AN ABSTRACT OF THE DISSERTATION OF

William Curran for the degree of Doctor of Philosophy in Robotics and Computer Science presented on June 5, 2017.

Title: High-Dimensional Reinforcement Learning with Human Feedback

Abstract approved: ______________________________

William D. Smart Prasad Tadepalli

State-of-the-art personal robots need to perform complex manipulation tasks to be viable in complex scenarios. However, many of these robots, like the PR2, use manipulators with high degrees of freedom. High degrees of freedom are desirable from a functionality standpoint, but make the learning task more difficult by adding a high-dimensional state space. The problem is made worse in bimanual manipulation tasks. Our proposed approach is to scale existing reinforcement learning techniques to learn in high-dimensional robot control problems.

We propose reducing the state space by using demonstrations to discover a representative low-dimensional manifold in which to learn. This allows the agent to converge quickly to a good policy. We call this Dimensionality-Reduced Reinforcement Learning (DRRL). However, when performing dimensionality reduction, sometimes important state information is lost. We extend this work by first learning in a single dimension, and then transferring that knowledge to a higher-dimensional space. By using our Iterative DRRL (IDRRL) framework with an existing learning algorithm, the agent converges quickly to a better policy by iterating to increasingly higher dimensions. IDRRL is robust to demonstration quality and can learn efficiently using few demonstrations.

We use Principal Component Analysis (PCA) for our linear dimensionality reduction in DRRL and IDRRL. However, linear dimensionality reduction assumes that the underlying data can be represented by a lower dimension linear subspace. Robot state spaces typically include velocities and accelerations, whose equations of motion are inherently nonlinear. Standard linear dimensionality reduction techniques cannot accurately

represent complex nonlinear structures. However, nonlinear dimensionality reduction techniques are too computationally complex to use online. To overcome these limitations, we introduce a novel approach to dimensionality reduction based on a system of cascading autoencoders (CAE), producing the new algorithm IDRRL-CAE.

Optimization is useful, but fast learning doesn't help if the objective function is deceptive or difficult to define mathematically. In many cases, roboticists may not be able to predict all scenarios their robots may experience, and thus cannot design an objective function for every case apriori. In these situations it may be helpful to incorporate human feedback. To give effective feedback, users need an interface that is intuitive, time insensitive, and incorporates both fine-grained and coarse feedback.

To incorporate human feedback in our learning, we use timeline interfaces. Timeline interfaces that allow you to move backward and forward through a video have been used by video editors for years. They are simple and designed for both non-experts and video editing experts. These interfaces allow a user to cut, concatenate, rewind, fast forward, and perform many other tasks on videos. They speed up the editing process by decoupling the timescale of the editing process from the timescale of the video being edited. These same concepts can be used in human feedback mechanisms for robot control systems. Current human feedback mechanisms require the user to quickly respond to robot actions, work in only discrete spaces, or only allow for either coarse or detailed feedback. The timeline interface paradigm naturally accounts for fine-grained state spaces, does not require quick human feedback, allows the user to make both coarse and fine-grained edits to video, and decouples the speed of the video from the speed of feedback. In this dissertation we present a proof-of-concept movie reel interface that uses this timeline interface paradigm.

High-Dimensional Reinforcement Learning with Human Feedback

by

William Curran

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 5, 2017
Commencement June 2017

Doctor of Philosophy dissertation of William Curran presented on June 5, 2017.

APPROVED:

Co-Major Professor, representing Robotics

Co-Major Professor, representing Computer Science

Head of the School of Mechanical, Industrial and Manufacturing Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

William Curran, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

Our ultimate goal is to deploy personal robots into the world, and have members of the general public retask them without having to resort to explicitly programming them. Manipulation in the real world is an essential task for personal household robots. Robots designed for manipulation, such as the PR2, are constructed with high degrees of freedom to allow them to perform complex motions. For example, the PR2 robot has two 7 degree-of-freedom arms. When learning position and velocity control, this leads to a 14 dimensional state space per arm. Precise execution of manipulation tasks are crucial for these robots to be viable for personal use, but presents a challenging control problem.

A classical control approach would include modeling the robotic system and leveraging this model to move the robot to a desired target state. However this presents several problems when the robot is sufficiently complex. Dynamics models are not always available or sufficiently accurate. If they are available, classical controls approaches may still not be able to offer an analytical solution. Furthermore, even when a model and an analytical solution are available, the application of this model may be too computationally complex for real-time control [38, 60, 106].

Learning from demonstration (LfD) methods learn a policy using examples or demonstrations given by a human to speed up learning a custom task [6]. However, these demonstrations must be consistent and accurately represent solving the task. These methods also solve for a specific complex task, rather than solve for general control [6]. In addition, it is often necessary to give tools to a user who wishes to modify how the robot is executing a task. There are many scenarios in which a robot has learned an optimal trajectory with respect to an optimization function, but not with respect to human preference. In this dissertation we present two approaches to directly address the problems of learning good policies with reinforcement learning in high-dimensional state spaces and incorporating human feedback in reinforcement learning policies.

The first step in our research goals is to develop an efficient method for teaching the robot. Reinforcement learning provides a much faster implementation because it offers model-free reactive control. It can be used in many ways relevant to robot tasks, includ-

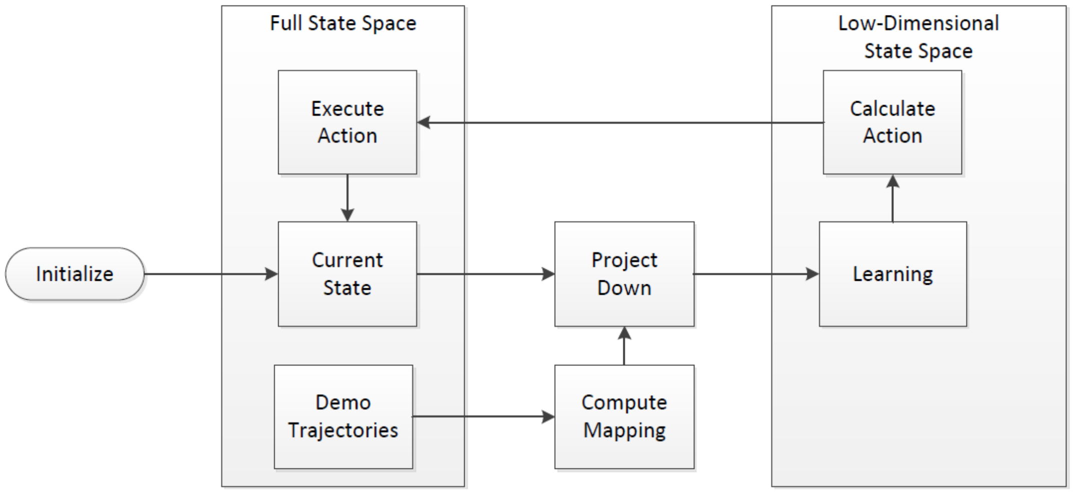

Figure 1.1: One iteration of the DRRL algorithm.

ing teaching a robot new skills that a human cannot demonstrate, finding novel ways to reach human-defined goals, and finding solutions to complex dynamics problems with no analytic formulation [65]. Though reinforcement learning offers a fast and flexible solution to finding robotic movements, it can be difficult to apply in the high-dimensional spaces commonly found in robotics. High-dimensional solution spaces significantly increase the time and memory requirements of many learning algorithms, and performance can suffer when the solution space cannot be fully explored [57].

We introduce two algorithms: Dimensionality-Reduced Reinforcement Learning (DRRL) and Iterative DRRL. In DRRL we use demonstrations to compute a projection of the state space to a low-dimensional subspace. In each learning iteration, we project the current state into this subspace, compute and execute an action, project the new state into the subspace, and perform a reinforcement learning update (Figure 1.1). This general approach has been shown to reduce the exploration needed and accelerates the learning rate of reinforcement learning algorithms [14, 27].

By learning using DRRL, the agent can learn quickly in the small low-dimensional subspace. However, this leads to a critical trade-off. By projecting onto a low-dimensional manifold, we are discarding potentially important data. By adding DRRL to an existing algorithm, we show that the robot can quickly converge to a good policy much faster.

However, since DRRL does not represent all dimensions, it could converge to a suboptimal policy.

In many learning domains, poor policies are undesirable. In robotics in particular, bad controllers can damage the robot. We propose a novel framework, IDRRL, combining learning from demonstration techniques, dimensionality reduction, and transfer learning. Instead of learning entirely in one manifold, we iteratively learn in all subspaces by using transfer learning. The robot can quickly learn in a low-dimensional space d, and transfer that knowledge from d dimensions to the $d + 1$ dimensional space using the known mapping between the spaces.

Our novel approach is a framework to improve other learning algorithms when working in high-dimensional spaces. It combines the speed of low-dimensional learning and the expressiveness of the full state space. We demonstrate DRRL and IDRRL converges quickly and to a better policy than policies learned only in the full dimensional space. We show this in the Mountain Car domain [41], a common benchmarking problem, the Swimmers domain [28], an abstraction of a robot control problem, and a Yaskawa Motoman MH5F robotic arm learning to balance a ball bearing on an acrylic plate.

Our early work used Principal Component Analysis (PCA) for its dimensionality reduction technique. Although this technique worked well, it did not capture the nonlinearities involved in complex robot state spaces. Alternative non-linear dimensionality reduction techniques were far too computationally expensive to use in this learning framework. This limited the performance of IDRRL when dynamics were highly nonlinear.

We introduce a novel non-linear and fast dimensionality reduction technique using a cascade of autoencoders. We then use this to extend IDRRL to handle nonlinear dynamics. We demonstrate the learning speed increase when a Q-learning algorithm is augmented with IDRRL in the Mountain Car domain and the Swimmers domain.

The second step of our research is to develop an effective mechanism to modify the objective functions our learning algorithms use. Formulating robot movement in terms of an objective function is a common approach in robotics. Both reinforcement learning and path planning optimize trajectories over these functions to perform many robot tasks [33, 42, 63]. In controlled scenarios, roboticists can easily define these functions since robots need to minimize predictable parameters, like power usage, or distance-to-goal.

However, in some situations, the exact objective function may not be clear. Or, the outcome of the optimization might not result in a desirable trajectory because of unforeseen interactions. To alleviate this issue, we believe objective functions need to consider human preference and avoid locally optimal solutions, but these concepts are difficult to define mathematically. In many cases roboticists may not be able to predict all scenarios their robots may experience, and thus cannot design an objective function for every case apriori. Human feedback can be incorporated to guide robot behavior in unexpected scenarios.

This dissertation proposes a style of tool humans can use to give this feedback. Many current feedback mechanisms only use high-level feedback such as voice or drawings, which are not detailed enough for fine-grained feedback [16, 21], do not account for continuous spaces [112], or require the user to give time-sensitive feedback [46]. In this work, we introduce a new interface design for human feedback that leverages a timeline interface paradigm. This paradigm has exited for years in non-linear video editing systems [54].

Non-linear video editing systems employ a timeline interface paradigm to allow users to make edits to film at any point and scan through the video easily. This interface decouples the speed of the video from the speed of the feedback, allowing for faster editing. It features multiple levels of feedback, so the user can step through a timeline of the video and make fine adjustments to small parts of the video or coarse adjustments to large sections. In this work, we use this timeline interface to allow the interactive design of objective functions.

This timeline interface paradigm naturally accounts for fine-grained state spaces, since video is typically at 30-60 frames-per-second. It also allows the user to scan through a video by fast forwarding, rewinding, and pausing, removing the need to quickly react as the robot is executing. Users can slowly step through the robot execution, and give fine-grained feedback, or choose a large area of time and give general feedback.

In this dissertation, we implement a proof-of-concept user interface using the timeline interface paradigm. We test this interface in a deceptive learning problem. Deceptive learning scenarios are standard reward design problems in which the agent must navigate through many negative rewards to find the solution [75]. The agent in this deceptive problem needs human feedback to modify its objective function to reduce the local maxima. We also use our interface to modify a planned path associated with dangerous navigation. In this problem, an agent plans a path close to a descending stairway and around a sharp corner. A natural human preference is to incentivize the robot to stay away from stairs. We use our interface to incorporate this feedback in the path planner's objective function.

The research objectives of this work are to create an end-to-end policy learning and policy feedback system. We wish to scale existing reinforcement learning techniques to learn in high-dimensional robot control problems and then incorporate human feedback in those objective functions to intelligently guide learning.

Chapter 2: Background

In our work there is a depth of existing literature we leverage. Our dimensionality-reduced reinforcement learning algorithms use concepts from the field of reinforcement learning, dimensionality reduction, transfer learning and learning from demonstration. Our timeline interface work use ideas from classic video editing techniques to fix many of the issues we discuss here in existing human feedback mechanisms.

2.1 Reinforcement Learning

Reinforcement learning is a tool within the field of multiagent or single-agent learning where agents take an action, observe the environment, and receive a reward based on the new environment [103]. Reinforcement learning addresses the problem of finding an optimal policy, $\pi^*(s,a) = P(a|s)$, that maximizes a reward function, R:

$$J(\pi) = \sum_{s,a} \mu^{\pi}(s)\pi(s,a)R(s,a) \tag{2.1}$$

where $\mu^{\pi}(s)$ is the probability distribution over policy π (Section 2.1.3).

To develop this optimal policy, reinforcement learning uses the current state of the agent, s, the action taken, a, the resultant state, s', and a reward corresponding to the system-level success or failure of the state and action, R.

In our reinforcement learning approach, we use the standard formulation of MDPs [57]. An MDP is a 4-tuple $\langle S, A, T, R\rangle$, where S is a set of states, A is a set of actions, T is a probabilistic state transition function $T(s,a,s')$, and R is the reward function $R(s,a)$. Reinforcement learners can learn with and without a model of the environment. In this work we use stateful reinforcement learning.

There are three main aspects when defining a stateful reinforcement learner: the policy, the reward function, and the state transitions.

2.1.1 Policy

The reinforcement learner's policy represents the actions an agent should take in each state. This policy is computed either by a value function representing the quality of a particular action within the state, or a direct search through the policy-space:

$$a = \pi(s) \tag{2.2}$$

2.1.1.1 Value Function

Learning algorithms based on value functions estimate a value function $V^{\pi}(s)$ that computes the long-term reward of state s to derive a policy $\pi(s)$ [58]. There are two common value function-based methods to estimate $V^{\pi}(s)$, policy iteration and temporal difference methods.

2.1.1.2 Policy Iteration

Policy iteration and value iteration are a model-based approaches that require the transition probabilities $T(s', a, s)$ (the model) and the reward function $R(s, a)$ to calculate the value function. Policy iteration consists of alternating between policy evaluation and policy improvement. Policy evaluation updates the value function by visiting each state in the current policy, and updating the estimate of $V(s)$ [18]:

$$V_i^{\pi}(s) \leftarrow R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s') \tag{2.3}$$

this update is based on the current value estimates of the successor states $V^{\pi}(s')$, the state transition probabilities $T(s, \pi(s), s')$, the current policy π, the reward function $R(s, \pi(s))$ and the discount factor γ. After the policy is evaluated, the policy is then greedily improved by selecting the best action in every state, according to $V(s)$. Policy evaluation and policy improvement is repeated until the policy has converged.

Value iteration combines the policy evaluation and policy improvement by directly updating $V(s)$ every time the state changed [18]:

$$V_i^{\pi}(s) \leftarrow \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^{\pi}(s') \right] . \tag{2.4}$$

Policy iteration and value iteration are model-based approaches that require the transition probabilities $T(s', a, s)$ (the model) and the reward function $R(s, a)$ to calculate the value function. Policy iteration consists of alternating between policy evaluation and policy improvement. Policy evaluation updates the value function by visiting each state in the current policy, and updating the estimate of $V(s)$ [18]. Policy improvement then picks the policy considered optimal. Learning stops when policy improvement converges.

Policy iteration and value iteration guarantee convergence to the optimal solution, assuming that we have a model of the environment and infinite time [94]. Both approaches are nearly exhaustive in the search for an optimal policy, and the time complexity includes the size of the state space and action space. In addition, the model of the environment must be completely accurate in order to avoid accumulation of error. This makes policy iteration difficult to perform in the continuous and large state spaces typically seen in the field of robotics.

Alternatively, temporal difference methods use the reward received at each time step to calculate the difference between the old estimation of the value function and the new estimation. One such example is Q-Learning [103]. Q-learning updates of the agents mapping are based on the reward received. Equation 2.5 show the action-value update:

$$Q_t(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha(R(s_t, a_t, s_{t+1}) + \gamma \max_a Q(s_{t+1}, a) - Q_t(s_t, a_t)) \ , \tag{2.5}$$

where $Q_t(s_t, a_t)$ is the updated state-action value table entry for action a and state s, $\max_a Q(s_{t+1}, a)$ is the next-best action, α is the learning rate between 0 and 1 (1 causing the agent to only take the most recent reward into account and 0 causing no update), and $R(s_t, a_t, s_{t+1})$ represents the reward received this time step for taking action a in state s. In a static environment, if the α term is chosen too high the agent will have a hard time finding the best long term policy, and if it is set too low an agent will take a long time to reach the optimal policy. In a constantly changing environment α needs to be set to a rate close to the rate of change. If set too low the environment will change faster than the agent is learning the environment, and if set too high the same problems arise as in the static environment [103].

In contrast to policy iteration approaches, the temporal difference learner does not perform a search of all successor states. The agent is required to arbitrarily explore previously unseen states or unused actions. In order to find an optimal policy, the agent visits as much of the state-action space as possible. This can result in large and sudden modifications to the current policy. This instability is not desired in robotics, since the robot can be easily damaged. Cautious exploration must be enforced by either avoiding significant changes in the policy (enforcing stability) [87], or strictly disallowing the entering of undesired states [34].

Temporal difference methods also guarantee convergence to the optimal solution, with the same assumptions of infinite time [94]. Since these methods do not require the search of all successor states, the computational complexity is much lower. Temporal difference learning methods are the most widely used methods in reinforcement learning due to the simplicity, minimal amount of computation, and fewer constraints than alternative methods [103].

Value function approaches are compact, simple, and have been proven to find an optimal policy [63] under reasonable conditions. However, in order to obtain these optimality guarantees, nearly the complete state-action space must be visited, which is often computationally intractable in robotics. In addition, the value function must be discretized when used in a continuous state or action space problem, which can lead to approximation errors. Approximation errors can lead to errors in the value function, which can propagate throughout the policy [33].

2.1.1.3 Policy Search

Policy search algorithms are the reinforcement learning approach of choice in robotics [33, 63]. Policy search methods use parameterized policies, rather than a value function, essentially performing local explorations through policy space.

Parameterized policies enable reinforcement learning to scale to high dimensional and continuous action spaces by reducing the space of possible policies. Although the localized policy search removes the convenient optimality guarantees of value function reinforcement learning, it includes the benefits of better scalability [33, 63], more intuitive policy initialization with expert knowledge [88], and stability [12].

Given that policy π is parameterized by the set of parameters w, the most general policy representation is:

$$\pi(s) = w^T\theta(s) \tag{2.6}$$

where $\theta(s)$ is a basis function vector, typically linear in simple problems, and nonlinear with additional free parameters in more complex problems. Alternatively, policies can be represented by a function approximation, such as a neural network [48].

Similar to value function approaches, policy search repeats a three step process: exploration of the state space, evaluation of the policy, and the update of the current policy. The exploration strategy is an important step in policy improvement for all reinforcement learning approaches and was discussed in the previous section.

There are two types of policy evaluation when performing policy search, step-based and episodic. The step-based policy evaluation is identical to the iterative policy evaluation approach used in value function learning. The quality of an action is given by the expected future reward of executing an action in a specific state. Value function approaches can be used in this case, as well as a Monte-Carlo estimation.

The episodic policy evaluation uses the expected return of the entire learning episode to update the parameter vector w. Rather than updating the value function associated with a specific state-action pair, episodic policy evaluation evaluates the policy parameters directly.

Once a policy is evaluated, it needs to be updated based on the resultant evaluation of the state-action pair or the policy parameters. Most policy search research focuses on improving the policy update step with respect to maximizing sample efficiency, minimizing the number of tunable parameters, and increasing learning stability [33]. There are many mathematical approaches for policy update, including policy gradient [10, 64, 88, 99, 117], expectation-maximization [62, 78], and information-theoretic approaches [31, 87].

Policy gradient approaches are direct, in that they directly compute the gradient to maximize policy performance. The REINFORCE algorithm was one of the first policy search algorithms, performs well, and computes policy updates without explicitly computing gradient estimates [117]. However, these approaches require the user to specify a learning rate, which can lead to instability or slow learning if not chosen correctly.

Expectation-Maximization methods remove the need for a learning rate. Variational Inference for Policy Search (VIP) is a typical example. It uses inference to autonomously determine the exploration rate and learning rate [78] but these approaches can potentially have an unstable learning process as well as tends to converge to local minima [33].

Information-theoretic approaches are considered the best for policy search, as they combine the benefits from both policy gradient and expectation-maximization approaches [33]. Information-theoretic approaches bound the distance between the prior policy and the new policy at each update step, enforcing stability in the learning process, and simultaneously updates its policy using expectation-maximization, removing the learning rate [33].

2.1.1.4 State Generalization

In robotics applications of reinforcement learning, states are defined as continuous parameters, such as joint angles, voltages, or torques. As a result, it quickly becomes prohibitively costly for learning approaches to directly represent the state-action value function. Function approximation techniques remove the need to discretely represent the value function by using examples to approximate the entire function. In this work, we use the tile coding function approximation technique, Cerebellar Model Arithmetic Computers (CMACs) [3].

CMACs partition a state space into a set of overlapping tiles, and maintain the weights (θ) of each tile. The accuracy of the generalization is improved as the number of tilings (n) increases. Each tile has an associated binary value (ϕ) to indicate whether that tile is present in the current state. The shape of the tile represents the generalization over the state space. In our work, the shape of all tiles remain the same static value (Figure 2.1).

To compute the active tiles associated with a state, we need to first normalize the state between $0 - 1$ and divide by the generalization factor, g (Algorithm 2). This generalization factor is a value between 0 and 1 that creates the possibility of generalization between any two states variables that are within the generalization factor of each other. We can then compute the active tiles and the estimate of the value function:

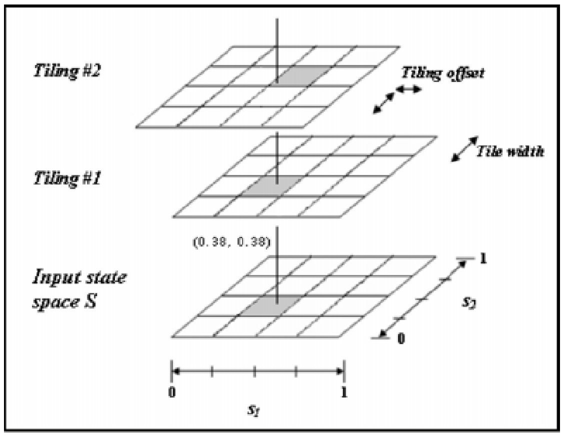

Figure 2.1: Tile Coding [43]

$$Q_t(s,a) = \sum_{i}^{n} \theta(i)\phi(i) \tag{2.7}$$

where $Q_t(s,a)$ is the estimated value function, θ is the weight vector and ϕ is a component vector. Given a learning example, we adjust the weights, θ, of the involved tiles to reduce the error. We use standard model-free Q-Learning to update our function approximation (Equation 2.5).

2.1.2 Reward Function

The reward function encompasses the high-level goal of the system. When an agent takes an action that is good for the system, the reward function ideally returns to the agent a positive reward proportional to how much it helped the system-level reward. Reinforcement learners also ideally receive a correspondingly smaller reward when their actions are detrimental to the system. In this way agents can iteratively update to converge to the optimal policy according to their reward design. This update is based on the action-value updates.

Reward functions can be given as a binary success or fail value at the end of an episode, and can be given iteratively throughout the simulation. Since success in a robot task involves many actions, rewarding each action is an ideal iterative way for a

Algorithm 1 Reinforcement Learning with Tile Coding Generalization

```
1: function LEARN
2:     Initialize $Q$ arbitrarily
3:     Initialize $\theta$ arbitrarily
4:     for $e \leftarrow 1$ to $episodes$ do
5:         $s \leftarrow initialize$
6:         for $t \leftarrow 1$ to $timesteps$ do
7:             $\{Q_t, \phi_t\} \leftarrow Approximate(s, \theta)$
8:             $a \leftarrow \max_a Q_t(s_t, a)$ with probability $1 - \epsilon$ otherwise $a \leftarrow$ random
9:             Take action a, observe $r$ and $s'$
10:            $\{Q_{t+1}, \phi_{t+1}\} \leftarrow Approximate(s', \theta)$
11:            $\delta \leftarrow r + \gamma \max_a Q_{t+1}(s', a) - Q_t(s, a)$
12:            $\theta[\phi_t] \leftarrow \frac{\alpha}{n}\delta$
13:            $s \leftarrow s_{t+1}$
14:        end for
15:    end for
16: end function
```

Algorithm 2 Tile Coding Approximation

```
1: function APPROXIMATE($s, \theta$)
2:     $s_n \leftarrow norm(s)$
3:     $s_n \leftarrow \frac{s_n}{g}$
4:     $\phi \leftarrow GetTiles(s_n, n)$
5:     $Q \leftarrow \sum_i^n \theta(i)\phi(i)$
6: return $\{Q, \phi\}$
7: end function
```

reinforcement learner to quickly converge. This approach of guiding the learning process with intermediate rewards is called reward shaping.

2.1.3 State Transitions

State transitions can be computed using a mathematical model or ignored by learning directly on the robot. Many intractable reinforcement learning problems can be made possible through the use of a model-based approach. Requiring fewer learning examples on the real robot is desirable, as robots need long execution times, manual intervention, and maintenance. However, models must be created from real-world data or computing the physical model of the dynamics. If the model is not accurate, model errors can accumulate and invalidate the learned policy.

2.2 Dimensionality Reduction

A core aspect of this work is the dimensionality reduction algorithm we use in DRRL and IDRRL. It needs to be fast enough for real-time execution and effective at reducing the state space. Here we explore multiple dimensionality reduction techniques and discus the applicability of each with respect to our proposed algorithms. We also give a brief outline of prior work in dimensionality reduction for learning.

2.2.1 Principal Component Analysis

PCA identifies patterns in data and reduces the dimensions of the dataset with minimal loss of information. It does this by computing a orthogonal transform to convert correlated data to linearly uncorrelated data. This transformation ensures that the first principal component captures the largest possible variance. Each additional component captures the largest possible variance uncorrelated with all previous components. Essentially, PCA represents as much as the demonstrated state space as possible in a lower dimension.

PCA deconstructs the data into eigenvectors and eigenvalues. For every eigenvector there is a corresponding eigenvalue. Each eigenvector represents the direction and the corresponding eigenvalue tells how much variance is represented in that direction.

The eigenvector with the highest eigeenvalue is the first principal component, and each subsequent eigenvector have depreciating eigenvalues.

The PCA transform is given by:

$$T = XW \tag{2.8}$$

where X is the demonstrated data, W is a p by p matrix whose columns are eigenvectors of X^TX and p is the number of principal components (in this case, the number of dimensions).

To transform to any arbitrary dimension, d, we can choose d eigenvectors from W with the largest eigenvalues to form a p by d dimensional matrix W_d:

$$T_d = XW_d \tag{2.9}$$

We then use reinforcement learning to learn trajectories in the new manifold. All learning is in a lower-dimensional space. For each learning iteration, we project state x down to a lower-dimensional space d:

$$x_d = W_d^T x \tag{2.10}$$

2.2.2 Sparse Random Projection

Sparse random projections reduce the dimensionality by projecting the original space using a sparse random matrix. This technique draws random elements for the transformation matrix from the distribution:

$$T(x) = \begin{cases} -\sqrt{\frac{s}{n}}, & & \frac{1}{2s} \\ 0, & \text{with probability} & 1 - \frac{1}{s} \\ \sqrt{\frac{s}{n}}, & & \frac{1}{2s} \end{cases}$$

where n is the number of components and s is the density recommended by Li et. al [67]: $\frac{1}{n}$.

The projection is identical to PCA and follows Equation 2.9. Since the projection is a simple matrix multiplication, this technique is fast and easy to use.

Linear dimensionality reduction assumes that the underlying data can be represented by a lower dimension linear subspace. However, robot state spaces typically include velocities and accelerations, whose equations of motion are inherently nonlinear. Standard linear dimensionality reduction techniques cannot accurately represent complex nonlinear structures which cannot be well represented in a linear subspace. Later, we explore nonlinear dimensionality reduction techniques to project complex state spaces for learning.

2.2.3 Kernel PCA

Kernel PCA [73] is a non-linear extension of PCA. The basic idea of Kernel PCA is to project the linearly inseparable space onto a higher dimensional space where it can be linearly separable. This nonlinear transformation function is known as the kernel function. The kernel function maps each point x from a d-dimensional space into an n-dimensional space, where $d < n$:

$$x_i \rightarrow \phi(x_i) \text{ where } \phi : R^d \rightarrow R^n \tag{2.11}$$

The kernel uses ϕ to map the original space onto a n-dimensional space by calculating the dot product of the samples:

$$k(x_i, x_j) = \phi(x_i)\phi(x_j)^T \tag{2.12}$$

where k is the kernel that creates nonlinear combinations of the original features. The kernel can be any function, but linear (2.13), d-degree polynomial (2.14), and radial basis functions are all common (2.15).

$$k(x, y) = x^T y \tag{2.13}$$

$$k(x, y) = (\gamma x^T y)^d \tag{2.14}$$

$$k(x, y) = \exp(-\gamma \|x - y\|^2) \tag{2.15}$$

Although Kernel PCA is a powerful dimensionality reduction technique, it will not work in our application. The pairwise comparison between each point is computationally expensive when initially creating the dimensionality reduction, and will be repeated for each out-of-sample point. Since we need Kernel PCA to run for each state-action-reward sample, this pairwise comparison is too expensive and quickly makes this approach computationally intractable.

2.2.4 Dimensionality Reduction for Reinforcement Learning

Previous work in dimensionality reduction focuses on reducing the space for classification or function approximation. Principal Component Analysis (PCA) [55] is effective in many machine learning and data mining applications at extracting features from large data sets [86, 113]. Feature selection, which aims at reducing the dimensionality by selecting a subset of most relevant features, has also been proven to be an effective and efficient way to handle high-dimensional data [26].

In our work, we use PCA to discover the low-dimensional representation of the state space during learning. It does this by computing a transform to convert correlated data to linearly uncorrelated data. This transformation ensures that the first principal component captures the largest possible variance. Each additional component captures the largest possible variance uncorrelated with all previous components. Essentially, PCA linearly represents as much of the demonstrated state space as possible in a lower dimension.

Rather than transferring knowledge from a simple representation to a complex one, Taylor, Kulis, and Sha [108] learn directly which states are irrelevant. They collect data while the agent explores the environment, calculate a state similarity metric, and ignore state variables that do not add additional information and scale those that do. Similarly, Thomas et al. [26] use demonstrations to develop a subset of features from the original space. The learning algorithm then used this subspace to predict the action that a human expert would take. Both approaches find state variables to ignore or emphasize. This works well if there are insignificant state variables, but would have issues where there are infrequent or non-demonstrated state variables with critically important data. In our IDRRL framework, we iteratively add additional state information until it learns in the full state space, essentially combining the speed of low-dimensional learning and the expressiveness of the full state space.

Similar to our approach, Colomé et al. [27] apply dimensionality reduction techniques

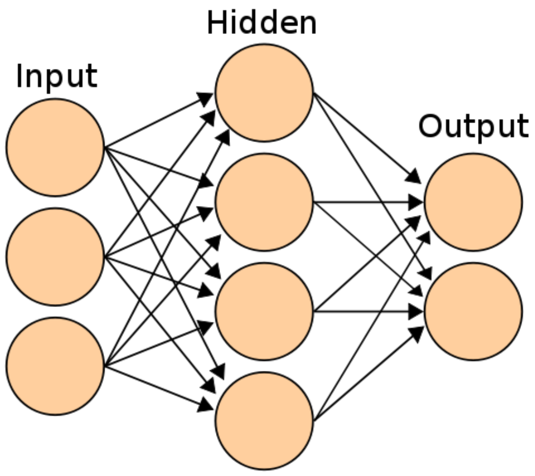

Figure 2.2: The standard structure of a feedforward artificial neural network with one hidden layer.

to exploit underlying manifolds in the state space. They learn probabilistic motor primitives in a low-dimensional space discovered by probabilistic dimensionality reduction. Bitzer et al. [14] also applies dimensionality reduction to solve reinforcement learning planning problems in a reduced space that automatically satisfies their task constraints. Although these approaches greatly accelerate learning, they learn entirely in the low-dimensional space and could discard potentially important data.

In this work, we introduce a fast non-linear dimensionality reduction technique for learning, to better accommodate nonlinearities often found in robot dynamics. To this end, we use neural network autoencoders [9].

A neural network is made up of many connected artificial neurons (Figure 2.2). Each input into a neuron is multiplied by a learned weight and summed together to create the activation value, which is then fed into the next neuron as an input. To limit the growth and size of these activation values, each value is fed into an activation function. Typical activation functions are sigmoid, hyperbolic tangent and gaussian.

Neural network autoencoders reduce the dimensionality of data by transforming high-dimensional data into a low-dimensional space (Figure 2.3). It trains by encoding the high-dimensional data and decoding it back to the original space with the least possible amount of distortion [9]. Hinton and Salakhutdinov found that you can reduce the

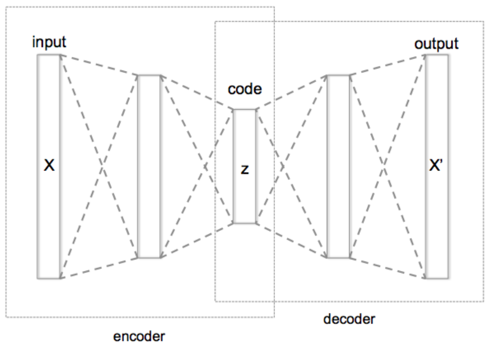

Figure 2.3: The structure of a simple autoencoder. An autoencoder projects (encodes) the input space into a smaller set of neurons (code), then reconstructs its own inputs to find the projection error (decoder). Once trained, the neural network no longer needs the decoder section and users retain only the encoder section.

dimensionality of the dataset by extracting important features [50]. They found that these features were more representative of the full state space than the features found with PCA.

Many of these autoencoder dimensionality reduction techniques take a very large input space and reduce the dimensionality with hundreds of neurons and many layers. In our work, we have a small input space, but need our network construction to be fast, be able to project the data onto many different dimensions, and encode the maximum amount of information in the first dimension, with each additional dimension strictly adding a superset of state space representation. In this dissertation, we introduce a cascade of autoencoders to fit these three requirements.

2.3 Transfer Learning

As RL problems become more complex, basic techniques may become slow or infeasible. A significant amount of research in RL focuses on increasing the speed of learning by taking advantage of certain domain knowledge. Some techniques include agent partitioning, which focuses mainly on how to divide the problem by the state space, actions, or goals [56, 92, 30]; generalizing over the state space with techniques such as tile coding [116], neural networks [47], or k-nearest neighbors [72]; and learning with temporally defined actions, such as options [105]. In contrast, we do not wish to leverage knowledge within the same problem, but instead transfer knowledge from a low-dimensional problem to a higher-dimensional one.

The core idea of transfer learning is that experience gained in learning to perform one task can help improve learning performance in a related, but different, task. If the relationship between the first (source) task and the second (target) task is not trivial, there must be a mapping between the two tasks so that a learner can apply the older knowledge to the new task [109].

The type of knowledge transferred is based on the precision required by the algorithm. High-precision knowledge, such as $< s, a, r, s' >$ rollouts or the learned Q-Function can help fully define an initial policy in the target domain. However, these techniques require accurate mappings between the source and target task. Higher-level feedback, such as action advice or shaping rewards are less restricting, but less detailed. In this work, we have an accurate mapping between the source and target task, so we only focus on the high-precision knowledge transfer.

2.3.1 Multi-Task Learning

Multi-Task Learning (MTL) problems assume that all domains experienced by the agent are from the same distribution [109]. This corresponds to the source and target task have the same state and action representation. MTL approaches are particularly suited for tasks that remain the same, but with varying dynamics, such as friction.

MTL techniques can greatly speed up learning in tasks with differing parameters but the same representation. For example, Dimitrakakis and Rothkopf used multi-task learning to control an inverted pendulum, where the dynamics of the pendulum was varied across tasks [40]. Deisenroth et al. applied multi-task policy search to control a robotic manipulator. The arm was required to stack blocks in tasks of varying complexity [32].

Sutton et al. demonstrated that this technique can be used in a divide-and-conquer approach [104]. They suggest that the problem can be split into a sequence of subtasks. The learner can learn the first subtask, and transfer that knowledge to the next subtask. This approach is similar to our Dimensionality-Reduced Reinforcement Learning in spirit.

2.3.2 Inter-Task Learning

It is not always possible for the source and target domain to be from the same distribution. An inter-task mapping is a general structure that defines how two tasks are related. The mappings χ_S and χ_A are defined as a mapping between state variables and actions in two tasks, respectively. This technique [109].

[110] developed inter-task mappings for policy search methods and learned these mappings when they were not available. They test their approach in the RoboCup Keepaway domain with a varying number of agents, leading to a varying number of state variables.

In this work, we transfer from a low-dimensional representation to a higher-dimensional representation. Therefore, the states between representations are not the same. Additionally, the number of state variables we represent change as we change the number of dimensions in the problem. To perform the state projection, we compute a mapping using dimensionality reduction. This mapping is χ_S, which is what we use during transfer. Our action representation remains the same, and therefore we do not need to compute χ_A.

2.4 Learning from Demonstration

Developing a policy using traditional straightforward reinforcement learning does not work well in robotics. Finding an optimal or near-optimal solution requires exploration throughout much of the state space and excessive exploring of the unknown state space risks damaging the robot. This can be avoided by taking small steps during exploration, but this brings the additional problem of taking longer to find an optimal solution. Initializing, or bootstrapping, the policy close to the desired robot behavior removes many of these problems. One particular approach to policy initialization is Learning from Demonstration (LfD).

LfD learns a policy using examples or demonstrations provided by a human or a robotic teacher. These examples are typically state-action pairs that are recorded during the teacher's demonstration. These state-action samples are used to initialize a policy that can then either be directly utilized, or improved using reinforcement learning [6].

There are two main divisions of research within LfD: developing methodology for acquiring the demonstration data, and deriving policies from this data. The data can be acquired from many different sources, such as another robot, another human, a human controlling a robot, or even a human controlling the learner robot. The data acquisition choice heavily affects the difficulty of mapping from real-world data to state-action pairs and will be discussed in Section 2.4.1.

There are a variety of techniques that LfD algorithms use when deriving a policy. Machine learning directly maps the demonstrated data to a policy to be sampled (Section 2.4.2.1). This approach attempts to directly learn the teacher's demonstration. On the other hand, reinforcement learning uses the demonstrated data to guide learning, rather than directly learning the demonstration (Section 2.4.2.2). The demonstrated data can be used to initialize a policy, develop a reward function, and build a state transition function. This approach takes the demonstrated data and attempts to optimize it.

2.4.1 Data Acquisition

When gathering demonstration data, the choice of demonstrator greatly impacts the complexity. If the teacher is directly controlling the learning robot, the learner's execution can be directly used as demonstration data. However, if the learner is required to observe, the learner must sense the teacher's execution and convert it to demonstration data. This can lead to a large computational overhead. Additionally, if the learner is not physically similar to the teacher, it may also have to convert demonstration data to its own action and state space, adding more computational complexity.

There are four different demonstration approaches at different levels of complexity: teleoperation, mimicking, sensors on the teacher, and simply observing the teacher. Teleoperation removes any complex mapping from the system, making it the most direct approach for data acquisition. However, teleoperation assumes that the teleoperator has the ability to properly control the low-level commands of robots. With higher degree of freedom robots, this teleoperation can become very difficult to perform accurately.

Teleoperation is one of the most commonly used data acquisition techniques [6] and is often performed by a human with a joystick. Some examples include using teleoperation demonstrations to control an inverted autonomous helicopter [79], robotic grasp modeling [107], object grasping [90], and simulated driving [1].

High-level teleoperation such as speech recognition and kinesthetic teaching can also be employed with great success. Natural language processing has been used to allow a human teacher to verbally teach the robot how to navigate a miniature town [66], to clarify ambiguous demonstrations [19], and general task training [95]. Kinesthetic teaching involves humans physically moving the robot's passive joints through the desired motions [2].

Mimicking requires the learner to observe the demonstrator, convert the demonstrator's execution to actions, and then execute those actions. This assumes the learner's set of actions are the same as the demonstrator's set of actions. Therefore, the only additional computation comes from observing and mapping from the teachers movements to demonstration data. Mimicking has been used analyze how biological animals learn through imitation [35], navigation tasks [77, 81], and to learn arm gestures [82].

By adding sensors directly onto the demonstrator, the learner no longer has to observe the teacher. Rather than having the learner observe the teacher, the teacher records its own execution with sensors. This is similar in computational complexity to mimicking, except the extra computation involves mapping from the demonstration data to the learner, rather than from the demonstrator to the demonstration data. This is especially useful when the application requires precise measurements of the teacher. Researchers have used this technique to teach robots tennis swings [51] and walking patterns [76] by using human teachers wearing sensors.

Lastly, the most general and computationally expensive data acquisition approach involves the robot observing the teacher. The learner must extract the teachers state-action pairs from the observed demonstration. Those state-action pairs then need to be mapped to the learners set of actions. This leads to an imprecise, computationally expensive and less reliable approach than other data acquisition methods [6]. However, this approach is the most general of all techniques. Mimicking requires the learner's set of actions to match the teacher's set of actions, while adding sensors to the teacher requires expensive sensors to accurately obtain demonstration data. Visual features analyzed with computer vision have been used to teach human movement, grasping tasks, and hand gestures [114, 4, 13]. Hybrid approaches combining computer vision with sensors on the teacher or speech recognition have also been successfully used. A force sensing glove on the teacher combined with computer vision motion tracking has been used to teach grasps and other motor skills [68, 70, 115] and visual cues combined with speech recognition has been used to learn human gestures and object tracking [36, 101].

The choice of approach is situationally dependent on the tools available and the users. Teleoperation is useful in low DoF robots and requires only some training and low cost tools. Mimicking, sensors on the teacher, and observation all require another robot or human to first perform the task, which may not always be possible.

2.4.2 Policy Derivation

Now that the learner has acquired demonstration data from the teacher, it can develop a policy. Many policy derivation approaches exist, but this dissertation will focus on machine learning and reinforcement learning. Machine learning attempts to develop a policy as close to the teacher's demonstration as possible while generalizing to states unseen by the teacher, and reinforcement learning uses the teacher's policy to initialize the policy, help develop the reward function, or build a state transition function, and then applies reinforcement learning techniques to optimize performance of the task.

2.4.2.1 Machine Learning

Machine learning approaches use function approximation to develop a mapping from the demonstration states to actions taken. There are two ways to accomplish this: classification and regression. Using both techniques, the learner can reproduce the teacher's policy and generalize to states not encountered by the teacher. These function approximations take the current state as input, and output a action. The key difference between classification and regression is small. Classification takes the set of state inputs and outputs discrete class labels, and regression takes the set of state inputs and outputs a continuous value. In the terms of policy derivation, this maps to discrete or continuous actions. Therefore, classification works well in applications where the robot executes high-level commands, such as "turn the knob" or "open the door", or the action space can be accurately discretized. Regression is a more general technique and works well with low-level motor primitives which need a continuous input. Each approach has benefits and issues, as we will now discuss.

Classification techniques require that the action space be discretized into a specific number of actions. This greatly reduces the generality of the approach. If the action space is not discretized enough, the policy will not adequately match the teacher's demonstration. If the action space is too discretized, the policy may not converge and will need more data. However, if the actions are discretized correctly, this approach can learn a policy with much less data than with regression.

Classification approaches use low-level actions include controlling a simulated car using Gaussian Mixture Models and demonstration requests for unknown parts of the state space [23], obstacle avoidance using interactive teaching methods [53], and using bio-inspired memory maps and k-Nearest Neighbors to construct dynamically sized action-selection mechanisms [97].

High-level actions are useful when sequences of low-level actions have already been developed. It is much easier to create a policy to accomplish a large task when the action selections represent a sequence of already learned low-level actions. This approach has been used to perform a box sorting task by classifying gestures representing prior learned sequences of actions [96], start a collaborative social dialogue with a robot and a human to learn a button pressing task [69], and to classify behaviors to perform a multi-robot ball rolling task by dynamically determining the need for new demonstrations [24].

Regression techniques are more general than classification methods, as the outputs are continuous actions. Different choices arise when using regression. Rather than choosing how to discretize the action space, in regression the designer must choose when to perform the policy function approximation. If the approximation is done during run time, the regression does not need to expend computation to discover state-action pairs in states that are never visited. However, all of the training data must be stored in memory. This is known as Lazy Learning [7]. If the approximation is done before run time, the entire function approximation must be performed using the entire demonstration data set, including states that will never end up being visited in practice. This is much more computationally expensive, but is generally performed off-line.

Lazy Learning techniques have been successfully used in practice. Some examples include using human policy critiquing to successfully intercept a ball [5], learning rhythmic patterns that robustly cope with external perturbations [52], and learning biped walking patterns [76]. In large problems with a continuous action and state spaces, the amount of demonstration data to process at every time step becomes too computationally complex for real-time robotic movement.

There are many approaches that perform policy function approximation before run time, with the most common being Neural Networks [114, 39, 74, 89]. Neural Network function approximation has been used to teach a robot the game of Kendama, a game similar to ball in a cup [74], control ALVINN (Autonomous Land Vehicle In a Neural Network) to drive in single-lane paved and unpaved roads as well as multi-lane lined and unlined roads [89], performing a peg in a hole task by preprocessing bad demonstrations to make them suitable for learning [39] and learning human full-body motion kinematic patterns [114]. Other approaches have used Gaussian Mixture Regression to teach human gestures to a humanoid robot via sensors on the teacher [22] and Sparse On-Line Gaussian Processes to teach an AIBO robot soccer skills through teleoperation [45]. Even though function approximation of the entire dataset is a computationally complex approach, it moves the processing from being real-time on the robot to before the execution, making this approach more ideal for real-time robotics in complex problems.

2.4.2.2 Reinforcement Learning

As mentioned in Section 2.1, reinforcement learning is a tool within the field of multiagent or single-agent learning where agents take an action, observe the environment, and receive a reward based on the new environment [103]. This approach requires exploration in order to improve upon the current policy, but in robotics, exploration is hazardous to the robot. Alternatively, the robot can learn in simulation, but that requires an accurate state transition model of the world, $T(s'|a, s)$. When applying reinforcement learning with LfD, the teacher's demonstration can help with both approaches. If the robot is learning in the real world, the teacher's demonstration can be used to initialize the reinforcement learning policy, and therefore learning can take small exploratory actions to improve upon this already accurate policy. If the robot is learning in simulation, the teacher's demonstration can be used to both initialize the policy, and be used as data to construct a state transition function. Reinforcement learning approaches include value function and policy search, which have both been explained in Section 2. In addition to policy initialization and state transition derivation, the teacher's demonstration data can be used to assist in engineering the reward function in both standard and inverse reinforcement learning.

In standard reinforcement learning, the reward function is hand-created by the user. Giving high rewards to states around the goal, and low rewards to states around obstacles are intuitive choices, but it leads to a sparse reward function. States that are not directly nearby obstacles or the goal are arbitrarily explored, and can lead to damage to the robot. Teacher demonstrations can be used to prevent blind exploration caused by sparse reward functions by showing the robot rewarded areas of the state space [100]. Learners have also been taught to ask for demonstration help when they are in an arbitrary area of the state space [21, 25] or exchange advice with other agents [83]. Lastly, the user can weigh the reward function based off of whether the teacher had visited the state [83]. All of these approaches attempt to alleviate the issues of arbitrary action selection and sparse reward functions.

An alternative approach is to not build the reward function manually. Inverse Reinforcement Learning is a field within Reinforcement Learning that learns the reward function. LfD can assist in learning the reward function by analyzing the teacher's demonstrations. Approaches include simply associating higher rewards to states similar to those during the demonstration [8], developing a reward function based off of the similarity between the demonstrated and learner executed policy [84], and learning rewards based purely on human feedback [112].

2.5 Human Feedback

Even with training speed improvements, reinforcement learning can take a long time to converge. Human intervention can accelerate this learning process. There are two key steps for using human feedback: the mechanism for giving the feedback and how that feedback is integrated into the learning process.

2.5.1 Human Feedback Mechanisms

Human feedback mechanisms are tools that allow users to give feedback to an agent. The agent can use this feedback to modify its learned policy based on user preference. It then executes the newly learned policy, and the user can give additional feedback. This cycle continues until the agent has learned an adequate policy, as defined by the user. Researchers have developed a variety of these tools, and focused on making them both intuitive and detailed. Intuitive feedback tools tend to use mechanisms that are natural to users, such as voice [21], kinesthetics [2, 21] or drawing [16], while detailed mechanisms use graphical interfaces that are less natural, but allow for high-fidelity feedback [46, 112]. However, many feedback mechanisms are not detailed enough for fine-grained feedback, do not account for continuous spaces, or require the user to give time-sensitive feedback.

The work by Cakmak and Thomaz used Active Learning (AL) and Learning from Demonstration (LfD) as a mechanism for natural feedback [21]. In that work, the robot could query the user to guide exploration and learning, while the user could also query the robot to better understand what was being learned. The robot could also ask the user to kinesthetically demonstrate how to perform tasks in areas of the state space of high uncertainty. Boniardi et al. also use this same intuitive style human feedback [16]. Participants give hand-drawn sketches that are a high-level description of the environment. This allows the robot to navigate when it is not given a full description of the scene. While intuitive and time-insensitive, these approaches do not allow the user to give detailed feedback on important parts of the state space. In our work, the movie reel interface allows the user to give fine-grained feedback.

Rather than use natural feedback, in the work by Harutyunyan et al. the humans interactively advise a video game agent via a button that gives the agent positive rewards [46]. The human gives advice for the first 5 episodes, and then the agent continues learning on its own for the rest of the trial. The authors find that with this initial advice, the Mario agent learns a better policy at a faster rate. Thomaz and Breazeal developed a more advanced interface that lets a human reward the agent with a variety of rewards during policy execution [112]. In that work, the human can, at any point in the operation, reward the agent with a scalar reward between -1 and 1. This allows the human to give continuous feedback over a range of states. However, both of these works require quick actions by the human. The human must be paying close attention during the policy execution, and can miss giving feedback on early actions that affect later states. Our movie-reel interface alleviates this issue by using fast forwarding, rewinding and pausing.

2.5.2 Human Feedback Integration

Once a human has given feedback, there are a variety of techniques that integrate this feedback into the learning algorithm. Griffith et al. developed a policy shaping technique called Advise [44] that leverages human feedback directory by using the feedback as policy labels. They assume that the human adviser knows the different optimal actions in a state and the state transition of taking an action. They then use these policy labels to guide learning. Advise effectively incorporates human feedback into the learning process, but requires the user to understand the effects of taking an action, and knowing whether those effects are positive or negative. In our work, we require feedback on motor primitives. Motor primitive actions do not have an intuitive state transition, especially in higher dimensional problems.

Knox and Stone developed *Action Biasing* and *Control Sharing* strategies for using human feedback to modify the policy [61]. *Action Biasing* uses positive and negative human feedback to bias the action selection mechanism of Q-Learning. They add the human feedback reward to the learned Q-Value when choosing the next action to take. Alternatively, *Control Sharing* modifies the action selection mechanism by transitioning between using the best action during feedback and the learned best action.

Chapter 3: Experimental Domains

In this section we will describe the experimental domains we used throughout the dissertation. Early in prototyping this research we used two toy domains, Mountain Car and Mario (Section 3.1 and Section 3.2). In Mountain Car the agent learns how to drive an underpowered car up a steep hill and in Mario the agent learns how to play the Mario video game. Later in this work we used more compelling experiments to prepare for real-world experiments. For scalability experiments we used Swimmers, where the agent learned to control the speed and position of an underwater n-link fish using motor torques (Section 3.3). We control the difficulty of the problem by scaling up the number of links the agent can control. Last, for our real-world experiment we taught a robot arm how to balance a ball bearing on an acrylic plate (Section 3.4).

3.1 Mountain Car

To test the efficacy of the algorithms introduced in this work, we apply it to the Mountain Car 3D domain, a common reinforcement learning benchmarking problem [41]. In Mountain Car, an underpowered car must drive up a steep hill (Figure 3.1). The problem is engineered such that the car cannot overcome the effects of gravity, and cannot simply drive up the hill. Since the car starts in a valley, the agent must learn to build up enough inertia by driving partially up the opposite hill before it is able to make it to the goal.

In 2D Mountain Car, there are two states defined as the continuous position ($-1.2 \leq x \leq 0.6$) and velocity ($-.007 \leq v \leq .007$) of the car. There are three actions: Accelerate left, accelerate right, and neutral. The starting state is a random position at a random velocity. Lastly, the reward is -1 at each time step, and 100 at the goal. The 3D variant of Mountain Car incorporates these dynamics, but there are also two new states (position and velocity) and two new actions (accelerate/decelerate) to represent movement across

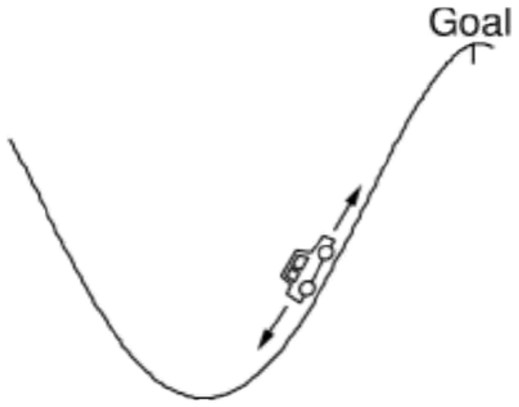

Figure 3.1: Mountain Car. An underpowered car learns to drive up a steep hill by using a shorter hill to build up enough inertia.

the third dimension. The car moved according to the following dynamics:

$$\ddot{x} = Action$$
$$\dot{x} = \dot{x} + 0.001 * \ddot{x} + -0.0025 * cos(3 * x)$$
$$x = x + \dot{x}$$

3.2 Mario Benchmark Problem

The Mario benchmark problem [59] is based on Infinite Mario Bros, which is a public reimplementation of the original 80's game Super Mario Bros®. In this task, Mario needs to collect as many points as possible, this is done by killing an enemy (10), devouring a mushroom (58) or a fireflower (64), grabbing a coin (16), finding a hidden block (24), finishing the level (1024), getting hurt by a creature (−42) or dying (−512). The actions available to Mario correspond to the buttons on the NES controller, which are (left, right, no direction), (jump, don't jump), and (run/fire, don't run/fire). Mario can take one action from each of these groups simultaneously, resulting in 12 distinct combined or 'super' actions. The state space in Mario is complex, as Mario observes the exact locations of all enemies on the screen and their type. He also observes all information about himself, such as what mode he is in (small, big, fire). Lastly, he has a gridlike receptive field in which each cell indicates what type of object is in it (such as a brick,

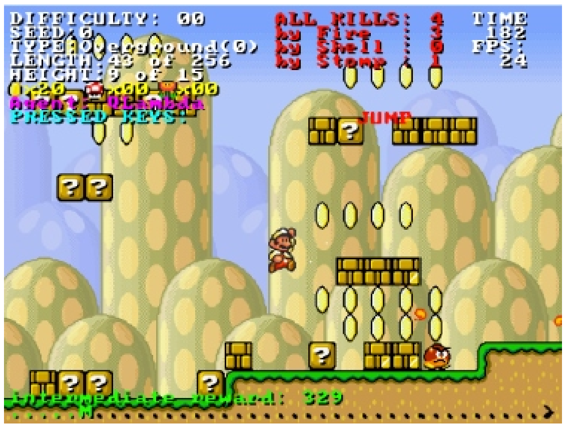

Figure 3.2: A screenshot of Mario.

a coin, a mushroom, a goomba (enemy), etc.). A screenshot is shown in Figure 3.2.

The part of the state space the agent considers consists of these variables:

- is Mario able to jump $(0 - 1)$
- is Mario on the ground $(0 - 1)$
- is Mario able to shoot fireballs $(0 - 1)$
- Mario's current direction, 8 directions and standing still $(0 - 8)$
- enemies closeby (within one gridcell) in 8 directions $(2^8 \rightarrow 0 - 255)$
- enemies at midrange (within one to three gridcells) in 8 directions $(0 - 255)$
- whether there is an obstacle in four vertical gridcells in front of Mario $(2^4 \rightarrow 0 - 16)$
- closest enemy position within 21x21 grid surrounding Mario + 1 for absent enemy $(21^2 + 1 \rightarrow 0 - 441)$

This makes for 3.24×10^{10} possible states, and 4×10^{11} Q-values (12 actions in each state). The size of the state space is large, but some states are more important and frequent than others. This variable state importance makes the Mario Benchmark Problem good for analyzing our dimensionality reduction approach.

3.3 Swimmers

The Swimmers domain [28] is a more complex system than Mountain Car, and serves as an abstraction of a robot arm positioning problem because the goal is to control position through a simulated viscous fluid using motor torques. It includes complex physics with a large state and action space. In the Swimmers domain, there is a simple swimmer (Figure 3.3) connected by joints that moves in a two dimensional pool. The action space is a torque applied at each joint. The goal of the Swimmers domain is to swim as fast as possible to the right, by using the friction of the water.

We define the state of the swimmer as the angular position and velocity at each joint, as well as the center x- and y-velocity. Therefore a n-link swimmer has $2n + 2$ states. The action space consists of the $n - 1$ control torques at each joint. At each control step the learner chooses between a $-3Nm$, $0Nm$ or $3Nm$ torque, making $3^{(n-1)}$ actions. We reward the swimmer for moving as fast as possible to the right (Δx). In this work we use a 3-link and 6-link swimmer.

To move the swimmer, the reinforcement learner must learn to control an n-link object using the torques at each joint. It must learn to leverage the viscous friction and learn the nonlinear dynamics of the system. The state and action spaces are identical and involve passive dynamics acting on the agent.

We use the Swimmers domain as an abstraction of a robot arm that we can scale to high dimensionality. Since we can arbitrarily scale the complexity of the Swimmers, we can find how our algorithms scale. Swimmers also has a similar state and action space to a robot arm. The reinforcement learner must learn to control an n-link object using the torques at each joint. The Swimmers experiments also include a model of viscous friction. The reinforcement learner must learn to take advantage of the viscous friction of the water to move.

Coulom [28] modeled the system dynamics of the swimmer according to the following. Let G_i be the center of mass in between each pair of links A_{i-1} and A_i:

$$G_i = \frac{A_{i-1} + A_i}{2}, \tag{3.1}$$

and f_i be the force applied by segment $i + 1$ to segment i and :

$$\forall_i \in \{1, ..., n\} : -f_{i+1} + f_i + F_i = m_i \ddot{G}_i, \tag{3.2}$$

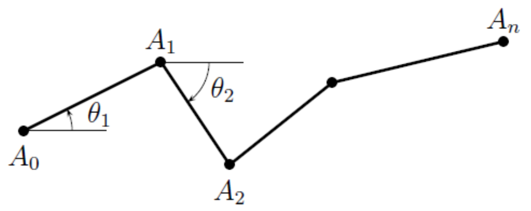

Figure 3.3: N-link swimmer. The swimmer must learn to leverage the viscous friction of the water to move quickly to the goal.

we can now express the state variables as:

$$f_0 = 0$$
$$\forall_i \in \{1, ..., n\} : f_i = f_{i+1} - F_i + m_i \ddot{G}_i,$$

and the system of equations becomes a set of $n + 2$ linear equations, where n is the number of links:

$$\begin{cases} f_n = 0, \\ m_i \frac{l_i}{12} \ddot{\theta} = det(G_i A_i, f_i + f_{i-1}) + \omega_i - u_i + u_{i-1} \end{cases}$$

where ω_i is the moment at point G_i and u_i is the torque that the agent applies to joint i. Lastly, Coulom's model of viscous friction calculates the total force and moment at each moving part:

$$\omega_i = -k\dot{\theta}_i \frac{l_i^3}{12},$$

where l_i is the length for segment i and k is the viscous-friction coefficient. For all experiments we used standard values of $l_i = 1$, $m_i = 1$, and $k = 10$.

3.4 Ball Balancing

To experimentally validate our learning approach on a real-world robotics platform, we teach a robotic arm with a plate attachment to balance a ball bearing. We used a 6 degree-of-freedom Yaskawa Motoman MH5F with a F1000 controller with an custom plate end-effector (Figure 3.5). Each link on the robot is a parent to its successor child link, connected by a constrained revolute joint which maintains a unique reference frame governing its successor links. In our domain, this robot arm had to learn to balance a 1/8 inch ball bearing on a .25m x .5m plate.

We used Robot Operating System (ROS) to model the robots joints, joint limits, and kinematics to ensure accurate motion. ROS [91] is a collection of packages that provides an abstract and fast framework for robot software development. Traditionally, software development in robotics had been entirely custom-made for each robot, slowing research and development. Quigley et al. [91] developed ROS to avoid this problem by using a generic communication infrastructure between high-level and low-level hardware, essentially abstracting away the hardware layer and allowing users to use the same code on many different robots.

ROS has a directed graph architecture, where edges represent messages and nodes represent processes. Nodes perform computation and communicate with each other by passing strictly-typed data structures called messages. Low-level hardware nodes either publish messages, such as laser scanner data, or subscribe to messages, such as motor controls. A roboticist typically builds a high-level node in ROS by subscribing to messages published by low-level hardware nodes, performing some computation, and publishing its own message, which is in turn subscribed to by another node. This makes ROS highly modular and abstract, promoting encapsulation and code reuse.

We define the state of the arm as the angular position and velocity of 4 joints and the position and velocity of the ball bearing in each dimension. Of the 6 joints, two corresponded to yaw, which were not useful for the learning problem. For each control step the learning algorithm chooses an angular acceleration (-0.5$\frac{\theta}{s^2}$, 0.5$\frac{\theta}{s^2}$) to apply to one of the degrees of freedom. The agent must learn to use these 12 state dimensions and 8 actions to balance the ball for as long as possible. The agent received a reward of positive 1 for each time step the ball was on the plate and a reward of -1000 if the ball falls.

For ease-of-application we used a simulated robot for learning of the policy (Figure 3.4) and a physical robot for the execution (Figure 3.5). For each simulated experiment, we use forward kinematics to determine the position (θ) and velocity (v) of each degree-of-freedom i at time t at a resolution of $\Delta t = 0.01$ (Equations 3.3).

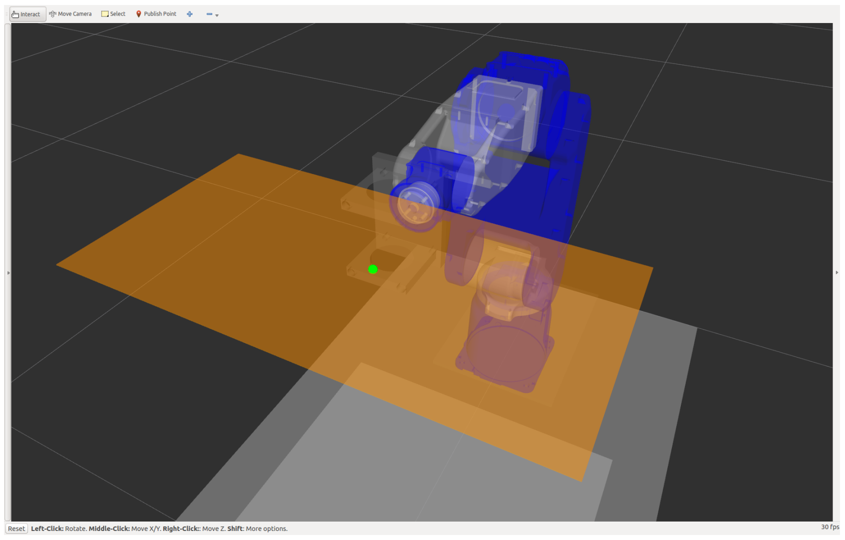
Interact
Move Camera
Select
Publish Point
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click: Move Z. Shift: More options.
30 fps

Figure 3.4: Simulated Yaskawa Motoman MH5F Arm balancing a simulated ball (green) on a frictionless plane (orange).

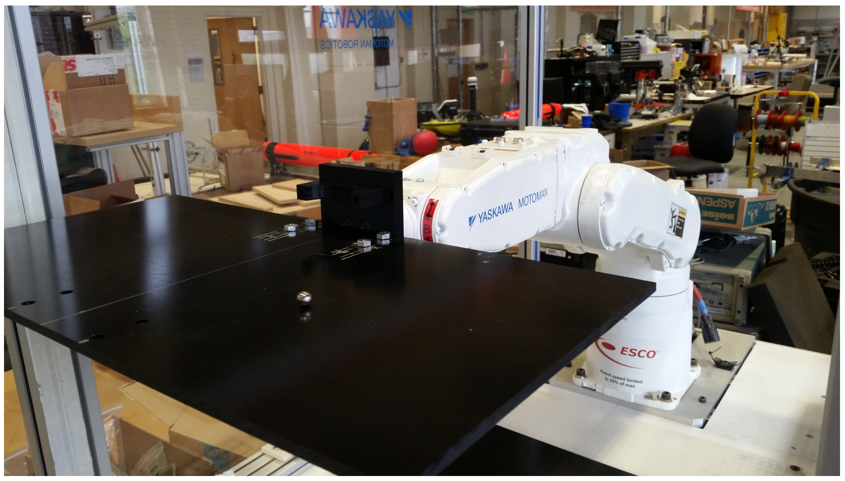

Figure 3.5: Physical Yaskawa Motoman MH5F Arm balancing a ball bearing on an .25m x .5m acrylic sheet.

$$\begin{aligned}\ddot{\theta(i)}_t &= action(i)\\ \dot{\theta(i)}_t &= \dot{\theta(i)}_{t-1} + \ddot{\theta(i)}_t * \Delta t\\ \theta(i)_t &= \theta(i)_{t-1} + \dot{\theta(i)}t - 1 * \Delta t + \frac{1}{2} * \ddot{\theta(i)}_t * \Delta t^2\end{aligned} \tag{3.3}$$

We also used forward kinematics to determine the position and velocity of the ball in the x and y plane. We determined the plate angle from the roll and pitch angles of the arm (Equation 3.4).

$$\begin{aligned}pitch &= \theta(2) + \theta(4)\\ roll &= \theta(3) + \theta(5)\end{aligned} \tag{3.4}$$

The ball slid on a frictionless plane according to standard forward kinematics (Equations 3.5 and 3.6)

$$\begin{aligned}\ddot{x}_t &= g * sin(roll)\\ \dot{x}_t &= \dot{x}_{t-1} + \ddot{x}_t * \Delta t\\ x_t &= x_{t-1} + \dot{x}_{t-1} * \Delta t + \frac{1}{2} * \ddot{x}_t * \Delta t^2\end{aligned} \tag{3.5}$$

$$\begin{aligned}\ddot{y}_t &= g * sin(pitch)\\ \dot{y}_t &= \dot{y}_{t-1} + \ddot{y}_t * \Delta t\\ y_t &= y_{t-1} + \dot{y}_{t-1} * \Delta t + \frac{1}{2} * \ddot{y}_t * \Delta t^2\end{aligned} \tag{3.6}$$

Chapter 4: Dimensionality-Reduced Reinforcement Learning

State spaces in robotics problems are often tremendously large as they scale exponentially with the number of state variables and often are continuous. This challenge of exponential growth is often referred to as the curse of dimensionality [11]. As the number of dimensions grows, exponentially more data and computation are needed to cover the complete state-action space. For example, if we assume that each dimension of a state-space is discretized into 10 levels, we have 10 states per dimension of the state space, for 10^n total states. Evaluating every state quickly becomes infeasible with growing dimensionality, even for discrete states.

The curse of dimensionality is a notoriously difficult issue in Reinforcement Learning. In this dissertation we introduce two algorithms: Dimensionality-Reduced Reinforcement Learning (DRRL) and Iterative DRRL. Both techniques use Principal Component Analysis to project the state space down to a lower-dimensional manifold. Reinforcement Learning in this smaller state space reduces the exploration, accelerates the learning rate, and helps alleviate the curse of dimensionality.

However, if the reduced state space does not fully represent the full state space, the agent cannot learn an optimal policy (Section 4.1). IDRRL is the iterative version of DRRL. After the agent learns a policy quickly in the low-dimensional space we transfer that knowledge to the next, more complex, dimension. Learning continues in this higher-dimension and is bootstrapped with knowledge learned from earlier dimensions. This cycle continues until IDRRL is learning in the full state space (Section 4.2).

We show that our optimization speeds up learning when added to an existing algorithm, but we wanted to ensure this addition did not remove any existing PAC-MDP performance guarantees. We show that DRRL and IDRRL maintains polynomial state, action and sample complexity, thus retaining PAC-MDP guarantees (Section 4.3).

Our early work used Principal Component Analysis (PCA) as the dimensionality reduction technique. This technique captured much of the low-dimensional state space in DRRL and IDRRL, leading to faster learning. However, it did not capture the non-linearities involved in complex robot state spaces. Alternative non-linear dimensional-

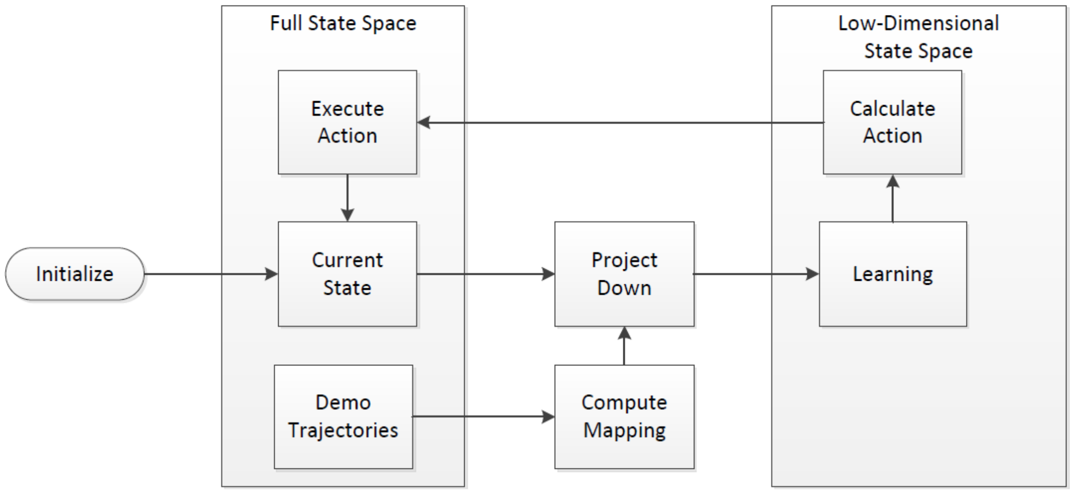

Figure 4.1: One iteration of the DRRL algorithm.

ity reduction techniques were far too computationally expensive to use in this learning framework. This limited the performance of IDRRL when dynamics were highly nonlinear. We developed an extension to IDRRL to use our novel Cascade of Autoencoders (CAE) dimensionality reduction technique. This is a nonlinear dimensionality reduction technique with the same guarantees as the PCA dimensionality reduction (Section 4.4).

4.1 Dimensionality-Reduced Reinforcement Learning

To learn in high-dimensional state spaces, our algorithm first computes a mapping between the full space and a lower-dimensional space. To perform this computation, we need trajectories across a representative set of the agent's state space. We can then use a dimensionality reduction technique to learn the transform. In this work, a variety of dimensionality reduction techniques are explored, but we use Principal Component Analysis (PCA).

First, we project the state down onto a low-dimensional manifold. We then compute the action using a reinforcement learning algorithm, and execute that action. The environment calculates the new state given the executed action, which we then project that state down to the same lower-dimensional space. We can then perform a learning update (Algorithm 3 and Figure 4.1).

Algorithm 3 Dimensionality-Reduced Reinforcement Learning

```
1: function DIMENSIONALITY-REDUCED REINFORCEMENT LEARNING($d$)
2:     Initialize $Q_d$ arbitrarily
3:     for $e \leftarrow 1$ to $episodes$ do
4:         $s \leftarrow initialize$
5:         for $t \leftarrow 1$ to $timesteps$ do
6:             $s_d \leftarrow DimensionalityReduction(s)$
7:             $a \leftarrow \max_a Q_d(s_d, a)$ with probability $1 - \epsilon$ otherwise $a \leftarrow$ random
8:             Take action a, observe $r$ and $s'$
9:             $s'_d \leftarrow DimensionalityReduction(s')$
10:            $r \leftarrow GetReward(s')$
11:            $Q_d(s_d, a) \leftarrow Update(s_d, a, s'_d, r)$
12:            $s \leftarrow s'$
13:        end for
14:    end for
15: end function
```

When learning in a smaller space, reinforcement learning algorithms converge faster. However, in most cases a low dimensional manifold cannot represent the entire state space. Therefore, even given infinite time, the converged learning performance in a non-trivial case will be strictly worse than learning in the full space. This is based on the reprojection error. The reprojection error represents how much data is lost during the transformation into the subspace. You compute the reprojected point by transforming the point into the low-dimensional subspace, then project it back up to the full space. The error is the difference between the original point and the reprojected point.

If the reprojection error is low, then no information is lost in the transformation, and the reinforcement learning algorithm improves speed at no cost. Otherwise, there is some information loss. This leads to a trade-off. By projecting onto a lower-dimensional manifold, we are throwing away potentially important data. Even so, we experimentally validate that with our DRRL framework, learning can still converge to a good policy much faster than the reinforcement learning algorithm alone.

4.1.1 Results and Analysis

To analyze the efficacy of DRRL, we test it in the Mario and Mountain Car domains. Mario has a large discrete state space, and will show the scalability of DRRL. We first test the applicability of using dimensionality reduction in learning by running PCA on a variety of Mario demonstrations and analyzing the principal components. We then demonstrate in both Mario and Mountain Car how DRRL learns faster than standard Q-Learning, but converges to a worse performance.

4.1.1.1 Mario Benchmark

As a preliminary analysis, we calculated all the principal components of the Mario domain to see which dimensions PCA weighed the highest during learning (Figure 4.2). The *jump*, *ground* and *current direction* features are heavily represented in the first few principal components. This is intuitive, as this state changes frequently throughout a game of Mario. These features are also fundamental skills required to play a game of Mario.

PCA also associated features related to enemies within close proximity to Mario entirely in the last few principal components. These features are only important in very specific scenarios where Mario needs to quickly react to many nearby enemies. If only one enemy is nearby, it is also represented in the *closest enemy X* and *closest enemy Y* features.

This analysis legitimizes our approach in the Mario Benchmarking Domain. It demonstrates that we should initially learn using the fundamental skills required to play Mario. We will show that we can learn these skills quickly. The skills represented in the higher principal components are better for strict optimization in specific scenarios. Analyzing the principal components of the demonstrations is a sanity check as well as validation of the richness of the demonstration.

Our reinforcement learning agent for Mario is inspired by Liao's and Brys' previous work [20]. We use a $Q(\lambda)$-learner with a tabular state representation and α=0.01, λ=0.5, γ=0.9 and ϵ=0.05. In the experiments, we run every learning episode on a procedurally generated level based on a random seed $\in [0, 10^6]$, with difficulty 0. Also, we randomly select the mode Mario starts in (small, large, fire) for each episode. Making an agent

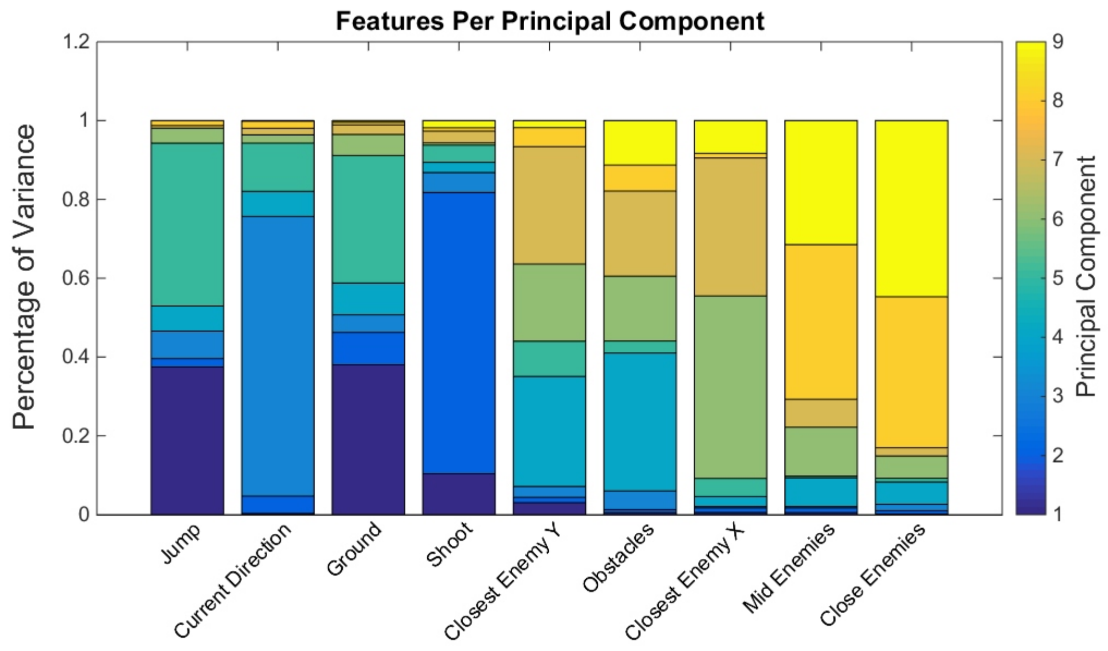

Figure 4.2: The emphasis of each feature relative to the principal component.

learn to play Mario this way helps avoid overfitting on a specific level, and makes for a more generally applicable Mario agent. Our results are always averaged over 100 different trials.

When using our approach in the Mario domain, results were as expected. By projecting the state down to a low-dimensional manifold (less than 4), the learning algorithm converged quickly to bad policies (Figure 4.3). However, when using a manifold of 4 dimensions or greater, we converged quickly to a much better policy. These are promising results although these well-performing dimensions may still converge to a suboptimal policy after more than 5000 episodes.

Since learning was poor within the first two manifolds, it shows us that the *jump* and *ground* features are not informative enough alone to learn an effective Mario policy. Yet, when projecting onto a 3 or 4 dimensional manifold, we see large increase in policy performance. In these manifolds, the features *jump*, *ground*, *current direction*, *shoot*, *closest enemy Y*, and *obstacles* are all represented. It is intuitive that these features are important for having basic skill in Mario. The remaining features are important, but

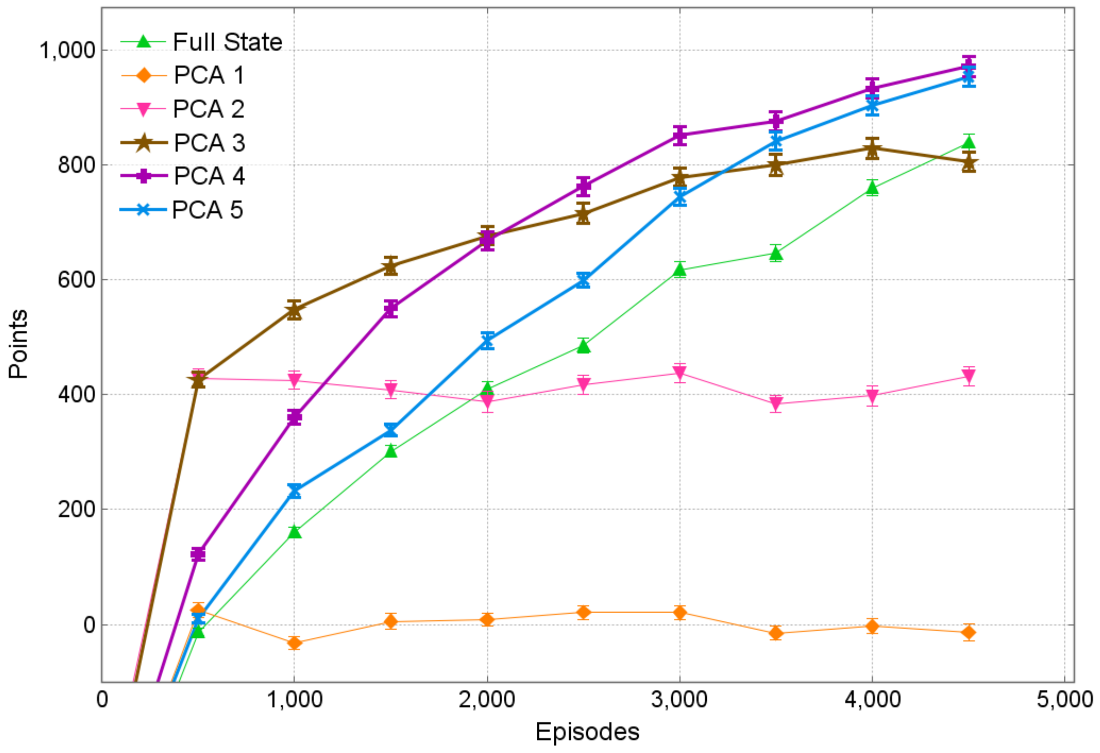

Figure 4.3: Results with lower-dimension manifolds. PCA n represents projecting the state space to a n-dimensional space. Lines in bold are experiments that performed better or equal to learning with the full state. Dimensions greater than 5 performed similarly to the full state space, and were not included for clarity. Error bars are the variance over 100 statistical runs.

only for fine tuning policies.

4.1.1.2 Mountain Car

In our formulation of Mountain Car we used 16 tiles and a 10^n tiling, where n is the number of state variables. There are 4 state variables and 5 actions in the 3D variant. We first learn using good demonstrations and single dimensions to test the efficacy of DRRL. We also show that if the subspace does not represent enough variance in the data, the learning algorithm will converge to a poor policy. To gather the demonstrations we learned good and bad policies with Q-Learning and computed random policies. We define good and bad policies by the reward they received during learning. Good demonstrations reached the goal within 300 time steps, and bad demonstrations within 500–1000 steps. Bad demonstrations reached the goal state, just less efficiently.

In 3D Mountain Car there was no single state variable more important than all other state variables. Our principal components showed that the first two principal components weighed all of the state variables equally, independent of demonstration quality. Since the demonstrations explored much of the configuration space of the agent, this showed us which states were important to the agents general movement.

Learning in only one d-dimensional manifold converged faster than learning in the full state space (Figure 4.4). DRRL converged to the optimal solution using only 2 or 3 dimensional subspace, rather than the full dimensional space of 4. This tells us that the Mountain Car domain is simple enough to be learned in a 2 dimensional space. It also converged very quickly to a poor solution in a 1 dimensional manifold. However, by using only a single dimension, DRRL does not have a rich enough state space to learn optimally.

4.2 Iterative Dimensionality-Reduced Reinforcement Learning

To account for this loss of information in the lower-dimensional manifold, we developed Iterative Dimensionality Reduced Reinforcement Learning (IDRRL). By using iterative learning, we transfer what was learned in the low-dimensional space to a higher-dimensional space. To do this, we use transfer learning. In this work, we want to transfer all of the knowledge from the source (low-dimension) to the target (higher-dimension)

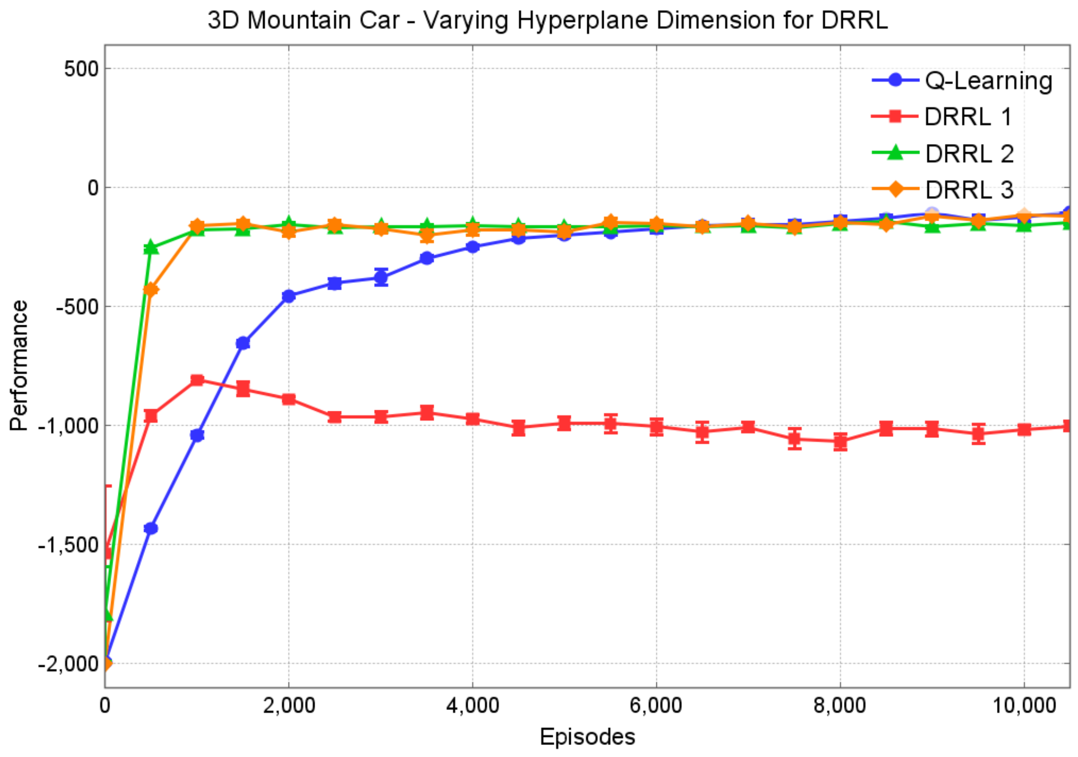

Figure 4.4: We compare Q-Learning to Q-Learning combined with DRRL using good quality demonstrations. Q-Learning combined with DRRL converged faster to an optimal solution when learning in the 2 and 3 dimensional manifold. The single dimensional manifold did not contain enough information to learn effectively.

task. Additionally, the mapping from the source to the target task ($\chi_S(s)$) is given by the dimensionality reduction mapping. This eases the transfer problem greatly. We only need to choose *when* to transfer. If we transfer too early, the value function in the low-dimensional representation is far from optimal. This bad knowledge can be spread throughout the value function in the high-dimensional case.

We can use any transfer learning approach with IDRRL within the constraints of the learning algorithm. We use Q-Value Reuse [110]. Q-Value Reuse is a simple technique applicable when the source and target task both use temporal difference learning, as in our case. In Q-Value Reuse, a copy of the source task's value function is retained and used to calculate the target task's Q-Value. This computed Q-Value is a combination of the source task's saved value function and the target task's value function:

$$Q(s,a) = Q_{source}(\chi_S(s), a) + Q_{target}(s,a) \tag{4.1}$$

where χ_S is the transfer function between the source and target tasks' states. This transfer function is the dimensionality reduction projection. We then compute the Q-Learning update step as normal, but we only update the target's value function (line 7-10, Algorithm 4).

To choose when to transfer, we borrow from the definition of convergence in Policy Iteration [103]. In Policy Iteration, the value function is updated at each iteration until the policy does not change between updates. We make this policy comparison and ensure that the most recently executed policy is at least 90% converged (unchanged before and after an update) before transferring to the higher space (line 17-23, Algorithm 4).

By iteratively transferring the policies learned in the low-dimensional manifold, IDRRL learns as fast as DRRL without the loss of information. Eventually learning is performed in the full dimensional space. The computational cost associated with this transfer is negligible. In each dimension IDRRL performs d matrix multiplications to project the state down onto each low-dimensional manifold. Although this is a $O(dn^2)$ computation, the size of the matrix is typically small and so the computation is fast.

Algorithm 4 Iterative Dimensionality-Reduced Reinforcement Learning

```
 1: function ITERATIVE DIMENSIONALITY-REDUCED REINFORCEMENT LEARNING
 2:     Initialize $Q_d$ arbitrarily for all $d$
 3:     for $e \leftarrow 1$ to $episodes$ do
 4:         $s \leftarrow initialize$
 5:         for $t \leftarrow 1$ to $timesteps$ do
 6:             $Q(s) \leftarrow \{\}$
 7:             for $i \leftarrow 1$ to $d$ do                                   ▷ Q-Value Reuse
 8:                 $s_i \leftarrow DimensionalityReduction_i(s)$
 9:                 $Q(s) \leftarrow Q(s) + Q_i(s_i)$
10:             end for
11:             $a \leftarrow \max_a Q(s,a)$ with probability $1 - \epsilon$ otherwise $a \leftarrow$ random
12:             Take action a, observe $r$ and $s'$
13:             $s_d \leftarrow DimensionalityReduction_d(s)$
14:             $s'_d \leftarrow DimensionalityReduction_d(s')$
15:             $r \leftarrow GetReward(s')$
16:             $Q_d(s_d, a) \leftarrow Update(s_d, a, s'_d, r)$
17:             $s \leftarrow s'$
18:             $a' \leftarrow \max_{a'} Q(s_d, a')$                              ▷ Convergence Test
19:             if $a = a'$ then
20:                 $\delta \leftarrow \delta + 1$
21:             end if
22:         end for
23:         if $\frac{\delta}{timesteps} \geq .90$ then
24:             $d \leftarrow d + 1$
25:         end if
26:     end for
27: end function
```

4.2.1 Results and Analysis

4.2.1.1 Mountain Car

Combining standard reinforcement learning with IDRRL led the agent to converge faster with the same converged performance (Figure 4.5). DRRL converged slightly faster than IDRRL, but IDRRL benefits by eventually learning in the entire state space. This means IDRRL does not lose any state information to speed up learning. Moreover, it does not require the algorithm designer to know beforehand which is the best manifold to learn in.

Since IDRRL represents the state space in low-dimensional and sparse manifolds, it converges very quickly. With each additional dimension, it starts with a richer state space and the experience gained from all previous dimensions. By episode 1,000, IDRRL bootstrapped learning in the full dimensional space, and was near an optimal solution. This results in much faster convergence than learning entirely in the full dimensional space (Figure 4.5).

To analyze the robustness of the approach, we varied the quality of demonstration data as well as the amount. Figure 4.5 shows the relationship between demonstration quality and performance. There is no significant performance difference between good, bad, and random demonstrations. IDRRL is robust to demonstration quality because we are showing the learning algorithm the low-dimensional manifold in which to learn in. This is alternative to learning from demonstration approaches, where the agent learns based on the demonstrated policy and learning is susceptible to bad demonstrations.

To test the sample robustness of IDRRL we also varied the amount of demonstration data. For this analysis, we used random demonstrations and varied the amount of demonstration data used between 1,000 and 25,000 demonstration states. We only test with random demonstrations, since demonstration quality was not a factor in performance for Mountain Car 3D. None of the random demonstrations reached the goal state, and each demonstration trajectory was approximately 2,000 samples. The experiment with 1,000 demonstration points converged slightly slower, and there were no significant differences between 10,000 and 25,000 states.

IDRRL scales well with the size of the state space. We modified Mountain Car 3D to add an additional fourth dimension. Mountain Car 4D has 6 continuous states and 7

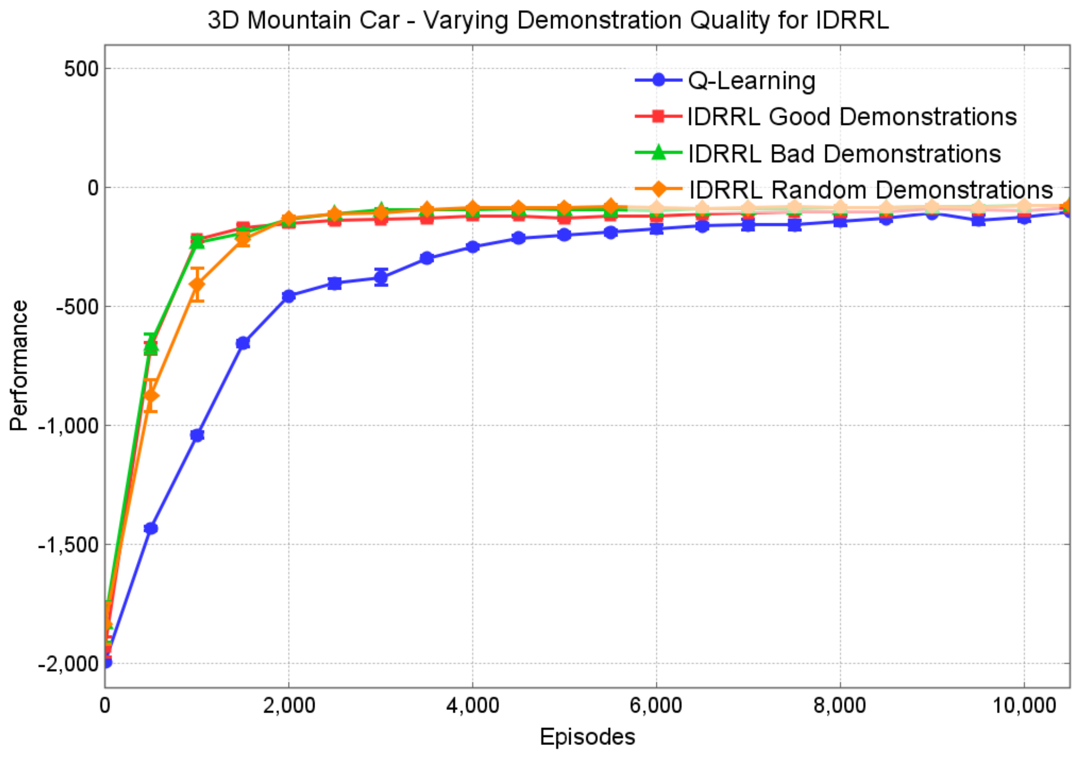

Figure 4.5: We compare Q-Learning to Q-Learning when combined with IDRRL with demonstrations of varying quality. IDRRL converged at the same speed to the optimal solution when given good, bad or random demonstrations. In Mountain Car 3D IDRRL is robust to suboptimal demonstrations.

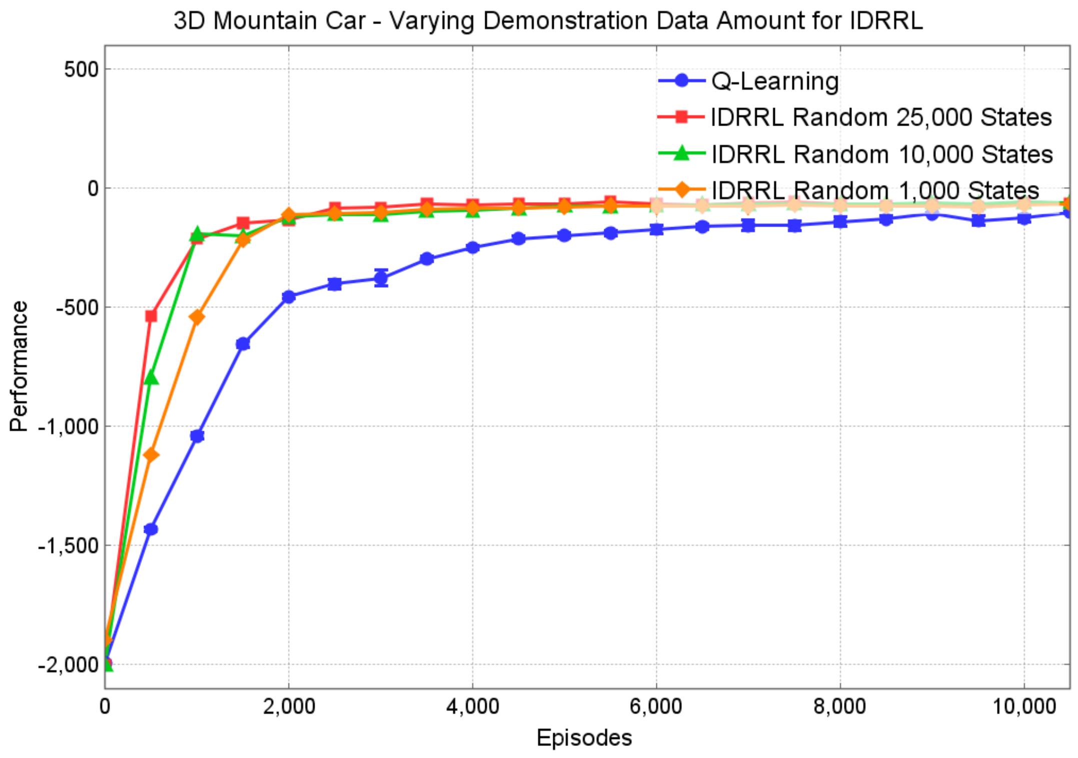

Figure 4.6: In Mountain Car 3D, there is no significant difference in the performance of IDRRL when using 10,000 or 25,000 demonstration states.

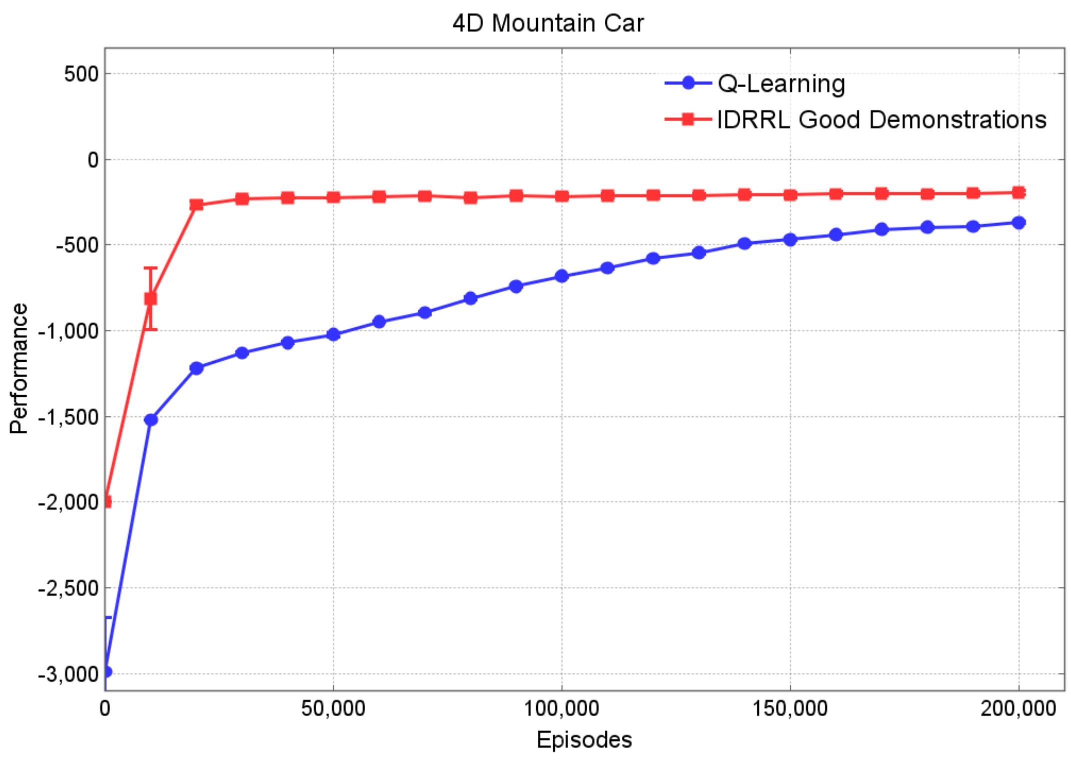

Figure 4.7: In Mountain Car 4D, Q-Learning with IDRRL scales well with the size of the state space.

actions. There are position and velocity states and acceleration/deceleration actions for each of the x, y and z dimensions. The trends seen previously in Mountain Car 3D are emphasized with additional states (Figure 4.7). By using IDRRL, the agent converges much faster to a good solution.

4.2.1.2 Swimmers

In our formulation of Swimmers we used 32 tiles and a 10^n tiling, where n is the number of state variables. There are 6 state variables and 9 actions in the 3-link swimmer and 12 state variables and 243 actions per state in the 6-link swimmer. Similar to our Mountain Car experiments, we gather demonstrations by learning in the domain with Q-Learning and collecting demonstrations. These demonstrations represent the best policies found

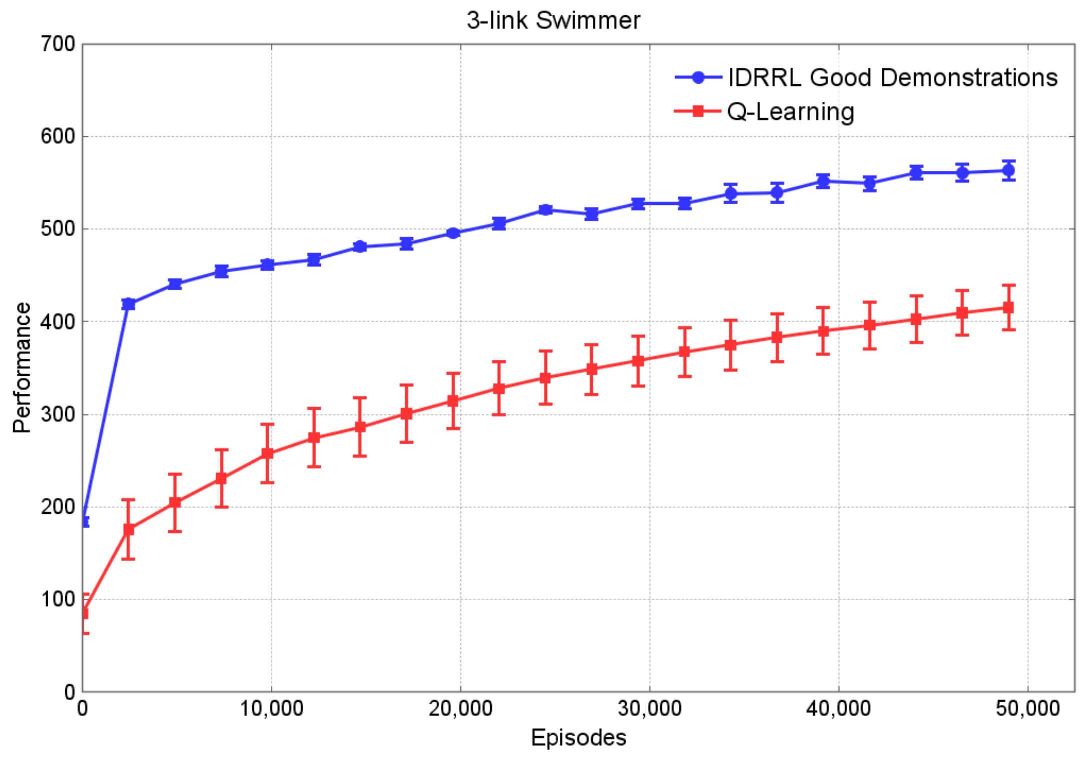

Figure 4.8: Q-Learning with IDRRL converges faster than standard Q-Learning when learning a control policy for a 3-link swimmer with 6 state dimensions and 9 actions.

with Q-Learning. In swimmers, the performance is measured by how far the swimmer has moved to the right (Δx). In the 3-link swimmer problem, the best policy performed well, but in the 6-link problem the policies were highly suboptimal due to the large state and action space.

The Swimmers Domain is a more complex system than Mountain Car. It includes complex physics with a large state and action space. This is where IDRRL can greatly increase learning performance when added to an existing algorithm. When adding IDRRL to Q-Learning, the new learning algorithm can learn a controller for a 3-link Swimmer faster (Figure 4.8).

IDRRL explored half the number of new states as Q-Learning (Figure 4.9). Although this is expected in the low-dimensional states, it was unexpected when projected to the full state space. This demonstrates that IDRRL was able to focus exploration in a high-

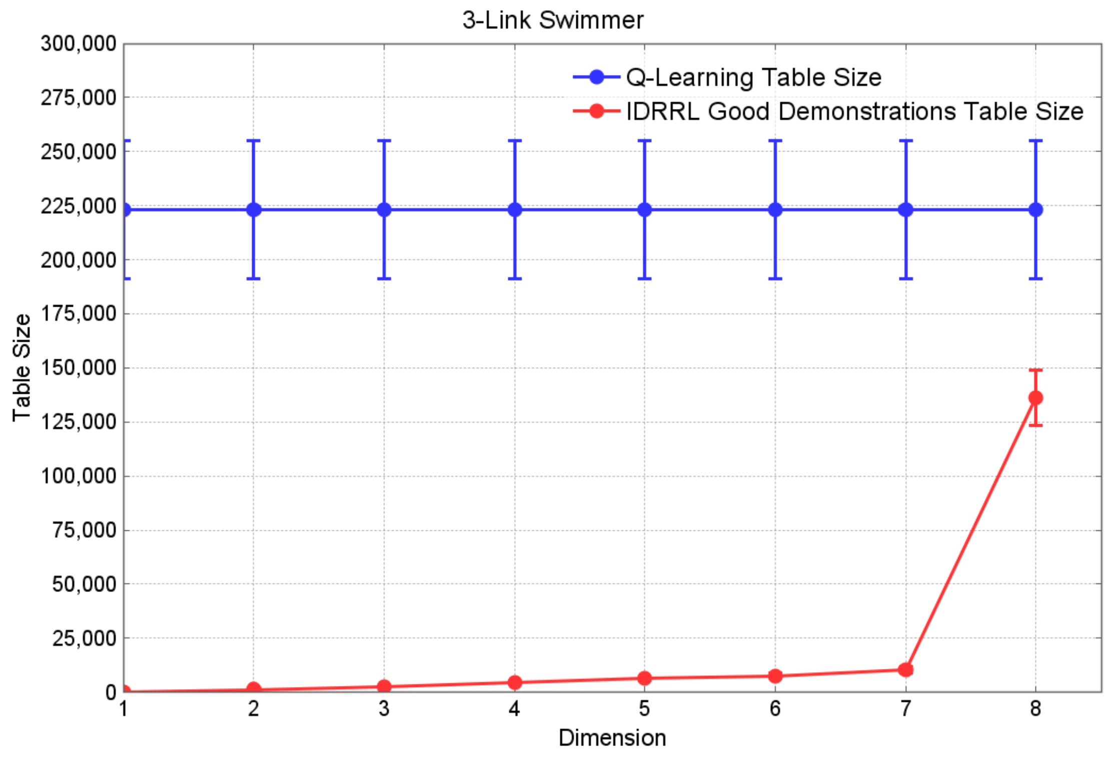

Figure 4.9: Q-Learning with IDRRL explored half as many states as Q-Learning alone. This explains the increase in learning speed we demonstrate in the Swimmers domain.

utility area of the state space. This reduces the amount of exploration in the learning algorithm and speeds up learning.

IDRRL scales effectively to a state space of 12 dimensions with 243 possible actions at each state (Figure 4.10). By initially projecting the state space onto a manifold, IDRRL samples many of the actions in a smaller space. It then generalizes what was learned in the low-dimensional space to the higher-dimensions. This generalization causes IDRRL to scale well with the size of both the state and action space.

4.2.1.3 Ball Balancing

In our formulation of the ball balancing problem we used 64 tiles and a 8^n tiling, where n is the number of state variables. For IDRRL we gathered demonstrations by collecting

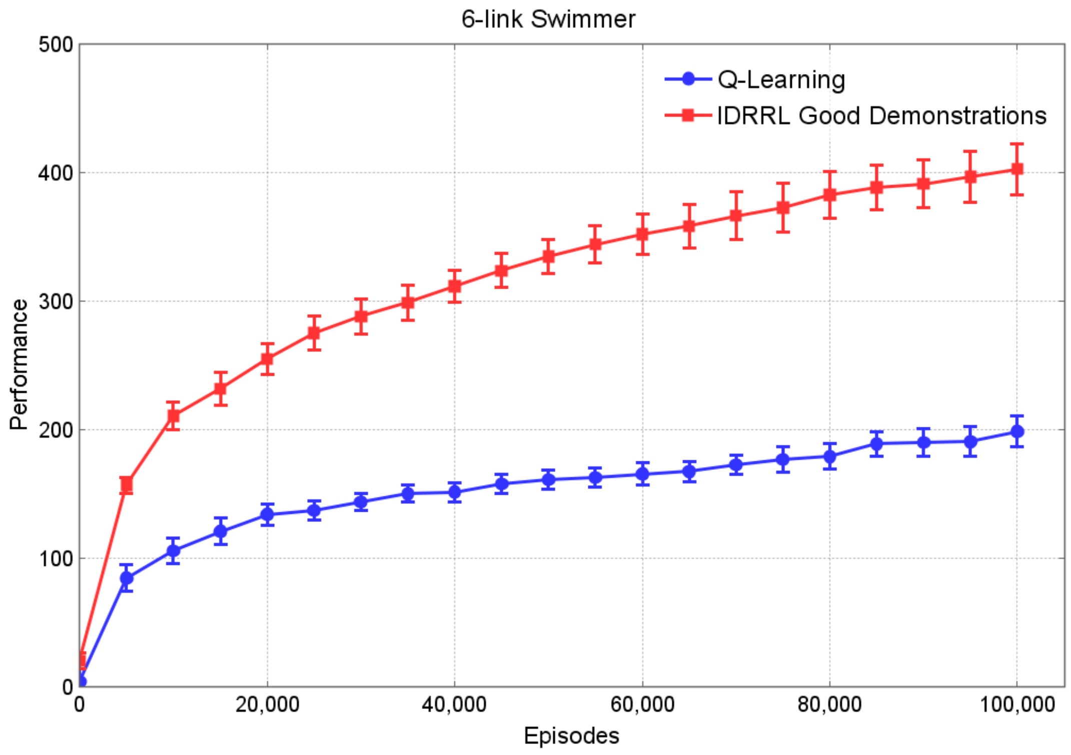

Figure 4.10: A 6-link swimmer is difficult to control due to an extremely large state and action space. Q-Learning with IDRRL learns a swimmers control policy quickly in 12 state dimensions and 243 actions per state.

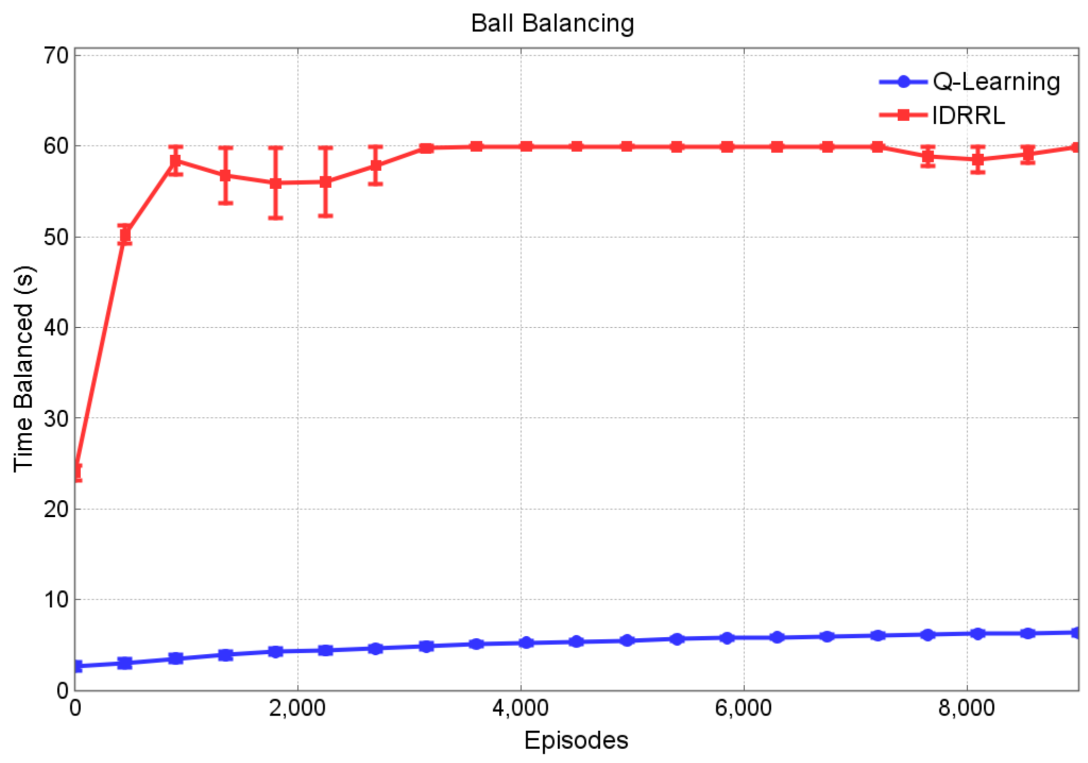

Figure 4.11: The simulated robot arm learned to balance the ball for 60 seconds when learning with IDRRL. With standard Q-Learning, the agent improved, but could not balance the ball for longer than 8 seconds.

learned trajectories from Q-Learning. The agent controls the robot by choosing small accelerations at each joint each time step. To balance the ball, the agent has to correlate these accelerations to the change in position and velocity of the arm as well as how these states impact the ball position and velocity.

We simulated the robot arm, ball and plate for 60 seconds. Each episode terminated when the ball falls, or at the end of 60 simulated seconds. We reward the agent 1 for each time step the ball remains on the plate and -1000 if the ball falls. Standard Q-Learning could not learn a policy in which the ball could remain on the plate for the full 60 seconds. However, by using IDRRL we quickly converged to the optimal solution (Figure 4.11).

IDRRL performed better than expected, so we analyzed the principal components

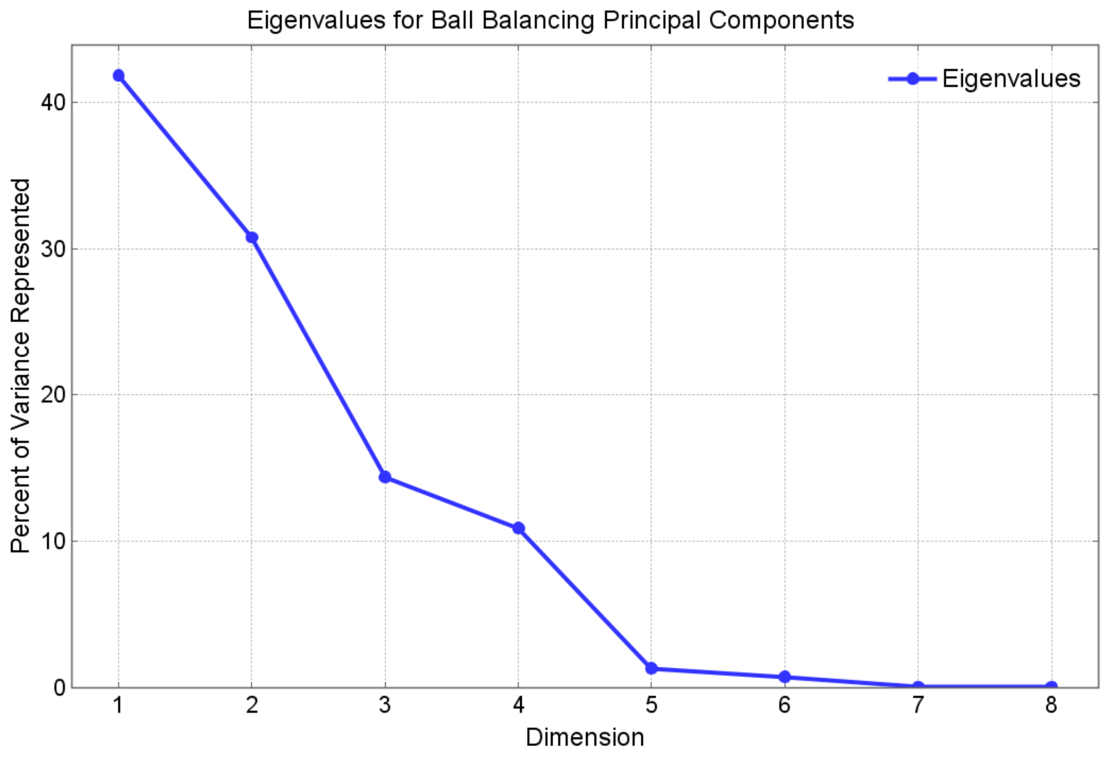

Figure 4.12: The eigenvalues associated with the ball balancing principal components. The full state space can mostly be represented within the first 4 dimensions.

in the dimensionality reduction to elucidate the reason. We attribute the increase of performance to the representational power of the first four dimensions and accurate associations between different state dimensions. According to the eigenvalues, the first 4 principal components can represent 98% of the variance in the demonstration data (Figure 4.12 and Figure 4.13). This reduces number of dimensions in the state space by half.

The strongest evidence for the increased performance are the associations PCA found between states. The first principal component (Table 4.1, row 1) associates the movement of the joint positions to the movement of the ball. The rolling joint (joint 5) position and velocity is positively correlated with the ball position and velocity across the x plane. This makes intuitive sense, since the rolling joints change the ball along the x direction. Similarly, there is also a positive correlation between the pitch joints and the

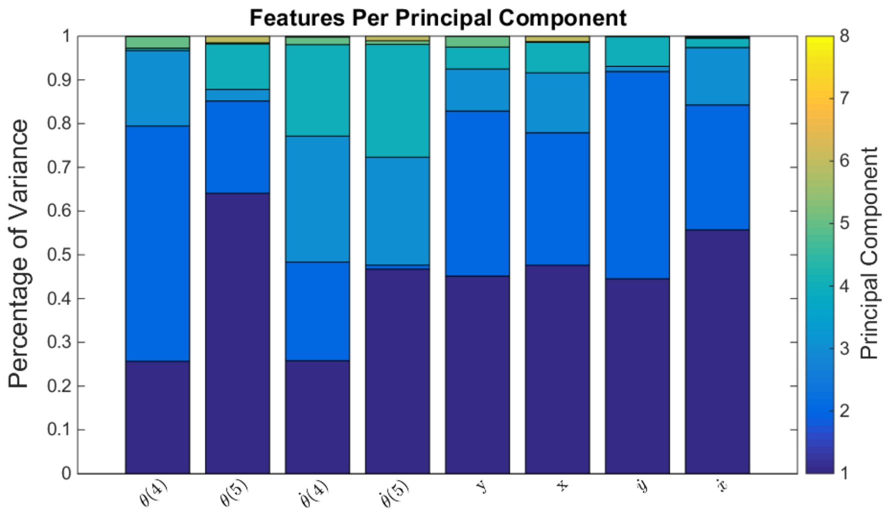

Figure 4.13: A visualization of the ball balancing principal component eigenvectors weighted by the eigenvalue. The full state space can mostly be represented within the first 4 dimensions.

PC	Eigenvector							
	Arm States				Ball States			
	$\theta(4)$	$\theta(5)$	$\dot{\theta}(4)$	$\dot{\theta}(5)$	y	x	$\dot{y}$	$\dot{x}$
1	-0.1858	**0.4815**	0.1892	**0.3144**	**-0.3664**	**0.3950**	**-0.3365**	**0.4422**
2	**0.5299**	0.2157	0.2246	-0.0082	**0.4164**	**0.3415**	**0.4871**	**0.3082**
3	**0.3633**	-0.0579	**0.6145**	**0.4826**	-0.2277	**-0.3312**	-0.0263	**-0.3032**
4	0.0154	**0.3010**	**-0.5899**	**0.6667**	0.1567	-0.2222	0.1963	-0.0652
5	-0.6224	0.0582	0.4092	0.1814	0.6343	-0.0421	-0.0037	-0.0686
6	0.0211	0.6697	0.0896	-0.4180	-0.0105	-0.6016	0.0138	0.0775
7	-0.4055	-0.0181	0.1107	-0.0020	-0.4612	0.0103	0.7811	0.0063
8	-0.0152	-0.4188	0.0199	0.1265	0.0285	-0.4523	-0.0035	0.7763

Table 4.1: The eigenvectors associated with the ball balancing principal components. The numbers in bold are the primary principal components for the most representative projections.

movement across the y plane. The Motoman arm pitch and roll joints are independent, causing a negative correlation between the roll impacts and the pitch impacts. Lastly, the magnitude of rolling joints and movement across the x plane are the largest. This large magnitude is due to the plate being twice as long across the x dimension than the y (.5m vs .25m). Since there are no low eigenvector values in the first principal component, all states are important for the low-dimensional representation. The eigenvalue of this first principal component represents 41% of the variance in the data, meaning that this projection is unlikely by itself to learn a good policy, but can learn a rough state transition to transfer to later principal components.

The rest of the principal components represent variance that previous principal components do not represent. The second principal component (Table 4.1, row 2) takes into account the pitch effect of the arm that the first principal component missed. It correlates the pitch of the arm and some roll with the movement of the ball. This principal component represents 30% of the data, showing that it is able to extract a lot of variance in pitch that the first principal component missed.

The third and fourth principal components focus on the impact arm velocity has on the ball movement (Table 4.1, row 3 and 4). Each of these principal components represent around 10% of the data, showing that arm velocity is important for representation, but not as important as arm position.

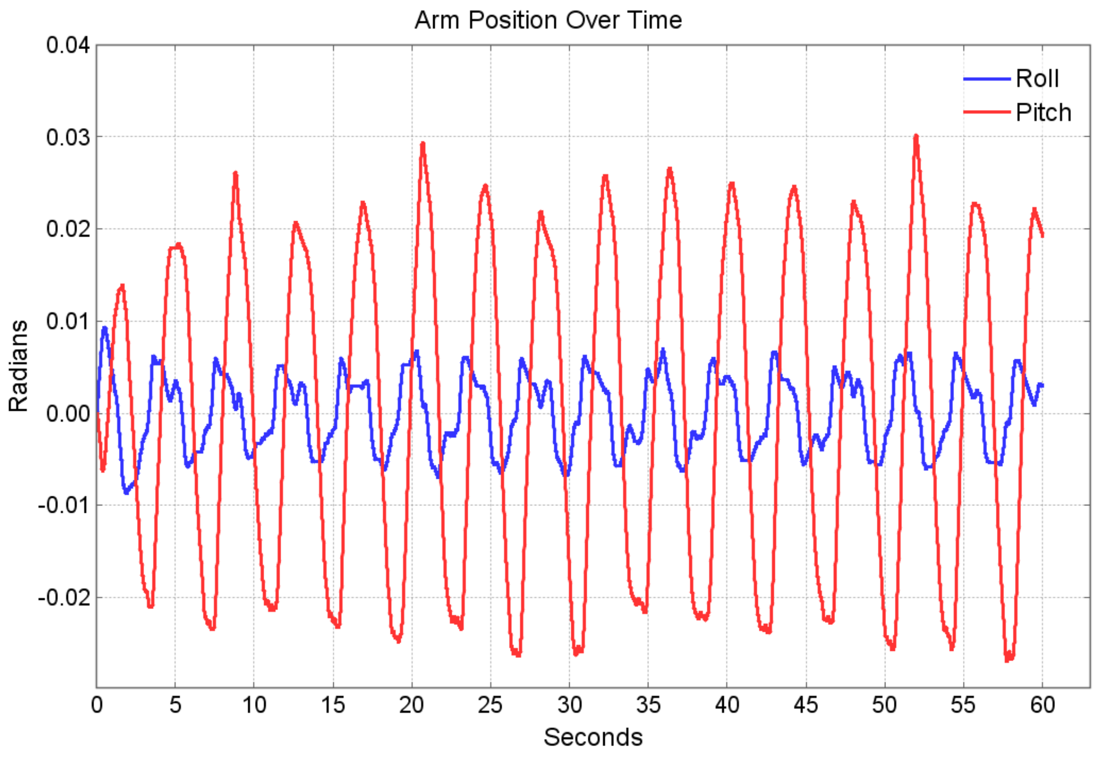

Figure 4.14: The agent learned to “flick” the ball in the roll direction to remove momentum from the system, and move the ball into the center of the plate. Meanwhile, the pitch cycles remained static throughout with differing heights in the peaks and valleys.

We tested our learned policy in an open-loop implementation on the robot arm. For the open-loop experiments we wrote 60 seconds of our simulated policy to file and ran it directly on the robot with no feedback. We found that even with no feedback, the arm was able to balance the ball for the full 60 seconds.

More interestingly, we found that whether the ball started on the left side, middle, or right side of the board, the agent was still able to balance the ball. The agent learned an effective general cyclic policy that moved the ball closer to the middle of the plate throughout the simulation (Figure 4.14). The agent learned to manipulate the momentum of the ball by “flicking” the roll of the arm. This essentially created a basin of attraction in the center of the plate.

4.3 PAC-MDP Analysis

There are three metrics to quantify the performance of a reinforcement learning algorithm: computational complexity, space complexity and sample complexity [102]. These metrics determine if a learning algorithm is effective, and is used in Probably Approximately Correct in Markov Decision Processes (PAC-MDP) analysis. Given the state space S, action space A, error rate ϵ, confidence δ, and discount factor γ, an algorithm is said to be PAC-MDP if:

Definition 4.3.1. An Algorithm A is said to be an PAC-MDP algorithm if, for any $\epsilon > 0$ and $0 < \delta < 1$, the per-timestep space complexity and sample complexity of A are less than some polynomial in the relevant quantities $(S, A, \frac{1}{\epsilon}, \frac{1}{\delta}, \frac{1}{1-\gamma})$, with probability at least $1 - \delta$. It is **efficiently** PAC-MDP if we add computational complexity to the same constraints. [98].

In this section, we show that adding DRRL or IDRRL does not remove the efficient PAC-MDP classification to an existing learning algorithm.

4.3.1 Computational Complexity

Computational complexity represents the per-timestep computation the algorithm uses during learning. In our work, the computational complexity relies on the learning algorithm, dimensionality reduction algorithm, and state generalization approach. In this analysis, we hold the learning algorithm and state generalization approach static, and analyze the additional computational cost when adding dimensionality reduction.

The additional computational complexity when we apply DRRL to an existing learning algorithm is based on the dimensionality reduction algorithm. For each learning iteration, the algorithm projects the current state and the next state onto a low dimensional manifold. For principal component analysis, this is a cost of $O(n^2)$ where n is total number of dimensions (Equation 2.9). Since n is typically low ($n \leq 24$), this is a low additional cost for the increase in performance.

The additional computational complexity when we apply IDRRL to an existing learning algorithm is larger than in DRRL. In order to apply Q-Value Reuse we need to project the current state and next state onto each dimension between 1 and the current learning dimension. For principal component analysis, this is a cost of $O(dn^2)$ where n is total number of dimensions and d is the current learning dimension.

Since both DRRL and IDRRL add at most a 2 degree polynomial complexity to the existing learning algorithm, the algorithm retains its computational complexity PAC-MDP guarantee.

4.3.2 Space Complexity

Space complexity is the measure of the amount of working storage an algorithm needs. This depends on how much memory, in the worst case, is needed at any point in the algorithm. Again, we will hold the learning algorithm and state generalization approach static, as many are already polynomial.

The additional complexity when we apply DRRL and IDRRL to an existing learning algorithm is also based on the dimensionality reduction algorithm. For both algorithms, the the principal component analysis space complexity would be the nxn matrix of principal components. This is a cost of $O(n^2)$, within the polynomial requirements to maintain efficient PAC-MDP guarantees.

4.3.3 Sample Complexity

The sample complexity of an algorithm directly measures how many times an agent acts suboptimally (ϵ). Researchers use sample complexity in PAC-MDP theory to determine how quickly an agent learns. Here, we show that by adding DRRL or IDRRL to an existing learning algorithm, the sample complexity remains polynomial.

Definition 4.3.2. Let $c = (s_1, a_1, r_1, s_2, a_2, r_2, ...)$ be a random path generated by executing an algorithm A in an MDP, M. For any fixed $\epsilon > 0$ the **sample complexity** of A is the number of timesteps t such that the policy at time t, A_t, satisfies $V^{A_t}(s_t) < V*(s_t) - \epsilon$ [102].

Strehl, Li, and Littman [102] demonstrated that Q-Value initialization can decrease the sample complexity while maintaining PAC-MDP guarantees if the action-values are admissible. Admissible heuristics provide valuable prior knowledge to PAC-MDP RL algorithms, but the specified prior knowledge does not need to be exact. An initialization heuristic (H) is said to be admissible if:

$$V^*(s) \leq Q^*(s,a) \leq H^*(s,a) \leq \frac{R_{max}}{1-\gamma} \tag{4.2}$$

where $V^*(s)$ is the optimal value function, $Q^*(s,a)$ is the optimal action-value function, $H^*(s,a)$ is the initialization heuristic, and $\frac{R_{max}}{1-\gamma}$ is the maximum possible value.

Mann and Choe [71] extend PAC-MDP theory to intertask transfer learning. They introduce the concept of weakly admissible heuristics and show that they can still maintain PAC-MDP guarantees. To be weakly admissible, a heuristic needs to be admissible for only one action in each state. They combine weakly admissible heuristics and intertask mappings and prove they are also PAC-MDP if for each state s there is an action $\widetilde{a}$ such that:

$$V^*_{trg}(s) - \alpha \leq Q^*_{trg}(s,\widetilde{a}) \leq Q^*_{src}(\chi_S(s), \chi_A(\widetilde{a})) \tag{4.3}$$

where α is the smallest non-negative value satisfying this inequality.

Q-Value Reuse is the intertask transfer learning technique we use in this work and is an example of action-value initialization. We still enforce an admissible heuristic to make use of the *optimism in the face of uncertainty* bias [17]. This bias has been shown to reduce the chance of converging to a locally optimal policy. As it stands, Q-Value Reuse is not admissible. It is guaranteed to be less than $\frac{R_{max}}{1-\gamma}$, but is not greater than $Q^*(s,a)$. We remove this issue by simply adding R_{max} to each $Q_{source}(\chi_S(s), a)$ function.

4.4 Cascading Autoencoders for IDRRL

To overcome the limitations of PCA and Kernel PCA, we introduce a novel approach to dimensionality reduction based on a system of cascading autoencoders. In this section we explain the structure and function of this dimensionality reduction technique.

4.4.1 Dimensionality Reduction Requirements for Learning

There are three requirements for our dimensionality reduction technique. First, it needs to be fast. Iterative Dimensionality Reduced Reinforcement Learning uses its dimensionality reduction technique at each action selection step. This causes minor increases in computational complexity but can increase the entire learning runtime by magnitudes.

This computational cost means that traditional non-linear dimensionality reduction techniques, such as ISOMAP [111], Kernel PCA [73], Locally Linear Embedding [93], are computationally infeasible during learning. These techniques were developed to perform dimensionality reduction as a pre-processing step for large data.

Second, the representation of each dimension needs to be a superset of all dimensions below it. If the dimensionality reduction does not strictly contain the superset, the learning algorithm will unlearn the policy learned in the $d-1$ step. Third, the dimensionality reduction needs the first dimension to maximally encode information associated with the state space, and each additional dimension encoding less. This leads to the fastest rate of learning in the single dimension case, and more informative learning in the higher dimensions.

In this work, we develop a fast non-linear dimensionality reduction technique with a cascade of autoencoders (CAE). An autoencoder neural network is an unsupervised learning technique that learns a reduced representation of its inputs. It contains an encoder network to transform high-dimensional into a low-dimensional space, and a similar decoder network that takes the output of the encoder and attempts to reconstruct the original data (Figure 4.15). These networks are trained by minimizing the difference between the original data and its reconstructed output. Once the network is trained, the decoder section of the network is no longer needed, we only use the encoder.

4.4.2 Proposed Neural Network Constructions

We explore three ways to construct the autoencoder for dimensionality reduction in IDRRL. We could create one neural network per dimension. For example, each network would encode n inputs into d outputs, where d is the current dimension IDRRL is learning in. However, this would not satisfy our second requirement. Each autoencoder would represent more of the state space, but not necessarily a superset of all previous autoencoders.

To enforce the superset requirement, each dimension would have to remain static when we increase dimensionality. That is, when we add an additional dimension, all previous dimension representations do not change. We only learn the most recent autoencoder. One autoencoder that encodes n inputs to $n-1$ outputs would satisfy this requirement. We could take d values from the $n-1$ outputs and use that as our dimensionally reduced space. However, this does not satisfy the third requirement. Since we are using a single autoencoder, there is no guarantee that the first dimension encodes the maximum amount of information. In actuality, only the joint encoding layer encodes this maximum amount of information, which is not what we desire.

The third construction involves cascading n single output autoencoders together. Each autoencoder encodes the n-dimensional space to a 1-dimension output. Rather than executing one large autoencoder, this technique requires the execution of many small autoencoders, and concatenating the outputs (Figure 4.15). This technique is fast, since these autoencoders are small and the operation can be done in parallel. Since we concatenate the outputs of each autoencoder, each projection strictly represents a superset of all lower-dimension projections. Lastly, the first autoencoder represents as much of the full space as possible in a single dimension. All additional dimensions strictly add more space representation, but never more than the previous autoencoders.

Each autoencoder after the first dimension calculates an adjustment for the previous dimensions. Therefore, the addition of all outputs for each dimension produces the original training data. To train this behavior, we used the IDRRL demonstration as both the inputs and the desired outputs for the first dimension. However, for subsequent dimensions we used the cumulative reconstruction error of all previous dimensions as the desired output:

$$\begin{aligned} y_d &= x_{train} - CAE \\ &= x_{train} - \sum_{i=1}^{d-1} \text{autoencoder}_i(x_{train}) \end{aligned} \tag{4.4}$$

where y_d is the desired output for the autoencoder in dimension d, and x_{train} is the demonstration data. This ensures that each autoencoder learns to encode only what lower-dimension autoencoders could not. To prevent overfitting to the demonstration data, we used a validation set that was a uniform grid across the entire state space. This grid is domain dependent with a complexity trade-off. If the uniform grid is highly discretized the CAE algorithm is more robust to outliers, but training time grows exponentially with the size of the discretization. In this work, we use a 4-sized grid, leading to 4^n states, where n is the number of state dimensions. Although we ignore training time, since it is a preprocessing step, the small size of our neural networks result in fast training time.

Furthermore, neural network execution is a sequence of matrix multiplications, which has polynomial space and time complexity. This ensures that we still maintain PAC-MDP guarantees.

4.4.3 Results and Analysis

We applied IDRRL-CAE and IDRRL-PCA with Q-Learning to two domains: Mountain Car 3D and Swimmers. For each experiment we use the following parameter settings for 20 statistical runs: $\alpha = 0.1$ and $\gamma = 0.99$. Error bars are shown in each graph and represent error in the mean. If an error bar is not visible, the error was negligible.

In our formulation of Mountain Car we used 16 tiles and a 8^n tiling, where n is the number of state variables. There are 4 state variables and 5 actions in the 3D variant. Although this is not a high-dimensional problem, we use Mountain Car, a problem with a known solution, to demonstrate that IDRRL-CAE can reach the same optimal solution as IDRRL-PCA and Q-Learning at a faster rate.

IDRRL-CAE learns an optimal policy in Mountain Car 3D faster than IDRRL-PCA and standard Q-Learning (Figure 4.16). We show IDRRL with random projections as a baseline to demonstrate the usefulness of an accurate projection.

To further analyze why CAE performed better than PCA, we calculated the reconstruction error for each technique (Figure 4.17). We found that PCA does not reconstruct the original data as well as CAE in the lower dimensions. This fits well within the IDRRL framework. IDRRL first learns a coarse solution quickly, and bootstraps that knowledge in the higher dimensions. The better the dimensionality reduction technique is at representing the low-dimensional space, the better the coarse solution will be, and the boostrapping will be more effective.

In our formulation of Swimmers we used 32 tiles and a 2^n tiling, where n is the number of state variables. There are 8 state variables and 9 actions in the 3-link swimmer. Similar to our Mountain Car experiments, we gather demonstrations by learning in the domain with Q-Learning and collecting demonstrations. These demonstrations represent the best policies found with Q-Learning, but are still highly suboptimal. In swimmers, the performance is measured by how far the swimmer has moved to the right (Δx).

When learning how to control a 3D swimmer, IDRRL-CAE learns a 71% better policy

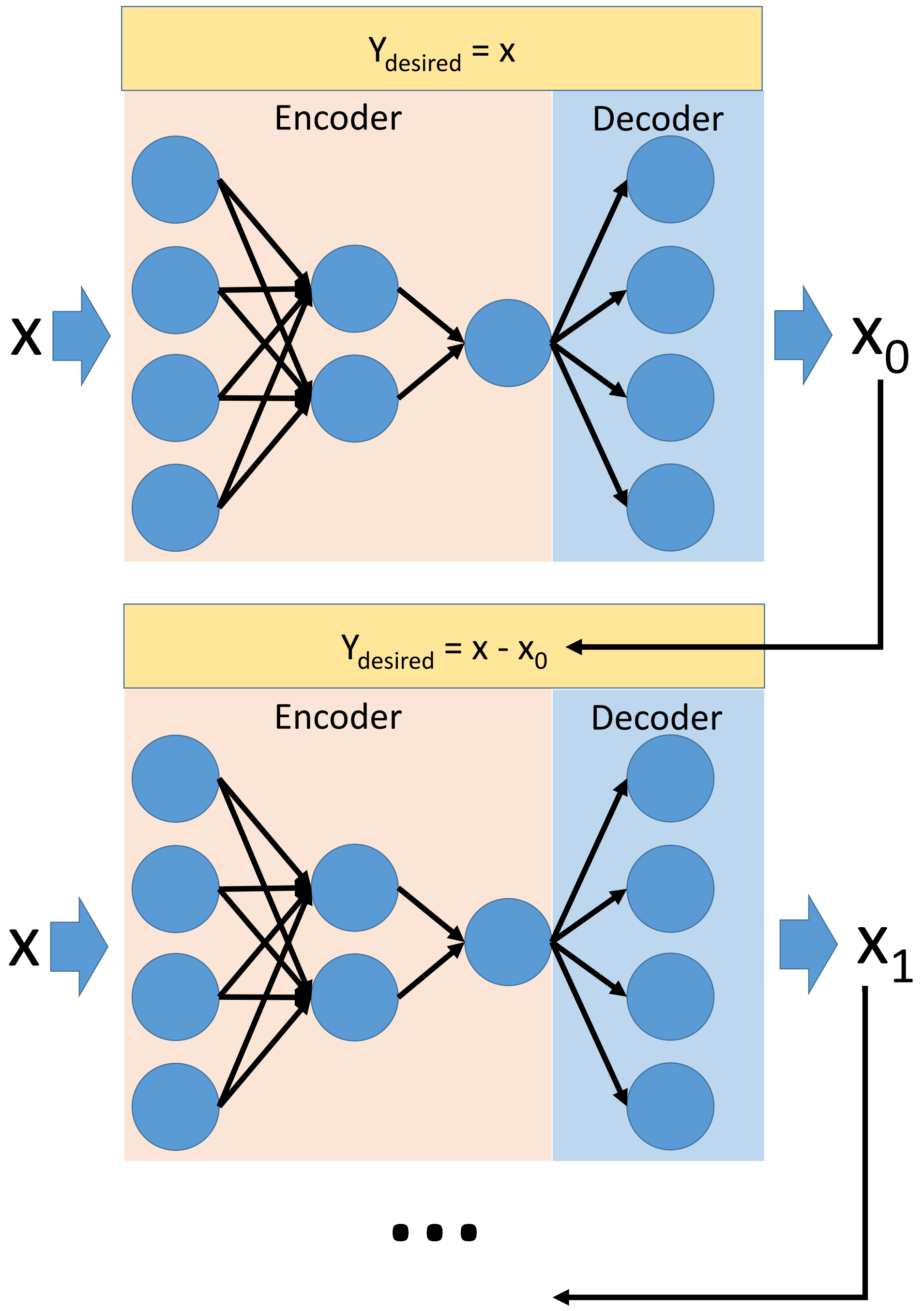

Figure 4.15: The CAE dimensionality reduction technique trains one autoencoder per dimension. The first autoencoder uses the original training data as the desired output, and each additional autoencoder learns to predict cumulative previous autoencoders reconstruction error.

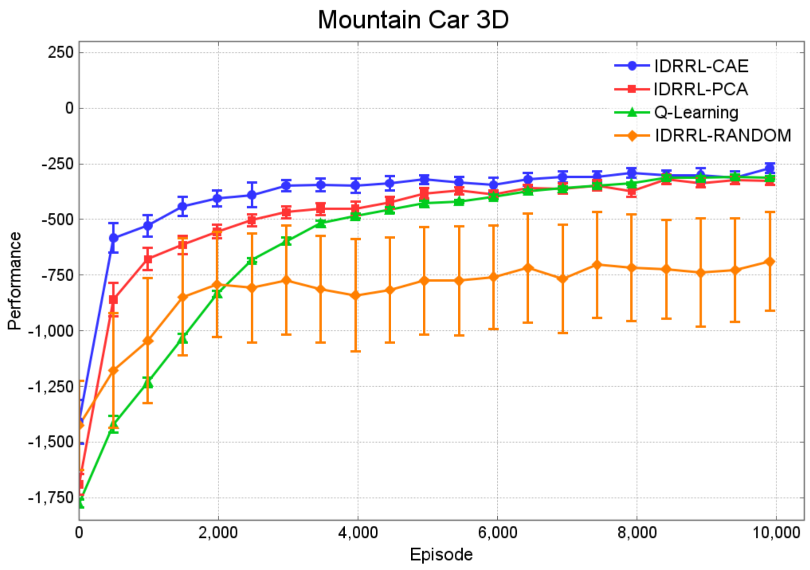

Figure 4.16: IDRRL-CAE converges faster than IDRRL-PCA or Q-Learning alone. This is due to CAE projecting the state space onto a non-linear and accurate representation. As a baseline we compared to sparse random projections.

at a faster rate than Q-Learning and a 26% better policy than IDRRL-PCA (Figure 4.18). Again, this is due to projecting the full state space onto a more representative non-linear manifold.

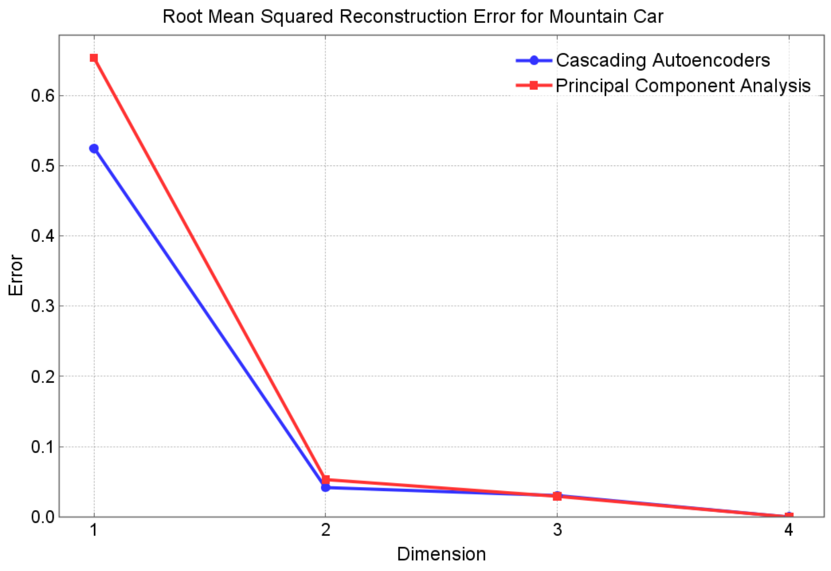

Figure 4.17: The reconstruction root mean squared error at each dimension for Mountain Car 3D using both PCA and CAE. The RMS of CAE is lower for the first dimension case, meaning that the learning algorithm will be able to learn more efficiently in this dimension, and bootstrap the next dimension with that knowledge.

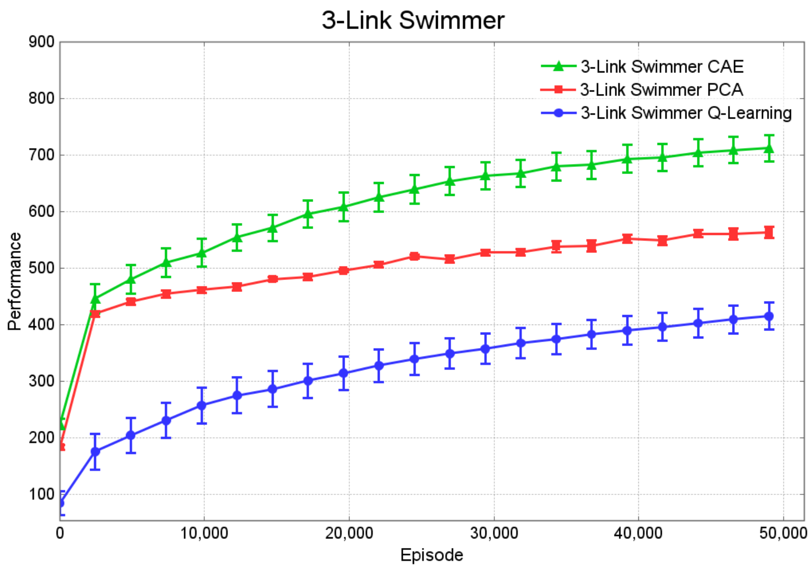

Figure 4.18: IDRRL-CAE converges faster than using IDRRL-PCA or Q-Learning alone. This is due to IDRRL-CAE projecting the state space onto a non-linear and accurate representation.

Chapter 5: Movie-Reel Interface

We had three requirements when designing our human feedback mechanism. We intend this mechanism for use in robotics, so it must work in continuous spaces. It also must tolerate delayed feedback and should not require the user to quickly react. Lastly, it must incorporate both coarse and detailed feedback.

5.1 Timeline Interface Paradigm

Interfaces for continuous space manipulation have been around for decades in post-production video editing applications. Video editing software such as iMovie, Windows Movie Maker, and Blender use a timeline interface paradigm for manipulating video streams. Video editing software naturally accounts for fine-grained state spaces, since video is typically at 30-60 frames-per-second. These interfaces are easy to use and have been used by end-users for editing videos in research [80] and K-12 education [29, 49] with success. They have also been the subject of multiple user studies associated with ease-of-use and efficiency [119].

The timeline interface paradigm allows users to edit sections of the video by annotating, concatenating, cutting, shifting and more. The user can also scan through a video by fast forwarding, rewinding, and pausing. These features remove the need to quickly react as the video is playing. Instead, users can make edits at their own pace. They can slowly step through the video, making small edits, or choose a large swath of video to make large edits. In this way, the timeline interface paradigm decouples the time of execution from the speed of editing. This expedites the editing process.

5.2 Movie-Reel Annotation

Incorporating concepts from the timeline interface paradigm into human feedback mechanisms accomplishes many of our goals. It works in continuous spaces, gives the user both coarse and fine-grained control over the executed policy, and does not require timely

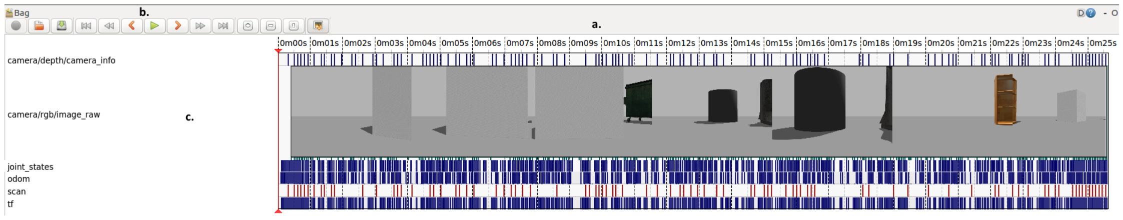

Figure 5.1: The user can drag the timeline (a) to select a subset of data. The toolbar consists of buttons for rewinding, fast forwarding, playing and pausing (b). The GUI also displays when messages are received and has visualization of video streams (c).

feedback. To test this paradigm, we build on the tool *rqt bag* [15]. *Rqt bag* (Figure 5.1) is a ROS package that provides the user with tools for scanning through recorded data, known as bags.

Rqt bag has key GUI components. There is a timeline at the top of the GUI (Figure 5.1, a.) showing the user how long the bag file is. The user can drag an area of this timeline to select a subset of data, which is useful for playing a small subset of data on repeat. The toolbar (Figure 5.1, b.) controls the playhead for fast forwarding, rewinding, slowing, pausing and playing the bag file. Lastly, the message view section of the GUI (Figure 5.1, c.) displays when messages are received by the robot, and visualizes messages. This provides a first-person view from the robot vision system. Throughout this work, we build heavily upon the *rqt bag* GUI. We add functionality to annotate recorded data with rewards, thus enabling *rqt bag* to be used in Reinforcement Learning applications.

We augmented the capabilities of *rqt bag* in multiple ways. The main features we add are the ability to create and export annotations. The user can use “annotation mode” to select a subset of the data (Figure 5.2, a.). The user can then drag an area of the timeline to reward this subset (Figure 5.2, b.). If the robot is making a small mistake, the user can highlight the respective area of the timeline and annotate it with a slightly negative reward. In this way, only a few states are given negative feedback. Likewise, if the robot is making a large mistake, the user can highlight the entire timeline of the mistake and give many states a large negative reward.

The user doesn’t have to always give negative feedback. If the user would like to incentivize the robot to move into a specific area of the state space they can use the same process to give the robot a positive reward on the timeline when the robot is in the

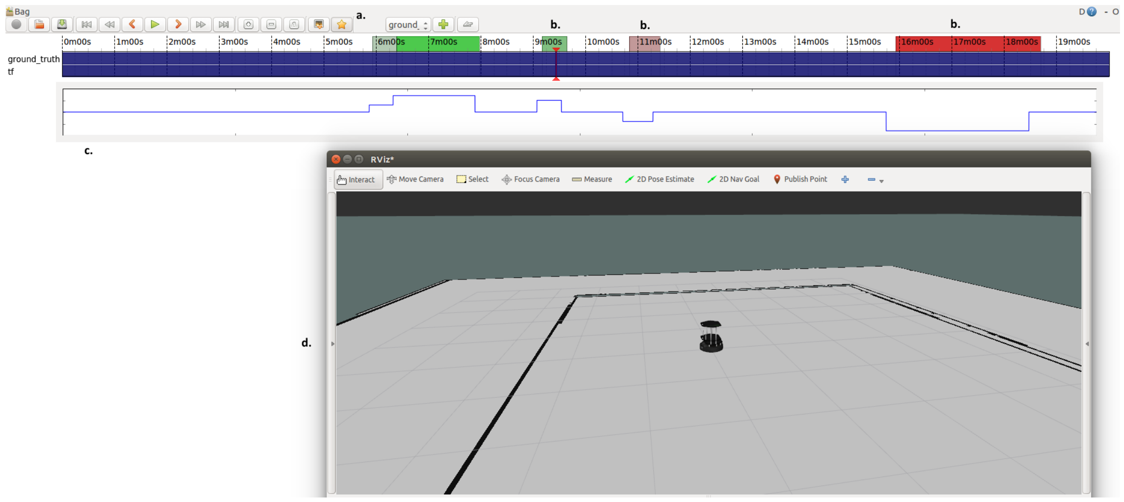

Figure 5.2: We added an annotation button (a) to the toolbar. When the user clicks this button they can drag over the timeline to annotate areas of the robot state (b). They can double click on this timeline to assign that robot state rewards. The user can see how the robot is performing over the entire execution by looking at the graph (c). We also include a GUI to visualize the state of the robot so the user can easily give feedback (d).

desired state. We add a graph of the feedback below the timeline so the user can easily see how the robot is performing (Figure 5.2, c.). Lastly, we include a GUI of the robot state so the user can visualize what the robot is doing (Figure 5.2, d.).

We incorporate this feedback as a reinforcement learning signal or a path planning cost. Although there are many approaches for human feedback integration, that is not the focus of this dissertation. To demonstrate the utility of our user feedback technique, we chose a general reinforcement learning approach, so we use standard reward shaping [103]. Every learning step, we add the human feedback to the domain reward and use that cumulative reward during policy updates. For path planning, we simply add the human feedback to the planners cost function. The planner will incorporate our human feedback by throwing out high cost trajectories.

5.3 Results and Analysis

We test our movie reel interface in two scenarios. First, the robot uses reinforcement learning to learn a policy to drive between two points. To guide the robot's learning, we develop a reward function that uses its distance to the goal. This is a standard reward function that works in general navigation scenarios. However, this reward function can potentially lead the agent to a locally optimal solution. We use our movie reel interface to modify this reward function to lower the local maxima.

In the second scenario we use our movie reel interface to incorporate human preference in path planning. When robots are navigating, they typically use path planning and optimize their trajectory by incorporating distance-to-goal and obstacles. This technique can cause the robot to navigate near dangerous situations, and a human may prefer to have the robot take a different route. We incorporate this human preference in the planner's cost function.

5.3.1 Removing Local Maxima

We use our movie-reel style interface to give human feedback in a learning scenario with a hard-to-define reward function. In this scenario, the robot is learning a policy in a deceptive problem [75], where there is a simple locally optimal policy that results in poor performance. In typical deceptive problems, the robot has to first pass through a deceptively bad region in order to reach the true goal. This can cause the robot to find a locally optimal solution, trapping it between the start location and the deceptive area. We give the robot negative feedback to reduce the value of the local maxima and positive feedback to incentivize the deceptive region.

We test how efficiently we can incorporate human feedback from our movie reel interface in a learning scenario. The robot starts with a random policy, and must learn to move from point S to point G (Figure 5.3). This is a deceptive problem, since the robot can easily get stuck in local maxima that leads to a poor policy.

The agent learned for 1000 time steps and 200 episodes. We reward the agent based on the distance it moved towards the goal at each time step, 1000 if it reached the goal and -10 if it hit a wall. The robot state consists of its position (x, y, θ) and its actions are the combination of three angular velocities (-1, 0, 1) and three x velocities (forward,

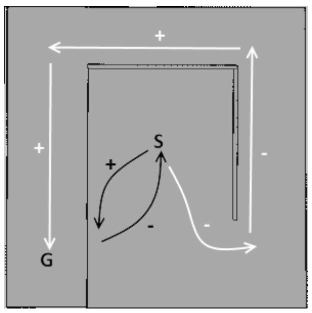

Figure 5.3: The robot spawns in a random orientation at the start location (S) and learns a policy to the goal location (G). Without human feedback, it gets stuck looping near the goal, but on the other side of the wall (black line). We give the robot negative feedback for initially moving toward the goal, incentivizing it to move away from the goal and find the positive rewards (white line). Plus and minus signs represent the distance-to-goal reward function of the robot.

neutral, reverse).

We gave the robot feedback during episode 10, 50, and 100. We gave coarse feedback when the robot moved in the correct (Figure 5.3, white line) or incorrect (Figure 5.3, black line) direction for a long period of time, and fine-grained feedback when it moved in the correct or incorrect direction for a short period of time. When the agent was not obviously moving toward or away from the goal, we gave no feedback. Since we wanted to test quick, yet detailed, feedback we limited our feedback to take no longer than 5 minutes.

When the agent is not given any feedback, it is able to easily find a policy with a converged performance of around 0 (Figure 5.4). It learns to drive in a circle, moving toward the goal, accumulating positive rewards, then once it gets close to the wall, it moves away from the goal. By learning this policy, it never reaches the goal. Alternatively, the correct policy moves away from the goal, accumulating many negative rewards, and begins receiving positive rewards once it reaches the top of the maze.

When first learning, the agent rarely moves in the correct direction. We give mainly negative feedback at episode 10. This incentivizes the agent to move away from the goal, accumulating negative rewards and worse performance according to our reward function. Since we needed to give coarse feedback at this point, we did not need the full 5 minutes. At episode 50, the agent consistently moves away from the goal, but not far enough.

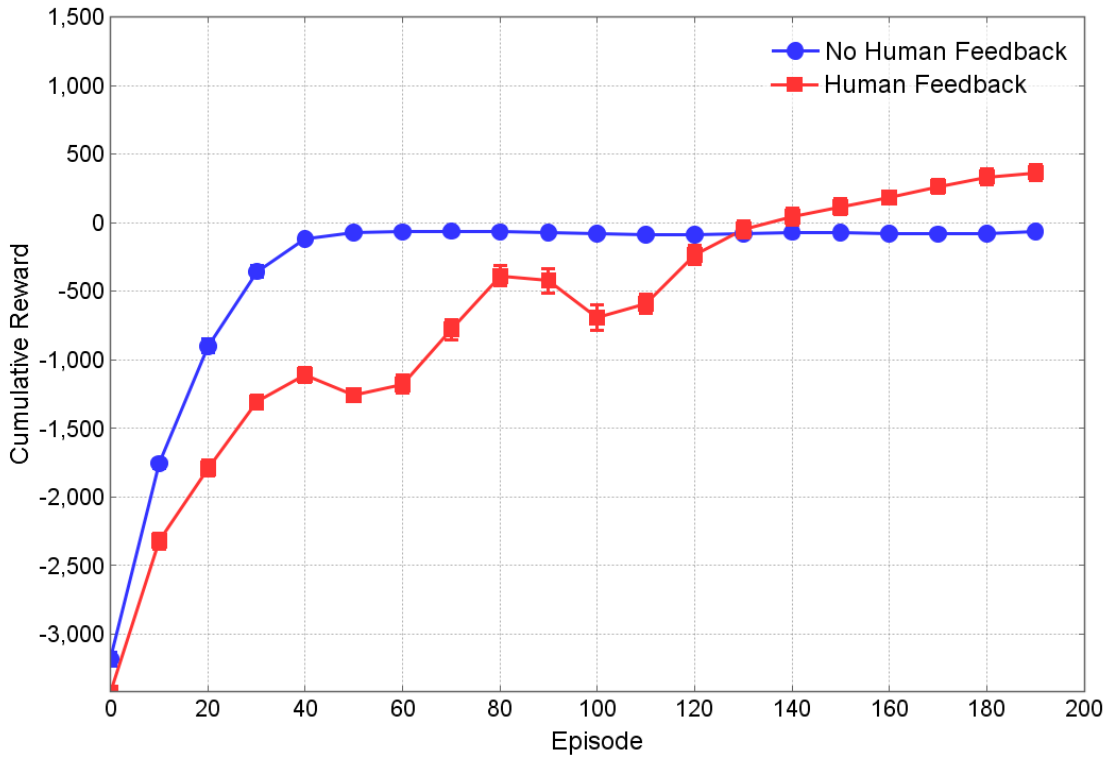

Figure 5.4: When the agent is not given feedback, it is able to easily learn a policy that leads to an overall high reward, but does not reach the goal. We give the robot feedback to reduce the value of this local maxima and learn the correct policy. We include error bars for 50 statistical runs.

This required us to use a fine-grained feedback approach and we gave both positive and negative feedback. In this feedback session we used the full 5 minutes. By episode 100, the agent inconsistently gets to the goal, and the reward signal begins propagating to earlier states. Once this happens, we do not need to give much feedback, since the agent has learned to initially move away from the goal and reach the top of the maze. At this stage, we only had to give a few feedback updates to the robot. By the end of episode 200, the agent gets to the goal nearly every episode.

5.3.2 Personalized Feedback

Sometimes navigation with costmaps should result in paths that are not strictly distance-optimal. The distance-optimal path ignores plans that move the robot close to dangerous situations, such as near a stairway. The human may prefer the robot to stay away from these dangerous situations, but the robot may not have the sensing capabilities to detect them. For example, the robot will need a downward facing sensor to detect a stairway. We show we can use our movie-reel interface to incorporate this human preference in the ROS navigation stack.

To simulate this human preference scenario, the robot uses the ROS navigation stack to optimally plan down a hallway and around a corner. This navigation software uses a combination of Dijkstra's planning algorithm for global planning and a weighted combination of distance-to-goal and obstacle costs as a cost function for local planning. In navigating, the robot moves close to a stairway, and drives closely around a sharp corner (Figure 5.5, dark line). Although this minimizes the time it takes for the robot to reach the goal, we consider this dangerous navigation. We use our movie-reel interface to give the robot feedback to negatively reinforce these undesirable states.

We incorporated this feedback in the ROS navigation local planner by including the human preference scores into the cost function. The local planner moves around locations with negative human feedback. The more extreme the negative feedback (a low negative value vs a high negative value) the more the planner is incentivized to avoid those states.

We give the robot negative feedback in states close to the stairs and close to the corner. This feedback was easy to give. In a process that took only seconds, we scanned the robot execution to where the robot moved in front of the stairs and around the corner. At these points we gave negative feedback. The local planner avoided these states and the robot navigated away from the stairs and the corner (Figure 5.5, light line).

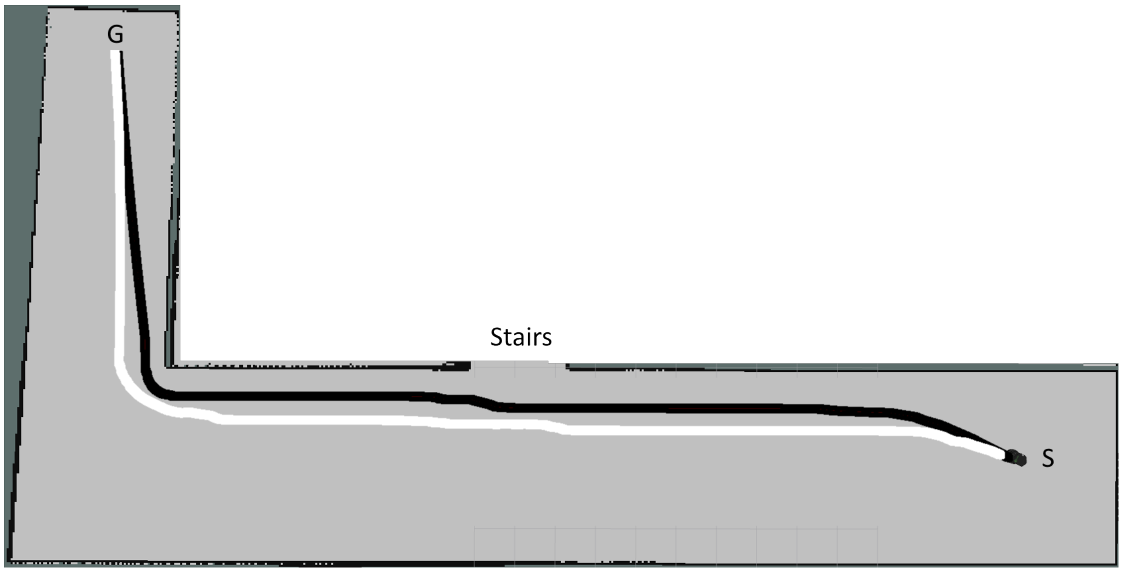

Figure 5.5: The path planning algorithm optimally plans the robot to move close to the stairs and too closely around the corner (dark line). We consider this behavior dangerous and give the robot feedback to negatively reinforce these states. The path planner uses this feedback in its cost function and plans away from both the stairs and the corner (light line).

Chapter 6: Conclusion

In this dissertation we introduced our DRRL and IDRRL frameworks to improve reinforcement learning performance in high-dimensional problems. We also introduced our movie-reel interface for human feedback on the learning algorithm's policy. This end-to-end architecture furthers the field of reinforcement learning and human feedback mechanisms for robotics by developing efficient and novel algorithms that alleviate existing issues in both fields discussed in this dissertation.

The DRRL and IDRRL frameworks improve the performance of an existing algorithm by combining the speed of low-dimensional learning and the expressiveness of the full state space (Section 4.1 and 4.2). By projecting the state space onto a low-dimensional manifold, our methods are able to represent a complex state with only a few state variables. Then, by incrementally transferring the knowledge from low-dimensional spaces into higher-dimensional ones, IDRRL learns good policies faster than the reinforcement learning algorithm alone.

We demonstrate in Mountain Car, the Mario Benchmark Problem, Swimmers and our Ball Balancing domain that by adding IDRRL to an existing learning algorithm we see an increase in speed of learning and performance (Section 4.2). We attribute the increase of performance to the representational power of the low-dimensional spaces and accurate associations between different states that the dimensionality reduction calculated. The reduction in the state space allows the learning algorithm to quickly find a high-utility section of the state space. This correspondingly reduced the number of new states IDRRL needed to explore, and increased learning speed. Additionally, we use demonstrations to find a low-dimensional manifold, rather than directly bootstrap learning. This leads our frameworks to be robust to demonstration quality, a classic issue in learning from demonstration literature [6].

We also show that when adding the DRRL or IDRRL framework to an existing learning algorithm, we retain efficient PAC-MDP guarantees (Section 4.3). Therefore, our frameworks retain any polynomial space, time and sample complexity guarantees of the original algorithm.

However, the effectiveness of learning in this low-dimensional subspace is dependent on the projection. With a projection that accurately represents the high-reward part of the state space, the learning algorithm is able to find and explore that good part of the state space quickly. With a poor projection, many dissimilar states are projected onto the same low-dimensional point. This misleads the learning algorithm and decreases learning speed in the low-dimensional space.

We introduce the novel CAE dimensionality reduction extension to IDRRL (Section 4.4). IDRRL-CAE learns in a dimensionally reduced space that is representative of the high-reward area of the full state space. This is due to the fact that autoencoders are able to learn a non-linear projection onto a low-dimensional manifold with a minimal loss of information [9]. We cascade many of these autoencoders and use them together as our dimensionally reduced state space. By using IDRRL-CAE, we reduce the RMS error of the first dimension. This leads to accurate and fast learning in that low-dimensional space. We demonstrate that this corresponds to better bootstrapping for the higher dimensional learning and therefore faster learning.

One important aspect of IDRRL not explored in this work is the extraction of extra state information. We believe that we could increase the number of state variables in the learning problem and IDRRL will begin learning by using a combination of the most representative state variables. IDRRL will leave the less important variables for the later high-dimensional spaces, after the learning algorithm has already found a high-utility area of the state space. Although these less important variables may not be useful for learning the general policy, they may be leveraged by the algorithm for fine-tuning.

In this dissertation we also discussed many of the issues with current human feedback mechanisms. Current mechanisms often do not work in continuous spaces, rely on quick feedback from the user, or only allow for either coarse or fine-grained feedback. The timeline interface paradigm alleviates many of these issues. It exists in many end-user applications and has been the subject of multiple user studies associated with ease-of-use and efficiency.

Our framework allows the user to scan through a robot execution by fast forwarding, rewinding, and pausing. These features remove the need to quickly react as the robot is executing a plan. Instead, users can give feedback at their own pace. This framework decouples the time of execution from the speed of giving annotations and expedites the feedback process.

We developed a proof-of-concept GUI for a timeline interface on the *rqt bag* architecture and test it in two scenarios. In both scenarios, it was easy for us to give feedback. In the Section 5.3.1, the robot used reinforcement learning to learn a policy to drive between two points. The robot was in a deceptive maze, and it easily found a locally optimal solution. We gave the robot feedback to reduce the local maxima and lead the robot out of the bad policy space. We give the robot feedback at three stages in learning, and limited ourselves to 5 minutes per feedback session. Although we limited ourselves to 5 minutes, we only used that full time in the second feedback session. In the first session the robot mainly needed coarse negative reinforcement and the last session it only needed a few positive samples.

Human preference plays an important role in objective functions. In the Section 5.3.2, we used our movie-reel interface to incorporate human preference. In this scenario, ROS navigation planned a path close to a stairway and quickly around a corner. The robot was perfectly safe planning close to the stairs. However, if the robot was expensive, or some sort of autonomous vehicle, the user may react in an unpredictable or dangerous manner to the robot moving toward the stairs. Incorporating human preference in the objective function alleviates this issue. We negatively reinforced these states and modified the ROS path planning algorithm to incorporate that human feedback in its objective function. The planner then navigated the robot away from the stairs and further away from the corner. By using our timeline interface, this feedback was quick and effective.

These scenarios had in common difficult to define objective functions. The first scenario needed two reward functions, one for initially moving away from the goal, and out of the local optimal, and one for later moving toward the goal. An objective function that specific will only work in that scenario. However, the more general distance-to-goal function we used was too general. Many times, these general objective functions cannot do not work effectively without human feedback.

Future work includes performing a user study to assess the usability of our framework in a robotics application. Related work addresses the ease-of-use of timeline interfaces [119], but this could change substantially when end-users are interacting with robots. This is due to the concept of trust in the human-robot interaction [85].

Human trust in robots depends on multiple factors. As robots become more effective in everyday scenarios, humans begin to place more trust in them [37]. However, the transparency of the reasoning behind robot autonomy also plays an important role in trust [118]. Our timeline interface gives end-users an efficient and transparent way to supply feedback to a robot.

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, page 1, 2004.

[2] Baris Akgun, Maya Cakmak, Jae Wook Yoo, and Andrea Lockerd Thomaz. Trajectories and keyframes for kinesthetic teaching: A human-robot interaction perspective. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction*, pages 391–398, 2012.

[3] James Sacra Albus. Brains, Behavior, and Robotics. Byte Books, 1981.

[4] R. Amit and Maja Matarić. Learning movement sequences from demonstration. In *The 2nd International Conference on Development and Learning*, pages 203–208, 2002.

[5] Brenna Argall, Brett Browning, and Manuela Veloso. Learning by demonstration with critique from a human teacher. In *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction*, pages 57–64, 2007.

[6] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[7] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1):11–73, 1997.

[8] Christopher G. Atkeson and Stefan Schaal. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 12–20, 1997.

[9] Pierre Baldi. Autoencoders, unsupervised learning and deep architectures. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop*, pages 37–50, 2011.

[10] Jonathan Baxter and Peter Bartlett. Direct gradient-based reinforcement learning: I. Gradient estimation algorithms. Technical report, National University, 1999.

[11] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, 1957.

[12] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 2nd edition, 2000.

[13] Aude Billard and Maja Matarić. Learning human arm movements by imitation: Evaluation of a biologically-inspired connectionist architecture. *Robotics and Autonomous Systems*, 941:1–16, 2001.

[14] Sebastian Bitzer, Matthew Howard, and Sethu Vijayakumar. Using dimensionality reduction to exploit constraints in reinforcement learning. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3219–3225, 2010.

[15] Aaron Blasdel, Tim Field, and Austin Hendrix. rqt bag. `http://wiki.ros.org/rqt_bag`.

[16] Federico Boniardi, Abhinav Valada, Wolfram Burgard, and Gian Diego Tipaldi. Autonomous indoor robot navigation using a sketch interface for drawing maps and routes. In *2016 IEEE International Conference on Robotics and Automation*, pages 2896–2901, 2016.

[17] Ronen I. Brafman and Moshe Tennenholtz. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2003.

[18] Darius Braziunas. POMDP Solution Methods. Technical report, Department of Computer Science, University of Toronto, 2003.

[19] Cynthia Breazeal, Matt Berlin, Andrew G. Brooks, Jesse Gray, and Andrea Lockerd Thomaz. Using perspective taking to learn from ambiguous demonstrations. *Robotics and Autonomous Systems*, 54(5):385–393, 2006.

[20] Tim Brys, Anna Harutyunyan, Peter Vrancx, Matthew E Taylor, Daniel Kudenko, and Ann Nowé. Multi-objectivization of reinforcement learning problems by reward shaping. In *International Joint Conference on Neural Networks*, pages 2315–2322, 2014.

[21] Maya Cakmak and Andrea L. Thomaz. Designing robot learners that ask good questions. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction*, pages 17–24, 2012.

[22] Sylvain Calinon and Aude Billard. Incremental learning of gestures by imitation in a humanoid robot. In *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction*, pages 255–262, 2007.

[23] Sonia Chernova and Manuela Veloso. Confidence-based policy learning from demonstration using gaussian mixture models. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 233:1–233:8, 2007.

[24] Sonia Chernova and Manuela Veloso. Teaching multi-robot coordination using demonstration of communication and state sharing. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1183–1186, 2008.

[25] Jeffery A. Clouse. On integrating apprentice learning and reinforcement learning. Technical report, University of Massachusetts, 1997.

[26] Luis C Cobo, Peng Zang, Charles L Isbell Jr, and Andrea L Thomaz. Automatic state abstraction from demonstration. In *Proceedings of the 22nd Second International Joint Conference on Articial Intelligence*, volume 22, page 1243, 2011.

[27] Adrià Colomé, Gerhard Neumann, Jan Peters, and Carme Torras. Dimensionality reduction for probabilistic movement primitives. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 794–800, 2014.

[28] Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

[29] Robyn Cox. Digital Literacies: Social Learning and Classroom Practices. *Literacy*, 46(1):56–56, 2012.

[30] William J. Curran, Adrian Agogino, and Kagan Tumer. Addressing hard constraints in the air traffic problem through partitioning and difference rewards. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multiagent Systems*, pages 1281–1282, 2013.

[31] Christian Daniel, Gerhard Neumann, Oliver Kroemer, and Jan Peters. Hierarchical relative entropy policy search. *Journal of Machine Learning Research*, 17(1):3190–3239, 2016.

[32] Marc Peter Deisenroth, Peter Englert, Jan Peters, and Dieter Fox. Multi-task policy search for robotics. In *Proceedings of 2014 IEEE International Conference on Robotics and Automation*, pages 3876–3881, 2014.

[33] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A Survey on Policy Search for Robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142, 2013.

[34] Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. *Robotics: Science and Systems*, 7:57–64, 2011.

[35] John Demiris and Gillian M. Hayes. Imitation in animals and artifacts. chapter Imitation As a Dual-route Process Featuring Predictive and Learning Components: A Biologically Plausible Computational Model, pages 327–361. MIT Press, 2002.

[36] Yiannis Demiris and Bassam Khadhouri. Hierarchical attentive multiple models for execution and recognition of actions. In *Robotics and Autonomous Systems*, pages 361–369, 2005.

[37] Munjal Desai, Kristen Stubbs, Aaron Steinfeld, and Holly Yanco. Creating trustworthy robots: Lessons and inspirations from automated systems. In *Proceedings of the AISB Convention: New Frontiers in Human-Robot Interaction*, 2009.

[38] Alexander Dietrich, Thomas Wimbock, Alin Albu-Schaffer, and Gerd Hirzinger. Reactive whole-body control: Dynamic mobile manipulation using a large number of actuated degrees of freedom. *IEEE Robotics and Automation Magazine*, 19(2):20–33, 2012.

[39] Rüdiger Dillmann, M. Kaiser, and A. Ude. Acquisition of elementary robot skills from human demonstration. In *International Symposium on Intelligent Robotics Systems*, pages 185–192, 1995.

[40] Christos Dimitrakakis and Constantin A. Rothkopf. Bayesian multitask inverse reinforcement learning. In *Proceedings of the 9th European Conference on Recent Advances in Reinforcement Learning*, pages 273–284, 2012.

[41] Alain Dutech, Tim Edmunds, Jelle Kok, Michail Lagoudakis, Michael Littman, Martin Riedmiller, Brian Russell, Bruno Scherrer, Rich Sutton, Stephan Timmer, Nikos Vlassis, Adam White, Shimon Whiteson, and Dinakar Jayarajan. NIPS workshop: Reinforcement Learning Benchmarks and Bake-offs II. 2005.

[42] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A guide to heuristic-based path planning. In *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling*, pages 9–18, 2005.

[43] Javier Garcıa, Iván López-Bueno, Fernando Fernández, and Daniel Borrajo. A comparative study of discretization approaches for state space generalization in the keepaway soccer task. *Reinforcement Learning: Algorithms, Implementations and Aplications. Nova Science Publishers*, 2010.

[44] Shane Griffith, Kaushik Subramanian, Jonathan Scholz, Charles Isbell, and Andrea L Thomaz. Policy shaping: Integrating human feedback with reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2625–2633, 2013.

[45] Daniel H Grollman and Odest Chadwicke Jenkins. Sparse incremental learning for interactive robot control policy estimation. In *IEEE International Conference on Robotics and Automation*, pages 3315–3320, 2008.

[46] Anna Harutyunyan, Tim Brys, Peter Vrancx, and Ann Nowé. Shaping mario with human advice. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1913–1914, 2015.

[47] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 2nd edition, 1998.

[48] Verena Heidrich-Meisner and Christian Igel. Neuroevolution strategies for episodic reinforcement learning. *Journal of Algorithms*, 64(4):152–168, 2009.

[49] Claudia Hindo, Ken Rose, and Louis M Gomez. Searching for steven spielberg: Introducing imovie to the high school english classroom: A closer look at what open-ended technology project designs can do to promote engaged learning. In *Proceedings of the 6th international conference on learning sciences*, pages 609–609, 2004.

[50] Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[51] Auke Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1398–1403, 2002.

[52] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning rhythmic movements by demonstration using nonlinear oscillators. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 958–963, 2002.

[53] Tetsunari Inamura, Masayuki Inaba, and Hirochika Inoue. Acquisition of probabilistic behavior decision model based on the interactive teaching method. In *Proceedings of the Ninth International Conference on Advanced Robotics*, 1999.

[54] Wallace Jackson. *The Tools of Digital Video: Non-Linear Editing Software*, pages 1–10. Apress, Berkeley, CA, 2016.

[55] Ian Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, 2002.

[56] Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. In *Proceedings of 1993 International Joint Conference on Neural Networks*, volume 2, pages 1339–1344, 1993.

[57] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4(1):237–285, May 1996.

[58] Shivaram Kalyanakrishnan and Peter Stone. An empirical analysis of value function-based and policy search reinforcement learning. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, volume 2, pages 749–756, 2009.

[59] Sergey Karakovskiy and Julian Togelius. The Mario AI benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[60] Jin-Oh Kim and Pradeep Khosla. Real-time obstacle avoidance using harmonic potential functions. *IEEE Transactions on Robotics and Automation*, 8(3):338–349, 1992.

[61] W Bradley Knox and Peter Stone. Reinforcement learning from simultaneous human and MDP reward. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 475–482, 2012.

[62] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, July 2011.

[63] Jens Kober and Jan Peters. Reinforcement Learning in Robotics: A Survey. In *Reinforcement Learning*, volume 12, pages 579–610. 2012.

[64] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2619–2624, 2004.

[65] Petar Kormushev, Sylvain Calinon, and Darwin G. Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122, 2013.

[66] Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, and Ewan Klein. Mobile robot programming using natural language. *Robotics and Autonomous Systems*, 38(3-4):171–181, 2002.

[67] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 287–296, 2006.

[68] Jeff Lieberman and Cynthia Breazeal. Improvements on action parsing and action interpolation for learning through demonstration. In *4th IEEE/RAS International Conference on Humanoid Robots*, volume 1, pages 342–365, 2004.

[69] Andrea Lockerd and Cynthia Breazeal. Tutelage and socially guided robot learning. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 4, pages 3475–3480, 2004.

[70] Manuel Lopes and José Santos-Victor. Visual learning by imitation with motor representations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 35(3):438–449, 2005.

[71] Timothy A. Mann and Yoonsuck Choe. Directed exploration in reinforcement learning with transferred knowledge. *JMLR Workshop and Conference Proceedings: EWRL*, 24:59–76, 2012.

[72] José Antonio Martín H., Javier Lope, and Darío Maravall. The kNN-TD reinforcement learning algorithm. In *Proceedings of the 3rd International Work-Conference on The Interplay Between Natural and Artificial Computation: Part I: Methods and Models in Artificial and Natural Computation. A Homage to Professor Mira's Scientific Legacy*, pages 305–314, 2009.

[73] Sebastian Mika, Bernhard Schölkopf, Alex Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, pages 536–542, 1999.

[74] Hiroyuki Miyamoto, Stefan Schaal, Francesca Gandolfo, Hiroaki Gomi, Yasuharu Koike, Rieko Osu, Eri Nakano, Yasuhiro Wada, and Mitsuo Kawato. A kendama learning robot based on bi-directional theory. *Neural Networks*, 9(8):1281–1302, 1996.

[75] Jean-Baptiste Mouret and Stéphane Doncieux. Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 627–634, 2009.

[76] Jun Nakanishi, Jun Morimoto, Gen Endo, Gordon Cheng, Stefan Schaal, and Mitsuo Kawato. Learning from demonstration and adaptation of biped locomotion. *Robotics and Autonomous Systems*, 47:79–91, 2004.

[77] Ulrich Nehmzow, Otar Akanyeti, Christoph Weinrich, Theocharis Kyriacou, and Stephen A Billings. Robot programming by demonstration through system identification. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 801–806, 2007.

[78] Gerhard Neumann. Variational inference for policy search in changing situations. In *Proceedings of the 28th international conference on machine learning*, pages 817–824, 2011.

[79] Andrew Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, pages 363–372, 2006.

[80] Aaron Nichols, Amber Billey, Peter Spitzform, and Catherine Tran. Kicking the tires: A usability study of the primo discovery tool. *Journal of Web Librarianship*, 8(2):172–195, 2014.

[81] Monica Nicolescu and Maja Matarić. Experience-based representation construction: learning from human and robot teachers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 740–745, 2001.

[82] Masaki Ogino, Hideki Toichi, Minoru Asada, and Yuichiro Yoshikawa. Imitation faculty based on a simple visuo-motor mapping towards interaction rule learning with a human partner. In *The 4th International Conference on Development and Learning*, pages 148–148, 2005.

[83] Eugénio Oliveira and Luis Nunes. Learning by exchanging advice. In *Design of Intelligent Multi-Agent Systems*, pages 279–313. Springer, 2005.

[84] Mark Ollis, Wesley H Huang, and Michael Happold. A bayesian approach to imitation learning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 709–714, 2007.

[85] Hancock P.A., Billings D.R., Schaefer K.E., Chen J.Y., De Visser E.J., and Parasuraman R. A Meta-analysis of Factors Affecting Trust in Human-Robot Interaction. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 53(5):517–527, 2011.

[86] Mykola Pechenizkiy, Seppo Puuronen, and Alexey Tsymbal. Feature extraction for classification in knowledge discovery systems. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 2773 of *Lecture Notes in Computer Science*, pages 526–532. Springer, 2003.

[87] Jan Peters, Katharina Mülling, and Yasemin Altün. Relative entropy policy search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1607–1612, 2010.

[88] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225, 2006.

[89] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

[90] Polly K. Pook and Dana H. Ballard. Recognizing teleoperated manipulations. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 578–585, 1993.

[91] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[92] Chandra Reddy and Prasad Tadepalli. Learning goal-decomposition rules using exercises. In *Proceedings of the 14th International Conference on Machine Learning*, pages 278–286, 1997.

[93] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[94] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, 3rd edition, 2009.

[95] Paul Rybski, Kevin Yoon, Jeremy Stolarz, and Manuela Veloso. Interactive robot task training through dialog and demonstration. In *2nd ACM/IEEE International Conference on Human-Robot Interaction*, 2007.

[96] Paul E Rybski and Richard M Voyles. Interactive task training of a mobile robot through human gesture recognition. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 664–669, 1999.

[97] Joe Saunders, Chrystopher L. Nehaniv, and Kerstin Dautenhahn. Teaching robots by moulding behavior and scaffolding the environment. In *Human-Robot Interaction*, pages 118–125, 2006.

[98] Sham M. Kakade. *On the Sample Complexity of Reinforcement Learning.* PhD thesis, University College London, 2003.

[99] Chang Shu, Hang Ding, and Ning Zhao. Numerical comparison of least square-based finite-difference (LSFD) and radial basis function-based finite-difference (RBFFD) methods. *Computers & Mathematics with Applications*, 51(8):1297–1310, 2006.

[100] William D Smart and Leslie Pack Kaelbling. Effective reinforcement learning for mobile robots. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 3404–3410, 2002.

[101] Jochen J Steil, Frank Röthling, Robert Haschke, and Helge Ritter. Situated robot learning for multi-modal instruction and imitation of grasping. *Robotics and autonomous systems*, 47(2):129–141, 2004.

[102] Alexander L. Strehl, Lihong Li, and Michael L. Littman. Reinforcement learning in finite MDPs: PAC analysis. *Journal of Machine Learning Research*, 10:2413–2444, 2009.

[103] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, 1st edition, 1998.

[104] Richard S. Sutton, Anna Koop, and David Silver. On the role of tracking in stationary environments. In *Proceedings of the 24th International Conference on Machine Learning*, pages 871–878, 2007.

[105] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning.

[106] Takahiro Suzuki and Yuji Ebihara. Casting control for hyper-flexible manipulation. In *IEEE International Conference on Robotics and Automation*, pages 1369–1374, 2007.

[107] John D Sweeney and Rod Grupen. A model of shared grasp affordances from demonstration. In *7th IEEE-RAS International Conference on Humanoid Robots*, pages 27–35, 2007.

[108] Matthew E Taylor, Brian Kulis, and Fei Sha. Metric learning for reinforcement learning agents. In *The 10th International Conference on Autonomous Agents and Multiagent Systems*, volume 2, pages 777–784, 2011.

[109] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685, December 2009.

[110] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 37:1–37:8, 2007.

[111] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[112] Andrea Lockerd Thomaz and Cynthia Breazeal. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Proceedings of the 21st National Conference on Artificial Intelligence*, volume 6, pages 1000–1005, 2006.

[113] Matthew A Turk and Alex P Pentland. Face recognition using eigenfaces. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 586–591, 1991.

[114] Aleš Ude, Christopher G. Atkeson, and Marcia Riley. Programming full-body movements for humanoid robots by observation. *Robotics and Autonomous Systems*, 47:93 – 108, 2004.

[115] Richard M. Voyles. A multi-agent system for programming robots by human demonstration. *Integrated Computer-Aided Engineering*, 8:59–67, 2001.

[116] Shimon Whiteson, Matthew E Taylor, and Peter Stone. *Adaptive tile coding for value function approximation.* Computer Science Department, University of Texas at Austin, 2007.

[117] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[118] Robert H Wortham, Andreas Theodorou, and Joanna J Bryson. What does the robot think? Transparency as a fundamental design requirement for intelligent systems. In *IJCAI-2016 Ethics for Artificial Intelligence Workshop*, 2016.

[119] Panayiotis Zaphiris. *Human Computer Interaction: Concepts, Methodologies, Tools and Applications.* Information Science Reference, 2008.