How to Make Software Agents Do the Right Thing: An Introduction to Reinforcement Learning

Satinder Singh Peter Norvig David Cohn

Adaptive Systems Group
Harlequin Inc.

July 5, 1996

1 Doing the Right Thing

You're about to let a "spider" loose on the Internet. How do you know if it will seek out the information you want, without disrupting the net? You're in charge of writing the scheduling algorithm for a bank of elevators. How do you know when to go up and when to go down? In general, how do you make software do the right thing for its users? How do you even know what the right thing is?

Many people see **agents** and agent-based programming ushering in a new era in computing, particularly in the environment of the internet. The optimists believe that all the protocols for data transfer, encryption, security, and payment will be sorted out in a year or so, and we can then go about writing new and exciting agent-based applications. This article explains why programming agents is not just business-as-usual; rather it requires a new way of looking at problems and their solutions.

When you hire a human agent to do something for you, you rarely spell out a detailed plan of action. Instead, you define the state of the environment that you want to achieve (e.g., you tell a contractor that you want a new front porch with comfortable seating, for under $2000). In more complex and uncertain situations, you specify your **preferences** rather than stating outright goals, as when you tell a stock broker agent that the more money you make the better, but count capital gains as, say, 30% better than dividend income. Your hired agent then takes actions on your behalf, even negotiates with other agents, all to help you achieve your preferences. We would like our software agents to behave the same way.

That means we will need a way to describe our preferences to software agents, and a methodology for building agents that best satisfy our preferences. The pleasant surprise is that for many problems, once we know the preferences, we're almost done! Given the preferences, a list of possible actions, and enough time to practice taking actions, we can apply the formalism of **Reinforcement Learning** (or RL) to build an agent that acts according to the preferences in a near-optimal way. This article shows how.

2 The Elevator Problem

As an example, consider the problem of programming a bank of elevators (borrowed from Crites and Barto [3]). Consider a ten floor building with three elevators (Figure 1). What program should we write to control them? By this, we don't mean what programming language, or what hardware configuration we should use — though those are certainly important questions. Instead, we mean the more fundamental question of how we approach the programming task.

We will think of this as an agent-based program. Once every, say, 1/2 second, the program will examine the information available to the elevator sensors, and decide what actions the elevators should take. The sensor information or **percepts** include the buttons which are lit on each floor and inside each elevator. (Some more advanced elevators can detect the weight of the passengers in each car and the breaking of an electric eye beam at each door, but we'll assume our elevators do not.) An empty elevator can either stop, go up, or go down. One constraint, driven by passenger expectation, is that a non-empty elevator must continue in its current direction and stop at all the floors requested by its passengers before it reverses directions. We assume that we don't need to explicitly control the doors: they will always open when the elevator is stopped at a floor with a corresponding lit button, and close before the elevator moves.

We could use separate agents for each elevator, but for the moment let us assume that there is one central program. On each iteration, the program receives as percepts an *on* or *off* signal for each button, and the current floor and direction of each elevator. The program's output is a list of actions, one for each elevator, e.g., [*up, stop, up*], that obey the constraints stated above.

In most complex problems, the percepts alone are not enough to decide on good actions. For example, it is useful to know how long a passenger has been waiting for an elevator to arrive. Since this is not available from the percepts, the system will need to have some memory. Let us use the term **state** to denote the available information, both percept and memory (in control theory, "state" implies certain technical constraints which we ignore for simplicity; see Barto, Bradtke and Singh [1]). The program's behavior is a feedback-loop (Figure 2). At each time step, the program has access to the current state of the system and has to decide what action to execute. After each action, the system moves to a new state, and the decision process is repeated. Technically, we say that

a mapping from the space of states to the space of actions is a **policy**. Our problem is that of finding an optimal or near-optimal policy.

What is optimal? There are many criteria we could consider: the number of passengers delivered to their destination, the time a passenger waits for an elevator to arrive, the time spent inside the elevator, the cost of the power consumed by the system, the wear and tear from moving and reversing directions, etc (or the resulting time and money spent on maintenance). Typically there are trade-offs among these; reducing wear and tear may increase wait time. Some criteria are best thought of as **costs** to be minimized while others are **rewards** to be maximized, but it will simplify things if we treat them all as rewards, with the understanding that a cost is treated as a negative reward.

We can then define an optimal policy as one that maximizes the expected sum of the rewards it receives over its lifetime. (Here, **expected** means that we have to average over all sources of randomness in the system; in complex systems the reward you get often depends on many random factors beyond the agent's control. In the elevator system, the number of passengers and their arrival and destination floors are random.)

Let's assume we want to minimize two quantities: the time spent by passengers waiting for an elevator, and the distance traveled by the elevators (which effects both energy and maintenance costs). To put this into the language of rewards, we need a common scale for the two quantities. For concreteness, let's say that on each time step we will receive a reward of -50 for every floor that has passengers waiting, and -1 for every elevator in motion.

In the traditional approach to this problem, the programmer would come up with a set of rules and procedures on when to take what action based on common-sense or rough heuristics. There are three drawbacks of this approach. First, it requires a lot of work on the part of the programmer. Second, when the environment changes in some way (perhaps faster elevators are installed), a programmer needs to be called in again. Third, there's no way to guarantee that this will lead to an optimal policy; sometimes common sense is just wrong. Even if the rules and procedures are quite sensible in general, they would not pay any attention to the traffic patterns of a *particular* building, which means they might be missing an opportunity for fine-tuning.

3 The Reinforcement Learning Approach

Reinforcement Learning (RL) corrects all three drawbacks of the traditional approach. First, RL requires very little programmer effort, because most of the work is done by a process of automatic training, not programming. Second, if the environment changes, the training can be re-run, without any additional programming. In fact, retraining can be done continuously, online, as the system runs. Third, RL is mathematically guaranteed to converge to an optimal policy (given certain assumptions described later).

What does an agent need to know in order to take optimal actions? Clearly, it would be a big help if the agent could predict the **reward function**, the reward it would receive from taking a given action in a given state. But that would not be enough. Consider an empty elevator that is approaching the seventh floor on its way up, and assume that there are passengers waiting to go down on the seventh, eighth, and ninth floors. Stopping at the seventh floor would give a larger immediate reward than either continuing on up or reversing direction. But if we did stop at the seventh floor, we would have to take the passenger there down, and return later for the other passengers. A wiser strategy—one with a better long-term reward—would be to bypass the seventh and eighth floors and pick up the passengers on the ninth floor first and then go down, picking up the passengers on the eighth and seventh floors on the way.

The problem is that the reward function just captures the immediate or short-term consequences of executing actions. What we need instead is a function that captures the long-term consequences. Such a function is called a **utility function.**

Intuitively, the utility of taking action a in some state s is the expected immediate reward for that action plus the sum of the long-term rewards over the rest of the agent's lifetime, assuming it acts using the best policy. Equations (1) and (2) in Figure 4 formally define the utility function.

If we knew the utility function, then the optimal policy would be to enumerate all possible actions and choose the action with the highest utility. Solving the elevator problem therefore reduces to learning the utility function. Theoretically, it is possible to exactly solve for the utility function. Equation (3) in Figure 4 shows how the utility of a state is related to immediate reward and the utility of the resulting state. If there are n actions and m states, this defines a system of nm nonlinear equations in nm unknowns, which can be solved by standard methods, such as dynamic programming (Bertsekas [2]). In practice, however, this rarely works.

One problem is that our simulation model of the elevator system consists of probability distributions for passenger arrivals and destinations, etc. In order to compute the expectations in Equation 4 to do dynamic programming, we must first convert our simulation model into a model of state transition probabilities. Doing so is computationally expensive. Another problem is that in our elevator example the state-transition probabilities are continuous, so even if we had them, computing the expectations would be very expensive. Finally, Equation 4 requires an iteration over the entire state space of the system. Our elevator example has more than 10^{22} states — iterating over all of them is impossible. If we did not have a simulation model to begin with, doing dynamic programming would first require estimating a model by experimenting with the real elevator system.

Reinforcement Learning is an iterative approach that eliminates these problems to comes up with an approximate solution to the utility function. The

basic idea is to start with some initial guess of the utility function, and to use experience with the elevator system to improve that guess. To gain experience, one needs either a simulated elevator system or the ability to select actions and acquire data from the real elevator system. In either case, the experience that the program acquires is a state-action trajectory with associated rewards (see Figure 3). On each step through the trajectory, the agent takes some action and receives some reward. The reward is information that it can use to update its estimate of the utility function, according to Equation (5) in Figure 4. By focusing computation on the states that actually occur in the simulation, RL is able to deal with larger state spaces than dynamic programming methods. As the program gains more experience, the difference between its estimated utility function and the true utility function decreases.

We are finally able to say exactly how RL allows us to write an elevator control program:

(1) Code up a simulation of the elevator system. Make some measurements or estimates of passenger arrival rates and other parameters, and write a program to simulate the elevator environment.

(2) Determine the reward function.

(3) Make an initial guess at a utility function. This can be based on simple rules like "it is better to move up if there is a passenger waiting above." The details are not important, because the guess will be improved.

(4) Run the simulation, updating the utility function on each step using Equation (5). Given enough time to explore in the simulated system, our program will converge to a near-optimal policy.

(5) The final version of the program can then be used to control the real elevator system. If desired, the program could continually monitor its performance and update its utility function to reflect changes in the real environment.

3.1 Issues

What type of problems are appropriate for RL? In a recent paper, Crites and Barto [3] have applied RL to the elevator problem defined above and shown that it produces significantly better performance than the best solutions previously available. RL has also been successfully applied in many areas, including *process control, scheduling, resource allocation, queuing, and adaptive games.* For example, we have applied RL to the problem of channel assignment in cellular telephone systems and shown that it yields better performance than previously available solutions (see Singh and Bertsekas [5]). Zhang and Dietterich [7] have shown similar results in a job-shop scheduling problem, and Tesauro [6] has used RL to develop the world's best computer backgammon player, which is nearly as good as the human champion. Numerous other applications are being presented at this year's Machine Learning and Neural Information Processing Society (NIPS) conferences.

When is convergence to an optimal policy guaranteed? Technically, RL will converge for all finite stationary Markov environments. What does that mean? First, it may not work if there are an infinite number of states, because it won't be able to explore them all. Second, it may not work if the environment is constantly changing, although in practice, small slow changes in the environment are accommodated well. Finally, the mathematics break down if the result of an action depends on states other than the current state. These limitations only affect the guarantee of convergence; there are many instances of RL working well despite violating these conditions. These limitations are actively being addressed in current research.

Is it sensible to treat all preferences as numeric rewards on a single scale? Theoretically, yes. There is a theorem (North [4]) that if you believe four fairly simple axioms about preferences, then you can derive the existence of a real-valued utility function. (The only mildly controversial axiom is substitutability: that if you prefer A to B, then you must prefer a coin flip between A and C to a coin flip between B and C.) Practically, it depends. Users often find it hard to articulate their preferences as numbers. (Example: you have to design the controller for a nuclear power plant. How many dollars is a human life worth?)

How should the program store the utility function? The utility function maps state-action pairs to real numbers. If the size of the state-action space is small enough, this function can be stored in the form of a table; otherwise, some form of compact function approximation is used. Statisticians have a variety of representations for this purpose: decision trees, polynomial approximations, neural networks, etc. Using such a representation not only saves space, it also gives us **generalization**: the program can take a sensible action from a state it has never seen before by interpolating or extrapolating from known states.

How much information should be kept as state? In some situations, there may be too many sensors, some of them giving redundant information. Too much information increases the size of the state space needlessly, and makes learning the utility function slower. Progress is being made in techniques to detect and remove redundancy. In other situations, there may be too few sensors. Too little information can prevent learning from making much progress. Memory can be used to augment the limited information in these situations.

What if you don't have a simulation of the elevator problem available? You could use state trajectories obtained by controlling and experimenting with the real elevator system to train the RL solution. In this case, methods for optimal experiment design (in choosing what actions to take) are important, because real life elevators run so much slower than simulations, and because we don't want to alienate passengers as the system is being trained.

How should agents communicate and interact? We designed the elevator solution as a single agent that runs all three elevators. But consider the problem of controlling cars on a freeway. A central controlling agent would be

unmanageable — we would need to use many interacting agents instead. Game theory and economic market theory can be used to extend the RL algorithm to handle these multi-agent situations.

So RL says I should always take the action with the highest estimated utility, right? Actually, no. If you know the true utility function, you should always take the action that maximizes it. But the estimated utility may be wrong. Taking non-optimal actions may get you "off the beaten track" enough to learn something new, thereby updating the estimate, and enabling much better actions in the future. So there is always a trade-off between exploitation of the best known action and exploration of the consequences of other actions.

4 Conclusion

The increasing complexity of the problems being tackled with software systems means that in many cases it is no longer cost-effective (or even possible) for the programmer to prescribe the best action in every state of the environment. To meet these new challenges, not only do we need better programming tools and protocols — we need a new way of thinking about problem solving. Just as when we hire human agents to solve problems for us, we want to specify only high-level preferences to our software agents, and have them figure out the best course of action. This is not only more cost-effective, but also leads to better and more robust performance in real-world, changing, and uncertain environments. In this article we have presented a brief introduction to reinforcement learning, a problem solving method that enables software agents to learn near-optimal solutions to complex problems from a specification of high-level preferences.

Sidebar: Genetic Algorithms

The field of **Genetic Algorithms** (GA) addresses the same set of problems as RL, but with a different methodology. Instead of learning a utility function, Genetic Algorithms learn the policy directly, representing it as an encoded bit string (or, in **Genetic Programming**, as a program in some computer language). The approach is to search the space of policies by generating a pool of policies and evaluating each one for a sequence of time steps in a simulated environment. A new pool of policies is constructed by combining the more successful existing policies, along with some random change. The algorithm iterates until a sufficiently good policy is found.

The name "Genetic Algorithms" comes from an analogy with biology: the bits representing a policy are like the bases in a strand of DNA, the creation of new policies is like the process of sexual reproduction, the random change is like mutation, and the propagation of successful policies is like natural selection.

The GA approach shares with RL the twin benefits of being based on training, not programming, and being responsive to changes in the environment through the possibility of online re-training. However, there are reasons to believe that RL makes better use of available computing resources. Consider, for example, a policy A that is in the pool of policies being evaluated by GA. Further, assume that the long-term cumulative reward from policy A is very poor. Then GA would simply discard policy A in favor of better policies in its pool without learning anything from the simulation of A. RL on the other hand has the potential of learning a lot from following policy A. Supposing policy A assigns action a to some state s, and that whenever action a is executed in state s the resulting next state always has high utility. RL will learn that action a is good in state s from executing policy A, even though the policy itself is bad. It can do this because it learns a state-based utility function. GA on the other hand ignores the state-based structure of the problem and therefore wastes information.

In summary, the GA approach needs a partial success in terms of total reward before it can make use of it, while the RL approach only needs a partial success in terms of reaching a high-utility intermediate state, which is easier to achieve.

It is not yet clear exactly when Genetic Algorithms are the preferred approach and when RL is, but in general, RL seems to do better as the problem becomes more complex. Genetic Algorithms may do well when there is a constrained, small space of policies to be searched.

References

[1] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[2] D. P. Bertsekas. *Dynamic Programming and Optimal Control: Vols 1 and 2*. Athena-Scientific, Belmont, MA, 1995.

[3] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances is Neural Information Processing Systems 8*, pages 1017–1023. MIT press, 1996.

[4] D. W. North. A tutorial introduction to decision theory. *IEEE Transactions on Systems Man and Cybernetics*, SSC-4(3), Sept. 1968.

[5] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. submitted.

[6] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–277, May 1992.

[7] W. Zhang and T. G. Dietterich. High-performance job-shop scheduling with a time-delay td(lambda) network. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances is Neural Information Processing Systems 8*, pages 1024–1030. MIT Press, 1996.

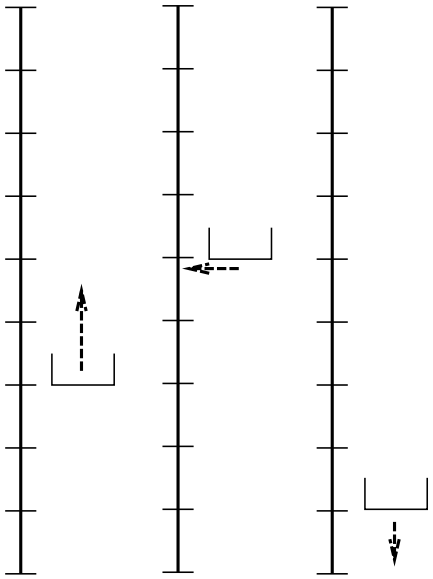

Figure 1: The Elevator Problem. A building with ten floors and three elevators. The software program is to control the actions of each elevator. The information available to the program are the state of each button on the floors and inside the elevators, as well as the time elapsed since each floor-button push.

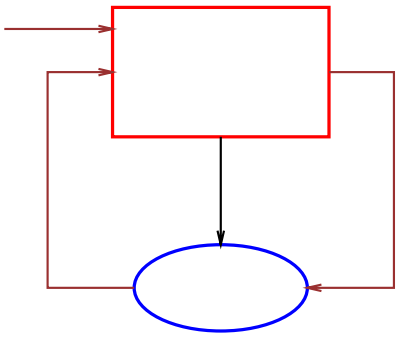

Figure 2: Abstract Agent Program. The program interacts with an environment in a feedback cycle. At each step, the agent perceives part of the environment (and may remember some), executes an action, and receives a reward. The environment changes to a new state, which is partially determined by the action (elevators move up or down) and partially unpredictable (buttons are pushed). The agent's goal is to execute actions to maximize the reward it accumulates over time.

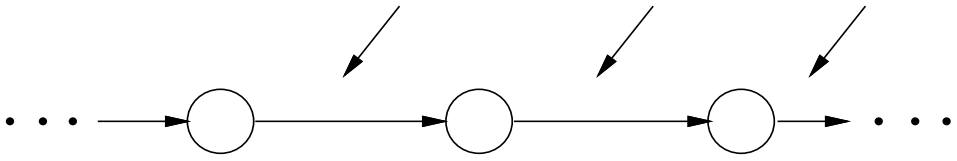

Figure 3: The program's experience consists of a trajectory through state space. At time step t, the state is s_t and the agent faces a choice of actions. Denote the action the agent chooses to execute at step t as a_t. The reward at step t, REWARD_t, is a function of s_t and a_t. The next state s_{t+1} depends on s_t, a_t and many random things such as passengers arriving at floors and pushing buttons. Reinforcement learning allows a program to use such a trajectory to incrementally improve its policy.

Utility (over a finite agent lifetime) is defined as the expected sum of the immediate reward and the long-time reward under the best possible policy:

$$\text{Util}(s_t, a) = E\left\{ \text{REWARD}(s_t, a) + \max_{\text{policies}} \sum_{j=1}^{N-1} \text{REWARD}_{t+j} \right\}, \tag{1}$$

where s_t is the state at time step t, $\text{REWARD}(s_t, a)$ is the immediate reward of executing action a in state s_t, N is the number of steps in the lifetime of the agent, and REWARD_t is the reward at time step t.). The operator $E\{\cdot\}$ stands for taking an expectation over all sources of randomness in the system.
Utility (over an infinite agent lifetime) is defined similarly:

$$\text{Util}(s_t, a) = E\left\{ \text{REWARD}(s_t, a) + \max_{\text{policies}} \sum_{j=1}^{\infty} \gamma^j \text{REWARD}_{t+j} \right\}. \tag{2}$$

To avoid the mathematical awkwardness of infinite sums, we introduce a **discount factor**, $0 \leq \gamma < 1$, which counts future rewards less than immediate rewards. This is similar to the compound interest that banks use.

The utility of a state can be defined in terms of the utility of the next state:

$$\text{Util}(s_t, a) = E\{\text{REWARD}(s_t, a) + \gamma \max_b \text{Util}(s_{t+1}, b)\}. \tag{3}$$

This gives a system of equations, called the *Bellman Optimality Equations* (e.g., Bertsekas [2]), one for each possible state-action pair, the solution to which is the utility function.

The dynamic programming method for solving the Bellman equations is to iterate the following, $\forall s, a$:

$$\text{Util}_{n+1}(s, a) = E\{\text{REWARD}(s, a) + \gamma \max_b \text{Util}_n(s, b)\} \tag{4}$$

The RL rule for updating the estimated utility is:

$$\begin{aligned} \text{Util}_{t+1}(s_t, a_t) &= (1-\alpha)\text{Util}_t(s_t, a_t) \\ &\quad +\alpha\left(\text{REWARD}(s_t, a_t) + \gamma \max_b \text{Util}_t(s_{t+1}, b)\right) \end{aligned} \tag{5}$$

where α is a small number less than one that determines the rate of change of the estimate. Notice that the second part of Equation 5 is a lot like Equation 3, except that there are no expectation signs E anywhere. See Barto, Bradtke, and Singh [1] for additional information on RL methods.

Figure 4: Equations for Reinforcement Learning