Efficient Motion-based Task Learning for a Serial Link Manipulator

Nicholas Malone[1] Aleksandra Faust[1] Brandon Rohrer[2] Ron Lumnia[3] John Wood[4] Lydia Tapia[1]

***Abstract*—Generating motions for robot arms in real-world complex tasks requires a combination of approaches to cope with the state-space complexity, task structure, environmental noise, and hardware imperfections. This article presents an efficient framework for adaptive motion task learning on physical hardware that consists of state-space dimensionality reduction with probabilistic roadmaps (PRM), task transfer from simulation to hardware, and an online reinforcement learning algorithm. Online refers to ongoing policy update and adaption to current state of the environment. The task transfer jump starts training on the hardware with knowledge learned in simulation. To achieve faster trainings speeds and improve scalability we integrate a PRM with the learning agent. For motion-based task learning, we use a reinforcement learning algorithm loosely based on human cognition. We demonstrate the framework by applying it to two pointing tasks on a 7 degree of freedom Barrett Whole Arm Manipulator (WAM) robot. The first task has a stationary target and illustrates the scalability and the ability of the framework to quickly adapt and compensate for hardware noise. The second task goes a step further and introduces a non-stationary target, demonstrating the framework's ability to adapt quickly to a new environment and new task.**

I. Introduction

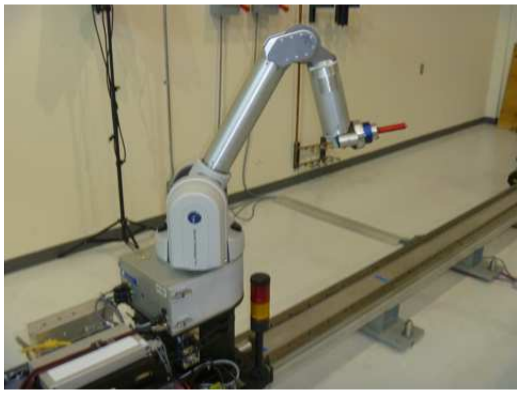

Fig. 1: Whole Arm Manipulator (WAM).

In order to perform tasks, robots must be able to adapt to changing environments and problems. In order to process real world information, online planning has to process higher volumes of data with tighter deadlines at every time step. The planning is subject to hardware imperfections and errors in reading sensory information. Online reinforcement learning (ORL), a machine learning technique, is a useful tool for robotics motion learning and planning. It provides a closed-loop feedback system continuously incorporating current environment information into the planning and producing the motions required to perform a task. However, online reinforcement learning comes with several challenges that make it potentially problematic to use on a physical system.

Implementation of an ORL algorithm must be carefully designed to be safe for the robot both in terms of collision avoidance and producing motions that don't strain hardware. Training the ORL agent from scratch on physical hardware can cause wear and tear to the hardware and thus change the dynamics of the system. Furthermore, motions take longer time to execute on hardware than in simulation, and the training phase could become impractically lengthy. Lastly, because the state space grows exponentially with the number of degrees of freedom, the sheer size of real world state spaces and physical laws of motion that need to be processed at every time step in real-time could make ORL prohibitively computationally expensive even for serial link manipulators with as little as 3 DoFs [7].

Reinforcement learning (RL) learns action (motion) sequences that maximize accumulated reward over the agent's lifetime. RL learns a policy, a mapping between robot's states and its actions with respect to some observed, and unknown to the agent, reward signal. The outcomes of the action-taking, transitions between the states, are also unknown a priori and are learned through experience. The RL problem is defined by specific state and action spaces, the ability to observe action effects on the states, and the reward associated with states. To accomplish task learning with RL, we need to engineer the reward structure that corresponds to the task, the state space that corresponds to the possible robot configurations, and the actions space that corresponds to possible robot motions.

Two major classes of RL methods are available, online and offline. Offline methods analyze and derive policy from an experience batch. Online RL, on the other hand, derives policy in an ongoing manner. It improves and changes the policy with every step. The advantage of the online RL is that it naturally adapts to the changing environment, something offline RL is not capable of doing. The adaptation comes at the price of longer convergence times to threshold performance. Being more computationally expensive, online RL is potentially prohibitive for systems with high degrees

[1]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, {nmalone, afaust, tapia}@cs.unm.edu
[2]Intelligent Systems, Robotics, and Cybernetics Group, Sandia National Laboratories, PO Box 5800, Albuquerque, NM, 87185-1010,brrohre@sandia.gov
[3]Faculty of Mechanical Engineering, University of New Mexico, Albuquerque, NM 87131, lumia@unm.edu
[4]The Manufacturing Engineering Program, University of New Mexico, Albuquerque, NM 87131, jw@unm.edu

of freedom. In this article, we take advantage of online RL, but address the slow convergence times with dimensionality reduction with PRMs for improved scalability, and with learning transfer by training first in simulation and moving goals.

Consolidating on our previous work [7], [8], this article presents framework based on ORL that successfully overcomes the challenges above and learns motion-based tasks suitable for a physical robot. To jump start the learning on hardware, and avoid a lengthy training phase, we transfer the knowledge from a task trained in simulation. To achieve performance suitable for a physical system, ensure the safety of the system, and address state space scalability, we rely on probabilistic roadmaps (PRM) for dimensionality reduction. The state space information reduced by the PRM is passed to our learning agent, which learns to produce efficient motion plans. We use a Brain-Emulating Cognition and Control Architecture (BECCA) [14] agent. It is an adaptive online reinforcement learning algorithm paired with an unsupervised hierarchical feature creator. BECCA's algorithm contains a decay feature, allowing the agent to forget features and motion plans over time. This feature is especially useful for changing environments, as the agent continuously learns and updates plans based on the current feedback from the environment.

To demonstrate the framework, we implement a pointing task on a 7 DoF WAM (Figure 1) using all 7 degrees of freedom. The robot needs to autonomously learn how to point at a target location in its environment regardless of the start position. We first formulate the task in terms of RL, and perform four sets of experiments. First, we compare the learning scalability with and without the PRM dimensionality reduction as a function of degrees of freedom. In the second series of the experiments, we assess learning transfer impact on a stationary target. We assess the performance of the framework by measuring how well the agent adapts to hardware imperfections and measurement noise. In the third series of experiments the target location moves and we assess the adaptability. Lastly, we examine the performance of the framework by looking into time savings obtained by using transfer learning.

Our results show that the system task performance does not change with increase in dimensions, and shows near-identical performance between simulation and transferred hardware runs. We show between 100 to 600 time steps of savings obtained by using transfer learning, and demonstrate an agile agent that quickly adapts to the new environment within 500 time steps.

The rest of this article is organized as follows: Section II provides necessary background and an overview of the related work. Section III discusses our methodology, and section IV presents our experimental results. Finally, section V concludes the article with the framework's benefits to online, reactive motion-based learning.

II. Preliminaries

This article presents a task learning framework for a serial link mainulator, based on BECCA [14], transfer learning [18], and PRMs [4]. This Section gives necessary background and context.

A. Barrett Whole Arm Manipulator Hardware Platform

The Barrett Whole Arm Manipulator (WAM) platform is a 7 degree of freedom (DoF) robotic arm. It is cable-driven and controlled with position encoders and torque estimation. The WAM has been connected to a GE Intelligent Platforms reflective memory network in a hub design that allows multiple computers to share memory at speeds ranging from 43 MB/s to 170 MB/s. The reflective memory networks allow remote computers to handle the planning and learning processing, while leaving a small and fast computer on-board the WAM to handle simple motion control.

There is active research on WAM training through human demonstration. A WAM system is represented as a canonical system of motor primitives [12]. The direct policy search class of reinforcement algorithms learns the parameters of the canonical system, while using the demonstration as an initial policy [12]. This line of research has produced a WAM capable of playing table-tennis [10], performing a ball-in-a-cup task [5], and flipping a pancake [6].

B. BECCA

Creating a general learning machine has been one of the grand goals of artificial intelligence (AI) since the field was born. Efforts to achieve this goal may be divided into two categories. The first category uses a depth first approach, solving problems that are complex, yet limited in scope, such as playing chess. The assumption underlying these efforts is that an effective solution to one problem may eventually be generalized to solve a broad set of problems. The second category emphasizes breadth over depth, solving large classes of simple problems. The assumption underlying these efforts is that a general solution to simple problems may be scaled up to address more complex ones. An example of the first category would be a master level chess playing agent, while an example of the second category would be an agent with the capabilities of an ant worker. The work described here falls into the second category, focusing on breadth. The motivating goal for this work is to find a solution to natural world interaction, the problem of navigating, manipulating, and interacting with arbitrary physical environments to achieve arbitrary goals. In this context, environment refers both to the physical embodiment of the agent and to its surroundings, which may include humans and other embodied agents. The agent design presented here is loosely based on the structure and function of the human brain and is referred to optimistically as a Brain-Emulating Cognition and Control Architecture (BECCA) [14], [15].

A Brain-Emulating Cognition and Control Architecture agent interacts with the world by taking in actions, making observations, and receiving reward (see Fig. 2). Formulated in this way, natural world interaction is a general reinforcement learning problem [17], and BECCA is a potential solution. Specifically, at each discrete time step, it performs three functions:

1) reads in an observation, a vector $o \in \Re^m \mid 0 \leq o_i \leq 1$.
2) receives a reward, a scalar $r \in \Re \mid -\infty \leq r \leq \infty$.
3) outputs an action, a vector $a \in \Re^n \mid 0 \leq a_i \leq 1$.

Because BECCA is intended for use in a wide variety of environments and tasks, it makes very few assumptions about the environment beforehand. Although it is a model-based learner, it must learn an appropriate model through experience. There are two key algorithms to do this: an unsupervised feature creation algorithm and a tabular model construction algorithm.

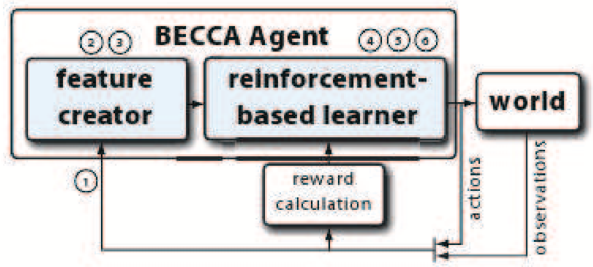

Fig. 2: At each timestep, the BECCA agent completes one iteration of the sensing-learning-planning-acting loop, consisting of six major steps: 1) Reading in observations and reward, 2) Updating feature set, 3) Expressing observations in terms of features, 4) Predicting likely outcomes based on an internal model, 5) Selecting an action based on the expected reward of action options, and 6) Updating the model.

The feature creator component identifies repeated patterns in the input vector [14]. It then groups loosely correlated elements of the input vector. The groups are treated as subspaces and unit vectors of these subspaces are features [14]. New inputs are also projected onto existing features and the single feature in each group which has the greatest response is turned on while all others in that group are turned off [13], [14], [16].

The reinforcement learning component receives feature activity, reward, and direct input from the environment. Each feature is associated with an approximate reward. It keeps track of recent actions and recent features in working memory which is then used to update the model. The actual model is a table of cause-effect pairs. The cause is the working memory and the effect is the current feature. Considering this in standard reinforcement learning language, the model can be thought of as a sequence of state-action pairs. Entries in the table which are rarely observed are deleted from the model [13], [14], [16].

To chose an action the reinforcement learner compares the current working memory to the entries in the model and selects the entry which both matches the current working memory and which has the highest recorded reward. With a set probability, an exploratory action is chosen instead [13], [14], [16].

In the context of traditional Markov Decision Process (MDP)-based reinforcement learning, the cause-effect pairs are equivalent to action-state pairs. The cause-effect table with the working memory and its expected reward roughly corresponds to a Q-function in traditional MDP-based reinforcement learning. However, BECCA's model does not assume the Markovian property and might depend on more than one previous state.

As time progresses, less frequently observed cause-effect transitions fade from the memory and the cause-effect table. This makes BECCA inherently able to adapt to new situations and environments at the cost of a steeper learning curve.

C. Probabilistic Roadmaps

PRMs [4] are a path planning technique used with robots with high DoFs to reduce the complexity searching in a high-dimensional and continuous space of possible configuration. They have been applied to a variety of complex robot types including manipulators [11], walking robots [3], nonholonomic robots [2].

PRMs tackle the planning problem by working in *configuration space* (*C-space*) rather than the workspace. *C-space* is set of all positions a robot can take. It is partitioned in set of feasible configurations (*C-free*), and configuration in collision. PRMs work by building a roadmap of possible feasible motions in *C-free* space. They do this by randomly selecting points in C-free. Then, nearby points are connected with a local planner. Only connections that fully belong to *C-free* are added to the roadmap. Nearby can be defined by low-cost Euclidean distance calculation to identify the k nearest neighbors. Connection can be achieved using straight-line interpolation. To solve the collision-free problem between start and goal configurations, a roadmap is queried. If the start and goal do not belong to the roadmap, they are added to it. Then the solution to the motion planning query is the shortest path in the roadmap with respect to some metric, often Euclidean distance, that connects the start with the goal configuration. This path is a sequence of configurations from *C-free* space that transforms start configuration to the goal configuration free of collision.

D. Transfer Learning

Transfer learning typically refers to utilizing information learned in the past on a task in the present [18]. This past learning can be transferred to a new task or to the same task under different constraints. Transfer learning has also been utilized in transferring knowledge from one robot to another robot that may have a different internal architecture to represent the world [18]. Taylor and Stone [18] define jump start and time to threshold performance as two metrics for transfer learning. Jump start defines the amount of gain an agent initially receives from transferred knowledge. Time to threshold performance defines the amount of time it takes

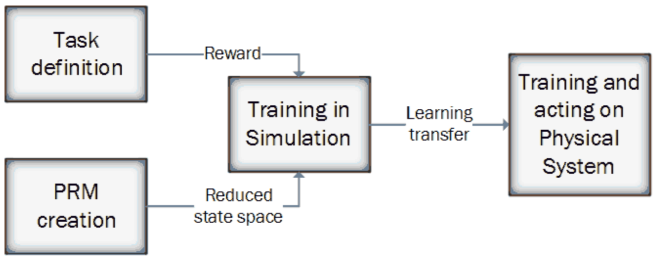

Fig. 3: Task learning framework.

an agent to reach the threshold performance, which is the best the agent can do at a given task.

III. Methods

We present a framework for online motion-based task learning. Figure 3 shows the framework's main components. Task definition describes process of constructing the reward structure, and state-action space encoding to describe the task. PRM creation segment generates a roadmap for a given environment and physical system. With the roadmap constructed and task encoded, the BECCA agent is deployed on a simulated system. Once the simulated agent's performance meets the satisfactory criteria, the entire agent is transferred to the physical system for ongoing task performing.

A. Probabilistic Roadmaps Creation

This article uses the PRMs combined with learning agent techniques to build a roadmap for the reinforcement learning agent to navigate by randomly sampling joint positions. The PRM for a task, environment, and system are created as outlined in Section II-C. The nodes in the roadmap are connected to *k* nearest neighbors using straight line local planner. The learning agent's state space is reduced to roadmap nodes, and actions are limited to edges between the nodes. The agent is constrained to making straight line movements along the edges in the adjacency matrix, thus constraining the reinforcement learner to learn how to navigate the roadmap. During a transfer, the previously learned roadmap is preserved. The PRM is the underlying state space provided to the learning agent.

B. Task definition

While BECCA is mostly automated, an engineer must design a task to interface with BECCA via sending sensory vectors and interpreting action vectors. Such an interface is called a task. A task simply defines what information from the world will be sent to the agent, and in what format. Note that BECCA is agnostic to the task semantics. The task also defines how to read an action vector and move the robotic actuators. Again, note that BECCA is agnostic to how this is defined, and it will learn whatever format the engineer devises. To demonstrate the framework we now define two pointing tasks and explain their setup.

1) Task with Stationary Target: The sensory vector is a n element binary vector, since the PRM contains n nodes. Each node represents a feasible, collision-free configuration of the robotic arm. When the robot is at a particular configuration the corresponding element in the sensory vector is set to 1.

Algorithm 1 shows how the pointing task is constructed. The action vector is a *k* element long binary vector and is parsed by the $interpret$ function. In this task, we have constrained BECCA to only return a single 1 in the action vector. The $interpret$ function in Algorithm 1 does the following: The 1 in the action vector represents BECCA selecting to move to one of the *k* neighbors, and the $(k+1)^{th}$ element is interpreted as staying at the current configuration. For example the action vector $[0, 1, 0, 0]$ is interpreted by the task as selection to move to the second neighbor of the current configuration in the roadmap. The function then returns the configuration of the selected neighbor.

The reward structure for the PRM task assigns a reward of 100 to the target node, a reward of 10 to all neighbors of the target node, and a reward of 1 to the neighbors of the neighbors. Every other node is given a reward of 0.

2) Pointing Task with Non-stationary Target: The formulation and the setup of the non-stationary target task is the same as in Section III-B1. The reinforcement learner is trained on an initial pointing task and then transferred to hardware, however upon being transferred the goal state is changed. Thus, the learning agent must compensate for the changed goal, while learning to adapt to the dynamics of the hardware system. Specifically, for this task the goal state is moved to one of the neighbors in the roadmap of the simulation goal state. The reward structure is changed so that the new goal state is reward 100 and the neighbors of the new goal 10 and the neighbors of the neighbors 0.1.

C. Transfer Learning from Simulation to Hardware

Taylor and Stone define a taxonomy of transfer learning in the reinforcement learning domain in [18]. Using that terminology, our source task is a simulated pointing task. We have two target tasks. In one, the target task has the same goal and algorithm in both simulation and hardware runs. In the other the target is moved but it still has the same algorithm. The transferred knowledge is a set of feature groups and a

Algorithm 1 Task Step

```
Require: Task
1: $Task.agent.action$ = $[0, 0, 0, 0]$
2: while $not\ coverging$ do
3:     $newLocation \leftarrow$ interpret($task.agent.action$)
4:     $sendToWAM(newLocation)$
5:     $task.currentPosition \leftarrow$ read current WAM location
6:     $task.SensoryInput \leftarrow task.currentPosition$
7:     $task.reward \leftarrow task.calculateReward()$
8:     $task.agent \leftarrow agent_{step}(SensoryInput, Reward)$;
9: end while
```

cause-effect pairs.

We transfer learned knowledge of a single task between a perfect simulation of a robot to imperfect robotic hardware. In simulation the robot always receives the exact same joint angles for a particular state, but in hardware the joint angles are subject to small error so re-entering the same state will not have the exact same state information. The source task uses the same learning agent, parameters, and reward function as the target task. The only difference is that the source task interacts with the WAM simulator while the target task interacts with the WAM hardware.

When performing the transfer, we transfer the entire agent with all its internal states and accumulated experience. We only change the world model that it interacts with from the simulator or the WAM interface.

D. *WAM Simulator*

The WAM simulator is a simple kinetic simulator, representing the arm with seven points each corresponding to one degree of freedom. The arm moves in the simulator by simply adding the state and action vectors. The simulator does not inject noise, and performs perfect movements. The WAM arm, on the other hand, performs the movements as described in the Sections II-A and III-E. The resulting motion is subject to error in performing the movement.

E. *WAM Interface*

The WAM is connected to a xPC Target Kernel running Matlab Simulink 7.7.0 R2008b [9]. The controller for the WAM is written in Simulink and interfaces with remote computers via the reflective memory network. The Simulink code responsible for directly issuing commands to the WAM, henceforth the WAM controller, receives a command vector by reading a specific block of reflective memory. The command vector is a length seven vector containing the desired joint angles in radians of each for the seven WAM joints.

The WAM controller, upon receiving a command vector, places the command vector into a buffer, which only stores one move. The command vector is first sanitized so that each entry is within the WAM's joint limits. If the WAM is not executing a move, it compares its current location to the command vector buffer. If the command vector buffer is sufficiently different from the current location, the WAM controller computes a linear interpolation in joint space between the two joint angles and executes the path within the allowable WAM workspace. However, the velocity follows a fifth-order smooth polynomial as seen in Fig 4, and is used both for safety and for mimicking biological motion [1]. Slow beginnings and endings to moves provide safe joint torques. In the current architecture a move cannot be interrupted.

IV. EXPERIMENTS

We perform four experiments. Two experiments involve the stationary target pointing task, one is a non-stationary pointing task, and the last evaluation assesses the training time benefits of the framework. First we, evaluate the utility

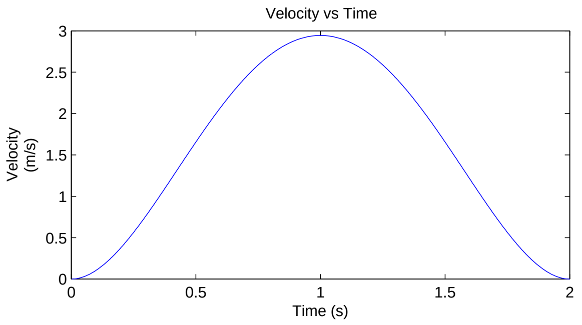

Fig. 4: Example Velocity Profile for a Single Joint.

of the dimensionality reduction using PRMs over standard state space binning. Then, we examine the benefits of transfer learning, including performance adaptability. All experimental results are averaged over five executions. Throughout the experiments, we measure the performance on the learning agent by measuring its cumulative reward. When the learning agent is transitioned from simulation to physical hardware, it is placed in a configuration that is as far as possible from the goal configuration.

We present the performance of the learning agent on hardware compared to performance in simulation. The agent executes in time steps but the graphs are shown in blocks, where 1 block equals 100 time steps. We look at the time savings brought on by using transfer learning, and the initial boost of performance that was obtained by knowledge transfer. In case of the non-stationary task, we will look at the time it takes the agent to react to a change in environment and recover to the previous level of performance

Each experimental run is executed on a new roadmap of 50 configurations generated using PRMs. Each configuration is connected to 3 neighbors and itself. A random point in the 50 configurations is chosen as the goal. The goal node is given a reward of 100.

A. *Dimensionality reduction utility*

This Section evaluates learning scalability when the state space is reduced to PRM nodes. The goal of incorporating PRMs is to help guide the searching. The state space dimensionality increases exponentially with DoFs. In [8], we showed that BECCA can learn up to 3 DoF pointing tasks with binned state space representation. Beyond that, the learning takes impractically long and the results are affected. Incorporating PRMs allows us to reduce the state space for 3-DoF tasks to the number of nodes in the roadmap. For the results shown, the state space has 50 nodes, but the number of nodes can be adjusted.

We compare the learning performance of the PRM based task, with two variants of the task with stationary point (Section III-B1). The task is reduced to 3 DoFs by limiting the WAM to use only three joints. Joints 1, 2, and 3, are mapped into a 3 dimensional C-space. Then, fifty random points are sampled in the C-space using a uniform distribution. The fifty points are then connected probabilistically based on the distance between the points, such that closer points have a higher probability of being connected. Fig. 5 shows an example of a PRM generated for the 3-DoF task.

Fig. 6 shows the cumulative reward per block for BECCA operating on the 3-DoF PRM task. The maximum reward that can be receive per iteration is 100, making the maximum per block 10,000 units of reward. The PRM covers a wide area in the WAM's range of motion, but only takes 900 iterations to reach a very high cumulative reward. 900 iterations is significantly fewer than the 5,000 iterations required for the 2-DoF task to converge [8], which indicates that PRM's are very effective at reducing the convergence time of BECCA.

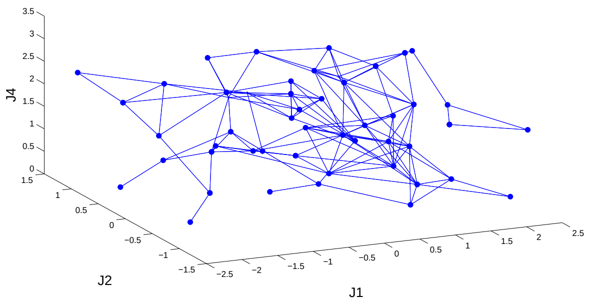

Fig. 5: Probabilistic Roadmap for a 3-DoF WAM Task. Vertices are possible configurations. Edges are possible transitions between configurations.

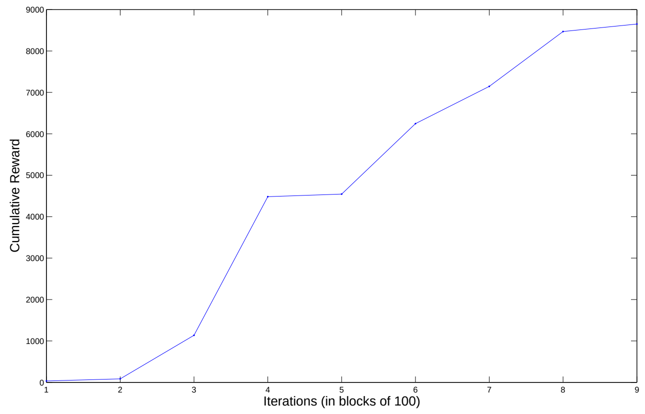

Fig. 6: PRM Cumulative Reward for a 3-DoF Task.

To compare the PRM based task to non-PRM tasks, two 3-DoF tasks are created, a simple and a hard task. The simple 3-DoF task has 3 bins per joint, and an action vector of length 12. The hard 3-DoF task has 4 bins per joint, and an action vector of length 18. The reward structure for both, the simple and hard tasks, parallel to the PRM task. It has a maximum reward of 100 per iteration and thus 10,000 per block. The simple task has 27 possible states. The hard task has 64 possible states and the PRM has 50 states. Thus, the simple and hard tasks frame the PRM in number of states. However, it is important to note that the simple and hard tasks have larger action vectors than the PRM task, 12 and 18 actions vs. 4.

Fig. 7 shows that the PRM method converges much faster than either the simple 3-DoF or hard 3-DoF task. The PRM method has reached the optimal of 7,000 units of reward by around 1,000 iterations while, the simple 3-DoF task has only reached approximately 6,000 units of reward by 7,000 iterations. The 3-DoF hard task has only reached approximately 3,500 by 7,000 iterations. Thus we can see that the PRM task converges much faster than either the simple or hard task.

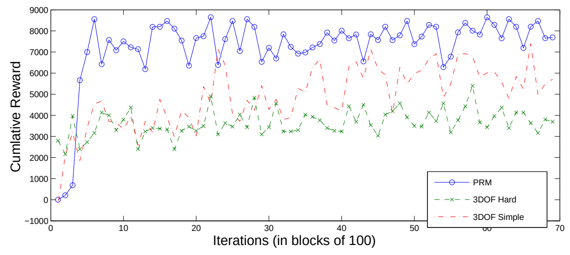

Fig. 7: Cumulative reward for PRM, 3-DoF Simple, and 3-DoF Hard tasks. The 3-DoF simple task has 3 bins per joint, giving a state space of 3^3. The 3-DoF hard task has 4 bins per and an action vector length of 18, giving a state space of 4^3. The PRM task has 50 points which correspond to 50 states.

To further show the scalability of the PRM approach we produce Fig. 8 which plots the average reward of 10 runs for each DoF from 1 to 7. This graph confirms that PRM-BECCA is unaffected by the Degrees of Freedom with a constant number of nodes. However, there is a problem with just testing the Degrees of Freedom and holding the number of nodes constant. By holding the number of nodes in the PRM constant, the density of nodes decreases as the DoF increases. Thus we must also test to see how BECCA scales with the number of nodes.

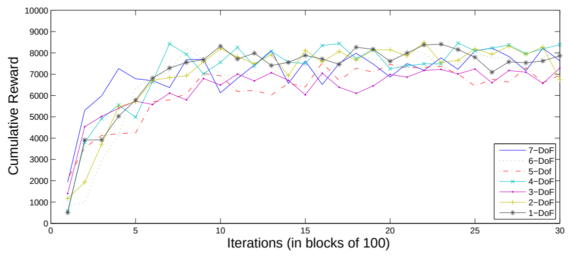

Fig. 8: Cumulative reward per block of 1-DoF to 7-DoF with PRMs.

In the following experiments we vary the number of nodes from 60 to 200 in steps of 20, and the k neighbor parameter is set to 4. Since the previous experiment showed that BECCA would converge at the same number of steps regardless of DoF we chose to do this experiment with 3 DoF. Again 10 runs are done for each number of nodes, and the results are averaged. Fig. 9 shows the average cumulative reward for each test. It shows that BECCA may converges at the same

TABLE I: Average cumulative rewards in simulation and on hardware after the stabilization for 7DoF task with a stationary target and 7DoF task with a non-stationary target

Task	Simulation	Hardware
Stationary Target	7460.3	7614.8
Nonstationary Target	7460.3	7491.5

time regardless of number of nodes in the graph. Fig. 9 is very similar to Fig. 8, thus showing that BECCA converges at the same rate regardless of DoF and regardless of the number of nodes in the PRM.

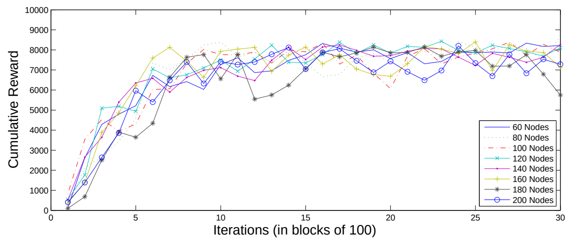

Fig. 9: Reward per Block for Varying Number of Nodes.

It is important to note that this is a novel use of PRMs. In previous work, they have been used to plan the motions for complex robot systems [2], [3], [11]. However, by integrating PRMs with BECCA, we are able to demonstrate automatic learning of controls to achieve motion in complex problems.

B. Transfer Learning on Pointing Task with Stationary Target

This Section assesses the effect of the transfer learning to the system performance. We learn first in simulation and then transfer the entire task to the physical system (Section III-C).

Figure 10 shows the cumulative reward of the pointing task with the stationary target in simulation and on hardware. The vertical line indicates the transition from the simulation to the hardware. The results show near-seamless transition, and the average performance of the agent on hardware very close to the performance in the simulation.

Table I shows the average cumulative reward for each experiment after stabilization, before and after transition to the physical hardware. Stabilization in simulation occurs at 20 blocks. The performance of the agent on the hardware outperforms the agent in simulation by 154 units of reward.

To better demonstrate the advantages of using the transfer learning in our framework, the pointing task with stationary target experiments were run again in a different manner. Five completely untrained learning agents were run on hardware for 20 blocks and the results averaged together. Then five agents which were trained for 100 blocks in a simulation were run on hardware for 20 more blocks and averaged together. Figure 11 shows the comparison of the stationary pointing task using transfer to the same task without using transfer. The advantages of using transfer are seen primarily in the jump start and the time to threshold metrics. Table II shows the transfer metrics for the three experiments. Jump start shows the immediate gain from using the transfer. The

TABLE II: Transfer Metrics for stationary and non stationary tasks. Jump start shows the gain from using transfer. Threshold gain shows the reduction in time steps needed to reach the threshold performance

Task	Metric	Average	min	max
Stationary	Jump Start (reward)	5716	2757	9280
	Threshold Gain (steps)	500	200	700
Non-stationary	Jump Start (reward)	1313	364	1702
	Threshold Gain (steps)	100	100	400

pointing task starts very close to the threshold performance using the transfer and has a jump start gain of 5716. In all random runs, the transferred learning agent outperforms the non-transferred learning agent (Table II). Furthermore, the transferred task reaches the threshold performance in 2 blocks compared to 7 blocks without transfer (Figure 11). It is important to note the time saved by using transfer learning. Table III shows the run times for simulation versus hardware for 20 blocks. It is clear that simulation is faster by up to 1 hour and 55 minutes. Using transfer learning it takes significantly less physical time on the robotic hardware for the agent to perform the given task as near optimal levels. This not only saves valuable time but it also saves valuable wear and tear on the hardware.

It is important to note that the learning algorithm is not executing pre-planned paths. It learns from experience which paths lead to highest reward and attempts to follow those paths. The paths learned in simulation provide BECCA with a strong foundation to work from, however each execution of the learning problem finds different paths due to the randomness of exploration. Thus, it is possible to witness executions of BECCA on the same underlying roadmap with slightly varying performances.

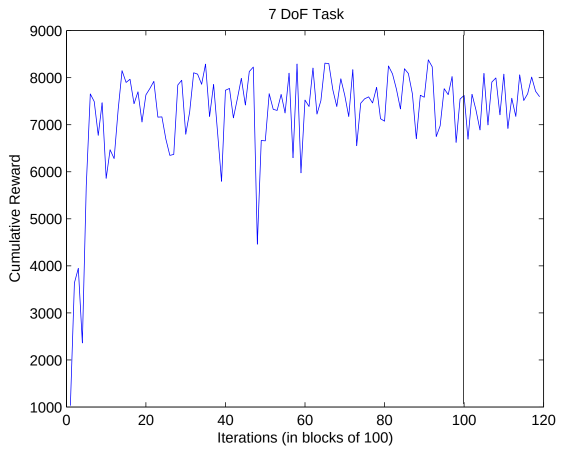

Fig. 10: Cumulative reward for the pointing task with stationary target per time step. The vertical line indicates where the learning agent was transitioned from simulation to physical hardware.

C. Adaptability Evaluation on Pointing Task with Non-stationary Target

In this experiment the reinforcement learner is trained on an initial pointing task and then transferred to hardware.

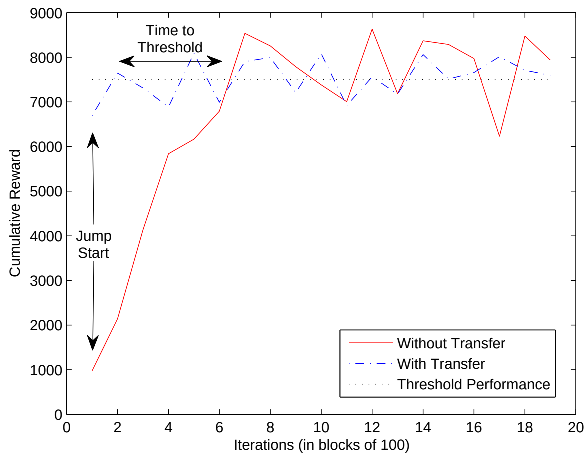

Fig. 11: Cumulative reward for the pointing task running on hardware with stationary target task with transfer and without transfer per time step. Transfer is when an agent trained in simulation is transferred to hardware. Jump start shows the initial gain obtained by using the transferred knowledge. Time to threshold indicates the time that the task without the transfer needs to achieve the same level of performance as the task with the transfer.

TABLE III: Average time in minutes to run 20 blocks in simulation and on hardware for 7DoF task with a stationary target and 7DoF task with a non-stationary target

Task	Simulation (min)	Hardware (min)
Stationary Target	23	122
Non-stationary Target	24	121

However, upon being transferred, the goal state is changed. Thus, the learning agent must compensate for the changed environment. The goal state is moved to one of the neighbors in the roadmap of the simulation goal state. The reward structure is changed so that the new goal state is reward 100 and the neighbors of the new goal 10 and the neighbors of the neighbors 0.1.

Figure 12 shows the results of 100 blocks of simulation and then 20 blocks of running on hardware where the goal has changed. Initially there is a steep performance drop, but the reward does not drop to zero. The agent quickly recovers and learns the new reward structure within 6 blocks. This shows the online nature of the BECCA algorithm. It is able to first learn one environment and when placed into a slightly different environment it is able to quickly compensate for the change.

Figure 13 is a comparison between the agent having previously learned a pointing task in a modified environment, to an agent without any prior knowledge. However, the agent with knowledge has learned to point to a different goal in simulation before being run on hardware. The untrained agent is also run on hardware but has a stationary target. Thus, the transferred agent has some information about the structure of the environment but it does not have the exact reward structure as the goal was moved before being placed on physical hardware. The figure shows that the agent with prior

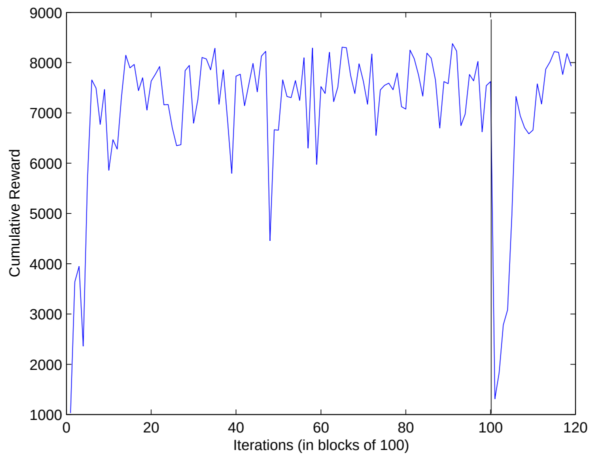

Fig. 12: Cumulative reward for running in simulation and then transferring the task to hardware. The transfer occurs at 100 blocks.

knowledge has a small jump start of 1313 units of reward and reaches the threshold performance 1 block faster than the agent without transferred knowledge.

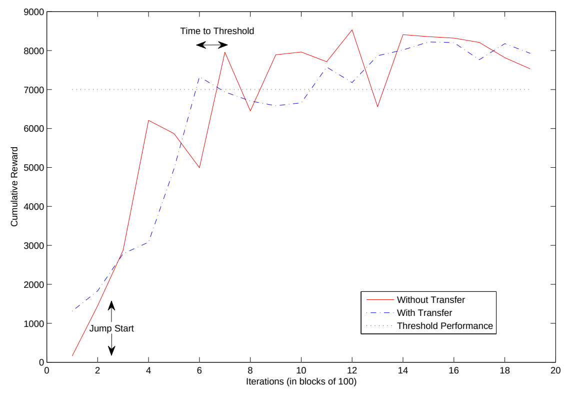

Fig. 13: Cumulative reward for the pointing task running on hardware with a non-stationary target task with transfer and without transfer per time step. Jump start shows the initial gain obtained by using the transferred knowledge. Time to threshold indicates the time that the task without the transfer needs to achieve the same level of performance as the task with the transfer.

Figure 14 shows a variant on the moving target. In this experiment the agent is trained in simulation until convergence to the threshold performance. After convergence in simulation the agent is moved to physical hardware with an unchanged goal (just like the stationary target experiments). The agent is then allowed to adapt to the hardware for 10 blocks, at which point the goal is moved while still on hardware. The agent must then adapt to this change in hardware. Figure 14 shows that the agent does very well with the initial transfer and does better when the goal is moved than Figure 12, where the agent is not allowed to adapt to the hardware before the goal is moved. The threshold performance is restored after 6 time-blocks, as previously.

But, the minimal reward of 2080 at that time-frame, is higher than the minimal reward of 1313 when the environment is changes right after the task transfer from the simulation to the hardware.

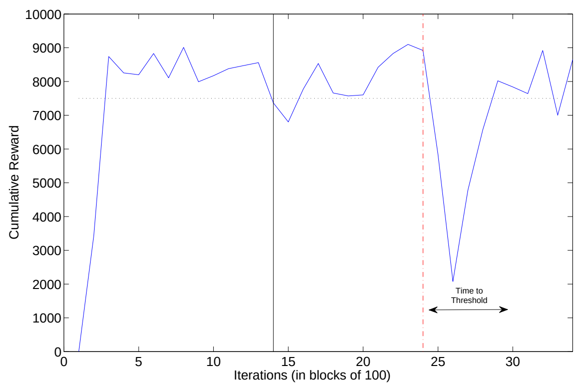

Fig. 14: Cumulative reward for the pointing task initially trained in simulation then transferred to the robot at the solid black bar. While running on the robot the goal is then changed at the red dashed line.

D. Timing

Timing data is collected by simply measuring the difference between start time and stop time for runs. Table III shows the timing data for running the learning algorithm in simulation versus on physical hardware. The run time on hardware is approximately 5 times longer due to the amount of time it takes to for the arm to move between configurations. Each move on the WAM takes approximately 3.5 seconds to compute and execute. This computation time includes the feature extraction and action decision time for the learning algorithm. In contrast, in simulation it only takes 0.5 seconds of time to execute a complete move.

Since BECCA is an online learning algorithm, it can adapt to changes in real time. However, because it is an unsupervised learning agent it still requires repeated examples of the new environment. The real time metric that we are interested in is the amount of time it takes to converge to the threshold performance. This time is important because it represents the amount of time in which the robot is learning instead of performing the desired task. This metric is recorded by simply measuring the difference between the start time of run and the time of each step. Table IV shows the average time for reaching the threshold performance with and without transfer learning. This table shows that transfer learning reduces the learning time by 29 minutes for a stationary target and 8 minutes for a non-stationary target.

For the experiment with both stationary and non-stationary targets, Table IV shows the convergence time after the target is changed for simulations training and without simulation training. This experiment first transfers the simulation to the robot with a station target, and then after stabilizing the target is moved (like the non-stationary test). The version without simulation runs the whole experiment on the physical

TABLE IV: Average Time for Convergence to Threshold Performance.

Task	w/o Transfer (min)	w/ Transfer (min)
Stationary Target	40.8	11.7
Non-stationary Target	41.1	33.0

Task	w/o Transfer (min)	w/ Transfer (min)
Both Targets: recovery	25.9	24.7
Both Targets: total time	129.5	82.1

robot. The times for the recovery are very similar, which is to be expected as they are just showing the recovery from the changed target experiment. At this point both the transfer run and the no-transfer run have the same knowledge and thus exhibit the same amount of recovery time. This demonstrates the online nature of the algorithm. However, the total run-time is very different as the transfer agent does 1400 iterations in simulation (11.44 minutes vs. 58.30 minutes). This is a difference of 47.37 minutes.

V. Discussion

We demonstrated an efficient online motion-based task learning framework based on reinforcement learning that works in high-dimensional spaces in real-time, is reactive to changes in the environment, performs safe hardware motions, and efficiently learns on hardware. We demonstrated the framework by implementing it on a 7 DoF WAM using all joints to produce pointing motions with both stationary and non-stationary targets. The framework is robust and extensible to other robotics systems as well as with different model formulations, and for a large variety of tasks as well.

Dimensionality reduction and collision checks can be handled through PRMs for any motion-based task. When PRMs are used in this manner, they impose hard limits on the system. For example, self-collision states tend to be invariant to the type of environment or the task, and are good candidates to be precomputed ahead of time. When there is error in the model used for simulation caused by noisy sensor data, the robot can explore the validity of the simulation's roadmap and learn how to efficiently navigate in the physical environment.

Transfer learning can be used to avoid early learning phases when the agent's performance tends to be erratic, to reduce wear and tear on robot, and to speed up the learning process on the physical robot. It can be a powerful techniques to mitigate the long convergence times of reinforcement learning. Combining transfer learning, reinforcement learning and probabilistic roadmap methods produces a powerful framework for solving complex robotic tasks. By harnessing each method's strengths, the weaknesses of the other methods can be mitigated.

An online reinforcement learning algorithm is a suitable candidate for a planner when paired with the above techniques. Such a reinforcement learner continuously learns and updates its policy by incorporating most recent experience from the environment and produces motion plans that are adaptive, real-time, and reactive.

References

[1] T. Flash and N. Hogans. The coordination of arm movements: An experimentally confirmed mathematical model. *J. of neuroscience*, 5(7):1688–1703, 1985.

[2] S. Hashim and T.-F. Lu. A new strategy in dynamic time-dependent motion planing for nonholonomic mobile robots. In *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1692–1697, Dec 2009.

[3] K. Hauser, T. Bretl, J.-C. Latombe, and B. Wilcox. Motion planning for a six-legged lunar robot. In S. Akella, N. Amato, W. Huang, and B. Mishra, editors, *Algorithmic Foundation of Robotics VII*, volume 47 of *Springer Tracts in Advanced Robotics*, pages 301–316. Springer Berlin Heidelberg, 2008.

[4] L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Automat.*, 12(4):566–580, August 1996.

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

[6] P. Kormushev, S. Calinon, and D. G. Caldwell. Robot motor skill coordination with em-based reinforcement learning. In *Proc. IEEE/RSJ Intl Conf. on Intelligent Robots and Systems (IROS)*, pages 3232–3237, Taipei, Taiwan, October 2010.

[7] N. Malone, A. Faust, B. Rohrer, J. Wood, and L. Tapia. Efficient motion-based task learning. In *Robot Motion Planning Workshop, IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012.

[8] N. Malone, B. Rohrer, L. Tapia, R. Lumia, and J. Wood. Implementation of an embodied general reinforcement learner on a serial link manipulator. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 862–869, May 2012.

[9] MathWorks. MATLAB version 7.7.0 r2008b, 2008.

[10] K. Mülling, J. Kober, and J. Peters. A biomimetic approach to robot table tennis. *Adaptive Behavior*, 19(5):359–376, 2011.

[11] J.-J. Park, J.-H. Kim, and J.-B. Song. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems*, 5(6):674–680, 2007.

[12] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682 – 697, 2008.

[13] B. Rohrer. Biologically inspired feature creation for multi-sensory perception. In *Proc. Int. Conf. on Biologically Inspired Cognitive Architectures*, volume 233 of *Frontiers in Artificial Intelligence and Applications*, pages 305–313. IOS Press, Nov 2011.

[14] B. Rohrer. A developmental agent for learning features, environment models, and general robotics tasks. In *IEEE Conference on Development and Learning and Epigenetic Robotics*, Aug 2011.

[15] B. Rohrer. An implemented architecture for feature creation and general reinforcement learning. In *Workshop on Self-Programming in AGI Syst., Int. Conf. on Artificial General Intelligence*, Aug 2011.

[16] B. Rohrer. BECCA: Reintegrating AI for natural world interaction. In *AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI 2012*, Mar 2012.

[17] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press, MIT, 1998.

[18] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, Dec 2009.