Mach Learn (2010) 81: 85–98
DOI 10.1007/s10994-010-5202-y

Dimension reduction and its application to model-based exploration in continuous spaces

Ali Nouri · Michael L. Littman

Received: 30 April 2010 / Accepted: 20 June 2010 / Published online: 22 July 2010
© The Author(s) 2010

Abstract The sample complexity of a reinforcement-learning algorithm is highly coupled to how proficiently it explores, which in turn depends critically on the effective size of its state space. This paper proposes a new exploration mechanism for model-based algorithms in continuous state spaces that automatically discovers the relevant dimensions of the environment. We show that this information can be used to dramatically decrease the sample complexity of the algorithm over conventional exploration techniques. This improvement is achieved by maintaining a low-dimensional representation of the transition function. Empirical evaluations in several environments, including simulation benchmarks and a real robotics domain, suggest that the new method outperforms state-of-the-art algorithms and that the behavior is robust and stable.

Keywords Reinforcement learning · Exploration · Kernel regression · Dimension reduction

1 Introduction

Reinforcement-learning (RL) agents need to explore their environments well to be effective (Thrun 1992). A particularly successful and versatile approach to action selection is for agents to plan to reach areas of the state space where their models are uncertain. Agents can be encouraged to reach these states by a type of exploration bonus (Sutton 1990) added to uncertain transitions, sometimes called "optimism in the face of uncertainty" and more recently referred to as RMAX exploration (Brafman and Tennenholtz 2002; Kakade 2003).

Unfortunately, even with sophisticated exploration techniques, the problem of the *curse of dimensionality* still looms large in complex environments. The "curse" refers to the mathematical fact that the volume of a space grows exponentially larger as the dimensionality

Editors: José L Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag.

A. Nouri (✉) · M.L. Littman
Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
e-mail: nouri@cs.rutgers.edu

M.L. Littman
e-mail: mlittman@cs.rutgers.edu

increases. Having to reason about a state space using a finite number of samples, algorithms face extreme challenges in high-dimensional environments. Without any remedy for handling this type of data, algorithms are doomed to failure in complex domains (Chow and Tsitsiklis 1989).

A classic approach to dealing with high-dimensional spaces in machine learning is to explicitly use a simpler representation of data by projecting it to lower dimensional spaces—known as *dimension reduction*. In fact, the history of using dimension reduction in machine learning goes back to several decades ago, with a large number of success stories (Jolliffe 1986). Methods, such as principal component analysis (PCA), have long been used in various scientific disciplines as a preprocessing step for handling high-dimensional data, and are now considered standard for dealing with complex data. More recently however, the applicability of these methods has been extended a great deal, thanks to advances in the field of statistical learning theory. Robust dimension reduction in regression using nonlinear kernel transformation functions is an example of such an advance (Weinberger and Tesauro 2007; Fukumizu et al. 2009).

The idea of dimension reduction has also been studied in the reinforcement-learning community. For example, Kolter and Ng (2009) and Parr et al. (2008) learned the relevant basis functions (for example, from a large pool), when approximating the value function in the context of least-squares temporal difference learning (LSTD). Discarding irrelevant basis functions reduces the number of free parameters and provides a more overfit-resistant estimation. Some research makes an even tighter connection to the dimension-reduction literature by directly using some of the existing techniques and tailoring them to the RL framework. For example, Smart (2004) used manifold learning for low-dimensional representation of the value function, and Mahadevan (2009) proposed a framework using Laplacian operators for representing and controlling Markov Decision Processes (MDPs).

The main contribution of this work is a method for using dimension reduction to attack the exploration problem in continuous state-space problems. Because of the focus on exploration, our research is orthogonal to existing dimension-reduction work in RL, which focused on either statistical efficiency of learning or exploitation. We present a model-based algorithm that discovers low-dimensional structures in system dynamics and uses this information to perform "self-aware" exploration in a more compact space, resulting in a dramatic decrease in the sample complexity.

In this work, we concern ourselves with the sample complexity of algorithms instead of their computational complexity. It is important to note that the algorithm we propose, like any model-based method, still needs an approximate planner to solve its internal model. While the structure-discovery component allows learning to take place in a smaller subspace with far fewer sample points, planning still needs to take place in the original space, which might be computationally burdensome. Of course, since the model-learning in the algorithm is independent of the planning step, any approximate planner that can handle generative models can be used. In this work, we use fitted value iteration (Gordon 1995) or FVI, but one can easily substitute the approximate planner most appropriate for the problem.

We consider reinforcement learning in environments that can be modeled as continuous state MDPs. These domains can be described by a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$, where $\mathcal{S}$ is the state space and is a bounded measurable subspace of $\mathbb{R}^{|\mathcal{S}|}$, $\mathcal{A}$ is a discrete set of actions, T is the transition function that determines the next state given the current state and action. We focus on transition functions that can be written in the form of $s_{t+1} = T(s_t, a_t) + \omega_t$, where the subscripts indicate time, and ω is a white noise. The function $R : \mathcal{S} \to \mathbb{R}$ is the bounded reward function, whose maximum we denote by $R_{\max}$. Finally, γ is the discount factor. The

agent knows about $\mathcal{S}$, $\mathcal{A}$, R and γ, and has to learn the optimal policy by interacting with the environment.[1]

In the first part of the paper, we discuss a particular method of performing dimension reduction for multivariate regression. We then show in the second part how to incorporate this idea into a model-based RL algorithm to estimate the transition function and drive exploration on several test domains.

2 Dimension reduction for regression

The task of dimension reduction for regression (DRR) is to find a low-dimensional representation of the input space such that the transformed data can predict the output independent of the original covariates. To be more precise, let us define the data as a set of observations of the form (x, y), where $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^l$, and the regression as the problem of estimating the conditional probability density of y given x. Therefore, the task of DRR is to find a *transformation function* $\Phi : \mathbb{R}^m \to \mathbb{R}^r$, with $r < m$, such that x and y are conditionally independent given the transformation $\Phi(x)$ (Fukumizu et al. 2009). Of particular interest are techniques that involve linear transformation functions, mainly because of their versatility and speed. For convenience, we use matrix notation $\mathbf{X}$ to denote the row-wise concatenation of x_i^T for $i = 1 \ldots n$. Linear transformation into r-dimensional subspace can then be specified using an $(r \times m)$ matrix.

In this section, we first consider multivariate kernel regression and how to embed dimension reduction in it, and then show how to tailor this method to be most effective in a model-based RL algorithm.

2.1 Metric learning for kernel regression (MLKR)

We build on the work of Weinberger and Tesauro (2007) on univariate kernel-metric learning. Given a query point x^*, the kernel regression outputs $\widehat{y^*}$ as follows:

$$\widehat{y^*} \equiv \hat{f}(x^*) = \frac{\sum_{i=1}^{n} k(x^*, x_i) y_i}{\sum_{i=1}^{n} k(x^*, x_i)}, \tag{1}$$

where $k(.,.) \geq 0$ is referred to as the *kernel* function. A lot of different kernel functions have been studied in the literature, but we focus on the *Gaussian kernel*:

$$k(x_i, x_j) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{d^2(x_i, x_j)}{\sigma^2}},$$

where d is the distance metric and σ is a constant that determines how fast the kernel decays with respect to d. Equation (1) uses the whole dataset for prediction, which might be computationally inefficient; to alleviate this problem, an approximation that uses only the c closest neighbors of x^*—denoted by $\mathcal{N}_c(x^*)$—can be used. This approximation sacrifices a little accuracy for a lot of computational advantage, since the kernel function usually decays very fast and far points don't have much contribution to the estimates. The approximate kernel regression can be written as:

$$\hat{f}(x^*) = \frac{\sum_{x_i \in \mathcal{N}_c(x^*)} k(x^*, x_i) y_i}{\sum_{x_i \in \mathcal{N}_c(x^*)} k(x^*, x_i)}. \tag{2}$$

[1] Extension to the case of unknown reward function is straightforward.

Metric learning refers to the tuning of the distance function in the kernel so as to minimize the regression error. For example, if one of the dimensions of the input space is irrelevant to the true function f, a distance metric that is oblivious to that dimension is expected to achieve better results. We shall use subscripts on k to indicate what metric is being used inside the kernel. For example, k_u denotes the kernel function with the Euclidian metric.

In order to tune the metric, we must first select a differentiable distance function with respect to some parameter θ. This setup allows us to perform gradient descent to find the optimal value. More precisely, let the loss function $\mathcal{L}$ be the cumulative leave-one-out error of the training set: $\mathcal{L} = \sum_i \|y_i - \hat{y}_i\|_2^2$. The metric-learning algorithm updates θ iteratively using the gradient descent rule: $\Delta\theta = -\alpha \frac{\partial \mathcal{L}}{\partial \theta}$, where α is the learning rate. Any differentiable distance function works in this procedure. Here, we use the *Mahalanobis* metric, which can be written as:

$$d_m^2(x_i, x_j) = \|\mathbf{A}(x_i - x_j)\|_2^2, \tag{3}$$

where $\mathbf{A}$ is an $(m \times m)$ parameter matrix. In this equation, we have rewritten the original Mahalanobis metric to make learning easier (Weinberger and Tesauro 2007). It can be shown that:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = 4 \sum_i (\hat{y}_i - y_i) \frac{\sum_j (\hat{y}_i - y_j) \mathbf{A} k_m(x_i, x_j; \mathbf{A}) x_{ij} x_{ij}^T}{\sum_{j \neq i} k_m(x_i, x_j; \mathbf{A})}, \tag{4}$$

where x_{ij} is a shorthand for $(x_i - x_j)$ and k_m is the Gaussian kernel with the Mahalanobis metric. Note that the optimized $\mathbf{A}$ provides a completely data-driven distance metric, so normalizing the data prior to regression—a vital step in regular kernel regression—is not necessary anymore. In fact, the parameter σ and the leading coefficient in the kernel function can also be omitted, as they are captured inside $\mathbf{A}$.

Equation (3) reveals that the kernel regression with Mahalanobis metric is equivalent to regular kernel regression after the transformation $\mathbf{X} \leftarrow \mathbf{X}\mathbf{A}^T$. So, $\mathbf{A}$ plays the role of the transformation function in DRR. It is important to note that this transformation does not explicitly reduce the dimensionality of the data as in a typical DRR application. But, since the kernel regression does not care about the dimensionality of the points as long as the correct distance metric is used, this transformation is enough.

Nevertheless, we can incorporate explicit dimension reduction into the metric learning process to allow for a more sophisticated regression process. This way, we can transform the data into a lower-dimensional subspace as a preprocessing step, and then use a more appropriate regressor afterward. Furthermore, it is generally known that simpler representations provide more noise-resistant learning along with better computational complexities.

If we know the desired target dimensionality, forcing $\mathbf{A}$ to be an $(r \times m)$ matrix ensures that a transformation that maps the data into the r-dimensional space is directly learned. If the target dimensionality is not known ahead of time, we can use an unsupervised dimension-reduction method after $\mathbf{X}\mathbf{A}_{m\times m}^T$ transformation. In particular, it can be shown that directly learning $\mathbf{A}_{r\times m}$ is similar to learning $\mathbf{A}_{m\times m}$ and mapping $(\mathbf{X}\mathbf{A}_{m\times m}^T)$ into an r-dimensional space using PCA (please refer to Weinberger and Tesauro 2007 for more details).

Algorithm 1 highlights the details of the whole process when PCA is used as a second step to provide explicit dimension reduction. The $\mathbf{W}$ in line 7 is the PCA transformation matrix.

We are now ready to introduce a variant of MLKR that is more suitable for model-based RL algorithms.

Algorithm 1 Multivariate MLKR Algorithm

function train($\mathbf{X}, \mathbf{Y}$)
Initialize $\mathbf{A}$.
repeat
  $\Delta\mathbf{A} \leftarrow -\alpha \frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ {using (4)}
until $\|\Delta\mathbf{A}\|_\infty <$ threshold
$\tilde{\mathbf{X}} \leftarrow \mathbf{X}\mathbf{A}^T$;
{explicit dimension reduction step:}
$\mathbf{W} = PCA(\tilde{\mathbf{X}})$;
$\tilde{\mathbf{X}} \leftarrow \tilde{\mathbf{X}}\mathbf{W}^T$;
end function

function test(x)
$\tilde{x} \leftarrow \mathbf{W}x$;
return $\hat{f}(\tilde{x})$ using (2) with k_u kernel, and trained on $(\tilde{\mathbf{X}}, \mathbf{Y})$;
end function

2.2 Factorization of MLKR

For a given regression problem, the minimal of all the input subspaces that maintains the conditional independency of y and x is called the *central subspace*. This concept provides an important insight into the statistical efficiency of the dimension reduction and the corresponding regression, as it signifies what portion of the input data is redundant or irrelevant.

Estimating the transition function of a continuous state-space MDP involves solving a regression problem from $\mathbb{R}^{|\mathcal{S}|}$ to itself, with the target covariates being the next-state components. A large class of real-life environments, including most of the physical control problems have a factorized transition function, in which the individual components of the next state are independent of each other (i.e. $p(y(i)|x, y(j)) = p(y(i)|x)$ when $i \neq j$). In fact, coming up with a control problem that is not in this class is not an easy task. For this type of environments, we introduce a factorized variation of MLKR, or F-MLKR, that achieves a better statistical efficiency. This improvement is achieved by breaking up the original regression into several easier ones with smaller central subspaces.

This extension is pretty straightforward to construct: The $\mathbb{R}^m \rightarrow \mathbb{R}^l$ regression is broken up into l univariate MLKR problems, one for each component of the output space. Upon receiving the training data set, the algorithm feeds $\{(x_1, y_1(j)), \ldots, (x_n, y_n(j))\}$ to the j-th MLKR learner, where $y_i(j)$ is the j-th component of y_i. To estimate the value of a query point x^*, it queries each MLKR learner and constructs $\widehat{y^*}$ using the output of the learners.

It can be shown that the central subspace of each univariate regression is smaller than or equal to the central subspace of the original multivariate formulation. In fact, since each univariate regression is dealing with only one component of the output, less information from the input space is typically needed, yielding smaller subspaces. We demonstrate this by a simple example. Consider learning a simple function $f(x) = I(x) + rd(x)$, where I is the identity function and rd shifts the components of x downward. For this specific function, the dependency set of all the output components is all of the input variables, and therefore the whole input space is required to describe f. However, each output component depends on only two dimensions of the input.

The factorized MLKR turns the regression into l sub-regressions and one might be concerned about accumulation of errors caused by this process. But fortunately, it is easy to see

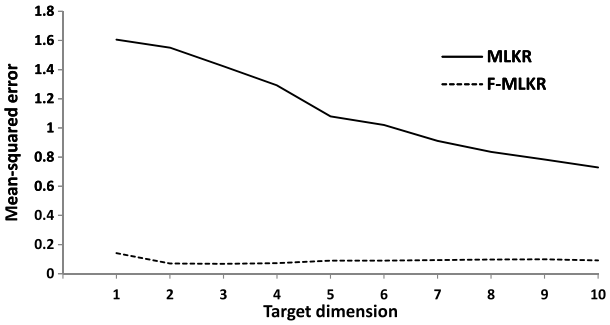

Fig. 1 Comparison of MLKR and F-MLKR on a simple regression problem

that the error of F-MLKR will not be more than the error of the original MLKR because of the independency of the output variables. On the other hand, smaller central subspaces of the factored regressors create exponentially better estimations due to the properties of the curse of dimensionality.

As an example, we estimated the above function in $\mathbb{R}^{10}$ using the two methods. Figure 1 shows what happens if we force different target dimensionalities on the regressors. To produce this graph, we generated 100 points uniformly distributed in the unit square and used the above function plus a small amount of Gaussian noise to construct the training set. The x-axis shows the internal dimensionality we forced on the regressors. The y-axis shows the mean-squared-error measured on another set of 100 randomly selected points.

Since only one dimension is statistically sufficient to output each component of the output (using the linear combination of the two dependent components), F-MLKR quickly achieves good performance, even when the algorithm has to map the input data into scalars. On the other hand, MLKR requires all the input dimensions in order to maintain the link between the input and output. That is why the result for MLKR improves as more dimensions are allowed in the transformation. MLKR cannot solve the regression problem as well as F-MLKR even when it uses the whole input space, because a training set of 100 points is simply not enough to cover a 10-dimensional space.

3 The proposed algorithm

This algorithm, which we call *Dimension Reduction in Exploration* (DRE), is derived in the spirit of several published papers in model-based reinforcement learning that use model uncertainty to drive the exploration toward parts of the state space in which the algorithm is uncertain about its predictions (Brafman and Tennenholtz 2002; Kakade 2003). More specifically, we build on the work of Nouri and Littman (2008) that introduced the concept of *continuous knownness* for exploration.

Knownness is a quantity in $[0 \ldots 1]$ defined on the state-action pairs—denoted by $\tau(s, a)$—and is a measure of how certain the algorithm is about the dynamics of the environment from state s when action a is performed. Depending on the knownness value, the algorithm mixes its internal estimation of the transition function with another function that transitions to an imaginary state with the highest possible value. This augmented estimation of the transition function creates bonus values that are inversely proportional to the knownness values; therefore, it encourages the agent to reach states that are less known.

More precisely, DRE uses $|\mathcal{A}|$ F-MLKR multivariate regressors to estimate the transition function, each responsible for estimating the next state for one action.[2] Each F-MLKR in turn consists of $|\mathcal{S}|$ MLKR regressors inside, each responsible for estimating one of the components of the next state. Let $\text{MLKR}(a, i)$ be the univariate MLKR regressor responsible for estimating the i-th component of the next state when action a is used. Upon receiving a query point (s, a), the estimated transition function—denoted by $\hat{T}$—returns $\hat{s'}$ as the next state, where $\hat{s'}(i)$ is the output of $\text{MLKR}(a, i)$.

Furthermore, let s_f be a new special state with self-loop transition on all actions and a reward of $R_{\max}$. DRE constructs its internal model as $\hat{M} = \langle \mathcal{S} + s_f, \mathcal{A}, \hat{T}', R, \gamma \rangle$, where the augmented transition function $\hat{T}'$ is computed as follows:

$$\hat{T}'(s, a) = \begin{cases} s^f, & \text{w.p. } 1 - \tau(s, a) \\ \hat{T}(s, a), & o.w. \end{cases} \tag{5}$$

The knownness function $\tau(s, a)$ is computed using the corresponding F-MLKR regressor. This regressor computes the knownness of s as the minimum of that state's knownness in all of its internal MLKRs:

$$\tau(s, a) = \min_i (\tau(\text{MLKR}(a, i); s)), \quad i = 1 \ldots |\mathcal{S}|. \tag{6}$$

The knownness function for each individual MLKR regressor is computed based on the same local smoothness principle that kernel regression is based on: nearby points have similar outputs. In this work, we use a simple function to capture this:

$$\tau(\text{MLKR}(a, i); s)) = \frac{1}{c} \sum_{x \in \mathcal{N}_c(\tilde{s})} k_u(\tilde{s}, x), \tag{7}$$

where $\tilde{s}$ is the transformation of $\text{MLKR}(a, i)$ applied to s.

After the construction of $\hat{M}$, the algorithm uses an approximate planner to find a near-optimal policy for it. It then takes the greedy action according to this policy. While any planning algorithm for solving continuous state MDPs with generative models can be applied, we use FVI in our implementation as mentioned earlier.

Algorithm 2 shows how the agent learns in the environment.

Algorithm 2 DRE: A model-based algorithm for continuous state space MDPs

```
for all timesteps t do
    Observe the transition ⟨s, a, r, s'⟩, and add ⟨s, s'⟩ to the history list h_a.
    if t mod planFreq = 0 then
        for all actions a:
        Use h_a to train a-th F-MLKR regressor.
        Construct M̂ and use FVI to solve it. Denote the optimal policy by π*.
    end if
    execute action π*(s').
end for
```

[2] We can construct DRE with MLKR instead of F-MLKR in a similar fashion.

3.1 Discussions

There are two important characteristics that we believe are vital to the success of DRE. First, the algorithm uses metric learning for estimating the transition function, and second, it computes the knownness function in a subspace of the original state space.

One of the most important properties of dimension-reduction techniques in regression is that they provide stable approximation when the sample size is small. In fact, many practical applications of these methods are when the number of samples is in the order of the number of variables, in which case the classic approaches typically fail. Therefore, DRE is able to build very realistic models of the world in the early stages of learning, due to the efficiency of dimension reduction in regression.

The space in which the knownness is computed directly affects the sample complexity of the algorithm. For a query point to have a high knownness value, several points need to exist in its vicinity. Therefore, covering a space with known points requires a training set that is exponential in size with respect to the dimensionality of that space. By reducing the dimensionality of the space in which the knownness is computed, far fewer samples are needed to get high knownness values for the entire space.

The computational complexity of relearning the metric every $planFreq$ steps seems burdensome, because $|\mathcal{A}| \times |\mathcal{S}|$ gradient descent instances need to be solved. However, our experiments indicate that the most time-consuming component of the algorithm is still the planning step. Part of this phenomenon stems from the way gradient descent searches the solution space. If we use the current $\mathbf{A}$ as the starting point of the gradient descent (Line 2 of Algorithm 1), after performing dimension-reduction once or twice, the starting point is usually very close to the optimal solution. As a result, gradient descent returns very quickly. If local optima are a concern and we can afford more computation, we can start the search using several initial matrices, though we did not see much improvement using this technique in practice.

DRE is closely related to factored RMAX (Strehl 2007) and SLF-RMAX (Strehl et al. 2007) in finite spaces. Factored RMAX takes in the dependency graph of input/output variables in the form of a Dynamic Bayes Net, or DBN, from the user and uses this structure to reduce the size of the model class of possible transition functions. This reduction results in exponential speedup in the sample complexity. SLF-RMAX is a followup of factored RMAX that discovers the structure of the DBN itself during the learning.

In the next section, we empirically evaluate the performance of DRE, particularly focusing on the characteristics we mentioned.

4 Experimental results

We empirically evaluate the performance of DRE in two widely used simulation domains in the RL community and a real robotics task.

4.1 Domains

Generalized Mountaincar is adapted from Mountaincar, which is a well-studied environment in the RL community (Sutton and Barto 1998). In this domain, an underpowered car tries to climb to a hilltop, but has to build up speed via several back and forth trips across the valley (Fig. 2(a)). This problem has two state variables (horizontal position and velocity) and three actions (accelerate left, neutral and forward). The agent receives -1 reward each timestep and the episode ends once the car clears the hilltop.

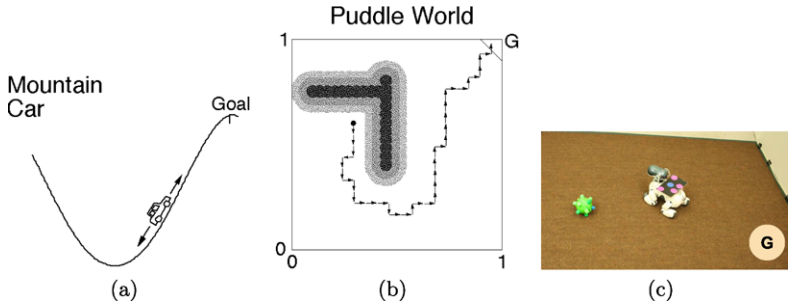

Fig. 2 (**a**) and (**b**) Mountaincar and Puddleworld borrowed from Sutton (1996), (**c**) The Bumbleball domain

To emphasize the effect of dimension reduction, we extend the original Mountaincar to n-Mountaincar, in which n cars are controlled simultaneously. Thus, the dimensionality of n-Mountaincar is $2n$ and there are $3n$ actions available. The goal of the environment is still to steer the first car to the hilltop, so the average number of timesteps to the goal is the same in all n-Mountaincar domains. By comparing the results of an algorithm for different values of n, we can determine how much increasing dimensionality degrades the performance of the algorithm.

Generalized Puddleworld is adapted from Puddleworld (Sutton 1996), in which an agent is placed inside a two-dimensional unit square and has to navigate to a small goal region while avoiding two puddles on the way (Fig. 2(b)). There are four available actions (U, D, R, L) that move the agent in the intended direction by 0.05 with the addition of a small Gaussian noise $\mathcal{N}(0, 0.01)$. Each step yields a -1 reward, but additional penalties are given for entering the puddles.

Generalized Puddleworld extends the original version to n-dimensional spaces in a natural way: In this version, the agent is placed in the unit square of the n-dimensional space and has $2n$ available actions. Actions $(2i - 1)$ and $2i$ move the agent along the i-dimension. Puddles are the projection of the original 2D puddles into the new space.

Bumbleball World is a robot navigation task carried out by a four-legged Sony Aibo robot. In this domain, the robot has to navigate to a goal region while avoiding a moving obstacle in the environment (we used a randomly moving toy ball called the Bumbleball as the obstacle). The state space has five dimensions: the position and orientation of the robot in the field, and the ball's position (see Fig. 2(c)). The state information is computed using an overhead camera. There are six available actions to the robot: move forward and backward, turn left and right, and strafe left and right. The low-level gait pattern for these actions were hard-coded into the robot and were executed for 1 second for each timestep. Each step results in -1 reward, unless the robot hits the ball in which case it gets -40. The episode ends when the robot reaches the goal, in which case it receives 20 reward, or 100 steps is passed.

4.2 Experiments

The first experiment was designed to demonstrate the statistical efficiency of learning the transition function using metric learning with dimension reduction. For this reason, we ignored the exploration problem and evaluated algorithms on offline data. For each i-Mountaincar, we collected 500 transitions using a policy that selects actions randomly from 100 random start states and runs for 5 timesteps. Each algorithm learned a model using the data. We then chose another 100 random start states as the test set, and evaluated

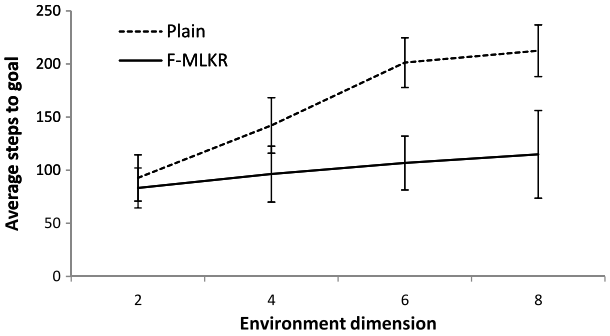

Fig. 3 Batch reinforcement learning in n-Mountaincar

each algorithm by running its learned policy from these points for 300 steps or until the goal was reached. For the sake of statistical significance, the experiment was repeated 20 times. The y-axis of Fig. 3 is the average steps to the goal from the test points. The x-axis shows the dimensionality of the environment. Error bars represent standard deviation.

We compared two algorithms: one that used regular kernel regression (denoted by *Plain*)[3] and DRE that used F-MLKR. The σ parameter of Plain was hand tuned to 0.3. For DRE, the c was set to 20 and the PCA cutoff to 70%. FVI used 20 points per dimension for both algorithms.

As mentioned earlier, the average number of steps to the goal is the same in all n-Mountaincars, and any performance degrade is only due to the model-approximation error. While kernel regression is a powerful technique, it is still susceptible to the curse of dimensionality. Thus, the performance of Plain degraded as the dimensionality increased. At 3-Mountaincar, the space became simply too large for the 500 training points to cover. F-MLKR on the other hand, managed to keep the dimensionality low even in 4-Mountaincar because of the fact that only a subset of dimensions were necessary to predict each component of the output. The internal dimension of the univariate MLKR's never went above 2 in this experiment.

The second experiment tested algorithms in an online setting. We first used the generalized Puddleworld. Each algorithm ran 50 episodes, with a cap of 300 steps, in each i-dimensional Puddleworld. Results are averaged over 20 runs. We compared two algorithms: DRE with F-MLKR and Plain. The exploration in Plain was handled by computing the knownness as:

$$\tau(s,a) = \frac{1}{c} \sum_{x \in \mathcal{N}_c(s)} k_u(s,x), \tag{8}$$

using the corresponding kernel regressor for action a. This is similar to (7), but with the kernel computed in the *original* space.

Both algorithms set *planFreq* to 100. Parameter c in the approximate kernel regression was set to 20. Figure 4 shows the average performance-per-episode (cumulative reward divided by number of episodes) when the dimensionality changes. Similar to the previous result, the F-MLKR maintained the low-level representation in all the domains (it never used more than one dimension in each MLKR). Therefore, DRE created high knownness values very quickly, as they were computed in small spaces. The Plain algorithm was not very successful in high-dimensional Puddleworlds.

[3]This algorithm is very similar to Jong and Stone (2007).

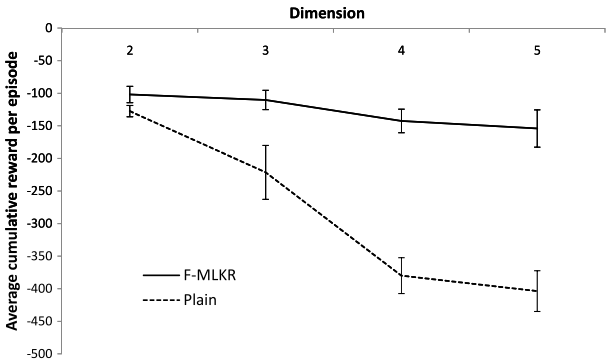

Fig. 4 Online evaluation of two algorithms in n-dimensional Puddleworld

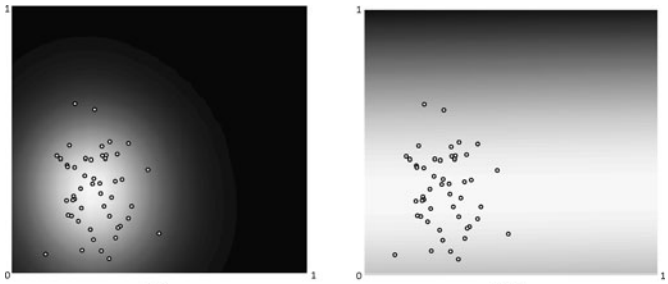

Fig. 5 Comparison of knownness using regular kernel regression (**a**) and MLKR (**b**)

In order to get more insight into why DRE was superior to Plain, we investigated the knownness function in a simple experiment. We selected part of the transition function of 2D Puddleworld: $f(x) = x(2) + 0.05 + \mathcal{N}(0, 0.01)$. We then compared the knownness function using the two equations (8) and (7) after training with 50 samples. Figure 5 shows the training points along with the knownness values across the entire space. The darker parts of the image indicate smaller knownness values (0 is black and 1 is white). The MLKR regressor discovered that the horizontal dimension is irrelevant to the function, and generalized the knownness values across this axis (see Fig. 5(b)). On the other hand, since the Plain kernel regressor works in 2D, it produced much smaller knownness values because of the weaker generalization.

We then tested the algorithms in generalized MountainCar with the same setup. Figure 6 shows a more detailed graph of learning which reveals the total reward obtained in each episode of learning. We ran Plain (Fig. 6(a)) and DRE (Fig. 6(b)) in 1-MountainCar (the solid lines in the two graphs) and 3-MountainCar (the dotted lines). As the graphs show, both algorithms performed fairly comparatively in 1-MountainCar; but, Plain completely failed to learn in 50 episodes in 3-MountainCar, whereas DRE didn't suffer much.

We have tried the proposed algorithm in several other benchmarks and have found it very robust and stable. One of the contributing factors to this behavior is the very few number of parameters to tune, and that they are not very sensitive. In fact, we used only one set of parameter values to run the algorithm across all the environments (except for *planFreq* for computational purposes). However, the kernel width typically plays an important role in the prediction accuracy of the regular kernel regression and has to be picked carefully every time a new problem is to be solved.

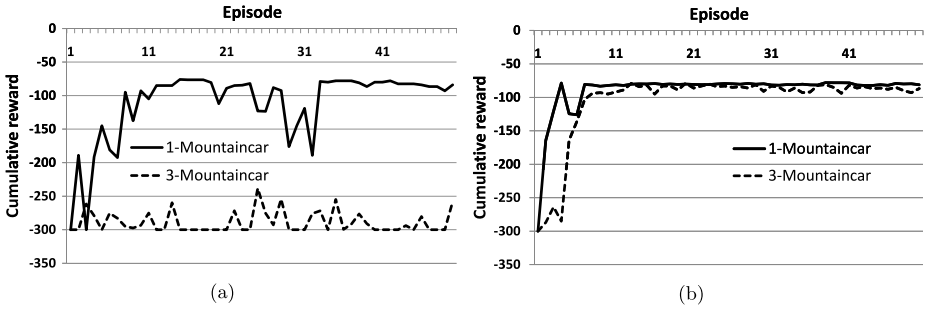

Fig. 6 The learning curve of (**a**) Plain and (**b**) DRE on two environments: 1-Mountaincar and 3-Mountaincar

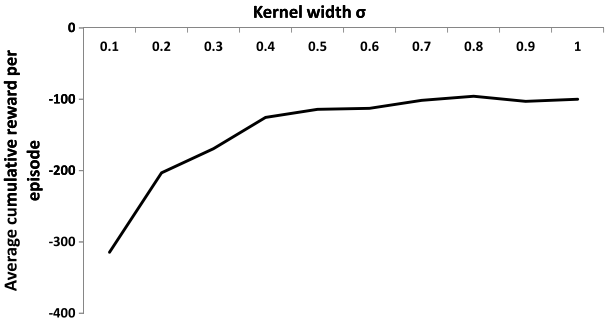

Fig. 7 Effect of different values of σ in the Plain algorithm in 2D Puddleworld

As an example, Fig. 7 shows the result of picking different values for σ in the Plain algorithm in the 2D Puddleworld. Considering the range between the optimal and random policies, which was between -23 and -453, we can see how sensitive the algorithm is to this parameter. To make matters more complicated, we note that the best kernel width for this domain will not usually be a good choice for other domains. (The best width we found that performed reasonable across the domains we tried was 0.3, while the best value for 2D Puddleworld was much higher.) This experiment shows how useful the role of automatic structure discovery is when not a lot of prior knowledge is available for the problem at hand.

The final experiment we considered was the Bumbleball world. The task was to learn as much as possible using 1000 samples. Three algorithms were evaluated: DRE, Plain, and a model-based algorithm that took in the dependency structure from the user a priori and used it as the transformation mapping—we called it $D_{\max}$. The dependency structure was hand crafted in several DBNs using the fact that the movement of the ball is generally independent of robot and vice versa. Each DBN described the dependency of one of the components of the next state on the current state when a particular action was chosen. While these DBNs are prohibitively many to include in the paper, it is very straightforward to generate them. We set the *planFreq* parameter to 20 for all the algorithms.

Table 1 summarizes the performance of these algorithms averaged over 3 runs. Obviously, $D_{\max}$ performed better than DRE because the structure was given to it, but DRE was also able to learn this 5-dimensional task using only 1000 samples, which is quite remarkable. Plain had no chance of learning given the very few number of samples. To get a sense of what the numbers mean, we note that a randomly moving dog collects about -850 reward in each episode.

Table 1 Performance of three algorithms in the Bumbleball domain

Algorithms:	$D_{\max}$	DRE	Plain
Total cumulative reward:	−1865.3	−2290.6	−7437.3
Number of collisions:	25.6	37	165.6
Percent finished episodes:	83.2%	76%	9.4%

One of the interesting behaviors of the algorithm was that it quickly learned to avoid the ball even though it hadn't learned the optimal policy quite yet.

5 Conclusion

In this paper, we proposed a new model-based algorithm that automatically discovers dependency structures in the dynamics of the environment. To achieve this objective, it uses a technique from the dimension-reduction literature to maintain a low dimensional representation of the transition function. A specially constructed knownness function in the reduced space is used as the exploration strategy of this algorithm.

We showed that this technique boosts the performance of the algorithm in two major ways: (1) using dimension reduction provides a statistically more efficient way of learning the transition function, and (2) the computation of the knownness function in the reduced subspace helps dramatically decrease the sample complexity of the algorithm. Experimental results in two simulation benchmarks and a real robotics task confirm that the new method significantly speeds up the learning process and that the behavior is robust and stable across domains.

References

Brafman, R. I., & Tennenholtz, M. (2002). R-MAX—a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, *3*, 213–231.

Chow, C. S., & Tsitsiklis, J. N. (1989). The complexity of dynamic programming. *Journal of Complexity*, *5*(4), 466–488.

Fukumizu, K., Bach, F. R., & Jordan, M. I. (2009). Kernel dimension reduction in regression. *Annals of Statistics*, *37*, 1871.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. In *Proceedings of the twelfth international conference on machine learning*.

Jolliffe, I. (1986). *Principal component analysis*. New York: Springer.

Jong, N. K., & Stone, P. (2007). Model-based exploration in continuous state spaces. In *The 7th symposium on abstraction, reformulation, and approximation*.

Kakade, S. M. (2003). *On the sample complexity of reinforcement learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London.

Kolter, J. Z., & Ng, A. Y. (2009). Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th international conference on machine learning*.

Mahadevan, S. (2009). Learning representation and control in Markov decision processes: new frontiers. *Foundations and Trends in Machine Learning*, *1*(4), 403–565. http://dx.doi.org/10.1561/2200000003.

Nouri, A., & Littman, M. L. (2008). Multi-resolution exploration in continuous spaces. In *Proceedings of the 22nd neural information processing systems*.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., & Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on machine learning*.

Smart, W. D. (2004). Explicit manifold representations for value-function approximation in reinforcement learning. In *Proceedings of the 8th international symposium on artificial intelligence and mathematics* (pp. 25–2004).

Strehl, A. L. (2007). Model-based reinforcement learning in factored-state MDPs. In *Proceedings of the 2007 IEEE international symposium on approximate dynamic programming and reinforcement learning*.

Strehl, A. L., Diuk, C., & Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. In *Proceedings of the 22nd national conference on artificial intelligence*.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th international conference on machine learning*.

Sutton, R. S. (1996). Generalization in reinforcement learning: successful examples using sparse coarse coding. In *Proceedings of the 8th advances in neural information processing systems*. Cambridge, MA.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction*. Cambridge: MIT.

Thrun, S. B. (1992). The role of exploration in learning control. In *Handbook of intelligent control: neural, fuzzy, and adaptive approaches* (pp. 527–559). New York: Van Nostrand Reinhold.

Weinberger, K. Q., & Tesauro, G. (2007). Metric learning for kernel regression. In *Proceedings of the 11th international conference on artificial intelligence and statistics*.