AAAI Technical Report SS-12-02
Designing Intelligent Robots: Reintegrating AI

TEXPLORE: Real-Time Sample-Efficient Reinforcement Learning for Robots

Todd Hester and Peter Stone
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712
{*todd,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 situations 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. In addition, the algorithm must learn efficiently in the face of noise, sensor/actuator delays and continuous state features. In this paper, we describe TEXPLORE, a model-based RL method that addresses these issues. It learns a random forest model of the domain which generalizes dynamics to unseen states. The agent targets exploration on states that are both promising for the final policy and uncertain in the model. With sample-based planning and a novel parallel architecture, TEXPLORE can select actions continually in real-time whenever necessary. We empirically evaluate TEXPLORE learning to control the velocity of an autonomous vehicle in real-time.

Introduction

Robots have the potential to solve many problems in society by working in dangerous places or performing unwanted jobs. 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) (Sutton and Barto 1998) 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.

Learning on robots poses many challenges for RL, as it requires an algorithm to learn very quickly in the face of noise, delays, and continuous state features. RL has been applied to a few carefully chosen robotic tasks that are achievable with limited training and infrequent action selections (e.g. (Kohl and Stone 2004)), or allow for an off-line learning phase (e.g. (Ng et al. 2003)). However, to the best of our knowledge, none of these methods allow for continual learning on the robot running in its environment.

In this paper, we bring together two threads of research to create an RL algorithm, TEXPLORE, that is *sample efficient*, while being able to act continually in *real-time*. These two properties not only make TEXPLORE applicable to robotic control tasks, but also many other real-world tasks. The key insights of TEXPLORE are 1) to learn multiple domain models that generalize the effects of actions across states and target exploration on uncertain and promising states; and 2) to combine Monte Carlo Tree Search and a parallel architecture to take actions continually in real-time. TEXPLORE has been released publicly as a ROS package at: `http://www.ros.org/wiki/rl-texplore-ros-pkg`.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

We adopt the standard Markov Decision Process (MDP) formalism for this work (Sutton and Barto 1998). 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 many domains, the state s has a factored representation, where it is represented by a vector of n state variables $s = \langle x_1, x_2, ..., x_n \rangle$. 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'. The new state s' is determined from the probability distribution $P(s'|s,a)$.

The value $Q^*(s,a)$ of a given state-action pair (s,a) is an estimate of the future reward that can be obtained from (s,a) and is determined by solving the Bellman equation:

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

where $0 < \gamma < 1$ is the discount factor. The goal of the agent 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 as follows:

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

Model-based RL methods learn a model of the domain by approximating $R(s,a)$ and $P(s'|s,a)$ for each state and action. The agent can then plan on this model through a method such as value iteration (Sutton and Barto 1998) or UCT (Kocsis and Szepesvári 2006), effectively updating the Bellman equations for each state using their model. RL algorithms can also work without a model, updating the values of actions 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.

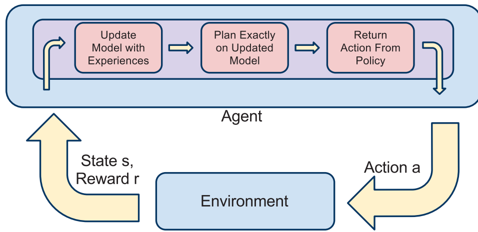

Figure 1: A diagram of how model learning and planning are typically interleaved in a model-based agent.

TEXPLORE

In this paper, we describe TEXPLORE (Hester and Stone 2010), a sample-efficient model-based real-time RL algorithm. We describe how TEXPLORE returns actions in real-time in the following section, and then describe its approach to model learning and exploration.

Real-Time Architecture

In this section, we describe TEXPLORE's real-time architecture, which can be used for a broad class of model-based RL algorithms that learn generative models. Most current model-based RL methods use a sequential architecture such as the one shown in Figure 1. In this architecture, the agent receives a new state and reward; updates its model with the new transition $\langle s, a, s', r \rangle$; plans exactly on the updated model (i.e. by computing the optimal policy 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 for the batch updates to be performed. Making the algorithm real-time requires two modifications to the standard sequential architecture: 1) utilizing sample-based approximate planning and 2) developing a novel parallel architecture called the Real-Time Model-Based Architecture (RTMBA).

First, instead of planning exactly with value iteration, RTMBA uses an anytime algorithm for approximate planning. It follows the approach of Silver et al. (2008) (among others) in using a sample-based planning algorithm from the Monte Carlo Tree Search (MCTS) family (such as UCT (Kocsis and Szepesvári 2006)) 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.

In addition to using MCTS for planning, we have developed a Real-Time Model Based Architecture (RTMBA) that parallelizes the model learning, planning, and acting such that the computation-intensive processes (model learning and planning) are spread out over time. Actions are selected as quickly as dictated by the robot control loop, while still being based on the most recent models and plans available. This architecture is general, allowing for any type of model learning method, and only requiring any method from the MCTS family for planning.

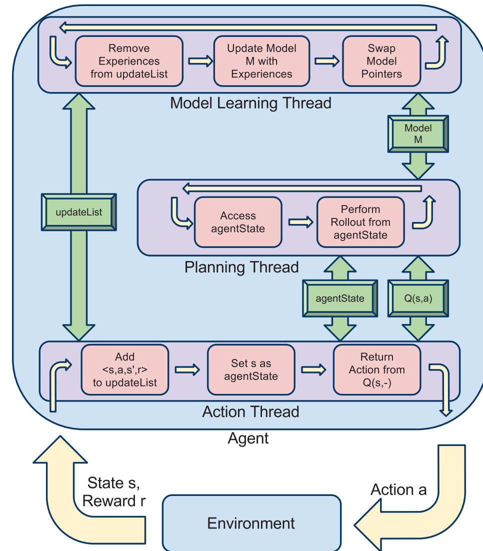

Figure 2: The Real-Time Model-Based Architecture (RTMBA).

Since both the model learning and planning can take significant computation (and thus also wall-clock time), RTMBA places both of those processes in their own parallel threads in the background, 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. 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 whenever one is required. In addition, this architecture enables the agent to take full 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 1.

The action thread receives the agent's new state and 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 roll-

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 1: This table shows all the variables that are protected under mutex locks in the proposed architecture, along with their purpose and which threads use them.

out, 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 same state the action thread wants, it releases it immediately after updating its values.

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 use any type of model, such as a tabular model, Gaussian Process regression model (Deisenroth and Rasmussen 2011), or the random forest model used by TEXPLORE. 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.

Though TEXPLORE uses a variant of UCT, the planning thread can use any MCTS planning algorithm. The thread loops, continually retrieving $agentState$ and performing planning rollouts from that state. Each rollout queries the latest model, M, to update the agent's value function. With more rollouts, the algorithm's estimates of action values improve, resulting in more accurate policies.

Model Learning

While the parallel architecture we just presented enables TEXPLORE to operate in real-time, the algorithm must learn an accurate model of the domain quickly to learn the task with high sample efficiency. While tabular models are a common approach, they require the agent to take every action from each state once (or multiple times in stochastic domains), since they learn a prediction for each state-action separately. If we assume that the transition dynamics are similar across state-action pairs, we can improve upon tabular models by incorporating *generalization* into the model learning, as has been done by past algorithms such as SPITI (Degris, Sigaud, and Wuillemin 2006) and FITTED R-MAX (Jong and Stone 2007). TEXPLORE achieves high sample efficiency by combining this generalization with aggressive exploration to improve the model as quickly as possible. TEXPLORE approaches model learning as a supervised learning problem with (s, a) as the input and s' and r as the outputs the supervised learner is predicting. One of the benefits of incorporating generalization is that the supervised learner will make predictions about the model for unseen or infrequently visited states based on the transitions it has been trained on.

In many domains, the *relative* transition effects of actions are similar across many states, making it easier to generalize actions' relative effects than their absolute ones. For example, in many gridworld domains, there is an EAST action that usually increases the agent's X variable by 1. It is easier to generalize the relative effect of this action than the absolute outcome. Relative transitions have been used to improve model learning in previous work such as RAM-R-MAX (Leffler, Littman, and Edmunds 2007) and FITTED R-MAX (Jong and Stone 2007). Instead of trying to predict s', TEXPLORE takes advantage of this idea by learning to predict the change in the state: $s^{rel} = s' - s$.

The algorithm learns a model of the domain by learning a separate prediction for each of the n state features and reward, similar to a Dynamic Bayes Network (DBN) model. Assuming that each of the state variables transition independently, these separate feature predictions can be combined to create a prediction of the complete state vector. The probability of the change in state is the product of the probabilities of the change in each of its n state features.

TEXPLORE uses decision trees to learn models of the transition and reward functions. It uses an implementation of the C4.5 algorithm (Quinlan 1986), which chooses the optimal split at each node of the tree based on information gain. Our implementation includes a modification to make the algorithm incremental. For continuous domains, the algorithm uses the M5 regression tree algorithm (Quinlan 1992), which learns a linear regression model in each leaf of the tree, enabling it to better model continuous dynamics by building a piecewise linear model.

Each tree makes predictions for the particular feature or reward it is given based on a vector containing the n features of the state s along with the action a: $\langle x_1, x_2, ..., x_n, a\rangle$. This same vector is used when querying the trees for the change in each feature and reward. The trees are built on-line while the agent is acting in the MDP. At the start, the tree will be empty, and then it will generalize broadly, making predictions about large parts of the statespace. It will continue to refine itself until it has leaves for individual states where the transition dynamics differ from the global dynamics.

We are particularly interested in applying TEXPLORE to robots, which commonly have sensor and actuator delays. For example, a robot's motors may be slow to start moving, and thus the robot may still be executing (or yet to execute) the last action given to it when the algorithm selects the next action. This is important, as the algorithm must take into account what the state of the robot will be when the action

actually gets executed. To address this problem, TEXPLORE provides its models with the past k actions as inputs in addition to the current state and action, similar to the U-TREE algorithm (McCallum 1996). The model can then learn which of these past actions is relevant, thus learning the delay dynamics in the domain.

Using decision trees to learn the model of the MDP provides us with a model that can be learned quickly with few samples. However, the model's generalization to unvisited or infrequently visited state-actions may be incorrect. Therefore, it is vital for the algorithm to have a good method for driving exploration to state-actions where the model is likely to be incorrect and needs improvement.

TEXPLORE builds multiple possible models of the domain in the form of a random forest (Breiman 2001). The random forest model is a collection of m decision trees. Each tree is trained on only a subset of the agent's *experiences* ($\langle s, a, s', r \rangle$ tuples), as it is updated with each new experience with probability w. To increase stochasticity in the models, at each split in the tree, the best feature is chosen from a random subset of the features, with each feature removed from this set with probability f. Each model in the forest can be used to drive exploration similar to the way Bayesian methods such as (Strens 2000; Asmuth et al. 2009) use samples from their distribution over possible models, but without the computational overhead of maintaining and sampling from such a distribution.

The random forest model's final prediction for a state-action is the average of the predictions of each of the trees. For example, if four trees predicted outcome A with probability 1.0, and a fifth tree predicted outcomes A and B with equal probability of 0.5, the final model will have a 0.9 probability of outcome A and a 0.1 probability of outcome B. Averaging multiple possible models of the domain inherently incorporates uncertainty in the model. If all the models agree on the outcome of a particular state-action, then it is likely to be correct. Moreover, averaging the models in this way also allows the trade-off between exploration costs and potential benefits to be handled naturally. For example, if the models disagree and the average model predicts there is a small chance of a particular high-valued outcome occurring, it may be worth exploring even if there is a low probability that it is real. On the other hand, if this outcome has a large negative value, the possibility that exploring it could be costly should make the agent avoid it. Thus, TEXPLORE will explore state-actions where both 1) some of its models predict good outcomes and 2) its models do not predict overwhelming exploration costs. Figure 3 shows a diagram of how the entire model learning system works.

Experiments

In this section, we evaluate the ability of TEXPLORE to learn velocity control on our autonomous vehicle (Beeson et al. 2008) and its simulation. This task has a continuous statespace, delayed action effects, and requires learning that is both sample efficient (to learn quickly) and computationally efficient (to learn on-line while controlling the car).

The experimental vehicle is an Isuzu VehiCross (Figure 4) that has been upgraded to run autonomously by adding shift-

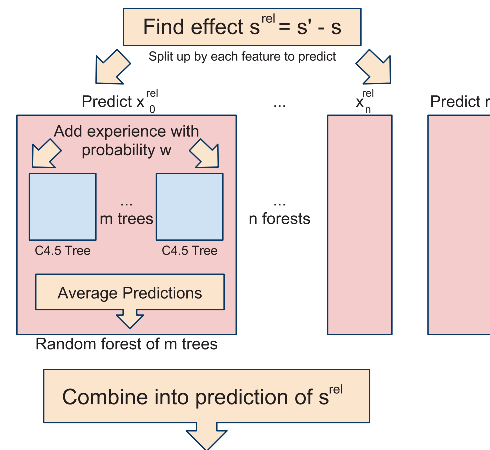

Figure 3: *Model Learning.* The agent calculates the difference between s' and s as the transition effect s^{rel}. Then it splits up the state vector and learns a random forest to predict each state feature and reward. Each random forest is made up of stochastic decision trees, which get each new experience with probability w. The random forest's predictions are made by averaging each tree's predictions, and then the predictions for each feature are combined into a complete model.

Figure 4: The autonomous vehicle operated by Austin Robot Technology and The University of Texas at Austin.

by-wire, steering, and braking actuators to the vehicle. The brake was actuated with a motor physically moving the pedal, which had a significant delay. ROS (Quigley et al. 2009) was used as the underlying middleware. Actions must be taken in real-time, as the car cannot wait for an action while a car is stopping in front of it or it approaches a turn in the road. To the best of our knowledge, no prior RL algorithm is able to learn in this domain *in real time*: with no prior data-gathering phase for training a model.

Since the autonomous vehicle was already running ROS as its middleware, we created a ROS package for interfacing with RL algorithms similar to the message system used by RL-Glue (Tanner and White 2009). 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 ROS messages to communicate with the learning algorithm. At each time step, the RL Interface node computes the current state and reward and publishes them as a ROS 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 RL agents using RTMBA respond with

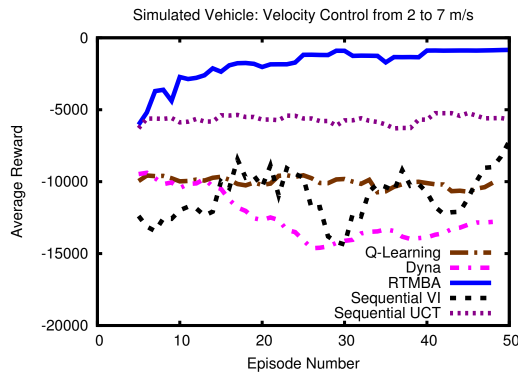

Figure 5: Average rewards of the algorithms controlling the autonomous vehicle in simulation from 2 to 7 m/s. Results are averaged over a 4 episode sliding window. Each episode consisted of 10 seconds of vehicle control.

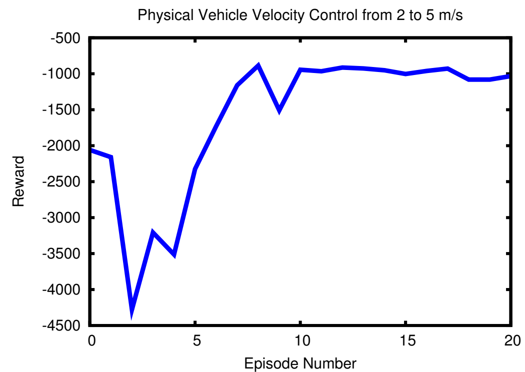

Figure 6: Average rewards of TEXPLORE learning to control the physical vehicle from 2 to 5 m/s. Results are averaged over 5 trials.

an action message immediately after receiving the state and reward message, other methods may have a long delay to complete model updates and planning before sending back an action message. In this case, the vehicle would continue with all the actuators in their current positions until it receives a new action message.

The task was to learn to drive the vehicle at a desired velocity by controlling the pedals. For learning this task, the RL agent's 4-dimensional state was the desired velocity of the vehicle, the current velocity, and the current position of the brake and accelerator pedals. For the discrete methods, desired velocity was discretized into 0.5 m/s increments, current velocity into 0.25 m/s increments, and the pedal positions into tenths of maximum position. The agent's reward at each step was -10.0 times the error in velocity in m/s. Each episode was run at 10 Hz for 10 seconds. The agent had 5 actions: one did nothing (no-op), two increased or decreased the desired brake position by 0.1 while setting the desired accelerator position to 0, and two increased or decreased the desired accelerator position by 0.1 while setting the desired brake position to 0. While these actions changed the desired positions of the pedals immediately, there was some delay before the brake and accelerator would reach their target positions.

First, we ran simulated experiments with the vehicle starting at 2 m/s with a target velocity of 7 m/s. We compared Q-LEARNING (Watkins 1989) and DYNA (Sutton 1990) with three algorithms using the TEXPLORE model. One used RTMBA, one used the sequential architecture shown in Figure 1 with value iteration planning, and another used the same architecture but planned by running UCT for 0.1 seconds. Figure 5 shows the average rewards per episode for this task. Here, the model-free methods cannot learn the task within the given number of episodes. Planning approximately with UCT is better than performing exact planning, but using RTMBA is better than either. In only 5 minutes (thirty 10-second episodes), TEXPLORE with RTMBA learns to quickly accelerate to and maintain a velocity of 7 m/s.

Finally, we ran five trials of Continuous TEXPLORE (using M5 regression trees) with $k = 2$ on the physical vehicle learning to drive at 5 m/s from a start of 2 m/s. Figure 6 shows the average rewards over 20 episodes. In all five trials, the agent learned the task within 11 episodes, which is less than 2 minutes of driving time. In 4 of the trials, the agent learned the task in only 7 episodes. This experiment shows that TEXPLORE can learn a robotic task that has continuous state and actuator delays in very few samples while selecting actions continually in real-time.

Related Work

Since TEXPLORE is addressing many challenges, there is ample related work. Deisenroth and Rasmussen (2011) use Gaussian Process regression to learn a model of the domain that generalizes experience to unknown states and represents uncertainty explicitly. This approach learns to control a physical cart-pole device in few actions, but it requires ten minutes of computation for every 2.5 seconds of experience.

Model-based Bayesian RL methods seek to solve the exploration problem by maintaining a posterior distribution over possible models. This approach is promising for solving the exploration problem because it provides a principled way to track the agent's uncertainty in different parts of the model. However, these methods have a few drawbacks. They must maintain a belief distribution over models and sample from it, both of which can be computationally expensive. In order to generalize, the user must design a model parametrization that ties the dynamics of different states together correctly. In addition, the user must provide a well-defined prior for the model.

Duff (2003) presents an "optimal probe" that solves the exploration problem optimally, using an augmented statespace that includes both the agent's state and its beliefs over its models (called a *belief state MDP*). Planning over this larger augmented statespace enables the agent to explore optimally, but can be very computationally expensive.

Other Bayesian methods use the model distribution to drive exploration without having to plan over a statespace that is augmented with model beliefs. Both Bayesian DP (Strens 2000) and Best of Sampled Set (BOSS) (Asmuth et al. 2009) approach the exploration problem by sampling from the distribution over world models and using these samples in different ways.

Bayesian DP samples a single model from the distribution, plans a policy using it, and follows that policy for a

number of steps before sampling a new model. In between sampling new models, the agent will follow a policy consistent with the sampled model, which may be more exploratory or exploitative depending on the sampled model.

BOSS, on the other hand, samples m models from the model posterior distribution and merges them into a single optimistic model with the same statespace, but an augmented action space of mA actions. Essentially, there is an action modeled by each of the predictions of the m models for each of the A actions. Planning over this model allows the agent to optimistically select at each state an action from any of the m sampled models.

Learning on a robot requires actions to be given at a specific control frequency, while maintaining sample efficiency so that learning does not take too long. Batch methods such as experience replay (Lin 1992) and LSPI (Lagoudakis and Parr 2003) 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 number of episodes, stop to perform their batch update, and then repeat.

The DYNA framework (Sutton 1990) incorporates some of the benefits of model-based methods while still running in real-time. DYNA saves its experiences, and then performs l Bellman updates on randomly selected experiences between each action. Thus, instead of performing full value iteration each time, its planning is broken up into a few updates between each action. However, it uses a simplistic model (saved experiences) and thus is not very sample efficient.

The DYNA-2 framework (Silver, Sutton, and Müller 2008) 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.

Discussion and Conclusion

This paper presents TEXPLORE, an RL algorithm which addresses the challenges necessary to be successful on robots. To achieve high sample efficiency, TEXPLORE learns random forest models that generalize transition and reward dynamics to unseen states. Unlike methods that guarantee optimality by exploring exhaustively, TEXPLORE explores quickly by targeting its exploration on states that are both promising for the final policy and uncertain in the model. TEXPLORE can take actions continually in real-time by using sample-based planning and a parallel architecture (RTMBA). These properties allow TEXPLORE to learn to control the velocity of an autonomous vehicle quickly. TEXPLORE represents an important step towards the applicability of RL to larger and more real-world tasks such as robotics problems.

Acknowledgements

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

Asmuth, J.; Li, L.; Littman, M.; Nouri, A.; and Wingate, D. 2009. A Bayesian sampling approach to exploration in reinforcement learning. In *UAI*.

Beeson, P.; O'Quin, J.; Gillan, B.; Nimmagadda, T.; Ristroph, M.; Li, D.; and Stone, P. 2008. Multiagent interactions in urban driving. *Journal of Physical Agents* 2(1):15–30.

Breiman, L. 2001. Random forests. *Machine Learning* 45(1):5–32.

Degris, T.; Sigaud, O.; and Wuillemin, P.-H. 2006. Learning the structure of factored Markov Decision Processes in reinforcement learning problems. In *ICML*, 257–264.

Deisenroth, M., and Rasmussen, C. 2011. PILCO: A model-based and data-efficient approach to policy search. In *ICML*.

Duff, M. 2003. Design for an optimal probe. In *ICML*, 131–138.

Hester, T., and Stone, P. 2010. Real time targeted exploration in large domains. In *ICDL*.

Jong, N., and Stone, P. 2007. Model-based function approximation for reinforcement learning. In *AAMAS*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *ECML*.

Kohl, N., and Stone, P. 2004. Machine learning for fast quadrupedal locomotion. In *AAAI*.

Lagoudakis, M., and Parr, R. 2003. Least-squares policy iteration. *Journal of Machine Learning Research* 4:1107–1149.

Leffler, B.; Littman, M.; and Edmunds, T. 2007. Efficient reinforcement learning with relocatable action models. In *AAAI*, 572–577.

Lin, L.-J. 1992. *Reinforcement learning for robots using neural networks*. Ph.D. Dissertation, Pittsburgh, PA, USA.

McCallum, A. 1996. Learning to use selective attention and short-term memory in sequential tasks. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*.

Ng, A.; Kim, H. J.; Jordan, M.; and Sastry, S. 2003. Autonomous helicopter flight via reinforcement learning. In *NIPS 16*.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. 2009. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

Quinlan, R. 1986. Induction of decision trees. *Machine Learning* 1:81–106.

Quinlan, R. 1992. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, 343–348. Singapore: World Scientific.

Silver, D.; Sutton, R.; and Müller, M. 2008. Sample-based learning and search with permanent and transient memories. In *ICML*.

Strens, M. 2000. A Bayesian framework for reinforcement learning. In *ICML*, 943–950.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Sutton, R. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML*.

Tanner, B., and White, A. 2009. RL-Glue : Language-independent software for reinforcement-learning experiments. *JMLR* 10.

Watkins, C. 1989. *Learning From Delayed Rewards*. Ph.D. Dissertation, University of Cambridge.