A Factored Model Learner for Robotic Tasks

Abstract—To apply reinforcement-learning techniques to real-life problems such as robotics tasks, we have to use generalization methods that allow the learner to reason about unvisited parts of the state space. Only a very restricted class of value-function approximation techniques are proven to be stable, so, in this work, we pursue another approach that applies function approximation on the parameters of the model instead of the value function. We define a model-based learning algorithm that can exploit prior structural knowledge and evaluate it on three tasks. The first is a simulated benchmark domain known as PuddleWorld, for which a wide array of algorithms are shown to perform well. When we move away from simulation and address two robotics tasks inspired by PuddleWorld, we find that the physical world introduces some complications and not all algorithms are successful. We demonstrate that our model-based learning algorithm successfully leverages the available information to speed up learning in our robotics tasks.

I. INTRODUCTION AND RELATED WORKS

Reinforcement learning (RL) [1] is a learning framework that is a good fit for robotics tasks, since often a robot is placed into an unknown environment and has to learn to behave optimally only through direct interaction. Nevertheless, RL has received little attention among roboticists, for at least two significant reasons. First, while a few challenging real-life applications have been successfully attacked in the RL framework [2, 3], most applications of RL methods have been in the realm of toy domains. As such, the complications and characteristics of robotics problems have not been addressed. Second, theoretical sound and practical algorithms for continuous spaces have been sparse in the RL literature, and continuous state spaces are crucial to robotics and control problems. Continuous domains cannot be treated as a simple variation of discrete ones, which are what most of the traditional algorithms address. When continuous domains are discretized naively, their sample complexity suffers—experience requirements typically grow exponentially with the dimensionality of the state space. Sample complexity is very important in robotics, since real-life experience is expensive to collect. In this work, we attempt to address these issues with the hope of bringing RL and robotics closer to each other.

PuddleWorld is a well-known test domain in the RL community [4]. Here, an agent is placed inside a 2d region and has to navigate to a small goal region while avoiding some obstacles along the way. In this work, we examine PuddleWorld along with two novel Sony-Aibo-based robotics tasks inspired by it. The first task, simple BumbleBall, replicates the PuddleWorld domain using an Aibo robot. We find that this simple transition from a simulation to a real robotics domain renders the problem much more challenging—the dimensionality increases and key assumptions such as full observability and Markovianness of the environment are violated. As a result, existing algorithms that work well in PuddleWorld fail in for simple BumbleBall. The second domain, hard BumbleBall, replaces the stationary obstacles with a dynamic one—a randomly moving ball. Whereas PuddleWorld is a 2-dimensional problem, hard BumbleBall is a very stochastic 5-dimensional domain. By examining existing and novel algorithms on these domains, we are able to get insight into the advantages of our factored model learner in robotics tasks.

Treating each state of the world independently by keeping lookup tables, the best studied RL algorithms are really only applicable to small, finite state spaces. Wise generalization across the states—using experience with some states to reason about unvisited parts of the space—is crucial to the success of the algorithms in continuous spaces. Recently, there have been some efforts in developing algorithms specifically for the continuous spaces [5, 6, 7].

Among different RL paradigms, value-based methods are the most widely used techniques in continuous domains. These methods apply function approximation directly to the value function. The work of Gordon [8] showed a small class of function approximators called averages are guaranteed to converge when applied to the value function. Even linear function approximators are not guaranteed to be stable [4]. As such, a lot of research has focused on finding function approximators inside the class of averages [6, 9] or developing rule along with more general function approximators [10] that do not have performance guarantees.

Model-based algorithms have shown to be very promising in terms of smart exploration and sample complexity in discrete domains. Kearns and Singh [11] introduced the first model-based algorithm that provably achieves near-optimal policy in polynomial time in the PAC (Probably Approximately Correct) setting. Later on, several other algorithms, including RMAX [12], also emerged that achieved the same kinds of bounds. Strehl and Littman [13] and ... showed that in the cases, where the MDP is represented in the factored form and there are structural constraints on the dependency of factors on each other, these information can be incorporated into the learning algorithm to gain exponential speed up in the sample complexity.

Given these interesting results, it is very appealing to use the same kind of ideas in the continuous setting with the help of a function approximator. In fact, the idea of function approximation in model-based algorithms is not new; Atkeson et al. [14] suggested applying locally weighted regression to learn the behavior of a control problem a decade ago.

Sony Aibo is a 4-legged dog robot.
However, we are not aware of follow up work that built on this suggestion. Recently, Kakade et al. [7] presented a general framework for model-based algorithms that provably converges to near-optimal policies in polynomial time in a continuous framework. In a similar vein, Strehl and Littman [15] presented a learning method that has a sample complexity that grows linearly to the number of dimensions if the transition function is known to have a linear form. Jong et al. [16] took a more general approach and showed that averagers can be used in a model-based setting to get the same results as value-based methods, often with lower sample complexity.

In this work, we examine three algorithms from the RL literature. Two model-based and one value-based algorithms are designed for finite MDPs, and a value-based algorithm is designed specifically for continuous domains. We then propose a practical model-based algorithm that has both a smart exploration built into it, and at the same time, has the ability to take advantage of available information from the underlying MDP to speed up the learning. While not significantly faster on the simulated PuddleWorld task, this new algorithm solves our robotics tasks much faster than the other methods.

The rest of the paper is organized as follows. In Section II, we introduce our experimental domains. Section III discusses different ways of applying function approximation within an RL algorithm. In Section IV, we introduce our new model-based algorithm, called MPA. Section V provides experimental results for all the three domains, and finally Section VI concludes the paper.

II. EXPERIMENTAL DOMAINS

Our experimental domains are variations of a well-known RL benchmark domain called PuddleWorld [4]. In PuddleWorld, an agent is placed inside the bounded region (0..1,0..1) and its goal is to move to a small goal region using 4 available actions: {north, east, south, west}. Along the way to the goal there are 2 puddles, which the agent has to avoid. The state space of this domain is the X and Y coordinates of the agent in the world. Each of the 4 available actions move the agent in the intended direction by some fixed step size to which a small amount of Gaussian noise is added. Each step in the environment counts as −1 reward if the agent is outside of a puddle and a big negative penalty of -50 if it steps into a puddle.

The simple BumbleBall task extends the PuddleWorld by implementing the same environment using a Sony Aibo robot. A box of approximately 4 × 6 feet is used as the environment and the robot has to go to one of the corners while avoiding two (imaginary) puddles. A top camera provides the location of the dog in the box in terms of X, Y and the orientation, but does not provide information about the location of the puddles. Figure 1 shows a snapshot of the environment captured by the top camera. The robot has 4 available actions (forward, backward, turn left, turn right). These actions are more restrictive than the ones in PuddleWorld, so the robot is no longer holonomic and has to mix turns and walks to go in a desired direction. This environment is relatively harder than the PuddleWorld due to its more restrictive set of actions and higher dimensionality.

In our setup, the learning algorithm runs on a computer that is connected to the camera and the Aibo using a wireless network, and a light-weight program on the Aibo executes the actions as they arrive over the network. The step time of each action is approximately 1100 ms, during which the following events happen: (1) a vision program uses the captured image from the camera to locate objects in the box and sends the information to the learner; (2) the learner observes the current state, computes the action for execution and sends it to the Aibo—the learner waits 1100 ms before asking for the next state from the camera; (3) the Aibo executes actions as they arrive—if it doesn’t receive an action for 1300ms, it stops. This setting introduces several unavoidable sources of noise to the system: The localization is not perfect because of the camera and the vision software we used, making the environment slightly unobservable. There is a delay between the time the image is taken by the camera and the time the next action is executed: this delay is caused by processing time of the vision software and the learner, the way programs run on the Aibos, and also network delays. During this time, the robot is moving in order to achieve a smooth movement, again making the environment slightly unobservable. Finally, the Aibo’s low level actions can have different effects depending on the current physical status of the robot. For example, a move forward command behaves differently depending on whether the Aibo is already moving forward or turning, making the environment mildly non-Markovian.

The hard BumbleBall introduces the BumbleBall, which is a mine-looking powered toy ball that moves around randomly. The main difference from the simple BumbleBall task is that the imaginary stationary puddles are replaced by this

2These high level actions are pre-built into the robot as gait patterns.
stochastically moving ball. The robot receives -40 penalty if it touches the ball. To give the Aibo a little more freedom, we introduced two more actions (strafe left/right), so that the robot has a better chance of escaping the fast moving ball. The location of the ball is provided to the Aibo as part of the state, whereas it’s sometimes not very easy to perceive the full state of the world using sensory information in the robot. Later on, we show how some RL methods can benefit from extra available information about the environment. In particular, we exploit a natural factorization of the transition function $T$.

While value-based methods try to directly estimate the value function (or similar ones like the Q-function [1]) without explicitly constructing $T$ and $R$, model-based methods try to estimate $T$ and $R$ from experience data and compute the value function only indirectly using this internal model of the environment. Most of these methods in their original form use tabular representations for maintaining their parameters [18, 12], but this method cannot be applied in continuous spaces. In order to be able to reason about the infinite set of states using only a finite representation, function approximation must be used to compactly represent the parameters of these functions [9, 7, 5].

### A. Model vs. Value Approximation

Here, we briefly compare pros and cons of applying function approximation in the two paradigms of model-based and value-based RL algorithms, focusing more on the issues encountered while designing and solving our robotics task rather than a general theoretical treatment.

Value-based methods have to only approximate one function, namely the optimal value function. This function is always deterministic no matter how complicated the environment is, which makes it simpler to represent than a full stochastic transition model. Value functions in the finite case can take on only $|S|$ different values, so the complexity of this function is proportional to the size of the state space (or $|S| \times |A|$ in the case of Q-functions). On the other hand, model-based algorithms have to learn $T$, which is a function from states to probability distributions over states with complexity proportional to $|S|^2 \times |A|$. Another advantage of value-based methods is that they can directly use the outcome of their function approximator for decision making, while model-based learners have to solve their internal model, specified implicitly by their function approximators—a task that is quite computationally challenging by itself. The collection of these advantages has made value-function approximation very appealing and the preferred approach in most RL research.

On the other hand, model approximation has some advantages over value approximation. The value function is the result of applying the nonlinear $\max$ operator to the two base

In the discrete case, where the state space is finite, there are several ways of computing the value function and optimal policy given the description of an MDP [17]. In the RL setting however, the description of the environment is not fully available and the agent has to learn the optimal policy only through direct interaction with the environment. Theoretical studies in the RL literature usually assume that the agent has prior knowledge of $S$, $\gamma$, and $R_{\text{max}}$, and not $T$ and $R$. Unfortunately, this set of information is usually altered when dealing with real-life systems. For example, it’s usually easy to give the robot access to the immediate reward function, whereas it’s sometimes not very easy to perceive the full state of the world using sensory information in the robot. Later on, we show how some RL methods can benefit from extra available information about the environment. In particular, we exploit a natural factorization of the transition function $T$.

In the reinforcement-learning framework, the environment is often modeled as a Markov Decision Process (MDP), which can be defined as a tuple $\langle S, A, T, R, \gamma \rangle$. In the continuous worlds that interest us, $S$ is a bounded subspace of $\mathbb{R}^m$ with $m$ denoting the dimensionality of the problem ($S$ is a finite set in the discrete case); $A$ is a discrete set of actions $^3$, $T$ is the transition function that governs the evolution of the system: given a state $s \in S$ and an action $a \in A$, $T(s', s, a)$ is the probability of going to $s' \in S$ given that we started in $s$ and applied action $a$. $R : S \rightarrow \mathbb{R}$ is the bounded reward function, whose maximum we denote by $R_{\text{max}}$. And finally, $\gamma$ is the discount factor in the range of $[0, 1)$, which signifies the importance of future rewards. Given this definition, a policy $\pi$ is a mapping from $S$ to $A$ that prescribes an action to be taken in each state. The optimal value of a state, denoted by $V^*(s)$, is the maximum expected discounted sum of future rewards an agent can garner starting from $s$ and following a policy. Any policy that achieves $V^*$ in all states is called an optimal policy, denoted $\pi^*$. It is known that under very mild assumptions, at least one $\pi^*$ exists for any MDP and its value function is the fixed point of the following equation, the Bellman equation [17]:

$$V^*(s) = R(s) + \gamma \max_a \int_{s' \in S} T(s', s, a)V^*(s')ds'.$$ (1)

$^3$A continuous action representation can also be used with our algorithm.
functions $T$ and $R$ in Equation 1. This makes it usually a lot more complex function to learn. Figure 3 shows both the value function and transition function in its relative form as learnt by MPA IV. In fact, even though we usually impose some smoothness assumptions on the dynamics of the system, the value function is not smooth in most cases [19, 20]. In contrast, the shape and complexity of the transition functions can often be estimated at design time, something that is very important for selecting the type of function approximator as well as its parameters. Extra information about the environment can greatly reduce the complexity of a model-based learner, as we’ll show later, whereas similar decompositions can be very difficult to apply to value-based methods. For example, reward functions are mostly available to the robot as we have hand-coded them in the design stage; this information can be directly imported to the model-based learner. (In fact, we do so in all our experiments in this paper.) Some basic information about the transition function is also usually available to the designer of the system, like what state variables might affect other variables. This type of information has proven to result in an exponential speedup in sample complexity of model-based learners.

But, perhaps the most important advantage of applying function approximation on the parameters of the model, instead of state values, comes from the way these functions are trained. Regressors estimate a function of the form $F: \mathbb{R}^m \rightarrow \Delta(\mathbb{R})$ via a finite set of training samples $D = \{(\mathbf{X}_l, y_l), l = 1, ..., n\}$. $\Delta(\mathbb{R})$ is a probability distribution over $\mathbb{R}$ and signifies the fact that $F$ could be a stochastic function. $m$ is the dimension of the regressor and $n$ is the sample size. In the supervised learning framework, the samples $(\mathbf{X}_l, y_l)$ are supposed to be drawn from the target function $F$. Unfortunately, in the case of value-function approximation, we don’t have access to the target function; after all, the whole purpose of learning is to compute this function, and if we had the ability to query it, we didn’t have to learn anything. Researchers have made efforts to adapt regressors to generate input samples for itself in a recursive manner [2, 8]. In such scenarios, Bellman backups are performed on the current estimate of the value function to generate more and more accurate estimations. Gordon [8] showed that very weak function approximators such as averages can be used successfully in this setting. However, even simple linear approximators are known to result in divergence [4]. Model-based learners on the other hand, work with functions that are governed directly by the environment, therefore samples are drawn from the target function itself and divergence can never happen. This fact allows us to use much more powerful function approximators. In our case, we successfully applied a regression method called LWPR [21] on the parameters of the model—something that cannot be done reliably on the value function.

It’s worth noting that neither of the paradigms above is superior to the other one in general. Given that model-based algorithms are known for their smart exploration and good sample complexity, it makes them a very good candidate for robotics domains, where acquiring samples is usually very expensive. In the next section, we specify a model-based learner constructed from the Factored RMAX algorithm, that uses a powerful function approximator for learning the transition function.

IV. Model Parameter Approximation (MPA)

Model Parameter Approximation re-implements the factored RMAX algorithm for use in continuous spaces, replacing all the table-based representations with function approximators to compactly represent underlying functions. In order to get maximum benefit out of possible available information about the dynamics of the environment, one function approximation is used for each pair of target state variable $l$ action. More precisely, let $s = (s_1, ..., s_m)$ be the vector representation of $s$. A transition-function estimator, TFGen, is a generative (possibly stochastic) model that can answer queries of the form $(s'_1, ..., s'_m) = \text{TFGen}((s_1, ..., s_m), a_i)$ for $a_1, a_{|A|} \in A$. Our estimator has a function approximator for each pair of $(s'_i, a_j)$, denoted by $\Theta_{ij}$. A dynamic Bayes net that describes dependencies of variables on each other is used to speed up learning of the transition function. For example, to compute the value of $s'_i$ after applying action $a_k$, only the variables that are connected to $i$th component in the corresponding dynamic Bayes net will be used in $\Theta_{ik}$. Figure 8 shows such DBN for our hard BumbleBall task. This design greatly enhances
the generalization ability in cases where only a subset of variables affect each other. Function approximators are trained and queried for relative transitions instead of absolute transitions [16]. Therefore, given a transition of the form \((s, a, s')\), TFGen updates \(\Theta_{i,a}\) using the sample \((s_i', s_i, a)\), and given a query of the form \((s, a)\), computes \(s_i + \Theta_{i,a}\) as the \(i\)th component of the answer. In order to incorporate RMAX’s smart exploration of optimism in the face of uncertainty, we allow the function approximators to return a special ‘I don’t know’ symbol, \(\Xi\), for the regions where they do not have enough data support for their predictions. Bandwidth for sample-based regression or confidence bound for some other function approximators, such as the one we employ here, can be used for this purpose. RFGen is either a function approximator to map \(S\) to \(R\), or the actual underlying reward function if available.

Each \(\Theta_{i,a}\) can be designed independently to best fit that particular dimension. For solving the BumbleBall task, we’ve used a linear regression method called Locally Weighted Projection Regression (LWPR) [21] for the prediction of robot location and a simple discrete maximum-likelihood model for the ball location. LWPR combines linear regression techniques and non-parametric local methods to achieve both fast query processing time and generalization ability of parametric methods, and yet at the same time, having the ability to analyze the accuracy of the prediction. The confidence bound of LWPR is used to output \(\Xi\) if it is below a threshold. Since the BumbleBall is moving randomly in each direction, using LWPR wouldn’t be useful at all, because it converges to the expected value of the target function, which is 0 in the case of the BumbleBall. Our discrete maximum likelihood model, on the other hand, builds a list of previous movements by the ball and randomly selects one of them to answer a query, and it never outputs \(\Xi\).

A. Planning in MPA

All model-based methods have to use some sort of planning algorithm for solving their internal model. Unfortunately, none of the traditional planning algorithms for finite MDPs [17] can be applied to continuous MDPs in their original form. Recently, some new techniques have emerged for solving continuous MDPs approximately. Forward search methods such as sparse sampling [22] and UCT [23] depend only on the horizon and not the state space size. Sample-based methods such as fitted value iteration [24] follow a similar approach to value-based methods for computing the value function. Multi-resolution discretization [20] converts continuous MDPs to discrete ones before solving them. These methods are just some attempts to make the planning phase faster in model-based algorithms. MPA is independent of the type of algorithm used in the planning phase, and leaves the selection to the hands of the designer to choose the one that fits the environment best. For the purpose of this paper, we have used a discretization method for planning, denoted by DPlan. The DPlan approach, described in Algorithm 1, uses our two estimators TFGen and RFGen to generate imaginary samples, which are in turn used for building a finite MDP. The MPA algorithm itself is described in Algorithm 2.

**Algorithm 1** DPlan: a discretization-based planning approach used in MPA

1: Input: TFGen, RFGen, \(R_{\text{max}}\)
2: Output: Queries of the form \(a=\text{getBestAction}(s)\), \(s \in S\) and \(a \in A\).
3: Partition the state space into a set of non-overlapping cells \(\zeta\) with resolution \(\sigma\).
4: Create fictitious state \(FS\) with all actions having a self loop with reward \(R_{\text{max}}\).
5: For each cell \(\zeta \in \zeta\), generate \(k\) samples, \((O^1, ..., O^k)\), uniformly distributed in \(\zeta\).
6: Let \(\hat{R}(O^i) = RFGen(O^i)\)
7: For each action \(a_i\), let \(P^i = TFGen(O^i, a_i)\), and cell(s) be the cell containing \(s\).
8: if \(P^i \in \Xi\) then
9: \(P^i = FS\)
10: end if
11: Using all the \(P^i\)'s, create the maximum likelihood transition function \(\hat{T}(\zeta, a_i)\).
12: Build the finite approximate MDP \(\hat{M} = \langle \zeta, A, \hat{T}, \hat{R}, \gamma \rangle\)
13: Solve \(\hat{M}\) using conventional value iteration algorithm.
14: Upon queries, compute \(\zeta = \text{cell}(s)\) and return the best action for \(\zeta\) in \(\hat{M}\)

**Algorithm 2** Model Parameter Approximation

Variables: TFGen, RFGen, DPlan

Observe a transition of the form \((s_t, a_t, r_t, s_{t+1})\)
Add \((s_t, a_t, s_{t+1})\) as a training sample to TFGen
Add \((s_{t+1}, r_t)\) as a training sample to RFGen
if \(t \text{ mod planTime} == 0\) then
DPlan.solveModel()
end if
Execute action DPlan.getBestAction\((s_{t+1})\)

V. EXPERIMENTAL RESULTS

We now return back to the problem of solving the BumbleBall tasks. Here, we first start with the PuddleWorld environment as the standard baseline domain. We then move to the simple and hard BumbleBall problems, examining both value-based and model-based learning. In order to better understand the power of MPA and function approximation in general, we start by testing the 4 algorithms in PuddleWorld. In addition to MPA, the other algorithms were as follows.

**RMAX** [12]: The RMAX algorithm is a powerful and yet simple model-based algorithm for learning finite MDPs. The algorithm can be applied in its original form in our continuous worlds by discretizing all the sensory information before feeding it to the learning algorithm. We chose to use this algorithm in our analyses to emphasize the fact that continuous domains are not just special cases of discrete ones, and simple
adaptation of traditional algorithms don’t always work. For all the experiments here, we set the $M$ parameter of the algorithm to 2 and discretized the state space by overlaying a grid that partitions each dimension into 15 equally-sized intervals. These parameters were chosen by informal experimentation. Value iteration is called every 50 steps to solve the internal model.

**Q-learning** [18]: This algorithm is a value-based learning algorithm widely used in RL community. Q-learning, like RMAX, is a table-based method, so we used the same approach for turning the environment into a finite MDP. While 15 was the best partitioning in the case of RMAX, the coarser discretization of 6 per dimension achieved the best performance for Q-learning. A decaying epsilon-greedy exploration was used that started with 20% random action selection and linearly decayed to 1% using the formula $\epsilon = \max\{\frac{2000}{t+1000}, 0.01\}$, where $t$ is the time step.

**Fitted Q-iteration** [9]: This algorithm is the first we considered that is designed specifically to tackle continuous domains. In its original form, it’s an off-line batch algorithm, but combined with an exploration strategy, it can also be used as an online algorithm. The algorithm uses regression trees to approximate the value function over the state space by partitioning it into non-uniform regions and predicting a constant value for each region. Our implementation of fitted Q-iteration is similar, except that we used a Gaussian kernel-regression inside each region to increase the accuracy of prediction.\(^4\) The batch learner is called every 50 steps to compute the value function, given the set of collected samples. The algorithm used the same exploration approach as Q-learning.

**A. PuddleWorld**

The results for PuddleWorld are averaged over 10 independent runs with different random start seeds. Each run consists of 100 episodes, each having a cap of 100 steps. Each episode ends either if the agent reaches the goal, or if the cap is reached. Figure 4 shows the learning curve of all four algorithms. It is obvious that both MPA and fitted Q-iteration perform noticeably better than their discretized counterparts, RMAX and Q-learning. Although MPA is performing better than all the other algorithms even in this simple 2D domain, all the algorithms are able to converge to a near-optimal policy after episode 35, and on the basis of this challenging simulated domain alone, there is not much support for choosing between MPA and fitted Q-iteration.

**B. Simple BumbleBall**

The analogous experiment was then carried out in the simple BumbleBall domain. The main difference was that only 1 run was used instead of 10 and each run had 50 episodes instead of 100 as it was very expensive to collect samples using the actual robot. The reward function was given to all the learners. Another kind of learning curve is plotted in Figure 5, in which the total reward per episode is plotted for each algorithm. Although these kinds of learning curves are informative and show the overall performance of the algorithms as the learning continues, they don’t answer some key questions that are particularly interesting when working with robots. For example, it’s important to see how many actual robot steps is required for a particular learning algorithm to find the near-optimal policy or learn to avoid the puddles. This issue is important because gathering actual samples by running the robot is very expensive. In an attempt to answer some of these questions, Figure 6 shows the total number of steps the agent took during the 50 episodes of learning. There’s a huge difference between the number of steps in MPA and the other algorithms. The fact that MPA has converged to a better policy using so many fewer samples reveals the superiority of the algorithm. To make sure the absence of statistical significance doesn’t obscure the validity of the experiment, a very simple simulation of this domain was also used to verify the results in Figure 6. Although the numbers from the simulation runs are not directly comparable to the ones from the real experiments, the error bars show that MPAs advantage in the real run is not the result of luck.

Figure 7, on the other hand, shows the amount of penalty the robot incurred during the whole run. Again, we see that MPA accrued much less penalty compared to other algorithms. This result clearly shows how much generalization can help avoiding bad experiences. Although fitted Q-iteration also applies generalization, since it’s trying to approximate the complex value function with only local averaging, it doesn’t compare favorably to MPA, which effectively applies generalization to the simple transition function.

**C. Hard BumbleBall**

Only fitted Q-iteration and MPA were used in this experiment, as it was infeasible to run the other two. Again, the reward function was given to both learners. MPA had access to a set of DBNs that described the dependency of dimensions on each other for each action—Figure 8. As it can be seen in the picture, the DBNs are very easy to come up with; they basically tell the algorithm that BumbleBall doesn’t affect the Aibo and vice versa. As expected, although fitted Q-iteration

\(^4\)Using more powerful regression methods inside each tree cell has been used before and has proven to be very useful [10].
learned to avoid the BumbleBall a little, it was unable to find a near-optimal policy during the short period of 50 episodes. MPA was able to quickly learn to avoid the ball after only two episodes of smashing into it. The top lines in Figure 9 show the number of steps spent to reach the goal in each episode, while the bottom lines show the amount of penalty the robot incurred by colliding with the ball in each episode.

In order to investigate the reasons for this significant difference, the value function of both algorithms is shown at step number 500 in the Figure 10. The location of the ball and the orientation of the robot is fixed in the graphs in order to plot them in 3D form. Due to the sparsity of the data points in the huge state space, most of the regions in the KD-tree structure of fitted Q-iteration had no points in them and therefore had to predict $V_{\text{max}}$. MPA used the imaginary data points generated by its TFGen and RFGen to cover a much larger portion of the state space, acquiring a much more accurate value function.

VI. Conclusion

In this work we considered the problem of solving two robot navigation tasks, inspired by one of the benchmark toy domains in RL called PuddleWorld, using several algorithms from the two paradigms of reinforcement learning: value-based and model-based. Despite the fact that most of the research in the RL community on continuous spaces has taken place in the realm of value-function approximation, in this work we showed that model approximation can sometimes
be more useful, specially for robotics domains, where sample complexity is a crucial factor. We also introduced MPA, which is a model-based learning algorithm that applies function approximation to the parameters of the model. This algorithm successfully solves all of our robotics tasks much faster than the other methods, for two reasons: (1) it can benefit from powerful function approximators without worrying about divergence—something that cannot be used reliably in value-based methods, and (2) extra information about the dynamics of the environment can greatly improve the performance of MPA, while it’s hard to take advantage of such information in a value-function approximation setting.

ACKNOWLEDGMENT

Acknowledgment is removed for the purpose of double-blind review process.

REFERENCES