

Learning to Race: Experiments with a Simulated Race Car

Larry D. Pyeatt Adele E. Howe
Colorado State University
Fort Collins, CO 80523
email: {pyeatt,howe}@cs.colostate.edu
URL: <http://www.cs.colostate.edu/~{pyeatt,howe}>

Abstract

Our focus is on designing adaptable agents for highly dynamic environments. We have implemented a reinforcement learning architecture as the reactive component of a two layer control system for a simulated race car. We found that separating the layers has expedited gradually improving performance. We ran experiments to test the tuning, decomposition and coordination of the low level behaviors. Our control system was then extended to allow passing of other cars and tested for its ability to avoid collisions. The best design used reinforcement learning with separate networks for each action, coarse coded input and a simple rule based coordination mechanism.

Introduction

Autonomous agents require a mix of behaviors, i.e., responses to different stimuli. This is especially true in situations where there are other agents present or where the environment is otherwise non-deterministic. For an agent to be effective in its environment, it must have a large repertoire of behaviors and must be able to coordinate the use of those behaviors effectively. Reactive systems have been favored for applications requiring quick responses, such as robotic or process control applications (Arkin 1994). Unfortunately, it is difficult to tune, decompose and coordinate reactive behaviors while ensuring consistency.

In this paper, we present a case study of designing a mostly reactive autonomous agent for learning to perform a task in a multi-agent environment. The agent controls a race car in the Robot Automobile Racing Simulator (RARS) system (Timin 1995). To be successful, the agent must quickly respond to its environment and must have mastered several skills: steering, accelerating, and passing. To address these requirements, our basic agent architecture (Figure 1) adopts the common two-layer control structure: low level behaviors (implemented

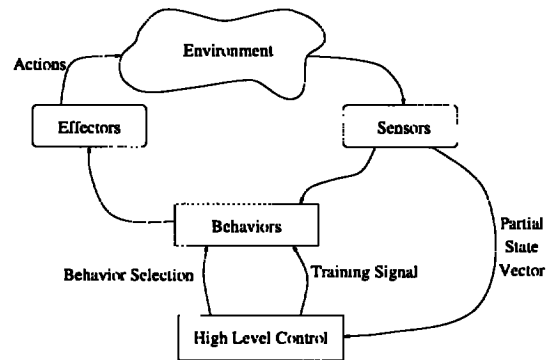


Figure 1: Agent architecture.

as reinforcement learning neural networks) and a high level control mechanism. The agent may have several behaviors which it has learned, but only one may be active at any time. The active behavior receives inputs from the environment and sends control signals to the effectors. The high level control mechanism selects which behavior is active and assigns reinforcements. This architecture is extensible and provides a means to let the agent learn by itself with no other agents in the environment. Once the agent has learned its "survival" behavior, new behaviors can be added for interaction with other agents.

Controlling Reactivity

Reactivity relies on the synergistic composition of low level behaviors. Authority for composing behaviors may be resident in a deliberative system. In Cypress (Wilkins *et al.* 1994), the planner is responsible for high level operation, leaving the details to an execution module. Control plans may direct the activation of situation-triggered behaviors (Hayes-Roth 1995). The Autonomous Robot Architecture used a deliberative planner to configure a modular reactive control system (Arkin 1990).

Because of the fast response and built-in tuning capabilities, neural network components have been used as both the controller and the behaviors. For example, a neural based robot controller showed better adaptability and noise rejection than a standard model based control algorithm (Poo *et al.* 1992).

Reinforcement learning of individual behaviors in a robot can sometimes outperform a hand coded solution (Mahadevan & Connell 1992). Dorigo and Colombetti (Dorigo & Colombetti 1994) used reinforcement learning to shape a robot to perform a predefined target behavior. They trained the agent to display five different behaviors and then defined four ways to combine these behaviors.

Simulated Automobile Racing

RARS simulates the activities of up to 16 agents competing against each other in automobile races. Each agent is implemented as a subroutine which is compiled into RARS. At each time step, RARS updates the position and velocity of each car. It then calls the subroutine for each agent in the race. Each agent routine receives a structure containing information about position, velocity, current speed, distance to the end of the current track segment, curvature of the track, and relative position and velocity of nearby cars. The agent routine calculates command signals for speed and steering, which are returned to RARS.

Several agents are included in the standard RARS distribution. These agents were contributed by programmers from all over the world. At this time, all of the agents are heuristic, and none have the capacity to learn. However, some of the agents exhibit quite good performance.

Low Level Behaviors

In our architecture, each low level behavior can be implemented as either a heuristic control strategy or as a reinforcement learning network similar to those used by Barto, Sutton and Watkins (Barto, Sutton, & Watkins 1989), Anderson (Anderson 1989) and Lin (Lin 1992). Behaviors are implemented as heuristic control strategies when the target behavior is simple to program or as a placeholder during testing and development. For the learned behaviors, we used a popular reinforcement learning strategy known as Q-learning.

Q-learning networks

Our implementation of Q-learning uses the standard back-propagation learning algorithm with the addition of *temporal difference* methods (Sutton 1988) to learn $Q(s, a)$, the value of performing action a in state s . In standard back-propagation, the error term for the output node is calculated by subtracting the actual output of the network from

the desired output. For temporal differencing, the error term is calculated as: $e_{t-1} = \delta o_t - o_{t-1} + \tau$ where o_t is the output of the network at time t , o_{t-1} is the output of the network at the previous time step, and τ is the reinforcement signal. δ is a scaling parameter between 0 and 1.

Q-learning uses a single network to select actions and estimate the value of those actions. The neural network has one output for each possible action. At each time step, $Q(s, a)$ for all a is calculated and the a with the highest value is chosen probabilistically.

Our implementation of Q-learning uses truncated returns and a TD(λ) approach. It stores the previous N states in a queue and performs learning on the N th previous state. The return R_{t-N} to state s_{t-N} is calculated by applying the following recursive equation N times:

$$R_{t-1} = \gamma \lambda R_t + \tau_t + \gamma(1 - \lambda)Q(s_t, a_t)$$

where R_t is the truncated return to state t of future rewards, τ_t is the reward given at time t , and $Q(s_t, a_t)$ is the value of the state action pair chosen at time t . γ is a decay term which discounts future returns. λ is a parameter which determines what proportion of the return is made up of future returns versus future state action values. Given R_{t-N} , the value of the state action pair $Q(s_{t-N}, a_{t-N})$ is updated using

$$\Delta Q(s_{t-N}, a_{t-N}) = \alpha [R_{t-N} - Q(s_{t-N}, a_{t-N})].$$

$\Delta Q(s_{t-N}, a_{t-N})$ is used to train the neural network output unit corresponding to the action a_{t-N} . The other neural network outputs are not trained for that time step.

Heuristic Control Strategies

The core control strategy for this domain is *race*. The *race* strategy must coordinate two complex interacting behaviors: *steering* and *accelerating*. The *race* strategy begins with no knowledge and learns purely by trial-and-error. This results in a lot of crashes while the neural networks are being trained. The system uses a heuristic control strategy to recover from crashes. The *recover* strategy is implemented as a set of simple rules. Another control strategy, *pass*, directs actions needed to pass another vehicle.

High level control

The purpose of high level control is to coordinate the behaviors so as to optimize speed and competitive advantage while maintaining safety. Coordination is achieved through two mechanisms: strategy selection and reinforcement. The strategy selection mechanism determines which behaviors are active at any given point, meaning that their control signals are passed to the simulator. The reinforcement signal coordinates the learning of separate networks so that their actions work together rather than clash.

Strategy Selection Mechanism

Two distinct control signals are required for driving the car: throttle and wheel. We divide each control signal into three possible actions. For throttle, the three possible actions are: accelerate by 10 ft/sec, remain at the same speed, and decelerate by 10 ft/sec. The possible wheel commands are: adjust steering left by 0.1 radians, do not adjust steering, or adjust steering right by 0.1 radians. Limiting the set to three possible actions limits the number of possible actions and, therefore, limits the control functions which must be learned.

At this stage, our strategy selection mechanism is quite simple. Strategy selection is a pre-defined heuristic which switches control based on the programmer's knowledge of when to switch.

Reinforcement signal

Without a good reinforcement signal, the reinforcement learning networks may not provide the correct response or may never converge. We used two different reinforcement signals during our system development. The reinforcement signal for *acceleration* is zero at all times unless the car is off the track or going less than 15 mph, in which case the reinforcement is -1 . Also, a reinforcement signal of 1 is given at the end of each lap. This function provides negative reinforcement for leaving the track and for going slowly. The slight bias in the network obviates the need for positive reinforcement.

The reinforcement signal for *steering* is 0 at all times unless the car is off the track, in which case the reinforcement is -1 . Also, a reinforcement signal of 1 is given at the end of each lap. Since the steering behavior is not directly responsible for the speed of the car, the negative reinforcement for going too slowly is not included in the reinforcement signal. However, the steering behavior can effect the time that it takes to complete a lap. For instance, by steering to the inside on curves, the steering behavior can reduce the distance traveled on a particular lap. For this reason, we include the reinforcement signal for each lap.

Experiments

We performed two experiments to test our design decisions. In both of them, the reinforcement schedule and action signals remained the same. Each network was tuned, trained and tested on a single track in RARS. For the first experiment, performance was measured in length of time needed to learn and the failure (leaving the track) rate after a certain number of races. For the second experiment, we substituted "damage" as the primary evaluation measure because that is the most direct measure of what we were trying to improve. In both experiments, we also measured the number of time steps required to complete each lap.

Experiment 1: Tuning Learning

We chose to implement separate networks for each possible action. This approach requires each network to learn the utility for a single action. Our pilot tests indicated that this approach was superior to using a single network with multiple output units. After the pilot tests, the agent was able to drive around the track, but still performed poorly when compared to all of the agents in the RARS distribution. Our next experiment investigated whether tuning of the learning networks could significantly improve performance; we changed the input representation and reinforcement protocol.

In the pilot tests, the networks were given car direction as a continuous input signal. Now, car direction is represented by a 21 bit vector, in which one of the bits is set to one to indicate direction while all other bits are set to zero. The other input signals were treated similarly. This technique is sometimes referred to as coarse coding.

The second change was to use *truncated returns*. The states are stored, and training is actually performed on the state that happened 15 time steps in the past, using information obtained from the 14 states which followed it. This allows us to compute a more accurate value for the return and causes the algorithm to converge more quickly.

Tuning the learning significantly improved performance. As shown in Figure 2, the new approach learns to avoid failure much more rapidly. The time to complete a lap is much lower with the new approach and is competitive with some of the heuristic control strategies which are supplied with RARS. Some of the best heuristic strategies can complete a lap in under 600 time steps on average while the worst heuristic strategies take more than 800 time steps. After sufficient training, the Q-learning approach can complete a lap in about 700 time steps on average.

Experiment 2: Adding a new behavior

We then began racing our agent with other cars on the track at the same time. Although it performed well when it was the only vehicle on the track, it did not have any strategy for avoiding other vehicles. RARS detects collisions between cars and calculates damage when a collision occurs. When enough damage points have been given to a car, it is removed from the race. To solve this problem, we added a behavior to direct the interaction with other agents on the track and allow it to safely pass the other cars.

When the high level control mechanism detects a car ahead, it gives control to *pass*, which steers around the other vehicle. *Pass* is given 15 time steps to move around the other car. If the car leaves the track or runs into the other car before the 15 time steps have expired, a reward of -10 is given,

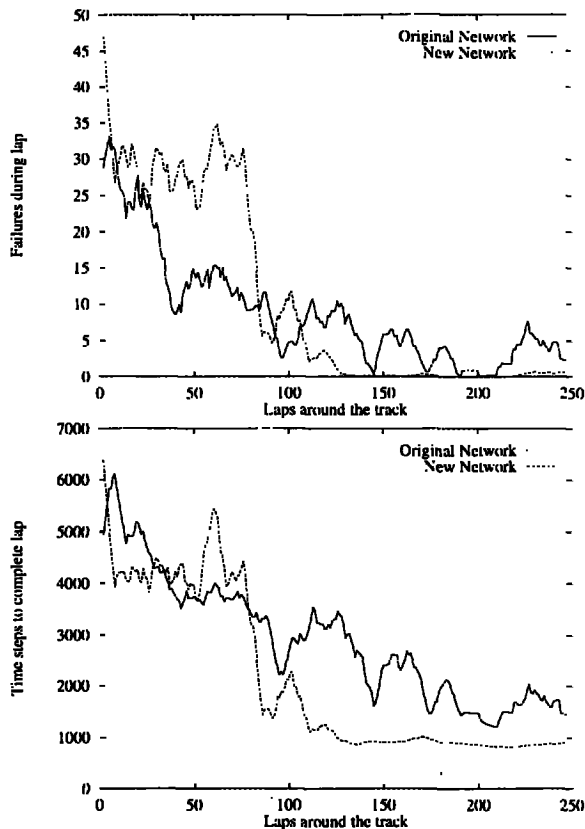


Figure 2: Learning performance for the improved learning strategy.

and the recover behavior is given control. Otherwise, the pass is given a reward of 1, and the race behavior is given control.

For training, three slow moving cars were created: blocker1, blocker2, and blocker3. These three cars were derived from the simple "cntrl0" car which comes with RARS. Blocker1 was adjusted to stay with 25% of the track to its left. Blocker2 was set to stay at the center of the track, and Blocker3 was set to stay with 75% of the track to its left. The blocker cars were set to run at 35 miles per hour; the average speed of a car in the RARS distribution is about 65 miles per hour. If only one blocker was used, the normal steering and accelerating behaviors would learn to stay in a part of the track where the blocker was absent, reducing the need for a passing behavior. Using three blocker cars forced our agent to make passing maneuvers.

The race behavior was trained alone on the track for 1000 laps, and then the pass behavior was initialized and trained on the track for 1000 laps with the three blocker cars. We saved the neural network weights and ran 100 races against the blocker cars. Each race ended when the slowest moving car completed 2 laps.

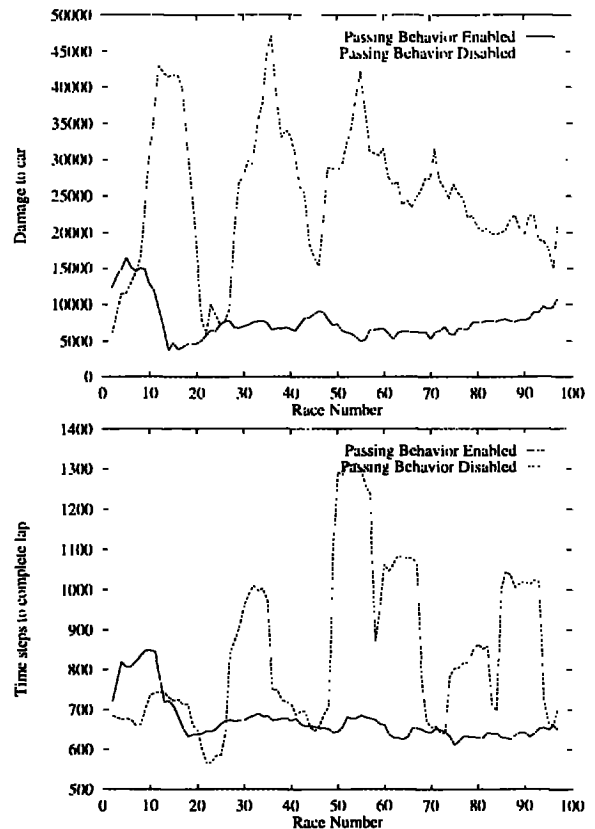


Figure 3: Using the passing behavior to improve performance.

The passing behavior greatly reduced the amount of damage and decreased the number of time steps required to complete a lap. Figure 3 shows the damage accrued by the car and the time steps required to complete a lap during the test races with and without the passing behavior enabled. The average speed for the car was 63.5 miles per hour, slightly slower than the 65 miles per hour average speed for cars in the RARS distribution, but a considerable improvement over earlier performance.

What We Learned

The long term goal of this research is to develop a system that is capable of generating new low-level behaviors in order to achieve its goals. The agent should be able to determine when a new low-level behavior is needed and to generate specifications for that behavior. The agent will add a new behavior module and use reinforcement learning to train it to match the new behavior specifications. The resulting behavior will be added to a repertoire of behaviors available to the high level controller.

The research described in this paper is a first step towards an agent that will control its own development. From the first experiment, we found that

reinforcement learning with neural networks can be made to work in this domain, but requires careful tuning of parameters. This is a disappointment because it means that for an agent to build its own low level behaviors it will need both considerable time to tune them and knowledge about how to do so. We are currently exploring other function approximation techniques to use with reinforcement learning, such as decision trees and CMACs.

The second experiment demonstrated that we, the human developers, can easily add new behaviors to the behavior set. The passing behavior was incorporated by creating a new network with an identical input representation and learning algorithm to the other behaviors. A reinforcement schedule was designed for the passing behavior, and a switch added to the heuristic control selection mechanism. The passing behavior was trained together with the normal steering and accelerating behaviors.

The "final agent" uses Q-learning with separate networks for each of three behaviors, coarse coded input, and a coordination mechanism combining simple rule-based selection of behaviors, reinforcement signals and synchronized training. This best agent differs significantly from the pilot agent that we built. By incrementally testing the design decisions, we developed an agent that is well suited to its environment. We also gained considerable insight into what will be required to automate the process and allow an agent to manage its own development.

Directions for Future Work

With the passing behavior in place, the agent learns to safely pass other cars, but can still get into situations which cause it to collide. For instance, if another car is overtaking ours, our agent does not avoid the other car. To reduce the damage from this type of collision, an avoidance behavior can be added just as the passing behavior was.

The agent can be further improved by finding a more appropriate function approximation technique. Neural networks do not perform localized learning. When the network adjusts the value of one state, the weight change may affect other, very different states. This is an undesirable feature in reinforcement learning. Table lookup methods provide localized learning with guaranteed convergence. However, table lookup does not scale well with the size of the input space. We need a good function approximation technique which combines the scalability of neural networks with the localized learning of table lookup methods.

We are extending the ideas proposed in this paper to a more general robot control architecture. The new system uses some higher level planning and can create and manage new low level behaviors. This new system is aimed at a more general

mobile robot platform, where more high level decision making is necessary and the robot's goals are more complex than in the RARS domain.

Acknowledgements

This research was supported in part by NSF Career Award IRI-930857. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein.

References

- Anderson, C. W. 1989. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, 103-114.
- Arkin, R. C. 1990. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems* 6:105-122.
- Arkin, R. C. 1994. Reactive robotic systems. available at <http://www.cc.gatech.edu/ai/robot-lab/mrl-online-publications.html>.
- Barto, A.; Sutton, R.; and Watkins, C. 1989. Learning and sequential decision making. Technical report, COINS Technical Report 89-95, Dept. of Computer and Information Science, University of Massachusetts.
- Dorigo, M., and Colombetti, M. 1994. Robot shaping: developing autonomous agents through learning. *Artificial Intelligence* 71(2):321-370.
- Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. *Artificial Intelligence* 72(1-2):329-365.
- Lin, L.-H. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning* 8(3/4):69-97.
- Mahadevan, S., and Connell, J. 1992. Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence* 55(2/3):311-365.
- Poo, A.; Jr., M. A.; Teo, C.; and Li, Q. 1992. Performance of a neuro-model-based robot controller: adaptability and noise rejection. *Intelligent Systems Engineering* 1(1):50-62.
- Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9-44.
- Timin, M. E. 1995. Robot Auto Racing Simulator. available from <http://www.ebc.ee/~mremm/rars/rars.htm>.
- Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1994. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI* 7.