Robot Behavioral Selection Using Q-learning *

Eric Martinson    Alexander Stoytchev    Ronald Arkin
Mobile Robot Laboratory
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 U.S.A.

Abstract
Q-learning has often been used to learn primitive behaviors, or to coordinate a limited set of motor skills. However, the complexity of the algorithm increases exponentially with the number of states the robot can be in and the number of actions that it can take. Therefore, it is natural to try to reduce the number of states and actions in order to improve the efficiency of the algorithm. Robot behaviors and behavioral assemblages provide a good level of abstraction which could be used to speed up robot learning. Instead of coordinating a set of primitives, we use Q-learning to coordinate a set of well tested behavioral assemblages to accomplish a robot mission. The domain for our experiments is a simple intercept mission. This paper also explores the effects of imperfect perceptual algorithms on learning when this approach is used.

1 Introduction
The driving force behind the use of Q-learning in mobile robot tasks is simplicity of design. Ideally, the application designer will not have to be a specialist to work with mobile robots. They will look at a list of possible perceptual triggers and pick out all the triggers that may or may not be useful and add them to the Q-learning module. These triggers combine to form the perceptual state space. Next the designer looks at a set of already built task assemblages or FSAs, such as traverse hallway, find biohazard, wander, etc. and chooses a subset of these that may prove useful to the robot and adds them to the Q-learning module. These prebuilt task solutions become the set of possible actions that the robot can choose from. Finally the user adds some reinforceers to define when the robot is rewarded or penalized, and lets the simulation run until the learning converges to a stable solution.

In this paper, we apply machine learning to learn when it is appropriate to select an appropriate set of behaviors (among several) that are suitable for the specific task at hand. Reinforcement learning in particular seems most appropriate to this type of problem. As such, this paper addresses the problem of coordination of high-level task solutions using a variant of reinforcement learning: Q-learning.

This research is part of an ongoing DARPA project for the Mobile Autonomous Robotics Software (MARS) Program that focuses on multi-level learning in hybrid deliberative/reactive architectures. Other related papers from our laboratory relevant to this effort include [18, 17, 16].

2 Definitions
In this section we define the terms that will be used in the remainder of this paper.

- **WorldState** - actual state of the physical world, including what the sensors of the robot can and cannot detect.
- **PerceptualState** - state of the world as represented by the perceptual triggers associated with this agent.

All combinations of perceptual triggers represent distinct perceptual states. If there are \( n \) perceptual triggers, then the number of distinct perceptual states is:

\[
\text{NumberOfStates} = 2^n
\]

- **InternalState** - the state of the world as recognized by the Q-learning coordination mechanism. It is different from **PerceptualState**, because the internal state could represent more information than is encapsulated by a single perceptual state. For instance, the internal state could represent a combination of perceptual state and the current action of the robot. Internal states could also be states separated through time as in Chirikjian's approach [11]. However, for the remainder of this document and the task described herein, the internal state will correspond directly to the recognizable perceptual states.
- **Action** - the behavioral assemblage selected by the Q-learning coordination mechanism. This assemblage could encapsulate a single behavioral primitive, or contain a set of behaviors or assemblages with its own coordination mechanism.
- **QvalueTable** - the table used (and updated) by the Q-learning algorithm to determine the next action of the robot. The table has size \( K \times A \), where \( K \) represents the number of distinct internal states, and \( A \) is the number of possible actions.
- **Reinforcer** - a tuple of perceptual state, action, and reward \( < s, a, r > \) which defines when and what reward is applied. When a perceptual state \( s \) is observed, and the robot is performing action \( a \), then a reward \( r \) is given to the Q-learner.
- **Policy** - the equivalent behavioral mapping defined by the Qvalue table, i.e., what actions would the Q-learner choose in each of the possible internal states.
3 Related Work

This section presents the related work on reinforcement learning in behavior-based robotics. For an overview of reinforcement learning in general see [29, 15, 16].

Reinforcement learning, as used today in behavior-based robotics, can be classified into the following categories depending on the task:

1. learning primitive behaviors
2. optimizing primitive behaviors
3. learning composite tasks
4. social learning

When learning a primitive behavior the task of the robot is to master one specific behavior based on a reinforcement function. A primitive behavior is a behavior that cannot be broken further into sub-behaviors. If a more complex task is desired, several primitive behaviors can be learned individually and then combined together by a sequencing mechanism (see Compositional Learning below).

Maladevan and Connell [22] used Q-learning to teach a behavior-based robot how to push boxes around a room without getting stuck. The task was broken manually into three subtasks (behaviors): finding a box, pushing a box, and recovering from stalled situations. The task decomposition method was compared to a monolithic approach where the task is learned as a single behavior. The results show that the task decomposition method learns the task about two times faster.

Asada et al. [6, 4], in the context of robot soccer, trained a robot to shoot a ball into a goal using visual feedback. Furthermore, Asada et al. [5] evaluated three different methods for learning new behaviors by coordinating existing behaviors learned separately by reinforcement learning.

Behavior optimization is useful when a robot almost knows how to achieve a task but the execution of the task needs to be fine tuned. The knowledge of the robot may come from prior experience or from a rough solution suggested by a human. Franklin [14] used reinforcement learning to refine robot motor control for nonlinear tasks. Smart and Kaelbling [28] use human generated control strategies to bootstrap a robot controller that uses Q-learning to refine these policies from experience.

The task of the robot is corridor following and the human uses a joystick to teach the robot an initial noisy and suboptimal control policy.

In Compositional Learning the assumption is that the robot already has a repertoire of primitive behaviors and its goal is to learn how to organize them to produce a more complex behavior which consists of primitive behaviors. Maes and Brooks [15] used this approach to train a 6-legged robot to walk. The robot was given positive reinforcement when it successfully moved forward and negative reinforcement when it lost balance and fell on the floor. Each leg had a set of associated behaviors for lifting and swinging forward. The robot learned to coordinate the behaviors and was able to achieve a stable tripod gait (i.e., three of its legs were on the ground at any moment: the middle leg on one side and the front and back leg on the other side).

An interesting variation of Compositional Learning is Robot Shaping [13, 25]. In this case a human trainer, or a training program, trains a robot to perform sequential tasks such that each new subtask in the sequence is either a known task or a slight modification of a known task. This roughly corresponds to the shaping procedures used by Skinner [27]. The majority of work in robot shaping [26, 7, 13], however, is more closely related to chaining of already learned or innate behaviors. A notable exception is [25] and [31], which provides a computational model for modifying the existing robot behaviors through selective reinforcement by a human trainer to achieve new behaviors.

In social learning the idea is to use the social environment of a robot colony as a context in which to provide reinforcement to the individual robot. Bak [8] was among the first to use Q-learning for learning behavioral diversity in teams of robots. Mataric [23] conducted experiments in social learning within a foraging context.

4 Overview of Q-Learning

Probably the most widely used reinforcement learning method for robotic systems is Q-Learning [30]. This is mostly due to its algorithmic simplicity and the ease of transitioning from a state to a value function to an optimal control policy by choosing in every state the action with the highest value. Following Kaelbling’s approach [15], at every time step the robot perceives the perceptual state $s$. Based on this information the robot chooses an action $a$ and executes it. The utility of this action is communicated to the robot through a scalar reinforcement value $r$. The goal of the robot is to choose actions that, in the long run, maximize the sum of the reinforcement value.

Let $S$ be the set of distinct internal states that the robot can be in and let $A$ be the set of actions that the robot can take. Let $T(s, a, s')$ be the probability of transitioning from state $s$ to state $s'$ using action $a$. If we are given a world model defined by the transition probabilities and the reward function $R(s, a)$ we can compute an optimal deterministic stationary policy using techniques from dynamic programming (like Value Iteration or Policy Iteration).

It is usually the case, however, that a world model is not known in advance and the robot needs to learn this model and simultaneously construct an optimal policy. Q-learning is an algorithm that does just that. Let $Q^*(s, a)$ be the expected value of the discounted reinforcement of taking action $a$ in state $s$. The value of this quantity can be estimated recursively with the following formula:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a') \quad (1)$$

The optimal policy in this case is:

$$p^* = \text{arg max}_{a} Q^*(s, a) \quad (2)$$

In other words, the best policy is, in each state, to take the action with the largest Q-value. Thus the $Q$-function makes the actions explicit, which allows us to compute them on-line using the following $Q$-learning update rule:

$$Q(s, a) = Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \quad (3)$$

where $\alpha$ is the learning rate, and gamma is the discount factor ($0 \leq \gamma < 1$). It can be proven [30] that this formula converges if each action is executed in each state
an infinite number of times and $\alpha$ is decayed appropriately. For a more detailed discussion of Q-learning refer to [30, 15].

5 Task Description

The Q-learning method described above was used to build a behavioral coordination mechanism for an intelligent anti-tank mine robot. The anti-tank mine is designed to intercept enemy tanks as they move down a nearby road and destroy them. The only sensor the mine has available to it determines the location of enemy tanks within a certain radius. From this data and a series of time-stamped readings, the velocity of the tank can be computed. This sensor information is then used in two perceptual triggers: CAN_INTERCEPT and NEAR. The first trigger, CAN_INTERCEPT, is true if the tank is interceptible given its current velocity and the maximum velocity of the mine. The NEAR trigger is true if the mine is within detonation range of the enemy tank. Combined, these two triggers imply 4 distinct perceptual states. As mentioned earlier, in this task, perceptual states directly correspond to individual internal states in the Q-learner. Appendix A provides a detailed pseudocode description of how the Q-learner was set up in the Missionlab software system [20, 21], and the settings used for this task.

After making an observation, the mine can choose three different actions to perform: WAIT, INTERCEPT, or TERMINATE. The WAIT action causes the mine to stop moving. The INTERCEPT action makes the mine move towards the nearest distance intercept point with the tank. Specifics on how the intercept point is dynamically calculated as the mine moves are provided in Appendix B. The TERMINATE action blows up the anti-tank mine and ends the learning scenario. Only when the TERMINATE action is chosen, and the NEAR trigger is ON is a reward of +10 provided to the robot.

The state space of this problem can be seen in Figure 1. The circles represent the 4 distinct perceptual states in binary format. 00 means no valid trigger, 01 means CAN_INTERCEPT only, 10 is NEAR only, and 11 means both triggers are valid. The arrows demonstrate the possible transitions the Q-learner can take in the scenario. For example, if the CAN_INTERCEPT trigger is valid (i.e., perceptual state 01), but the anti-tank mine fails to intercept the tank, the state transitions to 00. However, if the mine successfully catches the tank, then the tank stops and the mine remains in state 10 or 11 until the tank is destroyed.

Figure 2 displays a sequence of actions taken by a successful policy. The anti-tank mine starts in the circle marked start, and waits for the tank to appear. When the tank is detected, it moves towards the intercept point. The mine actually has to reach the intercept point before the tank, or it is less likely to perform a successful intercept, as seen in the third frame capture. When the tank is caught, the TERMINATE action is called and the world is reset for the next run.

6 Anti-tank Mine Characteristics

Three assemblages are used in the intercept scenario, INTERCEPT, WAIT, and TERMINATE. The TERMINATE behavior is used for the self-destruction of the mine when the tank is nearby, with the scenario being reset. Two perceptual triggers are also used: CAN_INTERCEPT, and NEAR.

6.1 INTERCEPT Behavioral Assemblage

The vector output of two primitive behaviors [6], INTERCEPT and AVOID_OBSTACLES, are summed together multiplied by their individual gains to form this assemblage.

- INTERCEPT behavior - uses the equations described in Appendix B to move towards the closest point of interception perpendicular to a line projected along the current detected direction of the enemy robot.
  
  Gain: 1.0
  Parameters:
  $\text{maxdist} = \text{NA}$
  $\text{maxspeed} = 0.1$

- AVOID_OBSTACLES - standard behavior provided with Missionlab.
  
  Gain: 1.0
  Parameters:
  $\text{sensor\_range} = 800$
  $\text{avoid\_obstacle\_sphere} = 1.0$
  $\text{avoid\_obstacle\_safety} = 0.5$

6.2 WAIT Assemblage

WAIT is the same as the STOP assemblage provided with Missionlab. The output is a vector of length 0 at every time-step, causing the robot to stay at the same location.

6.3 TERMINATE Assemblage

The output of the TERMINATE assemblage is the same as that of the WAIT assemblage, except that the scenario is reset if the TERMINATE action is performed, as if the robot had destroyed itself.
6.4 CAN_INTERCEPT Trigger

This trigger determines whether or not the anti-tank mine can reach the point of interception before the tank does. The specific algorithms for determining interceptibility are discussed in Appendix B.

Parameters:

max_velocity = 0.1
max_dist = 100

6.5 NEAR Trigger

NEAR uses the NEAR_ENEMY primitive in MissionLab parameterized as follows:

- NEAR_ENEMY - provided a set of enemy objects, determine whether the robot is within a given distance of those enemy objects.

Parameter:

dist_range = 0.6 m
7 Initial Results

To test and train the Q-learner, the learning process is divided into 200 learning scenarios. A learning scenario consists of 300 time-steps in which the mine is attempting to intercept the tank. If the TERMINATE action is chosen before 300 time-steps, reinforcement immediately ceases. At the end of each scenario the two agents (anti-tank mine and enemy tank) are restarted from their original starting positions and a new scenario begins.

The Q-value Table is initialized in the beginning of every experiment with a set of random values between 0 and 1. Every time the robot’s perceptual state changes, or a reward is received, the table is queried to determine the action with the highest Q-value. This is different from Kaelbling’s approach which queries the table at every time-step (Equation 3).

Whenever the Q-value table is queried, Watkins' update rule is also applied. The update rule uses a discount factor of $\gamma = 0.9$, and a decaying $\alpha$ value. The $\alpha$ value is initialized to 1.0 and is reduced each time the table is updated using the update formula $\alpha = 0.99\alpha$. An exploration method using a decaying exploration factor was used. The initial exploration rate, $e$, was set to 0.5 and the following decay formula was used: $e = 0.99e$. 

Figure 5: Graphical depiction of the CAN_INTERCEPT assemblage

Figure 6: Graphical depiction of the NEAR assemblage

Figure 7: Convergence properties a training session with no obstacles. Top: Reinforcement received, Middle: Changes in Policy, Bottom: Q-value Table. The spike in the reinforcement received (Top) around 160 runs occurred because the Q-learner randomly chose faulty actions in two nearby simulations.
7.1 Policy Extraction

Once the Q-value table has stabilized, a policy for the robot can be extracted by choosing for each internal state the action with the highest associated Q-value. An example is seen below.

<table>
<thead>
<tr>
<th>Perceptual State</th>
<th>Action</th>
<th>Q-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near, CanIntercept</td>
<td>WAIT</td>
<td>0.51537</td>
</tr>
<tr>
<td></td>
<td>INTERCEPT *</td>
<td>4.69892</td>
</tr>
<tr>
<td></td>
<td>TERMINATE</td>
<td>0.53961</td>
</tr>
<tr>
<td>False, False</td>
<td>WAIT</td>
<td>0.50203</td>
</tr>
<tr>
<td></td>
<td>INTERCEPT *</td>
<td>0.01844</td>
</tr>
<tr>
<td></td>
<td>TERMINATE</td>
<td>0.57419</td>
</tr>
<tr>
<td>True, False</td>
<td>WAIT</td>
<td>1.39008</td>
</tr>
<tr>
<td></td>
<td>INTERCEPT</td>
<td>1.20292</td>
</tr>
<tr>
<td></td>
<td>TERMINATE *</td>
<td>6.58522</td>
</tr>
<tr>
<td>True, True</td>
<td>WAIT</td>
<td>1.48664</td>
</tr>
<tr>
<td></td>
<td>INTERCEPT</td>
<td>1.19925</td>
</tr>
<tr>
<td></td>
<td>TERMINATE *</td>
<td>11.4678</td>
</tr>
</tbody>
</table>

Table 1: The Values of the Q-table in the final session using 5% Obstacles. A (*) indicates the best action to choose for each internal state.

7.2 Successful Policies

In this task, two possible policies exist in which the anti-tank mine receives the reward. The first solution is to perform the WAIT action until the CAN_INTERCEPT trigger becomes true. At that time, the mine performs an INTERCEPT action until the NEAR trigger becomes true and then executes the TERMINATE action. The second possibly correct policy never performs the WAIT action. It immediately chooses the INTERCEPT action even though the nearest distance intercept point is unknown. When the INTERCEPT action is performed without a point of intercept the mine moves in a straight line along whatever direction it is currently facing. Then when the INTERCEPT point is identified, the state changes and the INTERCEPT action is chosen again. This is analogous to patrolling versus lying in wait.

Because both the CAN_INTERCEPT trigger and INTERCEPT behavior take some number of runs to initialize, it is actually faster to immediately move to intercept the tank, rather than wait for the trigger. This means that the second policy achieves success in a shorter number of time-steps by initializing both at the same time. So in the absence of noise the Q-learner should always converge to the second policy.

7.3 Convergence Metrics

The success of the Q-learner was judged by the convergence properties of the Q-value table. Several convergence criteria were used:

- reinforcement received vs. trial number
- number of policy changes
- convergence of the Q-table

The first metric plots the amount of reinforcement received in every learning scenario. In general, the robot should receive reinforcement more often as the training progresses if the behavioral mapping is converging to a successful solution. It is evident from both top graphs in Figures 7 and 8, that the robot is receiving reinforcement more often as its training proceeds. Both graphs
were smoothed with a running window filter to show the general learning trend. As seen in both figures, it is impossible to determine where or if convergence occurred, or to identify the exact difference in learning rates between training sessions. The graph oscillates too much due to variations in sensor readings, obstacles in the environment, or continuing exploration. Another metric is needed to determine stabilization.

The second metric monitors the number of policy changes over time. This metric is derived by looking at the Q-table at each time-step in the simulation. During each simulation, how many times does the policy defined by the Q-table change? The middle graphs of Figures 7 and 8 show the number of policy changes that occur during each session. The purpose behind the metric is to show the volatility of the Q-learner. If it is still spiking with large number of changes each round, it is less likely that the Q-learner is nearing convergence.

The third metric monitors the change in the Q-values as learning progresses. According to the policy changes metric, convergence occurs where the equivalent policy stops changing. The lower graphs in Figures 7 and 8 demonstrate this effect. At the point of convergence, several values in the table see a marked increase in value, separating them from the other values in the graph.

### 7.4 Convergence in the presence of Obstacles

To further test the convergence properties of the algorithm the task was repeated in the presence of environmental obstacles. The MissionLab Simulator has the ability to randomly generate obstacles and artificial worlds with 0% through 30% obstacle coverage were generated. Figure 10 shows a world with 10% obstacle coverage within the area of concern. The box drawn in the figure demonstrates the maximum dimensions of the obstacle field through which the anti-tank mine has to pass. It is designed to make certain that the mine cannot pass around the edges of the field to catch the tank.

The presence of obstacles slows down the movement of the anti-tank mine robot and in general it takes more time steps for the robot to reach the intercept point. The number of time steps, 300, may not be enough in

![Figure 9: Oscillation of two Q-values in the presence of obstacles which leads to destabilization of the number of policy changes convergence metric.](image)

![Figure 10: A sample mission with 10% obstacle coverage. The dotted line represents the bounding area of the obstacle field.](image)

![Figure 11: Evaluation of the convergence of the algorithm with increasing obstacle densities. The top plot shows an average over 8 trials for each obstacle density. The bottom plot represents the percentage of possible successful intercepts to obstacle density. As the obstacle density increases, the number of chances the robot has to successfully intercept the tank falls off with the percentage obstacle density.](image)
this case since the task has a time component and if the robot mine does not reach its goal in a certain time window it cannot receive any reinforcement (i.e., the tank will pass by and there won’t be a second chance to intercept it in this learning scenario). Therefore, the convergence rates are expectedly slower than in the case with no obstacles. Figure 8 shows the results for 15% obstacle coverage. Figure 11 shows the degradation in performance for the number of policy changes metric with the increase of obstacle coverage. After 8 trials for each of the obstacle densities, the average number of simulations required for convergence appears to rise exponentially with the percentage of obstacles in the environment. Obstacle percentages greater than 30% are not shown in this graph, because in all of the testing with greater percentages, the landmine never once received a reward for a successful intercept. Additionally, with 30% obstacle testing, the robot was rewarded only once throughout all of the testing.

As mentioned earlier, the competing solutions problem becomes critical to the stabilization of the Q-learner when obstacles are involved. The greater the number of obstacles, the more often that the chosen policy fails to receive a reward. In other words, the two competing solutions will continue to oscillate until the alpha value decays completely. This phenomenon is seen in Figures 8 and 9, and was typical to the results of training sessions with obstacles.

7.5 Simulating a Noisy Trigger

In this experimental setup, the impact of a faulty trigger on the learning process can be dramatic. The Q-learner only changes actions when the triggers change; so if a trigger misfires, then the next chosen action may be incorrect as well. This becomes a hidden state problem. If the CAN_INTERCEPT trigger returns false after returning true, then the Q-learner may randomly change actions in the middle of what would have been a successful intercept. Because the Q-learner does not recognize this hidden state, the correct policy is not being rewarded as often as it should.

By removing the filter from the CAN_INTERCEPT trigger, we can simulate a noisy trigger. The synchronization errors between the tank robot and the Q-learning mine will cause the CAN_INTERCEPT trigger to occasionally return the wrong answer. Like the obstacle scenarios, the faulty trigger will sometimes return true when the tank is not interceptible. In addition, the trigger will sometimes return false when the tank is in fact interceptible.

Figure 12 (Top) demonstrates the results of this testing. The actual convergence testing required a much longer test space than previously, 750 runs as opposed to 200. This reason is due in part to an increase in the alpha value from 0.97 to 0.99. The change in the alpha value was necessary to guarantee convergence with 0% obstacles, as was demonstrated in the previous section. With a lower alpha value, the noise in the trigger was enough to stop convergence in some tests. Even without the presence of obstacles, the simulation did not always converge to a successful solution.

Figure 12 (Bottom) illustrates this problem by looking at the number of policy changes at each simulation. At 190 simulated runs, the Q-table seems to have stabilized. However, with the presence of noise in the trigger, we don’t see the clean separation of values seen in the Q-value tables of Figures 7 and 8. Instead, the actual table is more like Figure 9. There is a separation of values for a short time, but the high noise levels prevent a complete stabilization of the Q-learner. In this case, the problem only occurs because of the two competing solutions. If we just looked at the outcome of the policy, i.e., whether the robot was attempting the correct actions to intercept the enemy, then the policy never changes to an incorrect policy after appearing to stabilize at 190 runs.

8 Real Robot Results

The real robot experimentation used a Nomad 200 with a top mounted SICK laser as the intelligent anti-tank mine. A rolling can that could be manually moved in a straight line with a rope and pulley system took the place of the tank. At each time-step, the readings from the SICK laser were analyzed to detect cylindrical objects with a 3-ft radius. This is the radius of the rolling
Figure 13: A successful interception: (a) Tank moves in from the left (b)-(e) Mine moves in straight line toward nearest distance intercept (f) Interception.

Figure 14: Attempted Intercept with Obstacles: (a) Tank moves in from the right; (b)-(c) Because the pillar obstructs the view of the tank, the mine robot stops moving; (d)-(e) The tank is now visible, and the mine robot resumes moving towards the nearest distance point of interception. (f) The tank escapes.
can, as well as the approximate radius of a Nomad 250, which was originally to be used as the tank object. Once the mine-robot decided to attempt an interception of the tank robot, it only had to move approximately 8 feet to reach the nearest distance point of interception.

To calculate intercceptibility, the simulation uses the tank position as reported by the MissionLab environment. Because the tank and the mine are not synchronized in the beginning, the reported position of the tank is not always up to date. However, the reported position is never off the actual path of the tank. With the real robot testing, the readings from a SICK laser mounted on top of the robot were searched for objects of the desired radius. This gave us the estimated position of the tank, relative to the current location of the mine. Unfortunately, because of the extra calculations required, the behaviors on the real robot have to work with data from previous times and extrapolate to the current position.

![Real Robot Q-value Table](image)

**Figure 15:** Resulting Q-value table of one set of robot testing. Notice how three of the values separate from the rest around the 12th simulation. This is where the Q-learner stabilizes.

The use of the SICK laser to find suspected tank positions required two modifications to the software used in simulation. First, the filters had to be adjusted in both the CAN_INTERCEPT and INTERCEPT algorithms. Where the CAN_INTERCEPT trigger already had a filter to account for the synchronization problem between tank and anti-tank mine, the number of points analyzed by the filter had to be increased to 10. This was particularly necessary because of the irregular velocity at which the tank object was manually moved across the room. Sudden changes in tank speed while the human operator changes hands cause the trigger to misfire, ultimately slowing down the learning process.

The adjustments to the INTERCEPT behavior were also necessary because of the imperfect line of travel. The tank object did not move in a straight line, as was done in the simulation. This was unintentional, but also a more realistic scenario than a perfectly straight path. To fix the problem of the moving point of INTERCEPT, the number of points through which a line was fitted had to be increased to 10.

The second adjustment to both routines had to be included for when the tank was not found. In the curve fitting routines, an estimated error was returned with the detected tank position. When the view of the tank became obstructed, or the tank moved too far away, the accuracy with which the position of the tank was reported decreased. By thresholding the error, false tank positions could be eliminated. However, this meant that the position of the tank was not always reported as done in the simulation. When this happens, both behaviors return the same result returned in the previous timestep. If this happens for 10 rounds, then the INTERCEPT behavior outputs a zero velocity, and the CAN_INTERCEPT trigger outputs false. All of the adjustments made to the behavioral routines on the actual robots were also tested in simulation.

After the differences between simulation and hardware were sorted out, the actual learning function was applied in the experiments. In the initial testing, a successful solution derived in simulation was tested on the robot. The tank-robot was manually moved across the room in a straight line, along a path the mine-robot could intercept. When the mine-robot determined that the intercept was possible, it moved in a straight line towards the intercept point. It then successfully terminated the enemy robot when it came in range. Both of the correct solutions appearing in simulation were tested in this manner. Figure 14 demonstrates this learned policy in the presence of obstacles. When the tank is not visible, the mine robot stops moving. When the tank reappears, the mine robot again moves toward the estimated nearest distance intercept point. Finally, the speed of the tank was increased so that the mine could not catch the tank. When that happened, the tank became non-interceptible and the mine robot stopped in place.

The second set of testing with the Q-learner was designed to show that learning on the real robot was equivalent to the learning in simulation. Using the improved version of the Q-learner, the intercept scenario was run 20 times on the robot on three separate occasions (Figure 13). This time the Q-learner was started from scratch, with no prior simulation experience. The mine-robot converged to a successful solution by the 15th run in all three tests. Figure 15 shows the properties of the Q-value table for one of the runs. Notice that the actions which result in a successful intercept have diverged significantly from the incorrect solutions. Despite the alpha value remaining relatively high, if the robot failed to receive a reward for a successful intercept then the Q-table values are still separated enough that the solution would not change.

### 9 Conclusion

This work shows that Q-learning is a valid approach for robot behavioral selection. In both simulation and on the real robot, we have demonstrated learning a successful policy given a limited set of behaviors and perceptual triggers. Additionally, the policies learned by the simulation could be directly ported to a robot platform with the equivalent behaviors and perceptual triggers. In other words, if all of the behaviors and triggers used by the Q-learner work in both environments, then the results of simulation testing, perhaps involving hundreds or thousands of trials, could be used for more complex tasks that would be impossible to learn on the robot itself.

In addition, we demonstrated the effects of environmental noise on the convergence rate of the Q-learner. We found that the number of simulations required for
convergence increases exponentially with the percentage of noise in the environment. Given that the percentage of noise was inversely related to the number of possible successful scenarios, the convergence rate increases exponentially as the number of possible successes decline.

However, noise in the environment does not have as dramatic an effect on the learning rate as noise from the sensors or perceptual triggers themselves. In the presence of a malfunctioning component, i.e., a trigger or sensor returning useless or faulty information, the Q-learner is not guaranteed to converge. Even when there exists a successful policy which ignores the malfunctioning component, the Q-learner was not guaranteed to converge. In fact, the possibility of multiple, competing solutions can do just the opposite, destabilizing the Q-learner handicapped with a noisy trigger.

10 Future Work

There are obviously several issues that remain to be addressed. The first question is the problem of complexity. As the number of states in the Q-learner increases, how can we guarantee that the Q-learners will converge? One possibility is to use the Q-learner incrementally, from simple to more complex, incorporating each learned task into a larger task description. Once a relatively simple task is learned, that solution can be saved as a behavioral assemblage to be incorporated into another Q-learner working on a more complex task.

Another line of research pursues the faulty component questions raised by the noisy trigger. This problem seems very similar to the hidden state problem commonly encountered by Q-learning researchers. Can the Q-learner learn to recognize the existence of a faulty trigger and use the alternative solution? Alternatively, if the failing always happens under the same conditions, can the Q-learner learn to ignore the failing? If the learner can learn to add new internal states at run time, as was done in the hidden state research by Chrisman [11], and McCallum [24], then the faulty trigger problem might be overcome.

Finally, a particularly important question to this research is the application of reward. When should continuous reward be applied versus a delayed reward? When should negative reinforcement be applied? As the noise in the environment, or with the components themselves, increases, a naive solution is to increase the reward so that the fewer successful experiences have a greater impact. However, this can also prevent a better solution from being learned later on in the testing. Preliminary work on this problem, especially with regards to negative reinforcement, shows that noise in general can have a great effect on performance with these alternative reward scenarios. Applying a penalty when a trigger has misfired, or obstacles in the environment are preventing success, can prevent the Q-learner from choosing that action ever again.

Acknowledgments

This research has been supported by DARPA under the Mobile Autonomous Robot Software (MARS) Program. The authors would like to thank Brian Ellenberger for his help with the implementation of an early version of the system.

References


A Q-learner Details

At the beginning of each training run, the table used by the Q-learner has to be initialized to some random set of values. Thereafter, the data is saved to a file at the end of each session, and reloaded at the beginning of each session. The alpha value is decayed once at the beginning of each session.

- If No Data exists from earlier sessions then initialize
  \[ Q_{value} = \text{rand(size([Number of States, Number of Actions]))} \]
  Initialize all values in \( Q_{value} \) table to \( 1 \leq \text{random} < 2 \)
  \( \alpha = 1.0 \)
  \( \gamma = 0.9 \)
  \( \text{Exploration Rate}, e = 0.5 \)
- Decay Alpha Value
  \( \alpha = 0.99^a \)

After the initialization has been performed, the Q-learner is active at every time-step. It is only queried and updated on time-steps where the internal state has changed or a reward is given. Otherwise, the action from a previous query is used until a new query is performed.

- Read Trigger
- Determine Internal State (Perceptual State)

\[
P_{state} = \begin{cases} 
00 & : \text{default} \\
01 & : \text{CAN\_INTERCEPT} \\
10 & : \text{NEAR} \\
11 & : \text{NEAR,CAN\_INTERCEPT} 
\end{cases}
\]

- Determine the Reward Value, for this state, action pair

\[
\text{Reward} = 0 \\
\text{for each Reinforcer do} \\
\text{if} \ [P_{state}, \text{last action}] = \text{[Reinforcer.state, Reinforcer.action]} \text{then} \\
\text{Reward} = \text{Reward} + \text{Reinforcer.reward}
\]

Query the \( Q_{value} \) table only if the state has changed, or the reward is not equal to zero. This sequence of actions is as follows. Choose the best next action, based on highest expected rewards, then update the table according to Watkins’s update formula. Finally, check the exploration factor.

\[
\text{if ( state has changed ) or ( Reward \neq 0 )} \\
\text{Determine best action, based on highest expected reward} \\
\text{Expected Reward, } V_{i} = \max_{i} (Q_{value}[P_{state}, i]) \\
\text{action} = \arg\max_{i} (Q_{value}[P_{state}, i]) \\
\text{Perform Watkins Update} \\
Q_{value}[lastState, lastAction] = (1 - \alpha) * Q_{value}[lastState, lastAction] \\
+ \alpha( \text{Reward} + \gamma V_{i} )
\]

- Decay the Exploration Rate

\[
\text{Exploration Rate, } e = 0.99^e \\
0 \leq \text{RandomNumber} < 1 \\
\text{if Chosen Random Number} \leq e \text{ then} \\
\text{Choose action randomly}
\]

After the Q-learner has finished, perform Action \( action \). If the Q-learner was not queried, then perform the action chosen in a previous round.
A.1 Q-learner Pseudocode

At every time-step do

If First Run then

\[
Q_{\text{value}} = \text{size}(\text{Number of States, Number of Actions})
\]
\[
\alpha = 1.0
\]
\[
\gamma = 0.9
\]
\[
e = 0.5
\]
\[
\text{action} = \text{WAIT}
\]

else

\[
\text{lastAction} = \text{action}
\]
\[
\alpha = 0.99\alpha
\]
\[
\text{Reward} = 0
\]

for each Reinforcer do

if \([P_{\text{state}}, \text{lastAction}] = [\text{Reinforcer.state, Reinforcer.action}]\) then

\[
\text{Reward} = \text{Reward} + \text{Reinforcer.reward}
\]

end {if}

end {for}

if (state has changed) or (Reward \neq 0)

\[
\text{Expected Reward, } V_z = \max_i (Q_{\text{value}}[P_{\text{state}}, i])
\]
\[
\text{action} = \text{argmax}_i (Q_{\text{value}}[P_{\text{state}}, i])
\]

\[
Q_{\text{value}}[\text{lastState}, \text{lastAction}] = (1 - \alpha) \times Q_{\text{value}}[\text{lastState}, \text{lastAction}]
\]
\[
+ \alpha (\text{Reward} + \gamma V_z)
\]

Exploration Rate, \(e = 0.99e\)
\(0 \leq \text{Random Number} < 1\)

if Random Number Chosen \leq e then

Choose action randomly

end {if}

end {if}

if lastAction = TERMINATE then

End Simulation
Reset Simulation Environment

end {else}

Perform Action action

end {do}
B Interceptibility
Interceptibility is determined at every time-step by the CAN_INTERCEPT trigger. If the tank is not within sensor range of the mine, then CAN_INTERCEPT returns false. To calculate the point of interception, the position of the enemy tank is recorded at 5 separate time-steps and a best-fit line drawn through them. Throughout the simulation, the intercept point is dynamically calculated from the previous 5 visible tank positions. The INTERCEPT behavior uses the same set of equations for dynamically determining the nearest-distance point of intercept, seen in the first half of the appendix.

B.1 Point of Interception
• Create the array of previous points. Initially, all positions returned by the sensor are vectors away from the current position of the anti-tank mine. These positions have to be stored using an absolute frame of reference. In this case, the starting position of the mine was used as the origin. Any origin point could have been used, as long as it remained constant throughout the simulation.

\[
X_{\text{position}} = [\text{last 5 recorded } X \text{ coordinates}]
\]
\[
Y_{\text{position}} = [\text{last 5 recorded } Y \text{ coordinates}]
\]

• Calculate average coordinates, and their cross-product information. The equations used here are the same as used in the Laserfit routines provided with MissionLab [12].

\[
\begin{align*}
t_z &= \sum_{i=1}^{n} (X_{\text{position}[i]} + X_{\text{ref}} - X_{\text{mine}}) \\
t_y &= \sum_{i=1}^{n} (Y_{\text{position}[i]} + Y_{\text{ref}} - Y_{\text{mine}}) \\
t_{xz} &= \sum_{i=1}^{n} (X_{\text{position}[i]} + X_{\text{ref}} - X_{\text{mine}})^2 \\
t_{yy} &= \sum_{i=1}^{n} (Y_{\text{position}[i]} + Y_{\text{ref}} - Y_{\text{mine}})^2 \\
t_{xy} &= \sum_{i=1}^{n} (X_{\text{position}[i]} + X_{\text{ref}} - X_{\text{mine}}) \cdot (Y_{\text{position}[i]} + Y_{\text{ref}} - Y_{\text{mine}})
\end{align*}
\]

• Normalize by the number of points, and output a direction in polar coordinates:

\[
\begin{align*}
m_x &= t_z/n \\
m_y &= t_y/n \\
S_{xz} &= t_{xz} - 2 \cdot m_x \cdot t_z + m_z \cdot m_x \cdot n \\
S_{yy} &= t_{yy} - 2 \cdot m_y \cdot t_y + m_y \cdot m_y \cdot n \\
S_{xy} &= t_{xy} - m_x \cdot t_y - t_z \cdot m_y + m_z \cdot m_y \cdot n
\end{align*}
\]

![Figure 16: Best fit line for a set of points](image)
\[
S_{diff} = S_{y1} - S_{x2} \\
\phi = 0.5 \cdot \tan(2(-2 \cdot S_{xy}, S_{diff})) \\
r = m_{x} \cdot \cos(\phi) + m_{y} \cdot \sin(\phi)
\]

At this point, the INTERCEPT behavior converts the polar output to cartesian coordinates, and outputs a vector in rectangular coordinate space.

**B.2 Time and Velocity Calculations**

To determine interceptability, the CAN INTERCEPT trigger still needs to calculate the enemy velocity, and distance to the intercept point. Then it can output a boolean value indicating whether the agent can reach the point of intercept before the enemy.

- Determine the tank velocity using position data. The following equation assumes that all measurements were made at equal time-steps. While not a perfect assumption in the Missionlab environment, it is a good estimate. All velocities are calculated in units of meters per time-step.

  \[
  V_{tank} = \sum_{i=1}^{n-1} \sqrt{(X_{position}[i+1] - X_{position}[i])^2 + (Y_{position}[i+1] - Y_{position}[i])^2} \over n - 1
  \]

- Calculate remaining distance to the intercept point from the tank’s current position.

  Remaining travel distance for mine = \(r\), (see Point-of-Interception calculations for details)

  Current tank distance from the mine

  \[
  dist_{n} = \sqrt{(X_{position}[n] + X_{ref} - X_{mine})^2 + (Y_{position}[n] + Y_{ref} - Y_{mine})^2}
  \]

  Travel Distance for the tank

  \[
  dist_{tank} = \sqrt{dist_{n}^2 - r^2}
  \]

- The direction of the tank’s movement can be determined using the difference between the first and last recorded vectors. If the tank is moving away from the intercept point, then the last recorded vector will be the largest.

  Oldest stored tank position with reference to anti-tank mine position,

  \[
  dist_{t} = \sqrt{(X_{position}[1] + X_{ref} - X_{mine})^2 + (Y_{position}[1] + Y_{ref} - Y_{mine})^2}
  \]

  Direction of the tank’s movement, in terms of moving towards or away from the point of interception

  \[
  dir_{tank} = \begin{cases} 
  0 & \text{if } dist_{t} - dist_{n} < 0 \\
  1 & \text{if } dist_{t} - dist_{n} \geq 0
  \end{cases}
  \]

- Compare the time for each robot to reach the point of interception in order to output TRUE or FALSE for the CAN INTERCEPT trigger.

  Anti-tank Mine Velocity, \(V_{mine} = 0.1\), given as a function parameter

  \[
  \text{if } \left( \frac{dist_{tank}}{V_{tank}} \right) > \left( \frac{1}{V_{mine}} \right) \text{ and } (dir_{tank} > 0) \text{ then}
  \]

  \[
  CAN \_INTERCEPT = TRUE
  \]

  \[
  \text{else}
  \]

  \[
  CAN \_INTERCEPT = FALSE
  \]
C AVOID_OBSTACLE Pseudocode

Function INPUT

\[
safety\_margin = \text{Margin of safety around obstacle edge} \\
sphere = \text{size of area where the obstacle affects the robot}
\]

Function DEFINITION
At every time-step do

\[
\text{OutputVector} = 0 \\
\text{for each Detected Obstacle do} \\
\text{RepulsionVector} = 0 \\
C\_Dist = \text{Distance from Robot Center to Obstacle Center} \\
\text{Radius} = \text{ObstacleRadius} + \text{safetyMargin} \\
\text{if } C\_Dist \leq \text{Radius} \text{ then} \\
\quad \text{magnitude} = \infty \\
\text{else if } C\_Dist \leq (\text{Radius} + \text{Sphere}) \text{ then} \\
\quad \text{magnitude} = (\text{Sphere} - (C\_Dist - \text{Radius}))/\text{Sphere} \\
\text{else} \\
\quad \text{magnitude} = 0 \\
\text{end } \{\text{else}\} \\
\text{if } \text{magnitude} \neq 0 \text{ then} \\
\quad \text{RepulsionVector} = \text{Unit Vector pointing away from obstacle} \\
\quad \text{RepulsionVector} = \text{magnitude} \times \text{RepulsionVector} \\
\quad \text{OutputVector} = \text{OutputVector} + \text{RepulsionVector} \\
\text{end } \{\text{if}\} \\
\text{end } \{\text{for}\} \\
\text{return OutputVector} \\
\text{end } \{\text{do}\}