Adaptive Robotics - Final Report
Extending Q-Learning to Infinite Spaces

Eric Christiansen
Michael Gorbach

May 13, 2008

Abstract

One of the drawbacks of standard reinforcement learning techniques is that they only operate when both the state and action spaces are finite. Q-learning is one such algorithm. We propose an extension of Q-learning to infinite state and action sets called CHAMPAGNE, using a simple "Local Expert" function approximation method. We then experimentally test the performance of the algorithm on several navigation tasks. The algorithm is able to successfully solve a T-maze in pyrobot after reasonable training. We present results from varying the input type, reinforcement delay, and maximum memory size for this algorithm.

1 Introduction

The essence of the Q-learning algorithm is the task of filling in entries in a 2 dimensional table, where one axis corresponds to the finite set of possible states the system can be in, and the other axis corresponds to the finite set of possible actions the system can take, and where the i, j entry in the table is estimates the intrinsic value of performing action j when in state i. Any particular estimate i, j in the table is reliant solely on observations made when the jth action is performed from the ith state; the canonical Q-learning algorithm for updating estimates assumes no relationship between different states and different actions. This often means it does not take into account important knowledge about the problem. In this paper, we implement a simple extension of Q-learning to infinite sets of states and actions.

2 Methods

We wish to extend Q-learning to infinite state and action sets. Clearly, the existing Q-learning algorithm, given a finite running time, could never produce an estimate for every entry in an infinitely large table. What is needed is a means of interpolating between the entries in the table.

The interpolation task can be understood as a function estimation task, where the domain is the set of all pairs of states and actions, and the codomain is $\mathbb{R}$, corresponding to reward. We seek an algorithm which can perform this function estimation well, so that we can fairly say our method extends Q-learning to infinite spaces. The simplest method for implementing this approximation, which we call Local Expert Regression, turns out to be effective in solving navigation tasks. The method is described below.

To implement a Q-learning like algorithm, the function approximation must be supplemented with another element: estimation maximization. Given the estimated function, which gives a reward for every state and action pair, estimation optimization determines which action the agent should take from a particular state. Estimation optimization chooses the best move for the agent each turn.

2.1 CHAMPAGNE

2.1.1 Function Approximation: Local Expert Regression

Given a training set of points $\{p_i\}$ from the domain, and a set of values $\{R(p_i)\}$ associated with each point, let p be a point in the domain. We estimate $R(p)$ by finding a $p_{\min} \in \{p_i\}$ such that $||p - p_{\min}|| \leq ||p - p_i||$ for every i, i.e. we let $p_{\min}$ be the point in the training set closest to p in the Euclidean metric. Our estimate for $R(p)$ is then $R(p_{\min})$.

If our domain is bounded, and the function f we are trying to estimate is uniformly continous, then this algorithm is guaranteed to yield an arbitrarily good approximation of f given enough training points. Here, uniform continuity has a precise analytical meaning, but can be interpreted to mean that knowing the value of f at a point yields information about the values of f near that point. This condition on f is entirely reasonable; the set of functions a neural net can model is in fact a subset of the set of uniformly continuous functions.

In practice, our domain is a space D that is a cartesian product of two spaces: $D = S \times A$. Here, S is the space of all possible (sensor) states of the robot, while A is the space of all possible robot actions. The Local Expert scheme gives an estimate $\hat{R}$ for the function $R : D \to \mathbb{R}$.

2.1.2 Estimation Optimization

As noted above, in order to create a complete learning system, a function approximation scheme (in this case, Local Expert Regression), must be combined with a scheme for estimation optimization. The current state $s_c \in S$ defines a subspace S_c of D. The goal of an estimation optimization scheme is to maximize the value of $\hat{R}$ on S_c.

In Q-Learning, this is a simple task because there is a finite set of actions associated with a state, so we can exhaustively evaluate $\hat{R}$ for each state/action pair in S_c. However, in our case S_c is infinite (of the same dimension as A), so we perform an approximate optimization of $\hat{R}$ by sampling n_{sample} points in S_c and returning the point with best $\hat{R}$. If R is reward, we try to maximize $\hat{R}$. If R is error, we try to minimize $\hat{R}$.

Note that Q-Learning additionally uses an exploration parameter, which determines how often the system will choose a random move instead of the estimated optimal move. In CHAMPAGNE, the n_{sample} parameter implicitly leads to exploration; the fact that n_{sample} is finite and so optimization isn't exact leads to non-optimal actions being explored. In CHAMPAGNE, n_{sample} also grows with the square root of the number of moves previously taken. This was done to allow more strongly random exploration initially, followed later by fine-tuning towards an optimal solution.

2.1.3 Delayed Reward

In practice, the function we want to estimate assigns a reward to each point in the domain. However, in many robotics applications, knowledge of reward corresponding to each point in the domain is unavailable. Instead, we have only reward for a sequence of moves by the robot. The natural question is how this summary reward should be assigned among the sequence of moves leading up to the reward. For example, a Chess-playing agent may make many moves before the game is over, but reinforcement takes place only at the end of the game, and it isn't obvious how each individual move affected the overall performance.

In CHAMPAGNE, we have a simple method of distributing cumulative reward. If the cumulative reward after a sequence of m moves is R_C, then the reward assigned to each move in the sequence is $\frac{R_C}{m}$.

2.1.4 Finite Memory

The Local Expert algorithm, as formulated above, requires all observations that the agent makes to be stored indefinitely. This implies unbounded memory requirements. We can easily sidestep this difficulty by limiting the number of observations stored. Specifically, we can keep only the last $n_{memsize}$ observations, and throw older ones out.

2.2 Specific Tasks

To validate CHAMPAGNE we needed to define a performance metric. Several different robot navigation tasks were used for this purpose.

2.2.1 Target Following Game

As an initial test for the algorithm, we used a simple target-following navigation task. The game is played in rounds, each m turns long. Each round, a target point (x_t, y_t) is randomly set in the unit square. The agent itself, represented as a point, is started at another random point (x_0, y_0) in the unit square. Each turn, the agent can move a distance of d_{max} in any direction from 0 to 2π to a new position (x_c, y_c). At the end of each turn, the agent is reinforced with the euclidean distance (error) between its current position and the target point. As "sensor values", the agent perceives both its location and that of the target point.

We thus have a 4 dimensional real sensor space $S = \{x_t, y_t, x_c, y_c\}$, and a 2 dimensional action space $A = \{d, \theta\}$, giving a total 6 dimensional space $D = S \times A$. Our

function approximation then maps $D \rightarrow R$, where R is the (scalar) reward, or in this case the error.

2.2.2 T-Maze Task

As a more difficult task for our algorithm, we used a T-Maze navigation problem similar to that discussed in the SODA paper [1]. See Figure 1 for a diagram of the T-Maze world.

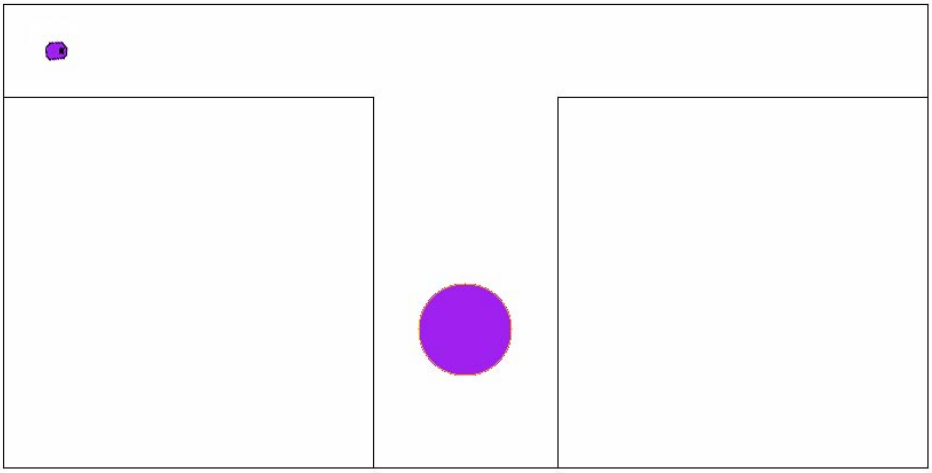

Figure 1: An illustration of the T-Maze task world. The robot and the target are in purple. The target is shown here as a purple light, but the robot can not perceive it.

The movement of the robot was set up similarly to that of the target-following task (2.2.1). The robot was able to move, each turn, in any direction up to an adjustable maximum distance d_{max}. The robot was repositioned to a fixed starting point in the top left corner at the beginning of each round, and then given a total of m moves to find the target. The moves of the robot were controlled a brain running the CHAMPAGNE algorithm discussed in 2.1.

To examine the performance of the algorithm with increasingly difficult problems, several variations of this task were studied. We varied three parameters: the values given as sensor input, the delay of the reinforcement, and memory size. Two types of sensor inputs were examined. Coordinate input involved giving the robot, as sensor data, the exact coordinates of its position and that of the target. Sonar input involved giving the robot readings from 4 of its 8 sonar sensors as input (skipping every other sensor, going around the robot's body). Obviously, we expected the use of coordinate input to make the task significantly easier to solve. We also varied the type of reinforcement given to the robot. In immediate reinforcement, the robot was given the error (euclidean distance from target) after each move. In delayed reinforcement, the robot was given the sum of its errors over a round as the only reinforcement, at the end of each round. The delayed reinforcement was distributed according the discussion in 2.1.3.

3 Results

Figure 2 presents a learning curve for the target-following task. Here, and in all further tasks, we used a round of $m = 10$. For this task, the maximum move distance was $d_{max} = 0.5$, and the world size was 1×1. As expected, we saw rapid convergence and this task served as a sanity check for the the method.

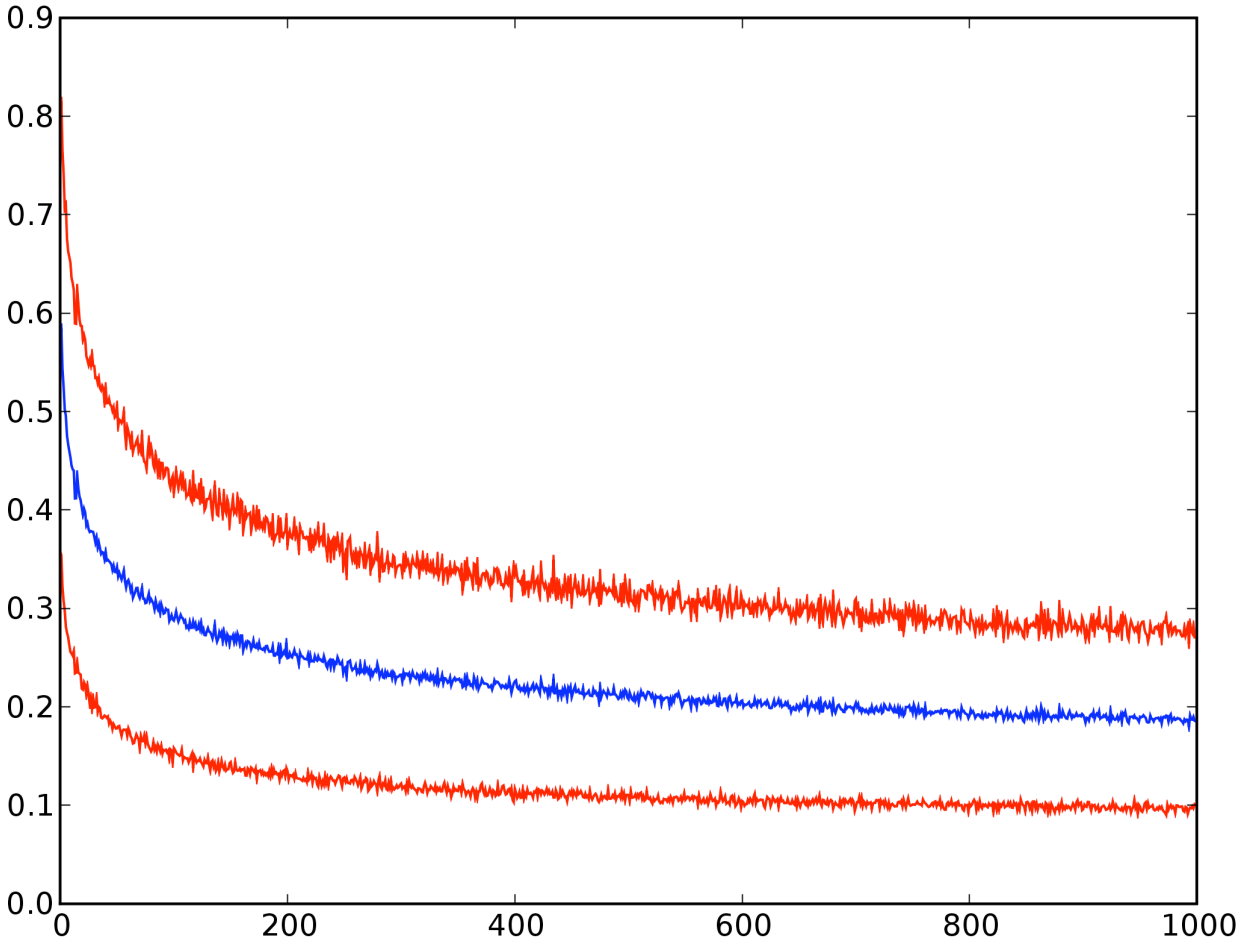

Figure 2: Performance in target following task. The x-axis is the number of training rounds and the y-axis shows cumulative error over each round. The blue curve is mean performance, and the red lines are plus and minus one standard deviation. The graph comes from 1200 independent runs.

Figures 3 and 4 present the results of measuring learning performance across different variants of the T-Maze task. For all the remaining trials, we used $d_{max} = 2$ and a world size of 10×20. We independently varied the type of sensor input and the reinforcement delay. As expected, convergence is very rapid for coordinate sensor and immediate reinforcement. Interestingly, sonar sensors with immediate reinforcement and coordinate sensor with delayed reinforcement performed about equally, suggesting that these sets of additional information have roughly equal value to the algorithm.

The task with delayed reinforcement and sonar sensor was significantly harder than any of the other tasks. However, the agent was able to learn achieved good performance with sufficient training (see Fig. 5). We see that the agent continues to improve to a significant degree with additional training.

Figures 5 and 6 present the results of varying the memory size. We restrict the memory size to 100, 1000, and 10,000 observations, as well as allowing for unlimited memory size. In nearly every case, increasing the memory size improved performance (see fig. 5). Extremely small memory sizes made learning difficult, though some learning is evident in these curves. It is interesting to note that going from 10,000 to unlimited memory size in the sonar sensors and delayed reinforcement condition did not significant improve performance. This may be due to the difficulty of the condition causing a large of amount non-useful data to be held in memory.

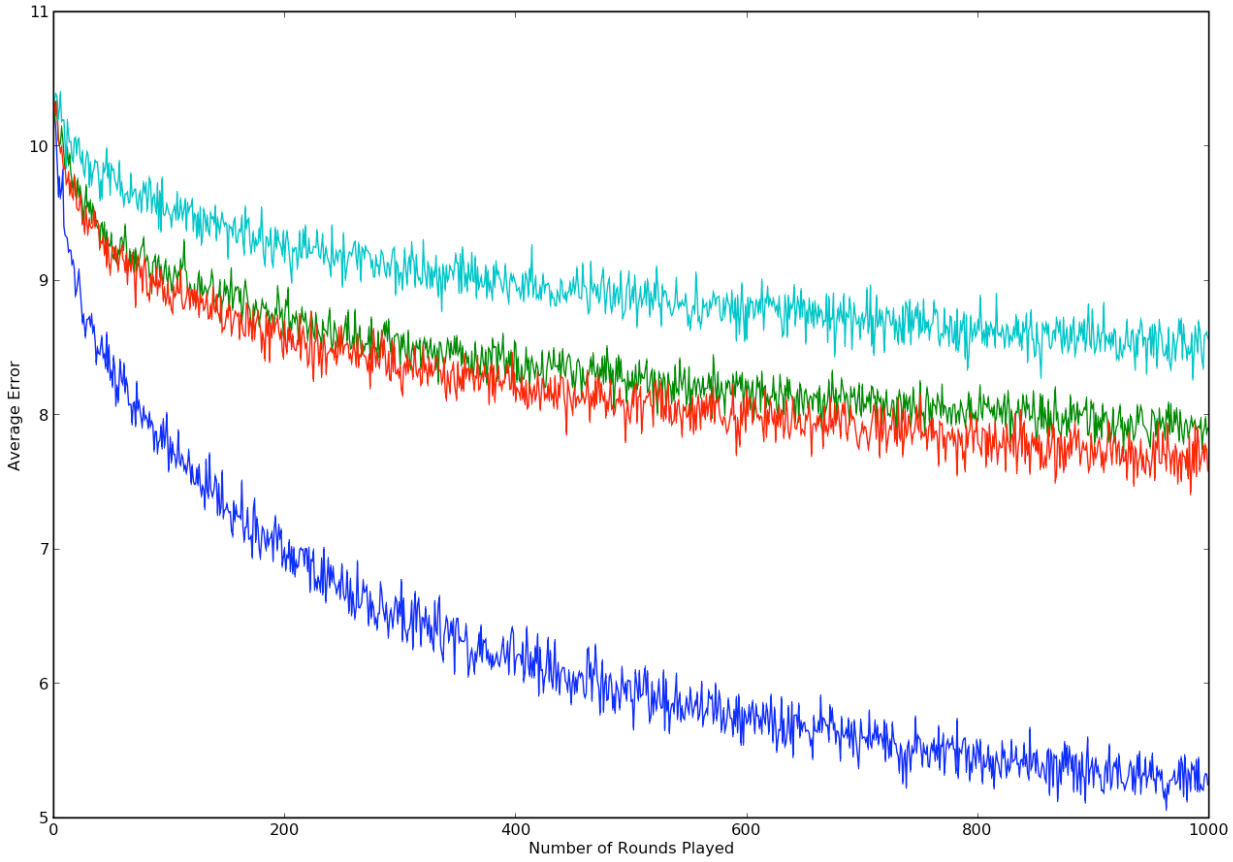

Figure 3: Performance versus sensor and reinforcement types. The blue curve shows the performance of an agent with coordinate sensors and immediate reinforcement. The red curve corresponds to sonar sensors and immediate reinforcement. The green curve corresponds to coordinate sensors and delayed reinforcement. The cyan curve corresponds to sonar sensors and delayed reinforcement. All robots had unlimited memory size. The graph comes from 100 independent runs.

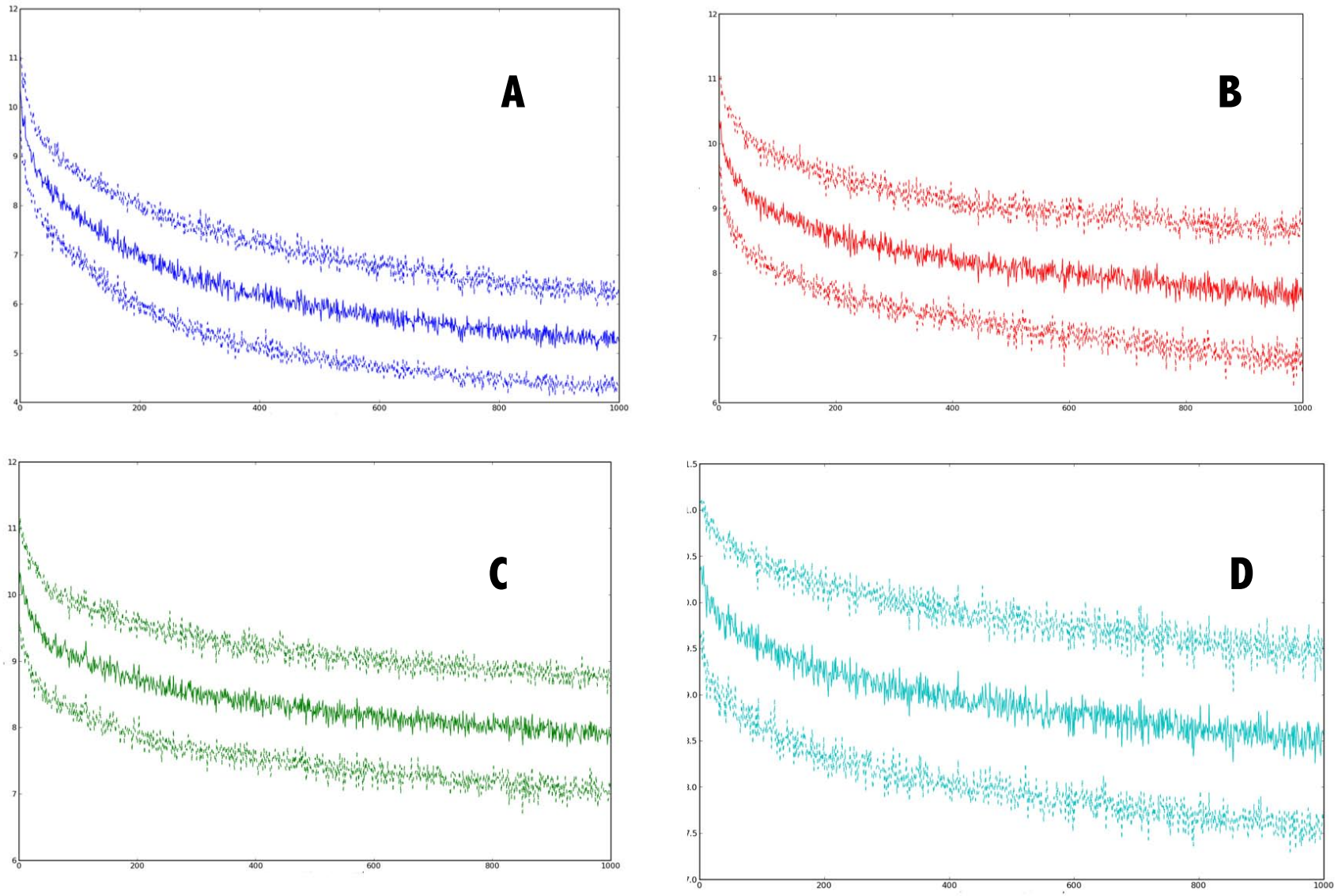

Figure 4: Performance versus sensor and reinforcement types. Each graph has the number of rounds trained on the x-axis, and total error on the y-axis. The lines bounding each central curve are plus and minus one standard deviation. Graph A shows the performance of an agent with coordinate sensors and immediate reinforcement. B corresponds to sonar sensors and immediate reinforcement. C corresponds to coordinate sensors and delayed reinforcement. D corresponds to sonar sensors and delayed reinforcement. All robots had unlimited memory size. The graph comes from 100 independent runs.

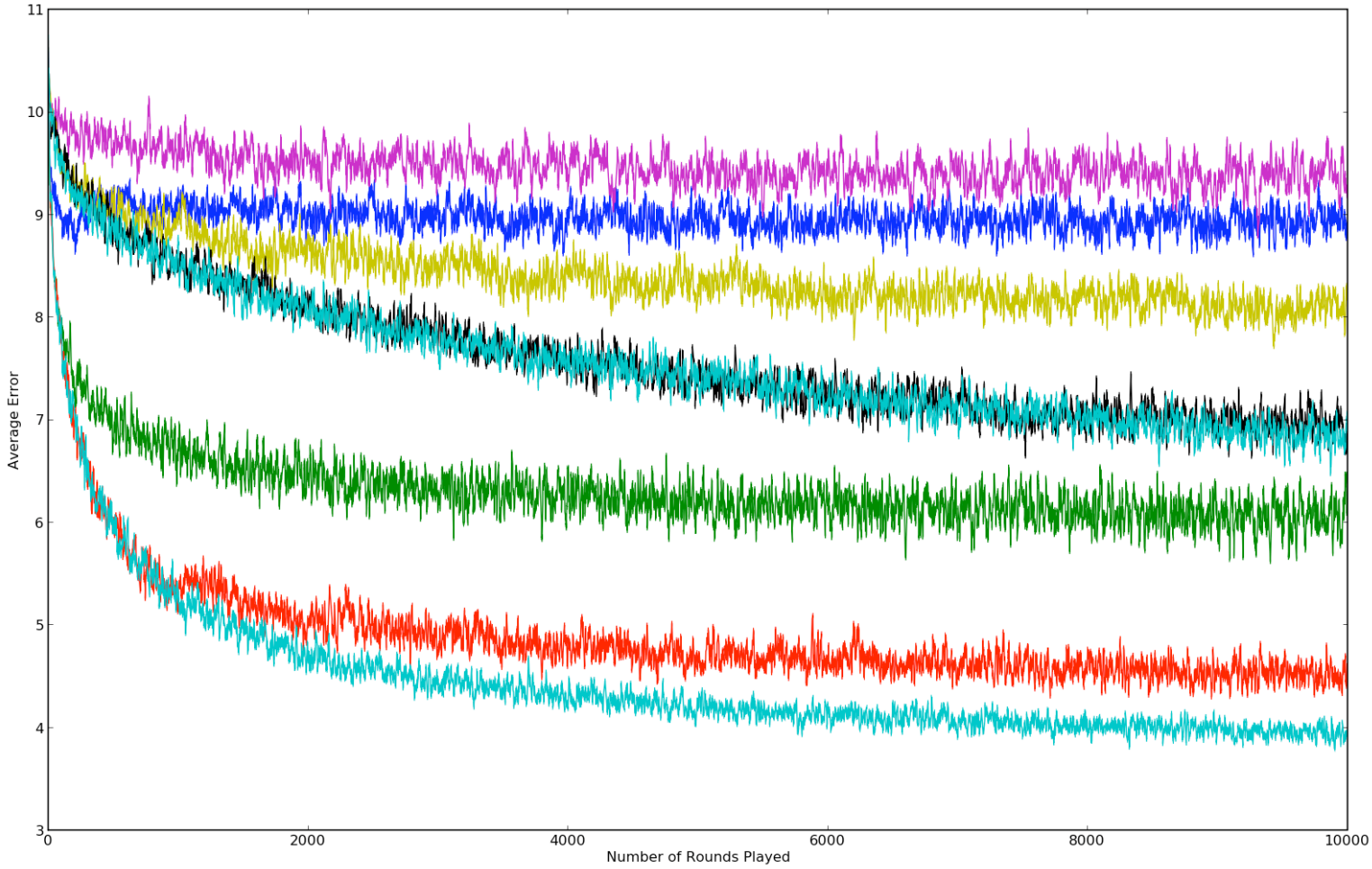

Figure 5: Performance versus memory size. Each graph has the number of rounds trained on the x-axis, and total error on the y-axis. Blue: performance of an agent with coordinate sensors, immediate reinforcement, and memory size of 100 observations. Green: coordinate sensors, immediate reinforcement, and memory size of 1000. Red: coordinate sensors, immediate reinforcement, and memory size of 10000. Lower cyan: coordinate sensors, immediate reinforcement, and unlimited memory. Purple: sonar sensors, delayed reinforcement, and memory size of 100. Yellow: sonar sensors, delayed reinforcement, and memory size of 1000. Black: sonar sensors, delayed reinforcement, and memory size of 10000. Upper cyan: sonar sensors, delayed reinforcement, and unlimited memory. The graph comes from 10 independent runs, and a 10-step moving average was used for smoothing.

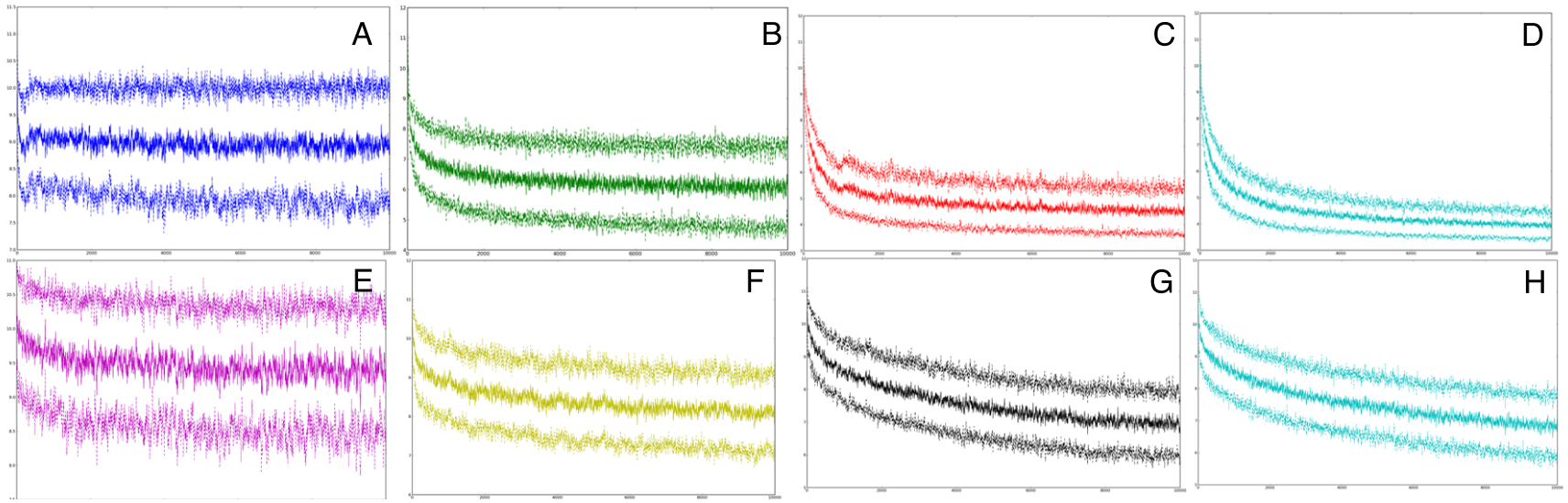

Figure 6: Performance versus memory size. Each graph has the number of rounds trained on the x-axis, and total error on the y-axis. The lines bounding each central curve are plus and minus one standard deviation. Graph A shows the performance of an agent with coordinate sensors, immediate reinforcement, and memory size of 100 observations. B corresponds to coordinate sensors, immediate reinforcement, and memory size of 1000. C corresponds to coordinate sensors, immediate reinforcement, and memory size of 10000. D corresponds to coordinate sensors, immediate reinforcement, and unlimited memory. E corresponds to sonar sensors, delayed reinforcement, and memory size of 100. F corresponds to sonar sensors, delayed reinforcement, and memory size of 1000. G corresponds to sonar sensors, delayed reinforcement, and memory size of 10000. H corresponds to sonar sensors, delayed reinforcement, and unlimited memory. The graph comes from 10 independent runs, and a 10-step moving average was used for smoothing.

4 Discussion

4.1 CHAMPAGNE and SODA

One of the original goals of CHAMPAGNE was to solve a task similar to the SODA T-Maze with a simpler algorithm. In a direct sense, CHAMPAGNE did succeed in solving the T-Maze, however the solution is in some senses not comparable with that of SODA. There are several important differences.

First, the reinforcement system is CHAMPAGNE is different from that of SODA. The SODA agent was given only a binary reinforcement indicating whether or not it had reachd a target area, at the end of every round. The CHAMPAGNE agent, while also being reinforced at the end of round in a delayed fashion, was reinforced not with a binary value like SODA, but with its sum distance from the target. The allowed CHAMPAGNE to gradually optimize the route, however it made the task significantly easier than in SODA.

Second, SODA was dealing with reinforcement having a significantly longer delay than in our experiments. In CHAMPAGNE, the step size and round length had to correspond so that the robot could cover the distance in the given time. The round length was 10, while the step size was 2, for a 10×20 world. The reinforcement delay in these trials was therefore 10 moves, which was significantly fewer than for SODA. One way to address this issue for CHAMPAGNE is to implement a concept of inertia. The change in "move orders" each turn can be limited. Specifically, the change in the move vector (d, θ) would be limited to small rectangular region in the action subspace. This would allow CHAMPAGNE to continue functioning well with significantly smaller step sizes, and would allow testing of CHAMPAGNE with correspondingly longer reinforcement delay.

Third, the SODA actions were at a different semantic level than those used CHAMPAGNE. While CHAMPAGNE actions consisted of giving a move distance and direction (d, θ), in SODA the agent started by considering low-level commands to engines, and abstracted them to higher-level trajectory-following control rules. The SODA agent was deciding between "keep going" actions and slight corrections.

Fourth, the way we simulated movement was not as physically accurate as the way the SODA simulator did. For reasons of code efficiency, we chose to "teleport" the robot to the target it specified at each turn, instead of driving the robot to that location. This let the robot cut corners (though it could not teleport past walls, as our code prevented the robot from leaving the world). Since the robot could only cut a corner if it had nearly passed it, we do not believe that this physical incongruity had much effect on our results.

4.2 Comparison with other Methods

It is important to compare the performance of CHAMPAGNE with other reinforcement learning methods. We explored two alternatives: Q-learning and a neural net approach.

4.2.1 Q-learning

In the Q-learning alternative, we first discretize our state and action space by imposing a lattice on the space separating each dimension into l equally-sized bins. This breaks our space up into l^d hypercubes, which we make into entries in a standard Q-table setup. We then perform Q-learning on this table.

In testing, we never achieved good performance with this algorithm. There are at least two reasons this could have happened. First, it could be that we had a bug in the implementation. Second, we may have encountered one of the fundamental problems with discretizing a space before anything is known about it. While CHAMPAGNE preferentially made observations and thus added table entries in regions corresponding to high performance, the Q-learning algorithm was stuck with the discretization it made at the very beginning of its training.

4.2.2 Neural Nets

The Local Expert Regression algorithm in CHAMPAGNE was simply a function approximator, and so can be swapped out with any other algorithm which performs regression. A standard method of nonparametric regression uses a neural net, where each training input is a point from the set of states and actions, and each target output is the estimated reward at that point. In practice, we were never able to find a neural net implementation that was nearly fast enough to compete with our CHAMPAGNE implementation, which used a kd-tree to store the observations. An advantage of using a neural net for function estimation is that, once the net is trained, the observations can be thrown away. The flip-side is that if we don't keep the observations, the system is vulnerable to catastrophic forgetting.

5 Conclusion

The simple CHAMPAGNE algorithm was able to successfully perform both the target-following and SODA T-Maze tasks. However, as noted in section 4.1, the experimental setup used here makes the CHAMPAGNE solution of the T-Maze task not entirely comparable to that in the SODA paper.

6 Future Work

- As noted in 4.2, it is important to complete the comparative analysis of champagne with algorithms such as Q-Learning and the neural net approach.
- Modify CHAMPAGNE such that earlier observations can be changed directly, updating using a rule similar to that of Q-Learning.
- It is possible to solve the estimation optimization problem described in section 2.1.2 exactly, as opposed to approximately. This would require an implementation that would keep full d-dimensional Voronoi diagram consistently updated at each observation. This may not be practical given the computational cost of Voronoi diagram algorithms. This may be able to be reduced due to the fact that we are interested in a maximizing only over a subspace defined by the current action.
- The Curse of Dimensionality is fairly universal, but it would be interesting to see how affected by the curse CHAMPAGNE is. One way to do this is to increase the number of sensors available to the robot.
- Every function estimation scheme puts an implicit prior over the set of possible data generators, and if that prior is not appropriate to a task, it will be difficult for the estimator to perform well on that task. It would be interesting to precisely state the assumptions our model makes.

References

1. Development navigation behavior through self-organizing distinctive state abstraction. Provost, et al. Artificial Intelligence Lab. University of Texas at Austin. 2005.