Effective Reinforcement Learning for Mobile Robots

William D. Smart and Leslie Pack Kaelbling

***Abstract*— Programming mobile robots can be a long, time-consuming process. Specifying the low-level mapping from sensors to actuators is prone to programmer misconceptions, and debugging such a mapping can be tedious. The idea of having a robot *learn* how to accomplish a task, rather than being told explicitly is an appealing one. It seems easier and much more intuitive for the programmer to specify *what* the robot should be doing, and to let it learn the fine details of *how* to do it. In this paper, we introduce a framework for reinforcement learning on mobile robots and describe our experiments using it to learn simple tasks.**

***Index Terms*— Mobile robots, machine learning, reinforcement learning, learning from demonstration.**

I. Introduction

Programming mobile robots can be a very time-consuming process. It often takes many iterations to fine-tune the low-level mapping from sensors to actuators. It is often difficult for a programmer to translate knowledge about how to complete a task into terms that are useful for the robot. Robot sensors and actuators are very different from those of humans, and misconceptions about how they operate can cause control code to fail. The idea of providing some high-level specification of the task and using machine learning techniques to "fill in the details" is an appealing one. It seems like it would take less time to write this high-level specification and the learned control policy, if it uses empirical observations of the world, should less prone to programmer bias than hand-written control code would be.

In this paper we briefly survey reinforcement learning, a machine learning paradigm that is especially well-suited to learning control policies for mobile robots. We discuss some of its shortcomings, and introduce a framework for effectively using reinforcement learning on mobile robots. We then go on to give experimental results of applying this framework to two mobile robot control tasks.

II. Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm that is particularly well-suited for use on mobile robots. It assumes that the world can be described by a set of states, S, and that the agent (the robot in this case) can take one of a fixed number of actions, A. Time is divided into discrete steps. At each time step, the agent observes the state of the world, s_t, (which might include the internal state of the robot) and chooses an action, a_t to take. After taking the action, the agent is given a reward, $r_{t+1} \in \mathbb{R}$, reflecting how good, in a very short-term sense, that action was, and observes the new state of the world, s_{t+1}. The goal of RL is to take these experience tuples, $(s_t, a_t, r_{t+1}, s_{t+1})$, and learn a mapping from states (or states and actions, depending on the particular algorithm used) to a measure of the long-term value of being in that state, known as the *optimal value function.*

The particular reinforcement learning algorithm that we use in this work is Q-learning [1]. The Q-learning optimal value function is defined as

$$Q^*(s,a) = E\left[R(s,a) + \gamma \max_{a'} Q^*(s',a')\right].$$

This represents the expected value of the reward for taking action a from state s, ending up in state s', and then acting optimally from then on. The parameter γ is known as the discount factor, and is a measure of how much attention we pay to possible rewards that we might get in the future. Once we have the optimal Q-function, $Q^*(s,a)$, it is easy to calculate the optimal policy, $\pi^*(s)$, by simply looking at all possible actions from a given state and selecting the one with the largest value,

$$\pi^*(s) = \arg\max_a Q(s,a).$$

The Q-function is typically stored in a table, indexed by state and action. Starting with arbitrary values, we can iteratively approximate the optimal Q-function based on our observations of the world. Every time that the robot takes an action, an experience tuple, $(s_t, a_t, r_{t+1}, s_{t+1})$, is generated. The table entry for state s and action a is then updated according to

$$\begin{aligned} Q(s_t,a_t) \quad \leftarrow \quad & (1-\alpha)Q(s_t,a_t) + \\ & \alpha\left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a')\right). \end{aligned}$$

Under some reasonable conditions [1] this is guaranteed to converge to the optimal Q-function, $Q^*(s,a)$. One of the most important features of Q-learning is that it is what is known as an *off-policy* algorithm. This means that the distribution from which the training samples are drawn has no effect, in the limit, on the policy that is learned. This will prove to be very important for our algorithm, as discussed below.

This description of Q-learning has been necessarily brief. For much more comprehensive coverage, see the book by Sutton and Barto [2] or the survey by Kaelbling, Littman and Moore [3].

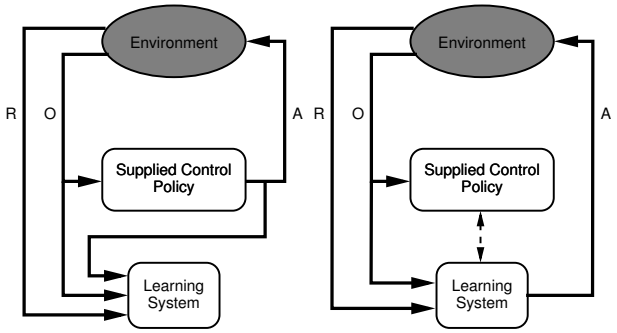
WDS is with the Department of Computer Science, Washington University in St. Louis, St. Louis, MO 63130, wds@cs.wustl.edu. LPK is with the Artificial Intelligence Laboratory, MIT, 200 Technology Square, Cambridge, MA 02139, lpk@ai.mit.edu. This work was supported under DARPA contract #DABT63-99-1-0012.

III. Reinforcement Learning on Robots

Reinforcement learning, and Q-learning in particular, seems to be a natural choice for learning control policies on mobile robots. Instead of designing a low-level control policy, we can design a much higher-level task description in the form of the reward function, $R(s, a)$. Designing a sparse reward function is generally easier than designing the low-level mapping from observations to actions. Often, for robot tasks, rewards correspond to physical events in the world. This makes is easy to come up with simple reward functions for many tasks. For example, for an obstacle avoidance task, the robot might get a reward of 1 for reaching the goal, and -1 for hitting an obstacle. In theory, this is all that is necessary for the robot to learn the optimal policy. However, there are a number of problems with simply using standard Q-learning techniques. We will call this type of reward function *sparse*. Sparse reward functions are zero everywhere, except for a few places (the obstacles and goal in the above example). The contrast with *dense* reward functions, which give non-zero rewards most of the time. A dense reward function for obstacle avoidance might be, for example, the sum of distances to the obstacles divided by the distance to the goal state. Dense reward functions give more information after each action, but are much more difficult to construct than sparse functions. In this work, we will mostly be interested in sparse reward functions, since they are generally much simpler to design.

One problem with RL on mobile robots is that the state space description of mobile robots is often best expressed in terms of vectors of real values, Q-learning requires discrete states and actions. One approach to overcome this problem, known as value-function approximation, replaces the tabular representation of $Q(s, a)$ with a general-purpose function approximator. However, it has been shown that this will not work for general function approximators, even in seemingly benign situations [4]. In previous work [5], [6], we presented an algorithm, HEDGER, that addresses the problems associated with value-function approximation. The algorithm is based on the observation by Gordon [7] that a function approximator can safely be used to replace the tabular value function representation if it never extrapolates from its training data. He showed that locally weighted averaging (LWA) is such a function approximator. HEDGER uses a more powerful function approximator, locally weighted regression (LWR) [8], supplemented with extrapolation checks, for value-function approximation. For every query, it checks to make sure that the query point is within the training data that it has already seen, and that the predicted value is within reasonable limits (given the rewards observed). If the query is outside of the training data, or the prediction is not reasonable, it returns a prediction using LWA (which is guaranteed safe). Otherwise, the prediction using LWR is returned. This algorithm has been shown to learn faster than one based only on LWA, and to be robust across a number of test domains [6].

In all of the work presented here, we use HEDGER as part of our Q-learning implementation. Previous work has generally solved this problem either by using domain knowledge to create a good discretization of the state space [9] or by hierarchically decomposing the problem by hand to make the learning task easier [10].



(a) Phase 1 (b) Phase 2

Fig. 1. The two learning phases.

The other main problem is that of incorporating prior knowledge into the learning system. The only way that Q-learning can find out information about its environment is to take actions and observe their effects. In the early stages of learning, the system knows nothing about the environment, or how to act in it. It is, therefore, forced to choose more-or-less arbitrary actions. As more information, in the form of rewards, $R(s, a)$, arrives, Q-learning can iteratively improve its approximation of the value function. However, if no rewards are observed, the value function approximation will never change. This problem is compounded by sparse reward functions. If there are only a few rewards, and the state-action space is large, the chances of finding a reward by chance are very small indeed. In section IV, we introduce a framework for RL on robots that addresses this problem.

IV. The Learning Framework

One of the major hurdles to implementing RL systems on real robots is the inclusion of prior knowledge in the learning system. Using a sparse reward function, and without some prior knowledge of the environment, the learning system is almost certainly doomed to fail. Our solution to this problem is to supply example trajectories to the learning system and split learning into two phases.

In the first phase of learning, shown in figure 1(a), the robot is being controlled by a supplied control policy. This can either be actual control code, or a human directly controlling the robot with a joystick. During this learning phase, the RL system is passively watching the states, actions and rewards that the supplied policy is generating. It uses these rewards to bootstrap information into its value-function approximation.

The key element of this phase is that the RL system is *not* in control of the robot. At this point, we assume that the value-function approximation is not complete enough to adequately control the robot. The rôle of the supplied control policy is to expose the RL system to the "interesting" parts of the state space, those parts where the reward is non-zero. It is important to note that we are not trying to learn the trajectories generated by the supplied policy, but simply using the to generate experiences with which to bootstrap the value-function approximation.

Once the value-function approximation is complete enough to control the robot effectively, the second learning phase starts. In this phase, shown in figure 1(b), the learned policy is in control of the robot, as it would be in a standard RL implementation. If the supplied control policy is a piece of software (as opposed to direct human control), it can be kept running in the background to offer advice on control decisions if needed.

By splitting the learning into two phases, we gain the ability to bootstrap information into the value-function approximation before committing to using the learned policy to control the robot. This allows us make sure that, once we move to the second learning phase, the robot will be capable of finding reward-giving states, and that learning will not stall. By observing the example trajectories, we remove the need to know about the robot sensor and actuator systems in detail. If phase one learning is done by direct control of the robot, the human controller does not need to know anything about sensor systems, inverse kinematics or reinforcement learning. Lin [11] used a similar method to bootstrap information into the value function, but relied on hand-coded sequences of experiences. This requires detailed knowledge about how the robot moves and how the sensors work. If any of this knowledge is incorrect, or based on faulty assumptions, the learning system will fail to produce the desired results. By observing actual trajectories through the state-action space, we are learning from empirical data, and are not subject to our own biases.

This sort of learning by demonstration has become quite popular recently [12], with the robots generally trying to learn the inverse kinematics of a task by observation [13]. The closest work to that reported here is by Lin [11], who used example trajectories to accelerate learning. However, these trajectories were assembled by hand and required a detailed knowledge of the robot dynamics and sensor systems. Our approach differs in the fact that we use the robot itself, under human guidance, to create the example trajectories from which the RL system learns.

V. Experimental Results

In this section, we present the results of using our framework to learn control policies for two simple robot tasks, corridor following and obstacle avoidance. The experiments were carried out on a Real World Interface B21r mobile robot. The robot has a synchronous-drive locomotion system and can rotate about its axis, and translate forwards and backwards. In all of the experiments reported here, the translation speed was either controlled by a fixed policy (for corridor following) or constant (for obstacle avoidance). Our goal is to learn a good policy for setting the rotation velocity of the robot.

The robot uses a scanning laser range-finder to identify walls and obstacles in the world. Higher-level features, described below, are computed from the raw laser information and used as inputs to the control system.

Every five training runs, learning was temporarily disabled, and the performance of the learned policy was evaluated. These evaluation runs were started from ten pre-specified starting poses. Our evaluation metric is number of steps taken to reach the goal state. The performance graphs below show the average number of steps to the goal, along with the 95% confidence bound on this average. Training runs were also started from a variety of poses similar to, but not the same as, those used for evaluation.

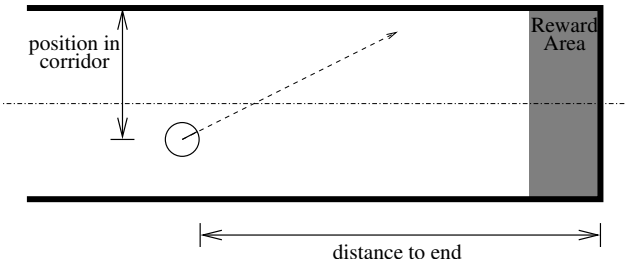


Fig. 2. The corridor following task.

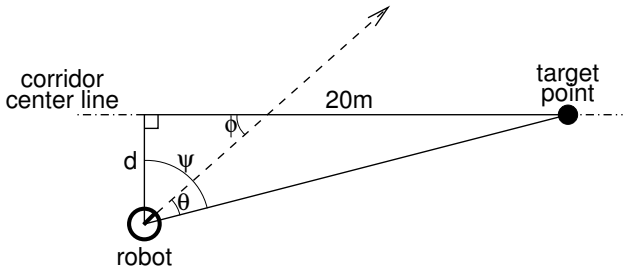


Fig. 3. Calculating the angle for the corridor task.

In all of the experiments presented here, the RL learning rate, α, is set to 0.2. In the corridor following experiments, the discount factor, γ, is set to 0.99, and in the obstacle avoidance experiments it is set to 0.9.

A. Corridor Following

The corridor following task is shown in figure 2. The translation speed, v_t, of the robot is controlled by a fixed policy that causes it to move quickly when it is in the middle of the corridor and more slowly as it gets closer to a wall. The state space of this problem has three dimensions, the distance to the end of the corridor, the distance from the left-hand wall as a fraction of the total corridor width and the angle to a target point, shown in figure 3. The angle to the target point, θ is

$$\theta = \tan^{-1}\frac{20}{d} + \phi - \frac{\pi}{2}.$$

We found that using the angle to a moving target point like this resulted in much smoother behavior than using the angle that the robot heading made with the corridor. The robot gets a reward of 10 for reaching the end of the corridor, and a reward of zero in all other situations.

Figure 4 shows the performance of the framework using a simple control policy for phase one learning. The supplied policy was a simple proportional controller

$$v_r = \alpha\theta.$$

The value of the gain, α, was varied from run to run in order to generate a wider range of experiences for the learning system.

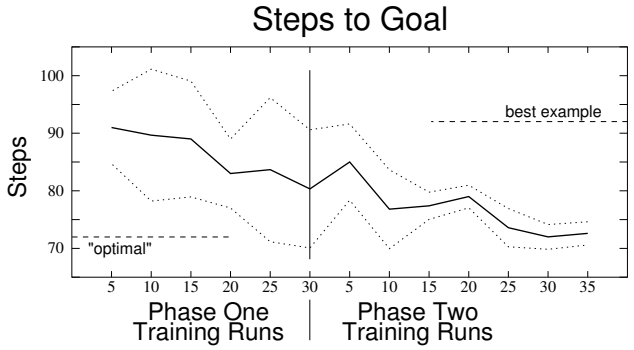


Fig. 4. Corridor following performance with simple policy examples.

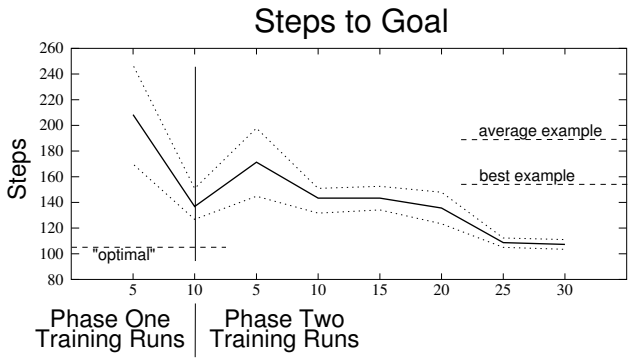


Fig. 5. Corridor following performance with direct control examples.

As can be seen from the figure, the performance of the learned policy improves with more experience. After 30 phase one training runs, the performance is significantly better (at the 95% level) than that of the shortest phase one training run (labelled "best example" in the figure).

After 30 phase one training runs, the learned policy was deemed to be competent enough to take over control of the robot. The first evaluation, after five phase two training runs, is worse than the last phase one evaluation. After this, however, the performance of the learned policy continues to improve. We have noticed this behavior in many domains, just after the change in learning phases. During phase one learning, the experiences generated by the example trajectories are likely to be fairly predictable. Thus, the number of novel experiences per run that he learning system must cope with is relatively small. However, after the phase transition, when the learned policy is in control, a greater number of novel experiences seem to be generated. This causes a temporary decrease in performance until the learning system has time to integrate the new data properly.

After 35 phase two training runs, the performance of the learned policy is indistinguishable (at the 95% level) from the best that one of the authors could achieve by directly controlling the robot with a joystick (labelled "optimal" in the figure). The confidence bounds on the average performance also narrow with more training. This corresponds to the performance of the robot becoming more predictable across different starting poses.

Figure 5 shows the performance of the framework on a another corridor following task. This time, however, phase one training was done by directly controlling the robot with a joystick, instead of using an hand-coded example policy. The corridor used in this experiment is longer than in the previous one, so the best possible performance will be different. Again, the graph shows the average and shortest training runs used during phase one, and the best one of the authors could do. The phase one training runs are all longer than this "optimal" value because no attempt was made to make them "good" examples. In fact, the robot was driven quite sloppily during phase one and, although it eventually reached the goal the path was often far from the shortest possible.

The basic profile of the performance is similar to that shown in figure 4, with two notable exceptions. There are fewer phase one training runs, and the performance improvement in phase one is much more rapid than in phase two. These are caused by the nature of the training data generated. The experiences generated by a human driving the robot are much more varied than those generated by an example control policy. This variety of training data means that our framework is able to generalize more effectively, which tends to lead to better performance in new situations.

As with the previous set of experiments, the slight performance decrease just after the learning phase change is evident. The final performance is, again, indistinguishable from the best that one of the authors could do by directly controlling the robot.

To see how effective phase one training is, in terms of time spent, we also ran some simulations of the corridor following task. Since we are dealing with a sparse reward function, learning time is dominated by the time taken to reach a reward-giving state for the first time. We simulated a robot in a corridor 10m long, approximately the length of the corridor in the first set of experiments described above, and timed how long it took to get to the reward states at the end of the corridor. The simulated robot had the same translation speed policy as in the real experiments, and used the idealized forward model for synchronous-drive systems. For each set of experiments, we started the robot in the middle of the corridor, pointing in a random direction in the half-circle towards the goal. In each set of experiments, we limited the magnitude of the rotation velocity, v_r (in the real robot experiments, $-1 \leq v_r \leq 1$). Actions were chosen from a uniform distribution over the appropriate range. Figure 6 shows the results from these simulations. The fastest that the simulation was able to reach the goal was slightly over two hours. Although it can be argued that a more informed action selection policy could do better, we are assuming that a standard RL approach will not have such domain-specific knowledge. In this case we are forced to take more-or-less arbitrary actions. In both of the real robot experiments reported above, all of the training was accomplished in approximately two hours. Since figure 6 shows the time until the *first* reward is found, it is clear that using phase one learning offers a huge time saving for this task.

B. Obstacle Avoidance

The obstacle avoidance task that we used is shown in figure 7. The translation speed, v_t, of the robot is fixed, and

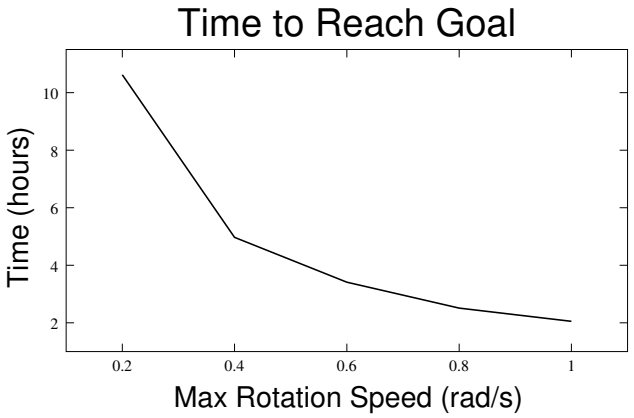


Fig. 6. Performance on the simulated corridor following task.

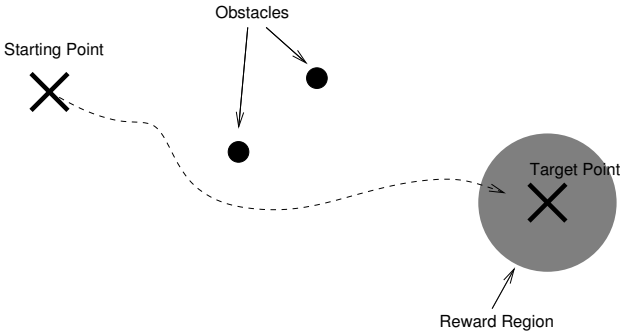


Fig. 7. The obstacle avoidance task.

we are trying to learn a policy for the rotation speed, v_r. The goal is to have the robot drive to the goal state, while avoiding the obstacles in the environment. A reward of 1 is given for reaching the goal, a reward of -1 is given for colliding with an obstacle and a zero reward is given in all other situations. An experimental run terminates when the robot either reaches the goal, or hits an obstacle. The inputs to the learning system are the direction and distance to the goal state and the obstacles, as shown in figure 8. The robot is started in a random pose approximately three meters from the goal point for each experimental run. Our performance metric is the number of time steps needed to reach the goal state. In all of the experiments reported in this section, one obstacle

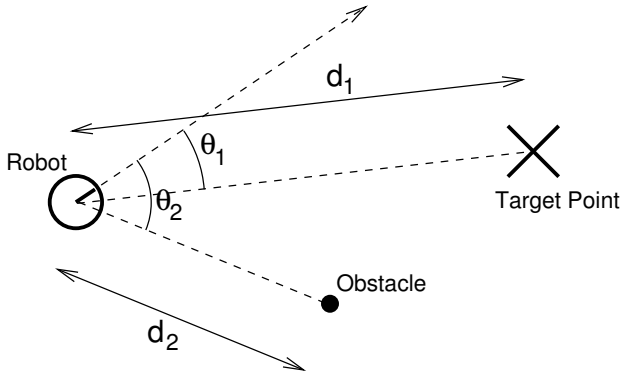


Fig. 8. Features for the obstacle avoidance task.

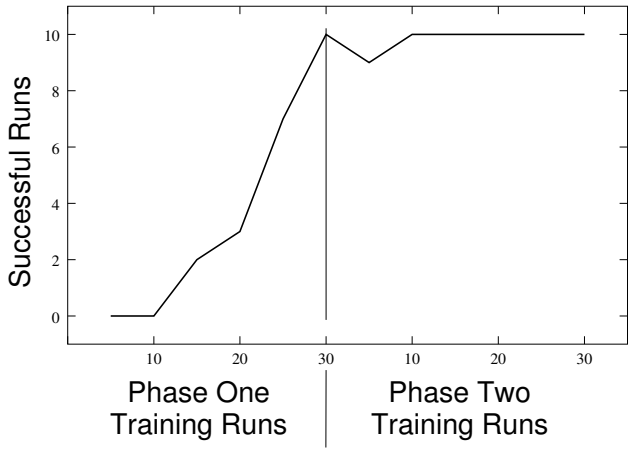


Fig. 9. Successful runs (out of 10) for the obstacle avoidance task.

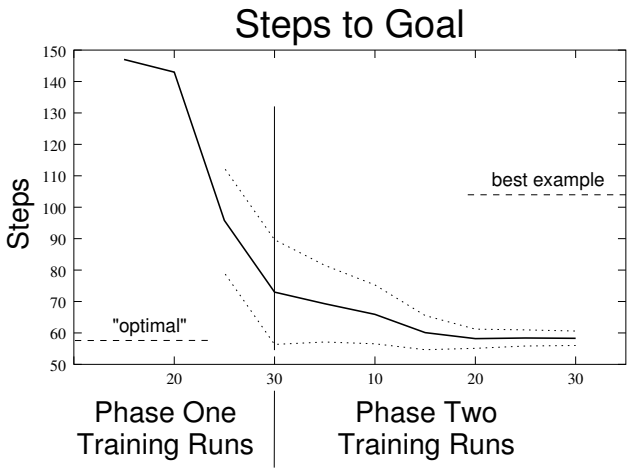


Fig. 10. Performance on the obstacle avoidance task.

was used and example trajectories were generated by direct joystick control of the robot.

The obstacle avoidance task is somewhat more difficult than corridor following. The robot is contained by the walls of the corridor and constrained by the dynamics of the robot. It is relatively difficult for it to get seriously lost, and to start driving down the corridor in the wrong direction. This means that it will (eventually) reach the reward-giving state at the end of the corridor in most experiments. However, in the obstacle avoidance task, there is no such help from the environment. In fact, policies that are almost perfect might just miss the goal state, and be unable to recover. This is reflected in figure 9, which shows the number of evaluation runs (out of ten) that actually manage to reach the goal state. Notice that th framework must see fifteen phase one example trajectories before it is capable of reaching the goal on its own. However, once it can find the goal state from one evaluation starting position, it quickly learns to reach it from all of them. As before, notice the slight performance decrease just after the change in learning phases.

For those evaluation runs that were able to reach the goal, figure 10 shows the average number of steps taken. The performance increase in the first learning phase is much

	Starting distance		
	1m	2m	3m
Successful	46.2%	25.0%	18.7%
Time (hours)	2.03	6.24	6.54

TABLE I

PERFORMANCE ON THE SIMULATED OBSTACLE AVOIDANCE TASK.

more dramatic than in the second phase. This is similar to the previous task when direct control is used for phase one training. Again, the final performance is not significantly different (at the 95% level) from the best that the authors could achieve with direct control of the robot. As with the corridor following task, the total time to reach the final performance level shown in figure 10 was approximately two hours.

As before, we ran a simulation of this task to see how long it would take the robot to reach the goal state without the benefit of phase one training. The robot was started at various distances from the goal state, pointing straight at it. No obstacles were present and rotation velocity, v_t, was chosen according to a normal distribution with mean zero and variance 0.05. Table I shows the number of runs that reached the goal within one week of simulated time, and the average time that these successful runs took. When starting at the same point as the real robot, less than one run in five was able to reach the goal state. Even those that did took an average of approximately six and a half hours. This is longer than the battery life of our robot. This demonstrates the necessity of example trajectories and phase one training when using reinforcement learning on a real robot.

VI. CONCLUSIONS

In this paper we have presented a framework for using reinforcement learning on mobile robots. The main feature of the system described in the paper is the use of example trajectories to bootstrap the value-function approximation, and the splitting learning into two phases. In the first learning phase, the robot is under the control of an example solution for the task, or is controlled directly by a human. The reinforcement learning system passively observes the states, actions and rewards encountered by the robot, and uses this information to bootstrap its value-function approximation. Once the learned policy is good enough to control the robot, the second phase of learning begins. The RL system is in control of the robot, and learning progresses as in the standard Q-learning framework.

Using example trajectories through the space allows us to easily incorporate human knowledge about how to perform a task in the learning system. The human guiding the robot does not need to know about the sensor and effector systems, or about reinforcement learning. They do not even have to show the robot the best solution for the task. In the experiments presented here, the final performance levels for both of the tasks are significantly better than any of the example trajectories used during phase one training. This underlines the point that we are not learning the trajectories that we are shown but are simply using them to generate experience that is then used by the reinforcement learning system.

Finally, the framework is capable of learning good control policies more quickly than moderately experienced programmers can hand-code them, at least for the tasks that we looked at. Anecdotally, we have observed that novice programmers can take up to a week to write a good wall-following program. In the experiments above, we managed to learn a good policy in approximately two hours.

We believe that our framework shows the promise of using RL techniques on real robots. However, there are still many open questions. How complex a task can be learned with sparse reward functions? How does the balance of "good" and "bad" phase one trajectories affect the speed of learning? Can we automatically determine when to change learning phases? We are addressing these questions in our current work.

REFERENCES

[1] Christopher J. C. H. Watkins and Peter Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[2] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, Adaptive Computations and Machine Learning. MIT Press, Cambridge, MA, 1998.

[3] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[4] Justin A. Boyan and Andrew W. Moore, "Generalization in reinforcement learning: Safely approximating the value function," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. S. Touretzky, and T. Leen, Eds. 1995, vol. 7, pp. 369–376, MIT Press.

[5] William D. Smart and Leslie Pack Kaelbling, "Practical reinforcement learning in continuous spaces," in *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000, pp. 903–910.

[6] William D. Smart, *Making Reinforcement Learning Work on Real Robots*, Ph.D. thesis, Department of Computer Science, Brown University, 2002.

[7] Geoffrey J. Gordon, *Approximate Solutions to Markov Decision Processes*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, June 1999, Also available as technical report CMU-CS-99-143.

[8] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal, "Locally weighted learning," *Artificial Intelligence Review*, vol. 11, pp. 11–73, 1997.

[9] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda, "Purposive behavior acquisition for a real robot by vision-based reinforcement learning," *Machine Learning*, vol. 23, pp. 279–303, 1996.

[10] Sridhar Mahadevan and Jonathan Connell, "Automatic programming of behavior-based robots using reinforcement learning," *Machine Learning*, vol. 55, no. 2–3, pp. 311–365, June 1992.

[11] Long-Ji Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, pp. 293–321, 1992.

[12] Paul Bakker and Yasuo Kinuyoshi, "Robot see, robot do: An overview of robot imitation," in *Proceedings of the AISB96 Workshop on Learning in Robots and Animals*, 1996, pp. 3–11.

[13] Stefan Schaal and Christopher G. Atkeson, "Robot learning by nonparametric regression," in *Proceedings of Intelligent Robots and Systems 1994 (IROS '94)*, V. Graefe, Ed., 1995, pp. 137–154.