In *IEEE International Conference on Robotics and Automation* (ICRA 2012),
St. Paul, Minnesota, USA, May 2012.

RTMBA: A Real-Time Model-Based Reinforcement Learning Architecture for Robot Control

Todd Hester, Michael Quinlan, and Peter Stone
Department of Computer Science
The University of Texas at Austin
{todd,mquinlan,pstone}@cs.utexas.edu

***Abstract*— Reinforcement Learning (RL) is a paradigm for learning decision-making tasks that could enable robots to learn and adapt to their situation on-line. For an RL algorithm to be practical for robotic control tasks, it must learn in very few samples, while continually taking actions in real-time. Existing model-based RL methods learn in relatively few samples, but typically take too much time between each action for practical on-line learning. In this paper, we present a novel parallel architecture for model-based RL that runs in real-time by 1) taking advantage of sample-based approximate planning methods and 2) parallelizing the acting, model learning, and planning processes in a novel way such that the acting process is sufficiently fast for typical robot control cycles. We demonstrate that algorithms using this architecture perform nearly as well as methods using the typical sequential architecture when both are given unlimited time, and greatly out-perform these methods on tasks that require real-time actions such as controlling an autonomous vehicle.**

I. Introduction

Robots have the potential to solve many problems in society by working in dangerous places or performing jobs that no one wants. One barrier to their widespread deployment is that they are mainly limited to tasks where it is possible to hand-program behaviors for every situation they may encounter. Reinforcement learning (RL) [1] is a paradigm for learning sequential decision making processes that could enable robots to learn and adapt to their environment online. An RL agent seeks to maximize long-term rewards through experience in its environment.

RL has been applied to a few carefully chosen robotic tasks that are achievable with limited training and infrequent action selections [2], or allow for an off-line learning phase [3]. However, none of these methods allow for continual learning on the robot running in its environment. For RL to be practical on tasks requiring lifelong continual control of a robot, such as low-level control tasks, it must meet at least the following two requirements: 1) it must learn in very few samples (be *sample efficient*); and 2) it must take actions continually in real-time, even while learning.

Model-based methods such as R-MAX [4] are a class of RL algorithms that meet the first requirement by learning a model of the domain from their experiences, and then planning a policy on that model. By updating their policy using their model rather than by taking actions in the world, they limit the number of real world samples needed to learn. However, most existing model-based methods fail to meet the second requirement because they take significant periods of wall-clock time to update their model and plan between each action. These action times are acceptable when learning in simulation or planning off-line, but for on-line robot control, actions must be given at a fixed, fast frequency. Some model-based methods that do take actions at this fast frequency have been applied to robots in the past [3], [5], but they perform learning off-line during pauses where they stop controlling the robot entirely. DYNA [6], which does run in real-time, uses a simplistic model and is not very sample efficient. Model-free methods can learn in real-time, but often take thousands of potentially expensive or dangerous real-world actions to learn: they meet our second requirement, but not the first.

The main contribution of this paper is a novel RL architecture, called Real-Time Model Based Architecture (RTMBA), that is the first to exhibit both sample efficient and real-time learning. It does so by leveraging sample-based approximate planning methods, and most uniquely, by parallelizing model-based methods to run in real-time. With RTMBA, the crucial computations needed to make model-based methods sample efficient are still performed, but threaded such that actions are not delayed. We compare RTMBA with other methods in simulation when they are all given unlimited time for computation between actions. We then demonstrate that it is the only algorithm among them that successfully learns to control an autonomous vehicle, both in simulation and on the robot. RTMBA has been implemented and publicly released as a ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`.

II. Background

We adopt the standard Markov Decision Process (MDP) formalism of RL [1]. An MDP consists of a set of states S, a set of actions A, a reward function $R(s,a)$, and a transition function $P(s'|s,a)$. In each state $s \in S$, the agent takes an action $a \in A$. Upon taking this action, the agent receives a reward $R(s,a)$ and reaches a new state s', determined from the probability distribution $P(s'|s,a)$.

The value $Q^*(s,a)$ of a state-action (s,a) is an estimate of the expected long-term rewards that can be obtained from (s,a) and is determined by solving the Bellman equation:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a') \quad (1)$$

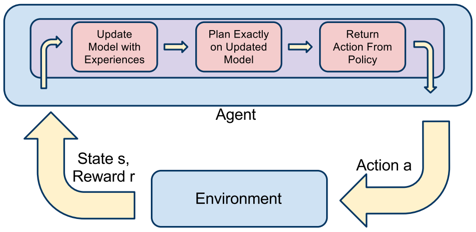


Fig. 1. A diagram of how model learning and planning are typically interleaved in a model-based agent.

where $0 < \gamma < 1$ is the discount factor. The agent's goal is to find the policy π mapping states to actions that maximizes the expected discounted total reward over the agent's lifetime. The optimal policy π is then:

$$\pi(s) = \text{argmax}_a Q^*(s, a) \tag{2}$$

Model-based RL methods learn a model of the domain by approximating $R(s, a)$ and $P(s'|s, a)$. The agent then computes a policy by planning on this model with a method such as value iteration [1]. RL algorithms can also work without a model, updating action-values only when taking them in the real task. Generally model-based methods are more sample efficient than model-free methods, as their sample efficiency is only constrained by how many samples it takes to learn a good model.

Figure 1 shows the typical model-based RL architecture. When the agent receives a new state and reward, it updates its model with the new transition $\langle s, a, s', r\rangle$; plans exactly on the updated model (i.e. with a method such as value iteration); and returns an action from its policy. Since both the model learning and planning can take significant time, this algorithm is not real-time. Alternatively, the agent may update its model and plan on batches of experiences at a time, but this requires long pauses to perform the batch updates.

The DYNA framework [6] presents an alternative to this approach. It incorporates some of the benefits of model-based methods while still running in real-time. DYNA saves its experiences, and then performs k Bellman updates on randomly selected experiences between each action. Instead of performing full value iteration between each action as above, its planning is broken up into a few updates between each action. However, it uses a simplistic model (saved experiences) and thus does not have very good sample efficiency. In the next section, we introduce a novel parallel architecture to allow more sophisticated model-based algorithms to run in real-time regardless of how long the model learning or planning may take.

III. The Architecture

We make two main modifications to the standard model-based paradigm that, together, allow it to run in real-time: 1) we limit planning time by using approximate instead of exact planning; 2) we parallelize the model learning, planning, and acting such that the computation-intensive processes are spread out over time.

First, instead of planning exactly with value iteration, RTMBA uses an anytime algorithm for approximate planning.

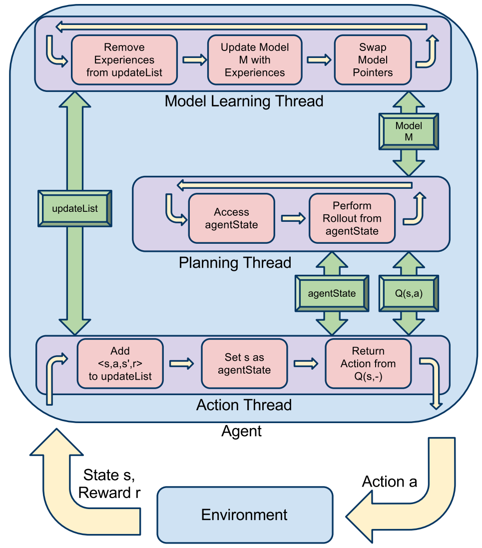


Fig. 2. A diagram of the Real-Time Model Based Architecture (RTMBA)

It follows the approach of [7] and [8] (among others) in using a sample-based planning algorithm from the Monte Carlo Tree Search (MCTS) family (such as UCT [9]) to plan *approximately*. These planners simulate trajectories (rollouts) from the agent's current state, updating the values of the sampled actions with the reward received. The agent performs as many rollouts as it can in the given time, with its value estimate improving with more rollouts. These methods can be more efficient than dynamic programming approaches in large domains because they focus their updates on states the agent is likely to visit soon rather than iterating over the entire statespace.

Second, since both the model learning and planning can take significant computation (and thus also wall-clock time), we take the straightforward but novel approach of placing both of those processes in their own parallel threads, shown in Figure 2. A third thread interacts with the environment, receiving the agent's new state and reward and returning the action given by the agent's current policy. Pseudo-code for all three threads is shown in Algorithm 1. By de-coupling this action thread from the time-consuming model-learning and planning processes, RTMBA releases the algorithm from the need to complete the model update and planning between actions. Now, it can return an action immediately when one is requested by the environment, while still selecting actions based on the most recent models and plans available. RTMBA enables the agent to take advantage of multi-core processors by running each thread on a separate core.

For the three threads to operate properly, they must share information while avoiding race conditions and data inconsistencies. The model learning thread must know which new transitions to add to its model, the planning thread must access the model being learned, the planner must know what state the agent is currently at, and the action thread must access the policy being planned. RTMBA uses mutex locks to control access to these variables, as summarized in Table I.

The action thread receives the agent's new state and

Algorithm 1 Real-Time Model-Based Architecture (RTMBA)

```
1: procedure INIT                                        ▷ Initialize variables
2:     Input: S, A
3:     Initialize s to a starting state in the MDP
4:     agentState ← s;  updateList ← ∅
5:     Initialize M to empty model
6:     UCT⇒INIT()
7: end procedure

8: procedure MODELLEARNINGTHREAD
9:     loop
10:        while updateList = ∅ do
11:            Wait for experiences to be added to list
12:        end while
13:        tmpModel ← M⇒COPY
14:        tmpModel⇒UPDATE-MODEL(updateList)
15:        updateList ← ∅
16:        M ← tmpModel
17:    end loop
18: end procedure

19: procedure PLANNINGTHREAD
20:    loop                                    ▷ Loop forever, performing rollouts
21:        UCT⇒SEARCH(M, agentState, 0)
22:    end loop
23: end procedure

24: procedure ACTIONTHREAD
25:    loop
26:        Choose a ← argmax_a Q(s, a)
27:        Take action a, Observe r, s'
28:        updateList ← updateList ∪ ⟨s, a, s', r⟩
29:        s ← s'
30:        agentState ← s
31:    end loop
32: end procedure
```

Variable	Threads	Use
$updateList$	Action, Model Learning	Store experiences to be updated into model
$agentState$	Action, Planning	Set current state to plan from
$Q(s, a)$	Action, Planning	Update policy used to select actions
M	Planning, Model Learning	Latest model to plan on

TABLE I

MUTEX PROTECTED VARIABLES, ALONG WITH THEIR PURPOSE AND WHICH THREADS USE THEM.

reward, and adds the new transition experience, $\langle s, a, s', r\rangle$, to the $updateList$ to be updated into the model. It then sets the agent's current state in $agentState$ for the planner and returns the action determined by the agent's value function, Q. When it is time to act, the action thread returns an action quickly. Although $updateList$, $agentState$, and Q are protected by mutex locks, $updateList$ is only used by the model learning thread between model updates, $agentState$ is only accessed by the planning thread between each rollout, and Q is under individual locks for each state. Thus, any given state is freely accessible most of the time. When the planner is using the state the action thread wants, it releases it immediately after updating the values for that state.

The model learning thread checks if there are any experiences in $updateList$ to be added to its model. If there are, it makes a copy of its model to $tmpModel$, updates $tmpModel$ with the new experiences, clears $updateList$, and replaces the original model with the updated copy. The other threads can continue accessing the original model while the copy is being updated, since only the swapping of the models requires locking the model mutex. After updating the model, the model learning thread repeats, checking for new experiences to add to the model.

The model learning thread can incorporate any type of model learning in the call on line 14, such as a tabular model [4], Gaussian Process regression [5], or random forests [10] (as used in this paper). Depending on how long the model update takes and how fast the agent is acting, the agent can add tens or hundreds of new experiences to its model at a time, or it can wait for long periods for a new experience. When adding many experiences at a time, full model updates are not performed between each individual action. In this case, the algorithm's sample efficiency is likely to suffer compared to that of sequential methods, but in exchange, it continues to act in real time.

The planning thread uses any MCTS planning algorithm to plan approximately (we use a variant of UCT). The thread retrieves the agent's current state ($agentState$) and its planner performs a rollout from that state. The rollout queries the latest model, M, to update the agent's value function. The thread repeats, continually performing rollouts from the agent's current state, even between action queries. With more rollouts, the algorithm's estimates of action values improve, resulting in more accurate policies.

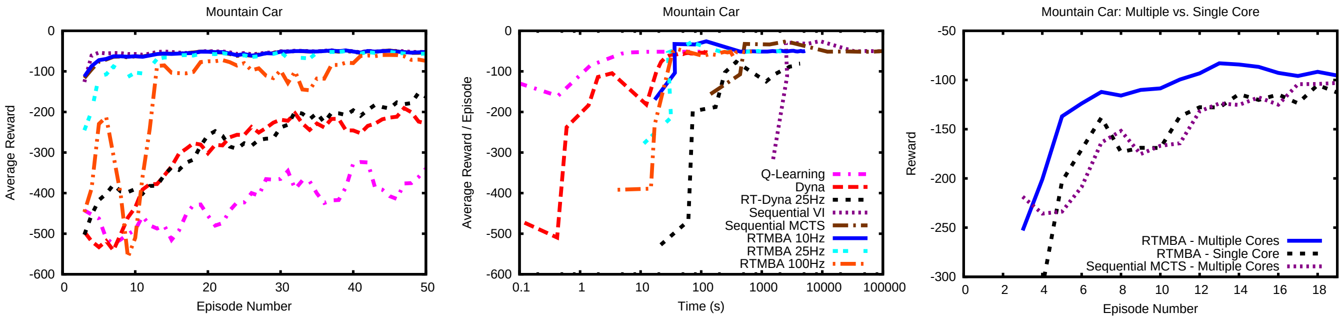
IV. EXPERIMENTS

To demonstrate the effectiveness of RTMBA, we performed experiments on two problems. Our first experiments measure the cost of parallelization in terms of environmental reward compared to a traditional sequential architecture. We use a simulated domain, which can wait as long as necessary for the agent to return an action (or it can execute actions as fast as the algorithm returns them). Our second set of experiments measures the performance gains due to parallelization on an autonomous vehicle, where real-time actions are absolutely necessary. We perform experiments, both in simulation and on the robot, that show that existing sequential approaches are not a viable option on this type of problem.

A. Mountain Car

Our first experiments were performed in the Mountain Car domain [1]. Mountain Car is a continuous task, where the agent controls an under-powered car that does not have enough power to drive directly up the hill to the goal. Instead, it must go up the opposite slope to gain momentum first. The agent has three actions, accelerating it leftward, rightward, or not at all. The agent's state is made up of two features: its POSITION and VELOCITY. The agent receives a reward of -1 each time step until it reaches the goal, when the episode terminates with a reward of 0. We discretized both state features into 100 values each, and ran the algorithms on the discretized domain. Following the evaluation methodology of Hester and Stone [10], each algorithm was initialized with one experience ($\langle s, a, s', r\rangle$ tuple) of the car reaching the goal to jump-start learning.

We ran experiments with a typical model-free RL algorithm (Q-LEARNING [11]), DYNA, two sequential model-based methods, and RTMBA. We ran two versions of DYNA:



(a) Average reward per episode. Results are averaged over a 4 episode sliding window.

(b) Average reward versus clock time. Each line starts when the first episode was *completed* (except Q-LEARNING, which completed 270 episodes after 0.1 seconds). The x-axis is in log scale.

(c) Multiple Core Experiments. Results are averaged over a 4 episode sliding window.

Fig. 3. Average rewards of the algorithms on Mountain Car, averaged over 30 trials.

DYNA performed updates on 1,000 saved experiences between each action; and RT-DYNA performed as many updates as it could while returning actions at 25 Hz. The sequential methods varied in their planning; one used value iteration for exact planning and one used MCTS for approximate planning. We modified MCTS to use UCT action selection [9], eligibility traces, and to generalize values across depths in the search tree. Between each action, the two sequential methods performed a full model update, then planned on their model by running value iteration to convergence or running MCTS for 0.1 seconds. We compared these algorithms with RTMBA using the same version of MCTS, running at three different action rates: 10 Hz, 25 Hz, and 100 Hz. All of the model-based algorithms acted greedily with respect to a random forest model of the domain, similar to the approach of Hester and Stone [10]. We ran 30 trials of each algorithm, with Q-LEARNING run for 2,000,000 episodes, DYNA for 4,000 episodes, and the remaining methods run for 1,000 episodes. Each trial was run on a single core of a machine with 2.4 - 2.66 GHz Intel Xeon processors and 4 GB of memory.

Our aim was to compare the real-time algorithms with the sequential methods when they were given the time needed to fully complete their computation between each step. Thus we can examine the performance lost by the real-time algorithms due to acting quickly. In contrast, the model-free methods could act as fast as they wanted, resulting in learning that took little wall clock time but many more samples. In these experiments, the environment waited for each algorithm to return its action, benefiting the sequential algorithms. This is only possible in simulation, whereas on a robot, the action rate is defined by the robot rather than the algorithm.

Figure 3(a) shows the average reward per episode for each algorithm over the first 50 episodes in the domain and Figure 3(b) shows the reward plotted against clock time in seconds (note the log scale on the x axis). The first plot shows that the two sequential methods perform better than RTMBA in sample efficiency, in particular, receiving significantly more reward per episode than RTMBA running at 25 and 100 Hz over the first 5 episodes ($p < 0.05$). RTMBA running at 10 Hz did not perform significantly worse than the sequential method using MCTS. However, Figure 3(b) shows that better performance of the sequential methods came at the cost of more computation time. For the sequential methods, switching from exact to approximate planning reduces the time to complete the first episode from 1541 to 142 seconds, but the MCTS method is still restricted by the need to perform complete model updates between actions. This restriction is removed with RTMBA, and all three versions using it complete the first episode within 20 seconds. In fact, all three RTMBA methods start performing well after 90 seconds, likely because they all took this much time to learn an accurate domain model. Compared with the sequential methods, RTMBA is only slightly worse in sample efficiency, while acting much faster.

The model-free approaches, Q-LEARNING and DYNA, select actions extremely quickly and converge to the optimal policy in less wall clock time than any version of RTMBA. However, Figure 3(a) shows that they are not as sample efficient. While RTMBA converges to the optimal policy within tens of episodes, DYNA takes approximately 650 episodes to converge, and Q-LEARNING takes approximately 22,000. Although RT-DYNA performs more planning updates between actions than DYNA, it is still not as sample efficient as TEXPLORE, taking approximately 300 episodes to converge. These methods learn in less wall clock time simply because they can take many more actions than RTMBA in a given amount of time. On an actual robot, it will not be possible to take actions faster than the robot's control frequency, and the poor sample efficiency of these methods will result in longer wall clock learning times as well. In comparison, RTMBA learns in fewer samples, meeting our requirement of high sample efficiency even while running at reasonable robot control rates between 10 and 100 Hz.

In addition to enabling real-time learning, another benefit of RTMBA is its ability to take advantage of multi-core processors, because each parallel thread can run on a separate core. We ran experiments comparing the performance of RTMBA when running on one versus multiple cores. These experiments were performed on a machine with four 2.6 GHz AMD Opteron processors. Figure 3(c) shows the average reward per episode for these experiments, running at 25 Hz. For comparison, we ran the sequential method using MCTS as a planner on the multi-core machine. It had unlimited time for model updates and then planned for 0.04 seconds (the same time given to RTMBA for both computations). Since the sequential architecture only has a single thread, it only used a single core even on the multi-core machine. Meanwhile, RTMBA utilized three processors with each thread running on its own core. Using the extra processors allowed the parallel version to perform more model updates and planning rollouts between actions than the single core version. Due to these advantages, the multi-core version performs better than the single core version, receiving significantly more rewards on every episode ($p < 0.005$). In addition, it performs better than the sequential method on episodes 3 to 14 ($p < 0.01$), possibly because its model is not changing every step.

These results demonstrate that the algorithms using RTMBA accomplish both requirements set forth in the introduction (sample efficiency and real-time action selection), while existing model-free and model-based methods only accomplish one of the two requirements. We have demonstrated that while using approximate planning reduces the time required by model-based methods, they do not reach real-time performance without RTMBA. Agents using RTMBA achieved similar sample efficiency to the sequential methods, while acting in real-time. Next, we look at how the algorithms compare on a task that *requires* real-time actions.

B. Autonomous Vehicle

Fig. 4. The vehicle.

Our next task was to control an autonomous vehicle. Here, actions must be taken in real-time, as the car cannot wait for an action while a car stops in front of it. This task was the main motivator for the creation of RTMBA. To the best of our knowledge, no prior RL algorithm can learn in this domain *in real time*: with no prior data-gathering phase for training a model. These experiments take place on the Austin Robot Technology autonomous vehicle [12], and on its simulation in ROS stage [13]. The vehicle is an Isuzu VehiCross (Figure 4) that has been upgraded to run autonomously by adding shift-by-wire, steering, and braking actuators to the vehicle.

The agent's goal was to learn to drive the vehicle at a desired velocity by controlling the pedals. The RL agent's state was the desired velocity of the vehicle, the current velocity, and the current position of the brake and accelerator pedals. Desired velocity was discretized into 0.5 m/s increments, current velocity into 0.1 m/s increments, and the pedal positions into tenths of maximum position. The agent's reward at each step was -10 times the error in velocity in m/s. Each episode was run at 20 Hz (the frequency that the vehicle receives new sensations) for 10 seconds. The agent had 5 actions: one did nothing, two increased or decreased the brake position by 0.1 while setting the accelerator to 0, and two increased or decreased the accelerator position by 0.1 while setting the brake position to 0.

The autonomous vehicle software uses ROS [13] as the underlying middleware. We created an RL Interface node that wraps sensor values into *states*, translates *actions* into actuator commands, and generates *reward*. This node uses a standard set of messages to communicate with the learning algorithm , similar to the messages used by RL-GLUE [14]. At each time step, it computes the current state and reward and publishes them as a message to the RL agent. The RL agent can then process this information and publish an action message, which the interface will convert into actuator commands. Whereas the RL agents using RTMBA respond with an action message immediately after receiving the state and reward message, the sequential methods may have a long delay to complete model updates and planning before sending back an action message. In this case, the vehicle continues with all the actuators in their current positions until it receives a new action message.

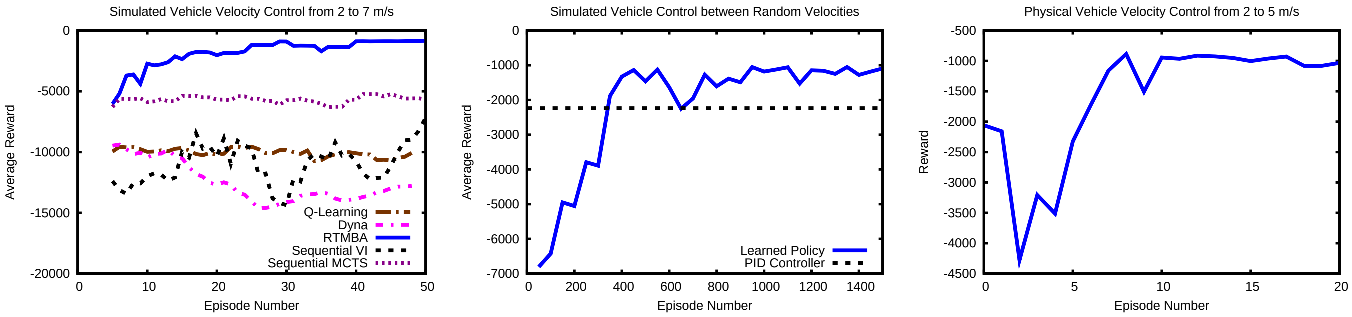
We ran the first experiment in the ROS stage simulation with the vehicle starting at 2 m/s with a target velocity of 7 m/s. Figure 5(a) shows the average rewards per episode for this task. Again the model-free methods cannot learn the task within the given number of episodes. As before, planning approximately with MCTS is better than performing exact planning, but using RTMBA is better than either. In only a few minutes, RTMBA learns to quickly accelerate to and maintain a velocity of 7 m/s.

Next, we evaluated RTMBA on the full velocity control problem, with starting and target velocities selected randomly from between 0 and 11 m/s. Figure 5(b) shows the reward accrued by the RL agent on each episode in the simulator while learning this task. For comparison, we show the reward that would be received by the PID controller that was previously used for controlling the car's velocity. The previous controller was hand-tuned for performance on the actual car. The learned controller received more reward than the PID controller after episode 350, which equates to about 1 hour of driving. It was significantly better than the PID controller ($p < 0.005$) after episode 750.

After testing in simulation, we ran 5 trials of learning in real-time on the physical vehicle, learning to drive at 5 m/s from a start of 2 m/s. Figure 5(c) shows the average rewards over 20 episodes. In all 5 trials, the agent learned the task within 11 episodes, which is less than 2 minutes of continual driving time. In 4 of the trials, the agent learned the task in only 7 episodes. This experiment shows that RTMBA enables agents to learn robot control tasks that require high sample efficiency and continual real-time action selection.

V. Related Work

Batch methods such as experience replay [15] and LSPI [16] improve the sample efficiency of model-free methods by saving experiences and re-using them in periodic batch updates. However, these methods typically run one policy for a while, stop to perform their batch update, and then repeat. While these methods take breaks to perform computation, RTMBA continues taking actions in real-time even while model and policy updates are occurring.



(a) Control from 2 to 7 m/s in simulation. Results are averaged over a 4 episode sliding window.

(b) Control between random velocities in simulation. Results are averaged over a 50 episode sliding window.

(c) Control from 2 to 5 m/s on the actual vehicle, averaged over 5 trials.

Fig. 5. Average rewards of the algorithms controlling the autonomous vehicle. Each episode consisted of 10 seconds of vehicle control.

DYNA [6] takes a similar approach to these methods, performing small batch updates between actions. DYNA-2 [7] extends DYNA to use UCT as its planning algorithm. This improves the performance of the algorithm compared to DYNA. However, to be sample-efficient, DYNA-2 must have a good model learning method, which may require large amounts of computation time between action selections.

Gaussian Process RL is a sample efficient model-based algorithm that uses Gaussian Process regression to compute the model and the policy [5]. The algorithm learns to control a physical cart-pole device with very few samples, but it runs in batch mode, pausing for 10 minutes of computation after every 2.5 seconds of action.

The Horde architecture [17] is an approach to real-time learning on robots than learns to predict or maximize the values of many different sensors in parallel, while running in real-time on a robot that is following a different policy. However, it cannot use these predictions as a model to plan better policies, and is not particularly sample efficient, as it takes 8.5 hours to learn a light-following policy.

Walsh et al. [8] combine model-based methods with sample-based planning. In order to maintain the RL methods' PAC-MDP guarantees, they create a more conservative version of UCT that guarantees ϵ-accurate policies and is nearly as fast as the original UCT. However, they still perform model and planning updates sequentially, rather than in real-time.

These methods all have drawbacks; they either have pauses in learning to perform batch updates, or require complete model update or planning steps between actions. None of these methods accomplish both goals of being sample efficient and acting continually in real-time.

VI. Conclusion

For RL to be practical for continual, on-line learning on a broad range of robotic tasks, it must both (1) be sample-efficient and (2) learn while taking actions continually in real-time. This paper introduces a novel parallel architecture for model-based RL that is the first to enable an agent to act in real-time while maintaining the sample efficiency of model-based RL. It uses sample-based approximate planning and performs model learning and planning in parallel threads, while a third thread returns actions at a rate dictated by the task. In addition, RTMBA enables RL algorithms to take advantage of the multi-core processors available on many robotic platforms. Our experiments, in simulation and on a real robot, demonstrate that RTMBA is necessary for learning on robots that require fast real-time actions. RTMBA is implemented and freely available for use as a ROS package. Our ongoing research agenda includes testing RTMBA on other robotic platforms, as well as testing other model learning and MCTS planning algorithms within the framework.

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (IIS-0917122), ONR (N00014-09-1-0658), and the FHWA (DTFH61-07-H-00030).

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.
[2] N. Kohl and P. Stone, "Machine learning for fast quadrupedal locomotion," in *AAAI*, 2004.
[3] A. Ng, H. J. Kim, M. Jordan, and S. Sastry, "Autonomous helicopter flight via reinforcement learning," in *NIPS 16*, 2003.
[4] R. Brafman and M. Tennenholtz, "R-Max - a general polynomial time algorithm for near-optimal reinforcement learning," in *IJCAI*, 2001.
[5] M. Deisenroth and C. Rasmussen, "PILCO: A model-based and data-efficient approach to policy search," in *ICML*, June 2011.
[6] R. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *ICML*, 1990.
[7] D. Silver, R. Sutton, and M. Müller, "Sample-based learning and search with permanent and transient memories," in *ICML*, 2008.
[8] T. Walsh, S. Goschin, and M. Littman, "Integrating sample-based planning and model-based reinforcement learning," in *AAAI*, 2010.
[9] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *ECML*, 2006.
[10] T. Hester and P. Stone, "Real time targeted exploration in large domains," in *ICDL*, August 2010.
[11] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, University of Cambridge, 1989.
[12] P. Beeson, *et al.*, "Multiagent interactions in urban driving," *Journal of Physical Agents*, vol. 2, no. 1, pp. 15–30, March 2008.
[13] M. Quigley, *et al.*, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.
[14] B. Tanner and A. White, "RL-Glue : Language-independent software for reinforcement-learning experiments," *JMLR*, vol. 10, Sep. 2009.
[15] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, Pittsburgh, PA, USA, 1992.
[16] M. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.
[17] R. Sutton, *et al.*, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *AAMAS*, 2011.