of Abbreviations and Symbols

## Symbols

| | |
|---|---|
| $\pi$ | Policy |
| $\gamma$ | Discount factor $\in [0, 1)$. |
| $\mathcal{S}$ | State space |
| $\mathcal{A}$ | Action space |
| $\mathcal{P}$ | Transition model |
| $\mathcal{R}$ | Reward function |
| $\mathcal{D}$ | Initial state distribution |

## Abbreviations

| | |
|---|---|
| RL | Reinforcement Learning |
| MDP | Markov Decision Process |
| FQI | Fitted-$Q$ iteration |
| LSPI | Least Squares Policy iteration |
| RBF | Radial Basis Function |
| PCA | Principal Component Analysis |

# Acknowledgments

In more ways than one, the past few years have been a journey. While the research included in this thesis was conducted while I was at the Technical University of Crete, the write-up was completed while I was at Duke University. As such, people at both sides of the Atlantic have influenced me in different yet positive ways.

First and foremost, I need to thank Michail Lagoudakis, my advisor, who has had by far the most influence in my career and in every other aspect of my life so far. During my undergrad years he was the one who introduced me to Reinforcement Learning, and later became my advisor for my undergrad thesis. As for my graduate career, both as a Master's student at Technical University of Crete and as a PhD student at Duke University, I can confidently say that I owe exclusively to Michail, without whom I wouldn't even have applied, let alone admitted, for either. During my transition from one university to another, Michail has been more than just an advisor to me. He has been a true friend, whose help I'll never be able to repay.

On the other side of the Atlantic, I'd like to thank my current advisor, Ronald Parr, whose input has significantly changed my views on many topics, and has helped me become a much better researcher. His knowledge and critical thinking have been invaluable in evaluating and improving both new and old ideas.

Apart from my advisors, with whom I've had the most frequent interaction, there have been many other faculty members, on both sides of the Atlantic who have helped me not only directly with feedback on my ideas, but also indirectly by teaching me

principles on how to think about and approach problems and research in general.

Last but not least, I'd like to thank my friends. Both the ones in Hania, who have now graduated and are pursuing academic and professional careers of their own, and the ones at Duke, who are now in a very similar phase in their lives as I am. Thank you for being there for me, and for helping me through some of the most demanding times in my life.

# 1

# Introduction

Learning from experience. An ability at which humans excel, and has allowed us to progress much faster than other species. Until recently, learning from experience was an ability exclusive to living things. The last few years have seen more and more applications of machines learning from data. One very simple application, we are all familiar with, are spam filters. Using "spam"/"not spam" flags from users, spam filters learn how to separate useful from spam emails, relieving users from having to go through every email on their own.

Machine learning has found many applications besides spam filters. However, two common traits that most successful applications have in common, is that the learning algorithm has some source of positive and negative examples, and that they are "one-shot". By one-shot, we mean that it is generally assumed, that the decision of whether to flag a particular email as spam does not change the distribution of emails we'll receive in the future. This type of learning is commonly known as supervised learning.

An arguably much more interesting setting is that of sequential decision making without an explicit source of correct or incorrect behavior. It becomes even more

interesting (and realistic) when we assume that we have no, or very little, knowledge about the environment. Problems that fall under this category include problems as diverse as scheduling a fleet of delivery trucks, flying an autonomous helicopter, or even controlling the electric power distribution of an entire country. Reinforcement learning is a subfield of machine learning that deals with these types of problems.

Perhaps somewhat surprisingly, even though Reinforcement Learning (RL) deals with a class of very interesting problems, it has yet to find a killer application. This can be attributed to a number of things. First of all, the problems that RL tries to address are generally more complex than many other machine learning problems. Secondly, RL is still a relatively young field. Even though it has shown much promise in toy domains in laboratory settings, there are a still obstacles it needs to overcome before it becomes successful in the real world.

This thesis deals with one particular obstacle that seems to universally cripple RL algorithms and appears in almost all interesting real world domains: making decisions efficiently, when the number of actions available at each timestep is large. A large action space can be the result of a finely discretized continuous action space, or a multidimensional action space yielding many action combinations, or a combination thereof.

## 1.1 Contribution

This thesis addresses the problem of learning and acting in domains with multidimensional and/or continuous state-action spaces, when we can make no convexity or other strong simplifying assumptions about the shape of the value function. The approach presented yields a sample-efficient, constant-time, limited-resource algorithm, suitable for real-world embedded and/or high-performance applications.

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, a state value function over an implied augmented MDP is learned, 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 binary search in the transformed MDP, whose complexity is linear in the number of action dimensions. It is shown that the representational complexity of the transformed MDP is within a factor of 2 from that of the original MDP, without relying on any assumptions about the shapes of the action space and/or the value function.

## 1.2 Thesis outline

Chapter 2 provides the necessary background of the ideas this thesis builds upon. A brief introduction is given to Reinforcement Learning and Markov Decision Processes. Additionally, Fitted-$Q$ iteration, and Least Squares Policy Iteration, the Reinforcement Learning algorithms used in the experiments throughout this thesis, are summarized.

Chapter 3 motivates the need for continuous and multidimensional action spaces and explains why this is such a difficult problem to solve.

Chapter 4 is an overview of previous related work in the area, along with a discussion of the strengths and weaknesses of each type of approach.

Chapter 5 presents the approach advocated in this thesis, states a number of important properties and gives a concrete algorithm implementation.

Chapter 6 demonstrates the applicability of the proposed approach when coupled with modern reinforcement learning algorithms such as Fitted-$Q$ iteration, and Least Squares Policy Iteration, in the inverted pendulum, double integrator, and bicycle balancing domains.

Chapter 7 discusses the strengths and weaknesses of the approach, gives a number of guiding directions for future work, and concludes this thesis with a brief summary.

Finally, Appendix A provides an equivalent $Q$-value formulation, which can be used as an alternative to the $V$-value formulation presented in Chapter 5.

# 2

# Background

## 2.1 Agents and environments

An agent can be anything that has the ability to perceive some aspect(s) of its environment and act. The environment is what our agent perceives as the "outside" world. It can be the real world, a room, a simulated labyrinth, or even an actuator. Considering an actuator as part of the environment, rather than part of the agent, may at first seem unnatural. However, in the cases that we are interested in, the agent will usually have no prior knowledge about the results of its own actions. This is very similar to the behavior of a newborn baby, who initially knows nothing about controlling his/her own arms and legs. Thus, our agents will treat everything as part of the environment, even parts of their bodies, and use their observations to learn how to interact with it in a beneficial way.

Environments are in most cases stochastic. What this means for our agent, is that, even if everything in the environment is the same between two repetitions of an experiment (we are in the same state), an action may have different outcomes. The difference between outcomes can be sharp as in the (discrete) case of a coin toss

that may produce heads or tails, or it can be more subtle and fine-grained, as in the (continuous) case of a free running motor that rotates within a certain speed range when voltage is applied to its terminals.

## 2.2 Markov Decision Processes (MDP)

A Markov Decision Process (Puterman, 1994), is a discrete-time mathematical decision-making modeling framework, particularly useful when the outcome of a process is a function of the agent's actions perturbed by noise. MDPs have found extensive use in areas such as economics, control, manufacturing, and Reinforcement Learning.

An MDP is defined as a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where:

- $\mathcal{S}$ is the state space of the process. It can be finite set, or it can be infinite as is the case when there is some state variable that can take values in a continuous range. The current state $s$ is assumed to be a complete description of the state of the environment at the current timestep.

- $\mathcal{A}$ is the action space of the process. Just like the state space, it can be finite or infinite. The set of actions are the possible choices an agent has at each timestep.

- $\mathcal{P}$ is a Markovian transition model, where $P(s'|s, a)$ denotes the probability of a transition to state $s'$ when taking action $a$ in state $s$. The Markov property, implies that the probability of making a transition to state $s'$ when taking action $a$ in state $s$, depends only on the current $s$ and $a$ and not on the history of the process.

- $\mathcal{R}$ is the reward function (scalar real-valued) of the process. It is Markovian as well, and $\mathcal{R}(s, a, s')$ represents the expected immediate reward for any transition from $s$ to $s'$ under action $a$ at any timestep. The expected reward for a

state-action pair $(s, a)$, is defined as:

$$\mathcal{R}(s,a) = \sum_{s' \in S} \mathcal{P}(s'|s,a)\mathcal{R}(s,a,s') \ .$$

- $\gamma \in [0, 1)$ is the discount factor. It is a way to express the fact that we care more about rewards received now, than in the distant future. The reward we receive at the current timestep is weighted by 1, while future rewards are exponentially discounted by $\gamma^t$. In the extreme case where $\gamma = 0$, the problem degenerates to picking the action that will yield the largest immediate reward (supervised learning). As $\gamma$ gets closer to 1, we may sacrifice short term benefits to achieve higher rewards later.

- $\mathcal{D}$ is the initial state distribution. It describes the probability each state in $\mathcal{S}$ has to be the initial state. In some problems most states have a zero probability, while few states (possibly only one) are candidates for being an initial state.

## 2.3 Policies and value functions

A policy $\pi$ is a mapping from states to actions. It defines the response (which may be deterministic or stochastic) of an agent in the environment for any state it encounters and it is sufficient to completely determine its behavior. In that sense $\pi(s)$ is the action chosen in state $s$ by the agent following policy $\pi$.

An optimal policy $\pi^*$, also known as an "undominated optimal policy", is a policy that yields the highest expected utility in the long run. That is, it maximizes the expected total discounted reward under all conditions (over the entire state space). For every MDP there is at least one such policy although it may not be unique (multiple policies can be undominated; hence yielding equal expected total discounted reward through different actions).

The value $V_\pi(s)$ of a state $s$ under a policy $\pi$ is defined as the expected, total, discounted reward when the process begins in state $s$ and all decisions are made according to policy $\pi$:

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

The value $Q_\pi(s, a)$ of a state-action pair $(s, a)$ under a policy $\pi$ 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 $\pi$:

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

The goal of the decision maker is to find an optimal policy $\pi^*$ 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 $V_{\pi*}$ 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 $Q_{\pi*}$ is known, a greedy policy, which simply selects actions that maximize $Q_{\pi*}$ 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, a problem known as planning, from a (not too large) discrete MDP model.

## 2.4 Reinforcement Learning (RL)

In Reinforcement Learning (Kaelbling et al., 1996; Sutton and Barto, 1998; Russel and Norvig, 2003), a learner interacts with a stochastic process modeled as an MDP.

It is usually assumed that the agent knows nothing about how the environment works or what the results of its actions are (does not have access to the model $P$ and reward function $R$ of the underlying MDP). Additionally, in contrast to supervised learning, there is no teacher to provide samples of correct or bad behavior.

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.

Two related problems fall within Reinforcement Learning: *prediction* and *control*. In prediction problems, the goal is to learn to predict the total reward for a given fixed policy, whereas in control the agent tries to maximize the total reward by finding a good policy. These two problems are often seen together, when a prediction algorithm evaluates a policy and a control algorithm subsequently tries to improve it.

The learning setting is what characterizes the problem as a Reinforcement Learning problem. Any method that can successfully reach a solution, is considered as a Reinforcement Learning method. This means that very diverse algorithms coming from different backgrounds can be used; and that is indeed the case. Most of the approaches can be distinguished into Model-Based learning and Model-Free learning.

In Model-Based learning, the agent uses its experiences in order to learn a model of the process and then find a good decision policy through planning. Model-Free learning on the other hand, tries to learn a policy directly without the help of a model. Both approaches have their strengths and weaknesses (in terms of guarantee of convergence, speed of convergence, ability to plan ahead, use of resources).

## 2.5 Learning algorithms

In this section, we provide a brief introduction to two reinforcement learning algorithms used in the experiments. We should stress that while these learning algorithms are a popular choice, the methods presented in this thesis are designed so that they can be combined with any reinforcement learning algorithm beyond these two.

### *2.5.1 Fitted Q Iteration (FQI)*

Fitted-$Q$ Iteration (Ernst et al., 2005), is an approximate version of the value iteration algorithm using $Q$-values. It uses a batch of samples and a supervised learning (regression) algorithm in the inner loop, to successively approximate and improve the value function. Instead of trying to learn the $V$ value function as is typical for exact value iteration, FQI learns the $Q$ value function from samples, allowing it to perform a model-free maximization step for policy improvement at each iteration. One of its biggest strengths is that it can be combined with any linear or non-linear function approximator (regressor). In particular, it has been demonstrated to achieve excellent performance when combined with random forest type approximators, as well as neural networks (Riedmiller, 2005). Figure 2.1 summarizes the Fitted-$Q$ Iteration algorithm.

### *2.5.2 Least-Squares Policy Iteration (LSPI)*

Least-Squares Policy Iteration (Lagoudakis and Parr, 2003) is a reinforcement learning algorithm based on the approximate policy iteration framework, whereby at each iteration an improved policy is produced as the greedy policy over an approximation of the value function of the previous policy. LSPI uses linear approximation architectures consisting of a weighted sum of a set of basis functions $\phi$ for representing value functions. The value function of each policy is learned by solving a linear system formed using a set of samples from the process and the fixed-point property of the

**Fitted $Q$-Iteration** $(\mathcal{D},\ \gamma,\ N)$
  // $D$ : Source of samples $(s, a, r, s')$
  // $N$ : Number of iterations
  $k \leftarrow 0$
  $Q^k \leftarrow \mathbf{0}$
  **do**
    $TS \leftarrow \varnothing$
    **for each** $(s, a, r, s')$ in $\mathcal{D}$
      $TS \leftarrow TS \cup \left\{\left((s, a), (r + \gamma\ \max_{a' \in \mathcal{A}} Q^k(s', a'))\right)\right\}$
    $Q^{k+1} \leftarrow$ use regression on $TS$
    $k \leftarrow k + 1$
  **while** $(k < N)$
  **return** $Q^N$

FIGURE 2.1: The Fitted $Q$-Iteration algorithm.

value function under the Bellman equation. Although, there is no guarantee that improvement will be monotonic, it is guaranteed that the iteration will not diverge, and indeed in most cases it converges in a few iterations, otherwise it oscillates within a small region. LSPI is summarized in Figure 2.2.

**LSPI** $(\mathcal{D},\ \phi,\ \gamma,\ \epsilon)$
  // $D$ : Source of samples $(s, a, r, s')$
  // $\phi$ : Basis functions
  // $\gamma$ : Discount factor
  // $\epsilon$ : Stopping criterion
  $w' \leftarrow \mathbf{0}$
  **repeat**
    $w \leftarrow w'$
    $\mathbf{A} \leftarrow \mathbf{0}$
    $b \leftarrow \mathbf{0}$
    **for each** $(s, a, r, s')$ in $\mathcal{D}$
      $a' = \arg\max_{a'' \in \mathcal{A}} w^\top \phi(s', a'')$
      $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi(s', a')\Big)^\top$
      $b \leftarrow b + \phi(s, a)r$
    $w' \leftarrow \mathbf{A}^{-1}b$
  **until** $(\|w - w'\| < \epsilon)$
  **return** $w$

FIGURE 2.2: The Least Squares Policy Iteration (LSPI) algorithm.

# 3

# Problem Statement

## 3.1 Why are continuous actions necessary?

Most benchmark domains in reinforcement learning are modeled with discrete actions. It is disheartening that some researchers are so out of touch with the requirements of real-world applications, that they even claim that "discrete actions seem to work well enough, what is the point of continuous actions"[1].

Let's look at one of the most popular benchmarks for reinforcement learning, the inverted pendulum problem (Wang et al., 1996). It 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 $\theta$ and the angular velocity $\dot{\theta}$ of the pendulum.

Typically three actions are used: force of $-50$ ,50 and 0 newtons. The domain is approached 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 and the pendulum falls. In simulation, this scheme does indeed work fine. Unfortunately it is very far from any real world application's requirements.

[1] The author has heard this particular phrase more than once.

The first problem, is that the mechanical stresses placed on the drivetrain of a controller that only has such extreme actions available to it, would wear the setup in a very short amount of time. On the other hand, a controller with a much more fine-grained action set, would be able to use much lower force magnitudes when correcting for small deviations. In addition the resulting controller may be more stable, especially in noisy situations. A controller with only three actions often has to choose between overcompensating for a small deviation, or waiting another timestep till the accumulated deviation is larger. A continuous controller does not face this predicament. It can compensate just the right amount.

Finally, in a real-world application, we would care about much more than just staying balanced. For instance we would probably also care about not spending a huge amount of energy. Since force used is directly related to energy used, this would require much more fine-grained actions that can correct for deviations while minimizing applied forces.

## 3.2 Are continuous actions possible?

One could argue that whenever a digital circuit is controlling something in the real world, we can never have truly continuous actions. For example, to control the position of a servo, the voltage at the terminals of a motor, or even the brightness of a light emitting diode (LED) we would be using pulse width modulation (PWM) with the duty cycle controlled by an 8-12 bit register. In such a situation we don't really have a "continuous" action space, but a finely discretized one. However, these spaces are more than just a collection of $2^8$ to $2^{12}$ unrelated actions. They are structured, and nearby actions have similar outcomes. For our purposes we will call such a finely discretized action space continuous.

## 3.3 Multidimensional action spaces

Apart from a finely discretized continuous action space, another reason why we may have a large number of actions, is the presence of many action variables. Consider for example the task of controlling an autonomous helicopter. In that case, we'll have (at least) four control variables. The yaw, pitch and roll rates, as well as the collective pitch. If every control variable has an 8-bit resolution, the combined state action space will have $2^8 \times 2^8 \times 2^8 \times 2^8 = 2^{32}$ actions.

One approach sometimes used in such cases, is to model the problem as a multi-agent domain. The reasoning is that it may be easier to learn multiple simple controllers, than one more complex one. Unfortunately not only does such an approach present additional design difficulties, but it can also lead to highly suboptimal solutions.

The first difficulty with the multiagent approach, is that in order to decompose the action space, we will also need to decompose the state space and reward function to the relevant components for each controller. Depending on the problem this may be far from trivial.

The second and far more dangerous issue, is that if cooperation between the different agents is required to achieve good performance, an approach that has not accounted for that may end up yielding highly suboptimal solutions. On the other hand, an approach that accounts for communication between the agents to achieve a globally good solution, will likely be more complex than solving the original single agent problem.

## 3.4 Why are large action spaces a problem?

From the discussion above, we can see that both continuous and multidimensional action spaces essentially boil down to a large action space. This section explains

why large action spaces present a problem for common, value-function-based RL algorithms.

Suppose we are using $Q$ value functions. Then at every timestep, when asked to select an action, the controller has to perform a maximization, namely $\arg\max_a Q(s, a)$ where $s$ is the current state. In the helicopter example above, this would mean evaluating the value function at $2^{32}$ points and picking the maximum. This would present a problem even in simulation, let alone in a real embedded platform, where we would only have a few milliseconds to make a decision.

Similarly, if we were using the $V$ value function, in the example above, we would need to evaluate the reward and transition models as well as the value function $2^{32}$ times per timestep.

Additionally, computing the value function would be a problem as well. Algorithms such as (approximate) value and policy iteration, perform a maximization step at each iteration. This would mean that we would have to perform $O(2^{32})$ operations per sample point (state or state-action pair), per iteration.

It should be obvious by now that action selection in multidimensional and/or continuous action spaces, is not a trivial problem that can be solved by brute force methods. A more elegant solution is required. The next chapter presents some related work that has appeared in the literature.

# 4

# Related Work

## 4.1 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.

## 4.2 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, three main categories can be identified.

### *4.2.1 Combined state-action approximators*

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.

In the simplest case, if the value function is assumed to be convex given the state, and the approximator used is differentiable, finding the maximum is straightforward. While there are many large scale problems for which such an assumption is true, or at least approximately true, it is not the case in the kinds of control tasks we are primarily interested in this thesis.

In the more general case, the maximization step 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 immediately notice, that the aforementioned approaches don't have any significant difference from approaches in other communities where the maximization step is recognized as a non-linear maximization task, 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, and their optimized versions have made it into popular mathematical packages.

### *4.2.2 Action mixing*

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.

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, hence 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.

While these schemes allow the agent to develop preferences for actions that fall between approximators, it is unclear under what conditions they 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.

#### *4.2.3 Policy gradient*

Policy gradient methods (Peters and Schaal, 2006) 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. In addition very few of these algorithms come with any kind of useful guarantees about the quality of the solution.

## 4.3 Common approaches

In this section, I provide a brief description of what I believe is a representative sample of approaches from the categories above, that have appeared in the RL literature. This discussion does not, in any way, attempt to be complete. The goal is to highlight the similarities and differences between these approaches and provide my own (biased) view of their strengths and weaknesses.

Santamaría, Sutton, and Ram (Santamaría et al., 1998) provide one of the earliest successful 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 (Albus, 1975), 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 (Sallans and Hinton, 2004; Kimura, 2007). 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 (Lazaric et al., 2008). 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 (Millán et al., 2002), 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> (Martí•n H. and de Lope, 2009) 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.

## 4.4 Non-stardard approaches

This section provides a brief description of a number of approaches that do not fall into the categories presented above.

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.). The work most relevant to that of this thesis is that of Bertsekas and Tsitsiklis Bertsekas and Tsitsiklis (1996). In

section 6.1.4 of their book they introduce 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 (de Farias and Roy, 2004) as an MDP transformation encoding multidimensional action selection into a series of simpler action choices. This thesis could be considered as a concrete implementation of the above ideas in a model-free setting.

Pazis and Parr (2011a) present a new type of value function (called the H-value function) which can be used to select actions in time logarithmic to the number of actions, and also takes up space logarithmic to the number of actions. Unfortunately its construction is only straightforward when approximate linear programming is used as the learning method, which requires noise free samples. In addition, because it only tries to approximate the values of the best actions, it does not seem very well suited for online learning or exploration.

Finally, some methods exploit certain domain properties, such as temporal locality of actions (Riedmiller, 1997; Pazis and Lagoudakis, 2009b), modifying the current action by some small amount $\Delta$ 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 one.

# 5

# Action Search

This chapter presents the $V$-value formulation of action search, consistent with Pazis and Lagoudakis (2011). For an alternative $Q$-value formulation consistent with Pazis and Lagoudakis (2009a), see Appendix A.

As described in the previous chapter, 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. Even though this problem is far from being solved, there has been significant progress.

The second issue, which becomes apparent when the number of available actions becomes too large, is that selecting the right action using a reasonable amount of computation becomes non-trivial. 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. While many approaches offer a reasonable compromise between computational effort and accuracy in action selection, there is room for significant improvement.

## 5.1 Designing for efficient action selection

It should be apparent from the previous discussion that most approaches deal with the two problems separately. I 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 explored in this thesis. 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.

The problem of generalizing among actions is transformed to a problem of generalizing among states in an equivalent MDP, 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[1].

[1] Note that this speedup also affects the learning phase of algorithms such as LSPI and FQI. It has been observed that when the number of actions grows beyond a certain point, the time to perform the maximization step may dominate the execution time of the algorithm.

## 5.2 Intuition

Before we formally describe the algorithm and the MDP transformation, we should give some intuition behind our approach.

Consider an MDP with $2^N$ actions. In the absence of any other information, $2^N$ evaluations and comparisons are required to find the maximizing action. Even for moderate $N$ this can be a problem. What if we could decompose the original action space into two halves. Then finding the optimal action in each half would only require $2^{N-1}$ evaluations and comparisons. Of course, to achieve any speedup, we have to be able to efficiently select which half of the action space we are interested in, without evaluating all of them. We can do so by storing two additional values per state. The maximum value over the actions in each halfspace.

Unfortunately $2^{N-1}$ evaluations may still be quite expensive. However all we have to do, is perform the above decomposition recursively to each halfspace. By the $N$th decomposition step, we'll be left with singleton action spaces.

In the original action space, the question we were asking was: "which of these $2^N$ actions is the optimal?". In the new action space, the first question we ask is: "which half of the action space contains the optimal action?", followed by: "I know one of these two quarters of the action space contain the optimal action. Which one is it?". This process continues until we reach the bottom level, at which point we have narrowed our choices down to two potential actions, and the answer to the last query leaves us with a unique action.

## 5.3 MDP transformation

In this section we formally present the MDP transformation procedure, and state a number of important properties.

Consider an MDP $\mathcal{M}(\mathcal{S}, \mathcal{A}, P, R, \gamma, \mathcal{D})$, where at each state $s \in \mathcal{S}$ 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 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 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[2]. 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.

## 5.4 Action selection

**Corollary 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.*

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

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 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, figure 5.1 shows the decomposition steps for a state with 8 actions. Figure 5.2 shows the original and transformed value functions for the same state.

## 5.5 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 $\mathcal{A}_1$,

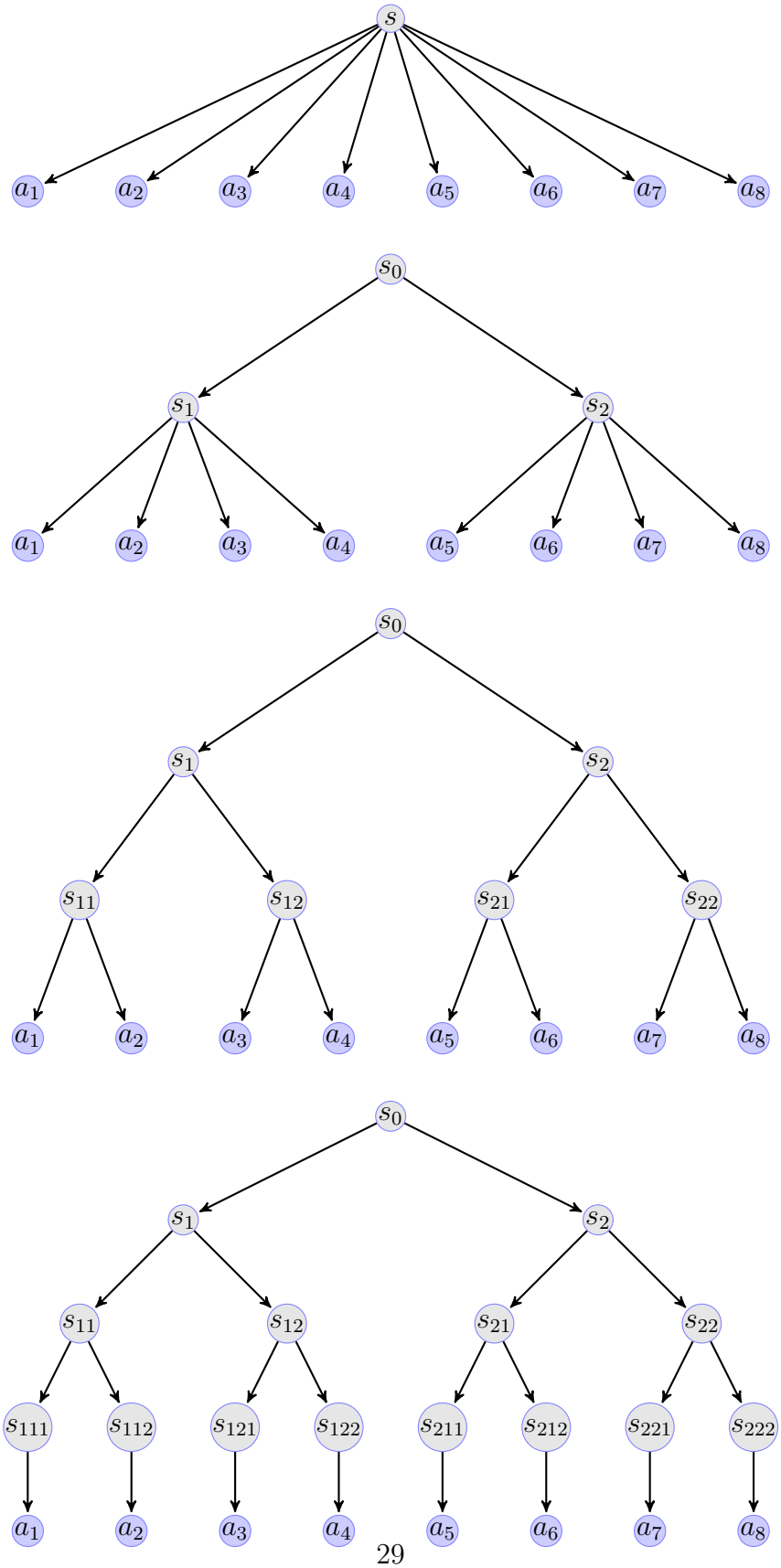




FIGURE 5.1: Decomposing a state with eight actions

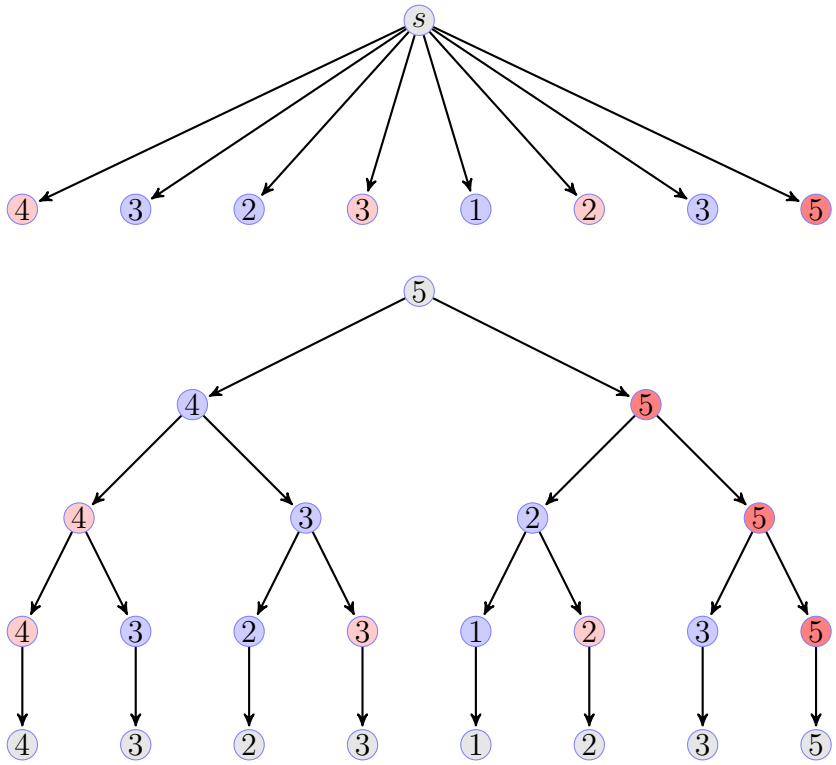


FIGURE 5.2: An example value function for the state in figure 5.1. States in gray do not need to be explicitly stored.

.

and so forth[3]. 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 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 maxi-

[3] 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.

mizing action is:

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

## 5.6 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 (Ernst et al., 2005), 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.

## 5.7 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.

### *5.7.1 Exact representation*

**Corollary 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.

### *5.7.2 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.

### *5.7.3 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 radial basis functions (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.

To our surprise, we've observed that a different approximator per level is not always necessary in practice. 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

**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
  $\Delta(j) \leftarrow \Big(a_{\max}(j) - a_{\min}(j)\Big) \dfrac{2^{N(j)-1}}{(2^{N(j)} - 1)}$ // 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** $a$

FIGURE 5.3: A practical implementation of the action search algorithm

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. Of course we don't expect this simplification to be possible in all situations. Its applicability will depend on our choice of domain and function approximator. However, as we will see from our experimental results, this scheme has proven to work very well in practice.

## 5.8 A practical action search implementation

A practical implementation for the general multidimensional case of the action search algorithm is provided in Figure 5.3. We are interested in dealing with action spaces where we are unable to store even a single instance of the tree in memory. 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.

## 5.9 Alternatives to binary search

For simplicity of exposition, we have thus far examined binary search for finding the optimal action. It should however be clear that any schedule for splitting the action space could be implemented. We could even have a non-regular split, where different nodes have different numbers of children.

As an example, if the number of actions is a power of 4, we could have each internal state have 4 actions leading deterministically to 4 children. In that case the computational requirements for finding the optimal action would be exactly the same as in the binary case, although the tree would have half the levels as in the binary case. Naturally, for splits with more than 4 children per node, the computational requirements would quickly increase.

## 5.10 Non-regular action space partitions

It should noted, that for continuous action spaces, the space does not need to be divided into equally sized intervals. On a number of applications there may be areas of the state space that require finer resolution, while others are less important. In such cases, the partitioning of the space should follow the relative importance of different areas, rather than absolute distances.

# 6

# Experimental Results

This section presents results from testing the proposed approach on three domains with large action spaces. A more realistic and difficult version of the inverted pendulum problem, the double integrator problem, as well as the bicycle balancing problem.

## 6.1 Inverted Pendulum

The inverted pendulum problem (figure 6.1), 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 $\theta$ 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 (Wang et al., 1996).

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

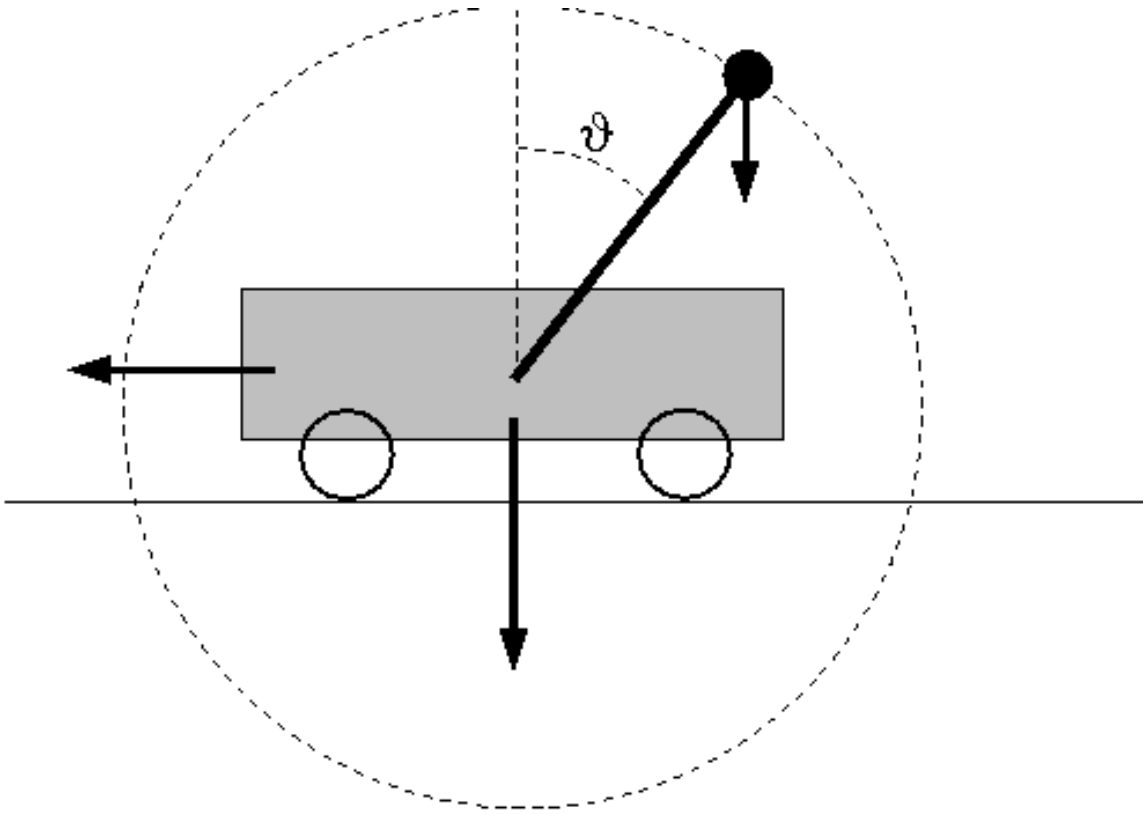


FIGURE 6.1: The Inverted pendulum problem.

configuration, and a negative reward when the controller fails and the pendulum falls. 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 for choosing action $u$, as long as $|\theta| \leqslant \pi/2$, 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[1] and kept only the first principal component $pc$. The

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

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 $5 \times 6$ regular grid over the state-action space:

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

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 6.2 shows the total accumulated reward as a function of the number of training episodes, when action search is combined with Least-Squares Policy Iteration (Lagoudakis and Parr, 2003) and Fitted-$Q$ iteration (Ernst et al., 2005).

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 steps, corresponding to 5 minutes of continuous balancing in real-time.

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 exhaustively 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 three-action controller learned with LSPI.

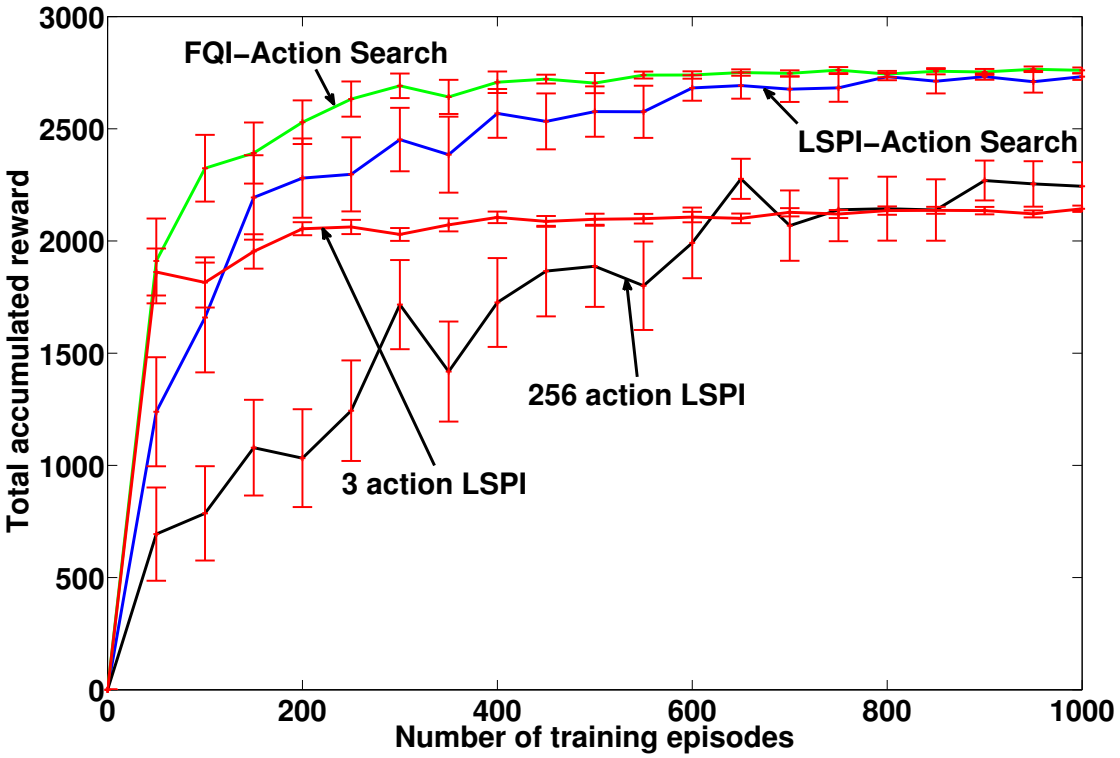


FIGURE 6.2: Total accumulated reward versus training episodes for the inverted pendulum regulation task. The green and blue lines represent the performance of action search when combined with FQI and LSPI respectively, while the red and black lines represent the performance of 3 and 256-action controllers learned with LSPI and evaluated for every possible action at each step.

It should come as no surprise that we are able to outperform the discrete three-action 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, the learning curve for the transformed MDP appears to be as steep as the one for the three-action 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 such 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.

## 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 the current velocity $v$. The goal is to bring the car to the equilibrium state $(0, 0)$ by controlling the acceleration $a$, under the constraints $|p| \leqslant 1$ and $|v| \leqslant 1$. 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 $\dot{v} = a$.

As the control frequency becomes lower, the car becomes more and more difficult to control. Large control inputs can easily make the car overshoot the target or even move outside its operating range. A control interval of 500msec was chosen in order to make the problem more challenging. Acceleration was restricted in the range $[-1, 1]$ and was approximated with an 8-bit resolution (256 values) resolution for the action search controllers. For this experiment we used the $Q$-value function formulation (see Appendix A), with a simple polynomial approximator with 10 terms for each action:

$$\phi = (1, p, v, a, p^2a, v^2a, a^2, pv, pa, va, a^2p, a^2v)^\top$$

For the discrete controllers, a similar polynomial approximator (without any $a$ terms) was used:

$$\phi = (1, p, v, pv)^\top$$

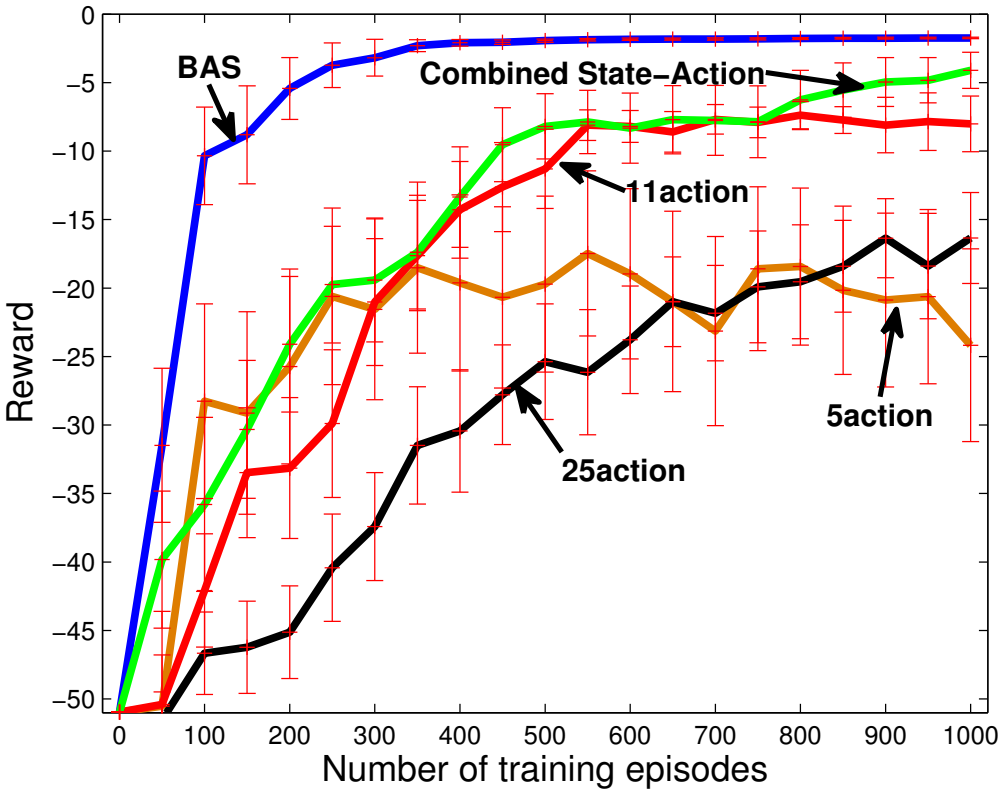


FIGURE 6.3: Double Integrator (LSPI) : Total accumulated reward.

Note that adding more terms to the approximator did not improve performance. Once again training samples were collected in advance from "random episodes" with a maximum length of 200 steps. For accurate assessment of performance, 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.

Figure 6.3 shows the total accumulated reward as a function of the number of training episodes. Once again, the BAS controllers learn much faster than their discrete counterparts and achieve far better rewards. Using a combined state-action approximator and evaluating for all 256 possible action choices yielded successful

policies for this domain, albeit at a 16-fold increase in computational cost compared to BAS. This is to be expected since it is very easy to generalize over neighboring actions for the linear dynamics of the Double Integrator. Even though the performance of the combined state-action approximator is better than that of the discrete controllers, it still falls short of the performance achieved by the BAS controllers.

## 6.3 Bicycle Balancing

The bicycle balancing problem (Ernst et al., 2005), has four state variables (angle $\theta$ and angular velocity $\dot{\theta}$ of the handlebar and angle $\omega$ 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 $|\omega| \leqslant \pi/15$, 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 and the control interval was set to 10msec. 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 $2^{16}$ (65,536) discrete actions, which brings us well beyond the reach of exhaustive enumeration. With the

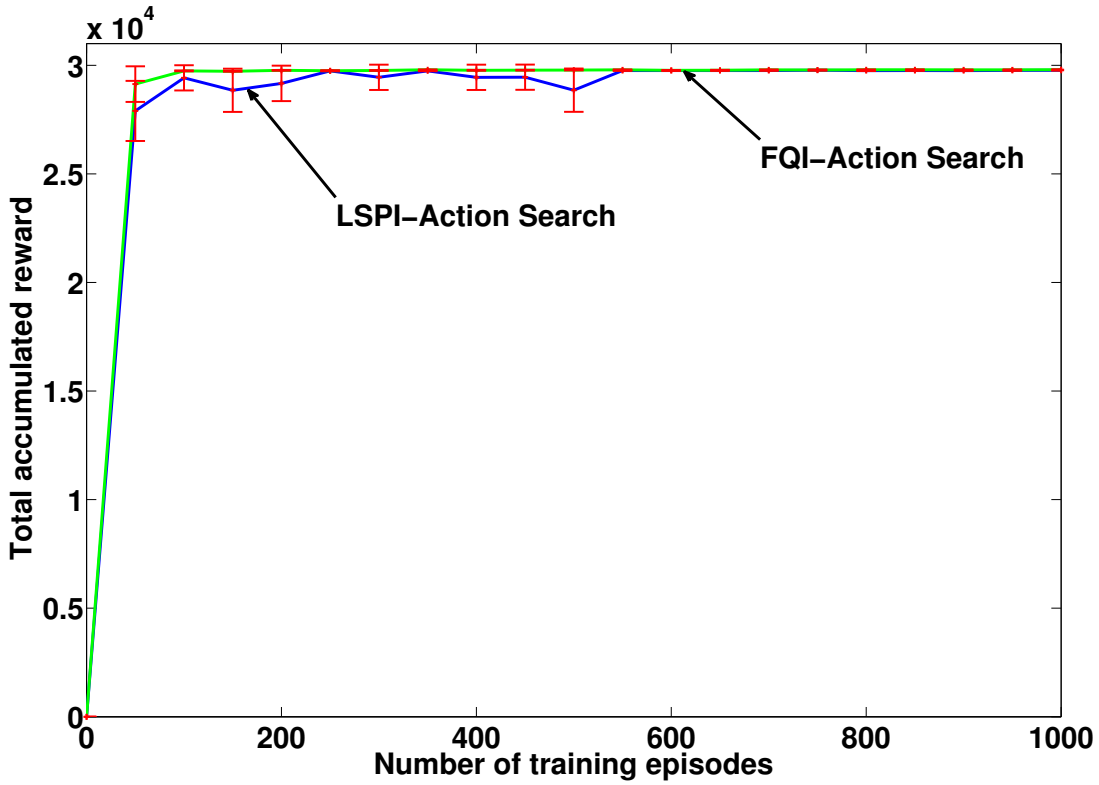


FIGURE 6.4: Total accumulated reward versus training episodes for the bicycle balancing task using action search combined with FQI and LSPI.

approach presented in this thesis we can reach a decision in just 16 value comparisons.

Figure 6.4 shows the total accumulated reward as a function of the number of training episodes when the proposed method is combined with LSPI and FQI. Once again the experiment was repeated 100 times for the entire horizontal axis to obtain average results and 95% confidence intervals over different sample sets. Training trajectories were truncated after 20 steps[2]. Each episode was allowed to run for a maximum of 30,000 steps corresponding to 5 minutes of continuous balancing in real-time. The learned controllers were almost always able to balance the bicycle for the entire time with as little as 50 training episodes.

[2] Practically all samples generated by the random policy after the first 20 steps are part of trajectories that have no chance of recovering, and provide no useful information.

# 7

# Discussion and Conclusion

## 7.1 Strengths and weaknesses

The method presented in this thesis has a number of unique advantages, that makes it a good candidate for domains with continuous and/or multidimensional action spaces:

- An easily overlooked advantage of action search is that it is a deterministic algorithm. Contrary to stochastic approaches to action selection, it does not suffer from transient dips in performance due to unlucky action sampling, nor does it require a source of random numbers which can be problematic in embedded platforms.

- One of the greatest advantages of the proposed scheme, is its simplicity. It is very easy to implement, both conceptually and in terms of the amount of code required.

- It requires only a selection between 2 actions from the Reinforcement Learning algorithm of choice. This allows for very fast and efficient implementations of the policy function.

- It easily achieves very fine, practically continuous action resolutions impossible to reach with naive discrete controllers.
- It is able to scale to multidimensional action spaces, without strong assumptions about the shape of the action space.
- It can be used in conjunction with any Reinforcement Learning algorithm with discrete actions and is not tied to any particular implementation.
- It can be used in an online, offline, on-policy or off-policy setting.
- It has very low computational and memory requirements.
- It requires essentially no tuning. The only tunable variable is the desired resolution for each control variable, which may be part of the problem statement.
- It results in smooth control, minimizing mechanical stresses and power consumption.
- In contrast to some other approaches that feature smooth control, it is able to take action immediately when a situation that requires it arises.

Of course there are things that were not addressed in this thesis. Here are the main potential weaknesses of our approach:

- Since action complexity is transformed to state complexity, state space complexity is increased.
- While action selection time is reduced, the representation complexity remains the same as with other naive approaches.
- We do not address the exploration problem, which is even more difficult in large action spaces.

The next section discusses some directions for future research that aim to alleviate some of these weaknesses.

## 7.2 Future work

Throughout the thesis we assume 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.

As mentioned in Section 5.7.2, an interesting observation is that only a small subset of the value function is accessed during policy execution. Pazis and Parr Pazis and Parr (2011a) demonstrate that this fact can be used to achieve an exponential reduction in representation complexity in addition to the reduction in action selection complexity. Unfortunately their construction is only straightforward when approximate linear programming is used as the learning method, which requires noise free samples. In addition, because it only tries to approximate the values of the best

actions, it does not seem very well suited for online learning or exploration. An interesting direction for future research would be to see if reductions in representation complexity are possible without sacrificing in other areas.

In our experiments we have used batch learning algorithms. One particularly promising direction is the combination of the research presented in this thesis with an optimistic exploration framework for online learning, similar in spirit to the pessimistic framework of Pazis and Parr Pazis and Parr (2011b). We believe that such a framework could be used to perform PAC[1]-optimal exploration in continuous state-action spaces.

The approach presented in this thesis transforms action complexity to state space complexity. This allows the learning algorithm to be oblivious to the fact that we have a large action space. While this is in general a benefit, it could potentially lead to lost opportunities for optimization. Action spaces have a number of unique properties. Often, even if the ambient dimension of our state spaces is large, the accessible states may lie in a lower dimensional manifold. On the other hand, the entire action space is accessible. During sampling/exploration in a high dimensional action space, it's up to the agent not to spend too much time exploring non-promising actions. One direction for future work would be to investigate how we can exploit properties of the transformed MDP to guide exploration.

## 7.3 Conclusion

This thesis presented a principled approach for efficiently learning and acting in domains with continuous and/or multidimensional control variables. We have shown that by using a simple MDP transformation, we can attack both the generalization and action selection problems simultaneously, with minimal cost.

I believe that this work brings us one step closer to the ultimate goal, of having

[1] PAC stands for Probably Approximately Correct.

black box reinforcement learning algorithms that can be used in real world problems. Of course as we have seen there is still much work to be done. Everything that was presented here is neither a first step (as it depends on good algorithms for learning in complex state spaces) nor the last. It is instead a bridge. Its purpose is to help bridge the gap between the abilities of modern algorithms and the requirements of real-world problems.

# Appendix A

## Q-Value Formulation

This appendix is meant to be a (mostly) self contained alternative to chapter 5. It provides the original $Q$-value formulation of the Binary Action Search algorithm (Pazis and Lagoudakis, 2009a), with the addition of support for multidimensional action spaces. It is included for reasons of completeness, and to show the motivation and intuition behind developing Binary Action Search.

### A.1 Intuition

At every timestep, a typical reinforcement learning agent decides what action to take. Unfortunately, as we saw in Chapter 3, when the number of actions is large, the computational cost of directly choosing an action can be prohibitive.

Some work has been able to break from the typical paradigm, by taking advantage of temporal locality. Taking advantage of the fact that in continuous action spaces successive actions are usually similar, a couple of algorithms opt to modify the current action, instead of choosing an action directly. The modification steps can either be fixed (Riedmiller, 1997) or adaptive (Pazis and Lagoudakis, 2009b). Unfortunately,

when temporal locality is reduced even for some parts of the state space, these algorithms run into problems.

The key observation in designing an improved algorithm is that nothing prevents us from querying the internal binary policy more than once before we apply our decision. If, for example, the answer to the first question is to increase the action value, we can always compute the new value and check what the binary policy suggests for that new action value in the same state. If the suggestion is to decrease, then we have an over-shoot. If the suggestion is to increase, then we have an under-shoot. Repeated look-ahead queries will allow us to search for an action value that reduces the effects of over-shooting and under-shooting. Going a step further, we don't even have to use the previous action value as the starting point of our search, or have a constant step size. Instead, we can start at any point and use any convenient step sizes to search the action range and successively approximate the value of the best continuous action choice in the current state. In the absence of domain knowledge, this search can be accomplished in an optimal way using a binary search scheme. Starting at the center of the range, and halving the size of the step after each query, we can quickly get very close to the optimal continuous action.

## A.2 Binary Action Search

The proposed algorithm, called Binary Action Search (BAS), looks for the best action choice in the continuous action range $[a_{\min}, a_{\max}]$ using a finite number of binary search steps. The first query will be at the center of the range: "*When in state s, would you rather increase or decrease the value of the action* $a = (a_{\max} + a_{\min})/2$*?*" The answer will eliminate half of the action range. The next query will be at the center of the remaining range, and so on. In general, each binary decision will eliminate half of the remaining possible choices. The number of decisions required to come to a final decision is the same as the number of bits of the desired resolution. For

**Action Search**
**Input:** state $s$, value function $Q$, 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** $Q(s, a - \Delta) > Q(s, a + \Delta)$ // 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** $a$

FIGURE A.1: A practical implementation of the binary action search algorithm

example, if we want to have 256 values for the continuous action (8-bit resolution), we can use BAS to reach a final decision within 8 queries. The first query will eliminate 128 of the 256 potential choices, the second query will eliminate 64 of the remaining 128 choices, and so on, up to the eighth query which will leave us with just one choice. The Binary Action Search approach is summarized in figure A.1. Note that $\Delta$ is initialized to a value that allows for proper coverage of the entire action range (including $a_{\min}$ and $a_{\max}$) within a finite number of steps $N$.

## A.3 Learning

The binary policy required by BAS answers the question: "*When in state $s$, would you rather increase or decrease the continuous action $a$?*". Most existing RL algorithms can be used in conjunction with BAS to learn such binary-action policies. There are only real requirement on the learning algorithm of choice is that it must be able to handle continuous state spaces, since the original state space $\mathcal{S}$ will have to be augmented with the latest value of the continuous action $a$, therefore the binary pol-

icy must be learned over the augmented state space $(\mathcal{S}, \mathcal{A})$. Of course most domains with continuous actions already have continuous state variables, so this shouldn't be a problem.

One potentially non-obvious point that deserves some attention is samples generation. At first glance, even though we have as many binary decisions per step as we have resolution bits, there is only one interaction with the environment, and thus only one reward and one state transition observed. However, we should stress that a single step in the environment does not not correspond to a single step for the BAS learner.

From the BAS learner's point of view, every decision point corresponds to a sample. The action in each such sample is the corresponding binary decision (increase or decrease). The (augmented) state in each such sample is the combination of the state of the process with the current search point in the action range. The last sample of each decision cycle, includes the observed state transition in the original state space $\mathcal{S}$ after the interaction with the environment, as well as a resetting of the search point to the center of the action range for the observed next state. Resetting the search point is required in order to keep the samples along the entire trajectory of binary decisions "connected". The reward is zero for all samples, but the last, which carries the actual reward observed and therefore it is the only one that should be discounted.

As an example consider a resolution of 3 bits, the continuous action range $[1.0, 8.0]$, and an agent who internally makes the binary decisions $-1$, $-1$, $+1$ to find the continuous action $a = 2.0$ which is applied to state $s$. The sample from interacting with the environment will be $(s, 2.0, r, s')$, however the three samples used for learning the binary policy will be $\big((s, 4.5), -1, 0, (s, 2.5)\big)$, $\big((s, 2.5), -1, 0, (s, 1.5)\big)$, and $\big((s, 1.5), +1, r, (s', 4.5)\big)$.

## A.4 Relationship to the V-value function formulation

Even though the intuition behind the Q-value function formulation presented in this appendix, and the V-value function formulation presented in chapter 5 is very different, the resulting algorithms are practically identical. In fact, we can see that the values stored by the two formulations are the same. For the V-value formulation we have a binary tree of height $\log(|A|)$ whose root does not need to be explicitly stored. All the nodes and leaves of that tree have a one to one mapping to the two binary trees (one for each choice "increase" or "decrease") of height $\log(|A|) - 1$ present in the Q-value function formulation.

# Bibliography


Albus, J. S. (1975), "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller," *Journal of Dynamic Systems, Measurement and Control*.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming*, Athena Scientific.

de Farias, D. P. and Roy, B. V. (2004), "On Constraint Sampling in the Linear Programming Approach to Approximate Dynamic Programming," *Mathematics of Operations Research*, 29, 462–478.

Ernst, D., Geurts, P., and Wehenkel, L. (2005), "Tree-Based Batch Mode Reinforcement Learning," *Journal of Machine Learning Research*, 6, 503–556.

Kaelbling, L. P., Littman, M., and Moore, A. (1996), "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, 4, 237–285.

Kimura, H. (2007), "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)*, pp. 88–95.

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

Lazaric, A., Restelli, M., and Bonarini, A. (2008), "Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods," in *Advances in Neural Information Processing Systems (NIPS) 20*, pp. 833–840.

Martí•n H., J. A. and de Lope, J. (2009), "`Ex<a>`: An effective algorithm for continuous actions Reinforcement Learning problems," in *Proceedings of the 35th Annual Conference of IEEE on Industrial Electronics*, pp. 2063–2068.

Millán, J. D. R., Posenato, D., and Dedieu, E. (2002), "Continuous-Action $Q$-Learning," *Machine Learning*, 49, 247–265.

Pazis, J. and Lagoudakis, M. (2009a), "Binary Action Search for Learning Continuous-Action Control Policies," in *Proceedings of the 26th International Conference on Machine Learning*, eds. L. Bottou and M. Littman, pp. 793–800, Montreal, Omnipress.

Pazis, J. and Lagoudakis, M. G. (2009b), "Learning Continuous-Action Control Policies," in *Proceedings of the IEEE Intl Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 169–176.

Pazis, J. and Lagoudakis, M. G. (2011), "Reinforcement Learning in Multidimensional Continuous Action Spaces," in *IEEE Symposium Series in Computational Intelligence 2011 (SSCI 2011)*, Paris, France.

Pazis, J. and Parr, R. (2011a), "Generalized Value Functions for Large Action Sets," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pp. 1185–1192.

Pazis, J. and Parr, R. (2011b), "Non-Parametric Approximate Linear Programming for MDPs," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, (AAAI)*.

Peters, J. and Schaal, S. (2006), "Policy gradient methods for robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2219–2225.

Puterman, M. L. (1994), *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley-Interscience.

Riedmiller, M. (1997), "Application of a Self-Learning Controller with Continuous Control Signals Based on the DOE-Approach," in *Proceedings of the European Symposium on Neural Networks*, pp. 237–242.

Riedmiller, M. (2005), "Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method," in *Machine Learning: ECML 2005*, vol. 3720, pp. 317–328.

Russel, S. and Norvig, P. (2003), *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Sallans, B. and Hinton, G. E. (2004), "Reinforcement Learning with Factored States and Actions," *Journal of Machine Learning Research*, 5, 1063–1088.

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

Sutton, R. and Barto, A. (1998), *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Massachusetts.

Wang, H., Tanaka, K., and Griffin, M. (1996), "An Approach to Fuzzy Control of Nonlinear Systems: Stability and Design Issues," *IEEE Transactions on Fuzzy Systems*, 4, 14–23.

# Publications

- Jason Pazis and Michail G. Lagoudakis, Reinforcement Learning in Multidimensional Continuous Action Spaces, *Proceedings of the 2011 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL 2011), Paris, France, April 2011, pp. 97–104.*

- Jason Pazis and Michail G. Lagoudakis, Binary Action Search for Learning Continuous-Action Control Policies, *Proceedings of the 26th International Conference on Machine Learning (ICML 2009), Montreal, Quebec, Canada, June 2009, pp. 793–800.*